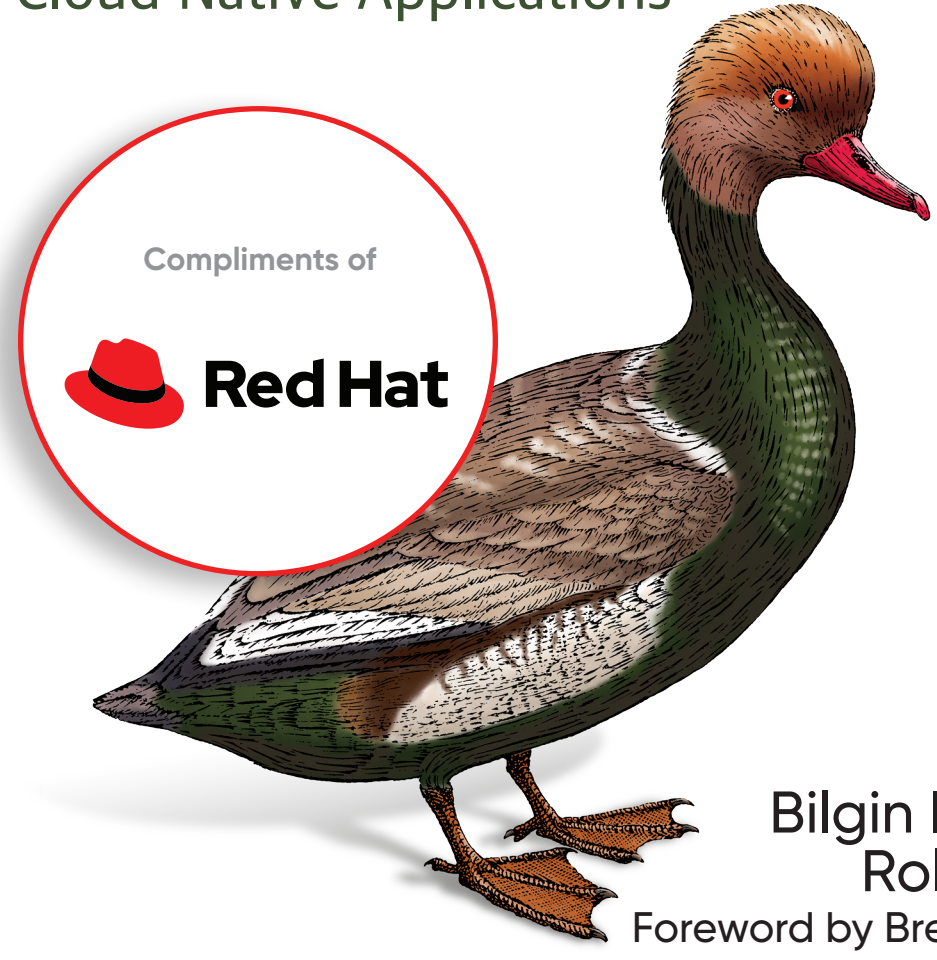
O'REILLY®

Second Edition

Kubernetes Patterns

Reusable Elements for Designing Cloud Native Applications



Bilgin Ibryam &
Roland Huß

Foreword by Brendan Burns

O'REILLY®

Kubernetes Patterns

The way developers design, build, and run software has changed significantly with the evolution of microservices and containers. These modern architectures offer new distributed primitives that require a different set of practices than many developers, tech leads, and architects are accustomed to. With this focused guide, Bilgin Ibryam and Roland Huß provide common reusable patterns and principles for designing and implementing cloud native applications on Kubernetes.

Each pattern includes a description of the problem and a Kubernetes-specific solution. All patterns are backed by and demonstrated with concrete code examples. This updated edition is ideal for developers and architects who are familiar with basic Kubernetes concepts but want to learn how to solve common cloud native challenges with proven design patterns.

You'll explore:

- Foundational patterns covering core principles and practices for building and running container-based cloud native applications
- Behavioral patterns for managing various types of container and platform interactions
- Structural patterns for organizing containers to address specific use cases
- Configuration patterns that provide insight into how application configurations can be handled in Kubernetes
- Security patterns for hardening applications running on Kubernetes and making them more secure
- Advanced patterns covering more complex topics such as operators, autoscaling, and in-cluster image builds

"Bilgin and Roland have written a wonderful, incredibly informative, and intensely useful book."

—**Grady Booch**
Chief Scientist for Software Engineering, IBM; Coauthor, *Unified Modeling Language*

"An updated set of patterns to enable developers to take full advantage of the capabilities and features found in Kubernetes."

—**Andrew Block**
Distinguished Architect, Red Hat

Bilgin Ibryam is a principal product manager at Diagrid, where he leads the company's product strategy.

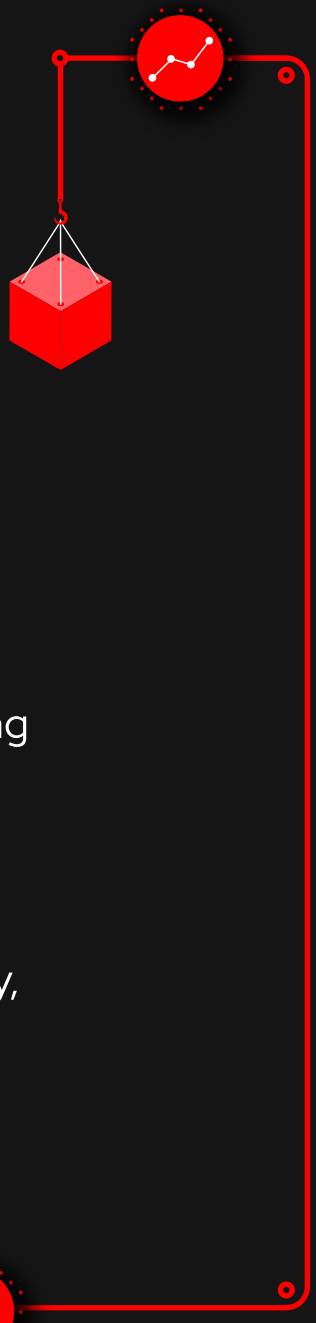
Dr. Roland Huß is a senior principal software engineer at Red Hat and the architect of OpenShift Serverless.

KUBERNETES

Twitter: @oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

ISBN: 978-1-098-13988-9

Red Hat



Build **Smarter.**
Ship **Faster.**

To make the most of the cloud, IT needs to approach applications in new ways. Cloud-native development means packaging with containers, adopting modern architectures, and using agile techniques.

Red Hat can help you arrange your people, processes, and technologies to build, deploy, and run cloud-ready applications anywhere they are needed. Discover how with **cloud-native development solutions.**

SECOND EDITION

Kubernetes Patterns

Reusable Elements for Designing Cloud Native Applications

Bilgin Ibryam and Roland Huß

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Kubernetes Patterns
by Bilgin Ibryam and Roland Huß

Printed in the United States of America.

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

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

Acquisitions Editor: John Devins
Development Editor: Rita Fernando
Production Editor: Beth Kelly
Copyeditor: Piper Editorial Consulting, LLC
Proofreader: Sharon Wilkey

Indexer: Judy McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

April 2019: First Edition
March 2023: Second Edition

Revision History for the Second Edition
2023-03-25: First Release
2023-05-26: Second Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098131685* for release details.

978-1-098-13988-9

[LSI]

Table of Contents

Part VI. Advanced Patterns

Foreword

When Craig, Joe, and I started Kubernetes nearly eight years ago, I think we all recognized its power to transform the way the world developed and delivered software. I don't think we knew, or even hoped to believe, how quickly this transformation would come. Kubernetes is now the foundation for the development of portable, reliable systems spanning the major public clouds, private clouds, and bare-metal environments. However, even as Kubernetes has become ubiquitous to the point where you can spin up a cluster in the cloud in less than five minutes, it is still far less obvious to determine where to go once you have created that cluster. It is fantastic that we have seen such significant strides forward in the operationalization of Kubernetes itself, but it is only a part of the solution. It is the foundation on which applications will be built, and it provides a large library of APIs and tools for building these applications, but it does little to provide the application architect or developer with any hints or guidance for how these various pieces can be combined into a complete, reliable system that satisfies their business needs and goals.

Although the necessary perspective and experience for what to do with your Kubernetes cluster can be achieved through past experience with similar systems, or via trial and error, this is expensive both in terms of time and the quality of systems delivered to our end users. When you are starting to deliver mission-critical services on top of a system like Kubernetes, learning your way via trial and error simply takes too much time and results in very real problems of downtime and disruption.

This then is why Bilgin and Roland's book is so valuable. *Kubernetes Patterns* enables you to learn from the previous experience that we have encoded into the APIs and tools that make up Kubernetes. Kubernetes is the by-product of the community's experience building and delivering many different, reliable distributed systems in a variety of different environments. Each object and capability added to Kubernetes represents a foundational tool that has been designed and purpose-built to solve a specific need for the software designer. This book explains how the concepts in Kubernetes solve real-world problems and how to adapt and use these concepts to build the system that you are working on today.

In developing Kubernetes, we always said that our North Star was making the development of distributed systems a CS 101 exercise. If we have managed to achieve that goal successfully, it is books like this one that are the textbooks for such a class. Bilgin and Roland have captured the essential tools of the Kubernetes developer and distilled them into segments that are easy to approach and consume. As you finish this book, you will become aware not just of the components available to you in Kubernetes but also the "why" and "how" of building systems with those components.

— Brendan Burns
Cofounder, Kubernetes

Preface

With the mainstream adoption of microservices and containers in recent years, the way we design, develop, and run software has changed radically. Today's applications are optimized for availability, scalability, and speed-to-market. Driven by these new requirements, today's modern applications require a different set of patterns and practices. This book aims to help developers discover and learn about the most common patterns for creating cloud native applications with Kubernetes. First, let's take a brief look at the two primary ingredients of this book: Kubernetes and design patterns.

Kubernetes

Kubernetes is a container orchestration platform. The origin of Kubernetes lies somewhere in the Google data centers where Google's internal container orchestration platform, Borg, was born. Google used Borg for many years to run its applications. In 2014, Google decided to transfer its experience with Borg into a new open source project called "Kubernetes" (Greek for "helmsman" or "pilot"). In 2015, it became the first project donated to the newly founded Cloud Native Computing Foundation (CNCF).

From the start, Kubernetes gained a whole community of users, and the number of contributors grew incredibly fast. Today, Kubernetes is considered one of the most popular projects on GitHub. It is fair to claim that Kubernetes is the most commonly used and feature-rich container orchestration platform. Kubernetes also forms the foundation of other platforms built on top of it. The most prominent of those Platform-as-a-Service systems is Red Hat OpenShift, which provides various additional capabilities to Kubernetes. These are only some reasons we chose Kubernetes as the reference platform for the cloud native patterns in this book.

This book assumes you have some basic knowledge of Kubernetes. In Chapter 1, we recapitulate the core Kubernetes concepts and lay the foundation for the following patterns.

Design Patterns

The concept of *design patterns* dates back to the 1970s and is from the field of architecture. Christopher Alexander, an architect and system theorist, and his team published the groundbreaking *A Pattern Language* (Oxford University Press) in 1977, which describes architectural patterns for creating towns, buildings, and other construction projects. Sometime later, this idea was adopted by the newly formed software industry. The most famous book in this area is *Design Patterns—Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides—the Gang of Four (Addison-Wesley). When we talk about the famous Singleton, Factories, or Delegation patterns, it's because of this defining work. Many other great pattern books have been written since then for various fields with different levels of granularity, like *Enterprise Integration Patterns* by Gregor Hohpe and Bobby Woolf (Addison-Wesley) or *Patterns of Enterprise Application Architecture* by Martin Fowler (Addison-Wesley).

In short, a *pattern* describes a *repeatable solution to a problem*.[1] This definition works for the patterns we describe in this book, except that we probably don't have as much variability in our solutions. A pattern is different from a recipe because instead of giving step-by-step instructions to solve a problem, it provides a blueprint for solving a whole class of similar problems. For example, the Alexandrian pattern *Beer Hall* describes how public drinking halls should be constructed where "strangers and friends are drinking companions" and not "anchors of the lonely." All halls built after this pattern look different but share common characteristics, such as open alcoves for groups of four to eight and a place where a hundred people can meet to enjoy beverages, music, and other activities.

However, a pattern does more than provide a solution. It is also about forming a language. The patterns in this book form a dense, noun-centric language in which each pattern carries a unique *name*. When this language is established, these names automatically evoke similar mental representations when people speak about these patterns. For example, when we talk about a table, anyone speaking English assumes we are talking about a piece of wood with four legs and a top on which you can put things. The same thing happens in software engineering when discussing a "factory." In an object-oriented programming language context, we immediately associate with a "factory" an object that produces other objects. Because we immediately know the solution behind the pattern, we can move on to tackle yet-unsolved problems.

1 Alexander and his team defined the original meaning in the context of architecture as follows: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." (*A Pattern Language*, Christopher Alexander et al., 1977.)

There are also other characteristics of a pattern language. For example, patterns are interconnected and can overlap so that they cover most of the problem space. Also, as already laid out in the original *A Pattern Language*, patterns have a different level of granularity and scope. More general patterns cover an extensive problem space and provide rough guidance on how to solve the problem. Granular patterns have a very concrete solution proposal but are less widely applicable. This book contains all sorts of patterns, and many patterns reference other patterns or may even include other patterns as part of the solution.

Another feature of patterns is that they follow a rigid format. However, each author defines a different form; unfortunately, there is no common standard for how patterns should be laid out. Martin Fowler gives an excellent overview of the formats used for pattern languages at "Writing Software Patterns".

How This Book Is Structured

We chose a simple pattern format for this book. We do not follow any particular pattern description language. For each pattern, we use the following structure:

Name
: Each pattern carries a name, which is also the chapter's title. The name is the center of the pattern's language.

Problem
: This section gives the broader context and describes the pattern space in detail.

Solution
: This section shows how the pattern solves the problem in a Kubernetes-specific way. This section also contains cross-references to other patterns that are either related or part of the given pattern.

Discussion
: This section includes a discussion about the advantages and disadvantages of the solution for the given context.

More Information
: This final section contains additional information sources related to the pattern.

We organized the patterns in this book as follows:

- Part I, "Foundational Patterns", covers the core concepts of Kubernetes. These are the underlying principles and practices for building container-based cloud native applications.

- Part II, "Behavioral Patterns", describes patterns that build on top of foundational patterns and add the runtime aspect concepts of managing various types of containers.
- Part III, "Structural Patterns", contains patterns related to organizing containers within a *Pod*, which is the atom of the Kubernetes platform.
- Part IV, "Configuration Patterns", gives insight into the various ways application configuration can be handled in Kubernetes. These are granular patterns, including concrete recipes for connecting applications to their configuration.
- Part V, "Security Patterns", addresses various security concerns that arise when an application is containerized and deployed on Kubernetes.
- Part VI, "Advanced Patterns", is a collection of advanced concepts, such as how the platform itself can be extended or how to build container images directly within the cluster.

Depending on the context, the same pattern might fit into several categories. Every pattern chapter is self-contained; you can read chapters in isolation and in any order.

Who This Book Is For

This book is for *developers* who want to design and develop cloud native applications and use Kubernetes as the platform. It is most suitable for readers who have some basic familiarity with containers and Kubernetes concepts and want to take it to the next level. However, you don't need to know the low-level details of Kubernetes to understand the use cases and patterns. Architects, consultants, and other technical personnel will also benefit from the repeatable patterns described here.

The book is based on use cases and lessons learned from real-world projects. It is an accumulation of best practices and patterns after years of working in this space. We want to help you understand the Kubernetes-first mindset and create better cloud native applications—not reinvent the wheel. It is written in a relaxed style and is similar to a series of essays that can be read independently.

Let's briefly look at what this book is *not*:

- This book is not an introduction to Kubernetes, nor is it a reference manual. We touch on many Kubernetes features and explain them in some detail, but we are focusing on the concepts behind those features. Chapter 1, "Introduction", offers a brief refresher on Kubernetes basics. If you are looking for a comprehensive book on Kubernetes, we highly recommend *Kubernetes in Action* by Marko Lukša (Manning Publications).
- This book is not a step-by-step guide on how to set up a Kubernetes cluster itself. Every example assumes you have Kubernetes up and running. You have several

options for trying out the examples. If you are interested in learning how to set up a Kubernetes cluster, we recommend *Kubernetes: Up and Running* by Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson (O'Reilly).

- This book is not about operating and governing a Kubernetes cluster for other teams. We deliberately skipped administrative and operational aspects of Kubernetes and took a developer-first view into Kubernetes. This book can help operations teams understand how a developer uses Kubernetes, but it is not sufficient for administering and automating a Kubernetes cluster. If you are interested in learning how to operate a Kubernetes cluster, we recommend *Kubernetes Best Practices* by Brendan Burns, Eddie Villalba, Dave Strebel, and Lachlan Evenson (O'Reilly).

What You Will Learn

There's a lot to discover in this book. Some patterns may read like excerpts from a Kubernetes manual at first glance, but upon closer look, you'll see the patterns are presented from a conceptual angle not found in other books on the topic. Other patterns are explained with detailed steps to solve a concrete problem, as in Part IV, "Configuration Patterns". In some chapters, we explain Kubernetes features that don't fit nicely into a pattern definition. Don't get hung up on whether it is a pattern or a feature. In all chapters, we look at the forces involved from the first principles and focus on the use cases, lessons learned, and best practices. That is the valuable part.

Regardless of the pattern granularity, you will learn everything Kubernetes offers for each particular pattern, with plenty of examples to illustrate the concepts. All these examples have been tested, and we tell you how to get the complete source code in "Using Code Examples" on page xix.

What's New in the Second Edition

The Kubernetes ecosystem has continued to grow since the first edition came out four years ago. As a result, there have been many Kubernetes releases, and more tools and patterns for using Kubernetes have become de facto standards.

Fortunately, most of the patterns described in our book have stood the test of time and remain valid. Therefore, we have updated these patterns, added new features up to Kubernetes 1.26, and removed obsolete and deprecated parts. For the most part, only minor changes were necessary, except for Chapter 29, "Elastic Scale", and Chapter 30, "Image Builder", which underwent significant changes due to new developments in these areas.

Additionally, we have included five new patterns and introduced a new category, Part V, "Security Patterns", which addresses a gap in the first edition and provides important security-related patterns for developers.

Our GitHub examples have been updated and extended. And, lastly, we added 50% more content for our readers to enjoy.

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.

As mentioned, patterns form a simple, interconnected language. To emphasize this web of patterns, each pattern is capitalized and set in italics, (e.g., *Sidecar*). When a pattern name is also a Kubernetes core concept (such as *Init Container* or *Controller*), we use this specific formatting only when we directly reference the pattern itself. Where it makes sense, we also interlink pattern chapters for ease of navigation.

We also use the following conventions:

- Everything you can type in a shell or editor is rendered in `constant width font`.
- Kubernetes resource names are always rendered in uppercase (e.g., Pod). If the resource is a combined name like ConfigMap, we keep it like this in favor of the more natural "config map" for clarity and to make it clear that it refers to a Kubernetes concept.
- Sometimes, a Kubernetes resource name is identical to a common concept like "service" or "node." In these cases, we use the resource name format only when referring to the resource itself.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Every pattern is backed with fully executable examples, which you can find on the accompanying web page. You can find the link to each pattern's example in each chapter's "More Information" section.

The "More Information" section also contains links to further information related to the pattern. We keep these lists updated in the example repository.

The source code for all examples in this book is available on GitHub. The repository and the website also have pointers and instructions on how to get a Kubernetes cluster to try out the examples. Please look at the provided resource files when you go through the examples. They contain many valuable comments that will further your understanding of the example code.

Many examples use a REST service called *random-generator* that returns random numbers when called. It is uniquely crafted to play well with the examples in this book. Its source can be found on GitHub as well, and its container image `k8spatterns/random-generator` is hosted on Docker Hub.

We use a JSON path notation to describe resource fields (e.g., `.spec.replicas` points to the `replicas` field of the resource's `spec` section).

If you find an issue in the example code or documentation or have a question, don't hesitate to open a ticket at the GitHub issue tracker. We monitor these GitHub issues and are happy to answer any questions.

We love code contributions too! If you think we can improve our examples, we are happy to hear from you. Just open a GitHub issue or create a pull request, and let's start a conversation.

O'Reilly Online Learning

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

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please 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
800-889-8969 (in the United States or Canada)
707-829-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://www.oreilly.com/about/contact.html

We have a web page for this book where we list errata, examples, and additional information. You can access this page at *https://oreil.ly/kubernetes_patterns-2e*.

For news and information about our books and courses, visit *https://oreilly.com*.

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

Follow us on Twitter: *https://twitter.com/oreillymedia*

Watch us on YouTube: *https://youtube.com/oreillymedia*

Follow the authors on Twitter: *https://twitter.com/bibryam*, *https://twitter.com/ro14nd*

Follow the authors on Mastodon: *https://fosstodon.org/@bilgin*, *https://hachyderm.io/@ro14nd*

Find the authors on GitHub: *https://github.com/bibryam*, *https://github.com/rhuss*

Follow their blogs: *https://www.ofbizian.com*, *https://ro14nd.de*

Acknowledgments

Bilgin is forever grateful to his wonderful wife, Ayshe, for her endless support and patience as he worked on yet another book. He is also thankful for his adorable daughters, Selin and Esin, who always know how to bring a smile to his face. You mean the world to him. Finally, Bilgin would like to thank his fantastic coauthor, Roland, for making this project a reality.

Roland is deeply grateful for his wife Tanja's unwavering support and forbearance throughout the writing process, and he also thanks his son Jakob for his encouragement. Furthermore, Roland wishes to extend special recognition to Bilgin for his exceptional insights and writing, without which the book would not have come to fruition.

Creating two editions of this book was a long multiyear journey, and we want to thank our reviewers who kept us on the right track.

For the first edition, special kudos to Paolo Antinori and Andrea Tarocchi for helping us through the journey. Big thanks to Marko Lukša, Brandon Philips, Michael Hüttermann, Brian Gracely, Andrew Block, Jiri Kremser, Tobias Schneck, and Rick Wagner, who supported us with their expertise and advice. Last but not least, big thanks to our editors Virginia Wilson, John Devins, Katherine Tozer, Christina Edwards, and all the awesome folks at O'Reilly for helping us push this book over the finish line.

Completing the second edition was no easy feat, and we are grateful to all who supported us in finishing it. We extend our thanks to our technical reviewers, Ali Ok, Dávid Šimanský, Zbyněk Roubalík, Erkan Yanar, Christoph Stäbler, Andrew Block, and Adam Kaplan, as well as to our development editor, Rita Fernando, for her patience and encouragement throughout the whole process. Many kudos go out to the O'Reilly production team, especially Beth Kelly, Kim Sandoval, and Judith McConville, for their meticulous attention in finalizing the book.

We want to express a special thank you to Abhishek Koserwal for his tireless and dedicated efforts in Chapter 26, "Access Control". His contributions came at a time when we needed them the most and made an impact.

CHAPTER 1

Introduction

In this introductory chapter, we set the scene for the rest of the book by explaining a few of the core Kubernetes concepts used for designing and implementing cloud native applications. Understanding these new abstractions, and the related principles and patterns from this book, is key to building distributed applications that can be automatable by Kubernetes.

This chapter is not a prerequisite for understanding the patterns described later. Readers familiar with Kubernetes concepts can skip it and jump straight into the pattern category of interest.

The Path to Cloud Native

Microservices is among the most popular architectural styles for creating cloud native applications. They tackle software complexity through modularization of business capabilities and trading development complexity for operational complexity. That is why a key prerequisite for becoming successful with microservices is to create applications that can be operated at scale through Kubernetes.

As part of the microservices movement, there is a tremendous amount of theory, techniques, and supplemental tools for creating microservices from scratch or for splitting monoliths into microservices. Most of these practices are based on *Domain-Driven Design* by Eric Evans (Addison-Wesley) and the concepts of bounded contexts and aggregates. *Bounded contexts* deal with large models by dividing them into different components, and *aggregates* help to further group bounded contexts into modules with defined transaction boundaries. However, in addition to these business domain considerations, for each distributed system—whether it is based on microservices or not—there are also technical concerns around its external structure, and runtime coupling. Containers and container orchestrators such as Kubernetes bring in new

primitives and abstractions to address the concerns of distributed applications, and here we discuss the various options to consider when putting a distributed system into Kubernetes.

Throughout this book, we look at container and platform interactions by treating the containers as black boxes. However, we created this section to emphasize the importance of what goes into containers. Containers and cloud native platforms bring tremendous benefits to your distributed applications, but if all you put into containers is rubbish, you will get distributed rubbish at scale. Figure 1-1 shows the mixture of the skills required for creating good cloud native applications and where Kubernetes patterns fit in.

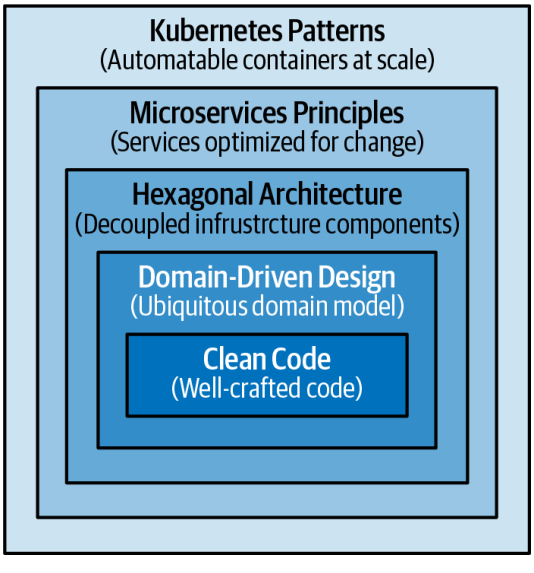


Figure 1-1. The path to cloud native

At a high level, creating good cloud native applications requires familiarity with multiple design techniques:

- At the lowest *code level*, every variable you define, every method you create, and every class you decide to instantiate plays a role in the long-term maintenance of the application. No matter what container technology and orchestration platform you use, the development team and the artifacts they create will have the most impact. It is important to grow developers who strive to write clean code, have the right number of automated tests, constantly refactor to improve code quality, and are guided by Software Craftsmanship principles at heart.
- *Domain-driven design* is about approaching software design from a business perspective with the intention of keeping the architecture as close to the real

world as possible. This approach works best for object-oriented programming languages, but there are also other good ways to model and design software for real-world problems. A model with the right business and transaction boundaries, easy-to-consume interfaces, and rich APIs is the foundation for successful containerization and automation later.

- The *hexagonal architecture* and its variations, such as Onion and Clean architectures, improve the flexibility and maintainability of applications by decoupling the application components and providing standardized interfaces for interacting with them. By decoupling the core business logic of a system from the surrounding infrastructure, hexagonal architecture makes it easier to port the system to different environments or platforms. These architectures complement domain-driven design and help arrange application code with distinct boundaries and externalized infrastructure dependencies.
- The *microservices architectural style* and the twelve-factor app methodology very quickly evolved to become the norm for creating distributed applications and they provide valuable principles and practices for designing changing distributed applications. Applying these principles lets you create implementations that are optimized for scale, resiliency, and pace of change, which are common requirements for any modern software today.
- *Containers* were very quickly adopted as the standard way of packaging and running distributed applications, whether these are microservices or functions. Creating modular, reusable containers that are good cloud native citizens is another fundamental prerequisite. *Cloud native* is a term used to describe principles, patterns, and tools to automate containerized applications at scale. We use *cloud native* interchangeably with *Kubernetes*, which is the most popular open source cloud native platform available today.

In this book, we are not covering clean code, domain-driven design, hexagonal architecture, or microservices. We are focusing only on the patterns and practices addressing the concerns of the container orchestration. But for these patterns to be effective, your application needs to be designed well from the inside by using clean code practices, domain-driven design, hexagonal architecture-like isolation of external dependencies, microservices principles, and other relevant design techniques.

Distributed Primitives

To explain what we mean by new abstractions and primitives, here we compare them with the well-known object-oriented programming (OOP), and Java specifically. In the OOP universe, we have concepts such as class, object, package, inheritance, encapsulation, and polymorphism. Then the Java runtime provides specific features and guarantees on how it manages the lifecycle of our objects and the application as a whole.

The Java language and the Java Virtual Machine (JVM) provide local, in-process building blocks for creating applications. Kubernetes adds an entirely new dimension to this well-known mindset by offering a new set of distributed primitives and runtime for building distributed systems that spread across multiple nodes and processes. With Kubernetes at hand, we don't rely only on the local primitives to implement the whole application behavior.

We still need to use the object-oriented building blocks to create the components of the distributed application, but we can also use Kubernetes primitives for some of the application behaviors. Table 1-1 shows how various development concepts are realized differently with local and distributed primitives in the JVM and Kubernetes, respectively.

Table 1-1. Local and distributed primitives

Concept	Local primitive	Distributed primitive
Behavior encapsulation	Class	Container image
Behavior instance	Object	Container
Unit of reuse	*.jar*	Container image
Composition	Class A contains Class B	Sidecar pattern
Inheritance	Class A extends Class B	A container's `FROM` parent image
Deployment unit	*.jar*/*.war*/*.ear*	Pod
Buildtime/Runtime isolation	Module, package, class	Namespace, Pod, container
Initialization preconditions	Constructor	Init container
Postinitialization trigger	Init-method	`postStart`
Predestroy trigger	Destroy-method	`preStop`
Cleanup procedure	`finalize()`, shutdown hook	-
Asynchronous and parallel execution	`ThreadPoolExecutor`, `ForkJoinPool`	Job
Periodic task	`Timer`, `ScheduledExecutorService`	CronJob
Background task	Daemon thread	DaemonSet
Configuration management	`System.getenv()`, `Properties`	ConfigMap, Secret

The in-process primitives and the distributed primitives have commonalities, but they are not directly comparable and replaceable. They operate at different abstraction levels and have different preconditions and guarantees. Some primitives are supposed to be used together. For example, we still have to use classes to create objects and put them into container images. However, some other primitives such as CronJob in Kubernetes can completely replace the `ExecutorService` behavior in Java.

Next, let's see a few distributed abstractions and primitives from Kubernetes that are especially interesting for application developers.

Containers

Containers are the building blocks for Kubernetes-based cloud native applications. If we make a comparison with OOP and Java, container images are like classes, and containers are like objects. The same way we can extend classes to reuse and alter behavior, we can have container images that extend other container images to reuse and alter behavior. The same way we can do object composition and use functionality, we can do container compositions by putting containers into a Pod and using collaborating containers.

If we continue the comparison, Kubernetes would be like the JVM but spread over multiple hosts, and it would be responsible for running and managing the containers. Init containers would be something like object constructors; DaemonSets would be similar to daemon threads that run in the background (like the Java Garbage Collector, for example). A Pod would be something similar to an Inversion of Control (IoC) context (Spring Framework, for example), where multiple running objects share a managed lifecycle and can access one another directly.

The parallel doesn't go much further, but the point is that containers play a fundamental role in Kubernetes, and creating modularized, reusable, single-purpose container images is fundamental to the long-term success of any project and even the containers' ecosystem as a whole. Apart from the technical characteristics of a container image that provide packaging and isolation, what does a container represent, and what is its purpose in the context of a distributed application? Here are a few suggestions on how to look at containers:

- A container image is the unit of functionality that addresses a single concern.
- A container image is owned by one team and has its own release cycle.
- A container image is self-contained and defines and carries its runtime dependencies.
- A container image is immutable, and once it is built, it does not change; it is configured.
- A container image defines its resource requirements and external dependencies.
- A container image has well-defined APIs to expose its functionality.
- A container typically runs as a single Unix process.
- A container is disposable and safe to scale up or down at any moment.

In addition to all these characteristics, a proper container image is modular. It is parameterized and created for reuse in the different environments in which it is going to run. Having small, modular, and reusable container images leads to the creation of more specialized and stable container images in the long term, similar to a great reusable library in the programming language world.

Pods

Looking at the characteristics of containers, we can see that they are a perfect match for implementing the microservices principles. A container image provides a single unit of functionality, belongs to a single team, has an independent release cycle, and provides deployment and runtime isolation. Most of the time, one microservice corresponds to one container image.

However, most cloud native platforms offer another primitive for managing the lifecycle of a group of containers—in Kubernetes, it is called a Pod. A *Pod* is an atomic unit of scheduling, deployment, and runtime isolation for a group of containers. All containers in a Pod are always scheduled to the same host, are deployed and scaled together, and can also share filesystem, networking, and process namespaces. This joint lifecycle allows the containers in a Pod to interact with one another over the filesystem or through networking via localhost or host interprocess communication mechanisms if desired (for performance reasons, for example). A Pod also represents a security boundary for an application. While it is possible to have containers with varying security parameters in the same Pod, typically all containers would have the same access level, network segmentation, and identity.

As you can see in Figure 1-2, at development and build time, a microservice corresponds to a container image that one team develops and releases. But at runtime, a microservice is represented by a Pod, which is the unit of deployment, placement, and scaling. The only way to run a container—whether for scale or migration—is through the Pod abstraction. Sometimes a Pod contains more than one container. In one such example, a containerized microservice uses a helper container at runtime, as Chapter 16, "Sidecar", demonstrates.

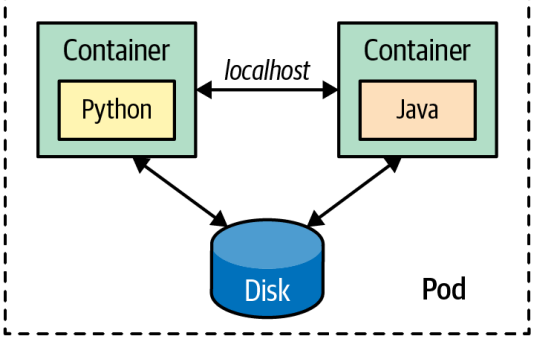


Figure 1-2. A Pod as the deployment and management unit

Containers, Pods, and their unique characteristics offer a new set of patterns and principles for designing microservices-based applications. We saw some of the characteristics of well-designed containers; now let's look at some characteristics of a Pod:

- A Pod is the atomic unit of scheduling. That means the scheduler tries to find a host that satisfies the requirements of all containers that belong to the Pod (we cover some specifics around init containers in Chapter 15, "Init Container"). If you create a Pod with many containers, the scheduler needs to find a host that has enough resources to satisfy all container demands combined. This scheduling process is described in Chapter 6, "Automated Placement".
- A Pod ensures colocation of containers. Thanks to the colocation, containers in the same Pod have additional means to interact with one another. The most common ways of communicating include using a shared local filesystem for exchanging data, using the localhost network interface, or using some host interprocess communication (IPC) mechanism for high-performance interactions.
- A Pod has an IP address, name, and port range that are shared by all containers belonging to it. That means containers in the same Pod have to be carefully configured to avoid port clashes, in the same way that parallel, running Unix processes have to take care when sharing the networking space on a host.

A Pod is the atom of Kubernetes where your application lives, but you don't access Pods directly—that is where Services enter the scene.

Services

Pods are ephemeral. They come and go at any time for all sorts of reasons (e.g., scaling up and down, failing container health checks, node migrations). A Pod IP address is known only after it is scheduled and started on a node. A Pod can be rescheduled to a different node if the existing node it is running on is no longer healthy. This means the Pod's network address may change over the life of an application, and there is a need for another primitive for discovery and load balancing.

That's where the Kubernetes Services come into play. The Service is another simple but powerful Kubernetes abstraction that binds the Service name to an IP address and port number permanently. So a Service represents a named entry point for accessing an application. In the most common scenario, the Service serves as the entry point for a set of Pods, but that might not always be the case. The Service is a generic primitive, and it may also point to functionality provided outside the Kubernetes cluster. As such, the Service primitive can be used for Service discovery and load balancing, and it allows altering implementations and scaling without affecting Service consumers. We explain Services in detail in Chapter 13, "Service Discovery".

Labels

We have seen that a microservice is a container image at build time but is represented by a Pod at runtime. So what is an application that consists of multiple microservices? Here, Kubernetes offers two more primitives that can help you define the concept of an application: labels and namespaces.

Before microservices, an application corresponded to a single deployment unit with a single versioning scheme and release cycle. There was a single file for an application in a *.war*, *.ear*, or some other packaging format. But then, applications were split into microservices, which are independently developed, released, run, restarted, or scaled. With microservices, the notion of an application diminishes, and there are no key artifacts or activities that we have to perform at the application level. But if you still need a way to indicate that some independent services belong to an application, *labels* can be used. Let's imagine that we have split one monolithic application into three microservices and another one into two microservices.

We now have five Pod definitions (and maybe many more Pod instances) that are independent of the development and runtime points of view. However, we may still need to indicate that the first three Pods represent an application and the other two Pods represent another application. Even the Pods may be independent, to provide a business value, but they may depend on one another. For example, one Pod may contain the containers responsible for the frontend, and the other two Pods are responsible for providing the backend functionality. If either of these Pods is down, the application is useless from a business point of view. Using label selectors gives us the ability to query and identify a set of Pods and manage it as one logical unit. Figure 1-3 shows how you can use labels to group the parts of a distributed application into specific subsystems.

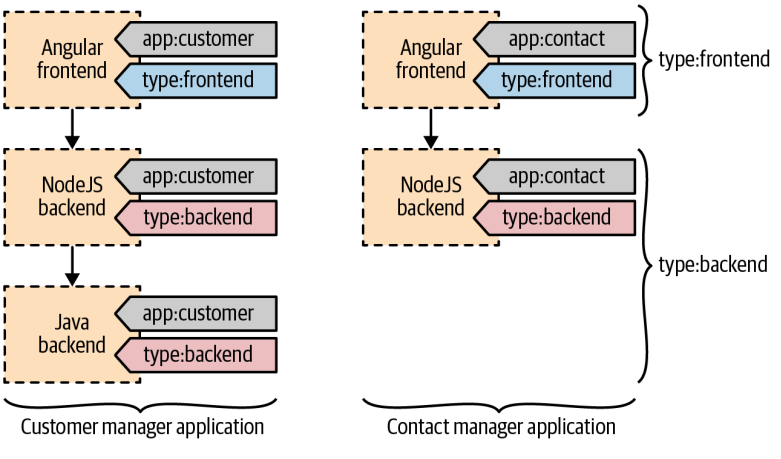


Figure 1-3. Labels used as an application identity for Pods

Here are a few examples where labels can be useful:

- Labels are used by ReplicaSets to keep some instances of a specific Pod running. That means every Pod definition needs to have a unique combination of labels used for scheduling.
- Labels are also heavily used by the scheduler. The scheduler uses labels for colocating or spreading Pods to the nodes that satisfy the Pods' requirements.
- A label can indicate a logical grouping of a set of Pods and give an application identity to them.
- In addition to the preceding typical use cases, labels can be used to store metadata. It may be difficult to predict what a label could be used for, but it is best to have enough labels to describe all important aspects of the Pods. For example, having labels to indicate the logical group of an application, the business characteristics and criticality, the specific runtime platform dependencies such as hardware architecture, or location preferences are all useful.

Later, these labels can be used by the scheduler for more fine-grained scheduling, or the same labels can be used from the command line for managing the matching Pods at scale. However, you should not go overboard and add too many labels in advance. You can always add them later if needed. Removing labels is much riskier as there is no straightforward way of finding out what a label is used for and what unintended effect such an action may cause.

Annotations

Another primitive very similar to labels is the *annotation*. Like labels, annotations are organized as a map, but they are intended for specifying nonsearchable metadata and for machine usage rather than human.

The information on the annotations is not intended for querying and matching objects. Instead, it is intended for attaching additional metadata to objects from various tools and libraries we want to use. Some examples of using annotations include build IDs, release IDs, image information, timestamps, Git branch names, pull request numbers, image hashes, registry addresses, author names, tooling information, and more. So while labels are used primarily for query matching and performing actions on the matching resources, annotations are used to attach metadata that can be consumed by a machine.

Namespaces

Another primitive that can also help manage a group of resources is the Kubernetes *namespace*. As we have described, a namespace may seem similar to a label, but in reality, it is a very different primitive with different characteristics and purposes.

Kubernetes namespaces allow you to divide a Kubernetes cluster (which is usually spread across multiple hosts) into a logical pool of resources. Namespaces provide scopes for Kubernetes resources and a mechanism to apply authorizations and other policies to a subsection of the cluster. The most common use case of namespaces is representing different software environments such as development, testing, integration testing, or production. Namespaces can also be used to achieve multitenancy and provide isolation for team workspaces, projects, and even specific applications. But ultimately, for a greater isolation of certain environments, namespaces are not enough, and having separate clusters is common. Typically, there is one nonproduction Kubernetes cluster used for some environments (development, testing, and integration testing) and another production Kubernetes cluster to represent performance testing and production environments.

Let's look at some of the characteristics of namespaces and how they can help us in different scenarios:

- A namespace is managed as a Kubernetes resource.
- A namespace provides scope for resources such as containers, Pods, Services, or ReplicaSets. The names of resources need to be unique within a namespace but not across them.
- By default, namespaces provide scope for resources, but nothing isolates those resources and prevents access from one resource to another. For example, a Pod from a development namespace can access another Pod from a production namespace as long as the Pod IP address is known. "Network isolation across namespaces for creating a lightweight multitenancy solution is described in Chapter 24, "Network Segmentation".
- Some other resources, such as namespaces, nodes, and PersistentVolumes, do not belong to namespaces and should have unique cluster-wide names.
- Each Kubernetes Service belongs to a namespace and gets a corresponding Domain Name Service (DNS) record that has the namespace in the form of `<service-name>.<namespace-name>.svc.cluster.local`. So the namespace name is in the URL of every Service belonging to the given namespace. That's one reason it is vital to name namespaces wisely.
- ResourceQuotas provide constraints that limit the aggregated resource consumption per namespace. With ResourceQuotas, a cluster administrator can control the number of objects per type that are allowed in a namespace. For example, a

developer namespace may allow only five ConfigMaps, five Secrets, five Services, five ReplicaSets, five PersistentVolumeClaims, and ten Pods.

- ResourceQuotas can also limit the total sum of computing resources we can request in a given namespace. For example, in a cluster with a capacity of 32 GB RAM and 16 cores, it is possible to allocate 16 GB RAM and 8 cores for the production namespace, 8 GB RAM and 4 cores for the staging environment, 4 GB RAM and 2 cores for development, and the same amount for testing namespaces. The ability to impose resource constraints decoupled from the shape and the limits of the underlying infrastructure is invaluable.

Discussion

We've only briefly covered a few of the main Kubernetes concepts we use in this book. However, there are more primitives used by developers on a day-by-day basis. For example, if you create a containerized service, there are plenty of Kubernetes abstractions you can use to reap all the benefits of Kubernetes. Keep in mind, these are only a few of the objects used by application developers to integrate a containerized service into Kubernetes. There are plenty of other concepts used primarily by cluster administrators for managing Kubernetes. Figure 1-4 gives an overview of the main Kubernetes resources that are useful for developers.

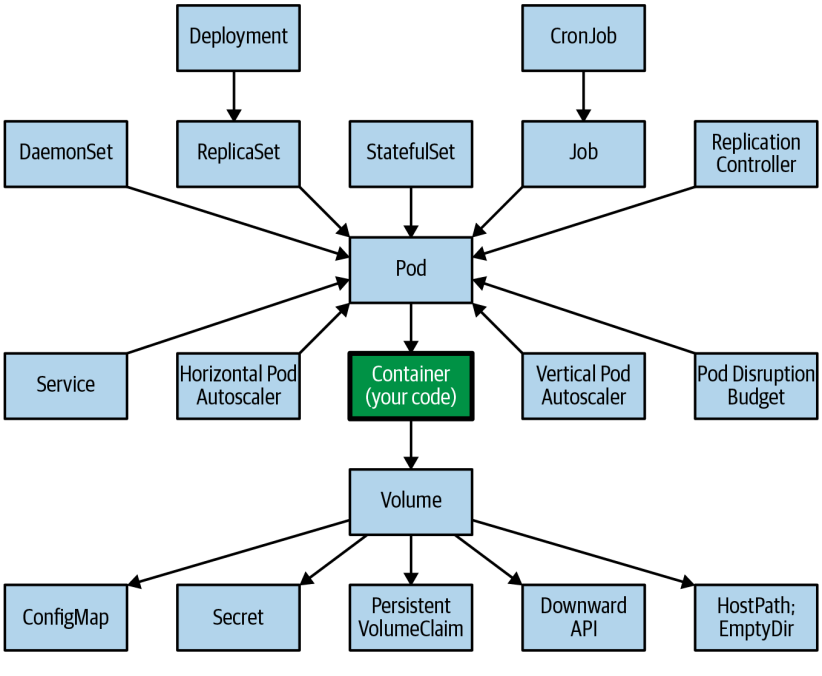


Figure 1-4. Kubernetes concepts for developers

With time, these new primitives give birth to new ways of solving problems, and some of these repetitive solutions become patterns. Throughout this book, rather than describing each Kubernetes resource in detail, we will focus on concepts that are proven as patterns.

More Information

- The Twelve-Factor App
- CNCF Cloud Native Definition v1.0
- Hexagonal Architecture
- *Domain-Driven Design: Tackling Complexity in the Heart of Software*
- Best Practices for Writing Dockerfiles
- Principles of Container-Based Application Design
- General Container Image Guidelines

PART I
Foundational Patterns

Foundational patterns describe a number of fundamental principles that containerized applications must comply with in order to become good cloud-native citizens. Adhering to these principles will help ensure that your applications are suitable for automation in cloud-native platforms such as Kubernetes.

The patterns described in the following chapters represent the foundational building blocks of distributed container-based Kubernetes-native applications:

- Chapter 2, "Predictable Demands", explains why every container should declare its resource requirements and stay confined to the indicated resource boundaries.
- Chapter 3, "Declarative Deployment", describes the different application deployment strategies that can be expressed in a declarative way.
- Chapter 4, "Health Probe", dictates that every container should implement specific APIs to help the platform observe and maintain the application healthily.
- Chapter 5, "Managed Lifecycle", explains why a container should have a way to read the events coming from the platform and conform by reacting to those events.
- Chapter 6, "Automated Placement", introduces the Kubernetes scheduling algorithm and the ways to influence the placement decisions from the outside.

CHAPTER 2

Predictable Demands

The foundation of successful application deployment, management, and coexistence on a shared cloud environment is dependent on identifying and declaring the application resource requirements and runtime dependencies. This *Predictable Demands* pattern indicates how you should declare application requirements, whether they are hard runtime dependencies or resource requirements. Declaring your requirements is essential for Kubernetes to find the right place for your application within the cluster.

Problem

Kubernetes can manage applications written in different programming languages as long as the application can be run in a container. However, different languages have different resource requirements. Typically, a compiled language runs faster and often requires less memory compared to just-in-time runtimes or interpreted languages. Considering that many modern programming languages in the same category have similar resource requirements, from a resource consumption point of view, more important aspects are the domain, the business logic of an application, and the actual implementation details.

Besides resource requirements, application runtimes also have dependencies on platform-managed capabilities like data storage or application configuration.

Solution

Knowing the runtime requirements for a container is important mainly for two reasons. First, with all the runtime dependencies defined and resource demands envisaged, Kubernetes can make intelligent decisions about where to place a container on the cluster for the most efficient hardware utilization. In an environment with shared resources among a large number of processes with different priorities, the only way to

ensure a successful coexistence is to know the demands of every process in advance. However, intelligent placement is only one side of the coin.

Container resource profiles are also essential for capacity planning. Based on the particular service demands and the total number of services, we can do some capacity planning for different environments and come up with the most cost-effective host profiles to satisfy the entire cluster demand. Service resource profiles and capacity planning go hand in hand for successful cluster management in the long term.

Before diving into resource profiles, let's look at declaring runtime dependencies.

Runtime Dependencies

One of the most common runtime dependencies is file storage for saving application state. Container filesystems are ephemeral and are lost when a container is shut down. Kubernetes offers volume as a Pod-level storage utility that survives container restarts.

The most straightforward type of volume is `emptyDir`, which lives as long as the Pod lives. When the Pod is removed, its content is also lost. The volume needs to be backed by another kind of storage mechanism to survive Pod restarts. If your application needs to read or write files to such long-lived storage, you must declare that dependency explicitly in the container definition using volumes, as shown in Example 2-1.

Example 2-1. Dependency on a PersistentVolume

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    volumeMounts:
    - mountPath: "/logs"
      name: log-volume
  volumes:
  - name: log-volume
    persistentVolumeClaim:  ❶
      claimName: random-generator-log
```

❶ Dependency of a PersistentVolumeClaim (PVC) to be present and bound.

The scheduler evaluates the kind of volume a Pod requires, which affects where the Pod gets placed. If the Pod needs a volume that is not provided by any node on

the cluster, the Pod is not scheduled at all. Volumes are an example of a runtime dependency that affects what kind of infrastructure a Pod can run and whether the Pod can be scheduled at all.

A similar dependency happens when you ask Kubernetes to expose a container port on a specific port on the host system through `hostPort`. The usage of a `hostPort` creates another runtime dependency on the nodes and limits where a Pod can be scheduled. `hostPort` reserves the port on each node in the cluster and is limited to a maximum of one Pod scheduled per node. Because of port conflicts, you can scale to as many Pods as there are nodes in the Kubernetes cluster.

Configurations are another type of dependency. Almost every application needs some configuration information, and the recommended solution offered by Kubernetes is through ConfigMaps. Your services need to have a strategy for consuming settings—either through environment variables or the filesystem. In either case, this introduces a runtime dependency of your container to the named ConfigMaps. If not all of the expected ConfigMaps are created, the containers are scheduled on a node, but they do not start up.

Similar to ConfigMaps, Secrets offer a slightly more secure way of distributing environment-specific configurations to a container. The way to consume a Secret is the same as it is for ConfigMaps, and using a Secret introduces the same kind of dependency from a container to a namespace.

ConfigMaps and Secrets are explained in more detail in Chapter 20, "Configuration Resource", and Example 2-2 shows how these resources are used as runtime dependencies.

Example 2-2. Dependency on a ConfigMap

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    env:
    - name: PATTERN
      valueFrom:
        configMapKeyRef:  ❶
          name: random-generator-config
          key: pattern
```

❶ Mandatory dependency on the ConfigMap `random-generator-config`.

While the creation of ConfigMap and Secret objects are simple deployment tasks we have to perform, cluster nodes provide storage and port numbers. Some of these dependencies limit where a Pod gets scheduled (if anywhere at all), and other dependencies may prevent the Pod from starting up. When designing your containerized applications with such dependencies, always consider the runtime constraints they will create later.

Resource Profiles

Specifying container dependencies such as ConfigMap, Secret, and volumes is straightforward. We need some more thinking and experimentation for figuring out the resource requirements of a container. Compute resources in the context of Kubernetes are defined as something that can be requested by, allocated to, and consumed from a container. The resources are categorized as *compressible* (i.e., can be throttled, such as CPU or network bandwidth) and *incompressible* (i.e., cannot be throttled, such as memory).

Making the distinction between compressible and incompressible resources is important. If your containers consume too many compressible resources such as CPU, they are throttled, but if they use too many incompressible resources (such as memory), they are killed (as there is no other way to ask an application to release allocated memory).

Based on the nature and the implementation details of your application, you have to specify the minimum amount of resources that are needed (called `requests`) and the maximum amount it can grow up to (the `limits`). Every container definition can specify the amount of CPU and memory it needs in the form of a request and limit. At a high level, the concept of `requests`/`limits` is similar to soft/hard limits. For example, similarly, we define heap size for a Java application by using the `-Xms` and `-Xmx` command-line options.

The `requests` amount (but not `limits`) is used by the scheduler when placing Pods to nodes. For a given Pod, the scheduler considers only nodes that still have enough capacity to accommodate the Pod and all of its containers by summing up the requested resource amounts. In that sense, the `requests` field of each container affects where a Pod can be scheduled or not. Example 2-3 shows how such limits are specified for a Pod.

Example 2-3. Resource limits

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    resources:
      requests:          ❶
        cpu: 100m
        memory: 200Mi
      limits:            ❷
        memory: 200Mi
```

❶ Initial resource request for CPU and memory.

❷ Upper limit until we want our application to grow at max. We don't specify CPU limits by intention.

The following types of resources can be used as keys in the `requests` and `limits` specification:

`memory`
: This type is for the heap memory demands of your application, including volumes of type `emptyDir` with the configuration `medium: Memory`. Memory resources are incompressible, so containers that exceed their configured memory limit will trigger the Pod to be evicted; i.e., it gets deleted and recreated potentially on a different node.

`cpu`
: The `cpu` type is used to specify the range of needed CPU cycles for your application. However, it is a compressible resource, which means that in an overcommit situation for a node, all assigned CPU slots of all running containers are throttled relative to their specified requests. Therefore, it is highly recommended that you set `requests` for the CPU resource but *no* `limits` so that they can benefit from all excess CPU resources that otherwise would be wasted.

`ephemeral-storage`
: Every node has some filesystem space dedicated for ephemeral storage that holds logs and writable container layers. `emptyDir` volumes that are not stored in a memory filesystem also use ephemeral storage. With this request and limit type, you can specify the application's minimal and maximal needs. `ephemeral-storage` resources are not compressible and will cause a Pod to be evicted from the node if it uses more storage than specified in its `limit`.

`hugepage-<size>`
: *Huge pages* are large, contiguous pre-allocated pages of memory that can be mounted as volumes. Depending on your Kubernetes node configuration, several sizes of huge pages are available, like 2 MB and 1 GB pages. You can specify a request and limit for how many of a certain type of huge pages you want to consume (e.g., `hugepages-1Gi: 2Gi` for requesting two 1 GB huge pages). Huge pages can't be overcommitted, so the request and limit must be the same.

Depending on whether you specify the `requests`, the `limits`, or both, the platform offers three types of Quality of Service (QoS):

Best-Effort
: Pods that do not have any requests and limits set for its containers have a QoS of *Best-Effort*. Such a *Best-Effort* Pod is considered the lowest priority and is most likely killed first when the node where the Pod is placed runs out of incompressible resources.

Burstable
: A Pod that defines an unequal amount for `requests` and `limits` values (and `limits` is larger than `requests`, as expected) are tagged as *Burstable*. Such a Pod has minimal resource guarantees but is also willing to consume more resources up to its `limit` when available. When the node is under incompressible resource pressure, these Pods are likely to be killed if no *Best-Effort* Pods remain.

Guaranteed
: A Pod that has an equal amount of `request` and `limit` resources belongs to the *Guaranteed* QoS category. These are the highest-priority Pods and are guaranteed not to be killed before *Best-Effort* and *Burstable* Pods. This QoS mode is the best option for your application's memory resources, as it entails the least surprise and avoids out-of-memory triggered evictions.

So the resource characteristics you define or omit for the containers have a direct impact on its QoS and define the relative importance of the Pod in the event of resource starvation. Define your Pod resource requirements with this consequence in mind.

Recommendations for CPU and Memory Resources

While you have many options for declaring the memory and CPU needs of your applications, we and others recommend the following rules:

- For memory, always set `requests` equal to `limits`.
- For CPU, set `requests` but no `limits`.

See the blog post "For the Love of God, Stop Using CPU Limits on Kubernetes" for a more in-depth explanation of why you should not use `limits` for the CPU, and see the blog post "What Everyone Should Know About Kubernetes Memory Limits" for more details about the recommended memory settings.

Pod Priority

We explained how container resource declarations also define Pods' QoS and affect the order in which the Kubelet kills the container in a Pod in case of resource starvation. Two other related concepts are Pod priority and preemption. *Pod priority* allows you to indicate the importance of a Pod relative to other Pods, which affects the order in which Pods are scheduled. Let's see that in action in Example 2-4.

Example 2-4. Pod priority

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority  ❶
value: 1000            ❷
globalDefault: false   ❸
description: This is a very high-priority Pod class
---
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
  labels:
    env: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
  priorityClassName: high-priority  ❹
```

❶ The name of the priority class object.

