

O'REILLY®

2nd Edition

# Certified Kubernetes Administrator (CKA) Study Guide

In-Depth Guidance and Practice



Early  
Release

RAW &  
UNEDITED

Benjamin Muschko

# **Certified Kubernetes Administrator (CKA) Study Guide**

SECOND EDITION

In-Depth Guidance and Practice

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

**Benjamin Muschko**

**O'REILLY®**

# **Certified Kubernetes Administrator (CKA) Study Guide**

by Benjamin Muschko

Copyright © 2025 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*.

Editors: John Devins and Corbin Collins

Production Editor: Beth Kelly

Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

June 2022: First Edition

March 2026: Second Edition

## Revision History for the Early Release

- 2025-04-04: First Release
- 2025-05-21: Second Release
- 2025-09-18: Third Release

See <http://oreilly.com/catalog/errata.csp?isbn=9798341608405> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Certified Kubernetes Administrator (CKA) 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.

979-834-1-60835-1

[FILL IN]



# Brief Table of Contents (Not Yet Final)

---

## Part 1: Introduction

Chapter 1: Exam Details and Resources (available)

Chapter 2: Kubernetes in a Nutshell (available)

Chapter 3: Interacting with Kubernetes (available)

## Part 2: Cluster Architecture, Installation, and Configuration

Chapter 4: Cluster Installation and Upgrade (available)

Chapter 5: Backing Up and Restoring etcd (available)

Chapter 6: Authentication, Authorization, and Admission Control (available)

Chapter 7: Operators and Custom Resource Definitions (CRDs) (available)

Chapter 8: Helm and Kustomize (available)

## Part 3: Workloads and Scheduling

Chapter 9: Pods and Namespaces (available)

Chapter 10: ConfigMaps and Secrets (available)

Chapter 11: Deployments and ReplicaSets (available)

Chapter 12: Scaling Workloads (available)

Chapter 13: Resource Requirements, Limits, and Quotas (available)

Chapter 14: Pod Scheduling(available)

## Part 4: Storage

Chapter 15: Volumes (available)

Chapter 16: Persistent Volumes (available)

Part 5: Servicing and Networking

Chapter 17: Services (available)

Chapter 18: Ingresses (available)

Chapter 19: Gateway API (available)

Chapter 20: Network Policies (available)

Part 6: Troubleshooting

Chapter 21: Troubleshooting Applications (available)

Chapter 22: Troubleshooting Clusters (available)

Appendix A: Answers to Review Questions (available)

Appendix B: Exam Review Guide (available)

# Preface

---

Kubernetes has become the de facto standard for container orchestration, with over 96% of organizations using or evaluating Kubernetes according to the [CNCF Annual Survey 2023](#). As organizations increasingly adopt cloud-native technologies, the demand for skilled Kubernetes administrators has skyrocketed. The ability to deploy, manage, troubleshoot, and secure Kubernetes clusters is now a critical skill for infrastructure engineers, DevOps professionals, and system administrators worldwide.

To provide a standardized way to validate these essential skills, the Cloud Native Computing Foundation (CNCF), in collaboration with the Linux Foundation, created the [Certified Kubernetes Administrator \(CKA\)](#) program. Since its launch in 2017, the CKA has become one of the most sought-after certifications in the industry. According to the [CNCF Annual Report 2024](#), over 250,000 professionals have earned CNCF certifications, with the Kubernetes certifications representing the majority of this achievement. The report also highlights that the Kubestronaut program, recognizing professionals who earn all Kubernetes certifications, reached 1,500 members in its first year alone, demonstrating the strong commitment to comprehensive Kubernetes expertise.

The CKA certification validates that you have the skills, knowledge, and competency to perform the responsibilities of a Kubernetes administrator. Unlike traditional multiple-choice exams, the CKA is a rigorous, performance-based test that requires you to solve real-world problems in a live Kubernetes environment under time pressure. With a pass rate historically hovering around 60%, earning this certification demonstrates genuine hands-on expertise that employers value.

This study guide will prepare you to pass the CKA certification exam by covering all topics in the curriculum. We'll explore cluster architecture, installation and configuration, workload management, networking, storage, and troubleshooting. Throughout this journey, you'll gain practical experience with the `kubectl` and `kubeadm` command-line tool and learn the best practices that will serve you well both in the exam and in real-world scenarios.

## Who This Book Is For

This book is for system administrators, DevOps engineers, infrastructure architects, and anyone who wants to prepare for the CKA exam. While the content covers all aspects of the exam curriculum, readers should have basic knowledge of Linux, containers, and fundamental Kubernetes concepts.

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).

The CKA should not be confused with the **Certified Kubernetes Application Developer (CKAD)**. While there is topic overlap, the CKA focuses on cluster administration, installation, and maintenance tasks rather than application deployment and management. For those interested in the complete Kubernetes certification path, the CNCF offers a **certification bundle** that includes CKA, CKAD, and CKS (Certified Kubernetes Security Specialist).

## What You Will Learn

The content of this book follows the official CKA exam curriculum, covering cluster architecture, installation and configuration, workload management, networking, storage, and troubleshooting. Given the breadth of Kubernetes functionality, it's impossible to cover every configuration option without duplicating the official documentation. Test takers are encouraged to reference the **Kubernetes documentation** during the exam, as it's an allowed resource.

Practical experience is crucial for passing the exam. Each chapter contains a "Sample Exercises" section with practice questions that



mirror the exam format. Solutions to these exercises are available in [Appendix A](#).

## What's New in the Second Edition

The CNCF regularly updates the CKA curriculum to keep pace with Kubernetes evolution and industry needs. The curriculum update effective February 18, 2025, brings significant changes that reflect how Kubernetes is actually used in production environments today. The exam now uses Kubernetes v1.33, and this second edition has been completely updated to cover all the new topics while removing outdated content.

Based on the [official program changes](#) and [CNCf announcements](#), here are the major curriculum changes:

### New topics added

#### *Gateway API*

A major addition to the curriculum, the Gateway API represents the future of ingress traffic management in Kubernetes. This modern alternative to traditional Ingress resources provides more expressive routing rules, better multi-tenancy support, and extensible traffic management capabilities. You'll learn how to implement and manage Gateway resources, HTTPRoutes, and GatewayClasses.

#### *Helm*

Package management with Helm is now part of the CKA curriculum. You'll need to understand how to use Helm for deploying applications, managing releases, and working with Helm charts. This reflects the reality that Helm has become the de facto standard for Kubernetes application packaging.

## *Kustomize*

Declarative configuration management using Kustomize is now required knowledge. The curriculum covers how to use Kustomize for managing environment-specific configurations without templating, a critical skill for managing applications across multiple environments.

## *Custom Resource Definitions (CRDs) and Operators*

Understanding how to extend Kubernetes through CRDs and work with Operators is now essential. This addition reflects the widespread adoption of the Operator pattern in production Kubernetes environments.

## *Container Runtime Interfaces*

The curriculum now includes coverage of extension interfaces like CNI (Container Network Interface), CSI (Container Storage Interface), and CRI (Container Runtime Interface). Understanding these interfaces is crucial for troubleshooting and configuring production clusters.

## *Expanded CoreDNS Coverage*

While DNS was always part of the curriculum, there's now expanded focus on CoreDNS configuration, troubleshooting DNS issues, and understanding service discovery mechanics in detail.

## *Network Policies*

Network policies are now explicitly mentioned in the curriculum. You'll need to understand how to define and enforce network policies to control traffic flow between Pods, namespaces, and external endpoints. This addition emphasizes the growing importance of network segmentation and security in multi-tenant clusters.

## *Pod Admission and Scheduling*

The topic “Configure Pod admission and scheduling” has been made more prominent in the curriculum. This includes deeper coverage of scheduling mechanisms, node affinity, taints and tolerations, pod priority and preemption, and admission controllers. The increased emphasis reflects the complexity of workload placement in production clusters.

## **Topics removed or de-emphasized**

The competency “Provision underlying infrastructure to deploy a Kubernetes cluster” has been removed, acknowledging that most organizations now use managed Kubernetes platforms rather than provisioning infrastructure from scratch.

## **Structural changes**

While the five main domains (Cluster Architecture, Installation & Configuration; Workloads & Scheduling; Services & Networking; Storage; and Troubleshooting) remain the same, the competencies within each domain have been updated to reflect modern practices.

The exam continues to be performance-based with a 2-hour time limit, but the scenarios now better reflect real-world tasks that administrators face when managing production Kubernetes clusters.

Additionally, I’ve included an “Exam Review Guide” in [Appendix B](#) that maps all curriculum topics to corresponding chapters and provides quick reference links to the Kubernetes documentation.

## **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.

**TIP**

This element signifies a tip or suggestion.

**NOTE**

This element signifies a general note.

**WARNING**

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 Administrator (CKA) Study Guide*, by Benjamin Muschko (O'Reilly). Copyright 2025 Automated Ascent, LLC, 978-1-098-XXXXX-X."

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](mailto:permissions@oreilly.com).



## O'Reilly Online Learning

### NOTE

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/cka-2ed>.

Email [bookquestions@oreilly.com](mailto: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 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 [Technical Reviewers TBD] for their detailed technical guidance and feedback. I would also like to thank the editors at O'Reilly Media, John Devins and Corbin Collins, for their continued support and encouragement.

# Part I. Introduction

---

The introduction part of the book touches on the most important aspects of the exam and orients Kubernetes beginners 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

---

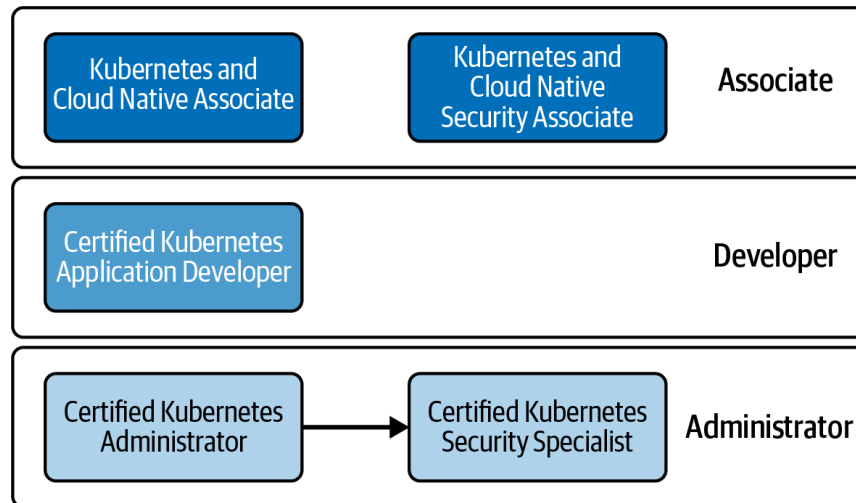
## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

This chapter addresses the most frequently asked questions by candidates preparing to successfully pass the **Certified Kubernetes Administrator (CKA)** 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 five 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 preexisting Kubernetes experience. Exams in this category require you to solve problems in multiple Kubernetes environments hands-on. You will find that the CKA is geared to administrators of a Kubernetes cluster and does not require any other certification as a prerequisite.

If you pass all Kubernetes certifications, you can call yourself a **Kubestronaut**. The Kubestronaut program recognizes community members that are able to demonstrate their skills in the most important core areas of Kubernetes.

Let's have a brief look at each certification to see if the CKA 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)**

The **KCSA** verifies your basic knowledge of security concepts and their application in a Kubernetes cluster. The breadth, depth, and format of the program is definitely more advanced than the KCNA. I personally would not categorize this exam as suitable to beginners to security concepts. You may want to take the KCSA after gaining more hands-on and in-depth exposure to Kubernetes, as the exam questions can be quite hard to answer.

## **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**

Kubernetes clusters need to be installed, configured, and maintained by skilled professionals. That's the job of a Kubernetes administrator. The CKA certification program verifies a deep understanding of the typical administration tasks encountered on the job, more specifically Kubernetes cluster maintenance, networking, storage solutions, and troubleshooting applications and cluster nodes.

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

## KUBERNETES VERSION USED DURING THE EXAM

At the time of writing, the exam is based on Kubernetes 1.33. 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:

- 25%: [Cluster Architecture, Installation and Configuration](#)
- 15%: [Workloads and Scheduling](#)
- 20%: [Servicing and Networking](#)
- 10%: [Storage](#)
- 30%: [Troubleshooting](#)

The next sections detail each domain.

## Cluster Architecture, Installation and Configuration

This section of the curriculum dives deep into all aspects of Kubernetes clusters. It covers the fundamental architecture of Kubernetes, including the distinction between the control plane and worker nodes, high-availability configurations, and the tools needed

for installing, upgrading, and maintaining a cluster. Additionally, you'll explore key extension interfaces and learn practical skills such as installing a cluster from scratch, upgrading its version, and backing up/restoring the etcd database.

The Cloud Native Computing Foundation (CNCF) has also incorporated related topics into this domain. For instance, mastering Role-Based Access Control (RBAC) is essential for administrators to effectively manage access to cluster resources. You'll also become proficient in installing Kubernetes operators and using tools like Kustomize and Helm to discover and deploy cluster components efficiently.

## **Workloads and Scheduling**

Administrators must have a solid understanding of Kubernetes concepts essential for managing cloud-native applications effectively. The domain focuses on these critical aspects. It covers key resources such as Deployments, ReplicaSets, and configuration management using ConfigMaps and Secrets.

When a new Pod is created, the Kubernetes scheduler assigns it to an available node based on predefined criteria. Scheduling rules, such as node affinity and taints/tolerations, help fine-tune this process to meet specific requirements. For exam preparation, it's crucial to grasp the various factors and concepts involved in Kubernetes' scheduling algorithm to ensure optimal workload placement and performance.

## **Servicing and Networking**

A cloud-native microservice rarely operates in isolation. More often than not, it interacts with other microservices or external systems. For administrators, understanding Pod-to-Pod communication, exposing applications to external clients, and configuring cluster networking is crucial for maintaining a fully functional system.

This domain of the exam evaluates your knowledge of essential Kubernetes networking primitives, including Services, Ingress, NetworkPolicy, and the Gateway API. Mastering these components ensures you can effectively manage and secure communication within and outside the cluster.

## **Storage**

This domain focuses on the various types of volumes used for reading and writing data in Kubernetes. As an administrator, you must understand how to create, configure, and manage these volumes effectively.

Persistent Volumes (PVs) play a crucial role in ensuring data persistence, even after a cluster node restarts. You'll need to demonstrate the ability to mount a Persistent Volume to a specific path within a container and understand the underlying mechanics. Additionally, it's essential to grasp the differences between static and dynamic provisioning to manage storage resources efficiently.

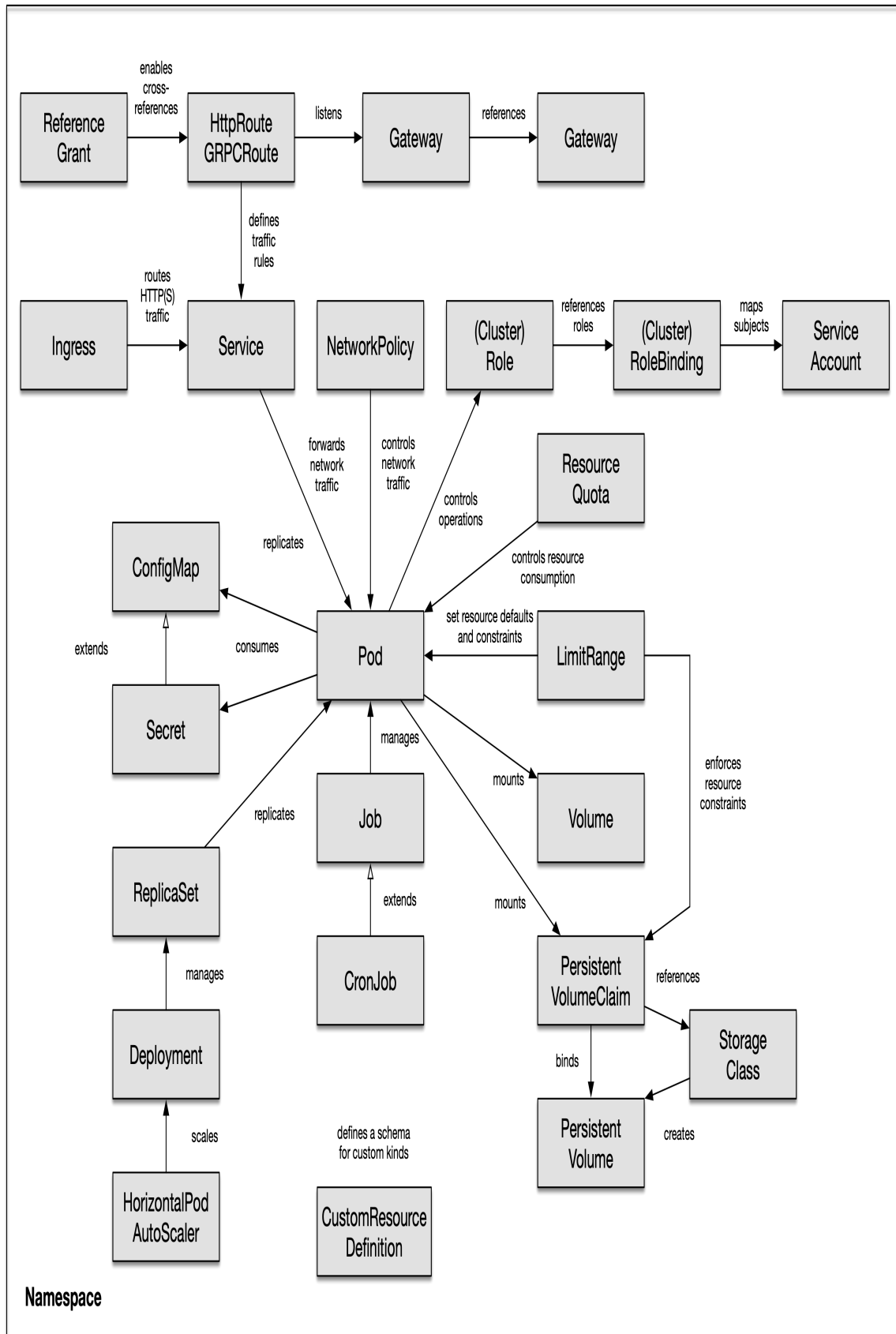
## **Troubleshooting**

In production Kubernetes clusters, issues are bound to arise. Applications may misbehave, become unresponsive, or even become completely inaccessible. Additionally, cluster nodes might crash or face configuration problems. Developing effective troubleshooting strategies is critical to quickly identifying and resolving these situations to minimize downtime and disruption.

This domain carries the highest scoring weight in the exam. You will encounter realistic scenarios that require you to diagnose problems and implement the appropriate solutions. Mastering these skills ensures you can maintain the stability and reliability of Kubernetes environments under pressure.

## **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, 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](#).

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 exam 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.

### *Kubernetes cluster maintenance tools*

Installing a Kubernetes cluster from scratch and upgrading the Kubernetes version of an existing cluster is performed using the tool `kubeadm`. It's important to understand its usage and the relevant process to walk through the process. Reference [Chapter 4](#) for more information. Additionally, you need to have a good understanding of the tools `etcdctl` and `etcdutl` including their command-line options for backing up and restoring the etcd database covered in [Chapter 5](#).

### *Working knowledge of a container runtime engine*

Kubernetes uses a container runtime engine for managing images. The default container runtime engine in Kubernetes is `containerd`. At a minimum, understand the difference between container images and containers, and their purpose. This topic goes beyond the coverage of this book and isn't directly tested for in 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 at least 66% of the answers to the tasks need to be correct to pass. Many tasks 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 tasks need 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"](#).

For specific tasks, e.g. the upgrade process of a cluster node, you will need to open an interactive shell to the host machine of a cluster node. Use the `ssh <nodename>` command to achieve that.

## 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  NAMESPACE
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 **CKA practice test in the form of interactive labs**.
- **Killercode** hosts interactive scenarios for the exam contributed by the Kubernetes community.

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.



# Chapter 2. Kubernetes in a Nutshell

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

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, I’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 (CI) pipeline.

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 9** to learn about those aspects.

## Features

The previous section touched on some features provided by Kubernetes. Here, I am going to dive a little deeper by explaining those features in 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*

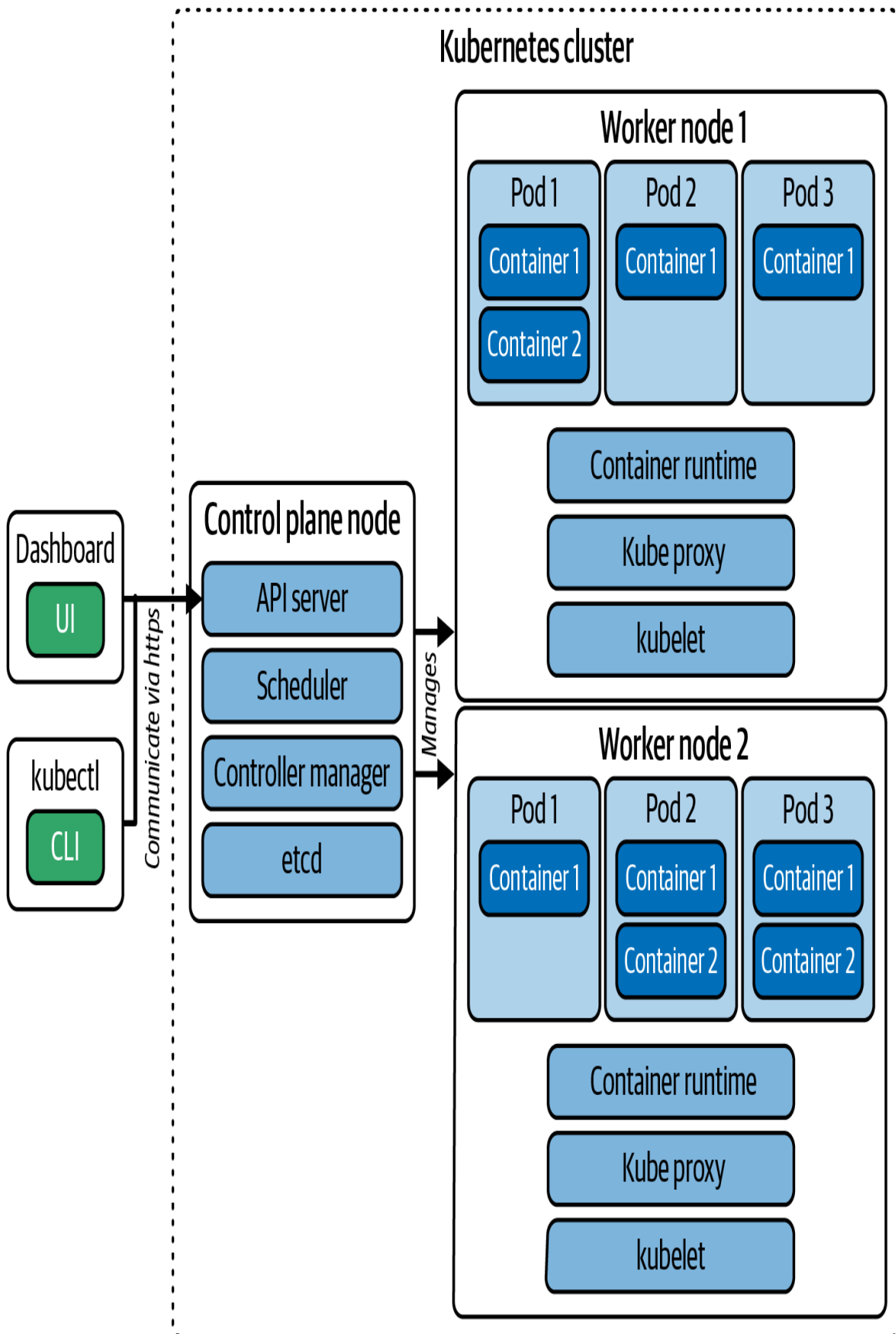
Containers offer only a temporary filesystem. Upon restart of the container, all data written to the filesystem is lost. Depending on the nature of your application, you may need to persist data for longer, for example, if your application interacts with a database. Kubernetes offers the ability to mount 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 look at 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 6](#).

### *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 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.

### *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 17](#).

### *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.

## **Advantages**

This section points out some of the most important 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.

# Chapter 3. Interacting with Kubernetes

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

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.

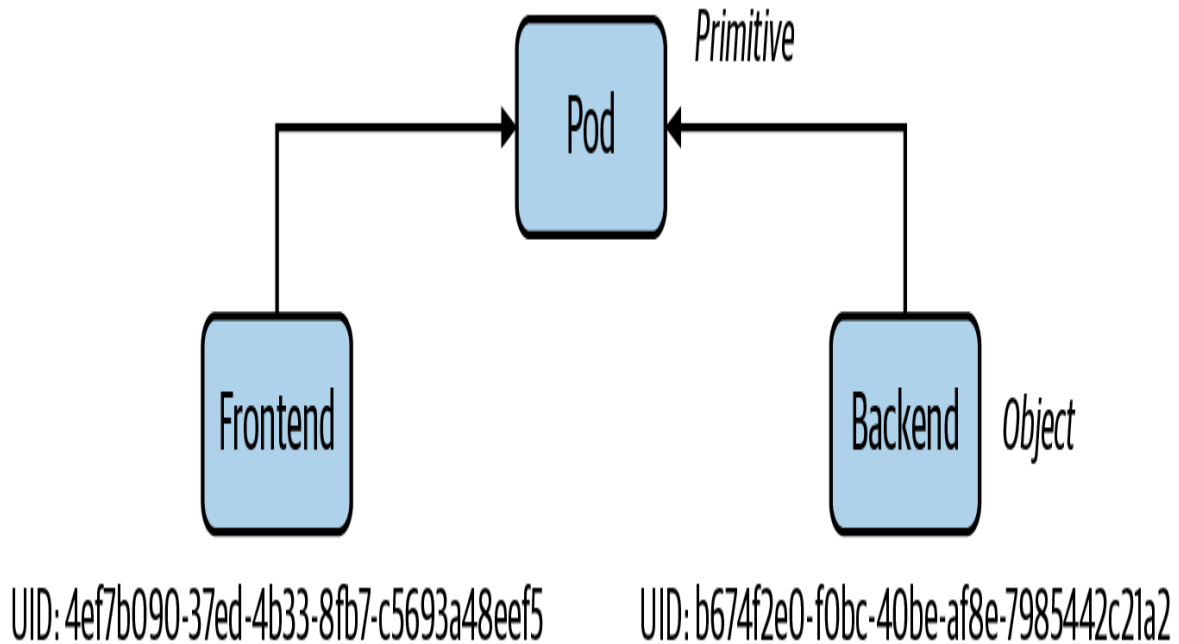
The exam does not test the use of a visual user interface for interacting with the Kubernetes cluster. Your only client for solving exam questions is `kubectl`. This chapter will touch on the Kubernetes API primitives and objects, as well as the different ways to manage objects with `kubectl`.

## 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.

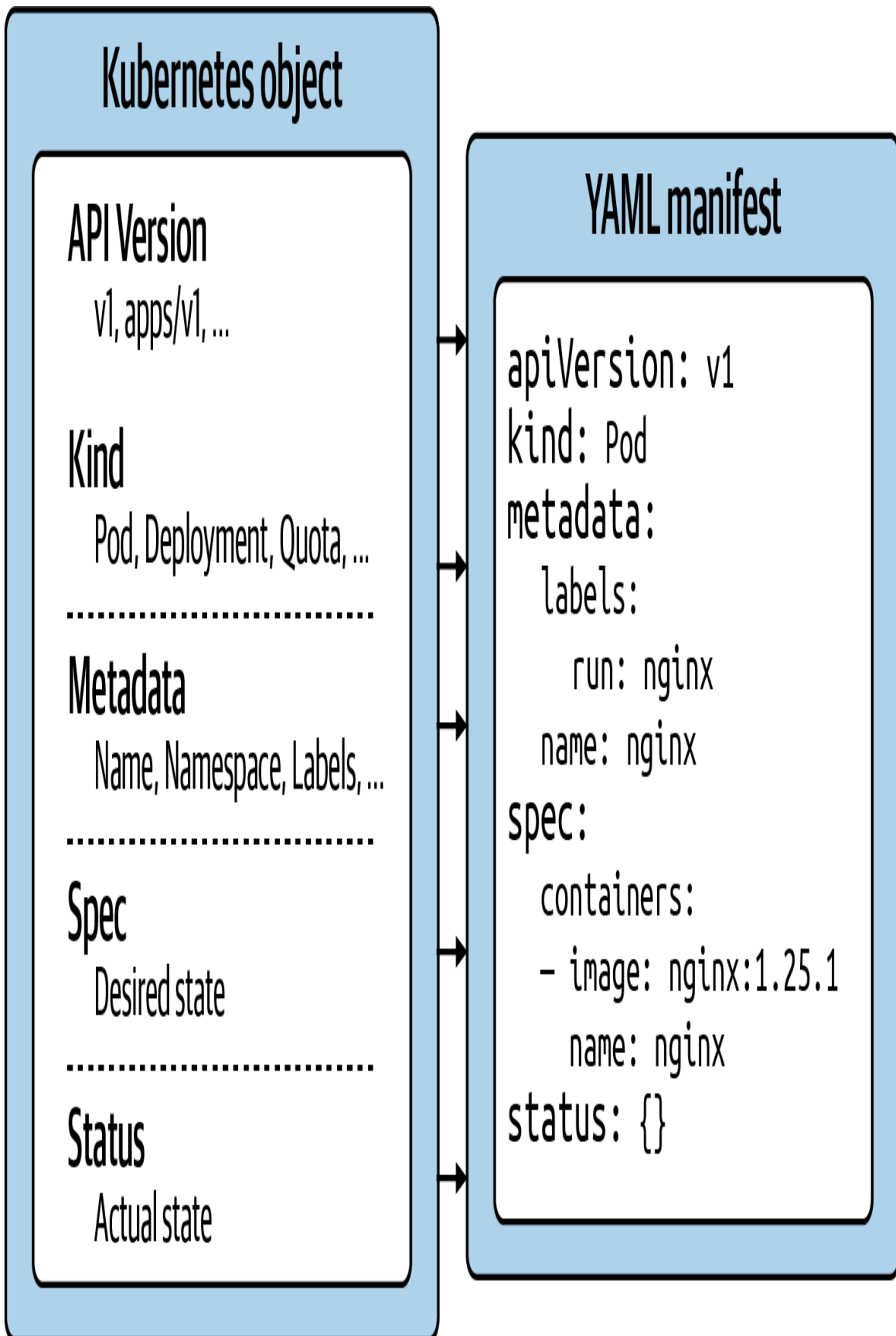
To draw an analogy, think back to the concepts of object-oriented programming. In object-oriented programming languages, a class defines the blueprint of a real-world functionality: its properties and behavior. A Kubernetes primitive is the equivalent of a class. The instance of a class in object-oriented 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.



The diagram illustrates the general usage pattern for the `kubectl` command. It shows the command `kubectl` followed by four components, each enclosed in square brackets and underlined with a curly brace. Below each bracketed component is a specific example: `get` under `[command]`, `pod` under `[TYPE]`, `app` under `[NAME]`, and `-o yaml` under `[flags]`.

*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/\
cka-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*. I 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"](#).

## 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 quickest 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. Cluster Architecture, Installation and Configuration

---

This domain explains the concepts you'd expect every Kubernetes administrator to be familiar with: the installation, maintenance, and configuration of the cluster. We'll discuss the processes, tools, and Kubernetes primitives and concepts involved by example.

The following chapters cover these concepts:

- **Chapter 4** expands on the discussion of architectural aspects applicable to a Kubernetes cluster. The content then provides an overview on the installation and update process for a Kubernetes cluster using `kubeadm`.
- **Chapter 5** focuses on a specific aspect of cluster maintenance: ensuring the data stored in etcd is backed up periodically and can be restored in case of a disaster that makes the data unreadable.
- **Chapter 6** first sets the stage on explaining the processing phases every API request has to go through before being handled. The chapter will then primarily concentrate on the authorization phase by explaining Role-Based Access Control (RBAC).
- **Chapter 7** touches on the Operator pattern, a concept used to extend the core functionality of the Kubernetes functionality. You will learn how to inspect and interact with

existing Custom Resource Definitions (CRDs) as a schema for instantiating objects to implement custom requirements not natively covered by Kubernetes.

- **Chapter 8** discusses the use of the open source tools Helm and Kustomize for deploying more complex application stacks or cluster components.

# Chapter 4. Cluster Installation and Upgrade

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

The first domain of the curriculum refers to typical tasks you’d expect of a Kubernetes administrator. Those tasks include understanding the architectural components of a Kubernetes cluster, setting up a cluster from scratch, and maintaining a cluster going forward. At the end of this chapter, you will understand the tools and procedures for installing and maintaining a Kubernetes cluster.

## COVERAGE OF CURRICULUM OBJECTIVES

This chapter addresses the following curriculum objective:

- Prepare underlying infrastructure for installing a Kubernetes cluster
- Understand extension interfaces (CNI, CSI, CRI, etc.)
- Create and manage Kubernetes clusters using kubeadm
- Manage the lifecycle of Kubernetes clusters
- Implement and configure a highly-available control plane

## Provisioning the Infrastructure

The infrastructure required by Kubernetes cluster nodes consist of servers, networks, and the storage. Provisioning of the infrastructure is done by configuring those resources on-premises or in the cloud, optimally in an automated fashion. That's the purpose of infrastructure automation tools like Ansible and Terraform.

Even though “provisioning the infrastructure” has been mentioned in the curriculum exam, a detailed discussion of this topic would go beyond the scope of this book and isn't directly related to Kubernetes. If you are more interested in the topic, have a look at the book *Infrastructure as Code*, 3rd Edition by Kief Morris (O'Reilly Media 2025).

## Understanding Extension Interfaces

**Chapter 2** laid the foundation for understanding the Kubernetes architecture, node types, and their cluster components. Kubernetes'

architecture has been designed to be flexible, modular, and extensible so that the core functionality of the platform can be expanded. You can extend the cluster using so-called extensions. Extensions are components that can be plugged into the platform to support custom functionality not native to Kubernetes. Those plugins usually include functions like network plugins, device plugins, and storage plugins.

This section talks about critical interfaces that you will need to understand as a Kubernetes administrator:

- [Container Network Interface \(CNI\)](#)
- [Container Runtime Interface \(CRI\)](#)
- [Container Storage Interface \(CSI\)](#)

## **Container Network Interface (CNI)**

The CNI is a specification and the corresponding libraries for writing plugins to configure network interfaces in Linux containers. In Kubernetes, the CNI is responsible for establishing network connections between Pods running in the cluster. You will need to install a CNI plugin in the control plane node(s) for them to function properly. ["Installing a Pod Network Add-on"](#) demonstrates the installation process of a CNI.

## **Container Runtime Interface (CRI)**

The container runtime is responsible for managing the lifecycle of containers and ensuring that containers receive the resources they requested. Kubernetes uses containerd as its default container runtime, however, you can swap it out with a different container runtime if needed. The CRI is an API facilitating the interaction between Kubernetes and different container runtimes.

## Container Storage Interface (CSI)

Despite the fact that Kubernetes provides a Volume plugin system, it was hard to integrate third-party storage solutions. In response, CSI was developed to as a standard to implement plugins for integrating arbitrary block and file storage systems to containerized workloads.

## Using kubeadm

The low-level command-line tool for performing cluster bootstrapping operations is called `kubeadm`. It is not meant for provisioning the underlying infrastructure.

To install `kubeadm`, follow the **installation instructions** in the official Kubernetes documentation. While not explicitly stated in the CKA frequently asked questions (FAQ) page, you can assume that the `kubeadm` executable has been preinstalled for you.

The following sections describe the processes for creating and managing a Kubernetes cluster on a high level and will use `kubeadm` heavily. For more detailed information, see the step-by-step Kubernetes reference documentation I will point out for each of the tasks.

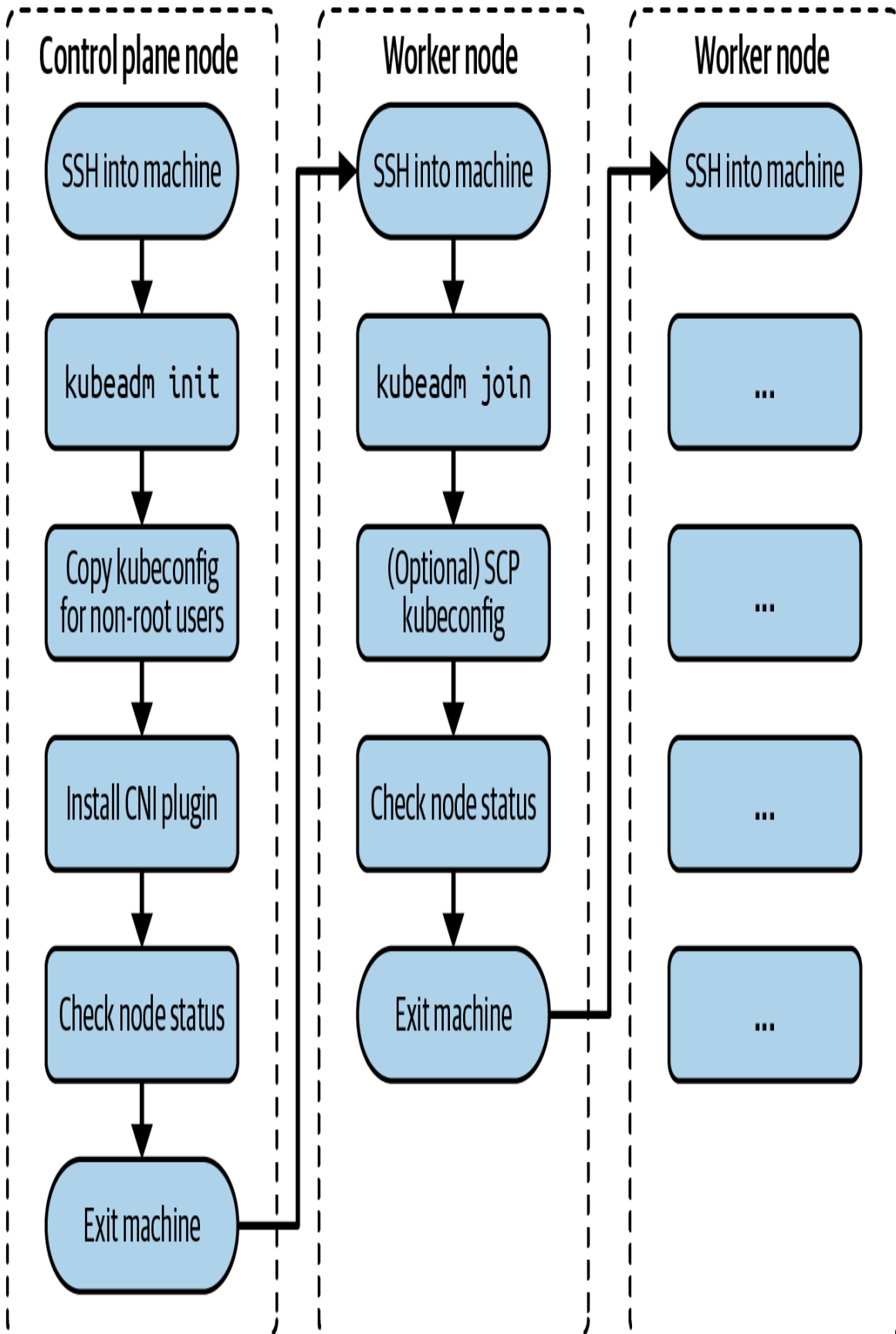
## Installing a Cluster

The most basic topology of a Kubernetes cluster consists of a single node that acts as the control plane and the worker node at the same time. By default, many developer-centric Kubernetes installations like minikube or Docker Desktop start with this configuration. While a single-node cluster may be a good option for a Kubernetes playground, it is not a good foundation for scalability and high-availability reasons. At the very least, you will want to



create a cluster with a single control plane and one or many nodes handling the workload.

This section explains how to install a cluster with a single control plane and one worker node. You can repeat the worker node installation process to add more worker nodes to the cluster. You can find a full description of the **installation steps** in the official Kubernetes documentation. **Figure 4-1** illustrates the installation process.



*Figure 4-1. Process for a cluster installation process*

## Initializing the Control Plane Node

Start by initializing the control plane on the control plane node. The control plane is the machine responsible for hosting the API server, etcd, and other components important to managing the Kubernetes cluster.

Open an interactive shell to the control plane node using the `ssh` command. The following command targets the control plane node named `kube-control-plane` running Ubuntu 24.10:

```
$ ssh kube-control-plane
Welcome to Ubuntu 24.10 (GNU/Linux 6.11.0-8-generic
aarch64)
...
```

Initialize the control plane using the `kubeadm init` command. You will need to add the following two command-line options: provide the IP addresses for the Pod network with the option `--pod-network-cidr`. With the option `--apiserver-advertise-address`, you can declare the IP address the API Server will advertise to listen on.

By default, `kubeadm` uses its own version to determine the control plane node's version. For example, if you use `kubeadm` version 1.31.1, then it will use version 1.31.1 for the initialized node. You can provide the desired Kubernetes version by providing the `--kubernetes-version` option, though it is recommended to use the `kubeadm` version you want to use for the nodes.

The console output renders a `kubeadm join` command. Keep that command around for later. It is important for joining worker nodes to the cluster in a later step.

## RETRIEVING THE JOIN COMMAND FOR WORKER NODES

You can always retrieve the `join` command by running `kubeadm token create --print-join-command` on the control plane node should you lose it.

The following command uses `10.244.0.0/16` for the Classless Inter-Domain Routing (CIDR):

```
$ sudo kubeadm init --pod-network-cidr=10.244.0.0/16
```

```
...
```

To start using your cluster, you need to run the following as a regular user:

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

You should now deploy a pod network to the cluster. Run `"kubectl apply -f [podnetwork].yaml"` with one of the options listed at:

<https://kubernetes.io/docs/concepts/cluster-administration/addons/>

Then you can join any number of worker nodes by running the following on \

each as root:

```
kubeadm join 172.16.0.5:6443 --token
fi8io0.dtkzsy9kws56dmzp \
    --discovery-token-ca-cert-hash \
    sha256:cc89ea1f82d5ec460e21b69476e0c052d691d0c52cce83fbd7e4
03559c1ebdac
```

After the `init` command has finished, run the necessary commands from the console output to start the cluster as nonroot user:

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

## Installing a Pod Network Add-on

You must deploy a **Container Network Interface (CNI) plugin** so that Pods can communicate with each other. You can pick from a wide range of networking plugins listed in the **Kubernetes documentation**. Popular plugins include Flannel, Calico, and Cilium. Sometimes you will see the term *add-ons* in the documentation, which is synonymous with plugin.

The exam will most likely ask you to install a specific add-on. Most of the installation instructions live on external web pages which are not permitted to be used during the exam. Make sure that you search for the relevant instructions in the official Kubernetes documentation. For example, you can find the installation instructions for Flannel [here](#).

### Installing Flannel using kubectl

The following command installs the Flannel objects via the released YAML manifest:

```
$ kubectl apply -f https://github.com/flannel-
io/flannel/releases/latest/\
download/kube-flannel.yml
namespace/kube-flannel created
serviceaccount/flannel created
clusterrole.rbac.authorization.k8s.io/flannel created
clusterrolebinding.rbac.authorization.k8s.io/flannel
created
```

```
configmap/kube-flannel-cfg created
daemonset.apps/kube-flannel-ds created
```

The YAML manifest hard-codes the Pod CIDR to the value `10.244.0.0/16`. If you want to change that value, you'd have to download the YAML manifest first, edit the corresponding value, and then apply the manifest.

