

O'REILLY®

Second
Edition

Certified Kubernetes Application Developer (CKAD) Study Guide

In-Depth Guidance and Practice



Benjamin Muschko
Foreword by Chris Aniszczyk

Praise for *Certified Kubernetes Application Developer (CKAD) Study Guide*

Benjamin Muschko is a cloud native guru and has expanded and sharpened this CKAD study guide. There is much to learn in Kubernetes, yet Ben keeps his focus on the concepts that are expected in the exam. Add this guide to ease your journey toward certification or to simply be a well-informed cloud native developer.

—*Jonathan Johnson, Independent Software Architect*

A direct, example-driven guide essential for CKAD exam preparation.

—*Bilgin Ibryam, Coauthor of Kubernetes Patterns,
Principal Product Manager at Diagrid*

As someone who oversaw the creation of the CKAD in CNCF, I'm happy to see an updated study guide that covers the latest version of the ever evolving certification.

—*Chris Aniszczyk, CTO and Cofounder, CNCF*

SECOND EDITION

Certified Kubernetes Application Developer (CKAD) Study Guide

In-Depth Guidance and Practice

Benjamin Muschko
Foreword by Chris Aniszczyk

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Certified Kubernetes Application Developer (CKAD) Study Guide

by Benjamin Muschko

Copyright © 2024 Automated Ascent LLC. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: John Devins

Proofreader: Helena Stirling

Development Editor: Virginia Wilson

Interior Designer: David Futato

Production Editor: Beth Kelly

Cover Designer: Karen Montgomery

Copyeditor: Piper Editorial Consulting, LLC

Illustrator: Kate Dullea

February 2021: First Edition

June 2024: Second Edition

Revision History for the Second Edition

2024-5-22: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098152864> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Certified Kubernetes Application Developer (CKAD) Study Guide*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

Table of Contents

Foreword.....	xv
Preface.....	xvii

Part I. Introduction

1. Exam Details and Resources.....	3
Kubernetes Certification Learning Path	3
Kubernetes and Cloud Native Associate (KCNA)	4
Kubernetes and Cloud Native Security Associate (KCSA)	4
Certified Kubernetes Application Developer (CKAD)	4
Certified Kubernetes Administrator (CKA)	4
Certified Kubernetes Security Specialist (CKS)	4
Exam Objectives	5
Curriculum	5
Application Design and Build	6
Application Deployment	6
Application Observability and Maintenance	6
Application Environment, Configuration, and Security	6
Services and Networking	7
Involved Kubernetes Primitives	7
Documentation	8
Exam Environment and Tips	8
Candidate Skills	9
Time Management	10
Command-Line Tips and Tricks	11
Setting a Context and Namespace	11

Using the Alias for kubectl	11
Using kubectl Command Auto-Completion	11
Internalize Resource Short Names	12
Practicing and Practice Exams	12
Summary	13
2. Kubernetes in a Nutshell.....	15
What Is Kubernetes?	15
Features	16
High-Level Architecture	17
Control Plane Node Components	18
Common Node Components	18
Advantages	19
Summary	20
3. Interacting with Kubernetes.....	21
API Primitives and Objects	21
Using kubectl	23
Managing Objects	24
Imperative Object Management	24
Declarative Object Management	26
Hybrid Approach	28
Which Approach to Use?	29
Summary	29

Part II. Application Design and Build

4. Containers.....	33
Container Terminology	34
Containerizing a Java-Based Application	35
Writing a Dockerfile	35
Building the Container Image	36
Listing Container Images	36
Running the Container	37
Listing Containers	37
Interacting with the Container	37
Publishing the Container Image	38
Saving and Loading a Container Image	39
Going Further	40
Summary	40

Exam Essentials	40
Sample Exercises	41
5. Pods and Namespaces.....	43
Working with Pods	43
Creating Pods	43
Listing Pods	46
Pod Life Cycle Phases	46
Rendering Pod Details	47
Accessing Logs of a Pod	48
Executing a Command in Container	48
Creating a Temporary Pod	49
Using a Pod's IP Address for Network Communication	49
Configuring Pods	50
Deleting a Pod	53
Working with Namespaces	53
Listing Namespaces	54
Creating and Using a Namespace	54
Setting a Namespace Preference	55
Deleting a Namespace	55
Summary	55
Exam Essentials	56
Sample Exercises	56
6. Jobs and CronJobs.....	59
Working with Jobs	59
Creating and Inspecting Jobs	60
Job Operation Types	61
Restart Behavior	62
Working with CronJobs	63
Creating and Inspecting CronJobs	64
Configuring Retained Job History	65
Summary	66
Exam Essentials	66
Sample Exercises	67
7. Volumes.....	69
Working with Storage	69
Volume Types	70
Ephemeral Volumes	70
Persistent Volumes	72

Storage Classes	77
Summary	79
Exam Essentials	80
Sample Exercises	80
8. Multi-Container Pods.....	83
Working with Multiple Containers in a Pod	83
Init Containers	84
The Sidecar Pattern	86
The Adapter Pattern	88
The Ambassador Pattern	90
Summary	93
Exam Essentials	93
Sample Exercises	94
9. Labels and Annotations.....	95
Working with Labels	95
Declaring Labels	96
Inspecting Labels	97
Modifying Labels for a Live Object	97
Using Label Selectors	98
Recommended Labels	100
Working with Annotations	101
Declaring Annotations	101
Inspecting Annotations	102
Modifying Annotations for a Live Object	102
Reserved Annotations	102
Summary	103
Exam Essentials	103
Sample Exercises	103

Part III. Application Deployment

10. Deployments.....	107
Working with Deployments	107
Creating Deployments	108
Listing Deployments and Their Pods	110
Rendering Deployment Details	111
Deleting a Deployment	112
Performing Rolling Updates and Rollbacks	112

Updating a Deployment's Pod Template	113
Rolling Out a New Revision	113
Adding a Change Cause for a Revision	115
Rolling Back to a Previous Revision	115
Scaling Workloads	116
Manually Scaling a Deployment	116
Autoscaling a Deployment	117
Creating Horizontal Pod Autoscalers	118
Listing Horizontal Pod Autoscalers	119
Rendering Horizontal Pod Autoscaler Details	120
Defining Multiple Scaling Metrics	120
Summary	122
Exam Essentials	122
Sample Exercises	122
11. Deployment Strategies.....	125
Rolling Deployment Strategy	126
Implementation	126
Use Cases and Trade-Offs	128
Fixed Deployment Strategy	128
Implementation	129
Use Cases and Trade-Offs	130
Blue-Green Deployment Strategy	130
Implementation	131
Use Cases and Trade-Offs	133
Canary Deployment Strategy	133
Implementation	134
Use Cases and Trade-Offs	135
Summary	135
Exam Essentials	135
Sample Exercises	136
12. Helm.....	139
Managing an Existing Chart	139
Identifying a Chart	140
Adding a Chart Repository	141
Searching for a Chart in a Repository	143
Installing a Chart	143
Listing Installed Charts	145
Upgrading an Installed Chart	145
Uninstalling a Chart	146

Summary	146
Exam Essentials	146
Sample Exercises	147
<hr/>	
Part IV. Application Observability and Maintenance	
13. API Deprecations.....	151
Understanding the Deprecation Policy	151
Listing Available API Versions	152
Handling Deprecation Warnings	152
Handling a Removed or Replaced API	153
Summary	155
Exam Essentials	155
Sample Exercises	155
14. Container Probes.....	157
Working with Probes	157
Probe Types	158
Health Verification Methods	159
Health Check Attributes	160
The Readiness Probe	160
The Liveness Probe	161
The Startup Probe	162
Summary	163
Exam Essentials	164
Sample Exercises	164
15. Troubleshooting Pods and Containers.....	165
Troubleshooting Pods	165
Retrieving High-Level Information	166
Inspecting Events	167
Using Port Forwarding	168
Troubleshooting Containers	169
Inspecting Logs	169
Opening an Interactive Shell	170
Interacting with a Distroless Container	171
Inspecting Resource Metrics	172
Summary	174
Exam Essentials	174
Sample Exercises	174

Part V. Application Environment, Configuration, and Security

16. CustomResourceDefinitions (CRDs).....	179
Working with CRDs	179
Example CRD	180
Implementing a CRD Schema	181
Instantiating an Object for the CRD	182
Discovering CRDs	183
Implementing a Controller	184
Summary	184
Exam Essentials	185
Sample Exercises	185
17. Authentication, Authorization, and Admission Control.....	187
Processing a Request	187
Authentication with kubectl	188
The Kubeconfig	188
Managing Kubeconfig Using kubectl	190
Authorization with Role-Based Access Control	191
RBAC Overview	191
Understanding RBAC API Primitives	192
Default User-Facing Roles	193
Creating Roles	193
Listing Roles	195
Rendering Role Details	195
Creating RoleBindings	195
Listing RoleBindings	196
Rendering RoleBinding Details	196
Seeing the RBAC Rules in Effect	197
Namespace-Wide and Cluster-Wide RBAC	198
Working with Service Accounts	198
The Default Service Account	199
Creating a Service Account	199
Setting Permissions for a Service Account	200
Admission Control	204
Summary	205
Exam Essentials	205
Sample Exercises	205

18. Resource Requirements, Limits, and Quotas.....	207
Working with Resource Requirements	208
Defining Container Resource Requests	208
Defining Container Resource Limits	209
Defining Container Resource Requests and Limits	210
Working with Resource Quotas	211
Creating ResourceQuotas	212
Rendering ResourceQuota Details	213
Exploring a ResourceQuota's Runtime Behavior	213
Working with Limit Ranges	215
Creating LimitRanges	216
Rendering LimitRange Details	217
Exploring a LimitRange's Runtime Behavior	217
Summary	219
Exam Essentials	220
Sample Exercises	220
19. ConfigMaps and Secrets.....	223
Working with ConfigMaps	224
Creating a ConfigMap	224
Consuming a ConfigMap as Environment Variables	226
Mounting a ConfigMap as a Volume	226
Working with Secrets	228
Creating a Secret	229
Consuming a Secret as Environment Variables	232
Mounting a Secret as a Volume	233
Summary	235
Exam Essentials	235
Sample Exercises	236
20. Security Contexts.....	237
Working with Security Contexts	238
Defining a Security Context on the Pod Level	239
Defining a Security Context on the Container Level	241
Defining a Security Context on the Pod and Container Level	242
Summary	243
Exam Essentials	244
Sample Exercises	244

Part VI. Services and Networking

21. Services.....	247
Working with Services	248
Service Types	249
Port Mapping	250
Creating Services	251
Listing Services	253
Rendering Service Details	253
The ClusterIP Service Type	254
Creating and Inspecting the Service	255
Accessing the Service	256
The NodePort Service Type	258
Creating and Inspecting the Service	259
Accessing the Service	260
The LoadBalancer Service Type	260
Creating and Inspecting the Service	261
Accessing the Service	262
Summary	263
Exam Essentials	263
Sample Exercises	263
22. Ingresses.....	265
Working with Ingresses	266
Installing an Ingress Controller	266
Deploying Multiple Ingress Controllers	267
Configuring Ingress Rules	267
Creating Ingresses	268
Defining Path Types	269
Listing Ingresses	270
Rendering Ingress Details	270
Accessing an Ingress	272
Summary	273
Exam Essentials	273
Sample Exercises	273
23. Network Policies.....	275
Working with Network Policies	275
Installing an Network Policy Controller	276
Creating a Network Policy	276

Listing Network Policies	279
Rendering Network Policy Details	279
Applying Default Network Policies	280
Restricting Access to Specific Ports	282
Summary	283
Exam Essentials	283
Sample Exercises	283
A. Answers to Review Questions.....	285
B. Exam Review Guide.....	339

Foreword

The software development landscape is rapidly evolving, and cloud-native technologies are at the forefront of this change. The Cloud Native Computing Foundation (CNCF) is one of the largest open source communities in the world with over 190 open source projects and has 200,000+ contributors worldwide improving the state of cloud native every day. From my experience as the cofounder and CTO, it can be fairly tricky to understand where you should begin to explore this vast **cloud native landscape**.

This study guide prepares you well for the Certified Kubernetes Application Developer (CKAD), which covers Kubernetes skills from the perspective of an application developer. In my opinion, having detailed examples is critical in passing the CKAD and this book has many of them. Ben covers all the topics required to successfully pass the certification and even lays some groundwork for future advanced certifications like the Certified Kubernetes Security Specialist (CKS). What's more, reading this book will give you the practice you need to become a more forward-thinking professional who embraces modern and open source development practices.

Kubernetes is one project that is at the heart of the open source cloud native movement, enabling developers to build, deploy, and scale applications with agility. Kubernetes certifications are incredibly popular with more than 100K+ registrations to date. We designed the CKAD certification to be focused on the application developer and provide a solid foundation for the fundamentals of Kubernetes and cloud native open source skills. Earning it validates your understanding of these critical technologies, and signifies your commitment to continuous learning and staying ahead of the curve. Furthermore, the knowledge you gain here will empower you to effectively deploy and manage cloud native applications (a skill in high demand across various industries), help open doors to exciting career opportunities, and demonstrate your ability to thrive in the constant dynamic world of cloud native.

I wish you luck in your cloud native career journey; and once you pass, come join us at the contribute.cncf.io community.

— *Chris Aniszczyk*
CTO and Cofounder, CNCF

Austin, TX

March 12, 2024

Preface

Microservices architecture is one of the hottest areas of application development today, particularly for cloud-based, enterprise-scale applications. The benefits of building applications using small, single-purpose services are well documented. But managing what can be enormous numbers of containerized services is no easy task and requires the addition of an “orchestrator” to keep it all together. Kubernetes is among the most popular and broadly used tools for this job, so it’s no surprise that the ability to use, troubleshoot, and monitor Kubernetes as an application developer is in high demand. To provide job seekers and employers a standard means to demonstrate and evaluate proficiency in developing with a Kubernetes environment, the Cloud Native Computing Foundation (CNCF) developed the [Certified Kubernetes Application Developer \(CKAD\)](#) program. To achieve this certification, you need to pass an exam.

The CKAD is not to be confused with the [Certified Kubernetes Administrator \(CKA\)](#). While there is some topic overlap, the CKA focuses mostly on Kubernetes cluster administration tasks rather than developing applications operated in a cluster.

In this study guide, I will explore the topics covered in the CKAD exam to fully prepare you to pass the certification exam. We’ll look at determining when and how you should apply the core concepts of Kubernetes to manage an application. We’ll also examine the `kubectl` command-line tool, a mainstay of the Kubernetes engineer. I will also offer tips to help you better prepare for the exam and share my personal experience with getting ready for all aspects of it.

The CKAD is different from the typical multiple-choice format of other certifications. It’s completely performance based and requires you to demonstrate deep knowledge of the tasks at hand under immense time pressure. Are you ready to pass the test on the first go?

Who This Book Is For

This book is for developers who want to prepare for the CKAD exam. The content covers all aspects of the exam curriculum, though basic knowledge of the Kubernetes architecture and its concepts is expected.

If you are completely new to Kubernetes, I recommend that you first read *Kubernetes: Up and Running* by Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson (O'Reilly) or *Kubernetes in Action* by Marko Lukša (Manning Publications).

What You Will Learn

The content of the book condenses the most important aspects of the CKAD exam. Given the plethora of configuration options available in Kubernetes, it's impossible to cover all use cases and scenarios without duplicating the official documentation. Test takers are encouraged to reference the [Kubernetes documentation](#) as the go-to compendium for broader exposure.

The outline of the book follows the CKAD curriculum. While there might be a more natural, didactic structure for learning Kubernetes in general, the curriculum outline will help test takers prepare for the exam by focusing on specific topics. As a result, you will cross-reference other chapters of the book depending on your existing knowledge level. Be aware that this book covers only the concepts relevant to the CKAD exam. Refer to the Kubernetes documentation or other books if you want to dive deeper.

Practical experience with Kubernetes is key to passing the exam. Each chapter contains a section named "Sample Exercises" with practice questions. Solutions to those questions can be found in [Appendix A](#).

What's New in the Second Edition

The CNCF periodically updates all Kubernetes certifications to keep up with the latest developments in the field. In September 2021, the CKAD curriculum received a [major overhaul](#). The organization of existing topics has been changed, and new topics have been added, making the certification more relevant and applicable to Kubernetes practitioners in real-world scenarios.

Compared to the first edition of the book, about 80% of the content hasn't changed. The new curriculum includes the following topics:

Deployment strategies

The coverage of deployment strategies goes beyond the ones directly supported by the Deployment primitive. You will need to understand how to implement and manage blue/green deployments and canary deployments.

Helm

Helm is a tool that automates the bundling, configuration, and deployment of Kubernetes applications by combining your configuration files into a single reusable package. You will need to understand how to use Helm for discovering and installing existing Helm charts.

API deprecations

You need to be aware of Kubernetes' release process and what it means to the usage of APIs that are deprecated and removed. You will learn how to handle situations that require you to switch to a newer or replaced API version.

Custom Resource Definitions (CRDs)

CRDs allow for extending the Kubernetes API by creating your own custom resource types. You need to aware of how to create CRDs, as well as how to manage objects based on the CRD type.

Authentication, authorization, and admission control

Every call to the Kubernetes API needs to be authenticated. As daily users of `kubectl`, application developers need to understand how to manage and use their credentials. Once authenticated, the request to the API also needs to pass the authorization phase. You need a rough understanding of role-based access control (RBAC), the concept that guards access to Kubernetes resources. Admission control is a topic covered by the CKA and Certified Kubernetes Security Specialist (CKS) exams, and therefore I'll just scratch the surface of this aspect.

Ingress

You will want to expose your customer-facing applications running in Kubernetes to outside consumers. The Ingress primitive routes HTTPS traffic to one of many Service backends. The Ingress is now part of the CKAD curriculum.

Additionally, I included an “Exam Review Guide” in [Appendix B](#) that maps curriculum topics to the corresponding chapters in the book plus coverage of the topic in the Kubernetes documentation.

The previous version of the CKAD curriculum included two topics that have been removed: “Create and configure basic Pods” and “Understand how to use Labels, Selectors, and Annotations.” This book covers both topics, even though they haven’t been mentioned explicitly. Pods are essential for running workload in Kubernetes, so you’ll definitely need to know about them. [Chapter 5](#) explains the ins and outs. Given that labels and label selection are such important concepts to understand for primitives like the Deployment, Service, and Network Policies, I kept a dedicated chapter named [Labels and Annotations](#).

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

Using Code Examples

The source code for all examples and exercises in this book is available on [GitHub](#). The repository is distributed under the Apache License 2.0. The code is free to use in commercial and open source projects. If you encounter an issue in the source code or if you have a question, open an issue in the [GitHub issue tracker](#). I'm happy to have a conversation and fix any issues that might arise.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not

need to contact us for permission unless you’re reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O’Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product’s documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Certified Kubernetes Application Developer (CKAD) Study Guide*, by Benjamin Muschko (O’Reilly). Copyright 2024 Automated Ascent, LLC, 978-1-098-16949-7.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-827-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://oreil.ly/ckad-2ed>.

Email bookquestions@oreilly.com to comment or ask technical questions about this book.

For news and information about our books and courses, visit <http://oreilly.com>.

Watch us on YouTube: <http://youtube.com/oreillymedia>

Find us on LinkedIn: <https://linkedin.com/company/oreilly-media>

Follow the author on Twitter: <https://twitter.com/bmuschko>

Follow the author on GitHub: <https://github.com/bmuschko>

Follow the author's blog: <https://bmuschko.com>

Acknowledgments

Every book project is a long journey and would not be possible without the help of the editorial staff and technical reviewers. Special thanks go to Jonathon Johnson, Bilgin Ibryam, Vladislav Bilay, Andrew Martin, and Michael Levan for their detailed technical guidance and feedback. I would also like to thank the editors at O'Reilly Media, John Devins and Virginia Wilson, for their continued support and encouragement.

PART I

Introduction

The *Introduction* section of the book touches on the most important aspects of the exam and orients beginners to Kubernetes to the lay of the land without introducing too much complexity.

The following chapters cover these concepts:

- [Chapter 1](#) discusses the exam objectives, curriculum, and tips and tricks for passing the exam.
- [Chapter 2](#) is a short and sweet overview on Kubernetes. This chapter summarizes the purpose and benefits of Kubernetes and provides an overview of its architecture and components.
- [Chapter 3](#) discusses how to interact with a Kubernetes cluster using the command line tool `kubectl`. The tool is going to be your only user interface during the exam. We'll compare imperative and declarative commands, their pros and cons, as well as time-saving techniques for the exam.

CHAPTER 1

Exam Details and Resources

This chapter addresses the most frequently asked questions by candidates preparing to successfully pass the [Certified Kubernetes Application Developer \(CKAD\)](#) exam. Later chapters will give you a summary of Kubernetes' benefits and architecture and how to [interact with a Kubernetes cluster using kubectl](#).

Kubernetes Certification Learning Path

The CNCF offers four different Kubernetes certifications. [Figure 1-1](#) categorizes each of them by target audience.

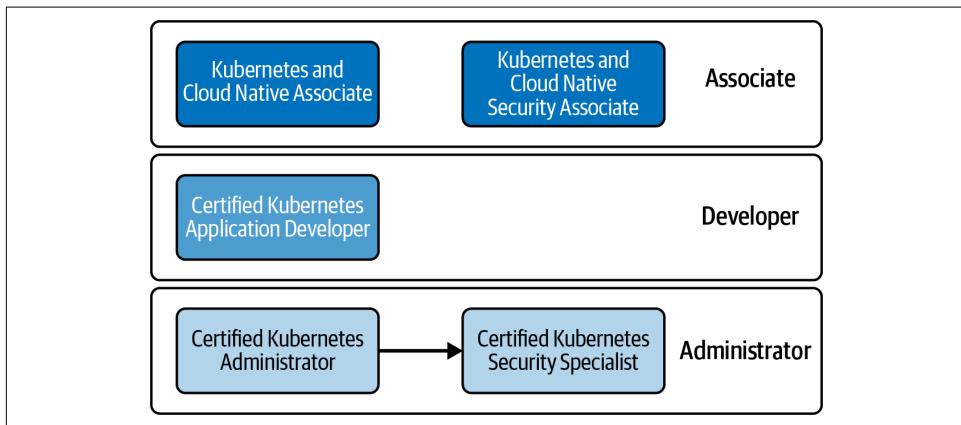


Figure 1-1. Kubernetes certifications learning path

The target audience for associate-level certifications is beginners to the cloud and Kubernetes. Associate-level certification exams use a multiple-choice format. You will not have to interact with a Kubernetes cluster in an interactive environment.

Practitioner-level certifications are meant for developers and administrators with pre-existing Kubernetes experience. Exams in this category require you to solve problems in multiple Kubernetes environments hands-on. You will find that the CKAD is geared to application developers and does not require any other certification as a prerequisite.

Let's have a brief look at each certification to see if the CKAD is the right fit for you.

Kubernetes and Cloud Native Associate (KCNA)

KCNA is an entry-level certification program for anyone interested in cloud-native application development, runtime environments, and tooling. While the exam does cover Kubernetes, it does *not* expect you to interact with a cluster hands-on. This exam consists of multiple-choice questions and is suitable to candidates interested in the topic with a broad exposure to the ecosystem.

Kubernetes and Cloud Native Security Associate (KCSA)

This certification focuses on basic knowledge of security concepts and their application in a Kubernetes cluster. The breadth, depth, and format of the program is comparable to the KCNA.

Certified Kubernetes Application Developer (CKAD)

The CKAD exam focuses on verifying your ability to build, configure, and deploy a microservices-based application to Kubernetes. You are not expected to actually implement an application; however, the exam is suitable for developers familiar with topics like application architecture, runtimes, and programming languages.

Certified Kubernetes Administrator (CKA)

The target audience for the CKA exam are DevOps practitioners, system administrators, and site reliability engineers. This exam tests your ability to perform in the role of a Kubernetes administrator, which includes tasks like cluster, network, storage, and beginner-level security management, with emphasis on troubleshooting scenarios.

Certified Kubernetes Security Specialist (CKS)

The CKS exam expands on the topics verified by the CKA exam. Passing the CKA is a prerequisite before you can sign up for the CKS exam. For this certification, you are expected to have a deeper knowledge of Kubernetes security. The curriculum covers topics like applying best practices for building containerized applications and ensuring a secure Kubernetes runtime environment.

Exam Objectives

This book focuses on getting you ready for the CKAD exam. I will give a little bit of background on why Kubernetes is important to application developers before dissecting the topics important to the exam.

More and more application developers find themselves in projects transitioning from a monolithic architectural model to bite-sized, cohesive, and containerized microservices. There are pros and cons to both approaches, but Kubernetes has become the de facto runtime platform for deploying and operating applications without needing to worry about the underlying physical infrastructure.

It is no longer the exclusive responsibility of an administrator or release manager to deploy and monitor their applications in target runtime environments. Application developers need to see their applications through from development to operation. Some organizations like Netflix live and breathe this culture, so you, the application developer, are fully responsible for making design decisions as well as fixing issues in production. It's more important than ever to understand the capabilities of Kubernetes, how to apply the relevant concepts properly, and how to interact with the platform.

The exam is designed specifically for application developers who need to design, build, configure, and manage cloud native applications on Kubernetes.



Kubernetes version used during the exam

At the time of writing, the exam is based on Kubernetes 1.28. All content in this book will follow the features, APIs, and command-line support for that version. It's possible that future versions will break backward compatibility. While preparing for the certification, review the [Kubernetes release notes](#) and practice with the Kubernetes version used during the exam to avoid unpleasant surprises. The exam environment will be aligned with the most recent Kubernetes minor version within approximately four to eight weeks of the Kubernetes release date.

Curriculum

The following overview lists the high-level sections, or domains, of the exam and their scoring weights:

- 20%: [Application Design and Build](#)
- 20%: [Application Deployment](#)
- 15%: [Application Observability and Maintenance](#)

- 25%: Application Environment, Configuration, and Security
- 20%: Services and Networking

The next sections detail each domain.

Application Design and Build

The first domain of the curriculum covers designing and building a containerized application and operating it in Kubernetes. You will need to be familiar with basic container concepts and how to define a container inside of a Pod. In addition, the domain covers more advanced use cases and Kubernetes concepts: the use of storage in Pods, the need for defining multiple containers inside of a Pod, and how to define and execute batch and periodic workloads.

Application Deployment

This domain primarily focuses on the Kubernetes primitive Deployment. A Deployment helps with scaling Pods with the same definition, so-called replicas, and managing the configuration across all replicas it controls. You need to understand managing deployments including strategies helpful for rolling out new versions of an application to replicas in a controlled fashion. Finally, you will need to be familiar with Helm, an open source tool for managing a set of manifests required to deploy and configure an application stack.

Application Observability and Maintenance

Deploying an application to Kubernetes is only the first step. You need to be able to monitor, inspect, and potentially debug Kubernetes objects. Probes are an important concept covered by this domain: they define health checks for applications running in a Pod. Furthermore, you need to be confident with identifying runtime issues for workload and how to fix them.

Application Environment, Configuration, and Security

Kubernetes offers security and resource management features configurable for a Pod. This includes the security context and resource requirements/constraints covered in this domain. Furthermore, you need to be able to demonstrate the use of ConfigMaps and Secrets to inject configuration data into a Pod to control its runtime behavior. The domain also touches on the rudimentary concepts and functionality of role-based access control (RBAC) and CustomResourceDefinitions (CRDs).

Services and Networking

The last domain of the curriculum deals with providing network access to your application from within and outside of the cluster. For that purpose, you'll need to demonstrate knowledge of Services and Ingresses. Finally, you'll need a rough understanding of network policies, which are essentially rules that deny or permit Pod-to-Pod communication.

Involved Kubernetes Primitives

Some of the exam objectives can be covered by understanding the relevant core Kubernetes primitives. Be aware that the exam combines multiple concepts in a single problem. Refer to [Figure 1-2](#) as a guide to the applicable Kubernetes resources and their relationships.

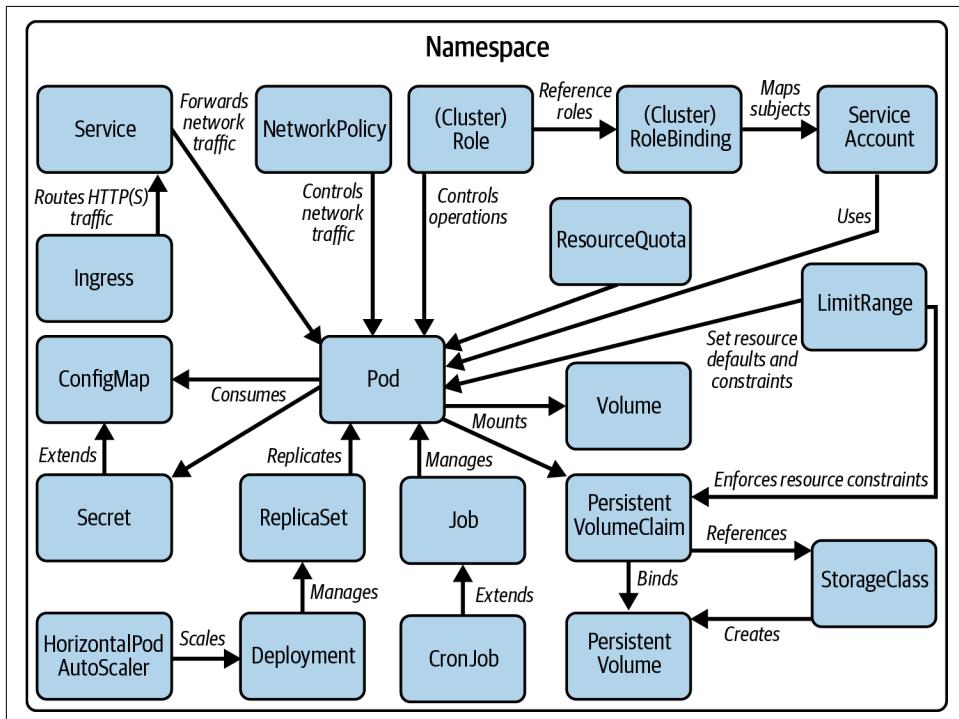


Figure 1-2. Kubernetes primitives relevant to the exam

Documentation

During the exam, you are permitted to open a well-defined list of web pages as a reference. You can freely browse those pages and copy-paste code to the exam terminal.

The official Kubernetes documentation includes the reference manual, the GitHub site, and the blog. In addition, you can also browse the Helm documentation.

- Reference manual: <https://kubernetes.io/docs>
- Blog: <https://kubernetes.io/blog>
- Helm: <https://helm.sh/docs>

Having the Kubernetes documentation pages at hand is extremely valuable, but make sure you know *where* to find the relevant information within those pages. In preparation for the test, read all the documentation pages from start to end at least once. Don't forget the search functionality of the official documentation pages. For reference, [Appendix B](#) maps the exam objectives to the book chapters covering the topics and the relevant Kubernetes documentation pages.



Using the documentation efficiently

Using a search term will likely lead you to the right documentation pages quicker than navigating the menu items. Copying and pasting code snippets from the documentation into the console of the exam environment works reasonably well. You may have to adjust the YAML indentation manually as the proper formatting can get lost in the process.

Exam Environment and Tips

To take the exam, you must purchase a registration voucher, which can be acquired on the [CNCF training and certification web page](#). On occasion, the CNCF offers discounts for the voucher (e.g., around the US Thanksgiving holiday). Those discount offers are often announced on the [Linux Foundation LinkedIn page](#) and the Twitter account [@LF_Training](#).

After you purchase the voucher, you can schedule a time for the exam with [PSI](#), the company conducting the test virtually. In-person exams at a testing facility are not available. On the day of your scheduled test, you'll be asked to log into the test platform with a URL provided to you by email. You'll be asked to enable the audio and video feed on your computer to discourage cheating. A proctor will oversee your actions via audio/video feed and terminate the session if they think you are not following the rules.



Exam attempts

The voucher you purchased grants two attempts to pass the exam. I recommend preparing reasonably well before taking the test on the first attempt. It will give you a fair chance to pass the test and provide a good impression of the exam environment and the complexity of the questions. Don't sweat it if you do not pass the test on the first attempt. You've got another free shot.

The CKAD has a time limit of two hours. During that time, you'll need to solve hands-on problems on a real, predefined Kubernetes cluster. Every question will state the cluster you need to work on. This practical approach to gauge a candidate's skill set is superior to tests with multiple-choice questions, as you can translate the knowledge directly on tasks performed on the job.

I highly recommend reading the [FAQ for the exam](#). You will find answers to most of your pressing questions there, including system requirements for your machine, scoring, certification renewal, and retake requirements.

Candidate Skills

The certification assumes that you have a basic understanding of Kubernetes. You should be familiar with Kubernetes internals, its core concepts, and the command-line tool `kubectl`. The CNCF offers a free "[Introduction to Kubernetes](#)" course for beginners to Kubernetes.

Your background is likely more on the end of an application developer, although it doesn't really matter which programming language you're most accustomed to. Here's a brief overview of the background knowledge you need to increase your likelihood of passing the exam:

Kubernetes architecture and concepts

The exam won't ask you to install a Kubernetes cluster from scratch. Read up on the basics of Kubernetes and its architectural components. Reference [Chapter 2](#) for a jump start on Kubernetes' architecture and concepts.

The kubectl CLI tool

The `kubectl` command-line tool is the central tool you will use during the exam to interact with the Kubernetes cluster. Even if you have only a little time to prepare for the exam, it's essential to practice how to operate `kubectl`, as well as its commands and their relevant options. You will have no access to the [web dashboard UI](#) during the exam. [Chapter 3](#) provides a short summary of the most important ways of interacting with a Kubernetes cluster.

Working knowledge of a container runtime engine

Kubernetes uses a container runtime engine for managing images. A widely used container runtime engine is Docker Engine. At a minimum, understand container files, container images, containers, and relevant CLI commands. [Chapter 4](#) explains all you need to know about containers for the exam.

Other relevant tools

Kubernetes objects are represented by YAML or JSON. The content of this book will use examples in YAML, as it is more commonly used than JSON in the Kubernetes world. You will have to edit YAML during the exam to create a new object declaratively or when modifying the configuration of a live object. Ensure that you have a good handle on basic YAML syntax, data types, and indentation conforming to the specification. How do you edit the YAML definitions, you may ask? From the terminal, of course. The exam terminal environment comes with the tools `vi` and `vim` preinstalled. Practice the keyboard shortcuts for common operations, (especially how to exit the editor). The last tool I want to mention is GNU Bash. It's imperative that you understand the basic syntax and operators of the scripting language. It's absolutely possible that you may have to read, modify, or even extend a multiline Bash command running in a container.

Time Management

Candidates have two hours to complete the exam, and 66% of the answers to the questions need to be correct to pass. Many questions consist of multiple steps. Although the Linux Foundation doesn't provide a scoring breakdown, I'd assume that partially correct answers will score a portion of the points.

When taking the test, you will notice that the given time limit will put you under a lot of pressure. That's intentional. The Linux Foundation expects Kubernetes practitioners to be able to apply their knowledge to real-world scenarios by finding solutions to problems in a timely fashion.

The exam will present you with a mix of problems. Some are short and easy to solve; others require more context and take more time. Personally, I tried to tackle the easy problems first to score as many points as possible without getting stuck on the harder questions. I marked any questions I could not solve immediately in the notepad functionality integrated in the exam environment. During the second pass, revisit the questions you skipped and try to solve them as well. Optimally, you will be able to work through all the problems in the allotted time.

Command-Line Tips and Tricks

Given that the command line is your solitary interface to the Kubernetes cluster, it's essential that you become extremely familiar with the `kubectl` tool and its available options. This section provides tips and tricks for making their use more efficient and productive.

Setting a Context and Namespace

The exam environment comes with multiple Kubernetes clusters already set up for you. Take a look at the [instructions](#) for a high-level, technical overview of those clusters. Each of the exam exercises needs to be solved on a designated cluster, as outlined in its description. Furthermore, the instructions will ask you to work in a namespace other than `default`. Make sure to set the context and namespace as the first course of action before working on a question. The following command sets the context and the namespace as a one-time action:

```
$ kubectl config set-context <context-of-question> \  
  --namespace=<namespace-of-question>  
$ kubectl config use-context <context-of-question>
```

You can find a more detailed discussion of the context concept and the corresponding `kubectl` commands in [“Authentication with kubectl” on page 188](#).

Using the Alias for `kubectl`

In the course of the exam, you will have to execute the `kubectl` command tens or even hundreds of times. You might be an extremely fast typist; however, there's no point in fully spelling out the executable over and over again. The exam environment already sets up the alias `k` for the `kubectl` command.

In preparation for the exam, you can set up the same behavior on your machine. The following `alias` command maps the letter `k` to the full `kubectl` command:

```
$ alias k=kubectl  
$ k version
```

Using `kubectl` Command Auto-Completion

Memorizing `kubectl` commands and command-line options takes a lot of practice. The exam environment comes with auto-completion enabled by default. You can find instructions for setting up auto-completion for the shell on your machine in the [Kubernetes documentation](#).

Internalize Resource Short Names

Many of the `kubectl` commands can be quite lengthy. For example, the command for managing Persistent volume claims is `persistentvolumeclaims`. Spelling out the full command can be error-prone and time-consuming. Thankfully, some of the longer commands come with a short-form usage. The command `api-resources` lists all available commands plus their short names:

```
$ kubectl api-resources
NAME           SHORTNAMES   APIGROUP   NAMESPACED   KIND
...
persistentvolumeclaims   pvc          true        PersistentVolumeClaim
...
```

Using `pvc` instead of `persistentvolumeclaims` results in a more concise and expressive command execution, as shown here:

```
$ kubectl describe pvc my-claim
```

Practicing and Practice Exams

Hands-on practice is extremely important when it comes to passing the exam. For that purpose, you'll need a functioning Kubernetes cluster environment. The following options stand out:

- I found it useful to run one or many virtual machines using [Vagrant](#) and [VirtualBox](#). Those tools help with creating an isolated Kubernetes environment that is easy to bootstrap and dispose on demand.
- It is relatively easy to install a simple Kubernetes cluster on your developer machine. The Kubernetes documentation provides various [installation options](#), depending on your operating system. Minikube is useful when it comes to experimenting with more advanced features like Ingress or storage classes, as it provides the necessary functionality as add-ons that can be installed with a single command. Alternatively, you can also give [kind](#) a try, another tool for running local Kubernetes clusters.
- If you're a subscriber to the [O'Reilly Learning Platform](#), you have unlimited access to scenarios running a [Kubernetes sandbox environment](#). In addition, you can test your knowledge with the help of the [CKAD practice test in the form of interactive labs](#).

You may also want to try one of the following commercial learning and practice resources:

- [Killer Shell](#) is a simulator with sample exercises for all Kubernetes certifications. If you purchase a voucher for the exam, you will be allowed two free sessions.

- Other online training providers offer video courses for the exam, some of which include an integrated Kubernetes practice environment. I would like to mention [KodeKloud](#) and [A Cloud Guru](#). You'll need to purchase a subscription to access the content for each course individually.

Summary

The exam is a completely hands-on test that requires you to solve problems in multiple Kubernetes clusters. You're expected to understand, use, and configure the Kubernetes primitives relevant to application developers. The exam curriculum subdivides those focus areas and puts different weights on topics, which determines their contributions to the overall score. Even though focus areas are grouped meaningfully, the curriculum doesn't necessarily follow a natural learning path, so it's helpful to cross-reference chapters in the book in preparation for the exam.

In this chapter, we discussed the exam environment and how to navigate it. The key to acing the exam is intense practice of `kubectl` to solve real-world scenarios. The next two chapters in [Part I](#) will provide a jump start to Kubernetes.

All chapters that discuss domain details give you an opportunity to practice hands-on. You will find sample exercises at the end of each chapter.

Kubernetes in a Nutshell

It's helpful to get a quick rundown of what Kubernetes is and how it works if you are new to the space. Many tutorials and 101 courses are available on the web, but I would like to summarize the most important background information and concepts in this chapter. In the course of this book, we'll reference cluster node components, so feel free to come back to this information at any time.

What Is Kubernetes?

To understand what Kubernetes is, first let's define microservices and containers.

Microservice architectures call for developing and executing pieces of the application stack as individual services, and those services have to communicate with one another. If you decide to operate those services in containers, you will need to manage a lot of them while at the same time thinking about cross-cutting concerns like scalability, security, persistence, and load balancing.

Tools like [buildkit](#) and [Podman](#) package software artifacts into a container image. Container runtime engines like [Docker Engine](#) and [containerd](#) use the image to run a container. This works great on developer machines for testing purposes or for ad-hoc executions, e.g., as part of a Continuous Integration pipeline. For more information on containers, refer to [Chapter 4](#).

Kubernetes is a container orchestration tool that helps with operating hundreds or even thousands of containers on physical machines, virtual machines, or in the cloud. Kubernetes can also fulfill those cross-cutting concerns mentioned earlier. The container runtime engine integrates with Kubernetes. Whenever a container creation is triggered, Kubernetes will delegate life cycle aspects to the container runtime engine.

The most essential primitive in a Kubernetes is a Pod. The Pod can run one or many containers while at the same time adding cross-cutting concerns like security requirements and resource consumption expectations. Have a look at [Chapter 5](#) to learn about those aspects.

Features

The previous section touched on some features provided by Kubernetes. Here, we are going to dive a little deeper by explaining those features with more detail:

Declarative model

You do not have to write imperative code using a programming language to tell Kubernetes how to operate an application. All you need to do as an end user is to declare a desired state. The desired state can be defined using a YAML or JSON manifest that conforms to an API schema. Kubernetes then maintains the state and recovers it in case of a failure.

Autoscaling

You will want to scale up resources when your application load increases, and scale down when traffic to your application decreases. This can be achieved in Kubernetes by manual or automated scaling. The most practical, optimized option is to let Kubernetes automatically scale resources needed by a containerized application.

Application management

Changes to applications, e.g., new features and bug fixes, are usually baked into a container image with a new tag. You can easily roll out those changes across all containers running them using Kubernetes' convenient replication feature. If needed, Kubernetes also allows for rolling back to a previous application version in case of a blocking bug or if a security vulnerability is detected.

Persistent storage

Containers offer only a temporary filesystem. Upon restart of the container, all data written to the filesystem is lost. Depending on the nature of your application, you may need to persist data for longer, for example, if your application interacts with a database. Kubernetes offers the ability to mount storage required by application workloads.

Networking

To support a microservices architecture, the container orchestrator needs to allow for communication between containers, and from end users to containers from outside of the cluster. Kubernetes employs internal and external load balancing for routing network traffic.

High-Level Architecture

Architecturally, a Kubernetes cluster consists of control plane nodes and worker nodes, as shown in [Figure 2-1](#). Each node runs on infrastructure provisioned on a physical or virtual machine, or in the cloud. The number of nodes you want to add to the cluster and their topology depends on the application resource needs.

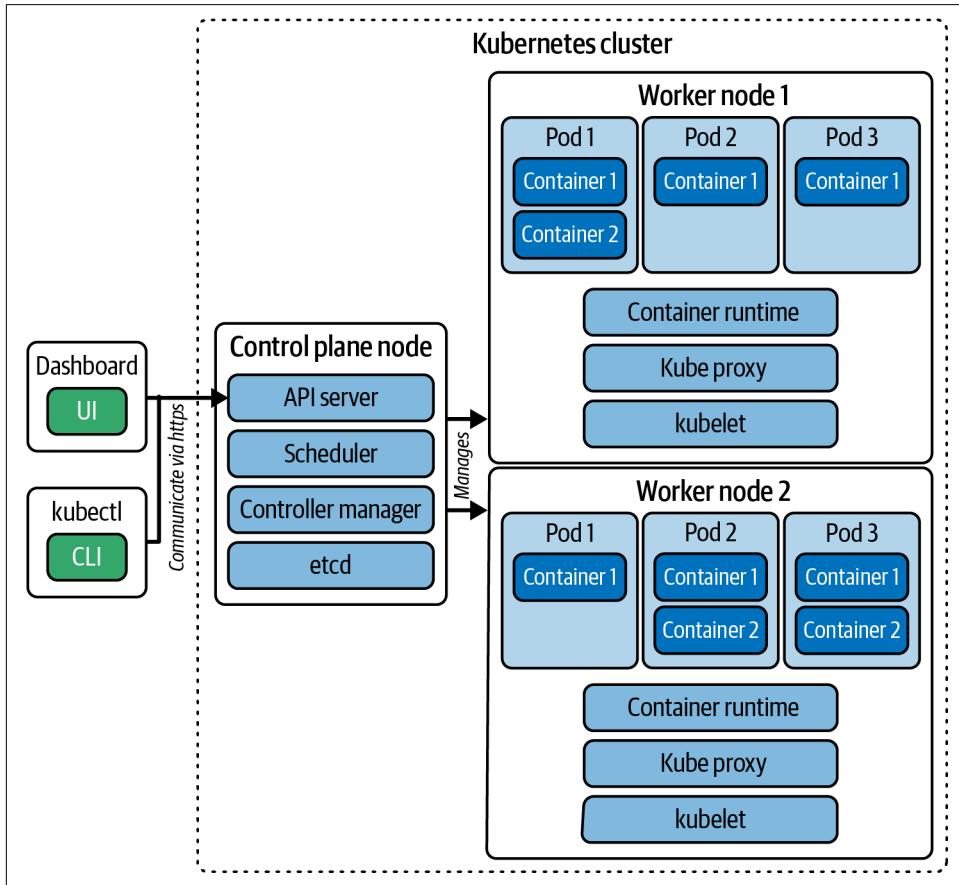


Figure 2-1. Kubernetes cluster nodes and components

Control plane nodes and worker nodes have specific responsibilities:

Control plane node

This node exposes the Kubernetes API through the API server and manages the nodes that make up the cluster. It also responds to cluster events, for example, when the end user requested to scale up the number of Pods to distribute the load for an application. Production clusters employ a [highly available \(HA\) architecture](#) that usually involves three or more control plane nodes.

Worker node

The worker node executes workload in containers managed by Pods. Every worker node needs a container runtime engine installed on the host machine to be able to manage containers.

In the next two sections, we'll talk about the essential components embedded in those nodes to fulfill their tasks. Add-ons like cluster DNS are not discussed explicitly here. See the [Kubernetes documentation](#) for more details.

Control Plane Node Components

The control plane node requires a specific set of components to perform its job. The following list of components will give you an overview:

API server

The API server exposes the API endpoints clients use to communicate with the Kubernetes cluster. For example, if you execute the tool `kubectl`, a command-line based Kubernetes client, you will make a RESTful API call to an endpoint exposed by the API server as part of its implementation. The API processing procedure inside of the API server will ensure aspects like authentication, authorization, and admission control. For more information on that topic, see [Chapter 17](#).

Scheduler

The scheduler is a background process that watches for new Kubernetes Pods with no assigned nodes and assigns them to a worker node for execution.

Controller manager

The controller manager watches the state of your cluster and implements changes where needed. For example, if you make a configuration change to an existing object, the controller manager will try to bring the object into the desired state.

Etcd

Cluster state data needs to be persisted over time so it can be reconstructed upon a node or even a full cluster restart. That's the responsibility of `etcd`, an open source software Kubernetes integrates with. At its core, etcd is a key-value store used to persist all data related to the Kubernetes cluster.

Common Node Components

Kubernetes employs components that are leveraged by all nodes independent of their specialized responsibility:

Kubelet

The kubelet runs on every node in the cluster; however, it makes the most sense to exist on a worker node. The reason is that the control plane node usually doesn't execute workload, and the worker node's primary responsibility is to run

workload. The kubelet is an agent that makes sure that the necessary containers are running in a Pod. You could say that the kubelet is the glue between Kubernetes and the container runtime engine and ensures that containers are running and healthy. We'll have a touch point with the kubelet in [Chapter 14](#).

Kube proxy

The kube proxy is a network proxy that runs on each node in a cluster to maintain network rules and enable network communication. In part, this component is responsible for implementing the Service concept covered in [Chapter 21](#).

Container runtime

As mentioned earlier, the container runtime is the software responsible for managing containers. Kubernetes can be configured to choose from a range of different container runtime engines. While you can install a container runtime engine on a control plane, it's not necessary as the control plane node usually doesn't handle workload. We'll use a container runtime in [Chapter 4](#) to create a container image and run a container with the produced image.

Advantages

This chapter points out a couple of advantages of Kubernetes, which are summarized here:

Portability

A container runtime engine can manage a container independent of its runtime environment. The container image bundles everything it needs to work, including the application's binary or code, its dependencies, and its configuration. Kubernetes can run applications in a container in on-premise and cloud environments. As an administrator, you can choose the platform you think is most suitable to your needs without having to rewrite the application. Many cloud offerings provide product-specific, opt-in features. While using product-specific features helps with operational aspects, be aware that they will diminish your ability to switch easily between platforms.

Resilience

Kubernetes is designed as a declarative state machine. Controllers are reconciliation loops that watch the state of your cluster, then make or request changes where needed. The goal is to move the current cluster state closer to the desired state.

Scalability

Enterprises run applications at scale. Just imagine how many software components retailers like Amazon, Walmart, or Target need to operate to run their businesses. Kubernetes can scale the number of Pods based on demand or automatically according to resource consumption or historical trends.

API based

Kubernetes exposes its functionality through APIs. We learned that every client needs to interact with the API server to manage objects. It is easy to implement a new client that can make RESTful API calls to exposed endpoints.

Extensibility

The API aspect stretches even further. Sometimes, the core functionality of Kubernetes doesn't fulfill your custom needs, but you can implement your own extensions to Kubernetes. With the help of specific extension points, the Kubernetes community can build custom functionality according to their requirements, e.g., monitoring or logging solutions.

Summary

Kubernetes is software for managing containerized applications at scale. Every Kubernetes cluster consists of at least a single control plane node and a worker node. The control plane node is responsible for scheduling the workload and acts as the single entrypoint to manage its functionality. Worker nodes handle the workload assigned to them by the control plane node.

Kubernetes is a production-ready runtime environment for companies wanting to operate microservice architectures while also supporting nonfunctional requirements like scalability, security, load balancing, and extensibility.

The next chapter will explain how to interact with a Kubernetes cluster using the command-line tool `kubectl`. You will learn how run it to manage objects, an essential skill for acing the exam.

Interacting with Kubernetes

As an application developer, you will want to interact with the Kubernetes cluster to manage objects that operate your application. Every call to the cluster is accepted and processed by the API server component. There are various ways to perform a call to the API server. For example, you can use a [web-based dashboard](#), a command-line tool like `kubectl`, or a direct HTTPS request to the RESTful API endpoints.

The exam does not test the use of a visual user interface for interacting with the Kubernetes cluster. Your only client for solving exam questions is `kubectl`. This chapter will touch on the Kubernetes API primitives and objects, as well as the different ways to manage objects with `kubectl`.

API Primitives and Objects

Kubernetes primitives are the basic building blocks anchored in the Kubernetes architecture for creating and operating an application on the platform. Even as a beginner to Kubernetes, you might have heard of the terms Pod, Deployment, and Service, all of which are Kubernetes primitives. There are many more that serve a dedicated purpose in the Kubernetes architecture.

To draw an analogy, think back to the concepts of object-oriented programming. In object-oriented programming languages, a class defines the blueprint of a real-world functionality: its properties and behavior. A Kubernetes primitive is the equivalent of a class. The instance of a class in object-oriented programming is an object, managing its own state and having the ability to communicate with other parts of the system. Whenever you create a Kubernetes object, you produce such an instance.

For example, a Pod in Kubernetes is the class of which there can be many instances with their own identity. Every Kubernetes object has a system-generated unique identifier (also known as UID) to clearly distinguish between the entities of a system.

Later, we'll look at the properties of a Kubernetes object. [Figure 3-1](#) illustrates the relationship between a Kubernetes primitive and an object.

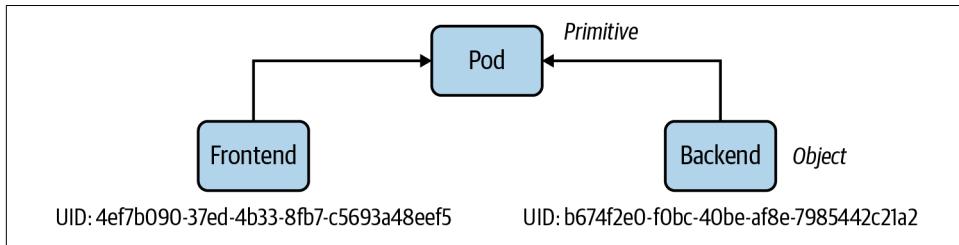


Figure 3-1. Kubernetes object identity

Every Kubernetes primitive follows a general structure, which you can observe if you look deeper at a manifest of an object, as shown in [Figure 3-2](#). The primary markup language used for a Kubernetes manifest is YAML.

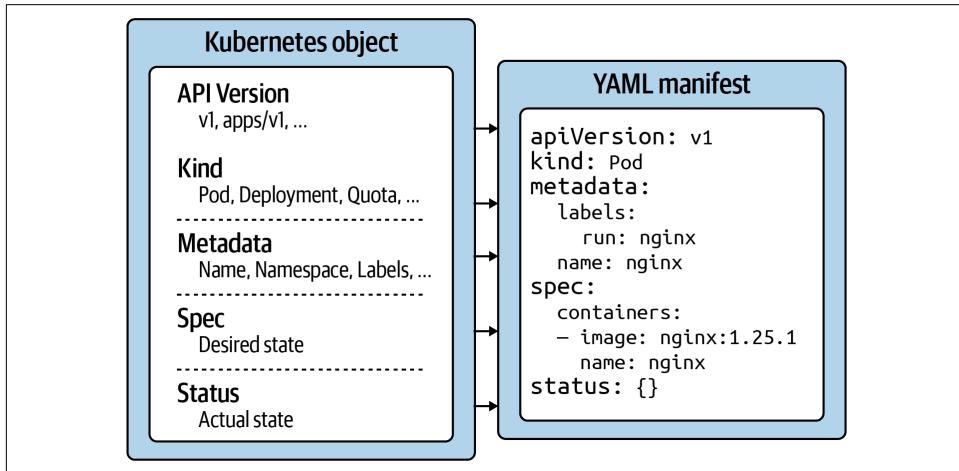


Figure 3-2. Kubernetes object structure

Let's look at each section and its relevance within the Kubernetes system:

API version

The Kubernetes API version defines the structure of a primitive and uses it to validate the correctness of the data. The API version serves a similar purpose as XML schemas to an XML document or JSON schemas to a JSON document. The version usually undergoes a maturity process—for example, from alpha to beta to final. Sometimes you see different prefixes separated by a slash (apps). You can list the API versions compatible with your cluster version by running the command `kubectl api-versions`.

Kind

The kind defines the type of primitive—e.g., a Pod or a Service. It ultimately answers the question, “What kinds of resource are we dealing with here?”

Metadata

Metadata describes higher-level information about the object—e.g., its name, what namespace it lives in, or whether it defines labels and annotations. This section also defines the UID.

Spec

The specification (“spec” for short) declares the desired state—e.g., how should this object look after it has been created? Which image should run in the container, or which environment variables should be set?

Status

The status describes the actual state of an object. The Kubernetes controllers and their reconciliation loops constantly try to transition a Kubernetes object from the desired state into the actual state. The object has not yet been materialized if the YAML status shows the value `[]`.

With this basic structure in mind, let’s look at how to create a Kubernetes object with the help of `kubectl`.

Using `kubectl`

`kubectl` is the primary tool for interacting with the Kubernetes clusters from the command line. The exam is exclusively focused on the use of `kubectl`. Therefore, it’s paramount to understand its ins and outs and practice its use heavily.

This section provides you with a brief overview of its typical usage pattern. Let’s start by looking at the syntax for running commands. A `kubectl` execution consists of a command, a resource type, a resource name, and optional command line flags:

```
$ kubectl [command] [TYPE] [NAME] [flags]
```

The command specifies the operation you’re planning to run. Typical commands are verbs like `create`, `get`, `describe`, or `delete`. Next, you’ll need to provide the resource type you’re working on, either as a full resource type or its short form. For example, you could work on a `service` here or use the short form, `svc`.

The name of the resource identifies the user-facing object identifier, effectively the value of `metadata.name` in the YAML representation. Be aware that the object name is not the same as the UID. The UID is an autogenerated, Kubernetes-internal object reference that you usually don’t have to interact with. The name of an object has to be unique across all objects of the same resource type within a namespace.

Finally, you can provide zero to many command line flags to describe additional configuration behavior. A typical example of a command-line flag is the `--port` flag, which exposes a Pod's container port.

Figure 3-3 shows a full `kubectl` command in action.

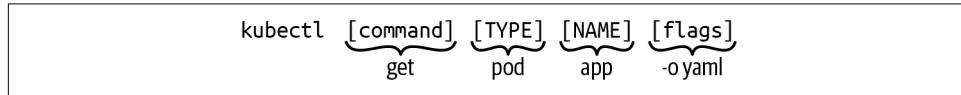


Figure 3-3. *Kubectl usage pattern*

Over the course of this book, we'll explore the `kubectl` commands that will make you the most productive during the exam. There are many more, however, and they usually go beyond the ones you'd use on a day-to-day basis as an application developer. Next up, we'll have a deeper look at the `create` command, the imperative way to create a Kubernetes object. We'll also compare the imperative object creation approach with the declarative approach.

Managing Objects

You can create objects in a Kubernetes cluster in two ways: imperatively or declaratively. The following sections describe each approach, including their benefits, drawbacks, and use cases.

Imperative Object Management

Imperative object management does not require a manifest definition. You'll use `kubectl` to drive the creation, modification, and deletion of objects with a single command and one or many command-line options. See the [Kubernetes documentation](#) for a more detailed description of imperative object management.

Creating objects

Use the `run` or `create` command to create an object on the fly. Any configuration needed at runtime is provided by command-line options. The benefit of this approach is the fast turnaround time without the need to wrestle with YAML structures. The following `run` command creates a Pod named `frontend` that executes the container image `nginx:1.24.0` in a container with the exposed port 80:

```
$ kubectl run frontend --image=nginx:1.24.0 --port=80
pod/frontend created
```

Updating objects

The configuration of live objects can still be modified. `kubectl` supports this use case by providing the `edit` or `patch` command.

The `edit` command opens an editor with the raw configuration of the live object. Changes to the configuration will be applied to the live object after exiting the editor. The command will open the editor defined by the `KUBE_EDITOR`, or `EDITOR` environment variables, or fall back to `vi` for Linux or `notepad` for Windows. This command demonstrates the use of the `edit` command for the Pod live object named `frontend`:

```
$ kubectl edit pod frontend
```

The `patch` command allows for fine-grained modification of a live object on an attribute level using a JSON merge patch. The following example illustrates the use of `patch` command to update the container image tag assigned to the Pod created earlier. The `-p` flag defines the JSON structure used to modify the live object:

```
$ kubectl patch pod frontend -p '{"spec":{"containers":[{"name":"frontend", "image":"nginx:1.25.1"}]}'  
pod/frontend patched
```

Deleting objects

You can delete a Kubernetes object at any time. During the exam, the need may arise if you made a mistake while solving a problem and want to start from scratch to ensure a clean slate. In a production Kubernetes environment, you'll want to delete objects that are no longer needed. The following `delete` command deletes the Pod object by its name `frontend`:

```
$ kubectl delete pod frontend  
pod "frontend" deleted
```

Upon execution of the `delete` command, Kubernetes tries to delete the targeted object gracefully so that there's minimal impact on the end user. If the object cannot be deleted within the default grace period (30 seconds), the `kubelet` attempts to forcefully kill the object.

During the exam, end user impact is not a concern. The most important goal is to complete all tasks in the time granted to the candidate. Therefore, waiting on an object to be deleted gracefully is a waste of time. You can force an immediate deletion of an object with the command-line option with the `--now` option. The following command kills the Pod named `nginx` using a `SIGKILL` signal:

```
$ kubectl delete pod nginx --now
```

Declarative Object Management

Declarative object management requires one or several manifests in the format of YAML or JSON describing the desired state of an object. You create, update, and delete objects using this approach.

The benefit of using the declarative method is reproducibility and improved maintenance, as the file is checked into version control in most cases. The declarative approach is the recommended way to create objects in production environments.

More information on declarative object management can be found in the [Kubernetes documentation](#).

Creating objects

The declarative approach creates objects from a manifest (in most cases, a YAML file) using the `apply` command. The command works by pointing to a file, a directory of files, or a file referenced by an HTTP(S) URL using the `-f` option. If one or more of the objects already exist, the command will synchronize the changes made to the configuration with the live object.

To demonstrate the functionality, we'll assume the following directories and configuration files. The following commands create objects from a single file, from all files within a directory, and from all files in a directory recursively. Refer to files in the book's GitHub repository if you want to give it a try. Later chapters will explain the purpose of the primitives used here:

```
.  
├── app-stack  
│   ├── mysql-pod.yaml  
│   ├── mysql-service.yaml  
│   ├── web-app-pod.yaml  
│   └── web-app-service.yaml  
└── nginx-deployment.yaml  
    └── web-app  
        ├── config  
        │   ├── db-configmap.yaml  
        │   └── db-secret.yaml  
        └── web-app-pod.yaml
```

Creating an object from a single file:

```
$ kubectl apply -f nginx-deployment.yaml  
deployment.apps/nginx-deployment created
```

Creating objects from multiple files within a directory:

```
$ kubectl apply -f app-stack/  
pod/mysql-db created  
service/mysql-service created
```

```
pod/web-app created
service/web-app-service created
```

Creating objects from a recursive directory tree containing files:

```
$ kubectl apply -f web-app/ -R
configmap/db-config configured
secret/db-creds created
pod/web-app created
```

Creating objects from a file referenced by an HTTP(S) URL:

```
$ kubectl apply -f https://raw.githubusercontent.com/bmuschko/\
ckad-study-guide/master/ch03/object-management/nginx-deployment.yaml
deployment.apps/nginx-deployment created
```

The `apply` command keeps track of the changes by adding or modifying the annotation with the key `kubectl.kubernetes.io/last-applied-configuration`. Here's an example of the annotation in the output of the `get pod` command:

```
$ kubectl get pod web-app -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":{}}, \
      "labels":{"app":"web-app","name":"web-app","namespace":"default"}, \
      "spec":{"containers":[{"envFrom":[{"configMapRef":{"name":"db-config"}}, \
      {"secretRef":{"name":"db-creds"}]}],"image":"bmuschko/web-app:1.0.1", \
      "name":"web-app","ports":[{"containerPort":3000,"protocol":"TCP"}]}, \
      "restartPolicy":"Always"}
```

...

Updating objects

Updating an existing object is done with the same `apply` command. All you need to do is to change the configuration file and then run the command against it. **Example 3-1** modifies the existing configuration of a Deployment in the file `nginx-deployment.yaml`. We added a new label with the key `team` and changed the number of replicas from 3 to 5.