❷ The priority value of the object.

❸ `globalDefault` set to `true` is used for Pods that do not specify a `priorityClassName`. Only one PriorityClass can have `globalDefault` set to `true`.

❹ The priority class to use with this Pod, as defined in PriorityClass resource.

We created a PriorityClass, a non-namespaced object for defining an integer-based priority. Our PriorityClass is named `high-priority` and has a priority of 1,000.

Now we can assign this priority to Pods by its name as `priorityClassName: high-priority`. PriorityClass is a mechanism for indicating the importance of Pods relative to one another, where the higher value indicates more important Pods.

Pod priority affects the order in which the scheduler places Pods on nodes. First, the priority admission controller uses the `priorityClassName` field to populate the priority value for new Pods. When multiple Pods are waiting to be placed, the scheduler sorts the queue of pending Pods by highest priority first. Any pending Pod is picked before any other pending Pod with lower priority in the scheduling queue, and if there are no constraints preventing it from scheduling, the Pod gets scheduled.

Here comes the critical part. If there are no nodes with enough capacity to place a Pod, the scheduler can preempt (remove) lower-priority Pods from nodes to free up resources and place Pods with higher priority. As a result, the higher-priority Pod might be scheduled sooner than Pods with a lower priority if all other scheduling requirements are met. This algorithm effectively enables cluster administrators to control which Pods are more critical workloads and place them first by allowing the scheduler to evict Pods with lower priority to make room on a worker node for higher-priority Pods. If a Pod cannot be scheduled, the scheduler continues with the placement of other lower-priority Pods.

Suppose you want your Pod to be scheduled with a particular priority but don't want to evict any existing Pods. In that case, you can mark a PriorityClass with the field `preemptionPolicy: Never`. Pods assigned to this priority class will not trigger any eviction of running Pods but will still get scheduled according to their priority value.

Pod QoS (discussed previously) and Pod priority are two orthogonal features that are not connected and have only a little overlap. QoS is used primarily by the Kubelet to preserve node stability when available compute resources are low. The Kubelet first considers QoS and then the PriorityClass of Pods before eviction. On the other hand, the scheduler eviction logic ignores the QoS of Pods entirely when choosing preemption targets. The scheduler attempts to pick a set of Pods with the lowest priority possible that satisfies the needs of higher-priority Pods waiting to be placed.

When Pods have a priority specified, it can have an undesired effect on other Pods that are evicted. For example, while a Pod's graceful termination policies are respected, the PodDisruptionBudget as discussed in Chapter 10, "Singleton Service", is not guaranteed, which could break a lower-priority clustered application that relies on a quorum of Pods.

Another concern is a malicious or uninformed user who creates Pods with the highest possible priority and evicts all other Pods. To prevent that, ResourceQuota has been extended to support PriorityClass, and higher-priority numbers are reserved for critical system-Pods that should not usually be preempted or evicted.

In conclusion, Pod priorities should be used with caution because user-specified numerical priorities that guide the scheduler and Kubelet about which Pods to place or to kill are subject to gaming by users. Any change could affect many Pods and could prevent the platform from delivering predictable service-level agreements.

Project Resources

Kubernetes is a self-service platform that enables developers to run applications as they see suitable on the designated isolated environments. However, working in a shared multitenanted platform also requires the presence of specific boundaries and control units to prevent some users from consuming all the platform's resources. One such tool is ResourceQuota, which provides constraints for limiting the aggregated resource consumption in a namespace. With ResourceQuotas, the cluster administrators can limit the total sum of computing resources (CPU, memory) and storage consumed. It can also limit the total number of objects (such as ConfigMaps, Secrets, Pods, or Services) created in a namespace. Example 2-5 shows an instance that limits the usage of certain resources. See the official Kubernetes documentation on Resource Quotas for the full list of supported resources for which you can restrict usage with ResourceQuotas.

Example 2-5. Definition of resource constraints

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
  namespace: default    ❶
spec:
  hard:
    pods: 4             ❷
    limits.memory: 5Gi  ❸
```

❶ Namespace to which resource constraints are applied.

❷ Allow four active Pods in this namespace.

❸ The sum of all memory limits of all Pods in this namespace must not be more than 5 GB.

Another helpful tool in this area is LimitRange, which allows you to set resource usage limits for each type of resource. In addition to specifying the minimum and maximum permitted amounts for different resource types and the default values for these resources, it also allows you to control the ratio between the `requests` and `limits`, also known as the *overcommit level*. Example 2-6 shows a LimitRange and the possible configuration options.

Example 2-6. Definition of allowed and default resource usage limits

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limits
  namespace: default
spec:
  limits:
  - min:                          ❶
      memory: 250Mi
      cpu: 500m
    max:                          ❷
      memory: 2Gi
      cpu: 2
    default:                      ❸
      memory: 500Mi
      cpu: 500m
    defaultRequest:               ❹
      memory: 250Mi
      cpu: 250m
    maxLimitRequestRatio:         ❺
      memory: 2
      cpu: 4
    type: Container               ❻
```

❶ Minimum values for requests and limits.

❷ Maximum values for requests and limits.

❸ Default values for limits when no limits are specified.

❹ Default values for requests when no requests are specified.

❺ Maximum ratio limit/request, used to specify the allowed overcommit level. Here, the memory limit must not be larger than twice the memory request, and the CPU limit can be as high as four times the CPU request.

❻ Type can be `Container`, `Pod`, (for all containers combined), or `PersistentVolumeClaim` (to specify the range for a request persistent volume).

LimitRanges help control the container resource profiles so that no containers require more resources than a cluster node can provide. LimitRanges can also prevent cluster users from creating containers that consume many resources, making the nodes not allocatable for other containers. Considering that the `requests` (and not `limits`) are the primary container characteristic the scheduler uses for placing, LimitRequestRatio allows you to control the amount of difference between the `requests` and `limits` of containers. A big combined gap between `requests` and `limits` increases the chances of overcommitting on the node and may degrade application performance when many containers simultaneously require more resources than initially requested.

Keep in mind that other shared node-level resources such as process IDs (PIDs) can be exhausted before hitting any resource limits. Kubernetes allows you to reserve a number of node PIDs for the system use and ensure that they are never exhausted by user workloads. Similarly, Pod PID limits allow a cluster administrator to limit the number of processes running in a Pod. We are not reviewing these in details here as they are set as Kubelet configurations options by cluster administrators and are not used by application developers.

Capacity Planning

Considering that containers may have different resource profiles in different environments, and a varied number of instances, it is evident that capacity planning for a multipurpose environment is not straightforward. For example, for best hardware utilization, on a nonproduction cluster, you may have mainly *Best-Effort* and *Burstable* containers. In such a dynamic environment, many containers are starting up and shutting down at the same time, and even if a container gets killed by the platform during resource starvation, it is not fatal. On the production cluster, where we want things to be more stable and predictable, the containers may be mainly of the *Guaranteed* type, and some may be *Burstable*. If a container gets killed, that is most likely a sign that the capacity of the cluster should be increased.

Table 2-1 presents a few services with CPU and memory demands.

Table 2-1. Capacity planning example

Pod	CPU request	Memory request	Memory limit	Instances
A	500 m	500 Mi	500 Mi	4
B	250 m	250 Mi	1000 Mi	2
C	500 m	1000 Mi	2000 Mi	2
D	500 m	500 Mi	500 Mi	1
Total	**4000 m**	**5000 Mi**	**8500 Mi**	**9**

Of course, in a real-life scenario, the more likely reason you are using a platform such as Kubernetes is that there are many more services to manage, some of which are about to retire, and some of which are still in the design and development phase. Even if it is a continually moving target, based on a similar approach as described previously, we can calculate the total amount of resources needed for all the services per environment.

Keep in mind that in the different environments, there are different numbers of containers, and you may even need to leave some room for autoscaling, build jobs, infrastructure containers, and more. Based on this information and the infrastructure provider, you can choose the most cost-effective compute instances that provide the required resources.

Discussion

Containers are useful not only for process isolation and as a packaging format. With identified resource profiles, they are also the building blocks for successful capacity planning. Perform some early tests to discover the resource needs for each container, and use that information as a base for future capacity planning and prediction.

Kubernetes can help you here with the *Vertical Pod Autoscaler* (VPA), which monitors the resource consumption of your Pod over time and gives a recommendation for requests and limits. The VPA is described in detail in "Vertical Pod Autoscaling" on page 325.

However, more importantly, resource profiles are the way an application communicates with Kubernetes to assist in scheduling and managing decisions. If your application doesn't provide any `requests` or `limits`, all Kubernetes can do is treat your containers as opaque boxes that are dropped when the cluster gets full. So it is more or less mandatory for every application to think about and provide these resource declarations.

Now that you know how to size our applications, in Chapter 3, "Declarative Deployment", you will learn multiple strategies to install and update our applications on Kubernetes.

More Information

- Predictable Demands Example
- Configure a Pod to Use a ConfigMap
- Kubernetes Best Practices: Resource Requests and Limits
- Resource Management for Pods and Containers
- Manage HugePages

- Configure Default Memory Requests and Limits for a Namespace
- Node-Pressure Eviction
- Pod Priority and Preemption
- Configure Quality of Service for Pods
- Resource Quality of Service in Kubernetes
- Resource Quotas
- Limit Ranges
- Process ID Limits and Reservations
- For the Love of God, Stop Using CPU Limits on Kubernetes
- What Everyone Should Know About Kubernetes Memory Limits

CHAPTER 3

Declarative Deployment

The heart of the *Declarative Deployment* pattern is the Kubernetes Deployment resource. This abstraction encapsulates the upgrade and rollback processes of a group of containers and makes its execution a repeatable and automated activity.

Problem

We can provision isolated environments as namespaces in a self-service manner and place the applications in these environments with minimal human intervention through the scheduler. But with a growing number of microservices, continually updating and replacing them with newer versions becomes an increasing burden too.

Upgrading a service to a next version involves activities such as starting the new version of the Pod, stopping the old version of a Pod gracefully, waiting and verifying that it has launched successfully, and sometimes rolling it all back to the previous version in the case of failure. These activities are performed either by allowing some downtime but not running concurrent service versions, or with no downtime but increased resource usage due to both versions of the service running during the update process. Performing these steps manually can lead to human errors, and scripting properly can require a significant amount of effort, both of which quickly turn the release process into a bottleneck.

Solution

Luckily, Kubernetes has automated application upgrades as well. Using the concept of *Deployment*, we can describe how our application should be updated, using different strategies and tuning the various aspects of the update process. If you consider that you do multiple Deployments for every microservice instance per release cycle

(which, depending on the team and project, can span from minutes to several months), this is another effort-saving automation by Kubernetes.

In Chapter 2, "Predictable Demands", we saw that, to do its job effectively, the scheduler requires sufficient resources on the host system, appropriate placement policies, and containers with adequately defined resource profiles. Similarly, for a Deployment to do its job correctly, it expects the containers to be good cloud native citizens. At the very core of a Deployment is the ability to start and stop a set of Pods predictably. For this to work as expected, the containers themselves usually listen and honor lifecycle events (such as SIGTERM; see Chapter 5, "Managed Lifecycle") and also provide health-check endpoints as described in Chapter 4, "Health Probe", which indicate whether they started successfully.

If a container covers these two areas accurately, the platform can cleanly shut down old containers and replace them by starting updated instances. Then all the remaining aspects of an update process can be defined in a declarative way and executed as one atomic action with predefined steps and an expected outcome. Let's see the options for a container update behavior.

Deployment Updates with kubectl rollout

In previous versions of Kubernetes, rolling updates were implemented on the client side with the `kubectl rolling-update` command. In Kubernetes 1.18, `rolling-update` was removed in favor of a `rollout` command for `kubectl`. The difference is that `kubectl rollout` manages an application update on the server side by updating the Deployment *declaration* and leaving it to Kubernetes to perform the update. The `kubectl rolling-update` command, in contrast, was *imperative*: the client `kubectl` told the server what to do for each update step.

A Deployment can be fully managed by updating the Kubernetes resources files. However, `kubectl rollout` comes in very handy for everyday rollout tasks:

`kubectl rollout status`
: Shows the current status of a Deployment's rollout.

`kubectl rollout pause`
: Pauses a rolling update so that multiple changes can be applied to a Deployment without retriggering another rollout.

`kubectl rollout resume`
: Resumes a previously paused rollout.

`kubectl rollout undo`
: Performs a rollback to a prevision revision of a Deployment. A rollback is helpful in case of an error during the update.

`kubectl rollout history`
: Shows the available revisions of a Deployment.

`kubectl rollout restart`
: Does not perform an update but restarts the current set of Pods belonging to a Deployment using the configured rollout strategy.

You can find usage examples for `kubectl rollout` commands in the examples.

Rolling Deployment

The declarative way of updating applications in Kubernetes is through the concept of Deployment. Behind the scenes, the Deployment creates a ReplicaSet that supports set-based label selectors. Also, the Deployment abstraction allows you to shape the update process behavior with strategies such as `RollingUpdate` (default) and `Recreate`. Example 3-1 shows the important bits for configuring a Deployment for a rolling update strategy.

Example 3-1. Deployment for a rolling update

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: random-generator
spec:
  replicas: 3                ❶
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1            ❷
      maxUnavailable: 1      ❸
  minReadySeconds: 60        ❹
  selector:
    matchLabels:
      app: random-generator
  template:
    metadata:
      labels:
        app: random-generator
    spec:
      containers:
      - image: k8spatterns/random-generator:1.0
        name: random-generator
        readinessProbe:      ❺
          exec:
            command: [ "stat", "/tmp/random-generator-ready" ]
```

❶ Declaration of three replicas. You need more than one replica for a rolling update to make sense.

❷ Number of Pods that can be run temporarily in addition to the replicas specified during an update. In this example, it could be a maximum of four replicas.

❸ Number of Pods that may be unavailable during the update. Here it could be that only two Pods are available at a time during the update.

❹ Duration in seconds of all readiness probes for a rolled-out Pod needs to be healthy until the rollout continues.

❺ Readiness probes that are very important for a rolling deployment to ensure zero downtime—don't forget them (see Chapter 4, "Health Probe").

`RollingUpdate` strategy behavior ensures there is no downtime during the update process. Behind the scenes, the Deployment implementation performs similar moves by creating new ReplicaSets and replacing old containers with new ones. One enhancement here is that with Deployment, it is possible to control the rate of a new container rollout. The Deployment object allows you to control the range of available and excess Pods through `maxSurge` and `maxUnavailable` fields.

These two fields can be either absolute numbers of Pods or relative percentages that are applied to the configured number of replicas for the Deployment and are rounded up (`maxSurge`) or down (`maxUnavailable`) to the next integer value. By default, `maxSurge` and `maxUnavailable` are both set to 25%.

Another important parameter that influences the rollout behavior is `minReadySeconds`. This field specifies the duration in seconds that the readiness probes of a Pod need to be successful until the Pod itself is considered to be available in a rollout. Increasing this value guarantees that your application Pod is successfully running for some time before continuing with the rollout. Also, a larger `minReadySeconds` interval helps in debugging and exploring the new version. A `kubectl rollout pause` might be easier to leverage when the intervals between the update steps are larger.

Figure 3-1 shows the rolling update process.

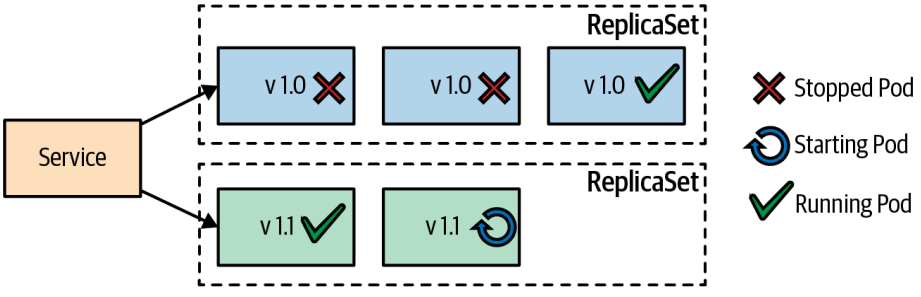


Figure 3-1. Rolling deployment

To trigger a declarative update, you have three options:

- Replace the whole Deployment with the new version's Deployment with `kubectl replace`.
- Patch (`kubectl patch`) or interactively edit (`kubectl edit`) the Deployment to set the new container image of the new version.
- Use `kubectl set image` to set the new image in the Deployment.

See also the full example in our repository, which demonstrates the usage of these commands and shows you how to monitor or roll back an upgrade with `kubectl rollout`.

In addition to addressing the drawbacks of the imperative way of deploying services, the Deployment has the following benefits:

- Deployment is a Kubernetes resource object whose status is entirely managed by Kubernetes internally. The whole update process is performed on the server side without client interaction.
- The declarative nature of Deployment specifies how the deployed state should look rather than the steps necessary to get there.
- The Deployment definition is an executable object and more than just documentation. It can be tried and tested on multiple environments before reaching production.
- The update process is also wholly recorded and versioned with options to pause, continue, and roll back to previous versions.

Fixed Deployment

A `RollingUpdate` strategy is useful for ensuring zero downtime during the update process. However, the side effect of this approach is that during the update process, two versions of the container are running at the same time. That may cause issues for the service consumers, especially when the update process has introduced backward-incompatible changes in the service APIs and the client is not capable of dealing with them. For this kind of scenario, you can use the `Recreate` strategy, which is illustrated in Figure 3-2.

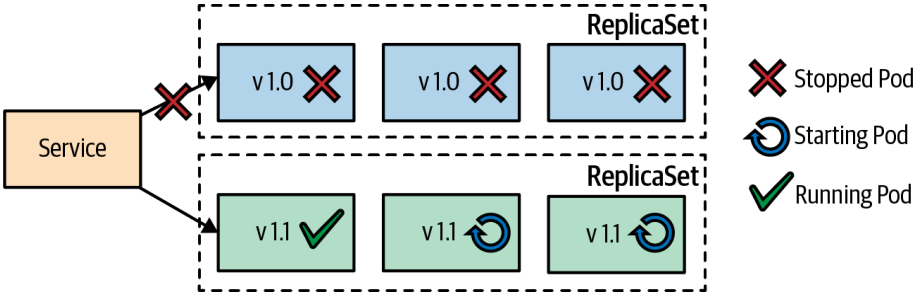


Figure 3-2. Fixed deployment using a `Recreate` strategy

The `Recreate` strategy has the effect of setting `maxUnavailable` to the number of declared replicas. This means it first kills all containers from the current version and then starts all new containers simultaneously when the old containers are evicted. The result of this sequence is that downtime occurs while all containers with old versions are stopped, and no new containers are ready to handle incoming requests. On the positive side, two different versions of the containers won't be running at the same time, so service consumers can connect only one version at a time.

Blue-Green Release

The *Blue-Green deployment* is a release strategy used for deploying software in a production environment by minimizing downtime and reducing risk. The Kubernetes Deployment abstraction is a fundamental concept that lets you define how Kubernetes transitions immutable containers from one version to another. We can use the Deployment primitive as a building block, together with other Kubernetes primitives, to implement this more advanced release strategy.

A Blue-Green deployment needs to be done manually if no extensions like a service mesh or Knative are used, though. Technically, it works by creating a second Deployment, with the latest version of the containers (let's call it *green*) not serving any requests yet. At this stage, the old Pod replicas from the original Deployment (called *blue*) are still running and serving live requests.

Once we are confident that the new version of the Pods is healthy and ready to handle live requests, we switch the traffic from old Pod replicas to the new replicas. You can do this in Kubernetes by updating the Service selector to match the new containers (labeled with green). As demonstrated in Figure 3-3, once the green (v1.1) containers handle all the traffic, the blue (v1.0) containers can be deleted and the resources freed for future Blue-Green deployments.

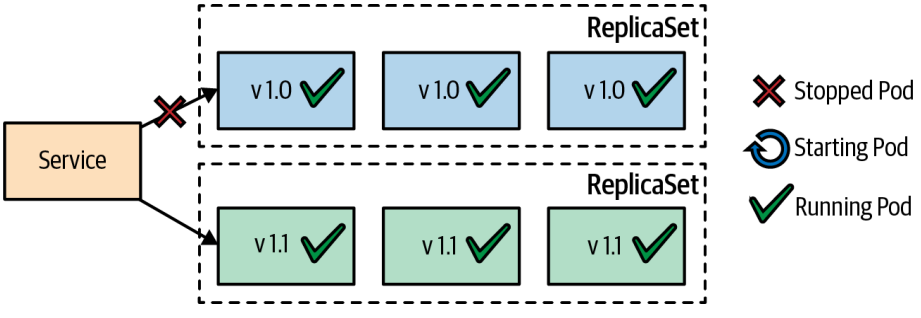


Figure 3-3. Blue-Green release

A benefit of the Blue-Green approach is that only one version of the application is serving requests at a time, which reduces the complexity of handling multiple concurrent versions by the Service consumers. The downside is that it requires twice the application capacity while both blue and green containers are up and running. Also, significant complications can occur with long-running processes and database state drifts during the transitions.

Canary Release

Canary release is a way to softly deploy a new version of an application into production by replacing only a small subset of old instances with new ones. This technique reduces the risk of introducing a new version into production by letting only some of the consumers reach the updated version. When we're happy with the new version of our service and how it performed with a small sample of users, we can replace all the old instances with the new version in an additional step after this canary release. Figure 3-4 shows a canary release in action.

In Kubernetes, this technique can be implemented by creating a new Deployment with a small replica count that can be used as the canary instance. At this stage, the Service should direct some of the consumers to the updated Pod instances. After the canary release and once we are confident that everything with the new ReplicaSet works as expected, we scale the new ReplicaSet up, and the old ReplicaSet down to zero. In a way, we're performing a controlled and user-tested incremental rollout.

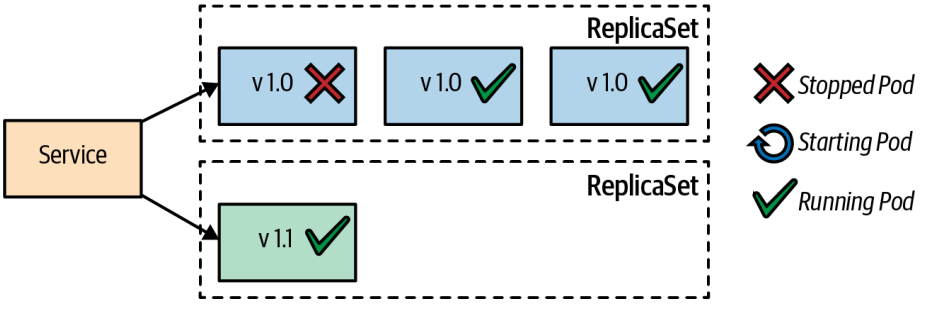


Figure 3-4. Canary release

Discussion

The Deployment primitive is an example of Kubernetes turning the tedious process of manually updating applications into a declarative activity that can be repeated and automated. The out-of-the-box deployment strategies (rolling and recreate) control the replacement of old containers by new ones, and the advanced release strategies (Blue-Green and canary) control how the new version becomes available to service consumers. The latter two release strategies are based on a human decision for the transition trigger and as a consequence are not fully automated by Kubernetes but require human interaction. Figure 3-5 summarizes of the deployment and release strategies, showing instance counts during transitions.

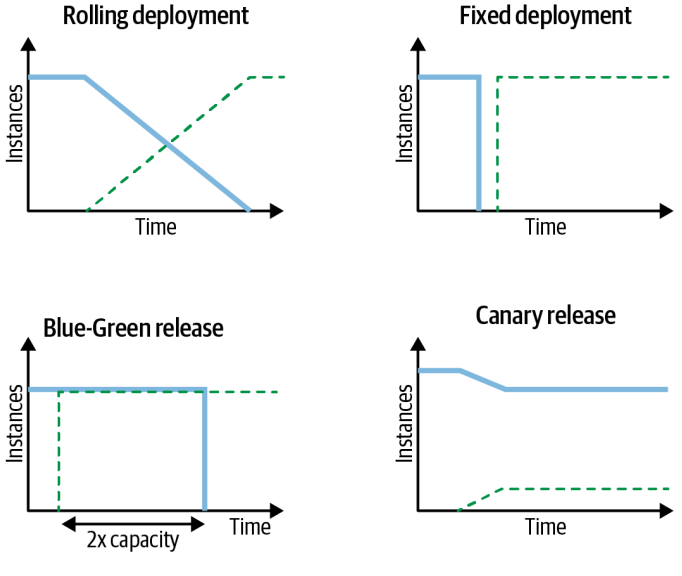


Figure 3-5. Deployment and release strategies

All software is different, and deploying complex systems usually requires additional steps and checks. The techniques discussed in this chapter cover the Pod update process, but do not include updating and rolling back other Pod dependencies such as ConfigMaps, Secrets, or other dependent services.

Pre and Post Deployment Hooks

In the past, there has been a proposal for Kubernetes to allow hooks in the deployment process. Pre and Post hooks would allow the execution of custom commands before and after Kubernetes has executed a deployment strategy. Such commands could perform additional actions while the deployment is in progress and would additionally be able to abort, retry, or continue a deployment. Those hooks are a good step toward new automated deployment and release strategies. Unfortunately, this effort has been stalled for some years (as of 2023), so it is unclear whether this feature will ever come to Kubernetes.

One approach that works today is to create a script to manage the update process of services and their dependencies using the Deployment and other primitives discussed in this book. However, this imperative approach that describes the individual update steps does not match the declarative nature of Kubernetes.

As an alternative, higher-level declarative approaches have emerged on top of Kubernetes. The most important platforms are described in the sidebar that follows. Those techniques work with operators (see Chapter 28, "Operator") that take a declarative description of the rollout process and perform the necessary actions on the server side, some of them also including automatic rollbacks in case of an update error. For advanced, production-ready rollout scenarios, it is recommended to look at one of those extensions.

Higher-Level Deployments

The Deployment resource is a good abstraction over ReplicaSets and Pods to allow a simple declarative rollout that a handful of parameters can tune. However, as we have seen, Deployment does not support more sophisticated strategies like canary or Blue-Green deployments directly. There are higher-level abstractions that enhance Kubernetes by introducing new resource types, enabling the declaration of more flexible deployment strategies. Those extensions all leverage the *Operator* pattern described in Chapter 28 and introduce their own custom resources for describing the desired rollout behavior.

As of 2023, the most prominent platforms that support higher-level Deployments include the following:

Flagger
: Flagger implements several deployment strategies and is part of the Flux CD GitOps tools. It supports canary and Blue-Green deployments and integrates with many ingress controllers and service meshes to provide the necessary traffic split between your app's old and new versions. It can also monitor the status of the rollout process based on a custom metric and detect if the rollout fails so that it can trigger an automatic rollback.

Argo Rollouts
: The focus on this part of the Argo family of tools is on providing a comprehensive and opinionated continuous delivery (CD) solution for Kubernetes. Argo Rollouts support advanced deployment strategies, like Flagger, and integrate into many ingress controllers and service meshes. It has very similar capabilities to Flagger, so the decision about which one to use should be based on which CD solution you prefer, Argo or Flux.

Knative
: Knative a serverless platform on top of Kubernetes. A core feature of Knative is traffic-driven autoscaling support, which is described in detail in Chapter 29, "Elastic Scale". Knative also provides a simplified deployment model and traffic splitting, which is very helpful for supporting high-level deployment rollouts. The support for rollout or rollbacks is not as advanced as with Flagger or Argo Rollouts but is still a substantial improvement over the rollout capabilities of Kubernetes Deployments. If you are using Knative anyway, the intuitive way of splitting traffic between two application versions is a good alternative to Deployments.

Like Kubernetes, all of these projects are part of the Cloud Native Computing Foundation (CNCF) project and have excellent community support.

Regardless of the deployment strategy you are using, it is essential for Kubernetes to know when your application Pods are up and running to perform the required sequence of steps to reach the defined target deployment state. The next pattern, *Health Probe*, in Chapter 4 describes how your application can communicate its health state to Kubernetes.

More Information

- Declarative Deployment Example
- Performing a Rolling Update
- Deployments
- Run a Stateless Application Using a Deployment
- Blue-Green Deployment

- Canary Release
- Flagger: Deployment Strategies
- Argo Rollouts
- Knative: Traffic Management

CHAPTER 4

Health Probe

The *Health Probe* pattern indicates how an application can communicate its health state to Kubernetes. To be fully automatable, a cloud native application must be highly observable by allowing its state to be inferred so that Kubernetes can detect whether the application is up and whether it is ready to serve requests. These observations influence the lifecycle management of Pods and the way traffic is routed to the application.

Problem

Kubernetes regularly checks the container process status and restarts it if issues are detected. However, from practice, we know that checking the process status is not sufficient to determine the health of an application. In many cases, an application hangs, but its process is still up and running. For example, a Java application may throw an `OutOfMemoryError` and still have the JVM process running. Alternatively, an application may freeze because it runs into an infinite loop, deadlock, or some thrashing (cache, heap, process). To detect these kinds of situations, Kubernetes needs a reliable way to check the health of applications—that is, not to understand how an application works internally, but to check whether the application is functioning as expected and capable of serving consumers.

Solution

The software industry has accepted the fact that it is not possible to write bug-free code. Moreover, the chances for failure increase even more when working with distributed applications. As a result, the focus for dealing with failures has shifted from avoiding them to detecting faults and recovering. Detecting failure is not a simple task that can be performed uniformly for all applications, as everyone has different

definitions of a failure. Also, various types of failures require different corrective actions. Transient failures may self-recover, given enough time, and some other failures may need a restart of the application. Let's look at the checks Kubernetes uses to detect and correct failures.

Process Health Checks

A *process health check* is the simplest health check the Kubelet constantly performs on the container processes. If the container processes are not running, the container is restarted on the node to which the Pod is assigned. So even without any other health checks, the application becomes slightly more robust with this generic check. If your application is capable of detecting any kind of failure and shutting itself down, the process health check is all you need. However, for most cases, that is not enough, and other types of health checks are also necessary.

Liveness Probes

If your application runs into a deadlock, it is still considered healthy from the process health check's point of view. To detect this kind of issue and any other types of failure according to your application business logic, Kubernetes has *liveness probes*—regular checks performed by the Kubelet agent that asks your container to confirm it is still healthy. It is important to have the health check performed from the outside rather than in the application itself, as some failures may prevent the application watchdog from reporting its failure. Regarding corrective action, this health check is similar to a process health check, since if a failure is detected, the container is restarted. However, it offers more flexibility regarding which methods to use for checking the application health, as follows:

HTTP probe
: Performs an HTTP GET request to the container IP address and expects a successful HTTP response code between 200 and 399.

TCP Socket probe
: Assumes a successful TCP connection.

Exec probe
: Executes an arbitrary command in the container's user and kernel namespace and expects a successful exit code (0).

gRPC probe
: Leverages gRPC's intrinsic support for health checks.

In addition to the probe action, the health check behavior can be influenced with the following parameters:

`initialDelaySeconds`
: Specifies the number of seconds to wait until the first liveness probe is checked.

`periodSeconds`
: The interval in seconds between liveness probe checks.

`timeoutSeconds`
: The maximum time allowed for a probe check to return before it is considered to have failed.

`failureThreshold`
: Specifies how many times a probe check needs to fail in a row until the container is considered to be unhealthy and needs to be restarted.

An example HTTP-based liveness probe is shown in Example 4-1.

Example 4-1. Container with a liveness probe

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-liveness-check
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    env:
    - name: DELAY_STARTUP
      value: "20"
    ports:
    - containerPort: 8080
      protocol: TCP
    livenessProbe:
      httpGet:                    ❶
        path: /actuator/health
        port: 8080
      initialDelaySeconds: 30     ❷
```

❶ HTTP probe to a health-check endpoint.

❷ Wait 30 seconds before doing the first liveness check to give the application some time to warm up.

Depending on the nature of your application, you can choose the method that is most suitable for you. It is up to your application to decide whether it considers itself healthy or not. However, keep in mind that the result of not passing a health check is that your container will restart. If restarting your container does not help, there is no benefit to having a failing health check as Kubernetes restarts your container without fixing the underlying issue.

Readiness Probes

Liveness checks help keep applications healthy by killing unhealthy containers and replacing them with new ones. But sometimes, when a container is not healthy, restarting it may not help. A typical example is a container that is still starting up and is not ready to handle any requests. Another example is an application that is still waiting for a dependency like a database to be available. Also, a container can be overloaded, increasing its latency, so you want it to shield itself from the additional load for a while and indicate that it is not ready until the load decreases.

For this kind of scenario, Kubernetes has *readiness probes*. The methods (HTTP, TCP, Exec, gRPC) and timing options for performing readiness checks are the same as for liveness checks, but the corrective action is different. Rather than restarting the container, a failed readiness probe causes the container to be removed from the service endpoint and not receive any new traffic. Readiness probes signal when a container is ready so that it has some time to warm up before getting hit with requests from the service. It is also useful for shielding the container from traffic at later stages, as readiness probes are performed regularly, similarly to liveness checks. Example 4-2 shows how a readiness probe can be implemented by probing the existence of a file the application creates when it is ready for operations.

Example 4-2. Container with readiness probe

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-readiness-check
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    readinessProbe:
      exec:  ❶
        command: [ "stat", "/var/run/random-generator-ready" ]
```

❶ Check for the existence of a file the application creates to indicate it's ready to serve requests. `stat` returns an error if the file does not exist, letting the readiness check fail.

Again, it is up to your implementation of the health check to decide when your application is ready to do its job and when it should be left alone. While process health checks and liveness checks are intended to recover from the failure by restarting the container, the readiness check buys time for your application and expects it to recover by itself. Keep in mind that Kubernetes tries to prevent your container from receiving new requests (when it is shutting down, for example), regardless of whether the readiness check still passes after having received a SIGTERM signal.

Custom Pod Readiness Gates

Readiness probes work on a per-container level, and a Pod is considered ready to serve requests when all containers pass their readiness probes. In some situations, this is not good enough—for example, when an external load balancer like the AWS LoadBalancer needs to be reconfigured and ready too. In this case, the `readinessGates` field of a Pod's specification can be used to specify extra conditions that need to be met for the Pod to become ready. Example 4-3 shows a readiness gate that will introduce an additional condition, `k8spatterns.io/load-balancer-ready`, to the Pod's `status` sections.

Example 4-3. Readiness gate for indicating the status of an external load balancer

```
apiVersion: v1
kind: Pod
...
spec:
  readinessGates:
  - conditionType: "k8spatterns.io/load-balancer-ready"
...
status:
  conditions:
  - type: "k8spatterns.io/load-balancer-ready" ❶
    status: "False"
    ...
  - type: Ready                                ❷
    status: "False"
    ...
```

❶ New condition introduced by Kubernetes and set to `False` by default. It needs to be switched to `True` externally, e.g., by a controller, as desribed in Chapter 27, "Controller", when the load balancer is ready to serve.

❷ The Pod is "ready" when all containers' readiness probes are passing and the readiness gates' conditions are `True`; otherwise, as here, the Pod is marked as nonready.

> Pod readiness gates are an advanced feature that are not supposed to be used by the end user but by Kubernetes add-ons to introduce additional dependencies on the readiness of a Pod.

In many cases, liveness and readiness probes are performing the same checks. However, the presence of a readiness probe gives your container time to start up. Only by passing the readiness check is a Deployment considered to be successful, so that, for example, Pods with an older version can be terminated as part of a rolling update.

For applications that need a very long time to initialize, it's likely that failing liveness checks will cause your container to be restarted before the startup is finished. To prevent these unwanted shutdowns, you can use *startup probes* to indicate when the startup is finished.

Startup Probes

Liveness probes can also be used exclusively to allow for long startup times by stretching the check intervals, increasing the number of retries, and adding a longer delay for the initial liveness probe check. This strategy, however, is not optimal since these timing parameters will also apply for the post-startup phase and will prevent your application from quickly restarting when fatal errors occur.

When applications take minutes to start (for example, Jakarta EE application servers), Kubernetes provides *startup probes*.

Startup probes are configured with the same format as liveness probes but allow for different values for the probe action and the timing parameters. The `periodSeconds` and `failureThreshold` parameters are configured with much larger values compared to the corresponding liveness probes to factor in the longer application startup. Liveness and readiness probes are called only after the startup probe reports success. The container is restarted if the startup probe is not successful within the configured failure threshold.

While the same probe action can be used for liveness and startup probes, a successful startup is often indicated by a marker file that is checked for existence by the startup probe.

Example 4-4 is a typical example of a Jakarta EE application server that takes a long time to start.

Example 4-4. Container with a startup and liveness probe

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-startup-check
spec:
  containers:
  - image: quay.io/wildfly/wildfly ❶
    name: wildfly
    startupProbe:
      exec:
        command: [ "stat", "/opt/jboss/wildfly/standalone/tmp/startup-marker" ] ❷
      initialDelaySeconds: 60 ❸
      periodSeconds: 60
      failureThreshold: 15
    livenessProbe:
      httpGet:
        path: /health
        port: 9990
        periodSeconds: 10 ❹
        failureThreshold: 3
```

❶ JBoss WildFly Jakarta EE server that will take its time to start.

❷ Marker file that is created by WildFly after a successful startup.

❸ Timing parameters that specify that the container should be restarted when it has not been passing the startup probe after 15 minutes (60-second pause until the first check, then maximal 15 checks with 60-second intervals).

❹ Timing parameters for the liveness probes are much smaller, resulting in a restart if subsequent liveness probes fail within 20 seconds (three retries with 10-second pauses between each).

The liveness, readiness, and startup probes are fundamental building blocks of the automation of cloud native applications. Application frameworks such as Quarkus SmallRye Health, Spring Boot Actuator, WildFly Swarm health check, Apache Karaf health check, or the MicroProfile spec for Java provide implementations for offering health probes.

Discussion

To be fully automatable, cloud native applications must be highly observable by providing a means for the managing platform to read and interpret the application health, and if necessary, take corrective actions. Health checks play a fundamental role in the automation of activities such as deployment, self-healing, scaling, and others. However, there are also other means through which your application can provide more visibility about its health.

The obvious and old method for this purpose is through logging. It is a good practice for containers to log any significant events to system out and system error and have these logs collected to a central location for further analysis. Logs are not typically used for taking automated actions but rather to raise alerts and further investigations. A more useful aspect of logs is the postmortem analysis of failures and detection of unnoticeable errors.

Apart from logging to standard streams, it is also a good practice to log the reason for exiting a container to */dev/termination-log*. This location is the place where the container can state its last will before being permanently vanished.[1] Figure 4-1 shows the possible options for how a container can communicate with the runtime platform.

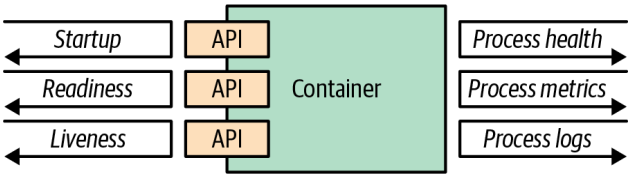


Figure 4-1. Container observability options

Containers provide a unified way for packaging and running applications by treating them like opaque systems. However, any container that is aiming to become a cloud native citizen must provide APIs for the runtime environment to observe the container health and act accordingly. This support is a fundamental prerequisite for automation of the container updates and lifecycle in a unified way, which in turn improves the system's resilience and user experience. In practical terms, that means, as a very minimum, your containerized application must provide APIs for the different kinds of health checks (liveness and readiness).

Even-better-behaving applications must also provide other means for the managing platform to observe the state of the containerized application by integrating with

1 Alternatively, you could change the `.spec.containers.terminationMessagePolicy` field of a Pod to `FallbackToLogsOnError`, in which case the last line of the log is used for the Pod's status message when it terminates.

tracing and metrics-gathering libraries such as OpenTracing or Prometheus. Treat your application as an opaque system, but implement all the necessary APIs to help the platform observe and manage your application in the best way possible.

The next pattern, *Managed Lifecycle*, is also about communication between applications and the Kubernetes management layer, but coming from the other direction. It's about how your application gets informed about important Pod lifecycle events.

More Information

- Health Probe Example
- Configure Liveness, Readiness, and Startup Probes
- Kubernetes Best Practices: Setting Up Health Checks with Readiness and Liveness Probes
- Graceful Shutdown with Node.js and Kubernetes
- Kubernetes Startup Probe—Practical Guide
- Improving Application Availability with Pod Readiness Gates
- Customizing the Termination Message
- SmallRye Health
- Spring Boot Actuator: Production-Ready Features
- Advanced Health Check Patterns in Kubernetes

CHAPTER 5

Managed Lifecycle

Containerized applications managed by cloud native platforms have no control over their lifecycle, and to be good cloud native citizens, they have to listen to the events emitted by the managing platform and adapt their lifecycles accordingly. The *Managed Lifecycle* pattern describes how applications can and should react to these lifecycle events.

Problem

In Chapter 4, "Health Probe", we explained why containers have to provide APIs for the different health checks. Health-check APIs are read-only endpoints the platform is continually probing to get application insight. It is a mechanism for the platform to extract information from the application.

In addition to monitoring the state of a container, the platform sometimes may issue commands and expect the application to react to them. Driven by policies and external factors, a cloud native platform may decide to start or stop the applications it is managing at any moment. It is up to the containerized application to determine which events are important to react to and how to react. But in effect, this is an API that the platform is using to communicate and send commands to the application. Also, applications are free to either benefit from lifecycle management or ignore it if they don't need this service.

Solution

We saw that checking only the process status is not a good enough indication of the health of an application. That is why there are different APIs for monitoring the health of a container. Similarly, using only the process model to run and stop a process is not good enough. Real-world applications require more fine-grained

interactions and lifecycle management capabilities. Some applications need help to warm up, and some applications need a gentle and clean shutdown procedure. For this and other use cases, some events, as shown in Figure 5-1, are emitted by the platform that the container can listen to and react to if desired.

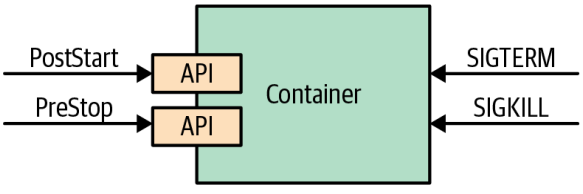


Figure 5-1. Managed container lifecycle

The deployment unit of an application is a Pod. As you already know, a Pod is composed of one or more containers. At the Pod level, there are other constructs such as init containers, which we cover in Chapter 15, "Init Container", that can help manage the container lifecycle. The events and hooks we describe in this chapter are all applied at an individual container level rather than the Pod level.

SIGTERM Signal

Whenever Kubernetes decides to shut down a container, whether that is because the Pod it belongs to is shutting down or simply because a failed liveness probe causes the container to be restarted, the container receives a SIGTERM signal. SIGTERM is a gentle poke for the container to shut down cleanly before Kubernetes sends a more abrupt SIGKILL signal. Once a SIGTERM signal has been received, the application should shut down as quickly as possible. For some applications, this might be a quick termination, and some other applications may have to complete their in-flight requests, release open connections, and clean up temp files, which can take a slightly longer time. In all cases, reacting to SIGTERM is the right moment to shut down a container in a clean way.

SIGKILL Signal

If a container process has not shut down after a SIGTERM signal, it is shut down forcefully by the following SIGKILL signal. Kubernetes does not send the SIGKILL signal immediately but waits 30 seconds by default after it has issued a SIGTERM signal. This grace period can be defined per Pod via the `.spec.terminationGracePeriodSeconds` field, but it cannot be guaranteed as it can be overridden while issuing commands to Kubernetes. The aim should be to design and implement containerized applications to be ephemeral with quick startup and shutdown processes.

PostStart Hook

Using only process signals for managing lifecycles is somewhat limited. That is why additional lifecycle hooks such as `postStart` and `preStop` are provided by Kubernetes. A Pod manifest containing a `postStart` hook looks like the one in Example 5-1.

Example 5-1. A container with `postStart` hook

```
apiVersion: v1
kind: Pod
metadata:
  name: post-start-hook
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    lifecycle:
      postStart:
        exec:
          command:  ❶
          - sh
          - -c
          - sleep 30 && echo "Wake up!" > /tmp/postStart_done
```

❶ The `postStart` command waits 30 seconds. `sleep` is just a simulation for any lengthy startup code that might run at this point. Also, it uses a trigger file to sync with the main application, which starts in parallel.

The `postStart` command is executed after a container is created, asynchronously with the primary container's process. Even if much of the application initialization and warm-up logic can be implemented as part of the container startup steps, `postStart` still covers some use cases. The `postStart` action is a blocking call, and the container status remains *Waiting* until the `postStart` handler completes, which in turn keeps the Pod status in the *Pending* state. This nature of `postStart` can be used to delay the startup state of the container while allowing time for the main container process to initialize.

Another use of `postStart` is to prevent a container from starting when the Pod does not fulfill certain preconditions. For example, when the `postStart` hook indicates an error by returning a nonzero exit code, Kubernetes kills the main container process.

The `postStart` and `preStop` hook invocation mechanisms are similar to the health probes described in Chapter 4, "Health Probe", and support these handler types:

exec
: Runs a command directly in the container

httpGet
: Executes an HTTP GET request against a port opened by one Pod container

You have to be very careful what critical logic you execute in the `postStart` hook as there are no guarantees for its execution. Since the hook is running in parallel with the container process, it is possible that the hook may be executed before the container has started. Also, the hook is intended to have at-least-once semantics, so the implementation has to take care of duplicate executions. Another aspect to keep in mind is that the platform does not perform any retry attempts on failed HTTP requests that didn't reach the handler.

PreStop Hook

The `preStop` hook is a blocking call sent to a container before it is terminated. It has the same semantics as the SIGTERM signal and should be used to initiate a graceful shutdown of the container when reacting to SIGTERM is not possible. The `preStop` action in Example 5-2 must complete before the call to delete the container is sent to the container runtime, which triggers the SIGTERM notification.

Example 5-2. A container with a `preStop` hook

```
apiVersion: v1
kind: Pod
metadata:
  name: pre-stop-hook
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    lifecycle:
      preStop:
        httpGet:  ❶
          path: /shutdown
          port: 8080
```

❶ Call out to a `/shutdown` endpoint running within the application.

Even though `preStop` is blocking, holding on it or returning an unsuccessful result does not prevent the container from being deleted and the process killed. The `preStop` hook is only a convenient alternative to a SIGTERM signal for graceful application shutdown and nothing more. It also offers the same handler types and guarantees as the `postStart` hook we covered previously.

Other Lifecycle Controls

In this chapter, so far we have focused on the hooks that allow you to execute commands when a container lifecycle event occurs. But another mechanism that is not at the container level but at the Pod level allows you to execute initialization instructions.

We describe the *Init Container* pattern in Chapter 15 in depth, but here we describe it briefly to compare it with lifecycle hooks. Unlike regular application containers, init containers run sequentially, run until completion, and run before any of the application containers in a Pod start up. These guarantees allow you to use init containers for Pod-level initialization tasks. Both lifecycle hooks and init containers operate at a different granularity (at the container level and Pod level, respectively) and can be used interchangeably in some instances, or complement one another in other cases. Table 5-1 summarizes the main differences between the two.

Table 5-1. Lifecycle hooks and init containers

Aspect	Lifecycle hooks	Init containers
Activates on	Container lifecycle phases.	Pod lifecycle phases.
Startup phase action	A `postStart` command.	A list of `initContainers` to execute.
Shutdown phase action	A `preStop` command.	No equivalent feature.
Timing guarantees	A `postStart` command is executed at the same time as the container's `ENTRY POINT`.	All init containers must be completed successfully before any application container can start.
Use cases	Perform noncritical startup/shutdown cleanups specific to a container.	Perform workflow-like sequential operations using containers; reuse containers for task executions.

If even more control is required to manage the lifecycle of your application containers, there is an advanced technique for rewriting the container entrypoints, sometimes also referred to as the *Commandlet* pattern. This pattern is especially useful when the main containers within a Pod have to be started in a certain order and need an extra level of control. Kubernetes-based pipeline platforms like Tekton and Argo CD require the sequential execution of containers that share data and support the inclusion of additional sidecar containers running in parallel (we talk more about sidecars in Chapter 16, "Sidecar").

For these scenarios, a sequence of init containers is not good enough because init containers don't allow sidecars. As an alternative, an advanced technique called *entrypoint rewriting* can be used to allow fine-grained lifecycle control for the Pod's main containers. Every container image defines a command that is executed by default when the container starts. In a Pod specification, you can also define this command directly in the Pod spec. The idea of entrypoint rewriting is to replace this command with a generic wrapper command that calls the original command and takes care of lifecycle concerns. This generic command is injected from another container image before the application container starts.

This concept is best explained by an example. Example 5-3 shows a typical Pod declaration that starts a single container with the given arguments.

Example 5-3. Simple Pod starting an image with a command and arguments

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-random-generator
spec:
  restartPolicy: OnFailure
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    command:
    - "random-generator-runner"  ❶
    args:                        ❷
    - "--seed"
    - "42"
```

❶ The command executed when the container starts.

❷ Additional arguments provided to the entrypoint command.

The trick is now to wrap the given command `random-generator-runner` with a generic supervisor program that takes care of lifecycle aspects, like reacting on `SIGTERM` or other external signals. Example 5-4 demonstrates a Pod declaration that includes an init container for installing a supervisor, which is then started to monitor the main application.

Example 5-4. Pod that wraps the original entrypoint with a supervisor

```
apiVersion: v1
kind: Pod
metadata:
  name: wrapped-random-generator
spec:
  restartPolicy: OnFailure
  volumes:
  - name: wrapper                          ❶
    emptyDir: { }
  initContainers:
  - name: copy-supervisor                  ❷
    image: k8spatterns/supervisor
    volumeMounts:
    - mountPath: /var/run/wrapper
      name: wrapper
    command: [ cp ]
    args: [ supervisor, /var/run/wrapper/supervisor ]
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    volumeMounts:
    - mountPath: /var/run/wrapper
      name: wrapper
    command:
    - "/var/run/wrapper/supervisor"        ❸
    args:                                  ❹
    - "random-generator-runner"
    - "--seed"
    - "42"
```

❶ A fresh `emptyDir` volume is created to share the supervisor daemon.

❷ Init container used for copying the supervisor daemon to the application containers.

❸ The original command `randomGenerator` as defined in Example 5-3 is replaced with supervisor daemon from the shared volume.

❹ The original command specification becomes the arguments for the supervisor commands.

This entrypoint rewriting is especially useful for Kubernetes-based applications that create and manage Pods programmatically, like Tekton, which creates Pods when running a continuous integration (CI) pipeline. That way, they gain much better control of when to start, stop, or chain containers within a Pod.

There are no strict rules about which mechanism to use except when you require a specific timing guarantee. We could skip lifecycle hooks and init containers entirely and use a bash script to perform specific actions as part of a container's startup or shutdown commands. That is possible, but it would tightly couple the container with the script and turn it into a maintenance nightmare. We could also use Kubernetes lifecycle hooks to perform some actions, as described in this chapter. Alternatively, we could go even further and run containers that perform individual actions using init containers or inject supervisor daemons for even more sophisticated control. In this sequence, the options require increasingly more effort, but at the same time offer stronger guarantees and enable reuse.

Understanding the stages and available hooks of containers and Pod lifecycles is crucial for creating applications that benefit from being managed by Kubernetes.

Discussion

One of the main benefits the cloud native platform provides is the ability to run and scale applications reliably and predictably on top of potentially unreliable cloud infrastructure. These platforms provide a set of constraints and contracts for an application running on them. It is in the interest of the application to honor these contracts to benefit from all of the capabilities offered by the cloud native platform. Handling and reacting to these events ensures that your application can gracefully start up and shut down with minimal impact on the consuming services. At the moment, in its basic form, that means the containers should behave as any well-designed POSIX process should. In the future, there might be even more events giving hints to the application when it is about to be scaled up or asked to release resources to prevent being shut down. It is essential to understand that the application lifecycle is no longer in the control of a person but is fully automated by the platform.

Besides managing the application lifecycle, the other big duty of orchestration platforms like Kubernetes is to distribute containers over a fleet of nodes. The next pattern, *Automated Placement*, explains the options to influence the scheduling decisions from the outside.

More Information

- Managed Lifecycle Example
- Container Lifecycle Hooks
- Attach Handlers to Container Lifecycle Events
- Kubernetes Best Practices: Terminating with Grace
- Graceful Shutdown of Pods with Kubernetes
- Argo and Tekton: Pushing the Boundaries of the Possible on Kubernetes
- Russian Doll: Extending Containers with Nested Processes

CHAPTER 6

Automated Placement

Automated Placement is the core function of the Kubernetes scheduler for assigning new Pods to nodes that match container resource requests and honor scheduling policies. This pattern describes the principles of the Kubernetes scheduling algorithm and how to influence the placement decisions from the outside.

Problem

A reasonably sized microservices-based system consists of tens or even hundreds of isolated processes. Containers and Pods do provide nice abstractions for packaging and deployment but do not solve the problem of placing these processes on suitable nodes. With a large and ever-growing number of microservices, assigning and placing them individually to nodes is not a manageable activity.

Containers have dependencies among themselves, dependencies to nodes, and resource demands, and all of that changes over time too. The resources available on a cluster also vary over time, through shrinking or extending the cluster or by having it consumed by already-placed containers. The way we place containers impacts the availability, performance, and capacity of the distributed systems as well. All of that makes scheduling containers to nodes a moving target.

Solution

In Kubernetes, assigning Pods to nodes is done by the scheduler. It is a part of Kubernetes that is highly configurable, and it is still evolving and improving. In this chapter, we cover the main scheduling control mechanisms, driving forces that affect the placement, why to choose one or the other option, and the resulting consequences. The Kubernetes scheduler is a potent and time-saving tool. It plays a fundamental role in the Kubernetes platform as a whole, but similar to other

Kubernetes components (API Server, Kubelet), it can be run in isolation or not used at all.

At a very high level, the main operation the Kubernetes scheduler performs is to retrieve each newly created Pod definition from the API Server and assign it to a node. It finds the most suitable node for every Pod (as long as there is such a node), whether that is for the initial application placement, scaling up, or when moving an application from an unhealthy node to a healthier one. It does this by considering runtime dependencies, resource requirements, and guiding policies for high availability; by spreading Pods horizontally; and also by colocating Pods nearby for performance and low-latency interactions. However, for the scheduler to do its job correctly and allow declarative placement, it needs nodes with available capacity and containers with declared resource profiles and guiding policies in place. Let's look at each of these in more detail.

Available Node Resources

First of all, the Kubernetes cluster needs to have nodes with enough resource capacity to run new Pods. Every node has capacity available for running Pods, and the scheduler ensures that the sum of the container resources requested for a Pod is less than the available allocatable node capacity. Considering a node dedicated only to Kubernetes, its capacity is calculated using the following formula in Example 6-1.

Example 6-1. Node capacity

```
Allocatable [capacity for application pods] =
    Node Capacity [available capacity on a node]
        - Kube-Reserved [Kubernetes daemons like kubelet, container runtime]
        - System-Reserved [Operating System daemons like sshd, udev]
        - Eviction Thresholds [Reserved memory to prevent system OOMs]
```

If you don't reserve resources for system daemons that power the OS and Kubernetes itself, the Pods can be scheduled up to the full capacity of the node, which may cause Pods and system daemons to compete for resources, leading to resource starvation issues on the node. Even then, memory pressure on the node can affect all Pods running on it through OOMKilled errors or cause the node to go temporarily offline. OOMKilled is an error message displayed when the Linux kernel's Out-of-Memory (OOM) killer terminates a process because the system is out of memory. Eviction thresholds are the last resort for the Kubelet to reserve memory on the node and attempt to evict Pods when the available memory drops below the reserved value.

Also keep in mind that if containers are running on a node that is not managed by Kubernetes, the resources used by these containers are not reflected in the node capacity calculations by Kubernetes. A workaround is to run a placeholder Pod that doesn't do anything but has only resource requests for CPU and memory

corresponding to the untracked containers' resource use amount. Such a Pod is created only to represent and reserve the resource consumption of the untracked containers and helps the scheduler build a better resource model of the node.

Container Resource Demands

Another important requirement for an efficient Pod placement is to define the containers' runtime dependencies and resource demands. We covered that in more detail in Chapter 2, "Predictable Demands". It boils down to having containers that declare their resource profiles (with `request` and `limit`) and environment dependencies such as storage or ports. Only then are Pods optimally assigned to nodes and can run without affecting one another and facing resource starvation during peak usage.

Scheduler Configurations

The next piece of the puzzle is having the right filtering or priority configurations for your cluster needs. The scheduler has a default set of predicate and priority policies configured that is good enough for most use cases. In Kubernetes versions before v1.23, a scheduling policy can be used to configure the predicates and priorities of a scheduler. Newer versions of Kubernetes moved to scheduling profiles to achieve the same effect. This new approach exposes the different steps of the scheduling process as an extension point and allows you to configure plugins that override the default implementations of the steps. Example 6-2 demonstrates how to override the `PodTopologySpread` plugin from the `score` step with custom plugins.

Example 6-2. A scheduler configuration

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
profiles:
  - plugins:
      score:                          ❶
        disabled:
        - name: PodTopologySpread     ❷
        enabled:
        - name: MyCustomPlugin        ❸
          weight: 2
```

❶ The plugins in this phase provide a score to each node that has passed the filtering phase.