## Installing Flannel using Helm

You may decide to install Flannel using Helm instead, either because you prefer the installation method or because you want to more conveniently provide a custom Pod CIDR. Helm offers the `--set` option to inject the custom value. The following command shows the use of Helm to install Flannel. Refer to ["Working with Helm"](#) to learn more about the use of Helm.

```
$ kubectl create ns kube-flannel
$ kubectl label --overwrite ns kube-flannel pod-
security.kubernetes.io/\
enforce=privileged
$ helm repo add flannel https://flannel-
io.github.io/flannel/
$ helm install flannel --set podCidr="10.244.0.0/16" --
namespace kube-flannel \
  flannel/flannel
namespace/kube-flannel created
namespace/kube-flannel labeled
"flannel" has been added to your repositories
NAME: flannel
LAST DEPLOYED: Wed Jan 29 23:03:33 2025
NAMESPACE: kube-flannel
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Both installation methods, the YAML manifest and the Helm chart, create the Flannel Pod into the namespace `kube-flannel`. Wait

until the Pod observes the “Running” status. You can check on the Pod’s details with the following command:

```
$ kubectl get pods -n kube-flannel
```

NAME	READY	STATUS	RESTARTS	AGE
kube-flannel-ds-h6455	1/1	Running	0	25s

Verify that the control plane node indicates the “Ready” status using the command `kubectl get nodes`. It might take a couple of seconds before the node transitions from the “NotReady” status to the “Ready” status. You have an issue with your node installation if the status transition does not occur. Refer to [Part VI](#) for debugging strategies:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE
VERSION			
kube-control-plane	Ready	control-plane	5m31s
v1.31.1			

Exit the control plane node using the `exit` command:

```
$ exit
logout
...
```

## Joining the Worker Nodes

Worker nodes are responsible for handling the workload scheduled by the control plane. Examples of workloads are Pods, Deployments, Jobs, and CronJobs. To add a worker node to the cluster so that it can be used, you will have to run a couple of commands, as described next.

Open an interactive shell to the worker node using the `ssh` command. The following command targets the worker node named `kube-worker-1` running Ubuntu 24.10:

```
$ ssh kube-worker-1
Welcome to Ubuntu 24.10 (GNU/Linux 6.11.0-8-generic
aarch64)
...
```

Run the `kubeadm join` command provided by the `kubeadm init` console output on the control plane node. The following command shows an example. Remember that the token and SHA256 hash will be different for you:

```
$ sudo kubeadm join 172.16.0.5:6443 --token
fi8io0.dtkzsy9kws56dmsp \
  --discovery-token-ca-cert-hash \

sha256:cc89ea1f82d5ec460e21b69476e0c052d691d0c52cce83fbd7e4
03559c1ebdac
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with \
'kubectl -n kube-system get cm kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file \
"/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with \
flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS
Bootstrap...
```

This node has joined the cluster:

- \* Certificate signing request was sent to apiserver and a response was received.
- \* The Kubelet was informed of the new secure connection details.



Run 'kubectl get nodes' on the control plane to see this node join the cluster.

You won't be able to run the `kubectl get nodes` command from the worker node without copying the administrator kubeconfig file from the control plane node. Follow the **instructions** in the Kubernetes documentation to do so or log back into the control plane node. Here, we are just going to log back into the control plane node. You should see that the worker node has joined the cluster and is in a "Ready" status:

```
$ ssh kube-control-plane
Welcome to Ubuntu 24.10 (GNU/Linux 6.11.0-8-generic
aarch64)
...
$ kubectl get nodes
NAME                                STATUS    ROLES    AGE
VERSION
kube-control-plane                 Ready    control-plane    2m14s
v1.31.1
kube-worker-1                      Ready    <none>          6m43s
v1.31.1
```

You can repeat the process for any other worker node you want to add to the cluster.

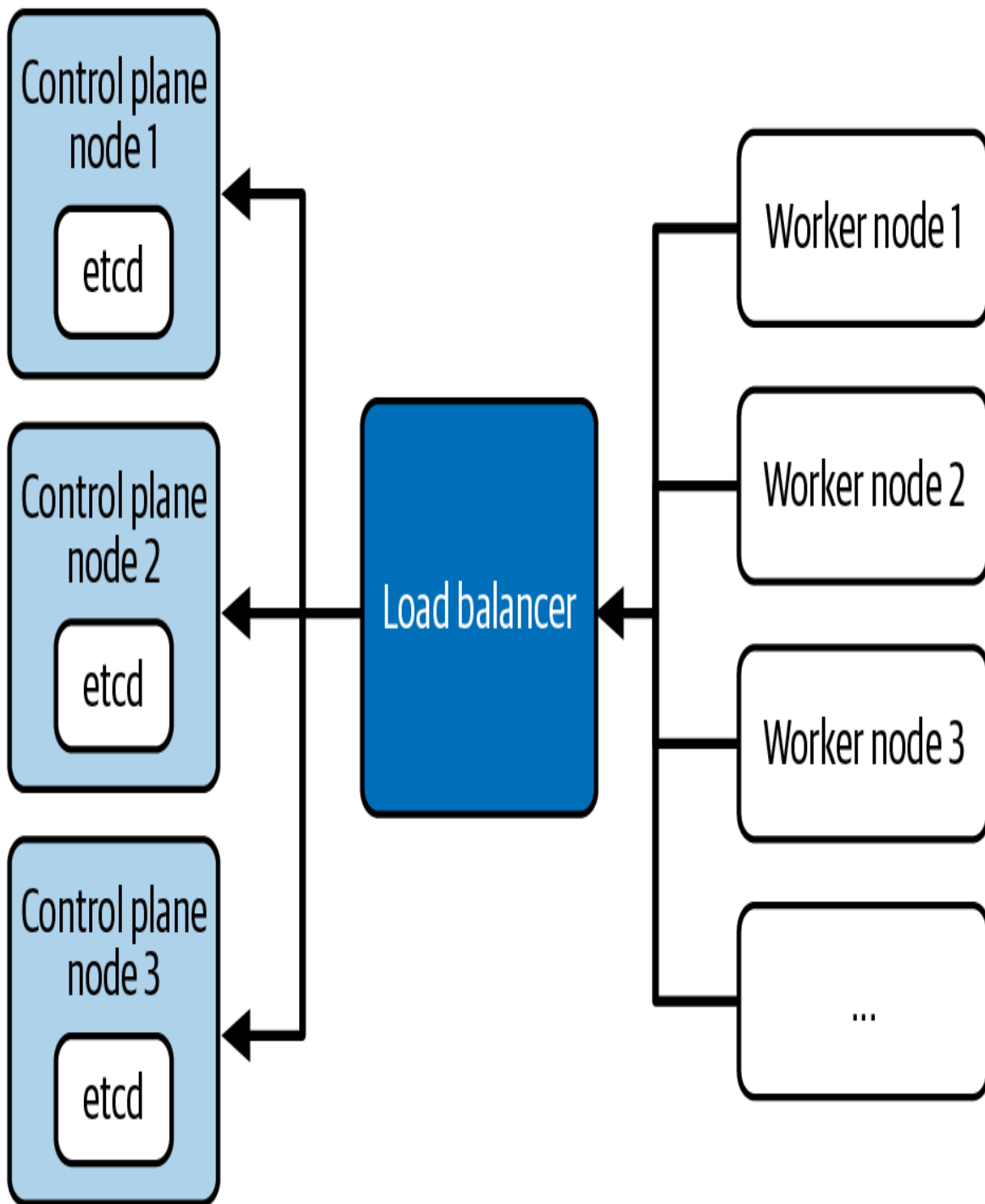
## Managing a Highly Available Cluster

Single control plane clusters are easy to install; however, they present an issue when the node is lost. Once the control plane node becomes unavailable, any ReplicaSet running on a worker node cannot re-create a Pod due to the inability to talk back to the scheduler running on a control plane node. Moreover, clusters cannot be accessed externally anymore (e.g., via `kubectl`), as the API server cannot be reached.

High-availability (HA) clusters help with scalability and redundancy. For the exam, you will need to have a basic understanding about configuring them and their implications. Given the complexity of standing up an HA cluster, it's unlikely that you'll be asked to perform the steps during the exam. For a full discussion on setting up HA clusters, see the [relevant page](#) in the Kubernetes documentation.

## Stacked etcd Topology

The *stacked etcd topology* involves creating two or more control plane nodes where etcd is colocated on the node. [Figure 4-2](#) shows a representation of the topology with three control plane nodes.



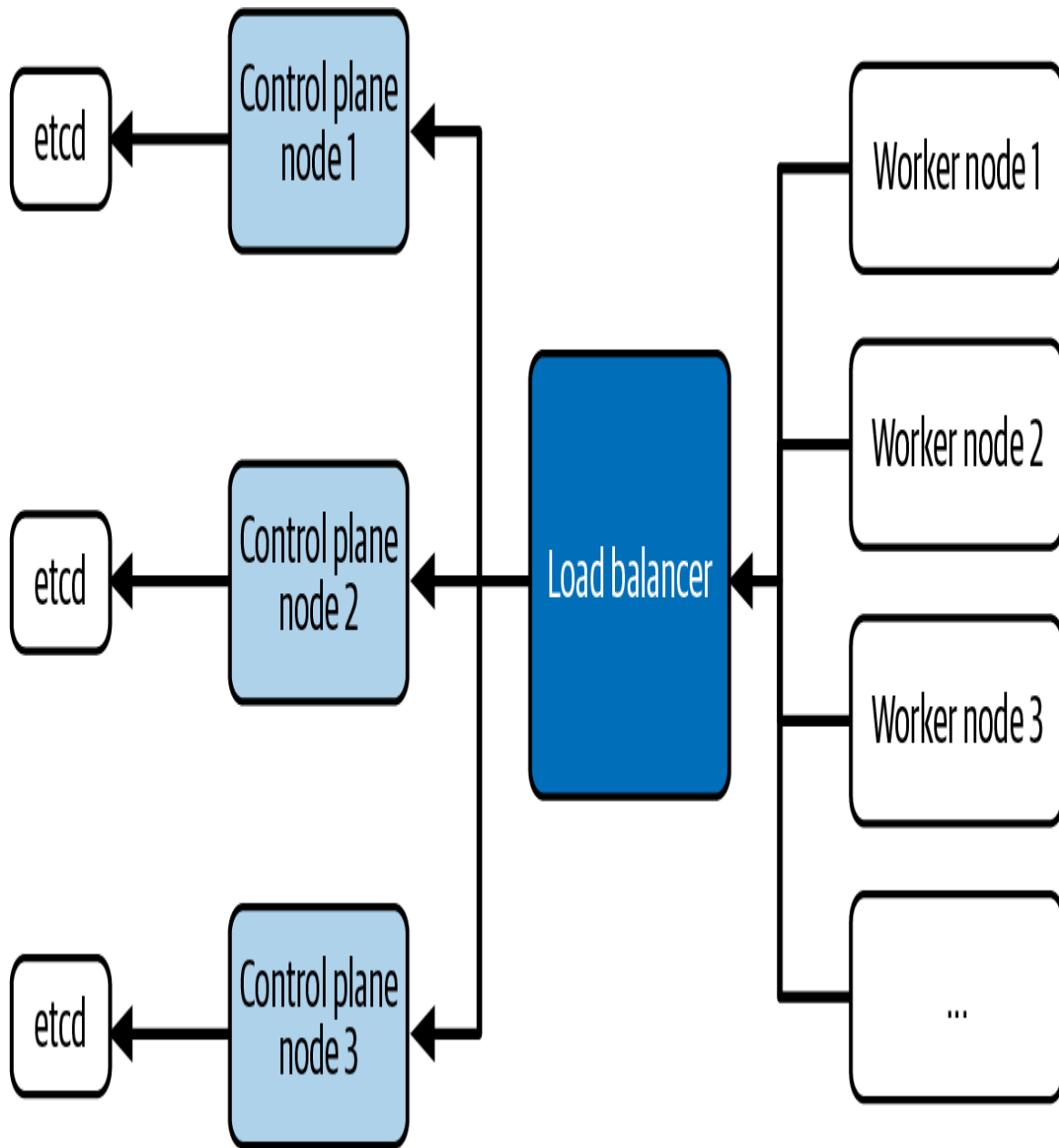
*Figure 4-2. Stacked etcd topology with three control plane nodes*

Each of the control plane nodes hosts the API server, the scheduler, and the controller manager. Worker nodes communicate with the API server through a load balancer. It is recommended to operate this cluster topology with a minimum of three control plane nodes

for redundancy reasons due to the tight coupling of etcd to the control plane node. By default, `kubeadm` will create an etcd instance when joining a control plane node to the cluster.

## External etcd Node Topology

The *external etcd node* topology separates etcd from the control plane node by running it on a dedicated machine. **Figure 4-3** shows a setup with three control plane nodes, each of which runs etcd on a different machine.



*Figure 4-3. External etcd node topology*

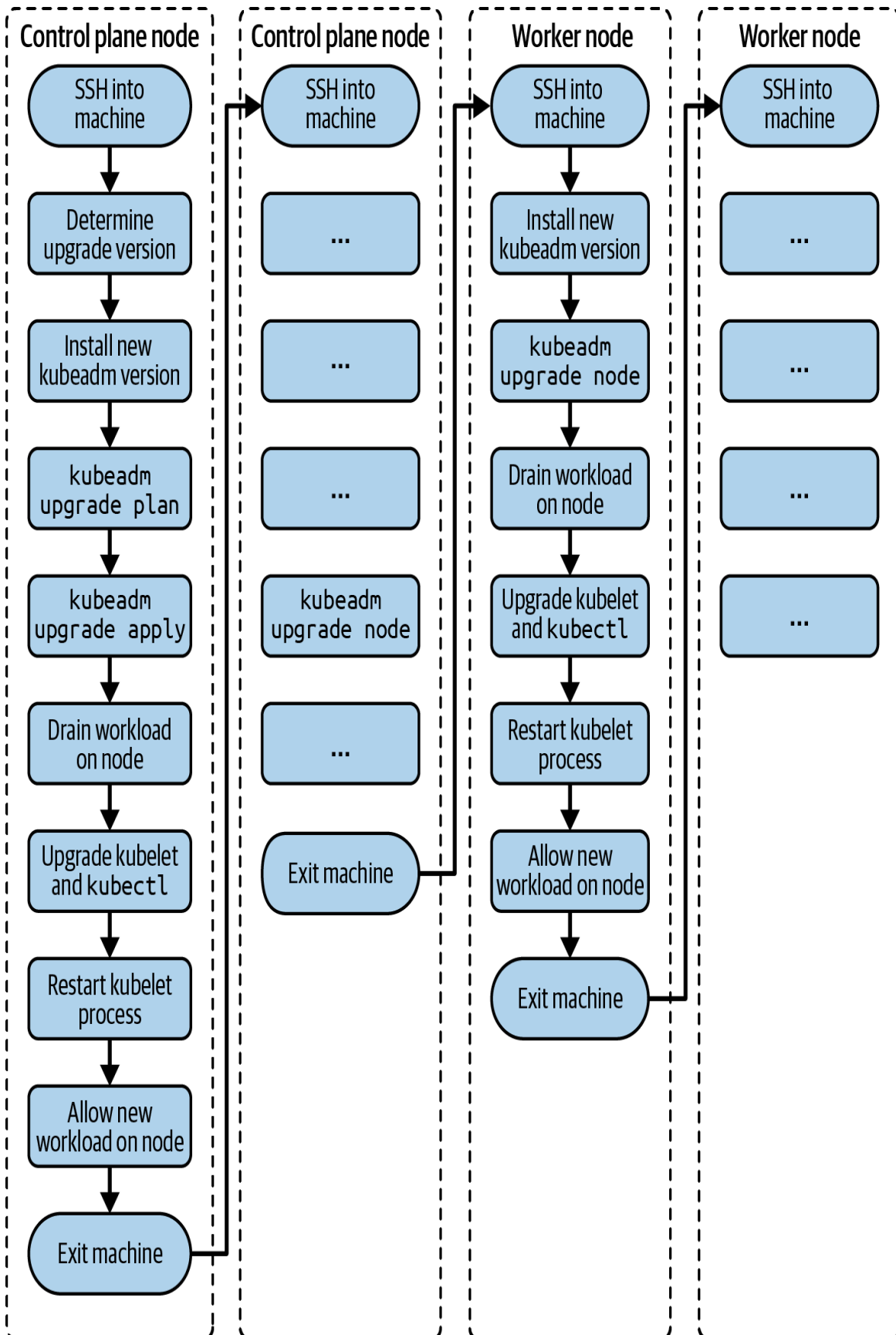
Similar to the stacked etcd topology, each control plane node hosts the API server, the scheduler, and the controller manager. The worker nodes communicate with them through a load balancer. The main difference here is that the etcd instances run on a separate host. This topology decouples etcd from other control plane functionality and therefore has less of an impact on redundancy when a control plane node is lost. As you can see in the illustration,

this topology requires twice as many hosts as the stacked etcd topology.

## Upgrading a Cluster Version

Over time, you will want to upgrade the Kubernetes version of an existing cluster to pick up bug fixes and new features. The upgrade process has to be performed in a controlled manner to avoid the disruption of workload currently in execution and to prevent the corruption of cluster nodes.

It is recommended to upgrade from a minor version to a next higher one (e.g., from 1.18.0 to 1.19.0), or from a patch version to a higher one (e.g., from 1.18.0 to 1.18.3). Abstain from jumping up multiple minor versions to avoid unexpected side effects. You can find a full description of the **upgrade steps** in the official Kubernetes documentation. **Figure 4-4** illustrates the upgrade process.



*Figure 4-4. Process for a cluster version upgrade*

## Upgrading control plane nodes

As explained earlier, a Kubernetes cluster may employ one or many control plane nodes to better support high-availability and scalability concerns. When upgrading a cluster version, this change needs to happen for control plane nodes one at a time.

Pick one of the control plane nodes that contains the kubeconfig file (located at `/etc/kubernetes/admin.conf`), and open an interactive shell to the control plane node using the `ssh` command. The following command targets the control plane node named `kube-control-plane` running Ubuntu 24.10:

```
$ ssh kube-control-plane
Welcome to Ubuntu 24.10 (GNU/Linux 6.11.0-8-generic
aarch64)
...
```

First, check the nodes and their Kubernetes versions. In this setup, all nodes run on version 1.31.1. We are dealing with only a single control plane node and a single worker node:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE
VERSION			
kube-control-plane	Ready	control-plane	4m54s
v1.31.1			
kube-worker-1	Ready	<none>	3m18s
v1.31.1			

Start by upgrading the `kubeadm` version. Identify the version you'd like to upgrade to. On Ubuntu machines, you can use the following `apt-get` command. The version format usually includes a patch



version (e.g., 1.20.7-00). Check the Kubernetes documentation if your machine is running a different operating system:

```
$ sudo apt update
...
$ sudo apt-cache madison kubeadm
  kubeadm | 1.31.5-1.1 |
https://pkgs.k8s.io/core:/stable:/v1.31/deb Packages
  kubeadm | 1.31.4-1.1 |
https://pkgs.k8s.io/core:/stable:/v1.31/deb Packages
  kubeadm | 1.31.3-1.1 |
https://pkgs.k8s.io/core:/stable:/v1.31/deb Packages
  kubeadm | 1.31.2-1.1 |
https://pkgs.k8s.io/core:/stable:/v1.31/deb Packages
  kubeadm | 1.31.1-1.1 |
https://pkgs.k8s.io/core:/stable:/v1.31/deb Packages
  kubeadm | 1.31.0-1.1 |
https://pkgs.k8s.io/core:/stable:/v1.31/deb Packages
```

Upgrade `kubeadm` to a target version. Say you'd want to upgrade to version 1.31.5-1.1. The following series of commands installs `kubeadm` with that specific version and checks the currently installed version to verify:

```
$ sudo apt-mark unhold kubeadm && sudo apt-get update &&
sudo apt-get install \
  -y kubeadm=1.31.5-1.1 && sudo apt-mark hold kubeadm
Canceled hold on kubeadm.
...
Unpacking kubeadm (1.31.5-1.1) over (1.31.1-1.1) ...
Setting up kubeadm (1.31.5-1.1) ...
kubeadm set on hold.
$ sudo apt-get update && sudo apt-get install -y --allow-
change-held-packages \
  kubeadm=1.31.5-1.1
...
kubeadm is already the newest version (1.31.5-1.1).
0 upgraded, 0 newly installed, 0 to remove and 94 not
```

upgraded.

**\$ kubectl version**

```
kubectl version: &version.Info{Major:"1", Minor:"31",
GitVersion:"v1.31.5", \
GitCommit:"af64d838aacd9173317b39cf273741816bd82377",
GitTreeState:"clean", \
BuildDate:"2025-01-15T14:39:21Z", GoVersion:"go1.22.10",
Compiler:"gc", \
Platform:"linux/arm64"}
```

Check which versions are available to upgrade to and validate whether your current cluster is upgradable. You can see in the output of the following command that we could upgrade to version 1.31.5:

**\$ kubectl upgrade plan**

```
...
[upgrade] Fetching available versions to upgrade to
[upgrade/versions] Cluster version: 1.31.5
[upgrade/versions] kubectl version: v1.31.5
I0130 22:26:53.887541    13574 version.go:261] remote
version is \
much newer: v1.32.1; falling back to: stable-1.31
[upgrade/versions] Target version: v1.31.5
[upgrade/versions] Latest version in the v1.31 series:
v1.31.
```

As described in the console output, we'll start the upgrade for the control plane. The process may take a couple of minutes. You may have to upgrade the CNI plugin as well. Follow the provider instructions for more information:

**\$ kubectl upgrade apply v1.31.5**

```
...
[upgrade/version] You have chosen to change the cluster
version to "v1.31.5"
[upgrade/versions] Cluster version: v1.31.5
[upgrade/versions] kubectl version: v1.31.5
```

...

[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.31.5". Enjoy!

[upgrade/kubelet] Now that your control plane is upgraded, please proceed \ with upgrading your kubelets if you haven't already done so.

**Drain the control plane node by evicting the workload. Any new workload won't be schedulable on the node until uncordoned:**

```
$ kubectl drain kube-control-plane --ignore-daemonsets
node/kube-control-plane cordoned
WARNING: ignoring DaemonSet-managed Pods: kube-
system/calico-node-qndb9, \
kube-system/kube-proxy-vpvm
evicting pod kube-system/calico-kube-controllers-
65f8bc95db-krp72
evicting pod kube-system/coredns-f9fd979d6-2brkq
pod/calico-kube-controllers-65f8bc95db-krp72 evicted
pod/coredns-f9fd979d6-2brkq evicted
node/kube-control-plane evicted
```

**Upgrade the kubelet and the kubectl tool to the same version:**

```
$ sudo apt-mark unhold kubelet kubectl && sudo apt-get
update && sudo \
  apt-get install -y kubelet=1.31.5-1.1 kubectl=1.31.5-1.1
&& sudo apt-mark \
  hold kubelet kubectl
...
Setting up kubelet (1.31.5-1.1) ...
Setting up kubectl (1.31.5-1.1) ...
kubelet set on hold.
kubectl set on hold.
```

**Restart the kubelet process:**

```
$ sudo systemctl daemon-reload
$ sudo systemctl restart kubelet
```

Reenable the control plane node back so that the new workload can become schedulable:

```
$ kubectl uncordon kube-control-plane
node/kube-control-plane uncordoned
```

The control plane nodes should now show the usage of Kubernetes 1.31.5:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kube-control-plane	Ready	control-plane	21h	v1.31.5
kube-worker-1	Ready	<none>	21h	v1.31.1

Exit the control plane node using the `exit` command:

```
$ exit
logout
...
```

## Upgrading worker nodes

Pick one of the worker nodes and open an interactive shell to the node using the `ssh` command. The following command targets the worker node named `kube-worker-1` running Ubuntu 24.10:

```
$ ssh kube-worker-1
Welcome to Ubuntu 24.10 (GNU/Linux 6.11.0-8-generic
aarch64)
...
```

Upgrade `kubeadm` to a target version. This is the same command you used for the control plane node, as explained earlier:

```
$ sudo apt-mark unhold kubeadm && sudo apt-get update &&
sudo apt-get install \
  -y kubeadm=1.31.5-1.1 && sudo apt-mark hold kubeadm
Canceled hold on kubeadm.
...
Unpacking kubeadm (1.31.5-1.1) over (1.31.1-1.1) ...
Setting up kubeadm (1.31.5-1.1) ...
kubeadm set on hold.
$ kubeadm version
kubeadm version: &version.Info{Major:"1", Minor:"31",
GitVersion:"v1.31.5", \
GitCommit:"af64d838aacd9173317b39cf273741816bd82377",
GitTreeState:"clean", \
BuildDate:"2025-01-15T14:39:21Z", GoVersion:"go1.22.10",
Compiler:"gc", \
Platform:"linux/arm64"}
```

Upgrade the kubelet configuration:

```
$ sudo kubeadm upgrade node
[upgrade] Reading configuration from the cluster...
[upgrade] FYI: You can look at this config file with \
'kubectl -n kube-system get cm kubeadm-config -o yaml'
[preflight] Running pre-flight checks
[preflight] Skipping prepull. Not a control plane node.
[upgrade] Skipping phase. Not a control plane node.
[upgrade] Skipping phase. Not a control plane node.
[upgrade] Backing up kubelet config file to \
/etc/kubernetes/tmp/kubeadm-kubelet-
config3058962439/config.yaml
[kubelet-start] Writing kubelet configuration to file \
"/var/lib/kubelet/config.yaml"
[upgrade] The configuration for this node was successfully
updated!
[upgrade] Now you should go ahead and upgrade the kubelet
```

```
package \  
using your package manager.
```

Drain the worker node by evicting the workload. Any new workload won't be schedulable on the node until uncordoned:

```
$ kubectl drain kube-worker-1 --ignore-daemonsets  
node/kube-worker-1 cordoned  
WARNING: ignoring DaemonSet-managed Pods: kube-  
system/calico-node-2hrxg, \  
kube-system/kube-proxy-qf6nl  
evicting pod kube-system/calico-kube-controllers-  
65f8bc95db-kggbr  
evicting pod kube-system/coredns-f9fd979d6-7zm4q  
evicting pod kube-system/coredns-f9fd979d6-tlmhq  
pod/calico-kube-controllers-65f8bc95db-kggbr evicted  
pod/coredns-f9fd979d6-7zm4q evicted  
pod/coredns-f9fd979d6-tlmhq evicted  
node/kube-worker-1 evicted
```

Upgrade the kubelet and the `kubectl` tool with the same command used for the control plane node:

```
$ sudo apt-mark unhold kubelet kubectl && sudo apt-get  
update && sudo apt-get \  
  install -y kubelet=1.31.5-1.1 kubectl=1.31.5-1.1 && sudo  
apt-mark hold kubelet \  
  kubectl  
...  
Setting up kubelet (1.31.5-1.1) ...  
Setting up kubectl (1.31.5-1.1) ...  
kubelet set on hold.  
kubectl set on hold.
```

Restart the kubelet process:

```
$ sudo systemctl daemon-reload
```

```
$ sudo systemctl restart kubelet
```

Reenable the worker node so that the new workload can become schedulable:

```
$ kubectl uncordon kube-worker-1
node/kube-worker-1 uncordoned
```

Listing the nodes should now show version 1.31.5 for the worker node. You won't be able to run `kubectl get nodes` from the worker node without copying the administrator kubeconfig file from the control plane node. Follow the [instructions](#) in the Kubernetes documentation to do so or log back into the control plane node:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kube-control-plane	Ready	control-plane	24h	v1.31.5
kube-worker-1	Ready	<none>	24h	v1.31.5

Exit the worker node using the `exit` command:

```
$ exit
logout
...
```

## Summary

As a Kubernetes administrator, you need to be familiar with typical tasks involving the management of the cluster nodes. The primary tool for installing new nodes and upgrading a node version is `kubeadm`. The cluster topology of such a cluster can vary. For optimal results with redundancy and scalability, consider configuring

the cluster with a high-availability setup that uses three or more control plane nodes and dedicated etcd hosts.

## **Exam Essentials**

### *Understand why you need to provision infrastructure*

Every cluster node needs to run on a physical or virtual machine. Provisioning the hardware is the job of an administrator, though you will not have to have hands-on experience for the exam. Explore the manual and automated approaches for provisioning hardware as you will need it for installing a cluster.

### *Know how to create a Kubernetes cluster from scratch*

Installing new cluster nodes and upgrading the version of an existing cluster node are typical tasks performed by a Kubernetes administrator. You do not need to memorize all the steps involved. The documentation provides a step-by-step, easy-to-follow manual for those operations. During the exam, pull the relevant documentation and copy-paste the commands.

### *Practice the cluster upgrade process*

The cluster upgrade process involves executing more commands than the installation process. It's important to remember that you only jump up by a single minor version or multiple patch versions before tackling the next higher version. I'd suggest you open the upgrade documentation page and walk through the process a couple of times.

### *Have a theoretical understanding of high-availability cluster topologies*

High-availability clusters help with redundancy and scalability. For the exam, you will need to understand the different HA



topologies, though it's unlikely that you'll have to configure one of them as the process would involve a suite of different hosts.

## Sample Exercises

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

1. Create a cluster with four nodes, one control plane node, and three worker nodes.

Create a Pod named `nginx` that uses the container image `nginx:1.27.4-alpine`.

Identify the node the Pod has been scheduled on.

Evict all Pods from the node that runs the Pod at once. Do not use the `kubectl delete pod` command to perform the operation. Ensure that the Pod is not running anymore.

*Prerequisite:* To create a cluster with 4 nodes using `minikube`, run the command `minikube start --nodes 4`.

2. Navigate to the directory `app-a/ch04/upgrade-cluster-version` of the checked-out GitHub repository [bmuschko/cka-study-guide](#).

Start up the VMs running the cluster using the command `vagrant up`. The cluster consists of a single control plane node named `kube-control-plane`, and one worker node named `kube-worker-1`. Open an interactive shell into the control plane node and inspect the currently-used Kubernetes version by listing all the nodes.

Upgrade all nodes of the cluster from Kubernetes 1.32.1 to 1.32.2.

Once done, shut down the cluster using `vagrant destroy -f`.

*Prerequisite:* This exercise requires the installation of the tools **Vagrant** and a **VMware provider**.

# Chapter 5. Backing Up and Restoring etcd

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

Kubernetes stores both the declared and observed states of the cluster in the distributed etcd key-value store. It’s important to have a backup plan in place that can help you with restoring the data in case of data corruption. Backing up the data should happen periodically in short time frames to avoid losing as little historical data as possible.

## COVERAGE OF CURRICULUM OBJECTIVES

This chapter addresses the following curriculum objective:

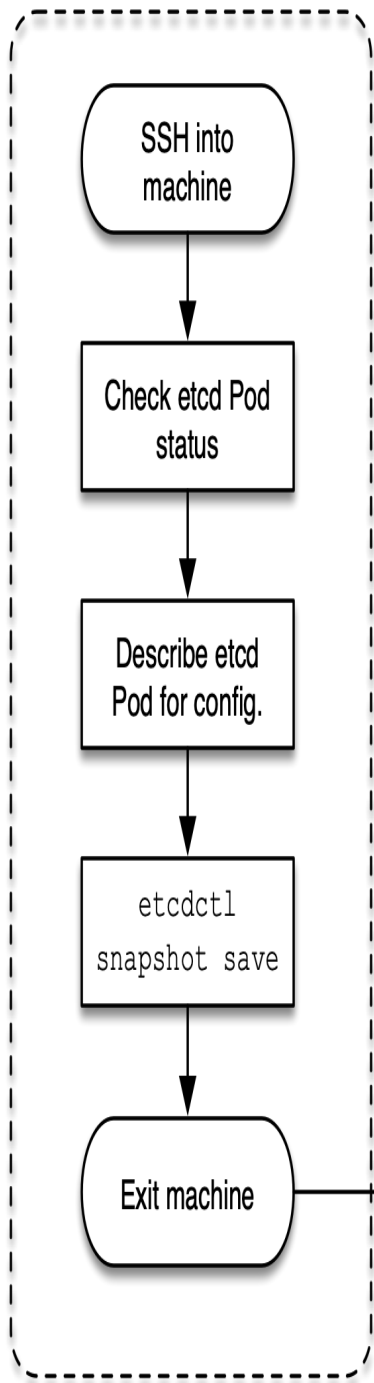
- Manage the lifecycle of Kubernetes clusters

## Using the etcd Administration Utilities

The backup process stores the etcd data in a so-called snapshot file. This snapshot file can be used to restore the etcd data at any given time. You can encrypt the snapshot file to protect sensitive information. The tool `etcdctl` is used to create a backup snapshot file. Restoring the etcd data from the snapshot file requires the use of the tool `etcdutl`.

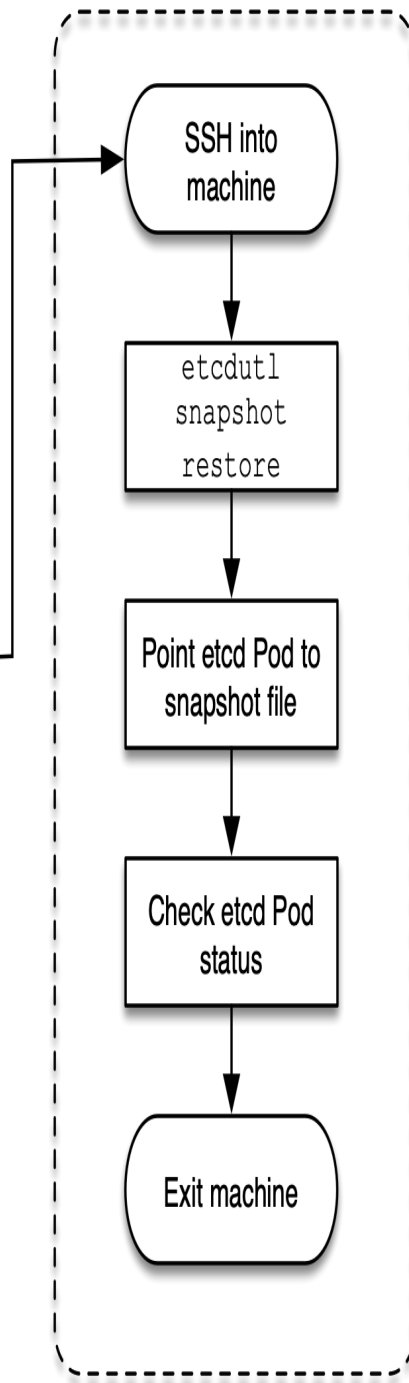
As an administrator, you will need to understand how to use the tools for both operations. You may need to install `etcdctl` and `etcdutl` if they are not available on the control plane node yet. You can find [installation instructions](#) in the etcd GitHub repository. **Figure 5-1** visualizes the etcd backup and restoration process.

*etcd Node*



*Backup*

*etcd Node*



*Restoration*

Disaster

*Figure 5-1. Process for a backing up and restoring etcd*

Depending on your cluster topology, your cluster may consist of one or many etcd instances. Refer to “[Managing a Highly Available Cluster](#)” for more information on how to set it up. The following sections explain a single-node etcd cluster setup. You can find [additional instructions](#) on the backup and restoration process for multinode etcd clusters in the official Kubernetes documentation.

## Backing Up etcd

Open an interactive shell to the machine hosting etcd using the `ssh` command. The following command targets the control plane node named `kube-control-plane` running Ubuntu 24.04 LTS:

```
$ ssh kube-control-plane
Welcome to Ubuntu 24.04 LTS (GNU/Linux 6.8.0-51-generic
x86_64)
...
```

Check the installed version of `etcdctl` to verify that the tool has been installed. On this node, the version is 3.5.15:

```
$ etcdctl version
etcdctl version: 3.5.15
API version: 3.5
```

Etcd is deployed as a Pod in the `kube-system` namespace. Inspect the version by describing the Pod. In the following output, you will find that the version is 3.5.15-0:

```
$ kubectl get pods -n kube-system
```

NAME	READY	STATUS
------	-------	--------

```

RESTARTS    AGE
...
etcd-kube-control-plane          1/1      Running
0                               33m
...
$ kubectl describe pod etcd-kube-control-plane -n kube-
system
...
Containers:
  etcd:
    Container ID:
containerd://47a6cf3ed27d455be6c9b782d2e35ee77b429ee5c0b \
3c6c3d6282628f6492b15
    Image:          registry.k8s.io/etcd:3.5.15-0
    Image ID:
registry.k8s.io/etcd@sha256:a6dc63e6e8cfa0307d7851762fa6 \
b629afb18f28d8aa3fab5a6e91b4af60026a
...

```

The same `describe` command reveals the configuration of the `etcd` service. Look for the value of the option `--listen-client-urls` for the endpoint URL. In the following output, the host is `localhost`, and the port is `2379`. The server certificate is located at `/etc/kubernetes/pki/etcd/server.crt` defined by the option `--cert-file`. The CA certificate can be found at `/etc/kubernetes/pki/etcd/ca.crt` specified by the option `--trusted-ca-file`:

```

$ kubectl describe pod etcd-kube-control-plane -n kube-
system
...
Containers:
  etcd:
    ...
    Command:
    etcd
    ...
    --cert-file=/etc/kubernetes/pki/etcd/server.crt

```

```
--key-file=/etc/kubernetes/pki/etcd/server.key
--listen-client-
urls=/etc/kubernetes/pki/etcd/server.key
--trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
...
```

Use the `etcdctl` command to create the backup with version 3 of the tool. For a good starting point, copy the command from the [official Kubernetes documentation](#). Provide the mandatory command-line options `--cacert`, `--cert`, and `--key`. The option `--endpoints` is not needed as we are running the command on the same server as `etcd`. After running the command, the file `/opt/etcd-backup.db` has been created:

```
$ sudo ETCDCTL_API=3 etcdctl --
cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  snapshot save /opt/etcd-backup.db
...
Snapshot saved at /opt/etcd-backup.db
```

Exit the node using the `exit` command:

```
$ exit
logout
...
```

## Restoring etcd

You created a backup of `etcd` and stored it in a safe space. There's nothing else to do at this time. Effectively, it's your insurance policy that becomes relevant when disaster strikes. In the case of a disaster scenario, the data in `etcd` gets corrupted or the machine



managing etcd experiences a physical storage failure. That's the time when you want to pull out the etcd backup for restoration.

To restore etcd from the backup, use the `etcdctl snapshot restore` command. At a minimum, provide the `--data-dir` command-line option.

### NOTE

The `snapshot restore` command for the `etcdctl` executable has been deprecated and will be **removed** with the release of etcd 3.6. Use the `etcdctl` executable instead.

Here, we are using the data directory */tmp/from-backup*. After running the command, you should be able to find the restored backup in the directory */var/lib/from-backup*:

```
$ sudo ETCDCTL_API=3 etcdctl --data-dir=/var/lib/from-backup snapshot restore \
/opt/etcd-backup.db
...
$ sudo ls /var/lib/from-backup
member
```

Edit the YAML manifest of the etcd Pod, which can be found at */etc/kubernetes/manifests/etcd.yaml*. Change the value of the attribute `spec.volumes.hostPath` with the name `etcd-data` from the original value */var/lib/etcd* to */var/lib/from-backup*:

```
$ cd /etc/kubernetes/manifests/
$ sudo vim etcd.yaml
...
spec:
  volumes:
```

```
...
- hostPath:
  path: /var/lib/etcd-from-backup
  type: DirectoryOrCreate
  name: etcd-data
...
```

The `etcd-kube-control-plane` Pod will be re-created, and it points to the restored backup directory:

```
$ kubectl get pod etcd-kube-control-plane -n kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
etcd-kube-control-plane	1/1	Running	0	5m1s

In case the Pod doesn't transition into the "Running" status, try to delete it manually with the command `kubectl delete pod etcd-kube-control-plane -n kube-system`.

Exit the node using the `exit` command:

```
$ exit
logout
...
```

## Summary

Backing up the etcd database should be performed as a periodic process to prevent the loss of crucial data in the event of a node or storage corruption. You can use the tool `etcdctl` to back up and the tool `etcdutl` restore etcd from the control plane node or via an API endpoint.

## Exam Essentials

### *Practice backing up etcd*

Backing up etcd requires the installation of a compatible version of the `etcdctl` executable. You can identify the version of etcd by inspecting the container image tag used to run the etcd Pod (assuming we are talking about a cluster without **high-availability characteristics**). The etcd process run in the container of the Pod also list the command line flags needed to drive the backup process. In the exam, you can assume that the `etcdctl` executable has been preinstalled.

### *Know how to restore etcd*

Restoring etcd requires the use of the executable `etcdctl`. You will need to point the command to the snapshot file created in the backup process, and a target directory used to extract the etcd data in. Just extracting the etcd data into a directory doesn't tell the etcd process to use it. You need to configure the host path to the directory in the configuration for etcd.

## Sample Exercises

Solutions to these exercises are available in **Appendix A**.

1. Navigate to the directory `app-a/ch05/etcd-backup-restore` of the checked-out GitHub repository ***bmuschko/cka-study-guide***. Start up the VMs running the cluster using the command `vagrant up`. The cluster consists of a single control plane node named `kube-control-plane`, and one worker node named `kube-worker-1`. Open an interactive shell into the control plane node and inspect the currently-used Kubernetes version by listing all the nodes.

SSH into the control plane node host machine. Identify the Pod that runs the etcd executable. Inspect the details of the Pod to find out which version of etcd it is running. Write the version to the file *etcd-version.txt*.

Once done, shut down the cluster using `vagrant destroy -f`.

*Prerequisite:* This exercise requires the installation of the tools **Vagrant** and a **VMware provider**.

2. Navigate to the directory *app-a/ch05/etcd-backup-restore* of the checked-out GitHub repository *bmuschko/cka-study-guide*. Start up the VMs running the cluster using the command `vagrant up`. The cluster consists of a single control plane node named `kube-control-plane`, and one worker node named `kube-worker-1`. Open an interactive shell into the control plane node and inspect the currently-used Kubernetes version by listing all the nodes.

The `etcdctl` and `etcdutl` tool have been preinstalled on the node `kube-control-plane`. Back up etcd to the snapshot file */opt/etcd.bak*. Restore etcd from the snapshot file. Use the data directory */var/bak*.

Once done, shut down the cluster using `vagrant destroy -f`.

*Prerequisite:* This exercise requires the installation of the tools **Vagrant** and a **VMware provider**.

# Chapter 6. Authentication, Authorization, and Admission Control

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

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, 2018).

## COVERAGE OF CURRICULUM OBJECTIVES

This chapter addresses the following curriculum objective:

- Manage role based access control (RBAC)

## Processing an API Request

Figure 6-1 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](#).

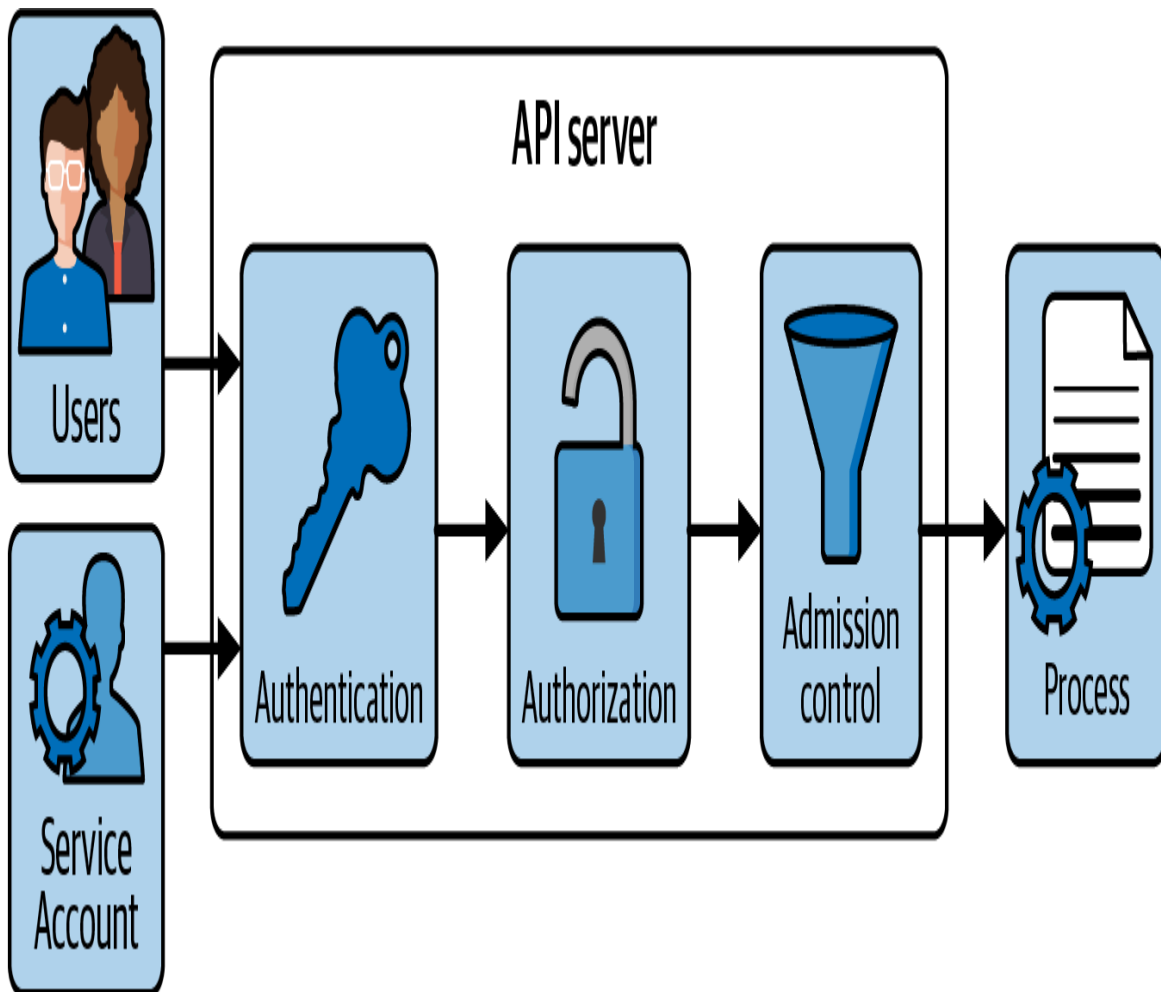


Figure 6-1. API server request processing

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 whether 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.

## NOTE

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 6-1** 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 6-1. A kubeconfig file*

---

```
apiVersion: v1
kind: Config
clusters: ❶
- 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: ❷
- 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 ❸
preferences: {}
users: ❹
- 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.

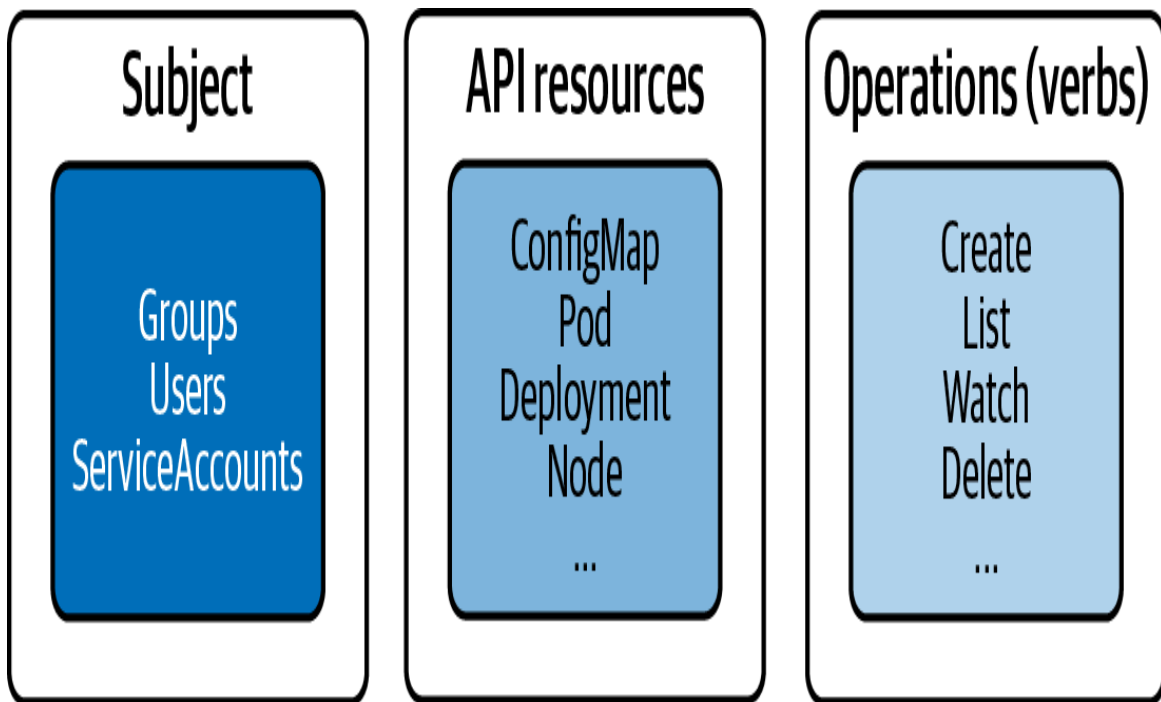
Setting permissions is the responsibility of a cluster administrator. The following sections briefly cover the effects of RBAC on requests from users and service accounts.

## 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 6-2**. 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 6-2. 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:

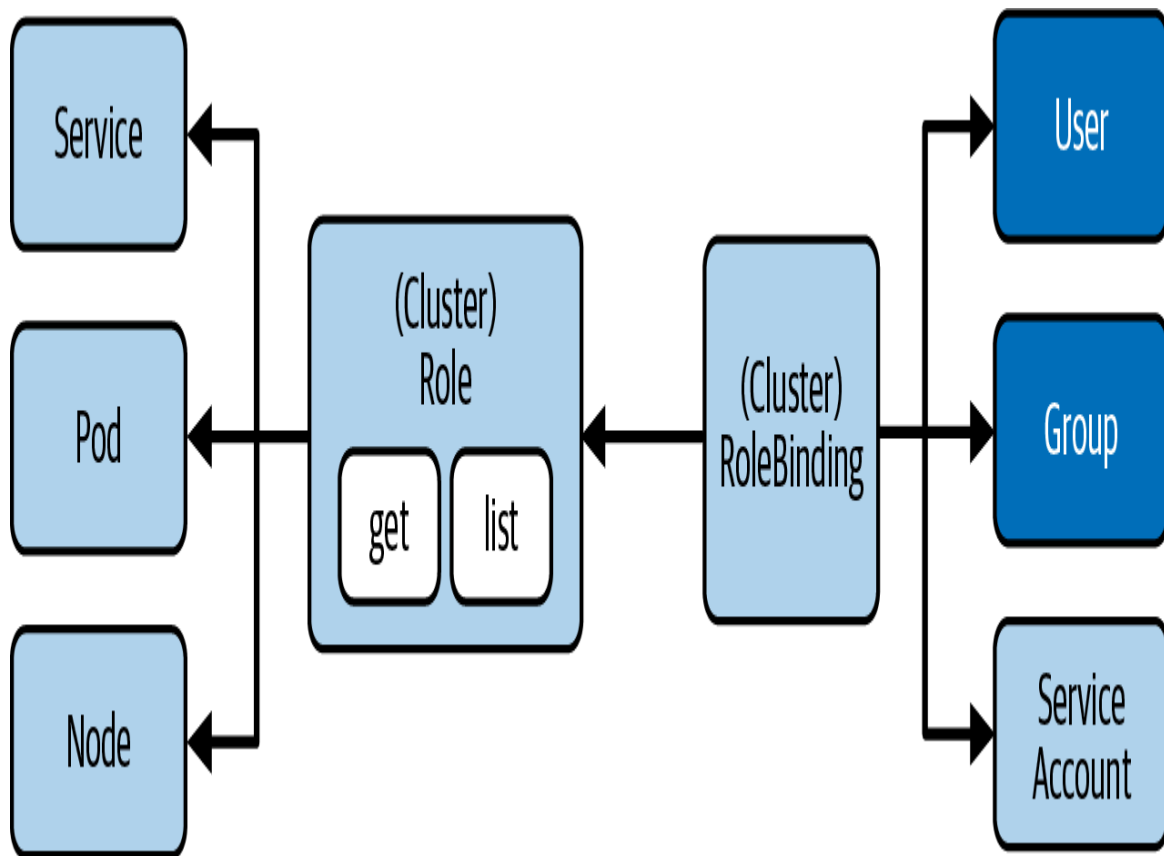
### *Role*

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 6-3** 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 6-3. 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”**.

## 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 6-1** describes the default user-facing Roles.

*Table 6-1. 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 6-2](#), 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 6-2. 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: ❶
  - apps
  resources:
  - deployments
  verbs:
  - list
  - get
  - watch

```

- ❶ Any resource that belongs to an API group needs 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 6-3** 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 6-3. 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 replicaset
Error from server (Forbidden): replicaset.apps is
forbidden: User "bmuschko" \
cannot list resource "replicaset" 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): deployment.apps "myapp" is
forbidden: User \
"bmuschko" cannot delete resource "deployment" 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]
$ kubect1 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.

## Aggregating RBAC Rules

Existing `ClusterRoles` can be aggregated to avoid having to redefine a new, composed set of rules that likely leads to duplication of instructions. For example, say you wanted to combine a user-facing role with a custom Role. An aggregated `ClusterRule` can merge rules

via label selection without having to copy-paste the existing rules into one.