Example 3-1. Modified configuration file for a Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
    team: red
spec:
```

```
replicas: 5
```

```
...
```

The following command applies the changed configuration file. As a result, the number of Pods controlled by the underlying ReplicaSet is 5:

```
$ kubectl apply -f nginx-deployment.yaml
deployment.apps/nginx-deployment configured
```

The Deployment's `kubectl.kubernetes.io/last-applied-configuration` annotation reflects the latest change to the configuration:

```
$ kubectl get deployment nginx-deployment -o yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment","metadata":{"annotations":{}}, \
      "labels":{"app":"nginx","team":"red"},"name":"nginx-deployment", \
      "namespace":"default","spec":{"replicas":5,"selector":{"matchLabels": \
      {"app":"nginx"}}, "template":{"metadata":{"labels":{"app":"nginx"}}, \
      "spec":{"containers":[{"image":"nginx:1.14.2","name":"nginx", \
      "ports":[{"containerPort":80}]}]}}}}
```

```
...
```

Deleting objects

While you can delete objects using the `apply` command by providing the options `--prune -l <labels>`, it is recommended to delete an object using the `delete` command and point it to the configuration file. The following command deletes a Deployment and the objects it controls (ReplicaSet and Pods):

```
$ kubectl delete -f nginx-deployment.yaml
deployment.apps "nginx-deployment" deleted
```

You can use the `--now` option to forcefully delete Pods, as described in “[Deleting objects](#)” on page 25.

Hybrid Approach

Sometimes you may want to go with a hybrid approach. You can start by using the imperative method to produce a manifest file without actually creating an object. You do so by executing the `run` or `create` command with the command-line options `-o yaml` and `--dry-run=client`:

```
$ kubectl run frontend --image=nginx:1.25.1 --port=80 \
-o yaml --dry-run=client > pod.yaml
```

You can now use the generate YAML manifest as a starting point to make further modifications before creating the object. Simply open the file with an editor, change the content, and execute the declarative `apply` command:

```
$ vim pod.yaml  
$ kubectl apply -f pod.yaml  
pod/frontend created
```

Which Approach to Use?

During the exam, using imperative commands is the most efficient and quick way to manage objects. Not all configuration options are exposed through command-line flags, which may force you into using the declarative approach. The hybrid approach can help here.



GitOps and Kubernetes

GitOps is a practice that leverages source code checked into Git repositories to automate infrastructure management, specifically in cloud-native environments powered by Kubernetes. Tools such as [Argo CD](#) and [Flux](#) implement GitOps principles to deploy applications to Kubernetes through a declarative approach. Teams responsible for overseeing real-world Kubernetes clusters and the applications within them are highly likely to adopt the declarative approach.

While creating objects imperatively can optimize the turnaround time, in a real-world Kubernetes environment you'll most certainly want to use the declarative approach. A YAML manifest file represents the ultimate source of truth of a Kubernetes object. Version-controlled files can be audited and shared, and they store a history of changes in case you need to revert to a previous revision.

Summary

Kubernetes represents its functionality for deploying and operating a cloud-native application with the help of primitives. Each primitive follows a general structure: the API version, the kind, the metadata, and the desired state of the resources, also called the spec. Upon creation or modification of the object, the Kubernetes scheduler automatically tries to ensure that the actual state of the object follows the defined specification. Every live object can be inspected, edited, and deleted.

`Kubectl` acts as a CLI-based client to interact with the Kubernetes cluster. You can use its commands and flags to manage Kubernetes objects. The imperative approach provides a fast turnaround time for managing objects with a single command, as long as you memorize the available flags. More complex configuration calls for the use of a

YAML manifest to define a primitive. Use the declarative command to instantiate objects from that definition. The YAML manifest is usually checked into version control and offers a way to track changes to the configuration.

PART II

Application Design and Build

The domain *Application Design and Build* summarizes all foundational topics important to designing a process or an application for the purpose of operating it in a Kubernetes Pod.

The following chapters cover these concepts:

- [Chapter 4](#) explains basic container terminology. The content uses Docker Engine to introduce commands for defining, building, running, and modifying container images.
- [Chapter 5](#) takes an existing container image and runs it in a Pod, the essential primitive for running applications in Kubernetes in a namespace.
- [Chapter 6](#) touches on modeling batch workload and periodically executed workload in Kubernetes.
- [Chapter 7](#) explains how to share data between containers running in a single Pod using ephemeral Volumes. The chapter will also explain how to store long-lived data with the help of persistent Volumes.
- [Chapter 8](#) illustrates the use cases for wanting to run multiple containers in a Pod and how to implement them with well-established design patterns.
- [Chapter 9](#) introduces how to assign labels and annotations to objects. After reading the chapter, you will understand the importance of labels to query, sort, and filter objects. Compared to labels, annotations add human-readable-only meta information to objects.

CHAPTER 4

Containers

Kubernetes is a container orchestrator that uses a container runtime to instantiate containers inside of Pods. Many Kubernetes clusters with version 1.24 or later use the container runtime [containerd](#).



Container runtime used on a Kubernetes node

You can fetch information about the container runtime used on any node of a Kubernetes cluster. Simply look at the output of the CONTAINER-RUNTIME column produced by running the command `kubectl get nodes -o wide`. Check the [Kubernetes documentation](#) to learn more about configuring a container runtime for a cluster.

For the exam, you are expected to understand the practical aspects of defining, building, and publishing container images, which this chapter covers. We'll also touch on running a container image inside of a container. For all of those operations, we'll use Docker Engine as the example container runtime though similar functionality is provided by other implementations.

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objective:

- Define, build, and modify container images

The discussion on containers in this book only scratches the surface. There's a lot more information on this topic if you want to fully immerse yourself. I can recommend the book *Docker: Up & Running* (O'Reilly) by Sean P. Kane and Karl Matthias for a detailed explanation of Docker.

Container Terminology

A *container* packages an application into a single unit of software including its runtime environment and configuration. This unit of software usually includes the operating system, the application's source code or the binary, its dependencies, and other needed system tools. The declared goal of a container is to decouple the runtime environment from the application to avoid the "but it works on my machine" problem.

The *container runtime engine* is the software component that can run containers on a host operating system. Examples include [Docker Engine](#) or [containerd](#). A *container orchestrator* uses a container runtime engine to instantiate a container while adding sophisticated features like scalability and networking across the workload. Kubernetes is an example of container orchestrators. Other tools like [Nomad](#) are capable of scheduling various types of workload including containers.

The process of bundling an application into a container is called *containerization*. Containerization works based on instructions defined in a *container file*. The Docker community calls this a Dockerfile. The Dockerfile explicitly spells out what needs to happen when the software is built. The result of the operation is a *container image*.

The container image is usually published to a *container registry* for consumption by other stakeholders. [Docker Hub](#) is the primary registry for container images available for public use. Other public registries like GCR and Quay are available. [Figure 4-1](#) illustrates the concepts in the context of containerizing an application.

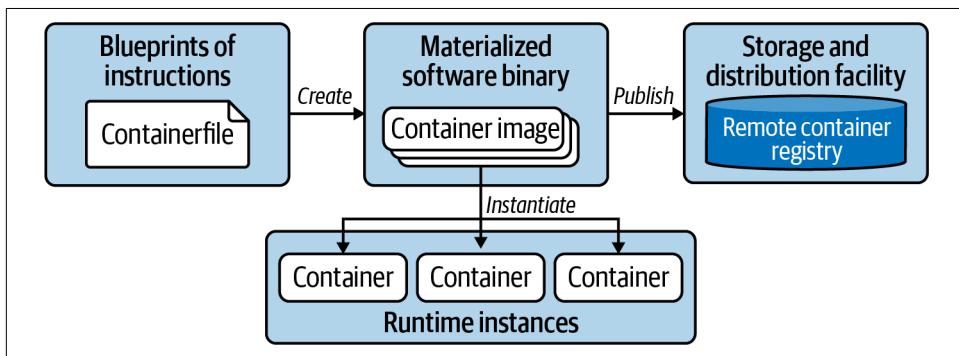


Figure 4-1. Containerization process

To summarize, the Dockerfile is a blueprint of how the software should be packaged, the image is the artifact produced by the process, and the container is a running instance of the image serving the application. We'll look at a more concrete example next.

Containerizing a Java-Based Application

Let's assume we want to containerize a web application written in Java. The application doesn't write core functionality from scratch but uses the [Spring Boot framework](#) as an external library. In addition, we want to control the runtime behavior with the help of environment variables. For example, you may want to provide URLs and credentials to connect to other services like a database. We'll talk through the process step by step and execute the relevant Docker commands from the terminal. If you want to follow along, you can download a sample application from the project generator [Spring Initializr](#).

Writing a Dockerfile

Before we can create the image, we have to write a Dockerfile. The Dockerfile can reside in any directory and is a plain-text file. The instructions that follow use the Azul JRE distribution of Java 21 as the base image. A base image contains the operating system and the necessary tooling, in this case Java.

Moreover, we include the binary file, an executable Java archive (JAR), into the directory `/app` of the image. Finally, we define the Java command that executes the program and expose the port 8080 to make the application accessible when run in a container. [Example 4-1](#) outlines a sample Dockerfile.

Example 4-1. Dockerfile for building a Java application

```
FROM azul/zulu-openjdk:21-jre          ①  
WORKDIR /app                           ②  
COPY target/java-hello-world-0.0.1.jar java-hello-world.jar ③  
ENTRYPOINT ["java", "-jar", "/app/java-hello-world.jar"] ④  
EXPOSE 8080                            ⑤
```

- ① Defines the base image.
- ② Sets the working directory of a container. Any RUN, CMD, ADD, COPY, or ENTRYPOINT instruction will be executed in the specified working directory.
- ③ Copies the JAR containing the compiled application code into the working directory.

- ④ Sets the default command that executes when a container starts from an image.
- ⑤ Documents the network port(s) the container should listen on.

While writing Dockerfile looks straightforward to beginners, optimizing the container image for a small footprint and security aspects isn't. You can find a more detailed list of best practices for writing Dockerfiles in the [Docker documentation](#).

Building the Container Image

With the Dockerfile in place, we can create the image. The following command provides the name of the image and the tag. The last argument points to the context directory. A context directory contains the Dockerfile as well as any directories and files to be included in the image. Here, the context directory is the current directory we reside in referenced by ".":

```
$ docker build -t java-hello-world:1.1.0 .
[+] Building 2.0s (9/9) FINISHED
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 284B
=> [internal] load metadata for docker.io/azul/zulu-openjdk:21-jre
=> [auth] azul/zulu-openjdk:pull token for registry-1.docker.io
=> [1/3] FROM docker.io/azul/zulu-openjdk:21-jre@sha256:d1e675cac0e5...
=> => resolve docker.io/azul/zulu-openjdk:21-jre@sha256:d1e675cac0e5...
=> => sha256:d1e675cac0e5ce9604283df2a6600d3b46328d32d83927320757ca7...
=> => sha256:67aa3090031eac26c946908c33959721730e42f9195f4f70409e4ce...
=> => sha256:ba408da684370e4d8448bec68b36fadf15c3819b282729df3bc8494...
=> [internal] load build context
=> => transferring context: 19.71MB
=> [2/3] WORKDIR /app
=> [3/3] COPY target/java-hello-world-0.0.1.jar java-hello-world.jar
=> exporting to image
=> => exporting layers
=> => writing image sha256:4b676060678b63de137536da24a889fc9d2d5fe0c...
=> => naming to docker.io/library/java-hello-world:1.1.0
```

What's Next?

[View a summary of image vulnerabilities and recommendations → ...](#)

Listing Container Images

As indicated by the terminal output, the image has been created. You might have noticed that the base image has been downloaded as part of the process. The generated image can be found in your local Docker Engine cache by running the following command:

```
$ docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
java-hello-world 1.1.0   4b676060678b  49 seconds ago  342MB
```

Running the Container

It's time to run the application in a container. The `run` command points to an image and executes its logic in a container:

```
$ docker run -d -p 8080:8080 java-hello-world:1.1.0
b0ee04accf078ea7c73cf3be0f9d1ac6a099ac4e0e903773bc6bf6258acbb66
```

We told the command to forward the port 8080 accessible on localhost to the container port 8080 using the `-p` CLI option. The `-d` CLI option runs the container in the background, which means it will detach from the container and return to the terminal prompt. This means we should now be able to resolve the application's endpoint from the local machine. As the following command shows, a simple `curl` to the root context path renders the message "Hello World!" :

```
$ curl localhost:8080
Hello World!
```

Listing Containers

Any running containers can be listed to display their runtime properties. The following command renders the container started earlier. The output includes the container ID for later reference. Add the flag `-a` to render terminated containers as well:

```
$ docker container ls
CONTAINER ID      IMAGE      COMMAND      ...
b0ee04accf07    java-hello-world:1.1.0    "java -jar /app/java..."    ...
```

Interacting with the Container

Once the container has been started, you can interact with it. All you need is the container ID. Use the `logs` command to inspect log messages produced by the application. Inspecting logs can be helpful for troubleshooting. The following command renders the log messages produced by Spring Boot upon container startup:

```
$ docker logs b0ee04accf07
...
2023-06-19 21:06:27.757  INFO 1 --- [nio-8080-exec-1] \
o.a.c.c.C.[Tomcat].[localhost].[/]           : Initializing \
Spring DispatcherServlet 'dispatcherServlet'
2023-06-19 21:06:27.757  INFO 1 --- [nio-8080-exec-1] \
o.s.web.servlet.DispatcherServlet          : Initializing \
Servlet 'dispatcherServlet'
2023-06-19 21:06:27.764  INFO 1 --- [nio-8080-exec-1] \
o.s.web.servlet.DispatcherServlet          : Completed \
initialization in 7 ms
```

You can dig deeper into the internals of running containers if the container image is packaged with a command-line shell. For example, you may want to inspect files consumed or produced by the application. Use the `exec` command to run a command in the container. The flag `-it` allows for iterating with the container until you are ready to exit out of it. The following command opens an iterative bash shell to the running container:

```
$ docker exec -it b0ee04accf07 bash
root@b0ee04accf07:/app# pwd
/app
root@b0ee04accf07:/app# exit
exit
```

To leave the interactive bash shell, run the `exit` command. You'll return to the terminal prompt on your host machine.

Publishing the Container Image

To publish an image to a registry, you'll have to do some prework. Most registries require you to provide a prefix that signifies the username or hostname as part of the container image name, which you can achieve with the `tag` command.

For example, Docker Hub requires you to provide the username. My username is `bmuschko` and therefore I have to retag my image before pushing it:

```
$ docker tag java-hello-world:1.1.0 bmuschko/java-hello-world:1.1.0
```

The `tag` command does not create a copy of the container image. It simply adds another identifier pointing to the existing container image, as shown in the following output. The image ID and size of the container image is the same for both entries:

```
$ docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
bmuschko/java-hello-world  1.1.0   4b676060678b  6 minutes ago  342MB
java-hello-world     1.1.0   4b676060678b  6 minutes ago  342MB
```

If the registry is protected, you'll also have to provide the credentials. For Docker Hub, we are logging in with username:

```
$ docker login --username=bmuschko
Password: *****
Login Succeeded
```

Finally, you can push the image to the registry using the `push` command:

```
$ docker push bmuschko/java-hello-world:1.1.0
The push refers to repository [docker.io/bmuschko/java-hello-world]
a7b86a39983a: Pushed
df1b2befef5f0: Pushed
e4db97f0e9ef: Mounted from azul/zulu-openjdk
```

```
8e87ff28f1b5: Mounted from azul/zulu-openjdk  
1.1.0: digest: sha256:6a5069bd9396a7edeb10bf8e24ab251df434c121f8f4293c2d3ef...
```

You can discover the published container image through the Docker Hub web page, as shown in [Figure 4-2](#). The “Tags” tab lists all available tags for the image including their details and quick reference to the `docker` command for pulling the image.

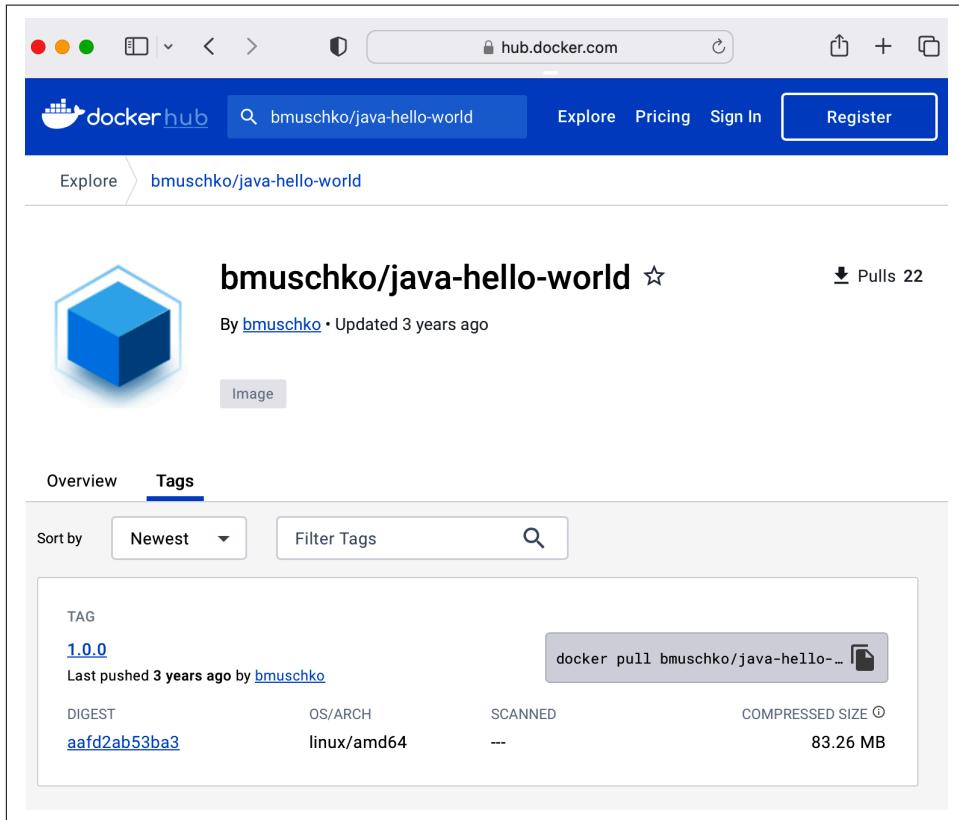


Figure 4-2. Discovering container images on Docker Hub

Anyone with access to the registry can now consume the container image using the `pull` command.

Saving and Loading a Container Image

Instead of publishing a container image to a container registry, you may want to save it to a file. Files can be easily stored and backed up on a shared drive and don’t require a container registry. The `save` command saves one or many images to a tar archive. The resulting archive file contains all parent layers, and all tags + versions.

The following command saves the container image to the file `java-hello-world.tar`:

```
$ docker save -o java-hello-world.tar java-hello-world:1.1.0
```

To load a container image from a tar archive, use the `load` command. The command restores both images and tags. The following command loads the container image from the file `java-hello-world.tar`:

```
$ docker load --input java-hello-world.tar
Loaded image: java-hello-world:1.1.0
```

The image is now available in the cache, as shown by running the `images` command:

```
$ docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
java-hello-world  1.1.0   4b676060678b  7 minutes ago  342MB
```

Going Further

Thus far you have experienced the most common developer workflows: containerizing an application and pushing the image to a registry. There's far more to learn about building and running containers, but that is outside the scope of this book, and we won't dive any deeper here. If you'd like to learn more, a good starting point is the [Docker documentation](#).

Summary

Application developers use containerization to bundle the application code into a container image so that it can be deployed to Kubernetes clusters as a single unit of runnable software. The containerization process defines, builds, runs, and publishes a container image using a container runtime engine. In this chapter, we used Docker Engine to demonstrate the process for a Java-based application; however, the steps involved would look similar for applications written in a different programming language.

Exam Essentials

Gain practical experience with the containerization process

Pods run container images inside of a container. You need to understand how to define, build, run, and publish a container image apart from Kubernetes. Practice the use of the container runtime engine's command-line tool to fulfill the workflow.

Compare the functionality of different container runtime engines

You should get familiar with Docker Engine specifically for understanding the containerization process. At the time of writing, Docker Engine is still the most widely used container runtime engine. Branch out by playing around with other container runtime engines like containerd or Podman.

Familiarize yourself with other workflows

As an application developer, you will deal with defining, building, and modifying container images daily. Container runtime engine support other, less-known features and workflows. It can't hurt to read through the container runtime engine's documentation to gain broader exposure.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. Navigate to the directory `app-a/ch04/containerized-java-app` of the checked-out GitHub repository [bmuschko/ckad-study-guide](#). Inspect the *Dockerfile*.

Build the container image from the *Dockerfile* with the tag `nodejs-hello-world:1.0.0`.

Run a container with the container image. Make the application available on port 80.

Execute a `curl` or `wget` command against the application's endpoint.

Retrieve the container logs.

2. Modify the *Dockerfile* from the previous exercise. Change the base image to the tag `20.4-alpine` and the working directory to `/node`.

Build the container image from the *Dockerfile* with the tag `nodejs-hello-world:1.1.0`.

Ensure that container image has been created by listing it.

3. Pull the container image `alpine:3.18.2` [available on Docker Hub](#).

Save the container image to the file `alpine-3.18.2.tar`.

Delete the container image. Verify the container image is not listable anymore.

Reinstate the container image from the file `alpine-3.18.2.tar`.

Verify that the container image can be listed.

Pods and Namespaces

The most important primitive in the Kubernetes API is the Pod. A Pod lets you run a containerized application. In practice, you'll often encounter a one-to-one mapping between a Pod and a container; however, the use cases discussed in [Chapter 8](#) benefit from declaring more than one container in a single Pod.

In addition to running a container, a Pod can consume other services like storage, configuration data, and much more. Therefore, think of a Pod as a wrapper for running containers while at the same time being able to mix in cross-cutting and specialized Kubernetes features.

Coverage of Curriculum Objectives

The curriculum doesn't explicitly mention coverage of Pods and namespaces. However, you will definitely need to understand those primitives as they are essential for running workload in Kubernetes.

Working with Pods

In this chapter, we will look at working with a Pod running only a single container. We'll discuss all important `kubectl` commands for creating, modifying, interacting, and deleting using imperative and declarative approaches.

Creating Pods

The Pod definition needs to state an image for every container. Upon creating the Pod object, imperatively or declaratively, the scheduler will assign the Pod to a node, and the container runtime engine will check if the container image already exists on

that node. If the image doesn't exist yet, the engine will download it from a container image registry. By default the registry is Docker Hub. As soon as the image exists on the node, the container is instantiated and will run. [Figure 5-1](#) demonstrates the execution flow.

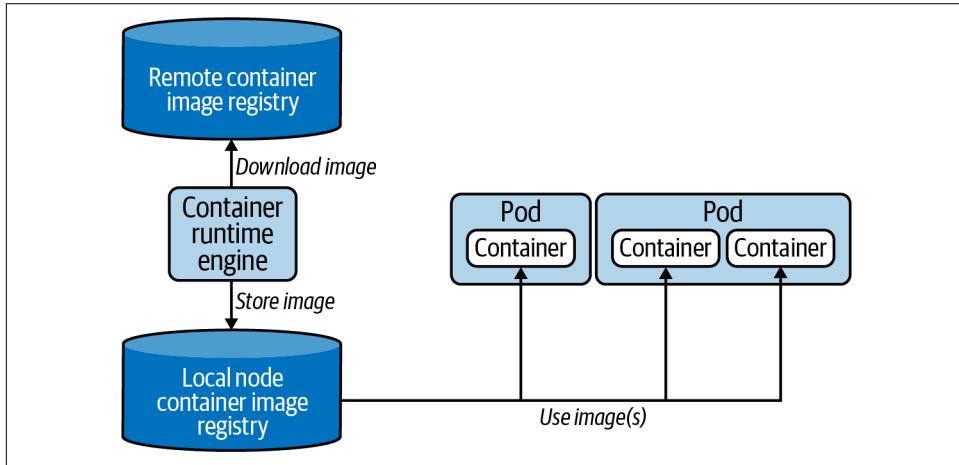


Figure 5-1. Container Runtime Interface interaction with container images

The `run` command is the central entry point for creating Pods imperatively. Let's talk about its usage and the most important command line options you should memorize and practice. Say you wanted to run a [Hazelcast instance](#) inside of a Pod. The container should use the latest [Hazelcast image](#), expose port 5701, and define an environment variable. In addition, we'll also want to assign two labels to the Pod. The following imperative command combines this information and does not require any further editing of the live object:

```
$ kubectl run hazelcast --image=hazelcast/hazelcast:5.1.7 \
--port=5701 --env="DNS_DOMAIN=cluster" --labels="app=hazelcast,env=prod"
```

The `run` command offers a wealth of command line options. Execute the `kubectl run --help` or refer to the Kubernetes documentation for a broad overview. For the exam, you'll not need to understand every command. [Table 5-1](#) lists the most commonly used options.

Table 5-1. Important kubectl run command line options

Option	Example value	Description
<code>--image</code>	<code>nginx:1.25.1</code>	The image for the container to run.
<code>--port</code>	<code>8080</code>	The port that this container exposes.
<code>--rm</code>	<code>N/A</code>	Deletes the Pod after command in the container finishes. See " Creating a Temporary Pod " on page 49 for more information.

Option	Example value	Description
--env	PROFILE=dev	The environment variables to set in the container.
--labels	app=frontend	A comma-separated list of labels to apply to the Pod. Chapter 9 explains labels in more detail.

Some developers are more used to creating Pods from a YAML manifest. Probably you're already accustomed to the declarative approach because you're using it at work. You can express the same configuration for the Hazelcast Pod by opening the editor, copying a Pod YAML code snippet from the Kubernetes online documentation, and modifying it to your needs. [Example 5-1](#) shows the Pod manifest saved in the file `pod.yaml`:

Example 5-1. Pod YAML manifest

```
apiVersion: v1
kind: Pod
metadata:
  name: hazelcast          ①
  labels:                   ②
    app: hazelcast
    env: prod
spec:
  containers:
    - name: hazelcast
      image: hazelcast/hazelcast:5.1.7 ③
      env:                      ④
        - name: DNS_DOMAIN
          value: cluster
      ports:                    ⑤
        - containerPort: 5701
```

- ① Assigns the name of `hazelcast` to the Pod.
- ② Specifies labels to the Pod.
- ③ Declares the container image to be executed in the container of the Pod.
- ④ Injects one or many environment variables to the container.
- ⑤ Number of port to expose on the Pod's IP address.

Creating the Pod from the manifest is straightforward. Simply use the `create` or `apply` command, as shown here and explained in “[Managing Objects](#)” on page 24:

```
$ kubectl apply -f pod.yaml
pod/hazelcast created
```

Listing Pods

Now that you have created a Pod, you can further inspect its runtime information. The `kubectl` command offers a command for listing all Pods running in the cluster: `get pods`. The following command renders the Pod named `hazelcast`:

```
$ kubectl get pods  
NAME      READY   STATUS    RESTARTS   AGE  
hazelcast  1/1     Running   0          17s
```

Real-world Kubernetes clusters can run hundreds of Pods at the same time. If you know the name of the Pod of interest, it's often easier to query by name. You would still see only a single Pod:

```
$ kubectl get pods hazelcast  
NAME      READY   STATUS    RESTARTS   AGE  
hazelcast  1/1     Running   0          17s
```

Pod Life Cycle Phases

Because Kubernetes is a state engine with asynchronous control loops, it's possible that the status of the Pod doesn't show a `Running` status right away when listing the Pods. It usually takes a couple of seconds to retrieve the image and start the container. Upon Pod creation, the object goes through several [life cycle phases](#), as shown in [Figure 5-2](#).

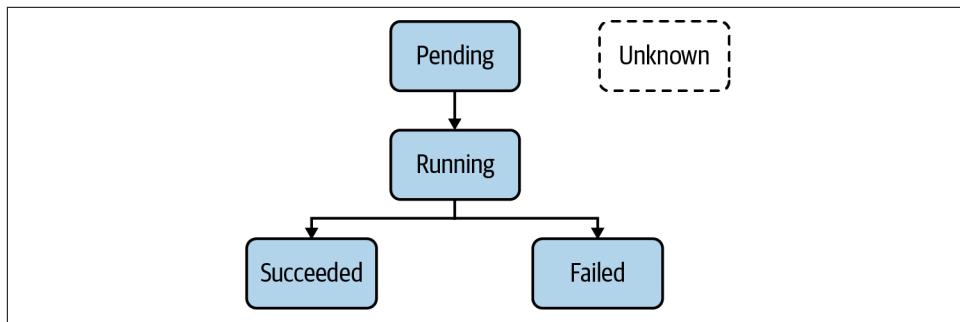


Figure 5-2. Pod life cycle phases

Understanding the implications of each phase is important as it gives you an idea about the operational status of a Pod. For example, during the exam you may be asked to identify a Pod with an issue and further debug the object. [Table 5-2](#) describes all Pod life cycle phases.

Table 5-2. Pod life cycle phases

Option	Description
Pending	The Pod has been accepted by the Kubernetes system, but one or more of the container images has not been created.
Running	At least one container is still running or is in the process of starting or restarting.
Succeeded	All containers in the Pod terminated successfully.
Failed	Containers in the Pod terminated; at least one failed with an error.
Unknown	The state of Pod could not be obtained.

The Pod life cycle phases should not be confused with container states within a Pod. Containers can have one of the three possible states: Waiting, Running, and Terminated. You can read more about container states in the [Kubernetes documentation](#).

Rendering Pod Details

The rendered table produced by the `get` command provides high-level information about a Pod. But what if you needed a deeper look at the details? The `describe` command can help:

```
$ kubectl describe pods hazelcast
Name:           hazelcast
Namespace:      default
Priority:       0
PriorityClassName: <none>
Node:           docker-desktop/192.168.65.3
Start Time:     Wed, 20 May 2020 19:35:47 -0600
Labels:         app=hazelcast
                env=prod
Annotations:    <none>
Status:         Running
IP:            10.1.0.41
Containers:
...
Events:
...
```

The terminal output contains the metadata information of a Pod, the containers it runs, and the event log, such as failures when the Pod was scheduled. The example output has been condensed to show just the metadata section. You can expect the output to be very lengthy.

There's a way to be more specific about the information you want to render. You can combine the `describe` command with a Unix `grep` command if you want to identify the image for running in the container:

```
$ kubectl describe pods hazelcast | grep Image:  
Image:          hazelcast/hazelcast:5.1.7
```

Accessing Logs of a Pod

As application developers, we know very well what to expect in the log files produced by the application we implemented. Runtime failures may occur when operating an application in a container. The `logs` command downloads the log output of a container. The following output indicates that the Hazelcast server started up successfully:

```
$ kubectl logs hazelcast  
...  
May 25, 2020 3:36:26 PM com.hazelcast.core.LifecycleService  
INFO: [10.1.0.46]:5701 [dev] [4.0.1] [10.1.0.46]:5701 is STARTED
```

It's very likely that more log entries will be produced as soon as the container receives traffic from end users. You can stream the logs with the command line option `-f`. This option is helpful if you want to see logs in real time.

Kubernetes tries to restart a container under certain conditions, such as if the image cannot be resolved on the first try. Upon a container restart, you won't have access to the logs of the previous container; the `logs` command renders the logs only for the current container. However, you can still get back to the logs of the previous container by adding the `-p` command line option. You may want to use the option to identify the root cause that triggered a container restart.

Executing a Command in Container

Some situations require you to get the shell to a running container and explore the filesystem. Maybe you want to inspect the configuration of your application or debug its current state. You can use the `exec` command to open a shell in the container to explore it interactively, as follows:

```
$ kubectl exec -it hazelcast -- /bin/sh  
# ...
```

Notice that you do not have to provide the resource type. This command only works for a Pod. The two dashes (--) separate the `exec` command and its options from the command you want to run inside of the container.

It's also possible to execute a single command inside of a container. Say you wanted to render the environment variables available to containers without having to be logged

in. Just remove the interactive flag `-it` and provide the relevant command after the two dashes:

```
$ kubectl exec hazelcast -- env  
...  
DNS_DOMAIN=cluster
```

Creating a Temporary Pod

The command executed inside of a Pod—usually an application implementing business logic—is meant to run infinitely. Once the Pod has been created, it will stick around. Under certain conditions, you want to execute a command in a Pod just for troubleshooting. This use case doesn’t require a Pod object to run beyond the execution of the command. That’s where temporary Pods come into play.

The `run` command provides the flag `--rm`, which will automatically delete the Pod after the command running inside of it finishes. Say you want to render all environment variables using `env` to see what’s available inside of the container. The following command achieves exactly that:

```
$ kubectl run busybox --image=busybox:1.36.1 --rm -it --restart=Never -- env  
...  
HOSTNAME=busybox  
pod "busybox" deleted
```

The last message rendered in the output clearly states that the Pod was deleted after command execution.

Using a Pod’s IP Address for Network Communication

Every Pod is assigned an IP address upon creation. You can inspect a Pod’s IP address by using the `-o wide` command-line option for the `get pod` command or by describing the Pod. The IP address of the Pod in the following console output is `10.244.0.5`:

```
$ kubectl run nginx --image=nginx:1.25.1 --port=80  
pod/nginx created  
$ kubectl get pod nginx -o wide  
NAME      READY   STATUS    RESTARTS   AGE     IP           NODE    \  
NOMINATED NODE   READINESS GATES  
nginx     1/1     Running   0          37s    10.244.0.5   minikube  \  
<none>        <none>  
$ kubectl get pod nginx -o yaml  
...  
status:  
  podIP: 10.244.0.5  
...
```

The IP address assigned to a Pod is unique across all nodes and namespaces. This is achieved by assigning a dedicated subnet to each node when registering it. When creating a new Pod on a node, the IP address is leased from the assigned subnet. This is

handled by the networking life cycle manager kube-proxy along with the Domain Name Service (DNS) and the Container Network Interface (CNI).

You can easily verify the behavior by creating a temporary Pod that calls the IP address of another Pod using the command-line tool `curl` or `wget`:

```
$ kubectl run busybox --image=busybox:1.36.1 --rm -it --restart=Never \
-- wget 172.17.0.4:80
Connecting to 172.17.0.4:80 (172.17.0.4:80)
saving to 'index.html'
index.html          100% |*****| 615  0:00:00 ETA
'index.html' saved
pod "busybox" deleted
```

It's important to understand that the IP address is not considered stable over time. A Pod restart leases a new IP address. Therefore, this IP address is often referred to as *virtual* IP address. Building a microservices architecture—where each of the applications runs in its own Pod with the need to communicate between each other with a stable network interface—requires a different concept: the Service. Refer to [Chapter 21](#) for more information.

Configuring Pods

The curriculum expects you to feel comfortable with editing YAML manifests either as files or as live object representations. This section shows you some typical configuration scenarios you may face during the exam. Later chapters will deepen your knowledge by touching on other configuration aspects.

Declaring environment variables

Applications need to expose a way to make their runtime behavior configurable. For example, you may want to inject the URL to an external web service or declare the username for a database connection. Environment variables are a common option to provide this runtime configuration.



Avoid creating container images per environment

It might be tempting to say, “Hey, let’s create a container image for any target deployment environment we need, including its configuration.” That’s a bad idea. One of the practices of [continuous delivery](#) and the [Twelve-Factor App principles](#) is to build a deployable artifact for a commit just once. In this case, the artifact is the container image. Deviating configuration runtime behavior should be controllable by injecting runtime information when instantiating the container. You can use environment variables to control the behavior as needed.

Defining environment variables in a Pod YAML manifest is relatively easy. Add or enhance the section `env` of a container. Every environment variable consists of a key-value pair, represented by the attributes `name` and `value`. Kubernetes does not enforce or sanitize typical naming conventions for environment variable keys, though it is recommended to follow the standard of using upper-case letters and the underscore character (`_`) to separate words.

To illustrate a set of environment variables, look at [Example 5-2](#). The code snippet describes a Pod that runs a Java-based application using the Spring Boot framework.

Example 5-2. YAML manifest for a Pod defining environment variables

```
apiVersion: v1
kind: Pod
metadata:
  name: spring-boot-app
spec:
  containers:
    - name: spring-boot-app
      image: bmuschko/spring-boot-app:1.5.3
      env:
        - name: SPRING_PROFILES_ACTIVE
          value: prod
        - name: VERSION
          value: '1.5.3'
```

The first environment variable named `SPRING_PROFILES_ACTIVE` defines a pointer to a so-called profile. A profile contains environment-specific properties. Here, we are pointing to the profile that configures the production environment. The environment variable `VERSION` specifies the application version. Its value corresponds to the tag of the image and can be exposed by the running application to display the value in the user interface.

Defining a command with arguments

Many container images already define an `ENTRYPOINT` or `CMD` instruction. The command assigned to the instruction is automatically executed as part of the container startup. For example, the Hazelcast image we used earlier defines the instruction `CMD ["/opt/hazelcast/start-hazelcast.sh"]`.

In a Pod definition, you can either redefine the image `ENTRYPOINT` and `CMD` instructions or assign a command to execute for the container if it hasn't been specified by the image. You can provide this information with the help of the `command` and `args` attributes for a container. The `command` attribute overrides the image's `ENTRYPOINT` instruction. The `args` attribute replaces the `CMD` instruction of an image.

Imagine you wanted to provide a command to an image that doesn't provide one yet. As usual, there are two different approaches: imperative and declarative. We'll generate the YAML manifest with the help of the `run` command. The Pod should use the `busybox:1.36.1` image and execute a shell command that renders the current date every 10 seconds in an infinite loop:

```
$ kubectl run mypod --image=busybox:1.36.1 -o yaml --dry-run=client \
> pod.yaml -- /bin/sh -c "while true; do date; sleep 10; done"
```

You can see in the generated but condensed `pod.yaml` file shown in [Example 5-3](#) that the command has been turned into an `args` attribute. Kubernetes specifies each argument on a single line.

Example 5-3. A YAML manifest containing an args attribute

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mypod
      image: busybox:1.36.1
      args:
        - /bin/sh
        - -c
        - while true; do date; sleep 10; done
```

You could have achieved the same by a combination of the `command` and `args` attributes if you were to handcraft the YAML manifest. [Example 5-4](#) shows a different approach.

Example 5-4. A YAML manifest containing command and args attributes

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mypod
      image: busybox:1.36.1
      command: ["/bin/sh"]
      args: ["-c", "while true; do date; sleep 10; done"]
```

You can quickly verify if the declared command actually does its job. First, create the Pod instance, then tail the logs:

```
$ kubectl apply -f pod.yaml
pod/mypod created
$ kubectl logs mypod -f
Fri May 29 00:49:06 UTC 2020
Fri May 29 00:49:16 UTC 2020
Fri May 29 00:49:26 UTC 2020
...
...
```

Deleting a Pod

Sooner or later you'll want to delete a Pod. During the exam, you may be asked to remove a Pod. Or possibly, you made a configuration mistake and want to start the question from scratch:

```
$ kubectl delete pod hazelcast
pod "hazelcast" deleted
```

Keep in mind that Kubernetes tries to delete a Pod *gracefully*. This means that the Pod will try to finish active requests to the Pod to avoid unnecessary disruption to the end user. A graceful deletion operation can take anywhere from 5 to 30 seconds, time you don't want to waste during the exam. See [Chapter 1](#) for more information on how to speed up the process.

An alternative way to delete a Pod is to point the `delete` command to the YAML manifest you used to create it. The behavior is the same:

```
$ kubectl delete -f pod.yaml
pod "hazelcast" deleted
```

To save time during the exam, you can circumvent the grace period by adding the `--now` option to the `delete` command. Avoid using the `--now` flag in production Kubernetes environments.

Working with Namespaces

Namespaces are an API construct to avoid naming collisions, and they represent a scope for object names. A good use case for namespaces is to isolate the objects by team or responsibility.



Namespaces for objects

The content in this chapter demonstrates the use of namespaces for Pod objects. Namespaces are not a concept applicable only to Pods though. Most object types can be grouped by a namespace.

Most questions in the exam will ask you to execute the command in a specific namespace that has been set up for you. The following sections briefly touch on the basic operations needed to deal with a namespace.

Listing Namespaces

A Kubernetes cluster starts out with a couple of initial namespaces. You can list them with the following command:

```
$ kubectl get namespaces
NAME      STATUS   AGE
default   Active   157d
kube-node-lease   Active   157d
kube-public    Active   157d
kube-system    Active   157d
```

The `default` namespace hosts objects that haven't been assigned to an explicit namespace. Namespaces starting with the prefix `kube-` are not considered end user-namespaces. They have been created by the Kubernetes system. You will not have to interact with them as an application developer.

Creating and Using a Namespace

To create a new namespace, use the `create namespace` command. The following command uses the name `code-red`:

```
$ kubectl create namespace code-red
namespace/code-red created
$ kubectl get namespace code-red
NAME      STATUS   AGE
code-red  Active   16s
```

Example 5-5 shows the corresponding representation as a YAML manifest.

Example 5-5. Namespace YAML manifest

```
apiVersion: v1
kind: Namespace
metadata:
  name: code-red
```

Once the namespace is in place, you can create objects within it. You can do so with the command line option `--namespace` or its short-form `-n`. The following commands create a new Pod in the namespace `code-red` and then list the available Pods in the namespace:

```
$ kubectl run pod --image=nginx:1.25.1 -n code-red
pod/pod created
$ kubectl get pods -n code-red
NAME     READY   STATUS    RESTARTS   AGE
pod      1/1     Running   0          13s
```

Setting a Namespace Preference

Providing the `--namespace` or `-n` command line option for every command is tedious and error-prone. You can set a permanent namespace preference if you know that you need to interact with a specific namespace you are responsible for. The first command shown sets the permanent namespace `code-red`. The second command renders the currently set permanent namespace:

```
$ kubectl config set-context --current --namespace=code-red
Context "minikube" modified.
$ kubectl config view --minify | grep namespace:
    namespace: hello
```

Subsequent `kubectl` executions do not have to spell out the namespace `code-red`:

```
$ kubectl get pods
NAME    READY    STATUS    RESTARTS   AGE
pod     1/1      Running   0          13s
```

You can always switch back to the default namespace or another custom namespace using the `config set-context` command:

```
$ kubectl config set-context --current --namespace=default
Context "minikube" modified.
```

Deleting a Namespace

Deleting a namespace has a cascading effect on the object existing in it. Deleting a namespace will automatically delete its objects:

```
$ kubectl delete namespace code-red
namespace "code-red" deleted
$ kubectl get pods -n code-red
No resources found in code-red namespace.
```

Summary

The exam puts a strong emphasis on the concept of a Pod, a Kubernetes primitive responsible for running an application in a container. A Pod can define one or many containers that use a container image. Upon its creation, the container image is resolved and used to bootstrap the application. Every Pod can be further customized with the relevant YAML configuration.

Exam Essentials

Know how to interact with Pods

A Pod runs an application inside of a container. You can check on the status and the configuration of the Pod by inspecting the object with the `kubectl get` or `kubectl describe` commands. Get familiar with the life cycle phases of a Pod to be able to quickly diagnose errors. The command `kubectl logs` can be used to download the container log information without having to shell into the container. Use the command `kubectl exec` to further explore the container environment, e.g., to check on processes or to examine files.

Understand advanced Pod configuration options

Sometimes you have to start with the YAML manifest of a Pod and then create the Pod declaratively. This could be the case if you wanted to provide environment variables to the container or declare a custom command. Practice different configuration options by copy-pasting relevant code snippets from the Kubernetes documentation.

Practice using a custom namespace

Most questions in the exam will ask you to work within a given namespace. You need to understand how to interact with that namespace from `kubectl` using the options `--namespace` and `-n`. To avoid accidentally working on the wrong namespace, know how to permanently set a namespace.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. Create a new Pod named `nginx` running the image `nginx:1.17.10`. Expose the container port 80. The Pod should live in the namespace named `ckad`.

Get the details of the Pod including its IP address.

Create a temporary Pod that uses the `busybox:1.36.1` image to execute a `wget` command inside of the container. The `wget` command should access the endpoint exposed by the `nginx` container. You should see the HTML response body rendered in the terminal.

Get the logs of the `nginx` container.

Add the environment variables `DB_URL=postgresql://mydb:5432` and `DB_USER_NAME=admin` to the container of the `nginx` Pod.

Open a shell for the `nginx` container and inspect the contents of the current directory `ls -l`. Exit out of the container.

2. Create a YAML manifest for a Pod named `loop` that runs the `busybox:1.36.1` image in a container. The container should run the following command: `for i in {1..10}; do echo "Welcome $i times"; done`. Create the Pod from the YAML manifest. What's the status of the Pod?

Edit the Pod named `loop`. Change the command to run in an endless loop. Each iteration should echo the current date.

Inspect the events and the status of the Pod `loop`.

Jobs and CronJobs

A Job models a one-time process—for example, a batch operation. The Pod and its encompassed containers stop running after the work has been completed. CronJobs run periodically according to their defined schedules. A good application for a CronJob is a task that needs to occur periodically (for example, a process that exports data). In this chapter you will learn how to configure, run, and inspect a Job and a CronJob.

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objective:

- Understand Jobs and CronJobs

Working with Jobs

A Job is a Kubernetes primitive that runs functionality until a specified number of completions has been reached, making it a good fit for one-time operations like import/export data processes or I/O-intensive processes with a finite end. The actual work managed by a Job is still running inside of a Pod. Therefore, you can think of a Job as a higher-level coordination instance for Pods executing the workload. [Figure 6-1](#) shows the parent-child relationship between a Job and the Pod(s) it manages.

Upon completion of a Job and its Pods, Kubernetes does not automatically delete the objects—they will stay until they’re explicitly deleted. Keeping those objects helps with debugging the command run inside of the Pod and gives you a chance to inspect the logs.

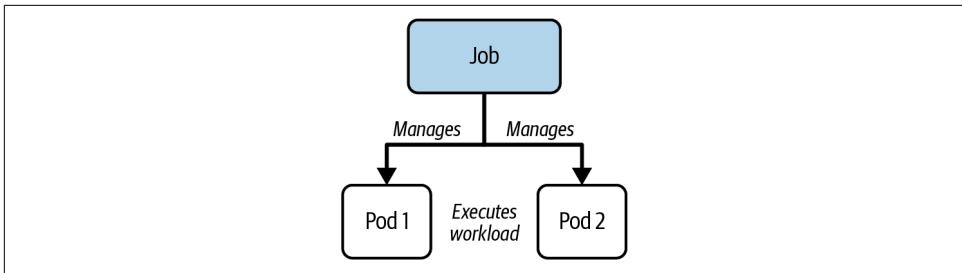


Figure 6-1. Relationship between a Job and its Pods



Auto-cleanup of Jobs and Pods

Kubernetes supports an [auto-cleanup mechanism](#) for Jobs and their controlled Pods by specifying the YAML attribute `spec.ttlSecondsAfterFinished`.

Creating and Inspecting Jobs

Let's first create a Job and observe its behavior in practice before delving into details. To create a Job imperatively, simply use the `create job` command. If the provided image doesn't run any commands, you may want to append a command to be executed in the corresponding Pod.

The following command creates a Job that runs an iteration process. For each iteration of the loop, a variable named `counter` is incremented. The command execution finishes after reaching the counter value 3:

```
$ kubectl create job counter --image=nginx:1.25.1 \
-- /bin/sh -c 'counter=0; while [ $counter -lt 3 ]; do \
counter=$((counter+1)); echo "$counter"; sleep 3; done;'
job.batch/counter created
```

[Example 6-1](#) shows the YAML manifest equivalent for the Job if you prefer the declarative approach.

Example 6-1. A Job executing a loop command

```
apiVersion: batch/v1
kind: Job
metadata:
  name: counter
spec:
  template: ①
  spec:
    containers:
      - name: counter
        image: nginx:1.25.1
```

```
command:
- /bin/sh
- -c
- counter=0; while [ $counter -lt 3 ]; do counter=$((counter+1)); \
  echo "$counter"; sleep 3; done;
restartPolicy: Never
```

- ➊ The Pod template uses the same attributes available in a Pod definition.

The output of listing the Job shows the current number of completions and the expected number of completions. The default number of completions is 1. This means if the Pod executing the command was successful, a Job is considered completed. As you can see in the following terminal output, a Job uses a single Pod by default to perform the work. The corresponding Pod can be identified by name—it uses the Job name as a prefix in its own name:

```
$ kubectl get jobs
NAME      COMPLETIONS  DURATION   AGE
counter   0/1          13s        13s
$ kubectl get jobs
NAME      COMPLETIONS  DURATION   AGE
counter   1/1          15s        19s
$ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
counter-z6kdj  0/1    Completed  0          51s
```

To verify the correct behavior of the Job, you can download its logs. As expected, the output renders the counter for each iteration:

```
$ kubectl logs counter-z6kdj
1
2
3
```

You can further tweak the runtime behavior of a Job. The next two sections discuss configuring the Job operation types and restart behavior.

Job Operation Types

The default behavior of a Job is to run the workload in a single Pod and expect one successful completion. That's what Kubernetes calls a *non-parallel* Job. Internally, those parameters are tracked by the attributes `spec.completions` and `spec.parallelism`, each with the assigned value 1. The following command renders the parameters of the Job we created earlier:

```
$ kubectl get jobs counter -o yaml | grep -C 1 "completions"
...
completions: 1
parallelism: 1
...
```

You can tweak any of those parameters to fit the needs of your use case. For example, if you expected the workload to complete successfully multiple times, then you'd increase the value of `spec.completions` to at least 2. Sometimes, you'll want to execute the workload by multiple pods in parallel. In those cases, you'd bump up the value assigned to `spec.parallelism`. This is referred to as a *parallel job*. Remember that you can use any combination of assigned values for both attributes. [Table 6-1](#) summarizes the different use cases.

Table 6-1. Configuration for different Job operation types

Type	spec.completions	spec.parallelism	Description
Non-parallel with one completion count	1	1	Completes as soon as its Pod terminates successfully.
Parallel with a fixed completion count	≥ 1	≥ 1	Completes when specified number of tasks finish successfully.
Parallel with worker queue	unset	≥ 1	Completes when at least one Pod has terminated successfully and all Pods are terminated.

Restart Behavior

The `spec.backoffLimit` attribute determines the number of retries a Job attempts to successfully complete the workload until the executed command finishes with an exit code 0. The default is 6, which means it will execute the workload 6 times before the Job is considered unsuccessful.

The Job manifest needs to explicitly declare the restart policy by using `spec.template.spec.restartPolicy`. The default restart policy of a Pod is `Always`, which tells the Kubernetes scheduler to *always* restart the Pod even if the container exits with a 0 exit code. The restart policy of a Job can be only `OnFailure` or `Never`.

Restarting the container on failure

[Figure 6-2](#) shows the behavior of a Job that uses the restart policy `OnFailure`. Upon a container failure, this policy will simply rerun the container.

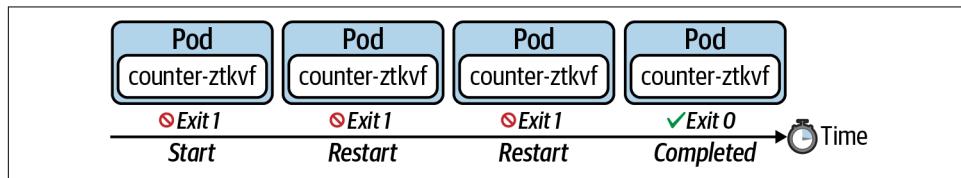


Figure 6-2. Restart policy onFailure

Starting a new Pod on failure

Figure 6-3 shows the behavior of a Job that uses the restart policy Never. This policy does not restart the container upon a failure. It starts a new Pod instead.

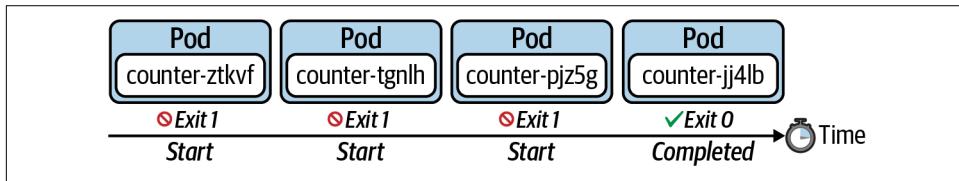


Figure 6-3. Restart policy Never

Working with CronJobs

A Job represents a finite operation. Once the operation can be executed successfully, the work is done and the Job will create no more Pods. A CronJob creates a new Job object periodically based a schedule. The Pods controlled by the Job handle the actual workload. Figure 6-4 illustrates the relationship between a CronJob, the Job it manages, and the Pods that execute the workload.

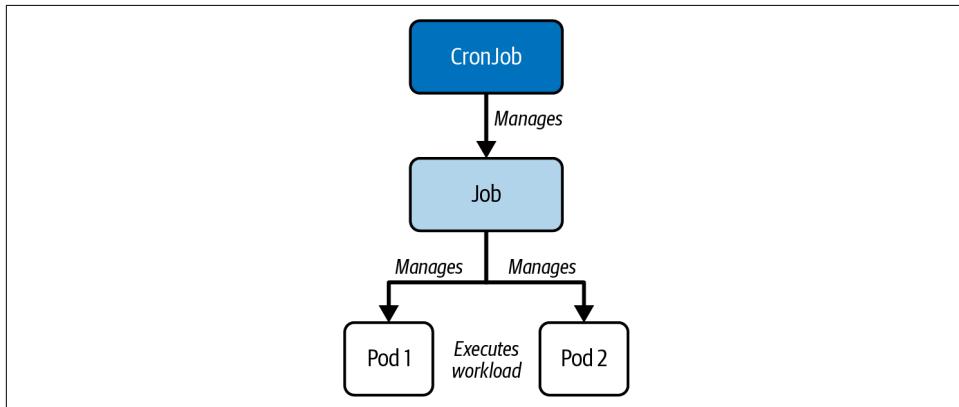


Figure 6-4. Relationship between a CronJob, Job, and its Pods

The schedule can be defined with a cron-expression you may already know from Unix cron jobs. Figure 6-5 shows a CronJob that executes every hour. For every execution, the CronJob creates a new Pod that runs the task and finishes with a 0 or non-zero exit code.

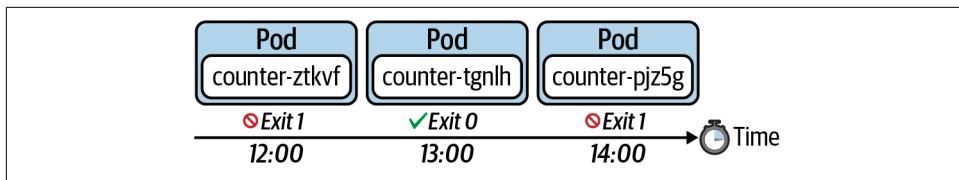


Figure 6-5. Executing a Job based on a schedule

Creating and Inspecting CronJobs

You can use the imperative `create cronjob` command to create a new CronJob. The following command schedules the CronJob to run every minute. The Pod created for every execution renders the current date to standard output using the Unix `echo` command:

```
$ kubectl create cronjob current-date --schedule="* * * * *" \
  --image=nginx:1.25.1 -- /bin/sh -c 'echo "Current date: $(date)"'
cronjob.batch/current-date created
```

To create a CronJob from the YAML manifest, use the definition shown in [Example 6-2](#).

Example 6-2. A CronJob printing the current date

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: current-date
spec:
  schedule: "* * * * *"      ①
  jobTemplate:                 ②
    spec:
      template:
        spec:
          containers:
            - name: current-date
              image: nginx:1.25.1
              args:
                - /bin/sh
                - -c
                - 'echo "Current date: $(date)"'
  restartPolicy: OnFailure
```

- ① Defines the cron expression that determines when a new Job object needs to be created.
- ② The section that describes the Job template.

If you list the existing CronJob with the `get cronjobs` command, you will see the schedule, the last scheduled execution, and whether the CronJob is currently active:

```
$ kubectl get cronjobs
NAME      SCHEDULE    SUSPEND   ACTIVE   LAST SCHEDULE   AGE
current-date * * * * *  False     0        <none>   28s
$ kubectl get cronjobs
NAME      SCHEDULE    SUSPEND   ACTIVE   LAST SCHEDULE   AGE
current-date * * * * *  False     1        14s     53s
```

It's easy to match Jobs and Pods created by a CronJob. You can simply identify them by the name prefix. In this case, the prefix is `current-date-`:

```
$ kubectl get jobs,pods
NAME                           COMPLETIONS   DURATION   AGE
job.batch/current-date-28473049 1/1          3s         2m23s
job.batch/current-date-28473050 1/1          3s         83s
job.batch/current-date-28473051 1/1          3s         23s

NAME                           READY   STATUS    RESTARTS   AGE
pod/current-date-28473049-l6hc7 0/1    Completed  0          2m23s
pod/current-date-28473050-csq7n  0/1    Completed  0          83s
pod/current-date-28473051-jg8st  0/1    Completed  0          23s
```

Configuring Retained Job History

Even after a task in a Pod controlled by a CronJob completes, it will not be deleted automatically. Keeping a historical record of Pods can be tremendously helpful for troubleshooting failed workloads or inspecting the logs. By default, a CronJob retains the last three successful Pods and the last failed Pod:

```
$ kubectl get cronjobs current-date -o yaml | grep successfulJobsHistoryLimit:
successfulJobsHistoryLimit: 3
$ kubectl get cronjobs current-date -o yaml | grep failedJobsHistoryLimit:
failedJobsHistoryLimit: 1
```

To reconfigure the job retention history limits, set new values for the `spec.successfulJobsHistoryLimit` and `spec.failedJobsHistoryLimit` attributes. [Example 6-3](#) keeps the last five successful executions and the last three failed executions.

Example 6-3. A CronJob configuring retention history limits

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: current-date
spec:
  successfulJobsHistoryLimit: 5      ①
  failedJobsHistoryLimit: 3          ②
  schedule: "* * * * *"
  jobTemplate:
```

```
spec:  
  template:  
    spec:  
      containers:  
        - name: current-date  
          image: nginx:1.25.1  
          args:  
            - /bin/sh  
            - -c  
            - 'echo "Current date: $(date)"'  
      restartPolicy: OnFailure
```

- ① Defines the number of successful Jobs kept in the history.
- ② Defines the number of failed Jobs kept in the history.

Summary

Jobs are well suited for implementing batch processes run in one or many Pods as a finite operation. Both objects, the Job and the Pod, will not be deleted after the work is completed to support inspection and troubleshooting activities. A CronJob is very similar to a Job but executes on a schedule, defined as a Unix cron expression.

Exam Essentials

Understand practical use cases of Jobs and CronJobs

Jobs and CronJobs manage Pods that should finish the work at least once or periodically. You will need to understand the creation of those objects and how to inspect them at runtime. Make sure to play around with the different configuration options and how they affect the runtime behavior.

Practice different Job operational modes

Jobs can operate in three modes: non-parallel with one completion count, in parallel with a fixed completion count, and in parallel with worker queue. The default behavior of a Job is to run the workload in a single Pod and expect one successful completion (non-parallel Job). The attribute `spec.completions` controls the number of required successful completions. The attribute `spec.parallelism` allows for executing the workload by multiple Pods in parallel.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. Create a Job named `random-hash` using the container image `alpine:3.17.3` that executes the shell command `echo $RANDOM | base64 | head -c 20`. Configure the Job to execute with two Pods in parallel. The number of completions should be set to 5.

Identify the Pods that executed the shell command. How many Pods do you expect to exist?

Retrieve the generated hash from one of the Pods.

Delete the Job. Will the corresponding Pods continue to exist?

2. Create a new CronJob named `google-ping`. When executed, the Job should run a `curl` command for `google.com`. Pick an appropriate image. The execution should occur every two minutes.

Watch the creation of the underlying Jobs managed by the CronJob. Check the command-line options of the relevant command or consult the [Kubernetes documentation](#).

Reconfigure the CronJob to retain a history of seven executions.

Reconfigure the CronJob to disallow a new execution if the current execution is still running. Consult the [Kubernetes documentation](#) for more information.

Volumes

A container's temporary filesystem is isolated from any other container or Pod and is not persisted beyond a Pod restart. A Pod can define a Volume and mount it to a container.

Ephemeral Volumes exist for the lifespan of a Pod. They are useful if you want to share data between multiple containers running in the Pod. Persistent Volumes preserve data beyond the lifespan of a Pod. They are a good option for applications that require data to exist longer, e.g., in the form of storage for a database-driven application. In this chapter, we'll exercise the use of different Volumes types in a Pod.

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objective:

- Utilize persistent and ephemeral volumes

Working with Storage

Applications running in a container can use the temporary filesystem to read and write files. In case of a container crash or a cluster/node restart, the kubelet will restart the container. Any data that had been written to the temporary filesystem is lost and cannot be retrieved. The container effectively starts with a clean slate.

There are many uses cases for wanting to mount a Volume in a container. We'll see one of the most prominent use cases in [Chapter 8](#): using an ephemeral Volume to exchange data between a main application container and a sidecar. [Figure 7-1](#) illustrates the differences between the temporary filesystem of a container and the use of a Volume.

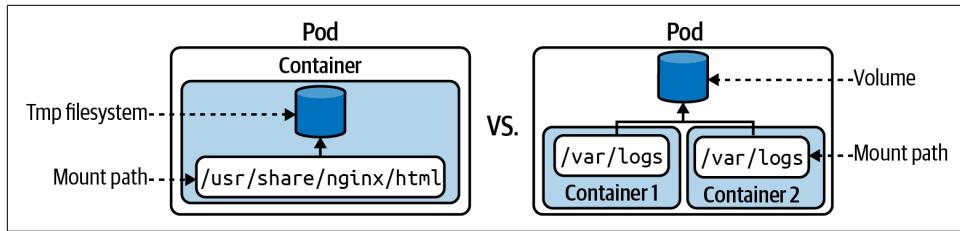


Figure 7-1. A container using the temporary filesystem versus a Volume

Volume Types

Every Volume needs to define a type. The type determines the medium that backs the Volume and its runtime behavior.

Ephemeral Volumes

These exist for the lifespan of a Pod. Ephemeral Volumes are useful if you want to share data between multiple containers running in the Pod or if you can easily reconstruct the data stored on the Volume upon a Pod restart.

Persistent Volumes

These preserve data beyond the lifespan of a Pod. Persistent Volumes are a good option for applications that require data to exist longer, for example, in the form of storage for a database-driven application.

The Kubernetes documentation offers a long list of Volume types. [Table 7-1](#) provides a select list of Volume types that I have found to be most relevant to the exam.

Table 7-1. Volume types most relevant to exam

Type	Description
emptyDir	Empty directory in Pod with read/write access. Only persisted for the lifespan of a Pod. A good choice for cache implementations or data exchange between containers of a Pod.
hostPath	File or directory from the host node's filesystem. Supported only on single-node clusters and not meant for production.
configMap, secret	Provides a way to inject configuration data. For practical examples, see Chapter 19 .
nfs	An existing NFS (Network File System) share. Preserves data after Pod restart.
persistentVolumeClaim	Claims a Persistent Volume. For more information, see " Creating PersistentVolumeClaims " on page 75.

Ephemeral Volumes

Defining an ephemeral Volume for a Pod requires two steps. First, you need to declare the Volume itself using the attribute `spec.volumes[]`. As part of the definition, you provide the name and the type. Just declaring the Volume won't be

sufficient. Second, the Volume needs to be mounted to a path of the consuming container via `spec.containers[].volumeMounts[]`. The mapping between the Volume and the Volume mount occurs through the matching name.

Creating and mounting an ephemeral Volume

In [Example 7-1](#), stored in the file `pod-with-volume.yaml` here, you can see the definition of a Volume with type `emptyDir`. The Volume has been mounted to the path `/var/log/nginx` inside the container named `nginx`.

Example 7-1. A Pod defining and mounting a ephemeral Volume

```
apiVersion: v1
kind: Pod
metadata:
  name: business-app
spec:
  volumes:
    - name: logs-volume
      emptyDir: {}
  containers:
    - image: nginx:1.25.1
      name: nginx
      volumeMounts:
        - mountPath: /var/log/nginx
          name: logs-volume
```

Interacting with the Volume

Let's create the Pod and see if we can interact with the mounted Volume. The following commands open an interactive shell after the Pod's creation, then navigate to the mount path. You can see that the Volume type `emptyDir` initializes the mount path as an empty directory. New files and directories can be created as needed without limitations:

```
$ kubectl create -f pod-with-volume.yaml
pod/business-app created
$ kubectl get pod business-app
NAME           READY   STATUS    RESTARTS   AGE
business-app   1/1     Running   0          43s
$ kubectl exec business-app -it -- /bin/sh
# cd /var/log/nginx
# pwd
/var/log/nginx
# ls
# touch app-logs.txt
# ls
app-logs.txt
```

For an illustrative use case of the `emptyDir` Volume type mounted by more than one container, see [Chapter 8](#).

Persistent Volumes

Data stored on Volumes outlives a container restart. In many applications, the data lives far beyond the life cycles of the applications, container, Pod, nodes, and even the clusters themselves. Data persistence ensures the life cycles of the data are decoupled from the life cycles of the cluster resources. A typical example would be data persisted by a database. That's the responsibility of a persistent Volume. Kubernetes models persist data with the help of two primitives: the PersistentVolume and the PersistentVolumeClaim.

The PersistentVolume is the primitive representing a piece of storage in a Kubernetes cluster. It is completely decoupled from the Pod and therefore has its own life cycle. The object captures the source of the storage (e.g., storage made available by a cloud provider). A PersistentVolume is either provided by a Kubernetes administrator or assigned dynamically by mapping to a storage class.

The PersistentVolumeClaim requests the resources of a PersistentVolume—for example, the size of the storage and the access type. In the Pod, you will use the type `persistentVolumeClaim` to mount the abstracted PersistentVolume by using the PersistentVolumeClaim.

[Figure 7-2](#) shows the relationship between the Pod, the PersistentVolumeClaim, and the PersistentVolume.

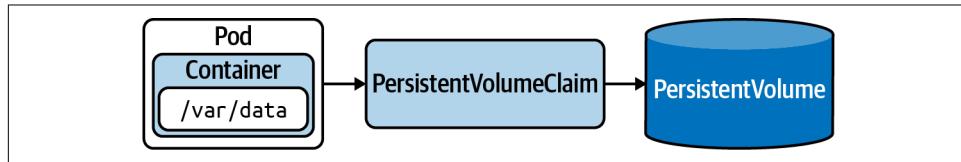


Figure 7-2. Claiming a PersistentVolume from a Pod

Static versus dynamic provisioning

A PersistentVolume can be created statically or dynamically. If you go with the static approach, then you first need to create a storage device and then reference it by explicitly creating an object of kind PersistentVolume. The dynamic approach doesn't require you to create a PersistentVolume object. It will be automatically created from the PersistentVolumeClaim by setting a storage class name using the attribute `spec.storageClassName`.

A storage class is an abstraction concept that defines a class of storage device (e.g., storage with slow or fast performance) used for different application types. It's the job of a Kubernetes administrator to set up storage classes. For a deeper discussion on

storage classes, see “[Storage Classes](#)” on page 77. For now, we’ll focus on the static provisioning of PersistentVolumes.

Creating PersistentVolumes

When you create a PersistentVolume object yourself, we refer to the approach as static provisioning. A PersistentVolume can be created only by using the manifest-first approach. At this time, `kubectl` doesn’t allow the creation of a PersistentVolume using the `create` command. Every PersistentVolume needs to define the storage capacity using `spec.capacity` and an access mode set via `spec.accessModes`. See “[Configuration options for a PersistentVolume](#)” on page 74 for more information on the configuration options available to a PersistentVolume.

Example 7-2 creates a PersistentVolume named `db-pv` with a storage capacity of `1Gi` and read/write access by a single node. The attribute `hostPath` mounts the directory `/data/db` from the host node’s filesystem. We’ll store the YAML manifest in the file `db-pv.yaml`.

Example 7-2. YAML manifest defining a PersistentVolume

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: db-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /data/db
```

When you inspect the created PersistentVolume, you’ll find most of the information you provided in the manifest. The status `Available` indicates that the object is ready to be claimed. The reclaim policy determines what should happen with the PersistentVolume after it has been released from its claim. By default, the object will be retained. The following example uses the short-form command `pv` to avoid having to type `persistentvolume`:

```
$ kubectl create -f db-pv.yaml
persistentvolume/db-pv created
$ kubectl get pv db-pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      \
        CLAIM      STORAGECLASS   REASON           AGE
db-pv     1Gi        RWO            Retain          Available   \
                    10s
```

Configuration options for a PersistentVolume

A PersistentVolume offers a variety of configuration options that determine their innate runtime behavior. For the exam, it's important to understand the volume mode, access mode, and reclaim policy configuration options.

Volume mode. The volume mode handles the type of device. That's a device either meant to be consumed from the filesystem or backed by a block device. The most common case is a filesystem device. You can set the volume mode using the attribute `spec.volumeMode`. [Table 7-2](#) shows all available volume modes.

Table 7-2. PersistentVolume volume modes

Type	Description
Filesystem	Default. Mounts the volume into a directory of the consuming Pod. Creates a filesystem first if the volume is backed by a block device and the device is empty.
Block	Used for a volume as a raw block device without a filesystem on it.

The volume mode is not rendered by default in the console output of the `get pv` command. You will need to provide the `-o wide` command-line option to see the `VOLUMEMODE` column, as shown here:

```
$ kubectl get pv -o wide
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      \
CLAIM    STORAGECLASS   REASON     AGE      VOLUMEMODE
db-pv    1Gi          RWO        19m     Filesystem  Available \
```

Access mode. Each PersistentVolume can express how it can be accessed using the attribute `spec.accessModes`. For example, you can define that the volume can be mounted only by a single Pod in a read or write mode or that a volume is read-only but accessible from different nodes simultaneously. [Table 7-3](#) provides an overview of the available access modes. The short-form notation of the access mode is usually rendered in outputs of specific commands, e.g., `get pv` or `describe pv`.

Table 7-3. PersistentVolume access modes

Type	Short Form	Description
ReadWriteOnce	RWO	Read/write access by a single node
ReadOnlyMany	ROX	Read-only access by many nodes
ReadWriteMany	RWX	Read/write access by many nodes
ReadWriteOncePod	RWOP	Read/write access mounted by a single Pod

The following command parses the access modes from the PersistentVolume named db-pv. As you can see, the returned value is an array underlining the fact that you can assign multiple access modes at once:

```
$ kubectl get pv db-pv -o jsonpath='{.spec.accessModes}'  
["ReadWriteOnce"]
```

Reclaim policy. Optionally, you can also define a reclaim policy for a PersistentVolume. The reclaim policy specifies what should happen to a PersistentVolume object when the bound PersistentVolumeClaim is deleted (see [Table 7-4](#)). For dynamically created PersistentVolumes, the reclaim policy can be set via the attribute `.reclaimPolicy` in the storage class. For statically created PersistentVolumes, use the attribute `spec.persistentVolumeReclaimPolicy` in the PersistentVolume definition.

Table 7-4. PersistentVolume reclaim policies

Type	Description
Retain	Default. When PersistentVolumeClaim is deleted, the PersistentVolume is “released” and can be reclaimed.
Delete	Deletion removes PersistentVolume and its associated storage.
Recycle	This value is deprecated. You should use one of the other values.

This command retrieves the assigned reclaim policy of the PersistentVolume named db-pv:

```
$ kubectl get pv db-pv -o jsonpath='{.spec.persistentVolumeReclaimPolicy}'  
Retain
```

Creating PersistentVolumeClaims

The next object we’ll need to create is the PersistentVolumeClaim. Its purpose is to bind the PersistentVolume to the Pod. Let’s look at the YAML manifest stored in the file `db-pvc.yaml`, shown in [Example 7-3](#).

Example 7-3. Definition of a PersistentVolumeClaim

```
kind: PersistentVolumeClaim  
apiVersion: v1  
metadata:  
  name: db-pvc  
spec:  
  accessModes:  
    - ReadWriteOnce  
  storageClassName: ""  
  resources:  
    requests:  
      storage: 256Mi
```

What we're saying is: "Give me a PersistentVolume that can fulfill the resource request of 256Mi and provides the access mode ReadWriteOnce." Static provisioning should use an empty string for the attribute `spec.storageClassName` if you do not want it to automatically assign the default storage class. The binding to an appropriate PersistentVolume happens automatically based on those criteria.