❷ This plugin implements topology spread constraints that we will see later in the chapter.

❸ The disabled plugin in the previous step is replaced by a new one.

Scheduler plugins and custom schedulers should be defined only by an administrator as part of the cluster configuration. As a regular user deploying applications on a cluster, you can just refer to predefined schedulers.

By default, the scheduler uses the default-scheduler profile with default plugins. It is also possible to run multiple schedulers on the cluster, or multiple profiles on the scheduler, and allow Pods to specify which profile to use. Each profile must have a unique name. Then when defining a Pod, you can add the field `.spec.schedulerName` with the name of your profile to the Pod specification, and the Pod will be processed by the desired scheduler profile.

Scheduling Process

Pods get assigned to nodes with certain capacities based on placement policies. For completeness, Figure 6-1 visualizes at a high level how these elements get together and the main steps a Pod goes through when being scheduled.

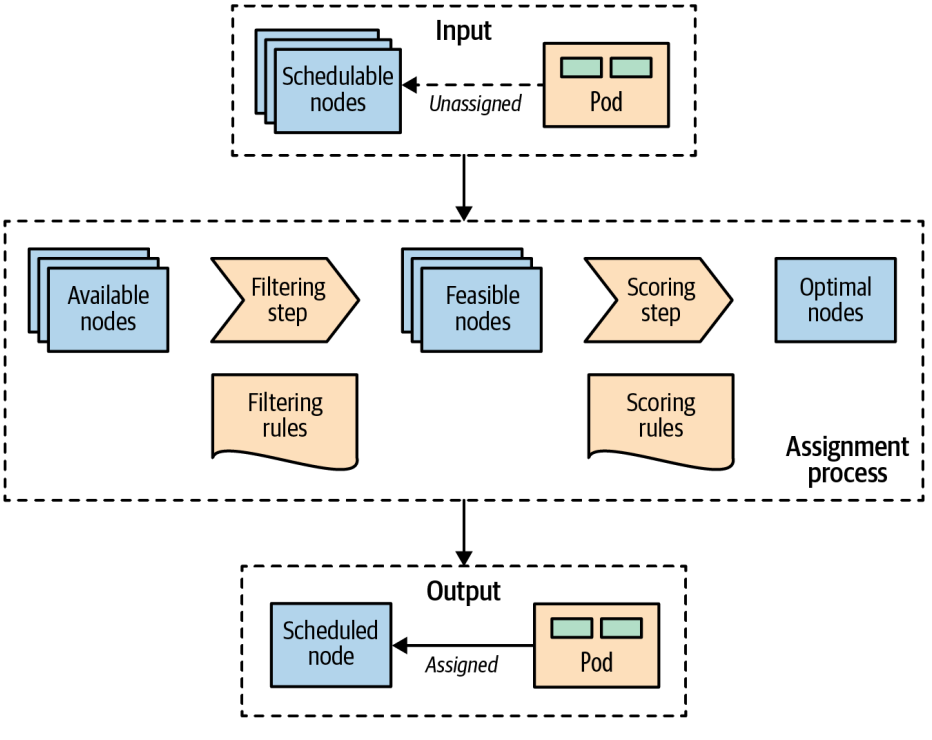


Figure 6-1. A Pod-to-node assignment process

As soon as a Pod is created that is not assigned to a node yet, it gets picked by the scheduler together with all the available nodes and the set of filtering and priority policies. In the first stage, the scheduler applies the filtering policies and removes all nodes that do not qualify. Nodes that meet the Pod's scheduling requirements are called *feasible nodes*. In the second stage, the scheduler runs a set of functions to score the remaining feasible nodes and orders them by weight. In the last stage, the scheduler notifies the API server about the assignment decision, which is the primary outcome of the scheduling process. This whole process is also referred to as *scheduling*, *placement*, *node assignment*, or *binding*.

In most cases, it is better to let the scheduler do the Pod-to-node assignment and not micromanage the placement logic. However, on some occasions, you may want to force the assignment of a Pod to a specific node or group of nodes. This assignment can be done using a node selector. The `.spec.nodeSelector` Pod field specifies a map of key-value pairs that must be present as labels on the node for the node to be eligible to run the Pod. For example, let's say you want to force a Pod to run on a specific node where you have SSD storage or GPU acceleration hardware. With the Pod definition in Example 6-3 that has `nodeSelector` matching `disktype: ssd`, only nodes that are labeled with `disktype=ssd` will be eligible to run the Pod.

Example 6-3. Node selector based on type of disk available

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
  nodeSelector:
    disktype: ssd       ❶
```

❶ Set of node labels a node must match to be considered the node of this Pod.

In addition to specifying custom labels to your nodes, you can use some of the default labels that are present on every node. Every node has a unique `kubernetes.io/hostname` label that can be used to place a Pod on a node by its hostname. Other default labels that indicate the OS, architecture, and instance type can be useful for placement too.

Node Affinity

Kubernetes supports many more flexible ways to configure the scheduling processes. One such feature is *node affinity*, which is a more expressive way of the node selector approach described previously that allows specifying rules as either required or preferred. *Required rules* must be met for a Pod to be scheduled to a node, whereas preferred rules only imply preference by increasing the weight for the matching nodes without making them mandatory. In addition, the node affinity feature greatly expands the types of constraints you can express by making the language more expressive with operators such as `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt`, or `Lt`. Example 6-4 demonstrates how node affinity is declared.

Example 6-4. Pod with node affinity

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:    ❶
        nodeSelectorTerms:
        - matchExpressions:                              ❷
          - key: numberCores
            operator: Gt
            values: [ "3" ]
      preferredDuringSchedulingIgnoredDuringExecution:   ❸
      - weight: 1
        preference:
          matchFields:
          - key: metadata.name
            operator: NotIn
            values: [ "control-plane-node" ]
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
```

❶ Hard requirement that the node must have more than three cores (indicated by a node label) to be considered in the scheduling process. The rule is not reevaluated during execution if the conditions on the node change.

❷ Match on labels. In this example, all nodes are matched that have a label `numberCores` with a value greater than 3.

❸ Soft requirements, which is a list of selectors with weights. For every node, the sum of all weights for matching selectors is calculated, and the highest-valued node is chosen, as long as it matches the hard requirement.

Pod Affinity and Anti-Affinity

Pod affinity is a more powerful way of scheduling and should be used when `nodeSelector` is not enough. This mechanism allows you to constrain which nodes a Pod can run based on label or field matching. It doesn't allow you to express dependencies among Pods to dictate where a Pod should be placed relative to other Pods. To express how Pods should be spread to achieve high availability, or be packed and colocated together to improve latency, you can use Pod affinity and anti-affinity.

Node affinity works at node granularity, but Pod affinity is not limited to nodes and can express rules at various topology levels based on the Pods already running on a node. Using the `topologyKey` field, and the matching labels, it is possible to enforce more fine-grained rules, which combine rules on domains like node, rack, cloud provider zone, and region, as demonstrated in Example 6-5.

Example 6-5. Pod with Pod affinity

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:  ❶
      - labelSelector:                                 ❷
          matchLabels:
            confidential: high
        topologyKey: security-zone                     ❸
    podAntiAffinity:                                   ❹
      preferredDuringSchedulingIgnoredDuringExecution: ❺
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchLabels:
              confidential: none
          topologyKey: kubernetes.io/hostname
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
```

❶ Required rules for the Pod placement concerning other Pods running on the target node.

❷ Label selector to find the Pods to be colocated with.

❸ The nodes on which Pods with labels `confidential=high` are running are supposed to carry a `security-zone` label. The Pod defined here is scheduled to a node with the same label and value.

❹ Anti-affinity rules to find nodes where a Pod would *not* be placed.

❺ Rule describing that the Pod should not (but could) be placed on any node where a Pod with the label `confidential=none` is running.

Similar to node affinity, there are hard and soft requirements for Pod affinity and anti-affinity, called `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution`, respectively. Again, as with node affinity, the `IgnoredDuringExecution` suffix is in the field name, which exists for future extensibility reasons. At the moment, if the labels on the node change and affinity rules are no longer valid, the Pods continue running,[1] but in the future, runtime changes may also be taken into account.

Topology Spread Constraints

Pod affinity rules allow the placement of unlimited Pods to a single topology, whereas Pod anti-affinity disallows Pods to colocate in the same topology. Topology spread constraints give you more fine-grained control to evenly distribute Pods on your cluster and achieve better cluster utilization or high availability of applications.

Let's look at an example to understand how topology spread constraints can help. Let's suppose we have an application with two replicas and a two-node cluster. To avoid downtime and a single point of failure, we can use Pod anti-affinity rules to prevent the coexistence of the Pods on the same node and spread them into both nodes. While this setup makes sense, it will prevent you from performing rolling upgrades because the third replacement Pod cannot be placed on the existing nodes because of the Pod anti-affinity constraints. We will have to either add another node or change the Deployment strategy from rolling to recreate. Topology spread constraints would be a better solution in this situation as they allow you to tolerate some degree of uneven Pod distribution when the cluster is running out of resources. Example 6-6 allows the placement of the third rolling deployment Pod on one of the two nodes because it allows imbalances—i.e., a skew of one Pod.

1 However, if node labels change and allow for unscheduled Pods to match their node affinity selector, these Pods are scheduled on this node.

Example 6-6. Pod with topology spread constraints

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
  labels:
    app: bar
spec:
  topologySpreadConstraints:                    ❶
  - maxSkew: 1                                  ❷
    topologyKey: topology.kubernetes.io/zone    ❸
    whenUnsatisfiable: DoNotSchedule            ❹
    labelSelector:                              ❺
      matchLabels:
        app: bar
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
```

❶ Topology spread constraints are defined in the `topologySpreadConstraints` field of the Pod spec.

❷ `maxSkew` defines the maximum degree to which Pods can be unevenly distributed in the topology.

❸ A topology domain is a logical unit of your infrastructure. And a `topologyKey` is the key of the Node label where identical values are considered to be in the same topology.

❹ The `whenUnsatisfiable` field defines what action should be taken when `maxSkew` can't be satisfied. `DoNotSchedule` is a hard constraint preventing the scheduling of Pods, whereas `ScheduleAnyway` is a soft constraint that gives scheduling priority to nodes that reduce cluster imbalance.

❺ `labelSelector` Pods that match this selector are grouped together and counted when spreading them to satisfy the constraint.

Topology spread constraints is a feature that is still evolving at the time of this writing. Built-in cluster-level topology spread constraints allow certain imbalances based on default Kubernetes labels and give you the ability to honor or ignore node affinity and taint policies.

Taints and Tolerations

A more advanced feature that controls where Pods can be scheduled and allowed to run is based on taints and tolerations. While node affinity is a property of Pods that allows them to choose nodes, taints and tolerations are the opposite. They allow the nodes to control which Pods should or should not be scheduled on them. A *taint* is a characteristic of the node, and when it is present, it prevents Pods from scheduling onto the node unless the Pod has toleration for the taint. In that sense, taints and tolerations can be considered an *opt-in* to allow scheduling on nodes that by default are not available for scheduling, whereas affinity rules are an *opt-out* by explicitly selecting on which nodes to run and thus exclude all the nonselected nodes.

A taint is added to a node by using `kubectl`: `kubectl taint nodes control-plane-node node-role.kubernetes.io/control-plane="true":NoSchedule`, which has the effect shown in Example 6-7. A matching toleration is added to a Pod as shown in Example 6-8. Notice that the values for `key` and `effect` in the `taints` section of Example 6-7 and the `tolerations` section in Example 6-8 are the same.

Example 6-7. Tainted node

```
apiVersion: v1
kind: Node
metadata:
  name: control-plane-node
spec:
  taints:                                    ❶
  - effect: NoSchedule
    key: node-role.kubernetes.io/control-plane
    value: "true"
```

❶ Mark this node as unschedulable except when a Pod tolerates this taint.

Example 6-8. Pod tolerating node taints

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
  tolerations:
  - key: node-role.kubernetes.io/control-plane ❶
    operator: Exists
    effect: NoSchedule                         ❷
```

❶ Tolerate (i.e., consider for scheduling) nodes, which have a taint with key `node-role.kubernetes.io/control-plane`. On production clusters, this taint is set on the control plane node to prevent scheduling of Pods on this node. A toleration like this allows this Pod to be installed on the control plane node nevertheless.

❷ Tolerate only when the taint specifies a `NoSchedule` effect. This field can be empty here, in which case the toleration applies to every effect.

There are hard taints that prevent scheduling on a node (`effect=NoSchedule`), soft taints that try to avoid scheduling on a node (`effect=PreferNoSchedule`), and taints that can evict already-running Pods from a node (`effect=NoExecute`).

Taints and tolerations allow for complex use cases like having dedicated nodes for an exclusive set of Pods, or force eviction of Pods from problematic nodes by tainting those nodes.

You can influence the placement based on the application's high availability and performance needs, but try not to limit the scheduler too much and back yourself into a corner where no more Pods can be scheduled and there are too many stranded resources. For example, if your containers' resource requirements are too coarse-grained, or nodes are too small, you may end up with stranded resources in nodes that are not utilized.

In Figure 6-2, we can see node A has 4 GB of memory that cannot be utilized as there is no CPU left to place other containers. Creating containers with smaller resource requirements may help improve this situation. Another solution is to use the Kubernetes *descheduler*, which helps defragment nodes and improve their utilization.

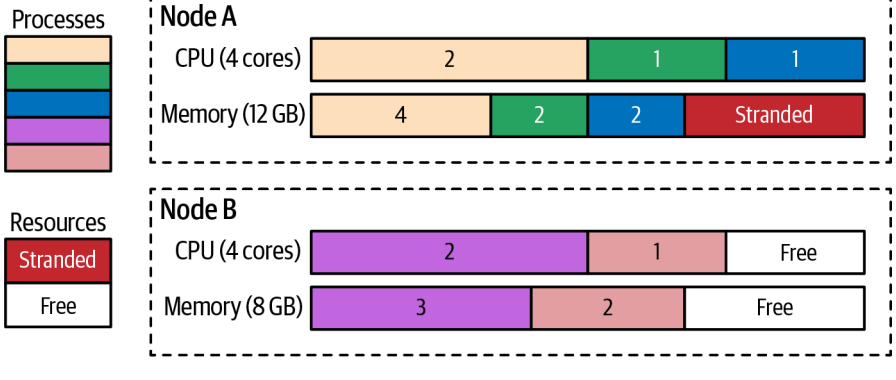


Figure 6-2. Processes scheduled to nodes and stranded resources

Once a Pod is assigned to a node, the job of the scheduler is done, and it does not change the placement of the Pod unless the Pod is deleted and recreated without a node assignment. As you have seen, with time, this can lead to resource fragmentation and poor utilization of cluster resources. Another potential issue is that the scheduler decisions are based on its cluster view at the point in time when a new Pod is scheduled. If a cluster is dynamic and the resource profile of the nodes changes or new nodes are added, the scheduler will not rectify its previous Pod placements. Apart from changing the node capacity, you may also alter the labels on the nodes that affect placement, but past placements are not rectified.

All of these scenarios can be addressed by the descheduler. The Kubernetes descheduler is an optional feature that is typically run as a Job whenever a cluster administrator decides it is a good time to tidy up and defragment a cluster by rescheduling the Pods. The descheduler comes with some predefined policies that can be enabled and tuned or disabled.

Regardless of the policy used, the descheduler avoids evicting the following:

- Node- or cluster-critical Pods
- Pods not managed by a ReplicaSet, Deployment, or Job, as these Pods cannot be recreated
- Pods managed by a DaemonSet
- Pods that have local storage
- Pods with PodDisruptionBudget, where eviction would violate its rules
- Pods that have a non-nil `DeletionTimestamp` field set
- Deschedule Pod itself (achieved by marking itself as a critical Pod)

Of course, all evictions respect Pods' QoS levels by choosing *Best-Efforts* Pods first, then *Burstable* Pods, and finally *Guaranteed* Pods as candidates for eviction. See Chapter 2, "Predictable Demands", for a detailed explanation of these QoS levels.

Discussion

Placement is the art of assigning Pods to nodes. You want to have as minimal intervention as possible, as the combination of multiple configurations can be hard to predict. In simpler scenarios, scheduling Pods based on resource constraints should be sufficient. If you follow the guidelines from Chapter 2, "Predictable Demands", and declare all the resource needs of a container, the scheduler will do its job and place the Pod on the most feasible node possible.

However, in more realistic scenarios, you may want to schedule Pods to specific nodes according to other constraints such as data locality, Pod colocality, application

high availability, and efficient cluster resource utilization. In these cases, there are multiple ways to steer the scheduler toward the desired deployment topology.

Figure 6-3 shows one approach to thinking and making sense of the different scheduling techniques in Kubernetes.

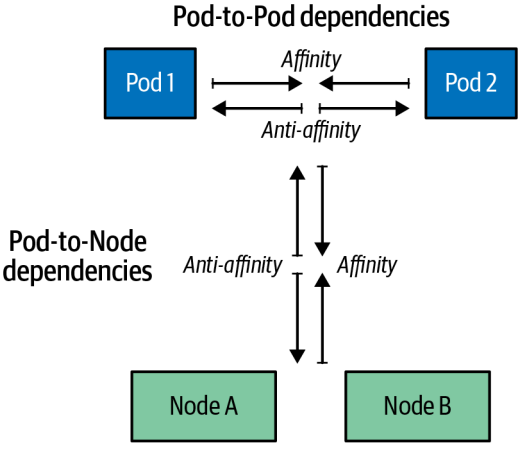


Figure 6-3. Pod-to-Pod and Pod-to-Node and dependencies

Start by identifying the forces and dependencies between the Pod and the nodes (for example, based on dedicated hardware capabilities or efficient resource utilization). Use the following node affinity techniques to direct the Pod to the desired nodes, or use anti-affinity techniques to steer the Pod away from the undesired nodes:

nodeName
: This field provides the simplest form of hard wiring a Pod to a node. This field should ideally be populated by the scheduler, which is driven by policies rather than manual node assignment. Assigning a Pod to a node through this approach prevents the scheduling of the Pod to any other node. If the named node has no capacity, or the node doesn't exist, the Pod will never run. This throws us back into the pre-Kubernetes era, when we explicitly needed to specify the nodes to run our applications. Setting this field manually is not a Kubernetes best practice and should be used only as an exception.

nodeSelector
: A node selector is a label map. For the Pod to be eligible to run on a node, the Pod must have the indicated key-value pairs as the label on the node. Having put some meaningful labels on the Pod and the node (which you should do anyway), a node selector is one of the simplest recommended mechanisms for controlling the scheduler choices.

Node affinity
: This rule improves the manual node assignment approaches and allows a Pod to express dependency toward nodes using logical operators and constraints that provides fine-grained control. It also offers soft and hard scheduling requirements that control the strictness of node affinity constraints.

Taints and tolerations
: Taints and tolerations allow the node to control which Pods should or should not be scheduled on them without modifying existing Pods. By default, Pods that don't have tolerations for the node taint will be rejected or evicted from the node. Another advantage of taints and tolerations is that if you expand the Kubernetes cluster by adding new nodes with new labels, you don't need to add the new labels on all Pods but only on those that should be placed on the new nodes.

Once the desired correlation between a Pod and the nodes is expressed in Kubernetes terms, identify the dependencies between different Pods. Use Pod affinity techniques for Pod colocation for tightly coupled applications, and use Pod anti-affinity techniques to spread Pods on nodes and avoid a single point of failure:

Pod affinity and anti-affinity
: These rules allow scheduling based on Pods' dependencies on other Pods rather than nodes. Affinity rules help for colocating tightly coupled application stacks composed of multiple Pods on the same topology for low-latency and data locality requirements. The anti-affinity rule, on the other hand, can spread Pods across your cluster among failure domains to avoid a single point of failure, or prevent resource-intensive Pods from competing for resources by avoiding placing them on the same node.

Topology spread constraints
: To use these features, platform administrators have to label nodes and provide topology information such as regions, zones, or other user-defined domains. Then, a workload author creating the Pod configurations must be aware of the underlying cluster topology and specify the topology spread constraints. You can also specify multiple topology spread constraints, but all of them must be satisfied for a Pod to be placed. You must ensure that they do not conflict with one another. You can also combine this feature with NodeAffinity and NodeSelector to filter nodes where evenness should be applied. In that case, be sure to understand the difference: multiple topology spread constraints are about calculating the result set independently and producing an AND-joined result, while combining it with NodeAffinity and NodeSelector, on the other hand, filters results of node constraints.

In some scenarios, all of these scheduling configurations might not be flexible enough to express bespoke scheduling requirements. In that case, you may have to customize and tune the scheduler configuration or even provide a custom scheduler implementation that can understand your custom needs:

Scheduler tuning
: The default scheduler is responsible for the placement of new Pods onto nodes within the cluster, and it does it well. However, it is possible to alter one or more stages in the filtering and prioritization phases. This mechanism with extension points and plugins is specifically designed to allow small alterations without the need for a completely new scheduler implementation.

Custom scheduler
: If none of the preceding approaches is good enough, or if you have complex scheduling requirements, you can also write your own custom scheduler. A custom scheduler can run instead of, or alongside, the standard Kubernetes scheduler. A hybrid approach is to have a "scheduler extender" process that the standard Kubernetes scheduler calls out to as a final pass when making scheduling decisions. This way, you don't have to implement a full scheduler but only provide HTTP APIs to filter and prioritize nodes. The advantage of having your scheduler is that you can consider factors outside of the Kubernetes cluster like hardware cost, network latency, and better utilization while assigning Pods to nodes. You can also use multiple custom schedulers alongside the default scheduler and configure which scheduler to use for each Pod. Each scheduler could have a different set of policies dedicated to a subset of the Pods.

To sum up, there are lots of ways to control the Pod placement, and choosing the right approach or combining multiple approaches can be overwhelming. The takeaway from this chapter is this: size and declare container resource profiles, and label Pods and nodes for the best resource-consumption-driven scheduling results. If that doesn't deliver the desired scheduling outcome, start with small and iterative changes. Strive for a minimal policy-based influence on the Kubernetes scheduler to express node dependencies and then inter-Pod dependencies.

More Information

- Automated Placement Example
- Assigning Pods to Nodes
- Scheduler Configuration
- Pod Topology Spread Constraints
- Configure Multiple Schedulers
- Descheduler for Kubernetes

- Disruptions
- Guaranteed Scheduling for Critical Add-On Pods
- Keep Your Kubernetes Cluster Balanced: The Secret to High Availability
- Advanced Kubernetes Pod to Node Scheduling

PART II
Behavioral Patterns

The patterns in this category are focused on the communications and interactions between the Pods and the managing platform. Depending on the type of managing controller used, a Pod may run until completion or be scheduled to run periodically. It can run as a daemon or ensure uniqueness guarantees to its replicas. There are different ways to run a Pod on Kubernetes, and picking the right Pod-management primitives requires understanding their behavior. In the following chapters, we explore the patterns:

- Chapter 7, "Batch Job", describes how to isolate an atomic unit of work and run it until completion.
- Chapter 8, "Periodic Job", allows the execution of a unit of work to be triggered by a temporal event.
- Chapter 9, "Daemon Service", allows you to run infrastructure-focused Pods on specific nodes, before application Pods are placed.
- Chapter 10, "Singleton Service", ensures that only one instance of a service is active at a time and still remains highly available.
- Chapter 11, "Stateless Service", describes the building blocks used for managing identical application instances.
- Chapter 12, "Stateful Service", is all about how to create and manage distributed stateful applications with Kubernetes.

- Chapter 13, "Service Discovery", explains how client services can discover and consume the instances of providing services.
- Chapter 14, "Self Awareness", describes mechanisms for introspection and metadata injection into applications.

CHAPTER 7

Batch Job

The *Batch Job* pattern is suited for managing isolated atomic units of work. It is based on the Job resource, which runs short-lived Pods reliably until completion on a distributed environment.

Problem

The main primitive in Kubernetes for managing and running containers is the Pod. There are different ways of creating Pods with varying characteristics:

Bare Pod
: It is possible to create a Pod manually to run containers. However, when the node such a Pod is running on fails, the Pod is not restarted. Running Pods this way is discouraged except for development or testing purposes. This mechanism is also known as *unmanaged* or *naked Pods*.

ReplicaSet
: This controller is used for creating and managing the lifecycle of Pods expected to run continuously (e.g., to run a web server container). It maintains a stable set of replica Pods running at any given time and guarantees the availability of a specified number of identical Pods. ReplicaSets are described in detail in Chapter 11, "Stateless Service".

DaemonSet
: This controller runs a single Pod on every node and is used for managing platform capabilities such as monitoring, log aggregation, storage containers, and others. See Chapter 9, "Daemon Service", for a more detailed discussion.

A common aspect of these Pods is that they represent long-running processes that are not meant to stop after a certain time. However, in some cases there is a need to perform a predefined finite unit of work reliably and then shut down the container. For this task, Kubernetes provides the Job resource.

Solution

A Kubernetes Job is similar to a ReplicaSet as it creates one or more Pods and ensures they run successfully. However, the difference is that, once the expected number of Pods terminate successfully, the Job is considered complete, and no additional Pods are started. A Job definition looks like Example 7-1.

Example 7-1. A Job specification

```
apiVersion: batch/v1
kind: Job
metadata:
  name: random-generator
spec:
  completions: 5                ❶
  parallelism: 2                ❷
  ttlSecondsAfterFinished: 300  ❸
  template:
    metadata:
      name: random-generator
    spec:
      restartPolicy: OnFailure  ❹
      containers:
      - image: k8spatterns/random-generator:1.0
        name: random-generator
        command: [ "java", "RandomRunner", "/numbers.txt", "10000" ]
```

❶ Job should run five Pods to completion, which all must succeed.

❷ Two Pods can run in parallel.

❸ Keep Pods for five minutes (300 seconds) before garbage-collecting them.

❹ Specifying the `restartPolicy` is mandatory for a Job. The possible values are `OnFailure` or `Never`.

One crucial difference between the Job and the ReplicaSet definition is the `.spec.template.spec.restartPolicy`. The default value for a ReplicaSet is `Always`, which makes sense for long-running processes that must always be kept running. The value `Always` is not allowed for a Job, and the only possible options are `OnFailure` or `Never`.

So why bother creating a Job to run a Pod only once instead of using bare Pods? Using Jobs provides many reliability and scalability benefits that make them the preferred option:

- A Job is not an ephemeral in-memory task but a persisted one that survives cluster restarts.
- When a Job is completed, it is not deleted but is kept for tracking purposes. The Pods that are created as part of the Job are also not deleted but are available for examination (e.g., to check the container logs). This is also true for bare Pods but only for `restartPolicy: OnFailure`. You can still remove the Pods of a Job after a certain time by specifying `.spec.ttlSecondsAfterFinished`.
- A Job may need to be performed multiple times. Using the `.spec.completions` field, it is possible to specify how many times a Pod should complete successfully before the Job itself is done.
- When a Job has to be completed multiple times, it can also be scaled and executed by starting multiple Pods at the same time. That can be done by specifying the `.spec.parallelism` field.
- A Job can be suspended by setting the field `.spec.suspend` to `true`. In this case, all active Pods are deleted and restarted if the Job is resumed (i.e., `.spec.suspend` set to `false` by the user).
- If the node fails or when the Pod is evicted for some reason while still running, the scheduler places the Pod on a new healthy node and reruns it. Bare Pods would remain in a failed state as existing Pods are never moved to other nodes.

All of this makes the Job primitive attractive for scenarios requiring some guarantees for the completion of a unit of work.

The following two fields play major roles in the behavior of a Job:

`.spec.completions`
: Specifies how many Pods should run to complete a Job.

`.spec.parallelism`
: Specifies how many Pod replicas could run in parallel. Setting a high number does not guarantee a high level of parallelism, and the actual number of Pods may still be fewer (and in some corner cases, more) than the desired number (e.g., because of throttling, resource quotas, not enough completions left, and other reasons). Setting this field to 0 effectively pauses the Job.

Figure 7-1 shows how the Job defined in Example 7-1 with a completion count of 5 and a parallelism of 2 is processed.

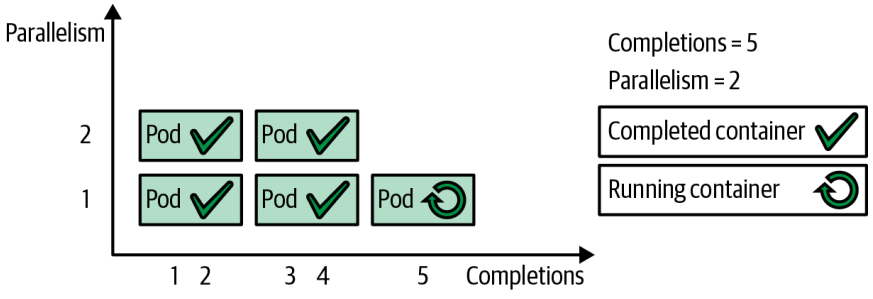


Figure 7-1. Parallel Batch Job with a fixed completion count

Based on these two parameters, there are the following types of Jobs:

Single Pod Jobs
: This type is selected when you leave out both `.spec.completions` and `.spec.parallelism` or set them to their default values of 1. Such a Job starts only one Pod and is completed as soon as the single Pod terminates successfully (with exit code 0).

Fixed completion count Jobs
: For a fixed completion count Job, you should set `.spec.completions` to the number of completions needed. You can set `.spec.parallelism`, or leave it unset and it will default to 1. Such a Job is considered completed after the `.spec.completions` number of Pods has completed successfully. Example 7-1 shows this mode in action and is the best choice when we know the number of work items in advance and the processing cost of a single work item justifies the use of a dedicated Pod.

Work queue Jobs
: For a work queue Job, you need to leave `.spec.completions` unset, and set `.spec.parallelism` to a number greater than one. A work queue Job is considered completed when at least one Pod has terminated successfully and all other Pods have terminated too. This setup requires the Pods to coordinate among themselves and determine what each one is working on so that they can finish in a coordinated fashion. For example, when a fixed but unknown number of work items is stored in a queue, parallel Pods can pick these up one by one to work on them. The first Pod that detects that the queue is empty and exits with success indicates the completion of the Job. The Job controller waits for all other Pods to terminate too. Since one Pod processes multiple work items, this Job type is an excellent choice for granular work items—when the overhead for one Pod per work item is not justified.

Indexed Jobs

Similar to *Work queue Jobs*, you can distribute work items to individual Jobs without needing an external work queue. When using a fixed completion count and setting the completion mode `.spec.completionMode` to `Indexed`, every Pod of the Job gets an associated index ranging from 0 to `.spec.completions - 1`. The assigned index is available to the containers through the Pod annotation `batch.kubernetes.io/job-completion-index` (see Chapter 14, "Self Awareness", to learn how this annotation can be accessed from your code) or directly via the environment variable `JOB_COMPLETION_INDEX` that is set to the index associated with this Pod. With this index at hand, the application can pick the associated work item without any external synchronization. Example 7-2 shows a Job that processes the lines of a single file individually by separate Pods. A more realistic example would be an indexed Job used for video processing, where parallel Pods are processing a certain frame range calculated from the index.

Example 7-2. An indexed Job selecting its work items based on a job index

```
apiVersion: batch/v1
kind: Job
metadata:
  name: file-split
spec:
  completionMode: Indexed      ❶
  completions: 5               ❷
  parallelism: 5
  template:
    metadata:
      name: file-split
    spec:
      containers:
      - image: alpine
        name: split
        command:               ❸
        - "sh"
        - "-c"
        - |
          start=$(expr $JOB_COMPLETION_INDEX \* 10000)      ❹
          end=$(expr $JOB_COMPLETION_INDEX \* 10000 + 10000)
          awk "NR>=$start && NR<$end" /logs/random.log \    ❺
              > /logs/random-$JOB_COMPLETION_INDEX.txt
        volumeMounts:
        - mountPath: /logs     ❻
          name: log-volume
      restartPolicy: OnFailure
```

❶ Enable an indexed completion mode.

❷ Run five Pods in parallel to completion.

❸ Execute a shell script that prints out a range of lines from a given file */logs/random.log*. This file is expected to have 50,000 lines of data.

❹ Calculate start and end line numbers.

❺ Use `awk` to print out a range of line numbers (`NR` is the `awk`-internal line number when iterating over the file).

❻ Mount the input data from an external volume. The volume is not shown here; you can find the full working definition in the example repository.

Partitioning the Work

As you have seen, we have multiple options for processing many work items by fewer worker Pods. While *Work queue Jobs* can operate on an unknown but finite set of work items, they need support from an external system that provides the work items. In that case, the external system has already divided the work into appropriately sized work items, so the worker Pods have to process those and stop when there is nothing left to do. The alternative is to use *Indexed Jobs*, which do not rely on an external work queue but have to split up the work on their own so that each Pod can separately work on a portion of the overall task. Each Pod needs to know its own identity (provided by the environment variable `JOB_COMPLETION_INDEX`), the total number of workers, and maybe the overall size of the work (like the size of a movie file to process). Unfortunately, the Job's application code cannot discover the total number of workers (i.e., the value specified in `.spec.completions`) for an Indexed Job. Therefore, something like a `JOB_COMPLETION_TOTAL` environment variable would be helpful to partition the work dynamically, but this is not supported as of 2023. However, there are two solutions to overcome this:

- Hardcode the knowledge of the total number of Pods working on a Job into the application code. While this might work for simple examples like Example 7-2, it's generally an imperfect solution as it couples the code in your container to the Kubernetes declaration. That is, if you want to change the number of completions in your Job definition, you would also have to create a new container image for your Job logic with an updated value.
- To access the value of `.spec.completions` in your application code, you can copy it to an environment variable or pass it as an argument to the container command in the Job's template specification. But if you plan to change the number of completions, you will need to update two places in the Job declaration.

There has been some discussion within the Kubernetes community about whether Kubernetes should provide the value of the `.spec.completions` field as an environment variable by default. The main concern with this approach is that environment variables cannot be modified at runtime, which could complicate support for resizable Jobs in the future. As a result, a `JOB_COMPLETION_TOTAL` environment variable is not provided by Kubernetes as of version 1.26.

If you have an unlimited stream of work items to process, other controllers like ReplicaSet are the better choice for managing the Pods processing these work items.

Discussion

The Job abstraction is a pretty basic but also fundamental primitive that other primitives such as CronJobs are based on. Jobs help turn isolated work units into a reliable and scalable unit of execution. However, a Job doesn't dictate how you should map individually processable work items into Jobs or Pods. That is something you have to determine after considering the pros and cons of each option:

One Job per work item
: This option has the overhead of creating Kubernetes Jobs and also means the platform has to manage a large number of Jobs that are consuming resources. This option is useful when each work item is a complex task that has to be recorded, tracked, or scaled independently.

One Job for all work items
: This option is right for a large number of work items that do not have to be independently tracked and managed by the platform. In this scenario, the work items have to be managed from within the application via a batch framework.

The Job primitive provides only the very minimum basics for scheduling work items. Any complex implementation has to combine the Job primitive with a batch application framework (e.g., in the Java ecosystem, we have Spring Batch and JBeret as standard implementations) to achieve the desired outcome.

Not all services must run all the time. Some services must run on demand, some at a specific time, and some periodically. Using Jobs can run Pods only when needed and only for the duration of the task execution. Jobs are scheduled on nodes that have the required capacity, satisfy Pod placement policies, and take into account other container dependency considerations. Using Jobs for short-lived tasks rather than using long-running abstractions (such as ReplicaSet) saves resources for other workloads on the platform. All of that makes Jobs a unique primitive, and Kubernetes a platform supporting diverse workloads.

More Information

- Batch Job Example
- Jobs
- Parallel Processing Using Expansions
- Coarse Parallel Processing Using a Work Queue
- Fine Parallel Processing Using a Work Queue
- Indexed Job for Parallel Processing with Static Work Assignment
- Spring Batch on Kubernetes: Efficient Batch Processing at Scale
- JBeret Introduction

CHAPTER 8
Periodic Job

The *Periodic Job* pattern extends the *Batch Job* pattern by adding a time dimension and allowing the execution of a unit of work to be triggered by a temporal event.

Problem

In the world of distributed systems and microservices, there is a clear tendency toward real-time and event-driven application interactions using HTTP and lightweight messaging. However, regardless of the latest trends in software development, job scheduling has a long history, and it is still relevant. Periodic jobs are commonly used for automating system maintenance or administrative tasks. They are also relevant to business applications requiring specific tasks to be performed periodically. Typical examples here are business-to-business integration through file transfer, application integration through database polling, sending newsletter emails, and cleaning up and archiving old files.

The traditional way of handling periodic jobs for system maintenance purposes has been to use specialized scheduling software or cron. However, specialized software can be expensive for simple use cases, and cron jobs running on a single server are difficult to maintain and represent a single point of failure. That is why, very often, developers tend to implement solutions that can handle both the scheduling aspect and the business logic that needs to be performed. For example, in the Java world, libraries such as Quartz, Spring Batch, and custom implementations with the `ScheduledThreadPoolExecutor` class can run temporal tasks. But similar to cron, the main difficulty with this approach is making the scheduling capability resilient and highly available, which leads to high resource consumption. Also, with this approach, the time-based job scheduler is part of the application, and to make the scheduler highly available, the whole application must be highly available. Typically, that involves running multiple instances of the application and at the same time

ensuring that only a single instance is active and schedules jobs—which involves leader election and other distributed systems challenges.

In the end, a simple service that has to copy a few files once a day may end up requiring multiple nodes, a distributed leader election mechanism, and more. Kubernetes CronJob implementation solves all that by allowing scheduling of Job resources using the well-known cron format and letting developers focus only on implementing the work to be performed rather than the temporal scheduling aspect.

Solution

In Chapter 7, "Batch Job", we saw the use cases and the capabilities of Kubernetes Jobs. All of that applies to this chapter as well since the CronJob primitive builds on top of a Job. A CronJob instance is similar to one line of a Unix crontab (cron table) and manages the temporal aspects of a Job. It allows the execution of a Job periodically at a specified point in time. See Example 8-1 for a sample definition.

Example 8-1. A CronJob resource

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: random-generator
spec:
  schedule: "*/3 * * * *"  ❶
  jobTemplate:
    spec:
      template:            ❷
        spec:
          containers:
          - image: k8spatterns/random-generator:1.0
            name: random-generator
            command: [ "java", "RandomRunner", "/numbers.txt", "10000" ]
          restartPolicy: OnFailure
```

❶ Cron specification for running every three minutes.

❷ Job template that uses the same specification as a regular Job.

Apart from the Job spec, a CronJob has additional fields to define its temporal aspects:

`.spec.schedule`
: Crontab entry for specifying the Job's schedule (e.g., `0 * * * *` for running every hour). You can also use shortcuts like `@daily` or `@hourly`. Please refer to the CronJob documentation for all available options.

`.spec.startingDeadlineSeconds`
: Deadline (in seconds) for starting the Job if it misses its scheduled time. In some use cases, a task is valid only if it executed within a certain timeframe, and it is useless when executed late. For example, if a Job is not executed in the desired time because of a lack of compute resources or other missing dependencies, it might be better to skip an execution because the data it is supposed to process is already obsolete. Don't use a deadline fewer than 10 seconds since Kubernetes will check the Job status only every 10 seconds.

`.spec.concurrencyPolicy`
: Specifies how to manage concurrent executions of Jobs created by the same CronJob. The default behavior `Allow` creates new Job instances even if the previous Jobs have not completed yet. If that is not the desired behavior, it is possible to skip the next run if the current one has not completed yet with `Forbid` or to cancel the currently running Job and start a new one with `Replace`.

`.spec.suspend`
: Field suspending all subsequent executions without affecting already-started executions. Note that this is different from a Job's `.spec.suspend` as the start of new Jobs will be suspended, not the Jobs themselves.

`.spec.successfulJobsHistoryLimit` *and* `.spec.failedJobsHistoryLimit`
: Fields specifying how many completed and failed Jobs should be kept for auditing purposes.

CronJob is a very specialized primitive, and it applies only when a unit of work has a temporal dimension. Even if CronJob is not a general-purpose primitive, it is an excellent example of how Kubernetes capabilities build on top of one another and support noncloud native use cases as well.

Discussion

As you can see, a CronJob is a pretty simple primitive that adds clustered, cron-like behavior to the existing Job definition. But when it is combined with other primitives such as Pods, container resource isolation, and other Kubernetes features such as those described in Chapter 6, "Automated Placement", or Chapter 4, "Health Probe", it ends up being a very powerful job-scheduling system. This enables developers to focus solely on the problem domain and implement a containerized application that is responsible only for the business logic to be performed. The scheduling is performed outside the application, as part of the platform with all of its added benefits, such as high availability, resiliency, capacity, and policy-driven Pod placement. Of course, similar to the Job implementation, when implementing a CronJob container, your application has to consider all corner and failure cases of duplicate runs, no runs, parallel runs, or cancellations.

More Information

- Periodic Job Example
- CronJob
- Cron
- Crontab Specification
- Cron Expression Generator

CHAPTER 9

Daemon Service

The *Daemon Service* pattern allows you to place and run prioritized, infrastructure-focused Pods on targeted nodes. It is used primarily by administrators to run node-specific Pods to enhance the Kubernetes platform capabilities.

Problem

The concept of a daemon in software systems exists at many levels. At an operating system level, a *daemon* is a long-running, self-recovering computer program that runs as a background process. In Unix, the names of daemons end in *d*, such as httpd, named, and sshd. In other operating systems, alternative terms such as *services-started tasks* and *ghost jobs* are used.

Regardless of what these programs are called, the common characteristics among them are that they run as processes and usually do not interact with the monitor, keyboard, and mouse and are launched at system boot time. A similar concept also exists at the application level. For example, in the Java Virtual Machine, daemon threads run in the background and provide supporting services to the user threads. These daemon threads have a low priority, run in the background without a say in the life of the application, and perform tasks such as garbage collection or finalization.

Similarly, Kubernetes also has the concept of a DaemonSet. Considering that Kubernetes is a distributed platform spread across multiple nodes and with the primary goal of managing application Pods, a DaemonSet is represented by Pods that run on the cluster nodes and provide some background capabilities for the rest of the cluster.

Solution

ReplicaSet and its predecessor ReplicationController are control structures responsible for making sure a specific number of Pods are running. These controllers constantly monitor the list of running Pods and make sure the actual number of Pods always matches the desired number. In that regard, a DaemonSet is a similar construct and is responsible for ensuring that a certain number of Pods are always running. The difference is that the first two run a specific number of Pods, usually driven by the application requirements of high availability and user load, irrespective of the node count.

On the other hand, a DaemonSet is not driven by consumer load in deciding how many Pod instances to run and where to run. Its main purpose is to keep running a single Pod on every node or specific nodes. Let's see such a DaemonSet definition next in Example 9-1.

Example 9-1. DaemonSet resource

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: random-refresher
spec:
  selector:
    matchLabels:
      app: random-refresher
  template:
    metadata:
      labels:
        app: random-refresher
    spec:
      nodeSelector:                ❶
        feature: hw-rng
      containers:
      - image: k8spatterns/random-generator:1.0
        name: random-generator
        command: [ "java", "RandomRunner", "/numbers.txt", "10000", "30" ]
        volumeMounts:              ❷
        - mountPath: /host_dev
          name: devices
      volumes:
      - name: devices
        hostPath:                  ❸
          path: /dev
```

❶ Use only nodes with the label `feature` set to value `hw-rng`.

❷ DaemonSets often mount a portion of a node's filesystem to perform maintenance actions.

❸ `hostPath` for accessing the node directories directly.

Given this behavior, the primary candidates for a DaemonSet are usually infrastructure-related processes, such as cluster storage providers, log collectors, metric exporters, and even kube-proxy, that perform cluster-wide operations. There are many differences in how DaemonSet and ReplicaSet are managed, but the main ones are the following:

- By default, a DaemonSet places one Pod instance on every node. That can be controlled and limited to a subset of nodes by using the `nodeSelector` or `affinity` fields.
- A Pod created by a DaemonSet already has `nodeName` specified. As a result, the DaemonSet doesn't require the existence of the Kubernetes scheduler to run containers. That also allows you to use a DaemonSet for running and managing the Kubernetes components.
- Pods created by a DaemonSet can run before the scheduler has started, which allows them to run before any other Pod is placed on a node.
- Since the scheduler is not used, the `unschedulable` field of a node is not respected by the DaemonSet controller.
- Pods created by a DaemonSet can have a `RestartPolicy` only set to `Always` or left unspecified, which defaults to `Always`. This is to ensure that when a liveness probe fails, the container will be killed and always restarted.
- Pods managed by a DaemonSet are supposed to run only on targeted nodes and, as a result, are treated with higher priority by many controllers. For example, the descheduler will avoid evicting such Pods, the cluster autoscaler will manage them separately, etc.

The main use case for DaemonSets is to run system-critical Pods on certain nodes in the cluster. The DaemonSet controller ensures that all eligible nodes run a copy of a Pod by assigning the Pod directly to the node by setting the `nodeName` field of the Pod specification. This allows DaemonSet Pods to be scheduled even before the default scheduler starts and keeps it immune to any scheduler customizations configured by the user. This approach works as long as there are enough resources on the nodes and it is done before other Pods are placed. When a node does not have enough resources, the DaemonSet controller cannot create a Pod for the node, and it cannot do anything such as preemption to release resources on the nodes. This duplication of scheduling logic in the DaemonSet controller and the scheduler creates maintenance challenges. The DaemonSet implementation also does not benefit from

new scheduler features such as affinity, anti-affinity, and preemption. As a result, with Kubernetes v1.17 and newer versions, DaemonSet uses the default scheduler for scheduling by setting the `nodeAffinity` field instead of the `nodeName` field to the DaemonSet Pods. This change makes the default scheduler a mandatory dependency for running DaemonSets, but at the same time it brings taints, tolerations, Pod priority, and preemption to DaemonSets and improves the overall experience of running DaemonSet Pods on the desired nodes even when there is resource starvation.

Typically, a DaemonSet creates a single Pod on every node or subset of nodes. Given that, there are several ways to reach Pods managed by DaemonSets:

Service
: Create a Service with the same Pod selector as a DaemonSet, and use the Service to reach a daemon Pod load-balanced to a random node.

DNS
: Create a headless Service with the same Pod selector as a DaemonSet that can be used to retrieve multiple A records from DNS containing all Pod IPs and ports.

Node IP with `hostPort`
: Pods in the DaemonSet can specify a `hostPort` and become reachable via the node IP addresses and the specified port. Since the combination of node IP and `hostPort` and `protocol` must be unique, the number of places where a Pod can be scheduled is limited.

Also, the application in the DaemonSets Pods can push data to a well-known location or service that's external to the Pod. No consumer needs to reach the DaemonSets Pods in this case.

Static Pods

Another way to run containers similar to the way a DaemonSet does is through the *static Pods* mechanism. The Kubelet, in addition to talking to the Kubernetes API Server and getting Pod manifests, can get the resource definitions from a local directory. Pods defined this way are managed by the Kubelet only and run on one node only. The API service is not observing these Pods, and no controller and no health checks are performed on them. The Kubelet watches such Pods and restarts them when they crash. Similarly, the Kubelet also periodically scans the configured directory for Pod definition changes and adds or removes Pods accordingly.

Static Pods can be used to spin off a containerized version of Kubernetes system processes or other containers. However, DaemonSets are better integrated with the rest of the platform and are recommended over static Pods.

Discussion

There are other ways to run daemon processes on every node, but they all have limitations. Static Pods are managed by the Kubelet but cannot be managed through Kubernetes APIs. Bare Pods (Pods without a controller) cannot survive if they are accidentally deleted or terminated, nor can they survive a node failure or disruptive node maintenance. Init scripts such as upstartd or systemd require different toolchains for monitoring and management and cannot benefit from the Kubernetes tools used for application workloads. All that makes Kubernetes and DaemonSet an attractive option for running daemon processes too.

In this book, we describe patterns and Kubernetes features primarily used by developers rather than platform administrators. A DaemonSet is somewhere in the middle, inclining more toward the administrator toolbox, but we include it here because it also has relevance to application developers. DaemonSets and CronJobs are also perfect examples of how Kubernetes turns single-node concepts such as crontab and daemon scripts into multinode clustered primitives for managing distributed systems. These are new distributed concepts developers must also be familiar with.

More Information

- Daemon Service Example
- DaemonSet
- Perform a Rolling Update on a DaemonSet
- DaemonSets and Jobs
- Create Static Pods

CHAPTER 10

Singleton Service

The *Singleton Service* pattern ensures that only one instance o an application is active at a time and yet is highly available. This pattern can be implemented from within the application or delegated fully to Kubernetes.

Problem

One of the main capabilities provided by Kubernetes is the ability to easily and transparently scale applications. Pods can scale imperatively with a single command such as `kubectl scale`, or declaratively through a controller definition such as ReplicaSet, or even dynamically based on the application load, as we describe in Chapter 29, "Elastic Scale". By running multiple instances of the same service (not a Kubernetes Service but a component of a distributed application represented by a Pod), the system usually increases throughput and availability. The availability increases because if one instance of a service becomes unhealthy, the request dispatcher forwards future requests to other healthy instances. In Kubernetes, multiple instances are the replicas of a Pod, and the Service resource is responsible for the request distribution and load balancing.

However, in some cases, only one instance of a service is allowed to run at a time. For example, if there is a periodically executed task in a service and multiple instances of the same service, every instance will trigger the task at the scheduled intervals, leading to duplicates rather than having only one task fired as expected. Another example is a service that performs polling on specific resources (a filesystem or database) and we want to ensure that only a single instance and maybe even a single thread performs the polling and processing. A third case occurs when we have to consume messages from a messages broker in an order-preserving manner with a single-threaded consumer that is also a singleton service.

In all these and similar situations, we need some control over how many instances \ of a service are active at a time (usually only one is required), while still ensuring high availability, regardless of how many instances have been started and kept running.

Solution

Running multiple replicas of the same Pod creates an *active-active* topology, where all instances of a service are active. What we need is an *active-passive* topology, where only one instance is active and all the other instances are passive. Fundamentally, this can be achieved at two possible levels: out-of-application and in-application locking.

Out-of-Application Locking

As the name suggests, this mechanism relies on a managing process that is outside of the application to ensure that only a single instance of the application is running. The application implementation itself is not aware of this constraint and is run as a singleton instance. From this perspective, it is similar to having a Java class that is instantiated only once by the managing runtime (such as the Spring Framework). The class implementation is not aware that it is run as a singleton, nor that it contains any code constructs to prevent instantiating multiple instances.

Figure 10-1 shows how to implement out-of-application locking with the help of a StatefulSet or ReplicaSet controller with one replica.

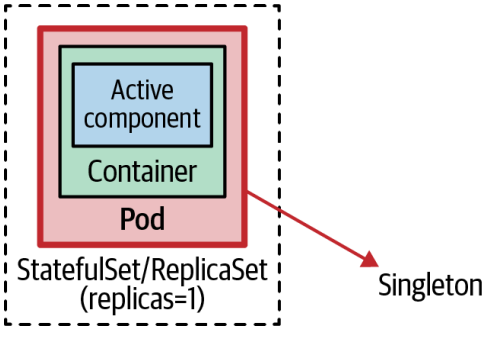


Figure 10-1. Out-of-application locking mechanism

The way to achieve this in Kubernetes is to start a single Pod. This activity alone does not ensure the singleton Pod is highly available. What we have to do is also back the Pod with a controller such as a ReplicaSet that turns the singleton Pod into a highly available singleton. This topology is not exactly *active-passive* (there is no passive instance), but it has the same effect, as Kubernetes ensures that one instance of the Pod is running at all times. In addition, the single Pod instance is highly available,

thanks to the controller performing health checks as described in Chapter 4, "Health Probe", and healing the Pod in case of failures.

The main thing to keep an eye on with this approach is the replica count, which should not be changed accidentally. In this section, you will see how we can voluntarily decrease the replica count through PodDisruptionBudget, but there is no platform-level mechanism to prevent an increase of the replica count.

It's not entirely true that only one instance is running at all times, especially when things go wrong. Kubernetes primitives such as ReplicaSet favor availability over consistency—a deliberate decision for achieving highly available and scalable distributed systems. That means a ReplicaSet applies "at least" rather than "at most" semantics for its replicas. If we configure a ReplicaSet to be a singleton with `replicas: 1`, the controller makes sure at least one instance is always running, but occasionally it can be more instances.

The most popular corner case here occurs when a node with a controller-managed Pod becomes unhealthy and disconnects from the rest of the Kubernetes cluster. In this scenario, a ReplicaSet controller starts another Pod instance on a healthy node (assuming there is enough capacity), without ensuring the Pod on the disconnected node is shut down. Similarly, when changing the number of replicas or relocating Pods to different nodes, the number of Pods can temporarily go above the desired number. That temporary increase is done with the intention of ensuring high availability and avoiding disruption, as needed for stateless and scalable applications.

Singletons can be resilient and recover, but by definition, they are not highly available. Singletons typically favor consistency over availability. The Kubernetes resource that also favors consistency over availability and provides the desired strict singleton guarantees is the StatefulSet. If ReplicaSets do not provide the desired guarantees for your application, and you have strict singleton requirements, StatefulSets might be the answer. StatefulSets are intended for stateful applications and offer many features, including stronger singleton guarantees, but they come with increased complexity as well. We discuss concerns around singletons and cover StatefulSets in more detail in Chapter 12, "Stateful Service".

Typically, singleton applications running in Pods on Kubernetes open outgoing connections to message brokers, relational databases, file servers, or other systems running on other Pods or external systems. However, occasionally, your singleton Pod may need to accept incoming connections, and the way to enable that on Kubernetes is through the Service resource.

We cover Kubernetes Services in depth in Chapter 13, "Service Discovery", but let's discuss briefly the part that applies to singletons here. A regular Service (with `type: ClusterIP`) creates a virtual IP and performs load balancing among all the Pod instances that its selector matches. However, a singleton Pod managed through

a StatefulSet has only one Pod and a stable network identity. In such a case, it is better to create a *headless Service* (by setting both `type: ClusterIP` and `clusterIP: None`). It is called *headless* because such a Service doesn't have a virtual IP address, kube-proxy doesn't handle these Services, and the platform performs no proxying.

However, such a Service is still useful because a headless Service with selectors creates endpoint records in the API Server and generates DNS A records for the matching Pod(s). With that, a DNS lookup for the Service does not return its virtual IP but instead the IP address(es) of the backing Pod(s). That enables direct access to the singleton Pod via the Service DNS record, and without going through the Service virtual IP. For example, if we create a headless Service with the name `my-singleton`, we can use it as `my-singleton.default.svc.cluster.local` to access the Pod's IP address directly.

To sum up, for nonstrict singletons with at least one instance requirement, defining a ReplicaSet with one replica would suffice. This configuration favors availability and ensures there is at least one available instance, and possibly more in some corner cases. For a strict singleton with an At-Most-One requirement and better performant service discovery, a StatefulSet and a headless Service would be preferred. Using StatefulSet will favor consistency and ensure there is an At-Most-One instance and occasionally none in some corner cases. You can find a complete example of this in Chapter 12, "Stateful Service", where you have to change the number of replicas to one to make it a singleton.

In-Application Locking

In a distributed environment, one way to control the service instance count is through a distributed lock, as shown in Figure 10-2. Whenever a service instance or a component inside the instance is activated, it can try to acquire a lock, and if it succeeds, the service becomes active. Any subsequent service instance that fails to acquire the lock waits and continuously tries to get the lock in case the currently active service releases it.

Many existing distributed frameworks use this mechanism for achieving high availability and resiliency. For example, the message broker Apache ActiveMQ can run in a highly available *active-passive* topology, where the data source provides the shared lock. The first broker instance that starts up acquires the lock and becomes active, and any other subsequently started instances become passive and wait for the lock to be released. This strategy ensures there is a single active broker instance that is also resilient to failures.

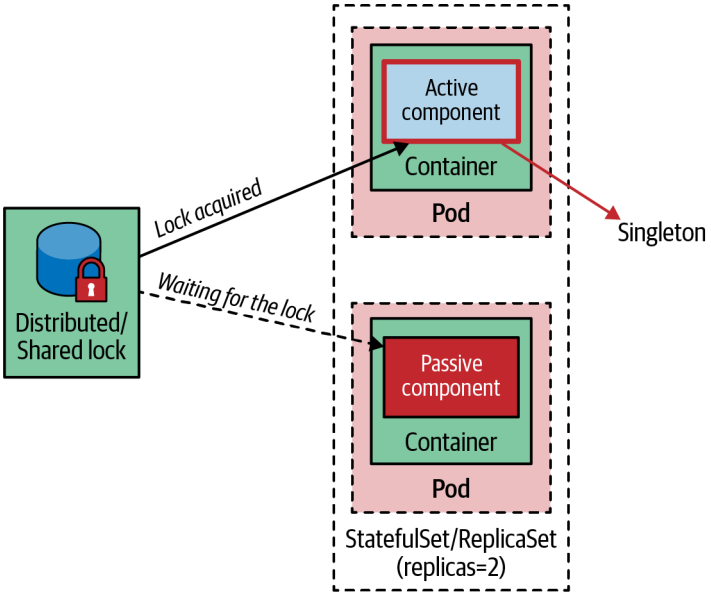


Figure 10-2. In-application locking mechanism

We can compare this strategy to a classic Singleton, as it is known in the object-oriented world: a *Singleton* is an object instance stored in a static class variable. In this instance, the class is aware of being a singleton, and it is written in a way that does not allow instantiation of multiple instances for the same process. In distributed systems, this would mean the containerized application itself has to be written in a way that does not allow more than one active instance at a time, regardless of the number of Pod instances that are started. To achieve this in a distributed environment, first we need a distributed lock implementation such as the one provided by Apache ZooKeeper, HashiCorp's Consul, Redis, or etcd.

The typical implementation with ZooKeeper uses ephemeral nodes, which exist as long as there is a client session and are deleted as soon as the session ends. The first service instance that starts up initiates a session in the ZooKeeper server and creates an ephemeral node to become active. All other service instances from the same cluster become passive and have to wait for the ephemeral node to be released. This is how a ZooKeeper-based implementation makes sure there is only one active service instance in the whole cluster, ensuring an active-passive failover behavior.

In the Kubernetes world, instead of managing a ZooKeeper cluster only for the locking feature, a better option would be to use etcd capabilities exposed through the Kubernetes API and running on the main nodes. etcd is a distributed key-value store that uses the Raft protocol to maintain its replicated state and provides the necessary building blocks for implementing leader election. For example, Kubernetes offers the

Lease object, which is used for node heartbeats and component-level leader election. For every node, there is a Lease object with a matching name, and the Kubelet on every node keeps running a heart beat by updating the Lease object's `renewTime` field. This information is used by the Kubernetes control plane to determine the availability of the nodes. Kubernetes Leases are also used in highly available cluster deployment scenarios for ensuring only single control plane components such as kube-controller-manager and kube-scheduler are active at a time and other instances remain on standby.

Another example is in Apache Camel, which has a Kubernetes connector that also provides leader election and singleton capabilities. This connector goes a step further, and rather than accessing the etcd API directly, it uses Kubernetes APIs to leverage ConfigMaps as a distributed lock. It relies on Kubernetes optimistic locking guarantees for editing resources such as ConfigMaps, where only one Pod can update a ConfigMap at a time. The Camel implementation uses this guarantee to ensure only one Camel route instance is active, and any other instance has to wait and acquire the lock before activating. It is a custom implementation of a lock but achieves the same goal: when there are multiple Pods with the same Camel application, only one of them becomes the active singleton, and the others wait in passive mode.

A more generic implementation of the *Singleton Service* pattern is provided by the Dapr project. Dapr's Distributed Lock building block provides APIs (HTTP and gRPC) with swappable implementations for mutually exclusive access to shared resources. The idea is that each application determines the resources the lock grants access to. Then, multiple instances of the same application use a named lock to exclusively access the shared resource. At any given moment, only one instance of an application can hold a named lock. All other instances of the application are unable to acquire the lock and therefore are not allowed to access the shared resource until the lock is released through unlock or the lock times out. Thanks to its lease-based locking mechanism, if an application acquires a lock, encounters an exception, and cannot free the lock, the lock is automatically released after a period of time using a lease. This prevents resource deadlocks in the event of application failures. Behind this generic distributed lock API, Dapr will be configured to use some kind of storage and lock implementation. This API can be used by applications to implement access to shared resources or in-application singletons.

An implementation with Dapr, ZooKeeper, etcd, or any other distributed lock implementation would be similar to the one described: only one instance of the application becomes the leader and activates itself, and other instances are passive and wait for the lock. This ensures that even if multiple Pod replicas are started and all are healthy, up, and running, only one service is active and performs the business functionality as a singleton, and other instances wait to acquire the lock in case the leader fails or shuts down.

Pod Disruption Budget

While singleton service and leader election try to limit the maximum number of instances a service is running at a time, the PodDisruptionBudget functionality of Kubernetes provides a complementary and somewhat opposite functionality—limiting the number of instances that are simultaneously down for maintenance.

At its core, PodDisruptionBudget ensures a certain number or percentage of Pods will not voluntarily be evicted from a node at any one point in time. *Voluntarily* here means an eviction that can be delayed for a particular time—for example, when it is triggered by draining a node for maintenance or upgrade (`kubectl drain`), or a cluster scaling down, rather than a node becoming unhealthy, which cannot be predicted or controlled.

The PodDisruptionBudget in Example 10-1 applies to Pods that match its selector and ensures two Pods must be available all the time.

Example 10-1. PodDisruptionBudget

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: random-generator-pdb
spec:
  selector:
    matchLabels:                ❶
      app: random-generator
  minAvailable: 2               ❷
```

❶ Selector to count available Pods.