Say we defined two ClusterRoles shown in [Example 6-4](#) and [Example 6-5](#). The ClusterRole `list-pods` allows for listing Pods and the ClusterRole `delete-services` allows for deleting Services.

*Example 6-4. A YAML manifest defining a ClusterRole for listing Pods*

---

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: list-pods
  namespace: rbac-example
  labels:
    rbac-pod-list: "true"
rules:
- apiGroups:
  - ""
  resources:
  - pods
  verbs:
  - list
```

*Example 6-5. A YAML manifest defining a ClusterRole for deleting Services*

---

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: delete-services
  namespace: rbac-example
  labels:
    rbac-service-delete: "true"
rules:
- apiGroups:
  - ""
  resources:
  - services
  verbs:
  - delete
```

To aggregate those rules, ClusterRoles can specify an `aggregationRule`. This attribute describes the label selection rules. **Example 6-6** shows an aggregated ClusterRole defined by an array of `matchLabels` criteria. The ClusterRole does not add its own rules as indicated by `rules: []`; however, there's no limiting factor that would disallow it.

*Example 6-6. A YAML manifest defining a ClusterRole with aggregated rules*

---

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pods-services-aggregation-rules
  namespace: rbac-example
aggregationRule:
  clusterRoleSelectors:
  - matchLabels:
      rbac-pod-list: "true"
  - matchLabels:
      rbac-service-delete: "true"
rules: []
```

We can verify the proper aggregation behavior of the ClusterRole by describing the object. You can see in the following output that both ClusterRoles, `list-pods` and `delete-services`, have been taken into account:

```
$ kubectl describe clusterroles pods-services-aggregation-rules -n rbac-example
```

```
Name:                pods-services-aggregation-rules
Labels:              <none>
Annotations:         <none>
PolicyRule:
  Resources  Non-Resource URLs  Resource Names  Verbs
  -----  -
  services   []                  []              [delete]
  pods       []                  []              [list]
```

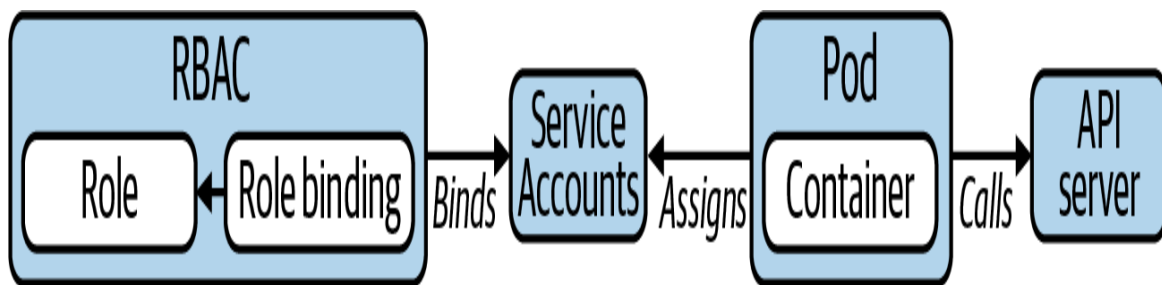


For more information on ClusterRole label selection rules, see the [official documentation](#). The page also explains how to aggregate the default user-facing ClusterRoles.

## 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 6-4](#).



*Figure 6-4. 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 a 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 6-7](#).

#### *Example 6-7. 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 `RoleBinding` 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 auto-mount 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 6-8** defines the namespace, the service account, and the Pod in a single YAML manifest file: *setup.yaml*.

*Example 6-8. 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
  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'] ❷
- 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']

```

❸

- ❶ The service account referenced by name used for communicating with the Kubernetes API.
- ❷ Performs an API call to retrieve the list of Pods in the namespace k97.
- ❸ 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 6-9** in the file *role.yaml*. The set of the rules adds only the Pod resource and the verb `list`.

### Example 6-9. 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 6-10** 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 6-10. 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
```

Developers will inadvertently use admission control plugins that have been configured by the administrator. One example is the `LimitRanger` and the `ResourceQuota`, I'll discuss in "[Working with Limit Ranges](#)" and "[Working with Resource Quotas](#)".

## 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 examined 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 automatically mount 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.

*Be aware of the purpose of admission control*

The API server comes with preconfigured admission control plugins that support the functionality of Kubernetes primitives like the LimitRange. For the exam, you will not have to have a deep understanding of enabling or configuring admission control plugins.

## Sample Exercises

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

1. Navigate to the directory *app-a/ch06/rbac-aggregation* of the checked-out GitHub repository *[bmuschko/cka-study-guide](#)*.

Create a ClusterRole with the name `service-view` to the API resources `services` with the operations `get` and `list`. Create the RoleBinding named `ellasmith-service-view` in the `development` namespace. Map the user `ellasmith` to the ClusterRole `service-view`.

Create a ClusterRole with the name `combined`. Aggregate cluster roles based on the matching label key-value pair `rbac.cka.cncf.com/aggregate: "true"`. Render the selected rules of the ClusterRole `combined`. How many rules do you see?

Create a ClusterRole with the name `deployment-modify` to the API resources `deployments` with the operations `create`, `delete`, `patch`, and `update`. Assign the label key-value pair `rbac.cka.cncf.com/aggregate: "true"`. Render the selected rules of the ClusterRole `combined`. How many rules do you see?

Run a command to figure out if the user `ellasmith` can list Services in the namespace `development`. Write the output of the command to the file *list-services-ellasmith.txt*. The output is either `no` or `yes`.

Run a command to figure out if the user `ellasmith` can watch Deployments in the namespace `production`. Write the output of the command to the file *watch-deployments-ellasmith.txt*. The output is either `no` or `yes`.

*Prerequisite:* This exercise requires the installation of the tools **Vagrant** and a **VMware provider**.

2. Create the ServiceAccount named `api-access` in a new namespace called `apps`.

Create a ClusterRole with the name `api-clusterrole`, and create a ClusterRoleBinding named `api-clusterrolebinding`. Map the ServiceAccount from the previous step to the API resources  `pods` with the operations `watch`, `list`, and `get`.

Create a Pod named `operator` with the image `nginx:1.21.1` in the namespace `apps`. Expose the container port 80. Assign the ServiceAccount `api-access` to the Pod. Create another Pod named `disposable` with the image `nginx:1.21.1` in the namespace `rm`. Do not assign the ServiceAccount to the Pod.

Open an interactive shell to the Pod named `operator`. Use the command-line tool `curl` to make an API call to list the Pods in the namespace `rm`. What response do you expect? Use the command-line tool `curl` to make an API call to delete the Pod `disposable` in the namespace `rm`. Does the response differ from the first call? You can find

information about how to interact with Pods using the API via HTTP in the [reference guide](#).

# Chapter 7. Operators and Custom Resource Definitions (CRDs)

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

Kubernetes comes with a core feature set to fulfill the basic needs for running application stacks with a standard set of primitives. For custom use cases, Kubernetes allows for installing extensions to the platform, so-called operators.

A Custom Resource Definition (CRD) is a Kubernetes extension mechanism (often bundled with an operator) for introducing custom API primitives to fulfill requirements not covered by built-in primitives.

This chapter will focus on the installation and configuration of operators, as well the interaction with provided CRDs.

## COVERAGE OF CURRICULUM OBJECTIVES

This chapter addresses the following curriculum objective:

- Understand CRDs, install and configure operators

## Working with Operators

Operators extend the core behavior of the Kubernetes cluster without actually changing the Kubernetes code. You can think of the operator as a plugin to the platform. Operators typically automate tasks that would have to be performed by humans, such as deploying, configuring, scaling, upgrading, and managing applications.

## The Operator Pattern

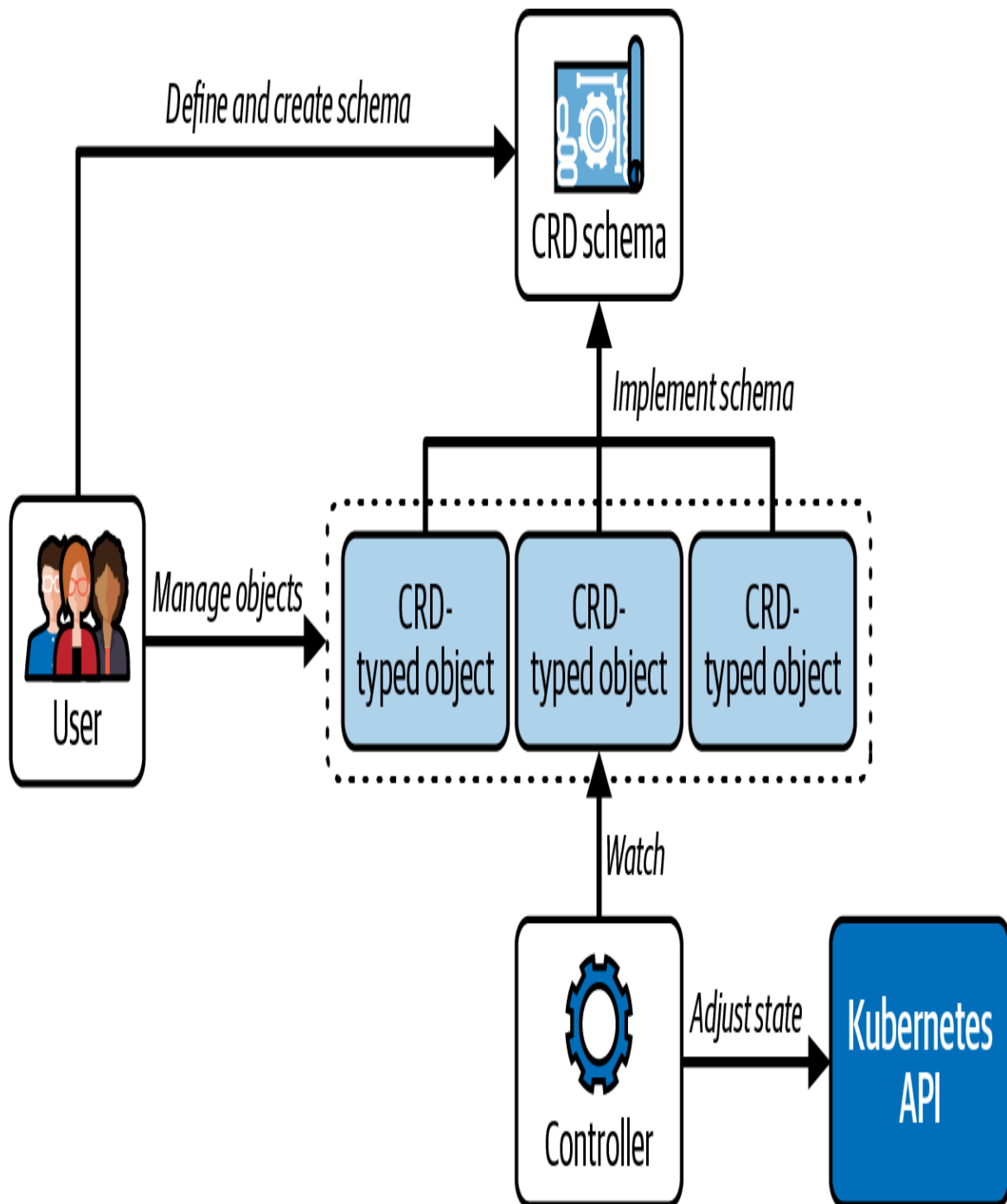
An operator usually consists of two components, one or more Custom Resource Definitions (CRD) and a controller. CRDs can be understood as the schema that defines the blueprint for a custom object, and then the instantiation of those objects with the newly introduced type, also called the Custom Resources (CR).

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 7-1** shows the operator patterns with all its moving parts.





*Figure 7-1. The Kubernetes operator pattern*

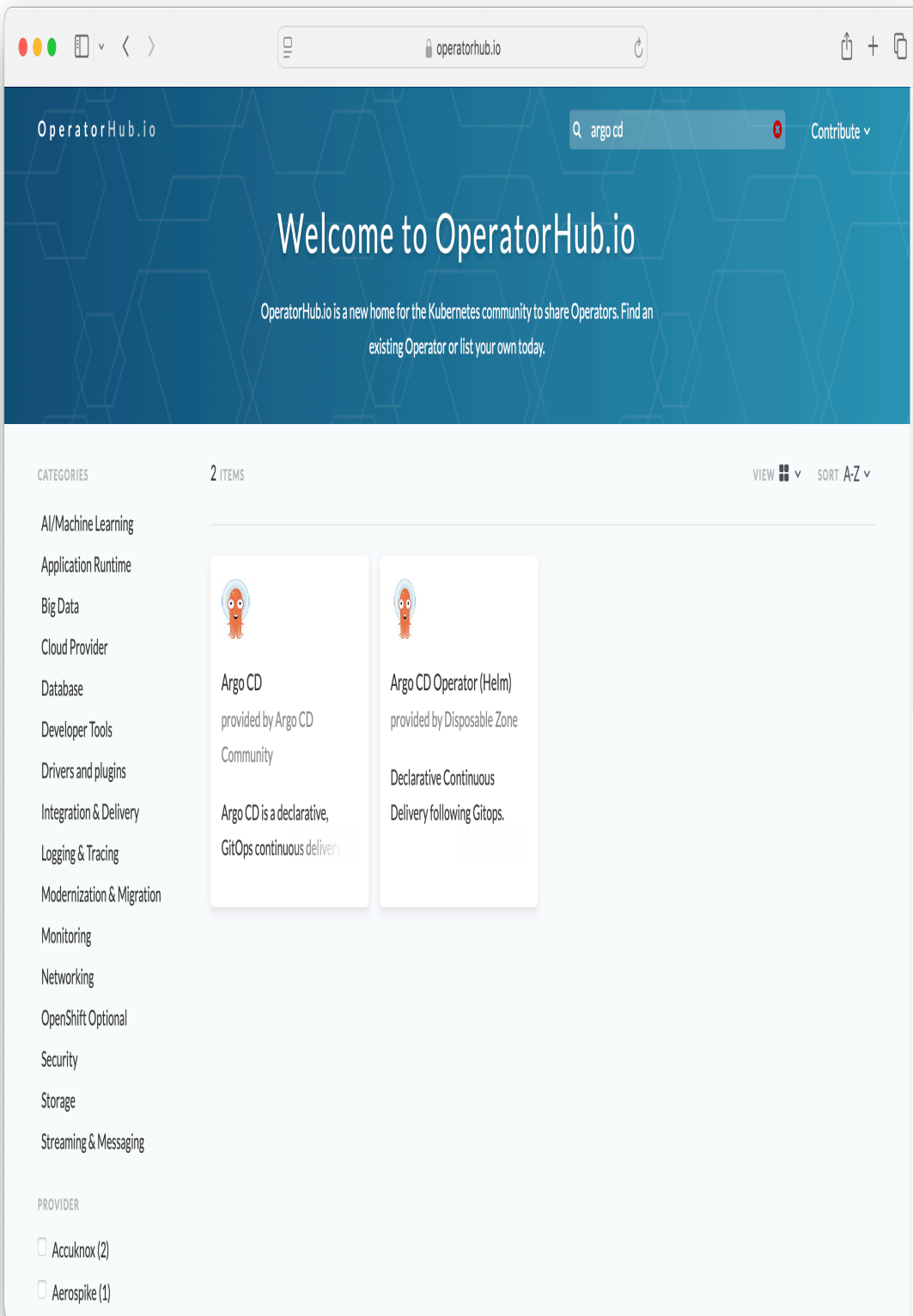
## Discovering Operators

The Kubernetes community has implemented many useful operators discoverable on [Operator Hub](#) or [Artifact Hub](#).

A prominent operator is the **External Secrets Operator** that helps with integrating external Secret managers, like AWS Secrets Manager and HashiCorp Vault, with Kubernetes. Another is the **Crossplane Operator** which helps with creating and managing cloud resources using declarative syntax.

To demonstrate the functionality of Operator Hub, we are going to search and install the popular **Argo CD Operator**, a declarative, GitOps continuous delivery tool for Kubernetes that automates deployments by continuously monitoring applications and synchronizing them with the desired state defined in Git repositories.

In your browser, open the URL for **Operator Hub** and enter the term *argo cd* into the search box named *Search OperatorHub....* You will receive a result, shown in **Figure 7-2**.




*Figure 7-2. A search result on Operator Hub*

Clicking on the *Argo CD* panel will bring you to the details of the **Argo CD Operator**, shown in **Figure 7-3**.

OperatorHub.io

Search OperatorHub...

Contribute ▾



# Argo CD

Argo CD is a declarative, GitOps continuous delivery tool for Kubernetes.

[Home](#) > [Argo CD](#)

## Argo CD

Install

### Overview

The Argo CD Operator manages the full lifecycle for [Argo CD](#) and it's components. The operator's goal is to automate the tasks required when operating an Argo CD cluster.

Beyond installation, the operator helps to automate the process of upgrading, backing up and restoring as needed and remove the human as much as possible. In addition, the operator aims to provide deep insights into the Argo CD environment by configuring Prometheus and Grafana to aggregate, visualize and expose the metrics already exported by Argo CD.

The operator aims to provide the following, and is a work in progress.

- Easy configuration and installation of the Argo CD components with sane defaults to get up and running quickly.
- Provide seamless upgrades to the Argo CD components.
- Ability to back up and restore an Argo CD cluster from a point in time or on a recurring schedule.
- Aggregate and expose the metrics for Argo CD and the operator itself using Prometheus and Grafana.
- Autoscale the Argo CD components as necessary to handle variability in demand.

CHANNEL

alpha

VERSION

0.13.0 (Current) ▾

CAPABILITY LEVEL ⓘ

✓ Basic Install

✓ Seamless Upgrades

✓ Full Lifecycle

✓ Deep Insights

○ Auto Pilot

PROVIDER

Argo CD Community

LINKS

*Figure 7-3. The Argo CD Operator on Operator Hub*

The page describes the functionality of the operator including a high-level overview on the provided CRDs. To be able to create CRs from the CRDs, we'll first need to install the operator.

## **Installing Operators**

You can install many of those operators with a single `kubectl` command execution or by using the Helm executable. For more information on using Helm, see [Chapter 8](#).

Clicking the *Install* button on Operator Hub brings up the installation instructions, as shown in [Figure 7-4](#).

OperatorHub.io

Search OperatorHub

Contribute

Home > Argo CD

Argo CD


Overview

The Argo CD Operator manages the lifecycle of Argo CD components, ensuring they are always up to date and running as expected.

Beyond installation, the operator provides a range of features to help you manage your Argo CD cluster as much as possible. In addition to managing the lifecycle of the components, the operator integrates with Prometheus and Grafana to aggregate, visualize and alert on the health of the cluster.

The operator aims to provide:

- Easy configuration and installation of the Argo CD components with sane defaults to get up and running quickly.
- Provide seamless upgrades to the Argo CD components.
- Ability to back up and restore an Argo CD cluster from a point in time or on a recurring schedule.
- Aggregate and expose the metrics for Argo CD and the operator itself using Prometheus and Grafana.
- Autoscale the Argo CD components as necessary to handle variability in demand.

Argo CD

0.13.0 provided by Argo CD Community

Install on Kubernetes

1. Install Operator Lifecycle Manager (OLM), a tool to help manage the Operators running on your cluster.

```
$ curl -sL https://github.com/operator-framework/operator-lifecycle-manager/releases/download/v0.31.0/install.sh | bash -s v0.31.0
```

2. Install the operator by running the following command:

[What happens when I execute this command?](#)

```
$ kubectl create -f https://operatorhub.io/install/argocd-operator.yaml
```

This Operator will be installed in the "operators" namespace and will be usable from all namespaces in the cluster.

3. After install, watch your operator come up using next command.

```
$ kubectl get csv -n operators
```

To use it, checkout the custom resource definitions (CRDs) introduced by this operator to start using it.

Deep Insights

Auto Pilot

PROVIDER

Argo CD Community

LINKS

*Figure 7-4. The Argo CD Operator installation instructions*

As shown on the installation page for the operator, we'll use the **Operator Lifecycle Manager (OLM)**, a tool to help manage the operators running on your cluster. This is a one-time operation:

```
$ curl -sL https://github.com/operator-framework/operator-  
lifecycle-manager/\n  
releases/download/v0.31.0/install.sh | bash -s v0.31.0
```

Next, we'll install the Argo CD Operator which will place the operator objects into the `operators` namespace:

```
$ kubectl create -f https://operatorhub.io/install/argocd-  
operator.yaml  
subscription.operators.coreos.com/my-argocd-operator  
created
```

You can follow the installation process by running the following command. A valid installation finishes with the `Succeeded` phase in the rendered output of the command.

```
$ kubectl get csv -n operators
```

NAME	DISPLAY	VERSION	REPLACES
argocd-operator.v0.13.0	Argo CD	0.13.0	argocd-operator.v0.12.0
	Succeeded		

You are now ready to use the operator. Please refer to the next sections in the chapter to interact with the installed CRDs.

## Working with Custom Resource Definitions



## (CRDs)

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.

## Discovering CRDs

The Argo CD Operator provides a couple of CRDs like the Application, ApplicationSet, AppProject, and some more. You can find their high-level purpose described below.

- *Application*: An Application is a group of Kubernetes resources as defined by a manifest.
- *ApplicationSet*: An ApplicationSet is a group or set of Application resources.
- *AppProject*: An AppProject is a logical grouping of Argo CD Applications.

Run the following command to list all installed CRDs. You will find the Argo CD CRDs in the command output:

```
$ kubectl get crds
NAME                                                    CREATED AT
applications.argoproj.io                               2025-03-
21T23:02:40Z
applicationsets.argoproj.io                           2025-03-
21T23:02:39Z
appprojects.argoproj.io                               2025-03-
21T23:02:39Z
argocdexports.argoproj.io                             2025-03-
21T23:02:39Z
argocds.argoproj.io                                   2025-03-
21T23:02:39Z
notificationsconfigurations.argoproj.io               2025-03-
21T23:02:39Z
```

As with any other Kubernetes object, you can render the details. The details of the CRD will reveal the kind, API group and version, and its properties. This command inspects the Application CRD.

```
$ kubectl describe crd applications.argoproj.io
```

For brevity, I do not show the lengthy output. In the next section, we are going to create a CR for the Application CRD.

## Instantiating a CR for one of the CRDs

The Application CRD describes a schema for defining an application representation that lives in a Git repository so that it can be deployed to a Kubernetes cluster.

For that purpose, create a new YAML manifest of kind Application in the file *nginx-application.yaml*. as shown in **Example 7-1**. You may recognize some of the properties used when you rendered the CRD schema.

### *Example 7-1. Instantiation of CR for the Application CRD*

---

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: nginx
spec:
  project: default
  source:
    repoURL: https://github.com/bmuschko/cka-study-guide.git
    targetRevision: HEAD
    path: ./ch07/nginx
  destination:
    server: https://kubernetes.default.svc
    namespace: default
```

Create a new object from the YAML manifest:

```
kubectl apply -f nginx-application.yaml
application.argoproj.io/nginx created
```

## Interacting with a CR

You can interact with the CR like any other object in Kubernetes. All `kubectl` create, read, update, and delete (CRUD) functions are available. For example, to list the object use the `describe` command. The following command shows the operations in action:

```
$ kubectl describe application nginx
Name:          nginx
Namespace:     default
Labels:        <none>
Annotations:   <none>
API Version:   argoproj.io/v1alpha1
Kind:          Application
...
```

To delete the object, use the `delete` command, as shown here:

```
$ kubectl delete application nginx
application.argoproj.io "nginx" deleted
```

We demonstrated that an operator can install CRDs into the cluster, and that we can create CR from the CRDs and interact with them.

Going deeper using Argo CD would require a chapter on its own and is beyond the coverage of the certification. For more information on Argo CD, check out the book *Argo CD: Up and Running* by Andrew Block and Christian Hernandez (O'Reilly, 2025).

## Inspecting the controller

The Argo CD CRs just represent data and won't be useful by themselves. A controller acts as a reconciliation process by inspecting the state of CR objects via calls to the Kubernetes API to perform a deployment process to the cluster.

The Argo CD Operator runs the controller logic within a Pod managed by a Deployment. You can discover the Argo CD controller objects as follows:

```
$ kubectl get deployments,pods -n operators
NAME                                                    READY
UP-TO-DATE      ...
deployment.apps/argocd-operator-controller-manager    1/1
1              ...

NAME
READY   STATUS      ...
pod/argocd-operator-controller-manager-6998544bff-zx8bg
1/1     Running     ...
```

Because we installed the operator using OLM, the controller Pod would be placed into the `operators` namespace to keep them separate from other objects in the cluster. You can scale the number of replicas as needed by changing the configuration of the Deployment.

## Summary

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.

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, so-called CRs, 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.

## Exam Essentials

### *Know how to install operators*

Operators built and managed by the Kubernetes community are available on searchable sites like Artifact Hub and Operator Hub. You will find installation instructions on the corresponding web pages. You do not have to memorize them for the exam. If you want to explore further, install an open source operator, such as the **Prometheus operator** or the **Jaeger operator**.

### *Acquire a high-level understanding of configurable options for a CRD schema*

You are not expected to implement a custom CRD schema. All you need to know is how to discover and interact with them using `kubectl`. Practice the definition of a CR in the form of a YAML manifest, and create the objects for it. Controller implementations are definitely outside the scope of the exam.

## 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/ch07/mongodb-operator* of the checked-out GitHub repository *[bmuschko/cka-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. Navigate to the directory *app-a/ch07/backup-crd* of the checked-out GitHub repository *[bmuschko/cka-study-guide](#)*.

Create the CRD from the file *backup-resource.yaml*. Retrieve the details for the `Backup` custom resource created in the previous step.

Create a CR named `nginx-backup` for the CRD in the default namespace. 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.

# Chapter 8. Helm and Kustomize

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 8th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

Kubernetes objects can be created, modified, and deleted by using imperative `kubectl` commands or by running a `kubectl` command against a configuration file declaring the desired state of an object, a so-called manifest. The primary definition language of a manifest is YAML, though you can opt for JSON, which is the less widely adopted format among the Kubernetes community. It’s recommended that development teams commit and push those configuration files to version control repositories as it will help with tracking and auditing changes over time.

Modeling an application in Kubernetes often requires a set of supporting objects, each of which can have its own manifest. For example, you may want to create a Deployment that runs the application on five Pods, a ConfigMap to inject configuration data as environment variables, and a Service for exposing network access.

It is not practical to manage a full application stack by running individual `kubectl` commands. That is where open source tools

like Helm and Kustomize come into play. They allow you conveniently manage the lifecycle of application stacks and cluster components as one single unit, while at the same time allowing for customization at the time of deployment.

### **COVERAGE OF CURRICULUM OBJECTIVES**

This chapter addresses the following curriculum objective:

- Use Helm and Kustomize to install cluster components

## **Working with Helm**

**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* that bundles 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 3.17.2.



## 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**. Revisit **Chapter 7** to refresh your memory on the moving parts of the operator pattern.

The following list shows the typical workflow for consuming and managing a Helm chart. Most of those steps need to use the `helm` executable:

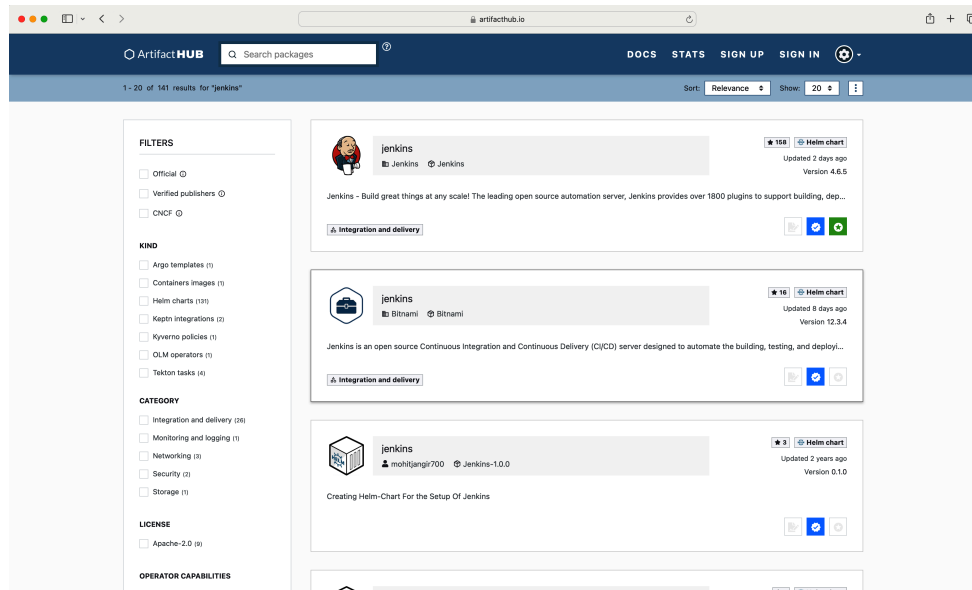
1. Identifying the chart you'd like to install
2. Adding the repository containing the chart
3. Installing the chart from the repository
4. Verifying the Kubernetes objects that have been installed by the chart
5. Rendering the list of installed charts
6. Upgrading an installed chart
7. Uninstalling 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 8-1** shows the list of results in Artifact Hub.



*Figure 8-1. 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 8-2** shows the page for the official Jenkins chart.

artifacthub.io

ArtifactHUB

Search packages

DOCS

STATS

SIGN UP

SIGN IN



jenkins

Helm chart

Integration and delivery

Jenkins

Jenkins

Star

158

Jenkins - Build great things at any scale! The leading open source automation server, Jenkins provides over 1800 plugins to support building, deploying and automating any project.

SUBSCRIPTIONS: 42

WEBHOOKS: 3

PRODUCTION USERS: 4

Jenkins

Report issue

INSTALL

TEMPLATES

DEFAULT VALUES

CHANGELOG



APPLICATION VERSION

2.414.2

CHART VERSIONS

4.6.5 (26 Sep, 2023)

4.6.4 (8 Sep, 2023)

4.6.3 (7 Sep, 2023)

See all (530)

LAST YEAR ACTIVITY

Oct'22 Mar'23 Sep'23

Artifact Hub jenkins

License Apache 2.0

downloads 7.6M

gitter join chat

Jenkins is the leading open source automation server, Jenkins provides over 1800 plugins to support building, deploying and automating any project.

This chart installs a Jenkins server which spawns agents on Kubernetes utilizing the Jenkins Kubernetes plugin.

Inspired by the awesome work of Carlos Sanchez.

Get Repository Info

helm repo add jenkins https://charts.jenkins.io

helm repo update

See `helm repo` for command documentation.

Install Chart

# Helm 3

\$ helm install [RELEASE\_NAME] jenkins/jenkins [flags]

See configuration below.

See `helm install` for command documentation.

*Figure 8-2. 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 the *Install* button to render repository details and the command for adding it. **Figure 8-3** shows the contextual pop-up that appears after clicking *Install*.

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
```



jenkins



## Helm v3

Add repository

```
helm repo add jenkinsci https://charts.jenkins.io/
```



Install chart

```
helm install my-jenkins jenkinsci/jenkins --version 4.6.5
```



***my-jenkins** corresponds to the release name, feel free to change it to suit your needs. You can also add additional flags to the **helm install** command if you need to.*

Need Helm?

You can also download this package's content directly using [this link](#).

× CLOSE

*Figure 8-3. Jenkins chart installation instructions*

As you can see from the pop-up, 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 discussed 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	5.8.26	2.492.2	...

At the time of writing, the latest version available is 5.8.26. 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, 5.8.25. 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 5.8.25
NAME: my-jenkins
LAST DEPLOYED: Wed Mar 26 13:48:50 2025
NAMESPACE: default
STATUS: deployed
REVISION: 1
...
```

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 the *Default Values* button on the chart page, as shown in **Figure 8-4**.



# Default values

Compare to version ... ▾



Search path



```
1 # Default values for jenkins.
2 # This is a YAML-formatted file.
3 # Declare name/value pairs to be passed into your templates.
4 # name: value
5
6 ## Overrides for generated resource names
7 # See templates/_helpers.tpl
8 # nameOverride:
9 # fullnameOverride:
10 # namespaceOverride:
11
12 # For FQDN resolving of the controller service. Change this value to match your existing configuration.
13 # ref: https://github.com/kubernetes/dns/blob/master/docs/specification.md
14 clusterZone: "cluster.local"
15
16 # The URL of the Kubernetes API server
17 kubernetesURL: "https://kubernetes.default"
18
19 renderHelmLabels: true
20
```



× CLOSE

*Figure 8-4. 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
my-jenkins	default	1	2023-09-28...	
CHART				
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
5.8.26
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.

## Working with Kustomize

Kustomize is a tool introduced with Kubernetes 1.14 that aims to make manifest management more convenient. It supports three different use cases:

- Generating manifests from other sources. For example, creating a ConfigMap and populating its key-value pairs from a properties file.
- Adding common configuration across multiple manifests. For example, adding a namespace and a set of labels for a Deployment and a Service.
- Composing and customizing a collection of manifests. For example, setting resource boundaries for multiple Deployments.

The central file needed for Kustomize to work is the kustomization file. The standardized name for the file is *kustomization.yaml* and cannot be changed. A kustomization file defines the processing rules Kustomize works upon.

Kustomize is fully integrated with `kubectl` and can be executed in two modes: rendering the processing output on the console or creating the objects. Both modes can operate on a directory, tarball, Git archive, or URL as long as they contain the kustomization file and referenced resource files:

*Rendering the produced output*

The first mode uses the `kustomize` subcommand to render the produced result on the console but does not create the objects. This command works similarly to the dry-run option you might know from the `run` command:

```
$ kubectl kustomize <target>
```

### *Creating the objects*

The second mode uses the `apply` command in conjunction with the `-k` command-line option to apply the resources processed by Kustomize, as explained in the previous section:

```
$ kubectl apply -k <target>
```

The following sections demonstrate each of the use cases by a single example. For a full coverage on all possible scenarios, refer to the [documentation](#) or the [Kustomize GitHub repository](#).

## **Composing Manifests**

One of the core functionalities of Kustomize is to create a composed manifest from other manifests. Combining multiple manifests into a single one may not seem that useful by itself, but many of the other features described later will build upon this capability. Say you wanted to compose a manifest from a Deployment and a Service resource file. All you need to do is to place the resource files into the same folder as the kustomization file:

```
.
├── kustomization.yaml
```

```
├─ web-app-deployment.yaml
└─ web-app-service.yaml
```

The kustomization file lists the resources in the `resources` section, as shown in **Example 8-1**.

### *Example 8-1. A kustomization file combining two manifests*

---

**resources:**

- web-app-deployment.yaml
- web-app-service.yaml

As a result, the `kustomize` subcommand renders the combined manifest containing all of the resources separated by three hyphens (`---`) to denote the different object definitions:

```
$ kubectl kustomize ./
apiVersion: v1
kind: Service
metadata:
  labels:
    app: web-app-service
  name: web-app-service
spec:
  ports:
  - name: web-app-port
    port: 3000
    protocol: TCP
    targetPort: 3000
  selector:
    app: web-app
  type: NodePort
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: web-app-deployment
  name: web-app-deployment
spec:
  replicas: 3
```

```

selector:
  matchLabels:
    app: web-app
template:
  metadata:
    labels:
      app: web-app
  spec:
    containers:
    - env:
      - name: DB_HOST
        value: mysql-service
      - name: DB_USER
        value: root
      - name: DB_PASSWORD
        value: password
      image: bmuschko/web-app:1.0.1
      name: web-app
      ports:
      - containerPort: 3000

```

## Generating Manifests from Other Sources

Earlier in this chapter, we learned that ConfigMaps and Secrets can be created by pointing them to a file containing the actual configuration data for it. Kustomize can help with the process by mapping the relationship between the YAML manifest of those configuration objects and their data. Furthermore, we'll want to inject the created ConfigMap and Secret in a Pod as environment variables. In this section, you will learn how to achieve this with the help of Kustomize.

The following file and directory structure contains the manifest file for the Pod and the configuration data files we need for the ConfigMap and Secret. The mandatory kustomization file lives on the root level of the directory tree:

```

.
├── config

```



```
|
|   └─ db-config.properties
|   └─ db-secret.properties
└─ kustomization.yaml
   └─ web-app-pod.yaml
```

In *kustomization.yaml*, you can define that the ConfigMap and Secret object should be generated with the given name. The name of the ConfigMap is supposed to be `db-config`, and the name of the Secret is going to be `db-creds`. Both of the generator attributes, `configMapGenerator` and `secretGenerator`, reference an input file used to feed in the configuration data. Any additional resources can be spelled out with the `resources` attribute. **Example 8-2** shows the contents of the kustomization file.

*Example 8-2. A kustomization file using a ConfigMap and Secret generator*

---

```
configMapGenerator:
- name: db-config
  files:
  - config/db-config.properties
secretGenerator:
- name: db-creds
  files:
  - config/db-secret.properties
resources:
- web-app-pod.yaml
```

Kustomize generates ConfigMaps and Secrets by appending a suffix to the name. You can see this behavior when creating the objects using the `apply` command. The ConfigMap and Secret can be referenced by name in the Pod manifest:

```
$ kubectl apply -k ./
configmap/db-config-t4c79h4mtt unchanged
secret/db-creds-4t9dmgtf9h unchanged
pod/web-app created
```

## NOTE

This naming strategy can be configured with the attribute `generatorOptions` in the kustomization file. See the [documentation](#) for more information.

Let's also try the `kustomize` subcommand. Instead of creating the objects, the command renders the processed output on the console:

```
$ kubectl kustomize ./
apiVersion: v1
data:
  db-config.properties: |-
    DB_HOST: mysql-service
    DB_USER: root
kind: ConfigMap
metadata:
  name: db-config-t4c79h4mtt
---
apiVersion: v1
data:
  db-secret.properties:
    REJfUEFTUldPUkQ6IGNHRnpjM2R2Y21RPQ==
kind: Secret
metadata:
  name: db-creds-4t9dmgtf9h
type: Opaque
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: web-app
    name: web-app
spec:
  containers:
    - envFrom:
        - configMapRef:
            name: db-config-t4c79h4mtt
```

```
- secretRef:
  name: db-creds-4t9dmgtf9h
image: bmuschko/web-app:1.0.1
name: web-app
ports:
- containerPort: 3000
  protocol: TCP
restartPolicy: Always
```

## Adding Common Configuration Across Multiple Manifests

Application developers usually work on an application stack set comprised of multiple manifests. For example, an application stack could consist of a frontend microservice, a backend microservice, and a database. It's common practice to use the same, cross-cutting configuration for each of the manifests. Kustomize offers a range of supported fields (e.g., namespace, labels, or annotations). Refer to the [documentation](#) to learn about all supported fields.

For the next example, we'll assume that a Deployment and a Service live in the same namespace and use a common set of labels. The namespace is called `persistence` and the label is the key-value pair `team: helix`. [Example 8-3](#) illustrates how to set those common fields in the kustomization file.

### *Example 8-3. A kustomization file using a common field*

---

```
namespace: persistence
commonLabels:
  team: helix
resources:
- web-app-deployment.yaml
- web-app-service.yaml
```

To create the referenced objects in the kustomization file, run the `apply` command. Make sure to create the `persistence` namespace beforehand:

```
$ kubectl create namespace persistence
namespace/persistence created
$ kubectl apply -k ./
service/web-app-service created
deployment.apps/web-app-deployment created
```

The YAML representation of the processed files looks as follows:

```
$ kubectl kustomize ./
apiVersion: v1
kind: Service
metadata:
  labels:
    app: web-app-service
    team: helix
  name: web-app-service
  namespace: persistence
spec:
  ports:
  - name: web-app-port
    port: 3000
    protocol: TCP
    targetPort: 3000
  selector:
    app: web-app
    team: helix
  type: NodePort
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: web-app-deployment
    team: helix
  name: web-app-deployment
  namespace: persistence
spec:
  replicas: 3
  selector:
    matchLabels:
```

```

    app: web-app
    team: helix
  template:
    metadata:
      labels:
        app: web-app
        team: helix
    spec:
      containers:
      - env:
        - name: DB_HOST
          value: mysql-service
        - name: DB_USER
          value: root
        - name: DB_PASSWORD
          value: password
        image: bmuschko/web-app:1.0.1
        name: web-app
        ports:
        - containerPort: 3000

```

## Customizing a Collection of Manifests

Kustomize can merge the contents of a YAML manifest with a code snippet from another YAML manifest. Typical use cases include adding security context configuration to a Pod definition or setting resource boundaries for a Deployment. The kustomization file allows for specifying different patch strategies like `patchesStrategicMerge` and `patchesJson6902`. For a deeper discussion on the differences between patch strategies, refer to the documentation.

**Example 8-4** shows the contents of a kustomization file that patches a Deployment definition in the file *nginx-deployment.yaml* with the contents of the file *security-context.yaml*.

*Example 8-4. A kustomization file defining a patch*

---

**resources:**

- nginx-deployment.yaml

```
patchesStrategicMerge:
- security-context.yaml
```

The patch file shown in **Example 8-5** defines a security context on the container-level for the Pod template of the Deployment. At runtime, the patch strategy tries to find the container named `nginx` and enhances the additional configuration.

#### *Example 8-5. The patch YAML manifest*

---

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  template:
    spec:
      containers:
      - name: nginx
        securityContext:
          runAsUser: 1000
          runAsGroup: 3000
          fsGroup: 2000
```

The result is a patched Deployment definition, as shown in the output of the `kustomize` subcommand shown next. The patch mechanism can be applied to other files that require a uniform security context definition:

```
$ kubectl kustomize ./
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
```

```
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
      - image: nginx:1.14.2
        name: nginx
        ports:
          - containerPort: 80
        securityContext:
          fsGroup: 2000
          runAsGroup: 3000
          runAsUser: 1000
```

## Key Differences Between Helm and Kustomize

On the surface, Helm and Kustomize may seem to solve the same problems. Here, we want to identify the key difference between those tools so that you can make an educated decision based on the use cases you are trying to fulfill.

### *Ease of use*

Kustomize is bundled with the `kubectl` command line. You do not need to install another tool, nor do you have learn another templating engine.

Helm requires the installation of an executable and requires you to become familiar with its commands and workflow.

### *Learning curve*

Kustomize builds upon the knowledge of an administrator or developer familiar with writing Kubernetes YAML manifests.

Helm has a steeper learning curve. You will have to become familiar with its package management system, the notation for its templating engine, and how to pass in user-defined values upon invocation.

## *Packaging*

Kustomize doesn't require the end user to produce an archive file. All you need is a set of YAML manifests and the *kustomization.yaml* file, which you would check into a Git repository.

Helm on the other hand requires the creation of a metadata file named *Chart.yaml*, default values represented in a file named *values.yaml*, and a set of template files in a *templates* subdirectory. To be able to distribute the Helm chart, you'll need to package it into a TAR file.

## *Release versioning*

Kustomize solely focuses on generating the desired state in a cluster through YAML manifests. You can track changes over time by Git commit hashes or tags to indicate a version.

Helm's rigid project structure requires the definition of a chart version inside of the *Chart.yaml* file. Every time you make a change, you'd bump up the version number, often represented by semantic versioning.

Helm and Kustomize are Kubernetes tools used to automate the process of deploying objects into a cluster. Carefully analyze your technical requirements, business objectives, and the skillset of team members before deciding for either of the tools. You may even determine that you should use both tools to manage application stacks in Kubernetes. Most Kubernetes GitOps tools, e.g. Argo CD or Flux, support Helm and Kustomize.

## **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.

Additional tools emerged for more convenient manifest management. Kustomize is fully integrated with the `kubectl` tool chain. It helps with the generation, composition, and customization of manifests.

## Exam Essentials

*Assume that the Helm and Kustomize executable is preinstalled*

Unfortunately, the **exam FAQ** does not mention any details about the Helm and Kustomize executable. It's fair to assume that it will be preinstalled for you and therefore you do not need to memorize installation instructions.

*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 `kubect` `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.

2. Create the directory named *manifests*. Within the directory, create two files: *pod.yaml* and *configmap.yaml*. The *pod.yaml* file should define a Pod named `nginx` with the image `nginx:1.21.1`. The *configmap.yaml* file defines a ConfigMap named `logs-config` with the key-value pair `dir=/etc/logs/traffic.log`. Create both objects with a single, declarative command.

Modify the ConfigMap manifest by changing the value of the key `dir` to `/etc/logs/traffic-log.txt`. Apply the changes. Delete both objects with a single declarative command.

Use Kustomize to set a common namespace `t012` for the resource file *pod.yaml*. The file *pod.yaml* defines the Pod named `nginx` with the image `nginx:1.21.1` without a namespace. Run the Kustomize command that renders the transformed manifest on the console.

# Part III. Workloads and Scheduling

---

The Workloads and Scheduling domain encompasses the foundational Kubernetes primitives used for deploying applications in enterprise environments. It also covers the key concepts and mechanisms that drive how workloads are efficiently admitted, scheduled and distributed across worker nodes, ensuring optimal resource utilization and performance.

The following chapters cover these concepts:

- **Chapter 9** evolves around the Pod, the essential primitive for running workload in Kubernetes. The chapter also explains the namespace primitive, a way to logically group objects.
- **Chapter 10** shows how to centrally define configuration data with ConfigMaps and Secrets and the different ways to consume the configuration data from a Pod.
- **Chapter 11** demonstrates how to manage a set of Pods to run workload using the Deployment and ReplicaSet primitives by example. You'll learn about the built-in deployment strategies supported by the Deployment to roll out changes across its replicas.
- **Chapter 12** explains how to scale the number of Pods controlled by a ReplicaSet manually by providing a discrete number and automatically by defining resource threshold metrics with a Horizontal Pod Autoscaler.

- **Chapter 13** is all about resource management. 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 14** expands on lessons learned in the previous chapter. You will get to know other concepts that influence if and how a Pod can be scheduled on specific nodes.

# Chapter 9. Pods and Namespaces

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

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 we discussed 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

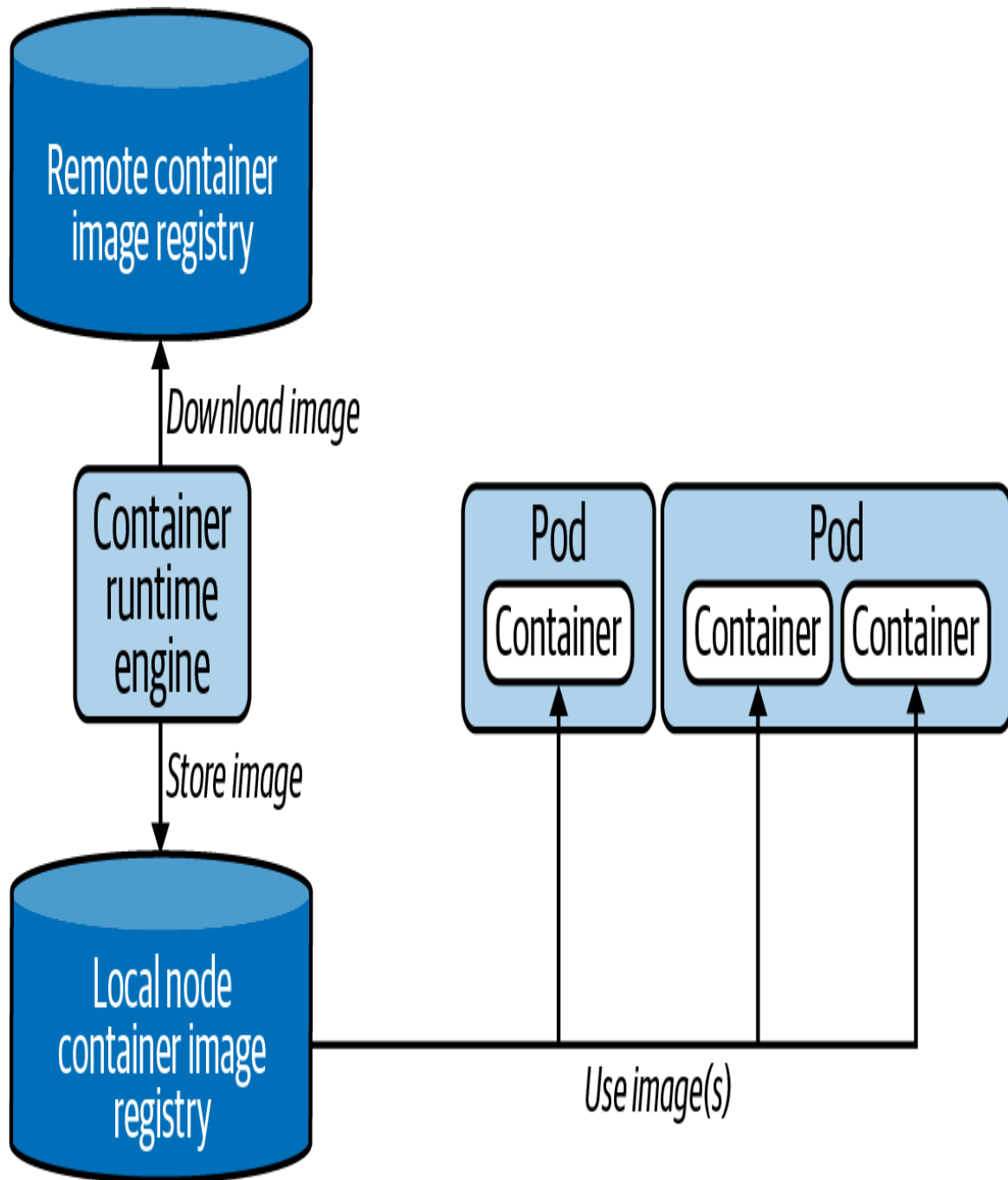
The curriculum doesn't explicitly mention coverage of Pods and namespaces. However, you will definitely need to understand those primitives as they are essential for running workload in Kubernetes.

## Working with Pods

In this chapter, we will look at working with a Pod running only a single container. I'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 9-1** demonstrates the execution flow.



*Figure 9-1. 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, you 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 9-1** lists the most commonly used options.

*Table 9-1. 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>	N/A	Deletes the Pod after command in the container finishes. See <a href="#">“Creating a Temporary Pod”</a> for more information.
<code>--env</code>	<code>PROFILE=dev</code>	The environment variables to set in the container.
<code>--labels</code>	<code>app=frontend</code>	A comma-separated list of labels to apply to the Pod.

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 9-1](#) shows the Pod manifest saved in the file *pod.yaml*:

*Example 9-1. Pod YAML manifest*