After creating the PersistentVolumeClaim, the status is set as Bound, which means that the binding to the PersistentVolume was successful. Once the associated binding occurs, nothing else can bind to it. The binding relationship is one-to-one. Nothing else can bind to the PersistentVolume once claimed. The following `get` command uses the short-form `pvc` instead of `persistentvolumeclaim`:

```
$ kubectl create -f db-pvc.yaml
persistentvolumeclaim/db-pvc created
$ kubectl get pvc db-pvc
NAME      STATUS    VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
db-pvc   Bound     db-pv    1Gi        RWO          <none>        111s
```

The PersistentVolume has not been mounted by a Pod yet. Therefore, inspecting the details of the object shows `<none>`. Using the `describe` command is a good way to verify if the PersistentVolumeClaim was mounted properly:

```
$ kubectl describe pvc db-pvc
...
Used By:      <none>
...
```

Mounting PersistentVolumeClaims in a Pod

All that's left is to mount the PersistentVolumeClaim in the Pod that wants to consume it. You already learned how to mount a volume in a Pod. The big difference here, shown in [Example 7-4](#), is using `spec.volumes[].persistentVolumeClaim` and providing the name of the PersistentVolumeClaim.

Example 7-4. A Pod referencing a PersistentVolumeClaim

```
apiVersion: v1
kind: Pod
metadata:
  name: app-consuming-pvc
spec:
  volumes:
  - name: app-storage
    persistentVolumeClaim:
      claimName: db-pvc
  containers:
  - image: alpine:3.18.2
    name: app
    command: ["/bin/sh"]
```

```
args: ["-c", "while true; do sleep 60; done;"]
volumeMounts:
- mountPath: "/mnt/data"
  name: app-storage
```

Let's assume we stored the configuration in the file *app-consuming-pvc.yaml*. After creating the Pod from the manifest, you should see the Pod transitioning into the Ready state. The `describe` command will provide additional information on the volume:

```
$ kubectl create -f app-consuming-pvc.yaml
pod/app-consuming-pvc created
$ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
app-consuming-pvc  1/1     Running   0          3s
$ kubectl describe pod app-consuming-pvc
...
Volumes:
  app-storage:
    Type:      PersistentVolumeClaim (a reference to a PersistentVolumeClaim \
              in the same namespace)
    ClaimName: db-pvc
    ReadOnly:   false
...

```

The PersistentVolumeClaim now also shows the Pod that mounted it:

```
$ kubectl describe pvc db-pvc
...
Used By:        app-consuming-pvc
...
```

You can now go ahead and open an interactive shell to the Pod. Navigating to the mount path at */mnt/data* gives you access to the underlying PersistentVolume:

```
$ kubectl exec app-consuming-pvc -it -- /bin/sh
/ # cd /mnt/data
/mnt/data # ls -l
total 0
/mnt/data # touch test.db
/mnt/data # ls -l
total 0
-rw-r--r--    1 root      root            0 Sep 29 23:59 test.db
```

Storage Classes

A storage class is a Kubernetes primitive that defines a specific type or “class” of storage. Typical storage characteristics can be the type (e.g., fast SSD storage versus remote cloud storage or the backup policy for storage). The storage class is used to provision a PersistentVolume dynamically based on its criteria. In practice, this means that you do not have to create the PersistentVolume object yourself. The

provisioner assigned to the storage class takes care of it. Most Kubernetes cloud providers come with a list of existing provisioners. Minikube already creates a default storage class named `standard`, which you can query with the following command:

```
$ kubectl get storageclass
NAME          PROVISIONER          RECLAIMPOLICY \
VOLUMEBINDINGMODE ALLOWVOLUMEEXPANSION AGE
standard (default) k8s.io/minikube-hostpath Delete      \
Immediate           false            108d
```

Creating storage classes

Storage classes can be created declaratively only with the help of a YAML manifest. At a minimum, you need to declare the provisioner. All other attributes are optional and use default values if not provided upon creation. Most provisioners let you set parameters specific to the storage type. [Example 7-5](#) defines a storage class on Google Compute Engine denoted by the provisioner `kubernetes.io/gce-pd`.

Example 7-5. Definition of a storage class

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
  replication-type: regional-pd
```

If you saved the YAML contents in the file `fast-sc.yaml`, then the following command will create the object. The storage class can be listed using the `get storageclass` command:

```
$ kubectl create -f fast-sc.yaml
storageclass.storage.k8s.io/fast created
$ kubectl get storageclass
NAME          PROVISIONER          RECLAIMPOLICY \
VOLUMEBINDINGMODE ALLOWVOLUMEEXPANSION AGE
fast           kubernetes.io/gce-pd Delete      \
Immediate           false            4s
...
```

Using storage classes

Provisioning a PersistentVolume dynamically requires assigning of the storage class when you create the PersistentVolumeClaim. [Example 7-6](#) shows the usage of the attribute `spec.storageClassName` for assigning the storage class named `standard`.

Example 7-6. Using a storage class in a PersistentVolumeClaim

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: db-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 512Mi
  storageClassName: standard
```

The corresponding PersistentVolume object will be created only if the storage class can provision a fitting PersistentVolume using its provisioner. It's important to understand that Kubernetes does not render an error or warning message if it isn't the case.

The following command renders the created PersistentVolumeClaim and PersistentVolume. As you can see, the name of the dynamically provisioned PersistentVolume uses a hash to ensure a unique naming:

```
$ kubectl get pv,pvc
NAME                                     CAPACITY \
ACCESS MODES   RECLAIM POLICY  STATUS   CLAIM           STORAGECLASS \
REASON AGE
persistentvolume/pvc-b820b919-f7f7-4c74-9212-ef259d421734  512Mi \
RWO          Delete        Bound    default/db-pvc  standard \
2s

NAME           STATUS  VOLUME \
CAPACITY ACCESS MODES STORAGECLASS AGE
persistentvolumeclaim/db-pvc Bound  pvc-b820b919-f7f7-4c74-9212-ef259d421734 \
512Mi   RWO       standard  2s
```

The steps for mounting the PersistentVolumeClaim from a Pod are the same as for static and dynamic provisioning. Refer to “[Mounting PersistentVolumeClaims in a Pod](#)” on page 76 for more information.

Summary

Kubernetes offers the concept of a Volume to implement the use case. A Pod mounts a Volume to a path in the container. Kubernetes offers a wide range of Volume types to fulfill different requirements.

PersistentVolumes store data beyond a Pod or cluster/node restart. Those objects are decoupled from the Pod's life cycle and are therefore represented by a Kubernetes primitive. The PersistentVolumeClaim abstracts the underlying implementation

details of a PersistentVolume and acts as an intermediary between Pod and PersistentVolume.

Exam Essentials

Practice defining and consuming ephemeral Volumes

Volumes are a cross-cutting concept applied in different areas of the exam. Know where to find the relevant documentation for defining a Volume as well as the multitude of ways to consume a Volume from a container. Definitely read [Chapter 19](#) for a deep dive on how to mount ConfigMaps and Secrets as a Volume, and [Chapter 8](#) for sharing a Volume between two containers.

Internalize the mechanics of defining and consuming a PersistentVolume

Creating a PersistentVolume involves a couple of moving parts. Understand the configuration options for PersistentVolumes and PersistentVolumeClaims and how they play together. Try to emulate situations that prevent a successful binding of a PersistentVolumeClaim. Then fix the situation by taking counteractions. Internalize the short-form commands `pv` and `pvc` to save precious time during the exam.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. Create a Pod YAML manifest with two containers that use the image `alpine:3.12.0`. Provide a command for both containers that keep them running forever. Define a Volume of type `emptyDir` for the Pod. Container 1 should mount the Volume to path `/etc/a`, and container 2 should mount the Volume to path `/etc/b`. Open an interactive shell for container 1 and create the directory `data` in the mount path. Navigate to the directory and create the file `hello.txt` with the contents “Hello World.” Exit out of the container. Open an interactive shell for container 2 and navigate to the directory `/etc/b/data`. Inspect the contents of file `hello.txt`. Exit out of the container.
2. Create a PersistentVolume named `logs-pv` that maps to the `hostPath /var/logs`. The access mode should be `ReadWriteOnce` and `ReadOnlyMany`. Provision a storage capacity of `5Gi`. Ensure that the status of the PersistentVolume shows `Available`. Create a PersistentVolumeClaim named `logs-pvc`. It uses `ReadWriteOnce` access. Request a capacity of `2Gi`. Ensure that the status of the PersistentVolume shows `Bound`.

Mount the PersistentVolumeClaim in a Pod running the image `nginx` at the mount path `/var/log/nginx`.

Open an interactive shell to the container and create a new file named `my-nginx.log` in `/var/log/nginx`. Exit out of the Pod.

Delete the Pod and re-create it with the same YAML manifest. Open an interactive shell to the Pod, navigate to the directory `/var/log/nginx`, and find the file you created before.

Multi-Container Pods

[Chapter 5](#) explained how to manage single-container Pods. That's the norm, as you'll want to run a microservice inside of a single Pod to reinforce separation of concerns and increase cohesion. Technically, a Pod allows you to configure and run multiple containers.

In this chapter, we'll discuss the need for multi-container Pods, their relevant use cases, and the design patterns that have emerged in the Kubernetes community. The exam outline specifically mentions prominent design patterns: the init container, the sidecar container, and others. We'll get a good grasp of their application with the help of representative examples.

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objective:

- Understand multi-container Pod design patterns

Working with Multiple Containers in a Pod

Especially for Kubernetes beginners, how to appropriately design a Pod isn't necessarily apparent. If you read the Kubernetes user documentation and tutorials on the internet, you'll quickly discover that you can create a Pod that runs multiple containers at the same time. The question then arises: "Should I deploy my microservices stack to a single Pod with multiple containers, or should I create multiple Pods, each running a single microservice?" The short answer is to operate a single microservice per Pod. This modus operandi promotes a decentralized, decoupled, and distributed

architecture. Furthermore, it helps with rolling out new versions of a microservice without necessarily interrupting other parts of the system.

So, what's the point of running multiple containers in a Pod? There are two common use cases. Sometimes, you'll want to initialize your Pod by executing setup scripts, commands, or any other kind of preconfiguration procedure before the application container should start. This logic runs in an init container. Other times, you'll want to provide helper functionality that runs alongside the application container to avoid the need to bake the logic into application code. For example, you may want to massage the log output produced by the application. Containers running helper logic are called *sidecars*.

Init Containers

Init containers provide initialization logic concerns to be run before the main application starts. To draw an analogy, let's look at a similar concept in programming languages. Many programming languages, especially object-oriented ones like Java or C++, come with a constructor or a static method block. Those language constructs initialize fields, validate data, and set the stage before a class can be created. Not all classes need a constructor, but they are equipped with the capability.

In Kubernetes, this functionality can be achieved with the help of init containers. Init containers are always started before the main application containers, which means they have their own life cycle. To split up the initialization logic, you can even distribute the work into multiple init containers that are run in the order of definition in the manifest. Of course, initialization logic can fail. If an init container produces an error, the whole Pod is restarted, causing all init containers to run again in sequential order. Thus, to prevent any side effects, making init container logic idempotent is a good practice. [Figure 8-1](#) shows a Pod with two init containers and the main application.

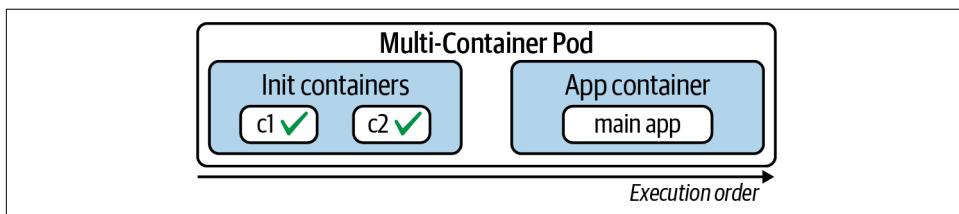


Figure 8-1. Sequential and atomic life cycle of init containers in a Pod

In the past couple of chapters, we've explored how to define a container within a Pod: you simply specify its configuration under `spec.containers`. For init containers, Kubernetes provides a separate section: `spec.initContainers`. Init containers are always executed before the main application containers, regardless of the definition order in the manifest.

The manifest shown in [Example 8-1](#) defines an init container and a main application container. For the most part, init containers use the same attributes as regular containers. There's one big difference, however. They cannot define probes, discussed in [Chapter 14](#). The init container sets up a configuration file in the directory `/usr/shared/app`. This directory has been shared through a Volume so that it can be referenced by a Node.js-based application running in the main container.

Example 8-1. A Pod defining an init container

```
apiVersion: v1
kind: Pod
metadata:
  name: business-app
spec:
  initContainers:
    - name: configurer
      image: busybox:1.36.1
      command: ['sh', '-c', 'echo Configuring application... && \
                  mkdir -p /usr/shared/app && echo -e "{\"dbConfig\": \" \
                  {\"host\":\"localhost\", \"port\":5432, \"dbName\":\"customers\"}\"} \
                  > /usr/shared/app/config.json']
  volumeMounts:
    - name: configdir
      mountPath: "/usr/shared/app"
  containers:
    - image: bmuschko/nodejs-read-config:1.0.0
      name: web
      ports:
        - containerPort: 8080
      volumeMounts:
        - name: configdir
          mountPath: "/usr/shared/app"
  volumes:
    - name: configdir
      emptyDir: {}
```

When starting the Pod, you'll see that the status column of the `get` command provides information on init containers as well. The prefix `Init:` signifies that an init container is in the process of being executed. The status portion after the colon character shows the number of init containers completed versus the overall number of init containers configured:

```
$ kubectl create -f init.yaml
pod/business-app created
$ kubectl get pod business-app
NAME      READY   STATUS    RESTARTS   AGE
business-app  0/1     Init:0/1  0          2s
$ kubectl get pod business-app
```

NAME	READY	STATUS	RESTARTS	AGE
business-app	1/1	Running	0	8s

Errors can occur during the execution of init containers. If any container fails in the sequential initialization chain, then the whole Pod will fail to start. You can always retrieve the logs of an init container by using the `--container` command-line option (or `-c` in its short form), as shown in [Figure 8-2](#).

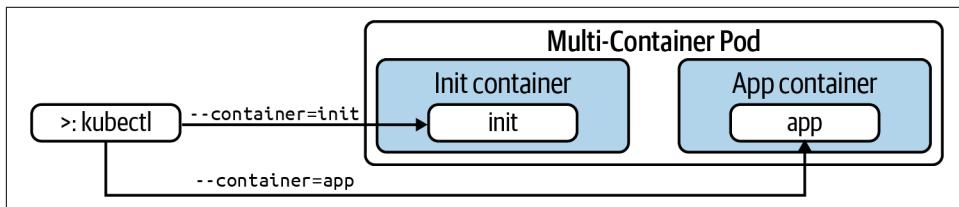


Figure 8-2. Targeting a specific container

The following command renders the logs of the `configurer` init container, which equates to the `echo` command we configured in the YAML manifest:

```
$ kubectl logs business-app -c configurer
Configuring application...
```

The Sidecar Pattern

The life cycle of an init container looks like this: it starts up, runs its logic, then terminates once the work has been done. Init containers are not meant to keep running over a longer period of time. But some scenarios call for a different usage pattern. For example, you may want to create a Pod that runs multiple containers continuously alongside one another.



Sidecar containers introduced in Kubernetes 1.29

Future exams using Kubernetes 1.29 or higher may cover the formalized **sidecar container**. Sidecar containers are secondary containers that will start with the Pod and remain running during the entire life of the Pod.

Typically, there are two different categories of containers: the container that runs the application and another container that provides helper functionality to the primary application. In the Kubernetes space, the container providing helper functionality is called a sidecar. The most commonly used capabilities of a sidecar container include file synchronization, logging, and watcher capabilities. The sidecars are not part of the main traffic or API of the primary application. They usually operate asynchronously and are not involved in the public API.

To illustrate the behavior of a sidecar, consider the following use case. The main application container runs a web server—in this case, NGINX. Once started, the web server produces two standard logfiles. The file `/var/log/nginx/access.log` captures requests to the web server’s endpoint. The other file, `/var/log/nginx/error.log`, records failures while processing incoming requests.

As part of the Pod’s functionality, we want to implement a monitoring service. The sidecar container polls the file’s `error.log` periodically and checks if any failures have been discovered. More specifically, the service tries to find failures assigned to the error log level, indicated by `[error]` in the log file. If an error is found, the monitoring service will react to it. For example, it could send a notification to the system administrators. We want to keep the functionality as simple as possible. The monitoring service will simply render an error message to standard output. The file exchange between the main application container and the sidecar container happens through a Volume, as shown in [Figure 8-3](#).

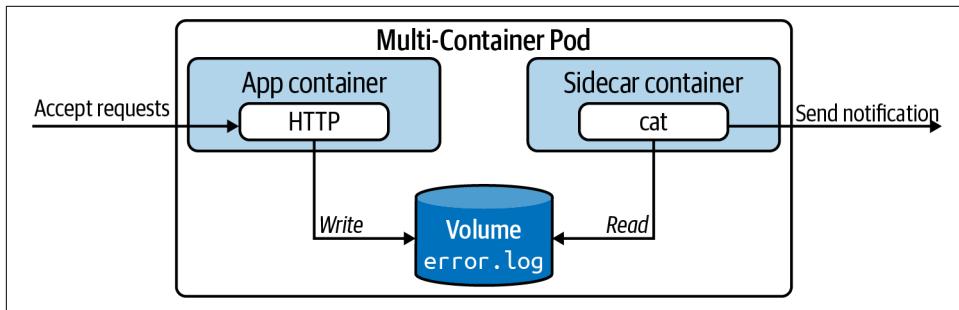


Figure 8-3. The sidecar pattern in action

The YAML manifest shown in [Example 8-2](#) sets up the described scenario. The trickiest portion of the code is the lengthy bash command. This command runs an infinite loop. As part of each iteration, we inspect the contents of the file `error.log`, `grep` for an error and potentially act on it. The loop executes every 10 seconds.

Example 8-2. An exemplary sidecar pattern implementation

```
apiVersion: v1
kind: Pod
metadata:
  name: webserver
spec:
  containers:
    - name: nginx
      image: nginx:1.25.1
      volumeMounts:
        - name: logs-vol
          mountPath: /var/log/nginx
```

```

- name: sidecar
  image: busybox:1.36.1
  command: ["sh", "-c", "while true; do if [ \"$(cat /var/log/nginx/error.log \
    | grep 'error')\" != \"\" ]; then echo 'Error discovered!'; fi; \
    sleep 10; done"]
  volumeMounts:
  - name: logs-vol
    mountPath: /var/log/nginx
  volumes:
  - name: logs-vol
    emptyDir: {}

```

When starting up the Pod, you'll notice that the overall number of containers will show 2. After all containers can be started, the Pod signals a `Running` status:

```

$ kubectl create -f sidecar.yaml
pod/webserver created
$ kubectl get pods webserver
NAME      READY   STATUS      RESTARTS   AGE
webserver  0/2     ContainerCreating   0          4s
$ kubectl get pods webserver
NAME      READY   STATUS      RESTARTS   AGE
webserver  2/2     Running   0          5s

```

You will find that `error.log` does not contain any failure to begin with. It starts out as an empty file. With the following commands, you'll provoke an error on purpose. After waiting for at least 10 seconds, you'll find the expected message on the terminal, which you can query for with the `logs` command:

```

$ kubectl logs webserver -c sidecar
$ kubectl exec webserver -it -c sidecar -- /bin/sh
/ # wget -O- localhost:unknown
Connecting to localhost (127.0.0.1:80)
wget: server returned error: HTTP/1.1 404 Not Found
/ # cat /var/log/nginx/error.log
2020/07/18 17:26:46 [error] 29#29: *2 open() "/usr/share/nginx/html/unknown" \
failed (2: No such file or directory), client: 127.0.0.1, server: localhost, \
request: "GET /unknown HTTP/1.1", host: "localhost"
/ # exit
$ kubectl logs webserver -c sidecar
Error discovered!

```

The Adapter Pattern

As application developers, we want to focus on implementing business logic. For example, as part of a two-week sprint, we're tasked with adding a shopping cart feature. In addition to the functional requirements, we also have to think about operational aspects such as exposing administrative endpoints or crafting meaningful and properly formatted log output. It's easy to fall into the habit of rolling all aspects into the application code, which makes it more complex and harder to maintain.

Cross-cutting concerns in particular need to be replicated across multiple applications and are often copied and pasted from one code base to another.

In Kubernetes, we can avoid bundling cross-cutting concerns into the application code by running them in another container apart from the main application container. The adapter pattern transforms the output produced by the application to make it consumable in the format needed by another part of the system. [Figure 8-4](#) illustrates a concrete example of the adapter pattern.

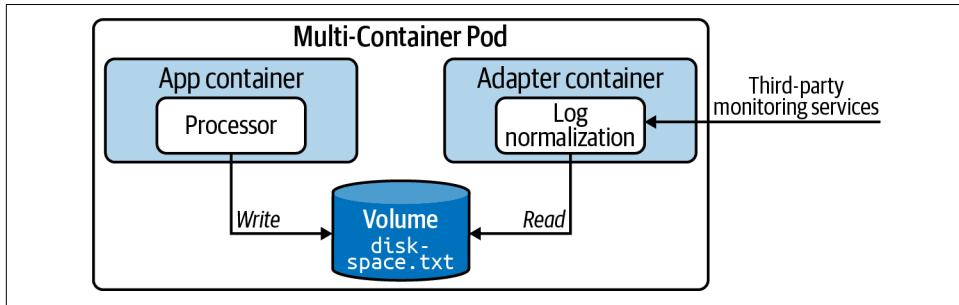


Figure 8-4. The adapter pattern in action

The business application running the main container produces timestamped information—in this case, the available disk space—and writes it to the file *diskspace.txt*. As part of the architecture, we want to consume the file from a third-party monitoring application. The problem is that the external application requires the information to exclude the timestamp. Of course, we could change the logging format to avoid writing the timestamp, but what if we actually want to know when the log entry has been written? This is where the adapter pattern can help. A sidecar container executes transformation logic that turns the log entries into the format needed by the external system without having to change application logic.

The YAML manifest in [Example 8-3](#) illustrates what this implementation of the adapter pattern could look like. The app container produces a new log entry every five seconds. The transformer container consumes the contents of the file, removes the timestamp, and writes it to a new file. Both containers have access to the same mount path through a Volume.

Example 8-3. An exemplary adapter pattern implementation

```
apiVersion: v1
kind: Pod
metadata:
  name: adapter
spec:
  containers:
    - args:
```

```

- /bin/sh
- -c
- 'while true; do echo "$(date) | $(du -sh ~)" >> /var/logs/diskspace.txt; \
  sleep 5; done;'
image: busybox:1.36.1
name: app
volumeMounts:
- name: config-volume
  mountPath: /var/logs
image: busybox:1.36.1
name: transformer
args:
- /bin/sh
- -c
- 'sleep 20; while true; do while read LINE; do echo "$LINE" | cut -f2 -d"|" \
  >> $(date +%Y-%m-%d-%H-%M-%S)-transformed.txt; done < \
  /var/logs/diskspace.txt; sleep 20; done;'
volumeMounts:
- name: config-volume
  mountPath: /var/logs
volumes:
- name: config-volume
  emptyDir: {}

```

After creating the Pod, we'll find two running containers. We should be able to locate the original file, `/var/logs/diskspace.txt`, after shelling into the `transformer` container. The transformed data exists in a separate file in the user home directory:

```

$ kubectl create -f adapter.yaml
pod/adapter created
$ kubectl get pods adapter
NAME      READY   STATUS    RESTARTS   AGE
adapter   2/2     Running   0          10s
$ kubectl exec adapter --container=transformer -it -- /bin/sh
/ # cat /var/logs/diskspace.txt
Sun Jul 19 20:28:07 UTC 2020 | 4.0K      /root
Sun Jul 19 20:28:12 UTC 2020 | 4.0K      /root
/ # ls -l
total 40
-rw-r--r-- 1 root root 60 Jul 19 20:28 2020-07-19-20-28-28-transformed.txt
...
/ # cat 2020-07-19-20-28-28-transformed.txt
4.0K   /root
4.0K   /root

```

The Ambassador Pattern

Another important design pattern covered by the CKAD is the ambassador pattern. The ambassador pattern provides a proxy for communicating with external services.

Many use cases can justify the introduction of the ambassador pattern. The overarching goal is to hide and/or abstract the complexity of interacting with other parts of the system. Typical responsibilities include retry logic upon a request failure, security concerns such as providing authentication or authorization, and monitoring latency or resource usage. [Figure 8-5](#) illustrates this pattern.

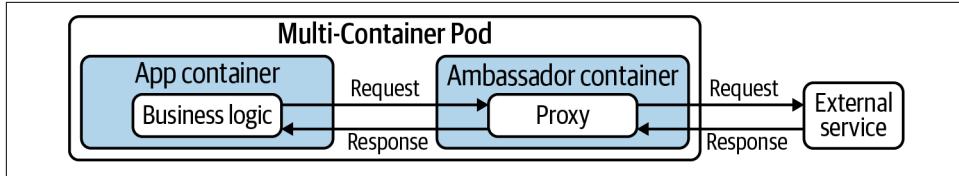


Figure 8-5. The ambassador pattern in action

In this example, we'll want to implement rate-limiting functionality for HTTP(S) calls to an external service. For example, the requirements for the rate limiter could say that an application can make a maximum of 5 calls every 15 minutes. Instead of strongly coupling the rate-limiting logic to the application code, it will be provided by an ambassador container. Any calls made from the business application need to be funneled through the ambassador container. [Example 8-4](#) shows a Node.js-based rate limiter implementation that makes calls to the external service Postman.

Example 8-4. Node.js HTTP rate limiter implementation

```
const express = require('express');
const app = express();
const rateLimit = require('express-rate-limit');
const https = require('https');

const rateLimiter = rateLimit({
  windowMs: 15 * 60 * 1000,
  max: 5,
  message:
    'Too many requests have been made from this IP, please try again after an hour'
});

app.get('/test', rateLimiter, function (req, res) {
  console.log('Received request...');
  var id = req.query.id;
  var url = 'https://postman-echo.com/get?test=' + id;
  console.log("Calling URL %s", url);

  https.get(url, (resp) => {
    let data = '';

    resp.on('data', (chunk) => {
      data += chunk;
    });
  });
});
```

```

    res.on('end', () => {
      res.send(data);
    });

  }).on("error", (err) => {
    res.send(err.message);
  });
}

var server = app.listen(8081, function () {
  var port = server.address().port
  console.log("Ambassador listening on port %s...", port)
})

```

The corresponding Pod shown in [Example 8-5](#) runs the main application container on a different port than the ambassador container. Every call to the HTTP endpoint of the container named `business-app` would delegate to the HTTP endpoint of the container named `ambassador`. It's important to mention that containers running inside of the same Pod can communicate via `localhost`. No additional networking configuration is required.

Example 8-5. An exemplary ambassador pattern implementation

```

apiVersion: v1
kind: Pod
metadata:
  name: rate-limiter
spec:
  containers:
    - name: business-app
      image: bmuschko/nodejs-business-app:1.0.0
      ports:
        - containerPort: 8080
    - name: ambassador
      image: bmuschko/nodejs-ambassador:1.0.0
      ports:
        - containerPort: 8081

```

Let's test the functionality. First, we'll create the Pod, shell into the container that runs the business application, and execute a series of `curl` commands. The first five calls will be allowed to the external service. On the sixth call, we'll receive an error message, as the rate limit has been reached within the given time frame:

```

$ kubectl create -f ambassador.yaml
pod/rate-limiter created
$ kubectl get pods rate-limiter
NAME          READY   STATUS    RESTARTS   AGE
rate-limiter  2/2     Running   0          5s
$ kubectl exec rate-limiter -it -c business-app -- /bin/sh

```

```
# curl localhost:8080/test
>{"args":{"test":"123"},"headers":{"x-forwarded-proto":"https", \
"x-forwarded-port": "443", "host": "postman-echo.com", \
"x-amzn-trace-id": "Root=1-5f177dba-e736991e882d12fcffd23f34"}, \
"url": "https://postman-echo.com/get?test=123"}
...
# curl localhost:8080/test
Too many requests have been made from this IP, please try again after an hour
```

Summary

Real-world scenarios call for running multiple containers inside of a Pod. An init container helps with setting the stage for the main application container by executing initializing logic. Once the initialized logic has been processed, the container will be terminated. The main application container starts only if the init container ran through its functionality successfully.

Other design patterns that involve multiple containers per Pod are the adapter pattern and the ambassador pattern. The adapter pattern helps with “translating” data produced by the application so that it becomes consumable by third-party services. The ambassador pattern acts as a proxy for the application container when communicating with external services by abstracting the “how.”

Exam Essentials

Understand the need for running multiple containers in a Pod

Pods can run multiple containers. You will need to understand the difference between init containers and sidecar containers and their respective life cycles. Practice accessing a specific container in a multi-container Pod with the help of the command-line option `--container` or `-c`.

Know how to create an init container

Init containers see a lot of use in enterprise Kubernetes cluster environments. Understand the need for using them in their respective scenarios. Practice defining a Pod with one or even more init containers and observe their linear execution when creating the Pod. It’s important to experience the behavior of a Pod in failure situations that occur in an init container.

Understand multi-container design patterns and how to implement them

Multi-container Pods are best understood by implementing a scenario for one of the established patterns. Based on what you’ve learned, come up with your own applicable use case and create a multi-container Pod to solve it. It’s helpful to be able to identify sidecar patterns and understand why they are important in practice and how to stand them up yourself. As you implement your own sidecars, you may notice that you have to brush up on your knowledge of bash.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. Create a YAML manifest for a Pod named `complex-pod`. The main application container named `app` should use the image `nginx:1.25.1` and expose the container port 80. Modify the YAML manifest so that the Pod defines an init container named `setup` that uses the image `busybox:1.36.1`. The init container runs the command `wget -O- google.com`.

Create the Pod from the YAML manifest.

Download the logs of the init container. You should see the output of the `wget` command.

Open an interactive shell to the main application container and run the `ls` command. Exit out of the container.

Force-delete the Pod.

2. Create a YAML manifest for a Pod named `data-exchange`. The main application container named `main-app` should use the image `busybox:1.36.1`. The container runs a command that writes a new file every 30 seconds in an infinite loop in the directory `/var/app/data`. The filename follows the pattern `{counter++}-data.txt`. The variable counter is incremented every interval and starts with the value 1.

Modify the YAML manifest by adding a sidecar container named `sidecar`. The sidecar container uses the image `busybox:1.36.1` and runs a command that counts the number of files produced by the `main-app` container every 60 seconds in an infinite loop. The command writes the number of files to standard output.

Define a Volume of type `emptyDir`. Mount the path `/var/app/data` for both containers.

Create the Pod. Tail the logs of the sidecar container.

Delete the Pod.

Labels and Annotations

The exam curriculum doesn't explicitly mention the concept of labels; however, it's an important one for understanding how certain Kubernetes primitives function internally. To avoid confusing labels with annotations, we'll also discuss the commonalities and differences among those concepts.

Labels are an essential tool for querying, filtering, and sorting Kubernetes objects. Annotations represent descriptive metadata for Kubernetes objects but can't be used for queries. In this chapter, you will learn how to assign and use both concepts.

Coverage of Curriculum Objectives

The curriculum doesn't explicitly mention coverage of labels and annotations. It's important to understand labels because primitives like the Deployment, Service, and NetworkPolicy can't function without them.

Working with Labels

Kubernetes lets you assign key-value pairs to objects so that you can use them later within a search query. Those key-value pairs are called *labels*. To draw an analogy, you can think of labels as tags for a blog post.

A label describes a Kubernetes object in distinct terms (e.g., a category like "frontend" or "backend"), but it is not meant for elaborate, multi-word descriptions of its functionality. As part of the specification, Kubernetes limits the length of a label to a maximum of 63 characters and a range of allowed alphanumeric and separator characters.

Figure 9-1 shows the Pods named `frontend`, `backend`, and `database`. Each of the Pods declares a unique set of labels.

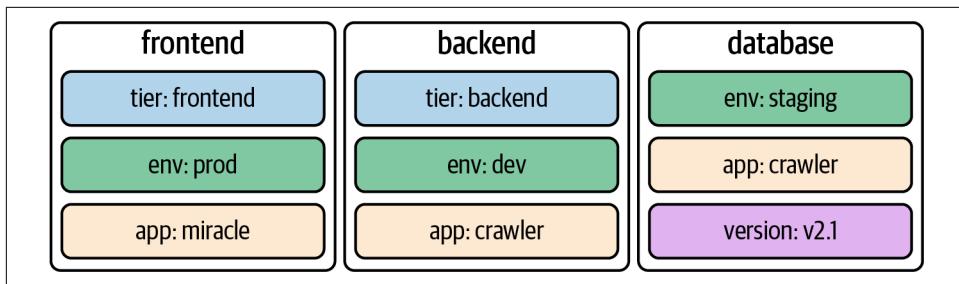


Figure 9-1. Pods with labels

It's common practice to assign one or many labels to an object at creation time; however, you can modify them as needed for a live object. When you see labels for the first time, they might seem insignificant—but their importance cannot be overstated. They're essential for understanding the runtime behavior of more advanced Kubernetes objects like a Deployment and a Service. Later in this chapter, we'll see the significance of labels in practice when learning about Deployments in more detail.

Declaring Labels

You can declare labels imperatively with the `run` command or declaratively in the `metadata.labels` section in the YAML manifest. The command-line option `--labels` (or `-l` in its short form) defines a comma-separated list of labels when creating a Pod. The following command creates a new Pod with two labels from the command line:

```
$ kubectl run labeled-pod --image=nginx:1.25.1 \
  --labels=tier=backend,env=dev
pod/labeled-pod created
```

Assigning labels to Kubernetes objects by editing the manifest requires a change to the `metadata` section. [Example 9-1](#) shows the Pod definition from the previous command if we were to start with the YAML manifest.

Example 9-1. A Pod defining two labels

```
apiVersion: v1
kind: Pod
metadata:
  name: labeled-pod
  labels:
    env: dev
    tier: backend
spec:
  containers:
    - image: nginx:1.25.1
      name: nginx
```

Inspecting Labels

You can inspect the labels assigned to a Kubernetes object from different angles. Here, we'll want to look at the most common ways to identify the labels of a Pod. As with any other runtime information, you can use the `describe` or `get` commands to retrieve the labels:

```
$ kubectl describe pod labeled-pod | grep -C 2 Labels:  
...  
Labels:      env=dev  
             tier=backend  
...  
$ kubectl get pod labeled-pod -o yaml | grep -C 1 labels:  
metadata:  
  labels:  
    env: dev  
    tier: backend  
...
```

If you want to list the labels for all object types or a specific object type, use the `--show-labels` command-line option. This option is convenient if you need to sift through a longer list of objects. The output automatically adds a new column named `LABELS`:

```
$ kubectl get pods --show-labels  
NAME      READY   STATUS    RESTARTS   AGE   LABELS  
labeled-pod  1/1     Running   0          38m   env=dev,tier=backend
```

Modifying Labels for a Live Object

You can add or remove a label from an existing Kubernetes object, or simply modify an existing label at any time. One way to achieve this is by editing the live object and changing the label definition in the `metadata.labels` section. The other option, which offers a slightly faster turnaround, is the `label` command. The following commands add a new label, change the value of the label, and then remove the label with the minus character:

```
$ kubectl label pod labeled-pod region=eu  
pod/labeled-pod labeled  
$ kubectl get pod labeled-pod --show-labels  
NAME      READY   STATUS    RESTARTS   AGE   LABELS  
labeled-pod  1/1     Running   0          22h   env=dev,region=eu,tier=backend  
$ kubectl label pod labeled-pod region=us --overwrite  
pod/labeled-pod labeled  
$ kubectl get pod labeled-pod --show-labels  
NAME      READY   STATUS    RESTARTS   AGE   LABELS  
labeled-pod  1/1     Running   0          22h   env=dev,region=us,tier=backend  
$ kubectl label pod labeled-pod region-  
pod/labeled-pod labeled  
$ kubectl get pod labeled-pod --show-labels
```

NAME	READY	STATUS	RESTARTS	AGE	LABELS
labeled-pod	1/1	Running	0	22h	env=dev, tier=backend

Using Label Selectors

Labels become meaningful only when combined with the selection feature. A label selector uses a set of criteria to query for Kubernetes objects. For example, you could use a label selector to express “select all Pods with the label assignment `env=dev`, `tier=frontend`, and have a label with the key `version` independent of the assigned value,” as shown in [Figure 9-2](#).

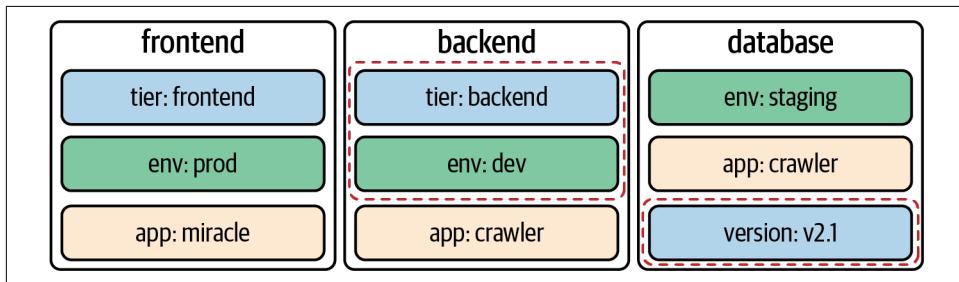


Figure 9-2. Selecting Pods by label criteria

Kubernetes offers two ways to select objects by labels: from the command line and within a manifest. Let’s talk about both options.

Label selection from the command line

On the command line, you can select objects by label using the `--selector` option (`-l` in its short-form notation). You can express a filter by providing an equality-based requirement or a set-based requirement. Both requirement types can be combined in a single query.

An *equality-based requirement* can use the operators `=`, `==`, or `!=`. You can separate multiple filter terms with a comma and then combine them with a boolean AND. At this time, equality-based label selection cannot express a boolean OR operation. A typical expression could say, “select all Pods with the label assignment `env=prod`.”

A *set-based requirement* can filter objects based on a set of values using the operators `in`, `notin`, and `exists`. The `in` and `notin` operators work based on a boolean OR. A typical expression could say, “select all Pods with the label key `env` and the value `prod` or `dev`.”

To demonstrate the functionality, we’ll start by setting up three different Pods with labels. All `kubectl` commands use the command-line option `--show-labels` to compare the results with our expectations. The `--show-labels` option is not needed for label selection:

```
$ kubectl run frontend --image=nginx:1.25.1 --labels=env=prod,team=shiny
pod/frontend created
$ kubectl run backend --image=nginx:1.25.1 --labels=env=prod,team=legacy, \
app=v1.2.4
pod/backend created
$ kubectl run database --image=nginx:1.25.1 --labels=env=prod,team=storage
pod/database created
$ kubectl get pods --show-labels
NAME      READY   STATUS    RESTARTS   AGE   LABELS
backend   1/1     Running   0          37s   app=v1.2.4,env=prod,team=legacy
database  1/1     Running   0          32s   env=prod,team=storage
frontend  1/1     Running   0          42s   env=prod,team=shiny
```

We'll start by filtering the Pods with an equality-based requirement. Here, we are looking for all Pods with the label assignment `env=prod`. The result returns all three Pods:

```
$ kubectl get pods -l env=prod --show-labels
NAME      READY   STATUS    RESTARTS   AGE   LABELS
backend   1/1     Running   0          37s   app=v1.2.4,env=prod,team=legacy
database  1/1     Running   0          32s   env=prod,team=storage
frontend  1/1     Running   0          42s   env=prod,team=shiny
```

The next filter operation uses a set-based requirement. We are asking for all Pods that have the label key `team` with the values `storage` or `shiny`. The result returns only the Pods named `backend` and `frontend`:

```
$ kubectl get pods -l 'team in (shiny, legacy)' --show-labels
NAME      READY   STATUS    RESTARTS   AGE   LABELS
backend   1/1     Running   0          19m   app=v1.2.4,env=prod,team=legacy
frontend  1/1     Running   0          20m   env=prod,team=shiny
```

Finally, we'll combine an equality-based requirement with a set-based requirement. The result returns only the `backend` Pod:

```
$ kubectl get pods -l 'team in (shiny, legacy)',app=v1.2.4 --show-labels
NAME      READY   STATUS    RESTARTS   AGE   LABELS
backend  1/1     Running   0          29m   app=v1.2.4,env=prod,team=legacy
```

Label selection in a manifest

Some advanced Kubernetes objects such as Deployments, Services, or network policies act as configuration proxies for Pods. They usually select a set of Pods by labels and then provide added value. For example, a network policy controls network traffic from and to a set of Pods. Only the Pods with matching labels will apply the network rules. [Example 9-2](#) applies the network policy to Pods with the equality-based requirement `tier=frontend`. For more details on network policies, see [Chapter 23](#).

Example 9-2. Label selection as part of the network policy API

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: frontend-network-policy
spec:
  podSelector:
    matchLabels:
      tier: frontend
...
...
```

The way you define label selection in a manifest is based on the API version of the Kubernetes resources and may differ between types. The content that follows in later chapters will use label selection heavily.

Recommended Labels

As you continue working with labels, you will likely find common key-value pairs you want to assign to objects. Many of those labels evolve around metadata for an application, for example, the name of the component you are deploying with a Pod, or its version.

Kubernetes proposes a list of **recommended labels**, all of which start with the key prefix `app.kubernetes.io`. Example 9-3 shows the assignment of version and component labels in a Pod definition.

Example 9-3. A Pod using recommended labels

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app.kubernetes.io/version: "1.25.1"
    app.kubernetes.io/component: server
spec:
  containers:
    - name: nginx
      image: nginx:1.25.1
```

Familiarize yourself with these recommended labels so you can use them across all objects you are managing. This provides the benefit of enabling tooling in the Kubernetes ecosystem to use the same terminology and enabling developers to use the same “language” when referring to application meta information.

Working with Annotations

Annotations are declared similarly to labels, but they serve a different purpose. They represent key-value pairs for providing descriptive metadata. The most important differentiator is that annotations cannot be used for querying or selecting objects. Typical examples of annotations may include SCM commit hash IDs, release information, or contact details for teams operating the object. Make sure to put the value of an annotation into single quotes or double quotes if it contains special characters or spaces. [Figure 9-3](#) illustrates a Pod with three annotations.

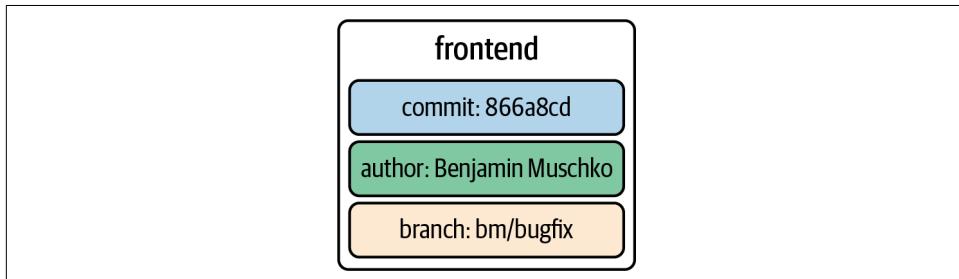


Figure 9-3. Pod with annotations

Kubernetes defines a list of reserved annotations that it will evaluate at runtime to control the runtime behavior of the object. You can find more information at [“Reserved Annotations” on page 102](#).

Declaring Annotations

The `kubectl run` command does not provide a command-line option for defining annotations that's similar to the one for labels. You will have to start by writing a YAML manifest and adding the desired annotations under `metadata.annotations`, as shown in [Example 9-4](#).

Example 9-4. A Pod defining three annotations

```
apiVersion: v1
kind: Pod
metadata:
  name: annotated-pod
  annotations:
    commit: 866a8dc
    author: 'Benjamin Muschko'
    branch: 'bm/bugfix'
spec:
  containers:
    - image: nginx:1.25.1
      name: nginx
```

Inspecting Annotations

Similar to labels, you can use the `describe` or `get` commands to retrieve the assigned annotations:

```
$ kubectl describe pod annotated-pod | grep -C 2 Annotations:  
...  
Annotations: author: Benjamin Muschko  
branch: bm/bugfix  
commit: 866a8dc  
...  
$ kubectl get pod annotated-pod -o yaml | grep -C 3 annotations:  
metadata:  
  annotations:  
    author: Benjamin Muschko  
    branch: bm/bugfix  
    commit: 866a8dc  
...
```

Modifying Annotations for a Live Object

The `annotate` command is the counterpart of the `labels` command but used for annotations. As you can see in the following examples, the usage pattern is the same:

```
$ kubectl annotate pod annotated-pod oncall='800-555-1212'  
pod/annotated-pod annotated  
$ kubectl annotate pod annotated-pod oncall='800-555-2000' --overwrite  
pod/annotated-pod annotated  
$ kubectl annotate pod annotated-pod oncall-  
pod/annotated-pod annotated
```

Reserved Annotations

Kubernetes itself and extensions to Kubernetes use the concept of annotations to configure runtime behavior for an object. For example, you can assign the reserved annotation `pod-security.kubernetes.io/enforce: "baseline"` to a namespace object to enforce **security standards** for all Pods that live in the namespace. [Example 9-5](#) shows a namespace definition that assigns the annotation.

Example 9-5. A Pod using a reserved annotation

```
apiVersion: v1  
kind: Namespace  
metadata:  
  name: secured  
  annotations:  
    pod-security.kubernetes.io/enforce: "baseline"
```

See the [Kubernetes documentation](#) for a full list of reserved annotations in Kubernetes.

Summary

Labels are a central concept for controlling the runtime behavior of more advanced Kubernetes objects. For example, in the context of a Deployment, Kubernetes requires you to use label selection to *select* the Pods the Deployment manages. You can use labels to select objects based on a query from the command line or within a manifest if supported by the primitive's API. Kubernetes suggests label keys for commonly used application metadata.

Annotations serve a different purpose; they provide human-readable, informative metadata. You cannot use annotations for querying objects. Kubernetes introduced reserved annotations as a means of flagging objects for special runtime treatment.

Exam Essentials

Practice labels declaration and selection

Labels are an extremely important concept in Kubernetes, as many other primitives work with label selection. Practice how to declare labels for different objects, and use the `-l` command-line option to query for them based on equality-based and set-based requirements. Label selection in a YAML manifest might look slightly different depending on the API version of the spec. Extensively practice label selection for primitives that use them heavily.

Understand the difference between labels and annotations

All you need to know about annotations is their declaration from the command line and in a YAML manifest. Be aware that annotations are meant only for assigning metadata to objects and they cannot be queried for.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. Create three Pods that use the image `nginx:1.25.1`. The names of the Pods should be `pod-1`, `pod-2`, and `pod-3`.

Assign the label `tier=frontend` to `pod-1` and the label `tier=backend` to `pod-2` and `pod-3`. All pods should also assign the label `team=artemidis`.

Assign the annotation with the key `deployer` to `pod-1` and `pod-3`. Use your own name as the value.

From the command line, use label selection to find all Pods with the team `artemidis` or `aircontrol` and that are considered a backend service.

2. Create a Pod with the image `nginx:1.25.1` that assigns two recommended labels: one for defining the application name with the value `F5-nginx`, and one for defining the tool used to manage the application named `helm`.

Render the assigned labels of the Pod object.

PART III

Application Deployment

Application deployment covers the Kubernetes concepts, techniques, and primitives that evolve around the deployment of an application in enterprise settings.

The following chapters cover these concepts:

- [Chapter 10](#) explains the features of the Deployment primitive by example. The chapter demonstrates how to scale Pods running an application manually and automatically. The chapter also shows the default deployment process for a new application version.
- [Chapter 11](#) expands on the lessons learned in the previous chapter. You will learn different deployment strategies, when they are suitable, and how to implement them with Kubernetes.
- [Chapter 12](#) discusses the use of the open source tool Helm for deploying more complex application stacks. The chapter focuses on the workflow of consuming an existing Helm chart available through Artifact Hub.

Deployments

A big selling point of Kubernetes is its scalability and replication. To support those features, Kubernetes offers the Deployment primitive. In this chapter we'll show the creation of a Deployment scaled to multiple replicas, how to roll out a revision of your application, how to roll back to a previous revision, and how to use auto-scalers to handle scaling concerns automatically based on the current workload.

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objective:

- Understand Deployments and how to perform rolling updates

Working with Deployments

The primitive for running an application in a container is the Pod. Using a single instance of a Pod to operate an application has its flaws—it represents a single point of failure because all traffic targeting the application is funneled to this Pod. This behavior is specifically problematic when the load increases due to higher demand (e.g., during peak shopping season for an e-commerce application or when an increasing number of microservices communicate with a centralized microservice functionality, e.g. an authentication provider).

Another important aspect of running an application in a Pod is failure tolerance. The scheduler cluster component will not reschedule a Pod in the case of a node failure, which can lead to a system outage for end users. In this chapter, we'll talk about the Kubernetes mechanics that support application scalability and failure tolerance.

A *ReplicaSet* is a Kubernetes API resource that controls multiple, identical instances of a Pod running the application, so-called replicas. It has the capability of scaling the number of replicas up or down on demand. Moreover, it knows how to roll out a new version of the application across all replicas.

A *Deployment* abstracts the functionality of ReplicaSet and manages it internally. In practice, this means you do not have to create, modify, or delete ReplicaSet objects yourself. The Deployment keeps a history of application versions and can roll back to an older version to counteract a blocking or potentially costly production issue. Furthermore, it offers the capability of scaling the number of replicas.

Figure 10-1 illustrates the relationship between a Deployment, a ReplicaSet, and its controlled replicas.

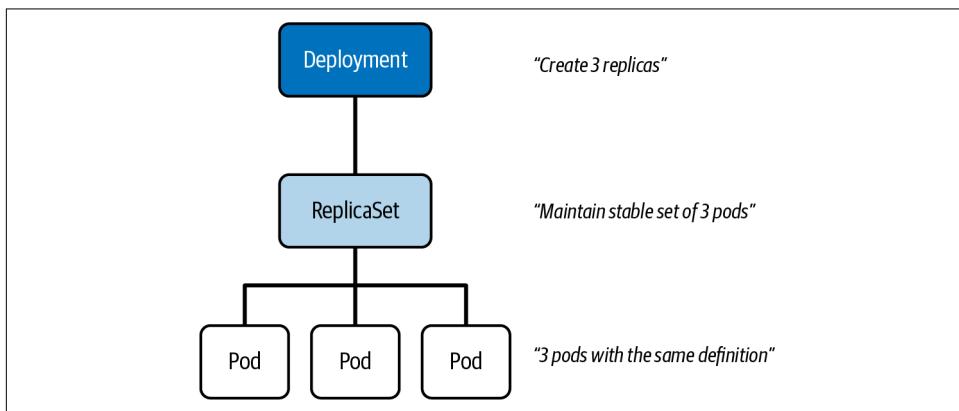


Figure 10-1. Relationship between a Deployment and a ReplicaSet

The following sections explain how to manage Deployments, including scaling and rollout features.

Creating Deployments

You can create a Deployment using the imperative command `create deployment`. The command offers a range of options, some of which are mandatory. At a minimum, you need to provide the name of the Deployment and the container image. The Deployment passes this information to the ReplicaSet, which uses it to manage the replicas. The default number of replicas created is 1; however, you can define a higher number of replicas using the option `--replicas`.

Let's observe the command in action. The following command creates the Deployment named `app-cache`, which runs the object cache `Memcached` inside the container on four replicas:

```
$ kubectl create deployment app-cache --image=memcached:1.6.8 --replicas=4
deployment.apps/app-cache created
```

The mapping between the Deployment and the replicas it controls happens through label selection. When you run the imperative command, `kubectl` sets up the mapping for you. [Example 10-1](#) shows the label selection in the YAML manifest. This YAML manifest can be used to create a Deployment declaratively or by inspecting the live object created by the previous imperative command.

Example 10-1. A YAML manifest for a Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-cache
  labels:
    app: app-cache
spec:
  replicas: 4
  selector:
    matchLabels:
      app: app-cache
  template:
    metadata:
      labels:
        app: app-cache
    spec:
      containers:
        - name: memcached
          image: memcached:1.6.8
```

When created by the imperative command, `app` is the label key the Deployment uses by default. You can find this key in three different places in the YAML output:

1. `metadata.labels`
2. `spec.selector.matchLabels`
3. `spec.template.metadata.labels`

For label selection to work properly, the assignment of `spec.selector.matchLabels` and `spec.template.metadata.labels` needs to match, as shown in [Figure 10-2](#).

The values of `metadata.labels` is irrelevant for mapping the Deployment to the Pod template. As you can see in the figure, the label assignment to `metadata.labels` has been changed deliberately to `deploy: app-cache` to underline that it is not important for the Deployment to Pod template selection.

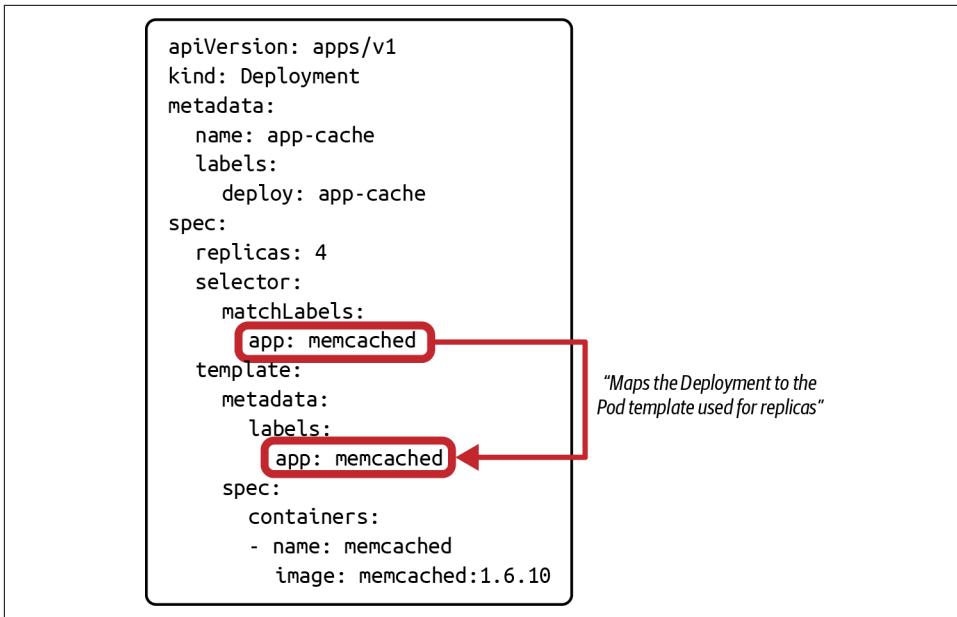


Figure 10-2. Deployment label selection

Listing Deployments and Their Pods

You can inspect a Deployment after its creation by using the `get deployments` command. The output of the command renders the important details of its replicas, as shown here:

```
$ kubectl get deployments
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
app-cache  4/4     4           4           125m
```

The column titles relevant to the replicas controlled by the Deployment are shown in [Table 10-1](#).

Table 10-1. Runtime replica information when listing deployments

Column Title	Description
READY	Lists the number of replicas available to end users in the format of <ready>/<desired>. The number of desired replicas corresponds to the value of <code>spec.replicas</code> .
UP-TO-DATE	Lists the number of replicas that have been updated to achieve the desired state.
AVAILABLE	Lists the number of replicas available to end users.

You can identify the Pods controlled by the Deployment by their naming prefix. In the case of the previously created Deployment, the Pods' names start with `app-cache-`. The hash following the prefix is autogenerated and appended to the name upon creation:

```
$ kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
app-cache-596bc5586d-84dkv 1/1     Running   0          6h5m
app-cache-596bc5586d-8bzfs 1/1     Running   0          6h5m
app-cache-596bc5586d-rc257 1/1     Running   0          6h5m
app-cache-596bc5586d-tvm4d 1/1     Running   0          6h5m
```

Rendering Deployment Details

You can render the details of a Deployment. Those details include the label selection criteria, which can be extremely valuable when troubleshooting a misconfigured Deployment. The following output provides the full gist:

```
$ kubectl describe deployment app-cache
Name:           app-cache
Namespace:      default
CreationTimestamp: Sat, 07 Aug 2021 09:44:18 -0600
Labels:         app=app-cache
Annotations:    deployment.kubernetes.io/revision: 1
Selector:       app=app-cache
Replicas:       4 desired | 4 updated | 4 total | 4 available | \
                0 unavailable
StrategyType:  RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=app-cache
  Containers:
    memcached:
      Image:      memcached:1.6.10
      Port:       <none>
      Host Port: <none>
      Environment: <none>
      Mounts:    <none>
      Volumes:   <none>
  Conditions:
    Type        Status  Reason
    ----        -----  -----
    Progressing True    NewReplicaSetAvailable
    Available   True    MinimumReplicasAvailable
    OldReplicaSets: <none>
    NewReplicaSet:  app-cache-596bc5586d (4/4 replicas created)
    Events:      <none>
```

You might have noticed that the output contains a reference to a ReplicaSet. The purpose of a ReplicaSet is to *replicate* a set of identical Pods. You do not need to deeply understand the core functionality of a ReplicaSet for the exam. Just be aware that the Deployment automatically creates the ReplicaSet and uses the Deployment's name as a prefix for the ReplicaSet, similar to the Pods it controls. In the case of the previous Deployment named app-cache, the name of the ReplicaSet is app-cache-596bc5586d.

Deleting a Deployment

A Deployment takes full charge of the creation and deletion of the objects it controls: Pods and ReplicaSets. When you delete a Deployment, the corresponding objects are deleted as well. Say you are dealing with the following set of objects shown in the output:

```
$ kubectl get deployments,pods,replicasets
NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/app-cache          4/4     4           4           6h47m
NAME                           READY   STATUS    RESTARTS   AGE
pod/app-cache-596bc5586d-84dkv  1/1    Running   0          6h47m
pod/app-cache-596bc5586d-8bzfs  1/1    Running   0          6h47m
pod/app-cache-596bc5586d-rc257  1/1    Running   0          6h47m
pod/app-cache-596bc5586d-tvm4d  1/1    Running   0          6h47m
NAME                         DESIRED   CURRENT   READY   AGE
replicaset.apps/app-cache-596bc5586d  4        4        4      6h47m
```

Run the `delete deployment` command for a cascading deletion of its managed objects:

```
$ kubectl delete deployment app-cache
deployment.apps "app-cache" deleted
$ kubectl get deployments,pods,replicasets
No resources found in default namespace.
```

Performing Rolling Updates and Rollbacks

A Deployment fully abstracts rollout and rollback capabilities by delegating this responsibility to the ReplicaSet(s) it manages. Once a user changes the definition of the Pod template in a Deployment, it will create a new ReplicaSet that applies the changes to the replicas it controls and then shut down the previous ReplicaSet. In this section, we'll talk about both scenarios: deploying a new version of an application and reverting to an old version of an application.

Updating a Deployment's Pod Template

You can choose from a range of options to update the definition of replicas controlled by a Deployment. Any of those options is valid, but they vary in ease of use and operational environment.

In real-world projects, you should check your manifest files into version control. Changes to the definition would then be made by directly editing the file. The `kubectl apply` can update a live object by pointing to the changed manifest:

```
$ kubectl apply -f deployment.yaml
```

The `kubectl edit` command lets you change the Pod template interactively by modifying the live object's manifest in an editor. To edit the Deployment live object named `web-server`, use the following command:

```
$ kubectl edit deployment web-server
```

The imperative `kubectl set image` command changes only the container image assigned to a Pod template by selecting the name of the container. For example, you could use his command to assign the image `nginx:1.25.2` to the container named `nginx` in the Deployment `web-server`:

```
$ kubectl set image deployment web-server nginx=nginx:1.25.2
```

The `kubectl replace` command lets you replace the existing Deployment with a new definition that contains your change to the manifest. The optional `--force` flag first deletes the existing object and then creates it from scratch. The following command assumes that you changed the container image assignment in `deployment.yaml`:

```
$ kubectl replace -f deployment.yaml
```

The command `kubectl patch` requires you to provide the merges as a patch to update a Deployment. The following command shows the operation in action. Here, you are sending the changes to be made in the form of a JSON structure:

```
$ kubectl patch deployment web-server -p '{"spec": {"template": {"spec": "\n        \"containers\": [{\"name\": \"nginx\", \"image\": \"nginx:1.25.2\"}]\n      }}}'
```

Rolling Out a New Revision

Deployments make it easy to roll out a new version of the application to all replicas it controls. Say you want to upgrade the version of Memcached from 1.6.8 to 1.6.10 to benefit from the latest features and bug fixes. All you need to do is change the desired state of the object by updating the Pod template. The Deployment updates all replicas to the new version one by one. This process is called the *rolling update* strategy.

The command `set image` offers a quick, convenient way to change the image of a Deployment, as shown in the following command:

```
$ kubectl set image deployment app-cache memcached=memcached:1.6.10
deployment.apps/app-cache image updated
```

You can check the current status of a rollout that's in progress using the command `rollout status`. The output indicates the number of replicas that have already been updated since emitting the command:

```
$ kubectl rollout status deployment app-cache
Waiting for rollout to finish: 2 out of 4 new replicas have been updated...
deployment "app-cache" successfully rolled out
```

Kubernetes keeps track of the changes you make to a Deployment over time in the rollout history. Every change is represented by a *revision*. When changing the Pod template of a Deployment—for example, by updating the image—the Deployment triggers the creation of a new ReplicaSet. The Deployment will gradually migrate the Pods from the old ReplicaSet to the new one. You can check the rollout history by running the following command. You will see two revisions listed:

```
$ kubectl rollout history deployment app-cache
deployment.apps/app-cache
REVISION  CHANGE-CAUSE
1          <none>
2          <none>
```

The first revision was recorded for the original state of the Deployment when you created the object. The second revision was added for changing the image tag.



By default, a Deployment persists for a maximum of 10 revisions in its history. You can change the limit by assigning a different value to `spec.revisionHistoryLimit`.

To get a more detailed view of the revision, run the following command. You can see that the image uses the value `memcached:1.6.10`:

```
$ kubectl rollout history deployments app-cache --revision=2
deployment.apps/app-cache with revision #2
Pod Template:
  Labels:      app=app-cache
               pod-template-hash=596bc5586d
  Containers:
    memcached:
      Image:      memcached:1.6.10
      Port:       <none>
      Host Port: <none>
      Environment: <none>
```

```
Mounts:      <none>
Volumes:     <none>
```

The rolling update strategy ensures that the application is always available to end users. This approach implies that two versions of the same application are available during the update process. As an application developer, you have to be aware that convenience doesn't come without potential side effects. If you happen to introduce a breaking change to the public API of your application, you might temporarily break consumers, as they could hit revision 1 or 2 of the application.

You can change the default update strategy of a Deployment by providing a different value to the attribute `spec.strategy.type`; however, consider the trade-offs. For example, the value `Recreate` kills all Pods first, then creates new Pods with the latest revision, causing potential downtime for consumers. See [Chapter 11](#) for a more detailed description of common deployment strategies.

Adding a Change Cause for a Revision

The rollout history renders the column `CHANGE-CAUSE`. You can populate the information for a revision to document *why* you introduced a new change or *which kubectl* command you use to make the change.

By default, changing the Pod template does not automatically record a change cause. To add a change cause to the current revision, add an annotation with the reserved key `kubernetes.io/change-cause` to the Deployment object. The following imperative `annotate` command assigns the change cause "Image updated to 1.6.10":

```
$ kubectl annotate deployment app-cache kubernetes.io/change-cause=\
"Image updated to 1.6.10"
deployment.apps/app-cache annotated
```

The rollout history now renders the change cause value for the current revision:

```
$ kubectl rollout history deployment app-cache
deployment.apps/app-cache
REVISION  CHANGE-CAUSE
1          <none>
2          Image updated to 1.6.10
```

Rolling Back to a Previous Revision

Problems can arise in production that require swift action. For example, the container image you just rolled out contains a crucial bug. Kubernetes gives you the option to roll back to one of the previous revisions in the rollout history. You can achieve this by using the `rollout undo` command. To pick a specific revision, provide the command-line option `--to-revision`. The command rolls back to the previous revision if you do not provide the option. Here, we are rolling back to revision 1:

```
$ kubectl rollout undo deployment app-cache --to-revision=1
deployment.apps/app-cache rolled back
```

As a result, Kubernetes performs a rolling update to all replicas with the revision 1.



Rollbacks and persistent data

The `rollout undo` command does not restore any persistent data associated with applications. Rather, it simply restores to a new instance of the previous declared state of the ReplicaSet.

The rollout history now lists revision 3. Given that we rolled back to revision 1, there's no more need to keep that entry as a duplicate. Kubernetes simply turns revision 1 into 3 and removes 1 from the list:

```
$ kubectl rollout history deployment app-cache
deployment.apps/app-cache
REVISION  CHANGE-CAUSE
2          Image updated to 1.16.10
3          <none>
```

Scaling Workloads

Scalability is one of Kubernetes' built-in capabilities. We'll learn how to manually scale the number of replicas as a reaction to increased application load. Furthermore, we'll talk about the API resource Horizontal Pod Autoscaler, which allows you to automatically scale the managed set of Pods based on resource thresholds such as CPU and memory.

Manually Scaling a Deployment

Scaling (up or down) the number of replicas controlled by a Deployment is a straightforward process. You can either manually edit the live object using the `edit deployment` command and change the value of the attribute `spec.replicas`, or you can use the imperative `scale deployment` command. In real-world production environments, you want to edit the Deployment YAML manifest, check it into version control, and apply the changes. The following command increases the number of replicas from four to six:

```
$ kubectl scale deployment app-cache --replicas=6
deployment.apps/app-cache scaled
```

You can observe the creation of replicas in real time using the `-w` command line flag. You'll see a change of status for the newly created Pods turning from `ContainerCreating` to `Running`:

\$ kubectl get pods -w					
NAME	READY	STATUS	RESTARTS	AGE	
app-cache-5d6748d8b9-6cc4j	1/1	ContainerCreating	0	11s	
app-cache-5d6748d8b9-6rmlj	1/1	Running	0	28m	
app-cache-5d6748d8b9-6z7g5	1/1	ContainerCreating	0	11s	
app-cache-5d6748d8b9-96dzf	1/1	Running	0	28m	
app-cache-5d6748d8b9-jkjsv	1/1	Running	0	28m	
app-cache-5d6748d8b9-svrxxw	1/1	Running	0	28m	

Manually scaling the number of replicas takes a bit of guesswork. You will still have to monitor the load on your system to see if your number of replicas is sufficient to handle the incoming traffic.

Autoscaling a Deployment

Another way to scale a Deployment is with the help of a Horizontal Pod Autoscaler (HPA). The HPA is an API primitive that defines rules for automatically scaling the number of replicas under certain conditions. Common scaling conditions include a target value, an average value, or an average utilization of a specific metric (e.g., for CPU and/or memory). Refer to the [MetricTarget API](#) for more information.

Let's say you want to define average CPU utilization of CPU as the scaling condition. At runtime, the HPA checks the metrics collected by the [metrics server](#) to determine if the average maximum CPU or memory usage across all replicas of a Deployment is less than or greater than the defined threshold. Make sure that you have the metrics server installed in the cluster. Collecting metrics may take a couple of minutes initially after installing the component. See [“Inspecting Resource Metrics”](#) on page 172 for more information.

Figure 10-3 shows an overview architecture diagram involving an HPA.

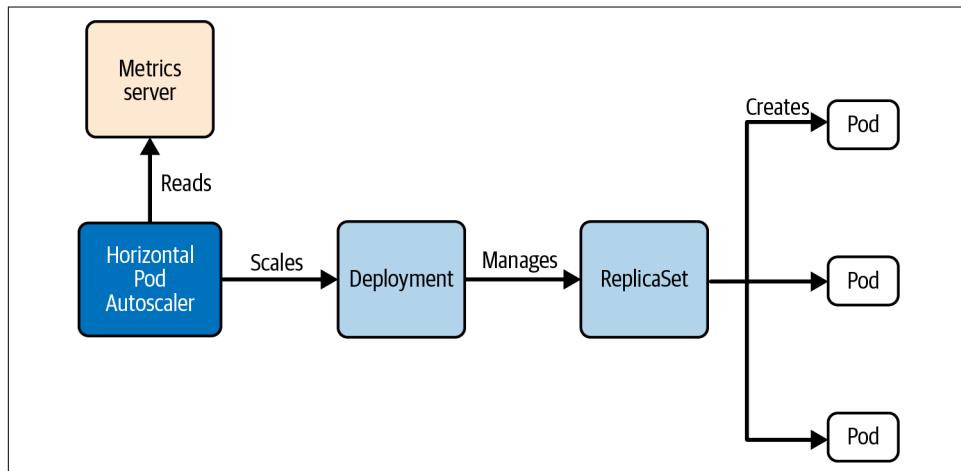


Figure 10-3. Autoscaling a Deployment

Creating Horizontal Pod AutoScalers

Figure 10-4 shows the use of an HPA that will scale up the number of replicas if an average of 80% CPU utilization is reached across all available Pods controlled by the Deployment.

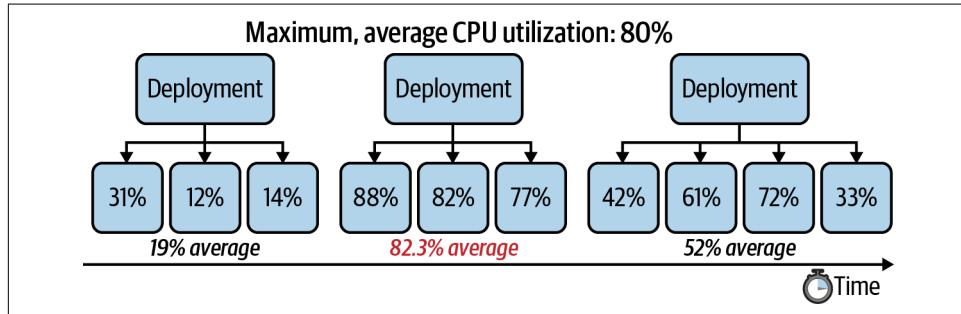


Figure 10-4. Autoscaling a Deployment horizontally

You can use the `autoscale deployment` command to create an HPA for an existing Deployment. The option `--cpu-percent` defines the average maximum CPU usage threshold. At the time of writing, the imperative command doesn't offer an option for defining the average maximum memory utilization threshold. The options `--min` and `--max` provide the minimum number of replicas to scale down to and the maximum number of replicas the HPA can create to handle the increased load, respectively:

```
$ kubectl autoscale deployment app-cache --cpu-percent=80 --min=3 --max=5
horizontalpodautoscaler.autoscaling/app-cache autoscaled
```

This command is a great shortcut for creating an HPA for a Deployment. The YAML manifest representation of the HPA object looks like Example 10-2.

Example 10-2. A YAML manifest for an HPA

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: app-cache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: app-cache
  minReplicas: 3
  maxReplicas: 5
  metrics:
  - resource:
      name: cpu
      target:
```

```
averageUtilization: 80
type: Utilization
type: Resource
```

Listing Horizontal Pod AutoScalers

The short-form command for a Horizontal Pod AutoScaler is `hpa`. Listing all of the HPA objects transparently describes their current state: the CPU utilization and the number of replicas at this time:

```
$ kubectl get hpa
NAME      REFERENCE          TARGETS      MINPODS   MAXPODS   REPLICAS  \
AGE
app-cache Deployment/app-cache <unknown>/80%    3          5          4          \
58s
```

If the Pod template of the Deployment does not define CPU resource requirements or if the CPU metrics cannot be retrieved from the metrics server, the left-side value of the column TARGETS says `<unknown>`. [Example 10-3](#) sets the resource requirements for the Pod template so that the HPA can work properly. You can learn more about defining resource requirements in “[Working with Resource Requirements](#)” on page [208](#).