❷ At least two Pods have to be available. You can also specify a percentage, like 80%, to configure that only 20% of the matching Pods might be evicted.

In addition to `.spec.minAvailable`, there is also the option to use `.spec.maxUnavailable`, which specifies the number of Pods from that set that can be unavailable after the eviction. Similar to `.spec.minAvailable`, it can be either an absolute number or a percentage, but it has a few additional limitations. You can specify only either `.spec.minAvailable` or `.spec.maxUnavailable` in a single PodDisruptionBudget, and then it can be used only to control the eviction of Pods that have an associated controller such as ReplicaSet or StatefulSet. For Pods not managed by a controller (also referred to as *bare* or *naked* Pods), other limitations around PodDisruptionBudget should be considered.

PodDisruptionBudget is useful for quorum-based applications that require a minimum number of replicas running at all times to ensure a quorum. Or maybe when an application is serving critical traffic that should never go below a certain percentage of the total number of instances.

PodDisruptionBudget is useful in the context of singletons too. For example, setting `maxUnavailable` to `0` or setting `minAvailable` to `100%` will prevent any voluntary eviction. Setting voluntary eviction to zero for a workload will turn it into an unevictable Pod and will prevent draining the node forever. This can be used as a step in the process where a cluster operator has to contact the singleton workload owner for downtime before accidentally evicting a not highly available Pod. StatefulSet, combined with PodDisruptionBudget, and headless Service are Kubernetes primitives that control and help with the instance count at runtime and are worth mentioning in this chapter.

Discussion

If your use case requires strong singleton guarantees, you cannot rely on the out-of-application locking mechanisms of ReplicaSets. Kubernetes ReplicaSets are designed to preserve the availability of their Pods rather than to ensure At-Most-One semantics for Pods. As a consequence, there are many failure scenarios that have two copies of a Pod running concurrently for a short period (efor example, when a node that runs the singleton Pod is partitioned from the rest of the cluster—such as when replacing a deleted Pod instance with a new one). If that is not acceptable, use StatefulSets or investigate the in-application locking options that provide you more control over the leader election process with stronger guarantees. The latter also mitigates the risk of accidentally scaling Pods by changing the number of replicas. You can combine this with PodDisruptionBudget and prevent voluntary eviction and disruption of your singleton workloads.

In other scenarios, only a part of a containerized application should be a singleton. For example, there might be a containerized application that provides an HTTP endpoint that is safe to scale to multiple instances, but also a polling component that must be a singleton. Using the out-of-application locking approach would prevent scaling the whole service. In such a situation, we either have to split the singleton component in its deployment unit to keep it a singleton (good in theory but not always practical or worth the overhead) or use the in-application locking mechanism and lock only the component that has to be a singleton. This would allow us to scale the whole application transparently, have HTTP endpoints scaled, and have other parts as *active-passive* singletons.

More Information

- Singleton Service Example
- Leases
- Specifying a Disruption Budget for Your Application
- Leader Election in Go Client
- Dapr: Distributed Lock Overview
- Creating Clustered Singleton Services on Kubernetes
- Akka: Kubernetes Lease

CHAPTER 11

Stateless Service

The *Stateless Service* pattern describes how to create and operate applications that are composed of identical ephemeral replicas. These applications are best suited for dynamic cloud environments where they can be rapidly scaled and made highly available.

Problem

The microservices architecture style is the dominant choice for implementing new greenfield cloud native applications. Among the driving principles of this architecture are things such as how it addresses a single concern, how it owns its data, how it has a well-encapsulated deployment boundary, and others. Typically, such applications also follow the twelve-factor app principles, which makes them easy to operate with Kubernetes on dynamic cloud environments.

Applying some of these principles requires understanding the business domain, identifying the service boundary, or applying domain-driven design or a similar methodology during the service implementation. Implementing some of the other principles may involve making the services ephemeral, which means the service can be created, scaled, and destroyed with no side effects. These latter concerns are easier to address when a service is stateless rather than stateful.

A stateless service does not maintain any state internally within the instance across service interactions. In our context, it means a container is stateless if it does not hold any information from requests in its internal storage (memory or temporary filesystem) that is critical for serving future requests. A stateless process has no stored knowledge of or reference to past requests, so each request is made as if from scratch. Instead, if the process needs to store such information, it should store it in an external storage such as a database, message queue, mounted filesystem,

or some other data store that can be accessed by other instances. A good thought experiment is to imagine the instances of your services deployed on different nodes and a load-balancer that randomly distributes the requests to the instances without any sticky session (i.e., without an affinity between a client and a specific service instance). If the service can fulfill its purpose in this setup, it is likely a stateless service (or it has a mechanism for state distribution among the instances, such as a data grid).

Stateless services are made of identical, replaceable instances that often offload state to external permanent storage systems and use load-balancers for distributing incoming requests among themselves. In this chapter, we will see specifically which Kubernetes abstractions can help operate such stateless applications.

Solution

In Chapter 3, "Declarative Deployment", you learned how to use the concept of Deployment to control how an application should be updated to the next version, using the `RollingUpdate` and `Recreate` strategies. But this is only the upgrading aspect of Deployment. At a broader level, a Deployment represents an application deployed in the cluster. Kubernetes doesn't have the notion of an `Application` or a `Container` as top-level entities. Instead, an application is typically composed of a collection of Pods managed by a controller such as ReplicaSet, Deployment, or StatefulSet, combined with ConfigMap, Secret, Service, PersistentVolumeClaim, etc. The controller that is used for managing stateless Pods is ReplicaSet, but that is a lower-level internal control structure used by a Deployment. Deployment is the recommended user-facing abstraction for creating and updating stateless applications, which creates and manages the ReplicaSets behind the scene. A ReplicaSet should be used when the update strategies provided by Deployment are not suitable, or a custom mechanism is required, or no control over the update process is needed at all.

Instances

The primary purpose of a ReplicaSet is to ensure a specified number of identical Pod replicas running at any given time. The main sections of a ReplicaSet definition include the number of replicas indicating how many Pods it should maintain, a selector that specifies how to identify the Pods it manages, and a Pod template for creating new Pod replicas. Then, a ReplicaSet creates and deletes Pods as needed to maintain the desired replica count using the given Pod template, as demonstrated in Example 11-1.

Example 11-1. ReplicaSet definition for a stateless Pod

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rg
  labels:
    app: random-generator
spec:
  replicas: 3                    ❶
  selector:                      ❷
    matchLabels:
      app: random-generator
  template:                      ❸
    metadata:
      labels:
        app: random-generator
    spec:
      containers:
      - name: random-generator
        image: k8spatterns/random-generator:1.0
```

❶ Desired number of Pod replicas to maintain running.

❷ Label selector used to identify the Pods to manage.

❸ Template specifying the data for creating new Pods.

The template is used when the ReplicaSet needs to create new Pods to meet the desired number of replicas. But a ReplicaSet is not limited to managing the Pods specified by the template. If a bare Pod has no owner reference (meaning it is not managed by a controller), and it matches the label selector, it will be acquired by setting the owner reference and managed by the ReplicaSet. This setup can lead to a ReplicaSet owning a nonidentical set of Pods created by different means, and terminate existing bare Pods that exceed the declared replica count. To avoid such undesired side effects, it is recommended that you ensure bare Pods do not have labels matching ReplicaSet selectors.

Regardless of whether you create a ReplicaSet directly or through a Deployment, the end result will be that the desired number of identical Pod replicas are created and maintained. The added benefit of using Deployment is that we can control how the replicas are upgraded and rolled back, which we described in detail in Chapter 3, "Declarative Deployment". Next, the replicas are scheduled to the available nodes as per the policies we covered in Chapter 6, "Automated Placement". The ReplicaSet's job is to restart the containers if needed and scale out or in when the number of replicas is increased or decreased, respectively. With this behavior, Deployment and ReplicaSet can automate the lifecycle management of stateless applications.

Networking

Pods created by ReplicaSet are ephemeral and may disappear at any time, such as when a Pod is evicted because of resource starvation or because the node the Pod is running on fails. In such a situation, the ReplicaSet will create a new Pod that will have a new name, hostname, and IP address. If the application is stateless, as we've defined earlier in the chapter, new requests should be handled from the newly created Pod the same way as by any other Pod.

Depending on how the application within the container connects to the other systems to accept requests or poll for messages, for example, you may require a Kubernetes Service. If the application is starting an egress connection to a message broker or database, and that is the only way it exchanges data, then there is no need for a Kubernetes Service. But more often, stateless services are contacted by other services over synchronous request/response-driven protocols such as HTTP and gRPC. Since the Pod IP address changes with every Pod restart, it is better to use a permanent IP address based on a Kubernetes Service that service consumers can use. A Kubernetes Service has a fixed IP address that doesn't change during the lifetime of the Service, and it ensures the client requests are always load-balanced across instances and routed to the healthy and ready-to-accept-requests Pods. We cover different types of Kubernetes Services in Chapter 13, "Service Discovery". In Example 11-2, we use a simple Service to expose the Pods internally within the cluster to other Pods.

Example 11-2. Exposing a stateless service

```
apiVersion: v1
kind: Service
metadata:
  name: random-generator      ❶
spec:
  selector:                   ❷
    app: random-generator
  ports:
  - port: 80
    targetPort: 8080
    protocol: TCP
```

❶ Name of the service that can be used to reach the matching Pods.

❷ Selector matching the Pod labels from the ReplicaSet.

The definition in this example will create a Service named `random-generator` that accepts TCP connections on port 80 and routes them to port 8080 on all the matching Pods with selector `app: random-generator`. Once a Service is created, it is assigned a `clusterIP` that is accessible only from within the Kubernetes cluster, and that IP remains unchanged as long as the Service definition exists. This acts as a

permanent entrypoint to all matching Pods that are ephemeral and have changing IP addresses.

Notice that Deployment and the resulting ReplicaSet are only responsible for maintaining the desired number of stateless Pods that match the label selector. They are unaware of any Kubernetes Service that might be directing traffic to the same set of Pods or a different combination of Pods.

Storage

Few stateless services don't need any state and can process requests based only on the data provided in every request. Most stateless services require state, but they are stateless because they offload the state to some other stateful system or data store, such as a filesystem. Any Pod, whether it is created by a ReplicaSet or not, can declare and use file storage through volumes. Different types of volumes can be used to store state. Some of these are cloud-provider-specific storage, while others allow mounting network storage or even sharing filesystems from the node where the Pod is placed. In this section, we'll look at the `persistentVolumeClaim` volume type, which allows you to use manually or dynamically provisioned persistent storage.

A PersistentVolume (PV) represents a storage resource abstraction in a Kubernetes cluster that has a lifecycle independent of any Pod lifecycle that is using it. A Pod cannot directly refer to a PV; however, a Pod uses PersistentVolumeClaim (PVC) to request and bind to the PV, which points to the actual durable storage. This indirect connection allows for a separation of concerns and Pod lifecycle decoupling from PV. A cluster administrator can configure storage provisioning and define PVs. The developer creating Pod definitions can use PVC to use the storage. With this indirection, even if the Pod is deleted, the ownership of the PV remains attached to the PVC and continues to exist. Example 11-3 shows a storage claim that can be used in a Pod template.

Example 11-3. A claim for a PersistentVolume

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: random-generator-log  ❶
spec:
  storageClassName: "manual"
  accessModes:
   - ReadWriteOnce            ❷
  resources:
   requests:
    storage: 1Gi              ❸
```

❶ Name of the claim that can be referenced from a Pod template.

❷ Indicates that only a single node can mount the volume for reading and writing.

❸ Requesting 1 GiB of storage.

Once a PVC is defined, it can be referenced from a Pod template through the `persistentVolumeClaim` field. One of the interesting fields of PersistentVolumeClaim is `accessModes`. It controls how the storage is mounted to the nodes and consumed by the Pods. For example, network filesystems can be mounted to multiple nodes and can allow reading and writing to multiple applications at the same time. Other storage implementations can be mounted to only a single node at a time and can be accessed only by the Pods scheduled on that node. Let's look at different `accessModes` offered by Kubernetes:

ReadWriteOnce
: This represents a volume that can be mounted to a single node at a time. In this mode, one or multiple Pods running on the node could carry out read and write operations.

ReadOnlyMany
: The volume can be mounted to multiple nodes, but it allows read-only operations to all Pods.

ReadWriteMany
: In this mode, the volume can be mounted by many nodes and allows both read and write operations.

ReadWriteOncePod
: Notice that all of the access modes described so far offer per-node granularity. Even `ReadWriteOnce` allows multiple Pods on the same node to read from and write to the same volume simultaneously. Only `ReadWriteOncePod` access mode guarantees that only a single Pod has access to a volume. This is invaluable in scenarios where at most one writer application is allowed to access data for data-consistency guarantees. Use this mode with caution as it will turn your services into a singleton and prevent scaling out. If another Pod replica uses the same PVC, the Pod will fail to start because the PVC is already in use by another Pod. As of this writing, `ReadWriteOncePod` doesn't honor preemption either, which means a lower-priority Pod will hold on to the storage and not be preempted from the node in favor of a higher-priority Pod waiting on the same `ReadWriteOncePod` claim.

In a ReplicaSet, all Pods are identical; they share the same PVC and refer to the same PV. This is in contrast to StatefulSets covered in the next chapter, where PVCs are created dynamically for each stateful Pod replica. This is one of the major differences between how stateless and stateful workloads are handled in Kubernetes.

Discussion

A complex distributed system is usually composed of multiple services, some of which will be stateful and perform some form of distributed coordination, some of which might be short-lived jobs, and some of which might be highly scalable stateless services. Stateless services are composed of identical, swappable, ephemeral, and replaceable instances. They are ideal for handling short-lived requests and can scale up and down rapidly without having any dependencies among the instances. As shown in Figure 11-1, Kubernetes offers a number of useful primitives to manage such applications.

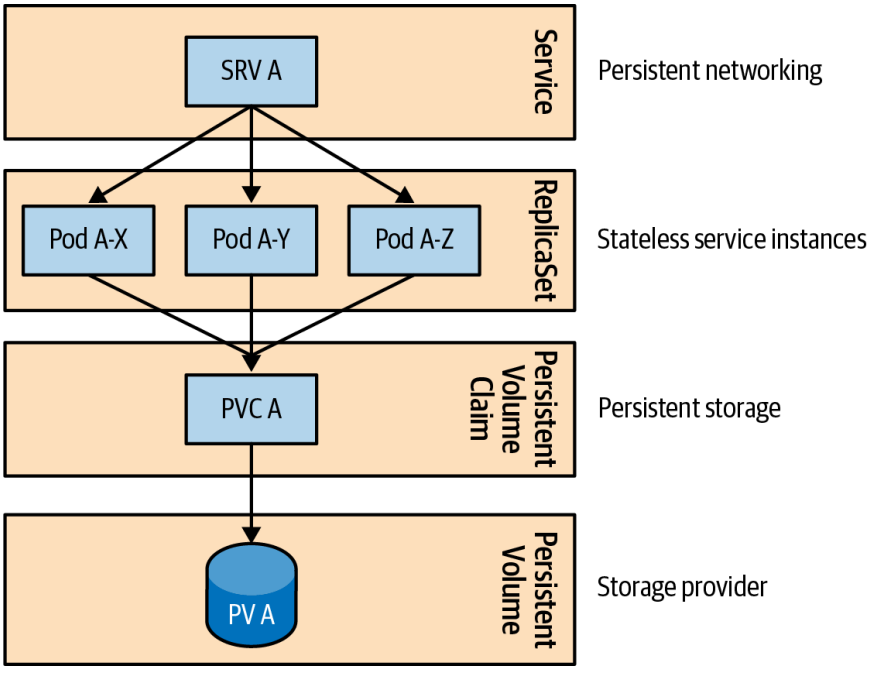


Figure 11-1. A distributed stateless application on Kubernetes

At the lowest level, the Pod abstraction ensures that one or more containers are observed with liveness checks and are always up and running. Building on that, the ReplicaSet also ensures that the desired number of stateless Pods are always running on the healthy nodes. Deployments automate the upgrade and rollback mechanism of Pod replicas. When there is incoming traffic, the Service abstraction discovers and distributes traffic to healthy Pod instances with passing readiness probes. When a persistent file storage is required, PVCs can request and mount storage.

Although Kubernetes offers these building blocks, it will not enforce any direct relationship between them. It is your responsibility to combine them to match the application nature. You have to understand how liveness checks and ReplicaSet control Pods' lifecycles, and how they relate to readiness probes and Service definitions controlling how the traffic is directed to the Pods. You should also understand how PVCs and `accessMode` control where the storage is mounted and how it is accessed. When Kubernetes primitives are not sufficient, you should know how to combine it with other frameworks such as Knative and KEDA and how to autoscale and even turn stateless applications into serverless. The latter frameworks are covered in Chapter 29, "Elastic Scale".

More Information

- Stateless Service Example
- ReplicaSet
- Persistent Volumes
- Storage Classes
- Access Modes

CHAPTER 12

Stateful Service

Distributed stateful applications require features such as persistent identity, networking, storage, and ordinality. The *Stateful Service* pattern describes the StatefulSet primitive that provides these building blocks with strong guarantees ideal for the management of stateful applications.

Problem

We have seen many Kubernetes primitives for creating distributed applications: containers with health checks and resource limits, Pods with multiple containers, dynamic cluster-wide placements, batch jobs, scheduled jobs, singletons, and more. The common characteristic of these primitives is that they treat the managed application as a stateless application composed of identical, swappable, and replaceable containers and comply with the twelve-factor app principles.

It is a significant boost to have a platform taking care of the placement, resiliency, and scaling of stateless applications, but there is still a large part of the workload to consider: stateful applications in which every instance is unique and has long-lived characteristics.

In the real world, behind every highly scalable stateless service is a stateful service, typically in the shape of a data store. In the early days of Kubernetes, when it lacked support for stateful workloads, the solution was placing stateless applications on Kubernetes to get the benefits of the cloud native model and keeping stateful components outside the cluster, either on a public cloud or on-premises hardware, managed with the traditional noncloud native mechanisms. Considering that every enterprise has a multitude of stateful workloads (legacy and modern), the lack of support for stateful workloads was a significant limitation in Kubernetes, which was known as a universal cloud native platform.

But what are the typical requirements of a stateful application? We could deploy a stateful application such as Apache ZooKeeper, MongoDB, Redis, or MySQL by using a Deployment, which could create a ReplicaSet with `replicas=1` to make it reliable, use a Service to discover its endpoint, and use PersistentVolumeClaim (PVC) and PersistentVolume (PV) as permanent storage for its state.

While that is mostly true for a single-instance stateful application, it is not entirely true, as a ReplicaSet does not guarantee At-Most-One semantics, and the number of replicas can vary temporarily. Such a situation can be disastrous and lead to data loss for distributed stateful applications. Also, the main challenges arise when it is a distributed stateful service that is composed of multiple instances. A stateful application composed of multiple clustered services requires multifaceted guarantees from the underlying infrastructure. Let's see some of the most common long-lived persistent prerequisites for distributed stateful applications.

Storage

We could easily increase the number of `replicas` in a ReplicaSet and end up with a distributed stateful application. However, how do we define the storage requirements in such a case? Typically, a distributed stateful application such as those mentioned previously would require dedicated, persistent storage for every instance. A ReplicaSet with `replicas=3` and a PVC definition would result in all three Pods attached to the same PV. While the ReplicaSet and the PVC ensure the instances are up and the storage is attached to whichever node the instances are scheduled on, the storage is not dedicated but shared among all Pod instances.

A workaround is for the application instances to share storage and have an in-app mechanism to split the storage into subfolders and use it without conflicts. While doable, this approach creates a single point of failure with the single storage. Also, it is error-prone as the number of Pods changes during scaling, and it may cause severe challenges around preventing data corruption or loss during scaling.

Another workaround is to have a separate ReplicaSet (with `replicas=1`) for every instance of the distributed stateful application. In this scenario, every ReplicaSet would get its PVC and dedicated storage. The downside of this approach is that it is intensive in manual labor: scaling up requires creating a new set of ReplicaSet, PVC, or Service definitions. This approach lacks a single abstraction for managing all instances of the stateful application as one.

Networking

Similar to the storage requirements, a distributed stateful application requires a stable network identity. In addition to storing application-specific data into the storage space, stateful applications also store configuration details such as hostname and

connection details of their peers. That means every instance should be reachable in a predictable address that should not change dynamically, as is the case with Pod IP addresses in a ReplicaSet. Here we could address this requirement again through a workaround: create a Service per ReplicaSet and have `replicas=1`. However, managing such a setup is manual work, and the application itself cannot rely on a stable hostname because it changes after every restart and is also not aware of the Service name it is accessed from.

Identity

As you can see from the preceding requirements, clustered stateful applications depend heavily on every instance having a hold of its long-lived storage and network identity. That is because in a stateful application, every instance is unique and knows its own identity, and the main ingredients of that identity are the long-lived storage and the networking coordinates. To this list, we could also add the identity/name of the instance (some stateful applications require unique persistent names), which in Kubernetes would be the Pod name. A Pod created with ReplicaSet would have a random name and would not preserve that identity across a restart.

Ordinality

In addition to a unique and long-lived identity, the instances of clustered stateful applications have a fixed position in the collection of instances. This ordering typically impacts the sequence in which the instances are scaled up and down. However, it can also be used for data distribution or access and in-cluster behavior positioning such as locks, singletons, or leaders.

Other Requirements

Stable and long-lived storage, networking, identity, and ordinality are among the collective needs of clustered stateful applications. Managing stateful applications also carries many other specific requirements that vary case by case. For example, some applications have the notion of a quorum and require a minimum number of instances to always be available; some are sensitive to ordinality, and some are fine with parallel Deployments; and some tolerate duplicate instances, and some don't. Planning for all these one-off cases and providing generic mechanisms is an impossible task, and that's why Kubernetes also allows you to create CustomResourceDefinitions (CRDs) and *Operators* for managing applications with bespoke requirements. The *Operator* pattern is explained in Chapter 28.

We have seen some common challenges of managing distributed stateful applications and a few less-than-ideal workarounds. Next, let's check out the Kubernetes native mechanism for addressing these requirements through the StatefulSet primitive.

Solution

To explain what StatefulSet provides for managing stateful applications, we occasionally compare its behavior to the already-familiar ReplicaSet primitive that Kubernetes uses for running stateless workloads. In many ways, StatefulSet is for managing pets, and ReplicaSet is for managing cattle. Pets versus cattle is a famous (but also a controversial) analogy in the DevOps world: identical and replaceable servers are referred to as cattle, and nonfungible unique servers that require individual care are referred to as pets. Similarly, StatefulSet (initially inspired by the analogy and named PetSet) is designed for managing nonfungible Pods, as opposed to ReplicaSet, which is for managing identical replaceable Pods.

Let's explore how StatefulSets work and how they address the needs of stateful applications. Example 12-1 is our random-generator service as a StatefulSet.[1]

Example 12-1. StatefulSet definition for a stateful application

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: rg                                  ❶
spec:
  serviceName: random-generator             ❷
  replicas: 2                               ❸
  selector:
    matchLabels:
      app: random-generator
  template:
    metadata:
      labels:
        app: random-generator
    spec:
      containers:
      - image: k8spatterns/random-generator:1.0
        name: random-generator
        ports:
        - containerPort: 8080
          name: http
        volumeMounts:
        - name: logs
          mountPath: /logs
  volumeClaimTemplates:                     ❹
  - metadata:
      name: logs
```

1 Let's assume we have invented a highly sophisticated way of generating random numbers in a distributed Random Number Generator (RNG) cluster with several instances of our service as nodes. Of course, that's not true, but for this example's sake, it's a good enough story.

```
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 10Mi
```

❶ Name of the StatefulSet is used as prefix for the generated node names.

❷ References the mandatory Service defined in Example 12-2.

❸ Two Pod members in the StatefulSet named *rg-0* and *rg-1*.

❹ Template for creating a PVC for each Pod (similar to the Pod's template).

Rather than going through the definition in Example 12-1 line by line, we explore the overall behavior and the guarantees provided by this StatefulSet definition.

Storage

While it is not always necessary, the majority of stateful applications store state and thus require per-instance-based dedicated persistent storage. The way to request and associate persistent storage with a Pod in Kubernetes is through PVs and PVCs. To create PVCs the same way it creates Pods, StatefulSet uses a `volumeClaimTemplates` element. This extra property is one of the main differences between a StatefulSet and a ReplicaSet, which has a `persistentVolumeClaim` element.

Rather than referring to a predefined PVC, StatefulSets create PVCs by using `volumeClaimTemplates` on the fly during Pod creation. This mechanism allows every Pod to get its own dedicated PVC during initial creation as well as during scaling up by changing the `replicas` count of the StatefulSets.

As you probably realize, we said PVCs are created and associated with the Pods, but we didn't say anything about PVs. That is because StatefulSets do not manage PVs in any way. The storage for the Pods must be provisioned in advance by an admin or provisioned on demand by a PV provisioner based on the requested storage class and ready for consumption by the stateful Pods.

Note the asymmetric behavior here: scaling up a StatefulSet (increasing the `replicas` count) creates new Pods and associated PVCs. Scaling down deletes the Pods, but it does not delete any PVCs (or PVs), which means the PVs cannot be recycled or deleted, and Kubernetes cannot free the storage. This behavior is by design and driven by the presumption that the storage of stateful applications is critical and that an accidental scale-down should not cause data loss. If you are sure the stateful application has been scaled down on purpose and has replicated/drained the data to other instances, you can delete the PVC manually, which allows subsequent PV recycling.

Networking

Each Pod created by a StatefulSet has a stable identity generated by the StatefulSet's name and an ordinal index (starting from 0). Based on the preceding example, the two Pods are named `rg-0` and `rg-1`. The Pod names are generated in a predictable format that differs from the ReplicaSet's Pod-name-generation mechanism, which contains a random suffix.

Dedicated scalable persistent storage is an essential aspect of stateful applications and so is networking.

In Example 12-2, we define a *headless* Service. In a headless Service, `clusterIP` is set to `None`, which means we don't want a kube-proxy to handle the Service, and we don't want a cluster IP allocation or load balancing. Then why do we need a Service?

Example 12-2. Service for accessing StatefulSet

```
apiVersion: v1
kind: Service
metadata:
  name: random-generator
spec:
  clusterIP: None          ❶
  selector:
    app: random-generator
  ports:
  - name: http
    port: 8080
```

❶ Declares this Service as headless.

Stateless Pods created through a ReplicaSet are assumed to be identical, and it doesn't matter on which one a request lands (hence the load balancing with a regular Service). But stateful Pods differ from one another, and we may need to reach a specific Pod by its coordinates.

A headless Service with selectors (notice `.selector.app == random-generator`) enables exactly this. Such a Service creates endpoint records in the API Server and creates DNS entries to return A records (addresses) that point directly to the Pods backing the Service. Long story short, each Pod gets a DNS entry where clients can directly reach out to it in a predictable way. For example, if our `random-generator` Service belongs to the `default` namespace, we can reach our `rg-0` Pod through its fully qualified domain name: `rg-0.random-generator.default.svc.cluster.local`, where the Pod's name is prepended to the Service name. This mapping allows other members of the clustered application or other clients to reach specific Pods if they wish to.

We can also perform DNS lookup for Service (SRV) records (e.g., through `dig SRV random-generator.default.svc.cluster.local`) and discover all running Pods registered with the StatefulSet's governing Service. This mechanism allows dynamic cluster member discovery if any client application needs to do so. The association between the headless Service and the StatefulSet is not only based on the selectors, but the StatefulSet should also link back to the Service by its name as `serviceName: "random-generator"`.

Having dedicated storage defined through `volumeClaimTemplates` is not mandatory, but linking to a Service through `serviceName` field is. The governing Service must exist before the StatefulSet is created and is responsible for the network identity of the set. You can always create other types of Services that also load balance across your stateful Pods if that is what you want.

As Figure 12-1 shows, StatefulSets offer a set of building blocks and guaranteed behavior needed for managing stateful applications in a distributed environment. Your job is to choose and use them in a meaningful way for your stateful use case.

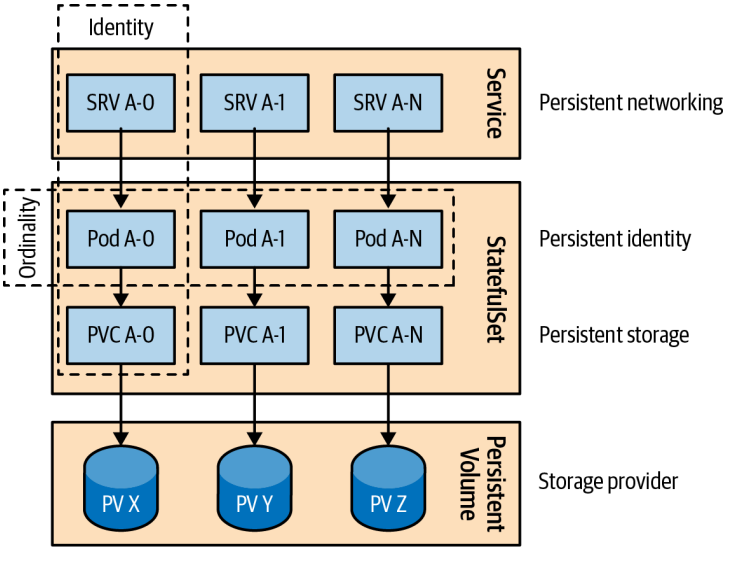


Figure 12-1. A distributed stateful application on Kubernetes

Identity

Identity is the meta building block all other StatefulSet guarantees are built upon. A predictable Pod name and identity is generated based on StatefulSet's name. We then use that identity to name PVCs, reach out to specific Pods through headless Services, and more. You can predict the identity of every Pod before creating it and use that knowledge in the application itself if needed.

Ordinality

By definition, a distributed stateful application consists of multiple instances that are unique and nonswappable. In addition to their uniqueness, instances may also be related to one another based on their instantiation order/position, and this is where the *ordinality* requirement comes in.

From a StatefulSet point of view, the only place where ordinality comes into play is during scaling. Pods have names that have an ordinal suffix (starting from 0), and that Pod creation order also defines the order in which Pods are scaled up and down (in reverse order, from n – 1 to 0).

If we create a ReplicaSet with multiple replicas, Pods are scheduled and started together without waiting for the first one to start successfully (running and ready status, as described in Chapter 4, "Health Probe"). The order in which Pods are starting and are ready is not guaranteed. It is the same when we scale down a ReplicaSet (either by changing the `replicas` count or deleting it). All Pods belonging to a ReplicaSet start shutting down simultaneously without any ordering and dependency among them. This behavior may be faster to complete but is not preferred for stateful applications, especially if data partitioning and distribution are involved among the instances.

To allow proper data synchronization during scale-up and -down, StatefulSet by default performs sequential startup and shutdown. That means Pods start from the first one (with index 0), and only when that Pod has successfully started is the next one scheduled (with index 1), and the sequence continues. During scaling down, the order reverses—first shutting down the Pod with the highest index, and only when it has shut down successfully is the Pod with the next lower index stopped. This sequence continues until the Pod with index 0 is terminated.

Other Features

StatefulSets have other aspects that are customizable to suit the needs of stateful applications. Each stateful application is unique and requires careful consideration while trying to fit it into the StatefulSet model. Let's see a few more Kubernetes features that may turn out to be useful while taming stateful applications:

Partitioned updates
: We described earlier the sequential ordering guarantees when scaling a StatefulSet. As for updating an already-running stateful application (e.g., by altering the `.spec.template` element), StatefulSets allow phased rollout (such as a canary release), which guarantees a certain number of instances to remain intact while applying updates to the rest of the instances.

By using the default rolling update strategy, you can partition instances by specifying a `.spec.updateStrategy.rollingUpdate.partition` number. The parameter (with a default value of 0) indicates the ordinal at which the StatefulSet should be partitioned for updates. If the parameter is specified, all Pods with an ordinal index greater than or equal to the `partition` are updated, while all Pods with an ordinal less than that are not updated. That is true even if the Pods are deleted; Kubernetes recreates them at the previous version. This feature can enable partial updates to clustered stateful applications (ensuring the quorum is preserved, for example) and then roll out the changes to the rest of the cluster by setting the `partition` back to 0.

Parallel deployments

When we set `.spec.podManagementPolicy` to `Parallel`, the StatefulSet launches or terminates all Pods in parallel and does not wait for Pods to run and become ready or completely terminated before moving to the next one. If sequential processing is not a requirement for your stateful application, this option can speed up operational procedures.

At-Most-One Guarantee

Uniqueness is among the fundamental attributes of stateful application instances, and Kubernetes guarantees that uniqueness by making sure no two Pods of a StatefulSet have the same identity or are bound to the same PV. In contrast, ReplicaSet offers the *At-Least-X-Guarantee* for its instances. For example, a ReplicaSet with two replicas tries to keep at least two instances up and running at all times. Even if there is occasionally a chance for that number to go higher, the controller's priority is not to let the number of Pods go below the specified number. It is possible to have more than the specified number of replicas running when a Pod is being replaced by a new one and the old Pod is still not fully terminated. Or, it can go higher if a Kubernetes node is unreachable with `NotReady` state but still has running Pods. In this scenario, the ReplicaSet's controller would start new Pods on healthy nodes, which could lead to more running Pods than desired. That is all acceptable within the semantics of At-Least-X.

A StatefulSet controller, on the other hand, makes every possible check to ensure there are no duplicate Pods—hence the *At-Most-One Guarantee*. It does not start a Pod again unless the old instance is confirmed to be shut down completely. When a node fails, it does not schedule new Pods on a different node unless Kubernetes can confirm that the Pods (and maybe the whole node) are shut down. The At-Most-One semantics of StatefulSets dictates these rules.

It is still possible to break these guarantees and end up with duplicate Pods in a StatefulSet, but this requires active human intervention. For example, deleting an unreachable node resource object from the API Server while the physical node is still running would break this guarantee. Such an action should be performed

only when the node is confirmed to be dead or powered down and no Pod processes are running on it. Or, for example, when you are forcefully deleting a Pod with `kubectl delete pods <pod> --grace-period=0 --force`, which does not wait for a confirmation from the Kubelet that the Pod is terminated. This action immediately clears the Pod from the API Server and causes the StatefulSet controller to start a replacement Pod that could lead to duplicates.

We discuss other approaches to achieving singletons in more depth in Chapter 10, "Singleton Service".

Discussion

In this chapter, we saw some of the standard requirements and challenges in managing distributed stateful applications on a cloud native platform. We discovered that handling a single-instance stateful application is relatively easy, but handling distributed state is a multidimensional challenge. While we typically associate the notion of "state" with "storage," here we have seen multiple facets of state and how it requires different guarantees from different stateful applications. In this space, StatefulSets is an excellent primitive for implementing distributed stateful applications generically. It addresses the need for persistent storage, networking (through Services), identity, ordinality, and a few other aspects. It provides a good set of building blocks for managing stateful applications in an automated fashion, making them first-class citizens in the cloud native world.

StatefulSets are a good start and a step forward, but the world of stateful applications is unique and complex. In addition to the stateful applications designed for a cloud native world that can fit into a StatefulSet, a ton of legacy stateful applications exist that have not been designed for cloud native platforms and have even more needs. Luckily Kubernetes has an answer for that too. The Kubernetes community has realized that rather than modeling different workloads through Kubernetes resources and implementing their behavior through generic controllers, it should allow users to implement their custom controllers and even go one step further and allow modeling application resources through custom resource definitions and behavior through operators.

In Chapters 27 and 28, you will learn about the related *Controller* and *Operator* patterns, which are better suited for managing complex stateful applications in cloud native environments.

More Information

- Stateful Service Example
- StatefulSet Basics
- StatefulSets
- Example: Deploying Cassandra with a Stateful Set
- Running ZooKeeper, a Distributed System Coordinator
- Headless Services
- Force Delete StatefulSet Pods
- Graceful Scaledown of Stateful Apps in Kubernetes

CHAPTER 13

Service Discovery

The *Service Discovery* pattern provides a stable endpoint through which consumers of a service can access the instances providing the service. For this purpose, Kubernetes provides multiple mechanisms, depending on whether the service consumers and producers are located on or off the cluster.

Problem

Applications deployed on Kubernetes rarely exist on their own, and usually they have to interact with other services within the cluster or systems outside the cluster. The interaction can be initiated internally within the service or through external stimulus. Internally initiated interactions are usually performed through a polling consumer: either after startup or later, an application connects to another system and starts sending and receiving data. Typical examples are an application running within a Pod that reaches a file server and starts consuming files, or a message that connects to a message broker and starts receiving or sending messages, or an application that uses a relational database or a key-value store and starts reading or writing data.

The critical distinction here is that the application running within the Pod decides at some point to open an outgoing connection to another Pod or external system and starts exchanging data in either direction. In this scenario, we don't have an external stimulus for the application, and we don't need any additional setup in Kubernetes.

To implement the patterns described in Chapter 7, "Batch Job", or Chapter 8, "Periodic Job", we often use this technique. In addition, long-running Pods in DaemonSets or ReplicaSets sometimes actively connect to other systems over the network. The more common use case for Kubernetes workloads occurs when we have long-running services expecting external stimulus, most commonly in the form of incoming HTTP connections from other Pods within the cluster or external systems. In these cases,

service consumers need a mechanism for discovering Pods that are dynamically placed by the scheduler and sometimes elastically scaled up and down.

It would be a significant challenge if we had to track, register, and discover endpoints of dynamic Kubernetes Pods ourselves. That is why Kubernetes implements the *Service Discovery* pattern through different mechanisms, which we explore in this chapter.

Solution

If we look at the "Before Kubernetes Era," the most common mechanism of service discovery was through client-side discovery. In this architecture, when a service consumer had to call another service that might be scaled to multiple instances, the service consumer would have a discovery agent capable of looking at a registry for service instances and then choosing one to call. Classically, that would be done, for example, either with an embedded agent within the consumer service (such as a ZooKeeper client, Consul client, or Ribbon) or with another colocated process looking up the service in a registry, as shown in Figure 13-1.

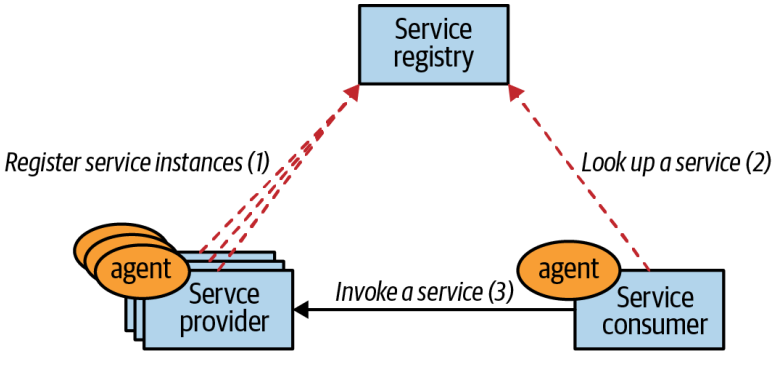


Figure 13-1. Client-side service discovery

In the "Post Kubernetes Era," many of the nonfunctional responsibilities of distributed systems such as placement, health checks, healing, and resource isolation are moving into the platform, and so is service discovery and load balancing. If we use the definitions from service-oriented architecture (SOA), a service provider instance still has to register itself with a service registry while providing the service capabilities, and a service consumer has to access the information in the registry to reach the service.

In the Kubernetes world, all that happens behind the scenes so that a service consumer calls a fixed virtual Service endpoint that can dynamically discover service instances implemented as Pods. Figure 13-2 shows how registration and lookup are embraced by Kubernetes.

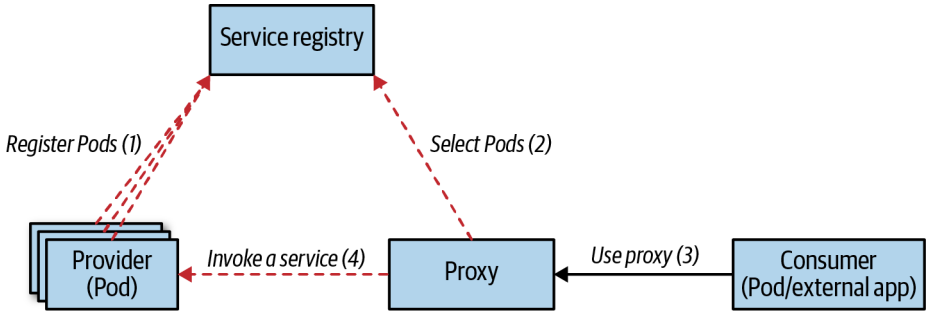


Figure 13-2. Server-side service discovery

At first glance, *Service Discovery* may seem like a simple pattern. However, multiple mechanisms can be used to implement this pattern, which depends on whether a service consumer is within or outside the cluster and whether the service provider is within or outside the cluster.

Internal Service Discovery

Let's assume we have a web application and want to run it on Kubernetes. As soon as we create a Deployment with a few replicas, the scheduler places the Pods on the suitable nodes, and each Pod gets a cluster-internal IP address assigned before starting up. If another client service within a different Pod wishes to consume the web application endpoints, there isn't an easy way to know the IP addresses of the service provider Pods in advance.

This challenge is what the Kubernetes Service resource addresses. It provides a constant and stable entry point for a collection of Pods offering the same functionality. The easiest way to create a Service is through `kubectl expose`, which creates a Service for a Pod or multiple Pods of a Deployment or ReplicaSet. The command creates a virtual IP address referred to as the `clusterIP`, and it pulls both Pod selectors and port numbers from the resources to create the Service definition. However, to have full control over the definition, we create the Service manually, as shown in Example 13-1.

Example 13-1. A simple Service

```
apiVersion: v1
kind: Service
metadata:
  name: random-generator
spec:
  selector:                  ❶
    app: random-generator
  ports:
  - port: 80                 ❷
    targetPort: 8080         ❸
    protocol: TCP
```

❶ Selector matching Pod labels.

❷ Port over which this Service can be contacted.

❸ Port on which the Pods are listening.

The definition in this example will create a Service named `random-generator` (the name is important for discovery later) and `type: ClusterIP` (which is the default) that accepts TCP connections on port 80 and routes them to port 8080 on all the matching Pods with the selector `app: random-generator`. It doesn't matter when or how the Pods are created—any matching Pod becomes a routing target, as illustrated in Figure 13-3.

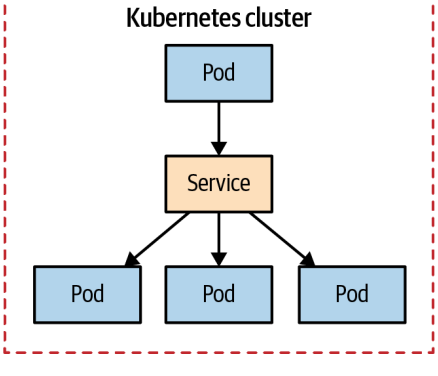


Figure 13-3. Internal service discovery

The essential points to remember here are that once a Service is created, it gets a `clusterIP` assigned that is accessible only from within the Kubernetes cluster (hence the name), and that IP remains unchanged as long as the Service definition exists. However, how can other applications within the cluster figure out what this dynamically allocated `clusterIP` is? There are two ways:

Discovery through environment variables
: When Kubernetes starts a Pod, its environment variables get populated with the details of all Services that exist up to that moment. For example, our `random-generator` Service listening on port 80 gets injected into any newly starting Pod, as the environment variables shown in Example 13-2 demonstrate. The application running that Pod would know the name of the Service it needs to consume and can be coded to read these environment variables. This lookup is a simple mechanism that can be used from applications written in any language and is also easy to emulate outside the Kubernetes cluster for development and testing purposes. The main issue with this mechanism is the temporal dependency on Service creation. Since environment variables cannot be injected into already-running Pods, the Service coordinates are available only for Pods started after the Service is created in Kubernetes. That requires the Service to be defined before starting the Pods that depend on the Service—or if this is not the case, the Pods need to be restarted.

Example 13-2. Service-related environment variables set automatically in Pod

```
RANDOM_GENERATOR_SERVICE_HOST=10.109.72.32
RANDOM_GENERATOR_SERVICE_PORT=80
```

Discovery through DNS lookup
: Kubernetes runs a DNS server that all the Pods are automatically configured to use. Moreover, when a new Service is created, it automatically gets a new DNS entry that all Pods can start using. Assuming a client knows the name of the Service it wants to access, it can reach the Service by a fully qualified domain name (FQDN) such as `random-generator.default.svc.cluster.local`. Here, `random-generator` is the name of the Service, `default` is the name of the namespace, `svc` indicates it is a Service resource, and `cluster.local` is the cluster-specific suffix. We can omit the cluster suffix if desired, and the namespace as well when accessing the Service from the same namespace.

The DNS discovery mechanism doesn't suffer from the drawbacks of the environment-variable-based mechanism, as the DNS server allows lookup of all Services to all Pods as soon as a Service is defined. However, you may still need to use the environment variables to look up the port number to use if it is a nonstandard one or unknown by the service consumer.

Here are some other high-level characteristics of the Service with `type: ClusterIP` that other types build upon:

Multiple ports
: A single Service definition can support multiple source and target ports. For example, if your Pod supports both HTTP on port 8080 and HTTPS on port 8443, there is no need to define two Services. A single Service can expose both ports on 80 and 443, for example.

Session affinity
: When there is a new request, the Service randomly picks a Pod to connect to by default. That can be changed with `sessionAffinity: ClientIP`, which makes all requests originating from the same client IP stick to the same Pod. Remember that Kubernetes Services performs L4 transport layer load balancing, and it cannot look into the network packets and perform application-level load balancing such as HTTP cookie-based session affinity.

Readiness probes
: In Chapter 4, "Health Probe", you learned how to define a `readinessProbe` for a container. If a Pod has defined readiness checks, and they are failing, the Pod is removed from the list of Service endpoints to call even if the label selector matches the Pod.

Virtual IP
: When we create a Service with `type: ClusterIP`, it gets a stable virtual IP address. However, this IP address does not correspond to any network interface and doesn't exist in reality. It is the kube-proxy that runs on every node that picks this new Service and updates the iptables of the node with rules to catch the network packets destined for this virtual IP address and replaces it with a selected Pod IP address. The rules in the iptables do not add ICMP rules, but only the protocol specified in the Service definition, such as TCP or UDP. As a consequence, it is not possible to `ping` the IP address of the Service as that operation uses the ICMP.

Choosing ClusterIP
: During Service creation, we can specify an IP to use with the field `.spec.clusterIP`. It must be a valid IP address and within a predefined range. While not recommended, this option can turn out to be handy when dealing with legacy

applications configured to use a specific IP address, or if there is an existing DNS entry we wish to reuse.

Kubernetes Services with `type: ClusterIP` are accessible only from within the cluster; they are used for discovery of Pods by matching selectors and are the most commonly used type. Next, we will look at other types of Services that allow discovery of endpoints that are manually specified.

Manual Service Discovery

When we create a Service with `selector`, Kubernetes tracks the list of matching and ready-to-serve Pods in the list of endpoint resources. For Example 13-1, you can check all endpoints created on behalf of the Service with `kubectl get endpoints random-generator`. Instead of redirecting connections to Pods within the cluster, we could also redirect connections to external IP addresses and ports. We can do that by omitting the `selector` definition of a Service and manually creating endpoint resources, as shown in Example 13-3.

Example 13-3. Service without selector

```
apiVersion: v1
kind: Service
metadata:
  name: external-service
spec:
  type: ClusterIP
  ports:
  - protocol: TCP
    port: 80
```

Next, in Example 13-4, we define an endpoint resource with the same name as the Service and containing the target IPs and ports.

Example 13-4. Endpoints for an external service