```
apiVersion: v1
kind: Pod
metadata:
```

```

name: hazelcast                                ❶
labels:                                         ❷
  app: hazelcast
  env: prod
spec:
  containers:
  - name: hazelcast
    image: hazelcast/hazelcast:5.1.7          ❸
    env:                                       ❹
    - name: DNS_DOMAIN
      value: cluster
    ports:
    - containerPort: 5701                     ❺

```

- ❶ Assigns the name of `hazelcast` to the Pod.
- ❷ Specifies labels to the Pod.
- ❸ Declares the container image to be executed in the container of the Pod.
- ❹ Injects one or many environment variables to the container.
- ❺ 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"](#):

```

$ 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

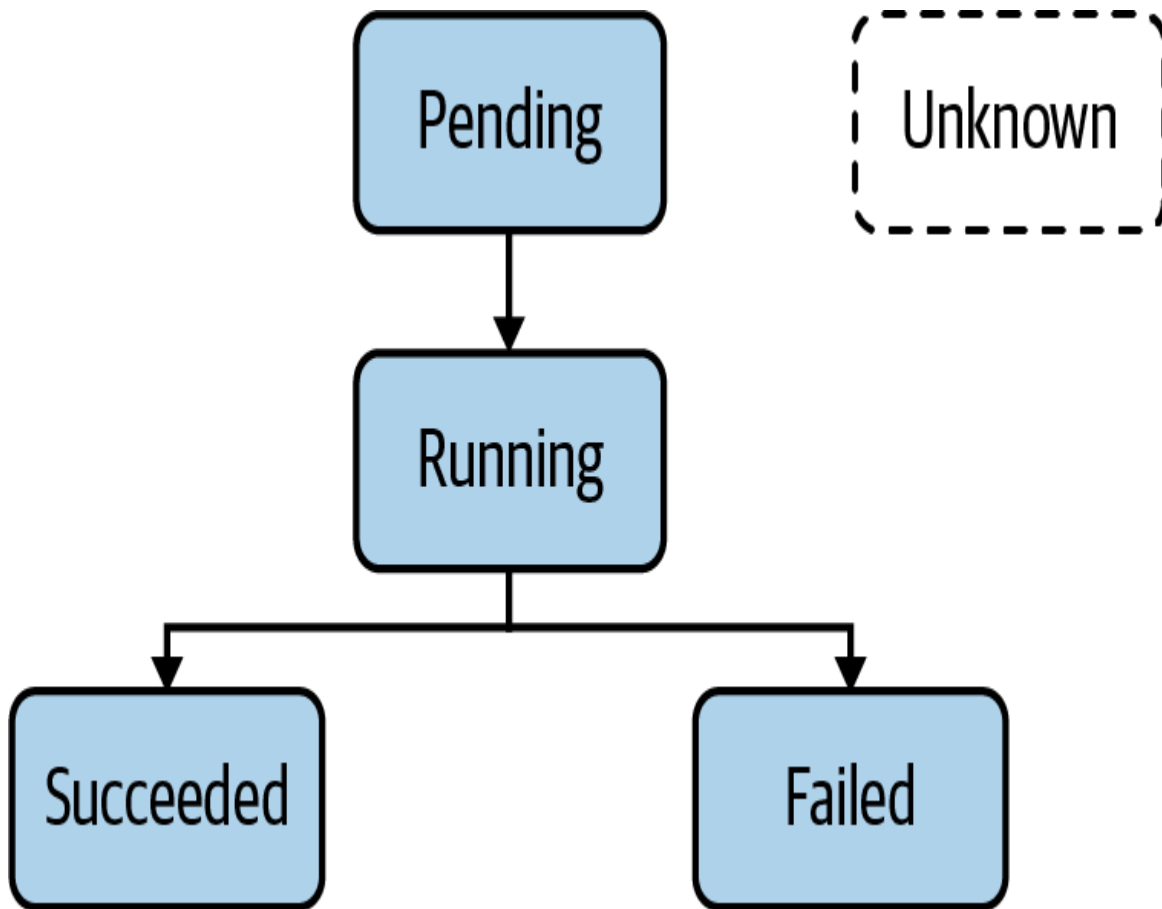
Real-world Kubernetes clusters can run hundreds of Pods at the same time. If you know the name of the Pod of interest, it's often easier to query by name. You would still see only a single Pod:

```
$ 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 9-2**.



*Figure 9-2. 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 9-2** describes all Pod life cycle phases.

*Table 9-2. 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](#).

## Container-Level Restarts

Every Pod provides container-level restart capabilities. If a container fails, the kubelet cluster component will restart it based on its configured [restart policy](#).

The restart policy of a Pod can be set using the attribute `spec.restartPolicy`. Possible values for this attribute include `Always`, `OnFailure`, and `Never`, as shown in [Table 9-3](#). The default value is `Always` if the attribute has not been set explicitly.

*Table 9-3. Container-level restart options*

Option	Description
Always	Automatically restarts the container after any termination.
OnFailure	Only restarts the container if it exits with an error (non-zero exit status).
Never	Does not automatically restart the terminated container.

**Example 9-2** shows the usage of the attribute in a Pod manifest.

*Example 9-2. Setting a Pod's restart policy*

```
apiVersion: v1
kind: Pod
metadata:
  name: hazelcast
spec:
  containers:
  - name: hazelcast
    image: hazelcast/hazelcast:5.1.7
    restartPolicy: Never
```

The attribute applies to a Pod's **application containers** and **init containers**. **Sidecar containers** ignore the attribute; they will automatically apply the value `Always`.

## 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 17](#) 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 9-3**. The code snippet describes a Pod that runs a Java-based application using the Spring Boot framework.

#### *Example 9-3. 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 9-4](#) that the command has been turned into an `args` attribute. Kubernetes specifies each argument on a single line.

#### *Example 9-4. 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 9-5](#) shows a different approach.

#### *Example 9-5. 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 delete 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 9-6** shows the corresponding representation as a YAML manifest.

### *Example 9-6. 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: code-red
```

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 `j43`.

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_USERNAME=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` in the namespace `j43` 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`.

# Chapter 10. ConfigMaps and Secrets

---

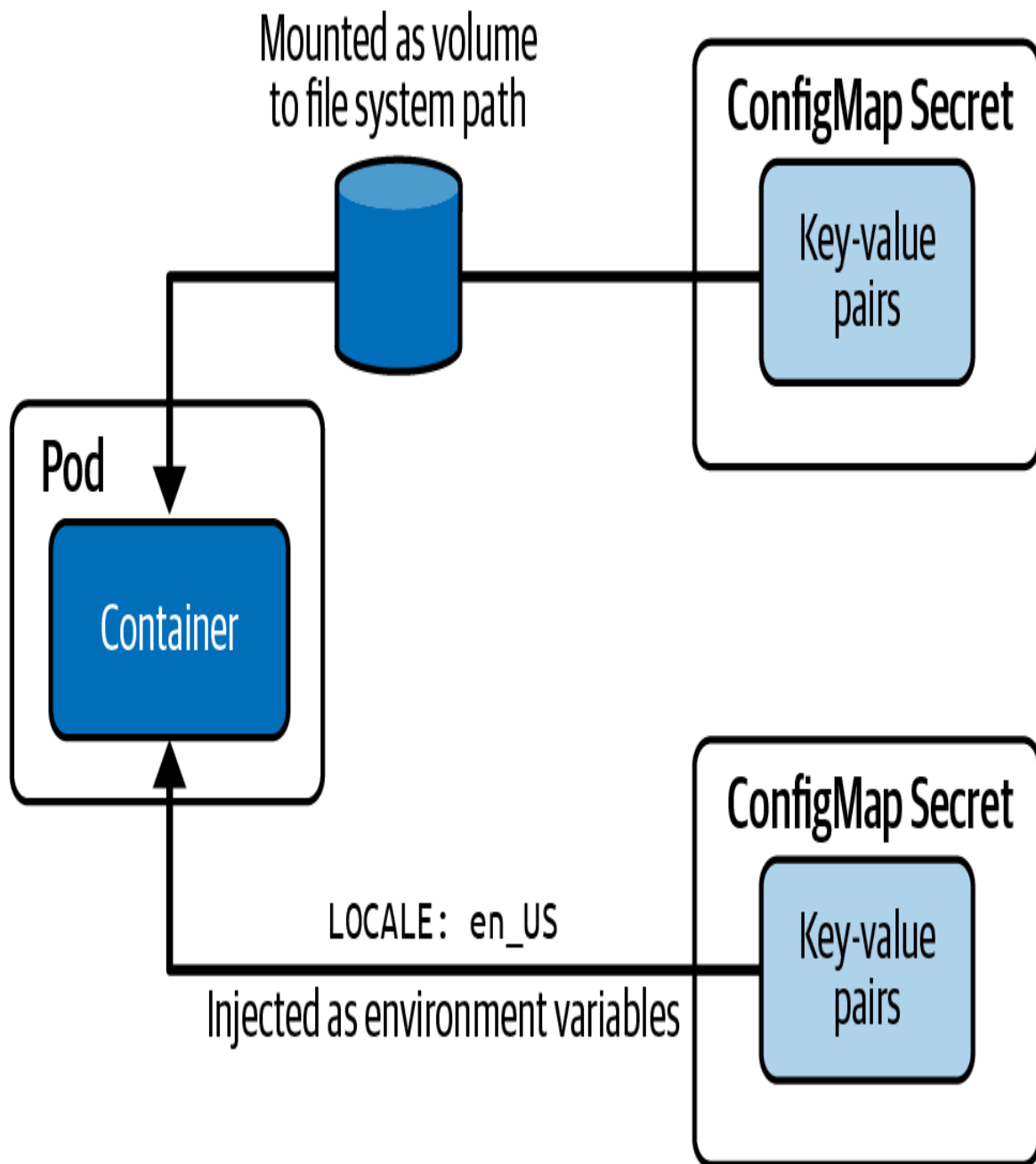
## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 10th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

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 10-1** illustrates the options.



*Figure 10-1. 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 15](#) 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:

- Use ConfigMaps and Secrets to configure applications

## **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 10-1](#).

*Table 10-1. Source options for data parsed by a ConfigMap*

Option	Example	Description
<code>--from-literal</code>	<code>--from-literal=locale=en_US</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=app-config.json</code>	A file with arbitrary contents
<code>--from-file</code>	<code>--from-file=config-dir</code>	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 10-1**. 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 10-1. 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 10-2** shows the use of `spec.containers[].envFrom[].configMapRef` to reference the ConfigMap by name.

*Example 10-2. 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 10-3**.

*Example 10-3. 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 10-4** 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 10-4. ConfigMap YAML manifest defining structured data*

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  db.json: |-
    {
      "db": {
        "host": "mysql-service",
        "user": "backend"
      }
    }
```

- ❶ 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 10-5** demonstrates the YAML definition.

#### *Example 10-5. 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
```

❶

- ❶ 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 -l /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 a 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 10-2** lists the different types. Kubernetes assigns the value in the Internal Type column to the `type` attribute in the live object. “**Specialized Secret types**” discusses other Secret types and their use cases.



*Table 10-2. 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/dockerconfig</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 10-3](#).

*Table 10-3. 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</code> <code>~/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=password=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 10-6](#) that the value `s3cre!` has been turned into `czNjc mUh`, the Base64-encoded equivalent.

*Example 10-6. A Secret with Base64-encoded values*

```
apiVersion: v1
kind: Secret
```

```
metadata:
  name: db-creds
  type: Opaque
  data:
    pwd: czNjcmUh
```

- ❶ The value `Opaque` for the type has been assigned to represent generic sensitive data.
- ❷ 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 is discussed 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 10-7](#).

### *Example 10-7. A Secret with plain-text values*

---

```
apiVersion: v1
kind: Secret
metadata:
  name: db-creds
type: Opaque
stringData: ❶
  pwd: s3cre! ❷
```

- 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 10-8](#), which you can retrieve with the command `kubectl get secret db-creds -o yaml`.

### *Example 10-8. 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. The next section discusses those specialized Secret types.

## 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 10-9** illustrates a YAML manifest for a Secret with type `kubernetes.io/basic-auth`.

### *Example 10-9. 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
```

- ❶ Uses the `stringData` attribute to allow for assigning plain-text values.
- ❷ 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 10-10** injects the Secret

named `secret-basic-auth` as environment variables into the container named `backend`.

### *Example 10-10. 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 10-**

**11** turns the key `username` into `USER` and the key `password` into `PWD`.

### *Example 10-11. 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 they 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 10-12](#).

### *Example 10-12. 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
  volumes:
    - name: ssh-volume
      secret:
        secretName: secret-ssh-auth
```

❶

❶

❷



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 -l /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 approaches*

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 approaches*

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/ch10/configmap* of the checked-out GitHub repository [bmuschko/cka-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=password`.

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.

# Chapter 11. Deployments and ReplicaSets

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 11th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

A major strength of Kubernetes lies in its ability to scale applications seamlessly and manage replication with ease. To enable these capabilities, Kubernetes provides two primitives: Deployments and ReplicaSets.

In this chapter, you’ll learn how to create a Deployment, which leverages a ReplicaSet behind the scenes to manage and scale a group of identical Pods. Deployments also make it easy to roll out updates to your application and, if needed, roll back to a previous version—all with minimal downtime and maximum control.

## COVERAGE OF CURRICULUM OBJECTIVES

This chapter addresses the following curriculum objectives:

- Understand application deployments and how to perform rolling update and rollbacks
- Understand the primitives used to create robust, self-healing, application deployments

## Working with Deployments

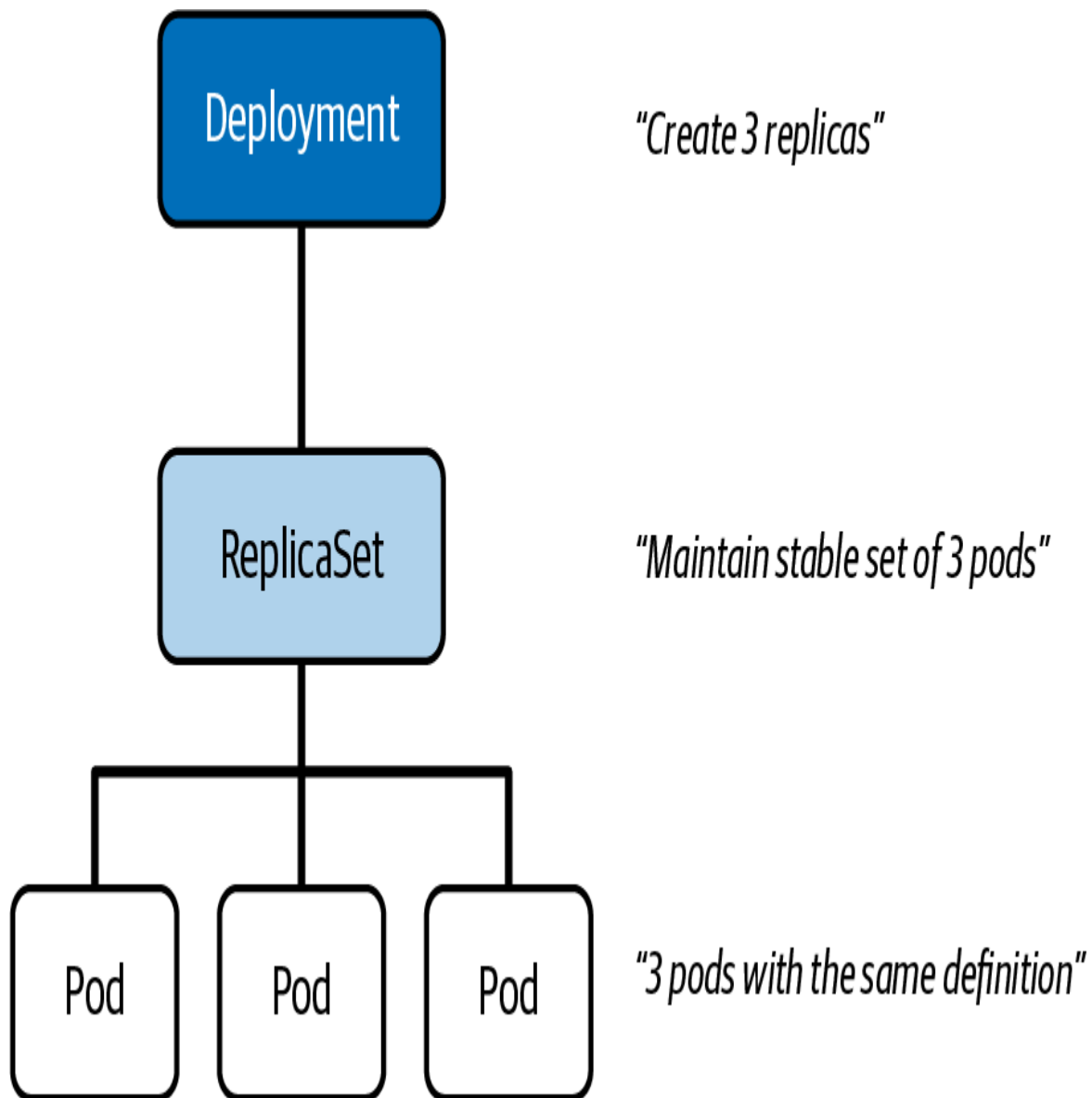
The primitive for running an application in a container is the Pod. Using a single instance of a Pod to operate an application has its flaws—it represents a single point of failure because all traffic targeting the application is funneled to this Pod. This behavior is specifically problematic when the 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 look at 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 11-1** illustrates the relationship between a *Deployment*, a *ReplicaSet*, and its controlled replicas.



*Figure 11-1. 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 11-1** 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 11-1. 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` needs to match, as shown in **Figure 11-2**.

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.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-cache
  labels:
    deploy: app-cache
spec:
  replicas: 4
  selector:
    matchLabels:
      app: memcached
  template:
    metadata:
      labels:
        app: memcached
    spec:
      containers:
        - name: memcached
          image: memcached:1.6.10
```

*"Maps the Deployment to the  
Pod template used for replicas"*

*Figure 11-2. Deployment label selection*

## Listing Deployments, ReplicaSets, 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 11-1](#).

*Table 11-1. Runtime replica information when listing deployments*

Column Title	Description
READY	Lists the number of replicas available to end users in the format of <code>&lt;ready&gt;/&lt;desired&gt;</code> . 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 names for ReplicaSet and its Pods start with `app-cache-`. The hash following the prefix is autogenerated and appended to the name upon creation:

```
$ kubectl get replicaset,pods
```

NAME	DESIRED	CURRENT
replicaset.apps/app-cache-596bc5586d	4	4

NAME	READY	STATUS	RESTARTS
app-cache-596bc5586d-84dkv	1/1	Running	0
app-cache-596bc5586d-8bzfs	1/1	Running	0
app-cache-596bc5586d-rc257	1/1	Running	0
app-cache-596bc5586d-tvm4d	1/1	Running	0

## 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`.

## Replica Replacement

The ReplicaSet in Kubernetes ensures that a specified number of replicas are running at all times. If a Pod fails or is deleted, Kubernetes automatically creates a replacement Pod to maintain the desired replica count. This behavior is one of Kubernetes' key *self-healing capabilities*.

You can easily observe this in action by manually deleting one of the Pods managed by the ReplicaSet. Let's assume the ReplicaSet and Pod objects shown here:

```
$ kubectl get replicaset,pods
```

NAME	DESIRED	CURRENT
replicaset.apps/app-cache-596bc5586d	4	4

NAME	READY	STATUS	RESTARTS
pod/app-cache-596bc5586d-84dkv	1/1	Running	0
pod/app-cache-596bc5586d-8bzfs	1/1	Running	0
pod/app-cache-596bc5586d-rc257	1/1	Running	0
pod/app-cache-596bc5586d-tvm4d	1/1	Running	0

You can delete any of the Pods controlled by the ReplicaSet, i.e. the Pod named `app-cache-596bc5586d-rc257`:

```
$ kubectl delete pod app-cache-596bc5586d-rc257
pod "app-cache-596bc5586d-rc257" deleted
```

Shortly after deleting the Pod manually, a new one will be created by the ReplicaSet. You can easily identify the newly-created Pod by

its “AGE” value. In the following output, the Pod is only 5 seconds old:

```
$ kubectl get replicaset,pods
```

NAME	DESIRED	CURRENT
replicaset.apps/app-cache-596bc5586d	4	4
4		

NAME	READY	STATUS	RESTARTS
pod/app-cache-596bc5586d-84dkv	1/1	Running	0
6h47m			
pod/app-cache-596bc5586d-8bzfs	1/1	Running	0
6h47m			
pod/app-cache-596bc5586d-lwflz	1/1	Running	0
5s			
pod/app-cache-596bc5586d-tvm4d	1/1	Running	0
6h47m			

Similar to the Deployment, other workload primitives like the **StatefulSet** and the **DaemonSet** manage a ReplicaSet to control a set of replicas.

For the StatefulSet, the replica replacement behavior is the same as for the Deployment. If a Pod fails as part of a DaemonSet, the control plane node ensures that the Pod replacement is run on the same node it was scheduled on before. The primitives StatefulSet and DaemonSet are out-of-scope for the exam, however, later chapters may mention them again.

## Deleting a Deployment

A Deployment takes full charge of the creation and deletion of the objects it controls: ReplicaSets and Pods. 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,replicasets,pods
```

NAME	READY	UP-TO-DATE	AVAILABLE
deployment.apps/app-cache	4/4	4	4
6h47m			

NAME	READY	AGE	DESIRED	CURRENT
replicaset.apps/app-cache-596bc5586d	4	6h47m	4	4

NAME	READY	STATUS	RESTARTS
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			

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,replicasets,pods
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, Kubernetes 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 see 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.

### Applying changes declaratively

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
```

### Editing the object from the default editor

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
```

### Updating the container image imperatively

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 this 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
```

## Replacing an existing object

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
```

## Updating fields of an object using a JSON patch

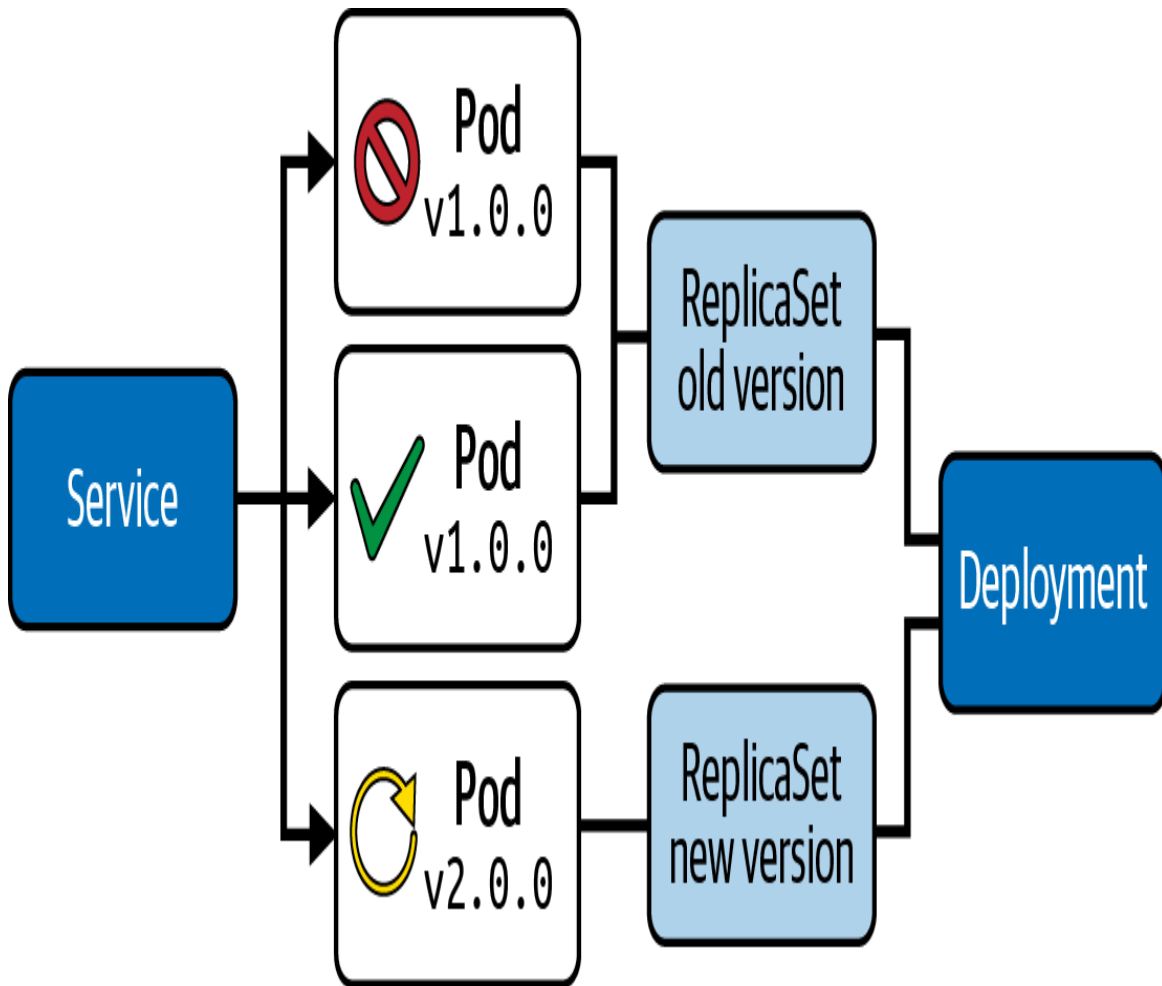
The command `kubectl patch` requires you to provide the merges as a patch to update a Deployment. The following command shows the operation in action. Here, you are sending the changes to be made in the form of a JSON structure:

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

## Rolling Out a New Revision

The Deployment primitive employs *rolling update* 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 11-3 shows a snapshot in time during the rollout process.



*Figure 11-3. The rolling update 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. The runtime behavior of the deployment strategy can be further customized. Refer to the [documentation](#) to see the configuration options.

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 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.

### NOTE

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 deployment strategy by providing a different value to the attribute `spec.strategy.type`; however, consider the trade-offs. The value `Recreate` kills all Pods first, then creates new Pods with the latest revision, causing potential downtime for consumers.

## 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>
```



## 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.

ReplicaSets 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.

## 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 and configure a Deployment.

### *Understand how the ReplicaSet supports replication*

Learn how to scale to multiple replicas. One of the superior features of a ReplicaSet 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.

### *Differentiate the built-in deployment strategies*

Learn how to configure the built-in strategies in the Deployment primitive and their options for fine-tuning the runtime behavior. 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, but will not be covered by the exam.

## Sample Exercises

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

1. A team member wrote a Deployment manifest but has trouble with creating the object from it. Help with finding the issue.

Navigate to the directory *app-a/ch11/misconfigured-deployment* of the checked-out GitHub repository [bmuschko/cka-study-guide](#).

Run a `kubectl` command to create the Deployment object defined in the file *fix-me-deployment.yaml*. Inspect the error message. Fix the Deployment manifest so that the object can be created.

2. 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.

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`.

# Chapter 12. Scaling Workloads

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 12th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

There are several reasons why scaling a workload becomes necessary, particularly to maintain optimal performance under increasing demand. For example, an application may experience a surge in users as it gains popularity, or it may need to process larger volumes of data over time.

In Kubernetes, scaling a workload can be achieved in two primary ways: by increasing the resources allocated to each Pod (vertical scaling), or by adjusting the number of Pods running concurrently (horizontal scaling). Horizontal scaling is especially effective for handling fluctuating workloads, ensuring the application remains responsive and resilient under varying levels of demand.

In this chapter, you’ll learn how to manually scale the number of replicas as a reaction to increased application load. Furthermore, we’ll explore the API primitive HorizontalPodAutoscaler, which allows you to automatically scale the managed set of Pods based on

resource thresholds such as CPU and memory. We won't get into vertical scaling as the exam won't cover it.

### **COVERAGE OF CURRICULUM OBJECTIVES**

This chapter addresses the following curriculum objective:

- Configure workload autoscaling

## **Manual Scaling of Workload**

Manual scaling of workloads requires specifying a fixed number of Pods to run. This number should be informed by real-world usage metrics gathered from production environments or estimated through load testing during development.

However, because application traffic can fluctuate unexpectedly—rising sharply or tapering off—you'll need to continuously monitor and adjust the number of Pods to match demand. Failing to do so can lead to over-provisioning, which wastes resources, or under-provisioning, which can degrade performance and user experience.

## **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-svrwx	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.

## Manually Scaling a StatefulSet

Another API primitive that can be scaled manually is the `StatefulSet`. `StatefulSets` are meant for managing stateful applications by a set of Pods (e.g., databases). Similar to a `Deployment`, the `StatefulSet` defines a Pod template; however, each of its replicas guarantees a unique and persistent identity. Similar to a `Deployment`, a `StatefulSet` uses a `ReplicaSet` to manage the replicas.

We are not going to look at StatefulSets in more detail, but you can read more about them in the [documentation](#). The reason I am discussing the StatefulSet primitive here is that it can be manually scaled in a similar fashion as the Deployment.

Let's say we'd deal with the YAML definition for a StatefulSet that runs and exposes a Redis database, as illustrated in [Example 12-1](#).

#### *Example 12-1. A YAML manifest for a StatefulSet*

---

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  selector:
    matchLabels:
      app: redis
  replicas: 1
  serviceName: "redis"
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
        - name: redis
          image: redis:6.2.5
          command: ["redis-server", "--appendonly", "yes"]
          ports:
            - containerPort: 6379
              name: web
          volumeMounts:
            - name: redis-vol
              mountPath: /data
      volumeClaimTemplates:
        - metadata:
            name: redis-vol
          spec:
            accessModes: ["ReadWriteOnce"]
            resources:
              requests:
                storage: 1Gi
```

After its creation, listing the StatefulSet shows the number of replicas in the "READY" column. As you can see in the following output, the number of replicas was set to 1:

```
$ kubectl apply -f redis.yaml
service/redis created
statefulset.apps/redis created
$ kubectl get statefulset redis
NAME      READY   AGE
redis     1/1     2m10s
$ kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
redis-0     1/1     Running   0           2m
```

The `scale` command we explored in the context of a Deployment works here as well. In the following command, we scale the number of replicas from one to three:

```
$ kubectl scale statefulset redis --replicas=3
statefulset.apps/redis scaled
$ kubectl get statefulset redis
NAME      READY   AGE
redis     3/3     3m43s
$ kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
redis-0     1/1     Running   0           101m
redis-1     1/1     Running   0           97m
redis-2     1/1     Running   0           97m
```

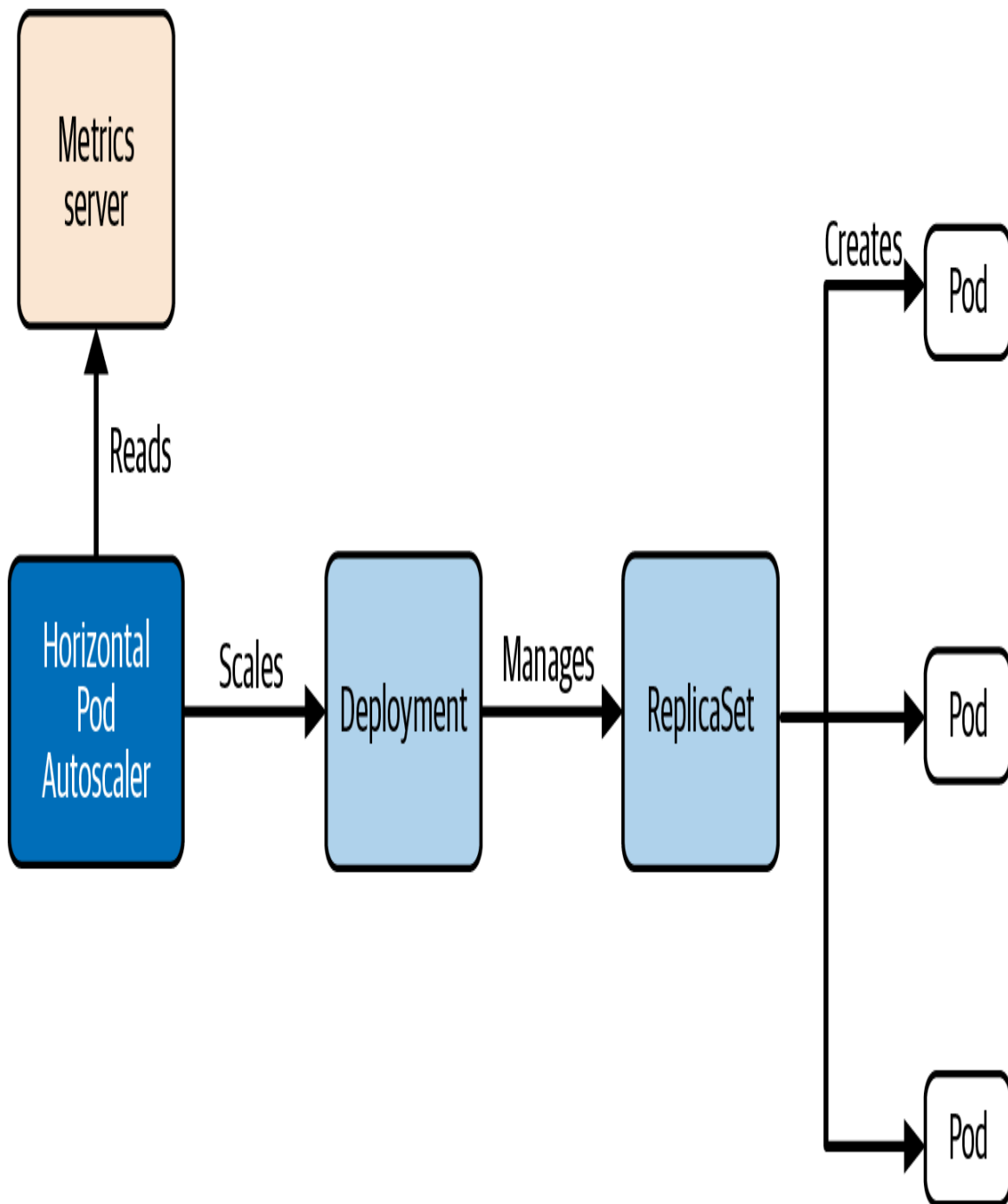
It's important to mention that the process for scaling down a StatefulSet requires all replicas to be in a healthy state. Any long-term, unresolved issues in Pods controlled by a StatefulSet can lead to a situation that can result in the application becoming unavailable to end users.

## Autoscaling of Workload

Another way to scale a Deployment is with the help of a HorizontalPodAutoscaler (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.

[Figure 12-1](#) shows an overview architecture diagram involving an HPA.





*Figure 12-1. Autoscaling a Deployment*

## Prerequisites for Autoscaling

An HPA only works on scalable resources like the Deployment, ReplicaSet, and the StatefulSet. It cannot scale standalone Pods.

For an HPA to work, a couple of prerequisites need to be fulfilled:

- The **metrics server** needs to be installed. Without it, the HPA cannot retrieve the necessary metrics to evaluate Pod performance. Collecting metrics may take a couple of minutes initially after installing the component.
- The **container resource requests** need to be defined. For CPU-based autoscaling, your Pods must define `spec.containers[].resources.requests.cpu`. For memory-based autoscaling, `spec.containers[].resources.requests.memory` is required. These values provide the baseline for calculating utilization.
- The cluster must have sufficient CPU and memory resources to schedule new Pods.

## Creating Horizontal Pod Autoscalers

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.

**Figure 12-2** 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.

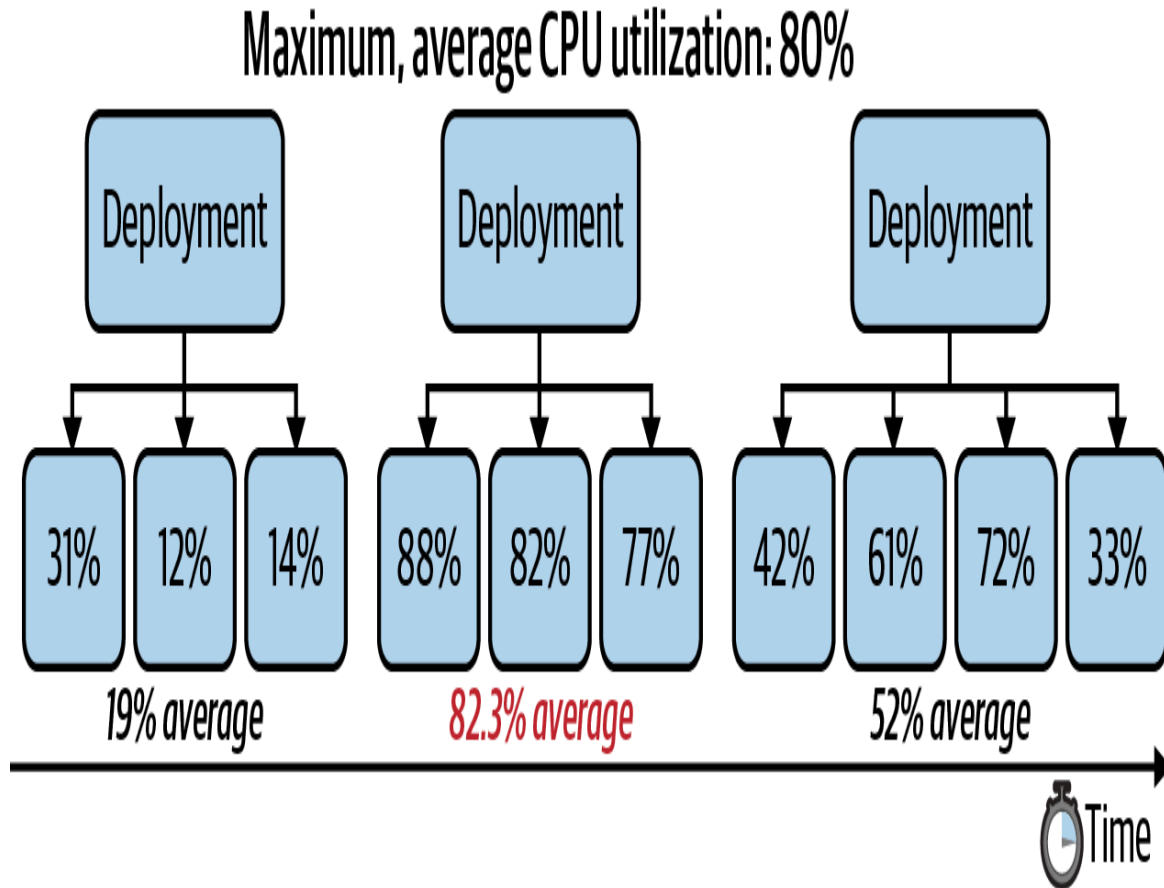


Figure 12-2. Autoscaling a Deployment horizontally

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 12-2](#).

### *Example 12-2. 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 12-3** 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**”.

***Example 12-3. 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: (current / target )

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
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 12-4](#), we are inspecting CPU and memory utilization to determine if the replicas of a Deployment need to be scaled up or down.

*Example 12-4. 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 12-5](#).

*Example 12-5. 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
app-cache Deployment/app-cache  1994752/500Mi, 0%/80%
3         5        \
3         2m14s

```

## Summary

Scaling workloads 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

*Understand the difference between manual and automatic scaling*



Kubernetes workloads can be scaled either manually or automatically. In real-world scenarios, autoscaling is the preferred approach, as it allows the number of replicas to adjust dynamically based on actual resource consumption relative to defined thresholds. This ensures optimal performance and resource efficiency without the need for constant manual oversight.

### *Identify if autoscaling prerequisites are fulfilled*

For the HPA to function correctly, it's essential to have the metrics server installed and to define resource requests for your containers. Without these in place, the HPA won't have the necessary data to make informed scaling decisions, rendering it ineffective.

### *Know how to instantiate and configure a Horizontal Pod Autoscaler*

An HPA YAML manifest is built around three essential components: the scaling target (the named resource the HPA will manage e.g. a Deployment or a ReplicaSet), the minimum and maximum number of replicas the HPA can scale between, and the scaling rules - the conditions that trigger scaling, typically based on resource usage thresholds like CPU or memory utilization.

## **Sample Exercises**

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

1. Define a Deployment named `hello-world` with 3 replicas in a YAML file named `hello-world-deployment.yaml`. The Pod template of the Deployment should use container image `bmuschko/nodejs-hello-world:1.0.0`. Create the object from the YAML file.

Scale the number of replicas to 8 by modifying the YAML file. Apply the changes and verify that all 8 replicas are running.

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?

# Chapter 13. Resource Requirements, Limits, and Quotas

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 13th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

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.

## NOTE

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. This chapter discusses 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:

- Configure Pod admission and scheduling (limits, node affinity, etc.)

## 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 13-1** describes the available options including an example value.

*Table 13-1. Options for resource requests*

YAML attribute	Description	Example value
<code>spec.containers[].resources.requests.cpu</code>	CPU resource type	500m (five hundred millicpu)
<code>spec.containers[].resources.requests.memory</code>	Memory resource type	64Mi (2 <sup>26</sup> bytes)
<code>spec.containers[].resources.requests.hugepages-&lt;size&gt;</code>	Huge page resource type	hugepages-2Mi: 60Mi
<code>spec.containers[].resources.requests.ephemeral-storage</code>	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 13-1](#) 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 13-1. 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 13-2** describes the available options including an example value.

*Table 13-2. Options for resource limits*

YAML attribute	Description	Example value
<code>spec.containers[].resource s.limits.cpu</code>	CPU resource type	500m (500 millicpu)
<code>spec.containers[].resource s.limits.memory</code>	Memory resource type	64Mi (2 <sup>26</sup> bytes)
<code>spec.containers[].resource s.limits.hugepages-&lt;size&gt;</code>	Huge page resource type	hugepages-2Mi: 60Mi
<code>spec.containers[].resource s.limits.ephemeral-storage</code>	Ephemeral storage resource type	4Gi

**Example 13-2** shows the definition of limits in action. Here, the container named `business-app` cannot use more than 256Mi of memory. The container named `ambassador` defines a limit of 64Mi of memory.

### *Example 13-2. 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 13-3** combines resource requests and limits in a single YAML manifest.

### *Example 13-3. 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 to maximize 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 13-4**.

#### Example 13-4. Defining hard resource limits with a ResourceQuota

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: awesome-quota
  namespace: team-awesome
spec:
  hard:
    pods: 2
    requests.cpu: "1"
    requests.memory: 1024Mi
    limits.cpu: "4"
    limits.memory: 4096Mi
```

- ❶ Limit the number of Pods to 2.
- ❷ Define the minimum resources requested across all Pods in a non-terminal state to 1 CPU and 1024Mi of RAM.
- ❸ 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 13-5](#).

### *Example 13-5. 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 13-6](#).

### *Example 13-6. 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 13-7](#) shows a YAML manifest for a LimitRange that uses some of the constraint attributes.

*Example 13-7. 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.



- ③ The default CPU resource limit value assigned to a container if not provided.
- ④ 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 13-8** shows a Pod definition without resource requirements.

#### *Example 13-8. 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 13-9](#).

### *Example 13-9. 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 limitranges`.

## 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/ch13/limitrange* of the checked-out GitHub repository *[bmuschko/cka-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?

# Chapter 14. Pod Scheduling

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 14th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

The *scheduler* is the cluster component responsible for deciding which node to select for running a Pod. In this chapter, you will learn about the general scheduling algorithm and the Kubernetes concepts that let you express soft and hard requirements for influencing the scheduling decisions.

## COVERAGE OF CURRICULUM OBJECTIVES

This chapter addresses the following curriculum objective:

- Configure Pod admission and scheduling (limits, node affinity, etc.)

## Pod Scheduling Algorithm

Initially a Pod created by an end user does not have a node assigned. That’s the job of the scheduler cluster component. The scheduler runs in a scheduling loop watching for unscheduled Pods.

It will then evaluate available nodes to choose the most suitable one.

Finding a fitting node follows a two-step approach: filtering and scoring. The filtering step determines the list of nodes feasible for running a Pod (e.g. by checking the available hardware capacity). The scoring step ranks the remaining nodes to select the most suitable to run the workload. Scheduling decisions include resource requirements, affinity and anti-affinity specifications, and many more. You can find a visualization of the Pod scheduling algorithm in [Figure 14-1](#).

## **Pod**

Object creation has  
been initiated



## **Filtering**

Filter out nodes that  
are not suitable



## **Scoring**

Score the remaining  
nodes



## **Selecting Node**

Select the highest  
scoring node

*Figure 14-1. Pod scheduling algorithm*

Pods and their container(s) can define requirements that help with determining the node. If the requirements cannot be met by any of the nodes, then the Pod stays unscheduled until the scheduler checks again. Otherwise, the scheduler picks the node with the highest score and places the Pod on it.

## Setting Up a Multi-Node Development Cluster

The effects of scheduling requirements is best explained by demonstrating them on a multi-node cluster. For the remainder of this chapter, I am going to use a cluster with one control plane node and three worker nodes, as shown here:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
multi-node	Ready	control-plane	2m33s	v1.32.2
multi-node-m02	Ready	<none>	2m22s	v1.32.2
multi-node-m03	Ready	<none>	2m15s	v1.32.2
multi-node-m04	Ready	<none>	2m9s	v1.32.2

Setting up a multi-node cluster with a Kubernetes environment is pretty easy to achieve. Most development cluster implementations like **minikube** and **kind** offer an option to initiate more than a single node. The following command creates a four-node cluster with the name prefix `multi-node` using minikube.

```
$ minikube start --kubernetes-version=v1.32.2 --nodes=4 -p multi-node
```

Refer to the documentation of the Kubernetes development cluster of your choice for more information on applicable configuration options.

## Determining the Node a Pod Runs on

The prerequisite for being able to schedule a Pod is that the scheduler is functional. The scheduler process runs in a Pod on the control plane node. You can use the following command to find the scheduler Pod. Make sure that the Pod has the *Running* status.

```
$ kubectl get pods -n kube-system
```

NAME	READY	STATUS	RESTARTS
AGE			
kube-scheduler-multi-node	1/1	Running	0
5m10s			

You can find out which node a Pod runs with the `kubectl get` or `kubectl describe` command. The following commands show variations of their usage:

```
$ kubectl get pod nginx -o=wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP
NODE		...			
nginx	1/1	Running	0	3m49s	10.244.2.2
multi-node-m03		...			