Example 10-3. Setting CPU resource requirements for Pod template

```
# ...
spec:
# ...
template:
# ...
spec:
  containers:
  - name: memcached
# ...
  resources:
    requests:
      cpu: 250m
    limits:
      cpu: 500m
```

Once traffic hits the replicas, the current CPU usage is shown as a percentage. Here the average maximum CPU utilization is 15%:

```
$ kubectl get hpa
NAME      REFERENCE          TARGETS      MINPODS   MAXPODS   REPLICAS  AGE
app-cache Deployment/app-cache 15%/80%    3          5          4          58s
```

Rendering Horizontal Pod Autoscaler Details

The event log of an HPA can provide additional insight into the rescaling activities. Rendering the HPA details can be a great tool for overseeing when the number of replicas was scaled up or down, as well as their scaling conditions:

```
$ kubectl describe hpa app-cache
Name:                      app-cache
Namespace:                 default
Labels:                    <none>
Annotations:               <none>
CreationTimestamp:         Sun, 15 Aug 2021 \
                          15:54:11 -0600
Reference:                Deployment/app-cache
Metrics:
  resource cpu on pods  (as a percentage of request): 0% (1m) / 80%
Min replicas:              3
Max replicas:              5
Deployment pods:           3 current / 3 desired
Conditions:
  Type      Status  Reason            Message
  ----      -----  ----             -----
  AbleToScale  True    ReadyForNewScale recommended size matches current size
  ScalingActive  True    ValidMetricFound the HPA was able to successfully \
                           calculate a replica count from cpu resource utilization (percentage of request)
  ScalingLimited  True    TooFewReplicas   the desired replica count is less \
                           than the minimum replica count
Events:
  Type      Reason          Age     From            Message
  ----      ----           ----   ----            -----
  Normal   SuccessfulRescale 13m    horizontal-pod-autoscaler  New size: 3; \
  reason: All metrics below target
```

Defining Multiple Scaling Metrics

You can define more than a single resource type as a scaling metric. As you can see in [Example 10-4](#), we are inspecting CPU and memory utilization to determine if the replicas of a Deployment need to be scaled up or down.

Example 10-4. A YAML manifest for a HPA with multiple metrics

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: app-cache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: app-cache
```

```

minReplicas: 3
maxReplicas: 5
metrics:
- type: Resource
  resource:
    name: cpu
    target:
      type: Utilization
      averageUtilization: 80
- type: Resource
  resource:
    name: memory
    target:
      type: AverageValue
      averageValue: 500Mi

```

To ensure that the HPA determines the currently used resources, we'll set the memory resource requirements for the Pod template as well, as shown in [Example 10-5](#).

Example 10-5. Setting memory resource requirements for Pod template

```

...
spec:
...
template:
...
spec:
  containers:
  - name: memcached
  ...
  resources:
    requests:
      cpu: 250m
      memory: 100Mi
    limits:
      cpu: 500m
      memory: 500Mi

```

Listing the HPA renders both metrics in the TARGETS column, as in the output of the get command shown here:

```

$ kubectl get hpa
  NAME      REFERENCE          TARGETS          MINPODS   MAXPODS  \
  REPLICAS   AGE               1994752/500Mi, 0%/80%   3           5           \
app-cache   Deployment/app-cache   2m14s

```

Summary

The Deployment is an essential primitive for providing declarative updates and life cycle management of Pods. The ReplicaSet performs the heavy lifting of managing those Pods, commonly referred to as replicas. Application developers do not have to interact directly with the ReplicaSet; a Deployment manages the ReplicaSet under the hood.

Deployments can easily roll out and roll back revisions of the application represented by an image running in the container. In this chapter you learned about the commands for controlling the revision history and its operations. Scaling a Deployment manually requires deep insight into the requirements and the load of an application. A Horizontal Pod Autoscaler can automatically scale the number of replicas based on CPU and memory thresholds observed at runtime.

Exam Essentials

Know the ins and outs of a Deployment

Given that a Deployment is such a central primitive in Kubernetes, you can expect that the exam will test you on it. Know how to create a Deployment and learn how to scale to multiple replicas. One of the superior features of a Deployment is its rollout functionality for new revisions. Practice how to roll out a new revision, inspect the rollout history, and roll back to a previous revision.

Understand the implications of using a Horizontal Pod Autoscaler

The number of replicas controlled by a Deployment can be scaled up or down using the Horizontal Pod Autoscaler (HPA). An HPA defines thresholds for resources like CPU and memory that will tell the object that a scaling event needs to happen. It's important to understand that the HPA functions properly only if you install the metrics server component and define resource requests and limits for containers.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. Create a Deployment named `nginx` with 3 replicas. The Pods should use the `nginx:1.23.0` image and the name `nginx`. The Deployment uses the label `tier=backend`. The Pod template should use the label `app=v1`.

List the Deployment and ensure that the correct number of replicas is running.

Update the image to `nginx:1.23.4`.

Verify that the change has been rolled out to all replicas.

Assign the change cause “Pick up patch version” to the revision.

Scale the Deployment to 5 replicas.

Have a look at the Deployment rollout history. Revert the Deployment to revision 1.

Ensure that the Pods use the image `nginx:1.23.0`.

2. Create a Deployment named `nginx` with 1 replica. The Pod template of the Deployment should use container image `nginx:1.23.4`; set the CPU resource request to 0.5 and the memory resource request/limit to 500Mi.

Create a HorizontalPodAutoscaler for the Deployment named `nginx-hpa` that scales to a minimum of 3 and a maximum of 8 replicas. Scaling should happen based on an average CPU utilization of 75% and an average memory utilization of 60%.

Inspect the HorizontalPodAutoscaler object and identify the currently-utilized resources. How many replicas do you expect to exist?

Deployment Strategies

Deploying an application (bundled into a container image) to one or many Pods is only the beginning of its life cycle within a Kubernetes cluster. Periodically, developers will produce and publish new container image tags to ship bug fixes and new features. Manually updating Pods with a new container image tag one by one would be extremely tedious. Kubernetes offers the Deployment primitive to streamline the process.

Chapter 10 explained how to automatically roll out a new release using the Deployment primitive. In this chapter, we will discuss the built-in deployment strategies supported by the primitive. We'll also talk about other deployment strategies that require deliberate human decisions. Each deployment strategy is presented with an example featuring their benefits and potential trade-offs. More deployment strategies exist, but they will not be covered in this book.



Some deployment strategies require the use of concepts not yet discussed. Jump to Chapter 14 for coverage of container probes. Reference Chapter 21 for more information on the purpose of Services.

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objective:

- Use Kubernetes primitives to implement common deployment strategies (e.g., blue/green or canary)

Rolling Deployment Strategy

The Deployment primitive employs rolling deployment as the default deployment strategy, also referred to as ramped deployment. It's called "ramped" because the Deployment gradually transitions replicas from the old version to a new version in batches. The Deployment automatically creates a new ReplicaSet for the desired change after the user updates the Pod template.

Figure 11-1 shows a snapshot in time during the rollout process.

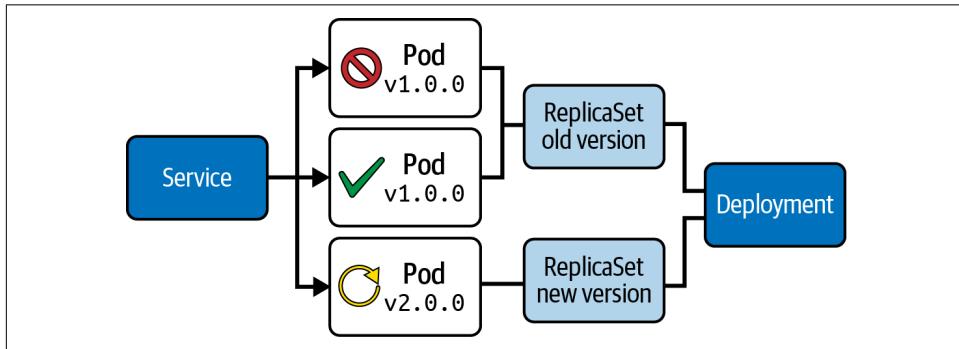


Figure 11-1. The rolling deployment strategy

In this scenario, the user initiated an update of the application version from 1.0.0 to 2.0.0. As a result, the Deployment creates a new ReplicaSet and starts up Pods running the new application version while at the same time scaling down the old version. The Service routes network traffic to either the old or new version of the application.

Implementation

A Deployment uses the rolling deployment strategy by default. The runtime value for the attribute `spec.strategy.type` is `RollingUpdate`. Users can fine-tune this strategy. You can use the attributes `spec.strategy.rollingUpdate.maxUnavailable` and `spec.strategy.rollingUpdate.maxSurge` to change the rollout rate. Both attributes can use a fixed integer (for example, 3) or assign a percentage of the total required number of Pods (for example, 33%). The default value for `maxUnavailable` and `maxSurge` is 25%.

The attribute `maxUnavailable` specifies the maximum number of Pods that can be unavailable during the update process. For example, if you set the value to 40%, then the old ReplicaSet can scale down to 60% immediately when the rolling update starts.

The attribute `maxSurge` specifies the maximum number of Pods that can be created over the desired number of Pods. For example, if you set the value to 10%, the total number of new and old Pods cannot exceed a total of 110% after the new ReplicaSet has been created.

Independent of the values assigned to the attributes `maxUnavailable` and `maxSurge`, all replicas controlled by the old ReplicaSet will be ramped down to 0 over time until all replicas controlled by the new ReplicaSet equals the value of `spec.replicas`.

It's recommended to define a readiness probe for the Pod template to ensure that a replica is ready to handle incoming requests. The attribute `spec.minReadySeconds` specifies the number of seconds a replica needs to be available for before it is made available to incoming requests.

Example 11-1 shows the usage of those attributes in the context of a full Deployment YAML manifest stored in the file `deployment-rolling-update.yaml`.

Example 11-1. A Deployment configured with a rolling update strategy

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  replicas: 4
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 40%    ①
      maxSurge: 10%          ②
      minReadySeconds: 60    ③
  selector:
    matchLabels:
      app: httpd
  template:
    metadata:
      labels:
        app: httpd
  spec:
    containers:
      - name: httpd
        image: httpd:2.4.23-alpine
        ports:
          - containerPort: 80
            protocol: TCP
        readinessProbe:      ④
          httpGet:
            path: /
            port: 80
```

- ① The percentage of Pods that can be unavailable during the update.
- ② The percentage of Pods that can temporarily exceed the total number of replicas.
- ③ The number of seconds for which the readiness probe in a Pod needs to be healthy until the rollout process can continue.
- ④ The readiness probe for all replicas referred to by `spec.minReadySeconds`.

The combination of assigned values to `maxUnavailable` and `maxSurge` determines the runtime behavior and speed of a rollout. You will adjust those parameters to find the most suitable combination for your application.

Use Cases and Trade-Offs

The rolling deployment is a fitting deployment strategy for rolling out a new application version with zero downtime. Depending on the number of replicas the Deployment manages, this process can be relatively slow, as old versions of the applications are ramped down and new versions of the application are ramped up in batches.

It's important to mention that this deployment strategy comes with a potential risk. Old and new versions of the application run in parallel. Breaking changes introduced with the new version can lead to unexpected and hard-to-debug errors for consumers if they haven't adapted their client software to the latest changes. It's a good idea to roll out a new application version in a backward-compatible fashion, for example by using a versioned API, to avoid running into this situation.

Fixed Deployment Strategy

The fixed deployment strategy will terminate replicas with the old application version at once before creating another ReplicaSet that controls replicas running the new application version.

[Figure 11-2](#) illustrates the rollout process while updating the Pod template from application version 1.0.0 to 2.0.0. All replicas of the old ReplicaSet are shut down simultaneously. Then, the replicas controlled by the new ReplicaSet are started. During this process, the Service may not be able to reach any of the replicas, which can lead to unnecessary downtime for consumers.

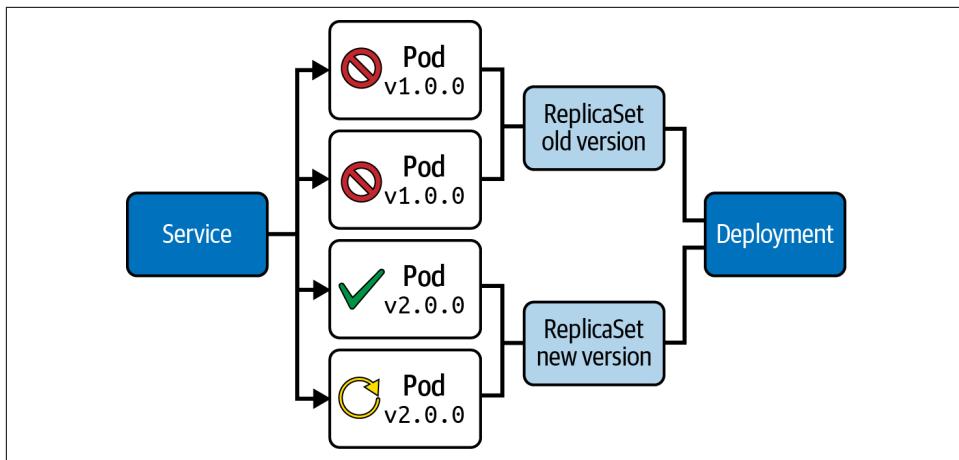


Figure 11-2. The fixed deployment strategy

Implementation

To configure the fixed deployment strategy for a Deployment, set the attribute `spec.strategy.type` to `Recreate`. Internally, this strategy type will automatically assign the total number of replicas to the attribute `maxUnavailable`. No other configuration options need to be provided.

[Example 11-2](#) shows the `Recreate` strategy type in the context of a full Deployment YAML manifest stored in the file `deployment-fixed.yaml`.

Example 11-2. A Deployment configured with a fixed deployment strategy

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  replicas: 4
  strategy:
    type: Recreate ①
  selector:
    matchLabels:
      app: httpd
  template:
    metadata:
      labels:
        app: httpd
  spec:
    containers:
      - name: httpd
        image: httpd:2.4.23-alpine

```

```
ports:
- containerPort: 80
protocol: TCP
```

- ① The strategy type for configuring fixed deployment.

Assigning a readiness probe to containers defined by the Pod template isn't strictly necessary because all replicas with the old application version will be shut down at once. Nevertheless, it still makes sense to verify that the application is up and running by defining a readiness probe before incoming traffic can reach the container.

Use Cases and Trade-Offs

The fixed deployment strategy is suitable for situations where application downtime is acceptable. For example, it's great if you want to roll out a new application to a developer environment for testing purposes. For production environments, this deployment strategy may work if you announce an outage time window to customers.

Blue-Green Deployment Strategy

The blue-green deployment strategy (sometimes referred to as red-black deployment strategy) figuratively uses blue as a representation of the old application version and green as a representation of the new application version. Both application versions will be operated at the same time with an equal number of replicas.

Kubernetes routes traffic to the blue deployment, while the development or test team rolls out and tests the green deployment. Traffic is switched over to the green deployment as soon as it is considered production-ready. At that point, the team managing the application can decommission the blue deployment.

Figure 11-3 shows two Deployments managing replicas with different application versions. The Services can switch network traffic from the old application version to the new application version by changing the label selection.

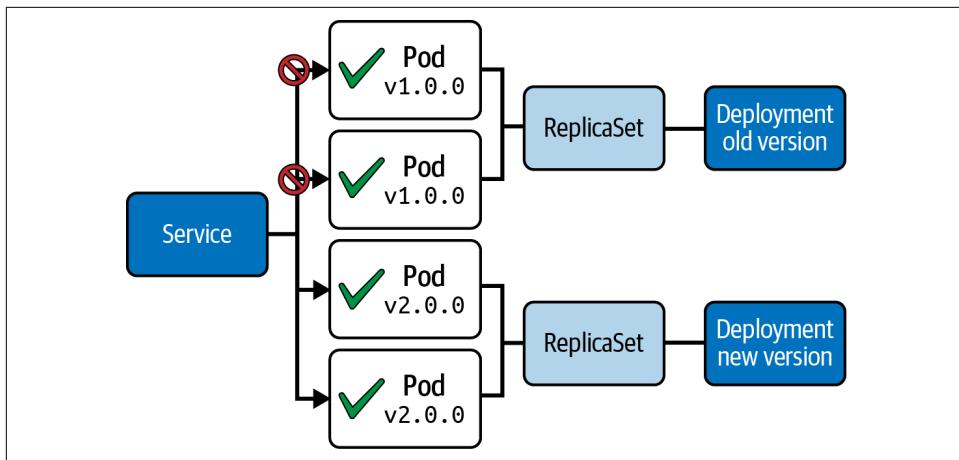


Figure 11-3. The blue-green deployment strategy

Implementation

Blue-green deployment is not a built-in strategy you can configure within the Deployment resource. You will have to create a Deployment object for both application versions. The Service routes traffic to replicas managed by either the blue or the green Deployment.

[Example 11-3](#) shows a blue Deployment YAML manifest stored in the file `deployment-blue.yaml` specifying the container image `httpd:2.4.23-alpine` in the Pod template.

Example 11-3. A blue Deployment

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server-blue
spec:
  replicas: 4
  selector:
    matchLabels:
      type: blue
  template:
    metadata:
      labels:
        type: blue
    spec:
      containers:
        - name: httpd
          image: httpd:2.4.23-alpine ②
          ports:

```

```
- containerPort: 80
  protocol: TCP
```

- ① Uses the label assignment type: `blue` to any replica managed by the corresponding ReplicaSet.
- ② The old application version `2.4.23-alpine`.

To set up a green deployment that runs the newer container image `httpd:2.4.57-alpine`, simply create another Deployment object. Note that the label used for the Pod template is different than for the blue deployment. [Example 11-4](#) shows the green Deployment definition in the file `deployment-green.yaml`.

Example 11-4. A green Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server-green
spec:
  replicas: 4
  selector:
    matchLabels:
      type: green
  template:
    metadata:
      labels:
        type: green
    spec:
      containers:
        - name: httpd
          image: httpd:2.4.57-alpine
          ports:
            - containerPort: 80
              protocol: TCP
```

- ① Uses the label assignment type: `green` to any replica managed by the corresponding ReplicaSet.
- ② The new application version `2.4.57-alpine`.

As mentioned earlier, the Service is the Kubernetes object responsible for routing network traffic to the old or new application version. [Example 11-5](#) shows a Service object.

Example 11-5. The Service routing network traffic to a blue deployment

```
apiVersion: v1
kind: Service
metadata:
  name: web-server
spec:
  selector:
    type: blue  ①
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

- ① The label selector pointing to replicas managed by the blue deployment.

The resource declaration currently points to the blue deployment; to switch to green, simply change the label selection from `type: blue` to `type: green`. At that point, you can delete the blue Deployment.

Use Cases and Trade-Offs

The blue-green deployment strategy is suitable for deployment scenarios where complex upgrades need to be performed without downtime to consumers. This situation may arise if a rollout requires a data migration or if multiple, dependent software components need to be changed at once. Should a rollback to the old application version be required, a simple change of the label selection in the Service will do.

On the downside, it's worth mentioning that you will need more hardware resources than for other deployment strategies. If you need five replicas to run the old application version, then you will need the same amount of resources for the new application version, assuming the resource requirements won't differ.

Canary Deployment Strategy

The canary deployment strategy is similar to the blue-green deployment; however, you'd make the new application version available to only a subset of consumers. With this approach, you can implement A/B testing of new features or if you need to gather metrics about consumer behavior. Based on the defined set of success criteria, traffic to the new application version can be increased gradually. The goal is to shut down the old application version completely.

Figure 11-4 shows how the Service sends traffic to both application versions.

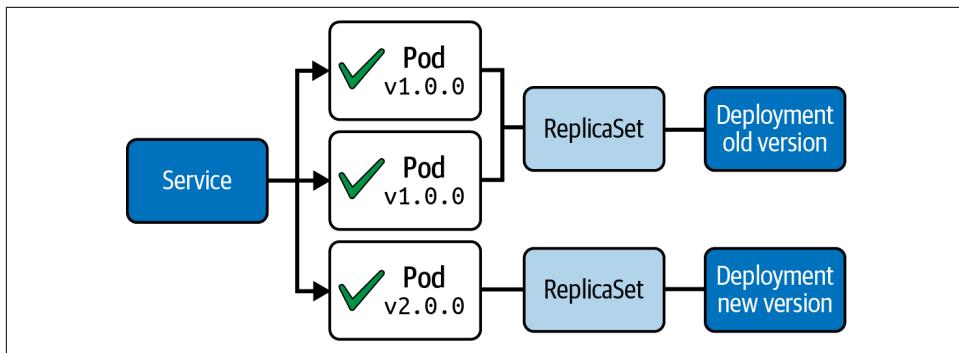


Figure 11-4. The canary deployment strategy

Deployment 1 controls application version v1.0.0. Deployment 2 controls application version v2.0.0. Deployments 1 and 2 use the same label assignment in their Pod template. The Service selects the label key-value pair(s) defined by both Deployments.

Implementation

In a Kubernetes cluster, you represent each application version with the help of a Deployment object. You want to roll out the new application version with fewer replicas than the current application version by assigning a smaller value to the attribute `spec.replicas`.

The following code snippet shows the truncated definition of the Deployment controlling the old application version:

```
kind: Deployment
spec:
  replicas: 4
  selector:
    matchLabels:
      app: httpd
```

For the new application version, assign a smaller number of replicas:

```
kind: Deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: httpd
```

To ensure that replicas for old and new application versions receive requests from consumers, the assigned Pod template label(s) in both Deployment objects need to be the same. Ensure that the Service selects those label(s), as shown in this truncated Service definition:

```
kind: Service
spec:
  selector:
    app: httpd ①
```

- ① Selects the label assigned to both Deployments.

Use Cases and Trade-Offs

Organizations typically use the canary deployment strategy to roll out experimental features or changes with potential impact on system performance. You can evaluate your success criteria while making the new feature available to only a subset of consumers. Implementing a canary deployment usually requires fewer hardware resources than the blue-green deployment as the number of replicas with the new application version is much lower.

Summary

A deployment is the process of making a software change available to end users or programs. You need to consider two aspects: the procedure of how to deploy a change and the routing of network traffic to the application. Select an appropriate deployment strategy based on use case, application type, and trade-offs.

With the Deployment primitive, Kubernetes natively supports two deployment strategies: the rolling deployment and the fixed deployment. The rolling deployment, specified by the `RollingUpdate` strategy, rolls out a change gradually in batches. The fixed deployment, configured by the `Recreate` strategy, first shuts down the old application version and then brings up the new application version.

The blue-green and canary deployment strategies can be set up by creating a second Deployment object that manages the new application version in parallel with the old one. The Service then routes network traffic to replicas of both application versions (blue-green) or transitions consumers to the new application version over time (canary).

Exam Essentials

Understand how to configure strategies native to the Deployment primitive

The exam may confront you with different deployment strategies. You need to understand how to implement the most common strategies and how to modify an existing deployment scenario. Learn how to configure the built-in strategies in the Deployment primitive and their options for fine-tuning the runtime behavior.

Practice multi-phased deployment strategies

You can implement even more sophisticated deployment scenarios with the help of the Deployment and Service primitives. Examples are the blue-green and canary deployment strategies, which require a multi-phased rollout process. Expose yourself to implementation techniques and rollout procedures. Operators provided by the Kubernetes community, e.g., [Argo Rollouts](#), offer higher-level abstractions for more sophisticated deployment strategies. The exam does not require you to understand external tooling to implement deployment strategies.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. One of your teammates created a Deployment YAML manifest to operate the container image `grafana/grafana:9.5.9`. Create the Deployment object from the YAML manifest file `deployment-grafana.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grafana
spec:
  replicas: 6
  selector:
    matchLabels:
      app: grafana
  template:
    metadata:
      labels:
        app: grafana
    spec:
      containers:
        - image: grafana/grafana:9.5.9
          name: grafana
          ports:
            - containerPort: 3000
```

You need to update all replicas with the container image `grafana/grafana:10.1.2`. Make sure that the rollout happens in batches of two replicas at a time. Ensure that a readiness probe is defined.

2. In this exercise, you will set up a blue-green Deployment scenario. You'll first create the initial (blue) Deployment and expose it with a Service. Later, you will create a second (green) Deployment and switch over traffic.

Create a Deployment named `nginx-blue` with 3 replicas. The Pod template of the Deployment should use container image `nginx:1.23.0` and assign the label `version=blue`.

Expose the Deployment with a Service of type ClusterIP named `nginx`. Map the incoming and outgoing port to 80. Select the Pod with label `version=blue`.

Run a temporary Pod with the container image `alpine/curl:3.14` to make a call against the Service using `curl`.

Create a second Deployment named `nginx-green` with 3 replicas. The Pod template of the Deployment should use container image `nginx:1.23.4` and assign the label `version=green`.

Change the Service's label selection so that traffic will be routed to the Pods controlled by the Deployment `nginx-green`.

Delete the Deployment named `nginx-blue`.

Run a temporary Pod with the container image `alpine/curl:3.14` to make a call against the Service.

Helm is a templating engine and package manager for a set of Kubernetes manifests. At runtime, it replaces placeholders in YAML template files with actual, end-user-defined values. The artifact produced by the Helm executable is a so-called *chart file* bundling the manifests that comprise the API resources of an application. You can upload the chart file to a *chart repository* so that other teams can use it to deploy the bundled manifests. The Helm ecosystem offers a wide range of reusable charts for common use cases searchable on [Artifact Hub](#) (for example, for running Grafana or PostgreSQL).

Due to the wealth of functionality available to Helm, we'll discuss only the basics. The exam does not expect you to be a Helm expert; rather, it wants you to be familiar with the workflow of installing existing packages with Helm. Building and publishing your own charts is outside the scope of the exam. For more detailed information on Helm, see the [user documentation](#). The version of Helm used to describe the functionality here is 3.13.0.

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objective:

- Use the Helm package manager to deploy existing packages

Managing an Existing Chart

As a developer, you want to reuse existing functionality instead of putting in the work to define and configure it yourself. For example, you may want to install the open source monitoring service Prometheus on your cluster.

Prometheus requires the installation of multiple Kubernetes primitives. Thankfully, the Kubernetes community provided a Helm chart making it very easy to install and configure all the moving parts in the form of a [Kubernetes operator](#).

The following list shows the typical workflow for consuming and managing a Helm chart. Most of those steps need to use the `helm` executable:

1. Identifying the chart you'd like to install
2. Adding the repository containing the chart
3. Installing the chart from the repository
4. Verifying the Kubernetes objects that have been installed by the chart
5. Rendering the list of installed charts
6. Upgrading an installed chart
7. Uninstalling a chart if its functionality is no longer needed

The following sections will explain each of the steps.

Identifying a Chart

Over the years, the Kubernetes community implemented and published thousands of Helm charts. Artifact Hub provides a web-based search capability for discovering charts by keyword.

Say you wanted to find a chart that installs the Continuous Integration solution Jenkins. All you'd need to do is to enter the term “jenkins” into the search box and press the enter key. [Figure 12-1](#) shows the list of results in Artifact Hub.

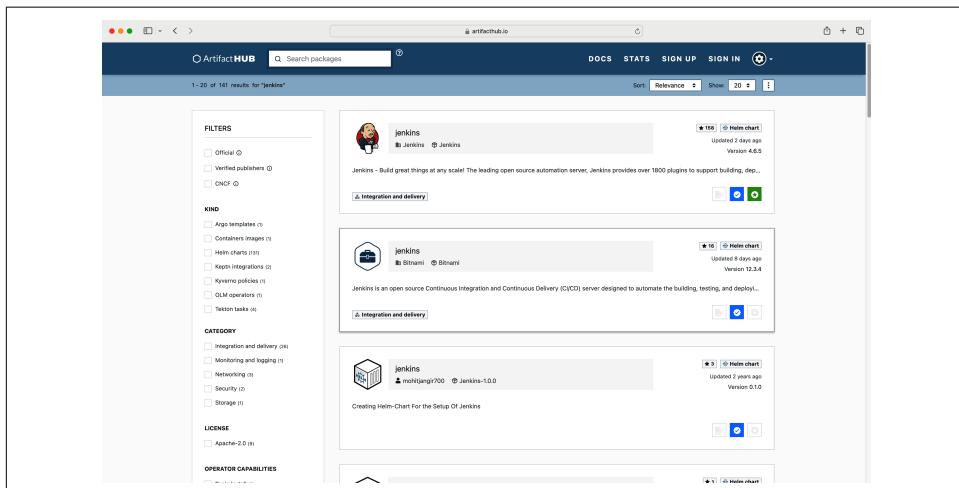


Figure 12-1. Searching for a Jenkins chart on Artifact Hub

At the time of writing, there are 141 matches for the search term. You will be able to inspect details about the chart by clicking on one of the search results, which includes a high-level description and the repository that the chart file resides in. Moreover, you can inspect the templates bundled with the chart file, indicating the objects that will be created upon installation and their configuration options. [Figure 12-2](#) shows the page for the official Jenkins chart.

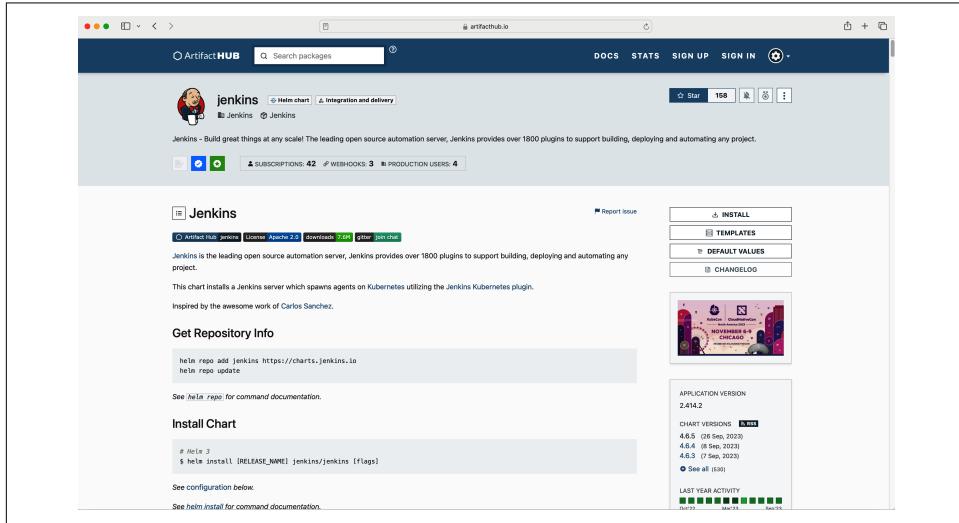


Figure 12-2. Jenkins chart details

You cannot install a chart directly from Artifact Hub. You must install it from the repository hosting the chart file.

Adding a Chart Repository

The chart description may mention the repository that hosts the chart file. Alternatively, you can click on the “Install” button to render repository details and the command for adding it. [Figure 12-3](#) shows the contextual pop-up that appears after clicking the “Install” button.

By default, a Helm installation defines no external repositories. The following command shows how to list all registered repositories. No repositories have been registered yet:

```
$ helm repo list  
Error: no repositories to show
```

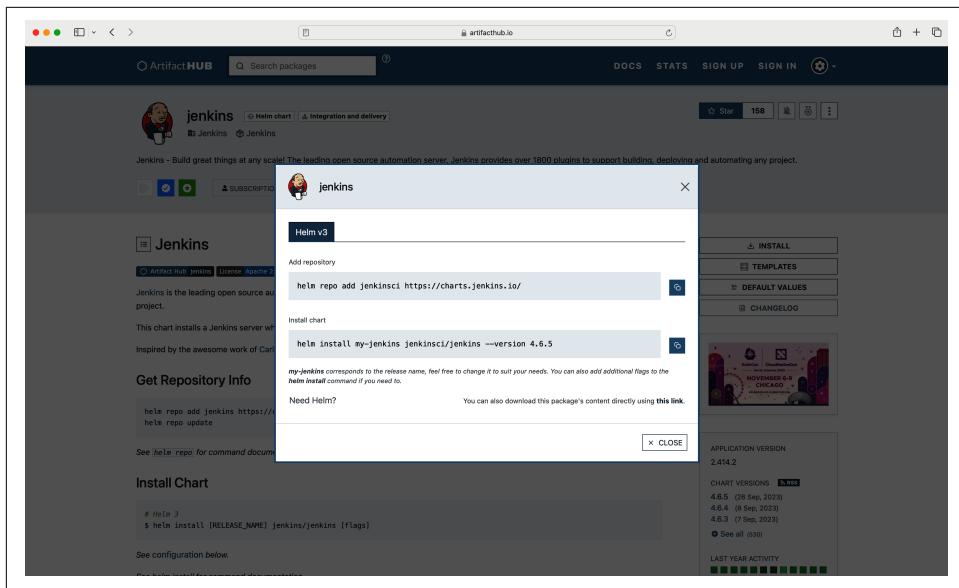


Figure 12-3. Jenkins chart installation instructions

As you can see from the screenshot, the chart file lives in the repository with the URL <https://charts.jenkins.io>. We will need to add this repository. This is an one-time operation. You can install other charts from that repository or you can update a chart that originated from that repository with commands we'll discuss in a later section.

You need to provide a name for the repository when registering one. Make the repository name as descriptive as possible. The following command registers the repository with the name `jenkinsci`:

```
$ helm repo add jenkinsci https://charts.jenkins.io/
"jenkinsci" has been added to your repositories
```

Listing the repositories now shows the mapping between name and URL:

```
$ helm repo list
NAME          URL
jenkinsci    https://charts.jenkins.io/
```

You permanently added the repository to the Helm installation.

Searching for a Chart in a Repository

The “Install” pop-up window already provided the command to install the chart. You can also search the repository for available charts in case you do not know their names or latest versions. Add the `--versions` flag to list all available versions:

```
$ helm search repo jenkinsci
NAME          CHART VERSION   APP VERSION   DESCRIPTION
jenkinsci/jenkins  4.6.5        2.414.2      ...
```

The latest version available is 4.6.5. This may be different if you run the command on your machine, given that the Jenkins project may have released a newer version.

Installing a Chart

Let’s assume that the latest version of the Helm chart contains a security vulnerability. Therefore, we decide to install the Jenkins chart with the previous version, 4.6.4. You need to assign a name to be able to identify an installed chart. The name we’ll use here is `my-jenkins`:

```
$ helm install my-jenkins jenkinsci/jenkins --version 4.6.4
NAME: my-jenkins
LAST DEPLOYED: Thu Sep 28 09:47:21 2023
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
...
```

The chart automatically created the Kubernetes objects in the `default` namespace. You can use the following command to discover the most important resource types:

```
$ kubectl get all
NAME           READY   STATUS    RESTARTS   AGE
pod/my-jenkins-0  2/2     Running   0          12m

NAME                  TYPE        CLUSTER-IP      EXTERNAL-IP   ...
service/my-jenkins    ClusterIP   10.99.166.189  <none>       ...
service/my-jenkins-agent ClusterIP  10.110.246.141  <none>       ...

NAME           READY   AGE
statefulset.apps/my-jenkins  1/1     12m
```

The chart has been installed with the default configuration options. You can inspect those default values by clicking on the “Default Values” button on the chart page, as shown in [Figure 12-4](#).

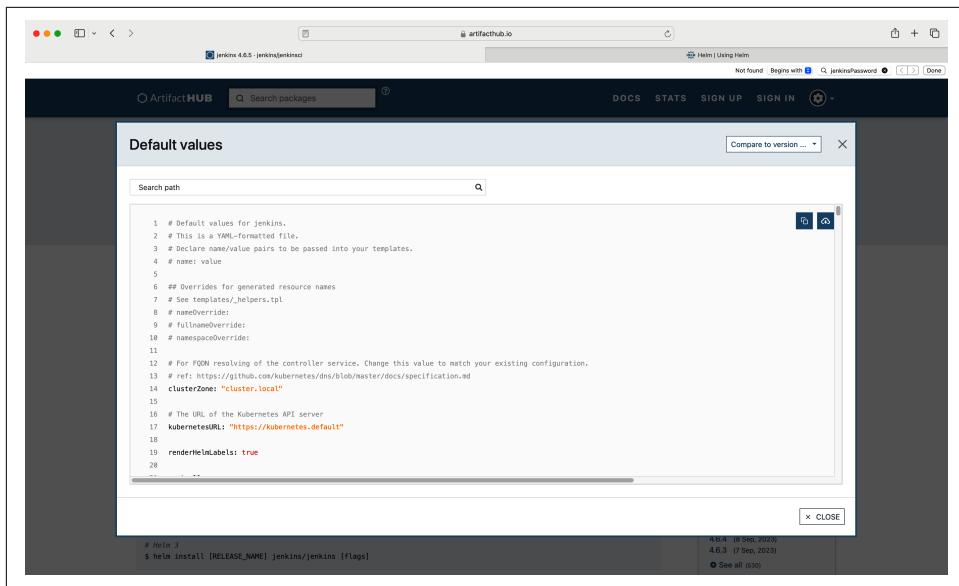


Figure 12-4. Jenkins chart default values

You can also discover those configuration options using the following command. The output shown renders only a subset of values, the admin username and its password, represented by `controller.adminUser` and `controller.adminPassword`:

```
$ helm show values jenkinsci/jenkins
...
controller:
  # When enabling LDAP or another non-Jenkins identity source, the built-in \
  # admin account will no longer exist.
  # If you disable the non-Jenkins identity store and instead use the Jenkins \
  # internal one,
  # you should revert controller.adminUser to your preferred admin user:
  adminUser: "admin"
  # adminPassword: <defaults to random>
...
```

You can customize any configuration value when installing the chart. To pass configuration data during the install processing use one of the following flags:

- `--values`: Specifies the overrides in the form of a pointer to a YAML manifest file.
- `--set`: Specifies the overrides directly from the command line.

For more information, see “[Customizing the Chart Before Installing](#)” in the Helm documentation.

You can decide to install the chart into a custom namespace. Use the `-n` flag to provide the name of an existing namespace. Add the flag `--create-namespace` to automatically create the namespace if it doesn't exist yet.

The following command shows how to customize some of the values and the namespace used during the installation process:

```
$ helm install my-jenkins jenkinsci/jenkins --version 4.6.4 \
--set controller.adminUser=boss --set controller.adminPassword=password \
-n jenkins --create-namespace
```

We specifically set the username and the password for the admin user. Helm created the objects controlled by the chart into the `jenkins` namespace.

Listing Installed Charts

Charts can live in the `default` namespace or a custom namespace. You can inspect the list of installed charts using the `helm list` command. If you do not know which namespace, simply add the `--all-namespaces` flag to the command:

```
$ helm list --all-namespaces
NAME      NAMESPACE  REVISION  UPDATED        STATUS    CHART
my-jenkins  default     1          2023-09-28...  deployed  jenkins-4.6.4
```

The output of the command includes the column `NAMESPACE` that shows the namespace used by a particular chart. Similar to the use of `kubectl`, the `helm list` command provides the option `-n` for spelling out a namespace. Providing no flag(s) with the command will return the result for the `default` namespace.

Upgrading an Installed Chart

Upgrading an installed chart usually means moving to a new chart version. You can poll for new versions available in the repository by running this command:

```
$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "jenkinsci" chart repository
Update Complete. *Happy Helming!*
```

What if you want to upgrade your existing chart installation to a newer chart version? Run the following command to upgrade the chart to that specific version with the default configuration:

```
$ helm upgrade my-jenkins jenkinsci/jenkins --version 4.6.5
Release "my-jenkins" has been upgraded. Happy Helming!
...
```

As with the `install` command, you will have to provide custom configuration values if you want to tweak the chart's runtime behavior when upgrading a chart.

Uninstalling a Chart

Sometimes you no longer need to run a chart. The command for uninstalling a chart is straightforward, as shown here. It will delete all objects controlled by the chart. Don't forget to provide the `-n` flag if you previously installed the chart into a namespace other than `default`:

```
$ helm uninstall my-jenkins
release "my-jenkins" uninstalled
```

Executing the command may take up to 30 seconds, as Kubernetes needs to wait for the workload grace period to end.

Summary

Helm has evolved to become a de facto tool for deploying application stacks to Kubernetes. The artifact that contains the manifest files, default configuration values, and metadata is called a chart. A team or an individual can publish charts to a chart repository. Users can discover a published chart through the Artifact Hub user interface and install it to a Kubernetes cluster.

One of the primary developer workflows when using Helm consists of finding, installing, and upgrading a chart with a specific version. You start by registering the repository containing chart files you want to consume. The `helm install` command downloads the chart file and stores it in a local cache. It also creates the Kubernetes objects described by the chart.

The installation process is configurable. A developer can provide overrides for customizable configuration values. The `helm upgrade` command lets you upgrade the version of an already installed chart. To uninstall a chart and delete all Kubernetes objects managed by the chart, run the `helm uninstall` command.

Exam Essentials

Assume that the Helm executable is preinstalled

Unfortunately, the [exam FAQ](#) does not mention any details about the Helm executable or the Helm version to expect. It's fair to assume that it will be preinstalled for you and therefore you do not need to memorize installation instructions. You will be able to browse the [Helm documentation pages](#).

Become familiar with Artifact Hub

Artifact Hub provides a web-based UI for Helm charts. It's worthwhile to explore the search capabilities and the details provided by individual charts, more specifically the repository the chart file lives in, and its configurable values. During the exam, you'll likely not be asked to navigate to Artifact Hub because its URL hasn't

been listed as one of the permitted documentation pages. You can assume that the exam question will provide you with the repository URL.

Practice commands needed to consume existing Helm charts

The exam does not ask you to build and publish your own chart file. All you need to understand is how to consume an existing chart. You will need to be familiar with the `helm repo add` command to register a repository, the `helm search repo` to find available chart versions, and the `helm install` command to install a chart. You should have a basic understanding of the upgrade process for an already installed Helm chart using the `helm upgrade` command.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. In this exercise, you will use Helm to install Kubernetes objects needed for the open source monitoring solution **Prometheus**. The easiest way to install Prometheus on top of Kubernetes is with the help of the **prometheus-operator** Helm chart.

You can search for the **kube-prometheus-stack** on Artifact Hub. Add the repository to the list of known repositories accessible by Helm with the name **prometheus-community**.

Update to the latest information about charts from the respective chart repository.

Run the Helm command for listing available Helm charts and their versions. Identify the latest chart version for **kube-prometheus-stack**.

Install the chart **kube-prometheus-stack**. List the installed Helm chart.

List the Service named **prometheus-operated** created by the Helm chart. The object resides in the **default** namespace.

Use the `kubectl port-forward` command to forward the local port 8080 to the port 9090 of the Service. Open a browser and bring up the Prometheus dashboard.

Stop port forwarding and uninstall the Helm chart.

PART IV

Application Observability and Maintenance

Application observability and maintenance entails concepts and techniques for maintaining, monitoring, and troubleshooting applications operated in a Kubernetes cluster.

The following chapters cover these concept:

- [Chapter 13](#) walks you through scenarios that may arise when using deprecated APIs to define objects. The chapter explains the measures you need to take to ensure operability of those objects with future Kubernetes versions.
- [Chapter 14](#) discusses the probes you can define for containers to automatically monitor applications to detect potentially problematic runtime issues.
- [Chapter 15](#) explains troubleshooting techniques for Pods and containers. You will be able to use these techniques to identify the root cause of application runtime issues and learn how to fix them.

API Deprecations

The Kubernetes project periodically releases new versions. Every release adds new features and bug fixes but may also introduce deprecations to existing APIs. An API is the interface that application developers interact with when defining Kubernetes objects.

Deprecations may come into effect if the Kubernetes team plans to change, replace, or completely remove support for an API. You need to understand how to handle API deprecations to avoid issues before updating nodes to a newer Kubernetes version.

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objective:

- Understand API deprecations

Understanding the Deprecation Policy

The Kubernetes project releases [three versions per calendar year](#). Optimally, the administrator of a Kubernetes cluster upgrades to the latest version as early as possible to incorporate enhancements and security fixes. However, upgrading a cluster doesn't come without potential cost and risk. You need to ensure that existing objects running in the cluster will still be compatible with the version you are upgrading to.

A Kubernetes release can deprecate an API, which means that it is scheduled for removal or replacement. The rules for introducing a deprecation follow the [deprecation policy](#) explained in the Kubernetes documentation.

The value you assign to the `version` attribute in a manifest specifies the API version. The use of a deprecated API renders a warning message when creating or updating the object. While you can still create or modify the object, a warning message informs the user about the action to take to ensure its future compatibility with newer Kubernetes versions. The [deprecated API migration guide](#) shows a list of deprecated APIs and the scheduled versions that will remove support for the API.

Listing Available API Versions

Kubectl provides a command for discovering available API versions. The `api-versions` command lists all API versions in the format group/version:

```
$ kubectl api-versions
admissionregistration.k8s.io/v1
apiextensions.k8s.io/v1
apiregistration.k8s.io/v1
apps/v1
authentication.k8s.io/v1
authorization.k8s.io/v1
autoscaling/v1
autoscaling/v2
batch/v1
certificates.k8s.io/v1
coordination.k8s.io/v1
discovery.k8s.io/v1
events.k8s.io/v1
flowcontrol.apiserver.k8s.io/v1beta2
flowcontrol.apiserver.k8s.io/v1beta3
networking.k8s.io/v1
node.k8s.io/v1
policy/v1
rbac.authorization.k8s.io/v1
scheduling.k8s.io/v1
storage.k8s.io/v1
v1
```

Certain APIs, like the API for the group `autoscaling`, exist with different versions, `v1` and `v2`. Generally speaking, you can make your manifests more future-proof by choosing a higher major version. At the time of writing, the deprecation status of any of those APIs is not included in the output of the `api-versions` command. You will need to look up the status in the Kubernetes documentation.

Handling Deprecation Warnings

Let's demonstrate the effects of using a deprecated API. The following example assumes that you are running a Kubernetes cluster with a version between 1.23 and

1.25. [Example 13-1](#) shows a manifest for a Horizontal Pod Autoscaler that uses a beta API version for the group `autoscaling`.

Example 13-1. A Horizontal Pod Autoscaler definition using a deprecated API

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
```

Creating a new object from the YAML manifest will not result in an error. Kubernetes will happily create the object, but the command will let you know what will happen with the API in a future Kubernetes version. As shown in the following output, it is suggested that you replace the use of the API with `autoscaling/v2`:

```
$ kubectl create -f hpa.yaml
Warning: autoscaling/v2beta2 HorizontalPodAutoscaler is deprecated in v1.23+, \
unavailable in v1.26+; use autoscaling/v2 HorizontalPodAutoscaler
```

In addition to the warning message, you may also want to look at the deprecated API migration guide. An easy way to find information about the deprecated API is to search the deprecation guide for it. For example, you will find the [following passage](#) about the API `autoscaling/v2beta2`:

The `autoscaling/v2beta2` API version of `HorizontalPodAutoscaler` is no longer served as of v1.26.

- Migrate manifests and API clients to use the `autoscaling/v2` API version, available since v1.23.
- All existing persisted objects are accessible via the new API

—Deprecated API Migration Guide

All you need to do to future-proof your manifest is to assign the new API version.

Handling a Removed or Replaced API

The administrator of a Kubernetes cluster may decide to jump up multiple minor versions at once when upgrading. It is possible that you won't catch that an API you are currently using has already been removed. It's important to verify the compatibility of

existing Kubernetes objects before upgrading the production cluster to avoid any disruptions.

Say you have been using the definition of a ClusterRole shown in [Example 13-2](#). Managing the object worked fine with Kubernetes 1.8; however, the administrator upgraded the cluster nodes all the way to 1.22.

Example 13-2. A ClusterRole using the API version rbac.authorization.k8s.io/v1beta1

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

Trying to create the object from the ClusterRole manifest file will not render a deprecation message. Instead, kubectl will return an error message. Consequently, the command did not create the object:

```
$ kubectl apply -f clusterrole.yaml
error: resource mapping not found for name: "pod-reader" namespace: "" from \
"clusterrole.yaml": no matches for kind "ClusterRole" in version \
"rbac.authorization.k8s.io/v1beta1"
```

You will not have a clear idea why the command failed just looking at the error message. It's a good idea to check with the deprecated API migration guide. Searching the page for the API version `rbac.authorization.k8s.io/v1beta1` will give you the [following information](#). The solution here is to assign the replacement API `rbac.authorization.k8s.io/v1` instead:

The `rbac.authorization.k8s.io/v1beta1` API version of ClusterRole, ClusterRoleBinding, Role, and RoleBinding is no longer served as of v1.22.

- Migrate manifests and API clients to use the `rbac.authorization.k8s.io/v1` API version, available since v1.8.
- All existing persisted objects are accessible via the new APIs
- No notable changes

—Deprecated API Migration Guide

Under certain conditions, a Kubernetes primitive may not provide a replacement API. As a representative use case I want to mention the PodSecurityPolicy primitive here. The feature has been replaced by a new Kubernetes-internal concept, the [Pod Security Admission](#). You should follow the release notes of upcoming Kubernetes releases to stay aware of more radical changes.

Summary

Inevitably, you will run into API deprecations when using Kubernetes long term. As a developer, you need to know how to interpret deprecation warning messages. The Kubernetes documentation provides all the information you need to identify an alternative API or feature. It's advisable to test all available YAML manifests for objects currently in use in production clusters before upgrading nodes to a newer Kubernetes version.

Exam Essentials

Keep the API deprecation documentation handy

The exam will likely expose you to an object that uses a deprecated API. You will need to keep the [Deprecated API Migration Guide](#) documentation page handy to tackle such scenarios. The page describes deprecated, removed, and replaced APIs categorized by Kubernetes version. Use the browser's search capability to quickly find relevant information about an API. The quick reference links on the right side of the page let you quickly navigate to a specific Kubernetes version.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. The Kubernetes administrator in charge of the cluster is planning to upgrade all nodes from Kubernetes 1.8 to 1.28. You defined multiple Kubernetes YAML manifests that operate an application stack. The administrator provided you with a Kubernetes 1.28 test environment. Make sure that all YAML manifests are compatible with Kubernetes version 1.28.

Navigate to the directory `app-a/ch13/deprecated` of the checked-out GitHub repository [bmuschko/ckad-study-guide](#). Inspect the files `deployment.yaml` and `configmap.yaml` in the current directory.

Create the objects from the YAML manifests. Modify the definitions as needed.

Verify that the objects can be instantiated with Kubernetes 1.28.

Container Probes

Applications running in containers do not operate under the premise of “fire and forget.” Once Kubernetes starts the container, you’ll want to know if the application is ready for consumption and is still working as expected in an hour, a week, or a month. A health probe is a periodically running mini-process that asks the application for its status and takes action upon certain conditions.

In this chapter, we’ll discuss container health probes—more specifically, readiness, liveness, and startup probes. You’ll learn about the different health verification methods and how to define them for the proper use cases.

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objective:

- Implement probes and health checks

Working with Probes

Even with the best testing strategy, it’s nearly impossible to find all bugs before deploying software to a production environment. That’s especially true for failure situations that occur only after operating the software for an extended period of time. It’s not uncommon to see memory leaks, deadlocks, infinite loops, and similar conditions crop up once end users put the application under load.

Proper monitoring can help with identifying those issues; however, you still need to act to mitigate the situation. First, you’ll likely want to restart the application to prevent further outages. Second, the development team needs to identify the underlying root cause and fix the application’s code.

Probe Types

Kubernetes provides a concept called *health probing* to automate the detection and correction of such issues. You can configure a container to execute a periodic mini-process that checks for certain conditions. These processes are defined as follows:

Readiness probe

Even after an application has started up, it may still need to execute configuration procedures—for example, connecting to a database and preparing data. This probe checks if the application is ready to serve incoming requests. [Figure 14-1](#) shows the readiness probe.

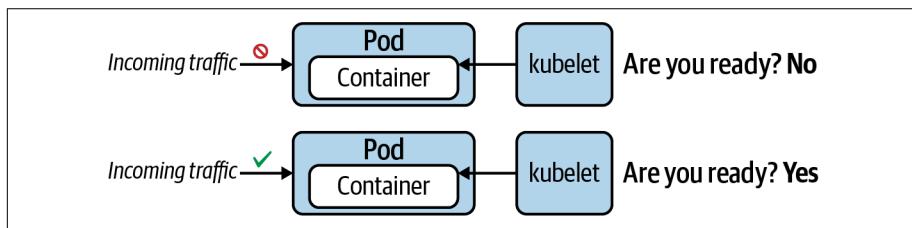


Figure 14-1. A readiness probe checks if the application is ready to accept traffic

Liveness probe

Once the application is running, you want to make sure that it still works as expected without issues. This probe periodically checks for the application's responsiveness. Kubernetes restarts the container automatically if the probe considers the application be in an unhealthy state, as shown in [Figure 14-2](#).

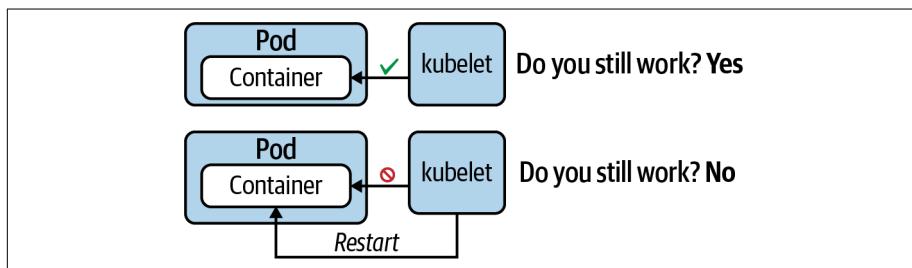


Figure 14-2. A liveness probe checks if the application is healthy

Startup probe

Legacy applications in particular can take a long time to start up—possibly several minutes. A startup probe can be instantiated to wait for a predefined amount of time before a liveness probe is allowed to start probing. By setting up a startup probe, you can prevent the application process from being overwhelmed with probing requests. Startup probes kill the container if the application can't start within the set time frame. [Figure 14-3](#) illustrates the behavior of a startup probe.

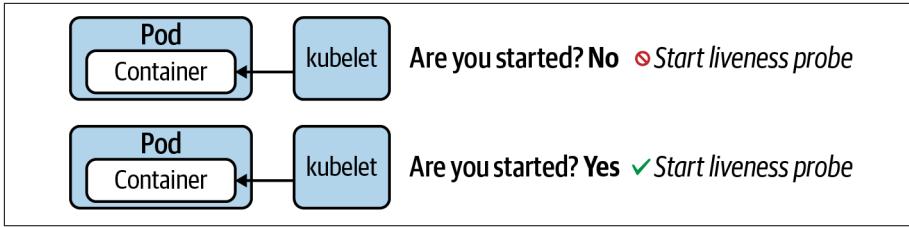


Figure 14-3. A startup probe holds off on starting the liveness probe

From an operational perspective, the most important probe to implement is the readiness probe. Without defining liveness and startup probes, the Kubernetes control plane components will handle the majority of the default behavior.

Each probe offers distinct methods to verify the health of a container, discussed in the next section.

Health Verification Methods

You can define one or many of the health verification methods for a container. [Table 14-1](#) describes the available health verification methods, their corresponding YAML attribute, and their runtime behavior.

Table 14-1. Available health verification methods

Method	Option	Description
Custom command	<code>exec.command</code>	Executes a command inside the container (e.g., a <code>cat</code> command) and checks its exit code. Kubernetes considers a zero exit code to be successful. A non-zero exit code indicates an error.
HTTP GET request	<code>httpGet</code>	Sends an HTTP GET request to an endpoint exposed by the application. An HTTP response code in the range of 200 to 399 indicates success. Any other response code is regarded as an error.
TCP socket connection	<code>tcpSocket</code>	Tries to open a TCP socket connection to a port. If the connection could be established, the probing attempt was successful. The inability to connect is accounted for as an error.
gRPC	<code>grpc</code>	The application implements the GRPC Health Checking Protocol , which verifies whether the server is able to handle a Remote Procedure Call (RPC).

Remember that you can combine any probe with any health check method. The health verification method you choose highly depends on the type of application you are running in the container. For example, the obvious choice for a web-based application is to use the HTTP GET request verification method.

Health Check Attributes

Every probe offers a set of attributes that can further configure the runtime behavior, as shown in [Table 14-2](#). For more information, see the API of the [Probe v1 core object](#).

Table 14-2. Attributes for fine-tuning the health check runtime behavior

Attribute	Default value	Description
initialDelaySeconds	0	Delay in seconds until first check is executed.
periodSeconds	10	Interval for executing a check (e.g., every 20 seconds).
timeoutSeconds	1	Maximum number of seconds until check operation times out.
successThreshold	1	Number of successful check attempts until probe is considered successful after a failure.
failureThreshold	3	Number of failures for check attempts before probe is marked failed and action taken.
terminationGracePeriodSeconds	30	Grace period before forcing a container to stop upon failure.

The following sections will demonstrate the usage of most verification methods for different probe types.

The Readiness Probe

In this scenario, we'll want to define a readiness probe for a Node.js application. The Node.js application exposes an HTTP endpoint on the root context path and runs on port 3000. Dealing with a web-based application makes an HTTP GET request a perfect fit for probing its readiness. You can find the source code of the application in the book's GitHub repository.

In the YAML manifest shown in [Example 14-1](#), the readiness probe executes its first check after two seconds and repeats checking every eight seconds thereafter. All other attributes use the default values. A readiness probe will continue to periodically check, even after the application has been successfully started.

Example 14-1. A readiness probe that uses an HTTP GET request

```
apiVersion: v1
kind: Pod
metadata:
  name: readiness-pod
spec:
  containers:
    - image: bmuschko/nodejs-hello-world:1.0.0
```

```

name: hello-world
ports:
- name: nodejs-port      ①
  containerPort: 3000
readinessProbe:
  httpGet:
    path: /
    port: nodejs-port ②
  initialDelaySeconds: 2
  periodSeconds: 8

```

- ① You can assign a name to a port so that it can be referenced in a probe.
- ② Instead of assigning port 3000 again, we simply use the port name.

Create a Pod by pointing the `apply` command to the YAML manifest. During the Pod's startup process, it's possible that the status shows `Running` but the container isn't ready to accept incoming requests, as indicated by `0/1` in the `READY` column:

```

$ kubectl apply -f readiness-probe.yaml
pod/readiness-pod created
$ kubectl get pod readiness-pod
NAME          READY   STATUS    RESTARTS   AGE
pod/readiness-pod  0/1    Running   0          6s
$ kubectl get pod readiness-pod
NAME          READY   STATUS    RESTARTS   AGE
pod/readiness-pod  1/1    Running   0          68s
$ kubectl describe pod readiness-pod
...
Containers:
  hello-world:
    ...
      Readiness:      http-get http://:nodejs-port/ delay=2s timeout=1s \
                      period=8s #success=1 #failure=3
    ...

```

The Liveness Probe

A liveness probe checks if the application is still working as expected. To demonstrate a liveness probe, we'll use a custom command. A custom command is the most flexible way to verify the health of a container, as it allows for calling any command available to the container. This can be either a command-line tool that comes with the base image or a tool that you install as part of the containerization process.

In [Example 14-2](#), we'll have the application create and update a file, `/tmp/heartbeat.txt`, to show that it's still alive. We'll do this by making it run the Unix `touch` command every five seconds. The probe will periodically check if the modification timestamp of the file is older than one minute. If it is, then Kubernetes can assume that the application isn't functioning as expected and will restart the container.