```
apiVersion: v1
kind: Endpoints
metadata:
  name: external-service    ❶
subsets:
  - addresses:
    - ip: 1.1.1.1
    - ip: 2.2.2.2
    ports:
    - port: 8080
```

❶ Name must match the Service that accesses these endpoints.

This Service is also accessible only within the cluster and can be consumed in the same way as the previous ones, through environment variables or DNS lookup. The difference is that the list of endpoints is manually maintained and those values usually point to IP addresses outside the cluster, as demonstrated in Figure 13-4.

While connecting to an external resource is this mechanism's most common use, it is not the only one. Endpoints can hold IP addresses of Pods but not virtual IP addresses of other Services. One good thing about the Service is that it allows you to add and remove selectors and point to external or internal providers without deleting the resource definition that would lead to a Service IP address change. So service consumers can continue using the same Service IP address they first pointed to while the actual service provider implementation is migrated from on-premises to Kubernetes without affecting the client.

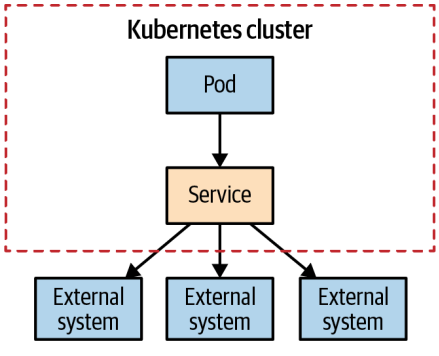


Figure 13-4. Manual service discovery

In this category of manual destination configuration, there is one more type of Service, as shown in Example 13-5.

Example 13-5. Service with an external destination

```
apiVersion: v1
kind: Service
metadata:
  name: database-service
spec:
  type: ExternalName
  externalName: my.database.example.com
  ports:
  - port: 80
```

This Service definition does not have a `selector` either, but its type is `ExternalName`. That is an important difference from an implementation point of view. This Service definition maps to the content pointed by `externalName` using DNS only, or more

specifically, `database-service.<namespace>.svc.cluster.local` will now point to `my.database.example.com`. It is a way of creating an alias for an external endpoint using DNS CNAME rather than going through the proxy with an IP address. But fundamentally, it is another way of providing a Kubernetes abstraction for endpoints located outside the cluster.

Service Discovery from Outside the Cluster

The service discovery mechanisms discussed so far in this chapter all use a virtual IP address that points to Pods or external endpoints, and the virtual IP address itself is accessible only from within the Kubernetes cluster. However, a Kubernetes cluster doesn't run disconnected from the rest of the world, and in addition to connecting to external resources from Pods, very often the opposite is also required—external applications wanting to reach to endpoints provided by the Pods. Let's see how to make Pods accessible for clients living outside the cluster.

The first method to create a Service and expose it outside of the cluster is through `type: NodePort`. The definition in Example 13-6 creates a Service as earlier, serving Pods that match the selector `app: random-generator`, accepting connections on port 80 on the virtual IP address and routing each to port 8080 of the selected Pod. However, in addition to all of that, this definition also reserves port 30036 on all the nodes and forwards incoming connections to the Service. This reservation makes the Service accessible internally through the virtual IP address, as well as externally through a dedicated port on every node.

Example 13-6. Service with type NodePort

```
apiVersion: v1
kind: Service
metadata:
  name: random-generator
spec:
  type: NodePort              ❶
  selector:
    app: random-generator
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30036           ❷
    protocol: TCP
```

❶ Open port on all nodes.

❷ Specify a fixed port (which needs to be available) or leave this out to get a randomly selected port assigned.

While this method of exposing services (illustrated in Figure 13-5) may seem like a good approach, it has drawbacks.

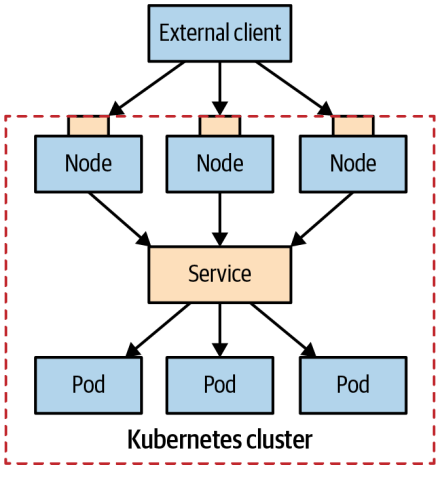


Figure 13-5. Node port service discovery

Let's see some of its distinguishing characteristics:

Port number
: Instead of picking a specific port with `nodePort: 30036`, you can let Kubernetes pick a free port within its range.

Firewall rules
: Since this method opens a port on all the nodes, you may have to configure additional firewall rules to let external clients access the node ports.

Node selection
: An external client can open connection to any node in the cluster. However, if the node is not available, it is the responsibility of the client application to connect to another healthy node. For this purpose, it may be a good idea to put a load balancer in front of the nodes that picks healthy nodes and performs failover.

Pods selection
: When a client opens a connection through the node port, it is routed to a randomly chosen Pod that may be on the same node where the connection was open or a different node. It is possible to avoid this extra hop and always force Kubernetes to pick a Pod on the node where the connection was opened by adding `externalTrafficPolicy: Local` to the Service definition. When this option is set, Kubernetes does not allow you to connect to Pods located on other nodes, which can be an issue. To resolve that, you have to either make sure there

are Pods placed on every node (e.g., by using daemon services) or make sure the client knows which nodes have healthy Pods placed on them.

Source addresses
: There are some peculiarities around the source addresses of packets sent to different types of Services. Specifically, when we use type `NodePort`, client addresses are source NAT'd, which means the source IP addresses of the network packets containing the client IP address are replaced with the node's internal addresses. For example, when a client application sends a packet to node 1, it replaces the source address with its node address, replaces the destination address with the Pod's address, and forwards the packet to node 2, where the Pod is located. When the Pod receives the network packet, the source address is not equal to the original client's address but is the same as node 1's address. To prevent this from happening, we can set `externalTrafficPolicy: Local` as described earlier and forward traffic only to Pods located on node 1.

Another way to perform Service Discovery for external clients is through a load balancer. You have seen how a `type: NodePort` Service builds on top of a regular Service with `type: ClusterIP` by also opening a port on every node. The limitation of this approach is that we still need a load balancer for client applications to pick a healthy node. The Service type `LoadBalancer` addresses this limitation.

In addition to creating a regular Service, and opening a port on every node, as with `type: NodePort`, it also exposes the service externally using a cloud provider's load balancer. Figure 13-6 shows this setup: a proprietary load balancer serves as a gateway to the Kubernetes cluster.

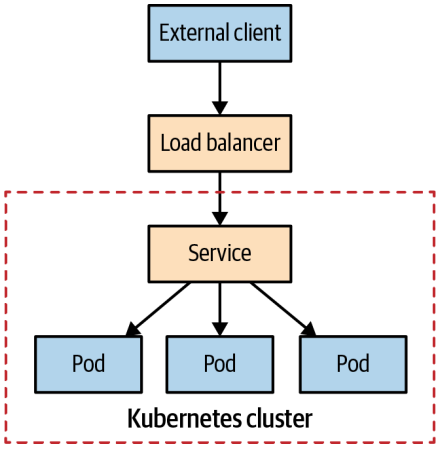


Figure 13-6. Load balancer service discovery

So this type of Service works only when the cloud provider has Kubernetes support and provisions a load balancer. We can create a Service with a load balancer by specifying the type `LoadBalancer`. Kubernetes then will add IP addresses to the `.spec` and `.status` fields, as shown in Example 13-7.

Example 13-7. Service of type LoadBalancer

```
apiVersion: v1
kind: Service
metadata:
  name: random-generator
spec:
  type: LoadBalancer
  clusterIP: 10.0.171.239          ❶
  loadBalancerIP: 78.11.24.19
  selector:
    app: random-generator
  ports:
  - port: 80
    targetPort: 8080
status:                            ❷
  loadBalancer:
    ingress:
    - ip: 146.148.47.155
```

❶ Kubernetes assigns `clusterIP` and `loadBalancerIP` when they are available.

❷ The `status` field is managed by Kubernetes and adds the Ingress IP.

With this definition in place, an external client application can open a connection to the load balancer, which picks a node and locates the Pod. The exact way that load-balancer provisioning and service discovery are performed varies among cloud providers. Some cloud providers will allow you to define the load-balancer address and some will not. Some offer mechanisms for preserving the source address, and some replace that with the load-balancer address. You should check the specific implementation provided by your cloud provider of choice.

Yet another type of Service is available: *headless* services, for which you don't request a dedicated IP address. You create a headless service by specifying `clusterIP None` within the Service's `spec` section. For headless services, the backing Pods are added to the internal DNS server and are most useful for implementing Services to StatefulSets, as described in detail in Chapter 12, "Stateful Service".

Application Layer Service Discovery

Unlike the mechanisms discussed so far, Ingress is not a service type but a separate Kubernetes resource that sits in front of Services and acts as a smart router and entry point to the cluster. Ingress typically provides HTTP-based access to Services through externally reachable URLs, load balancing, TLS termination, and name-based virtual hosting, but there are also other specialized Ingress implementations. For Ingress to work, the cluster must have one or more Ingress controllers running. A simple Ingress that exposes a single Service is shown in Example 13-8.

Example 13-8. An Ingress definition

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: random-generator
spec:
  defaultBackend:
    service:
      name: random-generator
      port:
        number: 8080
```

Depending on the infrastructure Kubernetes is running on, and the Ingress controller implementation, this definition allocates an externally accessible IP address and exposes the `random-generator` Service on port 80. But this is not very different from a Service with `type: LoadBalancer`, which requires an external IP address per Service definition. The real power of Ingress comes from reusing a single external load balancer and IP to service multiple Services and reduce the infrastructure costs. A simple fan-out configuration for routing a single IP address to multiple Services based on HTTP URI paths looks like Example 13-9.

Example 13-9. A definition for Nginx Ingress controller

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: random-generator
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:                          ❶
  - http:
      paths:
      - path: /                   ❷
        pathType: Prefix
        backend:
          service:
            name: random-generator
            port:
              number: 8080
      - path: /cluster-status     ❸
        pathType: Exact
        backend:
          service:
            name: cluster-status
            port:
              number: 80
```

❶ Dedicated rules for the Ingress controller for dispatching requests based on the request path.

❷ Redirect every request to Service `random-generator`…

❸ … except `/cluster-status`, which goes to another Service.

Since every Ingress controller implementation is different, apart from the usual Ingress definition, a controller may require additional configuration, which is passed through annotations. Assuming the Ingress is configured correctly, the preceding definition would provision a load balancer and get an external IP address that services two Services under two different paths, as shown in Figure 13-7.

Ingress is the most powerful and at the same time most complex service discovery mechanism on Kubernetes. It is most useful for exposing multiple services under the same IP address and when all services use the same L7 (typically HTTP) protocol.

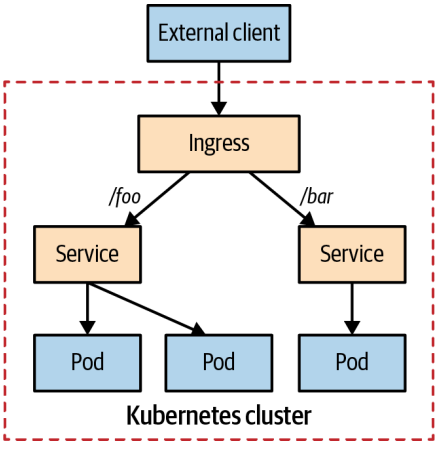


Figure 13-7. Application layer service discovery

OpenShift Routes

Red Hat OpenShift is a popular enterprise distribution of Kubernetes. Besides being fully compliant with Kubernetes, OpenShift provides some additional features. One of these features is Routes, which are very similar to Ingress. They are so similar, in fact, the differences might be difficult to spot. First of all, Routes predates the introduction of the Ingress object in Kubernetes, so Routes can be considered a kind of predecessor of Ingress.

However, some technical differences still exist between Routes and Ingress objects:

- A Route is picked up automatically by the OpenShift-integrated HAProxy load balancer, so there is no requirement for an extra Ingress controller to be installed.
- You can use additional TLS termination modes like re-encryption or pass-through for the leg to the Service.
- Multiple weighted backends for splitting traffic can be used.
- Wildcard domains are supported.

Having said all that, you can use Ingress on OpenShift too. So you have the choice when using OpenShift.

Discussion

In this chapter, we covered the favorite service discovery mechanisms on Kubernetes. Discovery of dynamic Pods from within the cluster is always achieved through the Service resource, though different options can lead to different implementations. The Service abstraction is a high-level cloud native way of configuring low-level details such as virtual IP addresses, iptables, DNS records, or environment variables. Service discovery from outside the cluster builds on top of the Service abstraction and focuses on exposing the Services to the outside world. While a `NodePort` provides the basics of exposing Services, a highly available setup requires integration with the platform infrastructure provider.

Table 13-1 summarizes the various ways service discovery is implemented in Kubernetes. This table aims to organize the various service discovery mechanisms in this chapter from more straightforward to more complex. We hope it can help you build a mental model and understand them better.

Table 13-1. Service Discovery mechanisms

Name	Configuration	Client type	Summary
ClusterIP	`type: ClusterIP` `.spec.selector`	Internal	The most common internal discovery mechanism
Manual IP	`type: ClusterIP` `kind: Endpoints`	Internal	External IP discovery
Manual FQDN	`type: ExternalName` `.spec.externalName`	Internal	External FQDN discovery
Headless Service	`type: ClusterIP` `.spec.clusterIP: None`	Internal	DNS-based discovery without a virtual IP
NodePort	`type: NodePort`	External	Preferred for non-HTTP traffic
LoadBalancer	`type: LoadBalancer`	External	Requires supporting cloud infrastructure
Ingress	`kind: Ingress`	External	L7/HTTP-based smart routing mechanism

This chapter gave a comprehensive overview of all the core concepts in Kubernetes for accessing and discovering services. However, the journey does not stop here. With the *Knative* project, new primitives on top of Kubernetes have been introduced, which help application developers with advanced serving and eventing.

In the context of the *Service Discovery* pattern, the *Knative Serving* subproject is of particular interest as it introduces a new Service resource with the same kind as the Services introduced here (but with a different API group). Knative Serving provides support for application revision but also for a very flexible scaling of services behind a load balancer. We give a short shout-out to Knative Serving in "Knative" on page 317, but a full discussion of Knative is beyond the scope of this book. In "More

Information" on page 333, you will find links that point to detailed information about Knative.

More Information

- Service Discovery Example
- Kubernetes Service
- DNS for Services and Pods
- Debug Services
- Using Source IP
- Create an External Load Balancer
- Ingress
- Kubernetes NodePort Versus LoadBalancer Versus Ingress? When Should I Use What?
- Kubernetes Ingress Versus OpenShift Route

CHAPTER 14

Self Awareness

Some applications need to be self-aware and require information about themselves. The *Self Awareness* pattern describes the Kubernetes *downward API* that provides a simple mechanism for introspection and metadata injection to applications.

Problem

For the majority of use cases, cloud native applications are stateless and disposable without an identity relevant to other applications. However, sometimes even these kinds of applications need to have information about themselves and the environment they are running in. That may include information known only at runtime, such as the Pod name, Pod IP address, and the hostname on which the application is placed. Or, other static information defined at Pod level such as the specific resource requests and limits, or some dynamic information such as annotations and labels that could be altered by the user at runtime.

For example, depending on the resources made available to the container, you may want to tune the application thread-pool size, or change the garbage collection algorithm or memory allocation. You may want to use the Pod name and the hostname while logging information, or while sending metrics to a central server. You may want to discover other Pods in the same namespace with a specific label and join them into a clustered application. For these and other use cases, Kubernetes provides the downward API.

Solution

The requirements that we've described and the following solution are not specific only to containers but are present in any dynamic environment where the metadata of resources changes. For example, AWS offers Instance Metadata and User Data services that can be queried from any EC2 instance to retrieve metadata about the EC2 instance itself. Similarly, AWS ECS provides APIs that can be queried by the containers and retrieve information about the container cluster.

The Kubernetes approach is even more elegant and easier to use. The *downward API* allows you to pass metadata about the Pod to the containers and the cluster through environment variables and files. These are the same mechanisms we used for passing application-related data from ConfigMaps and Secrets. But in this case, the data is not created by us. Instead, we specify the keys that interest us, and Kubernetes populates the values dynamically. Figure 14-1 gives an overview of how the downward API injects resource and runtime information into interested Pods.

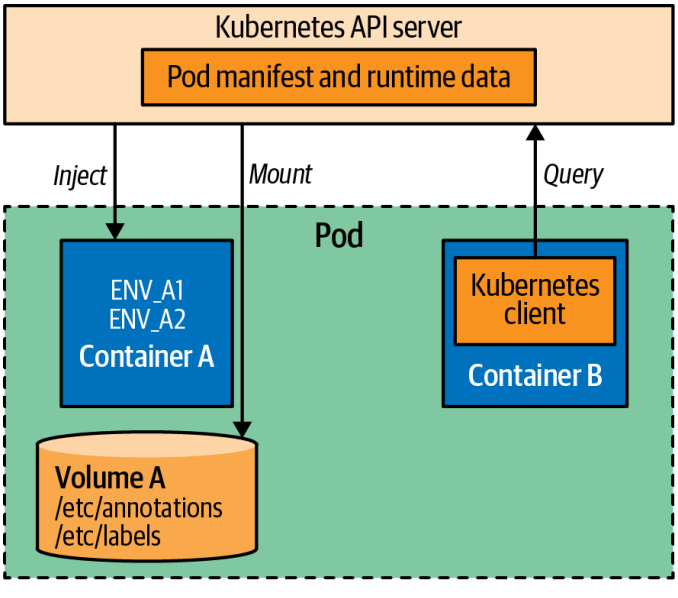


Figure 14-1. Application introspection mechanisms

The main point here is that with the downward API, the metadata is injected into your Pod and made available locally. The application does not need to use a client and interact with the Kubernetes API and can remain Kubernetes-agnostic. Let's see how easy it is to request metadata through environment variables in Example 14-1.

Example 14-1. Environment variables from downward API

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    env:
    - name: POD_IP
      valueFrom:
        fieldRef:                              ❶
          fieldPath: status.podIP
    - name: MEMORY_LIMIT
      valueFrom:
        resourceFieldRef:
          containerName: random-generator      ❷
          resource: limits.memory
```

❶ The environment variable `POD_IP` is set from the properties of this Pod and comes into existence at Pod startup time.

❷ The environment variable `MEMORY_LIMIT` is set to the value of the memory resource limit of this container; the actual limit declaration is not shown here.

In this example, we use `fieldRef` to access Pod-level metadata. The keys shown in Table 14-1 are available for `fieldRef.fieldPath` both as environment variables and `downwardAPI` volumes.

Table 14-1. Downward API information available in `fieldRef.fieldPath`

Name	Description
`spec.nodeName`	Name of node hosting the Pod
`status.hostIP`	IP address of node hosting the Pod
`metadata.name`	Pod name
`metadata.namespace`	Namespace in which the Pod is running
`status.podIP`	Pod IP address
`spec.serviceAccountName`	ServiceAccount that is used for the Pod
`metadata.uid`	Unique ID of the Pod
`metadata.labels['key']`	Value of the Pod's label *key*
`metadata.annotations['key']`	Value of the Pod's annotation *key*

As with `fieldRef`, we use `resourceFieldRef` to access metadata specific to a container's resource specification belonging to the Pod. This metadata is specific to a container and is specified with `resourceFieldRef.container`. When used as an environment variable, by default the current container is used. Possible keys for `resourceFieldRef.resource` are shown in Table 14-2. Resource declarations are explained in Chapter 2, "Predictable Demands".

Table 14-2. Downward API information available in `resourceFieldRef.resource`

Name	Description
`requests.cpu`	A container's CPU request
`limits.cpu`	A container's CPU limit
`requests.memory`	A container's memory request
`limits.memory`	A container's memory limit
`requests.hugepages-<size>`	A container's hugepages request (e.g., `requests.hugepages-1Gi`)
`limits.hugepages-<size>`	A container's hugepages limit (e.g., `limits.hugepages-1Gi`)
`requests.ephemeral-storage`	A container's ephemeral-storage request
`limits.ephemeral-storage`	A container's ephemeral-storage limit

A user can change certain metadata such as labels and annotations while a Pod is running. Unless the Pod is restarted, environment variables will not reflect such a change. But `downwardAPI` volumes can reflect updates to labels and annotations. In addition to the individual fields described previously, `downwardAPI` volumes can capture all Pod labels and annotations into files with `metadata.labels` and `metadata.annotations` references. Example 14-2 shows how such volumes can be used.

Example 14-2. Downward API through volumes

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    volumeMounts:
    - name: pod-info                    ❶
      mountPath: /pod-info
  volumes:
  - name: pod-info
    downwardAPI:
      items:
      - path: labels                    ❷
        fieldRef:
          fieldPath: metadata.labels
```

```
  - path: annotations                    ❸
    fieldRef:
      fieldPath: metadata.annotations
```

❶ Values from the downward API can be mounted as files into the Pod.

❷ The file `labels` contain all labels, line by line, in the format `name=value`. This file gets updated when labels are changing.

❸ The `annotations` file holds all annotations in the same format as the labels.

With volumes, if the metadata changes while the Pod is running, it is reflected in the volume files. But it is still up to the consuming application to detect the file change and read the updated data accordingly. If such a functionality is not implemented in the application, a Pod restart still might be required.

Discussion

Often, an application needs to be self-aware and have information about itself and the environment in which it is running. Kubernetes provides nonintrusive mechanisms for introspection and metadata injection. One of the downsides of the downward API is that it offers a fixed number of keys that can be referenced. If your application needs more data, especially about other resources or cluster-related metadata, it has to be queried on the API Server. This technique is used by many applications that query the API Server to discover other Pods in the same namespace that have certain labels or annotations. Then the application may form a cluster with the discovered Pods and sync state. It is also used by monitoring applications to discover Pods of interest and then start instrumenting them.

Many client libraries are available for different languages to interact with the Kubernetes API Server to obtain more self-referring information that goes beyond what the downward API provides.

More Information

- Self Awareness Example
- AWS EC2: Instance Metadata and User Data
- Expose Pod Information to Containers Through Files
- Expose Pod Information to Containers Through Environment Variables
- Downward API: Available Fields

PART III
Structural Patterns

Container images and containers are similar to classes and objects in the object-oriented world. Container images are the blueprint from which containers are instantiated. But these containers do not run in isolation; they run in other abstractions called Pods, where they interact with other containers.

The patterns in this category are focused on structuring and organizing containers in a Pod to satisfy different use cases. Pods provide unique runtime capabilities. The forces that affect containers in Pods result in the patterns discussed in the following chapters:

- Chapter 15, "Init Container", introduces a lifecycle for initialization-related tasks, decoupled from the main application responsibilities.
- Chapter 16, "Sidecar", describes how to extend and enhance the functionality of a preexisting container without changing it.
- Chapter 17, "Adapter", takes a heterogeneous system and makes it conform to a consistent unified interface that can be consumed by the outside world.
- Chapter 18, "Ambassador", describes a proxy that decouples access to external services.

CHAPTER 15

Init Container

The *Init Container* pattern enables separation of concerns by providing a separate lifecycle for initialization-related tasks distinct from the main application containers. In this chapter, we look closely at this fundamental Kubernetes concept that is used in many other patterns when initialization logic is required.

Problem

Initialization is a widespread concern in many programming languages. Some languages have it covered as part of the language, and some use naming conventions and patterns to indicate a construct as the initializer. For example, in the Java programming language, to instantiate an object that requires some setup, we use the constructor (or static blocks for fancier use cases). Constructors are guaranteed to run as the first thing within the object, and they are guaranteed to run only once by the managing runtime (this is just an example; we don't go into detail here on the different languages and corner cases). Moreover, we can use the constructor to validate preconditions such as mandatory parameters. We also use constructors to initialize the instance fields with incoming arguments or default values.

Init containers are similar but are at the Pod level rather than at the Java class level. So if you have one or more containers in a Pod that represent your main application, these containers may have prerequisites before starting up. These may include special permissions setup on the filesystem, database schema setup, or application seed data installation. Also, this initializing logic may require tools and libraries that cannot be included in the application image. For security reasons, the application image may not have permissions to perform the initializing activities. Alternatively, you may want to delay the startup of your application until an external dependency is satisfied. For all these kinds of use cases, Kubernetes uses init containers as implementation

of this pattern, which allow separation of initializing activities from the main application duties.

Solution

Init containers in Kubernetes are part of the Pod definition, and they separate all containers in a Pod into two groups: init containers and application containers. All init containers are executed in a sequence, one by one, and all of them have to terminate successfully before the application containers are started up. In that sense, init containers are like constructor instructions in a Java class that help object initialization. Application containers, on the other hand, run in parallel, and the startup order is arbitrary. The execution flow is demonstrated in Figure 15-1.

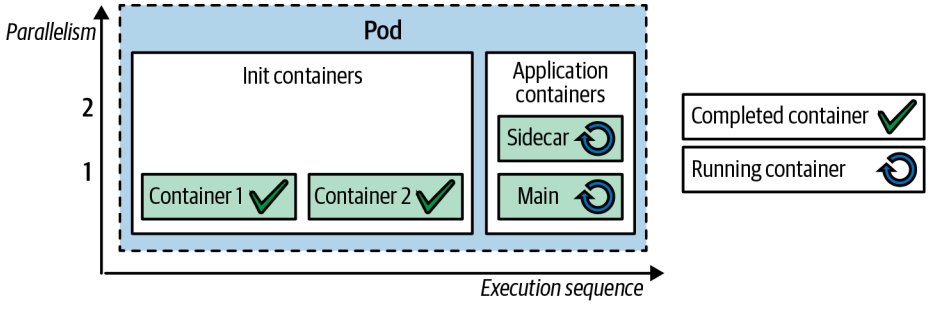


Figure 15-1. Init and application containers in a Pod

Typically, init containers are expected to be small, run quickly, and complete successfully, except when an init container is used to delay the start of a Pod while waiting for a dependency, in which case it may not terminate until the dependency is satisfied. If an init container fails, the whole Pod is restarted (unless it is marked with `RestartNever`), causing all init containers to run again. Thus, to prevent any side effects, making init containers idempotent is a good practice.

On one hand, init containers have all of the same capabilities as application containers: all of the containers are part of the same Pod, so they share resource limits, volumes, and security settings and end up placed on the same node. On the other hand, they have slightly different lifecycle, health-checking, and resource-handling semantics. There is no `livenessProbe`, `readinessProbe`, or `startupProbe` for init containers, as all init containers must terminate successfully before the Pod startup processes can continue with application containers.

Init containers also affect the way Pod resource requirements are calculated for scheduling, autoscaling, and quota management. Given the ordering in the execution of all containers in a Pod (first, init containers run a sequence, then all application

containers run in parallel), the effective Pod-level request and limit values become the highest values of the following two groups:

- The highest init container request/limit value
- The sum of all application container values for request/limit

A consequence of this behavior is that if you have init containers with high resource demands and application containers with low resource demands, the Pod-level request and limit values affecting the scheduling will be based on the higher value of the init containers, as demonstrated in Figure 15-2.

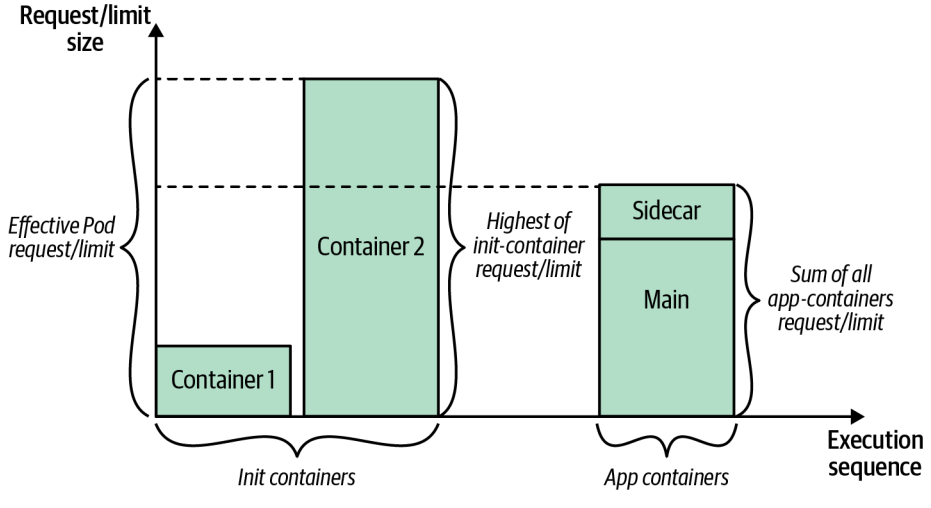


Figure 15-2. Effective Pod request/limit calculation

This setup is not resource-efficient. Even if init containers run for a short period of time and there is available capacity on the node for the majority of the time, no other Pod can use it.

Moreover, init containers enable separation of concerns and allow you to keep containers single-purposed. An application container can be created by the application engineer and focus on the application logic only. A deployment engineer can author an init container and focus on configuration and initialization tasks only. We demonstrate this in Example 15-1, which has one application container based on an HTTP server that serves files.

The container provides a generic HTTP-serving capability and does not make any assumptions about where the files to serve might come from for the different use cases. In the same Pod, an init container provides Git client capability, and its sole purpose is to clone a Git repo. Since both containers are part of the same Pod, they

can access the same volume to share data. We use the same mechanism to share the cloned files from the init container to the application container.

Example 15-1 shows an init container that copies data into an empty volume.

Example 15-1. Init Container

```
apiVersion: v1
kind: Pod
metadata:
  name: www
  labels:
    app: www
spec:
  initContainers:
  - name: download
    image: bitnami/git
    command:                          ❶
    - git
    - clone
    - https://github.com/mdn/beginner-html-site-scripted
    - /var/lib/data
    volumeMounts:                     ❷
    - mountPath: /var/lib/data
      name: source
  containers:
  - name: run
    image: centos/httpd
    ports:
    - containerPort: 80
    volumeMounts:                     ❷
    - mountPath: /var/www/html
      name: source
  volumes:                            ❸
  - emptyDir: {}
    name: source
```

❶ Clone an external Git repository into the mounted directory.

❷ Shared volume used by both init container and the application container.

❸ Empty directory used on the node for sharing data.

We could have achieved the same effect by using ConfigMap or PersistentVolumes but want to demonstrate how init containers work here. This example illustrates a typical usage pattern of an init container sharing a volume with the main container.

For debugging the outcome of init containers, it helps if the command of the application container is replaced temporarily with a dummy `sleep` command so that you have time to examine the situation. This trick is particularly useful if your init container fails to start up and your application fails to start because the configuration is missing or broken. The following command within the Pod declaration gives you an hour to debug the volumes mounted by entering the Pod with `kubectl exec -it <pod> sh`:

```
command:
- /bin/sh
- "-c"
- "sleep 3600"
```

A similar effect can be achieved by using a sidecar, as described next in Chapter 16, "Sidecar", where the HTTP server container and the Git container are running side by side as application containers. But with the sidecar approach, there is no way of knowing which container will run first, and sidecar is meant to be used when containers run side by side continuously. We could also use a sidecar and init container together if both a guaranteed initialization and a constant update of the data are required.

More Initialization Techniques

As you have seen, an init container is a Pod-level construct that gets activated after a Pod has been started. A few other related techniques used to initialize Kubernetes resources are different from init containers and are worth listing here for completeness:

Admission controllers
: This set of plugins intercepts every request to the Kubernetes API Server before persistence of the object and can mutate or validate it. There are many admission controllers for applying checks, enforcing limits, and setting default values, but all are compiled into the `kube-apiserver` binary and configured by a cluster administrator when the API Server starts up. This plugin system is not very flexible, which is why admission webhooks were added to Kubernetes.

Admission webhooks
: These components are external admission controllers that perform HTTP callbacks for any matching request. There are two types of admission webhooks: the *mutating webhook* (which can change resources to enforce custom defaults) and the *validating webhook* (which can reject resources to enforce custom admission policies). This concept of external controllers allows admission webhooks to be developed out of Kubernetes and configured at runtime.

There used to be other techniques for initializing Kubernetes resources, such as Initializers and PodPresets, which were eventually deprecated and removed. Nowadays other projects such as Metacontroller and Kyverno use admission webhooks or the *Operator* pattern to mutate Kubernetes resources and intervene in the initialization process. These techniques differ from init containers because they validate and mutate resources at creation time.

In contrast, the *Init Container* pattern discussed in this chapter is something that activates and performs its responsibilities during startup of the Pod. You could use admission webhooks, for example, to inject an init container into any Pod that doesn't have one already. For example, Istio, which is a popular service mesh project, uses a combination of techniques discussed in this chapter to inject its proxies into application Pods. Istio uses Kubernetes mutating admission webhooks for automatic sidecar and init container injection into the Pod definition at Pod definition creation time. When such a Pod is starting up, Istio's init container configures the Pod environment to redirect inbound and outbound traffic from the application to the Envoy proxy sidecar. The init container runs before any other container and configures iptable rules to insert the Envoy proxy in the request path of the application before any traffic reaches the application. This separation of containers is good for lifecycle management and also because the init container in this case requires elevated permissions to configure traffic redirection, which can pose a security threat. This is an example of how many initialization activities can be performed before an application container starts up.

In the end, the most significant difference is that init containers can be used by developers deploying on Kubernetes, whereas admission webhooks help administrators and various frameworks control and alter the container initialization process.

Discussion

So why separate containers in a Pod into two groups? Why not just use an application container with a bit of scripting in a Pod for initialization if required? The answer is that these two groups of containers have different lifecycles, purposes, and even authors in some cases.

Having init containers run before application containers, and more importantly, having init containers run in stages that progress only when the current init container completes successfully, means you can be sure at every step of the initialization that the previous step has completed successfully, and you can progress to the next stage. Application containers, in contrast, run in parallel and do not provide similar guarantees as init containers. With this distinction in hand, we can create containers focused on initialization or application-focused tasks, and reuse them in different contexts by organizing them in Pods with predictable guarantees.

More Information

- Init Container Example
- Init Containers
- Configuring Pod Initialization
- Admission Controllers Reference
- Dynamic Admission Control
- Metacontroller
- Kyverno
- Demystifying Istio's Sidecar Injection Model
- Object Initialization in Swift

CHAPTER 16
Sidecar

A sidecar container extends and enhances the functionality of a preexisting container without changing it. The *Sidecar* pattern is one of the fundamental container patterns that allows single-purpose containers to cooperate closely together. In this chapter, you'll learn all about the basic sidecar concept. The specialized follow-up patterns, *Adapter* and *Ambassador*, are discussed in Chapters 17 and 18, respectively.

Problem

Containers are a popular packaging technology that allow developers and system administrators to build, ship, and run applications in a unified way. A container represents a natural boundary for a unit of functionality with a distinct runtime, release cycle, API, and team owning it. A proper container behaves like a single Linux process—solves one problem and does it well—and is created with the idea of replaceability and reuse. This last part is essential as it allows us to build applications more quickly by leveraging existing specialized containers.

Today, to make an HTTP call, we don't have to write a client library but can use an existing one. In the same way, to serve a website, we don't have to create a container for a web server but can use an existing one. This approach allows developers to avoid reinventing the wheel and create an ecosystem with a smaller number of better-quality containers to maintain. However, having single-purpose reusable containers requires ways of extending the functionality of a container and a means for collaboration among containers. The sidecar pattern describes this kind of collaboration, where a container enhances the functionality of another preexisting container.

Solution

In Chapter 1, we described how the Pod primitive allows us to combine multiple containers into a single unit. Behind the scenes, at runtime, a Pod is a container as well, but it starts as a paused process (literally with the `pause` command) before all other containers in the Pod. It is not doing anything other than holding all the Linux namespaces the application containers use to interact throughout the Pod's lifetime. Apart from this implementation detail, what is more interesting is all the characteristics that the Pod abstraction provides.

The Pod is such a fundamental primitive that it is present in many cloud native platforms under different names but always with similar capabilities. A Pod as the deployment unit puts certain runtime constraints on the containers belonging to it. For example, all containers end up deployed to the same node, and they share the same Pod lifecycle. In addition, a Pod allows its containers to share volumes and communicate over the local network or host IPC. These are the reasons users put a group of containers into a Pod. *Sidecar* (sometimes also called *Sidekick*) is used to describe the scenario of a container being put into a Pod to extend and enhance another container's behavior.

A typical example demonstrating this pattern is of an HTTP server and a Git synchronizer. The HTTP server container is focused only on serving files over HTTP and does not know how or where the files are coming from. Similarly, the Git synchronizer container's only goal is to sync data from a Git server to the local filesystem. It does not care what happens once synced—its only concern is keeping the local folder in sync with the remote Git server. Example 16-1 shows a Pod definition with these two containers configured to use a volume for file exchange.

Example 16-1. Pod with a sidecar

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: app
    image: centos/httpd                ❶
    volumeMounts:
    - mountPath: /var/www/html         ❸
      name: git
  - name: poll
    image: bitnami/git                 ❷
    volumeMounts:
    - mountPath: /var/lib/data         ❸
      name: git
    env:
```

```
    - name: GIT_REPO
      value: https://github.com/mdn/beginner-html-site-scripted
    command: [ "sh", "-c" ]
    args:
    - |
      git clone $(GIT_REPO) .
      while true; do
        sleep 60
        git pull
      done
    workingDir: /var/lib/data
  volumes:
  - emptyDir: {}
    name: git
```

❶ Main application container serving files over HTTP.

❷ Sidecar container running in parallel and pulling data from a Git server.

❸ Shared location for exchanging data between the sidecar and main application container as mounted in the `app` and `poll` containers, respectively.

This example shows how the Git synchronizer enhances the HTTP server's behavior with content to serve and keeps it synchronized. We could also say that both containers collaborate and are equally important, but in a *Sidecar* pattern, there is a main container and a helper container that enhance the collective behavior. Typically, the main container is the first one listed in the containers list, and it represents the default container (e.g., when we run the command `kubectl exec`).

This simple pattern, illustrated in Figure 16-1, allows runtime collaboration of containers and at the same time enables separation of concerns for both containers, which might be owned by separate teams, using different programming languages, with different release cycles, etc. It also promotes replaceability and reuse of containers as the HTTP server, and the Git synchronizer can be reused in other applications and different configuration either as a single container in a Pod or again in collaboration with other containers.

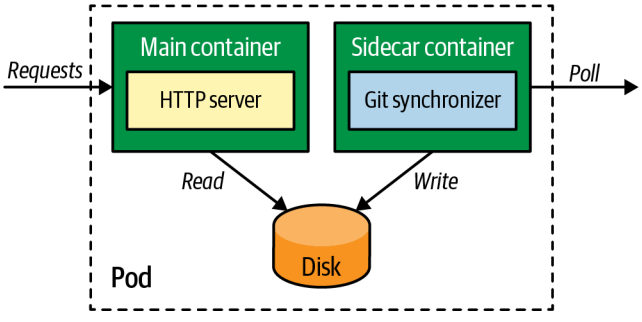


Figure 16-1. Sidecar pattern

Discussion

Previously we said that container images are like classes, and containers are like the objects in object-oriented programming (OOP). If we continue this analogy, extending a container to enhance its functionality is similar to inheritance in OOP, and having multiple containers collaborating in a Pod is similar to composition in OOP. While both approaches allow code reuse, inheritance involves tighter coupling between containers and represents an "is-a" relationship between containers.

On the other hand, a composition in a Pod represents a "has-a" relationship, and it is more flexible because it doesn't couple containers together at build time, giving you the ability to later swap containers in the Pod definition. With the composition approach, you have multiple containers (processes) running, health checked, restarted, and consuming resources, as the main application container does. Modern sidecar containers are small and consume minimal resources, but you have to decide whether it is worth running a separate process or whether it is better to merge it into the main container.

We see two dominating approaches for using sidecars: transparent sidecars that are invisible to the application, and explicit sidecars that the main application interacts with over well-defined APIs. Envoy proxy is an example of a transparent sidecar that runs alongside the main container and abstracts the network by providing common features such as Transport Layer Security (TLS), load balancing, automatic retries, circuit breaking, global rate limiting, observability of L7 traffic, distributed tracing, and more. All of these features become available to the application by transparently attaching the sidecar container and intercepting all the incoming and outgoing traffic to the main container. This is similar to aspect-oriented programming, in that with additional containers, we introduce orthogonal capabilities to the Pod without touching the main container.

An example of an explicit proxy that uses the sidecar architecture is Dapr. A Dapr sidecar container is injected into a Pod and offers features such as reliable service

invocation, publish-subscribe, bindings to external systems, state abstraction, observability, distributed tracing, and more. The primary difference between Dapr and Envoy proxy is that Dapr does not intercept all the networking traffic going in and out of the application. Rather, Dapr features are exposed over HTTP and gRPC APIs, which the application invokes or subscribes to.

More Information

- Sidecar Example
- Pods
- Design Patterns for Container-Based Distributed Systems
- Prana: A Sidecar for Your Netflix PaaS-Based Applications and Services
- Tin-Can Phone: Patterns to Add Authorization and Encryption to Legacy Applications
- Envoy
- Dapr
- The Almighty Pause Container
- Sidecar Pattern

CHAPTER 17

Adapter

The *Adapter* pattern takes a heterogeneous containerized system and makes it conform to a consistent, unified interface with a standardized and normalized format that can be consumed by the outside world. The *Adapter* pattern inherits all its characteristics from the *Sidecar* pattern but has the single purpose of providing adapted access to the application.

Problem

Containers allow us to package and run applications written in different libraries and languages in a unified way. Today, it is common to see multiple teams using different technologies and creating distributed systems composed of heterogeneous components. This heterogeneity can cause difficulties when all components have to be treated in a unified way by other systems. The *Adapter* pattern offers a solution by hiding the complexity of a system and providing unified access to it.

Solution

The best way to illustrate the *Adapter* pattern is through an example. A major prerequisite for successfully running and supporting distributed systems is providing detailed monitoring and alerting. Moreover, if we have a distributed system composed of multiple services we want to monitor, we may use an external monitoring tool to poll metrics from every service and record them.

However, services written in different languages may not have the same capabilities and may not expose metrics in the same format expected by the monitoring tool. This diversity creates a challenge for monitoring such a heterogeneous application from a single monitoring solution that expects a unified view of the whole system. With the *Adapter* pattern, it is possible to provide a unified monitoring interface by exporting

metrics from various application containers into one standard format and protocol. In Figure 17-1, an adapter container translates locally stored metrics information into the external format the monitoring server understands.

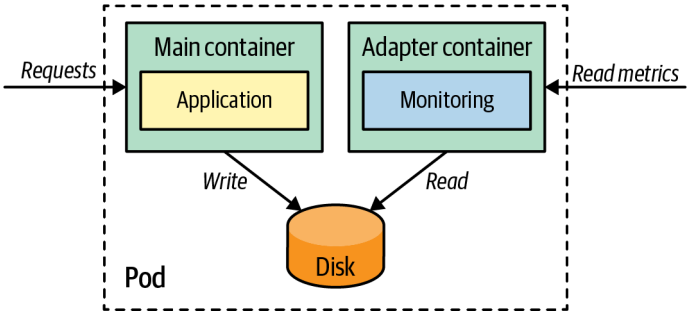


Figure 17-1. Adapter pattern

With this approach, every service represented by a Pod, in addition to the main application container, would have another container that knows how to read the custom application-specific metrics and expose them in a generic format understandable by the monitoring tool. We could have one adapter container that knows how to export Java-based metrics over HTTP and another adapter container in a different Pod that exposes Python-based metrics over HTTP. For the monitoring tool, all metrics would be available over HTTP and in a common, normalized format.

For a concrete implementation of this pattern, let's add the adapter shown in Figure 17-1 to our sample random generator application. When appropriately configured, it writes out a log file with the random-number generator and includes the time it took to create the random number. We want to monitor this time with Prometheus. Unfortunately, the log format doesn't match the format Prometheus expects. Also, we need to offer this information over an HTTP endpoint so that a Prometheus server can scrape the value.

For this use case, an adapter is a perfect fit: a sidecar container starts a small HTTP server and on every request, reads the custom log file and transforms it into a Prometheus-understandable format. Example 17-1 shows a Deployment with such an adapter. This configuration allows a decoupled Prometheus monitoring setup without the main application needing to know anything about Prometheus. The full example in the book's GitHub repository demonstrates this setup together with a Prometheus installation.

Example 17-1. Adapter delivering Prometheus-conformant output

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: random-generator
spec:
  replicas: 1
  selector:
    matchLabels:
      app: random-generator
  template:
    metadata:
      labels:
        app: random-generator
    spec:
      containers:
      - image: k8spatterns/random-generator:1.0          ❶
        name: random-generator
        env:
        - name: LOG_FILE                                 ❷
          value: /logs/random.log
        ports:
        - containerPort: 8080
          protocol: TCP
        volumeMounts:                                    ❸
        - mountPath: /logs
          name: log-volume
      # ------------------------------------------
      - image: k8spatterns/random-generator-exporter     ❹
        name: prometheus-adapter
        env:
        - name: LOG_FILE                                 ❺
          value: /logs/random.log
        ports:
        - containerPort: 9889
          protocol: TCP
        volumeMounts:                                    ❻
        - mountPath: /logs
          name: log-volume
      volumes:
      - name: log-volume                                 ❼
        emptyDir: {}
```

❶ Main application container with the random generator service exposed on 8080.

❷ Path to the log file containing the timing information about random-number generation.

❸ Directory shared with the Prometheus Adapter container.

❹ Prometheus exporter image, exporting on port 9889.

❺ Path to the same log file to which the main application is logging.

❻ Shared volume is also mounted in the adapter container.

❼ Files are shared via an `emptyDir` volume from the node's filesystem.

Another use of this pattern is logging. Different containers may log information in different formats and levels of detail. An adapter can normalize that information, clean it up, enrich it with contextual information by using the *Self Awareness* pattern described in Chapter 14, and then make it available for pickup by the centralized log aggregator.

Discussion

The *Adapter* is a specialization of the *Sidecar* pattern explained in Chapter 16. It acts as a reverse proxy to a heterogeneous system by hiding its complexity behind a unified interface. Using a distinct name different from the generic *Sidecar* pattern allows us to more precisely communicate the purpose of this pattern.

In the next chapter, you'll get to know another sidecar variation: the *Ambassador* pattern, which acts as a proxy to the outside world.

More Information

- Adapter Example

CHAPTER 18

Ambassador

The *Ambassador* pattern is a specialized sidecar responsible for hiding external complexities and providing a unified interface for accessing services outside the Pod. In this chapter, you will see how the *Ambassador* pattern can act as a proxy and decouple the main container from directly accessing external dependencies.

Problem

Containerized services don't exist in isolation and very often have to access other services that may be difficult to reach in a reliable way. The difficulty in accessing other services may be due to dynamic and changing addresses, the need for load balancing of clustered service instances, an unreliable protocol, or difficult data formats. Ideally, containers should be single-purposed and reusable in different contexts. But if we have a container that provides some business functionality and consumes an external service in a specialized way, the container will have more than one responsibility.

Consuming the external service may require a special service discovery library that we do not want to put in our container. Or we may want to swap different kinds of services by using different kinds of service-discovery libraries and methods. This technique of abstracting and isolating the logic for accessing other services in the outside world is the goal of this *Ambassador* pattern.

Solution

To demonstrate the pattern, we will use a cache for an application. Accessing a local cache in the development environment may be a simple configuration, but in the production environment, we may need a client configuration that can connect to the different shards of the cache. Another example is consuming a service by looking

it up in a registry and performing client-side service discovery. A third example is consuming a service over a nonreliable protocol such as HTTP, so to protect our application, we have to use circuit-breaker logic, configure timeouts, perform retries, and more.

In all of these cases, we can use an ambassador container that hides the complexity of accessing the external services and provides a simplified view and access to the main application container over localhost. Figures 18-1 and 18-2 show how an ambassador Pod can decouple access to a key-value store by connecting to an ambassador container listening on a local port. In Figure 18-1, we see how data access can be delegated to a fully distributed remote store like etcd.

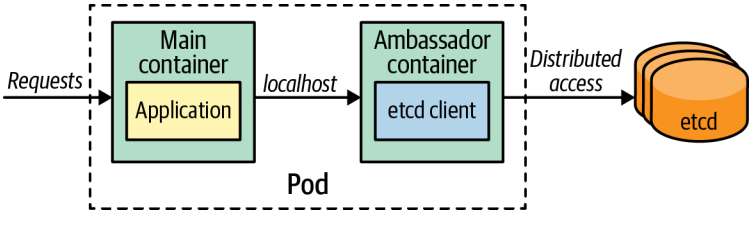


Figure 18-1. Ambassador for accessing a remote distributed cache

For development purposes, this ambassador container can be easily exchanged with a locally running in-memory key-value store like memcached (as shown in Figure 18-2).

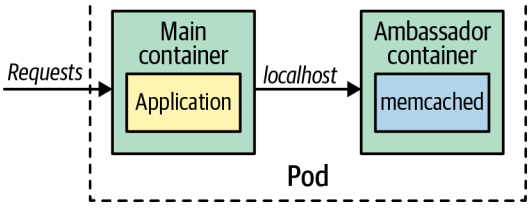


Figure 18-2. Ambassador for accessing a local cache

Example 18-1 shows an ambassador that runs parallel to a REST service. Before returning its response, the REST service logs the generated data by sending it to a fixed URL: *http://localhost:9009*. The ambassador process listens in on this port and processes the data. In this example, it prints the data out just to the console, but it could also do something more sophisticated like forward the data to a full logging infrastructure. For the REST service, it doesn't matter what happens to the log data, and you can easily exchange the ambassador by reconfiguring the Pod without touching the main container.

Example 18-1. Ambassador processing log output

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
  labels:
    app: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0                    ❶
    name: main
    env:
    - name: LOG_URL                                            ❷
      value: http://localhost:9009
    ports:
    - containerPort: 8080
      protocol: TCP
  - image: k8spatterns/random-generator-log-ambassador         ❸
    name: ambassador
```

❶ Main application container providing a REST service for generating random numbers.

❷ Connection URL for communicating with the ambassador via localhost.

❸ Ambassador running in parallel and listening on port 9009 (which is not exposed to the outside of the Pod).

Discussion

At a higher level, the *Ambassador* pattern is a *Sidecar* pattern. The main difference between ambassador and sidecar is that an ambassador does not enhance the main application with additional capability. Instead, it acts merely as a smart proxy to the outside world (this pattern is sometimes referred to as the *Proxy* pattern). This pattern can be useful for legacy applications that are difficult to modify and extend with modern networking concepts such as monitoring, logging, routing, and resiliency patterns.

The benefits of the *Ambassador* pattern are similar to those of the *Sidecar* pattern—both allow you to keep containers single-purposed and reusable. With such a pattern, our application container can focus on its business logic and delegate the responsibility and specifics of consuming the external service to another specialized container. This also allows you to create specialized and reusable ambassador containers that can be combined with other application containers.

More Information

- Ambassador Example
- How to Use the Ambassador Pattern to Dynamically Configure Services on CoreOS
- Modifications to the CoreOS Ambassador Pattern

PART IV
Configuration Patterns

Every application needs to be configured, and the easiest way to do this is by storing configurations in the source code. However, this approach has the side effect of code and configuration living and dying together. We need the flexibility to adapt configurations without modifying the application and recreating its container image. In fact, mixing code and configuration is an antipattern for a continuous delivery approach, where the application is created once and then moves unaltered through the various stages of the deployment pipeline until it reaches production. The way to achieve this separation of code and configuration is by using external configuration data, which is different for each environment. The patterns in the following chapters are all about customizing and adapting applications with external configurations for various environments:

- Chapter 19, "EnvVar Configuration", uses environment variables to store configuration data.
- Chapter 20, "Configuration Resource", uses Kubernetes resources like ConfigMaps or Secrets to store configuration information.
- Chapter 21, "Immutable Configuration", brings immutability to large configuration sets by putting them into containers linked to the application at runtime.
- Chapter 22, "Configuration Template", is useful when large configuration files need to be managed for multiple environments that differ only slightly.

CHAPTER 19

EnvVar Configuration

In this *EnvVar Configuration* pattern, we look into the simplest way to configure applications. For small sets of configuration values, the easiest way to externalize configuration is by putting them into universally supported environment variables. We'll see different ways of declaring environment variables in Kubernetes but also the limitations of using environment variables for complex configurations.

Problem

Every nontrivial application needs some configuration for accessing data sources, external services, or production-level tuning. And we knew well before the twelve-factor app manifesto that it is a bad thing to hardcode configurations within the application. Instead, the configuration should be *externalized* so that we can change it even after the application has been built. That provides even more value for containerized applications that enable and promote sharing of immutable application artifacts. But how can this be done best in a containerized world?

Solution

The twelve-factor app manifesto recommends using environment variables for storing application configurations. This approach is simple and works for any environment and platform. Every operating system knows how to define environment variables and how to propagate them to applications, and every programming language also allows easy access to these environment variables. It is fair to claim that environment variables are universally applicable. When using environment variables, a typical usage pattern is to define hardcoded default values during build time, which we can then overwrite at runtime. Let's see some concrete examples of how this works in Docker and Kubernetes.

For Docker images, environment variables can be defined directly in Dockerfiles with the `ENV` directive. You can define them line by line or all in a single line, as shown in Example 19-1.

Example 19-1. Example Dockerfile with environment variables

```
FROM openjdk:11
ENV PATTERN "EnvVar Configuration"
ENV LOG_FILE "/tmp/random.log"
ENV SEED "1349093094"

# Alternatively:
ENV PATTERN="EnvVar Configuration" LOG_FILE=/tmp/random.log SEED=1349093094
...
```

Then a Java application running in such a container can easily access the variables with a call to the Java standard library, as shown in Example 19-2.

Example 19-2. Reading environment variables in Java

```
public Random initRandom() {
  long seed = Long.parseLong(System.getenv("SEED"));
  return new Random(seed);       ❶
}
```

❶ Initializes a random-number generator with a seed from an EnvVar.

Directly running such an image will use the default hardcoded values. But in most cases, you want to override these parameters from outside the image.

When running such an image directly with Docker, environment variables can be set from the command line by calling Docker, as in Example 19-3.

Example 19-3. Set environment variables when starting a Docker container

```
docker run -e PATTERN="EnvVarConfiguration" \
           -e LOG_FILE="/tmp/random.log" \
           -e SEED="147110834325" \
           k8spatterns/random-generator:1.0
```

For Kubernetes, these types of environment variables can be set directly in the Pod specification of a controller like Deployment or ReplicaSet (as shown in Example 19-4).