```
$ kubectl get pod nginx -o yaml | grep nodeName:
  nodeName: multi-node-m03
```

```
$ kubectl describe pod nginx
```

Name: nginx  
Namespace: default  
Priority: 0  
Service Account: default  
Node: multi-node-m03/192.168.49.4  
...

## Pod Scheduling Options

The scheduler does a reasonably good job of assigning a Pod to a feasible node. Under certain conditions, you may want to restrict

which node a Pod can run on, or define a preferred selection criteria. This is usually expressed by using label selection.

Kubernetes offers a variety of Pod scheduling options, each of which can be combined with one another. In this chapter, I am going to discuss the following concepts, however, there are more you can select from:

- **Node selector:** A hard requirement to determine which node the Pod needs to run on.
- **Node affinity and anti-affinity:** A more flexible requirement for defining hard or soft requirements for node selection.
- **Taints and tolerations:** A way to safe-guard specific nodes from scheduling Pods on them based on conditions and requirements.
- **Pod topology spread constraints:** Defines how to spread Pods across different topologies, i.e. regions and zones.

## Working with Node Selectors

The node selector defines a hard requirement for scheduling a Pod on specific nodes. To use the node selector, label one or many nodes with a specific label key-value pair. When defining a Pod in a YAML manifest, select the label from the Pod via the attribute `spec.nodeSelector`.

A typical example for using the node selector is to ensure that Pods end up running on a node with specific hardware. Input/output-intensive applications that require fast disk access, e.g. as supported by SSD volumes, could make good use of this concept.

**Figure 14-2** illustrates a cluster with three nodes. Node 1 has the label `disk=ssd` and node 2 has the label `disk=hdd`. Pod 1 can only be scheduled on node 1 as it defines the matching node

selector, and therefore on none of the other nodes. Pod 2 cannot be scheduled on any of the nodes, as the node selector doesn't match any of the nodes' labels. Pod 1 and Pod 2 cannot be scheduled on node 3.



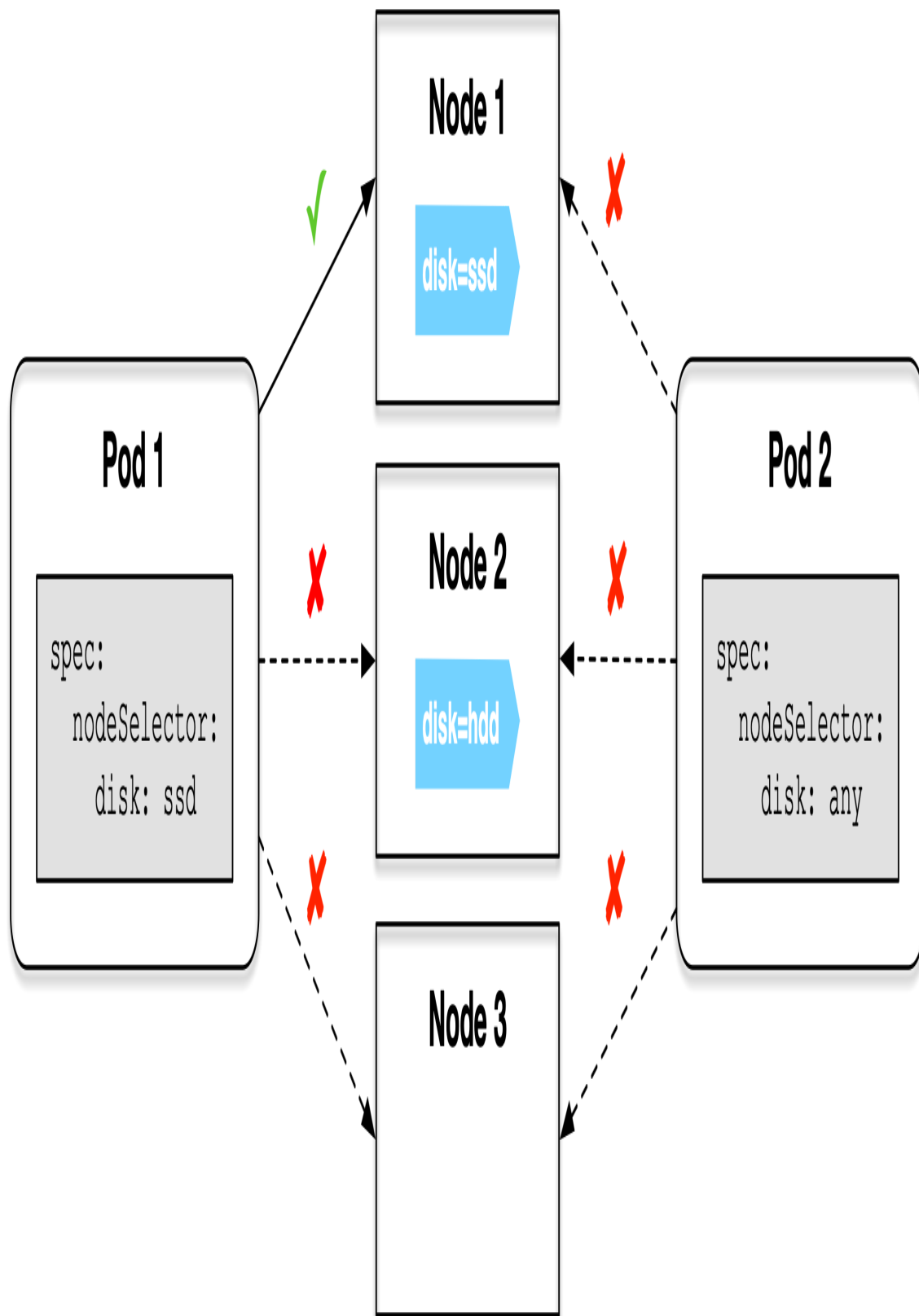


Figure 14-2. Node selector scenarios

## Labeling a node

Following the described use case of running applications only on nodes that provide SSD volume access, we'll first need to assign a specific label key-value pair. The following command uses the imperative `kubectl label` command to assign the label

`disk=ssd` to the node named `multi-node-m03`:

```
$ kubectl label node multi-node-m03 disk=ssd
node/multi-node-m03 labeled
```

You can find the assigned labels for all nodes when listing them with the `--show-labels` option:

```
$ kubectl get nodes --show-labels
NAME                STATUS    ROLES    AGE   VERSION
LABELS
multi-node          Ready    control-plane   14m   v1.32.2
...
multi-node-m02      Ready    <none>         14m   v1.32.2
...
multi-node-m03      Ready    <none>         14m   v1.32.2
...
multi-node-m04      Ready    <none>         14m   v1.32.2
...,disk=ssd,...
```

Next up, you will need to select this label from the Pod that should be scheduled on the node `multi-node-m03`.

## Assigning a node selector to a Pod

The only addition to the typical structure is the definition of the `spec.nodeSelector` attribute, as show in [Example 14-1](#).

### *Example 14-1. Assigning a node selector*

---

```
apiVersion: v1
kind: Pod
metadata:
  name: app
spec:
  nodeSelector:           ❶
    disk: ssd             ❷
  containers:
  - name: nginx
    image: nginx:1.27.1
```

- ❶ Defines a node selector for the Pod
- ❷ The label key-value pair used to determine a suitable node

#### **NOTE**

The node selector is not limited to a single label key-value pair. Selecting a map of labels is completely valid.

The Pod with this definition can only be scheduled on the node that provides the matching label. You can verify that the Pod runs on the expected node with the same command described in “[Determining the Node a Pod Runs on](#)”.

## **Working with Node Affinity and Anti-Affinity**

In Kubernetes, the `spec.nodeSelector` field is used to define strict scheduling constraints—it allows you to specify hard requirements that a node must satisfy for a Pod to be scheduled on it. While simple and straightforward, the node selector is limited to exact key-value label matches and does not support advanced logic.

For more flexible and expressive scheduling rules, you should use node affinity, defined under `spec.affinity.nodeAffinity` in

the Pod specification. Node affinity allows you to match nodes using label selector expressions, enabling more complex criteria such as **logical operators** (In, NotIn, Exists, etc.) and prioritized preferences.

Coming back to the previous use case, you may want to run Pods on nodes that support SSD-based volumes *or* volumes with slower storage devices if the primary preference cannot be fulfilled.

**Figure 14-3** shows the node affinity concept in action. In this scenario, Pod 1 can be scheduled on node 1 or node 2 based on the defined set-based label selection.

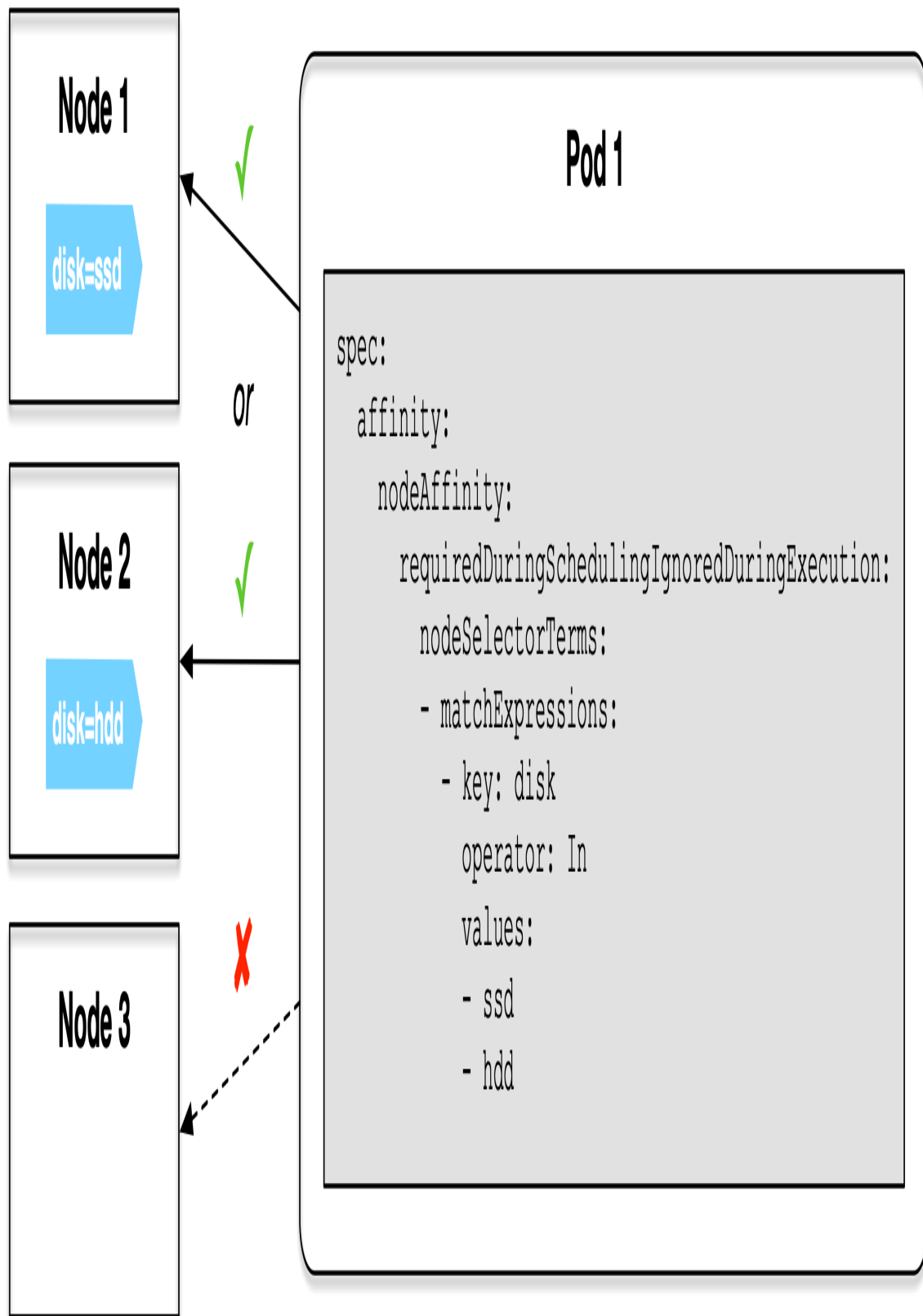


Figure 14-3. Node affinity scenarios

Node anti-affinity is useful in situation where you want to prevent Pods to be scheduled on specific nodes. This is particularly helpful in high-availability situations where you want to spread Pods across different zones or regions.

## Assigning a node affinity to a Pod

In short, node affinity effectively replaces the node selector for most use cases, offering greater precision and control over workload placement. **Example 14-2** shows the same Pod definition we saw earlier, but in this case it allows for placing the Pod on a node where the assigned label key-value pair is `disk=ssd` or `disk=hdd`.

### *Example 14-2. Assigning node affinity*

---

```
apiVersion: v1
kind: Pod
metadata:
  name: app
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: disk
                operator: In
                values:
                  - ssd
                  - hdd
  containers:
    - name: nginx
      image: nginx:1.27.1
```

- ❶ Defines a node affinity for the Pod
- ❷ The node affinity type that will be adhered to when scheduling the Pod

- ③ The expression for finding matching nodes
- ④ A set-based label requirement

## Node affinity types

Beyond supporting set-based expressions, node affinity also introduces specific types that control when the rules are applied. One commonly used type is:

`requiredDuringSchedulingIgnoredDuringExecution`. This setting means that the affinity rules are strictly enforced only at the time of scheduling—when the Pod is initially assigned to a node. Once the Pod is running, any changes to the node affinity rules are ignored and will not trigger rescheduling.

This isn't the only available node affinity type. [Table 14-1](#) shows the list. Types starting with `requiredDuringScheduling` express a hard requirement, and types starting with `preferredDuringScheduling` express a soft requirement, a preference which the schedule doesn't have to adhere to in case no fitting node can be determined.

*Table 14-1. Node affinity types*

Type	Description
<code>requiredDuringSchedulingIgnoredDuringExecution</code>	Rules that must be met for a Pod to be scheduled onto a node.
<code>preferredDuringSchedulingIgnoredDuringExecution</code>	Rules that specify preferences that the scheduler will try to enforce but will not guarantee.

At the time of writing, none of the node affinity types support modifying an already scheduled Pod, as indicated by the `IgnoredDuringExecution` suffix. The Kubernetes team may decide to change that in a future release.

## Node affinity operators

In the previous example, you saw one of the node affinity operators in action, the `In` operator. The `In` operator is an operator that defines a requirement to find a label value as part of a given set of strings. You can select other operators to define node affinity requirements, as listed in [Table 14-2](#).



*Table 14-2. Node affinity operators*

Operator	Behavior
<code>In</code>	A node has an assigned label value in the given set of strings.
<code>NotIn</code>	Only those nodes are selected that do not have an assigned label value in the given set of strings.
<code>Exists</code>	A node has a label key assigned to it that matches the given string.
<code>DoesNotExist</code>	A node does not have a label key assigned to it that matches the given string.
<code>Gt</code>	A node does not have a label key assigned to it that matches the given string.
<code>Lt</code>	A node does not have a label key assigned to it that matches the given string.

There are two operators, `NotIn` and `DoesNotExist`, that negate the selective effects of their counterparts `In` and `Exists`. Those operators are used to define a node anti-affinity behavior.

## Assigning a node anti-affinity to a Pod

Node anti-affinity rules are typically used to prevent certain Pods from being scheduled on the same nodes, based on labels. An essential tool for defining anti-affinity behavior is the operator.

**Example 14-3** uses the `NotIn` operator to repel Pods from a set of nodes with the given label values.

### *Example 14-3. Assigning node anti-affinity*

---

```
apiVersion: v1
kind: Pod
metadata:
  name: app
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: disk
                operator: NotIn
                values:
                  - ssd
                  - ebs
  containers:
    - name: nginx
      image: nginx:1.27.1
```

**❶** Defines a node anti-affinity by way of using a negating condition

In essence, node anti-affinity does not require you to learn a new API or new attributes when defining a Pod. Its usage primary boils down to the operator you select to define the node affinity rule.

## **Working with Taints and Tolerations**

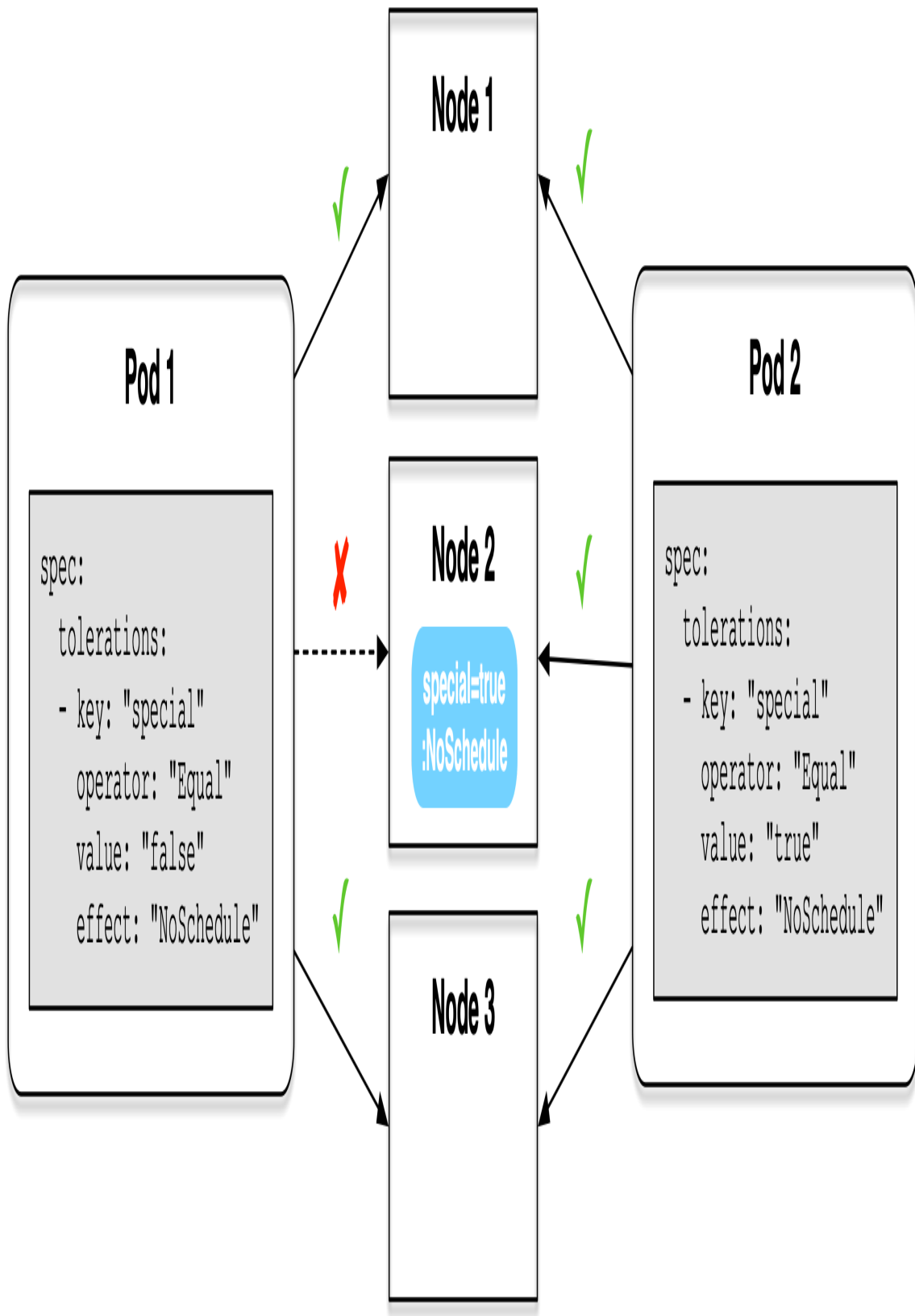
Similar to node anti-affinity, taints and tolerations represent another way in Kubernetes to influence where Pods can be scheduled, but they serve different purposes and work in fundamentally different ways. Node anti-affinity is meant to be used to spread or separate workload across nodes, whereas taints and tolerations are used for node isolation and workload protection.

The main purpose of taints and tolerations is to prevent Pods from being scheduled on a node unless they explicitly *tolerate* that node's taint. You'd add the taint to a node to say, "don't schedule anything here unless it tolerates the taint." Then in the Pod, you'd add a toleration if you want it to become schedulable on the tainted node.

A typical use case for using taints and tolerations is the need to ensure that Pods are not scheduled on control plane nodes. Control plane nodes use the taint `node-`

`role.kubernetes.io/control-plane:NoSchedule` to prevent Pods from being scheduled on them unless they provide a corresponding toleration.

In **Figure 14-4**, you can see a scenario that demonstrates the use of taints and tolerations. Node 1 and 3 tolerates Pod 1 and Pod 2. Node 2 only accepts Pod 1.



*Figure 14-4. Taints and tolerations scenarios*

Let's demonstrate the process of adding a taint to a node and a toleration to a Pod by example.

## Tainting a node

A taint on a node marks it as unsuitable for certain Pods unless those Pods explicitly state they can tolerate it. A taint consists of three parts—key, value and effect—formatted as `key=value:effect`. The key and value portions represent a simple, free-form key-value pair, similar to a label assignment. The effect describes the runtime treatment of the taint by the scheduler.

Use the imperative `kubectl taint` command to add a taint to a node. The following command adds the taint `special=true:NoSchedule` to the node named `multi-node-m02`:

```
$ kubectl taint node multi-node-m02 special=true:NoSchedule
node/multi-node-m02 tainted
```

The scheduler now considers this taint during Pod placement. To view the assigned taints of a node, run the `kubectl get` or `kubectl describe` command. The following command renders the YAML representation of the node `multi-node-m02` and then finds the relevant information in the output by combining it with the Linux `grep` command:

```
$ kubectl get node multi-node-m02 -o yaml | grep -C 3
taints:
...
spec:
  taints:
```

```
- effect: NoSchedule
  key: special
  value: "true"
```

## Taint effects

The taint shown in the previous example used the `NoSchedule` effect, which is a hard block for any Pod that doesn't come with the relevant toleration. You can choose from other taint effects, explained in [Table 14-3](#).

*Table 14-3. Taint effects*

Effect	Description
<code>NoSchedule</code>	Unless a Pod has matching toleration, it won't be scheduled on the node.
<code>PreferNoSchedule</code>	Try not to place a Pod that does not tolerate the taint on the node, but it is not required.
<code>NoExecute</code>	Evict Pod from node if already running on it. No future scheduling on the node.

In short, you can think of the available taint effects and their runtime enforcement in the following way. The effect `NoSchedule` is a hard block. `PreferNoSchedule` offers a soft suggestion. Lastly, `NoExecute` not only blocks Pods without the corresponding toleration but also evicts already running Pods that can't fulfill the requirement.

## Assigning a toleration to a Pod

To enable a Pod to run on a tainted node, you must add a toleration to the Pod specification that precisely matches the key, value, and effect of the node's taint. This tells the scheduler that the Pod is allowed to tolerate the taint and may be placed on the node despite the restriction.

**Example 14-4** shows a matching toleration assigned to a Pod for the taint `special=true:NoSchedule`.

### *Example 14-4. Assigning a toleration*

---

```
apiVersion: v1
kind: Pod
metadata:
  name: app
spec:
  tolerations:
    - key: "special"
      operator: "Equal"
      value: "true"
      effect: "NoSchedule"
  containers:
    - name: nginx
      image: nginx:1.27.1
```

- ❶ The attribute that lets you define one or more tolerations
- ❷ The key of the toleration that needs to match the taint
- ❸ A toleration “matches” a taint if the keys are the same and the effects are the same
- ❹ The value of the toleration that needs to match the taint
- ❺ The runtime effect

The usage of the effect in a toleration is required in most practical cases. If your toleration is missing the effect, it won't match any taint. Leaving off the effect is considered optional only when using

the operator `Exists` is used and you want to tolerate all taints with a specific key (regardless of value). Nevertheless, even in this scenario, specifying the effect is recommended.

## Working with Pod Topology Spread Constraints

Pod topology spread constraints control how Pods are distributed across your cluster to improve availability, resilience, and resource utilization. They define rules for how Pods of a certain group (typically in the same Deployment) should be spread across topology domains (like zones, nodes, or racks). You can define the Pod topology spread constraint in the Pod API with the attribute `spec.topologySpreadConstraints`.

### Assigning a topology spread constraint to a Pod

**Example 14-5** shows an example for a Deployment YAML definition with 6 replicas of an app and ensures that they are evenly spread across availability zones.

*Example 14-5. Assigning a topology spread constraint to a Pod*

---

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  replicas: 6
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
```



```

topologySpreadConstraints:
- maxSkew: 1
  topologyKey: topology.kubernetes.io/zone
  whenUnsatisfiable: DoNotSchedule
  labelSelector:
    matchLabels:
      app: web
containers:
- name: nginx
  image: nginx:1.27.1

```

①  
②  
③  
④  
④  
④

- ① The difference in Pod count between zones must not exceed 1.
- ② Uses the node label, in this case the reserved label for defining the Pods spread across availability zones
- ③ What to do if a spread cannot be satisfied
- ④ Applies the spread rule only for Pods with the given label(s)

## Topology spread constraint effects at runtime

There are a couple of important facts to mention on how the concept behaves at runtime. Pod topology spread constraints only affect newly-scheduled Pods, they do not rebalance existing Pods. You can define multiple constraints, e.g. a spread by zones and node hostnames. Be aware that the use of the concept can lead to unschedulable Pods if constraints are too strict and resources are limited.

## Summary

Kubernetes Pod scheduling is the process of assigning Pods to available nodes in a cluster based on various criteria. By default, the Kubernetes scheduler considers resource requirements, node availability, and constraints like taints, tolerations, and node selectors.

Developers can influence scheduling using node affinity, which expresses preferences for particular node labels, and Pod affinity/anti-affinity, which controls whether Pods are co-located or separated. Taints and tolerations allow nodes to repel certain pods unless those Pods explicitly tolerate the taint. Resource requests and limits guide the scheduler in ensuring Pods are only placed on nodes with sufficient CPU and memory. Topology spread constraints help evenly distribute Pods across failure domains like zones or nodes.

## Exam Essentials

*Be able to identify the node a Pod runs on*

As an end user to Kubernetes, you can easily find out which node a Pod runs on. Become familiar with the relevant `kubectl` commands that give you access to this information. During the exam, you may be asked which Pods run on which nodes of the cluster.

*Understand the ins and outs of Pod scheduling options*

You'll need to be familiar with a wide range of Pod scheduling concepts. Tasks in the exam may ask you select the most suitable concept to define soft or hard requirements for specific scheduling scenarios. Most likely, the Pod scheduling concept is spelled out explicitly, and you will need to be able to apply the syntax appropriately.

## Sample Exercises

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

1. Inspect the existing nodes and their assigned labels. Pick one available node and label it with the key-value pair

`color=green`. Pick a second node and label it with the key-value pair `color=red`.

Define a Pod with the image `nginx:1.27.1` in the YAML manifest file *pod.yaml*. Use the `nodeSelector` assignment to schedule the Pod on the node with the label `color=green`. Create the Pod and ensure that the correct node has been used to run the Pod.

Change the Pod definition to schedule it on nodes with the label `color=green` or `color=red`. Verify that the Pod runs on the correct node.

2. Define a Pod with the image `nginx:1.27.1` in the YAML manifest file *pod.yaml*. Create the Pod and check which node the Pod is running on.

Add a taint to the node. Set it to `exclusive: yes`. The effect should be `NoExecute`.

Modify the live Pod object by adding the following toleration: It should be equal to the taint key-value pair and have the effect `NoExecute`.

Observe the running behavior of the Pod. If your cluster has more than a single node where do you expect the Pod to run?

Remove the taint from the node. Do you expect the Pod to still run on the node?

# Part IV. Storage

---

The Storage domain in the curriculum focuses on managing persistent data in Kubernetes clusters. This critical area ensures applications can reliably store and access data beyond the lifecycle of individual Pods.

The following chapters cover these concepts:

- **Chapter 15** explains the general mechanism of making a directory available to containers in a Pod for the purpose of persisting and sharing data.
- **Chapter 16** describes the provisioning of a storage resource that provides durable data storage for applications running in Pods that survives Pod restarts, rescheduling, and failures.

# Chapter 15. Volumes

---

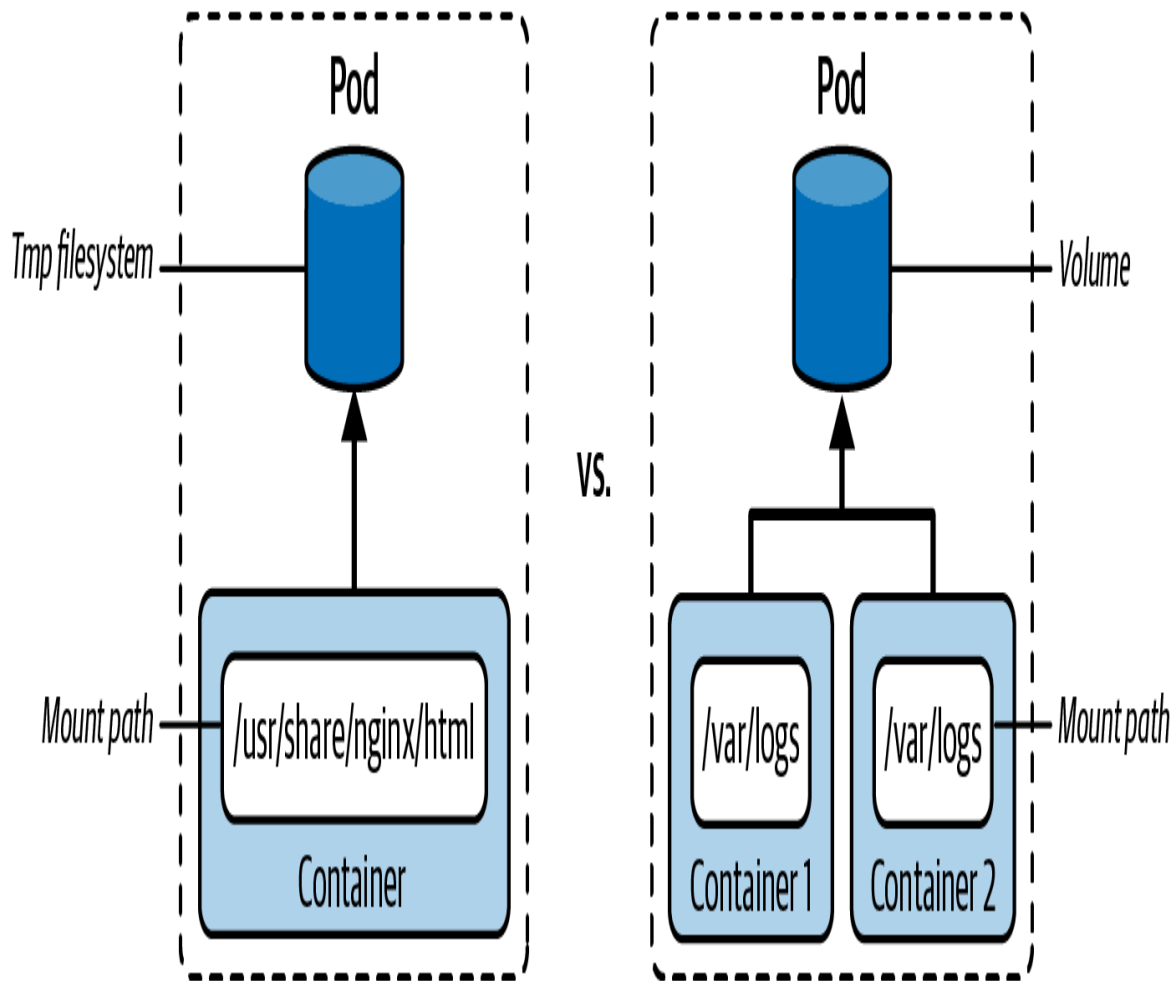
## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 15th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

When container images are instantiated as containers, the container needs context—context to CPU, memory, and I/O resources. Pods provide the network and the filesystem context for the containers within. The network is provided as the Pod’s virtual IP address, and the filesystem is mounted to the hosting node’s filesystem.

Applications running in the container can interact with the filesystem as part of the Pod context. A container’s temporary filesystem is isolated from any other container or Pod and is not persisted beyond a Pod restart.

Essentially, a volume is a directory that’s shareable between multiple containers of a Pod. **Figure 15-1** illustrates the differences between the temporary filesystem of a container and the use of a volume.



*Figure 15-1. A container using the temporary filesystem versus a volume*

In this chapter, you will learn about the different volume types and the process for defining and mounting a volume in a container.

### **COVERAGE OF CURRICULUM OBJECTIVES**

The curriculum doesn't explicitly mention coverage of volume basics. However, you will definitely need to understand this concept for persistent volumes described in the next chapter.

## Purpose of Volumes

In a nutshell, the Kubernetes volume concept aims to fulfill the goals described below.

### *Data Persistence*

Applications running in a container can use the temporary filesystem to read and write files. In the 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 anymore. The container effectively starts with a clean slate again. Volumes can provide persistent storage that survives container restarts, ensuring important data isn't lost.

### *Data Sharing*

There are many use cases for wanting to mount a volume in a container. One of the most prominent use cases are **multi-container Pods** that use a volume to exchange data between a main application container and a sidecar, enabling them to share files and communicate through the filesystem.

### *Decoupling storage from containers*

Volumes abstract storage details from the application, allowing you to change storage backends without modifying container images.

## Volume Types

Every volume needs to define a type. The type determines the medium that backs the volume and its runtime behavior. The **Kubernetes documentation** offers a long list of volume types. Some of the types—for example, `azureDisk`, `awsElasticBlockStore`, or `gcePersistentDisk`—are available only when running the

Kubernetes cluster in a specific cloud provider. Many of those cloud provider-specific volume types have already been deprecated.

**Table 15-1** shows a reduced list of volume types that I deem to be most relevant to the exam.

*Table 15-1. Volume types relevant to exam*

Type	Description
<code>emptyDir</code>	Empty directory in Pod with read/write access. Persisted for only the lifespan of a Pod. A good choice for cache implementations or data exchange between containers of a Pod.
<code>hostPath</code>	File or directory from the host node's filesystem.
<code>configMap</code> , <code>secret</code>	Provides a way to inject configuration data. For practical examples, see <a href="#">Chapter 10</a> .
<code>nfs</code>	An existing Network File System (NFS) share. Preserves data after Pod restart.
<code>persistentVolumeClaim</code>	Claims a persistent volume. For more information, see <a href="#">"Creating PersistentVolumeClaims"</a> .

## Creating and Accessing Volumes

Defining a 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, though. 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 by the matching name.

From the YAML manifest stored in the file *pod-with-volume.yaml* and shown in **Example 15-1**, you can see the definition of a volume with type `emptyDir`. The volume has been mounted to the path `/usr/share/nginx/html` inside of the container named `nginx` and to the path `/data` inside of the container named `sidecar`.

### *Example 15-1. A Pod defining and mounting a volume*

---

```
apiVersion: v1
kind: Pod
metadata:
  name: business-app
spec:
  volumes:
  - name: shared-data
    emptyDir: {}
  containers:
  - name: nginx
    image: nginx:1.27.1
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html
  - name: sidecar
    image: busybox:1.37.0
    volumeMounts:
    - name: shared-data
      mountPath: /data
```

- ❶ Specifies a volume of type `emptyDir`. The curly braces mean that we don't want to provide additional configuration, e.g. a size limit.
- ❷ Mounts the volume to containers with different mount paths inside of the container.

Let's create the Pod from the YAML definition above. The Pod runs two containers.

```
$ kubectl apply -f pod-with-volume.yaml
pod/business-app created
$ kubectl get pod business-app
NAME             READY   STATUS    RESTARTS   AGE
business-app     2/2     Running   0           43s
```

The following commands open an interactive shell to the container named `nginx` after the Pod's creation and then navigate to the mount path. You can see that the volume type `emptyDir` initializes the mount path as an empty directory.

```
$ kubectl exec business-app -it -c nginx -- /bin/sh
# cd /usr/share/nginx/html
# pwd
/usr/share/nginx/html
# ls
# touch example.html
# ls
example.html
```

New files and directories can be created as needed without limitations.

## Read-only Volume Mounts

Some data is only meant for consumption, for example configuration data provided through a volume. You can mark a volume mount to be read-only. Kubernetes will prevent any write operation on that volume mount. It's important to understand that other containers may use the same volume in read-write mode.

To make a volume mount read-only, assign the value `true` to the attribute `spec.containers[].volumeMounts[].readOnly`, as shown in **Example 15-2**.

### *Example 15-2. A mount path marked as read-only*

---

```
apiVersion: v1
kind: Pod
metadata:
  name: business-app
spec:
  volumes:
  - name: shared-data
    emptyDir: {}
  containers:
  - name: nginx
    image: nginx:1.27.1
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html
      readOnly: true
```

❶

❶ Prevents write operations for this mount path.

Read-only mounts are not recursively read-only. You can force the behavior though by setting the attribute

`.spec.containers[].volumeMounts[].recursiveReadOnly` to `true`.

## 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. You can select from ephemeral volumes and persistent volumes.

## Exam Essentials

### *Understand the need and use cases for a volume*

Many production-ready application stacks running in a cloud-native environment need to persist data. Read up on common use cases and explore recipes that describe typical scenarios. You can find some examples in the O'Reilly books [Kubernetes Best Practices](#), and [Cloud Native DevOps with Kubernetes](#).

### *Practice defining and consuming 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 and the multitude of ways to consume a volume from a container. Definitely revisit [Chapter 10](#) for a deep dive on how to mount ConfigMaps and Secrets as a volume.

## 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.

# Chapter 16. Persistent Volumes

---

## A NOTE FOR EARLY RELEASE READERS

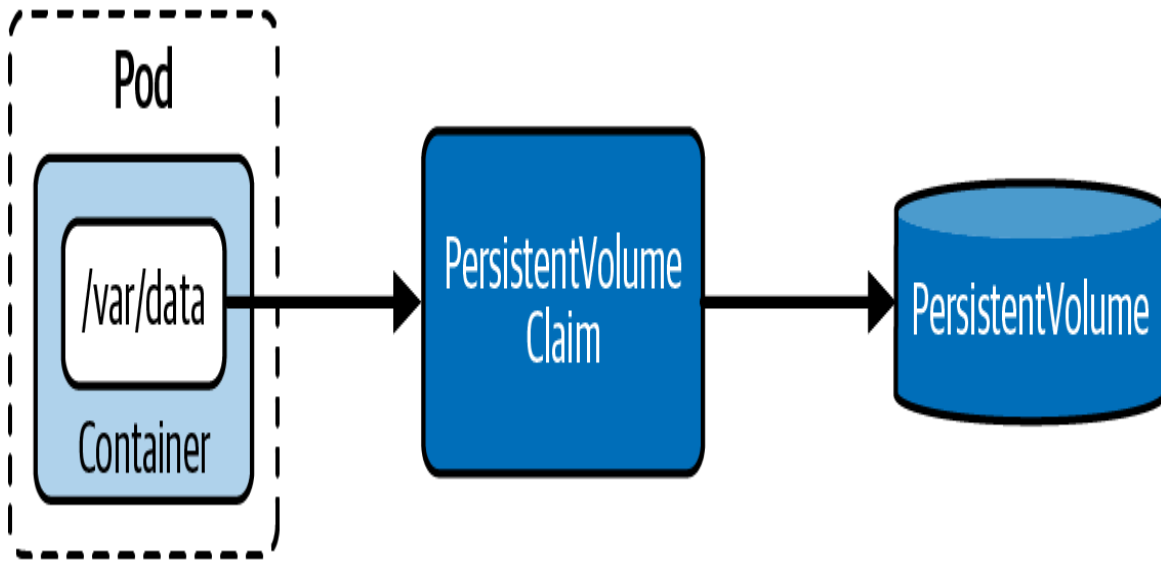
With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 16th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

Persistent volumes are a specific category of the wider concept of volumes with the capability of persisting data beyond a Pod restart. The mechanics for persistent volumes are slightly more complex. The persistent volume is the resource that actually persists the data to an underlying physical storage.

The persistent volume claim represents the connecting resource between a Pod and a persistent volume responsible for requesting the storage.

Finally, the Pod needs to *claim* the persistent volume and mount it to a directory path available to the containers running inside of the Pod.

**Figure 16-1** shows the relationship between the Pod, the persistent volume claim, and the persistent volume.



*Figure 16-1. Claiming a PersistentVolume from a Pod*

## **COVERAGE OF CURRICULUM OBJECTIVES**

This chapter addresses the following curriculum objective:

- Implement storage classes and dynamic volume provisioning
- Configure volume types, access modes and reclaim policies
- Manage persistent volumes and persistent volume claims

## **Working with 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 persistent 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`.

## Volume Types

Kubernetes supports several types of persistent volumes to accommodate different storage backends and use cases. Each type has its own characteristics and is suitable for specific scenarios.

**Table 15-1** lists the most commonly-used persistent volume types that are not deprecated.



*Table 16-1. Persistent volume types*

Type	Description
<code>hostPath</code>	Mounts a file or directory from the host node's filesystem into the Pod. Useful for development and testing but not recommended for production multi-node clusters as it ties the Pod to a specific node.
<code>local</code>	Represents a mounted local storage device such as a disk, partition, or directory. Provides better performance than remote storage but requires node affinity to ensure Pods are scheduled on the correct node.
<code>nfs</code>	Allows multiple Pods to share the same Network File System (NFS) mount. Supports <code>ReadWriteMany</code> access mode and is useful for sharing data between Pods across nodes.
<code>csi</code>	Container Storage Interface driver that provides a standardized way to expose storage systems to containerized workloads. Most modern storage solutions use CSI drivers.
<code>fc</code>	Fibre Channel volume that allows existing FC storage to be attached to Pods. Requires FC hardware and proper configuration on nodes.
<code>iscsi</code>	iSCSI (Internet Small Computer Systems Interface) volume that allows existing iSCSI

Type	Description
	storage to be mounted to Pods. Provides block-level storage over IP networks.

The choice of volume type depends on your infrastructure, performance requirements, and whether you need the storage to be shared across multiple Pods or nodes. For cloud environments, CSI drivers are typically provided by cloud providers (like AWS EBS CSI driver, GCE PD CSI driver, or Azure Disk CSI driver) to integrate with their native storage services.

## 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](#)". 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"](#) for more information on the configuration options available to a `PersistentVolume`.

**Example 16-1** 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 16-1. YAML manifest defining a PersistentVolume*

---

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

- ❶ The storage capacity available to persistent volume.
- ❷ The read-write access modes applicable to the persistent volume.

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 apply -f db-pv.yaml
persistentvolume/db-pv created
$ kubectl get pv db-pv
```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS
db-pv	1Gi	RWO	Retain	Available

```
\
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 16-2](#) shows all available volume modes.

Table 16-2. 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    Retain
Available \
19m    Filesystem
```

## 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 16-3](#) 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 16-3. 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 16-4](#)). 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 16-4. 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
```

## Node affinity

Node affinity allows you to constrain which nodes a PersistentVolume can be accessed from. This is particularly important for local storage types like `local` and `hostPath` volumes, which are physically tied to specific nodes. By defining

node affinity rules, you ensure that Pods using the PersistentVolume are scheduled only on nodes that can actually access the underlying storage.

The node affinity is specified using the `spec.nodeAffinity` field in the PersistentVolume definition. It uses the same syntax as Pod node affinity, with `required` rules that must be satisfied for the volume to be accessible.

**Example 16-2** illustrates an example of a PersistentVolume with node affinity that restricts it to specific nodes:

#### *Example 16-2. Defining node affinity for a PersistentVolume*

---

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  local: ❶
    path: /mnt/data
  nodeAffinity: ❷
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname ❸
              operator: In
              values:
                - node01
                - node02
```

❶ Uses the `local` volume type, which requires node affinity.

❷ Defines node affinity rules for this PersistentVolume.

❸



Restricts the volume to nodes with hostnames `node01` or `node02`.

Common use cases for node affinity with PersistentVolumes include:

- Local volumes: Must specify node affinity to indicate which node contains the storage.
- Zone constraints: Ensuring volumes are accessed only from nodes in specific availability zones.
- Hardware requirements: Restricting volumes to nodes with specific storage hardware (e.g., SSD nodes).

When a PersistentVolumeClaim binds to a PersistentVolume with node affinity, any Pod using that claim will be scheduled according to these constraints. If no suitable node is available, the Pod will remain in `Pending` status.

Important considerations:

- Node affinity is required for `local` volume types.
- The scheduler considers both the Pod's node selectors and the PersistentVolume's node affinity.
- Changes to node labels after binding don't affect existing mounted volumes.
- For high availability, avoid overly restrictive node affinity rules.

## 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 16-3**.

### Example 16-3. Definition of a PersistentVolumeClaim

---

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

- ❶ The access modes we're asking an unbound persistent volume to provide.
- ❷ Uses an empty string assignment to indicate that we want to use static provisioning.
- ❸ The minimum amount of storage an unbound persistent volume needs to have available.

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 apply -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
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>
...
```

## Binding by volume name

When creating a PersistentVolumeClaim, you can optionally specify the exact PersistentVolume you want to bind to using the `spec.volumeName` attribute. This creates a direct binding between the PersistentVolumeClaim and a specific PersistentVolume, bypassing the normal matching algorithm that considers storage size, access modes, and storage class.

This is useful in scenarios where:

- You have pre-provisioned PersistentVolumes with specific characteristics.
- You need to ensure a PersistentVolumeClaim binds to a particular PersistentVolume for compliance or data locality reasons.

- You want to reuse an existing PersistentVolume that contains important data.

**Example 16-4** shows an example of a PersistentVolumeClaim that explicitly binds to a PersistentVolume named `db-pv`:

*Example 16-4. Binding a PersistentVolume to a PersistentVolumeClaim by name*

---

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: specific-pvc
spec:
  volumeName: db-pv
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

- ❶ Explicitly binds this PersistentVolumeClaim to the PersistentVolume named `db-pv`.

When using `volumeName`, ensure that:

- The specified PersistentVolume exists and is in `Available` status
- The PersistentVolume's capacity is greater than or equal to the PersistentVolumeClaim's requested storage.
- The access modes are compatible.
- The storage classes match (or both are unset).

If these conditions aren't met, the PersistentVolumeClaim will remain in `Pending` status.

## 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 16-5](#), is using `spec.volumes[].persistentVolumeClaim` and providing the name of the PersistentVolumeClaim.

### *Example 16-5. A Pod referencing a PersistentVolumeClaim*

---

```
apiVersion: v1
kind: Pod
metadata:
  name: app-consuming-pvc
spec:
  volumes:
  - name: app-storage
    persistentVolumeClaim: ❶
      claimName: db-pvc ❷
  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
```

- ❶ The volume type that selects a persistent volume claim by name.
- ❷ The name of the persistent volume claim object we want to bind to.

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 apply -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 16-6** defines a storage class on Google Compute Engine denoted by the provisioner `kubernetes.io/gce-pd`.

### *Example 16-6. 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 16-7** shows the usage of the attribute `spec.storageClassName` for assigning the storage class named `standard`.

### Example 16-7. Using a storage class in a PersistentVolumeClaim

```

kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: db-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:

```



```
storage: 512Mi
storageClassName: standard ❶
```

- ❶ Uses the storage class by its name to enable dynamic provisioning.

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"** for more information.

## Summary

PersistentVolumes even store data beyond a Pod or cluster/node restart. Those objects are decoupled from the Pod's lifecycle and are therefore represented by a Kubernetes primitive. The PersistentVolumeClaim abstracts the underlying implementation details of a PersistentVolume and acts as an intermediary between the Pod and PersistentVolume. A PersistentVolume can be provisioned statically by creating the object or dynamically with the help of a provisioner assigned to a storage class.

## Exam Essentials

*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.

*Know the differences between static and dynamic provisioning of a PersistentVolume*

A PersistentVolume can be created statically by creating the object from a YAML manifest using the `create` command. Alternatively, you can let Kubernetes provision a PersistentVolume dynamically without your direct involvement. For this to happen, assign a storage class to the PersistentVolumeClaim. The provisioner of the storage class takes care of creating the PersistentVolume object for you.

## Sample Exercises

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

1. 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.

2. Navigate to the directory `app-a/ch16/dynamic-provisioning` of the checked-out GitHub repository [bmuschko/cka-study-guide](#). Inspect the YAML manifest definition in the file `local-path-storage-0.0.31.yaml`. Create the objects from the YAML manifest file.

Create a PersistentVolumeClaim named `db-pvc` in the namespace `persistence`. The access it uses is `ReadWriteOnce`. Request a capacity of 10Mi. Use the storageclass name `local-path`.

Ensure that the status of the PersistentVolumeClaim object shows `Pending`. A PersistentVolume object should not have been provisioned yet.

Mount the PersistentVolumeClaim in a Pod named `app-consuming-pvc` in the namespace `persistence` at the mount path `/mnt/data`. The container should use the image `alpine:3.21.3`.

Wait until the Pod transitions into the "Running" status. Ensure that the PersistentVolume object has been created dynamically.

Open an interactive shell to the container, and create a new file named `test.db` in `/mnt/data`. Exit out of the Pod.