Example 14-2. A liveness probe that uses a custom command

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-pod
spec:
  containers:
    - image: busybox:1.36.1
      name: app
      args:
        - /bin/sh
        - -c
        - 'while true; do touch /tmp/heartbeat.txt; sleep 5; done;'
  livenessProbe:
    exec:
      command:
        - test `find /tmp/heartbeat.txt -mmin -1` > /dev/null
    initialDelaySeconds: 5
    periodSeconds: 30
```

The next command uses the YAML manifest shown in [Example 14-2](#), stored in the file *liveness-probe.yaml*, to create the Pod. Describing the Pod renders information on the liveness probe. We can not only inspect the custom command and its configuration, but also see how many times the container has been restarted upon a probing failure:

```
$ kubectl apply -f liveness-probe.yaml
pod/liveness-pod created
$ kubectl get pod liveness-pod
NAME          READY   STATUS    RESTARTS   AGE
pod/liveness-pod  1/1     Running   0          22s
$ kubectl describe pod liveness-pod
...
Containers:
  app:
    ...
    Restart Count:  0
    Liveness:      exec [test `find /tmp/heartbeat.txt -mmin -1`] delay=5s \
                    timeout=1s period=30s #success=1 #failure=3
...
...
```

The Startup Probe

The purpose of a startup probe is to figure out when an application is fully started. Defining the probe is useful for an application that takes a long time to start up. The kubelet puts the readiness and liveness probes on hold while the startup probe is running. A startup probe finishes its operation under one of the following conditions:

1. If it can verify that the application has been started
2. If the application doesn't respond within the timeout period

To demonstrate the functionality of the startup probe, [Example 14-3](#) defines a Pod that runs the [Apache HTTP server](#) in a container. By default, the image exposes the container port 80, and that's what we're probing for using a TCP socket connection.

Example 14-3. A startup probe that uses a TCP socket connection

```
apiVersion: v1
kind: Pod
metadata:
  name: startup-pod
spec:
  containers:
    - image: httpd:2.4.46
      name: http-server
      startupProbe:
        tcpSocket:
          port: 80
        initialDelaySeconds: 3
        periodSeconds: 15
      livenessProbe:
        ...

```

As you can see in the following terminal output, the `describe` command can retrieve the configuration of a startup probe as well:

```
$ kubectl apply -f startup-probe.yaml
pod/startup-pod created
$ kubectl get pod startup-pod
NAME           READY   STATUS    RESTARTS   AGE
pod/startup-pod 1/1     Running   0          31s
$ kubectl describe pod startup-pod
...
Containers:
  http-server:
    ...
    Startup:      tcp-socket :80 delay=3s timeout=1s period=15s \
                  #success=1 #failure=3
    ...

```

Summary

In this chapter, we looked at all available health probe types you can define for a Pod. A health probe is a periodically running mini-process that asks the application running in a container for its status. Think of it as taking the pulse of your system.

The readiness probe ensures that the container accepts incoming traffic only if the application runs properly. The liveness probe makes sure that the application is functioning as expected and will restart the container if necessary. The startup probe pauses a liveness probe until application startup has been completed. In practice, you'll often find that a container defines all three probes.

Exam Essentials

Understand the purpose of all probes

To prepare for this section of the exam, focus on understanding and using health probes. You should understand the purpose of startup, readiness, and liveness probes and practice how to configure them. In your Kubernetes cluster, try to emulate success and failure conditions to see the effects of probes and the actions they take.

Practice the use of different verification methods

You can choose from a variety of verification methods applicable to probes. Gain a high-level understanding when to apply which verification method, and how to configure each one of them.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. Define a new Pod named `web-server` with the image `nginx:1.23.0` in a YAML manifest. Expose the container port 80. Do not create the Pod yet.

For the container, declare a startup probe of type `httpGet`. Verify that the kubelet can make a request to the root context endpoint. Use the default configuration for the probe.

For the container, declare a readiness probe of type `httpGet`. Verify that the kubelet can make a request to the root context endpoint. Wait five seconds before checking for the first time.

For the container, declare a liveness probe of type `httpGet`. Verify that the kubelet can make a request to the root context endpoint. Wait 10 seconds before checking for the first time. The probe should run the check every 30 seconds.

Create the Pod and follow the life cycle phases of the Pod during the process.

Inspect the runtime details of the Pod's probes.

Troubleshooting Pods and Containers

When operating an application in a production Kubernetes cluster, failures are almost inevitable. You can't completely leave this job up to the Kubernetes administrator—it's your responsibility as an application developer to be able to troubleshoot issues for the Kubernetes objects you designed and deployed.

In this chapter, we'll look at troubleshooting strategies that can help with identifying the root cause of an issue so that you can take action and correct the failure appropriately. The strategies discussed here start with the high-level perspective of a Kubernetes object and then drill into more detail as needed.

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objectives:

- Use provided tools to monitor Kubernetes applications
- Utilize container logs
- Debugging in Kubernetes

Troubleshooting Pods

In most cases, creating a Pod is no issue. You simply emit the `run`, `create`, or `apply` commands to instantiate the Pod. If the YAML manifest is formed properly, Kubernetes accepts your request, so the assumption is that everything works as expected. To verify the correct behavior, the first thing you'll want to do is to check the Pod's high-level runtime information. The operation could involve other Kubernetes objects like a Deployment responsible for rolling out multiple replicas of a Pod.

Retrieving High-Level Information

To retrieve the information, run either the `kubectl get pods` command for just the Pods running in the namespace or the `kubectl get all` command to retrieve the most prominent object types in the namespace (which includes Deployments). You will want to look at the columns READY, STATUS, and RESTARTS. In the optimal case, the number of ready containers matches the number of containers you defined in the spec. For a single-container Pod, the READY column would say 1/1.

The status should say Running to indicate that the Pod entered the proper life cycle state. Be aware that it's totally possible that a Pod renders a Running state, but the application isn't actually working properly. If the number of restarts is greater than 0, then you might want to check the logic of the liveness probe (if defined) and identify the reason a restart was necessary.

The following Pod observes the status `ErrImagePull` and makes 0/1 containers available to incoming traffic. In short, this Pod has a problem:

```
$ kubectl get pods
NAME           READY   STATUS      RESTARTS   AGE
misbehaving-pod  0/1     ErrImagePull   0          2s
```

After working with Kubernetes for a while, you'll automatically recognize common error conditions. [Table 15-1](#) lists some of those error statuses and explains how to fix them.

Table 15-1. Common Pod error statuses

Status	Root cause	Potential fix
ImagePullBackOff or <code>ErrImagePull</code>	Image could not be pulled from registry.	Check correct image name, check that image name exists in registry, verify network access from node to registry, ensure proper authentication.
<code>CrashLoopBackOff</code>	Application or command run in container crashes.	Check command executed in container, ensure that image can properly execute (e.g., by creating a container with Docker).
<code>CreateContainerConfigError</code>	ConfigMap or Secret referenced by container cannot be found.	Check correct name of the configuration object, verify the existence of the configuration object in the namespace.

Inspecting Events

You might not encounter any of those error statuses. But there's still a chance of the Pod having a configuration issue. You can retrieve detailed information about the Pod using the `kubectl describe pod` command to inspect its events.

The following output belongs to a Pod that tries to mount a Secret that doesn't exist. Instead of rendering a specific error message, the Pod gets stuck with the status `ContainerCreating`:

```
$ kubectl get pods
NAME      READY   STATUS        RESTARTS   AGE
secret-pod 0/1    ContainerCreating  0          4m57s
$ kubectl describe pod secret-pod
...
Events:
Type  Reason     Age           From           Message
----  ----       --           --            --
Normal  Scheduled <unknown>   default-scheduler  Successfully assigned
                                         default/secret-pod to minikube
Warning FailedMount 3m15s      kubelet, minikube  Unable to attach or \
                                         mount volumes: \
                                         unmounted \
                                         volumes=[mysecret], \
                                         unattached volumes= \
                                         [default-token-bf8rh \
                                         mysecret]: timed out \
                                         waiting for the \
                                         condition
Warning FailedMount 68s (x10 over 5m18s)  kubelet, minikube  MountVolume.SetUp \
                                         failed for volume \
                                         "mysecret" : secret \
                                         "mysecret" not found
Warning FailedMount 61s      kubelet, minikube  Unable to attach or \
                                         mount volumes: \
                                         unmounted volumes= \
                                         [mysecret], \
                                         unattached \
                                         volumes=[mysecret \
                                         default-token-bf8rh \
                                         ]: timed out \
                                         waiting for the \
                                         condition
```

Another helpful command is `kubectl get events`. The output of the command lists the events across all Pods for a given namespace. You can use additional command-line options to further filter and sort events:

```
$ kubectl get events
LAST SEEN    TYPE      REASON          OBJECT                MESSAGE
3m14s        Warning   BackOff         pod/custom-cmd   Back-off \
                           restarting \
                           failed container
2s           Warning   FailedNeedsStart cronjob/google-ping Cannot determine \
                           if job needs to \
                           be started: too \
                           many missed start \
                           time (> 100). Set \
                           or decrease \
                           .spec. \
                           startingDeadline \
                           Seconds or check \
                           clock skew
```

Sometimes troubleshooting won't be enough. You may have to dig into the application runtime behavior and configuration in the container.

Using Port Forwarding

In production environments, you'll operate an application in multiple Pods controlled by a ReplicaSet. It's not unusual that one of those replicas experiences a runtime issue. Instead of troubleshooting the problematic Pod from a temporary Pod from within the cluster, you can also forward the traffic to a Pod through a tunneled HTTP connection. This is where the **port-forward command** comes into play.

Let's demonstrate the behavior. The following command creates a new Deployment running nginx in three replicas:

```
$ kubectl create deployment nginx --image=nginx:1.24.0 --replicas=3 --port=80
deployment.apps/nginx created
```

The resulting Pod will have unique names derived from the name of the Deployment. Say the Pod `nginx-595dff4799-ph4js` has an issue you want to troubleshoot:

```
$ kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
nginx-595dff4799-pfgdg  1/1     Running   0          6m25s
nginx-595dff4799-ph4js  1/1     Running   0          6m25s
nginx-595dff4799-s76s8  1/1     Running   0          6m25s
```

The **port-forward** command forwards HTTP connections from a local port to a port exposed by a Pod. This exemplary command forwards port 2500 on your local machine to the container port 80 running in the Pod `nginx-595dff4799-ph4js`:

```
$ kubectl port-forward nginx-595dff4799-ph4js 2500:80
Forwarding from 127.0.0.1:2500 -> 80
Forwarding from [::1]:2500 -> 80
```

The `port-forward` command does not return. You have to open another terminal to perform calls to the Pod via port forwarding. The following command simply checks if the Pod is accessible from your local machine using `curl`:

```
curl -Is localhost:2500 | head -n 1
HTTP/1.1 200 OK
```

The HTTP response code 200 clearly shows that we can access the Pod from outside of the cluster. The `port-forward` command is not meant to run for a long time. Its primary purpose is for testing or troubleshooting a Pod without having to expose it with the help of a Service.

Troubleshooting Containers

You can interact with the container for a deep-dive into the application's runtime environment. The next sections will discuss how to inspect logs, open an interactive shell to a container, and debug containers that do not provide a shell.



The commands described in the following sections apply to init and sidecar containers as well. Use the `-c` or `--container` command line flag to target a specific container if you are running more than a single one. See [Chapter 8](#) for more information on multi-container Pods.

Inspecting Logs

When troubleshooting a Pod, you can retrieve the next level of details by downloading and inspecting its logs. You may or may not find additional information that points to the root cause of a misbehaving Pod, but it's definitely worth a look. The YAML manifest shown in [Example 15-1](#) defines a Pod running a shell command.

Example 15-1. A Pod running a failing shell command

```
apiVersion: v1
kind: Pod
metadata:
  name: incorrect-cmd-pod
spec:
  containers:
    - name: test-container
      image: busybox:1.36.1
      command: ["/bin/sh", "-c", "unknown"]
```

After creating the object, the Pod fails with the status `CrashLoopBackOff`. Running the `logs` command reveals that the command run in the container has an issue:

```
$ kubectl create -f crash-loop-backoff.yaml
pod/incorrect-cmd-pod created
$ kubectl get pods incorrect-cmd-pod
NAME           READY   STATUS      RESTARTS   AGE
incorrect-cmd-pod  0/1    CrashLoopBackOff  5          3m20s
$ kubectl logs incorrect-cmd-pod
/bin/sh: unknown: not found
```

The `logs` command provides two helpful options. The option `-f` streams the logs, meaning you'll see new log entries as they're being produced in real time. The option `--previous` gets the logs from the previous instantiation of a container, which is helpful if the container has been restarted.

Opening an Interactive Shell

If any of the previous commands don't point you to the root cause of the failing Pod, it's time to open an interactive shell to a container. As an application developer, you'll know best what behavior to expect from the application at runtime. Inspect the running processes by using the Unix or Windows utility tools, depending on the image run in the container.

Say you encounter a situation where a Pod seems to work properly on the surface, as shown in [Example 15-2](#).

Example 15-2. A Pod periodically writing the current date to a file

```
apiVersion: v1
kind: Pod
metadata:
  name: failing-pod
spec:
  containers:
    - args:
        - /bin/sh
        - -c
        - while true; do echo $(date) >> ~/tmp/curr-date.txt; sleep \
          5; done;
      image: busybox:1.36.1
      name: failing-pod
```

After creating the Pod, you check the status. It says `Running`; however, when making a request to the application, the endpoint reports an error. Next, you check the logs. The log output renders an error message that points to a nonexistent directory. Apparently, the directory that the application needs hasn't been set up correctly:

```
$ kubectl create -f failing-pod.yaml
pod/failing-pod created
$ kubectl get pods failing-pod
NAME      READY   STATUS    RESTARTS   AGE
failing-pod 1/1     Running   0          5s
$ kubectl logs failing-pod
/bin/sh: can't create /root/tmp/curr-date.txt: nonexistent directory
```

The `exec` command opens an interactive shell to further investigate the issue. In the following code, we're using the Unix tools `mkdir`, `cd`, and `ls` inside of the running container to fix the problem. Obviously, the better mitigation strategy is to create the directory from the application or provide an instruction in the Dockerfile:

```
$ kubectl exec failing-pod -it -- /bin/sh
# mkdir -p ~/tmp
# cd ~/tmp
# ls -l
total 4
-rw-r--r-- 1 root root 112 May 9 23:52 curr-date.txt
```

Interacting with a Distroless Container

Some images run in containers are designed to be very minimal for security reasons. For example, the [Google distroless](#) images don't have any Unix utility tools preinstalled. You can't even open a shell to a container, as it doesn't come with a shell.



Incorporating security best practices for container images

Shipping container images with accessible shells and running with the `root` user is commonly discouraged as these aspects can be used as potential attack vectors. Check out the [CKS certification](#) to learn more about security concerns in Kubernetes.

One of Google's distroless images is `k8s.gcr.io/pause:3.1`, shown in [Example 15-3](#).

Example 15-3. Running a distroless image

```
apiVersion: v1
kind: Pod
metadata:
  name: minimal-pod
spec:
  containers:
    - image: k8s.gcr.io/pause:3.1
      name: pause
```

As you can see in the following exec command, the image doesn't provide a shell:

```
$ kubectl create -f minimal-pod.yaml
pod/minimal-pod created
$ kubectl get pods minimal-pod
NAME        READY   STATUS    RESTARTS   AGE
minimal-pod  1/1     Running   0          8s
$ kubectl exec minimal-pod -it -- /bin/sh
OCI runtime exec failed: exec failed: container_linux.go:349: starting \
container process caused "exec: \"/bin/sh\": stat /bin/sh: no such file \
or directory": unknown
command terminated with exit code 126
```

Kubernetes offers the concept of **ephemeral containers**. Those containers are meant to be disposable and have no resilience features like probes. You can deploy an ephemeral container for troubleshooting minimal containers that would usually not allow the use of the exec command.

Kubernetes 1.18 introduced a new debug command that can inject an ephemeral container to a running Pod for debugging purposes. The following command adds the ephemeral container running the image busybox to the Pod named minimal-pod and opens an interactive shell for it:

```
$ kubectl alpha debug -it minimal-pod --image=busybox
Defaulting debug container name to debugger-jf98g.
If you don't see a command prompt, try pressing enter.
/ # pwd
/
/ # exit
Session ended, resume using 'kubectl alpha attach minimal-pod -c \
debugger-jf98g -i -t' command when the pod is running
```

Inspecting Resource Metrics

Deploying software to a Kubernetes cluster is only the start of operating an application long term. Developers need to understand their applications' resource consumption patterns and behaviors, with the goal of providing a scalable and reliable service.

In the Kubernetes world, monitoring tools like **Prometheus** and **Datadog** help with collecting, processing, and visualizing information over time. The exam does not expect you to be familiar with third-party monitoring, logging, tracing, and aggregation tools; however, it is helpful to have a basic understanding of the underlying Kubernetes infrastructure responsible for collecting usage metrics. The following are examples of typical metrics:

- Number of nodes in the cluster
- Health status of nodes

- Node performance metrics such as CPU, memory, disk space, network
- Pod-level performance metrics such as CPU and memory consumption

This responsibility falls to the [metrics server](#), a cluster-wide aggregator of resource usage data. As shown in [Figure 15-1](#), kubelets running on nodes collect metrics and send them to the metrics server.

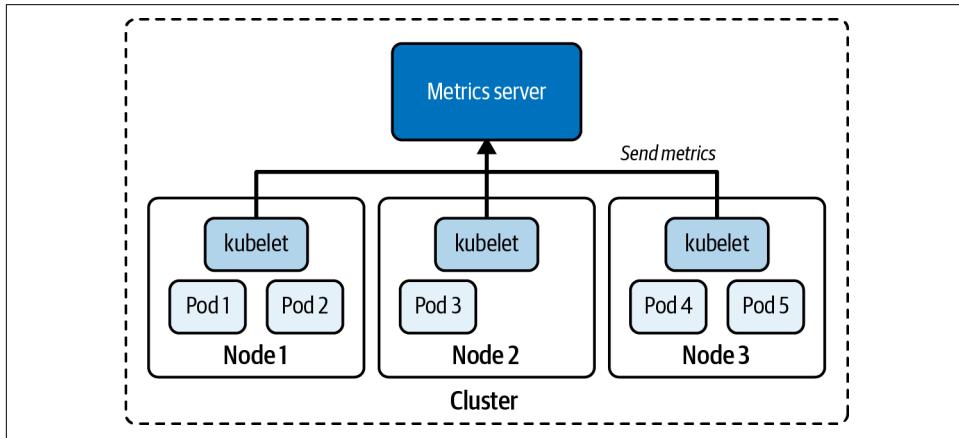


Figure 15-1. Data collection for the metrics server

The metrics server stores data in memory and does not persist data over time. If you are looking for a solution that keeps historical data, then you need to look into commercial options. Refer to the documentation for more information on its [installation process](#).

If you're using Minikube as your practice environment, [enabling the metrics-server add-on](#) is straightforward using the following command:

```
$ minikube addons enable metrics-server
The 'metrics-server' addon is enabled
```

You can now query for metrics of cluster nodes and Pods with the `kubectl top` command:

```
$ kubectl top nodes
NAME      CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
minikube  283m        14%    1262Mi          32%
$ kubectl top pod frontend
NAME      CPU(cores)   MEMORY(bytes)
frontend  0m           2Mi
```

It takes a couple of minutes after the installation of the metrics server before it has gathered information about resource consumption. Rerun the `kubectl top` command if you receive an error message.

Summary

We discussed strategies for approaching failed or misbehaving Pods. The main goal is to diagnose the root cause of a failure and then fix it by taking the right action. Troubleshooting Pods doesn't have to be hard. With the right `kubectl` commands in your tool belt, you can rule out root causes one by one to get a clearer picture.

The Kubernetes ecosystem provides a lot of options for collecting and processing metrics of your cluster over time. Among those options are commercial monitoring tools like Prometheus and Datadog. Many of those tools use the metrics server as the source of truth for those metrics. We also briefly touched on the installation process and the `kubectl top` command for retrieving metrics from the command line.

Exam Essentials

Know how to debug Pod objects

In this chapter, we mainly focused on troubleshooting problematic Pods and containers. Practice all relevant `kubectl` commands that can help with diagnosing issues. Refer to the [Kubernetes documentation](#) to learn more about debugging other Kubernetes resource types.

Learn how to retrieve and interpret resource metrics

Monitoring a Kubernetes cluster is an important aspect of successfully operating in a real-world environment. You should read up on commercial monitoring products and which data the metrics server can collect. You can assume that the exam environment provides you with an installation of the metrics server. Learn how to use the `kubectl top` command to render Pod and node resource metrics and how to interpret them.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. In this exercise, you will practice your troubleshooting skills by inspecting a misconfigured Pod. Navigate to the directory `app-a/ch15/troubleshooting` of the checked-out GitHub repository [bmuschko/ckad-study-guide](#).

Create a new Pod from the YAML manifest in the file `pod.yaml`. Check the Pod's status. Do you see any issue?

Render the logs of the running container and identify an issue. Shell into the container. Can you verify the issue based on the rendered log message?

Suggest solutions that can fix the root cause of the issue.

2. You will inspect the metrics collected by the metrics server. Navigate to the directory `app-a/ch15/stress-test` of the checked-out GitHub repository [bmuschko/ckad-study-guide](#). The current directory contains the YAML manifests for three Pods, `stress-1-pod.yaml`, `stress-2-pod.yaml`, and `stress-3-pod.yaml`. Inspect those manifest files.

Create the namespace `stress-test` and the Pods inside of the namespace.

Use the data available through the metrics server to identify which of the Pods consumes the most memory.

Prerequisite: You will need to install the metrics server if you want to be able to inspect actual resource metrics. You can find [installation instructions](#) on the project's GitHub page.

PART V

Application Environment, Configuration, and Security

The primary focus of the domain *Application Environment, Configuration, and Security* is configuring an application with security settings, defining and injecting configuration data, and specifying resource requirements. Two other aspects relevant to this domain are extending the Kubernetes API with custom resources, and the inner workings of processing requests to the Kubernetes API.

The following chapters cover these concepts:

- [Chapter 16](#) touches on extending the Kubernetes API by introducing your own primitives. You will learn how to define, inspect, and create a CustomResourceDefinition (CRD) as a schema for instantiating objects to implement custom requirements not natively covered by Kubernetes.
- [Chapter 17](#) describes the three phases that spring into action whenever a client like `kubectl` makes a call to the API server. More specifically, we'll talk about the authentication phase and the authorization phase (including controlling permissions with RBAC). The chapter will also take you on a short excursion into the admission phase.
- [Chapter 18](#) is all about resource management relevant to application developers. You will learn about container resource requirements and their impact on Pod scheduling and runtime behavior. This chapter will also cover enforcing aggregate resource consumption of objects living in a specific namespace with the help

of resource quotas. Finally, we'll dive into governing resource consumption for specific resource types with limit ranges.

- [Chapter 19](#) shows how to centrally define configuration data with ConfigMaps and Secrets and the different ways to consume the configuration data from a Pod.
- [Chapter 20](#) discusses the security context concept, a way to define security settings for containers.

CustomResourceDefinitions (CRDs)

Kubernetes provides primitives that support the most common use cases required by operators of an application stack. For custom use cases, Kubernetes allows for implementing and installing extensions to the platform.

A CustomResourceDefinition (CRD) is a Kubernetes extension mechanism for introducing custom API primitives to fulfill requirements not covered by built-in primitives. This chapter will primarily focus on the implementation of and interaction with CRDs.

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objective:

- Discover and use resources that extend Kubernetes (CRD)

Working with CRDs

CRDs can be understood as the schema that defines the blueprint for a custom object, and then the instantiation of those objects with the newly introduced type. For a CRD to be useful, it has to be backed by a *controller*. Controllers interact with the Kubernetes API and implement the reconciliation logic that interacts with CRD objects. The combination of CRDs and controllers is commonly referred to as the *operator pattern*. The exam does not require you to have an understanding of controllers; therefore, their implementation won't be covered in this chapter. [Figure 16-1](#) shows the operator patterns with all its moving parts.

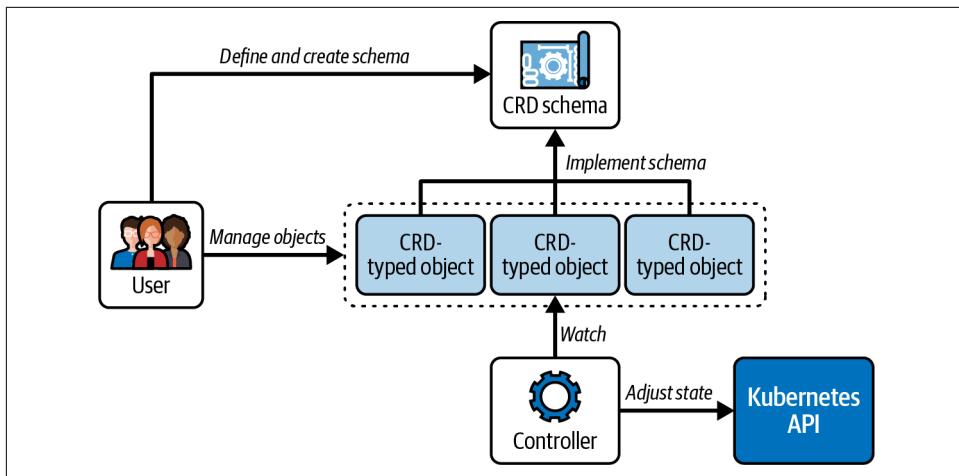


Figure 16-1. The Kubernetes operator pattern

The Kubernetes community has implemented many useful operators discoverable on [OperatorHub.io](#). You can install many of those operators with a single `kubectl`. A prominent operator is the [External Secrets Operator](#) that helps with integrating external Secret managers, like AWS Secrets Manager and HashiCorp Vault, with Kubernetes. For the exam, you will need to understand how to discover CRD schemas provided by external operators and how to interact with objects that follow the CRD schema. We'll go a little further in this chapter and talk about creating your own CRD schemas.

Example CRD

The implementation of and interaction with a CRD is best explained by example. We will implement and instantiate a CRD that represents a smoke test for a Service object to be executed after an application stack has been deployed to Kubernetes.

Technical baseline

Assume you are in charge of managing a web-based application. The Kubernetes objects needed to manage the application consist of a Deployment for running an application in Pods and the Service object that routes the network traffic to the replicas.

Desired functionality

A smoke test should be triggered automatically after deploying the Kubernetes objects responsible for operating the application. At runtime, the smoke test executes HTTPS calls to an endpoint of the application by targeting the Service's DNS name. The result of the smoke test (in the case of success or failure) will be

sent to an external service so it can be rendered as charts and graphs in a dashboard.

Goal

To implement this functionality, you could decide to write a CRD and controller. In this chapter, we are only going to cover the CRD for a smoke test, not the controller that would do the heavy lifting of executing the smoke test.

Implementing a CRD Schema

To make a CRD operational, you have to create two objects: the custom resource schema and the custom resource object. The custom resource schema specifies the CRD blueprint in the form of an OpenAPI v3 specification. The custom resource object follows the specification of the CRD schema and assigns values to properties available to the schema.



Using Kubebuilder to generate CRDs

A CRD schema specification can become quite extensive. To avoid writing those specification by hand, take a look at the [Kubebuilder](#) project. One Kubebuilder functionality is the ability to generate CRD schemas with minimal input provided by the user. Using the tool can help save time getting a Kubernetes operator project started.

First, let's look at the custom resource schema of the smoke test. The schema uses the kind CustomResourceDefinition, as shown in [Example 16-1](#).

While I can't explain the meaning of every attribute of a CustomResourceDefinition here, I want to point out some important aspects. The CustomResourceDefinition specifies the group, version, and names of the custom primitive. It also spells out all configurable attributes, including their data types. For a more detailed description of the CustomResourceDefinition see the [Kubernetes documentation](#).

Example 16-1. A custom resource schema to represent a smoke test

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: smoketests.stable.bmuschko.com ①
spec:
  group: stable.bmuschko.com            ②
  versions:
    - name: v1                           ③
      served: true
      storage: true
```

```

schema:
  openAPIV3Schema:
    type: object
    properties:
      spec: ❸
        type: object
        properties:
          service:
            type: string
          path:
            type: string
          timeout:
            type: integer
          retries:
            type: integer
      scope: Namespaced
      names: ❹
        plural: smoketests
        singular: smoketest
        kind: SmokeTest
      shortNames:
        - st

```

- ❶ The combination of the identifiers <plural>.<group>.
- ❷ The API group to be used by CRD.
- ❸ The versions supported by the CRD. A version can define 0..n attributes.
- ❹ The attributes to be set by the custom type.
- ❺ The identifiers for the custom type, e.g., the kind and the singular/plural/short names.

The file *smoketest-resource.yaml* contains the YAML content shown in the previous example. You can now create the object for the schema with the typical `kubectl` commands:

```
$ kubectl apply -f smoketest-resource.yaml
customresourcedefinition.apiextensions.k8s.io/smoketests.stable.bmuschko.com \
created
```

Instantiating an Object for the CRD

Once the schema object has been created, you can create objects for the new custom type. The YAML manifest in [Example 16-2](#) defines a primitive of type `SmokeTest` named `backend-smoke-test` for the Service named `backend`. As you can see in the manifest, additional attributes can be specified to fine-tune its runtime behavior.

Example 16-2. Instantiation of CRD kind SmokeTest

```
apiVersion: stable.bmuschko.com/v1      ①
kind: SmokeTest                         ②
metadata:
  name: backend-smoke-test
spec:
  service: backend                      ③
  path: /health                          ③
  timeout: 600                           ③
  retries: 3                            ③
```

- ① The group and Version of the custom kind.
- ② The kind defined by CRD.
- ③ The attributes and their values that make the custom kind configurable.

Go ahead and create the SmokeTest object from the file *smoketest.yaml*:

```
$ kubectl apply -f smoketest.yaml
smoketest.stable.bmuschko.com/backend-smoke-test created
```

You can interact with the `backend-smoke-test` like any other object in Kubernetes. For example, to list the object use the `get` command. To delete the object, use the `delete` command. The following commands show those operations in action:

```
$ kubectl get smoketest backend-smoke-test
NAME          AGE
backend-smoke-test 12s
$ kubectl delete smoketest backend-smoke-test
smoketest.stable.bmuschko.com "backend-smoke-test" deleted
```

You can create more objects of this type as needed to execute smoke tests against other Services, although each of the objects needs a unique name. We now have the CRD schema object in place. In the next section, we'll interact with it.

Discovering CRDs

A CRD schema registers a new API resource. Every API resource can be discovered. You can list custom API resources in the same way as you would for built-in API resources. The following command lists API resources by the API group `stable.bmuschko.com`:

```
$ kubectl api-resources --api-group=stable.bmuschko.com
NAME      SHORTNAMES   APIVERSION      NAMESPACED   KIND
smoketests  st           stable.bmuschko.com/v1  true        SmokeTest
```

CRD objects can be interacted with like any other object using `kubectl`. You can create, read, update, and delete them. The following command lists all CRDs installed in the cluster:

```
$ kubectl get crds
NAME                           CREATED AT
smoketests.stable.bmuschko.com  2023-05-04T14:49:40Z
```

The CRD you see in the output is the one representing a smoke test. If you see other CRDs in the output then they may have been provided by the external operator installed in your cluster.

Implementing a Controller

The smoke test object just represents data and won't be useful by itself. You need to add a controller implementation that does something with those objects. In a nutshell, a controller acts as a reconciliation process by inspecting the state of CRD objects via calls to the Kubernetes API.

At runtime, a controller implementation needs to poll for new SmokeTest objects and perform the HTTPS requests to the configured Service endpoint. Finally, the controller inspects the response from the request, evaluates the result, and sends it to an external service to be recorded in a database. A dashboard can then tap into historical results and render them as needed, e.g., as charts and graphs.

Controllers can use one of the Kubernetes [client libraries](#), written in Go or Python, to access custom resources. Visit the relevant documentation for more information and examples on how to implement a controller.

Summary

A CRD schema defines the structure of a custom resource. The schema includes the group, name, version, and its configurable attributes. New objects of this kind can be created after registering the schema. You can interact with a custom object using `kubectl` with the same CRUD commands used by any other primitive.

CRDs realize their full potential when combined with a controller implementation. The controller implementation inspects the state of specific custom objects and reacts based on their discovered state. Kubernetes refers to the CRD and the corresponding controller as the operator pattern. The Kubernetes community has implemented many operators to fulfill custom requirements. You can install them into your cluster to reuse the functionality.

Exam Essentials

Acquire a high-level understanding of configurable options for a CRD schema

You are not expected to implement a CRD schema. All you need to know is how to discover and use them with `kubectl`. Controller implementations are definitely outside the scope of the exam.

Practice the commands for installing and discovering CRDs

Learn how to use the `kubectl get crds` command to discover installed CRDs, and how to create objects from a CRD schema. If you want to explore further, install an open source CRD, such as the [Prometheus operator](#) or the [Jaeger operator](#), and inspect its schema.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. You decide to manage a [MongoDB](#) installation in Kubernetes with the help of the [official community operator](#). This operator provides a CRD. After installing the operator, you will interact with the CRD.

Navigate to the directory `app-a/ch16/mongodb-operator` of the checked-out GitHub repository [bmuschko/ckad-study-guide](#). Install the operator using the following command: `kubectl apply -f mongodbcommunity.mongodb.com_mongodb_community.yaml`.

List all CRDs using the appropriate `kubectl` command. Can you identify the CRD that was installed by the installation procedure?

Inspect the schema of the CRD. What are the type and property names of this CRD?

2. As an application developer, you may want to install Kubernetes functionality that extends the platform using the Kubernetes operator pattern. The objective of this exercise is to familiarize yourself with creating and managing CRDs. You will not need to write a controller.

Create a CRD resource named `backup.example.com` with the following specifications:

- Group: `example.com`
- Version: `v1`
- Kind: `Backup`
- Singular: `backup`

- Plural: backups
- Properties of type `string`: `cronExpression`, `podName`, `path`

Retrieve the details for the `Backup` custom resource created in the previous step.

Create a custom object named `nginx-backup` for the CRD. Provide the following property values:

- `cronExpression: 0 0 * * *`
- `podName: nginx`
- `path: /usr/local/nginx`

Retrieve the details for the `nginx-backup` object created in the previous step.

Authentication, Authorization, and Admission Control

The API server is the gateway to the Kubernetes cluster. Any human user, client (e.g., `kubectl`), cluster component, or service account will access the API server by making a RESTful API call via HTTPS. It is *the* central point for performing operations like creating a Pod or deleting a Service.

In this chapter, we'll focus on the security-specific aspects relevant to the API server. For a detailed discussion on the inner workings of the API server and use of the Kubernetes API, refer to *Managing Kubernetes* by Brendan Burns and Craig Tracey (O'Reilly).

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objectives:

- Understand authentication, authorization, and admission control
- Understand SecurityContext

Processing a Request

[Figure 17-1](#) illustrates the stages a request goes through when a call is made to the API server. For reference, you can find more information in the [Kubernetes documentation](#).

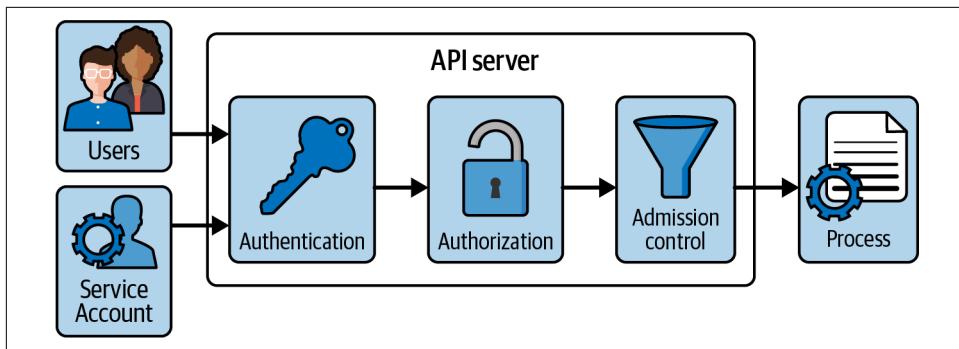


Figure 17-1. API server request processing

The first stage of request processing is *authentication*. Authentication validates the identity of the caller by inspecting the client certificates or bearer tokens. If the bearer token is associated with a service account, then it will be verified here.

The second stage determines if the identity provided in the first stage can access the verb and HTTP path request. Therefore, stage two deals with *authorization* of the request, which is implemented with the standard Kubernetes RBAC model. Here, we ensure that the service account is allowed to list Pods or create a new Service object if that has been requested.

The third stage of request processing deals with *admission control*. Admission control verifies if the request is well formed or potentially needs to be modified before the request is processed. An admission control policy could, for example, ensure that the request for creating a Pod includes the definition of a specific label. If the request doesn't define the label, then it is rejected.

Authentication with kubectl

Developers interact with the Kubernetes API by running the `kubectl` command line tool. Whenever you execute a command with `kubectl`, the underlying HTTPS call to the API server needs to authenticate.

The Kubeconfig

Credentials for the use of `kubectl` are stored in the file `$HOME/.kube/config`, also known as the *kubeconfig file*. The kubeconfig file defines the API server endpoints of the clusters we want to interact with, as well as a list of users registered with the cluster, including their credentials in the form of client certificates. The mapping between a cluster and user for a given namespace is called a *context*. `kubectl` uses the currently selected context to know which cluster to talk to and which credentials to use.



You can point the environment variable `KUBECONFIG` to a set of kubeconfig files. At runtime, `kubectl` will merge the contents of the set of defined kubeconfig files and use them. By default, `KUBECONFIG` is not set and falls back to `$HOME/.kube/config`.

[Example 17-1](#) shows a kubeconfig file. Be aware that file paths assigned in the example are user-specific and may differ in your own environment. You can find a detailed description of all configurable attributes in the [Config resource type](#) API documentation.

Example 17-1. A kubeconfig file

```
apiVersion: v1
kind: Config
clusters:          ①
- cluster:
  certificate-authority: /Users/bmuschko/.minikube/ca.crt
  extensions:
  - extension:
    last-update: Mon, 09 Oct 2023 07:33:01 MDT
    provider: minikube.sigs.k8s.io
    version: v1.30.1
    name: cluster_info
  server: https://127.0.0.1:63709
  name: minikube
contexts:          ②
- context:
  cluster: minikube
  user: bmuschko
  name: bmuschko
- context:
  cluster: minikube
  extensions:
  - extension:
    last-update: Mon, 09 Oct 2023 07:33:01 MDT
    provider: minikube.sigs.k8s.io
    version: v1.30.1
    name: context_info
  namespace: default
  user: minikube
  name: minikube
current-context: minikube  ③
preferences: {}
users:              ④
- name: bmuschko
  user:
    client-key-data: <REDACTED>
- name: minikube
  user:
```

```
client-certificate: /Users/bmuschko/.minikube/profiles/minikube/client.crt  
client-key: /Users/bmuschko/.minikube/profiles/minikube/client.key
```

- ❶ A list of referential names to clusters and their API server endpoints.
- ❷ A list of referential names to contexts (a combination of cluster and user).
- ❸ The currently selected context.
- ❹ A list of referential names to users and their credentials.

User management is handled by the cluster administrator. The administrator creates a user representing the developer and hands the relevant information (username and credentials) to the human wanting to interact with the cluster via `kubectl`. Alternatively, it is also possible to integrate with external identity providers for authentication purposes, e.g., via [OpenID Connect](#).

Creating a new user manually consists of multiple steps, as described in the [Kubernetes documentation](#). The developer would then add the user to the `kubeconfig` file on the machine intended to interact with the cluster.

Managing Kubeconfig Using `kubectl`

You do not have to manually edit the `kubeconfig` file(s) to change or add configuration. `Kubectl` provides commands for reading and modifying its contents. The following commands provide an overview. You can find additional examples for commands in the [kubectl cheatsheet](#).

To view the merged contents of the `kubeconfig` file(s), run the following command:

```
$ kubectl config view
apiVersion: v1
kind: Config
clusters:
...
...
```

To render the currently selected context, use the `current-context` subcommand. The context named `minikube` is the active one:

```
$ kubectl config current-context
minikube
```

To change the context, provide the name with the `use-context` subcommand. Here, we are switching to the context `bmuschko`:

```
$ kubectl config use-context bmuschko
Switched to context "bmuschko".
```

To register a user with the kubeconfig file(s), use the `set-credentials` subcommand. We are choosing to assign the username `myuser` and point to the client certificate by providing the corresponding CLI flags:

```
$ kubectl config set-credentials myuser \
--client-key=myuser.key --client-certificate=myuser.crt \
--embed-certs=true
```

For the exam, familiarize yourself with the `kubectl config` command. Every task in the exam will require you to work with a specific context and/or namespace.

Authorization with Role-Based Access Control

We've learned that the API server will try to authenticate any request sent using `kubectl` by verifying the provided credentials. An authenticated request will then need be checked against the permissions assigned to the requestor. The authorization phase of the API processing workflow checks if the operation is permitted against the requested API resource.

In Kubernetes, those permissions can be controlled using Role-Based Access Control (RBAC). In a nutshell, RBAC defines policies for users, groups, and service accounts by allowing or disallowing access to manage API resources. Enabling and configuring RBAC is mandatory for any organization with a strong emphasis on security.

Setting permissions is the responsibility of a cluster administrator. In the following sections, we'll briefly talk about the effects of RBAC on requests from users and service accounts.

RBAC Overview

RBAC helps with implementing a variety of use cases:

- Establishing a system for users with different roles to access a set of Kubernetes resources
- Controlling processes (associated with a service account) running in a Pod and performing operations against the Kubernetes API
- Limiting the visibility of certain resources per namespace

RBAC consists of three key building blocks, as shown in [Figure 17-2](#). Together, they connect API primitives and their allowed operations to the so-called subject, which is a user, a group, or a service account.

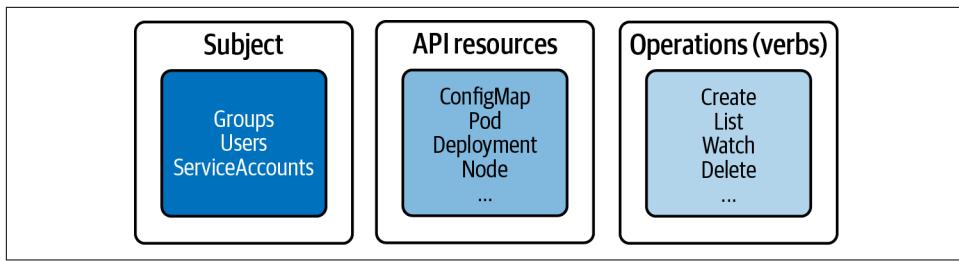


Figure 17-2. RBAC key building blocks

Each block's responsibilities are as follows:

Subject

The user or service account that wants to access a resource

Resource

The Kubernetes API resource type (e.g., a Deployment or node)

Verb

The operation that can be executed on the resource (e.g., creating a Pod or deleting a Service)

Understanding RBAC API Primitives

With these key concepts in mind, let's look at the Kubernetes API primitives that implement the RBAC functionality:

Role

The Role API primitive declares the API resources and their operations this rule should operate on in a specific namespace. For example, you may want to say “allow listing and deleting of Pods,” or you may express “allow watching the logs of Pods,” or even both with the same Role. Any operation that is not spelled out explicitly is disallowed as soon as it is bound to the subject.

RoleBinding

The RoleBinding API primitive *binds* the Role object to the subject(s) in a specific namespace. It is the glue for making the rules active. For example, you may want to say “bind the Role that permits updating Services to the user John Doe.”

Figure 17-3 shows the relationship between the involved API primitives. Keep in mind that the image renders only a selected list of API resource types and operations.

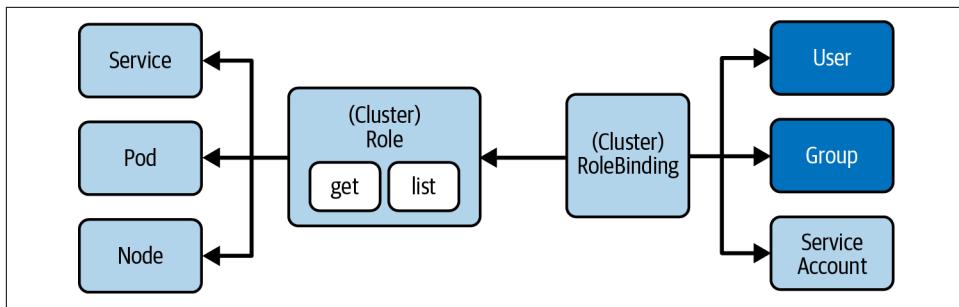


Figure 17-3. RBAC primitives

The following sections demonstrate the namespace-wide usage of Roles and RoleBindings, but the same operations and attributes apply to cluster-wide Roles and RoleBindings, discussed in “[Namespace-Wide and Cluster-Wide RBAC](#)” on page 198.

Default User-Facing Roles

Kubernetes defines a set of default Roles. You can assign them to a subject via a RoleBinding or define your own, custom Roles depending on your needs. [Table 17-1](#) describes the default user-facing Roles.

Table 17-1. Default user-facing Roles

Default ClusterRole	Description
cluster-admin	Allows read and write access to resources across all namespaces.
admin	Allows read and write access to resources in namespace including Roles and RoleBindings.
edit	Allows read and write access to resources in namespace except Roles and RoleBindings. Provides access to Secrets.
view	Allows read-only access to resources in namespace except Roles, RoleBindings, and Secrets.

To define new Roles and RoleBindings, you will have to use a context that allows for creating or modifying them, that is, cluster-admin or admin.

Creating Roles

Roles can be created imperatively with the `create role` command. The most important options for the command are `--verb` for defining the verbs, aka operations, and `--resource` for declaring a list of API resources (core primitives as well as CRDs). The following command creates a new Role for the resources Pod, Deployment, and Service with the verbs `list`, `get`, and `watch`:

```
$ kubectl create role read-only --verb=list,get,watch \
  --resource=pods,deployments,services
role.rbac.authorization.k8s.io/read-only created
```

Declaring multiple verbs and resources for a single imperative `create` `role` command can be declared as a comma-separated list for the corresponding command-line option or as multiple arguments. For example, `--verb=list,get,watch` and `--verb=list --verb=get --verb=watch` carry the same instructions. You also can use the wildcard `*` to refer to all verbs or resources.

The command-line option `--resource-name` spells out one or many object names that the policy rules should apply to. A name of a Pod could be `nginx` and listed here with its name. Providing a list of resource names is optional. If no names have been provided, then the provided rules apply to all objects of a resource type.

The declarative approach can become a little lengthy. As you can see in [Example 17-2](#), the section `rules` lists the resources and verbs. Resources with an API group, like Deployments that use the API version `apps/v1`, need to explicitly declare it under the attribute `apiGroups`. All other resources (e.g., Pods and Services), simply use an empty string, as their API version doesn't contain a group. Be aware that the imperative command for creating a Role automatically determines the API group.

Example 17-2. A YAML manifest defining a Role

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: read-only
rules:
- apiGroups:
  - ""
    resources:
    - pods
    - services
    verbs:
    - list
    - get
    - watch
- apiGroups: ①
  - apps
    resources:
    - deployments
    verbs:
    - list
    - get
    - watch
```

- ① Any resource that belongs to an API group need to be listed as an explicit rule in addition to the API resources that do not belong to an API group.

Listing Roles

Once the Role has been created, its object can be listed. The list of Roles renders only the name and the creation timestamp. Each of the listed roles does not give away any of its details:

```
$ kubectl get roles
NAME      CREATED AT
read-only  2021-06-23T19:46:48Z
```

Rendering Role Details

You can inspect the details of a Role using the `describe` command. The output renders a table that maps a resource to its permitted verbs:

```
$ kubectl describe role read-only
Name:           read-only
Labels:         <none>
Annotations:   <none>
PolicyRule:
  Resources      Non-Resource URLs  Resource Names  Verbs
  -----          -----          -----
  pods            []                []              [list get watch]
  services        []                []              [list get watch]
  deployments.apps []               []              [list get watch]
```

This cluster has no resources created, so the list of resource names in the following console output is currently empty.

Creating RoleBindings

The imperative command creating a RoleBinding object is `create rolebinding`. To bind a Role to the RoleBinding, use the `--role` command-line option. The subject type can be assigned by declaring the options `--user`, `--group`, or `--serviceaccount`. The following command creates the RoleBinding with the name `read-only-binding` to the user called `bmuschko`:

```
$ kubectl create rolebinding read-only-binding --role=read-only --user=bmuschko
rolebinding.rbac.authorization.k8s.io/read-only-binding created
```

[Example 17-3](#) shows a YAML manifest representing the RoleBinding. You can see from the structure that a role can be mapped to one or many subjects. The data type is an array indicated by the dash character under the attribute `subjects`. At this time, only the user `bmuschko` has been assigned.

Example 17-3. A YAML manifest defining a RoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-only-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: read-only
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: bmuschko
```

Listing RoleBindings

The most important information the list of RoleBindings displays is the associated Role. The following command shows that the RoleBinding `read-only-binding` has been mapped to the Role `read-only`:

```
$ kubectl get rolebindings
NAME          ROLE          AGE
read-only-binding  Role/read-only  24h
```

The output does not provide an indication of the subjects. You will need to render the details of the object for more information, as described in the next section.

Rendering RoleBinding Details

RoleBindings can be inspected using the `describe` command. The output renders a table of subjects and the assigned role. The following example renders the descriptive representation of the RoleBinding named `read-only-binding`:

```
$ kubectl describe rolebinding read-only-binding
Name:           read-only-binding
Labels:         <none>
Annotations:   <none>
Role:
  Kind:  Role
  Name:  read-only
Subjects:
  Kind  Name      Namespace
  ----  --       -----
  User  bmuschko
```

Seeing the RBAC Rules in Effect

Let's see how Kubernetes enforces the RBAC rules for the scenario we set up so far. First, we'll create a new Deployment with the `cluster-admin` permissions. In Minikube, those permissions are available to the context `minikube` by default:

```
$ kubectl config current-context
minikube
$ kubectl create deployment myapp --image=:1.25.2 --port=80 --replicas=2
deployment.apps/myapp created
```

Now, we'll switch the context for the user `bmuschko`:

```
$ kubectl config use-context bmuschko-context
Switched to context "bmuschko-context".
```

Remember that the user `bmuschko` is permitted to list Deployments. We'll verify that by using the `get deployments` command:

```
$ kubectl get deployments
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
myapp    2/2       2           2           8s
```

The RBAC rules allow listing Deployments, Pods, and Services only. The following command tries to list the ReplicaSets, which results in an error:

```
$ kubectl get replicasesets
Error from server (Forbidden): replicasesets.apps is forbidden: User "bmuschko" \
cannot list resource "replicasesets" in API group "apps" in the namespace "default"
```

A similar behavior can be observed when trying to use verbs other than `list`, `get`, or `watch`. The following command tries to delete a Deployment:

```
$ kubectl delete deployment myapp
Error from server (Forbidden): deployments.apps "myapp" is forbidden: User \
"bmuschko" cannot delete resource "deployments" in API group "apps" in the \
namespace "default"
```

At any given time, you can check a user's permissions with the `auth can-i` command. The command gives you the option to list all permissions or check a specific permission:

```
$ kubectl auth can-i --list --as bmuschko
Resources          Non-Resource URLs      Resource Names      Verbs
...
pods              []                      []                  [list get watch]
services          []                      []                  [list get watch]
deployments.apps  []                      []                  [list get watch]
$ kubectl auth can-i list pods --as bmuschko
yes
```

Namespace-Wide and Cluster-Wide RBAC

Roles and RoleBindings apply to a particular namespace. You will have to specify the namespace when creating both objects. Sometimes, a set of Roles and RoleBindings needs to apply to multiple namespaces or even to the whole cluster. For a cluster-wide definition, Kubernetes offers the API resource types ClusterRole and ClusterRoleBinding. The configuration elements are effectively the same. The only difference is the value of the kind attribute:

- To define a cluster-wide Role, use the imperative subcommand `clusterrole` or the kind `ClusterRole` in the YAML manifest.
- To define a cluster-wide RoleBinding, use the imperative subcommand `clusterrolebinding` or the kind `ClusterRoleBinding` in the YAML manifest.

ClusterRoles and ClusterRoleBindings not only set up cluster-wide permissions to a namespaced resource, but they can also be used to set up permissions for non-namespaced resources like CRDs and nodes.

Working with Service Accounts

We've been using the `kubectl` executable to run operations against a Kubernetes cluster. Under the hood, its implementation calls the API server by making an HTTP call to the exposed endpoints. Some applications running inside of a Pod may have to communicate with the API server as well. For example, the application may ask for specific cluster node information or available namespaces.

Pods can use a service account to authenticate with the API server through an authentication token. A Kubernetes administrator assigns rules to a service account via RBAC to authorize access to specific resources and actions as illustrated in [Figure 17-4](#).

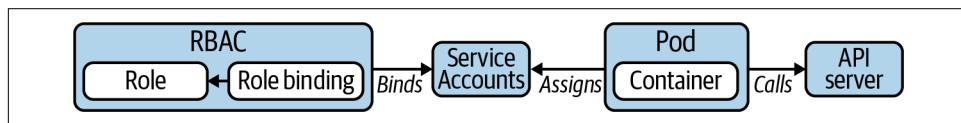


Figure 17-4. Using a service account to communicate with an API server

A Pod doesn't necessarily need to be involved in the process. Other use cases call for leveraging a service account outside of a Kubernetes cluster. For example, you may want to communicate with the API server as part of CI/CD pipeline automation step. The service account can provide the credentials to authenticate with the API server.

The Default Service Account

So far, we haven't defined a service account for a Pod. If not assigned explicitly, a Pod uses the **default service account**, which has the same permissions as an unauthenticated user. This means that the Pod cannot view or modify the cluster state or list or modify any of its resources. The default service account can however request basic cluster information via the assigned `system:discovery` Role.

You can query for the available service accounts with the subcommand `serviceaccounts`. You should see only the default service account listed in the output:

```
$ kubectl get serviceaccounts  
NAME      SECRETS   AGE  
default    0          4d
```

While you can execute the `kubectl` operation to delete the default service account, Kubernetes will reinstantiate the service account immediately.

Creating a Service Account

You can create a custom service account object using the imperative and declarative approach. This command creates a service account object with the name `cicd-bot`. The assumption here is to use the service account for calls to the API server made by a CI/CD pipeline:

```
$ kubectl create serviceaccount cicd-bot  
serviceaccount/cicd-bot created
```

You can also represent the service account in the form of a manifest. In its simplest form, the definition assigns the kind `ServiceAccount` and a name, as shown in [Example 17-4](#).

Example 17-4. YAML manifest for a service account

```
apiVersion: v1  
kind: ServiceAccount  
metadata:  
  name: cicd-bot
```

You can set a couple of **configuration options** for a service account. For example, you may want to disable automounting of the authentication token when assigning the service account to a Pod. Although you will not need to understand those configuration options for the exam, it makes sense to dive deeper into security best practices by reading up on them in the Kubernetes documentation.

Setting Permissions for a Service Account

It's important to limit the permissions to only the service accounts that are necessary for the application to function. The next sections will explain how to achieve this to minimize the potential attack surface.

For this scenario to work, you'll need to create a `ServiceAccount` object and assign it to the Pod. Service accounts can be tied in with RBAC and assigned a Role and Role-Binding to define which operations they should be allowed to perform.

Binding the service account to a Pod

As a starting point, we will set up a Pod that lists all Pods and Deployments in the namespace `k97` by calling the Kubernetes API. The call is made as part of an infinite loop every ten seconds. The response from the API call will be written to standard output accessible via the Pod's logs.



Accessing the API server endpoint

Accessing the Kubernetes API from a Pod is straightforward. Instead of using the IP address and port for the API server Pod, you can simply refer to a Service named `kubernetes.default.svc` instead. This special Service lives in the `default` namespace and is stood up by the cluster automatically.

To authenticate against the API server, we'll send a bearer token associated with the service account used by the Pod. The default behavior of a service account is to automount API credentials on the path `/var/run/secrets/kubernetes.io/serviceaccount/token`. We'll simply get the contents of the file using the `cat` command-line tool and send them along as a header for the HTTP request. [Example 17-5](#) defines the namespace, the service account, and the Pod in a single YAML manifest file: `setup.yaml`.

Example 17-5. YAML manifest for assigning a service account to a Pod

```
apiVersion: v1
kind: Namespace
metadata:
  name: k97
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sa-api
  namespace: k97
---
apiVersion: v1
kind: Pod
```

```

metadata:
  name: list-objects
  namespace: k97
spec:
  serviceAccountName: sa-api    ①
  containers:
  - name: pods
    image: alpine/curl:3.14
    command: ['sh', '-c', 'while true; do curl -s -k -m 5 -H \
      "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/ \
      serviceaccount/token)" https://kubernetes.default.svc.cluster. \
      local/api/v1/namespaces/k97/pods; sleep 10; done'] ②
  - name: deployments
    image: alpine/curl:3.14
    command: ['sh', '-c', 'while true; do curl -s -k -m 5 -H \
      "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/ \
      serviceaccount/token)" https://kubernetes.default.svc.cluster. \
      local/apis/apps/v1/namespaces/k97/deployments; \
      sleep 10; done'] ③

```

- ① The service account referenced by name used for communicating with the Kubernetes API.
- ② Performs an API call to retrieve the list of Pods in the namespace k97.
- ③ Performs an API call to retrieve the list of Deployments in the namespace k97.

Create the objects from the YAML manifest with the following command:

```
$ kubectl apply -f setup.yaml
namespace/k97 created
serviceaccount(sa-api) created
pod/list-objects created
```

Verifying the default permissions

The Pod named `list-objects` makes a call to the API server to retrieve the list of Pods and Deployments in dedicated containers. The container `pods` performs the call to list Pods. The container `deployments` sends a request to the API server to list Deployments.

As explained in the [Kubernetes documentation](#), the default RBAC policies do not grant any permissions to service accounts outside of the `kube-system` namespace. The logs of the containers `pods` and `deployments` return an error message indicating that the service account `sa-api` is not authorized to list the resources:

```
$ kubectl logs list-objects -c pods -n k97
{
  "kind": "Status",
  "apiVersion": "v1",
```

```

"metadata": {},
"status": "Failure",
"message": "pods is forbidden: User \\"system:serviceaccount:k97:sa-api\\" \
            cannot list resource \\"pods\\" in API group \\"\\\" in the \
            namespace \\"k97\\\"",
"reason": "Forbidden",
"details": {
    "kind": "pods"
},
"code": 403
}
$ kubectl logs list-objects -c deployments -n k97
{
    "kind": "Status",
    "apiVersion": "v1",
    "metadata": {},
    "status": "Failure",
    "message": "deployments.apps is forbidden: User \
                \\"system:serviceaccount:k97:sa-api\\" cannot list resource \
                \\"deployments\\" in API group \\"apps\\" in the namespace \
                \\"k97\\\"",
    "reason": "Forbidden",
    "details": {
        "group": "apps",
        "kind": "deployments"
    },
    "code": 403
}

```

Next up, we'll stand up a Role and RoleBinding object with the required API permissions to perform the necessary calls.

Creating the Role

Start by defining the Role named `list-pods-role` shown in [Example 17-6](#) in the file `role.yaml`. The set of the rules adds only the Pod resource and the verb `list`.

Example 17-6. YAML manifest for a Role that allows listing Pods

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: list-pods-role
  namespace: k97
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["list"]

```

Create the object by pointing to its corresponding YAML manifest file:

```
$ kubectl apply -f role.yaml
role.rbac.authorization.k8s.io/list-pods-role created
```

Creating the RoleBinding

Example 17-7 defines the YAML manifest for the RoleBinding in the file `rolebinding.yaml`. The RoleBinding maps the Role `list-pods-role` to the service account named `sa-pod-api` and applies it only to the namespace `k97`.

Example 17-7. YAML manifest for a RoleBinding attached to a service account

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: serviceaccount-pod-rolebinding
  namespace: k97
subjects:
- kind: ServiceAccount
  name: sa-api
roleRef:
  kind: Role
  name: list-pods-role
  apiGroup: rbac.authorization.k8s.io
```

Create both RoleBinding objects using the `apply` command:

```
$ kubectl apply -f rolebinding.yaml
rolebinding.rbac.authorization.k8s.io/serviceaccount-pod-rolebinding created
```

Verifying the granted permissions

With the granted `list` permissions, the service account can now properly retrieve all the Pods in the `k97` namespace. The `curl` command in the `pods` container succeeds, as shown in the following output:

```
$ kubectl logs list-objects -c pods -n k97
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "628"
  },
  "items": [
    {
      "metadata": {
        "name": "list-objects",
        "namespace": "k97",
        ...
      }
    ]
}
```

We did not grant any permissions to the service account for other resources. Listing the Deployments in the k97 namespace still fails. The following output shows the response from the curl command in the deployments namespace:

```
$ kubectl logs list-objects -c deployments -n k97
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
  "message": "deployments.apps is forbidden: User \\"system:serviceaccount:k97:sa-api\\" cannot list resource \\"deployments\\" in API group \\"apps\\" in the namespace \\"k97\\\"",
  "reason": "Forbidden",
  "details": {
    "group": "apps",
    "kind": "deployments"
  },
  "code": 403
}
```

Feel free to modify the Role object to allow listing Deployment objects as well.

Admission Control

The last phase of processing a request to the API server is admission control. Admission control is implemented by admission controllers. An admission controller provides a way to approve, deny, or mutate a request before it takes effect.

Admission controllers can be registered with the configuration of the API server. By default, the configuration file can be found at `/etc/kubernetes/manifests/kube-apiserver.yaml`. It is the cluster administrator's job to manage the API server configuration. The following command-line invocation of the API server enables the **admission control plugins** named `NamespaceLifecycle`, `PodSecurity` and `LimitRanger`:

```
$ kube-apiserver --enable-admission-plugins=NamespaceLifecycle,PodSecurity, \
  LimitRanger
```

As a developer, you are inadvertently using admission control plugins that have been configured for you. One example is the `LimitRanger` and the `ResourceQuota`, we'll discuss in “Working with Limit Ranges” on page 215 and “Working with Resource Quotas” on page 211.

Summary

The API server processes requests to the Kubernetes API. Every request has to go through three phases: authentication, authorization, and admission control. Every phase can short-circuit the processing. For example, if the credentials sent with the request cannot be authenticated, then the request will be dropped immediately.

We looked at examples of all phases. The authentication phase covered `kubectl` as the client making a call to the Kubernetes API. The `kubeconfig` file serves as configuration source for named cluster, users, and their credentials. In Kubernetes, authorization is handled by RBAC. We learned the Kubernetes primitives that let you configure permissions for API resources tied to one or many subjects. Finally, we briefly covered the purpose of admission control and listed some plugins that act as controllers for validating or mutating a request to the Kubernetes API.

Exam Essentials

Practice interacting with the Kubernetes API

This chapter demonstrated some ways to communicate with the Kubernetes API. We performed API requests by switching to a user context and with the help of a RESTful API call using `curl`. Explore the [Kubernetes API](#) and its endpoints on your own for broader exposure.

Understand the implications of defining RBAC rules for users and service accounts

Anonymous user requests to the Kubernetes API will not allow any substantial operations. For requests coming from a user or a service account, you will need to carefully analyze permissions granted to the subject. Learn the ins and outs of defining RBAC rules by creating the relevant objects to control permissions. Service accounts automount a token when used in a Pod. Expose the token as a volume only if you are intending to make API calls from the Pod.

Learn about the basic need for admission control

For the exam you will not need to understand how to configure admission control plugins in the API server. Developers interact with them, but configuration tasks are up to the cluster administrator. Read up on different plugins to gain a better understanding of the admission control landscape.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. The premise of this exercise is to create a new user and add her to the `kubeconfig` file. You will then define a context that uses the user, switch to the context, and execute a `kubectl` command.

Create a certificate for a user named `mary`. Do not provide any permissions to the user.

Add the user to the kubeconfig file. Define the context named `mary-context` that assigns the user to a cluster already available in the kubeconfig file.

Set the currently selected context to `mary-context`. Create a Pod using `kubectl`. What result do you expect to see?

2. You will use RBAC to grant permissions to a service account. The permissions should apply only to certain API resources and operations.

Create a new namespace named `t23`. Create a Pod named `service-list` in the namespace `t23`. The container uses the image `alpine/curl:3.14` and makes a `curl` call to the Kubernetes API that lists Service objects in the default namespace in an infinite loop.

Create and attach the service account `api-call` to the Pod.

Inspect the container logs after the Pod has been started. What response do you expect to see from the `curl` command?

Assign a ClusterRole and RoleBinding to the service account that allows only the operation needed by the Pod. Note the response from the `curl` command.

3. Identify the admission controller plugins that have been configured for the API server.

Locate the configuration file of the API server.

Inspect the command-line flag that defines the admission controller plugins. Capture the value.

Resource Requirements, Limits, and Quotas

Workload executed in Pods will consume a certain amount of resources (e.g., CPU and memory). You should define resource requirements for those applications. On a container level, you can define a minimum amount of resources needed to run the application, as well as the maximum amount of resources the application is allowed to consume. Application developers should determine the right-sizing with load tests or at runtime by monitoring the resource consumption.



Kubernetes measures CPU resources in millicores and memory resources in bytes. That's why you might see resources defined as 600m or 100Mi. For a deep dive on those resource units, it's worth cross-referencing the section "[Resource units in Kubernetes](#)" in the official documentation.

Kubernetes administrators can put measures in place to enforce the use of available resource capacity. We'll discuss two Kubernetes primitives in this realm, the ResourceQuota and the LimitRange. The ResourceQuota defines aggregate resource constraints on a namespace level. A LimitRange is a policy that constrains or defaults the resource allocations for a single object of a specific type (such as for a Pod or a PersistentVolumeClaim).

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objective:

- Understand and define resource requirements, limits, and quotas

Working with Resource Requirements

It's recommended practice that you specify resource requests and limits for every container. Determining those resource expectations is not always easy, specifically for applications that haven't been exercised in a production environment yet. Load testing the application early during the development cycle can help with analyzing the resource needs. Further adjustments can be made by monitoring the application's resource consumption after deploying it to the cluster.

Defining Container Resource Requests

One metric that comes into play for workload scheduling is the *resource request* defined by the containers in a Pod. Commonly used resources that can be specified are CPU and memory. The scheduler ensures that the node's resource capacity can fulfill the resource requirements of the Pod. More specifically, the scheduler determines the sum of resource requests per resource type across all containers defined in the Pod and compares them with the node's available resources.

Each container in a Pod can define its own resource requests. [Table 18-1](#) describes the available options including an example value.

Table 18-1. Options for resource requests

YAML attribute	Description	Example value
spec.containers[].resources.requests.cpu	CPU resource type	500m (five hundred millicpu)
spec.containers[].resources.requests.memory	Memory resource type	64Mi (2^26 bytes)
spec.containers[].resources.requests.hugepages-<size>	Huge page resource type	hugepages-2Mi:60Mi
spec.containers[].resources.requests.ephemeral-storage	Ephemeral storage resource type	4Gi

To clarify the uses of these resource requests, we'll look at an example definition. The Pod YAML manifest shown in [Example 18-1](#) defines two containers, each with its own resource requests. Any node that is allowed to run the Pod needs to be able to support a minimum memory capacity of 320Mi and 1250m CPU, the sum of resources across both containers.

Example 18-1. Setting container resource requests

```
apiVersion: v1
kind: Pod
metadata:
  name: rate-limiter
spec:
```

```

containers:
- name: business-app
  image: bmuschko/nodejs-business-app:1.0.0
  ports:
    - containerPort: 8080
  resources:
    requests:
      memory: "256Mi"
      cpu: "1"
- name: ambassador
  image: bmuschko/nodejs-ambassador:1.0.0
  ports:
    - containerPort: 8081
  resources:
    requests:
      memory: "64Mi"
      cpu: "250m"

```

It's certainly possible that a Pod cannot be scheduled due to insufficient resources available on the nodes. In those cases, the event log of the Pod will indicate this situation with the reasons `PodExceedsFreeCPU` or `PodExceedsFreeMemory`. For more information on how to troubleshoot and resolve this situation, see the relevant [section in the documentation](#).

Defining Container Resource Limits

Another metric you can set for a container is the *resource limits*. Resource limits ensure that the container cannot consume more than the allotted resource amounts. For example, you could express that the application running in the container should be constrained to 1000m of CPU and 512Mi of memory.

Depending on the container runtime used by the cluster, exceeding any of the allowed resource limits results in a termination of the application process running in the container or results in the system preventing the allocation of resources beyond the limits. For an in-depth discussion on how resource limits are treated by the container runtime Docker Engine, see the [documentation](#).

Table 18-2 describes the available options including an example value.

Table 18-2. Options for resource limits

YAML attribute	Description	Example value
<code>spec.containers[].resources.limits.cpu</code>	CPU resource type	500m (500 millicpu)
<code>spec.containers[].resources.limits.memory</code>	Memory resource type	64Mi (2^26 bytes)
<code>spec.containers[].resources.limits.hugepages-<size></code>	Huge page resource type	hugepages-2Mi:60Mi

YAML attribute	Description	Example value
spec.containers[].resources.limits.ephemeral-storage	Ephemeral storage resource type	4Gi

Example 18-2 shows the definition of limits in action. Here, the container named `business-app` cannot use more than 512Mi of memory. The container named `ambassador` defines a limit of 128Mi of memory.

Example 18-2. Setting container resource limits

```
apiVersion: v1
kind: Pod
metadata:
  name: rate-limiter
spec:
  containers:
    - name: business-app
      image: bmuschko/nodejs-business-app:1.0.0
      ports:
        - containerPort: 8080
      resources:
        limits:
          memory: "256Mi"
    - name: ambassador
      image: bmuschko/nodejs-ambassador:1.0.0
      ports:
        - containerPort: 8081
      resources:
        limits:
          memory: "64Mi"
```

Defining Container Resource Requests and Limits

To provide Kubernetes with the full picture of your application’s resource expectations, you must specify resource requests and limits for every container. **Example 18-3** combines resource requests and limits in a single YAML manifest.

Example 18-3. Setting container resource requests and limits

```
apiVersion: v1
kind: Pod
metadata:
  name: rate-limiter
spec:
  containers:
    - name: business-app
      image: bmuschko/nodejs-business-app:1.0.0
      ports:
```

```

- containerPort: 8080
resources:
  requests:
    memory: "256Mi"
    cpu: "1"
  limits:
    memory: "256Mi"
- name: ambassador
image: bmuschko/nodejs-ambassador:1.0.0
ports:
- containerPort: 8081
resources:
  requests:
    memory: "64Mi"
    cpu: "250m"
  limits:
    memory: "64Mi"

```

Assigning static container resource requirements is an approximation process. You want maximizing an efficient utilization of resources in your Kubernetes cluster. Unfortunately, the Kubernetes documentation does not offer a lot of guidance on best practices. The blog post “[For the Love of God, Stop Using CPU Limits on Kubernetes](#)” provides the following guidance:

- Always define memory requests.
- Always define memory limits.
- Always set your memory requests equal to your limit.
- Always define CPU requests.
- Do not use CPU limits.

After launching your application to production, you still need to monitor your application resource consumption patterns. Review resource consumption at runtime and keep track of actual scheduling behavior and potential undesired behaviors once the application receives load. Finding a happy medium can be frustrating. Projects like [Goldilocks](#) and [KRR](#) emerged to provide recommendations and guidance on appropriately determining resource requests. Other options, like the [container resize policies](#) introduced in Kubernetes 1.27, allow for more fine-grained control over how containers’ CPU and memory resources are resized automatically at runtime.

Working with Resource Quotas

The Kubernetes primitive ResourceQuota establishes the usable, maximum amount of resources per namespace. Once put in place, the Kubernetes scheduler will take care of enforcing those rules. The following list should give you an idea of the rules that can be defined:

- Setting an upper limit for the number of objects that can be created for a specific type (e.g., a maximum of three Pods).
- Limiting the total sum of compute resources (e.g., 3Gi of RAM).
- Expecting a Quality of Service (QoS) class for a Pod (e.g., `BestEffort` to indicate that the Pod must not make any memory or CPU limits or requests).