Example 19-4. Deployment with environment variables set

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    env:
    - name: LOG_FILE
      value: /tmp/random.log                    ❶
    - name: PATTERN
      valueFrom:
        configMapKeyRef:                        ❷
          name: random-generator-config         ❸
          key: pattern                          ❹
    - name: SEED
      valueFrom:
        secretKeyRef:                           ❺
          name: random-generator-secret
          key: seed
```

❶ EnvVar with a literal value.

❷ EnvVar from a ConfigMap.

❸ ConfigMap's name.

❹ Key within the ConfigMap to look for the EnvVar value.

❺ EnvVar from a Secret (lookup semantic is the same as for a ConfigMap).

In such a Pod template, you not only can attach values directly to environment variables (as for `LOG_FILE`), but also can use a delegation to Kubernetes Secrets and ConfigMaps. The advantage of ConfigMap and Secret indirection is that the environment variables can be managed independently from the Pod definition. Secret and ConfigMap and their pros and cons are explained in detail in Chapter 20, "Configuration Resource".

In the preceding example, the `SEED` variable comes from a Secret resource. While that is a perfectly valid use of Secret, it is also important to point out that environment variables are not secure. Putting sensitive, readable information into environment variables makes this information easy to read, and it may even leak into logs.

About Default Values

Default values make life easier, as they take away the burden of selecting a value for a configuration parameter you might not even know exists. They also play a significant role in the *convention over configuration* paradigm. However, defaults are not always a good idea. Sometimes they might even be an antipattern for an evolving application.

This is because *changing* default values retrospectively is a difficult task. First, changing default values means replacing them within the code, which requires a rebuild. Second, people relying on defaults (either by convention or consciously) will always be surprised when a default value changes. We have to communicate the change, and the user of such an application probably has to modify the calling code as well.

Changes in default values, however, often make sense, because it is hard to get default values right from the very beginning. It's essential that we consider a change in a default value as a *major change*, and if semantic versioning is in use, such a modification justifies a bump in the major version number. If unsatisfied with a given default value, it is often better to remove the default altogether and throw an error if the user does not provide a configuration value. This will at least break the application early and prominently instead of it doing something different and unexpected silently.

Considering all these issues, it is often the best solution to *avoid default values* from the very beginning if you cannot be 90% sure that a reasonable default will last for a long time. Passwords or database connection parameters are good candidates for not providing default values, as they depend highly on the environment and often cannot be reliably predicted. Also, if we do not use default values, the configuration information has to be provided explicitly, which serves as documentation too.

Instead of individually referring to configuration values from Secrets or ConfigMaps, you can also import *all* values of a particular Secret or ConfigMap with `envFrom`. We explain this field in Chapter 20, "Configuration Resource", when we talk about ConfigMaps and Secrets in detail.

Two other valuable features that can be used with environment variables are the downward API and *dependent variables*. You learned all about the downward API in Chapter 14, "Self Awareness", so let's have a look at dependent variables in Example 19-5 that allow you to reference previously defined variables in the value definition of other entries.

Example 19-5. Dependent environment variables

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    env:
    - name: PORT
      value: "8181"
    - name: IP                              ❶
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
    - name: MY_URL
      value: "https://$(IP):$(PORT)"        ❷
```

❶ Use the downward API to pick up the Pod's IP. The downward API is discussed in detail in Chapter 14, "Self Awareness".

❷ Include the previously defined environment variables `IP` and `PORT` to build up a URL.

With a `$(...)` notation, you can reference environment variables defined earlier in the `env` list or coming from an `envFrom` import. Kubernetes will resolve those references during the startup of the container. Be careful about the ordering, though: if you reference a variable defined later in the list, it will not be resolved, and the `$(...)` reference will be taken over literally. In addition, you can also reference environment variables with this syntax for Pod commands, as shown in Example 19-6.

Example 19-6. Using environment variables in a container's command definition

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
    - name: random-generator
      image: k8spatterns/random-generator:1.0
      command: [ "java", "RandomRunner", "$(OUTPUT_FILE)", "$(COUNT)" ] ❶
      env:                                                               ❷
        - name: OUTPUT_FILE
          value: "/numbers.txt"
        - name: COUNT
          valueFrom:
            configMapKeyRef:
              name: random-config
              key: RANDOM_COUNT
```

❶ Reference environment variables for the startup command of a container.

❷ Definition of the environment variables substituted in the commands.

Discussion

Environment variables are easy to use, and everybody knows about them. This concept maps smoothly to containers, and every runtime platform supports environment variables. But environment variables are not secure, and are good only for a decent number of configuration values. And when there are a lot of different parameters to configure, the management of all these environment variables becomes unwieldy.

In these cases, many people use an extra level of indirection and put configuration into various configuration files, one for each environment. Then a single environment variable is used to select one of these files. *Profiles* from Spring Boot are an example of this approach. Since these profile configuration files are typically stored within the application itself, which is within the container, it couples the configuration tightly with the application. This often leads to configuration for development and production ending up side by side in the same Docker image, which requires an image rebuild for every change in either environment. We do not recommend this setup (configuration should always be external to the application), but this solution indicates that environment variables are suitable for small to medium sets of configurations only.

The patterns *Configuration Resource*, *Immutable Configuration*, and *Configuration Template* described in the following chapters are good alternatives when more complex configuration needs come up.

Environment variables are universally applicable, and because of that, we can set them at various levels. This option leads to fragmentation of the configuration definitions and makes it hard to track for a given environment variable where it is set. When there is no central place where all environments variables are defined, it is hard to debug configuration issues.

Another disadvantage of environment variables is that they can be set only *before* an application starts, and we cannot change them later. On the one hand, it's a drawback that you can't change configuration "hot" during runtime to tune the application. However, many see this as an advantage, as it promotes *immutability* even to the configuration. Immutability here means you throw away the running application container and start a new copy with a modified configuration, very likely with a smooth Deployment strategy like rolling updates. That way, you are always in a defined and well-known configuration state.

Environment variables are simple to use, but are applicable mainly for simple use cases and have limitations for complex configuration requirements. The next patterns show how to overcome those limitations.

More Information

- EnvVar Configuration Example
- The Twelve-Factor App
- Expose Pod Information to Containers Through Environment Variables
- Define Dependent Environment Variables
- Spring Boot Profiles for Using Sets of Configuration Values

CHAPTER 20

Configuration Resource

Kubernetes provides native configuration resources for regular and confidential data, which allows you to decouple the configuration lifecycle from the application lifecycle. The *Configuration Resource* pattern explains the concepts of ConfigMap and Secret resources and how we can use them, as well as their limitations.

Problem

One significant disadvantage of the *EnvVar Configuration* pattern, discussed in Chapter 19, is that it's suitable for only a handful of variables and simple configurations. Another disadvantage is that because environment variables can be defined in various places, it is often hard to find the definition of a variable. And even if you find it, you can't be entirely sure it won't be overridden in another location. For example, environment variables defined within a OCI image can be replaced during runtime in a Kubernetes Deployment resource.

Often, it is better to keep all the configuration data in a single place and not scattered around in various resource definition files. But it does not make sense to put the content of a whole configuration file into an environment variable. So some extra indirection would allow more flexibility, which is what Kubernetes configuration resources offer.

Solution

Kubernetes provides dedicated configuration Resources that are more flexible than pure environment variables. These are the ConfigMap and Secret objects for general-purpose and sensitive data, respectively.

We can use both in the same way, as both provide storage and management of key-value pairs. When we are describing ConfigMaps, the same can be applied most of the time to Secrets too. Besides the actual data encoding (which is Base64 for Secrets), there is no technical difference for the use of ConfigMaps and Secrets.

Once a ConfigMap is created and holding data, we can use the keys of a ConfigMap in two ways:

- As a reference for *environment variables*, where the key is the name of the environment variable.
- As *files* that are mapped to a volume mounted in a Pod. The key is used as the filename.

The file in a mounted ConfigMap volume is updated when the ConfigMap is updated via the Kubernetes API. So, if an application supports hot reload of configuration files, it can immediately benefit from such an update. However, with ConfigMap entries used as environment variables, updates are not reflected because environment variables can't be changed after a process has been started.

In addition to ConfigMap and Secret, another alternative is to store configuration directly in external volumes that are then mounted.

The following examples concentrate on ConfigMap usage, but they can also be used for Secrets. There is one big difference, though: values for Secrets have to be Base64 encoded.

A ConfigMap resource contains key-value pairs in its `data` section, as shown in Example 20-1.

Example 20-1. ConfigMap resource

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: random-generator-config
data:
  PATTERN: Configuration Resource  ❶
  application.properties: |
    # Random Generator config
    log.file=/tmp/generator.log
    server.port=7070
  EXTRA_OPTIONS: "high-secure,native"
  SEED: "432576345"
```

❶ ConfigMaps can be accessed as environment variables and as a mounted file. We recommend using uppercase keys in the ConfigMap to indicate an EnvVar usage and proper filenames when used as mounted files.

We see here that a ConfigMap can also carry the content of complete configuration files, like the Spring Boot `application.properties` in this example. You can imagine that for a nontrivial use case, this section could get quite large!

Instead of manually creating the full resource descriptor, we can use `kubectl` to create ConfigMaps or Secrets too. For the preceding example, the equivalent `kubectl` command looks like that in Example 20-2.

Example 20-2. Create a ConfigMap from a file

```
kubectl create cm spring-boot-config \
   --from-literal=PATTERN="Configuration Resource" \
   --from-literal=EXTRA_OPTIONS="high-secure,native" \
   --from-literal=SEED="432576345" \
   --from-file=application.properties
```

This ConfigMap then can be read in various places—everywhere environment variables are defined, as demonstrated Example 20-3.

Example 20-3. Environment variable set from ConfigMap

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - env:
    - name: PATTERN
      valueFrom:
        configMapKeyRef:
          name: random-generator-config
          key: PATTERN
....
```

If a ConfigMap has many entries that you want to consume as environment variables, using a certain syntax can save a lot of typing. Rather than specifying each entry individually, as shown in the preceding example in the `env` section, `envFrom` allows you to expose all ConfigMap entries that have a key that also can be used as a valid environment variable. We can prepend this with a prefix, as shown in Example 20-4. Any key that cannot be used as an environment variable is ignored (e.g., `"illeg.al"`). When multiple ConfigMaps are specified with duplicate keys, the last entry in `envFrom` takes precedence. Also, any same-named environment variable set directly with `env` has higher priority.

Example 20-4. Setting all entries of a ConfigMap as environment variables

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
    envFrom:                          ❶
    - configMapRef:
        name: random-generator-config
      prefix: CONFIG_                 ❷
```

❶ Pick up all keys from the ConfigMap `random-generator-config` that can be used as environment variable names.

❷ Prefix all suitable ConfigMap keys with `CONFIG_`. With the ConfigMap defined in Example 20-1, this leads to three exposed environment variables: `CONFIG_PATTERN_NAME`, `CONFIG_EXTRA_OPTIONS`, and `CONFIG_SEED`.

Secrets, as with ConfigMaps, can also be consumed as environment variables, either per entry or for all entries. To access a Secret instead of a ConfigMap, replace `configMapKeyRef` with `secretKeyRef`.

When a ConfigMap is used as a volume, its complete content is projected into this volume, with the keys used as filenames. See Example 20-5.

Example 20-5. Mount a ConfigMap as a volume

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    volumeMounts:
    - name: config-volume
      mountPath: /config
  volumes:
  - name: config-volume
    configMap:  ❶
      name: random-generator-config
```

❶ A ConfigMap-backed volume will contain as many files as entries, with the map's keys as filenames and the map's values as file content.

The configuration in Example 20-1 that is mounted as a volume results in four files in the */config* folder: an *application.properties* file with the content defined in the ConfigMap and the files *PATTERN*, *EXTRA_OPTIONS*, and *SEED*, each with a single line of content.

The mapping of configuration data can be fine-tuned more granularly by adding additional properties to the volume declaration. Rather than mapping all entries as files, you can also individually select every key that should be exposed, the filename, and permissions under which it should be available. Example 20-6 demonstrates how you can granularly select which parts of a ConfigMap are exposed as volumes.

Example 20-6. Expose ConfigMap entries selectively as volumes

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    volumeMounts:
    - name: config-volume
      mountPath: /config
  volumes:
  - name: config-volume
    configMap:
      name: random-generator-config
      items:                                  ❶
      - key: application.properties           ❷
        path: spring/myapp.properties
        mode: 0400
```

❶ List of ConfigMap entries to expose as volumes.

❷ Expose only `application.properties` from the ConfigMap under the path `spring/myapp.properties` with file mode 0400.

As you have seen, changes to a ConfigMap are directly reflected in a projected volume that contains the ConfigMap's content as files. An application can watch those files and immediately pick up any changes. This hot reload is very useful to avoid a redeployment of an application, which can cause an interruption of the service. On the other hand, such live changes are not tracked anywhere and can easily get lost during a restart. These ad hoc chances can cause configuration drift that is hard to detect and analyze. That is one of the reasons many people prefer an *immutable configuration* that stays constant once deployed. We have dedicated a whole pattern in

Chapter 21, "Immutable Configuration", to this paradigm, but there is a cheap way to easily achieve this with ConfigMap and Secrets too.

How Secure Are Secrets?

Secrets hold Base64-encoded data and decode it before passing it to a Pod either as environment variables or mounted volume. This is very often confused as a security feature. Base64 encoding is not an encryption method, and from a security perspective, it is considered the same as plain text. Base64 encoding in Secrets allows you to store binary data, so why are Secrets considered more secure than ConfigMaps? There are a number of other implementation details of Secrets that make them secure. Constant improvements are occurring in this area, but the main implementation details currently are as follows:

- A Secret is distributed only to nodes running Pods that need access to the Secret.
- On the nodes, Secrets are stored in memory in a `tmpfs` and never written to physical storage, and they are removed when the Pod is removed.
- In etcd, the backend storage for the Kubernetes API, Secrets can be stored in encrypted form.

Regardless of all that, there are still ways to get access to Secrets as a root user, or even by creating a Pod and mounting a Secret. You can apply role-based access control (RBAC) to Secrets (as you can do to ConfigMaps or other resources) and allow only certain Pods with predefined service accounts to read them. We explain RBAC in great length in Chapter 26, "Access Control". But users who have the ability to create Pods in a namespace can still escalate their privileges within that namespace by creating Pods. They can run a Pod under a greater-privileged service account and still read Secrets. A user or a controller with Pod-creation access in a namespace can impersonate any service account and access all Secrets and ConfigMaps in that namespace. Thus, additional encryption of sensitive information is often done at the application level too. In Chapter 25, "Secure Configuration", you'll learn several ways to make Secrets more secure, especially in a GitOps context.

Since version 1.21, Kubernetes supports an `immutable` field for ConfigMaps and Secrets that, if set to `true`, prevents the resource from being updated once created. Besides preventing unwanted updates, using immutable ConfigMaps and Secrets considerably improves a cluster's performance as the Kubernetes API server does not need to monitor changes on those immutable objects. Example 20-7 shows how to declare a Secret immutable. The only way to change such a Secret after it has been stored on the cluster is to delete and recreate the updated Secret. Any running Pod referencing this secret needs to be restarted too.

Example 20-7. Immutable Secret

```
apiVersion: v1
kind: Secret
metadata:
  name: random-config
data:
  user: cm9sYW5k
immutable: true  ❶
```

❶ Boolean flag declaring the mutability of the Secret (default is `false`).

Discussion

ConfigMaps and Secrets allow you to store configuration information in dedicated resource objects that are easy to manage with the Kubernetes API. The most significant advantage of using ConfigMaps and Secrets is that they decouple the *definition* of configuration data from its *usage*. This decoupling allows us to manage the objects that use the configuration independently of the configuration definition. Another benefit of ConfigMaps and Secrets is that they are intrinsic features of the platform. No custom construct like that in Chapter 21, "Immutable Configuration", is required.

However, these configuration resources also have their restrictions: with a 1 MB size limit for Secrets, they can't store arbitrarily large data and are not well suited for nonconfiguration application data. You can also store binary data in Secrets, but since they have to be Base64 encoded, you can use only around 700 KB data for it. Real-world Kubernetes clusters also put an individual quota on the number of ConfigMaps that can be used per namespace or project, so ConfigMap is not a golden hammer.

The next two chapters show how to deal with large configuration data by using the *Immutable Configuration* and *Configuration Template* patterns.

More Information

- Configuration Resource Example
- Configure a Pod to Use a ConfigMap
- Secrets
- Encrypting Secret Data at Rest
- Distribute Credentials Securely Using Secrets
- Immutable Secrets
- How to Create Immutable ConfigMaps and Secrets
- Size Limit for a ConfigMap

CHAPTER 21

Immutable Configuration

The *Immutable Configuration* pattern offers two ways to make configuration data immutable so that your application's configuration is always in a well-known and recorded state. With this pattern, we can not only use immutable and versioned configuration data, but also overcome the size limitation of configuration data stored in environment variables or ConfigMaps.

Problem

As you saw in Chapter 19, "EnvVar Configuration", environment variables provide a simple way to configure container-based applications. And although they are easy to use and universally supported, as soon as the number of environment variables exceeds a certain threshold, managing them becomes hard.

This complexity can be handled to some degree by using *Configuration Resources*, as described in Chapter 20, "Configuration Resource", which since Kubernetes 1.21 can be declared as *immutable*. However, ConfigMaps still have a size limitation, so if you work with large configuration data (like precomputed data models in a machine learning context), then ConfigMaps are not suitable even when marked as immutable.

Immutability here means that we can't change the configuration after the application has started, in order to ensure that we always have a well-defined state for our configuration data. In addition, immutable configuration can be put under version control and follow a change control process.

Solution

There are several options to address the concern of configuration immutability. The simplest and preferred option is to use ConfigMaps or Secrets that are marked as immutable in their declaration. You learned about immutable ConfigMaps in Chapter 20. ConfigMaps should be the first choice if your configuration fits into a ConfigMap and is reasonably easy to maintain. In real-world scenarios, however, the amount of configuration data can increase quickly. Although a WildFly application server configuration might still fit in a ConfigMap, it is quite huge. It becomes really ugly when you have to nest XML or YAML within YAML—i.e., when the content of your configuration is also YAML and you embed this as within the ConfigMaps YAML section. Editor support for such use cases is limited, so you have to be very careful about the indentation, and even then, you will probably mess it up more than once (believe us!). Another nightmare is having to maintain tens or hundreds of entries in a single ConfigMap because your application requires many different configuration files. Although this pain can be mitigated to some degree with good tooling, large configuration data sets like pretrained machine learning data models are just impossible with ConfigMap because of the backend size restriction of 1 MB.

To address the concern of complex configuration data, we can put all environment-specific configuration data into a single, passive data image that we can distribute as a regular container image. During runtime, the application and the data image are linked together so that the application can extract the configuration from the data image. With this approach, it is easy to craft different configuration data images for various environments. These images then combine all configuration information for specific environments and can be versioned like any other container image.

Creating such a data image is trivial, as it is a simple container image that contains only data. The challenge is the linking step during startup. We can use various approaches, depending on the platform.

Docker Volumes

Before looking at Kubernetes, let's go one step back and consider the vanilla Docker case. In Docker, it is possible for a container to expose a *volume* with data from the container. With a `VOLUME` directive in a Dockerfile, you can specify a directory that can be shared later. During startup, the content of this directory within the container is copied over to this shared directory. As shown in Figure 21-1, this volume linking is an excellent way to share configuration information from a dedicated configuration container with another application container.

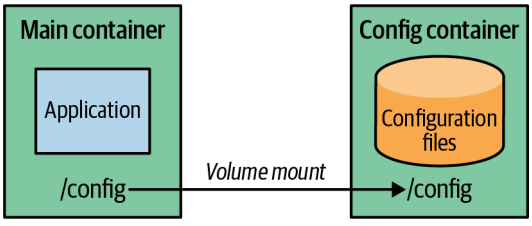


Figure 21-1. Immutable configuration with Docker volume

Let's have a look at an example. For the development environment, we create a Docker image that holds the developer configuration and creates a volume with mount point */config*. We can create such an image with `Dockerfile-config`, as in Example 21-1.

Example 21-1. Dockerfile for a configuration image

```
FROM scratch
ADD app-dev.properties /config/app.properties  ❶
VOLUME /config                                 ❷
```

❶ Add specified property.

❷ Create volume and copy property into it.

We now create the image itself and the Docker container with the Docker CLI in Example 21-2.

Example 21-2. Building the configuration Docker image

```
docker build -t k8spatterns/config-dev-image:1.0.1 -f Dockerfile-config .
docker create --name config-dev k8spatterns/config-dev-image:1.0.1 .
```

The final step is to start the application container and connect it to this configuration container (Example 21-3).

Example 21-3. Start application container with config container linked

```
docker run --volumes-from config-dev k8spatterns/welcome-servlet:1.0
```

The application image expects its configuration files to be within a */config* directory, the volume exposed by the configuration container. When you move this application from the development environment to the production environment, all you have to do is change the startup command. There is no need to alter the application image itself. Instead, you simply volume-link the application container with the production configuration container, as seen in Example 21-4.

Example 21-4. Use different configuration for production environment

```
docker build -t k8spatterns/config-prod-image:1.0.1 -f Dockerfile-config .
docker create --name config-prod k8spatterns/config-prod-image:1.0.1 .
docker run --volumes-from config-prod k8spatterns/welcome-servlet:1.0
```

Kubernetes Init Containers

In Kubernetes, volume sharing within a Pod is perfectly suited for this kind of linking of configuration and application containers. However, if we want to transfer this technique of Docker volume linking to the Kubernetes world, we will find that there is currently no support for container volumes in Kubernetes. Considering the age of the discussion and the complexity of implementing this feature versus its limited benefits, it's likely that container volumes will not arrive anytime soon.

So containers can share (external) volumes, but they cannot yet directly share directories located within the containers. To use immutable configuration containers in Kubernetes, we can use the *Init Containers* pattern from Chapter 15 that can initialize an empty shared volume during startup.

In the Docker example, we base the configuration Docker image on `scratch`, an empty Docker image with no operating system files. We don't need anything else because we only want the configuration data shared via Docker volumes. But for Kubernetes init containers, we need help from the base image to copy over the configuration data to a shared Pod volume. A good choice for this is `busybox`, which is still small but allows us to use a plain Unix `cp` command for this task.

So how does the initialization of shared volumes with configuration work under the hood? Let's have a look at an example. First, we need to create a configuration image again with a Dockerfile, as in Example 21-5.

Example 21-5. Development configuration image

```
FROM busybox
ADD dev.properties /config-src/demo.properties
ENTRYPOINT [ "sh", "-c", "cp /config-src/* $1", "--" ]  ❶
```

❶ Using a shell here in order to resolve wildcards.

The only difference from the vanilla Docker case in Example 21-1 is that we have a different base image and we add an `ENTRYPOINT` that copies the properties file to the directory given as an argument when the container image starts. This image can now be referenced in an init container within a Deployment's `.template.spec` (see Example 21-6).

Example 21-6. Deployment that copies configuration to destination in init container

```
initContainers:
- image: k8spatterns/config-dev:1
  name: init
  args:
  - "/config"
  volumeMounts:
  - mountPath: "/config"
    name: config-directory
containers:
- image: k8spatterns/demo:1
  name: demo
  ports:
  - containerPort: 8080
    name: http
    protocol: TCP
  volumeMounts:
  - mountPath: "/var/config"
    name: config-directory
volumes:
  - name: config-directory
    emptyDir: {}
```

The Deployment's Pod template specification contains a single volume and two containers:

- The volume `config-directory` is of the type `emptyDir`, so it's created as an empty directory on the node hosting this Pod.
- The init container Kubernetes calls during startup is built from the image we just created, and we set a single argument, `/config`, used by the image's `ENTRYPOINT`. This argument instructs the init container to copy its content to the specified directory. The directory */config* is mounted from the volume `config-directory`.
- The application container mounts the volume `config-directory` to access the configuration that was copied over by the init container.

Figure 21-2 illustrates how the application container accesses the configuration data created by an init container over a shared volume.

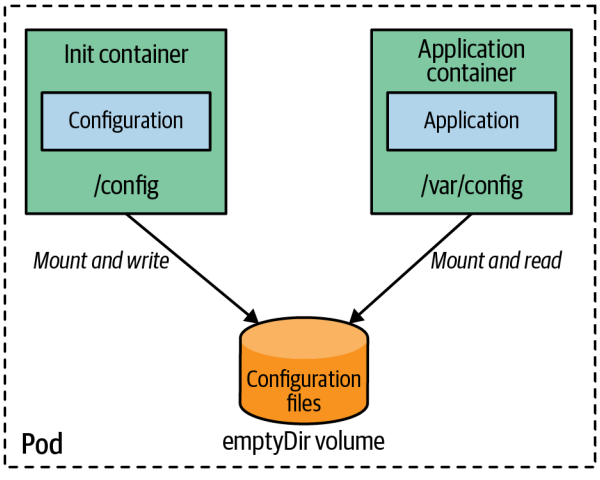


Figure 21-2. Immutable configuration with an init container

Now to change the configuration from the development to the production environment, all we need to do is exchange the image of the init container. We can do this either by changing the YAML definition or by updating with `kubectl`. However, it is not ideal to have to edit the resource descriptor for each environment. If you are on Red Hat OpenShift, an enterprise distribution of Kubernetes, *OpenShift Templates* can help address this. OpenShift Templates can create different resource descriptors for the different environments from a single template.

OpenShift Templates

OpenShift Templates are regular resource descriptors that are parameterized. As seen in Example 21-7, we can easily use the configuration image as a parameter.

Example 21-7. OpenShift Template for parameterizing config image

```
apiVersion: v1
kind: Template
metadata:
  name: demo
parameters:
  - name: CONFIG_IMAGE                          ❶
    description: Name of configuration image
    value: k8spatterns/config-dev:1
objects:
- apiVersion: apps/v1
  kind: Deployment
    // ....
    spec:
```

```
template:
    metadata:
      // ....
      spec:
        initContainers:
        - name: init
          image: ${CONFIG_IMAGE}          ❷
          args: [ "/config" ]
          volumeMounts:
          - mountPath: /config
            name: config-directory
        containers:
        - image: k8spatterns/demo:1
          // ...
          volumeMounts:
          - mountPath: /var/config
            name: config-directory
      volumes:
      - name: config-directory
        emptyDir: {}
```

❶ Template parameter `CONFIG_IMAGE` declaration.

❷ Use of the template parameter.

We show here only a fragment of the full descriptor, but you can quickly recognize the parameter `CONFIG_IMAGE` we reference in the init container declaration. If we create this template on an OpenShift cluster, we can instantiate it by calling `oc`, as in Example 21-8.

Example 21-8. Applying OpenShift template to create new application

```
oc new-app demo -p CONFIG_IMAGE=k8spatterns/config-prod:1
```

Detailed instructions for running this example, as well as the full Deployment descriptors, can be found as usual in our example Git repository.

Discussion

Using data containers for the *Immutable Configuration* pattern is admittedly a bit involved. Use these only if immutable ConfigMaps and Secret are not suitable for your use case.

Data containers have some unique advantages:

- Environment-specific configuration is sealed within a container. Therefore, it can be versioned like any other container image.

- Configuration created this way can be distributed over a container registry. The configuration can be examined even without accessing the cluster.
- The configuration is immutable, as is the container image holding the configuration: a change in the configuration requires a version update and a new container image.
- Configuration data images are useful when the configuration data is too complex to put into environment variables or ConfigMaps, since it can hold arbitrarily large configuration data.

As expected, the *Immutable Configuration* pattern also has certain drawbacks:

- It has higher complexity, because extra container images need to be built and distributed via registries.
- It does not address any of the security concerns around sensitive configuration data.
- Since no image volume support is actually available for Kubernetes workloads, the technique described here is still limited for use cases where the overhead of copying over data from init containers to a local volume is acceptable. We hope that eventually mounting container images directly as volumes will be possible in the future, but as of 2023, only experimental CSI support is available.
- Extra init container processing is required in the Kubernetes case, and hence we need to manage different Deployment objects for different environments.

All in all, you should carefully evaluate whether such an involved approach is really required.

Another approach for dealing with large configuration files that differ only slightly from environment to environment is described with the *Configuration Template* pattern, the topic of the next chapter.

More Information

- Immutable Configuration Example
- How to Mimic `--volumes-from` in Kubernetes
- Immutable ConfigMaps
- Feature Request: Image Volumes and Container Volumes
- docker-flexvol: A Kubernetes Driver That Supports Docker Volumes
- Red Hat OpenShift: Using Templates
- Kubernetes CSI Driver for Mounting Images

CHAPTER 22

Configuration Template

The *Configuration Template* pattern enables you to create and process large and complex configurations during application startup. The generated configuration is specific to the target runtime environment as reflected by the parameters used in processing the configuration template.

Problem

In Chapter 20, "Configuration Resource", you saw how to use the Kubernetes native resource objects ConfigMap and Secret to configure applications. But sometimes configuration files can get large and complex. Putting the configuration files directly into ConfigMaps can be problematic since they have to be correctly embedded in the resource definition. We need to be careful and avoid using special characters like quotes and breaking the Kubernetes resource syntax. The size of configurations is another consideration, as there is a limit on the sum of all values of ConfigMaps or Secrets, which is 1 MB (a limit imposed by the underlying backend store etcd).

Large configuration files typically differ only slightly for the different execution environments. This similarity leads to a lot of duplication and redundancy in the ConfigMaps because each environment has mostly the same data. The *Configuration Template* pattern we explore in this chapter addresses these specific use-case concerns.

Solution

To reduce duplication, it makes sense to store only the *differing* configuration values like database connection parameters in a ConfigMap or even directly in environment variables. During startup of the container, these values are processed with configuration templates to create the full configuration file (like a WildFly *standalone.xml*).

There are many tools like *Tiller* (Ruby) or *Gomplate* (Go) for processing templates during application initialization. Figure 22-1 is a configuration template example filled with data coming from environment variables or a mounted volume, possibly backed by a ConfigMap.

Before the application is started, the fully processed configuration file is put into a location where it can be directly used like any other configuration file.

There are two techniques for how such live processing can happen during runtime:

- We can add the template processor as part of the `ENTRYPOINT` to a Dockerfile so the template processing becomes directly part of the container image. The entry point here is typically a script that first performs the template processing and then starts the application. The parameters for the template come from environment variables.
- With Kubernetes, a better way to perform initialization is with an init container of a Pod in which the template processor runs and creates the configuration for the application containers in the Pod. The *Init Container* pattern is described in detail in Chapter 15.

For Kubernetes, the init container approach is the most appealing because we can use ConfigMaps directly for the template parameters. This technique is illustrated in Figure 22-1.

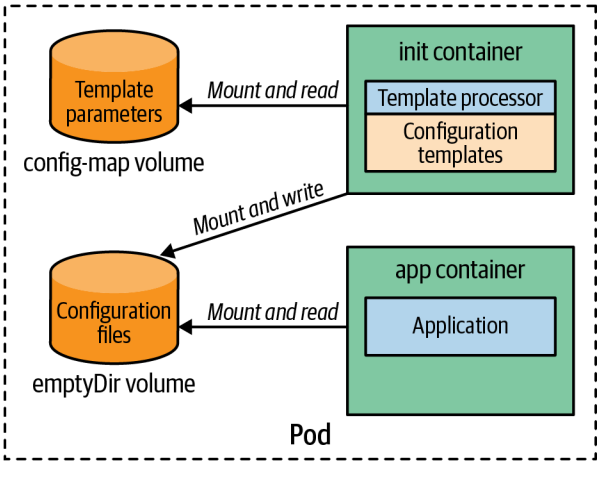


Figure 22-1. Configuration template

The application's Pod definition consists of at least two containers: one init container for the template processing and one for the application container. The init container contains not only the template processor but also the configuration templates themselves. In addition to the containers, this Pod also defines two volumes: one volume for the template parameters, backed by a ConfigMap, and an `emptyDir` volume used to share the processed templates between the init container and the application container.

With this setup, the following steps are performed during startup of this Pod:

1. The init container is started, and it runs the template processor. The processor takes the templates from its image, and the template parameters from the mounted ConfigMap volume, and stores the result in the `emptyDir` volume.
2. After the init container has finished, the application container starts up and loads the configuration files from the `emptyDir` volume.

The following example uses an init container for managing a full set of WildFly configuration files for two environments: a development environment and a production environment. Both are very similar to each other and differ only slightly. In fact, in our example, they differ only in the way logging is performed: each log line is prefixed with `DEVELOPMENT:` or `PRODUCTION:`, respectively.

You can find the full example along with complete installation instructions in the book's example GitHub repo. (We show only the main concept here; for the technical details, refer to the source repo.)

The log pattern in Example 22-1 is stored in *standalone.xml*, which we parameterize by using the Go template syntax.

Example 22-1. Log configuration template

```
....
<formatter name="COLOR-PATTERN">
  <pattern-formatter pattern="{{(datasource "config").logFormat}}"/>
</formatter>
....
```

Here we use Gomplate as a template processor, which uses the notion of a *data source* for referencing the template parameters to be filled in. In our case, this data source comes from a ConfigMap-backed volume mounted to an init container. Here, the ConfigMap contains a single entry with the key `logFormat`, from where the actual format is extracted.

With this template in place, we can now create the Docker image for the init container. The Dockerfile for the image *k8spatterns/example-configuration-template-init* is very simple (Example 22-2).

Example 22-2. Simple Dockerfile for template image

```
FROM k8spatterns/gomplate
COPY in /in
```

The base image *k8spatterns/gomplate* contains the template processor and an entrypoint script that uses the following directories by default:

- */in* holds the WildFly configuration templates, including the parameterized *standalone.xml*. These are added directly to the image.
- */params* is used to look up the Gomplate data sources, which are YAML files. This directory is mounted from a ConfigMap-backed Pod volume.
- */out* is the directory into which the processed files are stored. This directory is mounted in the WildFly application container and used for the configuration.

The second ingredient of our example is the ConfigMap holding the parameters. In Example 22-3, we just use a simple file with key-value pairs.

Example 22-3. Create ConfigMap with values to fill into the configuration template

```
kubectl create configmap wildfly-cm \
       --from-literal='config.yml=logFormat: "DEVELOPMENT: %-5p %s%e%n'
```

Finally, we need the Deployment resource for the WildFly server (Example 22-4).

Example 22-4. Deployment with template processor as init container

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    example: cm-template
  name: wildfly-cm-template
spec:
  replicas: 1
  template:
    metadata:
      labels:
        example: cm-template
    spec:
      initContainers:
      - image: k8spatterns/example-config-cm-template-init ❶
```

```
    name: init
    volumeMounts:
    - mountPath: "/params"                          ❷
      name: wildfly-parameters
    - mountPath: "/out"                             ❸
      name: wildfly-config
  containers:
  - image: jboss/wildfly:10.1.0.Final
    name: server
    command:
    - "/opt/jboss/wildfly/bin/standalone.sh"
    - "-Djboss.server.config.dir=/config"
    volumeMounts:
    - mountPath: "/config"                          ❹
      name: wildfly-config
  volumes:                                          ❺
  - name: wildfly-parameters
    configMap:
      name: wildfly-cm
  - name: wildfly-config
    emptyDir: {}
```

❶ Image holding the configuration templates that has been created from Example 22-2.

❷ Parameters are mounted from a volume `wildfly-parameters` declared in ❺.

❸ The target directory for writing out processed templates. This is mounted from an empty volume.

❹ The directory holding the generated full configuration files is mounted as `/config`.

❺ Volume declaration for the parameters' ConfigMap and the empty directory used for sharing the processed configuration.

This declaration is quite a mouthful, so let's drill down: the Deployment specification contains a Pod with our init container, the application container, and two internal Pod volumes:

- The first volume, `wildfly-parameters`, references the ConfigMap `wildfly-cm` with the parameter values that we created in Example 22-3.
- The other volume is an empty directory initially and is shared between the init container and the WildFly container.

If you start this Deployment, the following will happen:

- An init container is created, and its command is executed. This container takes the *config.yml* from the ConfigMap volume, fills in the templates from the */in* directory in an init container, and stores the processed files in the */out* directory. The */out* directory is where the volume `wildfly-config` is mounted.
- After the init container is done, a WildFly server starts with an option so that it looks up the complete configuration from the */config* directory. Again, */config* is the shared volume `wildfly-config` containing the processed template files.

It is important to note that we do *not* have to change these Deployment resource descriptors when going from the development to the production environment. Only the ConfigMap with the template parameters is different.

With this technique, it is easy to create a DRY configuration without copying and maintaining duplicated large configuration files.[1] For example, when the WildFly configuration changes for all environments, only a single template file in the init container needs to be updated. This approach has, of course, significant advantages on maintenance as there is no danger of configuration drift.

When working with Pods and volumes, as in this pattern, it is not obvious how to debug if things don't work as expected. So if you want to examine the processed templates, check out the directory */var/lib/kubelet/pods/{podid}/volumes/kubernetes.io~empty-dir/* on the node, as it contains the content of an `emptyDir` volume. Alternatively, just `kubectl exec` into the Pod when it is running, and examine the mounted directory (*/config* in our example) for any created files.

Discussion

The *Configuration Template* pattern builds on top of the *Configuration Resource* pattern and is especially suited when we need to operate applications in different environments with similar complex configurations. However, the setup with configuration templates is more complicated and has more moving parts that can go wrong. Use it only if your application requires huge configuration data. Such applications often require a considerable amount of configuration data from which only a small fraction is dependent on the environment. Even when copying over the whole configuration directly into the environment-specific ConfigMap works initially, it puts a

1 DRY is an acronym for "Don't Repeat Yourself."

burden on the maintenance of that configuration because it is doomed to diverge over time. For such a situation, this template approach is perfect.

If you are running on top of Red Hat OpenShift, an enterprise Kubernetes distribution, you have an alternative by using OpenShift templates for parameterizing resource descriptors. This approach does not solve the challenge of large configuration sets but is still very helpful for applying the same deployment resources to slightly varying environments.

More Information

- Configuration Template Example
- Tiller Template Engine
- Gomplate
- Go Template Syntax

PART V

Security Patterns

Security is a broad topic that has implications for all stages of the software development lifecycle, from development practices, to image scanning at build time, to cluster hardening through admission controllers at deployment time, to threat detection at runtime. Security also touches all the layers of the software stack, from cloud infrastructure security, to cluster security, to container security, to code security, also known as the 4C's of cloud native security. In this section, we focus on the intersection of an application with Kubernetes from the security point of view, as demonstrated in Figure V-1.

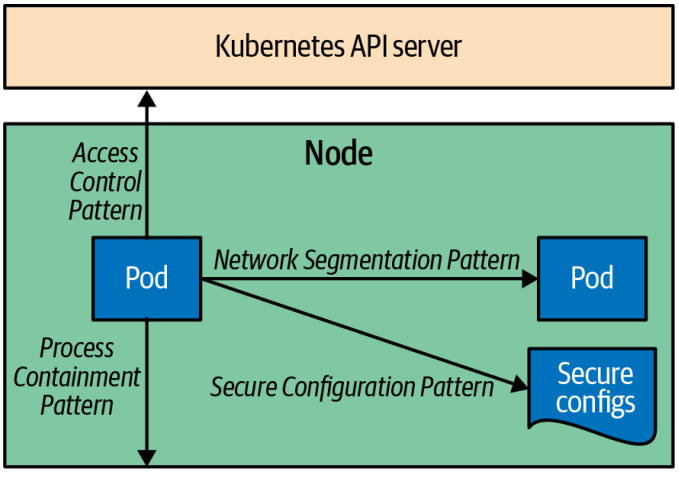


Figure V-1. Security patterns

We start by describing the *Process Containment* pattern to contain and limit the actions an application is allowed to perform on the node it is running on. Then we explore the techniques to limit what other Pods a Pod can talk to by doing *Network Segmentation*. In the *Secure Configuration* pattern, we discuss how an application within a Pod can access and use configurations in a secure way. And finally, we describe the *Access Control* pattern—how an application can authenticate and talk to the Kubernetes API server and interact with it in more advanced scenarios. These give you an overview of the main security dimensions of an application running on Kubernetes, and we discuss the resulting patterns in the following chapters:

- Chapter 23, "Process Containment", describes the ways to contain a process to the least privileges it is entitled to.
- Chapter 24, "Network Segmentation", applies network controls to limit the traffic a Pod is allowed to participate in.
- Chapter 25, "Secure Configuration", helps keep and use sensitive configuration data securely and safely.
- Chapter 26, "Access Control", allows users and application workloads to authenticate and interact with the Kubernetes API server.

CHAPTER 23

Process Containment

This chapter describes techniques that help apply the principle of least privilege to constrain a process to the minimum privileges it needs to run. The *Process Containment* pattern helps make applications more secure by limiting the attack surface and creating a line of defense. It also prevents any rogue process from running out of its designated boundary.

Problem

One of the primary attack vectors for Kubernetes workloads is through the application code. Many techniques can help improve code security. For example, static code analysis tools can check the source code for security flaws. Dynamic scanning tools can simulate malicious attackers with the goal of breaking into the system through well-known service attacks such as SQL injection (SQLi), cross-site request forgery (CSRF), and cross-site scripting (XSS). Then there are tools for regularly scanning the application's dependencies for security vulnerabilities. As part of the image build process, the containers are scanned for known vulnerabilities. This is usually done by checking the base image and all its packages against a database that tracks vulnerable packages. These are only a few of the steps involved in creating secure applications and protecting against malicious actors, compromised users, unsafe container images, or dependencies with vulnerabilities.

Regardless of how many checks are in place, new code and new dependencies can introduce new vulnerabilities, and there is no way to guarantee the complete absence of risks. Without runtime process-level security controls in place, a malicious actor can breach the application code and attempt to take control of the host or the entire Kubernetes cluster. The mechanisms we will explore in this chapter demonstrate how to limit a container only to the permissions it needs to run and apply the least-privilege principle. This way, Kubernetes configurations act as another line of

defense, containing any rogue process and preventing it from running outside its designated boundary.

Solution

Typically, a container runtime such as Docker assigns the default runtime permissions a container will have. When the container is managed by Kubernetes, the security configurations that will be applied to a container are controlled by Kubernetes and exposed to the user through the security context configurations of the Pod and the container specs. The Pod-level configurations apply to the Pod's volumes and all containers in the Pod, whereas container-level configurations apply to a single container. When the same configurations are set at both Pod and container levels, the values in the container spec take precedence.

As a developer creating cloud native applications, you typically should not need to deal with many fine-grained security configurations but instead have them validated and enforced as global policy. Fine-grained tuning is usually required when creating specialized infrastructure containers such as build systems and other plugins that need broader access to the underlying nodes. Therefore, we will review only the common security configurations that would be useful for running typical cloud native applications on Kubernetes.

Running Containers with a Non-Root User

Container images have a user, and can optionally have a group, to run the container process. These users and groups are used to control access to files, directories, and volume mounts. With some other containers, no user is created and the container image runs as root by default. In others, a user is created in the container image, but it is not set as the default user to run. These situations can be rectified by overriding the user at runtime using `securityContext`, as shown in Example 23-1.

Example 23-1. Setting a user and group for the containers of a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  securityContext:
    runAsUser: 1000   ❶
    runAsGroup: 2000  ❷
  containers:
  - name: app
    image: k8spatterns/random-generator:1.0
```

❶ Indicates the UID to run the container process.

❷ Specifies the GID to run the container process.

The configuration forces any container in the Pod to run with user ID 1000 and group ID 2000. This is useful when you want to swap the user that is specified in the container image. But there is also a danger in setting these values and making runtime decisions about which user to run the image. Often the user is set in conjunction with the directory structure containing files that have the same ownership IDs specified in the container image. To avoid having runtime failures due to lack of permissions, you should check the container image file and run the container with the user ID and group ID defined. This is one way to prevent a container from running as root, and matching it to the expected user in the image.

Instead of specifying a user ID to ensure that a container is not running as root, a less intrusive way is to set the `.spec.securityContext.runAsNonRoot` flag to `true`. When set, the Kubelet will validate at runtime and prevent any container from starting with a root user—that is, a user with UID 0. This latter mechanism doesn't change the user, but only ensures that a container is running as a non-root user. If you need to run as root to access files or volumes in the container, you can limit the exposure to root by running an init container that can run as root for a short time, and you can change the file access modes, before applications containers start up as non-root.

A container may not run as root, but it is possible to obtain root-like capabilities through privilege escalation. This is most similar to using the `sudo` command on Linux and executing commands with the root privileges. The way to prevent this in containers is by setting `.spec.containers[].securityContext.allowPrivilegeEscalation` to `false`. This configuration typically has no side effects because if an application is designed to run as non-root, it should not require privilege escalation during its lifetime.

The root user has special permissions and privileges in a Linux system, and preventing the root user from owning container processes, escalating privileges to become root, or limiting the root user lifetime with init containers will help prevent container breakout attacks and ensure adherence to the general security practices.

Restricting Container Capabilities

In essence, a container is a process that runs on a node, and it can have the same privileges a process can have. If the process requires a kernel-level call, it needs to have the privileges to do so in order to succeed. You can do this either by running the container as root, which grants all privileges to the container, or by assigning specific capabilities required for the application to function.

Containers with the `.spec.containers[].securityContext.privileged` flag set are essentially equivalent to root on the host and bypass the kernel permission checks. From a security point of view, this option bundles your container with the host system rather than isolating it. Therefore, this flag is typically set for containers with administrative capabilities—for example, to manipulate the network stack or access hardware devices. It is a better approach to avoid using privileged containers altogether and give specific kernel capabilities to containers that need them. In Linux, the privileges traditionally associated with the root user are divided into distinct capabilities, which can be independently enabled and disabled. Finding out what capabilities your container has is not straightforward. You can employ a whitelisting approach and start your container without any capabilities and gradually add capabilities when needed for every use case within the container. You might need the help of your security team, or you can use tools such as SELinux in permissive mode and check the audit logs of your application to discover what capabilities it needs, if any.

To make containers more secure, you should provide them with the least amount of privileges needed to run. The container runtime assigns a set of default privileges (capabilities) to the container. Contrary to what you might expect, if the `.spec.containers[].securityContext.capabilities` section is left empty, the default set of capabilities defined by the container runtime are far more generous than most processes need, opening them up to exploits. A good security practice for locking down the container attack surface is to drop all privileges and add only the ones you need, as shown in Example 23-2.

Example 23-2. Setting Pod permissions

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: app
    image: docker.io/centos/httpd
    securityContext:
      capabilities:
        drop: [ 'ALL' ]              ❶
        add: ['NET_BIND_SERVICE']    ❷
```

❶ Removes all default capabilities assigned to the container by the container runtime.

❷ Adds back only the `NET_BIND_SERVICE` capability.

In this example, we drop all the capabilities and add back only the `NET_BIND_SERVICE` capability, which allows binding to privileged ports with numbers lower than 1024. An alternative approach for addressing this scenario is to replace the container with one that binds to an unprivileged port number.

A Pod is more likely to be compromised if its Security Context is not configured or is too permissive. Limiting the capabilities of containers to the very minimum acts as an additional line of defense against known attacks. A malicious actor who breaches an application would have a harder time taking control of the host when the container process is not privileged or when the capabilities are severely limited.

Avoiding a Mutable Container Filesystem

In general, containerized applications should not be able to write to the container filesystem because containers are ephemeral and any state will be lost upon restart. As discussed in Chapter 11, "Stateless Service", state should be written to external persistence methods such as database or filesystems. Logs should be written to stdout or forward to a remote log collector. Such an application can limit the attack surface of the container further by having a read-only container filesystem. A read-only filesystem will prevent any rogue user from tampering with the application configuration or installing additional executables on the disk that can be used for further exploits. The way to do that is to set `.spec.containers[].securityContext.readOnlyRootFile` to `true`, which will mount the container's root filesystem as read-only. This prevents any writes to the container's root filesystem at runtime and enforces the principle of immutable infrastructure.

The complete list of values in the `securityContext` field has many more items and can vary between Pod and container configurations. It is beyond the scope of this book to cover all security configurations. The two other must-check security context options are `seccompProfile` and `seLinuxOptions`. The first one is a Linux kernel feature that can be used to limit the process running in a container to call only a subset of the available system calls. These system calls are configured as profiles and applied to a container or Pod.

The latter option, `seLinuxOptions`, can assign custom SELinux labels to all containers within the Pod as well as the volume. SELinux uses policies to define which processes can access other labeled objects in the system. In Kubernetes, it is typically used to label the container image in such a way as to restrict the process to access only files within the image. When SELinux is supported on the host environment, it can be strictly enforced to deny access, or it can be configured in permissive mode to log access violations.

Configuring these fields for every Pod or container causes them to be prone to human errors. Unfortunately, setting them is usually the responsibility of the workload authors who are not typically the security subject-matter experts in the organization. That is why there are also cluster-level, policy-driven means defined by cluster administrators for ensuring all Pods in a namespace meet the minimum security standards. Let's briefly review that next.

Enforcing Security Policies

So far, we've explored setting security parameters of the container runtime using the `securityContext` definition as part of the Pod and container specifications. These specifications are created individually per Pod and usually indirectly through higher abstractions such as Deployments, Jobs, and CronJobs. But how can a cluster administrator or a security expert ensure that a collection of Pods follows certain security standards? The answer is in the Kubernetes Pod Security Standards (PSS) and Pod Security Admission (PSA) controller. PSS defines a common understanding and consistent language around security policies, and PSA helps enforce them. This way, the policies are independent of the underlying enforcement mechanism and can be applied through PSS or other third-party tools. These policies are grouped in three security profiles that are cumulative, from highly permissive to highly restrictive, as follows:

Privileged
: This is an unrestricted profile with the widest possible level of permissions. It is purposely left open and offers allow-by-default mechanisms for trusted users and infrastructure workloads.

Baseline
: This profile is for common noncritical application workloads. It has a minimally restrictive policy and provides a balance between ease of adoption and prevention from known privilege escalations. For example, it won't allow privileged containers, certain security capabilities, and even other configurations outside of the `securityContext` field.

Restricted
: This is the most restrictive profile that follows the latest security-hardening best practices at the expense of adoption. It is meant for security-critical applications, as well as lower-trust users. On top of the Baseline profile, it puts restrictions on the fields we reviewed earlier, such as `allowPrivilegeEscalation`, `runAsNonRoot`, `runAsUser`, and other container configurations.

PodSecurityPolicy was the legacy security-policy-enforcement mechanism that was replaced with PSA in Kubernetes v1.25. Going forward, you can use a third-party admission plugin or the built-in PSA controller to enforce the security standards for

each namespace. The security standards are applied to a Kubernetes namespace using labels that define the standard level as described earlier and one or more actions to take when a potential violation is detected. Following are the actions you can take:

Warn
: The policy violations are allowed with a user-facing warning.

Audit
: The policy violations are allowed with an auditing log entry recorded.

Enforce
: Any policy violations will cause the Pod to be rejected.

With these options defined, Example 23-3 creates a namespace that rejects any Pods that don't satisfy the *baseline* standard, and also generates a warning for Pods that don't meet the *restricted* standards requirements.

Example 23-3. Set security standards for a namespace

```
apiVersion: v1
kind: Namespace
metadata:
  name: baseline-namespace
  labels:
    pod-security.kubernetes.io/enforce: baseline                ❶
    pod-security.kubernetes.io/enforce-version: v1.25           ❷
    pod-security.kubernetes.io/warn: restricted                 ❸
    pod-security.kubernetes.io/warn-version: v1.25
```

❶ Label hinting to the PSA controller to reject Pods that violate the *baseline* standard.

❷ Version of the security-standard requirements to use (optional).

❸ Label hinting to the PSA controller to warn about Pods that violate the *restricted* standard.

This example creates a new namespace and configures the security standards to apply to all Pods that will be created in this namespace. It is also possible to update the configuration of a namespace or apply the policy to one or all existing namespaces. For details on how to do this in the least distributive way, check out "More Information" on page 219.

Discussion

One of the common security challenges with Kubernetes is running legacy applications that are not implemented or containerized with Kubernetes security controls in mind. Running a privileged container can be a challenge on Kubernetes distributions or environments with strict security policies. Understanding how Kubernetes does process containment at runtime and configures security boundaries, as shown in Figure 23-1, will help you create applications that run on Kubernetes more securely. It is important to realize that a container is not only a packaging format and not only a resource isolation mechanism, but when configured properly, it is also a security fence.

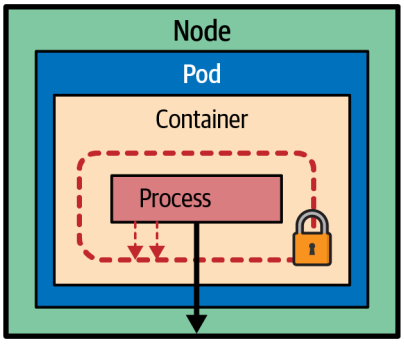


Figure 23-1. Process Containment pattern

The tendency of shifting left the security considerations and testing practices, including deploying into Kubernetes with the production security standards, is getting more popular. Such practices help identify and tackle security issues earlier in the development cycle and prevent last-minute surprises.

Shifting left is all about doing things earlier rather than later. It's about going leftward on the time ray that describes a development and deployment process. In our context, shift left implies that the developer already thinks about operational security when developing the application. See more details about the Shift Left model on Devopedia.

In this chapter, we hope that we have given you enough food for thought when creating secure cloud native applications. The guidelines in this chapter will help you design and implement applications that don't write to the local filesystem or require root privileges (for example, when containerizing applications, to ensure the container has a designated non-root user) and configure the security context. We hope that you understand exactly what your application needs and give it only the

minimum permissions. We also aimed to help you build boundaries between the workloads and the host, to reduce container privileges and configuring the runtime environment to limit resource utilization in the event of a breach. In this endeavor, the *Process Containment* pattern ensures "what happens in a container stays in a container," including any security breaches.

More Information

- Process Containment Example
- Configure a Security Context for a Pod or Container
- Pod Security Admission
- Pod Security Standards
- Enforce Pod Security Standards with Namespace Labels
- Admission Controllers Reference: PodSecurity
- Linux Capabilities
- Introduction to Security Contexts and SCCs
- 10 Kubernetes Security Context Settings You Should Understand
- Security Risk Analysis Tool for Kubernetes Resources

CHAPTER 24

Network Segmentation

Kubernetes is a great platform for running distributed applications that communicate with one another over the network. By default, the network space within Kubernetes is flat, which means that every Pod can connect to every other Pod in the cluster. In this chapter, we will explore how to structure this network space for improved security and a lightweight multitenancy model.

Problem

Namespaces are a crucial part of Kubernetes, allowing you to group your workloads together. However, they only provide a grouping concept, imposing isolation constraints on the containers associated with specific namespaces. In Kubernetes, every Pod can talk to every other Pod, regardless of their namespace. This default behavior has security implications, particularly when multiple independent applications operated by different teams run in the same cluster.

Restricting network access to and from Pods is essential for enhancing the security of your application because not everyone may be allowed to access your application via an ingress. Outgoing egress network traffic for Pods should also be limited to what is necessary to minimize the blast radius of a security breach.

Network segmentation plays a vital role in multitenancy setups where multiple parties share the same cluster. For example, the following sidebar addresses some of the challenges of multitenancy on Kubernetes, such as creating network boundaries for applications.

Multitenancy with Kubernetes

Multitenancy refers to platform's ability to support multiple isolated user groups, also known as *tenants*. Kubernetes does not provide extensive support for multitenancy out of the box, and the concept itself can be complex and difficult to define. The Kubernetes documentation on Multitenancy covers various aspects and the support within the platform, including namespaces and access control (Chapter 26), quotas to prevent noisy neighbor issues, storage and network isolation, and handling of shared resources like cluster-wide DNS or CustomResourceDefinitions. In this chapter, we will focus on the network isolation aspects, which offer a softer approach to multitenancy. Stricter isolation requirements may require a more encapsulated approach, such as a virtual control plane per tenant, as provided by vcluster.

In the past, shaping the network topology was primarily the responsibility of administrators who managed firewalls and iptable rules. The challenge with this model is that administrators need to understand the networking requirements of the applications. In addition, the network graph can get very complex in a microservices world with many dependencies, requiring deep domain knowledge about the application. In this sense, the developer must communicate and sync information about dependencies with administrators. A DevOps setup can help, but the definition of network topologies is still far away from the application itself and can change dynamically over time.

So, what does defining and establishing a network segmentation look like in a Kubernetes world?

Solution

The good news is that Kubernetes shifts left these networking tasks so that developers using Kubernetes fully define their applications' networking topology. You have already seen this process model described briefly in Chapter 23, when we discussed the *Process Containment* pattern.

The essence of this *Network Segmentation* pattern is how we, as developers, can define the network segmentation for our applications by creating "application firewalls."

There are two ways to implement this feature that are complementary and can be applied together. The first is through the use of core Kubernetes features that operate on the L3/L4 networking layers.[1] By defining resources of the type NetworkPolicy, developers can create ingress and egress firewall rules for workload Pods.

The other method involves the use of a service mesh and targets the L7 protocol layer, specifically HTTP-based communication. This allows for filtering based on HTTP verbs and other L7 protocol parameters. We will explore Istio's AuthorizationPolicy later in this chapter.

To start, let's focus on how to use NetworkPolicies to define the network boundaries for your application.

Network Policies

NetworkPolicy is a Kubernetes resource type that allows users to define rules for inbound and outbound network connections for Pods. These rules act like a custom firewall and determine which Pods can be accessed and which destinations they can connect to. The user-defined rules are picked up by the Container Network Interface (CNI) add-on used by Kubernetes for its internal networking. However, not all CNI plugins support NetworkPolicies; for example, the popular Flannel CNI plugin does not support it, but many others, like Calico, do. All hosted Kubernetes cloud offerings support NetworkPolicy (either directly or by configuring an add-on) as well as other distributions like Minikube.

The NetworkPolicy definition consists of a selector for Pods and lists of inbound (ingress) or outbound (egress) rules.

The *Pod selector* is used to match the Pods to which the NetworkPolicy should be applied. This selection is done by using labels, which are metadata attached to Pods. The labels allow for a flexible and dynamic grouping of Pods, meaning that the same NetworkPolicy can be applied to multiple Pods that share the same labels and are running in the same namespace as the NetworkPolicy. Pod selectors are described in detail in "Labels" on page 8.

The list of *ingress* and *egress* rules defines which inbound and outbound connections are allowed for the Pods matched by the Pod selector. These rules specify which sources and destinations are allowed to connect to and from the Pods. For example, a rule could allow connections from a specific IP address or range of addresses, or it could block connections to a specific destination.

1 Level 3 and Level 4 of the OSI Network stack are mostly about IP and TCP/UDP, respectively.

Let's start with the simple example in Example 24-1 that allows access to all database Pods only from backend Pods and nothing else.

Example 24-1. Simple NetworkPolicy allowing ingress traffic

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-database
spec:
  podSelector:          ❶
    matchLabels:
      app: chili-shop
      id: database
  ingress:              ❷
  - from:
    - podSelector:      ❸
        matchLabels:
          app: chili-shop
          id: backend
```

❶ Selector matching all Pods with the label `id: database` and `app: chili-shop`. All those Pods are affected by this NetworkPolicy.