# Part V. Servicing & Networking

---

The domain Servicing & Networking 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 17** 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.
- **Chapter 18** 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 19** discusses the replacement API for the traditional Ingress primitive. The chapter explains the limitations of Ingress while providing an overview of the Gateway API CRDs, as well as migration strategies.
- **Chapter 20** 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 17. Services

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 17th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

In “[Using a Pod’s IP Address for Network Communication](#)”, 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:

- Understand connectivity between Pods
- Use `ClusterIP`, `NodePort`, `LoadBalancer` service types and endpoints
- Understand and use CoreDNS

## 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](#)". Similar to a Deployment, the Service determines the Pods it works on with the help of label selection.

[Figure 17-1](#) 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.



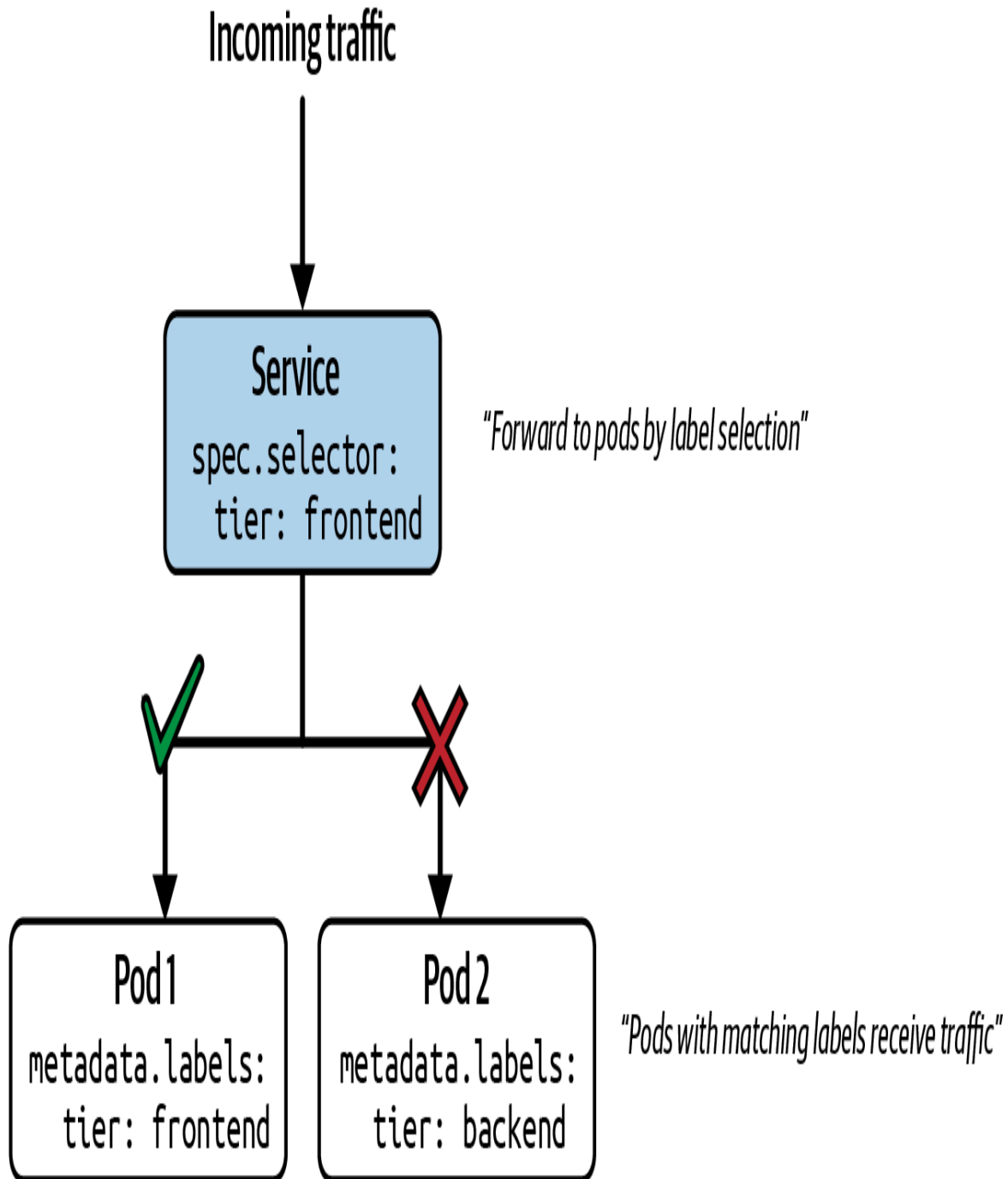


Figure 17-1. Service traffic routing based on label selection

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 17-1** lists the Service types relevant to the exam.

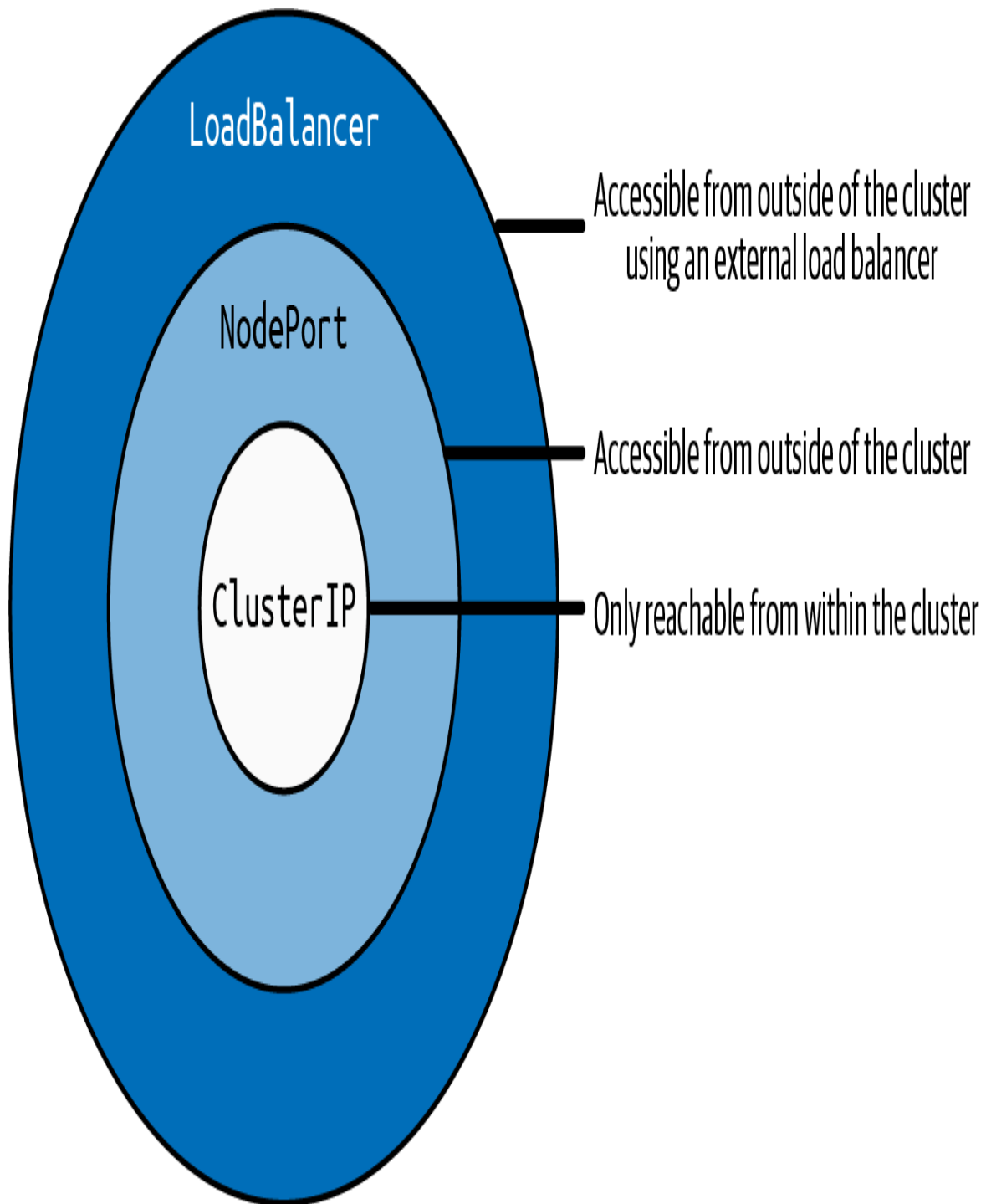
*Table 17-1. Service types*

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.
LoadBalancer	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 17-2](#) shows the relationship between different Service types.



*Figure 17-2. 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.

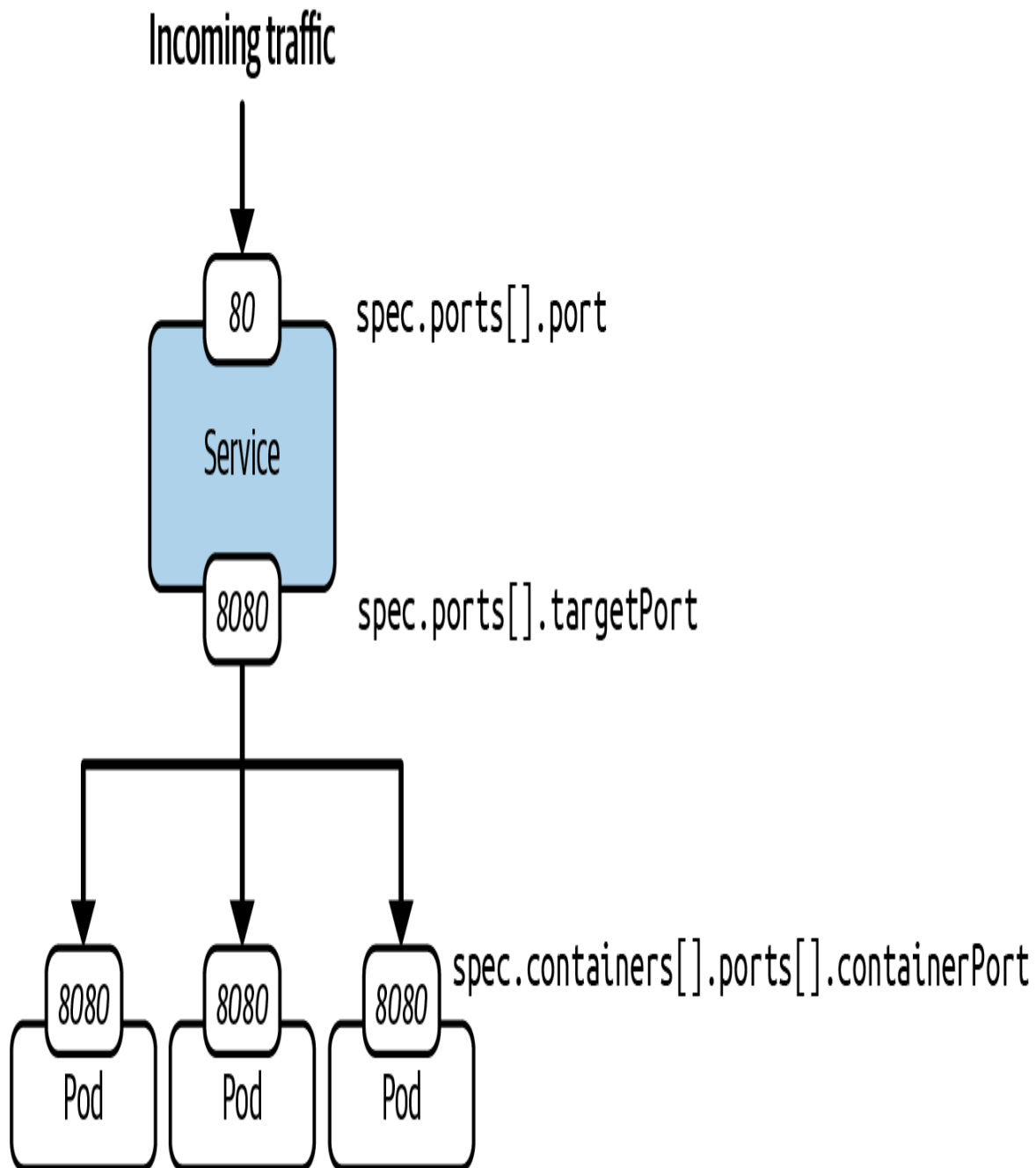
The `LoadBalancer` Service type makes the application available to outside consumers through an external IP address provided by an external load balancer. Network traffic will be distributed across multiple nodes in the cluster. This solution works great for production environments, but keep in mind 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 18](#).

## 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 17-3](#) 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 17-3. 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 17-1** 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 17-1. A Service defined by a YAML manifest*

---

```
apiVersion: v1
kind: Service
metadata:
  name: echoserver
spec:
  selector:
    run: echoserver
  ports:
  - port: 80
    targetPort: 8080
```

❶

❷

- ❶ Selects all Pods with the given label assignment.
- ❷ 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 17-4** illustrates the accessibility of a Service with type `ClusterIP`.

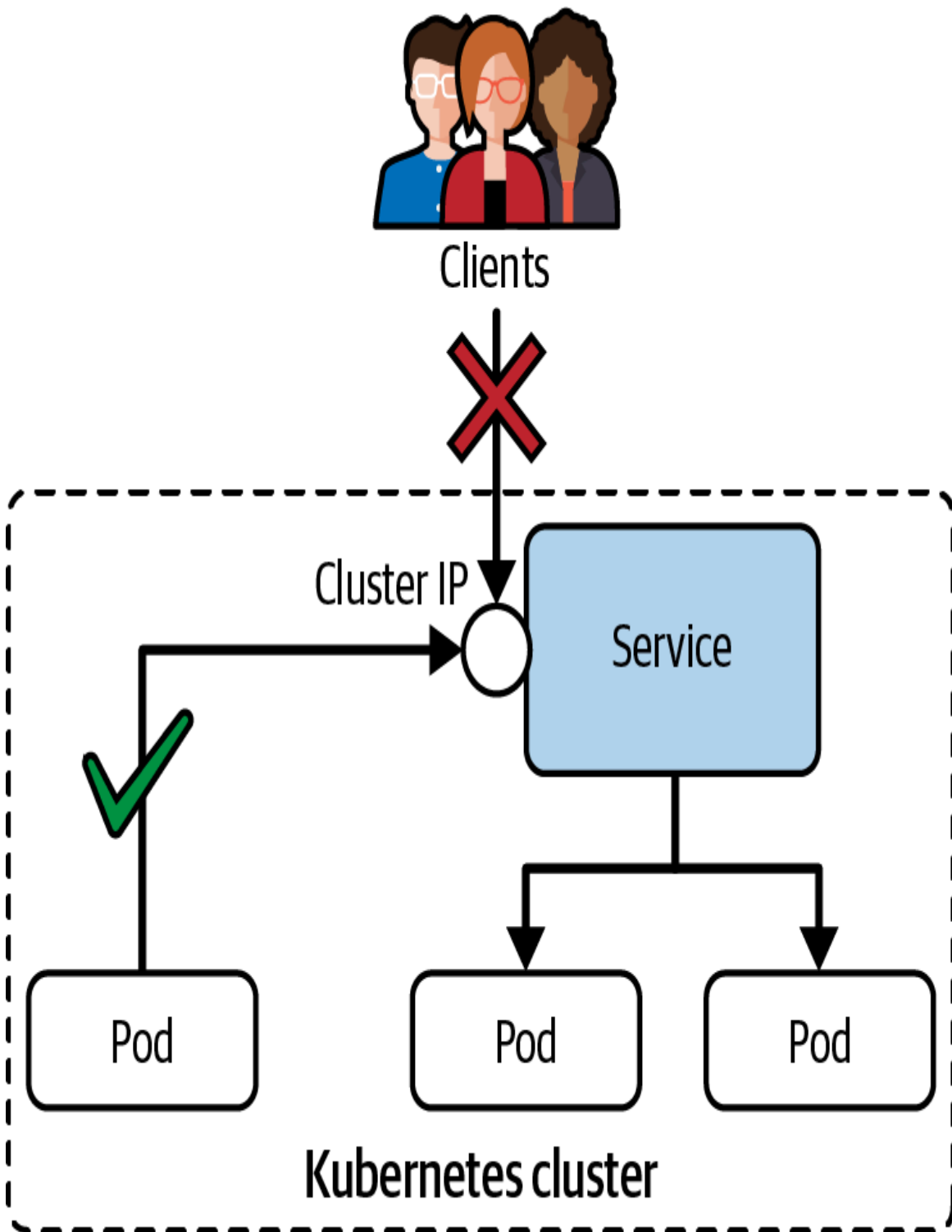


Figure 17-4. Accessibility of a Service with the type `ClusterIP`

## 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 17-2** shows an abbreviated version of the Service runtime representation.

### *Example 17-2. A ClusterIP Service object at runtime*

---

```
apiVersion: v1
kind: Service
metadata:
  name: echoserver
spec:
  type: ClusterIP ❶
  clusterIP: 10.96.254.0 ❷
  selector:
    app: echoserver
  ports:
  - port: 5005
    targetPort: 8080
    protocol: TCP
```

❶ The Service type set to `ClusterIP`.

- ② 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.

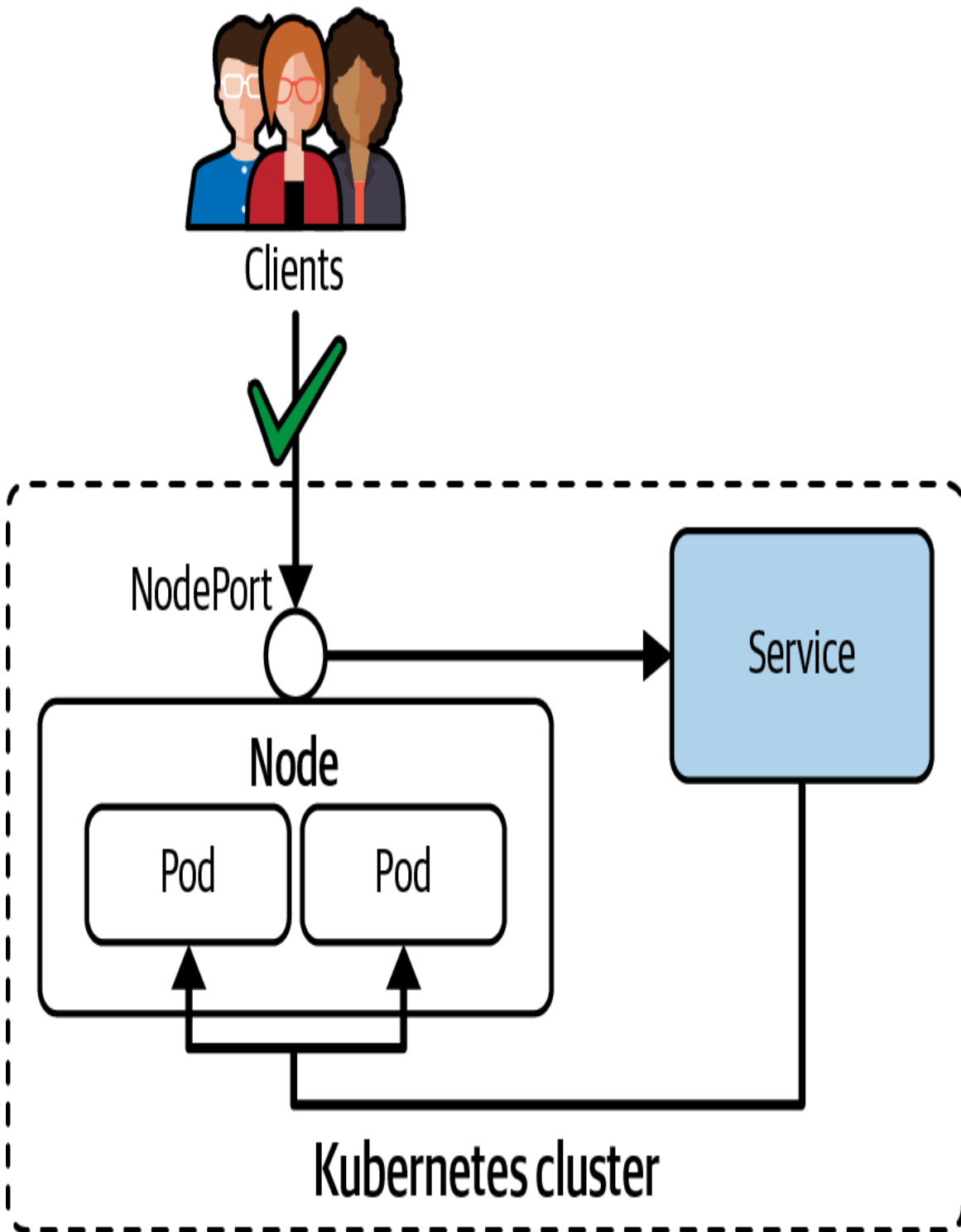
You can check on the actual key-value pairs by listing the environment variables of the container, as follows:

```
$ kubectl exec -it echoserver -- env
ECHOSEVER_SERVICE_HOST=10.96.254.0
ECHOSEVER_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 17-5](#) illustrates the routing of traffic to Pods via a NodePort-type Service.



*Figure 17-5. 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 17-3**. 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 17-3. A NodePort Service object at runtime*

---

```
apiVersion: v1
kind: Service
metadata:
  name: echoserver
spec:
  type: NodePort ❶
  clusterIP: 10.96.254.0
  selector:
    app: echoserver
  ports:
    - port: 5005
      nodePort: 30158 ❷
      targetPort: 8080
      protocol: TCP
```

- ❶ The Service type set to `NodePort`.
- ❷ 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

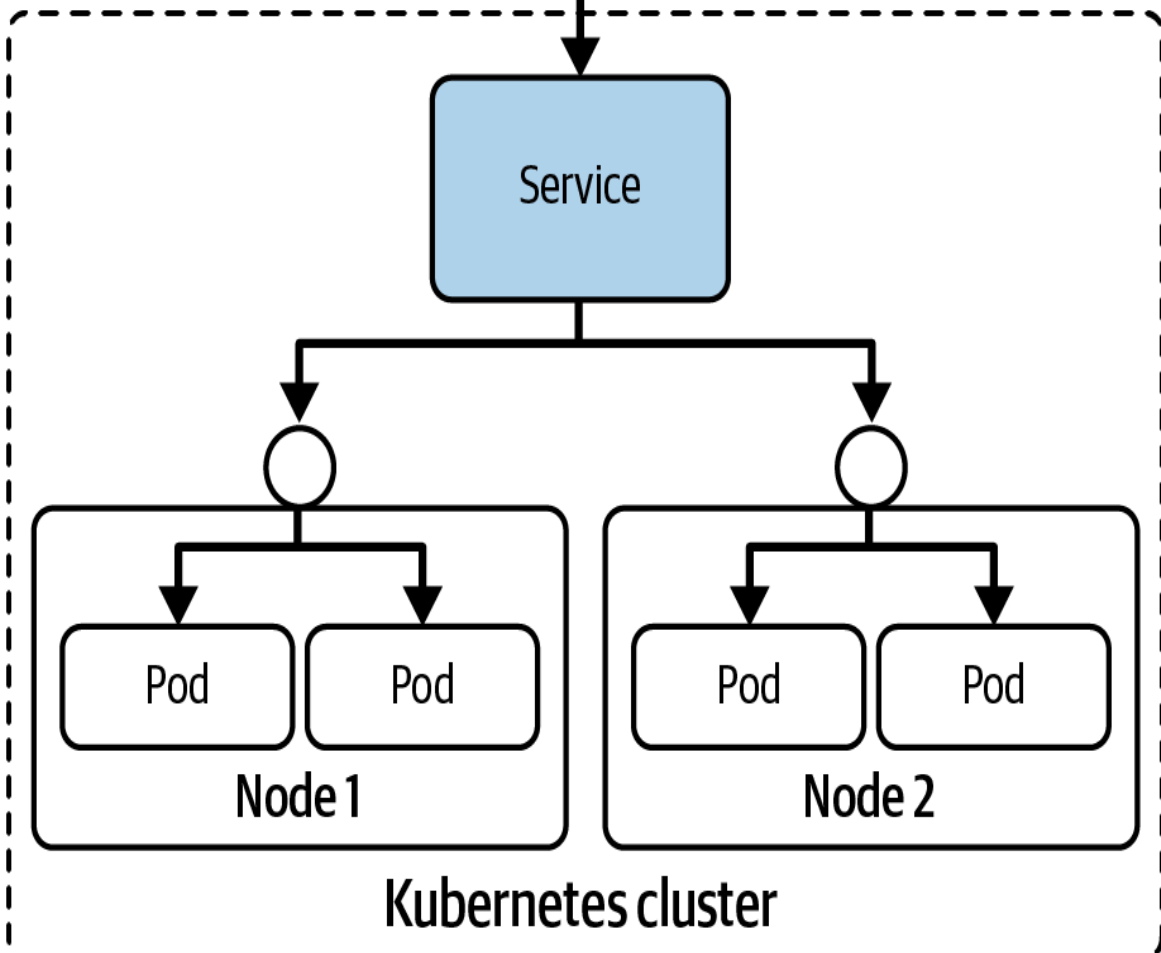
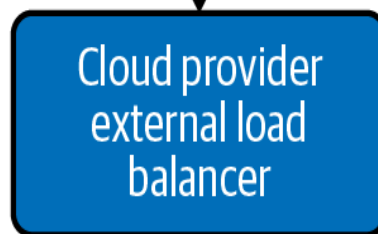
Kubernetes does not offer a native load balancer solution for on-premises clusters. Cloud providers are in charge of providing an appropriate implementation. The [MetalLB project](#) aims to fill the gap.

**Figure 17-6** shows an architectural overview of the `LoadBalancer` Service type.





Clients



*Figure 17-6. 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 17-4](#).

### *Example 17-4. A LoadBalancer Service object at runtime*

---

```
apiVersion: v1
kind: Service
metadata:
  name: echoserver
spec:
  type: LoadBalancer ❶
  clusterIP: 10.96.254.0
  loadBalancerIP: 10.109.76.157 ❷
  selector:
    app: echoserver
  ports:
    - port: 5005
```

```
targetPort: 8080
nodePort: 30158
protocol: TCP
```

- ❶ The Service type set to `LoadBalancer`.
- ❷ 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
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. You need to expose a web application that runs on port 8080 and has a metrics endpoint on port 9090. The application should be accessible from outside the cluster.

Create a Deployment named `webapp` using the image `nginxdemos/hello:0.4-plain-text` with 3 replicas.

Create a Service named `webapp-service` of type `NodePort` that exposes port 80 as port 80 (name it `web`), exposes port 9090 as port 9090 (name it `metrics`). Set the `NodePort` for the `web` port to 30080. Use appropriate selectors to target your Deployment.

Verify the Service is working by accessing it through the ClusterIP.

2. You have a backend database and a frontend application. The frontend needs to connect to the database using service discovery. Both should only be accessible within the cluster.

Create a Deployment named `database` with 1 replica using image `mysql:9.4.0` with these environment variables:

`MYSQL_ROOT_PASSWORD=secretpass` and

`MYSQL_DATABASE=myapp`.

Create a ClusterIP Service named `database-service` that exposes port 3306 and targets the database Pods.

Create a Deployment named `frontend` with 2 replicas using image `busybox:1.35` that runs a command to continuously test the database connection using the following command: `sh -c "while true; do nc -zv database-service 3306; sleep 5; done"`

Verify that the `frontend` Pods can resolve and connect to the database Service.

# Chapter 18. Ingresses

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 18th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

**Chapter 17** 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:

- Know how to use Ingress controllers and Ingress resources

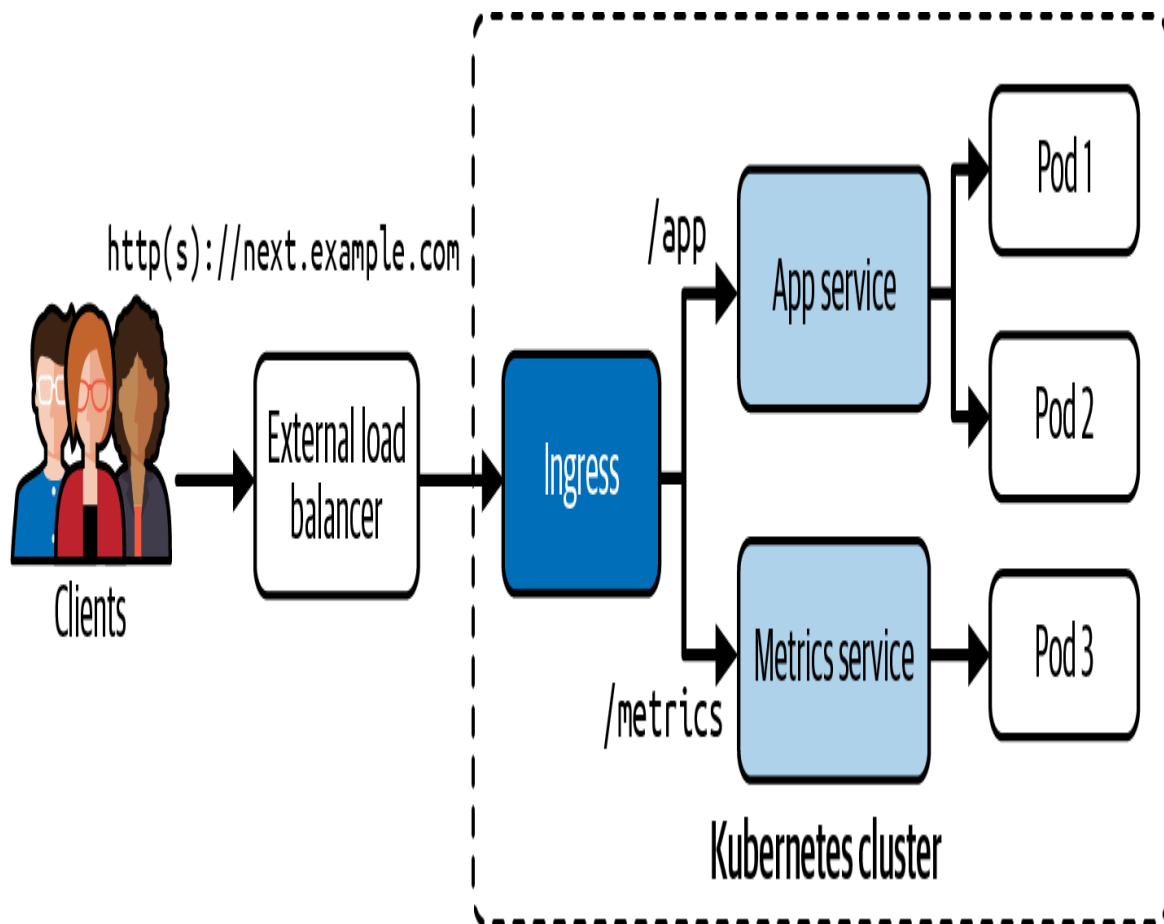
### **ACCESSING AN INGRESS IN MINIKUBE**

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 18-1](#).





*Figure 18-1. Managing external access to the Services via HTTP(S)*

The scenario depicted in **Figure 18-1** 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
ingress-nginx-admission-create-qghrp	0/1	Completed
ingress-nginx-admission-patch-56z26	0/1	Completed
ingress-nginx-controller-7c6974c4d8-2gg8c	1/1	Running

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 18-1](#) describes the three rules.

*Table 18-1. Ingress rules*

Type	Example	Description
An optional host	<code>next.example.com</code>	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	<code>/app</code>	Incoming traffic must match the host and path to correctly forward the traffic to a Service.
The backend	<code>app-service:8080</code>	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 18-1](#).

#### *Example 18-1. 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 ❶
spec:
  rules:
  - host: next.example.com ❷
    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

```



- ❶ Assigns a NGINX ingress-specific annotation for rewriting the URL.
- ❷ Defines the rule that maps the `app-service` backend to the URL `next.example.com/app`.
- ❸ 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 18-2](#) indicates the evaluation for incoming requests and their paths. See the [Kubernetes documentation](#) for a more comprehensive list.

*Table 18-2. 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           Type          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. Your company has two microservices running in the same Kubernetes cluster: A frontend application that serves the main website, and an API backend service that handles API requests. Both services need to be accessible from outside the cluster through the same domain name (app.example.com) but with different URL paths. Use the **NGINX Ingress controller**.

Create a namespace called `webapp`. Deploy two applications: a frontend application (use `nginx:1.29.1-alpine` image, name: `frontend`, 2 replicas), and an API application (use the `httpd:2.4.65-alpine` image, name: `api`, 2 replicas).

Create ClusterIP Services for both Deployments, a `frontend` Service on port 80, an API Service on port 80.

Create an Ingress resource that uses the host `app.example.com` and routes the paths `/` and `/app` to the `frontend` Service. It also routes `/api` to the `api` Service. Use the `nginx` Ingress class. Verify the proper Ingress configuration by making a call to it.

2. Your team wants to implement a blue-green deployment strategy using Ingress. You need to create an Ingress configuration that can switch traffic between two versions of an application, with the ability to do canary testing before full switchover. The team decided to use the **NGINX Ingress controller**. You may reference the controller-specific annotations from the **documentation**.

Create a namespace called `production-apps`. Deploy two versions of an application: A blue version (current) named `app-blue` with 3 replicas with the container image `nginxdemos/hello:0.3-plain-text`, and a green version (new) named `app-green` with 3 replicas with the container image `nginxdemos/hello:0.4-plain-text`.

Create 3 Ingress resources in the `production-apps` namespace. A main ingress routing to blue version, a canary ingress for testing green version (20% traffic). Assign the hostname `app.production.com`. The traffic routing should look as follows. Default traffic goes to blue version. 20% of traffic goes to green version (canary).

Add an entry in `/etc/hosts` that maps the load balancer IP address to the host `app.production.com`. Verify access to the Ingresses via this hostname. Test the blue-green traffic routing distribution.

# Chapter 19. Gateway API

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 19th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

The Gateway API is a collection of Kubernetes resources that provide advanced traffic management capabilities for cluster ingress. As a more expressive and extensible alternative to the traditional Ingress resource, the Gateway API offers role-oriented design, support for multiple protocols beyond HTTP/HTTPS, and enhanced traffic routing features.

The Gateway API replaces the Ingress primitive and is becoming increasingly important as organizations adopt this modern approach to managing external traffic.

As a candidate to the exam, you will need to understand how to deploy the Gateway API and instantiate the relevant objects to stand up ingress access to your cluster.

## COVERAGE OF CURRICULUM OBJECTIVES

This chapter addresses the following curriculum objective:

- Use the Gateway API to manage Ingress traffic

## Why is the Ingress Primitive not Sufficient?

The Ingress API is the standard Kubernetes way to configure external HTTP/HTTPS load balancing for Services, however, it has limitations. While the Ingress supports TLS termination and simple content-based request routing of HTTP traffic, real world use cases call for more advanced features. Enhancing the existing Ingress API model of the Ingress wouldn't allow adding those features easily for a variety of reasons.

### *Ingress controller-specific extensibility*

Advanced features like traffic splitting, rating limiting, request/response manipulation are provided by non-portable annotations for specific Ingress implementations.

### *Insufficient permission model*

The Ingress API is not well suited for multi-tenant environments that call for a strong permission model.

In this chapter, I am not going to discuss all possible configuration options and features of the Gateway API. That would go beyond the necessary coverage of the exam, however, I will explain how you can model ingress HTTP traffic similar to what you learned in [Chapter 18](#). See the [Gateway API documentation](#) that explains how to implement other use cases.



## Working with the Gateway API

The Gateway API is the official successor to the Ingress primitive representing a superset of Ingress functionality, while at the same time enabling more advanced use cases.

You can think of the Gateway API as a unified and standardized API for managing traffic into and out of a Kubernetes cluster instead of having to choose from individual Ingress implementations. Product-specific annotations are not needed anymore to configure routing options. The Gateway API offers a flexible way to incorporate similar features. Effectively, the Gateway API is a universal specification supported by a wide range of different implementations.

Instead of handling one single primitive like the Ingress, the Gateway API splits up responsibilities into multiple primitives explained in the next section.

## Gateway API Resources

The Gateway API introduces a layered approach to traffic management through four primary resource types that work together to handle incoming traffic.

### *Gateway*

Defines an instance of traffic handling infrastructure, such as a cloud load balancer.

### *GatewayClass*

Each Gateway is associated with a GatewayClass, which describes the actual kind of gateway controller that will handle traffic for the Gateway.

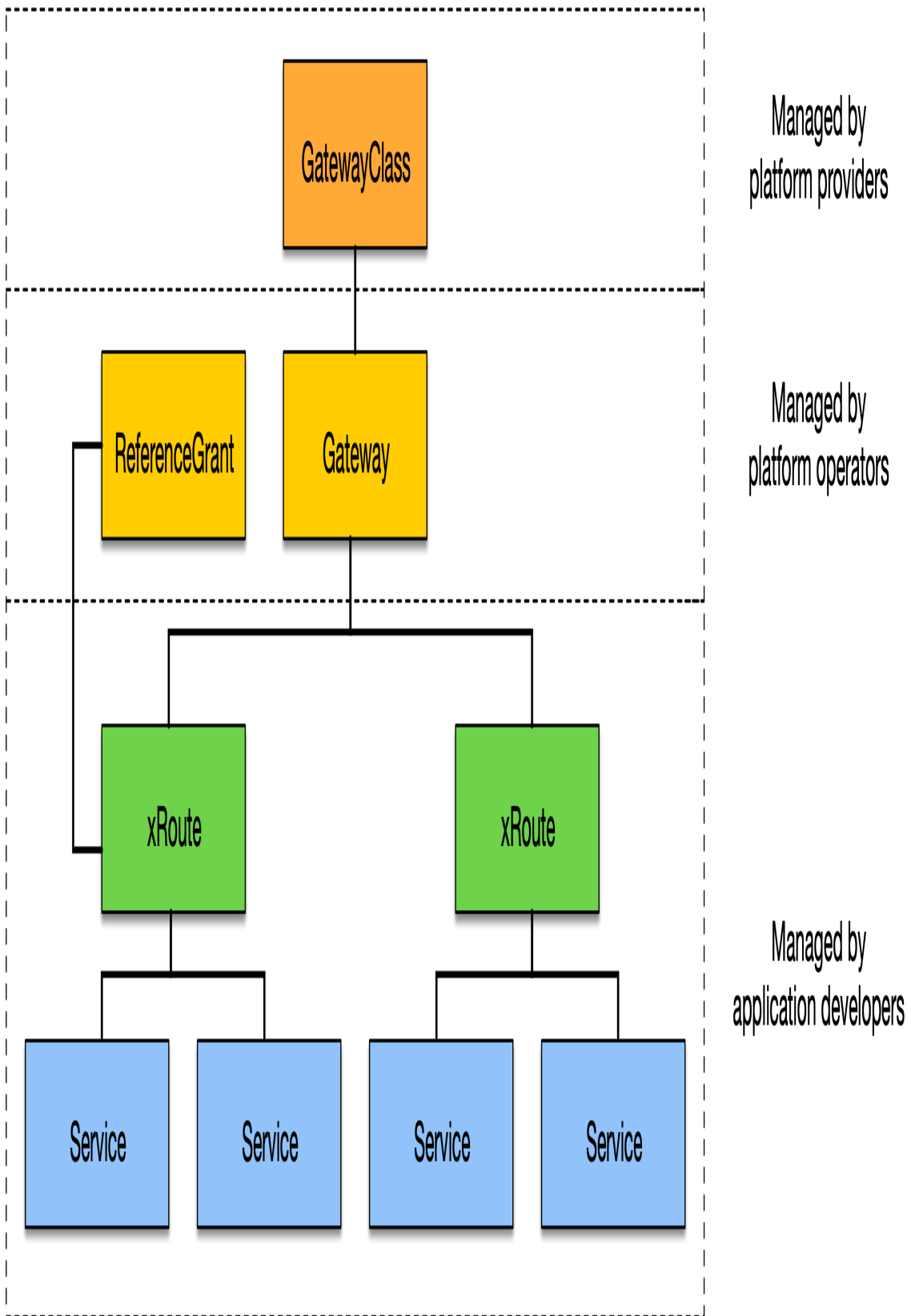
### *HTTPRoute/GRPCRoute*

Defines HTTP or GRPC-specific rules for mapping traffic from a Gateway listener to a representation of backend network endpoints. These endpoints are often represented as a Service.

### *ReferenceGrant*

Can be used to enable cross-namespace references within the Gateway API, e.g. routes may forward traffic to backends in other namespaces.

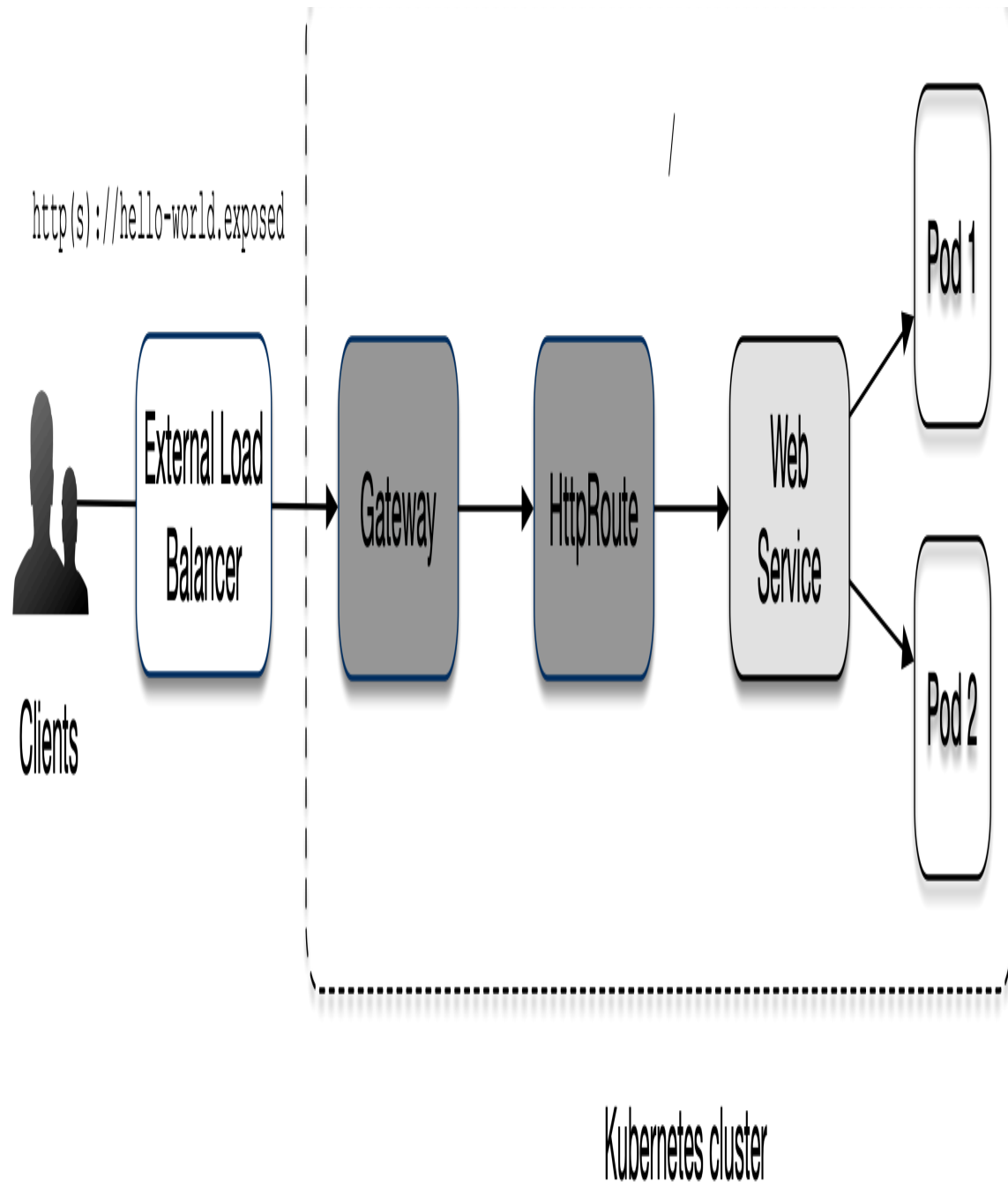
Managing those Gateway API resources falls within the responsibility of different personas, as shown in **Figure 19-1**.



*Figure 19-1. Gateway API resources managed by personas*

The GatewayClass is provided by platform providers, e.g. the cloud provider. The Gateway and ReferenceGrant is installed by the Kubernetes cluster administrator. Lastly, the HTTPRoute and GRPCRoute are created by application developers.

The prominent Gateway API use case I want to demonstrate in this chapter is how to route HTTP ingress traffic to multiple services based on a context URL. **Figure 19-2** illustrates the use case.



*Figure 19-2. Gateway API HTTP traffic routing*

Before we can actually create the objects shown in the visualization, we'll need to tell the Kubernetes cluster about the Gateway API CRDs.

## Installing the Gateway API CRDs

At the time of writing, the Gateway API resources do not ship with the standard set of Kubernetes API resources. You will have to install the Gateway API in the form of Custom Resource Definitions. The following command installs version 1.3.0 of the CRDs.

```
$ kubectl apply -f https://github.com/kubernetes-
sigs/gateway-api/releases/\
download/v1.3.0/standard-install.yaml
customresourcedefinition.apiextensions.k8s.io/\
gatewayclasses.gateway.networking.k8s.io created
customresourcedefinition.apiextensions.k8s.io/\
gateways.gateway.networking.k8s.io created
customresourcedefinition.apiextensions.k8s.io/\
grpcroutes.gateway.networking.k8s.io created
customresourcedefinition.apiextensions.k8s.io/\
httproutes.gateway.networking.k8s.io created
customresourcedefinition.apiextensions.k8s.io/\
referencegrants.gateway.networking.k8s.io created
```

You will now be able to list the CRDs by searching for the API group used by the Gateway API resources.

```
$ kubectl get crds | grep gateway.networking.k8s.io
gatewayclasses.gateway.networking.k8s.io    2025-08-
07T18:14:16Z
gateways.gateway.networking.k8s.io          2025-08-
07T18:14:16Z
grpcroutes.gateway.networking.k8s.io        2025-08-
07T18:14:16Z
httproutes.gateway.networking.k8s.io        2025-08-
07T18:14:16Z
```

See the [official Github repository](#) for more information on the Gateway API.

## Deploying a Gateway Controller

The Gateway API requires a [controller implementation](#) to function. Different controllers offer various features and performance characteristics. For this example, we'll use the [Envoy Gateway](#) implementation installed by Helm.

```
helm install eg oci://docker.io/envoyproxy/gateway-helm --  
version v1.4.2 \  
-n envoy-gateway-system --create-namespace
```

You will want to wait until the Gateway implementation becomes fully functional.

```
$ kubectl wait --timeout=5m -n envoy-gateway-system  
deployment/envoy-gateway \  
--for=condition=Available  
deployment.apps/envoy-gateway condition met
```

With the prerequisites in place, we'll set up everything needed to use the Gateway API to route ingress HTTP traffic to a Service backend.

## Creating GatewayClasses

Depending on the Kubernetes environment you are operating in, you may or may not have to create a GatewayClass. Cloud provider Kubernetes clusters should already come with one.

In case you want to create your own GatewayClass, you will first have to determine the Gateway controller name. The controller name depends on the controller implementation installed. You can usually look up the controller name in its documentation.

The controller name for Envoy is

`gateway.envoyproxy.io/gatewayclass-controller`.

Create the file *gateway-class.yaml* with the contents shown in **Example 19-1**.

*Example 19-1. A GatewayClass that uses the envoy GatewayClass controller*

---

```
apiVersion: gateway.networking.k8s.io/v1
kind: GatewayClass
metadata:
  name: envoy
spec:
  controllerName: gateway.envoyproxy.io/gatewayclass-
controller ❶
```

❶ The reference to the controller installed by the Helm chart.

Run the following command to create the GatewayClass object:

```
$ kubectl apply -f gateway-class.yaml
gatewayclass.gateway.networking.k8s.io/envoy created
```

To list GatewayClass objects, run the following command:

```
$ kubectl get gatewayclasses
NAME          CONTROLLER
ACCEPTED     AGE
envoy         gateway.envoyproxy.io/gatewayclass-controller
True          31s
```



You should see the object we created earlier, and potentially other GatewayClass objects that came with the Kubernetes environment.

## Creating Gateways

With the GatewayClass established, create a Gateway resource to handle incoming traffic. Create a file named *gateway.yaml* and populate the content, as shown in [Example 19-2](#).