Creating ResourceQuotas

Creating a `ResourceQuota` object is usually a task that a Kubernetes administrator would take on, but it's relatively easy to define and create such an object. First, create the namespace the quota should apply to:

```
$ kubectl create namespace team-awesome
namespace/team-awesome created
```

Next, define the `ResourceQuota` in YAML. To demonstrate the functionality of a `ResourceQuota`, add constraints to the namespace, as shown in [Example 18-4](#).

Example 18-4. Defining hard resource limits with a `ResourceQuota`

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: awesome-quota
  namespace: team-awesome
spec:
  hard:
    pods: 2          ①
    requests.cpu: "1" ②
    requests.memory: 1024Mi ②
    limits.cpu: "4"   ③
    limits.memory: 4096Mi ③
```

- ➊ Limit the number of Pods to 2.
- ➋ Define the minimum resources requested across all Pods in a non-terminal state to 1 CPU and 1024Mi of RAM.
- ➌ Define the maximum resources used by all Pods in a non-terminal state to 4 CPUs and 4096Mi of RAM.

You're ready to create a `ResourceQuota` for the namespace:

```
$ kubectl create -f awesome-quota.yaml
resourcequota/awesome-quota created
```

Rendering ResourceQuota Details

You can render a table overview of used resources vs. hard limits using the `kubectl describe` command:

```
$ kubectl describe resourcequota awesome-quota -n team-awesome
Name:          awesome-quota
Namespace:     team-awesome
Resource       Used   Hard
limits.cpu    0      4
limits.memory 0      4Gi
pods          0      2
requests.cpu  0      1
requests.memory 0      1Gi
```

The Hard column lists the same values you provided with the ResourceQuota definition. Those values won't change also long as you don't modify the object's specification. Under the Used column, you can find the actual aggregate resource consumption within the namespace. At this time, all values are 0 given that no Pods have been created yet.

Exploring a ResourceQuota's Runtime Behavior

With the quota rules in place for the namespace `team-awesome`, we'll want to see its enforcement in action. We'll start by creating more than the maximum number of Pods, which is two. To test this, we can create Pods with any definition we like. For example, we use a bare-bones definition that runs the image `nginx:1.25.3` in the container, as shown in [Example 18-5](#).

Example 18-5. A Pod without resource requirements

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: team-awesome
spec:
  containers:
  - image: nginx:1.25.3
    name: nginx
```

From that YAML definition stored in `nginx-pod.yaml`, let's create a Pod and see what happens. In fact, Kubernetes will reject the creation of the object with the following error message:

```
$ kubectl apply -f nginx-pod.yaml
Error from server (Forbidden): error when creating "nginx-pod.yaml": \
pods "nginx" is forbidden: failed quota: awesome-quota: must specify \
limits.cpu for: nginx; limits.memory for: nginx; requests.cpu for: \
nginx; requests.memory for: nginx
```

Because we defined minimum and maximum resource quotas for objects in the namespace, we have to ensure that Pod objects actually define resource requests and limits. Modify the initial definition by updating the instruction under `resources`, as shown in [Example 18-6](#).

Example 18-6. A Pod with resource requirements

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: team-awesome
spec:
  containers:
  - image: nginx:1.25.3
    name: nginx
    resources:
      requests:
        cpu: "0.5"
        memory: "512Mi"
      limits:
        cpu: "1"
        memory: "1024Mi"
```

We should be able to create two uniquely named Pods—`nginx1` and `nginx2`—with that manifest; the combined resource requirements still fit with the boundaries defined in the ResourceQuota:

```
$ kubectl apply -f nginx-pod1.yaml
pod/nginx1 created
$ kubectl apply -f nginx-pod2.yaml
pod/nginx2 created
$ kubectl describe resourcequota awesome-quota -n team-awesome
Name:          awesome-quota
Namespace:     team-awesome
Resource       Used   Hard
-----  -----  -----
limits.cpu    2      4
limits.memory 2Gi   4Gi
pods          2      2
requests.cpu   1      1
requests.memory 1Gi  1Gi
```

You may be able to imagine what would happen if we tried to create another Pod with the definition of `nginx1` and `nginx2`. It will fail for two reasons. The first reason is that we're not allowed to create a third Pod in the namespace, as the maximum number is set to two. The second reason is that we'd exceed the allotted maximum for `requests.cpu` and `requests.memory`. The following error message provides us with this information:

```
$ kubectl apply -f nginx-pod3.yaml
Error from server (Forbidden): error when creating "nginx-pod3.yaml": \
pods "nginx3" is forbidden: exceeded quota: awesome-quota, requested: \
pods=1,requests.cpu=500m,requests.memory=512Mi, used: pods=2,requests.cpu=1, \
requests.memory=1Gi, limited: pods=2,requests.cpu=1,requests.memory=1Gi
```

Working with Limit Ranges

In the previous section, you learned how a resource quota can restrict the consumption of resources within a specific namespace in aggregate. For individual Pod objects, the resource quota cannot set any constraints. That's where the limit range comes in. The enforcement of `LimitRange` rules happens during the **admission control phase** when processing an API request.

The `LimitRange` is a Kubernetes primitive that constrains or defaults the resource allocations for specific object types:

- Enforces minimum and maximum compute resources usage per Pod or container in a namespace
- Enforces minimum and maximum storage request per `PersistentVolumeClaim` in a namespace
- Enforces a ratio between request and limit for a resource in a namespace
- Sets default requests/limits for compute resources in a namespace and automatically injects them into containers at runtime



Defining more than one `LimitRange` in a namespace

It is best to create only a single `LimitRange` object per namespace. Default resource requests and limits specified by multiple `LimitRange` objects in the same namespace causes non-deterministic selection of those rules. Only one of the default definitions will win, but you can't predict which one.

Creating LimitRanges

The LimitRange offers a list of configurable constraint attributes. All are described in great detail in the Kubernetes API documentation for a [LimitRangeSpec](#). [Example 18-7](#) shows a YAML manifest for a LimitRange that uses some of the constraint attributes.

Example 18-7. A limit range defining multiple constraint criteria

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-resource-constraint
spec:
  limits:
    - type: Container      ❶
      defaultRequest:   ❷
      cpu: 200m
      default:        ❸
      cpu: 200m
      min:            ❹
      cpu: 100m
      max:            ❹
      cpu: "2"
```

- ❶ The context to apply the constraints to. In this case, to a container running in a Pod.
- ❷ The default CPU resource request value assigned to a container if not provided.
- ❸ The default CPU resource limit value assigned to a container if not provided.
- ❹ The minimum and maximum CPU resource request and limit value assignable to a container.

As usual, we can create an object from the manifest with the `kubectl create` or `kubectl apply` command. The definition of the LimitRange has been stored in the file `cpu-resource-constraint-limitrange.yaml`:

```
$ kubectl apply -f cpu-resource-constraint.yaml
limitrange/cpu-resource-constraint created
```

The constraints will be applied automatically when creating new objects. Changing the constraints for an existing LimitRange object won't have any effect on already running Pods.

Rendering LimitRange Details

Live LimitRange objects can be inspected using the `kubectl describe` command. The following command renders the details of the LimitRange object named `cpu-resource-constraint`:

```
$ kubectl describe limitrange cpu-resource-constraint
Name:          cpu-resource-constraint
Namespace:    default
Type:         Resource
              Min   Max   Default Request  Default Limit ...
-----  -----  -----  -----  -----
Container  cpu      100m  2     200m           200m           ...
```

The output of the command renders each limit constraint on a single line. Any constraint attribute that has not been set explicitly by the object will show a dash character (-) as the assigned value.

Exploring a LimitRange's Runtime Behavior

Let's demonstrate what effect the LimitRange has on the creation of Pods. We will walk through two different use cases:

1. Automatically setting resource requirements if they have not been provided by the Pod definition.
2. Preventing the creation of a Pod if the declared resource requirements are forbidden by the LimitRange.

Setting default resource requirements

The LimitRange defines a default CPU resource request of 200m and a default CPU resource limit of 200m. That means if a Pod is about to be created, and it doesn't define a CPU resource request and limit, the LimitRange will automatically assign the default values.

Example 18-8 shows a Pod definition without resource requirements.

Example 18-8. A Pod defining no resource requirements

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-without-resource-requirements
spec:
  containers:
  - image: nginx:1.25.3
    name: nginx
```

Creating the object from the contents stored in the file `nginx-without-resource-requirements.yaml` will work as expected:

```
$ kubectl apply -f nginx-without-resource-requirements.yaml
pod/nginx-without-resource-requirements created
```

The Pod object will be mutated in two ways. First, the default resource requirements set by the LimitRange are applied. Second, an annotation with the key `kubernetes.io/limit-ranger` will be added that provides meta information on what has been changed. You can find both pieces of information in the output of the `describe` command:

```
$ kubectl describe pod nginx-without-resource-requirements
...
Annotations:      kubernetes.io/limit-ranger: LimitRanger plugin set: cpu \
request for container nginx; cpu limit for container nginx
...
Containers:
  nginx:
    ...
    Limits:
      cpu: 200m
    Requests:
      cpu: 200m
    ...

```

Enforcing resource requirements

The LimitRange can enforce resource limits as well. For the LimitRange object we created earlier, the minimum amount of CPU was set to 100m, and the maximum amount of CPU was set to 2. To see the enforcement behavior in action, we'll create a new Pod as shown in [Example 18-9](#).

Example 18-9. A Pod defining CPU resource requests and limits

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-resource-requirements
spec:
  containers:
    - image: nginx:1.25.3
      name: nginx
      resources:
        requests:
          cpu: "50m"
        limits:
          cpu: "3"
```

The resource requirements of this Pod do not follow the constraints expected by the LimitRange object. The CPU resource request is less than 100m, and the CPU resource limit is higher than 2. As a result, the object won't be created and an appropriate error message will be rendered:

```
$ kubectl apply -f nginx-with-resource-requirements.yaml
Error from server (Forbidden): error when creating "nginx-with-resource-\
requirements.yaml": pods "nginx-with-resource-requirements" is forbidden: \
[minimum cpu usage per Container is 100 m, but request is 50 m, maximum cpu \
usage per Container is 2, but limit is 3]
```

The error message provides some guidance on expected resource definitions. Unfortunately, the message doesn't point to the name of the LimitRange object enforcing those expectations. Proactively check if a LimitRange object has been created for the namespace and what parameters have been set using `kubectl get limit ranges`.

Summary

Resource requests are one of the many factors that the kube-scheduler algorithm considers when making decisions on which node a Pod can be scheduled. A container can specify requests using `spec.containers[].resources.requests`. The scheduler chooses a node based on its available hardware capacity. The resource limits ensure that the container cannot consume more than the allotted resource amounts. Limits can be defined for a container using the attribute `spec.containers [].resources.limits`. Should an application consume more than the allowed amount of resources (e.g., due to a memory leak in the implementation), the container runtime will likely terminate the application process.

A resource quota defines the computing resources (e.g., CPU, RAM, and ephemeral storage) available to a namespace to prevent unbounded consumption by Pods running it. Accordingly, Pods have to work within those resource boundaries by declaring their minimum and maximum resource expectations. You can also limit the number of resource types (like Pods, Secrets, or ConfigMaps) that are allowed to be created. The Kubernetes scheduler will enforce those boundaries upon a request for object creation.

The limit range is different from the ResourceQuota in that it defines resource constraints for a single object of a specific type. It can also help with governance for objects by specifying resource default values that should be applied automatically should the API create request not provide the information.

Exam Essentials

Experience the effects of resource requirements on scheduling and autoscaling

A container defined by a Pod can specify resource requests and limits. Work through scenarios where you define those requirements individually and together for single- and multi-container Pods. Upon creation of the Pod, you should be able to see the effects on scheduling the object on a node. Furthermore, practice how to identify the available resource capacity of a node.

Understand the purpose and runtime effects of resource quotas

A ResourceQuota defines the resource boundaries for objects living within a namespace. The most commonly used boundaries apply to computing resources. Practice defining them and understand their effect on the creation of Pods. It's important to know the command for listing the hard requirements of a ResourceQuota and the resources currently in use. You will find that a ResourceQuota offers other options. Discover them in more detail for a broader exposure to the topic.

Understand the purpose and runtime effects of limit ranges

A LimitRange can specify resource constraints and defaults of specific primitives. Should you run into a situation where you receive an error message upon creation of an object, check if a limit range object enforces those constraints. Unfortunately, the error message does not point out the object that enforces it so you may have to proactively list LimitRange objects to identify the constraints.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. You have been tasked with creating a Pod for running an application in a container. During application development, you ran a load test for figuring out the minimum amount of resources needed and the maximum amount of resources the application is allowed to grow to. Define those resource requests and limits for the Pod.

Define a Pod named `hello-world` running the container image `bmuschko/nodejs-hello-world:1.0.0`. The container exposes the port 3000.

Add a Volume of type `emptyDir` and mount it in the container path `/var/log`.

For the container, specify the following minimum number of resources:

- CPU: 100m
- Memory: 500Mi
- Ephemeral storage: 1Gi

For the container, specify the following maximum number of resources:

- Memory: 500Mi
- Ephemeral storage: 2Gi

Create the Pod from the YAML manifest. Inspect the Pod details. Which node does the Pod run on?

2. In this exercise, you will create a resource quota with specific CPU and memory limits for a new namespace. Pods created in the namespace will have to adhere to those limits.

Create a ResourceQuota named `app` under the namespace `rq-demo` using the following YAML definition in the file `resourcequota.yaml`:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: app
spec:
  hard:
    pods: "2"
    requests.cpu: "2"
    requests.memory: 500Mi
```

Create a new Pod that exceeds the limits of the resource quota requirements, e.g., by defining 1Gi of memory but stays below the CPU, e.g., 0.5. Write down the error message.

Change the request limits to fulfill the requirements to ensure that the Pod can be created successfully. Write down the output of the command that renders the used amount of resources for the namespace.

3. A LimitRange can restrict resource consumption for Pods in a namespace, and assign default computing resources if no resource requirements have been defined. You will practice the effects of a LimitRange on the creation of a Pod in different scenarios.

Navigate to the directory `app-a/ch18/limitrange` of the checked-out GitHub repository [bmuscko/ckad-study-guide](#). Inspect the YAML manifest definition in the file `setup.yaml`. Create the objects from the YAML manifest file.

Create a new Pod named `pod-without-resource-requirements` in the namespace `d92` that uses the container image `nginx:1.23.4-alpine` without any resource requirements. Inspect the Pod details. What resource definitions do you expect to be assigned?

Create a new Pod named `pod-with-more-cpu-resource-requirements` in the namespace `d92` that uses the container image `nginx:1.23.4-alpine` with a CPU resource request of 400m and limits of 1.5. What runtime behavior do you expect to see?

Create a new Pod named `pod-with-less-cpu-resource-requirements` in the namespace `d92` that uses the container image `nginx:1.23.4-alpine` with a CPU resource request of `350m` and limits of `400m`. What runtime behavior do you expect to see?

ConfigMaps and Secrets

Kubernetes dedicates two primitives to defining configuration data: the ConfigMap and the Secret. Both primitives are completely decoupled from the life cycle of a Pod, which enables you to change their configuration data values without necessarily having to redeploy the Pod.

In essence, ConfigMaps and Secrets store a set of key-value pairs. Those key-value pairs can be injected into a container as environment variables, or they can be mounted as a Volume. [Figure 19-1](#) illustrates the options.

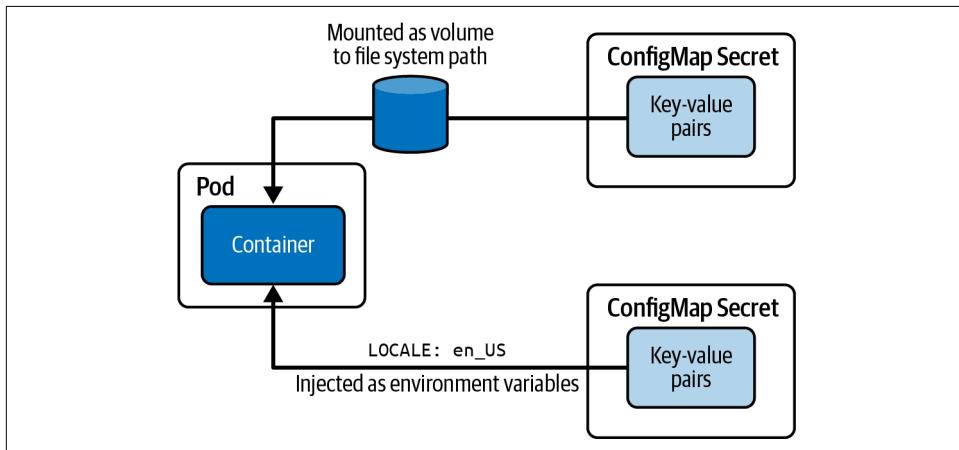


Figure 19-1. Consuming configuration data

The ConfigMap and Secret may look almost identical in purpose and structure on the surface; however, there is a slight but significant difference. A ConfigMap stores plain-text data, for example connection URLs, runtime flags, or even structured data like a JSON or YAML content. Secrets are better suited for representing sensitive data

like passwords, API keys, or SSL certificates and store the data in base64-encoded form.



Encryption of ConfigMap and Secret data

The cluster component that stores data of a ConfigMap and Secret object is etcd. Etcd manages this data in unencrypted form by default. You can configure encryption of data in etcd, as described in the [Kubernetes documentation](#). Etcd encryption is not within the scope of the exam.

This chapter references the concept of Volumes heavily. Refer to [Chapter 7](#) to refresh your memory on the mechanics of consuming a Volume in a Pod.

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objectives:

- Understand ConfigMaps
- Create and consume Secrets

Working with ConfigMaps

Applications often implement logic that uses configuration data to control runtime behavior. Examples for configuration data include a connection URL and network communication options (like the number of retries or timeouts) to third-party services that differ between target deployment environments.

It's not unusual that the same configuration data needs to be made available to multiple Pods. Instead of copy-pasting the same key-value pairs across multiple Pod definitions, you can choose to centralize the information in a ConfigMap object. The ConfigMap object holds configuration data and can be consumed by as many Pods as you want. Therefore, you will need to modify the data in only one location should you need to change it.

Creating a ConfigMap

You can create a ConfigMap by emitting the imperative `create configmap` command. This command requires you to provide the source of the data as an option. Kubernetes distinguishes the four different options shown in [Table 19-1](#).

Table 19-1. Source options for data parsed by a ConfigMap

Option	Example	Description
--from-literal	--from-literal=locale=en_US	Literal values, which are key-value pairs as plain text
--from-env-file	--from-env-file=config.env	A file that contains key-value pairs and expects them to be environment variables
--from-file	--from-file=app-config.json	A file with arbitrary contents
--from-dir	--from-dir=config-dir	A directory with one or many files

It's easy to confuse the options `--from-env-file` and `--from-file`. The option `--from-env-file` expects a file that contains environment variables in the format `KEY=value` separated by a new line. The key-value pairs follow typical naming conventions for environment variables (e.g., the key is uppercase, and individual words are separated by an underscore character). Historically, this option has been used to process [Docker Compose .env file](#), though you can use it for any other file containing environment variables.

The `--from-env-file` option does not enforce or normalize the typical naming conventions for environment variables. The option `--from-file` points to a file or directory containing *any* arbitrary content. It's an appropriate option for files with structured configuration data to be read by an application (e.g., a properties file, a JSON file, or an XML file).

The following command shows the creation of a ConfigMap in action. We are simply providing the key-value pairs as literals:

```
$ kubectl create configmap db-config --from-literal=DB_HOST=mysql-service \
  --from-literal=DB_USER=backend
configmap/db-config created
```

The resulting YAML object looks like the one shown in [Example 19-1](#). As you can see, the object defines the key-value pairs in a section named `data`. A ConfigMap does not have a `spec` section.

Example 19-1. ConfigMap YAML manifest

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  DB_HOST: mysql-service
  DB_USER: backend
```

You may have noticed that the key assigned to the ConfigMap data follows the typical naming conventions used by environment variables. The intention is to consume them as such in a container.

Consuming a ConfigMap as Environment Variables

With the ConfigMap created, you can now inject its key-value pairs as environment variables into a container. [Example 19-2](#) shows the use of `spec.containers[].envFrom[] .configMapRef` to reference the ConfigMap by name.

Example 19-2. Injecting ConfigMap key-value pairs into the container

```
apiVersion: v1
kind: Pod
metadata:
  name: backend
spec:
  containers:
    - image: bmuschko/web-app:1.0.1
      name: backend
      envFrom:
        - configMapRef:
            name: db-config
```

After creating the Pod from the YAML manifest, you can inspect the environment variables available in the container by running the `env` Unix command:

```
$ kubectl exec backend -- env
...
DB_HOST=mysql-service
DB_USER=backend
...
```

The injected configuration data will be listed among environment variables available to the container.

Mounting a ConfigMap as a Volume

Another way to configure applications at runtime is by processing a machine-readable configuration file. Say we have decided to store the database configuration in a JSON file named `db.json` with the structure shown in [Example 19-3](#).

Example 19-3. A JSON file used for configuring database information

```
{
  "db": {
    "host": "mysql-service",
    "user": "backend"
```

```
    }
}
```

Given that we are not dealing with literal key-value pairs, we need to provide the option `--from-file` when creating the ConfigMap object:

```
$ kubectl create configmap db-config --from-file=db.json
configmap/db-config created
```

[Example 19-4](#) shows the corresponding YAML manifest of the ConfigMap. You can see that the file name becomes the key; the contents of the file has used a multiline value.

Example 19-4. ConfigMap YAML manifest defining structured data

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  db.json: |-
    {
      "db": {
        "host": "mysql-service",
        "user": "backend"
      }
    }
```

①

- ① The multiline string syntax (`| -`) used in this YAML structure removes the line feed and removes the trailing blank lines. For more information, see the [YAML syntax for multiline string](#).

The Pod mounts the ConfigMap as a volume to a specific path inside of the container with read-only permissions. The assumption is that the application will read the configuration file when starting up. [Example 19-5](#) demonstrates the YAML definition.

Example 19-5. Mounting a ConfigMap as a volume

```
apiVersion: v1
kind: Pod
metadata:
  name: backend
spec:
  containers:
    - image: bmuschko/web-app:1.0.1
      name: backend
    volumeMounts:
      - name: db-config-volume
        mountPath: /etc/config
```

```
volumes:  
- name: db-config-volume  
  configMap:  
    name: db-config
```

- ① Assign the volume type for referencing a ConfigMap object by name.

To verify the correct behavior, open an interactive shell to the container. As you can see in the following commands, the directory `/etc/config` contains a file with the key we used in the ConfigMap. The content represents the JSON configuration:

```
$ kubectl exec -it backend -- /bin/sh  
# ls -1 /etc/config  
db.json  
# cat /etc/config/db.json  
{  
  "db": {  
    "host": "mysql-service",  
    "user": "backend"  
  }  
}
```

The application code can now read the file from the mount path and configure the runtime behavior as needed.

Working with Secrets

Data stored in ConfigMaps represent arbitrary plain-text key-value pairs. In comparison to the ConfigMap, the Secret primitive is meant to represent sensitive configuration data. A typical example for Secret data is a password or an API key for authentication.



Values stored in a Secret are only encoded, not encrypted

Secrets expect the value of each entry to be Base64-encoded. Base64 encodes only a value, but it doesn't encrypt it. Therefore, anyone with access to its value can decode it without problems. Therefore, storing Secret manifests in the source code repository alongside other resource files should be avoided.

It's somewhat unfortunate that the Kubernetes project decided to choose the term "Secret" to represent sensitive data. The nomenclature implies that data is actually secret and therefore encrypted. You can select from a range of options to keep sensitive data secure in real-world projects.

Bitnami Sealed Secrets is an production-ready and proven Kubernetes operator that uses asymmetric crypto encryption for data. The manifest representation of the data, the CRD SealedSecret, is safe to be stored in a public source code repository. You

cannot decrypt this data yourself. The controller installed with the operator is the only entity that can decrypt the data. Another option is to store sensitive data in external secrets managers, e.g., HashiCorp Vault or AWS Secrets Manager, and integrate them with Kubernetes. The [External Secrets Operator](#) synchronizes secrets from external APIs into Kubernetes. The exam only expects you to understand the built-in Secret primitive, covered in the following sections.

Creating a Secret

You can create a Secret with the imperative command `create secret`. In addition, a mandatory subcommand needs to be provided that determines the type of Secret. [Table 19-2](#) lists the different types. Kubernetes assigns the value in the Internal Type column to the `type` attribute in the live object. [“Specialized Secret types” on page 231](#) discusses other Secret types and their use cases.

Table 19-2. Options for creating a Secret

CLI option	Description	Internal Type
<code>generic</code>	Creates a secret from a file, directory, or literal value	Opaque
<code>docker-registry</code>	Creates a secret for use with a Docker registry, e.g., to pull images from a private registry when requested by a Pod	<code>kubernetes.io/dockercfg</code>
<code>tls</code>	Creates a TLS secret	<code>kubernetes.io/tls</code>

The most commonly used Secret type is `generic`. The options for a generic Secret are exactly the same as for a ConfigMap, as shown in [Table 19-3](#).

Table 19-3. Source options for data parsed by a Secret

Option	Example	Description
<code>--from-literal</code>	<code>--from-literal=password=secret</code>	Literal values, which are key-value pairs as plain text
<code>--from-env-file</code>	<code>--from-env-file=config.env</code>	A file that contains key-value pairs and expects them to be environment variables
<code>--from-file</code>	<code>--from-file=id_rsa=~/.ssh/id_rsa</code>	A file with arbitrary contents
<code>--from-file</code>	<code>--from-file=config-dir</code>	A directory with one or many files

To demonstrate the functionality, let’s create a Secret of type `generic`. The command sources the key-value pairs from the literals provided as a command-line option:

```
$ kubectl create secret generic db-creds --from-literal=pwd=s3cre!  
secret/db-creds created
```

When created using the imperative command, a Secret will automatically Base64-encode the provided value. This can be observed by looking at the produced YAML

manifest. You can see in [Example 19-6](#) that the value `s3cre!` has been turned into `czNjcmUh`, the Base64-encoded equivalent.

Example 19-6. A Secret with Base64-encoded values

```
apiVersion: v1
kind: Secret
metadata:
  name: db-creds
type: Opaque      ①
data:
  pwd: czNjcmUh  ②
```

- ① The value `Opaque` for the type has been assigned to represent generic sensitive data.
- ② The plain-text value has been Base64-encoded automatically if the object has been created imperatively.

If you start with the YAML manifest to create the Secret object, you will need to create the Base64-encoded value if you want to assign it to the `data` attribute. A Unix tool that does the job is `base64`. The following command achieves exactly that:

```
$ echo -n 's3cre!' | base64
czNjcmUh
```

As a reminder, if you have access to a Secret object or its YAML manifest then you can decode the Base64-encoded value at any time with the `base64` Unix tool. Therefore, you may as well specify the value in plain-text when defining the manifest, which we'll discuss in the next section.

Defining Secret data with plain-text values

Having to generate and assign Base64-encoded values to Secret manifests can become cumbersome. The Secret primitive offers the `stringData` attribute as a replacement for the `data` attribute. With `stringData`, you can assign plain-text values in the manifest file, as shown in [Example 19-7](#).

Example 19-7. A Secret with plain-text values

```
apiVersion: v1
kind: Secret
metadata:
  name: db-creds
type: Opaque
stringData:      ①
  pwd: s3cre!    ②
```

- ❶ The `stringData` attribute allows assigning plain-text key-value pairs.
- ❷ The value referenced by the `pwd` key was provided in plain-text format.

Kubernetes will automatically Base64-encode the `s3cre!` value upon creation of the object from the manifest. The result is the live object representation shown in [Example 19-8](#), which you can retrieve with the command `kubectl get secret db-creds -o yaml`.

Example 19-8. A Secret live object

```
apiVersion: v1
kind: Secret
metadata:
  name: db-creds
type: Opaque
data:           ❶
  pwd: czNjcmUh! ❷
```

- ❶ The live object of a Secret always uses the `data` attribute even though you may have used `stringData` in the manifest.
- ❷ The value has been Base64-encoded upon creation.

You can represent arbitrary Secret data using the `Opaque` type. Kubernetes offers specialized Secret types you can choose from should the data fit specific use cases. We'll discuss those specialized Secret types in the next section.

Specialized Secret types

Instead of using the `Opaque` Secret type, you can also use one of the [specialized types](#) to represent configuration data for particular use cases. The type `kubernetes.io/basic-auth` is meant for basic authentication and expects the keys `username` and `password`. At the time of writing, Kubernetes does not validate the correctness of the assigned keys.

The created object from this definition automatically Base64-encodes the values for both keys. [Example 19-9](#) illustrates a YAML manifest for a Secret with type `kubernetes.io/basic-auth`.

Example 19-9. Usage of the Secret type `kubernetes.io/basic-auth`

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-basic-auth
```

```
type: kubernetes.io/basic-auth
stringData:
  username: bmuschko
  password: secret
```

- ➊ Uses the `stringData` attribute to allow for assigning plain-text values.
- ➋ Specifies the mandatory keys required by the `kubernetes.io/basic-auth` Secret type.

Consuming a Secret as Environment Variables

Consuming a Secret as environment variables works similar to the way you'd do it for ConfigMaps. Here, you'd use the YAML expression `spec.containers[].envFrom[] .secretRef` to reference the name of the Secret. [Example 19-10](#) injects the Secret named `secret-basic-auth` as environment variables into the container named `backend`.

Example 19-10. Injecting Secret key-value pairs into the container

```
apiVersion: v1
kind: Pod
metadata:
  name: backend
spec:
  containers:
    - image: bmuschko/web-app:1.0.1
      name: backend
      envFrom:
        - secretRef:
            name: secret-basic-auth
```

Inspecting the environment variables in the container reveals that the Secret values do not have to be decoded. That's something Kubernetes does automatically. Therefore, the running application doesn't need to implement custom logic to decode the value. Note that Kubernetes does not verify or normalize the typical naming conventions of environment variables, as you can see in the following output:

```
$ kubectl exec backend -- env
...
username=bmuschko
password=secret
...
```

Remapping environment variable keys

Sometimes, key-value pairs stored in a Secret do not conform to typical naming conventions for environment variables or can't be changed without impacting running services. You can redefine the keys used to inject an environment variable into a Pod with the `spec.containers[].env[].valueFrom` attribute. [Example 19-11](#) turns the key `username` into `USER` and the key `password` into `PWD`.

Example 19-11. Remapping environment variable keys for Secret entries

```
apiVersion: v1
kind: Pod
metadata:
  name: backend
spec:
  containers:
    - image: bmuschko/web-app:1.0.1
      name: backend
      env:
        - name: USER
          valueFrom:
            secretKeyRef:
              name: secret-basic-auth
              key: username
        - name: PWD
          valueFrom:
            secretKeyRef:
              name: secret-basic-auth
              key: password
```

The resulting environment variables available to the container now follow the typical conventions for environment variables, and we changed how their are consumed by the application code:

```
$ kubectl exec backend -- env
...
USER=bmuschko
PWD=secret
...
```

The same mechanism of reassigning environment variables works for ConfigMaps. You'd use the attribute `spec.containers[].env[].valueFrom.configMapRef` instead.

Mounting a Secret as a Volume

To demonstrate mounting a Secret as a volume, we'll create a new Secret of type `kubernetes.io/ssh-auth`. This Secret type captures the value of an SSH private key that you can view using the command `cat ~/.ssh/id_rsa`. To process the SSH

private key file with the `create secret` command, it needs to be available as a file with the name `ssh-privatekey`:

```
$ cp ~/.ssh/id_rsa ssh-privatekey
$ kubectl create secret generic secret-ssh-auth --from-file=ssh-privatekey \
--type=kubernetes.io/ssh-auth
secret/secret-ssh-auth created
```

Mounting the Secret as a volume follows the two-step approach: define the volume first and then reference it as a mount path for one or many containers. The volume type is called `secret` as used in [Example 19-12](#).

Example 19-12. Mounting a Secret as a volume

```
apiVersion: v1
kind: Pod
metadata:
  name: backend
spec:
  containers:
    - image: bmuschko/web-app:1.0.1
      name: backend
      volumeMounts:
        - name: ssh-volume
          mountPath: /var/app
          readOnly: true      ①
  volumes:
    - name: ssh-volume
      secret:
        secretName: secret-ssh-auth ②
```

- ① Files provided by the Secret mounted as volume cannot be modified.
- ② Note that the attribute `secretName` that points to the Secret name is not the same as for the ConfigMap (which is `name`).

You will find the file named `ssh-privatekey` in the mount path `/var/app`. To verify, open an interactive shell and render the file contents. The contents of the file are not Base64-encoded:

```
$ kubectl exec -it backend -- /bin/sh
# ls -1 /var/app
ssh-privatekey
# cat /var/app/ssh-privatekey
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-128-CBC,8734C9153079F2E8497C8075289EBBF1
...
-----END RSA PRIVATE KEY-----
```

Summary

Application runtime behavior can be controlled either by injecting configuration data as environment variables or by mounting a volume to a path. In Kubernetes, this configuration data is represented by the API resources ConfigMap and Secret in the form of key-value pairs. A ConfigMap is meant for plain-text data, and a Secret encodes the values in Base64 to obfuscate the values. Secrets are a better fit for sensitive information like credentials and SSH private keys.

Exam Essentials

Practice creating ConfigMap objects with the imperative and declarative approach

The quickest ways to create those objects are the imperative `kubectl create configmap` commands. Understand how to provide the data with the help of different command line flags. The ConfigMap specifies plain-text key-value pairs in the `data` section of YAML manifest.

Practice creating Secret objects with the imperative and declarative approach

Creating a Secret using the imperative command `kubectl create secret` does not require you to Base64-encode the provided values. `kubectl` performs the encoding operation automatically. The declarative approach requires the Secret YAML manifest to specify a Base64-encoded value with the `data` section. You can use the `stringData` convenience attribute in place of the `data` attribute if you prefer providing a plain-text value. The live object will use a Base64-encoded value. Functionally, there's no difference at runtime between the use of `data` and `stringData`.

Understand the purpose of specialized Secret types

Secrets offer specialized types, e.g., `kubernetes.io/basic-auth` or `kubernetes.io/service-account-token`, to represent data for specific use cases. Read up on the different types in the Kubernetes documentation and understand their purpose.

Know how to inspect ConfigMap and Secret data

The exam may confront you with existing ConfigMap and Secret objects. You need to understand how to use the `kubectl get` or the `kubectl describe` command to inspect the data of those objects. The live object of a Secret will always represent the value in a Base64-encoded format.

Exercise the consumption of ConfigMaps and Secrets in Pods

The primary use case for ConfigMaps and Secrets is the consumption of the data from a Pod. Pods can inject configuration data into a container as environment variables or mount the configuration data as Volumes. For the exam, you need to be familiar with both consumption methods.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. In this exercise, you will first create a ConfigMap from a YAML configuration file as a source. Later, you'll create a Pod, consume the ConfigMap as Volume, and inspect the key-value pairs as files.

Navigate to the directory `app-a/ch19/configmap` of the checked-out GitHub repository [bmuschko/ckad-study-guide](#). Inspect the YAML configuration file named `application.yaml`.

Create a new ConfigMap named `app-config` from that file.

Create a Pod named `backend` that consumes the ConfigMap as Volume at the mount path `/etc/config`. The container runs the image `nginx:1.23.4-alpine`.

Shell into the Pod and inspect the file at the mounted Volume path.

2. You will first create a Secret from literal values in this exercise. Next, you'll create a Pod and consume the Secret as environment variables. Finally, you'll print out its values from within the container.

Create a new Secret named `db-credentials` with the key/value pair `db-password=passwd`.

Create a Pod named `backend` that uses the Secret as an environment variable named `DB_PASSWORD` and runs the container with the image `nginx:1.23.4-alpine`.

Shell into the Pod and print out the created environment variables. You should be able to find the `DB_PASSWORD` variable.

Security Contexts

Running a Pod in Kubernetes without implementing more restrictive security measures can pose a security risk. Without these measures, an attacker can potentially gain access to the host system or perform malicious activities, such as accessing files containing sensitive data. A security context defines privilege and access control settings for containers as part of a Pod specification. The following list provides some examples for security-related parameters:

- The user ID that should be used to run the Pod and/or container
- The group ID that should be used for filesystem access
- Granting a running process inside the container some privileges of the root user but not all of them

This chapter will give you an overview of defining security contexts and seeing their runtime effects in practice. Given the wide range of security settings, we won't be able to discuss all of them. You will find additional use cases and configuration options in the [Kubernetes documentation](#).

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objective:

- Understand SecurityContext

Working with Security Contexts

The security context is not a Kubernetes primitive. It is modeled as a set of attributes under the directive `securityContext` within the Pod specification. Security settings defined on the Pod level apply to all containers running in the Pod. When applied to a single container, it will have no effects on other containers running in the same Pod.

- You can apply security settings to all containers of a Pod with the attribute `spec.securityContext`.
- For individual containers, you can apply security settings with the attribute `spec.containers[].securityContext`.

Figure 20-1 shows the use of security settings `runAsUser` and `fsGroup` applied on the Pod and container level. A later section will describe the runtime effect of those settings by example.

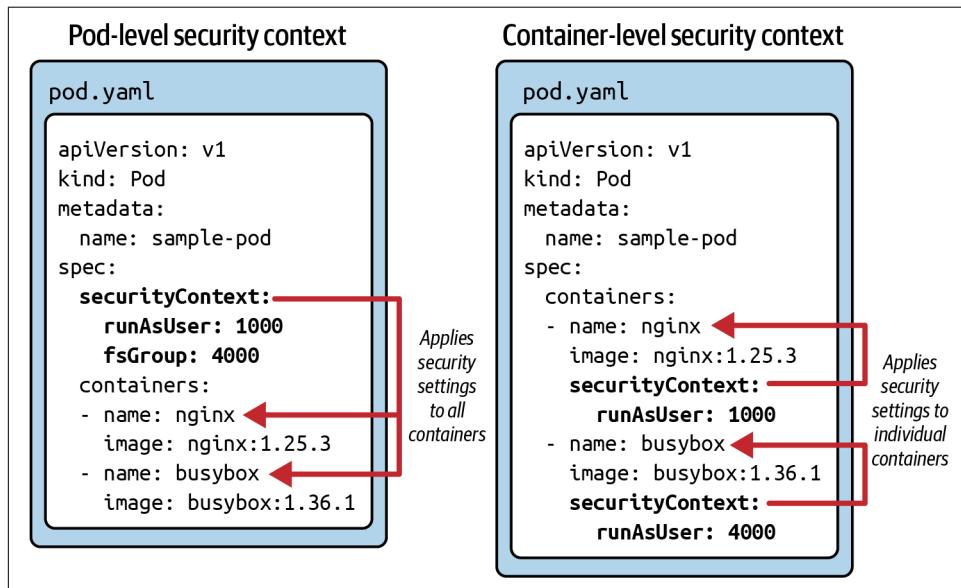


Figure 20-1. Applying security settings on the Pod and container level

Some security context attributes are available on the Pod *and* the container level. If you define the same security context on both levels then the value assigned on the container level will take precedence. Figure 20-2 shows an override of a container-level security context value for the container named `nginx`. For that container, the value `2000` will apply.

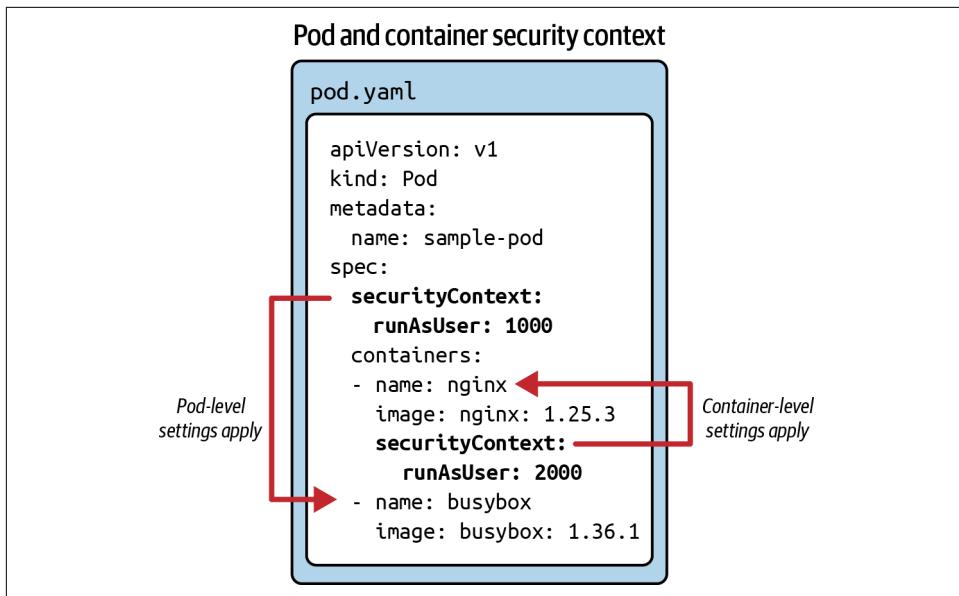


Figure 20-2. Overriding a security context attribute on the container level

For more information on the specifics of Pod-level security attributes, see the [PodSecurityContext](#) API. Container-level security attributes can be found in the [Security-Context](#) API.



Applying a security context to a Deployment's Pod template

The definition of a Deployment applies security context attributes in the same way as a vanilla Pod definition. You'd use the same Pod- and container-level attributes in the Pod template section of the Deployment.

Defining security-related parameters with the security context is always limited to a container. The Kubernetes ecosystem offers other ways to improve or govern security for applications, some of which directly tie in with the security context concept. For example, you can use the [Pod Security Admission](#) to enforce desired security settings for all Pods within a namespace.

Defining a Security Context on the Pod Level

Container images can define security-relevant instructions to reduce the attack vector for the running container. By default, containers run with root privileges, which provide supreme access to all processes and the container's filesystem. As a best practice, you should craft the corresponding `Dockerfile` in a such a way that the container

will be run with a user ID other than 0 with the help of the `USER` instruction. There are many other ways to secure a container on the container level, but we won't go into any more detail here. Refer to *Container Security* by Liz Rice (O'Reilly) for more information.

To make the functionality of a security-context more transparent, let's look at a use case. Some images, like the one for the open source reverse-proxy server **NGINX**, must be run with the root user. Say you wanted to enforce that containers cannot be run as a root user as a sensible security strategy. The YAML manifest shown in [Example 20-1](#) defines the security configuration on the Pod level as a direct child of the `spec` attribute. If you were to run other containers inside the Pod, then the `runAsNonRoot` setting apply to them as well.

Example 20-1. Setting a security context on the container level for the NGINX image

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-non-root
spec:
  securityContext:
    runAsNonRoot: true      ①
  containers:
  - image: nginx:1.25.3
    name: secured-container
```

- ① Enforces the use of a non-root user to run the container.

Creating the Pod from this manifest will work as expected:

```
$ kubectl apply -f container-nginx-root-user.yaml
pod/nginx-non-root created
```

Unfortunately, the image is not compatible. The container fails during the startup process with the status `CreateContainerConfigError`:

```
$ kubectl get pod nginx-non-root
NAME          READY   STATUS           RESTARTS   AGE
nginx-non-root  0/1    CreateContainerConfigError   0          7s
```

You will find the root cause for this issue in the event logs:

```
$ kubectl describe pod nginx-non-root
...
Events:
Type  Reason  Age           From           Message
----  -----  --           ----           -----
Normal  Scheduled  <unknown>  default-scheduler  Successfully assigned \
                                         default/non-root to \
                                         minikube
```

```

Normal  Pulling   18s      kubelet, minikube  Pulling image \
"nginx:1.25.3"
Normal  Pulled    14s      kubelet, minikube  Successfully pulled \
image "nginx:1.25.3"
Warning Failed    0s (x3 over 14s) kubelet, minikube  Error: container has \
runAsNonRoot and image \
will run as root

```

There are alternative NGINX images available that are not required to run with the root user. One example is [bitnami/nginx](#). Upon a closer look at the Dockerfile that produced the image, you will see that the container runs with the user ID 1001. [Example 20-2](#) shows the use of the Bitnami image.

Example 20-2. Setting a security context on the container level for the Bitnami NGIX image

```

apiVersion: v1
kind: Pod
metadata:
  name: bitnami-nginx-non-root
spec:
  securityContext:
    runAsNonRoot: true
  containers:
  - image: bitnami/nginx:1.25.3
    name: secured-container

```

Starting the container with the `runAsNonRoot` directive will work just fine:

```
$ kubectl apply -f container-bitnami-nginx-root-user.yaml
pod/bitnami-nginx-non-root created
```

The container will indicate the “Running” status:

```
$ kubectl get pod nginx-non-root
NAME                  READY   STATUS    RESTARTS   AGE
bitnami-nginx-non-root 1/1     Running   0          7s
```

The container could be executed with the user ID set by the container image that you easily surface by running the following command inside the container:

```
$ kubectl exec -it bitnami-nginx-non-root -- id -u
1001
```

The output of the command renders the user ID 1001, a non-root user ID.

Defining a Security Context on the Container Level

You can impose many other security restrictions on a container running in Kubernetes. For example, you may want to set the access control for files and directories. Say that, whenever a file is created on the filesystem, the owner of the file should be

the arbitrary group ID 3500. The YAML manifest shown in [Example 20-3](#) assigns the security context settings on the container level.

Example 20-3. Setting a security context on the container level

```
apiVersion: v1
kind: Pod
metadata:
  name: fs-secured
spec:
  containers:
    - image: nginx:1.25.3
      name: secured-container
      securityContext:
        fsGroup: 3500
      volumeMounts:
        - name: data-volume
          mountPath: /data/app
    volumes:
      - name: data-volume
        emptyDir: {}
```

Create the Pod object from the manifest file and inspect the status. The Pod should transition into the “Running” status:

```
$ kubectl apply -f pod-file-system-group.yaml
pod/fs-secured created
$ kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
fs-secured  1/1     Running   0          24s
```

You can easily verify the effect of setting the filesystem group ID. Open an interactive shell to the container, navigate to the mounted volume, and create a new file:

```
$ kubectl exec -it fs-secured -- /bin/sh
# cd /data/app
# touch logs.txt
# ls -l
-rw-r--r-- 1 root 3500 0 Jul  9 01:41 logs.txt
```

Inspecting the ownership of the file will show the group ID 3500 automatically assigned to it.

Defining a Security Context on the Pod and Container Level

Finally, let’s demonstrate the override behavior on the container level if you already defined the same attribute on the Pod level. [Example 20-4](#) shows the definition of the `runAsNonRoot` on both levels.

Example 20-4. Setting a security context on the Pod and container level

```
apiVersion: v1
kind: Pod
metadata:
  name: non-root-user-override
spec:
  securityContext:
    runAsNonRoot: true          ①
  containers:
  - image: nginx:1.25.3
    name: root
    securityContext:
      runAsNonRoot: false       ②
  - image: bitnami/nginx:1.25.3
    name: non-root
```

- ① Assign the default value `true` to all containers of the Pod.
- ② The value `false` will take precedence even though `true` has been assigned on the Pod level.

Create the Pod object from the manifest file:

```
$ kubectl apply -f pod-non-root-user-override.yaml
pod/non-root-user-override created
```

Open an interactive shell to the container and execute the command for rendering the user ID that runs the container:

```
$ kubectl exec -it -c root non-root-user-override -- id -u
0
$ kubectl exec -it -c non-root non-root-user-override -- id -u
1001
```

The container `root` returns the value 0, which is the user ID for the root user. The container `non-root` returns the user ID 1001, the ID set by the container image itself.

Summary

It's important to enforce security best practices for Pods. This chapter covered the security context concept. With the help of a security context, you can control container permissions to access objects such as files, run a container in privileged and unprivileged mode, specify Linux capabilities, and much more.

The security context can be declared on a Pod level and container level. The Pod level applies the provided security settings to all containers in the Pod. The container level applies only to individual containers. The container-level security settings override the Pod-level security settings if the same attribute value is specified on both levels.

Exam Essentials

Experiment with options available to security contexts

The Kubernetes user documentation and API documentation is a good starting point for exploring security context options. You will find that there's an overlap in the options available via the PodSecurityContext and a SecurityContext APIs. While working through the different use cases solved by a security context option, verify their outcome by running an operation that should either be permitted or disallowed.

Understand the implications of defining a security context on the Pod and container level

You can define a security context on the Pod level with `spec.securityContext`, and on the container level with `spec.containers[].securityContext`. If defined on the Pod level, settings can be overridden by specifying them with a different value on the container level. The exam may confront you with existing Pods that set a security context on both levels. Understand which value will take effect.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. Define a Pod named `busybox-security-context` that uses the image `busybox:1.36.1` for a single container running the command `sh -c sleep 1h`.

Add an ephemeral Volume of type `emptyDir`. Mount the Volume to the container at `/data/test`.

Define a security context that runs the container with user ID 1000, with group ID 3000, and the filesystem group ID 2000. Ensure that the container should not allow privilege escalation.

Create the Pod object and ensure that it transitions into the "Running" status.

Open a shell to the running container and create a new file named `logs.txt` in the directory `/data/test`. What's the file's user ID and group ID?

2. Create a Deployment named `nginx` in the namespace `h20` with the 3 replicas. The Pod template should use the image `nginx:1.25.3-alpine`.

Using the security context, assign the `drop` Linux capability for that Pod template. The attribute for the `drop` capabilities should use the value `all`.

Create the Deployment object and inspect its replicas. Does NGINX work as expected?

PART VI

Services and Networking

The last domain in the exam is named *Services and Networking*. It covers the Kubernetes primitives important for establishing and restricting communication between microservices running in the cluster, or outside consumers. More specifically, this domain covers the primitives Services and Ingresses, as well as network policies.

The following chapters cover these concepts:

- [Chapter 21](#) introduces the Services resource type. You will learn how to expose a microservice inside of the cluster to other portions of the system. Services also allows for making an application accessible to end users outside of the cluster. This chapter doesn't stop there. It also provides techniques for troubleshooting misconfigured Service objects.
- [Chapter 22](#) starts by explaining why a Service is often not good enough for exposing an application to outside consumers. The Ingress primitive can expose a load-balanced endpoint to consumers accessible via HTTP(S).
- [Chapter 23](#) explains the need for network policies from a security perspective. By default, Kubernetes' Pod-to-Pod communication is unrestricted; however, you want to implement the principle of least privilege to ensure that only those Pods can talk to other Pods required by your architectural needs. Limiting network communication between Pods will decrease the potential attack surface.

CHAPTER 21

Services

In “Using a Pod’s IP Address for Network Communication” on page 49, we learned that you can communicate with a Pod by its IP address. A restart of a Pod will automatically assign a new virtual cluster IP address. Therefore, other parts of your system cannot rely on the Pod’s IP address if they need to talk to one another.

Building a microservices architecture, where each of the components runs in its own Pod with the need to communicate with each other through a stable network interface, requires a different primitive, the Service.

The Service implements an abstraction layer on top of Pods, assigning a fixed virtual IP fronting all the Pods with matching labels, and that virtual IP is called Cluster IP. This chapter will focus on the ins and outs of Services, and most importantly the exposure of Pods inside and outside of the cluster based on their declared type.



Accessing a Service in minikube

Accessing Services of type NodePort and LoadBalancer in minikube requires special handling. Refer to the [documentation](#) for detailed instructions.

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objective:

- Provide and troubleshoot access to applications via Services

Working with Services

In a nutshell, Services provide discoverable names and load balancing to a set of Pods. The Service remains agnostic from IP addresses with the help of the Kubernetes DNS control-plane component, an aspect we'll discuss in "[Discovering the Service by DNS lookup](#)" on page 256. Similar to a Deployment, the Service determines the Pods it works on with the help of label selection.

[Figure 21-1](#) illustrates the functionality. Pod 1 receives traffic as its assigned label matches with the label selection defined in the Service. Pod 2 does not receive traffic as it defines a nonmatching label.

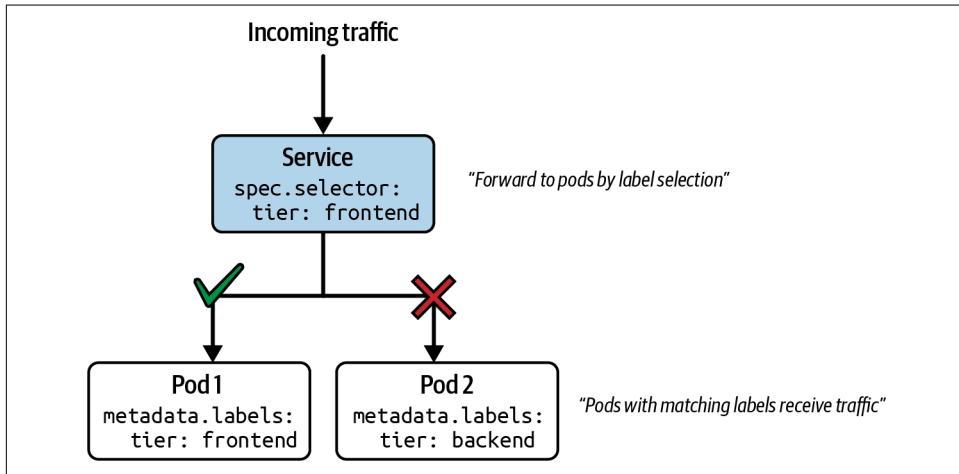


Figure 21-1. Service traffic routing based on label selection

Note that it is possible to create a Service without a label selector to support other scenarios. Refer to the relevant [Kubernetes documentation](#) for more information.



Services and Deployments

Services are a complementary concept to Deployments. Services route network traffic to a set of Pods, and Deployments manage a set of Pods, the replicas. While you can use both concepts in isolation, it is recommended to use Deployments and Services together. The primary reason is the ability to scale the number of replicas and at the same time being able to expose an endpoint to funnel network traffic to those Pods.

Service Types

Every Service defines a type. The type is responsible for exposing the Service inside and/or outside of the cluster. [Table 21-1](#) lists the Service types relevant to the exam.

Table 21-1. Service types

Type	Description
ClusterIP	Exposes the Service on a cluster-internal IP. Reachable only from within the cluster. Kubernetes uses a round-robin algorithm to distribute traffic evenly among the targeted Pods.
NodePort	Exposes the Service on each node's IP address at a static port. Accessible from outside of the cluster. The Service type does not provide any load balancing across multiple nodes.
Load Balancer	Exposes the Service externally using a cloud provider's load balancer.

Other Service types, e.g. `ExternalName` or the headless Service, can be defined; however, we'll not address them in this book as they are not within the scope of the exam. For more information, refer to the [Kubernetes documentation](#).

Service type inheritance

The Service types just mentioned, `ClusterIP`, `NodePort`, and `LoadBalancer`, make a Service accessible with different levels of exposure. It's imperative to understand that those Service types also build on top of each other. [Figure 21-2](#) shows the relationship between different Service types.

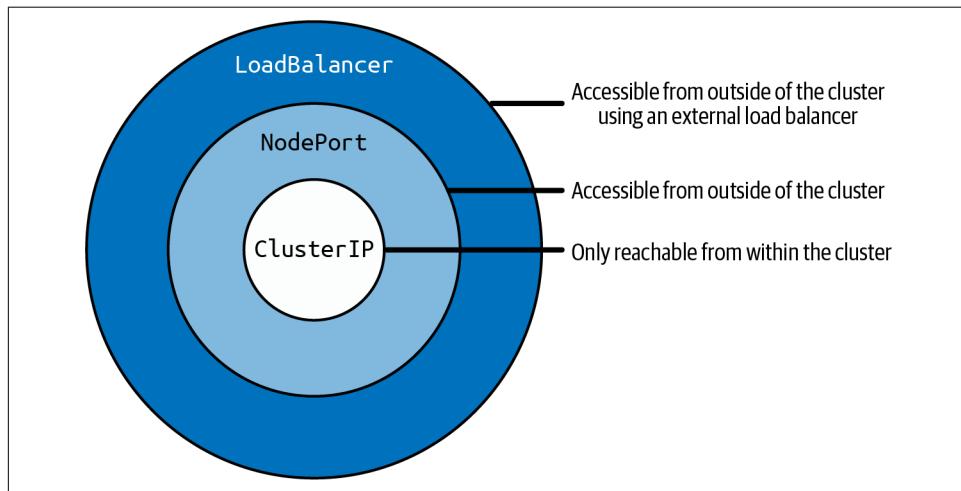


Figure 21-2. Network accessibility characteristics for Service types

For example, creating a Service of type `NodePort` means that the Service will bear the network accessibility characteristics of a `ClusterIP` Service type as well. In turn, a `NodePort` Service is accessible from within and from outside of the cluster. This chapter demonstrates each Service type by example. You will find references to the inherited exposure behavior in the following sections.

When to use which Service type?

When building a microservices architecture, the question arises which Service type to choose to implement certain use cases. We briefly discuss this question here.

The `ClusterIP` Service type is suitable for use cases that call for exposing a microservice to other Pods within the cluster. Say you have a frontend microservice that needs to connect to one or many backend microservices. To properly implement the scenario, you'd stand up a `ClusterIP` Service that routes traffic to the backend Pods. The frontend Pods would then talk to that Service.

The `NodePort` Service type is often mentioned as a way to expose an application to consumers external to the cluster. Consumers will have to know the node's IP address and the statically assigned port to connect to the Service. That's problematic for multiple reasons. First, the node port is usually allocated dynamically. Therefore, you won't typically know it in advance. Second, providing the node's IP address will funnel the network traffic only through a single node so you will not have load balancing at your disposal. Finally, by opening a publicly available node port, you are at risk of increasing the attack surface of your cluster. For all these reasons, a `NodePort` Service is primarily used for development or testing purposes, and less so in production environments.

The `LoadBalancer` Service type makes the application available to outside consumers through an external IP address provided by an external load balancer. Network traffic will be distributed across multiple nodes in the cluster. This solution works great for production environments, but keep in mind that every provisioned load balancer will accrue costs and can lead to an expensive infrastructure bill. A more cost-effective solution is the use of an Ingress, discussed in [Chapter 22](#).

Port Mapping

A Service uses label selection to determine the set of Pods to forward traffic to. Successful routing of network traffic depends on the port mapping.

[Figure 21-3](#) shows a Service that accepts incoming traffic on port 80. That's the port defined by the attribute `spec.ports[] .port` in the manifest. Any incoming traffic is then routed toward the target port, represented by `spec.ports[] .targetPort`.

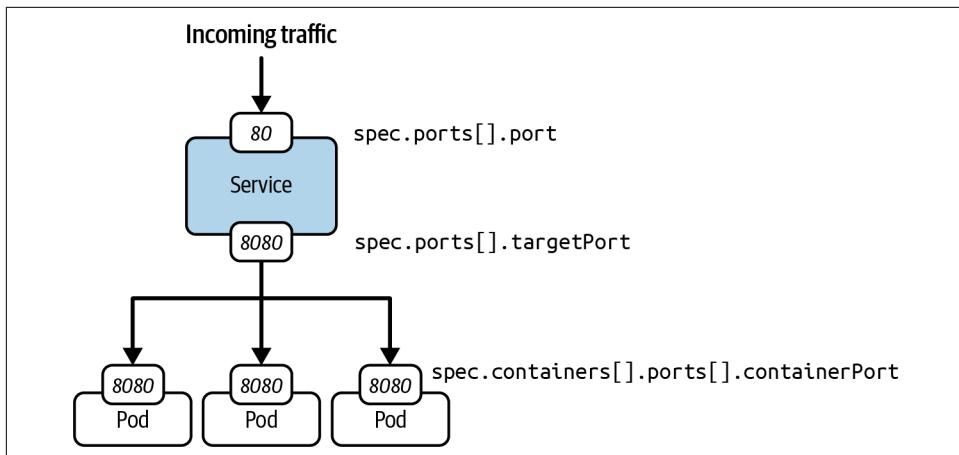


Figure 21-3. Service port mapping

The target port is the same port as defined by the container with `spec.containers[].ports[].containerPort` running inside the label-selected Pod. In this example, that's port 8080. The selected Pod(s) will receive traffic only if the Service's target port and the container port match.

Creating Services

You can create Services in a variety of ways, some of which are more appropriate for the exam as they provide a fast turnaround. Let's discuss the imperative approach first.

A Service needs to select a Pod by a matching label. The Pod created by the following `run` command is called `echoserver`, which exposes the application on the container port 8080. Internally, it automatically assigns the label key-value pair `run=echoserver` to the object:

```
$ kubectl run echoserver --image=k8s.gcr.io/echoserver:1.10 --restart=Never \
--port=8080
pod/echoserver created
```

You can create a Service object using the `create service` command. Make sure to provide the Service type as a mandatory argument. Here we are using the type `clusterip`. The command-line option `--tcp` specifies the port mapping. Port 80 exposes the Service to incoming network traffic. Port 8080 targets the container port exposed by the Pod:

```
$ kubectl create service clusterip echoserver --tcp=80:8080
service/echoserver created
```

An even faster workflow of creating a Pod and Service together can be achieved with a `run` command and the `--expose` option. The following command creates both objects in one swoop while establishing the proper label selection. This command-line option is a good choice during the exam to save time if you are asked to create a Pod and a Service:

```
$ kubectl run echoserver --image=k8s.gcr.io/echoserver:1.10 --restart=Never \
  --port=8080 --expose
service/echoserver created
pod/echoserver created
```

It's actually more common to use a Deployment and Service that work together. The following set of commands creates a Deployment with five replicas and then uses the `expose deployment` command to instantiate the Service object. The port mapping can be provided with the options `--port` and `--target-port`:

```
$ kubectl create deployment echoserver --image=k8s.gcr.io/echoserver:1.10 \
  --replicas=5
deployment.apps/echoserver created
$ kubectl expose deployment echoserver --port=80 --target-port=8080
service/echoserver exposed
```

Example 21-1 shows the representation of a Service in the form of a YAML manifest. The Service declares the key-value `app=echoserver` for label selection and defines the port mapping from 80 to 8080.

Example 21-1. A Service defined by a YAML manifest

```
apiVersion: v1
kind: Service
metadata:
  name: echoserver
spec:
  selector:
    run: echoserver ①
  ports: ②
  - port: 80
    targetPort: 8080
```

- ① Selects all Pods with the given label assignment.
- ② Defines incoming and outgoing ports of the Service. The outgoing port needs to match the container port of the selected Pods.

The Service YAML manifest shown does not assign an explicit type. A Service object that does not specify a value for the attribute `spec.type` will default to `ClusterIP` upon creation.

Listing Services

Listing all Services presents a table view that includes the Service type, the cluster IP address, an optional external IP address, and the incoming port(s). Here, you can see the output for the echoserver Pod we created earlier:

```
$ kubectl get services
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
echoserver  ClusterIP  10.109.241.68  <none>          80/TCP      6s
```

Kubernetes assigns a cluster IP address given that the Service type is `ClusterIP`. An external IP address is not available for this Service type. The Service is accessible on port 80.

Rendering Service Details

You may want to drill into the details of a Service for troubleshooting purposes. That might be the case if the incoming traffic to a Service isn't routed properly to the set of Pods you expect to handle the request.

The `describe service` command renders valuable information about the configuration of a Service. The configuration relevant to troubleshooting a Service is the value of the fields Selector, IP, Port, TargetPort, and Endpoints. A common source of misconfiguration is incorrect label selection and port assignment. Make sure that the selected labels are actually available in the Pods intended to route traffic to and that the target port of the Service matches the exposed container port of the Pods.

Take a look at the output of the following `describe` command. It's the details for a Service created for five Pods controlled by a Deployment. The Endpoints attribute lists a range of endpoints, one for each of the Pods:

```
$ kubectl describe service echoserver
Name:            echoserver
Namespace:       default
Labels:          app=echoserver
Annotations:     <none>
Selector:        app=echoserver
Type:            ClusterIP
IP Family Policy: SingleStack
IP Families:    IPv4
IP:              10.109.241.68
IPs:             10.109.241.68
Port:            <unset>  80/TCP
TargetPort:      8080/TCP
Endpoints:       172.17.0.4:8080,172.17.0.5:8080,172.17.0.7:8080 + 2 more...
Session Affinity: None
Events:          <none>
```

An endpoint is a resolvable network endpoint, which serves as the virtual IP address and container port of a Pod. If a Service does not render any endpoints then you are likely dealing with a misconfiguration.

Kubernetes represents endpoints by a dedicated primitive that you can query for. The Endpoint object is created at the same time you instantiate the Service object. The following command lists the endpoints for the Service named echoserver:

```
$ kubectl get endpoints echoserver
NAME      ENDPOINTS                                     AGE
echoserver  172.17.0.4:8080,172.17.0.5:8080,172.17.0.7:8080 + 2 more...  8m5s
```

The details of the endpoints give away the full list of IP addresses and ports combinations:

```
$ kubectl describe endpoints echoserver
Name:            echoserver
Namespace:       default
Labels:          app=echoserver
Annotations:    endpoints.kubernetes.io/last-change-trigger-time: \
                2021-11-15T19:09:04Z
Subsets:
  Addresses:     172.17.0.4,172.17.0.5,172.17.0.7,172.17.0.8,172.17.0.9
  NotReadyAddresses: <none>
  Ports:
    Name  Port  Protocol
    ----  ---   -----
    <unset>  8080  TCP
Events:  <none>
```

The ClusterIP Service Type

ClusterIP is the default Service type. It exposes the Service on a cluster-internal IP address. That means the Service can be accessed only from a Pod running inside of the cluster and not from outside of the cluster (e.g., if you were to make a call to the Service from your local machine). [Figure 21-4](#) illustrates the accessibility of a Service with type ClusterIP.

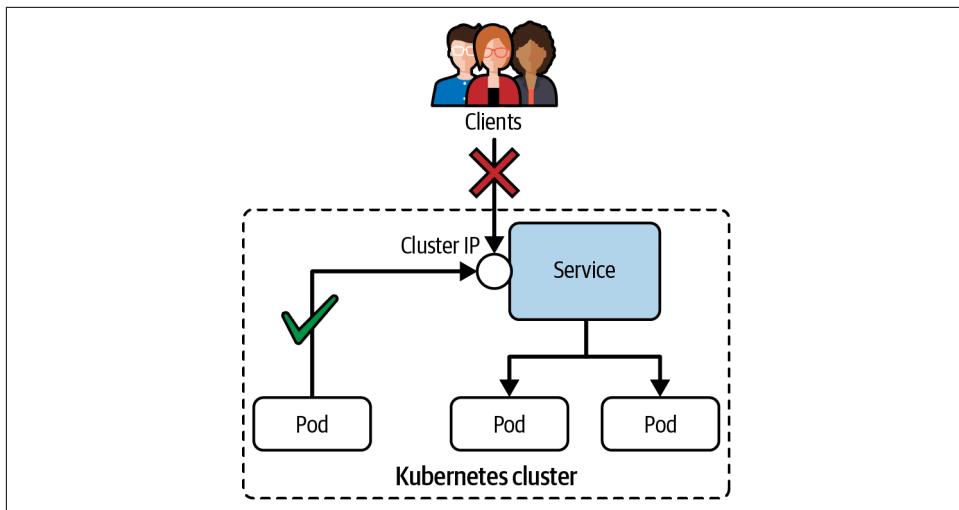


Figure 21-4. Accessibility of a Service with the type *ClusterIP*

Creating and Inspecting the Service

We will create a Pod and a corresponding Service to demonstrate the runtime behavior of the `ClusterIP` Service type. The Pod named `echoserver` exposes the container port `8080` and specifies the label `app=echoserver`. The Service defines port `5005` for incoming traffic, which is forwarded to outgoing port `8080`. The label selection matches the Pod we set up:

```
$ kubectl run echoserver --image=k8s.gcr.io/echoserver:1.10 --restart=Never \
--port=8080 -l app=echoserver
pod/echoserver created
$ kubectl create service clusterip echoserver --tcp=5005:8080
service/echoserver created
```

Inspecting the live object with the command `kubectl get service echoserver -o yaml` will render the assigned cluster IP address. [Example 21-2](#) shows an abbreviated version of the Service runtime representation.

Example 21-2. A ClusterIP Service object at runtime

```
apiVersion: v1
kind: Service
metadata:
  name: echoserver
spec:
  type: ClusterIP          ①
  clusterIP: 10.96.254.0    ②
  selector:
    app: echoserver
```

```
ports:
- port: 5005
  targetPort: 8080
  protocol: TCP
```

- ➊ The Service type set to ClusterIP.
- ➋ The cluster IP address assigned to the Service at runtime.

The cluster IP address that makes the Service available in this example is 10.96.254.0. Listing the Service object is an alternative way to render the information we need to make a call to the Service:

```
$ kubectl get service echoserver
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
echoserver  ClusterIP  10.96.254.0  <none>        5005/TCP  8s
```

Next up, we'll try to make a call to the Service.

Accessing the Service

You can access the Service using a combination of the cluster IP address and the incoming port: 10.96.254.0:5005. Making a request from any other machine residing outside of the cluster will fail, as illustrated by the following `wget` command:

```
$ wget 10.96.254.0:5005 --timeout=5 --tries=1
--2021-11-15 15:45:36-- http://10.96.254.0:5005/
Connecting to 10.96.254.0:5005... ]failed: Operation timed out.
Giving up.
```

Accessing the Service from a Pod from within the cluster properly routes the request to the Pod matching the label selection:

```
$ kubectl run tmp --image=busybox:1.36.1 --restart=Never -it --rm \
  -- wget 10.96.254.0:5005
Connecting to 10.96.254.0:5005 (10.96.254.0:5005)
saving to 'index.html'
index.html          100% [*****] 408  0:00:00 ETA
'index.html' saved
pod "tmp" deleted
```

Apart from using the cluster IP address and the port, you can also discover a Service by DNS name and environment variables available to containers.

Discovering the Service by DNS lookup

Kubernetes registers every Service by its name with the help of its DNS service named CoreDNS. Internally, CoreDNS will store the Service name as a hostname and maps it to the cluster IP address. Accessing a Service by its DNS name instead of an IP

address is much more convenient and expressive when building microservice architectures.

You can verify the correct service discovery by running a Pod in the same namespace that makes a call to the Service by using its hostname and incoming port:

```
$ kubectl run tmp --image=busybox:1.36.1 --restart=Never -it --rm \
  -- wget echoserver:5005
Connecting to echoserver:5005 (10.96.254.0:5005)
saving to 'index.html'
index.html      100% |*****| 408  0:00:00 ETA
'index.html' saved
pod "tmp" deleted
```

It's not uncommon to make a call from a Pod to a Service that lives in a different namespace. Referencing just the hostname of the Service does not work across namespaces. You need to append the namespace as well. The following makes a call from a Pod in the other namespace to the Service in the default namespace:

```
$ kubectl run tmp --image=busybox:1.36.1 --restart=Never -it --rm \
  -n other -- wget echoserver.default:5005
Connecting to echoserver.default:5005 (10.96.254.0:5005)
saving to 'index.html'
index.html      100% |*****| 408  0:00:00 ETA
'index.html' saved
pod "tmp" deleted
```

The full hostname for a Service is `echoserver.default.svc.cluster.local`. The string `svc` describes the type of resource we are communicating with. CoreDNS uses the default value `cluster.local` as a domain name (which is configurable if you want to change it). You do not have to spell out the full hostname when communicating with a Service.

Discovering the Service by environment variables

You may find it easier to use the Service connection information directly from the application running in a Pod. The kubelet makes the cluster IP address and port for every active Service available as environment variables. The naming convention for Service-related environments variable are `<SERVICE_NAME>_SERVICE_HOST` and `<SERVICE_NAME>_SERVICE_PORT`.



Availability of Service environment variables

Make sure you create the Service before instantiating the Pod. Otherwise, those environment variables won't be populated.

You can check on the actual key-value pairs by listing the environment variables of the container, as follows:

```
$ kubectl exec -it echoserver -- env  
ECHOSERVER_SERVICE_HOST=10.96.254.0  
ECHOSERVER_SERVICE_PORT=8080  
...
```

The name of the Service, `echoserver`, does not include any special characters. That's why the conversion to the environment variable key is easy; the Service name was simply upper-cased to conform to environment variable naming conventions. Any special characters (such as dashes) in the Service name will be replaced by underscore characters. You need to make sure that the Service has been created before starting a Pod if you want those environment variables populated.

The NodePort Service Type

Declaring a Service with type `NodePort` exposes access through the node's IP address and can be resolved from outside of the Kubernetes cluster. The node's IP address can be reached in combination with a port number in the range of 30000 and 32767 (also called the node port), assigned automatically upon the creation of the Service. [Figure 21-5](#) illustrates the routing of traffic to Pods via a `NodePort`-type Service.

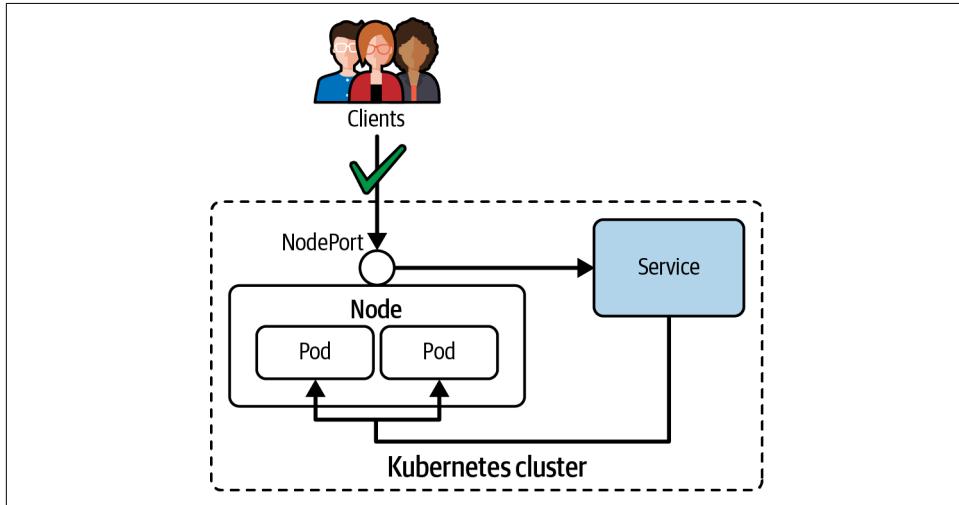


Figure 21-5. Accessibility of a Service with the type `NodePort`

The node port is opened on every node in the cluster, and its value is global and unique at the cluster-scope level. To avoid port conflicts, it's best to not define the exact node port and to let Kubernetes find an available port.

Creating and Inspecting the Service

The next two commands create a Pod and a Service of type NodePort. The only difference here is that `nodeport` is provided instead of `clusterip` as a command-line option:

```
$ kubectl run echoserver --image=k8s.gcr.io/echoserver:1.10 --restart=Never \
  --port=8080 -l app=echoserver
pod/echoserver created
$ kubectl create service nodeport echoserver --tcp=5005:8080
service/echoserver created
```

The runtime representation of the Service object is shown in [Example 21-3](#). It's important to point out that the node port will be assigned automatically. Keep in mind `NodePort` (capital *N*) is the Service type, whereas `nodePort` (lowercase *n*) is the key for the value.

Example 21-3. A NodePort Service object at runtime

```
apiVersion: v1
kind: Service
metadata:
  name: echoserver
spec:
  type: NodePort          ①
  clusterIP: 10.96.254.0
  selector:
    app: echoserver
  ports:
  - port: 5005
    nodePort: 30158        ②
    targetPort: 8080
    protocol: TCP
```

- ① The Service type set to `NodePort`.
- ② The statically-assigned node port that makes the Service accessible from outside of the cluster.

Once the Service is created, you can list it. You will find that the port representation contains the statically assigned port that makes the Service accessible:

```
$ kubectl get service echoserver
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
echoserver  NodePort  10.101.184.152  <none>           5005:30158/TCP  5s
```

In this output, the node port is 30158 (identifiable by the separating colon). The incoming port 5005 is still available for the purpose of resolving the Service from within the cluster.

Accessing the Service

From within the cluster, you can still access the Service using the cluster IP address and port number. This Service displays exactly the same behavior as if it were of type ClusterIP:

```
$ kubectl run tmp --image=busybox:1.36.1 --restart=Never -it --rm \
-- wget 10.101.184.152:5005
Connecting to 10.101.184.152:5005 (10.101.184.152:5005)
saving to 'index.html'
index.html      100% |*****| 414  0:00:00 ETA
'index.html' saved
pod "tmp" deleted
```

From outside of the cluster, you need to use the IP address of any worker node in the cluster and the statically assigned port. One way to determine the worker node's IP address is by rendering the node details. Another option is to use the `status.hostIP` attribute value of a Pod, which is the IP address of the worker node the Pod runs on.

The node IP address here is 192.168.64.15. It can be used to call the Service from outside of the cluster:

```
$ kubectl get nodes -o \
  jsonpath='{ .items[*].status.addresses[?(.type=="InternalIP")].address }'
192.168.64.15
$ wget 192.168.64.15:30158
--2021-11-16 14:10:16-- http://192.168.64.15:30158/
Connecting to 192.168.64.15:30158... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/plain]
Saving to: 'index.html'

...
```

The LoadBalancer Service Type

The last Service type to discuss in this book is the `LoadBalancer`. This Service type provisions an external load balancer, primarily available to Kubernetes cloud providers, which exposes a single IP address to distribute incoming requests to the cluster nodes. The implementation of the load balancing strategy (e.g., round robin) is up to the cloud provider.



Load balancers for on-premises Kubernetes clusters

Kubernetes does not offer a native load balancer solution for on-premises clusters. Cloud providers are in charge of providing an appropriate implementation. The [MetallLB project](#) aims to fill the gap.

Figure 21-6 shows an architectural overview of the LoadBalancer Service type.

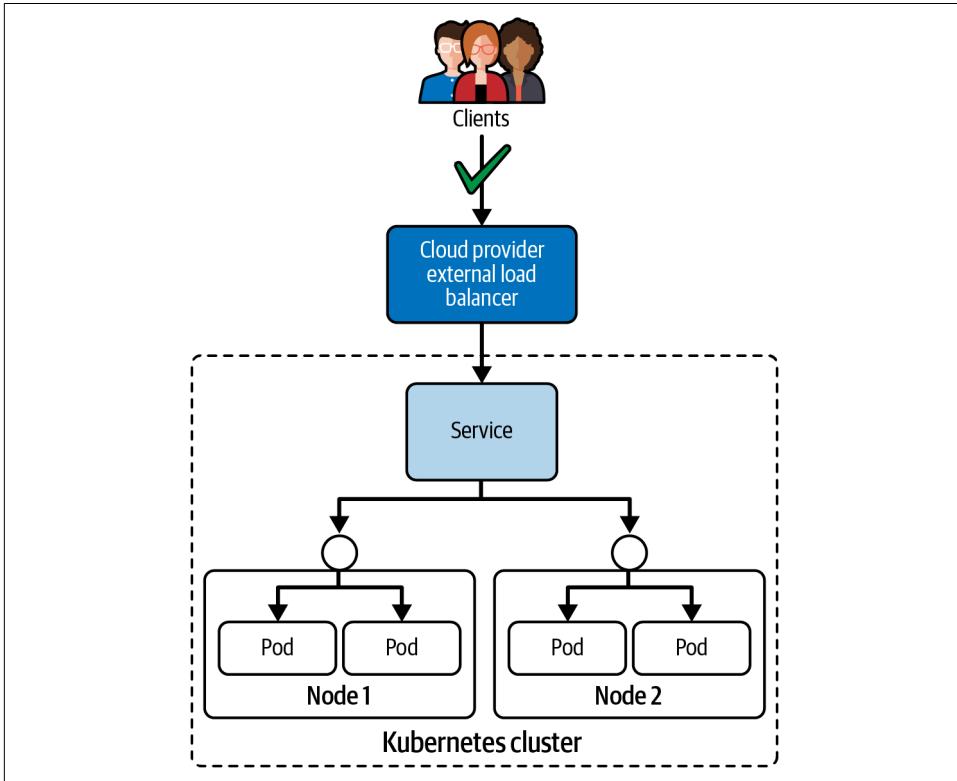


Figure 21-6. Accessibility of a Service with the type LoadBalancer

As you can see from the illustration, the load balancer routes traffic between different nodes, as long as the targeted Pods fulfill the requested label selection.

Creating and Inspecting the Service

To create a Service as a load balancer, set the type to LoadBalancer in the manifest or by using the `create service loadbalancer` command:

```
$ kubectl run echoserver --image=k8s.gcr.io/echoserver:1.10 --restart=Never \
--port=8080 -l app=echoserver
pod/echoserver created
$ kubectl create service loadbalancer echoserver --tcp=5005:8080
service/echoserver created
```

The runtime characteristics of a LoadBalancer Service type look similar to the ones provided by the NodePort Service type. The main difference is that the external IP address column has a value, as shown in Example 21-4.

Example 21-4. A LoadBalancer Service object at runtime

```
apiVersion: v1
kind: Service
metadata:
  name: echoserver
spec:
  type: LoadBalancer          ①
  clusterIP: 10.96.254.0
  loadBalancer: 10.109.76.157  ②
  selector:
    app: echoserver
  ports:
    - port: 5005
      targetPort: 8080
      nodePort: 30158
      protocol: TCP
```

- ① The Service type set to LoadBalancer.
- ② The external IP address assigned to the Service at runtime.

Listing the Service renders the external IP address, which is 10.109.76.157, as demonstrated by this command:

```
$ kubectl get service echoserver
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
echoserver   LoadBalancer   10.109.76.157   10.109.76.157   5005:30642/TCP   5s
```

Given that the external load balancer needs to be provisioned by the cloud provider, it may take a little time until the external IP address becomes available.

Accessing the Service

To call the Service from outside of the cluster, use the external IP address and its incoming port:

```
$ wget 10.109.76.157:5005
--2021-11-17 11:30:44--  http://10.109.76.157:5005/
Connecting to 10.109.76.157:5005... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/plain]
Saving to: 'index.html'

...
```

As discussed, a LoadBalancer Service is also accessible in the same way as you would access a ClusterIP or NodePort Service.

Summary

Kubernetes assigns a unique IP address for every Pod in the cluster. Pods can communicate with each other using that IP address; however, you cannot rely on the IP address to be stable over time. That's why Kubernetes provides the Service resource type.

A Service forwards network traffic to a set of Pods based on label selection and port mappings. Every Service needs to assign a type that determines how the Service becomes accessible from within or outside of the cluster. The Service types relevant to the exam are `ClusterIP`, `NodePort`, and `LoadBalancer`. CoreDNS, the DNS server for Kubernetes, allows Pods to access the Service by hostname from the same and other namespaces.

Exam Essentials

Understand the purpose of a Service

Pod-to-Pod communication via their IP addresses doesn't guarantee a stable network interface over time. A restart of the Pod will lease a new virtual IP address. The purpose of a Service is to provide that stable network interface so that you can operate complex microservice architecture that runs in a Kubernetes cluster. In most cases, Pods call a Service by hostname. The hostname is provided by the DNS server named CoreDNS running as a Pod in the `kube-system` namespace.

Practice how to access a Service for each type

The exam expects you to understand the differences between the Service types `ClusterIP`, `NodePort`, and `LoadBalancer`. Depending on the assigned type, a Service becomes accessible from inside the cluster or from outside the cluster.

Work through Service troubleshooting scenarios

It's easy to get the configuration of a Service wrong. Any misconfiguration won't allow network traffic to reach the set of Pod it was intended for. Common misconfigurations include incorrect label selection and port assignments. The `kubectl get endpoints` command will give you an idea which Pods a Service can route traffic to.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. Create a Service named `myapp` of type `ClusterIP` that exposes port 80 and maps to the target port 80.

Create a Deployment named `myapp` that creates 1 replica running the image `nginx:1.23.4-alpine`. Expose the container port 80. Scale the Deployment to 2 replicas.

Create a temporary Pod using the image `busybox:1.36.1` and execute a `wget` command against the IP of the service.

Change the service type to `NodePort` so that the Pods can be reached from outside of the cluster. Execute a `wget` command against the service from outside of the cluster.

2. Kate is a developer in charge of implementing a web-based application stack. She is not familiar with Kubernetes, and asked if you could help out. The relevant objects have been created; however, connection to the application cannot be established from within the cluster. Help Kate with fixing the configuration of her YAML manifests.

Navigate to the directory `app-a/ch21/troubleshooting` of the checked-out GitHub repository [`bmuschko/ckad-study-guide`](#). Create the objects from the YAML manifest `setup.yaml`. Inspect the objects in the namespace `y72`.

Create a temporary Pod using the image `busybox:1.36.1` in the namespace `y72`. The container command should make a `wget` call to the Service `web-app`. The `wget` call will not be able to establish a successful connection to the Service.

Identify the root cause for the connection issue and fix it. Verify the correct behavior by repeating the previous step. The `wget` call should return a successful response.

Ingresses

Chapter 21 delved into the purpose and creation of the Service primitive. Once there's a need to expose the application to external consumers, selecting an appropriate Service type becomes crucial. The most practical choice often involves creating a Service of type LoadBalancer. Such a Service offers load balancing capabilities by assigning an external IP address accessible to consumers outside the Kubernetes cluster.

However, opting for a LoadBalancer Service for each externally reachable application has drawbacks. In a cloud provider environment, each Service triggers the provisioning of an external load balancer, resulting in increased costs. Additionally, managing a collection of LoadBalancer Service objects can lead to administrative challenges, as a new object must be established for each externally accessible microservice.

To mitigate these issues, the Ingress primitive comes into play, offering a singular, load-balanced entry point to an application stack. An Ingress possesses the ability to route external HTTP(S) requests to one or more Services within the cluster based on an optional, DNS-resolvable host name and URL context path. This chapter will guide you through the creation and access of an Ingress.

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objective:

- Use Ingress rules to expose applications



Accessing an Ingress in minikube

Accessing an Ingress in minikube requires special handling. Refer to the Kubernetes tutorial “[Set up Ingress on Minikube with the NGINX Ingress Controller](#)” for detailed instructions.

Working with Ingresses

The Ingress exposes HTTP (and optionally HTTPS) routes to clients outside of the cluster through an externally-reachable URL. The routing rules configured with the Ingress determine *how* the traffic should be routed. Cloud provider Kubernetes environments will often deploy an external load balancer. The Ingress receives a public IP address from the load balancer. You can configure rules for routing traffic to multiple Services based on specific URL context paths, as shown in [Figure 22-1](#).

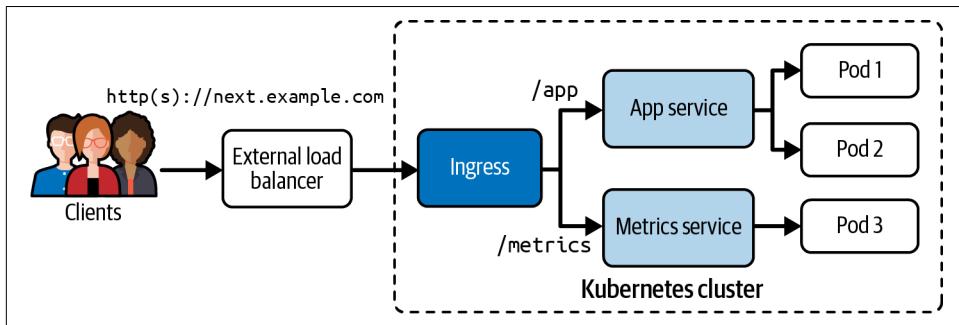


Figure 22-1. Managing external access to the Services via HTTP(S)

The scenario depicted in [Figure 22-1](#) instantiates an Ingress as the sole entry point for HTTP(S) calls to the domain name “next.example.com.” Based on the provided URL context, the Ingress directs the traffic to either of the fictional Services: one designed for a business application and the other for fetching metrics related to the application.

Specifically, the URL context path `/app` is routed to the App Service responsible for managing the business application. Conversely, sending a request to the URL context `/metrics` results in the call being forwarded to the Metrics Service, which is capable of returning relevant metrics.

Installing an Ingress Controller

For Ingress to function, an Ingress controller is essential. This controller assesses the set of rules outlined by an Ingress, dictating the routing of traffic. The choice of Ingress controller often depends on the specific use cases, requirements, and preferences of the Kubernetes cluster administrator. Noteworthy examples of production-grade Ingress controllers include the [F5 NGINX Ingress Controller](#) or the [AKS](#)

Application Gateway Ingress Controller. Additional options can be explored in the Kubernetes documentation.

You should find at least one Pod that runs the Ingress controller after installing it. This output renders the Pod created by the NGINX Ingress controller residing in the namespace `ingress-nginx`:

```
$ kubectl get pods -n ingress-nginx
NAME                               READY   STATUS    RESTARTS   AGE
ingress-nginx-admission-create-qqhrp   0/1     Completed   0          60s
ingress-nginx-admission-patch-56z26   0/1     Completed   1          60s
ingress-nginx-controller-7c6974c4d8-2gg8c   1/1     Running   0          60s
```

Once the Ingress controller Pod transitions into the “Running” status, you can assume that the rules defined by Ingress objects will be evaluated.

Deploying Multiple Ingress Controllers

Certainly, deploying multiple Ingress controllers within a single cluster is a feasible option, especially if a cloud provider has preconfigured an Ingress controller in the Kubernetes cluster. The Ingress API introduces the attribute `spec.ingressClassName` to facilitate the selection of a specific controller implementation by name. To identify all installed Ingress classes, you can use the following command:

```
$ kubectl get ingressclasses
NAME      CONTROLLER      PARAMETERS   AGE
nginx    k8s.io/ingress-nginx <none>    14m
```

Kubernetes determines the default Ingress class by scanning for the annotation `ingressclass.kubernetes.io/is-default-class: "true"` within all Ingress class objects. In scenarios where Ingress objects do not explicitly specify an Ingress class using the attribute `spec.ingressClassName`, they automatically default to the Ingress class marked as the default through this annotation. This mechanism provides flexibility in managing Ingress classes and allows for a default behavior when no specific class is specified in individual Ingress objects.

Configuring Ingress Rules

When creating an Ingress, you have the flexibility to define one or multiple rules. Each rule encompasses the specification of an optional host, a set of URL context paths, and the backend responsible for routing the incoming traffic. This structure allows for fine-grained control over how external HTTP(S) requests are directed within the Kubernetes cluster, catering to different services based on specified conditions. Table 22-1 describes the three rules.

Table 22-1. Ingress rules

Type	Example	Description
An optional host	next.example.com	If provided, the rules apply to that host. If no host is defined, all inbound HTTP(S) traffic is handled (e.g., if made through the IP address of the Ingress).
A list of paths	/app	Incoming traffic must match the host and path to correctly forward the traffic to a Service.
The backend	app-service:8080	A combination of a Service name and port.

An Ingress controller can optionally define a default backend that is used as a fallback route should none of the configured Ingress rules match. You can learn more about it in the [documentation of the Ingress primitive](#).

Creating Ingresses

You can create an Ingress with the imperative `create ingress` command. The main command-line option you need to provide is `--rule`, which defines the rules in a comma-separated fashion. The notation for each key-value pair is `<host>/<path>=<service>:<port>`. Let's create an Ingress object with two rules:

```
$ kubectl create ingress next-app \
  --rule="next.example.com/app=app-service:8080" \
  --rule="next.example.com/metrics=metrics-service:9090"
ingress.networking.k8s.io/next-app created
```

If you look at the output of the `create ingress --help` command, more fine-grained rules can be specified.



Support for TLS termination

Port 80 for HTTP traffic is implied, as we didn't specify a reference to a TLS Secret object. If you have specified `tls=mysecret` in the rule definition, then the port 443 would be listed here as well. For more information on enabling HTTPS traffic, see the [Kubernetes documentation](#). The exam does not cover configuring TLS termination for an Ingress.

Using a YAML manifest to define Ingress is often more intuitive and preferred by many. It provides a clearer and more structured way to express the desired configuration. The Ingress defined as a YAML manifest is shown in [Example 22-1](#).

Example 22-1. An Ingress defined by a YAML manifest

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
```

```

name: next-app
annotations:
  nginx.ingress.kubernetes.io/rewrite-target: /$1 ①
spec:
  rules:
    - host: next.example.com
      http:
        paths:
          - backend:
              service:
                name: app-service
                port:
                  number: 8080
            path: /app
            pathType: Exact
    - host: next.example.com
      http:
        paths:
          - backend:
              service:
                name: metrics-service
                port:
                  number: 9090
            path: /metrics
            pathType: Exact

```

- ① Assigns a NGNIX ingress-specific annotation for rewriting the URL.
- ② Defines the rule that maps the `app-service` backend to the URL `next.example.com/app`.
- ③ Defines the rule that maps the `metrics-service` backend to the URL `next.example.com/metrics`.

The Ingress YAML manifest contains one major difference from the live object representation created by the imperative command: the assignment of an Ingress controller annotation. Some Ingress controller implementations provide annotations to customize their behavior. You can find the full list of annotations that come with the NGINX Ingress controller in the [corresponding documentation](#).

Defining Path Types

The previous YAML manifest demonstrates one of the options for specifying a path type via the attribute `spec.rules[].http.paths[].pathType`. The path type defines how an incoming request is evaluated against the declared path. [Table 22-2](#) indicates the evaluation for incoming requests and their paths. See the [Kubernetes documentation](#) for a more comprehensive list.

Table 22-2. Ingress path types

Path Type	Rule	Incoming Request
Exact	/app	Matches /app but does not match /app/test or /app/
Prefix	/app	Matches /app and /app/ but does not match /app/test

The key distinction between the `Exact` and `Prefix` path types lies in their treatment of trailing slashes. The `Prefix` path type focuses solely on the provided prefix of a URL context path, allowing it to accommodate requests with URLs that include a trailing slash. In contrast, the `Exact` path type is more stringent, requiring an exact match of the specified URL context path without considering a trailing slash.

Listing Ingresses

Listing Ingresses can be achieved with the `get ingress` command. You will see some of the information you specified when creating the Ingress (e.g., the hosts):

```
$ kubectl get ingress
NAME      CLASS      HOSTS          ADDRESS      PORTS      AGE
next-app   nginx     next.example.com  192.168.66.4  80        5m38s
```

The Ingress automatically selected the default Ingress class `nginx` configured by the Ingress controller. You can find the information under the `CLASS` column. The value listed under the `ADDRESS` columns is the IP address provided by the external load balancer.

Rendering Ingress Details

The `describe ingress` command is a valuable tool for obtaining detailed information about an Ingress resource. It presents the rules in a clear table format, which aids in understanding the routing configurations. Additionally, when troubleshooting, it's essential to pay attention to any additional messages or events.

In the provided output, it's evident that there might be an issue with the Services named `app-service` and `metrics-service` that are mapped in the Ingress rules. This discrepancy between the specified services and their existence can lead to routing errors:

```
$ kubectl describe ingress next-app
Name:           next-app
Labels:         <none>
Namespace:      default
Address:        192.168.66.4
Ingress Class:  nginx
Default backend: <default>
Rules:
Host          Path  Backends
-----
next.example.com
```

```

/app      app-service:8080 (<error: endpoints \
"app-service" not found>)
/metrics  metrics-service:9090 (<error: endpoints \
"metrics-service" not found>)
Annotations: <none>
Events:
  Type   Reason  Age           From            ...
  ----  -----  --  -----
Normal  Sync    6m45s (x2 over 7m3s)  nginx-ingress-controller  ...

```

Furthermore, observing the event log that shows syncing activity by the Ingress controller is crucial. Any warnings or errors in this log can provide insights into potential issues during the synchronization process.

To address the problem, ensure that the specified Services in the Ingress rules actually exist and are accessible within the Kubernetes cluster. Additionally, review the event log for any relevant messages that might indicate the cause of the discrepancy.

Let's resolve the issue of not being able to route to the backends configured in the Ingress object. The following commands create the Pods and Services:

```

$ kubectl run app --image=k8s.gcr.io/echoserver:1.10 --port=8080 \
-l app=app-service
pod/app created
$ kubectl run metrics --image=k8s.gcr.io/echoserver:1.10 --port=8080 \
-l app=metrics-service
pod/metrics created
$ kubectl create service clusterip app-service --tcp=8080:8080
service/app-service created
$ kubectl create service clusterip metrics-service --tcp=9090:8080
service/metrics-service created

```

Inspecting the Ingress object doesn't show any errors for the configured rules. If you're now able to see a list of resolvable backends along with the corresponding Pod virtual IP addresses and ports, the Ingress object is correctly configured, and the backends are recognized and accessible:

```

$ kubectl describe ingress next-app
Name:          next-app
Labels:        <none>
Namespace:    default
Address:      192.168.66.4
Ingress Class: nginx
Default backend: <default>
Rules:
  Host          Path  Backends
  ----          ----  -----
  next.example.com
                  /app      app-service:8080 (10.244.0.6:8080)
                  /metrics  metrics-service:9090 (10.244.0.7:8080)
Annotations: <none>
Events:

```

Type	Reason	Age	From	Message
----	-----	---	----	-----
Normal	Sync	13m (x2 over 13m)	nginx-ingress-controller	Scheduled for sync

It's worth coming back to the Ingress details if you experience any issues with routing traffic through an Ingress endpoint.

Accessing an Ingress

To enable the routing of incoming HTTP(S) traffic through the Ingress and subsequently to the configured Service, it's crucial to set up a DNS entry mapping to the external address. This typically involves configuring either an A record or a CNAME record. The [ExternalDNS project](#) is a valuable tool that can assist in managing these DNS records automatically.

For local testing on a Kubernetes cluster on your machine, follow these steps:

1. Find the IP address of the load balancer used by the Ingress.
2. Add the IP address to hostname mapping to your `/etc/hosts` file.

By adding the IP address to your local `/etc/hosts` file, you simulate the DNS resolution locally, allowing you to test the behavior of the Ingress without relying on actual DNS records:

```
$ kubectl get ingress next-app \
  --output=jsonpath=".status.loadBalancer.ingress[0]['ip']"
192.168.66.4
$ sudo vim /etc/hosts
...
192.168.66.4    next-app
```

You can now send HTTP requests to the backend. This call matches the `Exact` path rule and therefore returns a HTTP 200 response code from the application:

```
$ wget next.example.com/app --timeout=5 --tries=1
--2021-11-30 19:34:57-- http://next.example.com/app
Resolving next.example.com (next.example.com)... 192.168.66.4
Connecting to next.example.com (next.example.com)|192.168.66.4|:80... \
connected.
HTTP request sent, awaiting response... 200 OK
```

This next call uses a URL with a trailing slash. The Ingress path rule doesn't support this case, and therefore the call doesn't go through. You will receive a HTTP 404 response code. For the second call to work, you'd have to change the path rule to `Prefix`:

```
$ wget next.example.com/app/ --timeout=5 --tries=1
--2021-11-30 15:36:26-- http://next.example.com/app/
Resolving next.example.com (next.example.com)... 192.168.66.4
Connecting to next.example.com (next.example.com)|192.168.66.4|:80... \
```

```
connected.  
HTTP request sent, awaiting response... 404 Not Found  
2021-11-30 15:36:26 ERROR 404: Not Found.
```

You can observe the same behavior for the Metrics Service configured with the URL context path `metrics`. Feel free to try that out as well.

Summary

The resource type Ingress defines rules for routing cluster-external HTTP(S) traffic to one or many Services. Each rule defines a URL context path to target a Service. For an Ingress to work, you first need to install an Ingress controller. An Ingress controller periodically evaluates those rules and ensures that they apply to the cluster. To expose the Ingress, a cloud provider usually stands up an external load balancer that lends an external IP address to the Ingress.

Exam Essentials

Know the difference between a Service and an Ingress

An Ingress is not to be confused with a Service. The Ingress is meant for routing cluster-external HTTP(S) traffic to one or many Services based on an optional hostname and mandatory path. A Service routes traffic to a set of Pods.

Understand the role of an Ingress controller

An Ingress controller needs to be installed before an Ingress can function properly. Without installing an Ingress controller, Ingress rules will have no effect. You can choose from a range of Ingress controller implementations, all documented on the Kubernetes documentation page. Assume that an Ingress controller will be preinstalled for you in the exam environment.

Practice the definition of Ingress rules

You can define one or many rules in an Ingress. Every rule consists of an optional host, the URL context path, and the Service DNS name and port. Try defining more than a single rule and how to access the endpoint. You will not have to understand the process for configuring TLS termination for an Ingress—this aspect is covered by the CKS exam.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. Create a new Deployment named `web` that controls a single replica running the image `bmuschko/nodejs-hello-world:1.0.0` on port 3000. Expose the Deployment with a Service named `web` of type ClusterIP. The Service routes traffic to

the Pods controlled by the Deployment `web`. Make a request to the endpoint of the application on the context path `/`. You should see the message “Hello World.”

Create an Ingress that exposes the path `/` for the host `hello-world.exposed`. The traffic should be routed to the Service created earlier. List the Ingress object.

Add an entry in `/etc/hosts` that maps the load balancer IP address to the host `hello-world.exposed`. Make a request to `http://hello-world.exposed`. You should see the message “Hello World.”

2. Any application has been exposed by an Ingress. Some of your end users report an issue with connecting to the application from outside of the cluster. Inspect the existing setup and fix the problem for your end users.

Navigate to the directory `app-a/ch22/troubleshooting` of the checked-out GitHub repository [`bmuschko/ckad-study-guide`](#). Create the objects from the YAML manifest `setup.yaml`. Inspect the objects in the namespace `s96`. Create an entry in `/etc/hosts` for the hostname `faulty.ingress.com`.

Perform a HTTP call to `faulty.ingress.com/` using `wget` or `curl`. Inspect the connection error.

Change the configuration to ensure that end users can connect to the Ingress. Verify proper connectivity by performing another HTTP call.

Network Policies

The uniqueness of the IP address assigned to a Pod is maintained across all nodes and namespaces. This is accomplished by allocating a dedicated subnet to each registered node during its creation. The Container Network Interface (CNI) plugin handles the leasing of IP addresses from the assigned subnet when a new Pod is created on a node. Consequently, Pods on a node can seamlessly communicate with all other Pods running on any node within the cluster.

Network policies in Kubernetes function similarly to firewall rules, specifically designed for governing Pod-to-Pod communication. These policies include rules specifying the direction of network traffic (ingress and/or egress) for one or multiple Pods within a namespace or across different namespaces. Additionally, these rules define the targeted ports for communication. This fine-grained control enhances security and governs the flow of traffic within the Kubernetes cluster.

Coverage of Curriculum Objectives

This chapter addresses the following curriculum objective:

- Demonstrate basic understanding of Network Policies

Working with Network Policies

Within a Kubernetes cluster, any Pod can talk to any other Pod without restrictions using its **IP address or DNS name**, even across namespaces. Not only does unrestricted inter-Pod communication pose a potential security risk, it also makes it harder to understand the mental communication model of your architecture. A network policy defines the rules that control traffic from and to a Pod, as illustrated in [Figure 23-1](#).

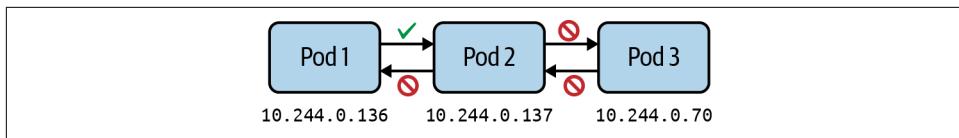


Figure 23-1. Network policies define traffic from and to a Pod

For example, there's no good reason to allow a backend application running in a Pod to talk directly to the frontend application running in another Pod. The communication should be directed from the frontend Pod to the backend Pod.

Installing an Network Policy Controller

A network policy cannot work without a network policy controller. The network policy controller evaluates the collection of rules defined by a network policy. You can find instructions for a wide range of network policy controllers in the [Kubernetes documentation](#).

Cilium is a CNI that implements a network policy controller. You can install Cilium on cloud provider and on-prem Kubernetes clusters. Refer to the [installation instructions](#) for detailed information. Once it is installed, you should find at least two Pods running Cilium and the Cilium Operator in the `kube-system` namespace:

```
$ kubectl get pods -n kube-system
NAME           READY   STATUS    RESTARTS   AGE
cilium-k5td6   1/1     Running   0          110s
cilium-operator-f5cdcc8d-njfbk   1/1     Running   0          110s
```

You can now assume that the rules defined by network policy objects will be evaluated. Additionally, you can use the Cilium command line tool to validate the proper installation.

Creating a Network Policy

Label selection plays a crucial role in defining which Pods a network policy applies to. We already saw the concept in action in other contexts (e.g., the [Deployment](#) and the [Service](#)). Furthermore, a network policy defines the direction of the traffic, to allow or disallow. In the context of a network policy, incoming traffic is called *ingress*, and outgoing traffic is called *egress*. For ingress and egress, you can whitelist the sources of traffic like Pods, IP addresses, or ports.



Network policies do not apply to Services

In most cases, you'd set up Service objects to funnel network traffic to Pods based on label and port selection. Network policies do not involve Services at all. All rules are namespace- and Pod-specific.

The creation of network policies is best explained by example. Let's say you're dealing with the following scenario: you're running a Pod that exposes an API to other consumers. For example, a Pod that processes payments for other applications. The company you're working for is migrating applications from a legacy payment processor to a new one. Therefore, you'll want to allow access only from the applications that are capable of properly communicating with it. Right now, you have two consumers—a grocery store and a coffee shop—each running their application in a separate Pod. The coffee shop is ready to consume the API of the payment processor, but the grocery store isn't. [Figure 23-2](#) shows the Pods and their assigned labels.

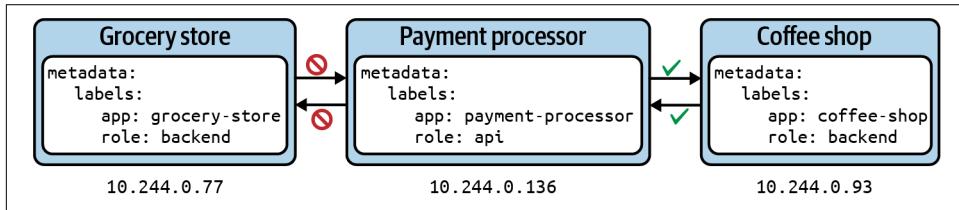


Figure 23-2. Limiting traffic to and from a Pod

Before creating a network policy, we'll stand up the Pods to represent the scenario:

```
$ kubectl run grocery-store --image=nginx:1.25.3-alpine \
-l app=grocery-store,role=backend --port 80
pod/grocery-store created
$ kubectl run payment-processor --image=nginx:1.25.3-alpine \
-l app=payment-processor,role=api --port 80
pod/payment-processor created
$ kubectl run coffee-shop --image=nginx:1.25.3-alpine \
-l app=coffee-shop,role=backend --port 80
```

Given Kubernetes' default behavior of allowing unrestricted Pod-to-Pod communication, the three Pods will be able to talk to one another. The following commands verify the behavior. The grocery store and coffee shop Pods perform a `wget` call to the payment processor Pod's IP address:

```
$ kubectl get pod payment-processor --template '{{.status.podIP}}'
10.244.0.136
$ kubectl exec grocery-store -it -- wget --spider --timeout=1 10.244.0.136
Connecting to 10.244.0.136 (10.244.0.136:80)
remote file exists
$ kubectl exec coffee-shop -it -- wget --spider --timeout=1 10.244.0.136
Connecting to 10.244.0.136 (10.244.0.136:80)
remote file exists
```

You cannot create a new network policy with the imperative `create` command. Instead, you will have to use the declarative approach. The YAML manifest in [Example 23-1](#), stored in the file `networkpolicy-api-allow.yaml`, shows a network policy for the scenario described previously.

Example 23-1. Declaring a NetworkPolicy with YAML

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: api-allow
spec:
  podSelector: ❶
    matchLabels:
      app: payment-processor
      role: api
  ingress: ❷
  - from:
    - podSelector:
        matchLabels:
          app: coffee-shop
```

- ❶ Selects the Pod the policy should apply to by label selection.
- ❷ Allows incoming traffic from the Pod with matching labels within the same namespace.

A network policy defines a couple of important attributes, which together form its set of rules. [Table 23-1](#) shows the attributes on the `spec` level.

Table 23-1. Spec attributes of a network policy

Attribute	Description
podSelector	Selects the Pods in the namespace to apply the network policy to.
policyTypes	Defines the type of traffic (i.e., ingress and/or egress) the network policy applies to.
ingress	Lists the rules for incoming traffic. Each rule can define <code>from</code> and <code>ports</code> sections.
egress	Lists the rules for outgoing traffic. Each rule can define <code>to</code> and <code>ports</code> sections.

You can specify ingress and egress rules independently using `spec.ingress.from[]` and `spec.egress.to[]`. Each rule consists of a Pod selector, an optional namespace selector, or a combination of both. [Table 23-2](#) lists the relevant attributes for the `to` and `from` selectors.

Table 23-2. Attributes of a network policy to and from selectors

Attribute	Description
podSelector	Selects Pods by label(s) in the same namespace as the network policy that should be allowed as ingress sources or egress destinations.
namespaceSelector	Selects namespaces by label(s) for which all Pods should be allowed as ingress sources or egress destinations.

Attribute	Description
namespaceSelector and podSelector	Selects Pods by label(s) within namespaces by label(s).

Let's see the effect of the network policy in action. Create the network policy object from the manifest:

```
$ kubectl apply -f networkpolicy-api-allow.yaml
networkpolicy.networking.k8s.io/api-allow created
```

The network policy prevents calling the payment processor from the grocery store Pod. Accessing the payment processor from the coffee shop Pod works perfectly, as the network policy's Pod selector matches the Pod's assigned label `app=coffee-shop`:

```
kubectl exec grocery-store -it -- wget --spider --timeout=1 10.244.0.136
Connecting to 10.244.0.136 (10.244.0.136:80)
wget: download timed out
command terminated with exit code 1
$ kubectl exec coffee-shop -it -- wget --spider --timeout=1 10.244.0.136
Connecting to 10.244.0.136 (10.244.0.136:80)
remote file exists
```

As a developer, you may be dealing with network policies that have been set up for you by other team members or administrators. You need to know about the `kubectl` commands for listing and inspecting network policy objects to understand their effects on the directional network traffic between microservices.

Listing Network Policies

Listing network policies works the same as any other Kubernetes primitive. Use the `get` command in combination with the resource type `networkpolicy`, or its short-form, `netpol`. For the previous network policy, you see a table that renders the name and Pod selector:

```
$ kubectl get networkpolicy api-allow
NAME      POD-SELECTOR          AGE
api-allow  app=payment-processor,role=api  83m
```

It's unfortunate that the output of the command doesn't give a lot of information about the ingress and egress rules. To retrieve more information, you have to dig into the details.

Rendering Network Policy Details

You can inspect the details of a network policy using the `describe` command. The output renders all the important information: Pod selector, and ingress and egress rules:

```
$ kubectl describe networkpolicy api-allow
Name:      api-allow
Namespace: default
Created on: 2024-01-10 09:06:59 -0700 MST
Labels:    <none>
Annotations: <none>
Spec:
  PodSelector: app=payment-processor,role=api
  Allowing ingress traffic:
    To Port: <any> (traffic allowed to all ports)
    From:
      PodSelector: app=coffee-shop
  Not affecting egress traffic
  Policy Types: Ingress
```

The network policy details don't draw a clear picture of the Pods that have been selected based on its rules. You can create Pods that match the rules and do not match the rules to verify the network policy's desired behavior.



Visualizing network policies

Defining the rules of network policies correctly can be challenging. The page [networkpolicy.io](#) provides a visual editor for network policies that renders a graphical representation in the browser.

As explained earlier, every Pod can talk to other Pods running on any node of the cluster, which exposes a potential security risk. An attacker able to gain access to a Pod theoretically can try to compromise another Pod by communicating with it by its virtual IP address.

Applying Default Network Policies

The principle of least privilege is a fundamental security concept, and it's highly recommended when it comes to restricting Pod-to-Pod network traffic in Kubernetes. The idea is to initially disallow all traffic and then selectively open up only the necessary connections based on the application's architecture and communication requirements.

You can lock down Pod-to-Pod communication with the help of a [default network policy](#). Default network policies are custom policies set up by administrators to enforce restrictive communication patterns by default.

To demonstrate the functionality of such a default network policy, we'll set up two Pods in the namespace `internal-tools`. Within the namespace, all Pods will be able to communicate with each other:

```

$ kubectl create namespace internal-tools
namespace/internal-tools created
$ kubectl run metrics-api --image=nginx:1.25.3-alpine --port=80 \
    -l app=api -n internal-tools
pod/metrics-api created
$ kubectl run metrics-consumer --image=nginx:1.25.3-alpine --port=80 \
    -l app=consumer -n internal-tools
pod/metrics-consumer created

```

Let's create a default network policy that denies all ingress and egress network traffic in the namespace. We'll store the network policy in the file *networkpolicy-deny-all.yaml*.

Example 23-2. Disallowing all traffic with the default policy

```

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
  namespace: internal-tools
spec:
  podSelector: {}          ①
  policyTypes:             ②
    - Ingress              ②
    - Egress               ②

```

- ① The curly braces for `spec.podSelector` mean “apply to all Pods in the namespace.”
- ② Defines the types of traffic the rule should apply to, in this case ingress and egress traffic.

Create the network policy from the manifest:

```

$ kubectl apply -f networkpolicy-deny-all.yaml
networkpolicy.networking.k8s.io/default-deny-all created

```

The network policy prevents any network communication between the Pods in the `internal-tools` namespace will, as shown here:

```

$ kubectl get pod metrics-api --template '{{.status.podIP}}' -n internal-tools
10.244.0.182
$ kubectl exec metrics-consumer -it -n internal-tools \
    -- wget --spider --timeout=1 10.244.0.182
Connecting to 10.244.0.182 (10.244.0.182:80)
wget: download timed out
command terminated with exit code 1
$ kubectl get pod metrics-consumer --template '{{.status.podIP}}' \
    -n internal-tools
10.244.0.70
$ kubectl exec metrics-api -it -n internal-tools \

```

```
-- wget --spider --timeout=1 10.244.0.70
Connecting to 10.244.0.70 (10.244.0.70:80)
wget: download timed out
command terminated with exit code 1
```

With those default deny constraints in place, you can define more detailed rules and loosen restrictions gradually. Network policies are additive. It's common practice to now set up additional network policies that will open up directional traffic, but only the ones that are really required.

Restricting Access to Specific Ports

Controlling access at the port level is a critical aspect of network security in Kubernetes. If not explicitly defined by a network policy, all ports are accessible, which can pose security risks. For instance, if you have an application running in a Pod that exposes port 80 to the outside world, leaving all other ports open widens the attack vector unnecessarily. Port rules can be specified for ingress and egress as part of a network policy. The definition of a network policy in [Example 23-3](#) allows access on port 80.

Example 23-3. Definition of a network policy allowing ingress access on port 8080

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: port-allow
  namespace: internal-tools
spec:
  podSelector:
    matchLabels:
      app: api
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: consumer
  ports:
  - protocol: TCP
    port: 80
```

①
①
①

- ❶ Only allows incoming traffic on port 80.

When defining network policies, only allow those ports that are required for implementing your architectural needs. All other ports should be locked down.

Summary

Intra-Pod communication or communication between two containers of the same Pod is completely unrestricted in Kubernetes. Network policies instate rules to control the network traffic either from or to a Pod. You can think of network policies as firewall rules for Pods. It's best practice to start with a "deny all traffic" rule to minimize the attack vector.

From there, you can open access as needed. Learning about the intricacies of network policies requires a bit of hands-on practice, as it is not directly apparent if the rules work as expected.

Exam Essentials

Understand the purpose and effects of network policies

By default, Pod-to-Pod communication is unrestricted. Instantiate a default deny rule to restrict Pod-to-Pod network traffic with the principle of least privilege. The attribute `spec.podSelector` of a network policy selects the target Pod the rules apply to based on label selection. The ingress and egress rules define Pods, namespaces, IP addresses, and ports for allowing incoming and outgoing traffic.

Know how to implement the principle of least privilege

Network policies can be aggregated. A default deny rule can disallow ingress and/or egress traffic. An additional network policy can open up those rules with a more fine-grained definition.

Explore common network policy scenarios

To explore common scenarios, look at the GitHub repository named "[Kubernetes Network Policy Recipes](#)". The repository comes with a visual representation for each scenario and walks you through the steps to set up the network policy and the involved Pods. This is a great practice resource.

Sample Exercises

Solutions to these exercises are available in [Appendix A](#).

1. You have been tasked with setting up a network policy for an existing application stack that consists of a frontend Pod in the namespace `end-user` and a backend Pod in the namespace `internal`.

Navigate to the directory `app-a/ch23/app-stack` of the checked-out GitHub repository [bmuschko/ckad-study-guide](#). Create the objects from the YAML manifest `setup.yaml`. Inspect the objects in both namespaces.

Create a network policy named `app-stack` in the `end-user` namespace. Allow egress traffic only from the `frontend` Pod to the `backend` Pod. The `backend` Pod should be reachable only on port 80.

2. Navigate to the directory `app-a/ch23/troubleshooting` of the checked-out GitHub repository [`bmuschko/ckad-study-guide`](#). Create the objects from the YAML manifest `setup.yaml`. Inspect the objects in the namespaces `k1` and `k2`.

Determine the virtual IP address of Pod `nginx` in namespace `k2`. Try to make a `wget` call on port 80 from the Pod `busybox` in namespace `k1` to the Pod `nginx` in namespace `k2`. The call will fail with the current setup.

Create a network policy that allows performing ingress calls for all Pods in namespace `k1` to the Pod `nginx` in namespace `k2`. Pods in all other namespaces should be denied to make ingress calls to Pods in namespace `k2`.

Verify that a network connection can be established.

APPENDIX A

Answers to Review Questions

Chapter 4, Containers

1. The given *Dockerfile* builds a nodejs-based application. All files necessary to run the application are available in the same directory. Upon further inspection, you will find that the *Dockerfile* exposes port 3000.

Build the container image with the following command:

```
$ docker build -t nodejs-hello-world:1.0.0 .
```

You will be able to find the container image by listing it:

```
$ docker images
REPOSITORY          TAG      IMAGE ID      CREATED        SIZE
nodejs-hello-world  1.0.0   0cc723ca8b06  15 seconds ago  180MB
```

Run the container in detached mode with the following command. Make sure to map port 80 to the exposed container port 3000:

```
$ docker run -d -p 80:3000 nodejs-hello-world:1.0.0
9e0f1abcef415e902422117de7644544cdd08ae158a1cd0b2a2d182fcf056cab
```

You can discover details about the container by listing them:

```
$ docker container ls
CONTAINER ID   IMAGE                  COMMAND           ...
9e0f1abcef41   nodejs-hello-world:1.0.0  "docker-entrypoint.s..." ...
```

You can now access the application on port 80 with either curl or wget:

```
$ curl localhost
Hello World
$ wget localhost
```

```
--2023-05-09 08:38:30-- http://localhost/
Resolving localhost (localhost)... ::1, 127.0.0.1
```

```
Connecting to localhost (localhost)|::1|:80... connected.  
HTTP request sent, awaiting response... 200 OK  
...  
2023-05-09 08:38:30 (2.29 MB/s) - ‘index.html’ saved [12/12]
```

You can retrieve logs written by the application with the following command:

```
$ docker logs 9e0f1abcef41  
Magic happens on port 3000
```

2. Change the first two lines of the *Dockerfile* as follows:

```
FROM node:20.4-alpine  
WORKDIR /node  
...
```

Build the container image with the following command:

```
$ docker build -t nodejs-hello-world:1.1.0 .
```

You will be able to find the container image by listing it. The size of the container image slightly increased by 1 MB:

```
$ docker images  
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE  
nodejs-hello-world  1.1.0  d332031cb5b6  4 seconds ago  181MB
```

3. Download the container image using the `docker pull` command:

```
$ docker pull alpine:3.18.2
```

The container image will now be available:

```
$ docker images alpine  
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE  
alpine          3.18.2   c1aabb73d233  4 weeks ago   7.33MB
```

To save the container image to a file, run the following command:

```
$ docker save -o alpine-3.18.2.tar alpine:3.18.2  
$ ls  
alpine-3.18.2.tar
```

Remove the container image locally:

```
$ docker image rm alpine:3.18.2  
Untagged: alpine:3.18.2  
Untagged: alpine@sha256:82d1e9d7ed48a7523bdebc18cf6290bdb97b82302a8a9 \\\n  c27d4fe885949ea94d1  
Deleted: sha256:c1aabb73d2339c5ebaa3681de2e9d9c18d57485045a4e311d9f80 \\\n  04bec208d67  
Deleted: sha256:78a822fe2a2d2c84f3de4a403188c45f623017d6a4521d23047c9 \\\n  fbb0801794c
```

The container image cannot be listed anymore:

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

Loading a container image from a file can be achieved with the following command:

```
$ docker load --input alpine-3.18.2.tar
78a822fe2a2d: Loading layer [=====] 7.622MB/7.622MB
Loaded image: alpine:3.18.2
```

The container image has been reinstated from the TAR file:

```
$ docker images alpine
REPOSITORY      TAG          IMAGE ID      CREATED        SIZE
alpine          3.18.2       c1aabb73d233   4 weeks ago   7.33MB
```

Chapter 5, Pods and Namespaces

1. You can either use the imperative approach or the declarative approach. First, we'll look at creating the namespace with the imperative approach:

```
$ kubectl create namespace ckad
```

Create the Pod:

```
$ kubectl run nginx --image=nginx:1.17.10 --port=80 --namespace=ckad
```

Alternatively, you can use the declarative approach. Create a new YAML manifest in the file called *ckad-namespace.yaml* with the following contents:

```
apiVersion: v1
kind: Namespace
metadata:
  name: ckad
```

Create the namespace from the YAML manifest:

```
$ kubectl apply -f ckad-namespace.yaml
```

Create a new YAML manifest in the file *nginx-pod.yaml* with the following contents:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.17.10
      ports:
        - containerPort: 80
```

Create the Pod from the YAML manifest:

```
$ kubectl apply -f nginx-pod.yaml --namespace=ckad
```

You can use the command-line option `-o wide` to retrieve the IP address of the Pod:

```
$ kubectl get pod nginx --namespace=ckad -o wide
```

The same information is available if you query for the Pod details:

```
$ kubectl describe pod nginx --namespace=ckad | grep IP:
```

You can use the command-line options `--rm` and `-it` to start a temporary Pod. The following command assumes that the IP address of the Pod named `nginx` is `10.1.0.66`:

```
$ kubectl run busybox --image=busybox:1.36.1 --restart=Never --rm -it \
-n ckad -- wget -O- 10.1.0.66:80
```

To download the logs, use a simple `logs` command:

```
$ kubectl logs nginx --namespace=ckad
```

Editing the live object is forbidden. You will receive an error message if you try to add the environment variables:

```
$ kubectl edit pod nginx --namespace=ckad
```

You will have to re-create the object with a modified YAML manifest, but first you'll have to delete the existing object:

```
$ kubectl delete pod nginx --namespace=ckad
```

Edit the existing YAML manifest in the file `nginx-pod.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.17.10
      ports:
        - containerPort: 80
      env:
        - name: DB_URL
          value: postgresql://mydb:5432
        - name: DB_USERNAME
          value: admin
```

Apply the changes:

```
$ kubectl apply -f nginx-pod.yaml --namespace=ckad
```

Use the `exec` command to open an interactive shell to the container:

```
$ kubectl exec -it nginx --namespace=ckad -- /bin/sh
# ls -l
```

- Combine the command-line options `-o yaml` and `--dry-run=client` to write the generated YAML to a file. Make sure to escape the double-quote characters of the string rendered by the echo command:

```
$ kubectl run loop --image=busybox:1.36.1 -o yaml --dry-run=client \
--restart=Never -- /bin/sh -c 'for i in 1 2 3 4 5 6 7 8 9 10; \
do echo "Welcome $i times"; done' \
> pod.yaml
```

Create the Pod from the YAML manifest:

```
$ kubectl apply -f pod.yaml --namespace=ckad
```

The status of the Pod will say `Completed`, as the executed command in the container does not run in an infinite loop:

```
$ kubectl get pod loop --namespace=ckad
```

The container command cannot be changed for existing Pods. Delete the Pod so you can modify the manifest file and re-create the object:

```
$ kubectl delete pod loop --namespace=ckad
```

Change the YAML manifest content:

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: loop
    name: loop
spec:
  containers:
  - args:
    - /bin/sh
    - -c
    - while true; do date; sleep 10; done
    image: busybox:1.36.1
    name: loop
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}
```

Create the Pod from the YAML manifest:

```
$ kubectl apply -f pod.yaml --namespace=ckad
```

You can describe the Pod events by grepping for the term:

```
$ kubectl describe pod loop --namespace=ckad | grep -C 10 Events:
```

You can simply delete the namespace, which will delete all objects within the namespace:

```
$ kubectl delete namespace ckad
```

Chapter 6, Jobs and CronJobs

1. Start by creating a YAML manifest file named `random-hash-job.yaml`. The contents of the file could look like this:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: random-hash
spec:
  parallelism: 2
  completions: 5
  backoffLimit: 4
  template:
    spec:
      containers:
        - name: random-hash
          image: alpine:3.17.3
          command: ["/bin/sh", "-c", "echo $RANDOM | base64 | head -c 20"]
      restartPolicy: Never
```

Create the Job from the YAML manifest:

```
$ kubectl apply -f random-hash-job.yaml
job.batch/random-hash created
```

The result will be five Pods that correspond to the number of completions. You can combine the `kubectl` command with the `grep` command to easily find the Pods that are controlled by the job named `random-hash`:

```
$ kubectl get pods | grep "random-hash-"
NAME           READY   STATUS    RESTARTS   AGE
random-hash-4qk96  0/1    Completed  0          46s
random-hash-ld2sl  0/1    Completed  0          39s
random-hash-xcmts 0/1    Completed  0          35s
random-hash-xxlhk  0/1    Completed  0          46s
random-hash-z9xc4  0/1    Completed  0          39s
```

You can pick one of the Pods by name and get its logs. The downloaded logs will contain the generated hash:

```
$ kubectl logs random-hash-4qk96
MTgxMTIK
```

Deleting the Job will also delete the Pods controlled by the Job:

```
$ kubectl delete job random-hash
job.batch "random-hash" deleted
$ kubectl get pods | grep -E "random-hash-" -c
0
```

2. You can use the image `nginx:1.25.1`, which has the command-line tool `curl` installed. The Unix cron expression for this job is `*/2 * * * *`:

```
$ kubectl create cronjob google-ping --schedule="*/2 * * * *" \
--image=nginx:1.25.1 -- /bin/sh -c 'curl google.com'
cronjob.batch/google-ping created
```

You can inspect when a CronJob is executed using the `-w` command-line option:

```
$ kubectl get cronjob -w
NAME      SCHEDULE      SUSPEND      ACTIVE      LAST SCHEDULE      AGE
google-ping  */2 * * * *  False        0           115s        2m10s
google-ping  */2 * * * *  False        1           6s          2m21s
google-ping  */2 * * * *  False        0           16s         2m31s
google-ping  */2 * * * *  False        1           6s          4m21s
google-ping  */2 * * * *  False        0           16s         4m31s
```

Explicitly assign the value 7 to the `spec.successfulJobsHistoryLimit` attribute of the live object. The resulting YAML manifest should have the following configuration:

```
...
spec:
  successfulJobsHistoryLimit: 7
...
```

Edit the default value of `spec.concurrencyPolicy` of the live object. The resulting YAML manifest should have the following configuration:

```
...
spec:
  concurrencyPolicy: Forbid
...
```

Chapter 7, Volumes

1. Start by generating the YAML manifest using the `run` command in combination with the `--dry-run` option:

```
$ kubectl run alpine --image=alpine:3.12.0 --dry-run=client \
--restart=Never -o yaml -- /bin/sh -c "while true; do sleep 60; \
done;" > multi-container-alpine.yaml
$ vim multi-container-alpine.yaml
```

After editing the Pod, the manifest could look like the following. The container names here are `container1` and `container2`:

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: alpine
    name: alpine
spec:
  containers:
```

```

- args:
  - /bin/sh
  - -c
  - while true; do sleep 60; done;
  image: alpine:3.12.0
  name: container1
  resources: {}
- args:
  - /bin/sh
  - -c
  - while true; do sleep 60; done;
  image: alpine:3.12.0
  name: container2
  resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}

```

Edit the YAML manifest further by adding the Volume and the mount paths for both containers.

In the end, the Pod definition could look like this:

```

apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: alpine
    name: alpine
spec:
  volumes:
    - name: shared-vol
      emptyDir: {}
  containers:
    - args:
        - /bin/sh
        - -c
        - while true; do sleep 60; done;
      image: alpine:3.12.0
      name: container1
      volumeMounts:
        - name: shared-vol
          mountPath: /etc/a
      resources: {}
    - args:
        - /bin/sh
        - -c
        - while true; do sleep 60; done;
      image: alpine:3.12.0
      name: container2
      volumeMounts:
        - name: shared-vol

```

```
    mountPath: /etc/b
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
  status: {}
```

Create the Pod and check if it has been created properly. You should see the Pod in `Running` status with two containers ready:

```
$ kubectl apply -f multi-container-alpine.yaml
pod/alpine created
$ kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
alpine   2/2     Running   0          18s
```

Use the `exec` command to shell into the container named `container1`. Create the file `/etc/a/data/hello.txt` with the relevant content:

```
$ kubectl exec alpine -c container1 -it -- /bin/sh
/ # cd /etc/a
/etc/a # ls -l
total 0
/etc/a # mkdir data
/etc/a # cd data/
/etc/a/data # echo "Hello World" > hello.txt
/etc/a/data # cat hello.txt
Hello World
/etc/a/data # exit
```

Use the `exec` command to shell into the container named `container2`. The contents of the file `/etc/b/data/hello.txt` should say “Hello World”:

```
$ kubectl exec alpine -c container2 -it -- /bin/sh
/ # cat /etc/b/data/hello.txt
Hello World
/ # exit
```

2. Start by creating a new file named `logs-pv.yaml`. The contents could look as follows:

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: logs-pv
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
    - ReadOnlyMany
  hostPath:
    path: /var/logs
```

Create the PersistentVolume object and check its status:

```
$ kubectl create -f logs-pv.yaml
persistentvolume/logs-pv created
$ kubectl get pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM \
STORAGECLASS   REASON   AGE
logs-pv      5Gi        RWO,ROX    Retain           Available \
                                         18s
```

Create the file *logs-pvc.yaml* to define the PersistentVolumeClaim. The following YAML manifest shows its contents:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: logs-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
  storageClassName: ""
```

Create the PersistentVolume object and check on its status:

```
$ kubectl create -f logs-pvc.yaml
persistentvolumeclaim/logs-pvc created
$ kubectl get pvc
NAME      STATUS   VOLUME                                     CAPACITY \
ACCESS MODES   STORAGECLASS   AGE
logs-pvc  Bound    pvc-47ac2593-2cd2-4213-9e31-450bc98bb43f  2Gi \
RWO          standard    11s
```

Create the basic YAML manifest using the `--dry-run` command-line option:

```
$ kubectl run nginx --image=nginx:1.25.1 --dry-run=client \
-o yaml > nginx-pod.yaml
```

Now, edit the file *nginx-pod.yaml* and bind the PersistentVolumeClaim to it:

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginx
    name: nginx
spec:
  volumes:
    - name: logs-volume
      persistentVolumeClaim:
        claimName: logs-pvc
  containers:
    - image: nginx:1.25.1
      name: nginx
```

```

volumeMounts:
  - mountPath: "/var/log/nginx"
    name: logs-volume
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
  status: {}

```

Create the Pod using the following command and check its status:

```

$ kubectl apply -f nginx-pod.yaml
pod/nginx created
$ kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
nginx    1/1      Running   0          8s

```

Use the exec command to open an interactive shell to the Pod and create a file in the mounted directory:

```

$ kubectl exec nginx -it -- /bin/sh
# cd /var/log/nginx
# touch my-nginx.log
# ls
access.log  error.log  my-nginx.log
# exit

```

After you re-create the Pod, the file stored on the PersistentVolume should still exist:

```

$ kubectl delete pod nginx
$ kubectl apply -f nginx-pod.yaml
pod/nginx created
$ kubectl exec nginx -it -- /bin/sh
# cd /var/log/nginx
# ls
access.log  error.log  my-nginx.log
# exit

```

Chapter 8, Multi-Container Pods

1. You can start by generating the YAML manifest in dry-run mode. The resulting manifest will set up the main application container:

```
$ kubectl run complex-pod --image=nginx:1.25.1 --port=80 \
--restart=Never -o yaml --dry-run=client > complex-pod.yaml
```

Edit the manifest file by adding the init container and changing some of the default settings that have been generated. The finalized manifest could look like this:

```

apiVersion: v1
kind: Pod
metadata:
  name: complex-pod

```

```

spec:
  initContainers:
  - image: busybox:1.36.1
    name: setup
    command: ['sh', '-c', 'wget -O- google.com']
  containers:
  - image: nginx:1.25.1
    name: app
    ports:
    - containerPort: 80
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
  status: {}

```

Run the `create` command to instantiate the Pod. Verify that the Pod is running without issues:

```

$ kubectl apply -f complex-pod.yaml
pod/complex-pod created
$ kubectl get pod complex-pod
NAME      READY   STATUS    RESTARTS   AGE
complex-pod  1/1     Running   0          27s

```

Use the `logs` command and point it to the init container to download the log output:

```

$ kubectl logs complex-pod -c setup
Connecting to google.com (172.217.1.206:80)
Connecting to www.google.com (172.217.2.4:80)
writing to stdout
...