❷ List of sources that are allowed for incoming traffic.

❸ Pod selector that will allow all Pods of the type `backend` to access the selected database Pods.

Figure 24-1 shows how the backend Pods can access the database Pods but frontend Pods can't.

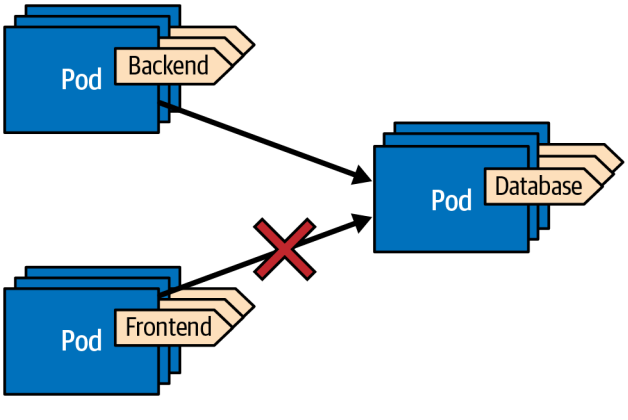


Figure 24-1. NetworkPolicy for ingress traffic

NetworkPolicy objects are namespace-scoped and match only Pods from within the NetworkPolicy's namespace. Unfortunately, there is no way to define cluster-wide defaults for all namespaces. However, some CNI plugins like Calico support customer extensions for defining cluster-wide behavior.

Network segment definition with labels

In Example 24-1, we can see how label selectors are used to dynamically define groups of Pods. This is a powerful concept in Kubernetes that allows users to easily create distinct networking segments.

Developers are typically the best ones to know which Pods belong to a specific application and how they communicate with one another. By carefully labeling the Pods, users can directly translate the dependency graphs of distributed applications into NetworkPolicies. These policies can then be used to define the network boundaries for an application, with well-defined entry and exit points.

To create network segmentation using labels, it's common to label all Pods in the application with a unique `app` label. The `app` label can be used in the selector of the NetworkPolicy to ensure that all Pods belonging to the application are covered by the policy. For example, in Example 24-1, the network segment is defined using an `app` label with the value `chili-shop`.

There are two common ways to consistently label workloads:

- Using workload-unique labels, you can directly model the dependency graph between application components such as other microservices or a database. These workloads can consist of multiple Pods, for example, when deployed in high availability. This technique is used to model the permission graph in Example 24-1, where we use a label `type` to identify the application component. Only one type of workload (e.g., Deployment or StatefulSet) is expected to carry the label `type: database`.
- In a more loosely coupled approach, you can define specific `role` or `permissions` labels that need to be attached to every workload that plays a certain role. Example 24-2 shows an example of this setup. This approach is more flexible and allows for new workloads to be added without updating the NetworkPolicy. However, the more straightforward approach of directly connecting workloads is often easier to understand by simply looking at the NetworkPolicy without having to look up all workloads that apply to a role.

Example 24-2. Role-based network segment definition

```
kind: Pod
metadata:
  label:
    app: chili-shop
    id: backend
    role-database-client: 'true'          ❶
    role-cache-client: 'true'
....
---
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-database-client
spec:
  podSelector:
    matchLabels:
      app: chili-shop
      id: database                        ❷
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: chili-shop
          role-database-client: 'true'    ❸
```

❶ Add all roles that enable this backend Pod to access the requested services.

❷ Selector matching the database Pods—i.e., Pods with the label `id: database`.

❸ Every Pod that is a database client (`role-database-client: 'true'`) is allowed to send traffic to the backend Pod.

Deny-all as default policy

In Examples 24-1 and 24-2, we have seen how to individually configure the allowed incoming connections for a selected set of Pods. This setup works fine as long as you don't forget to configure one Pod, since the default mode, when NetworkPolicy is not configured in the namespace, does not restrict incoming and outgoing traffic (allow-all). Also, for Pods that we might create in the future, it is problematic that it might be necessary to remember to add the respective NetworkPolicy.

Therefore, it is highly recommended to start with a deny-all policy, as shown in Example 24-3.

Example 24-3. Deny-all policy for incoming traffic

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-all
spec:
  podSelector: {}      ❶
  ingress: []          ❷
```

❶ An empty selector matches every Pod.

❷ An empty list of ingress rules implies that all incoming traffic gets dropped.

The list of allowed ingresses is set to an empty list (`[]`), which implies there is no ingress rule that allows incoming traffic. Note that an empty list `[]` is different from a list with a single empty element `[ {} ]`, which achieves the exact opposite since the single empty rule matches everything.

Ingress

Example 24-1 covers the primary use case of a policy that covers ingress traffic. We have already explained the `podSelector` field and given an example of an `ingress` list that matches Pods that are allowed to send traffic to the Pod under configuration. The selected Pod can receive traffic if any of the configured ingress rules in the list are matched.

Besides selecting Pods, you have additional options to configure the ingress rules. We already saw the `from` field for an ingress rule that can contain a `podSelector` for selecting all Pods that pass this rule. In addition, a `namespaceSelector` can be given to choose the namespaces in which the `podSelector` should be applied to identify the Pods that can send traffic.

Table 24-1 shows the effect of the various combinations of `podSelector` and `namespaceSelector`. Combining both fields allows for very flexible setups.

Table 24-1. Combinations of setting `podSelector` and `namespaceSelector` (`{}: empty`, `{...}: non-empty`, `---: unset`)

podSelector	namespaceSelector	Behavior
{}	{}	Every Pod in every namespace
{}	{ … }	Every Pod in the matched namespaces
{ … }	{}	Every matching Pod in all namespaces
{ … }	{ … }	Every matching Pod in the matching namespaces
---	{ … } / {}	Every Pod in the matching namespace/all namespaces
{ … } / {}	---	Matching Pods/every Pod in the NetworkPolicy's namespace

As an alternative for selecting Pods from the cluster, a range of IP addresses can be specified with a field `ipBlock`. We show IP ranges in Example 24-5.

Another option is to restrict the traffic to specific ports to the selected Pod. We can specify this list with a `ports` field that contains all allowed ports.

Egress

Not only can incoming traffic be regulated, but so can any request that a Pod sends in the outgoing direction. Egress rules are configured precisely with the same options as ingress rules. And as with ingress rules, starting with a very restrictive policy is recommended. However, denying all outgoing traffic is not practical. Every Pod needs interaction with Pods from the system namespace for DNS lookups. Also, if we use ingress rules to restrict incoming traffic, we would have to add mirrored egress rules for the source Pods. So let's be pragmatic and allow all egress within the cluster, forbid everything outside the cluster, and let ingress rules define the network boundaries.

Example 24-4 shows the definition of such a rule.

Example 24-4. Allow all internal egress traffic

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: egress-allow-internal-only
spec:
  policyTypes:                  ❶
  - Egress
  podSelector: {}               ❷
  egress:
  - to:
    - namespaceSelector: {}     ❸
```

❶ Add only `Egress` as policy type; otherwise, Kubernetes assumes that you want to specify ingress and egress.

❷ Apply NetworkPolicy to all Pods in the NetworkPolicy's namespace.

❸ Allow egress to every Pod in every other namespace.

Figure 24-2 illustrates the effect of this NetworkPolicy and how it prevents Pods from connecting to external services.

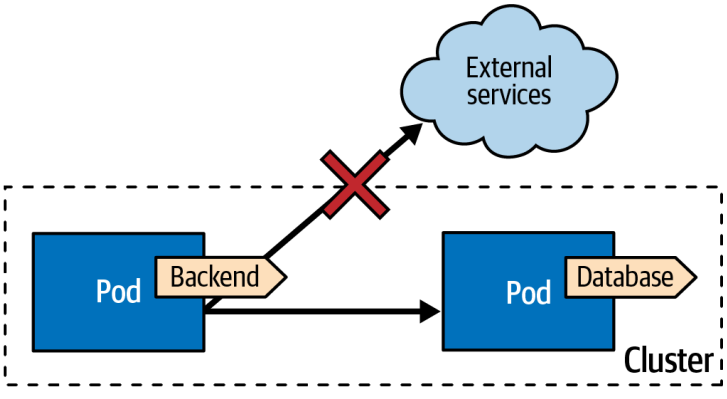


Figure 24-2. NetworkPolicy that allows only internal egress traffic

The `policyTypes` field in a NetworkPolicy determines the type of traffic the policy affects. It is a list that can contain the elements `Egress` and/or `Ingress`, and it specifies which rules are included in the policy. If the field is omitted, the default value is determined based on the presence of the `ingress` and `egress` rule sections:

- If an `ingress` section is present, the default value of `policyTypes` is `[Ingress]`.
- If an `egress` section is provided, the default value of `policyTypes` is `[Ingress, Egress]` regardless of whether ingress rules are provided.

This default behavior implies that to define an egress-only policy, you must explicitly set `policyTypes` to `[Egress]`, as in Example 24-4. Failing to do so would imply an empty `ingress` rules set, effectively forbidding all incoming traffic.

With this restriction for cluster-internal egress traffic in place, we can selectively activate access to external IP addresses for certain Pods that might require cluster-external network access. In Example 24-5, such an IP range block for allowing external egress access is defined.

Example 24-5. NetworkPolicy that allows access to all IP addresses, with some exceptions

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-external-ips
spec:
  podSelector: {}
  egress:
  - to:
    - ipBlock:
        cidr: 0.0.0.0/0    ❶
        except:
        - 192.168.0.0/16   ❷
        - 172.23.42.0/24
```

❶ Allow access to all IP addresses…

❷ …except IP addresses that belong to these subnets.

Some care must be taken if you decide to choose more strict egress rules and also want to restrict the cluster's internal egress traffic. First, it is essential to always allow access to the DNS server in the kube-system namespace. This configuration is best done by allowing access to port 53 for UDP and TCP to all ports in the system namespace.

For operators and controllers, the Kubernetes API server needs to be accessible. Unfortunately, no unique label would select the API server in the kube-system namespace, so the filtering should happen on the API server's IP address. The IP address can best be fetched from the `kubernetes` endpoints in the default namespace with `kubectl get endpoints -n default kubernetes`.

Tooling

Setting up the network topology with NetworkPolicies gets complex quickly since it involves creating many NetworkPolicy resources. It is best to start with some simple use cases that you can adapt to your specific needs. Kubernetes Network Policy Recipes is a good starting point.

Commonly, NetworkPolicies are defined along with the application's architecture. However, sometimes you must retrofit the policy schemas to an existing solution. In this case, policy advisor tools can be beneficial. They work by recording the network activity when playing through typical use cases. A comprehensive integration test suite with good test coverage pays off to catch all corner cases involving network connections. As of 2023, several tools can help you audit network traffic to create network policies.

Inspektor Gadget is a great tool suite for debugging and inspecting Kubernetes resources. It is entirely based on eBPF programs that enable kernel-level observability and provides a bridge from kernel features to high-level Kubernetes resources. One of Inspektor Gadget's features is to monitor network activity and record all UDP and TCP traffic for generating Kubernetes network policies. This technique works well but depends on the quality and depth of covered use cases.

What Is eBPF?

eBPF is a Linux technology that can run sandboxed programs in kernel space.[2] This technique extends the kernel's capabilities safely and allows for much faster innovation on top of this interface.

To some degree, eBPF is the next-generation plugin architecture for the Linux kernel. The flexibility of this API has fostered the evolution of many eBPF projects that cover a wide area of use cases, including observability and security.

Another great eBPF-based platform is Cilium, which has a dedicated audit mode that tracks all network traffic and matches it against a given network policy. By starting with a deny-all policy and audit mode enabled, Cilium will record all policy violations but will not block the traffic otherwise. The audit report helps create the proper NetworkPolicy to fit the traffic patterns exercised.

These are only two examples of the rich and growing landscape of tools for policy recommendation, simulations, and auditing.

Now that you have seen how we can model the network boundaries for our application on the TCP/UDP and IP levels, let's move up some levels in the OSI stack.

Authorization Policies

Until now, we looked at how we can control the network traffic between Pods on the TCP/IP level. However, it is sometimes beneficial to base the network restrictions on filtering on higher-level protocol parameters. This advanced network control requires knowledge of higher-level protocols like HTTP and the ability to inspect incoming and outgoing traffic. Kubernetes does not support this out of the box. Luckily, a whole family of add-ons extends Kubernetes to provide this functionality: service meshes.

2 eBPF was originally an acronym for "extended Berkeley Packet Filter" but is nowadays used as an independent term on its own.

Service Mesh

Some operational requirements like security, observability, or reliability affect all your applications. A service mesh takes care of these aspects in a generic way so that applications can focus on their business logic. Service meshes usually work by injecting sidecar containers into the workload Pods that act as the ambassador from Chapter 18 and adapter from Chapter 17 to intercept L7 incoming and outgoing traffic. Newer techniques to intercept the network traffic include node-wide proxies and a mesh data plane.

Prominent examples of service meshes are Istio, Gloo Mesh, and Linkerd. Still, many more are listed in the CNCF Cloud Native Interactive Landscape.

We chose Istio as our example service mesh, but you will find similar functionalities in other service meshes. We won't go into much detail about service meshes or Istio. Instead, we'll focus on a particular custom resource of Istio that helps us shape the networking segments on the HTTP protocol level.

Istio has a rich feature set for enabling authentication, transport security via mTLS, identity management with CERT rotations, and authorization.

As with other Kubernetes extensions, Istio leverages the Kubernetes API machinery by introducing its own CustomResourceDefinitions (CRDs) that are explained in detail in Chapter 28, "Operator". Authorization in Istio is configured with the AuthorizationPolicy resource. While AuthorizationPolicy is only one component in Istio's security model, it can be used alone and allows for partitioning the network space based on HTTP.

The schema of AuthorizationPolicy is very similar to NetworkPolicy but is more flexible and includes HTTP-specific filters. NetworkPolicy and AuthorizationPolicy should be used together. This can lead to a tricky debugging setup when two configurations must be checked and verified in parallel. Traffic will pass through to a Pod only if the two user-defined firewalls spanned by NetworkPolicy and AuthorizationPolicy definition will allow it.

An AuthorizationPolicy is a namespaced resource and contains a set of rules that control whether or not traffic is allowed or denied to a particular set of Pods in a Kubernetes cluster. The policy consists of the following three parts:

Selector
: Specifies which Pods the policy applies to. If no selector is specified, the policy applies to all Pods in the same namespace as the policy. If the policy is created in Istio's root namespace (`istio-system`), it applies to all matching Pods in all namespaces.

Action
: Defines what should be done with the traffic that matches the rules. The possible actions are `ALLOW`, `DENY`, `AUDIT` (for logging only), and `CUSTOM` (for user-defined actions).

List of rules
: These are evaluated for incoming traffic. All of the rules must be satisfied for the action to be taken. Each rule has three components: a `from` field that specifies the source of the request, a `to` field that specifies the HTTP operation that the request must match, and an optional `when` field for additional conditions (e.g., the identity associated with the request must match a particular value).

Example 24-6 shows a typical example that allows the monitoring operator access to application endpoints for collecting metric data.

Example 24-6. Authorization for a Prometheus setup

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: prometheus-scraper
  namespace: istio-system    ❶
spec:
  selector:                  ❷
    matchLabels:
      has-metrics: "true"
  action: ALLOW              ❸
  rules:
  - from:                    ❹
    - source:
        namespaces: ["prometheus"]
    to:
    - operation:             ❺
        methods: [ "GET" ]
        paths: ["/metrics/*"]
```

❶ When created in the namespace `istio-system`, the policy applies to all matching Pods in all namespaces.

❷ The policy is applied to all Pods with a `has-metrics` label set to `true`.

❸ The action should allow the request to pass if the rules match.

❹ Every request coming from a Pod from the `prometheus` namespace…

❺ …can perform a GET request on the `/metrics` endpoint.

In Example 24-6, every Pod that carries the label `has-metrics: "true"` allows traffic to its `/metrics` endpoint from each Pod of the `prometheus` namespace.

This policy has an effect only if, by default, all requests are denied. As for NetworkPolicy, the best starting point is to define a deny-all policy, as shown in Example 24-7, and then selectively build up the network topology by allowing dedicated routes.

Example 24-7. Deny-all policy as the default

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
 name: deny-all
 namespace: istio-system ❶
spec: {}                 ❷
```

❶ The policy applies to all namespaces since it is created in `istio-system`.

❷ Policies with an empty `spec` section deny all requests.

With the help of the proper labeling schema, AuthorizationPolicy helps define the application's network segments that are independent and isolated from one another. All that we said in "Network segment definition with labels" on page 225 also applies here.

However, AuthorizationPolicy can also be used for application-level authorization when we add an identity check to the rules. One crucial difference to the authorization that we describe in Chapter 26, "Access Control", is that AuthorizationPolicy is about *application authorization*, while the Kubernetes RBAC model is about securing the access to the Kubernetes API server. Access control is primarily helpful for operators monitoring their custom resources.

Discussion

In the early days of computing, network topologies were defined by physical wiring and devices like switches. This approach is secure but not very flexible. With the advent of virtualization, these devices were replaced by software-backed constructs to provide network security. *Software-defined networking* (SDN) is a type of computer networking architecture that allows network administrators to manage network services through abstraction of lower-level functionality. This abstraction is typically achieved by separating the control plane, which makes decisions about how data should be transmitted, from the data plane, which actually sends the data. Even with the use of SDN, administrators are still needed to set up and rearrange networking boundaries to effectively manage the network.

Kubernetes has the ability to overlay its flat cluster-internal network with network segments defined by users through the Kubernetes API. This is the next step in the evolution of network user interfaces. It shifts the responsibility to developers who understand the security requirements of their applications. This shift-left approach is beneficial in a world of microservices with many distributed dependencies and a complex network of connections. NetworkPolicies for L3/L4 network segmentation and AuthorizationPolicies for more granular control of network boundaries are essential for implementing this *Network Segmentation* pattern.

With the advent of eBPF-based platforms on top of Kubernetes, there is additional support for finding suitable network models. Cilium is an example of a platform that combines L3/L4 and L7 firewalling into a single API, making it easier to implement the pattern described in this chapter in future versions of Kubernetes.

More Information

- Network Segmentation Example
- Network Policies
- The Kubernetes Network Model
- Kubernetes Network Policy Recipes
- Using Network Policies
- Why You Should Test Your Kubernetes Network Policies
- Using the eBPF Superpowers to Generate Kubernetes Security Policies
- Using Advise Network-Policy with Inspektor Gadget
- You and Your Security Profiles; Generating Security Policies with the Help of eBPF
- kube-iptables-tailer
- Creating Policies from Verdicts
- Istio: Authorization Policy
- SIG Multitenancy Working Group

CHAPTER 25

Secure Configuration

No real-world application lives in isolation. Instead, each connects to external systems in one way or the other. Such external systems could include value-add services provided by the big cloud providers, other microservices that your service connects to, or a database. Regardless of which remote services your application connects to, you will likely need to go through authentication, which involves sending over credentials such as username and password or some other security token. This confidential information must be stored somewhere close to your application securely and safely. This chapter's *Secure Configuration* pattern is about the best ways to keep your credentials as secure as possible when running on Kubernetes.

Problem

As you learned in Chapter 20, "Configuration Resource", despite what its name implies, Secret resources are not encrypted but are only Base64 encoded. Nevertheless, Kubernetes does its best to restrict access to a Secret's content with the techniques described in "How Secure Are Secrets?" on page 190.

However, as soon as Secret resources are stored outside the cluster, they are naked and vulnerable. With the advent of GitOps as a prevalent paradigm for deploying and maintaining server-side applications, this security challenge is even more pressing. Should Secrets be stored on remote Git repositories? If so, then they must not be stored unencrypted. However, when those are committed encrypted in a source code management system like Git, where do they get decrypted on their way into a Kubernetes cluster?

Even when credentials are stored encrypted within the cluster, it is not guaranteed that nobody else can access that confidential information. While you can granularly

regulate access to Kubernetes resources with RBAC rules,[1] at least one person has access to all data stored in the cluster: your cluster administrator. You might or might not be able to trust the cluster administrator. It all depends on the context in which your application operates. Are you running a Kubernetes cluster in the cloud operated by somebody else? Or is your application deployed on a big company-wide Kubernetes platform, and you need to know who is running this cluster? Different solutions are required depending on these trust boundaries and confidentiality requirements.

Secrets are the Kubernetes answer for confidential configuration in-cluster storage. We talked in depth about Secrets in Chapter 20, "Configuration Resource", so let's now have a look at how we can improve various security aspects of Secrets with additional techniques.

Solution

The most straightforward solution for secure configuration is decoding encrypted information within the application itself. This approach always works, and not just when running on Kubernetes. But it takes considerable work to implement this within your code, and it couples your business logic with this aspect of securing your configuration. There are better, more transparent ways to do this on Kubernetes.

The support for secure configuration on Kubernetes falls roughly into two categories:

Out-of-cluster encryption
: This stores encrypted configuration information outside of Kubernetes, which nonauthorized persons can also read. The transformation into Kubernetes Secrets happens just before entering the cluster (e.g., when applying a resource via the API server) or inside the cluster by a permanently running operator process.

Centralized secret management
: This uses specialized services that are either already offered by cloud providers (e.g., AWS Secrets Manager or Azure Key Vault) or are part of an in-house vault service (e.g., HashiCorp Vault) for storing confidential configuration data.

While out-of-cluster encryption techniques always eventually create a Secret within the cluster that your application can use, the support for external secret management systems (SMSs) provided by Kubernetes add-ons uses various other techniques to bring the confidential information to the deployed workloads.

1 RBAC rules are explained in detail in Chapter 26, "Access Control".

Out-of-Cluster Encryption

The gist of the out-of-cluster technique is simple: pick up secret and confidential data from outside the cluster and transform it into a Kubernetes Secret. A lot of projects have been grown that implement this technique. This chapter looks at the three most prominent ones (as of 2023): Sealed Secrets, External Secrets, and sops.

Sealed Secrets

One of the oldest Kubernetes add-ons for helping with encrypted secrets is *Sealed Secrets*, introduced by Bitnami in 2017. The idea is to store the encrypted data for a Secret in a CustomResourceDefinition (CRD) SealedSecret. In the background, an operator monitors such resources and creates one Kubernetes Secret for each SealedSecret with the decrypted content. To learn more about CRDs and operators in general, check out Chapter 28, "Operator", which explains this pattern in detail. While the decryption happens within the cluster, the *encryption* happens outside by a CLI tool called `kubeseal`, which takes a Secret and translates it to a SealedSecret that can be stored safely in a source code management system like Git.

Figure 25-1 shows the setup for Sealed Secrets.

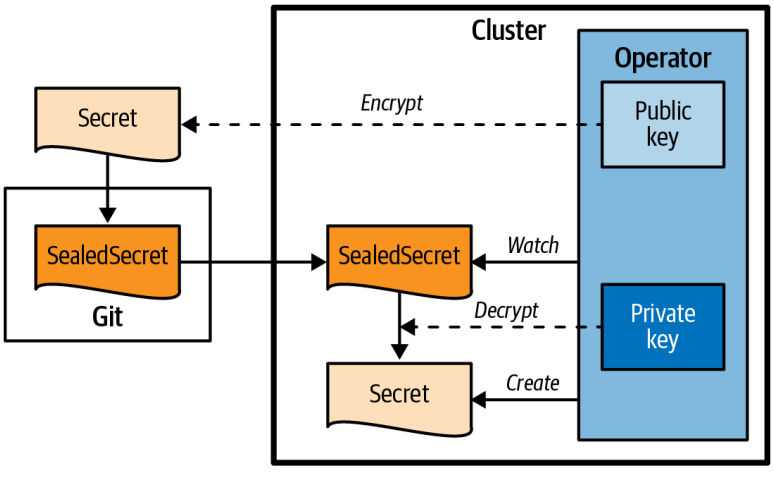


Figure 25-1. Sealed Secrets

Secrets are encrypted with AES-256-GCM symmetrically as a session key, and the session key is encrypted asymmetrically with RSA-OAEP, the same setup TLS uses.

The secret private key is stored within the cluster and is automatically created by the SealedSecret Operator. It is up to the administrator to back up this key and rotate it if needed. The public key used by `kubeseal` can be fetched directly from the cluster or

accessed directly from a file. You also can safely store the public key in Git along with your SealedSecret.

SealedSecrets support three scopes that you can select when creating a SealedSecret from a Secret:

Strict
: This freezes the namespace and name of the SealedSecret. This mode means you can create the SealedSecret only in the same namespace and with the same name as the original Secret in any target cluster. This mode is the default behavior.

Namespace-wide
: This allows you to apply the SealedSecret to a different name than the initial Secret but still pins it to the same namespace.

Cluster-wide
: This allows you to apply the SealedSecret to different namespaces, as it was initially created to do, and the name can be changed too.

These scopes can be selected when creating the SealedSecret with `kubeseal`. Still, you can also add the nonstrict scopes with the annotations listed in Table 25-1 on the original Secret before encryption or on the SealedSecret directly.

Table 25-1. Annotation

Annotation	Value	Description
sealedsecrets.bitnami.com/namespace-wide	`"true"`	Enable namespace-wide scope when set to `true`—i.e., different name but same namespace
sealedsecrets.bitnami.com/cluster-wide	`"true"`	Enable cluster-wide scope when set to `true`—i.e., name and namespace can be changed on the SealedSecret after encryption

Example 25-1 shows a SealedSecret created by `kubeseal` that can be directly stored in Git.

Example 25-1. SealedSecret created with `kubeseal`

```
# Command to create this sealed secret:
# kubeseal --scope cluster-wide -f mysecret.yaml          ❶
apiVersion: bitnami.com/v1alpha1
kind: SealedSecret
metadata:
  annotations:
    sealedsecrets.bitnami.com/cluster-wide: "true"        ❷
  name: DB-credentials
spec:
  encryptedData:
    password: AgCrKIIF2gA7tSR/gqw+FH6cEV..wPWWkHJbo=      ❸
    user: AgAmvgFQBBNPlt9Gmx..0DNHJpDIMUGgwaQroXT+o=
```

❶ Command to create a SealedSecret from the secret stored in *mysecret.yaml*.

❷ Annotation that indicates that this SealedSecret can have any name and be applied to any namespace.

❸ The secret values are encrypted individually (and shortened here for the sake of demonstration).

A Sealed Secret is a tool that allows you to store encrypted secrets in a publicly available location, such as a GitHub repository. It is important to properly back up the secret key, as without it, it will not be possible to decrypt the secrets if the operator is uninstalled. One potential drawback of Sealed Secrets is that they require a server-side operator to be continuously running in the cluster in order to perform the decryption.

External Secrets

The External Secrets Operator is a Kubernetes operator that integrates a growing list of external SMSs. The main difference between External Secrets and Sealed Secrets is that you do not manage the encrypted data storage yourself but rely on an external SMS to do the hard work, including encryption, decryption, and secure persistence. That way, you benefit from all the features of your cloud's SMS, like key rotation and a dedicated user interface. SMS also provides an excellent way of separating concerns so that different roles can manage the application deployments and the secrets separately.

Figure 25-2 shows the External Secrets architecture.

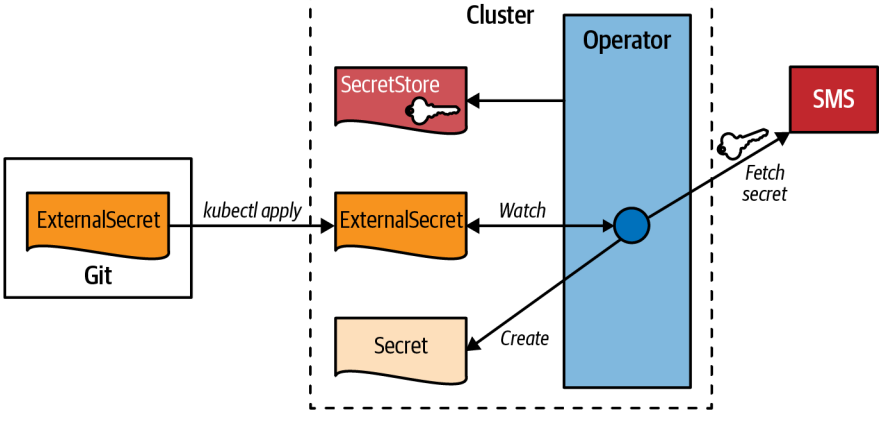


Figure 25-2. External Secrets

A central operator reconciles two custom resources:

- SecretStore is the resource that holds the type and configuration of the external SMS to access. Example 25-2 gives an example of a store that connects to AWS Secret Manager.
- ExternalSecret references a SecretStore, and the operator will create a corresponding Kubernetes Secret filled with the data fetched from the external SMS. For example, Example 25-3 references a secret in the AWS Secret Manager and exposes the value within the specified target Secret.

Example 25-2. SecretStore for connecting to AWS Secret Manager

```
apiVersion: external-secrets.io/v1beta1
kind: SecretStore
metadata:
  name: secret-store-aws
spec:
  provider:
    aws:                          ❶
      service: SecretsManager
      region: us-east-1
      auth:
        secretRef:
          accessKeyIDSecretRef:   ❷
            name: awssm-secret
            key: access-key
          secretAccessKeySecretRef:
            name: awssm-secret
            key: secret-access-key
```

❶ Provider `aws` configures the usage of the AWS Secret Manager.

❷ Reference to a Secret that holds the access keys for talking with the AWS Secret Manager. A Secret with the name `awssm-secret` contains the keys `access-key` and `secret-access-key` used to authenticate against the AWS Secret Manager.

Example 25-3. ExternalSecret that will be transformed into a Secret

```
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: db-credentials
spec:
  refreshInterval: 1h
  secretStoreRef:                 ❶
    name: secret-store-aws
    kind: SecretStore
  target:
```

```
    name: db-credentials-secrets ❷
    creationPolicy: Owner
  data:
    - key: cluster/db-username    ❸
      name: username
    - key: cluster/db-password
      name: password
```

❶ Reference to the SecretStore object that holds the connection parameters for AWS Secret Manager.

❷ Name of the Secret to create.

❸ The username that will be looked up under `cluster/DB-username` in AWS Secret Manager and put under the key `username` in the resulting Secret.

You have a lot of flexibility in defining the mapping of the external secret data to the content of the mirrored Secret—for example, using a template to create a configuration with a particular structure. See the External Secrets documentation for more information. One significant advantage of this solution over a client-side solution is that only the server-side operator knows the credentials to authenticate against the external SMS.

The External Secrets Operator project merges several other Secret-syncing projects. In 2023, it is already the dominant solution for this specific use case of mapping and syncing an externally defined secret to a Kubernetes Secret. However, it has the same cost as a server-side component that runs all the time.

Sops

Do we need a server-side component to work with Secrets in a GitOps world where all resources are stored in a Git repository? Luckily, solutions exist that work entirely outside of a Kubernetes cluster. A pure client-side solution is sops ("Secret OPerationS") by Mozilla. Sops is not specific to Kubernetes but allows you to encrypt and decrypt any YAML or JSON file to safely store those in a source code repository. It does this by encrypting all values of such a document but leaving the keys untouched.

We can use various methods for encryption with sops:

- Asymmetric local encryption via `age` with the keys stored locally.
- Storing the secret encryption key in a centralized key management system (KMS). Supported platforms are AWS KMS, Google KMS, and Azure Key Vault as external cloud providers and HashiCorp Vault as an SMS you can host on your own. The identity management of those platforms allows for fine-granular access control to the encryption key.

SMS Versus KMS

In the previous sections, we talked about *secret management systems* (SMSs), cloud services that do secret management for you. They provide an API for storing and accessing the secrets with granular and configurable access control. Those secrets are encrypted transparently for the user, and you don't have to worry about this. *Key management systems* (KMSs) are cloud services you can access with an API. However, in contrast to SMSs, KMSs are not databases for secure data but care about the discovery and storage of encryption keys, which you can use to encrypt data outside of a KMS. The GnuPG keyservers are good examples of a KMS. Each leading cloud provider offers both SMSs and KMSs. If you are sold to one of the big clouds, you also get good integration with its identity management for defining and assigning the access rules to SMS- and KMS-managed data.

Sops is a CLI tool you can run locally on your machine or within a cluster (e.g., as part of a CI pipeline). Especially for the latter use case and if you are running in one of the big clouds, leveraging one of their KMSs provides a smooth integration.

Figure 25-3 illustrates how sops handles encryption and decryption on the client side.

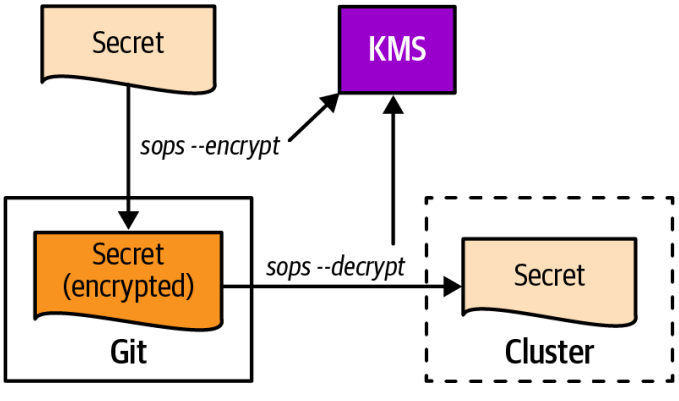


Figure 25-3. Sops for decrypting and encrypting resource files

Example 25-4 shows how to use sops to create an encrypted version of a ConfigMap.[2] This example uses `age` and a freshly generated keypair for the encryption, which should be stored safely.

2 In the real world, you should use a Secret for this kind of confidential information, but here we use a ConfigMap to demonstrate that you can use *any* resource file with sops.

Example 25-4. Sops for creating encrypted secrets

```
$ age-keygen -o keys.txt                    ❶
Public key: age1j49ugcg2rzyye07ksyvj5688m6hmv

$ cat configmap.yaml                        ❷
apiVersion: v1
kind: ConfigMap
metadata:
  name_unencrypted: db-auth                 ❸
data:
  # User and Password
  USER: "batman"
  PASSWORD: "r0b1n"

$ sops --encrypt \                          ❹
    --age age1j49ugcg2rzyye07ksyvj5688m6hmv \
    configmap.yaml > configmap_encrypted.yaml

$ cat configmap_encrypted.yaml
apiVersion: ENC[AES256_GCM,data:...,iv:...,tag:...,type:str] ❺
kind: ENC[AES256_GCM,data:...,iv:...,tag:...,type:str]
metadata:
    name_unencrypted: db-auth               ❻
data:
    #ENC[AES256_GCM,data:...,iv:...,tag:...,type:comment]
    USER: ENC[AES256_GCM,data:...,iv:...,tag:...=,type:str]
    PASSWORD: ENC[AES256_GCM,data:...,iv:...,tag:...,type:str]
sops:                                       ❼
    age:
        - recipient: age1j49ugcg2rzyye07ksyvj5688m6hmv
          enc: |                            ❽
            -----BEGIN AGE ENCRYPTED FILE-----
            YWdlLWVuY3J5cHRpb24ub3JnL3YxCi0+IFgyNTUxOSBqems3QkU4aXRyQWxaNER1
            TTdqcUZTeXFXNWhSY0E1T05XMUhVUzFjR1FnCmdMZmhlSlZCRHlqTzlNM0E1Z280
            Y0tqQ2VKYXdxdDZIZHpDbmxTYzhQSTgKLS0tIHlBYmloL2laZlA4Q05DTmRwQ0ls
            bURoU2xITHNzSXp5US9mUUV0Z0RackkKFtH+uNNe3A13pzSvHjT6n3q9av0pN7Nb
            i3AULtKvAGs6oAnH8qYbnwoj3qt/LFfnbqfeFk1zC2uqNONWkKxa2Q==
            -----END AGE ENCRYPTED FILE-----
    last modified: "2022-09-20T09:56:49Z"
    mac: ENC[AES256_GCM,data:...,iv:...,tag:...,type:str]
    unencrypted_suffix: _unencrypted
```

❶ Create a secret key with `age` and store it in *keys.txt*.

❷ The ConfigMap to encrypt.

❸ The `name` field is changed to `name_unencrypted` to prevent it from getting encrypted.

❹ Call sops with the public part of the `age` key, and store the result in *configmap_encrypted.yml*.

❺ Each value is replaced with an encrypted version in `ENC[...]` (output shortened for readability).

❻ The `name` of the ConfigMap is left untouched.

❼ An extra section, `sops` is appended to contain metadata that is needed for decryption.

❽ Encrypted session key that is used for symmetrical decryption. This key itself is encrypted asymmetrically by `age`.

As you can see, every value of the ConfigMap resource gets encrypted, even those that are not confidential, like resource types or the name of the resource. You can skip the encryption for specific values by appending an `_unencrypted` suffix to the key (which gets stripped off later when doing the decryption).

The generated *configmap_encrypted.yml* can safely be stored in Git or any other source control management. As shown in Example 25-5, you need the private key to decrypt the ciphered ConfigMap to apply it to the cluster.

Example 25-5. Decrypt sops-encoded resource and apply it to Kubernetes

```
$ export SOPS_AGE_KEY_FILE=keys.txt ❶
$ sops --decrypt configmap_encrypted.yaml | kubectl apply -f - ❷
configmap/db-auth created
```

❶ Point sops to the private key to decrypt the session key.

❷ Decrypt and apply to Kubernetes. Note that every `_unencrypted` suffix on the resource keys is removed during sops decryption.

Sops is an excellent solution for easy GitOps-style integration of Secrets without worrying about installing and maintaining Kubernetes add-ons. However, while your configuration can now be stored securely in Git, it is essential to understand that as soon as those configurations have been handed over to the cluster, anybody with elevated access rights can read that data directly via the Kubernetes API.

If this is not something you can tolerate, we need to dig deeper into the toolbox and look again at centralized SMSs.

Centralized Secret Management

As explained in "How Secure Are Secrets?" on page 190, Secrets are as secure as possible. Still, any administrator with cluster-wide read access can read every Secret stored unencrypted. Depending on your trust relationship with your cluster operators and security requirements, this might or might not be a problem.

Besides baking individual secret handling into your application code, an alternative is to keep the secure information outside the cluster in the external SMS and request the confidential information on demand over secure channels.

There is a growing number of such SMSs out there, and every cloud provider offers its variant. We won't go into many details here for those individual offerings but focus on the mechanism of how such systems integrate into Kubernetes. You will find a list of relevant products as of 2023 in "More Information" on page 252.

Secrets Store CSI Driver

The Container Storage Interface (CSI) is a Kubernetes API for exposing storage systems to containerized applications. CSI shows the path for third-party storage providers to plug in new types of storage that can be mounted as volumes in Kubernetes. Of particular interest in the context of this pattern is the Secrets Store CSI Driver. This driver, developed and maintained by the Kubernetes community, allows access to various centralized SMSs and mounts them as regular Kubernetes volumes. The difference from a mounted Secret volume as described in Chapter 20, "Configuration Resource", is that nothing is stored in the Kubernetes etcd database but securely outside the cluster.

The Secrets Store CSI Driver supports the SMS from major cloud vendors (AWS, Azure, and GCP) and HashiCorp Vault.

The Kubernetes setup for connecting a secret manager via the CSI driver involves performing these two administrative tasks:

- Installing the Secrets Store CSI Driver and configuration for accessing a specific SMS. Cluster-admin permissions are required for the installation process.
- Configuring access rules and policies. Several provider-specific steps need to be completed, but the result is that a Kubernetes service account is mapped to a secret manager-specific role that allows access to the secrets.

Figure 25-4 shows the overall setup needed for enabling the Secrets Store CSI Driver with a HashiCorp Vault backend.

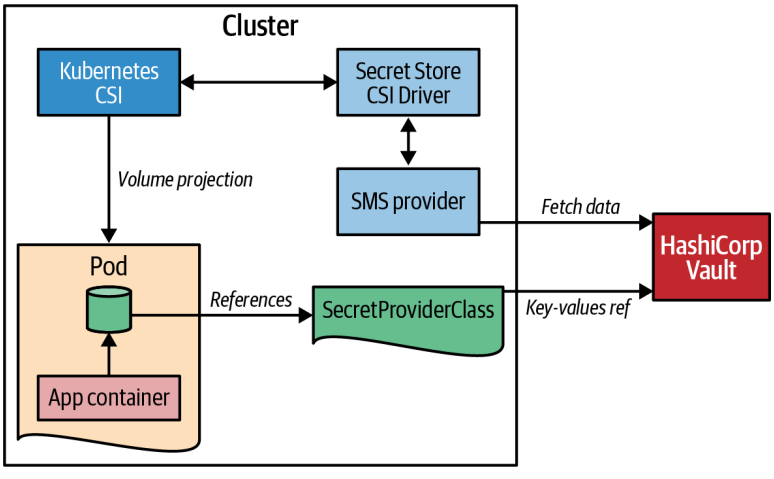


Figure 25-4. Secrets Store CSI Driver

After the setup is done, the usage of secret volumes is straightforward. First, you must define a SecretProviderClass, as demonstrated in Example 25-6. In this resource, you select the backend provider for the secret manager. For our example, we selected HashiCorp's Vault. In the `parameters` section, the provider-specific configuration is added, which contains the connection parameter to the vault, the role to impersonate, and a pointer to the secret information that Kubernetes will mount into a Pod.

Example 25-6. Configuration of how to access a secret manager

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: vault-database
spec:
  provider: vault                                    ❶
  parameters:
    vaultAddress: "http://vault.default:8200"        ❷
    roleName: "database"                             ❸
    objects: |
      - objectName: "database-password"              ❹
        secretPath: "secret/data/database-creds"     ❺
        secretKey: "password"                        ❻
```

❶ Type of provider to use (`azure`, `gcp`, `aws`, or `vault` as of 2023).

❷ Connection URL to the Vault service instance.

❸ Vault-specific authentication role contains the Kubernetes service account allowed to connect.

❹ Name of the file that should be mapped into the mounted volume.

❺ Path to the stored secret in the vault.

❻ Key to pick from the Vault secret.

This secret manager configuration can then be referenced by its name when used as a Pod volume. Example 25-7 shows a Pod that mounts the secrets configured in Example 25-6. One key aspect is the service account `vault-access-sa` with which this Pod runs. This service account must be configured on the Vault side to be part of the role `database` referenced in the SecretProviderClass.

You can find this Vault configuration in our complete working and self-contained example, along with setup instructions.

Example 25-7. Pod mounting a CSI volume from Vault

```
kind: Pod
apiVersion: v1
metadata:
  name: shell-pod
spec:
  serviceAccountName: vault-access-sa  ❶
  containers:
  - image: k8spatterns/random
    volumeMounts:
    - name: secrets-store
      mountPath: "/secrets-store"      ❷
  volumes:
    - name: secrets-store
      csi:                             ❸
        driver: secrets-store.csi.k8s.io
        readOnly: true
        volumeAttributes:
          secretProviderClass: "vault-database" ❹
```

❶ Service account that is used to authenticate against Vault.

❷ Directory in which to mount the secrets.

❸ Declaration of a CSI Driver, which points to the Secret Store CSI driver.

❹ Reference to the SecretProviderClass that provides the connection to the Vault service.

While the setup for a CSI Secret Storage drive is quite complex, the usage is straightforward, and you can avoid storing confidential data within Kubernetes. However, there are more moving parts than with Secrets alone, so more things can go wrong, and it's harder to troubleshoot.

Let's look at a final alternative for offering secrets to applications via well-known Kubernetes abstractions.

Pod injection

As mentioned, an application can always access external SMSs via proprietary client libraries. This approach's disadvantage is that you still have to store the credentials to access the SMS along your application and add a hard dependency within your code to a particular SMS. The CSI abstraction for projecting secret information into volumes visible as files for the deployed application is much more decoupled.

Alternative solutions leverage other well-known patterns described in this book:

- An *Init Container* (see Chapter 15) fetches the confidential data from an SMS and then copies it to a shared local volume that is mounted by the application container. The secret data is fetched only once before the main container starts.
- A *Sidecar* (see Chapter 16) syncs the secret data from the SMS to a local ephemeral volume that is also accessed by the application. The benefit of the sidecar approach is that it can update the secrets locally in case the SMS starts to rotate the secrets.

You can leverage these patterns on your own for your applications, but this is tedious. It is much better to let an external controller inject the init container or sidecar into your application.

An excellent example of such an injector is the HashiCorp Vault Sidecar Agent Injector. This injector is implemented as a so-called *mutating webhook*, a variant of a controller (see Chapter 27, "Controller"), that allows modification of any resource when it is created. When a Pod specification contains a particular, vault-specific annotation, the vault controller will modify this specification to add a container for syncing with Vault and to mount a volume for the secret data.

Figure 25-5 visualizes this technique, which is entirely transparent to the user.

While you still need to install the Vault Injector controller, it has fewer moving parts than hooking up a CSI secret storage volume with the provider deployment for a particular SMS product. Still, you can access all the secrets by just reading a file without using a proprietary client library.

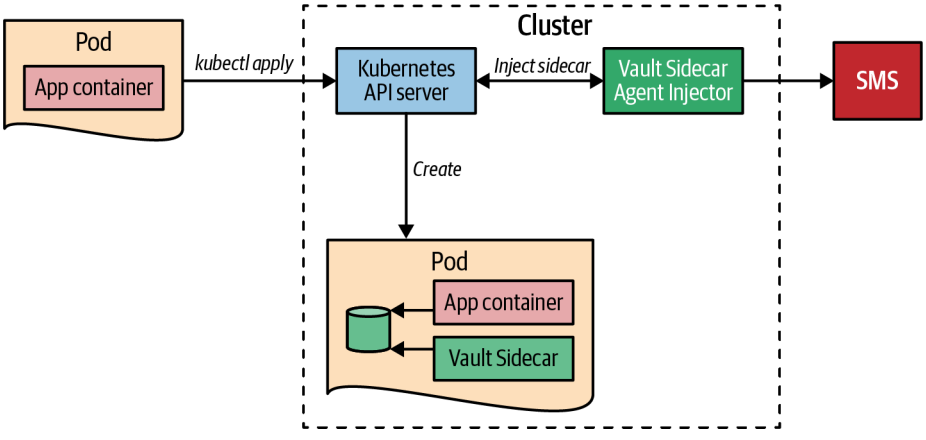


Figure 25-5. Vault Injector

Discussion

Now that we have seen the many ways you can make access to your confidential information more secure, the question is, which one is the best?

As usual, it depends:

- If your main goal is a simple way to encrypt Secrets stored in public-readable places like a remote Git repository, the pure *client-side* encryption that *Sops* offers is perfect.
- The secret synchronization that the *External Secrets Operator* implements is a good choice when separating the concerns of retrieving credentials in a remote SMS and using them is essential.
- The ephemeral volume projection of secret information provided by *Secret Storage CSI Providers* is the right choice for you when you want to ensure that no confidential information is stored permanently in the cluster except the access tokens for accessing external vaults.
- Sidebar injections like the *Vault Sidecar Agent Injector* have the benefit of shielding from a direct access to an SMS. They are easily approachable at the cost of blurring the boundary between developers and administrator because of security annotations leaking into application deployment.

Note that the listed projects are the most prominent as of this writing in 2023. The landscape is constantly evolving, so by the time you read this book, there might be new contenders (or some existing projects might have stopped). However, the techniques used (client-side encryption, Secret synchronization, volume projections, and sidecar injections) are universal and will be part of future solutions.

But a clear warning at the end: regardless of how securely and safely you can access your secret configuration, if somebody with evil intentions has full root access to your cluster and containers, a means to get to that data will always exist. This pattern makes these kinds of exploits as difficult as possible by adding an extra layer on the Kubernetes Secret abstraction.

More Information

- Secure Configuration Example
- Alex Soto Bueno and Andrew Block's *Kubernetes Secrets Management* (Manning, 2022)
- Kubernetes: Sealed Secrets
- Sealed Secrets
- External Secrets Operator
- Kubernetes External Secrets
- Sops
- Kubernetes Secrets Store CSI Driver
- Retrieve HashiCorp Vault Secrets with Kubernetes CSI
- HashiCorp Vault
- Secret Management Systems:
  - Azure Key Vault
  - AWS Secrets Manager
  - AWS Systems Manager Parameter Store
  - GCP Secret Manager

CHAPTER 26

Access Control

As the world becomes increasingly reliant on cloud infrastructure and containerization, the importance of security can never be understated. In 2022, security researchers made a troubling discovery: nearly one million Kubernetes instances were left exposed on the internet due to misconfigurations.[1] Using specialized security scanners, researchers were able to easily access these vulnerable nodes, highlighting the need for stringent access-control measures to protect the Kubernetes control plane. But while developers often focus on application-level authorization, they sometimes also need to extend Kubernetes capabilities using the *Operator* pattern from Chapter 28. In these cases, access control on the Kubernetes platform becomes critical. In this chapter, we delve into the *Access Control* pattern and explore the concepts of Kubernetes authorization. With the potential risks and consequences at stake, it's never been more important to ensure the security of your Kubernetes deployment.

Problem

Security is a crucial concern when it comes to operating applications. At the core of security are two essential concepts: authentication and authorization.

Authentication focuses on identifying the subject, or *who*, of an operation and preventing access by unauthorized actors. *Authorization*, on the other hand, involves determining the permissions for *what* actions are allowed on resources.

In this chapter, we will discuss authentication briefly, as it is primarily an administrative concern that involves integrating various identity-management techniques with Kubernetes. On the other hand, developers are typically more concerned with

1 See the blog post "Exposed Kubernetes Clusters".

authorization, such as who can perform which operations in the cluster and access specific parts of an application.

To secure access to their applications running on top of Kubernetes, developers must consider a range of security strategies, from simple web-based authentication to sophisticated single-sign-on scenarios involving external providers for identity and access management. At the same time, access control to the Kubernetes API server is also an essential concern for applications running on Kubernetes.

Misconfigured access can lead to privilege escalation and deployment failures. High-privilege deployments can access or modify configuration and resources for other deployments, increasing the risk of a cluster compromise.[2] It is important for developers to understand the authorization rules set up by administrators and consider security when making configuration changes and deploying new workloads to meet the organization-wide policies in the Kubernetes cluster.

Furthermore, as more and more Kubernetes-native applications extend the Kubernetes API and offer their services via CustomResourceDefinitions (CRDs) to users, as described in "Controller and Operator Classification" on page 297, access control becomes even more critical. Kubernetes patterns like Chapter 27, "Controller", and Chapter 28, "Operator", require high privileges to observe the state of cluster-wide resources, making it crucial to have fine-grained access management and restrictions in place to limit the impact of any potential security breaches.

Solution

Every request to the Kubernetes API server has to pass through three stages—Authentication, Authorization, and Admission Control, as shown in Figure 26-1.

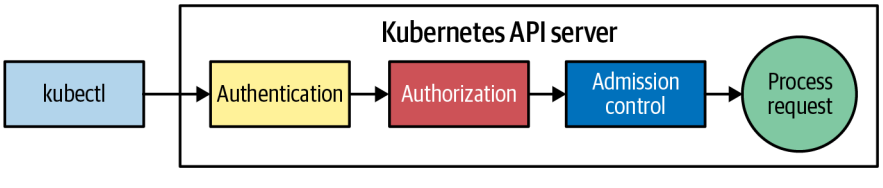


Figure 26-1. A request to the Kubernetes API server must pass through these stages

Once a request passes the Authentication and Authorization stages described in the following sections, a final check is done by Admission controllers before the request is eventually processed. Let's look at these stages separately.

2 An attacker with escalated privileges on a node can compromise a full Kubernetes cluster.

Authentication

As mentioned, we won't go into too much detail about authentication because it is mainly an administration concern. But it's good to know which options are available, so let's have a look at the pluggable authentication strategies Kubernetes has to offer that an administrator can configure:

Bearer Tokens (OpenID Connect) with OIDC Authenticators
: OpenID Connect (OIDC) Bearer Tokens can authenticate clients and grant access to the API Server. OIDC is a standard protocol that allows clients to authenticate with an OAuth2 provider that supports OIDC. The client sends the OIDC token in the Authorization header of their request, and the API Server validates the token to allow access. For the entire flow, see the Kubernetes documentation at OpenID Connect Tokens.

Client certificates (X.509)
: By using client certificates, the client presents a TLS certificate to the API Server, which is then validated and used to grant access.

Authenticating Proxy
: This configuration option refers to using a custom authenticating proxy to verify the client's identity before granting access to the API Server. The proxy acts as an intermediary between the client and the API Server and performs authentication and authorization checks before allowing access.

Static Token files
: Tokens can also be stored in standard files and used for authentication. In this approach, the client presents a token to the API Server, which is then used to look up the token file and search for a match.

Webhook Token Authentication
: A webhook can authenticate clients and grant access to the API Server. In this approach, the client sends a token in the Authorization header of their request, and the API Server forwards the token to a configured webhook for validation. The client is granted access to the API Server if the webhook returns a valid response. This technique is similar to the Bearer Token option, except that you can use an external custom service for performing the token validation.