### *Example 19-2. A Gateway exposing a HTTP listener*

---

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: hello-world-gateway
spec:
  gatewayClassName: envoy ❶
  listeners:
    - name: http
      protocol: HTTP ❷
      port: 80 ❷
```

- ❶ References the GatewayClass by name.
- ❷ The listener accepting HTTP traffic on port 80.

Run the following command to create the Gateway object:

```
$ kubectl apply -f gateway.yaml
gateway.gateway.networking.k8s.io/hello-world-gateway
created
```

You list the Gateway object with the command below:

```
$ kubectl get gateways
NAME                                CLASS    ADDRESS    PROGRAMMED    AGE
hello-world-gateway                envoy                False         16s
```

There's only one additional object to set up to finalize the HTTP traffic routing, the HTTPRoute object.

## Creating HTTPRoutes

Store the definition of the HTTPRoute in the YAML manifest file named *httproute.yaml*, shown in [Example 19-3](#). For brevity, this chapter will not demonstrate how to create Service backends. Please reference [Chapter 17](#) for more information.

### *Example 19-3. A HTTPRoute routing traffic to a Service backend*

---

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: hello-world-httproute
spec:
  parentRefs:
    - name: hello-world-gateway
  hostnames:
    - "hello-world.exposed" ❶
  rules:
    - backendRefs:
        - group: "" ❷
          kind: Service ❷
          name: web ❷
          port: 3000 ❷
          weight: 1 ❸
      matches:
        - path:
            type: PathPrefix ❹
            value: / ❹
```

- ❶ The hostname to make traffic routing available for.
- ❷ The Service backend to route the traffic to.
- ❸ Determines the proportion of traffic that will be sent to that specific Service.
- ❹ The routing rules based on the requested URL context.

Run the following command to create the HTTPRoute object:

```
$ kubectl apply -f httproute.yaml
httproute.gateway.networking.k8s.io/hello-world-httproute
created
```

You can list all HTTPRoute objects with the command shown below:

```
$ kubectl get httproutes
NAME                                HOSTNAMES                                AGE
hello-world-httproute              ["hello-world.exposed"]                 64s
```

## Accessing the Gateway

Accessing the Gateway differs depending on the Kubernetes cluster you are using. Follow the instructions in the section based on your Kubernetes cluster setup and assumes that you don't have **external load balancer support**.

Get the name of the Envoy Service created by the example Gateway and assign it to the environment variable `ENVOY_SERVICE`.

```
$ export ENVOY_SERVICE=$(kubectl get svc -n envoy-gateway-
system \
--selector=gateway.envoyproxy.io/owning-gateway-
namespace=default,\
gateway.envoyproxy.io/owning-gateway-name=hello-world-
gateway \
-o jsonpath='{.items[0].metadata.name}')
```

Port forward to the Envoy Service:

```
$ kubectl -n envoy-gateway-system port-forward
service/${ENVOY_SERVICE} 8889:80 &
[2] 93490
```

```
Forwarding from 127.0.0.1:8889 -> 10080
Forwarding from [::1]:8889 -> 10080
```

With the relevant entry in `/etc/hosts`, you should be able to make a call to the Gateway.

```
$ curl hello-world.exposed:8889
Handling connection for 8889
Hello World
```

In this example, the backend returns with the message “Hello World”. A successful response may look different depending on the implementation details of your application.

## Ingress to Gateway API Migration

The Gateway API represents the next evolution of traffic management in Kubernetes, designed to eventually replace the traditional Ingress resource.

The transition typically involves replacing Ingress resources with Gateway and HTTPRoute combinations. A single Gateway resource can handle multiple applications through different Route resources, similar to how an Ingress Controller works but with cleaner separation of responsibilities.

The Gateway API graduated to stable status for core features in late 2023. While Ingress remains supported and widely used, major ingress controllers like NGINX, Istio, and Envoy Gateway now offer Gateway API implementations. Organizations should evaluate migration based on their need for advanced traffic management features and operational complexity requirements.

Most migrations follow a phased approach: deploying Gateway API alongside existing Ingress, gradually migrating applications, and eventually deprecating Ingress resources. Many controllers support

both APIs simultaneously, allowing for smooth transitions without service disruption.

The official conversion tool named **ingress2gateway** automatically translates Ingress resources to Gateway API equivalents. It handles basic conversions and provides a foundation for more complex migrations.

## Summary

The Gateway API represents the future of Kubernetes ingress management, offering powerful features while maintaining clarity through its role-oriented design. As more organizations adopt this standard, proficiency with the Gateway API becomes an essential skill for Kubernetes administrators.

## Exam Essentials

### *Check on existing GatewayClasses*

Remember that exam environment may have pre-installed Gateway controllers, so always check available GatewayClasses before creating resources.

### *Understand the Gateway API CRDs*

Practice creating different Gateway configurations, experiment with various HTTPRoute patterns, and understand how the components work together to build a solid foundation for managing production traffic.

## Sample Exercises

Solutions to these exercises are available in **Appendix A**.

1. You are tasked with setting up ingress traffic management for a new application using the Gateway API. The application consists of two services that need to be accessible from outside the cluster. Use the **NGINX Gateway Fabric** controller implementation.

Create a Deployment named `web-app` with 2 replicas using the `nginx:1.21` image in the `default` namespace.

Create a Deployment named `api-app` with 2 replicas using the `httpd:2.4` image in the `default` namespace. Expose both Deployments as ClusterIP Services on port 80. You can shortcut the setup by applying the YAML file *setup.yaml* in the directory *app-a/ch19/basic-gateway* of the checked-out GitHub repository *[bmuschko/cka-study-guide](#)*.

Create a Gateway named `main-gateway` in the `default` namespace. Configure it to listen on port 80 for HTTP traffic. Use the hostname `example.local`. Create an HTTPRoute named `app-routes` that routes traffic with path `/web` to the `web-app` Service. It also routes traffic with path `/api` to the `api-app` Service. Define path prefix matching.

Check that the Gateway is ready and has an assigned IP/hostname. Verify that the HTTPRoute is accepted and bound to the Gateway, then test connectivity to the Gateway.

2. Your organization needs to set up a production-ready Gateway configuration that handles HTTP traffic for applications across multiple namespaces. Use the **NGINX Gateway Fabric** controller implementation.

Create two namespaces named `production` and `staging`. In the `production` namespace, create a Deployment named `prod-web` with 3 replicas using `nginx:1.22` image. Create a ClusterIP Service exposing

port 80 which routes traffic to the replicas of `prod-web`. In the `staging` namespace, create a Deployment named `staging-web` with 2 replicas using `nginx:1.21` image. Create a ClusterIP Service exposing port 80. Create a ClusterIP Service exposing port 80 which routes traffic to the replicas of `staging-web`. You can shortcut the setup by applying the YAML file *setup.yaml* in the directory *app-a/ch19/cross-namespace-gateway* of the checked-out GitHub repository *[bmuschko/cka-study-guide](#)*.

Create a Gateway named `gateway` in the `production` namespace. Configure a HTTP listener on port 80. Use the hostname `example.com`.

Create an HTTPRoute named `prod-route` in the `production` namespace. The object should route traffic with path `/app` to the `prod-web` Service, uses exact path matching, and attaches to the `gateway`. Create another HTTPRoute named `staging-route` in the `staging` namespace. The object should route traffic with path `/staging` to the `staging-web` Service, uses path prefix matching, and attaches to the `gateway` in the `production` namespace. Configure proper cross-namespace reference policy. Add request headers to identify the environment (production vs. staging).

Verify the correct HTTP routing. The production route should only respond to the exact path `/app`. The staging route should respond to `/staging`. Both routes should successfully serve the nginx welcome page.

# Chapter 20. Network Policies

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 20th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

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.



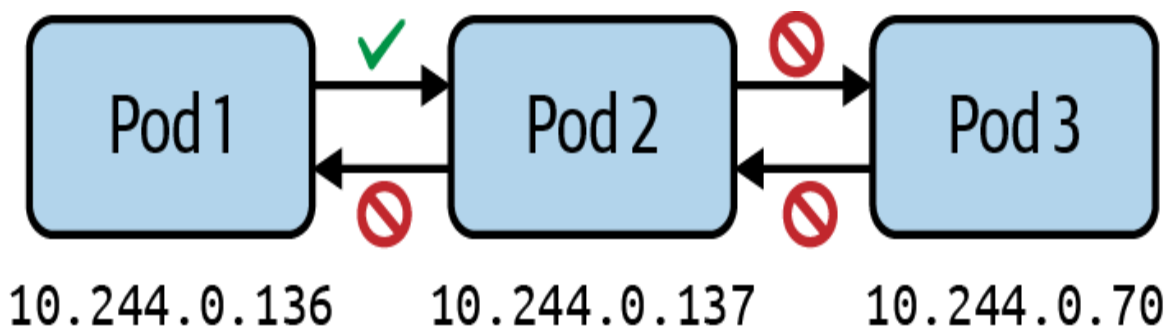
## COVERAGE OF CURRICULUM OBJECTIVES

This chapter addresses the following curriculum objective:

- Define and enforce 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 20-1**.



*Figure 20-1. 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 a 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
<div>RESTARTS      AGE</div> <div>cilium-k5td6</div> <div>110s</div>	1/1	Running
<div>cilium-operator-f5dcdcc8d-njfbk</div> <div>110s</div>	1/1	Running

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 Deployments 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 20-2** shows the Pods and their assigned labels.

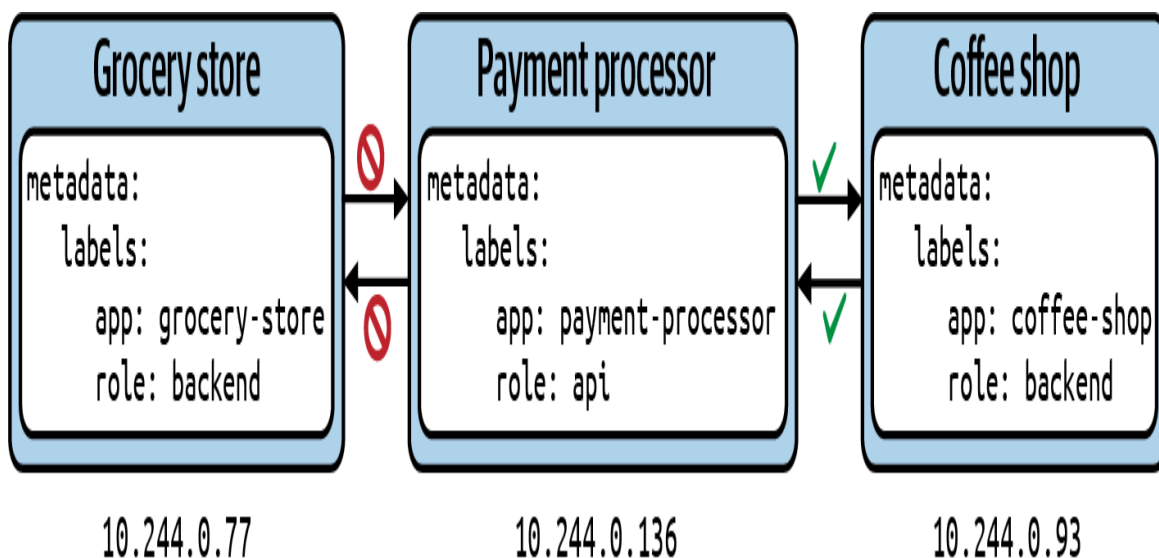


Figure 20-2. Limiting traffic to and from a Pod

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 20-1](#), stored in the file *networkpolicy-api-allow.yaml*, shows a network policy for the scenario described previously.

### *Example 20-1. 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 20-1** shows the attributes on the `spec` level.

*Table 20-1. Spec attributes of a network policy*

Attribute	Description
<code>podSelector</code>	Selects the Pods in the namespace to apply the network policy to.
<code>policyTypes</code>	Defines the type of traffic (i.e., ingress and/or egress) the network policy applies to.
<code>ingress</code>	Lists the rules for incoming traffic. Each rule can define <code>from</code> and <code>ports</code> sections.
<code>egress</code>	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 20-2** lists the relevant attributes for the `to` and `from` selectors.

*Table 20-2. Attributes of a network policy *to* and *from* selectors*

Attribute	Description
<code>podSelector</code>	Selects Pods by label(s) in the same namespace as the network policy that should be allowed as ingress sources or egress destinations.
<code>namespaceSelector</code>	Selects namespaces by label(s) for which all Pods should be allowed as ingress sources or egress destinations.
<code>namespaceSelector</code> and <code>podSelector</code>	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](https://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 20-2. Disallowing all traffic with the default policy*

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
  namespace: internal-tools
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
```

❶

❷

❷

❷

- ❶ The curly braces for `spec.podSelector` mean “apply to all Pods in the namespace.”
- ❷ 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, 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 20-3](#) allows access on port 80.

*Example 20-3. 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
```

❶  
❶  
❶

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. Your cluster has two teams working in separate namespaces. You need to implement network policies that control cross-namespace communication.

Navigate to the directory *app-a/ch20/cross-namespace-control* of the checked-out GitHub repository [bmuschko/cka-study-guide](#). Create the objects from the YAML manifest *setup.yaml*. Inspect the objects in the namespaces `team-alpha` and `team-beta`.

Create a NetworkPolicy in `team-alpha` namespace that allows the `alpha-app` Pod to connect only to the `team-beta` namespace and denies all other egress traffic except DNS.

Create a NetworkPolicy in the `team-beta` namespace that allows the `beta-app` Pod to receive traffic from `team-alpha` namespace on port 80 and denies all other ingress traffic.

Test the policies to ensure that the `alpha-app` Pod can reach the `beta-app` Pod. The `alpha-app` Pod cannot reach external sites and the `beta-app` Pod cannot receive traffic from other sources.

2. You have a three-tier application with frontend, backend, and database components. You need to implement network policies to ensure only the `backend` Pod can access the database, while the `frontend` can only communicate with the `backend` Pod.

Navigate to the directory *app-a/ch20/database-access-control* of the checked-out GitHub repository *bmuschko/cka-study-guide*. Create the objects from the YAML manifest *setup.yaml*. Inspect the objects in the namespace `production`.

Create a NetworkPolicy named `database-policy` that applies to Pods with label `tier=database`. It should only allow ingress from Pods with label `tier=backend` on port 6379.

Create a NetworkPolicy named `backend-policy` that applies to Pods with label `tier=backend`. It should only allow ingress from Pods with label `tier=frontend` on port 80. Allow egress to Pods with label `tier=database` on port 6379. Allow DNS resolution on port 53.

Create a default deny-all Ingress policy for the `production` namespace.

Verify your policies by testing connectivity: The `frontend` Pod should *not* be able to reach database. The `frontend` Pod should be able to reach the `backend` Pod. The `backend` Pod should be able to reach database Pod.

# Part VI. Troubleshooting

---

The Troubleshooting domain of the exam focuses on diagnosing and resolving issues across the entire Kubernetes stack, from cluster-level problems to application-specific failures. Key competencies include identifying and fixing problems with cluster components, troubleshooting node failures and resource constraints, monitoring resource utilization for both cluster infrastructure and applications, analyzing container logs and output streams for debugging, and resolving service connectivity and networking issues such as DNS problems, network policies, and service endpoint configurations.

The following chapters cover these concepts:

- **Chapter 21** covers troubleshooting strategies for pods and containers in Kubernetes, teaching developers to diagnose issues systematically from high-level object analysis down to specific details, including using metrics server for resource monitoring and troubleshooting services and networking to identify root causes.
- **Chapter 22** covers troubleshooting Kubernetes infrastructure issues, including diagnosing and fixing problems with cluster components and resolving node failures. You'll learn techniques using `kubectl`, system logs, and service status checks to identify root causes and quickly restore cluster functionality in production environments.



# Chapter 21. Troubleshooting Applications

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 21th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

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:

- Manage and evaluate container output streams
- Troubleshoot services and networking
- Monitor cluster and application resource usage

## 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 21-1](#) lists some of those error statuses and explains how to fix them.

*Table 21-1. Common Pod error statuses*

Status	Root cause	Potential fix
<code>ImagePullBackOff</code> 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.

## NOTE

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.

## 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 21-1](#) defines a Pod running a shell command.

### *Example 21-1. 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 21-2](#).

*Example 21-2. 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 21-3**.

#### *Example 21-3. 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
```

## Troubleshooting Services and Networking

A Service provides a unified network interface for Pods. For full coverage on networking aspects in Kubernetes, see **Chapter 17**. Here, I want to point out troubleshooting techniques for this primitive.

Service and networking problems in Kubernetes are among the most challenging issues to diagnose because they involve multiple layers: Pod networking, service discovery, DNS resolution, network policies, and ingress controllers. These issues typically manifest as connection timeouts, refused connections, or intermittent failures that can cripple application functionality.

## Diagnosing Service-to-Pod Label Selection

In case you can't reach the Pods that should map to the Service, start by ensuring that the label selector matches with the assigned labels of the Pods. You can query the information by describing the Service and then render the labels of the available Pods with the option `--show-labels`. The following example does not have matching labels and therefore wouldn't apply to any of the Pods running in the namespace:

```
$ kubectl describe service myservice
...
Selector:                app=myapp
...
$ kubectl get pods --show-labels
```

NAME	READY	STATUS	RESTARTS	AGE
myapp-68bf896d89-qfhlv	1/1	Running	0	7m39s
app=hello				
myapp-68bf896d89-tzt55	1/1	Running	0	7m37s
app=world				

## Diagnosing Service-to-Pod Port Mapping

Services must correctly map ports between the Service definition and the selected Pod's containers. Check if the port mapping from the target port of the Service to the container port of the Pod is configured correctly. Both ports need to match or the network traffic wouldn't be routed properly:

```
$ kubectl get service myapp -o yaml | grep targetPort:
  targetPort: 80
$ kubectl get pods myapp-68bf896d89-qfhlv -o yaml | grep
containerPort:
  - containerPort: 80
```

## Inspecting the Service's Endpoints

A straightforward way to check if label selection and port mapping is set up properly is to inspect the Service's endpoints. Service Endpoints are API objects that represent the network addresses (IP addresses and ports) of the Pods backing a Service.

Use the `get endpoints` command to render a Service's endpoints. Ensure that the output lists the number of Pod IP addresses and container ports expected to be selected by the Service.

```
$ kubectl get endpoints myservice
NAME                ENDPOINTS                                AGE
myservice           172.17.0.5:80,172.17.0.6:80            9m31s
```

If the output of the command does not render any endpoints then you know that either label selection or port mapping has not been configured correctly in the Service.

## Verifying Accessibility Scope

Different service types (ClusterIP, NodePort, LoadBalancer) have different accessibility scopes. By default, the Service type is `ClusterIP`, which means that a Pod can be reached through the Service only if queried from the same node inside of the cluster. First, check the Service type.

```
$ kubectl get services
NAME                TYPE                CLUSTER-IP      EXTERNAL-IP
PORT(S)            AGE
myservice           ClusterIP           10.99.155.165   <none>
```

```
80/TCP      15m
...
```

If you think that `ClusterIP` is the proper type you wanted to assign, open an interactive shell from a temporary Pod inside the cluster and run a `curl` or `wget` command:

```
$ kubectl run tmp --image=busybox:1.37.0 -it --rm -- wget
10.99.155.165:80
```

## DNS Resolution Problems

Instead of using the cluster-internal IP address to access a `ClusterIP` Service, you will likely want to use the Services' DNS name instead.

```
$ kubectl run tmp --image=busybox:1.37.0 -it --rm -- wget
myservice:80
```

Before jumping to any conclusions, remember that Services in different namespaces require fully qualified domain names (FQDN). The more bullet proof option to check accessibility to a Service is to include the namespace.

```
$ kubectl run tmp --image=busybox:1.37.0 -it --rm -- wget \
myservice.default.svc.cluster.local:80
```

DNS issues can prevent Pods from resolving service names, causing application failures. It's possible that the CoreDNS Pods are currently not running. You can check by executing the following command:

```
$ kubectl get pods -n kube-system -l k8s-app=kube-dns
```

To scan the CoreDNS logs for error message, run the following command:



```
$ kubectl logs -n kube-system -l k8s-app=kube-dns --tail=50
```

CoreDNS logs reveal DNS-related problems through various error patterns that help identify the root cause of service discovery failures. Common issues include upstream connectivity problems where CoreDNS cannot reach external DNS servers, resolution failures for Services that don't exist or are incorrectly referenced, communication breakdowns with the Kubernetes API server preventing Service discovery, configuration problems causing routing loops or processing errors, and performance issues from either excessive query loads or insufficient resources. By recognizing these patterns in the logs, you can quickly determine whether the problem lies in network connectivity, configuration, or resource allocation.

## Network Policy Restrictions

Network Policies in Kubernetes act as a firewall at the Pod level, controlling which Pods can communicate with each other and what external traffic is allowed. When network policies are implemented, they follow a “default deny” model—once any network policy selects a Pod, that Pod becomes isolated and can only receive traffic explicitly allowed by policies.

This security feature, while essential for production environments, frequently causes mysterious connectivity issues where applications that previously worked suddenly experience timeouts or connection refused errors. Common symptoms include pods being unable to reach services they depend on, health checks failing, cross-namespace communication breaking, or external traffic being blocked unexpectedly.

The challenge with troubleshooting network policies lies in their implicit nature—there's no immediate error message indicating that a network policy is blocking traffic; instead, connections simply fail silently.

Debugging requires systematically checking if network policies exist in the namespace, understanding which Pods they select through their `podSelector`, examining the ingress and egress rules to see what traffic is permitted, and testing connectivity from different source Pods to identify exactly where the policy enforcement is occurring.

The following command checks if any network policies exist in the namespace `production`:

```
$ kubectl get networkpolicies -n production
```

To see which Pods are affected by network policies, render its details. The following command assumes that we found the network policy `api-policy` to exist:

```
$ kubectl describe networkpolicy api-policy -n production
```

Lastly, try to test connectivity from one Pod to another one either directly or through a Service in the same namespace. The following command runs a `curl` command against a Service named `backend-service`:

```
$ kubectl exec -it frontend-pod -n production -- curl  
http://backend-service:8080
```

If this times out, but the backend is running, it's fair to assume that the network policy comes into play here.

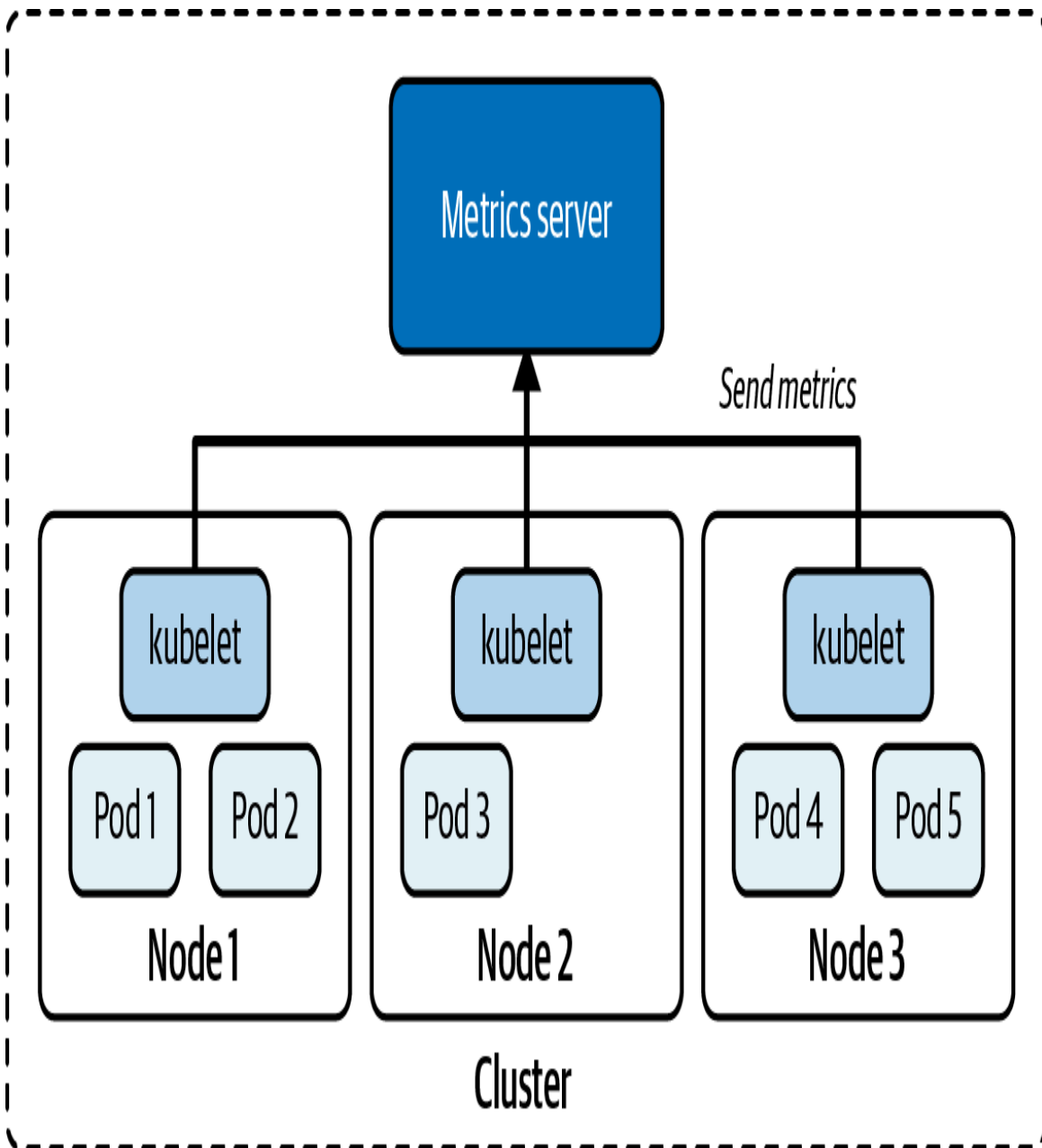
## Inspecting Resource Metrics

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 21-1**, kubelets running on nodes collect metrics and send them to the metrics server.



*Figure 21-1. 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 `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.

Effective Kubernetes network troubleshooting follows a systematic approach that begins with verifying the fundamentals, ensuring pods are running, Services have healthy endpoints, and DNS resolution is functioning, before moving to more complex investigations. Most importantly, understand the capabilities and limitations of each service type (ClusterIP for internal traffic, NodePort for node-level access, and LoadBalancer for external

exposure) to set appropriate expectations and choose the right troubleshooting approach for each scenario.

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 troubleshoot applications*

Applications running in a Pod can easily break due to misconfiguration. Think of possible scenarios that can occur and try to model them proactively to represent a failure situation. Then using the commands `get`, `logs`, and `exec`, get to the bottom of the issue and fix it. Try to dream up obscure scenarios to become more comfortable with finding and fixing application issues for different resource types. Refer to the [Kubernetes documentation](#) to learn more about debugging other Kubernetes resource types.

### *Make accessing container logs your daily bread and butter*

Accessing container logs is straightforward. Simply use the `logs` command. Practice the use of all relevant command-line options. The option `-c` targets a specific container. The option does not have to be used explicitly for single-container Pods. The option `-f` tails the log entries if you want to see live processing in an application. The `-p` option can be used for accessing logs if the container needed to be restarted, but you still want to take a look at the previous container logs.

### *Know how to diagnose networking issues*

Follow a systematic approach: first verify basic connectivity and DNS, then check service endpoints and selectors, examine port configurations, investigate any network policies, and finally test the complete traffic path from source to destination. Always remember that the exam environment may have pre-configured network policies or specific networking plugins that affect troubleshooting strategies.

### *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/ch21/troubleshooting-pod* of the checked-out GitHub repository [bmuschko/cka-study-guide](#).

Create a new Pod from the YAML manifest in the file *setup.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. 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-service* of the checked-out GitHub repository *bmuschko/cka-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.

3. You will inspect the metrics collected by the metrics server. Navigate to the directory *app-a/ch21/stress-test* of the checked-out GitHub repository *bmuschko/cka-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.

# Chapter 22. Troubleshooting Clusters

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 22th chapter of the final book. Please note that the GitHub repo will be made active later on. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

When a Kubernetes cluster experiences infrastructure-level failures, the impact can be catastrophic. Pods refuse to schedule, applications become unreachable, and entire nodes disappear from the cluster, potentially affecting hundreds of workloads simultaneously. Unlike application-specific issues that might affect a single service, problems with cluster components and nodes strike at the very foundation of your Kubernetes environment, making their swift diagnosis and resolution critical for maintaining system availability.

This chapter equips you with the skills to troubleshoot these infrastructure-level problems. You’ll learn to decode node conditions, investigate why nodes transition to NotReady states, resolve resource pressure situations, and trace Pod scheduling failures back to their root causes.

## COVERAGE OF CURRICULUM OBJECTIVES

This chapter addresses the following curriculum objectives:

- Troubleshoot clusters and nodes
- Troubleshoot cluster components

## Inspecting the Status of Cluster Nodes

There are many influencing factors that can render a Kubernetes cluster faulty on the component level. It's a good idea to list the nodes available in the cluster to identify potential issues:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE
control-plane-node	Ready	control-plane	2m45s
worker-node-1	Ready	<none>	2m36s
worker-node-2	Ready	<none>	2m29s
worker-node-3	Ready	<none>	2m22s

The output will give you a lay of the land. You can easily identify the responsibility of each node from the `ROLES` column, the Kubernetes version used, and the current health status.

There are a couple of things to look out for when identifying issues at a high level:

- Is the health status for the node anything other than "Ready"?

- Does the version of a node deviate from the version of other nodes?

In the following sections you can find individual sections on troubleshooting control plane nodes versus worker nodes.

## Inspecting the Status of Cluster Components

Among those **components available on the control plane node** are the following:

- kube-apiserver: Exposes the Kubernetes API used by clients like `kubectl` for managing objects.
- etcd: A key-value store for storing the cluster data.
- kube-scheduler: Selects nodes for Pods that have been scheduled but not created.
- kube-controller-manager: Runs controller processes (e.g., the job controller responsible for Job object execution).
- cloud-controller-manager (optional): Links cloud provider-specific API to the Kubernetes cluster. This controller is not available in on-premise cluster installations of Kubernetes.

In addition to the components running specifically on the control-plane nodes, **every node** (control-plane and worker nodes) can contain the following components:

- kubelet: Ensures that all Pods are running, including their containers.
- kube-proxy (optional): Maintains the network rules on nodes to implement Services.
- Container runtime: The software component responsible for running containers.

To discover those components and their status, list the Pods available in the namespace `kube-system`. Be aware that some components do not run in a Pod, e.g. the kubelet or the container runtime. Here, you can find the list of components:

```
$ kubectl get pods -n kube-system
```

NAME	READY	STATUS	RESTARTS
etcd	1/1	Running	1 (11d ago)
kube-apiserver	1/1	Running	1 (11d ago)
kube-controller-manager	1/1	Running	1 (11d ago)
kube-scheduler	1/1	Running	1 (11d ago)
...			

Any status that does not show “Running” should be inspected further. You can retrieve the logs for control-plane component Pods in the same fashion you do for any other Pod, using the `logs` command. The following command downloads the logs for the kube-apiserver component:

```
$ kubectl logs kube-apiserver -n kube-system
```

## Troubleshooting Node Issues

Control plane nodes are the critical components for keeping a cluster operational. As described in “[Managing a Highly Available Cluster](#)”, a cluster can consist of more than one control plane node to ensure a high degree of uptime. Detecting that one of the control plane nodes is faulty should be treated with extreme urgency to avoid compromising high-availability characteristics. For more information on troubleshooting techniques and root-cause analysis, reference the [Kubernetes documentation](#).

## Rendering cluster information

To further diagnose issues on the control plane node, run the command `kubectl cluster-info`. As you can see in the following output, the command renders the addresses of the control plane and other cluster services:

```
$ kubectl cluster-info
Kubernetes control plane is running at
https://192.168.64.21:8443
CoreDNS is running at
https://192.168.64.21:8443/api/v1/namespaces/ \
kube-system/services/kube-dns:dns/proxy
```

To further debug and diagnose cluster problems, use `kubectl cluster-info dump`.

For a detailed view of the cluster logs, append the `dump` subcommand. Due to the pages and pages of log messages, we won't render the output in this book. Parse through the message to see if you can find any errors:

```
$ kubectl cluster-info dump
```

## Node showing NotReady status

Worker nodes are responsible for managing the workload. Make sure you have a sufficient number of worker nodes available to distribute the load. For a deeper discussion on how to join worker nodes to a cluster, see [Chapter 4](#).

Any of the nodes available in a cluster can transition into an error state. It's your job as a Kubernetes administrator to identify those situations and fix them in a timely manner. When listing the nodes of a cluster, you may see that a worker node is not in the "Ready" state, which is a good indicator that it's not available to handle the

workload. In the output of the `get nodes` command, you can see that the node named `worker-1` is in the “NotReady” state:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
control-plane	Ready	control-plane	4d20h	v1.33.2
worker-1	NotReady	<none>	4d20h	v1.33.2
worker-2	Ready	<none>	4d20h	v1.33.2

The “NotReady” state means that the node is unused and will accumulate operational costs without actually scheduling workload. There might be a variety of reasons why the node entered this state. The following list shows the most common reasons:

- Insufficient resources: The node may be low on memory or disk space.
- Issues with the kubelet process: The process may have crashed or stopped on the node. Therefore, it cannot communicate with the API server running on any of the control plane nodes anymore.
- Issues with kube-proxy: The Pod running kube-proxy is responsible for network communication from within the cluster and from the outside. The Pod transitioned into a nonfunctional state.

SSH into the relevant worker node(s) and start your investigation.

## Checking available resources

A good way to identify the root cause of an unavailable worker node is to look at its details. The `describe node` command renders the section labeled “Conditions”:

```
$ kubectl describe node worker-1
....
```

```

Conditions:
  Type                                Status  LastHeartbeatTime
\
  LastTransitionTime                  Reason
Message
-----
\
  -----
-----
  NetworkUnavailable    False    Thu, 20 Jan 2022 18:12:13
+0000 \
    Thu, 20 Jan 2022 18:12:13 +0000    CalicoIsUp
\
    Calico is running on this node
  MemoryPressure        False    Tue, 25 Jan 2022 15:59:18
+0000 \
    Thu, 20 Jan 2022 18:11:47 +0000
KubeletHasSufficientMemory \
    kubelet has sufficient memory available
  DiskPressure          False    Tue, 25 Jan 2022 15:59:18
+0000 \
    Thu, 20 Jan 2022 18:11:47 +0000
KubeletHasNoDiskPressure \
    kubelet has no disk pressure
  PIDPressure           False    Tue, 25 Jan 2022 15:59:18
+0000 \
    Thu, 20 Jan 2022 18:11:47 +0000
KubeletHasSufficientPID \
    kubelet has sufficient PID available
  Ready                 True     Tue, 25 Jan 2022 15:59:18
+0000 \
    Thu, 20 Jan 2022 18:12:07 +0000    KubeletReady
\
    kubelet is posting ready status. AppArmor enabled
...

```

The table contains information about the resources available to the node, as well as an indication of other services like networking. See if any of the resource types render the status `True` or `Unknown`, which means that there's an issue with the particular resource. You



can further troubleshoot unavailable resources with a system-level command.

To check on memory and the number of processes running, use the `top` command:

```
$ top
top - 18:45:09 up 1 day,  2:21,  1 user,  load average:
0.13, 0.13, 0.15
Tasks: 116 total,   3 running,  70 sleeping,   0 stopped,
0 zombie
%Cpu(s):  1.5 us,   0.8 sy,   0.0 ni, 97.7 id,   0.0 wa,   0.0
hi,   0.0 si,   0.0 st
KiB Mem : 1008552 total,   134660 free,   264604 used,
609288 buff/cache
KiB Swap:          0 total,          0 free,          0 used.
611248 avail Mem
...
```

To check on the available disk space, use the command `df`:

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            480M     0  480M   0% /dev
tmpfs           99M   1.0M   98M   2% /run
/dev/sda1       39G   2.7G   37G   7% /
tmpfs           493M     0  493M   0% /dev/shm
tmpfs           5.0M     0   5.0M   0% /run/lock
tmpfs           493M     0  493M   0% /sys/fs/cgroup
vagrant         1.9T  252G   1.6T  14% /vagrant
tmpfs           99M     0   99M   0% /run/user/1000
```

## Checking the kubelet process

Some conditions rendered by the `describe node` command mention the kubelet process. If you look at the `Message` column, you might get an idea if the kubelet process is running properly. To troubleshoot a misbehaving kubelet process, run the following `systemctl` command:

```

$ systemctl status kubelet
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/lib/systemd/system/kubelet.service;
   enabled; \
   vendor preset: enabled)
   Drop-In: /etc/systemd/system/kubelet.service.d
            └─10-kubeadm.conf
   Active: active (running) since Thu 2022-01-20 18:11:41
   UTC; 5 days ago
     Docs: https://kubernetes.io/docs/home/
   Main PID: 6537 (kubelet)
     Tasks: 15 (limit: 1151)
    CGroup: /system.slice/kubelet.service
            └─6537 /usr/bin/kubelet \
               --bootstrap-
   kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf \
               --kubeconfig=/etc/kubernetes/kubelet.conf \
               --config=/var/lib/kubelet/config.yaml --network-
lines 1-10/10

```

The most important information in the output is the value of the `Active` attribute. If it says something other than “active (running),” then you will need to dig deeper. Use `journalctl` to take a look at the log files of the process:

```

$ journalctl -u kubelet.service
-- Logs begin at Thu 2022-01-20 18:10:41 UTC, end at
Tue 2022-01-25 18:44:05 UTC. --
Jan 20 18:11:31 worker-1 systemd[1]: Started kubelet: The
Kubernetes Node Agent.
Jan 20 18:11:31 worker-1 systemd[1]: kubelet.service:
Current command vanished \
from the unit file, execution of the command list won't be
resumed.
Jan 20 18:11:31 worker-1 systemd[1]: Stopping kubelet: The
Kubernetes
Node Agent...
Jan 20 18:11:31 worker-1 systemd[1]: Stopped kubelet: The
Kubernetes Node Agent.
Jan 20 18:11:31 worker-1 systemd[1]: Started kubelet: The

```

```
Kubernetes Node Agent.  
....
```

You will want to restart the process once you have identified the issue in the logs and fixed it:

```
$ systemctl restart kubelet
```

## Checking certificate validity

Sometimes, the certificate used by the kubelet can expire. Make sure that the values for the attributes `Issuer` and `Not After` are correct:

```
$ openssl x509 -in /var/lib/kubelet/pki/kubelet.crt -text  
Certificate:  
  Data:  
    Version: 3 (0x2)  
    Serial Number: 2 (0x2)  
    Signature Algorithm: sha256WithRSAEncryption  
    Issuer: CN = worker-1-ca@1642702301  
    Validity  
      Not Before: Jan 20 17:11:41 2022 GMT  
      Not After : Jan 20 17:11:41 2023 GMT  
    Subject: CN = worker-1@1642702301  
    ...
```

For a quick rundown of all certificates in the cluster's Public Key Infrastructure (PKI), you can use the following command.

```
$ kubeadm certs check-expiration  
[check-expiration] Reading configuration from the  
cluster...  
[check-expiration] FYI: You can look at this config file  
with 'kubectl -n \n kube-system get cm kubeadm-config -o yaml'
```

CERTIFICATE	EXPIRES
-------------	---------

RESIDUAL TIME	...				
admin.conf		Aug 31, 2026 14:28 UTC		364d	
...					
apiserver		Aug 31, 2026 14:28 UTC		364d	
...					
apiserver-etcd-client		Aug 31, 2026 14:28 UTC		364d	
...					
apiserver-kubelet-client		Aug 31, 2026 14:28 UTC		364d	
...					
controller-manager.conf		Aug 31, 2026 14:28 UTC		364d	
...					
etcd-healthcheck-client		Aug 31, 2026 14:28 UTC		364d	
...					
etcd-peer		Aug 31, 2026 14:28 UTC		364d	
...					
etcd-server		Aug 31, 2026 14:28 UTC		364d	
...					
front-proxy-client		Aug 31, 2026 14:28 UTC		364d	
...					
scheduler.conf		Aug 31, 2026 14:28 UTC		364d	
...					
super-admin.conf		Aug 31, 2026 14:28 UTC		364d	
...					

CERTIFICATE AUTHORITY	EXPIRES	RESIDUAL TIME	...
ca	Aug 29, 2035 14:28 UTC	9y	
...			
etcd-ca	Aug 29, 2035 14:28 UTC	9y	
...			
front-proxy-ca	Aug 29, 2035 14:28 UTC	9y	
...			

You can renew all certificates necessary to run the control plane with the following command:

```
$ kubeadm certs renew all
```

You must restart the kube-apiserver, kube-controller-manager, kube-scheduler and etcd, so that they can use the new certificates.

## Checking the kube-proxy Pod

The kube-proxy components run in a set of dedicated Pods in the namespace `kube-system`. You can clearly identify the Pods by their naming prefix `kube-proxy` and the appended hash. Verify if any of the Pods states a different status than “Running.” Each of the kube-proxy Pods runs on a dedicated worker node. You can add the `-o wide` option to render the node the Pod is running on in a new column:

```
$ kubectl get pods -n kube-system
NAME                                READY   STATUS    RESTARTS   AGE
...
kube-proxy-csrww                    1/1     Running   0           4d22h
kube-proxy-fjd48                    1/1     Running   0           4d22h
kube-proxy-tvf52                    1/1     Running   0           4d22h
```

Take a look at the event log for kube-proxy Pods that seem to have an issue. The following command describes the Pod named `kube-proxy-csrww`. In addition, you might find more information in the event log of the corresponding DaemonSet:

```
$ kubectl describe pod kube-proxy-csrww -n kube-system
$ kubectl describe daemonset kube-proxy -n kube-system
```

The logs may come in handy as well. You will be able to check the logs only for the kube-proxy Pod that runs on the specific worker node:

```
$ kubectl describe pod kube-proxy-csrww -n kube-system |
grep Node:
Node:                                worker-1/10.0.2.15
$ kubectl logs kube-proxy-csrww -n kube-system
```

## Summary

Troubleshooting Kubernetes clusters and nodes requires a deep understanding of component interactions, systematic investigation skills, and familiarity with various debugging tools. The scenarios covered in this chapter represent common real-world issues you'll encounter when managing Kubernetes infrastructure. By mastering these troubleshooting techniques, you'll be able to quickly diagnose and resolve cluster-level problems, minimizing downtime and maintaining service reliability.

## Exam Essentials

### *Master quick node diagnostics and recovery*

In the exam, you must rapidly identify why nodes are "NotReady" and fix them within minutes. Memorize this sequence: `kubectl get nodes`, `kubectl describe node <node-name>`, check the "Conditions" section for specific issues (MemoryPressure, DiskPressure, PIDPressure, Ready). Know how to SSH into nodes and use `systemctl status kubelet`, `systemctl restart kubelet`, and `journalctl -u kubelet | tail -50` to diagnose and fix kubelet issues.

### *Know system Pod locations and recovery commands*

You must instantly recognize and fix control plane component failures. Remember that static pods (API server, controller-manager, scheduler, etcd) have their manifests in the directory `/etc/kubernetes/manifests` and their logs accessible via `kubectl logs <component>-<node-name> -n kube-system`. For crashed components, know that editing the manifest file directly will trigger the kubelet to restart the Pod automatically.

## *Fix Pod scheduling issues*

Understand how node issues prevent Pod scheduling and know the quick fixes. When Pods are “Pending”, check for node taints (`kubectl describe node | grep Taint`), verify node capacity (`kubectl describe node | grep -A5 "Allocated resources"`), and look for cordoned nodes (`kubectl get nodes | grep SchedulingDisabled`). Master these recovery commands: `kubectl uncordon <node>` to enable scheduling, `kubectl taint nodes <node> <taint-key>-` to remove taints (note the minus sign), and `kubectl drain <node> --ignore-daemonsets --delete-emptydir-data` for node maintenance.

## **Sample Exercises**

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

1. Your team reports that applications are not scaling properly. Upon investigation, you notice one of the worker nodes is having issues and new Pods are not being scheduled there.

This exercise assumes that you run a cluster with 3 nodes: one control-plane node, and two worker nodes. The names of the worker node are `worker-node-1` and `worker-node-2`.

Navigate to the directory *app-a/ch22/troubleshooting-worker-node* of the checked-out GitHub repository [bmuschko/cka-study-guide](#). Transfer the file *setup.sh* to the `worker-node-2`. SSH into the host machine of `worker-node-2` node. Execute the script. For proper execution of the script, make sure that you have `kubectl` installed on

`worker-node-2` beforehand and can authenticate with the API server of the control-plane node.

Identify why `worker-node-2` is not accepting new Pods. Fix all issues preventing the node from functioning properly. Ensure the node can accept new Pod scheduling. Verify the fix by running a test Pod on that specific node.

2. The cluster is experiencing issues with Pod scheduling. New deployments are creating ReplicaSets, but Pods remain in "Pending" state despite having sufficient resources on nodes.

This exercise assumes that you run a cluster with at least one control-plane node, and one worker node. The name of the control-plane node is `control-plane`.

Navigate to the directory *app-a/ch22/troubleshooting-control-plane-node* of the checked-out GitHub repository *[bmuschko/cka-study-guide](#)*. Transfer the file *setup.sh* to the `control-plane` node. SSH into the host machine of `control-plane`. Execute the script.

Create a Deployment named `test-app` with the container image `nginx:1.29.1` with 3 replicas.

Identify which cluster component is failing on the `control-plane` node. Diagnose the root cause of the failure. Fix the component to restore proper cluster operation. Verify that Pod scheduling is working again.



# **Appendix A. Answers to Review Questions**

---

## Chapter 4, Cluster Installation and Upgrade

1. List the cluster nodes using the following command. You should see

```
$ kubectl get nodes
NAME                STATUS    ROLES    AGE
VERSION
minikube             Ready     control-plane  2m20s
v1.32.2
minikube-m02         Ready     <none>        2m10s
v1.32.2
minikube-m03         Ready     <none>        2m3s
v1.32.2
minikube-m04         Ready     <none>        116s
v1.32.2
```

Schedule the Pod using the following imperative command.

```
$ kubectl run nginx --image=nginx:1.27.4-alpine
```

You can identify the node the Pod is running, as shown below. The Pod is running on the node named `minikube-m02`.

```
$ kubectl get pod nginx -o
jsonpath='{.spec.nodeName}'
minikube-m02
```

Use the `kubectl drain` command to evict all Pods on a node. Here, the node name is `minikube-m02`. You'll need to use the `--ignore-daemonsets` flag to only delete Pods that are not managed by a `DaemonSet`, and the `--force` flag to delete Pods without a controller.

```
$ kubectl drain minikube-m02 --ignore-daemonsets --
```

```
force
evicting pod default/nginx
pod/nginx evicted
node/minikube-m02 drained
```

2. The solution to this sample exercise requires a lot of manual steps. The following commands do not render their output.

Open an interactive shell to the control plane node using Vagrant:

```
$ vagrant ssh kube-control-plane
```

Upgrade `kubeadm` to version 1.32.2 and apply it:

```
$ sudo apt-mark unhold kubeadm && sudo apt-get
update && sudo apt-get \
  install -y kubeadm=1.32.2-1.1 && sudo apt-mark
hold kubeadm
$ sudo kubeadm upgrade apply v1.32.2
```

Drain the node, upgrade the kubelet and `kubect1`, restart the kubelet, and uncordon the node:

```
$ kubect1 drain kube-control-plane --ignore-
daemonsets
$ sudo apt-get update && sudo apt-get install -y \
  --allow-change-held-packages kubelet=1.32.2-1.1
kubect1=1.32.2-1.1
$ sudo systemctl daemon-reload
$ sudo systemctl restart kubelet
$ kubect1 uncordon kube-control-plane
```

The version of the node should now say v1.32.2. Exit the node:

```
$ kubect1 get nodes
$ exit
```

Open an interactive shell to the worker node using Vagrant.  
Repeat all of the following steps for the other worker nodes:

```
$ vagrant ssh kube-worker-1
```

Upgrade `kubeadm` to version 1.32.2 and apply it to the node:

```
$ sudo apt-get update && sudo apt-get install -y \
  --allow-change-held-packages kubeadm=1.32.2-1.1
$ sudo kubeadm upgrade node
```

Drain the node, upgrade the kubelet and `kubect1`, restart the kubelet, and uncordon the node:

```
$ kubect1 drain kube-worker-1 --ignore-daemonsets
$ sudo apt-get update && sudo apt-get install -y \
  --allow-change-held-packages kubelet=1.32.2-1.1
kubect1=1.32.2-1.1
$ sudo systemctl daemon-reload
$ sudo systemctl restart kubelet
$ kubect1 uncordon kube-worker-1
```

The version of the node should now say v1.32.2. Exit the node:

```
$ kubect1 get nodes
$ exit
```

## Chapter 5, Backing Up and Restoring etcd

1. Open an interactive shell to the control plane node using Vagrant.

```
$ vagrant ssh kube-control-plane
```

The etcd Pod runs in the `kube-system` namespace. Identify the Pod by listing all Pods in the namespace. You should find the Pod named `etcd-kube-control-plane`.

```
$ kubectl get pods -n kube-system
```

The etcd Pod only runs a single container. You can render the details of the Pod using the `kubectl get` or `kubectl describe` command. To select only the container image, use the following JSONPath.

```
$ kubectl get pod etcd-kube-control-plane -n kube-system \
  -o jsonpath="{.spec.containers[0].image}"
```

The container uses the image `registry.k8s.io/etcd:3.5.16-0`. The etcd version used is 3.5.16. Write the version to the file named `etcd-version.txt`. You can view all released etcd versions in the corresponding [GitHub repository](#).

```
$ echo "3.5.16" > etcd-version.txt
```

Exit the node:

```
$ exit
```

2. The solution to this sample exercise requires a lot of manual steps. The following commands do not render their output.