```

You can target the main application as well. Here you'll open an interactive shell and run the `ls` command:

```

$ kubectl exec complex-pod -it -c app -- /bin/sh
# ls
bin dev docker-entrypoint.sh home lib64 mnt proc run \
srv tmp var boot docker-entrypoint.d etclib media opt \
root sbin sys usr
# exit

```

Avoid graceful deletion of the Pod by adding the options `--grace-period=0` and `--force`:

```

$ kubectl delete pod complex-pod --grace-period=0 --force
warning: Immediate deletion does not wait for confirmation that the \
running resource has been terminated. The resource may continue to run \
on the cluster indefinitely.
pod "complex-pod" force deleted

```

2. You can start by generating the YAML manifest in dry-run mode. The resulting manifest will set up the main application container:

```
$ kubectl run data-exchange --image=busybox:1.36.1 --restart=Never \
-o yaml --dry-run=client > data-exchange.yaml
```

Edit the manifest file by adding the sidecar container and changing some of the default settings that have been generated. The finalized manifest could look like the following:

```
apiVersion: v1
kind: Pod
metadata:
  name: data-exchange
spec:
  containers:
    - image: busybox:1.36.1
      name: main-app
      command: ['sh', '-c', 'counter=1; while true; do touch \
                  "/var/app/data/$counter-data.txt"; counter=$((counter+1)); \
                  sleep 30; done']
      resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
  status: {}
```

Simply add the sidecar container alongside the main application container with the proper command. Add to the existing YAML manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: data-exchange
spec:
  containers:
    - image: busybox:1.36.1
      name: main-app
      command: ['sh', '-c', 'counter=1; while true; do touch \
                  "/var/app/data/$counter-data.txt"; counter=$((counter+1)); \
                  sleep 30; done']
      resources: {}
    - image: busybox:1.36.1
      name: sidecar
      command: ['sh', '-c', 'while true; do ls -dq /var/app/data/*-data.txt \
                  | wc -l; sleep 30; done']
  dnsPolicy: ClusterFirst
  restartPolicy: Never
  status: {}
```

Modify the manifest so that a Volume is used to exchange the files between the main application container and sidecar container:

```
apiVersion: v1
kind: Pod
metadata:
  name: data-exchange
spec:
```

```

containers:
- image: busybox:1.36.1
  name: main-app
  command: ['sh', '-c', 'counter=1; while true; do touch \
    "/var/app/data/$counter-data.txt"; counter=$((counter+1)); \
    sleep 30; done']
volumeMounts:
- name: data-dir
  mountPath: "/var/app/data"
resources: {}
- image: busybox:1.36.1
  name: sidecar
  command: ['sh', '-c', 'while true; do ls -d /var/app/data/*-data.txt \
    | wc -l; sleep 30; done']
volumeMounts:
- name: data-dir
  mountPath: "/var/app/data"
volumes:
- name: data-dir
  emptyDir: {}
dnsPolicy: ClusterFirst
restartPolicy: Never
status: {}

```

Create the Pod, check for its existence, and tail the logs of the sidecar container. The number of files will increment over time:

```

$ kubectl apply -f data-exchange.yaml
pod/data-exchange created
$ kubectl get pod data-exchange
NAME           READY   STATUS    RESTARTS   AGE
data-exchange   2/2     Running   0          31s
$ kubectl logs data-exchange -c sidecar -f
1
2
...

```

Delete the Pod:

```

$ kubectl delete pod data-exchange
pod "data-exchange" deleted

```

Chapter 9, Labels and Annotations

1. Start by creating the Pods. You can assign labels at the time of creation:

```

$ kubectl run pod-1 --image=nginx:1.25.1 \
  --labels=tier=frontend,team=artemidis
pod/pod-1 created
$ kubectl run pod-2 --image=nginx:1.25.1 \
  --labels=tier=backend,team=artemidis
pod/pod-2 created
$ kubectl run pod-3 --image=nginx:1.25.1 \

```

```
--labels=tier=backend,team=artemidis
pod/pod-3 created
$ kubectl get pods --show-labels
NAME    READY   STATUS    RESTARTS   AGE   LABELS
pod-1   1/1     Running   0          30s   team=artemidis,tier=frontend
pod-2   1/1     Running   0          24s   team=artemidis,tier=backend
pod-3   1/1     Running   0          16s   team=artemidis,tier=backend
```

You can either edit the live objects to add an annotation or use the `annotate` command. We'll use the imperative command here:

```
$ kubectl annotate pod pod-1 pod-3 deployer='Benjamin Muschko'
pod/pod-1 annotated
pod/pod-3 annotated
$ kubectl describe pod pod-1 pod-3 | grep Annotations:
Annotations: deployer: Benjamin Muschko
Annotations: deployer: Benjamin Muschko
```

The label selection requires you to combine equality- and set-based criteria to find the Pods:

```
$ kubectl get pods -l tier=backend,'team in (artemidis,aircontrol)' \
--show-labels
NAME    READY   STATUS    RESTARTS   AGE   LABELS
pod-2   1/1     Running   0          6m38s  team=artemidis,tier=backend
pod-3   1/1     Running   0          6m30s  team=artemidis,tier=backend
```

2. Define a Pod in the file `pod-well-known.yaml`. Add the reserved labels `app.kubernetes.io/name` and `app.kubernetes.io/managed-by`. The resulting YAML manifest could look like the following:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app.kubernetes.io/name: F5-nginx
    app.kubernetes.io/managed-by: helm
spec:
  containers:
  - image: nginx:1.25.1
    name: nginx
```

Create the object from the YAML manifest:

```
$ kubectl apply -f pod-well-known.yaml
pod/nginx created
```

To look at the assigned labels, use the `describe` command or the `get` command in combination with the `--show-labels` option:

```
$ kubectl describe pod nginx
...
Labels:           app.kubernetes.io/managed-by=helm
                  app.kubernetes.io/name=F5-nginx
```

```
...
$ kubectl get pod nginx --show-labels
NAME    READY   STATUS    RESTARTS   AGE    LABELS
nginx   1/1     Running   0          18s    app.kubernetes.io/\
managed-by=helm,app.kubernetes.io/name=F5-nginx
```

Chapter 10, Deployments

1. Create the YAML manifest for a Deployment in the file *nginx-deployment.yaml*. The label selector should match the labels assigned to the Pod template. The following code snippet shows the contents of the YAML manifest file:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    tier: backend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: v1
  template:
    metadata:
      labels:
        app: v1
    spec:
      containers:
        - image: nginx:1.23.0
          name: nginx
```

Create the deployment by pointing it to the YAML manifest. Check on the Deployment status:

```
$ kubectl apply -f nginx-deployment.yaml
deployment.apps/nginx created
$ kubectl get deployment nginx
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   3/3     3           3           10s
```

Set the new image and check the revision history:

```
$ kubectl set image deployment/nginx nginx=nginx:1.23.4
deployment.apps/nginx image updated
$ kubectl rollout history deployment nginx
deployment.apps/nginx
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
$ kubectl rollout history deployment nginx --revision=2
deployment.apps/nginx with revision #2
Pod Template:
```

```
Labels:      app=v1
            pod-template-hash=5bd95c598
Containers:
  nginx:
    Image:      nginx:1.23.4
    Port:       <none>
    Host Port: <none>
    Environment:  <none>
    Mounts:     <none>
    Volumes:    <none>
```

Add the change cause to the current revision by annotating the Deployment object:

```
$ kubectl annotate deployment nginx kubernetes.io/change-cause=\
"Pick up patch version"
deployment.apps/nginx annotated
```

The revision change cause can be inspected by rendering the rollout history:

```
$ kubectl rollout history deployment nginx
deployment.apps/nginx
REVISION  CHANGE-CAUSE
1        <none>
2        Pick up patch version
```

Now, scale the Deployment to 5 replicas. You should find 5 Pods controlled by the Deployment:

```
$ kubectl scale deployment nginx --replicas=5
deployment.apps/nginx scaled
$ kubectl get pod -l app=v1
NAME          READY   STATUS    RESTARTS   AGE
nginx-5bd95c598-25z4j  1/1     Running   0          3m39s
nginx-5bd95c598-46mlt  1/1     Running   0          3m38s
nginx-5bd95c598-bszvp  1/1     Running   0          48s
nginx-5bd95c598-dwr8r  1/1     Running   0          48s
nginx-5bd95c598-kjrvf  1/1     Running   0          3m37s
```

Roll back to revision 1. You will see the new revision. Inspecting the revision should show the image `nginx:1.23.0`:

```
$ kubectl rollout undo deployment/nginx --to-revision=1
deployment.apps/nginx rolled back
$ kubectl rollout history deployment nginx
deployment.apps/nginx
REVISION  CHANGE-CAUSE
2        Pick up patch version
3        <none>
$ kubectl rollout history deployment nginx --revision=3
deployment.apps/nginx with revision #3
Pod Template:
  Labels:      app=v1
                pod-template-hash=f48dc88cd
  Containers:
```

```
nginx:  
  Image:      nginx:1.23.0  
  Port:       <none>  
  Host Port: <none>  
  Environment: <none>  
  Mounts:     <none>  
  Volumes:    <none>
```

2. Define the Deployment in the file *nginx-deployment.yaml*, as shown:

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: nginx  
spec:  
  replicas: 1  
  selector:  
    matchLabels:  
      app: nginx  
  template:  
    metadata:  
      labels:  
        app: nginx  
    spec:  
      containers:  
        - image: nginx:1.23.4  
          name: nginx  
          resources:  
            requests:  
              cpu: "0.5"  
              memory: "500Mi"  
            limits:  
              memory: "500Mi"
```

Create the Deployment object from the manifest file:

```
$ kubectl apply -f nginx-deployment.yaml  
deployment.apps/nginx created
```

Ensure that all Pods controlled by the Deployment transition into “Running” status:

```
$ kubectl get deploy  
NAME      READY   UP-TO-DATE   AVAILABLE   AGE  
nginx    1/1     1           1           49s  
$ kubectl get pods  
NAME                  READY   STATUS    RESTARTS   AGE  
nginx-5bbd9746c-9b4np  1/1     Running   0          24s
```

Next, define the HorizontalPodAutoscaler with the given resource thresholds in the file *nginx-hpa.yaml*. The final manifest is shown here:

```
apiVersion: autoscaling/v2  
kind: HorizontalPodAutoscaler  
metadata:
```

```

name: nginx-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
  minReplicas: 3
  maxReplicas: 8
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 75
    - type: Resource
      resource:
        name: memory
        target:
          type: Utilization
          averageUtilization: 60

```

Create the HorizontalPodAutoscaler object from the manifest file:

```
$ kubectl apply -f nginx-hpa.yaml
horizontalpodautoscaler.autoscaling/nginx-hpa created
```

When you inspect the HorizontalPodAutoscaler object, you will find that the number of replicas will be scaled up to the minimum number 3 even though the Deployment defines only a single replica. At the time of running this command, Pods are not using a significant amount of CPU and memory. That's why the current metrics show 0%:

```
$ kubectl get hpa nginx-hpa
NAME      REFERENCE           TARGETS           MINPODS   MAXPODS \
REPLICAS   AGE
nginx-hpa  Deployment/nginx  0%/60%, 0%/75%  3         8         \ 
            2m19s
```

Chapter 11, Deployment Strategies

1. Create the Deployment object from the YAML manifest:

```
$ kubectl apply -f deployment-grafana.yaml
deployment.apps/grafana created
```

Inspect the created Deployment object and replicas. You should find six replicas:

```
$ kubectl get deployments,pods
NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/grafana        6/6     6           6           39s
NAME                         READY   STATUS    RESTARTS   AGE

```

pod/grafana-5f6b77b687-4h7bq	1/1	Running	0	39s
pod/grafana-5f6b77b687-88bnb	1/1	Running	0	39s
pod/grafana-5f6b77b687-97d6g	1/1	Running	0	39s
pod/grafana-5f6b77b687-h8mhq	1/1	Running	0	39s
pod/grafana-5f6b77b687-lfgcf	1/1	Running	0	39s
pod/grafana-5f6b77b687-v9nkq	1/1	Running	0	39s

Multiple changes need to be made to the live object of the Deployment. One of the easiest options is to open an editor and make changes interactively using the command `kubectl edit deployment grafana`. Alternatively, you can edit the Deployment YAML manifest directly and then run the command `kubectl apply -f deployment-grafana.yaml`. The modified YAML manifest could look like this:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: grafana
spec:
  replicas: 6
  strategy:
    rollingUpdate:
      maxSurge: 2
      maxUnavailable: 2
      type: RollingUpdate
    minReadySeconds: 20
  selector:
    matchLabels:
      app: grafana
  template:
    metadata:
      labels:
        app: grafana
    spec:
      containers:
        - image: grafana/grafana:10.1.2
          name: grafana
          ports:
            - containerPort: 3000
          readinessProbe:
            httpGet:
              path: /
              port: 3000

```

The `RollingUpdate` strategy will transition all replicas to the new container image tag. This process may take a little time. To watch the changes play out, run the command `kubectl get deployment grafana --watch`.

2. Define the initial Deployment in the file `blue-deployment.yaml`, as shown here:

```

apiVersion: apps/v1
kind: Deployment

```

```

metadata:
  name: nginx-blue
spec:
  replicas: 3
  selector:
    matchLabels:
      version: blue
  template:
    metadata:
      labels:
        version: blue
    spec:
      containers:
        - image: nginx:1.23.0
          name: nginx
          ports:
            - containerPort: 80

```

Create the Deployment object using the following command. Wait until all replicas transition into the “Running” status:

```

$ kubectl apply -f blue-deployment.yaml
deployment.apps/nginx-blue created
$ kubectl get pods -l version=blue
NAME           READY   STATUS    RESTARTS   AGE
nginx-blue-99f499479-h9wq4   1/1     Running   0          9s
nginx-blue-99f499479-trsjf   1/1     Running   0          9s
nginx-blue-99f499479-wndkg   1/1     Running   0          9s

```

Define the Service in the file `service.yaml`, as shown:

```

apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  selector:
    version: blue
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80

```

Create the Service object using the following command. Wait until all replicas transition into “Running” status:

```

$ kubectl apply -f service.yaml
service/nginx created

```

Now, check to ensure that the Pods can be reached using a `curl` command from a temporary Pod. The returned headers will include the nginx server version:

```

$ kubectl run tmp --image=alpine/curl:3.14 --restart=Never -it \
  --rm -- curl -sI nginx.default.svc.cluster.local | grep Server
Server: nginx/1.23.0

```

Create a second Deployment in the file *green-deployment.yaml*. Make sure to change the labels and the container image tag. The manifest is shown here:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-green
spec:
  replicas: 3
  selector:
    matchLabels:
      version: green
  template:
    metadata:
      labels:
        version: green
    spec:
      containers:
        - image: nginx:1.23.4
          name: nginx
          ports:
            - containerPort: 80
```

Create the Deployment object using the following command. Wait until all replicas transition into “Running” status:

```
$ kubectl apply -f green-deployment.yaml
deployment.apps/nginx-green created
$ kubectl get pods -l version=green
NAME                  READY   STATUS    RESTARTS   AGE
nginx-green-658cfdc9c6-8pvpp  1/1    Running   0          11s
nginx-green-658cfdc9c6-fdgm6  1/1    Running   0          11s
nginx-green-658cfdc9c6-zg6gl  1/1    Running   0          11s
```

Change the existing *service.yaml* file by changing the label value with the key `version` from `blue` to `green`:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  selector:
    version: green
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

Apply the changes to the Service object:

```
$ kubectl apply -f service.yaml
service/nginx configured
```

Delete the initial Deployment using the following command:

```
$ kubectl delete deployment nginx-blue
deployment.apps "nginx-blue" deleted
```

Incoming traffic to the Service endpoint should now be switched over to the Pods controlled by the green Deployment:

```
$ kubectl run tmp --image=alpine/curl:3.14 --restart=Never -it --rm \
-- curl -sI nginx.default.svc.cluster.local | grep Server
Server: nginx/1.23.4
```

Chapter 12, Helm

1. Add the Helm chart repository using the provided URL. The name assigned to the URL is `prometheus-community`:

```
$ helm repo add prometheus-community https://prometheus-community.\
github.io/helm-charts
"prometheus-community" has been added to your repositories
```

Update the chart information with the following command:

```
$ helm repo update prometheus-community
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "prometheus-community" \
chart repository
Update Complete. *Happy Helming!*
```

You can search published chart versions in the the repository named `prometheus-community`:

```
$ helm search hub prometheus-community
URL                                     CHART VERSION ...
https://artifacthub.io/packages/helm/prometheus...  45.28.1    ...
```

Install the latest version of the chart `kube-prometheus-stack`:

```
$ helm install prometheus prometheus-community/kube-prometheus-stack
NAME: prometheus
LAST DEPLOYED: Thu May 18 11:32:31 2023
NAMESPACE: default
STATUS: deployed
REVISION: 1
...
```

The installed charts can be listed with the following command:

```
$ helm list
NAME      NAMESPACE  REVISION  UPDATED    ...
prometheus  default     1        2023-05-18 ...
```

One of the objects created by the chart is the Service named `prometheus-operated`. This Service exposes the Prometheus dashboard on port 9090:

```
$ kubectl get service prometheus-operated
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP  ...
prometheus-operated  ClusterIP  None        <none>     ...
```

Set up port forwarding from port 8080 to port 9090:

```
$ kubectl port-forward service/prometheus-operated 8080:9090
Forwarding from 127.0.0.1:8080 -> 9090
Forwarding from [::1]:8080 -> 9090
```

Open a browser and enter the URL <http://localhost:8080/>. You will be presented with the Prometheus dashboard.

A screenshot of a web browser displaying the Prometheus dashboard at localhost. The dashboard has a dark header with the Prometheus logo and navigation links for Alerts, Graph, Status, and Help. Below the header are several configuration checkboxes: 'Use local time' (unchecked), 'Enable query history' (unchecked), 'Enable autocomplete' (checked), 'Enable highlighting' (checked), and 'Enable linter' (checked). A search bar contains the placeholder 'Expression (press Shift+Enter for newlines)'. Below the search bar are two tabs: 'Table' (selected) and 'Graph'. Underneath the tabs is a date range selector with 'Evaluation time' and arrows for navigating between dates. The main content area displays the message 'No data queried yet'. At the bottom right of the dashboard is a blue button labeled 'Remove Panel'. At the very bottom of the browser window, there is a blue button labeled 'Add Panel'.

Uninstall the chart with the following command:

```
$ helm uninstall prometheus
release "prometheus" uninstalled
```

Chapter 13, API Deprecations

1. This exercise verifies your comfort level with handling API deprecations. You will find that the API version `apps/v1beta2` assigned to the Deployment definition cannot be found when trying to create the objects:

```
$ kubectl apply -f ./configmap/data-config
configmap/data-config created
error: resource mapping not found for name: "nginx" namespace: "" \
from "deployment.yaml": no matches for kind "Deployment" in version \
"apps/v1beta2"
ensure CRDs are installed first
```

Don't let the error message mislead you. A Deployment is a built-in API primitive and therefore doesn't require you to install a CRD for it. Checking on the available API version confirms that `apps/v1beta2` indeed does not exist:

```
$ kubectl api-versions
admissionregistration.k8s.io/v1
apiextensions.k8s.io/v1
apiregistration.k8s.io/v1
apps/v1
authentication.k8s.io/v1
authorization.k8s.io/v1
autoscaling/v1
autoscaling/v2
batch/v1
certificates.k8s.io/v1
coordination.k8s.io/v1
discovery.k8s.io/v1
events.k8s.io/v1
flowcontrol.apiserver.k8s.io/v1beta2
flowcontrol.apiserver.k8s.io/v1beta3
networking.k8s.io/v1
node.k8s.io/v1
policy/v1
rbac.authorization.k8s.io/v1
scheduling.k8s.io/v1
storage.k8s.io/v1
v1
```

Checking on the [Kubernetes Blog](#), you will find that the API version `apps/v1beta2` has been removed with Kubernetes 1.16. The replacement API, `apps/v1`, has been introduced in Kubernetes 1.9:

Migrate to use the `apps/v1` API version, available since v1.9. Existing persisted data can be retrieved/updated via the new version.

—Deprecated API Migration Guide

Modify the Deployment manifest file by replacing the API version:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        run: app
    spec:
      containers:
        - image: nginx:1.23.4
          name: nginx
```

```
  ports:
    - containerPort: 80
  envFrom:
    - configMapRef:
        name: data-config
```

Trying to create the Deployment object reveals that you need to define a label selector with API version apps/v1:

```
$ kubectl apply -f ./
configmap/data-config unchanged
The Deployment "nginx" is invalid:
* spec.selector: Required value
* spec.template.metadata.labels: Invalid value: \
map[string]string{"run":"app"}: `selector` does \
not match template `labels`
```

Change the Deployment definition so that it selects the label of the Pod template:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      run: app
  template:
    metadata:
      labels:
        run: app
  spec:
    containers:
      - image: nginx:1.23.4
        name: nginx
        ports:
          - containerPort: 80
        envFrom:
          - configMapRef:
              name: data-config
```

Both objects can now be created:

```
$ kubectl apply -f ./
configmap/data-config unchanged
deployment.apps/nginx created
$ kubectl get deployments,configmaps
NAME                   READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx   2/2     2            2           45s

NAME                  DATA   AGE
configmap/data-config   2     19m
```

Chapter 14, Container Probes

1. You can start by generating the YAML manifest in dry-run mode. The resulting manifest will create the container with the proper image:

```
$ kubectl run web-server --image=nginx:1.23.0 --port=80 -o yaml \
--dry-run=client > probed-pod.yaml
```

Edit the manifest by defining a startup probe. The finalized manifest could look as follows:

```
apiVersion: v1
kind: Pod
metadata:
  name: web-server
spec:
  containers:
  - image: nginx:1.23.0
    name: web-server
    ports:
    - containerPort: 80
      name: nginx-port
    startupProbe:
      httpGet:
        path: /
        port: nginx-port
```

Further edit the manifest by defining a readiness probe. The finalized manifest could look as follows:

```
apiVersion: v1
kind: Pod
metadata:
  name: web-server
spec:
  containers:
  - image: nginx:1.23.0
    name: web-server
    ports:
    - containerPort: 80
      name: nginx-port
    startupProbe:
      httpGet:
        path: /
        port: nginx-port
    readinessProbe:
      httpGet:
        path: /
        port: nginx-port
    initialDelaySeconds: 5
```

Further edit the manifest by defining a liveness probe. The finalized manifest could look as follows:

```

apiVersion: v1
kind: Pod
metadata:
  name: web-server
spec:
  containers:
    - image: nginx:1.23.0
      name: web-server
      ports:
        - containerPort: 80
          name: nginx-port
      startupProbe:
        httpGet:
          path: /
          port: nginx-port
      readinessProbe:
        httpGet:
          path: /
          port: nginx-port
      initialDelaySeconds: 5
      livenessProbe:
        httpGet:
          path: /
          port: nginx-port
      initialDelaySeconds: 10
      periodSeconds: 30

```

Create the Pod, then check its READY and STATUS columns. The container will transition from ContainerCreating to Running. At some point, a 1/1 container will be available:

```

$ kubectl create -f probed-pod.yaml
pod/probed-pod created
$ kubectl get pod web-server
NAME      READY   STATUS      RESTARTS   AGE
web-server  0/1    ContainerCreating   0          7s
$ kubectl get pod web-server
NAME      READY   STATUS      RESTARTS   AGE
web-server  0/1    Running   0          8s
$ kubectl get pod web-server
NAME      READY   STATUS      RESTARTS   AGE
web-server  1/1    Running   0          38s

```

You should find the configuration of the probes when executing the `describe` command:

```

$ kubectl describe pod web-server
...
Containers:
  web-server:
    ...
    Ready:           True
    Restart Count:  0

```

```

Liveness:      http-get http://:nginx-port/ delay=10s timeout=1s \
               period=30s #success=1 #failure=3
Readiness:     http-get http://:nginx-port/ delay=5s timeout=1s \
               period=10s #success=1 #failure=3
Startup:       http-get http://:nginx-port/ delay=0s timeout=1s \
               period=10s #success=1 #failure=3
...

```

Chapter 15, Troubleshooting Pods and Containers

- First, create the Pod using the given YAML content:

```
$ kubectl apply -f pod.yaml
pod/date-recorder created
```

Inspecting the Pod's status exposes no obvious issues. The status is "Running":

```
$ kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
date-recorder  1/1     Running   0          5s
```

Render the logs of the container. The returned error message indicates that the file or directory `/root/tmp/startup-marker.txt` does not exist:

```
$ kubectl logs date-recorder
[Error: ENOENT: no such file or directory, open \
'/root/tmp/startup-marker.txt'] {
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: '/root/tmp/curr-date.txt'
}
```

We could try to open a shell to the container; however, the container image does not provide a shell:

```
$ kubectl exec -it date-recorder -- /bin/sh
OCI runtime exec failed: exec failed: unable to start container \
process: exec: "/bin/sh": stat /bin/sh: no such file or \
directory: unknown
command terminated with exit code 126
```

We can use the debug command to create a debugging container for troubleshooting. The `--share-processes` flag lets us share the running nodejs process:

```
$ kubectl debug -it date-recorder --image=busybox --target=debian \
--share-processes
Targeting container "debian". If you don't see processes from this \
container it may be because the container runtime doesn't support \
this feature.
Defaulting debug container name to debugger-rns89.
If you don't see a command prompt, try pressing enter.
/ # ps
 PID  USER      TIME  COMMAND
```

```
1 root      4:21 /nodejs/bin/node -e const fs = require('fs'); \
let timestamp = Date.now(); fs.writeFile('/root/tmp/startup-m
35 root      0:00 sh
41 root      0:00 ps
```

Apparently, the directory we want to write to does indeed not exist:

```
$ kubectl exec failing-pod -it -- /bin/sh
/ # ls /root/tmp
ls: /root/tmp: No such file or directory
```

We'll likely want to change the command running the original container to point to the directory that does exist upon container start. Alternatively, it may make sense to mount an ephemeral Volume to provide the directory, as shown here:

```
apiVersion: v1
kind: Pod
metadata:
  name: date-recorder
spec:
  containers:
    - name: debian
      image: gcr.io/distroless/nodejs20-debian11
      command: ["/nodejs/bin/node", "-e", "const fs = require('fs'); \
let timestamp = Date.now(); fs.writeFile('/var/startup/\
startup-marker.txt', timestamp.toString(), err => { if (err) { \
console.error(err); } while(true) {}});"]
      volumeMounts:
        - mountPath: /var/startup
          name: init-volume
    volumes:
      - name: init-volume
        emptyDir: {}
```

2. Create the namespace with the imperative command:

```
$ kubectl create ns stress-test
namespace/stress-test created
```

Create all Pods by pointing the `apply` command to the current directory:

```
$ kubectl apply -f ./
pod/stress-1 created
pod/stress-2 created
pod/stress-3 created
```

Retrieve the metrics for the Pods from the metrics server using the `top` command:

```
$ kubectl top pods -n stress-test
NAME      CPU(cores)   MEMORY(bytes)
stress-1  50m          77Mi
stress-2  74m          138Mi
stress-3  58m          94Mi
```

The Pod with the highest amount of memory consumption is the Pod named `stress-2`. The metrics will look different on your machine given that the amount of consumed memory is randomized by the command executed per container.

Chapter 16, Custom Resource Definitions (CRDs)

1. Create the CRD from the provided URL:

```
$ kubectl apply -f https://raw.githubusercontent.com/mongodb/\
  mongodb-kubernetes-operator/master/config/crd/bases/mongodbcommunity.\
  mongodb.com_mongodbcommunity.yaml
  customresourcedefinition.apiextensions.k8s.io/mongodbcommunity.\
  mongodbcommunity.mongodb.com created
```

You can find the installed CRD named `mongodbcommunity.mongodbcommunity.mongodb.com` with the following command:

```
$ kubectl get crds
NAME                                     CREATED AT
mongodbcommunity.mongodbcommunity.mongodb.com  2023-12-18T23:44:04Z
```

One way to inspect the schema of the CRD is to use the `kubectl describe` command:

```
$ kubectl describe crds mongodbcommunity.mongodbcommunity.mongodb.com
```

You will find that the output of the command is very lengthy. Looking through the details, you will see that the type is called MongoDBCommunity. The CRD offers a lot of properties that can be set when instantiating an object of this type. See the documentation of the operator for more information.

2. Create the definition of the CRD. The resulting YAML manifest stored in the file `backup-resource.yaml` looks like this:

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: backups.example.com
spec:
  group: example.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronExpression:
```

```
        type: string
podName:
        type: string
path:
        type: string
scope: Namespaced
names:
        kind: Backup
singular: backup
plural: backups
shortNames:
- bu
```

Create the object from the YAML manifest file:

```
$ kubectl apply -f backup-resource.yaml
customresourcedefinition.apiextensions.k8s.io/backups.example.com created
```

You can interact with CRD using the following command. Make sure to spell out the full name of the CRD, `backups.example.com`:

```
$ kubectl get crd backups.example.com
NAME          CREATED AT
backups.example.com  2023-05-24T15:11:15Z
$ kubectl describe crd backups.example.com
...
```

Create the YAML manifest in file `backup.yaml` that uses the CRD kind `Backup`:

```
apiVersion: example.com/v1
kind: Backup
metadata:
    name: nginx-backup
spec:
    cronExpression: "0 0 * * *"
    podName: nginx
    path: /usr/local/nginx
```

Create the object from the YAML manifest file:

```
$ kubectl apply -f backup.yaml
backup.example.com/nginx-backup created
```

You can interact with the object using the built-in `kubectl` commands for any other Kubernetes API primitive:

```
$ kubectl get backups
NAME      AGE
nginx-backup   24s
$ kubectl describe backup nginx-backup
...
```

Chapter 17, Authentication, Authorization, and Admission Control

1. Follow the instructions “[How to issue a certificate for a user](#)” in the Kubernetes documentation to create a certificate for the user `mary`. You can find a list of executable commands in the file `app-a/ch17/mary-context/create-user-context.sh`. Avoid creating the corresponding Role and RoleBinding, as described in “[Create Role and RoleBinding](#)”.

Say you generated the client key file `mary.key` and the client certificate file `mary.crt` in the previous step. Use the following command to add the credentials to the kubeconfig file referenced by the username `mary`:

```
$ kubectl config set-credentials mary --client-key=mary.key \
--client-certificate=mary.crt --embed-certs=true
```

Next, add the content for the user `mary`. This command uses the context name `mary-context` that binds the cluster `kubernetes` to the user `mary`. You may want to check the kubeconfig file to see all available cluster names:

```
$ kubectl config set-context mary-context --cluster=kubernetes \
--user=mary
```

Change the context to `mary-context` with the following command:

```
$ kubectl config use-context mary-context
```

Trying to create a new Pod with the selected context won’t be allowed as the user doesn’t have the appropriate permissions to perform the action:

```
$ kubectl run nginx --image=nginx:1.25.2 --port=80
Error from server (Forbidden): pods is forbidden: User "mary" cannot \
create resource "pods" in API group "" in the namespace "default"
```

2. Create the namespace `t23`:

```
$ kubectl create namespace t23
```

Create the service account `api-call` in the namespace:

```
$ kubectl create serviceaccount api-call -n t23
```

Define a YAML manifest file with the name `pod.yaml`. The contents of a file define a Pod that makes a HTTPS GET call to the API server to retrieve the list of Services in the `default` namespace:

```
apiVersion: v1
kind: Pod
metadata:
  name: service-list
  namespace: t23
spec:
  serviceAccountName: api-call
```

```

containers:
- name: service-list
  image: alpine/curl:3.14
  command: ['sh', '-c', 'while true; do curl -s -k -m 5 -H \
    "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/\
    serviceaccount/token)" https://kubernetes.default.svc.cluster.\
    local/api/v1/namespaces/default/services; sleep 10; done']

```

Create the Pod with the following command:

```
$ kubectl apply -f pod.yaml
```

Check the logs of the Pod. The API call is not authorized, as shown in the log output here:

```

$ kubectl logs service-list -n t23
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
  "message": "services is forbidden: User \\"system:serviceaccount:t23 \\
  :api-call\\" cannot list resource \\"services\\" in API \\
  group \\"\\\" in the namespace \\"default\\\"",

  "reason": "Forbidden",
  "details": {
    "kind": "services"
  },
  "code": 403
}

```

Create the YAML manifest in the file *clusterrole.yaml*, as shown here:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: list-services-clusterrole
rules:
- apiGroups: []
  resources: ["services"]
  verbs: ["list"]

```

Reference the ClusterRole in a RoleBinding defined in the file *rolebinding.yaml*.

The subject should list the service account **api-call** in the namespace **t23**:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: serviceaccount-service-rolebinding
subjects:
- kind: ServiceAccount
  name: api-call
  namespace: t23
roleRef:
  kind: ClusterRole

```

```
name: list-services-clusterrole
apiGroup: rbac.authorization.k8s.io
```

Create both objects from the YAML manifests:

```
$ kubectl apply -f clusterrole.yaml
$ kubectl apply -f rolebinding.yaml
```

The API call running inside of the container should now be authorized and allowed to list the Service objects in the default namespace. As shown in the following output, the namespace currently hosts at least one Service object, the kubernetes.default Service:

```
$ kubectl logs service-list -n t23
{
  "kind": "ServiceList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "1108"
  },
  "items": [
    {
      "metadata": {
        "name": "kubernetes",
        "namespace": "default",
        "uid": "30eb5425-8f60-4bb7-8331-f91fe0999e20",
        "resourceVersion": "199",
        "creationTimestamp": "2022-09-08T18:06:52Z",
        "labels": {
          "component": "apiserver",
          "provider": "kubernetes"
        }
      },
      ...
    }
  ]
}
```

3. You can find the API server configuration file at `/etc/kubernetes/manifests/kube-apiserver.yaml` on the control plane node. If you are using minikube, then you will have to **log into the minikube environment** with `minikube ssh`.

The attribute `spec.containers[0].command` runs the `kube-apiserver` executable. The command line flag `--enable-admission-plugins` used by the command lists the enabled admission controller plugins.

For minikube, the default plugin configuration looks like the following:

```
spec:
  containers:
  - command:
    - kube-apiserver
    - --enable-admission-plugins=NamespaceLifecycle,LimitRanger, \
      ServiceAccount,DefaultStorageClass,DefaultTolerationSeconds,\
```

```
NodeRestriction,MutatingAdmissionWebhook,ValidatingAdmissionWebhook,\  
ResourceQuota
```

Overall nine plugins have been enabled in a comma-separated form. You can read about their functionality in the [Kubernetes documentation](#). The list of enabled plugins in your cluster may differ.

Chapter 18, Resource Requirements, Limits, and Quotas

1. Start by creating a basic definition of a Pod. The following YAML manifest defines the Pod named `hello` with a single container running the image `bmuschko/nodejs-hello-world:1.0.0`. Add a Volume of type `emptyDir` to the Pod and mount it in the container. Finally, define the resource requirements for the container:

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: hello  
spec:  
  containers:  
    - image: bmuschko/nodejs-hello-world:1.0.0  
      name: hello  
      ports:  
        - name: nodejs-port  
          containerPort: 3000  
      volumeMounts:  
        - name: log-volume  
          mountPath: "/var/log"  
      resources:  
        requests:  
          cpu: 100m  
          memory: 500Mi  
          ephemeral-storage: 1Gi  
        limits:  
          memory: 500Mi  
          ephemeral-storage: 2Gi  
  volumes:  
    - name: log-volume  
      emptyDir: {}
```

Create the Pod object with the following command:

```
$ kubectl apply -f pod.yaml  
pod/hello created
```

The cluster in this scenario consists of three nodes: one control plane node and two worker nodes. Be aware that your setup will likely look different:

```
$ kubectl get nodes  
NAME      STATUS   ROLES      AGE      VERSION  
minikube  Ready    control-plane  65s    v1.28.2
```

```
minikube-m02 Ready <none> 44s v1.28.2  
minikube-m03 Ready <none> 26s v1.28.2
```

The `-o wide` flag renders the node that the Pod is running on, in this case the node named `minikube-m03`:

```
$ kubectl get pod hello -o wide  
NAME READY STATUS RESTARTS AGE IP NODE  
hello 1/1 Running 0 25s 10.244.2.2 minikube-m03
```

The details of the Pod provide information about the container's resource requirements:

```
$ kubectl describe pod hello  
...  
Containers:  
  hello:  
    ...  
    Limits:  
      ephemeral-storage: 2Gi  
      memory: 500Mi  
    Requests:  
      cpu: 100m  
      ephemeral-storage: 1Gi  
      memory: 500M  
    ...
```

2. First, create the namespace and the resource quota in the namespace:

```
$ kubectl create namespace rq-demo  
namespace/rq-demo created  
$ kubectl apply -f resourcequota.yaml --namespace=rq-demo  
resourcequota/app created
```

Inspect the details of the resource quota:

```
$ kubectl describe quota app --namespace=rq-demo  
Name: app  
Namespace: rq-demo  
Resource Used Hard  
-----  
pods 0 2  
requests.cpu 0 2  
requests.memory 0 500Mi
```

Next, create the YAML manifest in the file `pod.yaml` with more requested memory than available in the quota. Start by running the command `kubectl run mypod --image=nginx -o yaml --dry-run=client --restart=Never > pod.yaml`, and then edit the produced file. Remember to replace the `resources` attribute that has been created automatically:

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: mypod
```

```

spec:
  containers:
  - image: nginx
    name: mypod
    resources:
      requests:
        cpu: "0.5"
        memory: "1Gi"
    restartPolicy: Never

```

Create the Pod and observe the error message:

```

$ kubectl apply -f pod.yaml --namespace=rq-demo
Error from server (Forbidden): error when creating "pod.yaml": pods \
"mypod" is forbidden: exceeded quota: app, requested: \
requests.memory=1Gi, used: requests.memory=0, limited: \
requests.memory=500Mi

```

Lower the memory settings to less than 500Mi (e.g., 255Mi) and create the Pod:

```

$ kubectl apply -f pod.yaml --namespace=rq-demo
pod/mypod created

```

The consumed resources of the Pod can be viewed in column Used:

```

$ kubectl describe quota --namespace=rq-demo
Name:          app
Namespace:     rq-demo
Resource       Used   Hard
-----  -----
pods           1      2
requests.cpu   500m   2
requests.memory 255Mi 500Mi

```

3. Create the objects from the given YAML manifest. The file defines a namespace and a LimitRange object:

```

$ kubectl apply -f setup.yaml
namespace/d92 created
limitrange/cpu-limit-range created

```

Describing the LimitRange object gives away its container configuration details:

```

$ kubectl describe limitrange cpu-limit-range -n d92
Name:          cpu-limit-range
Namespace:     d92
Type          Resource  Min   Max   Default Request  Default Limit  ...
-----  -----
Container     cpu       200m  500m  500m            500m          ...

```

Define a Pod in the file *pod-without-resource-requirements.yaml* without any resource requirements:

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-without-resource-requirements

```

```
namespace: d92
spec:
  containers:
    - image: nginx:1.23.4-alpine
      name: nginx
```

Create the Pod object using the apply command:

```
$ kubectl apply -f pod-without-resource-requirements.yaml
pod/pod-without-resource-requirements created
```

A Pod without specified resource requirements will use the default request and limit defined by LimitRange, in this case 500m:

```
$ kubectl describe pod pod-without-resource-requirements -n d92
...
Containers:
  nginx:
    Limits:
      cpu: 500m
    Requests:
      cpu: 500m
```

The Pod defined in the file *pod-with-more-cpu-resource-requirements.yaml* specifies a higher CPU resource limit than allowed by the LimitRange:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-more-cpu-resource-requirements
  namespace: d92
spec:
  containers:
    - image: nginx:1.23.4-alpine
      name: nginx
      resources:
        requests:
          cpu: 400m
        limits:
          cpu: 1.5
```

As a result, the Pod will not be allowed to be scheduled:

```
$ kubectl apply -f pod-with-more-cpu-resource-requirements.yaml
Error from server (Forbidden): error when creating \
"pod-with-more-cpu-resource-requirements.yaml": pods \
"pod-with-more-cpu-resource-requirements" is forbidden: \
maximum cpu usage per Container is 500m, but limit is 1500m
```

Finally, define a Pod in the file *pod-with-less-cpu-resource-requirements.yaml*. The CPU resource request and limit fits within the boundaries of the LimitRange:

```
apiVersion: v1
kind: Pod
metadata:
```

```

name: pod-with-less-cpu-resource-requirements
namespace: d92
spec:
  containers:
    - image: nginx:1.23.4-alpine
      name: nginx
      resources:
        requests:
          cpu: 350m
        limits:
          cpu: 400m

```

Create the Pod object using the `apply` command:

```
$ kubectl apply -f pod-with-less-cpu-resource-requirements.yaml
pod/pod-with-less-cpu-resource-requirements created
```

The Pod uses the provided CPU resource request and limit:

```
$ kubectl describe pod pod-with-less-cpu-resource-requirements -n d92
...
Containers:
  nginx:
    Limits:
      cpu: 400m
    Requests:
      cpu: 350m
```

Chapter 19, ConfigMaps and Secrets

1. Create the ConfigMap and point to the text file upon creation:

```
$ kubectl create configmap app-config --from-file=application.yaml
configmap/app-config created
```

The ConfigMap defines a single key-value pair. The key is the name of the YAML file, and the value is the contents of *application.yaml*:

```
$ kubectl get configmap app-config -o yaml
apiVersion: v1
data:
  application.yaml: |->
    dev:
      url: http://dev.bar.com
      name: Developer Setup
    prod:
      url: http://foo.bar.com
      name: My Cool App
kind: ConfigMap
metadata:
  creationTimestamp: "2023-05-22T17:47:52Z"
  name: app-config
  namespace: default
```

```
resourceVersion: "7971"
uid: 00cf4ce2-ebec-48b5-a721-e1bde2aabd84
```

Execute the run command in combination with the `--dry-run` flag to generate the file for the Pod:

```
$ kubectl run backend --image=nginx:1.23.4-alpine -o yaml \
--dry-run=client --restart=Never > pod.yaml
```

The final YAML manifest should look similar to the following code snippet:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: backend
  name: backend
spec:
  containers:
  - image: nginx:1.23.4-alpine
    name: backend
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
  volumes:
  - name: config-volume
    configMap:
      name: app-config
```

Create the Pod by pointing the `apply` command to the YAML manifest:

```
$ kubectl apply -f pod.yaml
pod/backend created
```

Log into the Pod and navigate to the directory `/etc/config`. You will find the file `application.yaml` with the expected YAML content:

```
$ kubectl exec backend -it -- /bin/sh
/ # cd /etc/config
/etc/config # ls
application.yaml
/etc/config # cat application.yaml
dev:
  url: http://dev.bar.com
  name: Developer Setup
prod:
  url: http://foo.bar.com
  name: My Cool App
/etc/config # exit
```

2. It's easy to create the Secret from the command line:

```
$ kubectl create secret generic db-credentials --from-literal=\
db-password=password
secret/db-credentials created
```

The imperative command automatically Base64-encodes the provided value of the literal. You can render the details of the Secret object from the command line. The assigned value to the key db-password is cGFzc3dk:

```
$ kubectl get secret db-credentials -o yaml
apiVersion: v1
data:
  db-password: cGFzc3dk
kind: Secret
metadata:
  creationTimestamp: "2023-05-22T16:47:33Z"
  name: db-credentials
  namespace: default
  resourceVersion: "7557"
  uid: 2daf580a-b672-40dd-8c37-a4adb57a8c6c
type: Opaque
```

Execute the run command in combination with the --dry-run flag to generate the file for the Pod:

```
$ kubectl run backend --image=nginx:1.23.4-alpine -o yaml \
--dry-run=client --restart=Never > pod.yaml
```

Edit the YAML manifest and create an environment that reads the key from the Secret while assigning a new name for it.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: backend
  name: backend
spec:
  containers:
    - image: nginx:1.23.4-alpine
      name: backend
    env:
      - name: DB_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-credentials
            key: db-password
```

Create the Pod by pointing the apply command to the YAML manifest:

```
$ kubectl apply -f pod.yaml
pod/backend created
```

You can find the environment variable in Base64-decoded form by shelling into the container and running the env command:

```
$ kubectl exec -it backend -- env
DB_PASSWORD=passwd
```

Chapter 20, Security Contexts

1. Start by creating the Pod definition as a YAML manifest in the file `pod.yaml`. Initially, you will only define the container with its command:

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox-security-context
spec:
  containers:
    - name: busybox
      image: busybox:1.28
      command: ["sh", "-c", "sleep 1h"]
```

Enhance the Pod definition by adding the volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox-security-context
spec:
  containers:
    - name: busybox
      image: busybox:1.36.1
      command: ["sh", "-c", "sleep 1h"]
      volumeMounts:
        - name: vol
          mountPath: /data/test
  volumes:
    - name: vol
      emptyDir: {}
```

Finally, define the security context. Some of the security context attributes can be set only on the Pod level, while some others can only be defined on the container level:

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox-security-context
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  containers:
    - name: busybox
      image: busybox:1.36.1
      command: ["sh", "-c", "sleep 1h"]
      volumeMounts:
        - name: vol
          mountPath: /data/test
```

```
    securityContext:  
      allowPrivilegeEscalation: false  
    volumes:  
    - name: vol  
      emptyDir: {}
```

Create the Pod with the following command:

```
$ kubectl apply -f pod.yaml  
pod/busybox-security-context created
```

Open an interactive shell to the container. Create the file in the directory of the volume mount. The file user ID should be 1000 and the group ID should be 2000, as defined by the security context:

```
$ kubectl exec -it busybox-security-context -- sh  
/ $ cd /data/test  
/data/test $ touch logs.txt  
/data/test $ ls -l  
total 0  
-rw-r--r-- 1 1000 2000 0 May 23 01:10 logs.txt  
/data/test $ exit  
command terminated with exit code 1
```

2. Define a starting point for the Deployment by creating a new YAML manifest named *deployment_security_context.yaml*. Populate the file with an example from the Kubernetes document. The initial Deployment manifest may look like this:

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: nginx  
  namespace: h20  
spec:  
  replicas: 3  
  selector:  
    matchLabels:  
      app: nginx  
  template:  
    metadata:  
      labels:  
        app: nginx  
    spec:  
      containers:  
      - name: nginx  
        image: nginx:1.25.3-alpine
```

Keep editing the YAML manifest file. Next, add the security context. Linux capabilities can only be added on the container level. You can find out by checking the API documentation:

```
apiVersion: apps/v1  
kind: Deployment  
metadata:
```

```

name: nginx
namespace: h20
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.25.3-alpine
        securityContext:
          capabilities:
            drop:
              - all

```

Make sure to create the namespace first if you haven't done so yet. Create the Deployment with the following command:

```
$ kubectl apply -f deployment-security-context.yaml
deployment.apps/nginx created
```

You will find that all Pods controlled by the Deployment will end up in a failure status:

```
$ kubectl get pods -n h20
NAME           READY   STATUS      RESTARTS   AGE
nginx-674df44dfc-5l7jl  0/1     CrashLoopBackOff  4 (18s ago)  111s
nginx-674df44dfc-fmlrh  0/1     CrashLoopBackOff  4 (27s ago)  111s
nginx-674df44dfc-rqdkp  0/1     CrashLoopBackOff  4 (22s ago)  111s
```

The reason is that some of the operations required by NGINX are no longer permitted:

```
$ kubectl logs nginx-674df44dfc-rqdkp -n h20
...
2023/12/15 23:59:56 [emerg] 1#1: chown("/var/cache/nginx/client_temp", 101) failed (1: Operation not permitted)
nginx: [emerg] chown("/var/cache/nginx/client_temp", 101) failed (1: Operation not permitted)
```

Chapter 21, Services

1. Expose the service with the type ClusterIP on port 80:

```
$ kubectl create service clusterip myapp --tcp=80:80
service/myapp created
```

```
kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
myapp	ClusterIP	10.109.149.59	<none>	80/TCP	4s

Create a Deployment with a single Pod:

```
$ kubectl create deployment myapp --image=nginx:1.23.4-alpine --port=80
deployment.apps/myapp created
```

```
$ kubectl get deployments,pods
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/myapp	1/1	1	1	79s

NAME	READY	STATUS	RESTARTS	AGE
pod/myapp-7d6cd46d65-jrc2q	1/1	Running	0	78s

Scale the Deployment to 2 replicas:

```
$ kubectl scale deployment myapp --replicas=2
deployment.extensions/myapp scaled
```

```
$ kubectl get deployments,pods
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/myapp	2/2	2	2	107s

NAME	READY	STATUS	RESTARTS	AGE
pod/myapp-7d6cd46d65-8vr8t	1/1	Running	0	5s
pod/myapp-7d6cd46d65-jrc2q	1/1	Running	0	106s

Determine the cluster IP and the port for the Service. In this case, it's 10.109.149.59:80. Alternatively, you can use the DNS name myapp. Use the information with the wget command:

```
$ kubectl run tmp --image=busybox:1.36.1 --restart=Never -it --rm \
-- wget -O- 10.109.149.59:80
Connecting to 10.109.149.59:80 (10.109.149.59:80)
writing to stdout
...
written to stdout
pod "tmp" deleted
```

Turn the type of the service into NodePort to expose it outside of the cluster. Now, the service should expose a port in the 30000 range:

```
$ kubectl edit service myapp
```

```
...
```

```
spec:
```

```
  type: NodePort
```

```
...
```

```
$ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
myapp	NodePort	10.109.149.59	<none>	80:31205/TCP	6m44s

Get the internal IP address of one of the nodes of the cluster. That's 192.168.49.2 in this case:

```
$ kubectl get nodes -o wide
NAME      STATUS   ROLES      AGE     VERSION   INTERNAL-IP   ...
minikube  Ready    control-plane  11s    v1.28.3  192.168.49.2  ...
```

Run a `wget` or `curl` command against the service by using the node's internal IP address and the node port. For the setup explained in the previous section, it's 192.168.49.2:31205:

```
$ wget -O- 192.168.49.2:31205
--2019-05-10 16:32:35-- http://192.168.49.2:31205/
Resolving localhost (localhost)... ::1, 127.0.0.1
Connecting to localhost (localhost)|::1|:31205... connected.
HTTP request sent, awaiting response... 200 OK
Length: 612 [text/html]
...
2019-05-10 16:32:35 (24.3 MB/s) - written to stdout [612/612]
```

2. Create the objects from the `setup.yaml` file. You will see from the output that at least three objects have been created: a namespace, a Deployment, and a Service:

```
$ kubectl apply -f setup.yaml
namespace/y72 created
deployment.apps/web-app created
service/web-app created
```

You can list all objects relevant to the scenario using the following command:

```
$ kubectl get all -n y72
NAME          READY   STATUS    RESTARTS   AGE
pod/web-app-5f77f59c78-8svdm  1/1     Running   0          10m
pod/web-app-5f77f59c78-mhvjz  1/1     Running   0          10m

NAME        TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)
service/web-app  ClusterIP  10.106.215.153  <none>           80/TCP

NAME          READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/web-app  2/2       2           2          10m

NAME          DESIRED   CURRENT   READY   AGE
replicaset.apps/web-app-5f77f59c78  2         2         2        10m
```

The Service named `web-app` is of type `ClusterIP`. You can access the Service only from within the cluster. Trying to connect to the Service by its DNS name from a temporary Pod in the same namespace won't be allowed:

```
$ kubectl run tmp --image=busybox --restart=Never -it --rm -n y72 \
  -- wget web-app
Connecting to web-app (10.106.215.153:80)
wget: can't connect to remote host (10.106.215.153): Connection refused
pod "tmp" deleted
pod y72/tmp terminated (Error)
```

The endpoint for the Service `web-app` cannot be resolved, as shown by the following command:

```
$ kubectl get endpoints -n y72
NAME      ENDPOINTS   AGE
web-app   <none>       15m
```

Describing the Service object provides you will additional information, e.g., the label selector and the target port:

```
$ kubectl describe service web-app -n y72
Name:            web-app
Namespace:       y72
Labels:          <none>
Annotations:    <none>
Selector:        run=myapp
Type:            ClusterIP
IP Family Policy: SingleStack
IP Families:    IPv4
IP:              10.106.215.153
IPs:             10.106.215.153
Port:            <unset>  80/TCP
TargetPort:      3001/TCP
Endpoints:       <none>
Session Affinity: None
Events:          <none>
```

Upon inspecting the Deployment, you will find that the Pod template uses the label assignment `app=webapp`. The container port is set to 3000. This information doesn't match the configuration of the Service. The endpoints of the `web-app` Service now point to the IP address and container port of the replicas controlled by the Deployment:

```
$ kubectl get endpoints -n y72
NAME      ENDPOINTS   AGE
web-app   10.244.0.3:3000,10.244.0.4:3000   24m
```

Edit the live object of the Service. Change the label selector from `run=myapp` to `app=webapp`, and the target port from 3001 to 3000:

```
$ kubectl edit service web-app -n y72
service/web-app edited
```

After changing the Service configuration, you will find that you can open a connection to the Pod running the application:

```
$ kubectl run tmp --image=busybox:1.36.1 --restart=Never -it --rm -n y72 \
-- wget web-app
Connecting to web-app (10.106.215.153:80)
saving to 'index.html'
index.html      100% |*****| ...
'index.html' saved
pod "tmp" deleted
```

Chapter 22, Ingresses

1. Create the Deployment with the following command:

```
$ kubectl create deployment web --image=bmuschko/nodejs-hello-world:1.0.0
deployment.apps/web created
```

```
$ kubectl get deployment web
NAME    READY    UP-TO-DATE    AVAILABLE    AGE
web     1/1      1            1            6s
```

Afterward, expose the application with a Service:

```
$ kubectl expose deployment web --type=ClusterIP --port=3000
service/web exposed
```

```
$ kubectl get service web
NAME    TYPE      CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
web     ClusterIP  10.100.86.59  <none>        3000/TCP   6s
```

Determine the cluster IP and the port for the Service. In this case, it's `10.109.149.59:3000`. Alternatively, you can use the DNS name `web`. Use the information to execute a `wget` command from another Pod:

```
$ kubectl run tmp --image=busybox:1.36.1 --restart=Never -it --rm \
  -- wget -O- web:3000
Connecting to web:3000 (10.100.86.59:3000)
writing to stdout
Hello World
...
pod "tmp" deleted
```

Create an Ingress using the following manifest in the file `ingress.yaml`. The following content assumes that you use the NGINX Ingress controller:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: hello-world-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
  rules:
    - host: hello-world.exposed
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: web
                port:
                  number: 3000
```

Create the Ingress object from the YAML manifest:

```
$ kubectl apply -f ingress.yaml
ingress.networking.k8s.io/hello-world-ingress created
```

List the Ingress object. The value for the IP address will populate after a little while. You may have to run the command multiple times:

```
$ kubectl get ingress hello-world-ingress
NAME           CLASS      HOSTS          ADDRESS      ...
hello-world-ingress  nginx    hello-world.exposed  192.168.64.38 ...
```

Edit the file `/etc/hosts` via `sudo vim /etc/hosts`. Add the following entry to map the host name `hello-world.exposed` to the load balancer's IP address:

```
192.168.64.38 hello-world.exposed
```

Make a `curl` call to the host name mapped by the Ingress. The call should be routed toward the backend and respond with the message "Hello World":

```
$ curl hello-world.exposed
Hello World
```

2. Start by creating the objects in the manifest file `setup.yaml`. The manifest defines a Deployment, Service, and an Ingress in the namespace `s96`:

```
$ kubectl apply -f setup.yaml
namespace/s96 created
deployment.apps/nginx created
service/nginx created
ingress.networking.k8s.io/nginx created
```

Sending a HTTP request to the hostname `faulty.ingress.com` will result in an error. The call responds with a HTTP 503 code:

```
$ wget faulty.ingress.com/
--2024-01-04 09:45:44-- http://faulty.ingress.com/
Resolving faulty.ingress.com (faulty.ingress.com)... 127.0.0.1
Connecting to faulty.ingress.com (faulty.ingress.com)|127.0.0.1|:80...
HTTP request sent, awaiting response... 503 Service Temporarily \
Unavailable
2024-01-04 09:45:44 ERROR 503: Service Temporarily Unavailable.
```

The Ingress defines a single rule for the root URL context path, which targets the backend with the Service name `nginx` and port `3333`. The Service does not expose port `3333`. It uses port `9999`. You can fix the issue by changing the backend port in the Ingress manifest to `9999`. Make the change in the `setup.yaml` file and apply the changes to the live object:

```
$ kubectl apply -f setup.yaml
namespace/s96 unchanged
deployment.apps/nginx unchanged
service/nginx unchanged
ingress.networking.k8s.io/nginx configured
```

The call to to the domain main will now properly forward the traffic to the Service. You will receive a HTTP 200 response code:

```
$ wget faulty.ingress.com/
--2024-01-04 09:43:03-- http://faulty.ingress.com/
Resolving faulty.ingress.com (faulty.ingress.com)... 127.0.0.1
Connecting to faulty.ingress.com (faulty.ingress.com)|127.0.0.1|:80...
HTTP request sent, awaiting response... 200 OK
...
```

Chapter 23, Network Policies

1. Create the objects from the `setup.yaml` file. The manifest creates two Pods in different namespaces:

```
$ kubectl apply -f setup.yaml
namespace/end-user created
namespace/internal created
pod/frontend created
pod/backend created
```

Ensure that the `frontend` Pod can reach the `backend` Pod before establishing the network policy:

```
$ kubectl get pod backend --template '{{.status.podIP}}' -n internal
10.244.0.49
$ kubectl exec frontend -it -n end-user \
    -- wget --spider --timeout=1 10.244.0.49
Connecting to 10.244.0.49 (10.244.0.49:80)
remote file exists
```

Define a network policy in the namespace `end-user` using the file `app-stack-networkpolicy.yaml`. The Pod selector should use the label assigned to the front end Pod. The egress rule needs to select the Pod by assigned label and the corresponding namespace it runs in. Ensure that you list the allowed ports.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: app-stack
  namespace: end-user
spec:
  podSelector:
    matchLabels:
      app: frontend
  policyTypes:
  - Egress
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: backend
```

```

namespaceSelector:
  matchLabels:
    access: inside
ports:
- protocol: TCP
  port: 80

```

Create the object from the YAML manifest:

```
$ kubectl apply -f allow-egress-networkpolicy.yaml
networkpolicy.networking.k8s.io/allow-egress-networkpolicy created
```

The frontend Pod should still be able to reach the backend Pod if you define the network policy correctly:

```
$ kubectl exec frontend -it -n end-user \
  -- wget --spider --timeout=1 10.244.0.49
Connecting to 10.244.0.49 (10.244.0.49:80)
remote file exists
```

2. Create the objects from the *setup.yaml* file. The manifest creates two Pods and a network policy:

```
$ kubectl apply -f setup.yaml
namespace/k1 created
namespace/k2 created
pod/busybox created
pod/nginx created
networkpolicy.networking.k8s.io/default-deny-ingress created
```

Check on the Pods in namespace k1 and k2:

```
$ kubectl get pod -n k1
NAME      READY   STATUS    RESTARTS   AGE
busybox   1/1     Running   0          3s

$ kubectl get pod nginx --template '{{.status.podIP}}' -n k2
10.0.0.101
```

Opening a connection to the Pod nginx won't be allowed and it will time out:

```
$ kubectl exec -it busybox -n k1 -- wget --timeout=5 10.0.0.101:80
Connecting to 10.0.0.101:80 (10.0.0.101:80)
wget: download timed out
command terminated with exit code 1
```

Define a network policy in *allow-ingress-networkpolicy.yaml* that will allow ingress access from the namespace k1 to k2:

```

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-ingress-networkpolicy
  namespace: k2
spec:
  podSelector:

```

```
matchLabels:  
  app: backend  
policyTypes:  
  - Ingress  
ingress:  
  - from:  
    - namespaceSelector:  
      matchLabels:  
        role: consumer  
ports:  
  - protocol: TCP  
  port: 80
```

Create the object from the YAML manifest:

```
$ kubectl apply -f allow-ingress-networkpolicy.yaml  
networkpolicy.networking.k8s.io/allow-ingress-networkpolicy created
```

You can now make a call from any Pod in namespace k1 to the Pod nginx in namespace k2:

```
$ kubectl exec -it busybox -n k1 -- wget --timeout=5 10.0.0.101:80  
Connecting to 10.0.0.101:80 (10.0.0.101:80)  
saving to 'index.html'  
...
```


APPENDIX B

Exam Review Guide

This book covers all objectives relevant to the exam and more. The tables in this appendix map the exam objective to the corresponding book chapter. Furthermore, you will also find a reference to the Kubernetes documentation. Some foundational objectives important to the exam, such as Pods and namespaces, have not been listed explicitly in the curriculum; however, the book does cover them. You can use the mapping as a reference to review specific objectives in more detail.

Application Design and Build

Exam Objective	Chapter	Reference Documentation	Tutorial
Define, build, and modify container images	Chapter 4	Containers	N/A
Understand Jobs and CronJobs	Chapter 6	Job, CronJob	Indexed Job for Parallel Processing with Static Work Assignment, Automatic Cleanup for Finished Jobs, Running Automated Tasks with a CronJob
Understand multi-container Pod design patterns	Chapter 8	Init Containers, How Pods manage multiple containers	N/A
Utilize persistent and ephemeral volumes	Chapter 7	Ephemeral Volumes, Persistent Volumes, Storage Classes	N/A

Application Deployment

Exam Objective	Chapter	Reference Documentation	Tutorial
Use Kubernetes primitives to implement common deployment strategies	Chapter 11	Deployment Spec Strategy, Canary Deployment	Zero-downtime Deployment in Kubernetes with Jenkins
Understand Deployments and how to perform rolling updates	Chapter 10	Deployments	Using kubectl to Create a Deployment, Performing a Rolling Update, Running Multiple Instances of Your App
Use the Helm package manager to deploy existing packages	Chapter 12	Helm Project	Helm Charts: making it simple to package and deploy common applications on Kubernetes

Application Observability and Maintenance

Exam Objective	Chapter	Reference Documentation	Tutorial
Understand API deprecations	Chapter 13	Kubernetes Deprecation Policy, Deprecated API Migration Guide	N/A
Implement probes and health checks	Chapter 14	Container probes	Configure Liveness, Readiness and Startup Probes
Use provided tools to monitor Kubernetes applications	Chapter 15	Metrics Server Project	Metrics Server
Utilize container logs	Chapter 15	Interacting with running Pods	Examining Pod Logs
Debugging in Kubernetes	Chapter 15	N/A	Troubleshooting Applications, Debug Running Pods, Debug Pods, Use Port Forwarding to Access Applications in a Cluster

Application Environment, Configuration, and Security

Exam Objective	Chapter	Reference Documentation	Tutorial
Discover and use resources that extend Kubernetes (CRD)	Chapter 16	Custom Resources	Use Custom Resources
Understand authentication, authorization and admission control	Chapter 17	Controlling Access to the Kubernetes API, Authenticating, Using RBAC Authorization, Admission Controllers Reference	N/A
Understand and defining resource requirements, limits and quotas	Chapter 18	Resource Management for Pods and Containers, Limit Ranges, Resource Quotas	N/A
Understand ConfigMaps	Chapter 19	ConfigMaps	Configure a Pod to Use a ConfigMap

Exam Objective	Chapter	Reference Documentation	Tutorial
Create and consume Secrets	Chapter 19	Secrets	Managing Secrets using kubectl, Managing Secrets using Configuration File
Understand ServiceAccounts	Chapter 17	Service Accounts, ServiceAccount permissions	Configure Service Accounts for Pods
Understand SecurityContext	Chapter 20	N/A	Configure a Security Context for a Pod or Container

Services and Networking

Exam Objective	Chapter	Reference Documentation	Tutorial
Demonstrate basic understanding of NetworkPolicies	Chapter 23	Network Policies	Declare Network Policy
Provide and troubleshoot access to applications via services	Chapter 21	Service, DNS for Services and Pods	Connecting Applications with Services, Debug Services
Use Ingress rules to expose applications	Chapter 22	Ingress, Ingress Controllers	Set up Ingress on Minikube with the NGINX Ingress Controller

About the Author

Benjamin Muschko is a software engineer, consultant, and trainer with more than 20 years of experience in the industry. He's passionate about project automation, testing, and continuous delivery. Ben is an author, a frequent speaker at conferences, and an avid open source advocate. He holds the CKAD, CKA, and CKS certifications and is a CNCF Ambassador.

Software projects sometimes feel like climbing a mountain. In his free time, Ben loves hiking [Colorado's 14ers](#) and enjoys conquering long-distance trails.

Colophon

The animal on the cover of *Certified Kubernetes Application Developer (CKAD) Study Guide* is a common porpoise (*Phocoena phocoena*). It is the smallest of the seven species of porpoise and one of the smallest marine mammals. Adults are 4.5 to 6 feet long and weigh between 130 and 170 pounds. They are dark gray with lightly speckled sides and white undersides. Females are larger than males.

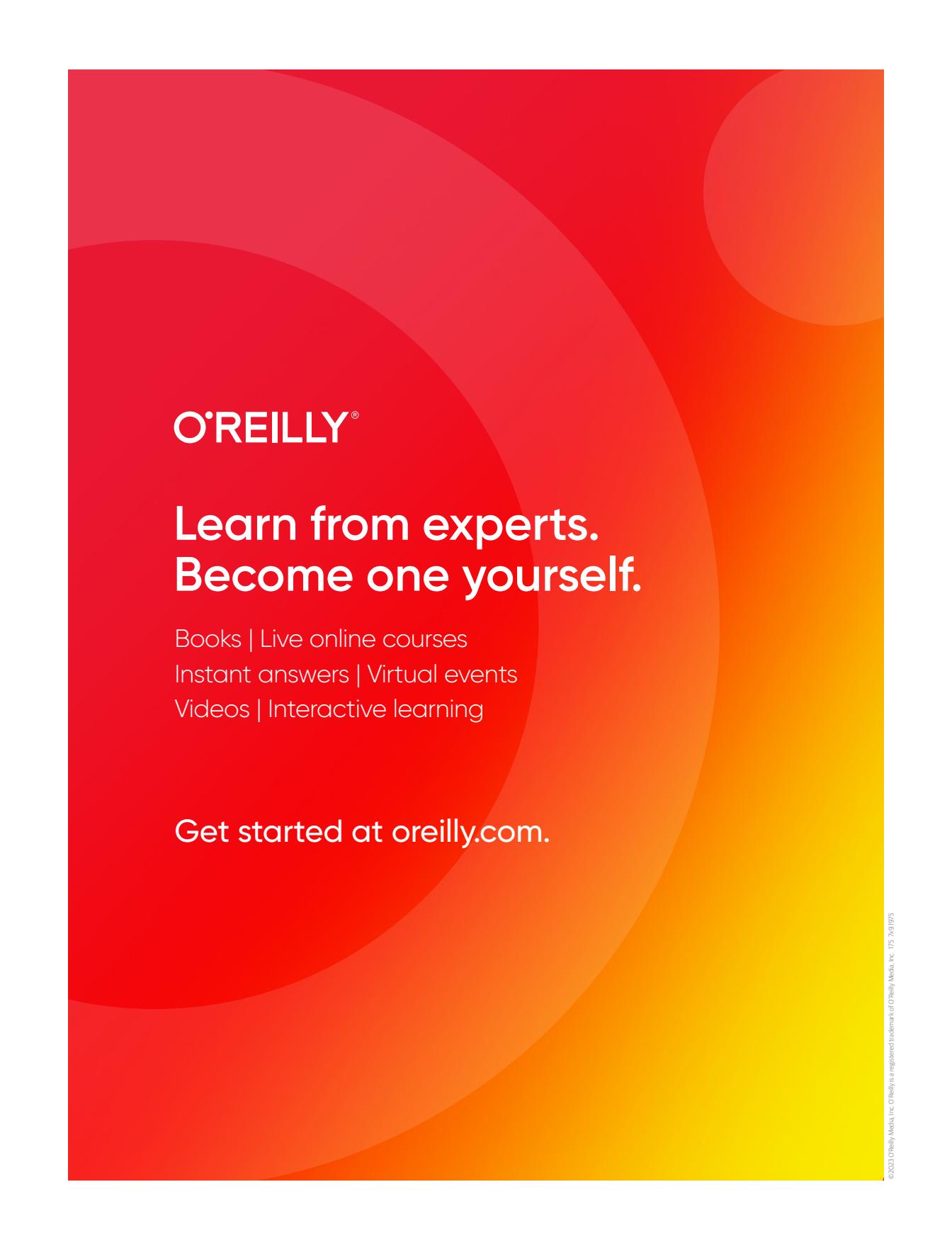
The common porpoise lives in the coastal waters of the North Atlantic, North Pacific, and Black Sea. They are also known as harbor porpoises since they inhabit fjords, bays, estuaries, and harbors. These marine mammals eat very small schooling fish and will hunt several hundred fish per hour throughout the day. They are usually solitary hunters but will occasionally form small packs.

Porpoises use ultrasonic clicks for echolocation (for both navigation and hunting) and social communication. A mass of adipose tissue in the skull, known as a melon, focuses and modulates their vocalizations.

Porpoises are conscious breathers; if they are unconscious for a long time, they may drown. In captivity, they have been known to sleep with one side of their brain at a time so that they can still swim and breathe consciously.

The conservation status of the common porpoise is of least concern. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *British Quadrupeds*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.



O'REILLY®

**Learn from experts.
Become one yourself.**

Books | Live online courses
Instant answers | Virtual events
Videos | Interactive learning

Get started at oreilly.com.