Open an interactive shell to the control plane node using Vagrant.

```
$ vagrant ssh kube-control-plane
```

Determine the parameters of the Pod `etcd-kube-control-plane` by describing it. Use the correct parameter values to create a snapshot file:

```
$ kubectl describe pod etcd-kube-control-plane -n kube-system
$ sudo ETCDCTL_API=3 etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key snapshot
save /opt/etcd.bak
```

Restore the backup from the snapshot file. Edit the `etcd` YAML manifest and change the value of `spec.volumes.hostPath.path` for the volume named `etcd-data` to `/var/bak`:

```
$ sudo ETCDCTL_API=3 etcdctl --data-dir=/var/bak
snapshot restore \
/opt/etcd.bak
$ sudo vim /etc/kubernetes/manifests/etcd.yaml
```

After a short while, the Pod `etcd-kube-control-plane` should transition back into the “Running” status. Exit the node:

```
$ kubectl get pod etcd-kube-control-plane -n kube-system
$ exit
```



## Chapter 6, Authentication, Authorization, and Admission Control

1. Use the `kubectl create clusterrole` command to create the Role imperatively:

```
$ kubectl create clusterrole service-view --  
verb=get,list \  
--resource=services
```

If you'd rather start with the YAML file, use content shown in the file *service-view-clusterrole.yaml*:

```
apiVersion: rbac.authorization.k8s.io/v1  
kind: ClusterRole  
metadata:  
  name: service-view  
rules:  
- apiGroups: [""]  
  resources: ["services"]  
  verbs: ["get", "list"]
```

Create the ClusterRole from the YAML file:

```
$ kubectl apply -f service-view-clusterrole.yaml
```

Use the `kubectl create rolebinding` command to create the RoleBinding imperatively:

```
$ kubectl create rolebinding ellasmith-service-view  
--user=ellasmith \  
--clusterrole=service-view -n development
```

The declarative approach of the RoleBinding could look like the one in the file *ellasmith-service-view-rolebinding.yaml*:

```
apiVersion: rbac.authorization.k8s.io/v1  
kind: RoleBinding
```



```

metadata:
  name: ellasmith-service-view
  namespace: development
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: service-view
subjects:
- kind: User
  name: ellasmith
  apiGroup: rbac.authorization.k8s.io

```

Create the RoleBinding from the YAML file:

```
$ kubectl apply -f ellasmith-service-view-
rolebinding.yaml
```

Use the `kubectl create clusterrole` command to create the ClusterRole imperatively:

```
$ kubectl create clusterrole combined \
  --aggregation-
rule="rbac.cka.cncf.com/aggregate=true"
```

If you'd rather start with the YAML file, use content shown in the file *combined-clusterrole.yaml*:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: combined
aggregationRule:
  clusterRoleSelectors:
  - matchLabels:
      rbac.cka.cncf.com/aggregate: "true"
rules: []

```

Create the ClusterRole from the YAML file:

```
$ kubectl apply -f combined-clusterrole.yaml
```

Rendering the rules shows an empty list as no ClusterRoles were selected by label selection:

```
$ kubectl describe clusterrole combined
Name:          combined
Labels:        <none>
Annotations:   <none>
PolicyRule:
  Resources    Non-Resource URLs  Resource Names
Verbs
-----
-
```

Use the `kubectl create clusterrole` command to create the ClusterRole imperatively. The command does not provide an option for adding a label. Therefore, generating the YAML manifest using the dry-run option is a good start:

```
$ kubectl create clusterrole deployment-modify \
  --verb=create,delete,patch,update --
resource=deployments \
  --dry-run=client -o yaml > deployment-modify-
clusterrole.yaml
```

Modify the content shown in the file *deployment-modify-clusterrole.yaml*:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: deployment-modify
  labels:
    rbac.cka.cncf.com/aggregate: "true"
rules:
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["create", "delete", "patch", "update"]
```

Create the ClusterRole from the YAML file:

```
$ kubectl apply -f deployment-modify-
```

**clusterrole.yaml**

Rendering the rules for the Clusterrole combined lists the rules defined by the ClusterRole deployment-modify:

```
$ kubectl describe clusterrole combined
Name:          combined
Labels:        <none>
Annotations:   <none>
PolicyRule:
  Resources            Non-Resource URLs  Resource
Names  Verbs
-----
-  -----
    deployments.apps  []
[create \
 delete patch update]
```

Listing Services is allowed for the user ellasmith in the namespace development. This is taken care by RoleBinding ellasmith-service-view:

```
$ kubectl auth can-i list services --as=ellasmith --
namespace=development
yes
```

Write the output the file *list-services-ellasmith.txt*.

```
$ echo 'yes' > list-services-ellasmith.txt
```

Watching Deployments is not allowed for the user ellasmith in the namespace production. The ClusterRole named deployment-modify only allows the verbs create, delete, patch, and update.

```
$ kubectl auth can-i watch deployments --
as=ellasmith --namespace=production
no
```

Write the output the file *watch-deployments-ellasmith.txt*.

```
$ echo 'no' > watch-deployments-ellasmith.txt
```

Deployments are allowed to be created, deleted, patched, and updated.

2. First, create the namespace named `apps`. Then, we'll create the ServiceAccount:

```
$ kubectl create namespace apps
$ kubectl create serviceaccount api-access -n apps
```

Alternatively, you can use the declarative approach. Create the namespace from the definition in the file `apps-`

```
namespace.yaml:
  apiVersion: v1
  kind: Namespace
  metadata:
    name: apps
```

Create the namespace from the YAML file:

```
$ kubectl apply -f apps-namespace.yaml
```

Create a new YAML file called `api-serviceaccount.yaml` with the following contents:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: api-access
  namespace: apps
```

Run the `apply` command to instantiate the ServiceAccount from the YAML file:

```
$ kubectl create -f api-serviceaccount.yaml
```

Use the `create clusterrole` command to create the ClusterRole imperatively:

```
$ kubectl create clusterrole api-clusterrole --  
verb=watch,list,get \  
--resource=pods
```

If you'd rather start with the YAML file, use content shown in the file `api-clusterrole.yaml`:

```
apiVersion: rbac.authorization.k8s.io/v1  
kind: ClusterRole  
metadata:  
  name: api-clusterrole  
rules:  
- apiGroups: [""]  
  resources: ["pods"]  
  verbs: ["watch", "list", "get"]
```

Create the ClusterRole from the YAML file:

```
$ kubectl apply -f api-clusterrole.yaml
```

Use the `create clusterrolebinding` command to create the ClusterRoleBinding imperatively.

```
$ kubectl create clusterrolebinding api-  
clusterrolebinding \  
--serviceaccount=apps:api-access --  
clusterrole=api-clusterrole
```

The declarative approach of the ClusterRoleBinding could look like the one in the file `api-`

`clusterrolebinding.yaml`:

```
apiVersion: rbac.authorization.k8s.io/v1  
kind: ClusterRoleBinding  
metadata:  
  name: api-clusterrolebinding  
roleRef:  
  apiGroup: rbac.authorization.k8s.io
```

```

    kind: ClusterRole
    name: api-clusterrole
  subjects:
  - apiGroup: ""
    kind: ServiceAccount
    name: api-access
    namespace: apps

```

Create the ClusterRoleBinding from the YAML file:

```
$ kubectl apply -f api-clusterrolebinding.yaml
```

Execute the `run` command to create the Pods in the different namespaces. You will need to create the namespace `rm` before you can instantiate the Pod disposable:

```

$ kubectl run operator --image=nginx:1.21.1 --
restart=Never \
  --port=80 --serviceaccount=api-access -n apps
$ kubectl create namespace rm
$ kubectl run disposable --image=nginx:1.21.1 --
restart=Never \
  -n rm

```

The following YAML manifest shows the `rm` namespace definition stored in the file `rm-namespace.yaml`:

```

apiVersion: v1
kind: Namespace
metadata:
  name: rm

```

The YAML representation of those Pods stored in the file `api-pods.yaml` could look as follows:

```

apiVersion: v1
kind: Pod
metadata:
  name: operator
  namespace: apps

```

```
spec:
  serviceAccountName: api-access
  containers:
  - name: operator
    image: nginx:1.21.1
    ports:
    - containerPort: 80
---
apiVersion: v1
kind: Pod
metadata:
  name: disposable
  namespace: rm
spec:
  containers:
  - name: disposable
    image: nginx:1.21.1
```

Create the namespace and Pods from the YAML files:

```
$ kubectl create -f rm-namespace.yaml
$ kubectl create -f api-pods.yaml
```

Determine the API server endpoint and the Secret access token of the ServiceAccount. You will need this information for making the API calls:

```
$ kubectl config view --minify -o \
  jsonpath='{.clusters[0].cluster.server}'
https://192.168.64.4:8443
$ kubectl get secret $(kubectl get serviceaccount
api-access -n apps \
  -o jsonpath='{.secrets[0].name}') -o
jsonpath='{.data.token}' -n apps \
  | base64 --decode
eyJhbGciOiJSUzI1NiIsImtpZCI6Ii1hOUhI...
```

Open an interactive shell to the Pod named `operator`:

```
$ kubectl exec operator -it -n apps -- /bin/sh
```

Emit API calls for listing all Pods and deleting the Pod disposable living in the rm namespace. You will find that while the list operation is permitted, the delete operation isn't:

```
# curl
https://192.168.64.4:8443/api/v1/namespaces/rm/pods
--header \
"Authorization: Bearer
eyJhbGciOiJSUzI1NiIsImtpZCI6Ii1hOUhI..." \
--insecure
{
  "kind": "PodList",
  "apiVersion": "v1",
  ...
}
# curl -X DELETE
https://192.168.64.4:8443/api/v1/namespaces \
/rm/pods/disposable --header \
"Authorization: Bearer
eyJhbGciOiJSUzI1NiIsImtpZCI6Ii1hOUhI..." \
--insecure
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

  },
  "status": "Failure",
  "message": "pods \"disposable\" is forbidden: User
\
\"system:serviceaccount:apps:api-access\" cannot
delete \
resource \"pods\" in
API group \"\" in the namespace \"rm\"",
  "reason": "Forbidden",
  "details": {
    "name": "disposable",
    "kind": "pods"
  },
  "code": 403
}
```





## Chapter 7, Custom Resource Definitions (CRDs) and Operators

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/mongod
bcommunity.\
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 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 8, Helm and Kustomize

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...
70.3.0           ...
```

Install the latest version of the chart `kube-prometheus-stack`:

```
$ helm install prometheus prometheus-community/kube-
prometheus-stack
NAME: prometheus
LAST DEPLOYED: Wed Mar 26 14:14:53 2025
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           2025-03-26  ...
```

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.

localhost

 Prometheus Alerts Graph Status ▾ Help



☐ Use local time

☐ Enable query history

☒ Enable autocomplete

☒ Enable highlighting

☒ Enable linter

Q

Expression (press Shift+Enter for newlines)

Execute

Table

Graph

<

Evaluation time

>

No data queried yet

Remove Panel

Add Panel

Uninstall the chart with the following command:

```
$ helm uninstall prometheus
release "prometheus" uninstalled
```

2. Navigate to the folder containing the `manifests` directory. Create all objects contained in the `manifests` directory using the recursive `apply` command:

```
$ kubectl apply -f manifests/ -R
configmap/logs-config created
pod/nginx created
```

Modify the value of the key `dir` in the file `configmap.yaml` using an editor. Then update the live object of the ConfigMap using the following command:

```
$ vim manifests/configmap.yaml
$ kubectl apply -f manifests/configmap.yaml
configmap/logs-config configured
```

Delete all objects that have been created from the `manifests` directory using the recursive `delete` command:

```
$ kubectl delete -f manifests/ -R
configmap "logs-config" deleted
pod "nginx" deleted
```

Create the file `kustomization.yaml`. It should define the common attribute for the namespace and reference the resource with the file `pod.yaml`. The following YAML file shows its contents:

```
namespace: t012
resources:
- pod.yaml
```



Run the following `kustomize` command to render the transformed manifest as console output:

```
$ kubectl kustomize ./
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: t012
spec:
  containers:
  - image: nginx:1.21.1
    name: nginx
```

## Chapter 9, 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 j43
```

Create the Pod:

```
$ kubectl run nginx --image=nginx:1.17.10 --port=80  
--namespace=j43
```

Alternatively, you can use the declarative approach. Create a new YAML manifest in the file called *namespace.yaml* with the following contents:

```
apiVersion: v1  
kind: Namespace  
metadata:  
  name: j43
```

Create the namespace from the YAML manifest:

```
$ kubectl apply -f 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=j43
```

You can use the command-line option `-o wide` to retrieve the IP address of the Pod:

```
$ kubectl get pod nginx --namespace=j43 -o wide
```

The same information is available if you query for the Pod details:

```
$ kubectl describe pod nginx --namespace=j43 | 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 j43 -- wget -O- 10.1.0.66:80
```

To download the logs, use a simple `logs` command:

```
$ kubectl logs nginx --namespace=j43
```

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=j43
```

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=j43
```

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=j43
```

Use the `exec` command to open an interactive shell to the container:

```
$ kubectl exec -it nginx --namespace=j43 -- /bin/sh
# ls -l
```

2. 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=j43
```

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=j43
```

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=j43
```

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=j43
```

You can describe the Pod events by grepping for the term:

```
$ kubectl describe pod loop --namespace=j43 | grep -C 10 Events:
```

You can simply delete the namespace, which will delete all objects within the namespace:

```
$ kubectl delete namespace j43
```

## Chapter 10, 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 11, Deployments and ReplicaSets

1. Execute the command to create the Deployment object. You will see an error message.

```
$ kubectl apply -f fix-me-deployment.yaml
The Deployment "nginx-deployment" is invalid:
spec.template.metadata.labels:\
  Invalid value: map[string]string{"app":"nginx"}:
`selector` does not \
match template `labels`
```

The label selector does not match with the assigned labels in the Pod template. Change the manifest so that they line up. The manifest shown below uses `run: server` under `spec.selector.matchLabels` and `spec.template.metadata.labels`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      run: server
  template:
    metadata:
      labels:
        run: server
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

You should now be able to create the object.

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

2. 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
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>

## Chapter 12, Scaling Workloads

1. Create a new YAML file named *hello-world-deployment.yaml* with the contents shown below:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  replicas: 3
  selector:
    matchLabels:
      run: hello-world
  template:
    metadata:
      labels:
        run: hello-world
    spec:
      containers:
      - image: bmuschko/nodejs-hello-world:1.0.0
        name: hello-world
```

Create the Deployment object from the manifest file:

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

Ensure the correct number of replicas. You should see 3 replicas ready.

```
$ kubectl get deployment hello-world
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
hello-world	3/3	3	3	32s

Modify the YAML file *hello-world-deployment.yaml* by editing it with an editor. Change the value of the attribute `spec.replicas` to 8. The following command uses vim.



```
$ vim hello-world-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  replicas: 8
...
```

Apply the changes to the existing object.

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

Ensure the correct number of replicas. You should see 8 replicas ready.

```
$ kubectl get deployment hello-world
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
hello-world	8/8	8	8	56s

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 deployment nginx
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%
8	\	3
3	2m19s	

## Chapter 13, 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
v1.32.2
minikube-m02       Ready    <none>      44s
v1.32.2
minikube-m03       Ready    <none>      26s
v1.32.2
```

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 14, Pod Scheduling

1. The following solution demonstrates the use of a multi-node Minikube cluster. Listing the nodes renders the following output.

```
$ kubectl get nodes
NAME                STATUS    ROLES    AGE
VERSION
minikube            Ready    control-plane    3m16s
v1.32.2
minikube-m02       Ready    <none>          3m6s
v1.32.2
minikube-m03       Ready    <none>          2m59s
v1.32.2
```

Let's assign the label `color=green` to the node `minikube-m02` and the label `color=red` to the node `minikube-m03`.

```
$ kubectl label nodes minikube-m02 color=green
node/minikube-m02 labeled
$ kubectl label nodes minikube-m03 color=red
node/minikube-m03 labeled
```

You can render the assign labels for all nodes using the `--show-labels` command line option.

```
$ kubectl get nodes --show-labels
NAME                STATUS    ROLES    AGE
VERSION    LABELS
minikube            Ready    control-plane    27m
v1.32.2    ...
minikube-m02       Ready    <none>          27m
v1.32.2    ...,color=green,...
minikube-m03       Ready    <none>          26m
v1.32.2    ...,color=red,...
```

The Pod manifest could look as such:

```
apiVersion: v1
kind: Pod
metadata:
  name: app
spec:
  nodeSelector:
    color: green
  containers:
  - name: nginx
    image: nginx:1.27.1
```

Create the Pod and inspect the assigned node.

```
$ kubectl apply -f pod.yaml
pod/app created
$ kubectl get pod app -o=wide
NAME      READY   STATUS    RESTARTS   AGE   IP
NODE
app       1/1     Running   0           21s   10.244.1.2
10.244.1.2 minikube-m02   ...
```

Change Pod manifest to use node affinity to schedule it on node minikube-m02 or minikube-m03. The resulting YAML definition would look like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: app
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: color
            operator: In
            values:
            - green
            - red
```

```
containers:
- name: nginx
  image: nginx:1.27.1
```

First delete the Pod, then recreate it. The Pod should be scheduled on one of nodes.

```
$ kubectl delete -f pod.yaml
pod "app" deleted
$ kubectl apply -f pod.yaml
pod/app created
$ kubectl get pod app -o=wide
NAME      READY   STATUS    RESTARTS   AGE   IP
NODE
app       1/1     Running   0          12s   10.244.1.3
minikube-m02    ...
```

2. The following solution demonstrates the use of a multi-node minikube cluster. Listing the nodes renders the following output.

```
$ kubectl get nodes
NAME                STATUS    ROLES          AGE
VERSION
minikube            Ready    control-plane  3m16s
v1.32.2
minikube-m02        Ready    <none>         3m6s
v1.32.2
minikube-m03        Ready    <none>         2m59s
v1.32.2
```

The Pod definition in the file *pod.yaml* could look as follows.

```
apiVersion: v1
kind: Pod
metadata:
  name: app
spec:
  containers:
  - name: nginx
    image: nginx:1.27.1
```

Create the Pod and inspect the assigned node.

```
$ kubectl apply -f pod.yaml
pod/app created
$ kubectl get pod app -o=wide
NAME      READY   STATUS    RESTARTS   AGE   IP
NODE
app       1/1     Running   0           89s   10.244.2.2
minikube-m03    ...
```

Add the taint to the node. In this case, the node is called minikube-m03.

```
$ kubectl taint nodes minikube-m03
exclusive=yes:NoExecute
node/minikube-m03 tainted
```

The taint causes the Pod to be evicted.

```
$ kubectl get pods
No resources found in default namespace.
```

Change the Pod manifest and add the toleration.

```
apiVersion: v1
kind: Pod
metadata:
  name: app
spec:
  tolerations:
  - key: "exclusive"
    operator: "Equal"
    value: "yes"
    effect: "NoExecute"
  containers:
  - name: nginx
    image: nginx:1.27.1
```

With the toleration, the Pod will be allowed to be scheduled on the node minikube-m03 again.

```
$ kubectl apply -f pod.yaml
pod/app created
$ kubectl get pod app -o=wide
NAME      READY   STATUS    RESTARTS   AGE   IP
NODE
app       1/1     Running   0           9s    10.244.2.3
minikube-m03    ...
```

Remove the taint from the node. The Pod will continue to run on the node.

```
$ kubectl taint nodes minikube-m03 exclusive-
node/minikube-m03 untainted
$ kubectl get pod app -o=wide
NAME      READY   STATUS    RESTARTS   AGE   IP
NODE
app       1/1     Running   0           37s    10.244.2.3
minikube-m03    ...
```

## Chapter 15, 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
```

## Chapter 16, Persistent Volumes

1. 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 apply -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 apply -f logs-pvc.yaml
persistentvolumeclaim/logs-pvc created
$ kubectl get pvc
NAME          STATUS    VOLUME    CAPACITY   ACCESS
MODES        STORAGECLASS  ...
logs-pvc     Bound       logs-pv    5Gi        RWO,ROX
...

```

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
pod "nginx" deleted
$ 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

```

2. Create the file *db-pvc.yaml* to define the PersistentVolumeClaim. The following YAML manifest shows its contents:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: db-pvc
  namespace: persistence
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: local-path
  resources:
    requests:
      storage: 128Mi
```

Create the PersistentVolumeClaim object from the YAML manifest:

```
$ kubectl apply -f db-pvc.yaml
persistentvolumeclaim/db-pvc created
```

Check the status of the PersistentVolumeClaim object:

```
$ kubectl get pvc -n persistence
NAME          STATUS    VOLUME   CAPACITY   ACCESS MODES
STORAGECLASS  ...
db-pvc        Pending
local-path    ...
```

The corresponding PersistentVolume object should not have been created yet.

```
$ kubectl get pv -n persistence
No resources found
```

Create the basic YAML manifest using the `--dry-run` command-line option:

```
$ kubectl run app-consuming-pvc --
image=alpine:3.21.3 -n persistence \
--dry-run=client --restart=Never -o yaml \
-- /bin/sh -c "while true; do sleep 60; done;" \
> alpine-pod.yaml
```

Now, edit the file *alpine-pod.yaml* and bind the PersistentVolumeClaim to it:

```
apiVersion: v1
kind: Pod
metadata:
  name: app-consuming-pvc
  namespace: persistence
spec:
  volumes:
    - name: app-storage
      persistentVolumeClaim:
        claimName: db-pvc
  containers:
    - image: alpine:3.21.3
      name: app
      command: ["/bin/sh"]
      args: ["-c", "while true; do sleep 60; done;"]
      volumeMounts:
        - mountPath: "/mnt/data"
          name: app-storage
  restartPolicy: Never
```

Create the Pod using the following command:

```
$ kubectl apply -f alpine-pod.yaml
pod/app-consuming-pvc created
```

Ensure that the Pod transitions into the "Running" status:

```
$ kubectl get pods -n persistence
```

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

The PersistentVolume object has been provisioned dynamically.

```
$ kubectl get pv -n persistence
```

NAME	CAPACITY
ACCESS MODES	...
pvc-af39068d-0cc2-4625-8a56-7b5207b79ace	128Mi
RWO	...

Use the `exec` command to open an interactive shell to the Pod, and create a file in the mounted directory:

```
$ kubectl exec app-consuming-pvc -n persistence -it  
-- /bin/sh  
# cd /mnt/data  
# touch test.db  
# ls  
test.db  
# exit
```

The file will be persisted in the PersistentVolume.



## Chapter 17, Services

1. Create the Deployment definition in the file *deployment.yaml*. Ensure to expose two container ports, 8080 and 9090.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: webapp
        image: nginxdemos/hello:0.4-plain-text
        ports:
        - containerPort: 80
        - containerPort: 9090
```

Define the Service in the file *service.yaml*. The most important part is the port mapping. Assign the static node port 30080 only to the `web` port mapping.

```
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
spec:
  type: NodePort
  selector:
    app: webapp
  ports:
  - name: web
```

```

    port: 80
    targetPort: 80
    nodePort: 30080
-   name: metrics
    port: 9090
    targetPort: 9090

```

Create the application Deployment and Service objects.

```

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

```

Listing the Deployment and Service objects are going to look similar as shown below:

```

$ kubectl get deployment,service
NAME                                     READY   UP-TO-DATE
AVAILABLE   AGE
deployment.apps/webapp                 3/3     3           3
6s

NAME                                     TYPE          CLUSTER-IP
EXTERNAL-IP   \
PORT(S)          AGE
service/webapp-service   NodePort      10.111.80.190
<none>          \
80:30080/TCP,9090:31231/TCP   6s

```

To test accessibility from your host machine, you can configure port forwarding for the Service. Here, we are mapping the host port 9091 to the Service port 80.

```

$ kubectl port-forward service/webapp-service
9091:80 &
Forwarding from 127.0.0.1:9091 -> 80
Forwarding from [::1]:9091 -> 80
Handling connection for 9091

```

You can now access the Service from your host machine using `curl`.

```
$ curl localhost:9091
Server address: 127.0.0.1:80
Server name: webapp-6dc64898b-hw116
Date: 21/Aug/2025:02:09:01 +0000
URI: /
Request ID: b5e0f6324ad7bac1b513dba9d2c1cf64
```

2. Create the database Deployment definition in the file *database-deployment.yaml*. Set the environment variables and expose the container port 3306.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: database
spec:
  replicas: 1
  selector:
    matchLabels:
      app: database
  template:
    metadata:
      labels:
        app: database
    spec:
      containers:
      - name: mysql
        image: mysql:9.4.0
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: secretpass
        - name: MYSQL_DATABASE
          value: myapp
        ports:
        - containerPort: 3306
```

Define the database Service in the file *database-service.yaml*. Map the port 3306 to the target port 3306.

```
apiVersion: v1
kind: Service
metadata:
  name: database-service
```

```
spec:
  type: ClusterIP
  selector:
    app: database
  ports:
  - port: 3306
    targetPort: 3306
```

Define the frontend Deployment in the file *frontend-deployment.yaml*. Ensure to define the command in the correct format.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: busybox:1.35
        command:
        - sh
        - -c
        - "while true; do nc -zv database-service
3306; sleep 5; done"
```

Create the application Deployment and Service objects.

```
$ kubectl apply -f database-deployment.yaml
$ kubectl apply -f database-service.yaml
$ kubectl apply -f frontend-deployment.yaml
```

The frontend Pods can be selected by the label `app=frontend`. The logs of both Pods should indicate that the connection to the database Service could be established. An error message in the logs indicate a configuration issue.

```
$ kubectl logs -l app=frontend  
database-service (10.101.125.103:3306) open  
database-service (10.101.125.103:3306) open
```

## Chapter 18, Ingresses

1. Define the namespace in the file *namespace.yaml*:

```
apiVersion: v1
kind: Namespace
metadata:
  name: webapp
```

The file *deployment.yaml* shows the frontend and api Deployments.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  namespace: webapp
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: nginx:1.29.1-alpine
        ports:
        - containerPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
  namespace: webapp
spec:
  replicas: 2
  selector:
```

```

    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
      - name: api
        image: httpd:2.4.65-alpine
        ports:
        - containerPort: 80

```

The file *services.yaml* defines the frontend and api Services.

```

apiVersion: v1
kind: Service
metadata:
  name: frontend-service
  namespace: webapp
spec:
  selector:
    app: frontend
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
  type: ClusterIP

```

---

```

apiVersion: v1
kind: Service
metadata:
  name: api-service
  namespace: webapp
spec:
  selector:
    app: api
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
  type: ClusterIP

```

Lastly, define the Ingress in the file *ingress.yaml*.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: webapp-ingress
  namespace: webapp
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
    - host: app.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: frontend-service
                port:
                  number: 80
          - path: /app
            pathType: Prefix
            backend:
              service:
                name: frontend-service
                port:
                  number: 80
          - path: /api
            pathType: Prefix
            backend:
              service:
                name: api-service
                port:
                  number: 80
```

Create all objects from the YAML manifests.

```
$ kubectl apply -f namespace.yaml
$ kubectl apply -f deployments.yaml
$ kubectl apply -f services.yaml
$ kubectl apply -f ingress.yaml
```



Map `localhost` to the hostname used by the Ingress object to make it accessible for local machine.

```
sudo vim /etc/hosts
127.0.0.1          app.example.com
```

Inspect the Ingress object. Make sure that the value in the column "ADDRESS" has been populated.

```
$ kubectl get ingress webapp-ingress -n webapp
NAME                CLASS    HOSTS                ADDRESS
PORTS    AGE
webapp-ingress      nginx    app.example.com
192.168.49.2    80          2m
```

Test all Ingress endpoints by sending a `curl` request.

```
$ curl -H "Host: app.example.com" http://localhost/
$ curl -H "Host: app.example.com"
http://localhost/app
$ curl -H "Host: app.example.com"
http://localhost/api
```

2. We'll use imperative commands to set up the backend objects to speed up the process. First, we'll need to create the namespace `production-apps`.

```
$ kubectl create namespace production-apps
```

Then we'll create the blue and green Deployment objects.

```
$ kubectl create deployment app-blue \
  --image=nginxdemos/hello:0.3-plain-text \
  --replicas=3 -n production-apps
$ kubectl create deployment app-green \
  --image=nginxdemos/hello:0.4-plain-text \
  --replicas=3 -n production-apps
```

Next up, create the Service objects using the `expose deployment` command.

```
$ kubectl expose deployment app-blue --name=app-blue-svc \
  --port=80 -n production-apps
$ kubectl expose deployment app-green --name=app-green-svc \
  --port=80 -n production-apps
```

Define the main Ingress in the file *blue-ingress.yaml*.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-main
  namespace: production-apps
spec:
  ingressClassName: nginx
  rules:
  - host: app.production.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: app-blue-svc
            port:
              number: 80
```

Define the canary Ingress in the file *green-ingress.yaml*.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-canary-weight
  namespace: production-apps
  annotations:
    nginx.ingress.kubernetes.io/canary: "true"
    nginx.ingress.kubernetes.io/canary-weight: "20"
spec:
  ingressClassName: nginx
```

```

rules:
- host: app.production.com
  http:
    paths:
    - path: /
      pathType: Prefix
      backend:
        service:
          name: app-green-svc
          port:
            number: 80

```

Create the Ingress objects from the YAML manifests.

```

$ kubectl apply -f blue-ingress.yaml
$ kubectl apply -f green-ingress.yaml

```

You can map `localhost` to the hostname used by the Ingress objects to make them accessible for local machine.

```

sudo vim /etc/hosts
127.0.0.1          app.production.com

```

Inspect the Ingress objects. Make sure that the value in the column "ADDRESS" has been populated.

```

$ kubectl get ingresses -n production-apps

```

NAME	CLASS	HOSTS
app-canary-weight	nginx	app.production.com
192.168.49.2 80	...	
app-main	nginx	app.production.com
192.168.49.2 80	...	

To see the traffic routing distribution in action, run a `curl` command a `for` loop. You will see that most requests are routed to the "blue" Pod, and a subset of the requests are routed to the "green" Pods.

```
for i in {1..10}; do curl -s -H "Host:
app.production.com" \
    http://localhost | grep -o "Server name:.*"; done
```

## Chapter 19, Gateway API

1. You can find the installation instructions for the NGINX Gateway Fabric controller in the [official documentation](#). Follow the instructions of your preferred installation method.

Create the application Deployment and Service objects. You can use file *setup.yaml* to set them up at once. The solution provided in this file creates content on the container mount path at */usr/share/nginx/html/web* for the *web-app* Pods, and content on the container mount path at */usr/local/apache2/htdocs/api* for the *api-app* Pods.

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

Ensure that the objects are available.

```
$ kubectl get deployments,services,pods
```

Configure Gateway API resources. Define the Gateway in the file *gateway.yaml*:

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: main-gateway
spec:
  gatewayClassName: nginx
  listeners:
  - name: http
    port: 80
    protocol: HTTP
    hostname: example.local
```

Define the Gateway in the file *httproute.yaml*:

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
```

```

    name: app-routes
spec:
  parentRefs:
  - name: main-gateway
  hostnames:
  - example.local
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /web
      backendRefs:
      - name: web-app
        port: 80
  - matches:
    - path:
        type: PathPrefix
        value: /api
      backendRefs:
      - name: api-app
        port: 80

```

Create the Gateway objects with the following command:

```

$ kubectl apply -f gateway.yaml
$ kubectl apply -f httproute.yaml

```

Check if Gateway received the assignment of an IP address.  
This may take a few minutes.

```

$ kubectl get gateway main-gateway -o \
  jsonpath='{.status.addresses[0].value}'

```

The following command verifies that the status for the  
Accepted condition of the HTTPRoute renders True.

```

$ kubectl get httproute app-routes -o \
  jsonpath='{.status.parents[*].conditions[?
  (@.type=="Accepted")].status}'

```

Port forward to the Gateway:

```
$ kubectl port-forward svc/main-gateway-nginx
8080:80 &
```

You should be able to access the Gateway using `curl`.

```
$ curl -H "Host: example.local"
http://localhost:8080/web/
$ curl -H "Host: example.local"
http://localhost:8080/api/
```

2. Create the namespaces and the applications running in them from the file *setup.yaml*.

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

Configure the Gateway in the file *gateway.yaml*. Allow that routing is accepted from all namespaces.

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: gateway
  namespace: production
spec:
  gatewayClassName: nginx
  listeners:
  - name: http
    port: 80
    protocol: HTTP
    hostname: example.com
    allowedRoutes:
      namespaces:
        from: All
```

Create the HTTPRoute in the `production` namespace in the file *production-route.yaml*.

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
```

```

    name: prod-route
    namespace: production
spec:
  parentRefs:
  - name: gateway
  hostnames:
  - example.com
  rules:
  - matches:
    - path:
        type: Exact
        value: /app
    backendRefs:
    - name: prod-web
      port: 80
    filters:
    - type: URLRewrite
      urlRewrite:
        path:
          type: ReplaceFullPath
          replaceFullPath: /
    - type: RequestHeaderModifier
      requestHeaderModifier:
        set:
        - name: X-Environment
          value: production

```

Create the HTTPRoute in the staging namespace in the file *staging-route.yaml*.

```

apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: staging-route
  namespace: staging
spec:
  parentRefs:
  - name: gateway
    namespace: production
  hostnames:
  - example.com
  rules:
  - matches:

```



```

- path:
    type: PathPrefix
    value: /staging
backendRefs:
- name: staging-web
  port: 80
filters:
- type: URLRewrite
  urlRewrite:
    path:
      type: ReplacePrefixMatch
      replacePrefixMatch: /
- type: RequestHeaderModifier
  requestHeaderModifier:
    set:
    - name: X-Environment
      value: staging

```

Create the ReferenceGrant for cross-namespace access in the file *reference-grant.yaml*.

```

apiVersion: gateway.networking.k8s.io/v1beta1
kind: ReferenceGrant
metadata:
  name: allow-staging-to-gateway
  namespace: production
spec:
  from:
  - group: gateway.networking.k8s.io
    kind: HTTPRoute
    namespace: staging
  to:
  - group: gateway.networking.k8s.io
    kind: Gateway
    name: gateway

```

Create all HTTPRoute objects.

```

$ kubectl apply -f gateway.yaml
$ kubectl apply -f production-route.yaml
$ kubectl apply -f staging-route.yaml
$ kubectl apply -f reference-grant.yaml

```

Port forward to the Gateway:

```
$ kubectl port-forward -n production svc/gateway-nginx 8080:80 &
```

You should be able to access the Gateway using `curl`.

```
$ curl -H "Host: example.com"
http://localhost:8080/app
$ curl -H "Host: example.com"
http://localhost:8080/staging
```

## Chapter 20, Network Policies

1. Create the NetworkPolicy for the Pods with the label assignment `app=alpha-app` in the file *alpha-app-policy.yaml*. The NetworkPolicy allows egress traffic to Pods in the namespace with the assigned label `team=beta`. It denies all other egress traffic except DNS signified by the port 53.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: alpha-app-policy
  namespace: team-alpha
spec:
  podSelector:
    matchLabels:
      app: alpha-app
  policyTypes:
  - Egress
  egress:
  - to:
    - namespaceSelector:
        matchLabels:
          team: beta
  - to:
    ports:
    - protocol: UDP
      port: 53
    - protocol: TCP
      port: 53
```

The NetworkPolicy defined in the file *beta-app-policy.yaml* looks a little simpler. It allows the `beta-app` Pod to receive traffic from `team-alpha` namespace on port 80.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: beta-app-policy
```

```

    namespace: team-beta
spec:
  podSelector:
    matchLabels:
      app: beta-app
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          team: alpha
    ports:
    - protocol: TCP
      port: 80

```

Create both NetworkPolicy objects.

```

$ kubectl apply -f alpha-app-policy.yaml
$ kubectl apply -f beta-app-policy.yaml

```

Use the `curl` command to test the `alpha-app` Pod to make a call the `beta-app` Service in the `team-beta` namespace. The command should succeed.

```

$ kubectl exec -it alpha-app -n team-alpha -- \
  curl -v --connect-timeout 2 \
  http://beta-app.team-beta.svc.cluster.local:8080

```

Use the `curl` command to test the `alpha-app` Pod to an address on the internet. The command should fail.

```

$ kubectl exec -it alpha-app -n team-alpha -- \
  curl -v --connect-timeout 2 http://google.com

```

Use the `curl` command to test the connectivity to the `beta-app` Service from a temporary Pod in the same namespace. The command should fail.

```
$ kubectl run test-pod --image=alpine/curl:8.14.1 -n
team-beta --rm -it \
  --restart=Never -- curl -v --connect-timeout 2
http://beta-app:8080
```

2. Network policies are additive which means you can apply them in any order. For make the changes more understandable, you may want to start with the deny policy first. The file *default-deny-all.yaml* shows the contents of the NetworkPolicy that denies all ingress communication within the `production` namespace.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
  namespace: production
spec:
  podSelector: {}
  policyTypes:
    - Ingress
```

Create the NetworkPolicy for the database component in the file *database-policy.yaml*. The NetworkPolicy applies to the Pods with the label assignment `tier=database` and only allows ingress traffic from Pods with the label assignment `tier=backend` on port 6379.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: database-policy
  namespace: production
spec:
  podSelector:
    matchLabels:
      tier: database
  policyTypes:
    - Ingress
  ingress:
    - from:
```

```

- podSelector:
    matchLabels:
        tier: backend
ports:
- protocol: TCP
  port: 6379

```

Create the NetworkPolicy for the backend component in the file *backend-policy.yaml*. This policy applies to Pods with the label assignment `tier=backend`. It allows incoming traffic from the frontend microservice and outgoing traffic to the database microservice.

```

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: backend-policy
  namespace: production
spec:
  podSelector:
    matchLabels:
        tier: backend
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - podSelector:
            matchLabels:
                tier: frontend
      ports:
        - protocol: TCP
          port: 80
  egress:
    - to:
        - podSelector:
            matchLabels:
                tier: database
      ports:
        - protocol: TCP
          port: 6379
    - to:

```

```
ports:
- protocol: UDP
  port: 53
- protocol: TCP
  port: 53
```

Create all NetworkPolicy objects.

```
$ kubectl apply -f default-deny-all.yaml
$ kubectl apply -f database-policy.yaml
$ kubectl apply -f backend-policy.yaml
```

Use the `curl` command to test the `frontend` Pod to test connectivity to the `database` Pod. We are using the `telnet` protocol here to ensure that can connect to the database port. The command should fail.

```
$ kubectl exec -it frontend -n production -- curl -v
--connect-timeout 2 \
telnet://database:3306
```

Use the `curl` command to test the `frontend` Pod to test connectivity to the `backend` Pod. The command should succeed.

```
$ kubectl exec -it frontend -n production -- curl -v
--connect-timeout 2 \
http://backend:80
```

Use the `curl` command to test the `backend` Pod to test connectivity to the `database` Pod. The command should succeed. Similar to the first command, we are using the `telnet` protocol.

```
$ kubectl exec -it backend -n production -- curl -v --
connect-timeout 2 \
telnet://database:3306
```





## Chapter 21, Troubleshooting Applications

1. First, create the Pod using the given YAML content:

```
$ kubectl apply -f setup.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 use 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 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-mhvjj	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
deployment.apps/web-app	2/2	2
10m		

NAME	READY	AGE	DESIRED
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 with 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
```

### 3. 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 22, Troubleshooting Clusters

1. List all nodes in the cluster. You will see that the node `worker-2` shows the status `NotReady, SchedulingDisabled`.

```
$ kubectl get nodes
```

NAME	STATUS	ROLES
control-plane	Ready	
control-plane	23m v1.33.2	
worker-1	Ready	<none>
worker-1	23m v1.33.2	
worker-2	NotReady, SchedulingDisabled	<none>
worker-2	23m v1.33.2	

Check why scheduling is disabled. The command renders a taint that indicates that workload cannot be scheduled on the node.

```
$ kubectl describe node worker-node-2 | grep -i taint
```

Check the node for any other conditions. You will see the message "Kubelet stopped posting node status."

```
$ kubectl describe node worker-node-2 | grep -A10 Conditions
```

Uncordon the node to allow scheduling on the node again.

```
$ kubectl uncordon worker-node-2
```

Check if the kubelet is healthy on the worker node. SSH into the node and check kubelet status. You will see that the kubelet process is inactive.

```
$ ssh worker-node-2
$ sudo systemctl status kubelet
```

Restart the kubelet process and verify that it is running properly.

```
$ sudo systemctl restart kubelet
$ sudo systemctl status kubelet
```

Check the node status. All nodes should show "Ready" without "SchedulingDisabled".

```
$ kubectl get nodes
```

Run a test pod specifically on `worker-node-2`. Verify that the Pod is running on the correct node.

```
$ kubectl run test-pod --image=nginx:1.29.1 --
overrides=\
'{"spec":{"nodeSelector":
{"kubernetes.io/hostname":"worker-node-2"}}}'
$ kubectl get pod test-pod -o wide
```

2. Create the Deployment with the parameters provided in the instructions.

```
$ kubectl create deployment test-app --
image=nginx:1.29.1 --replicas=3
```

Check the status of the Pods controlled by the ReplicaSet. All Pods will have the status "Pending".

```
$ kubectl get pods
```

NAME	READY	STATUS
test-app-5d4d5b6c7b-h2x4m	0/1	Pending
test-app-5d4d5b6c7b-k9p3n	0/1	Pending



```
2m
test-app-5d4d5b6c7b-x7v2q    0/1    Pending    0
2m
```

Check why Pods are pending. Describe a pending Pod to see its events. The output indicates “no scheduler found”.

```
$ kubectl describe pod test-app-5d4d5b6c7b-h2x4m |
tail -10
```

Check the Pods in the `kube-system` namespace. The scheduler is in `ImagePullBackOff` state.

```
$ kubectl get pods -n kube-system | grep -E \
"scheduler|controller|apiserver|etcd"
```

Investigate the scheduler Pod. The scheduler is trying to use a non-existent image version.

```
$ kubectl describe pod kube-scheduler-master-node -n
kube-system \
| grep -A5 Events
```

Fix the scheduler manifest. SSH into the control-plane node. Edit the scheduler configuration manifest. Fix the scheduler manifest by changing the image tag to the version of control-plane node, e.g. `v1.33.2`.

```
$ ssh control-plane
$ sudo vi /etc/kubernetes/manifests/kube-
scheduler.yaml
```

Wait a moment for the kubelet to detect the change and restart the scheduler Pod. Once the scheduler Pod transitions back to normal operations, you can check for the `test-app` Pods again.

```
$ kubectl get pods
```

NAME	READY	STATUS	
RESTARTS      AGE			
test-app-5d4d5b6c7b-h2x4m 8m	1/1	Running	0
test-app-5d4d5b6c7b-k9p3n 8m	1/1	Running	0
test-app-5d4d5b6c7b-x7v2q 8m	1/1	Running	0

All Pods should be back to the “Running” status.

# 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.

## **Cluster Architecture, Installation &**

## Configuration

Exam Objective	Chapter	Reference Documentation	Tutorial
Manage role based access control (RBAC)	Chapter 6	Using RBAC Authorization, Role Based Access Control Good Practices	N/A
Prepare underlying infrastructure for installing a Kubernetes cluster	Chapter 4	N/A	N/A
Create and manage Kubernetes clusters using kubeadm	Chapter 4	kubeadm	Creating a cluster with kubeadm, Upgrading kubeadm clusters
Manage the lifecycle of Kubernetes clusters	Chapter 5	etcd	Backing up an etcd cluster, Restoring an etcd cluster

<b>Exam Objective</b>	<b>Chapter</b>	<b>Reference Documentation</b>	<b>Tutorial</b>
Implement and configure a highly-available control plane	Chapter 4	kubeadm, Options for Highly Available Topology	Creating Highly Available Clusters with kubeadm
Use Helm and Kustomize to install cluster components	Chapter 8	Helm, Kustomize	Declarative Management of Kubernetes Objects Using Kustomize, Managing Secrets using Kustomize
Understand extension interfaces (CNI, CSI, CRI, etc.)	Chapter 4	Infrastructure extensions, Compute, Storage, and Networking Extensions	N/A
Understand CRDs, install and configure operators	Chapter 7	Custom Resources, Operator pattern	Use Custom Resources

## Workloads & Scheduling

Exam Objective	Chapter	Reference Documentation	Tutorial
Understand application deployments and how to perform rolling update and rollbacks	Chapter 9, Chapter 11	Pods, Deployments, ReplicaSet	Performing a Rolling Update
Use ConfigMaps and Secrets to configure applications	Chapter 10	ConfigMaps, Secrets	Configure a Pod to Use a ConfigMap, Managing Secrets using kubectl, Managing Secrets using Configuration File
Configure workload autoscaling	Chapter 12	Scaling a Deployment	Running Multiple Instances of Your App, Horizontal Pod Autoscaling
Understand the primitives	Chapter 11	Kubernetes Self-Healing,	N/A

<b>Exam Objective</b>	<b>Chapter</b>	<b>Reference Documentation</b>	<b>Tutorial</b>
used to create robust, self-healing, application deployments		ReplicaSet	
Configure Pod admission and scheduling (limits, node affinity, etc.)	Chapter 13, Chapter 14	Resource Management for Pods and Containers, Limit Ranges, Resource Quotas, Assigning Pods to Nodes, Taints and Tolerations, Pod Topology Spread Constraints	Assign Pods to Nodes using Node Affinity

## Storage

Exam Objective	Chapter	Reference Documentation	Tutorial
Implement storage classes and dynamic volume provisioning	Chapter 16	Persistent Volumes, Storage Classes	N/A
Configure volume types, access modes and reclaim policies	Chapter 16	Persistent Volumes	N/A
Manage persistent volumes and persistent volume claims	Chapter 16	PersistentVolumeClaims	Configure to Use a Persistent for Storag



## Servicing & Networking

Exam Objective	Chapter	Reference Documentation	Tutorial
Understand connectivity between Pods	Chapter 17	DNS for Services and Pods	N/A
Define and enforce Network Policies	Chapter 20	Network Policies	Declare Network Policy
Understand ClusterIP, NodePort, LoadBalancer service types and endpoints	Chapter 17	Service	Connecting Applications with Services
Use the Gateway API to manage Ingress traffic	Chapter 19	Gateway API	N/A
Know how to use Ingress controllers and Ingress resources	Chapter 18	Ingress, Ingress Controllers	Set up Ingress on Minikube with the NGINX Ingress Controller

<b>Exam Objective</b>	<b>Chapter</b>	<b>Reference Documentation</b>	<b>Tutorial</b>
Understand and use CoreDNS	Chapter 17	CoreDNS	Using CoreDNS for Service Discovery

# Troubleshooting

Exam Objective	Chapter	Reference Documentation	Tutorial
Troubleshoot clusters and nodes	Chapter 22	N/A	Troubleshooting Clusters
Troubleshoot cluster components	Chapter 22	N/A	Troubleshooting Clusters
Monitor cluster and application resource usage	Chapter 21	Metrics Server, kubectl top	N/A
Manage and evaluate container output streams	Chapter 21	Pod and container logs	N/A
Troubleshoot services and networking	Chapter 21	N/A	Debug Services



## 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 and an avid open source advocate. He is a KubeStronaut, holding all Kubernetes certifications.

Software projects sometimes feel like climbing a mountain. In his free time, Ben loves hiking **Colorado's 14ers** and enjoys conquering long-distance trails.