Kubernetes allows you to use multiple authentication plugins simultaneously, such as Bearer Tokens and Client certificates. If the Bearer Token strategy authenticates a request, Kubernetes won't check the Client certificates, and vice versa. Unfortunately, the order in which these strategies are evaluated is not fixed, so it's impossible to know which one will be checked first. When evaluating the strategies, the process will stop after one is successful, and Kubernetes will forward the request to the next stage.

After authentication, the authorization process will begin.

Authorization

Kubernetes provides RBAC as a standard way to manage access to the system. RBAC allows developers to control and execute actions in a fine-grained manner. The authorization plugin in Kubernetes also provides easy pluggability, allowing users to switch between the default RBAC and other models, such as attribute-based access control (ABAC), webhooks, or delegation to a custom authority.

The ABAC-based approach requires a file containing policies in a JSON per-line format. However, this approach requires the server to be reloaded for any changes, which can be a disadvantage. This static nature is one of the reasons ABAC-based authorization is used only in some cases.

Instead, nearly every Kubernetes cluster uses the default RBAC-based access control, which we describe in great detail in "Role-Based Access Control" on page 263.

Before we focus on authorization in the rest of this chapter, let's quickly look at the last stage performed by admission controllers.

Admission Controllers

Admission controllers are a feature of the Kubernetes API server that allows you to intercept requests to the API server and take additional actions based on those requests. For example, you can use them to enforce policies, perform validations, and modify incoming resources.

Kubernetes uses Admission controller plugins for implementing various functions. The functionality ranges from setting default values on specific resources (like the default storage class on persistent volumes), to validations (like the allowed resource limits for Pods), by calling external web hooks.

These external webhooks can be configured with dedicated resources and are used for validation (ValidatingWebhookConfiguration) and updating (MutatingWebhookConfiguration) API resources. The details of configuring such webhooks are explained in detail in the Kubernetes documentation "Dynamic Admission Control".

We won't go into more detail here as Admission controllers are mostly an administrative concept, and many other good resources describe Admission controllers in particular (see "More Information" on page 275 for some references).

Instead, for the remainder of the chapter, we will focus on the authorization aspect and how we can configure a fine-grained permission model for securing access to the Kubernetes API server.

As mentioned, authentication has two fundamental parts and authorization: the *who*, represented by a subject that can be either a human person or a workload identity, and the *what*, representing the actions those subjects can trigger at the Kubernetes

API server. In the next section, we discuss the who before diving into the details of the *what*.

Subject

A *subject* is all about the *who*, the identity associated with a request to the Kubernetes API server. In Kubernetes, there are two kinds of subjects, as shown in Figure 26-2: human *users* and *service accounts* that represent the workload identity of Pods.

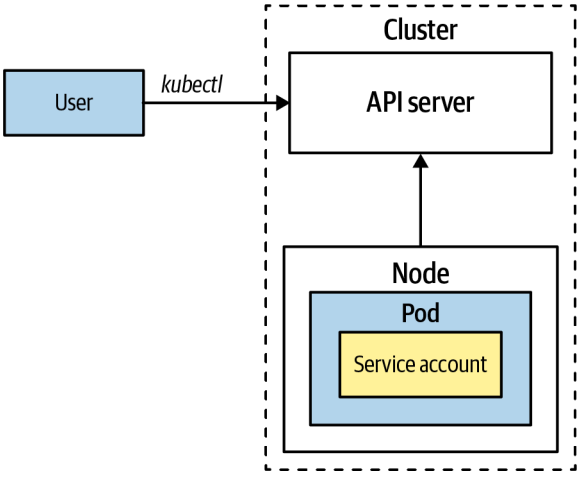


Figure 26-2. Subject (user or service account) requests to API Server

Human users and ServiceAccounts can be separately grouped in *user groups* and *service account groups*, respectively. Those groups can act as a single subject in which all members of the group share the same permission model. We will talk about groups later in this chapter, but first, let's look closely at how human users are represented in the Kubernetes API.

Users

Unlike many other entities in Kubernetes, human users are not defined as explicit resources in the Kubernetes API. This design decision implies that you can't manage users via an API call. The authentication and mapping to a user subject happens outside the usual Kubernetes API machinery by external user management.

As we have seen, Kubernetes supports many ways of authenticating an external user. Each component knows how to extract the subject information after successful authentication. Although this mechanism is different for each authentication component, they will eventually create the same user representation and add it to the actual API request to verify by later stages, as shown in Example 26-1.

Example 26-1. Representation of an external user after successful authentication

```
alice,4bc01e30-406b-4514,"system:authenticated,developers","scopes:openid"
```

This comma-separated list is a representation of the user and contains the following parts:

- The username (`alice`)
- A unique user id (UID) (`4bc01e30-406b-4514`)
- A list of groups that this user belongs to (`system:authenticated,developers`)
- Additional information as comma-separated key-value pairs (`scopes:openid`)

This information is evaluated by the Authorization plugin against the authorization rules associated with the user or via its membership to a user group. In Example 26-1, a user with the username `alice` has the default access associated with the group `system:authenticated` and the group `developers`. The extra information `scope:openid` indicates OIDC is being used to verify the user's identity.

Certain usernames are reserved for internal Kubernetes use and are distinguished by the special prefix `system:`. For example, the username `system:anonymous` represents anonymous requests to the Kubernetes API server. It is recommended to avoid creating your own users or groups with the `system:` prefix to avoid conflicts. Table 26-1 lists the default usernames in Kubernetes that are used when internal Kubernetes components communicate to one another.

Table 26-1. Default usernames in Kubernetes

Username	Purpose
`system:anonymous`	Represents anonymous requests to the Kubernetes API server
`system:apiserver`	Represents the API server itself
`system:kube-proxy`	Represents process identity of the kube-proxy service
`system:kube-controller-manager`	Represents the user agent of the controller manager
`system:kube-scheduler`	Represents the user of the scheduler

While the management and authentication of external users can vary depending on the specific setup of a Kubernetes cluster, the management of workload identities for Pods is a standardized part of the Kubernetes API and is consistent across all clusters.

Service accounts

Service accounts in Kubernetes represent nonhuman actors within the cluster and are used as workload identities. They are associated with Pods and allow running processes inside a Pod to communicate with the Kubernetes API Server. In contrast

to the many ways that Kubernetes can authenticate human users, service accounts always use an OpenID Connect handshake and JSON Web Tokens to prove their identity.

Service accounts in Kubernetes are authenticated by the API server using a username in the following format: `system:serviceaccount:<namespace>:<name>`. For example, if you have a service account, `random-sa`, in the `default` namespace, the service account's username would be `system:serviceaccount:default:random-sa`.

JSON Web Tokens in Kubernetes

JSON Web Tokens (JWTs) are digitally signed tokens that carry a payload. They consist of a header, payload, and signature and are represented as a sequence of Base64 URL-encoded parts separated by periods. Tools like jwt.io can decode, validate, and inspect JWTs.

In the context of Kubernetes, JWTs are used as Bearer Tokens in the `Authorization` HTTP header of API requests to specify the identity of the workload making the request and additional information, such as the expiration time or issuer. The Kubernetes API server verifies the signature of the JWT by comparing it with a public key published in a JSON Web Key Set (JWKS). This process is governed by the JSON Web Key (JWK) specification, which defines the cryptographic algorithms used in the verification process in RFC 7517.

The tokens issued by Kubernetes contain helpful information in the payload of the JWT, such as the issuer of the token, its expiration time, all the user information described in Example 26-1, and the associated service accounts (if any).

A ServiceAccount is a standard Kubernetes resource, as shown in Example 26-2.

Example 26-2. ServiceAccount definition

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: random-sa                    ❶
  namespace: default
automountServiceAccountToken: false  ❷
...
```

❶ Name of the service account.

❷ Flag indicating whether the service account token should be mounted by default into a Pod. The default is set to `true`.

A ServiceAccount has a simple structure and serves all identity-related information needed for a Pod when talking with the Kubernetes API server. Every namespace has a default ServiceAccount with the name `default` used to identify any Pod that does not define an associated ServiceAccount.

Each ServiceAccount has a JWT associated with it that is fully managed by the Kubernetes backend. A Pod's associated ServiceAccount's token is automatically mounted into the filesystem of each Pod. Example 26-3 shows the relevant part of a Pod specification that Kubernetes has automatically added for every Pod created.

Example 26-3. ServiceAccount token mounted as a file for a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: random
spec:
  serviceAccountName: default          ❶
  containers:
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount ❷
      name: kube-api-access-vzfp7      ❸
      readOnly: true
  ...
  volumes:
  - name: kube-api-access-vzfp7
    projected:                         ❹
      defaultMode: 420
      sources:
      - serviceAccountToken:
          expirationSeconds: 3600      ❺
          path: token                  ❻
    ...
```

❶ `serviceAccountName` to set the name of the service account (`serviceAccount` is a deprecated alias for `serviceAccountName`).

❷ */var/run/secrets/kubernetes.io/serviceaccount* is the directory under which the service account token is mounted.

❸ Kubernetes assigns a random Pod-unique name to the auto-generated volume.

❹ A projected volume injects the ServiceAccount token directly into the filesystem.

❺ Expiration time of the token in seconds. After this time, the token expires, and the mounted token file is updated with a new token.

❻ The name of the file that will contain the token.

To view the mounted token, we can execute a `cat` on the mounted file in the running Pod, as shown in Example 26-4.

Example 26-4. Print out the service account JWT (output is shortened)

```
$ kubectl exec random -- \
    cat /var/run/secrets/kubernetes.io/serviceaccount/token
eyJhbGciOiJSUzI1NiIsImtpZCI6InVHYV9NZEVYOEZteUNUZFl...
```

In Example 26-3, the token is mounted into the Pod as a projected volume. Projected volumes allow you to merge multiple volume sources, such as Secret and ConfigMap volumes (described in Chapter 20, "Configuration Resource"), into a single directory. With this volume type, the ServiceAccount token can also be directly mapped into the Pod's filesystem using a `serviceAccountToken` subtype. This method has several benefits, including reducing the attack surface by eliminating the need for an intermediate representation of the token and by providing the ability to set an expiration time for the token, which the Kubernetes token controller will rotate after it expires. Furthermore, the token injected into the Pod will be valid only for the duration of the Pod's existence, further reducing the risk of unauthorized inspection of the service account's token.

Before Kubernetes 1.24, Secrets were used to represent these tokens and were mounted directly with a `secret` volume type, which had the disadvantage of long lifetimes and lack of rotation. Thanks to the new projected volume type, the token is available only to the Pod and is not exposed as an additional resource, which reduces the attack surface. You can still create a Secret manually to contain a ServiceAccount's token, as demonstrated in Example 26-5.

Example 26-5. Create a Secret for ServiceAccount `random-sa`

```
apiVersion: v1
kind: Secret
type: kubernetes.io/service-account-token              ❶
metadata:
  name: random-sa
  annotations:
    kubernetes.io/service-account.name: "random-sa"    ❷
```

❶ Special type to indicate that this Secret is about holding a ServiceAccount.

❷ Reference to ServiceAccount, whose token should be added.

Kubernetes will fill in the token and the public key for validation into the secret. Also, the lifecycle of this Secret is now bound to the ServiceAccount itself. If you delete the ServiceAccount, Kubernetes will also delete this secret.

The ServiceAccount resource has two additional fields for specifying credentials for pulling container images and defining the secrets allowed to be mounted:

Image pull secrets
: Image pull secrets allow a workload to authenticate with a private registry when pulling images. Typically, you would need to manually specify the pull secrets as part of the Pod specification in the fields `.spec.imagePullSecrets`. However, Kubernetes provides a shortcut by allowing you to attach a pull secret directly to a ServiceAccount in the top-level field `imagePullSecrets`. Every Pod associated with the ServiceAccount will automatically have the pull secrets injected into its specification when it is created. This automation eliminates the need to manually include the image pull secrets in the Pod specification every time a new Pod is created in the namespace, reducing the manual effort required.

Mountable secrets
: The `secrets` field in the ServiceAccount resource allows you to specify which secrets a Pod associated with the ServiceAccount can mount. You can enable this restriction by adding the `kubernetes.io/enforce-mountable-secrets` annotation to the ServiceAccount. If this annotation is set to `true`, only the Secrets listed will be allowed to be mounted by Pods associated with the ServiceAccount.

Groups

Both user and service accounts in Kubernetes can belong to one or more groups. Groups are attached to requests by the authentication system and are used to grant permissions to all group members. As seen in Example 26-1, group names are plain strings that represent the group name.

As mentioned earlier, groups can be freely defined and managed by the identity provider to create groups of subjects with the same permission model. A set of predefined groups in Kubernetes are also implicitly defined and have a `system:` prefix in their name. These predefined groups are listed in Table 26-2.

We will see how group names can be used in a RoleBinding to grant permissions to all group members in "RoleBinding" on page 267.

Table 26-2. System groups in Kubernetes

Group	Purpose
`system:unauthenticated`	Group assigned to every unauthenticated request
`system:authenticated`	Group assigned to an authenticated user
`system:masters`	Group whose members have unrestricted access to the Kubernetes API server
`system:serviceaccounts`	Group with all ServiceAccounts of the cluster
`system:serviceaccounts:<namespace>`	Group with all ServiceAccounts of this namespace

Now that you have a clear understanding of users, ServiceAccounts, and groups, let's examine how these subjects can be associated with Roles that define the actions they are allowed to perform against the Kubernetes API server.

Role-Based Access Control

In Kubernetes, Roles define the specific actions that a subject can perform on particular resources. You can then assign these Roles to subjects, such as users or service accounts, as described in "Subject" on page 257, through the use of RoleBindings. Roles and RoleBindings are Kubernetes resources that can be created and managed like any other resource. They are tied to a specific namespace and apply to its resources.

Figure 26-3 illustrates the relationship between subjects, Roles, and RoleBindings.

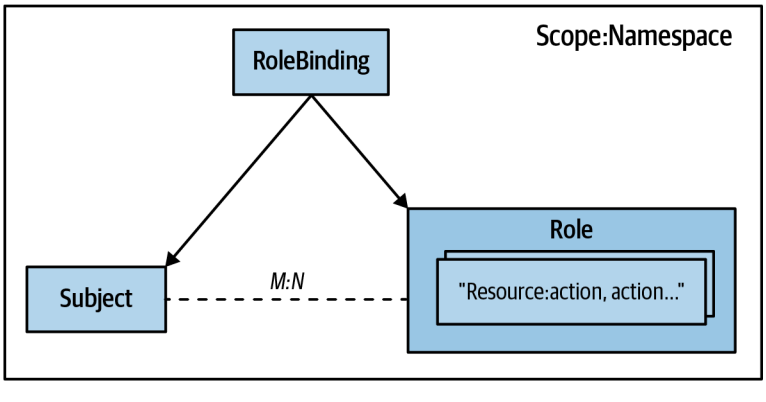


Figure 26-3. Relationship between Role, RoleBinding, and subjects

In Kubernetes RBAC, it is important to understand that there is a many-to-many relationship between subjects and Roles. This means that a single subject can have multiple Roles, and a single Role can be applied to multiple subjects. The relationship between a subject and a Role is established using a RoleBinding, which contains references to a list of subjects and a specific Role.

The RBAC concepts are best explained with a concrete example. Example 26-6 shows the definition of a Role in Kubernetes.

Example 26-6. Role for allowing access to core resources

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: developer-ro ❶
  namespace: default ❷
rules:
- apiGroups:
  - ""              ❸
  resources:        ❹
  - pods
  - services
  verbs:            ❺
  - get
  - list
  - watch
```

❶ The name of the Role, which is used to reference it.

❷ Namespace to which this Role applies. Roles are always connected to a namespace.

❸ An empty string indicates the core API group.

❹ List of Kubernetes core resources to which the rule applies.

❺ API actions are represented by verbs allowed by subjects associated with this Role.

The Role defined in Example 26-6 specifies that any user or service account associated with this Role can perform read-only operations on Pods and Services.

This Role can then be referenced in the RoleBinding shown in Example 26-7 to grant access to both the user, `alice`, and the ServiceAccount, `contractor`.

Example 26-7. RoleBinding specification

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: dev-rolebinding
subjects:           ❶
- kind: User        ❷
  name: alice
```

```
  apiGroup: "rbac.authorization.k8s.io"
- kind: ServiceAccount  ❸
  name: contractor
  apiGroup: ""
roleRef:
  kind: Role           ❹
  name: developer-ro
  apiGroup: rbac.authorization.k8s.io
```

❶ List of subjects to connect to a Role.

❷ Human user reference for a user named `alice`.

❸ Service account with name `contractor`.

❹ Reference to the Role with the name `developer-ro` that has been defined in Example 26-6.

Now that you have a basic understanding of the relationship between subjects, Roles, and RoleBindings, let's delve deeper into the specifics of Roles and RoleBindings.

Role

Roles in Kubernetes allow you to define a set of permitted actions for a group of Kubernetes resources or subresources. Typical activities on Kubernetes resources include the following:

- Getting Pods
- Deleting Secrets
- Updating ConfigMaps
- Creating ServiceAccounts

You have already seen a Role in Example 26-6. Besides metadata, such as names and namespaces, a Role definition consists of a list of rules that describe which resources can be accessed.

Only one rule must match a request to grant access to this Role. Three fields describe each `rule`:

apiGroups
: This list is used rather than a single value because wildcards can specify all resources of multiple API groups. For example, an empty string (`""`) is used for the core API group, which contains primary Kubernetes resources such as Pods and Services. A wildcard character (*) can match all available API groups the cluster is aware of.

resources
: This list specifies the resources that Kubernetes should grant access to. Each entry should belong to at least one of the configured `apiGroups`. A single `*` wildcard entry means all resources from all configured `apiGroups` are allowed.

verbs
: Allowed actions in a system are defined using verbs that are similar to HTTP methods. These verbs include CRUD operations on resources (CRUD stands for *Create-Read-Update-Delete* and describes the usual read-write operations that you can perform on persistent entities), and separate actions for operations on collections, such as `list` and `deletecollection`. Additionally, a `watch` verb allows access to resource change events and is separate from directly reading the resource with `get`. This `watch` verb is crucial for operators to receive notifications about the current status of resources they are managing. Chapter 27, "Controller", and Chapter 28, "Operator", has more on this topic. Table 26-3 lists the most common verbs. Using the `*` wildcard character is also possible to allow all operations on the configured resources for a given rule.

Table 26-3. Kubernetes verb mapping to HTTP request methods for CRUD operations

Verbs	HTTP request methods
get, watch, list	GET
create	POST
patch	PATCH
update	PUT
delete, delete collection	DELETE

Wildcard permissions make it easier to define all operations without listing each option individually. All of the properties of a Role's `rule` element allow for an `*` wildcard, which matches everything. Example 26-8 allows for all operations on all resources in the core and `networking.k8s.io` API group. If a wildcard is used, this list should have only this wildcard as its single entry.

Example 26-8. Wildcard permission for resources and permitted operations

```
rules:
- apiGroups:
  - ""
  - "networking.k8s.io"
  resources:
  - "*"  ❶
  verbs:
  - "*"  ❷
```

❶ All Resources in the listed API groups, core, and `networking.k8s.io`.

❷ All actions are allowed on those resources.

Wildcards help developers to configure rules quickly. But they come with the security risk of privilege escalation. Such broader privileges can cause security gaps and allow users to perform any operations that can compromise the Kubernetes cluster or cause unwanted changes.

Now that we have looked into the *what* (Roles) and *who* (subjects) of the Kubernetes RBAC model, let's have a closer look at how we can combine both concepts with RoleBindings.

RoleBinding

In Example 26-7, we saw how RoleBindings link one or more subjects to a given Role.

Each RoleBinding can connect a list of subjects to a Role. The `subjects` list field takes resource references as elements. Those resource references have a `name` field plus `kind` and `apiGroup` fields for defining the resource type to reference.

A subject in a RoleBinding can be one of the following types:

User
: A user is a human or system authenticated by the API server, as described in "Users" on page 257. User entries have a fixed `apiGroup` value of `rbac.authorization.k8s.io`.

Group
: A group is a collection of users, as explained in "Groups" on page 262. As for users, the group entries carry a `rbac.authorization.k8s.io` as `apiGroup`.

ServiceAccount
: We discussed ServiceAccount in depth in "Service accounts" on page 258. ServiceAccounts belong to the core API Group that is represented by an empty string (`""`). One unique aspect of ServiceAccounts is that it is the only subject type that can also carry a `namespace` field. This allows you to grant access to Pods from other namespaces.

Table 26-4 summarizes the possible field values for entries in a RoleBinding's `subject` list.

Table 26-4. Possible types for an element `subjects` list in a RoleBinding

Kind	API Group	Namespace	Description
User	rbac.authorization.k8s.io	N/A	`name` is a reference to a user.
Group	rbac.authorization.k8s.io	N/A	`name` is a reference to a group of users.
ServiceAccount	""	Optional	`name` is a reference to a ServiceAccount resource in the configured namespace.

The other end of a RoleBinding points to a single Role. This Role can either be a Role resource within the same namespace as the RoleBinding or a ClusterRole resource shared across multiple bindings in the cluster. ClusterRoles are described in detail in "ClusterRole" on page 269.

Similar to the subjects list, Role references are specified by `name`, `kind`, and `apiGroup`. Table 26-5 shows the possible values for the `roleRef` field.

Table 26-5. Possible types for a `roleRef` field in a RoleBinding

Kind	API Group	Description
Role	rbac.authorization.k8s.io	`name` is a reference to a Role in the same namespace.
ClusterRole	rbac.authorization.k8s.io	`name` is a reference to cluster-wide ClusterRole.

Privilege-Escalation Prevention

The RBAC subsystem is responsible for managing Roles and RoleBindings (as well as ClusterRoles and ClusterRoleBindings). To prevent privilege escalation, in which users with permissions to control the RBAC resource elevate their permissions, the following restrictions apply:

- Users can update a Role only if they already have all the permissions in that Role or if they have permission to use the `escalate` verb on all resources in the `rbac.authorization.k8s` API group.
- For RoleBindings, a similar restriction applies: users must have all the permissions granted in the referenced Role, or they must have the `bind` verb allowance on the RBAC resources.

More information about these restrictions and how they help prevent privilege escalation can be found in the Kubernetes documentation "Privilege Escalation Prevention and Bootstrapping".

ClusterRole

ClusterRoles in Kubernetes are similar to regular Roles but are applied cluster-wide rather than to a specific namespace. They have two primary uses:

- Securing cluster-wide resources such as CustomResourceDefinitions or StorageClasses. These resources are typically managed at the cluster-admin level and require additional access control. For example, developers may have read access to these resources but need help writing to them. ClusterRoleBindings are used to grant subjects access to cluster-wide resources.
- Defining typical Roles that are shared across namespaces. As we saw in "RoleBinding" on page 267, RoleBindings can refer only to Roles defined in the same namespace. ClusterRoles allow you to define general-access control Roles (e.g., "view" for read-only access to all resources) that can be used in multiple RoleBindings.

Example 26-9 shows a ClusterRole that can be reused in multiple RoleBindings. It has the same schema as a Role except that it ignores any `.meta.namespace` field.

Example 26-9. ClusterRole

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: view-pod ❶
rules:
- apiGroups:     ❷
  - ""
  resources:
  - pods
  verbs:
  - get
  - list
```

❶ Name of the ClusterRole but no namespace declaration.

❷ Rule that allows reading operations on all Pods.

Figure 26-4 shows how a single ClusterRole can be shared across multiple RoleBindings in different namespaces. In this example, the ClusterRole allows the reading of Pods in the `dev-1` and `dev-2` namespaces by a ServiceAccount in the `test` namespace.

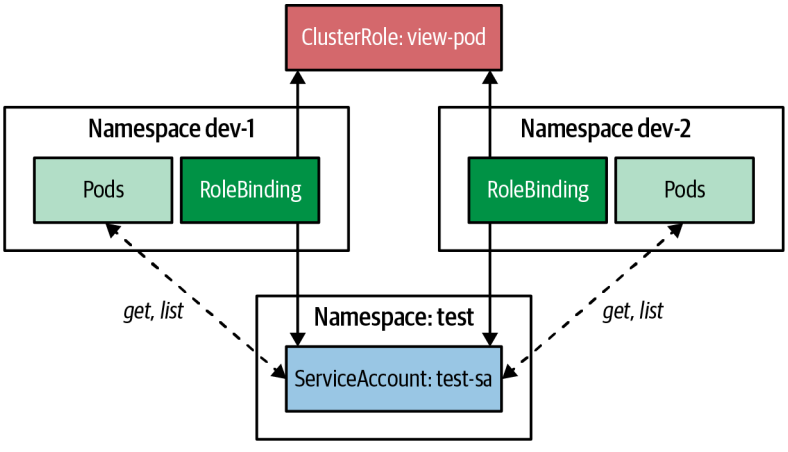


Figure 26-4. Sharing a ClusterRole in multiple namespaces

Using a single ClusterRole in multiple RoleBindings allows you to create typical access-control schemes that can be easily reused. For example, Table 26-6 includes a selection of useful user-facing ClusterRoles that Kubernetes provides out of the box. You can view the complete list of ClusterRoles available in a Kubernetes cluster using the `kubectl get clusterroles` command, or refer to the Kubernetes documentation for a list of default ClusterRoles.

Table 26-6. Standard user-facing ClusterRoles

ClusterRole	Purpose
`view`	Allows reading for most resources in a namespace, except Role, RoleBinding, and Secret
`edit`	Allows reading and modifying most resources in a namespace, except Role and RoleBinding
`admin`	Grants full control of all resources in a namespace, including Role and RoleBinding
`cluster-admin`	Grants full control of all namespace resources, including cluster-wide resources

Sometimes you may need to combine the permissions defined in two ClusterRoles. One way to do this is to create multiple RoleBindings that refer to both ClusterRoles. However, there is a more elegant way to achieve this using aggregation.

To use aggregation, you can define a ClusterRole with an empty `rules` field and a populated `aggregationRule` field containing a list of label selectors. Then, the rules defined by every other ClusterRole that has labels matching these selectors will be combined and used to populate the `rules` field of the aggregated ClusterRole.

When you set the `aggregationRule` field, you are handing ownership of the `rules` field over to Kubernetes, which will fully manage it. Therefore, any manual changes to the rules field will be constantly overwritten with the aggregated rules from the selected ClusterRoles in the `aggregationRule`.

This aggregation technique allows you to dynamically and elegantly build up large rule sets by combining smaller, more focused ClusterRoles.

Example 26-10 shows how the default `view` role uses aggregation to pick up more specific ClusterRoles labeled with `rbac.authorization.k8s.io/aggregate-to-view`. The `view` role itself also has the label `rbac.authorization.k8s.io/aggregate-to-edit`, which is used by the `edit` role to include the aggregated rules from the `view` ClusterRole.

Example 26-10. Aggregated ClusterRole

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: view
  labels:
    rbac.authorization.k8s.io/aggregate-to-edit: "true"    ❶
aggregationRule:
  clusterRoleSelectors:
  - matchLabels:
      rbac.authorization.k8s.io/aggregate-to-view: "true" ❷
rules: []  ❸
```

❶ This label exposes the ClusterRole as eligible for inclusion in the `edit` role.

❷ All ClusterRoles that match this selector will be picked up for the `view` ClusterRole. Note that this ClusterRole declaration does not need to be changed if you want to add additional permissions to the `view` ClusterRole—you can create a new ClusterRole with the appropriate label.

❸ The `rules` field will be managed by Kubernetes and populated with the aggregated rules.

This technique allows you to quickly compose more specialized ClusterRoles by aggregating a set of basic ClusterRoles. Example 26-10 also demonstrates how aggregation can be nested to build an inheritance chain of permission rule sets.

Since all of the user-facing default ClusterRoles use this aggregation technique, you can quickly hook into the permission model of custom resources (as described in

Chapter 28, "Operator") by simply adding the aggregation-triggering labels of the standard ClusterRoles (e.g., `view`, `edit`, and `admin`).

Now that we've covered the creation of a flexible and reusable permission model using ClusterRoles and RoleBindings, the final piece of the puzzle is establishing cluster-wide access rules with ClusterRoleBindings.

ClusterRoleBinding

The schema for a ClusterRoleBinding is similar to that of a RoleBinding, except that it ignores the `namespace` field. The rules defined in a ClusterRoleBinding apply to all namespaces in the cluster.

Example 26-11 shows a ClusterRoleBinding that connects a ServiceAccount `test-sa` with the ClusterRole `view-pod` defined in Example 26-9.

Example 26-11. ClusterRoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: test-sa-crb
subjects:                ❶
- kind: ServiceAccount
  name: test-sa
  namespace: test
roleRef:                 ❷
  kind: ClusterRole
  name: view-pod
  apiGroup: rbac.authorization.k8s.io
```

❶ Connects ServiceAccount `test-sa` from the `test` namespace.

❷ Allows the rules from the ClusterRole `view-pod` for every namespace.

The rules defined in the ClusterRole `view-pod` apply to all namespaces in the cluster so that any Pod associated with the ServiceAccount `test-sa` can read all Pods in every namespace, which is illustrated in Figure 26-5. However, it is crucial to use ClusterRoleBindings with caution, as they grant wide-ranging permissions across the entire cluster. Therefore, it is recommended that you carefully consider whether using a ClusterRoleBinding is necessary.

Using a ClusterRoleBinding may be convenient as it automatically grants permissions to newly created namespaces. However, using individual RoleBindings per namespace is generally better for more granular control over permissions. This extra effort allows you to omit specific namespaces, such as `kube-system`, from unauthorized access.

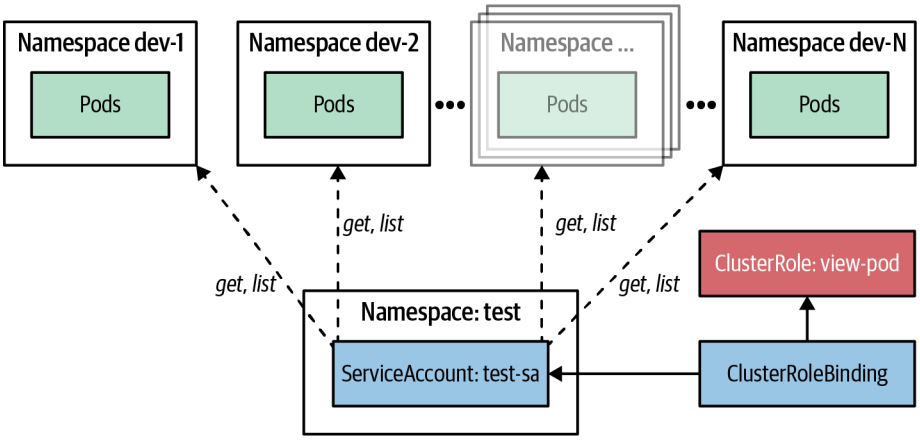


Figure 26-5. ClusterRoleBinding for reading all Pods

ClusterRoleBindings should be used only for administrative tasks, such as managing cluster-wide resources like Nodes, Namespaces, CustomResourceDefinitions, or even ClusterRoleBindings.

These final warnings conclude our tour through the world of Kubernetes RBAC. This machinery is mighty, but it's also complex to understand and sometimes even more complicated to debug. The following sidebar gives you some tips for better understanding a given RBAC setup.

Debugging RBAC Rules

In a Kubernetes cluster, many RBAC objects define the overall security model for accessing the API server. Understanding the authorization decisions made by the Kubernetes API server can be challenging, but the Access Review API can help by allowing you to query the authorization subsystem for permissions.

One way to use this API is through the `kubectl auth can-i` command. For example, you can use it to check whether a ServiceAccount named `test-sa` in the `test` namespace has permission to list all pods in the `dev-1` namespace. The command would look like Example 26-12. This command will return a simple "yes" or "no" indicating whether the ServiceAccount has the specified permission.

Example 26-12. Check access permissions with `kubectl`

```
kubectl auth can-i \
    list pods --namespace dev-1 --as system:serviceaccount:test:test-sa
```

Behind the scenes, a resource of the type SubjectAccessReview is created, and the Kubernetes authorization controller updates the `status` section of this resource with

the result of the authorization check. You can read more about this API in the Kubernetes RBAC documentation.

While `kubectl auth can-i` helps check specific permissions, it can be tedious and does not provide a comprehensive overview of a subject's permissions across the cluster. To better understand what actions a subject can perform on all resources, tools like rakkess can be helpful. Rakkess is available as a `kubectl` plugin and can be run with the command `kubectl access-matrix`. It provides a matrix view of the actions a subject can perform on specific resources.

Another tool to help visualize and verify the application of fine-grained permissions is KubiScan, which allows you to scan a Kubernetes cluster for risky permissions in the RBAC configuration.

The final section will discuss some general tips for properly using Kubernetes RBAC.

Discussion

Kubernetes RBAC is a powerful tool for controlling access to API resources. However, it can be challenging to understand which definition objects to use and how to combine them to fit a particular security setup. Here are some guidelines to help you navigate these decisions:

- If you want to secure resources in a specific namespace, use a Role with a RoleBinding that connects to a user or ServiceAccount. The ServiceAccount does not have to be in the same namespace, allowing you to grant access to Pods from other namespaces.
- If you want to reuse the same access rules in multiple namespaces, use a RoleBinding with a ClusterRole that defines these shared-access rules.
- If you want to extend one or more existing predefined ClusterRoles, create a new ClusterRole with an `aggregationRule` field that refers to the ClusterRoles you wish to extend, and add your permissions to the `rules` field.
- If you want to grant a user or ServiceAccount access to all resources of a specific kind in all namespaces, use a ClusterRole and a ClusterRoleBinding.
- If you want to manage access to a cluster-wide resource like a CustomResourceDefinition, use a ClusterRole and a ClusterRoleBinding.

We have seen how RBAC allows us to define fine-grained permissions and manage them. It can reduce risk by ensuring the applied permission does not leave gaps for the escalation path. On the other hand, defining any broad open permissions can lead to security escalations. Let's close this chapter with a summary of some general RBAC advice:

Avoid wildcard permissions
: We recommend following the principle of least privilege when composing the fine-grained access control in the Kubernetes cluster. To avoid unintentional operations, avoid wildcard permissions when defining the Role and ClusterRoles. For rare occasions, it might make sense to use wildcards (i.e., to secure all resources of an API group), but it is a good practice to establish a general "no wildcard" policy that could be relaxed for well-reasoned exceptions.

Avoid `cluster-admin` *ClusterRole*
: ServiceAccounts with high privileges can allow you to perform actions over any resources, like modifying permissions or viewing secrets in any namespace, which can lead to severe security implications. Therefore, never assign the `cluster-admin` ClusterRole to a Pod. Never.

Don't automount ServiceAccount tokens
: By default, tokens of ServiceAccounts are mounted within a container's file-system at */var/run/secrets/kubernetes.io/serviceaccount/token*. If such a Pod gets compromised, any attacker can talk with the API server with the permissions of the Pod's associated ServiceAccount. However, many applications don't need that token for business operations. For such a use case, avoid the token mount by setting the ServiceAccount's field `automountServiceAccountToken` to `false`.

Kubernetes RBAC is a flexible and powerful method for controlling access to the Kubernetes API. Therefore, even if your application is not directly interacting with the API Server to install your application and connect it to other Kubernetes servers, *Access Control* is a valuable pattern to secure the operation of your application.

More Information

- Access Control Example
- Escalation Paths
- Controlling Access to the Kubernetes API
- Auditing
- Admission Controllers Reference
- Dynamic Admission Control
- Kubernetes: Authentication Strategies
- RBAC Good Practices
- Workload Creation
- Bound Service Account Tokens
- BIG Change in K8s 1.24 About ServiceAccounts and Their Secrets

- Efficient Detection of Changes
- Add ImagePullSecrets to a Service Account
- RBAC Dev
- Rakkess
- How the Basics of Kubernetes Auth Scale for Organizations
- Kubernetes CVE-2020-8559 Proof of Concept PoC Exploit
- OAuth Is Not Authentication

PART VI
Advanced Patterns

The patterns in this category cover more complex topics that do not fit in any of the other categories. Some of the patterns here such as *Controller* or *Operator* are timeless, and Kubernetes itself is built on them. However, some of the other pattern implementations are still evolving. To keep up with this, we will keep our online examples up to date and reflect the latest developments in this space.

In the following chapters, we explore these advanced patterns:

- Chapter 27, "Controller", is essential to Kubernetes itself and shows how custom controllers can extend the platform.
- Chapter 28, "Operator", combines a controller with custom domain-specific resources to encapsulate operational knowledge in an automated form.
- Chapter 29, "Elastic Scale", describes how Kubernetes can handle dynamic loads by scaling in various dimensions.
- Chapter 30, "Image Builder", moves the aspect of building application images onto the cluster itself.

CHAPTER 27
Controller

A controller actively monitors and maintains a set of Kubernetes resources in a desired state. The heart of Kubernetes itself consists of a fleet of controllers that regularly watch and reconcile the current state of applications with the declared target state. In this chapter, we see how to leverage this *Controller* pattern to extend the platform for our needs.

Problem

You've already seen that Kubernetes is a sophisticated and comprehensive platform that provides many features out of the box. However, it is a general-purpose orchestration platform that does not cover all application use cases. Luckily, it provides natural extension points where specific use cases can be implemented elegantly on top of proven Kubernetes building blocks.

The main questions that arise here are how to extend Kubernetes without changing and breaking it and how to use its capabilities for custom use cases.

By design, Kubernetes is based on a declarative resource-centric API. What exactly do we mean by *declarative*? As opposed to an *imperative* approach, a declarative approach does not tell Kubernetes how it should act but instead describes how the target state should look. For example, when we scale up a Deployment, we do not actively create new Pods by telling Kubernetes to "create a new Pod." Instead, we change the Deployment resource's `replicas` property via the Kubernetes API to the desired number.

So, how are the new Pods created? This is done internally by the controllers. For every change in the resource status (like changing the `replicas` property value of a Deployment), Kubernetes creates an event and broadcasts it to all interested listeners. These listeners can then react by modifying, deleting, or creating new resources,

which in turn creates other events, like Pod-created events. These events are then potentially picked up again by other controllers, which perform their specific actions.

The whole process is also known as *state reconciliation*, where a target state (the number of desired replicas) differs from the current state (the actual running instances), and it is the task of a controller to reconcile and reach the desired target state again. When looked at from this angle, Kubernetes essentially represents a distributed state manager. You give it the desired state for a component instance, and it attempts to maintain that state should anything change.

How can we now hook into this reconciliation process without modifying Kubernetes code and create a controller customized for our specific needs?

Solution

Kubernetes comes with a collection of built-in controllers that manage standard Kubernetes resources like ReplicaSets, DaemonSets, StatefulSets, Deployments, or Services. These controllers run as part of the controller manager, which is deployed (as a standalone process or a Pod) on the control plane node. These controllers are not aware of one another. They run in an endless reconciliation loop, to monitor their resources for the actual and desired state and to act accordingly to get the actual state closer to the desired state.

However, in addition to these out-of-the-box controllers, the Kubernetes event-driven architecture allows us to natively plug in other custom controllers. Custom controllers can add extra functionality to the behavior by reacting to state-changing events, the same way that internal controllers do. A common characteristic of controllers is that they are reactive and react to events in the system to perform their specific actions. At a high level, this reconciliation process consists of the following main steps:

Observe
: Discover the actual state by watching for events issued by Kubernetes when an observed resource changes.

Analyze
: Determine the differences from the desired state.

Act
: Perform operations to drive the actual state to the desired state.

For example, the ReplicaSet controller watches for ReplicaSet resource changes, analyzes how many Pods need to be running, and acts by submitting Pod definitions to the API Server. The Kubernetes backend is then responsible for starting up the requested Pod on a node.

Figure 27-1 shows how a controller registers itself as an event listener for detecting changes on the managed resources. It observes the current state and changes it by calling out to the API Server to get closer to the target state (if necessary).

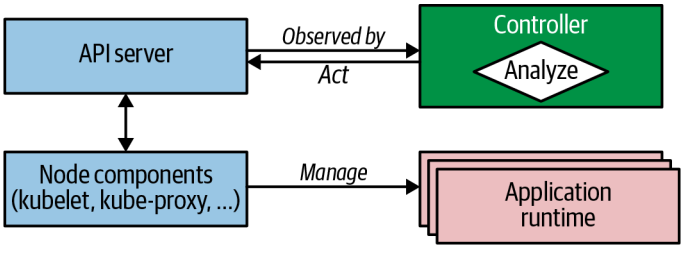


Figure 27-1. Observe-Analyze-Act cycle

Controllers are part of the Kubernetes control plane, and it became clear early on that they would also allow you to extend the platform with custom behavior. Moreover, they have become the standard mechanism for extending the platform and enable complex application lifecycle management. And as a result, a new generation of more sophisticated controllers was born, called *Operators*. From an evolutionary and complexity point of view, we can classify the active reconciliation components into two groups:

Controllers
: A simple reconciliation process that monitors and acts on standard Kubernetes resources. More often, these controllers enhance platform behavior and add new platform features.

Operators
: A sophisticated reconciliation process that interacts with CustomResourceDefinitions (CRDs), which are at the heart of the *Operator* pattern. Typically, these operators encapsulate complex application domain logic and manage the full application lifecycle.

As stated previously, these classifications help introduce new concepts gradually. Here, we focus on the simpler controllers, and in Chapter 28, we introduce CRDs and build up to the *Operator* pattern.

To avoid having multiple controllers acting on the same resources simultaneously, controllers use the *Singleton Service* pattern explained in Chapter 10. Most controllers are deployed just as Deployments but with one replica, as Kubernetes uses optimistic locking at the resource level to prevent concurrency issues when changing resource objects. In the end, a controller is nothing more than an application that runs permanently in the background.

Because Kubernetes itself is written in Go, and a complete client library for accessing Kubernetes is also written in Go, many controllers are written in Go too. However, you can write controllers in any programming language by sending requests to the Kubernetes API Server. We see a controller written in a pure shell script later in Example 27-1.

The most straightforward kind of controllers extend the way Kubernetes manages its resources. They operate on the same standard resources and perform similar tasks as the Kubernetes internal controllers operating on the standard Kubernetes resources, but they are invisible to the user of the cluster. Controllers evaluate resource definitions and conditionally perform some actions. Although they can monitor and act upon any field in the resource definition, metadata and ConfigMaps are most suitable for this purpose. The following are a few considerations to keep in mind when choosing where to store controller data:

Labels
: Labels as part of a resource's metadata can be watched by any controller. They are indexed in the backend database and can be efficiently searched for in queries. We should use labels when a selector-like functionality is required (e.g., to match Pods of a Service or a Deployment). A limitation of labels is that only alphanumeric names and values with restrictions can be used. See the Kubernetes documentation for which syntax and character sets are allowed for labels.

Annotations
: Annotations are an excellent alternative to labels. They have to be used instead of labels if the values do not conform to the syntax restrictions of label values. Annotations are not indexed, so we use annotations for nonidentifying information not used as keys in controller queries. Preferring annotations over labels for arbitrary metadata also has the advantage that it does not negatively impact the internal Kubernetes performance.

ConfigMaps
: Sometimes controllers need additional information that does not fit well into labels or annotations. In this case, ConfigMaps can be used to hold the target state definition. These ConfigMaps are then watched and read by the controllers. However, CRDs are much better suited for designing the custom target state specification and are recommended over plain ConfigMaps. For registering CRDs, however, you need elevated cluster-level permissions. If you don't have these, ConfigMaps are still the best alternative to CRDs. We will explain CRDs in detail in Chapter 28, "Operator".

Here are a few reasonably simple example controllers you can study as a sample implementation of this pattern:

jenkins-x/exposecontroller
: This controller watches Service definitions, and if it detects an annotation named `expose` in the metadata, the controller automatically exposes an Ingress object for external access of the Service. It also removes the Ingress object when someone removes the Service. This project is now archived but still serves as a good example of implementing a simple controller.

stakater/Reloader
: This is a controller that watches ConfigMap and Secret objects for changes and performs rolling upgrades of their associated workloads, which can be Deployment, DaemonSet, StatefulSet and other workload resources. We can use this controller with applications that are not capable of watching the ConfigMap and updating themselves with new configurations dynamically. That is particularly true when a Pod consumes this ConfigMap as environment variables or when your application cannot quickly and reliably update itself on the fly without a restart. As a proof of concept, we implement a similar controller with a plain shell script in Example 27-2.

Flatcar Linux Update Operator
: This is a controller that reboots a Kubernetes node running on Flatcar Container Linux when it detects a particular annotation on the Node resource object.

Now let's take a look at a concrete example: a controller that consists of a single shell script and that watches the Kubernetes API for changes on ConfigMap resources. If we annotate such a ConfigMap with `k8spatterns.io/podDeleteSelector`, all Pods selected with the given label selector are deleted when the ConfigMap changes. Assuming we back these Pods with a high-order resource like Deployment or ReplicaSet, these Pods are restarted and pick up the changed configuration.

For example, the following ConfigMap would be monitored by our controller for changes and would restart all Pods that have a label `app` with value `webapp`. The ConfigMap in Example 27-1 is used in our web application to provide a welcome message.

Example 27-1. ConfigMap use by web application

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: webapp-config
  annotations:
    k8spatterns.io/podDeleteSelector: "app=webapp"  ❶
data:
  message: "Welcome to Kubernetes Patterns !"
```

❶ Annotation used as selector for the controller in Example 27-2 to find the application Pods to restart.

Our controller shell script now evaluates this ConfigMap. You can find the source in its full glory in our Git repository. In short, the controller starts a *hanging GET* HTTP request for opening an endless HTTP response stream to observe the lifecycle events pushed by the API Server to us. These events are in the form of plain JSON objects, which are then analyzed to detect whether a changed ConfigMap carries our annotation. As events arrive, the controller acts by deleting all Pods matching the selector provided as the value of the annotation. Let's have a closer look at how the controller works.

The main part of this controller is the reconciliation loop, which listens on ConfigMap lifecycle events, as shown in Example 27-2.

Example 27-2. Controller script

```
namespace=${WATCH_NAMESPACE:-default}  ❶

base=http://localhost:8001             ❷
ns=namespaces/$namespace

curl -N -s $base/api/v1/${ns}/configmaps?watch=true | \
while read -r event                    ❸
do
   # ...
done
```

❶ Namespace to watch (or *default* if not given).

❷ Access to the Kubernetes API via a proxy running in the same Pod.

❸ Loop with watches for events on ConfigMaps.

The environment variable `WATCH_NAMESPACE` specifies the namespace in which the controller should watch for ConfigMap updates. We can set this variable in the Deployment descriptor of the controller itself. In our example, we're using the Downward API described in Chapter 14, "Self Awareness", to monitor the namespace in which we have deployed the controller as configured in Example 27-3 as part of the controller Deployment.

Example 27-3. `WATCH_NAMESPACE` extracted from the current namespace

```
env:
 - name: WATCH_NAMESPACE
   valueFrom:
     fieldRef:
       fieldPath: metadata.namespace
```

With this namespace, the controller script constructs the URL to the Kubernetes API endpoint to watch the ConfigMaps.

Note the `watch=true` query parameter in Example 27-2. This parameter indicates to the API Server not to close the HTTP connection but to send events along the response channel as soon as they happen (*hanging GET* or *Comet* are other names for this kind of technique). The loop reads every individual event as it arrives as a single item to process.

As you can see, our controller contacts the Kubernetes API Server via localhost. We won't deploy this script directly on the Kubernetes API control plane node, but then how can we use localhost in the script? As you may have probably guessed, another pattern kicks in here. We deploy this script in a Pod together with an ambassador container that exposes port 8001 on localhost and proxies it to the real Kubernetes Service. See Chapter 18 for more details on the *Ambassador* pattern. We see the actual Pod definition with this ambassador in detail later in this chapter.

Watching events this way is not very robust, of course. The connection can stop anytime, so there should be a way to restart the loop. Also, one could miss events, so production-grade controllers should not only watch on events but from time to time should also query the API Server for the entire current state and use that as the new base. For the sake of demonstrating the pattern, this is good enough.

Within the loop, the logic shown in Example 27-4 is performed.

Example 27-4. Controller reconciliation loop

```
curl -N -s $base/api/v1/${ns}/configmaps?watch=true | \
while read -r event
do
  type=$(echo "$event"        | jq -r '.type')                    ❶
  config_map=$(echo "$event"  | jq -r '.object.metadata.name')
  annotations=$(echo "$event" | jq -r '.object.metadata.annotations')

  if [ "$annotations" != "null" ]; then
    selector=$(echo $annotations | \                              ❷
     jq -r "\
        to_entries                                          |\
```

```
        .[]                                                    |\
        select(.key == \"k8spatterns.io/podDeleteSelector\") |\
        .value                                                 |\
         @uri                                                   \
    ")
  fi

  if [ $type = "MODIFIED" ] && [ -n "$selector" ]; then          ❸
    pods=$(curl -s $base/api/v1/${ns}/pods?labelSelector=$selector |\
           jq -r .items[].metadata.name)

    for pod in $pods; do                                         ❹
      curl -s -X DELETE $base/api/v1/${ns}/pods/$pod
    done
  fi
done
```

❶ Extract the type and name of the ConfigMap from the event.

❷ Extract all annotations on the ConfigMap with the key `k8spatterns.io/podDeleteSelector`. See the following sidebar for an explanation of this `jq` expression.

❸ If the event indicates an update of the ConfigMap and our annotation is attached, then find all Pods matching this label selector.

❹ Delete all Pods that match the selector.

First, the script extracts the event type that specifies what action happened to the ConfigMap. Then, we derive the annotations with `jq`. `jq` is an excellent tool for parsing JSON documents from the command line, and the script assumes it is available in the container the script is running in.

If the ConfigMap has annotations, we check for the annotation `k8spatterns.io/podDeleteSelector` by using a more complex `jq` query. The purpose of this query is to convert the annotation value to a Pod selector that can be used in an API query option in the next step: an annotation `k8spatterns.io/podDeleteSelector: "app=webapp"` is transformed to `app%3Dwebapp` that is used as a Pod selector. This conversion is performed with `jq` and is explained next if you are interested in how this extraction works.

If the script can extract a `selector`, we can now use it directly to select the Pods to delete. First, we look up all Pods that match the selector, and then we delete them one by one with direct API calls.

This shell script-based controller is, of course, not production-grade (e.g., the event loop can stop any time), but it nicely reveals the base concepts without too much boilerplate code for us.

Some jq Fu

Extracting the ConfigMap's `k8spatterns.io/podDeleteSelector` annotation value and converting it to a Pod selector is performed with `jq`. This is an excellent JSON command-line tool, but some concepts can be a bit confusing. Let's have a close look at how the expressions work in detail:

```
selector=$(echo $annotations | \
   jq -r "\
    to_entries                                          |\
    .[]                                                 |\
    select(.key == \"k8spatterns.io/podDeleteSelector\") |\
    .value                                              |\
     @uri                                                \
")
```

- `$annotations` holds all annotations as a JSON object, with annotation names as properties.
- With `to_entries`, we convert a JSON object like `{ "a": "b"}` into an array with entries like `{ "key": "a", "value": "b" }`. See the jq documentation for more details.
- `.[]` selects the array entries individually.
- From these entries, we pick only the ones with the matching key. There can be only zero or one matches that survive this filter.
- Finally, we extract the value (`.value`) and convert it with `@uri` so that it can be used as part of a URI.

This expression converts a JSON structure such as

```
{
  "k8spatterns.io/pattern": "Controller",
  "k8spatterns.io/podDeleteSelector": "app=webapp"
}
```

to a selector, `app%3Dwebapp`.

The remaining work is about creating resource objects and container images. The controller script itself is stored in a ConfigMap `config-watcher-controller`, and can be easily edited later if required.

We use a Deployment to create a Pod for our controller with two containers:

- One Kubernetes API ambassador container that exposes the Kubernetes API on localhost on port 8001. The image `k8spatterns/kubeapi-proxy` is an Alpine Linux with a local `kubectl` installed and `kubectl proxy` started with the proper

CA and token mounted. The original version, kubectl-proxy, was written by Marko Lukša, who introduced this proxy in *Kubernetes in Action*.

- The main container that executes the script contained in the just-created ConfigMap. Here, we use an Alpine base image with `curl` and `jq` installed.

You can find the Dockerfiles for the `k8spatterns/kubeapi-proxy` and `k8spatterns/curl-jq` images in the example Git repository.

Now that we have the images for our Pod, the final step is to deploy the controller by using a Deployment. We can see the main parts of the Deployment in Example 27-5 (the full version is available in our example repository).

Example 27-5. Controller Deployment

```
apiVersion: apps/v1
kind: Deployment
# ....
spec:
  template:
    # ...
    spec:
      serviceAccountName: config-watcher-controller ❶
      containers:
      - name: kubeapi-proxy                         ❷
        image: k8spatterns/kubeapi-proxy
      - name: config-watcher                        ❸
        image: k8spatterns/curl-jq
        # ...
        command:                                    ❹
        - "sh"
        - "/watcher/config-watcher-controller.sh"
        volumeMounts:                               ❺
        - mountPath: "/watcher"
          name: config-watcher-controller
      volumes:
      - name: config-watcher-controller             ❻
        configMap:
          name: config-watcher-controller
```

❶ ServiceAccount with proper permissions for watching events and restarting Pods.

❷ Ambassador container for proxying localhost to the Kubeserver API.

❸ Main container holding all tools and mounting the controller script.

❹ Startup command calling the controller script.

❺ Volume mapped to the ConfigMap holding our script.

❻ Mount of the ConfigMap-backed volume into the main Pod.

As you can see, we mount the `config-watcher-controller-script` from the ConfigMap we created previously and directly use it as the startup command for the primary container. For simplicity, we omitted any liveness and readiness checks as well as resource limit declarations. Also, we need a ServiceAccount `config-watcher-controller` that is allowed to monitor ConfigMaps. Refer to the example repository for the full security setup.

Let's see the controller in action. For this, we are using a straightforward web server, which serves the value of an environment variable as the only content. The base image uses plain `nc` (netcat) for serving the content. You can find the Dockerfile for this image in the example repository. We deploy the HTTP server with a ConfigMap and Deployment, as is sketched in Example 27-6.

Example 27-6. Sample web app with Deployment and ConfigMap

```
apiVersion: v1
kind: ConfigMap                                          ❶
metadata:
  name: webapp-config
  annotations:
    k8spatterns.io/podDeleteSelector: "app=webapp"       ❷
data:
  message: "Welcome to Kubernetes Patterns !"            ❸
---
apiVersion: apps/v1
kind: Deployment                                         ❹
# ...
spec:
  # ...
  template:
    spec:
      containers:
      - name: app
        image: k8spatterns/mini-http-server              ❺
        ports:
        - containerPort: 8080
        env:
        - name: MESSAGE                                  ❻
          valueFrom:
            configMapKeyRef:
              name: webapp-config
              key: message
```

❶ ConfigMap for holding the data to serve.

❷ Annotation that triggers a restart of the web app's Pod.

❸ Message used in web app in HTTP responses.

❹ Deployment for the web app.

❺ Simplistic image for HTTP serving with netcat.

❻ Environment variable used as an HTTP response body and fetched from the watched ConfigMap.

This concludes our example of our ConfigMap controller implemented in a plain shell script. Although this is probably the most complex example in this book, it also shows that it does not take much to write a basic controller.

Obviously, for real-world scenarios, you would write this sort of controller in a real programming language that provides better error-handling capabilities and other advanced features.

Discussion

To sum up, a controller is an active reconciliation process that monitors objects of interest for the world's desired state and the world's actual state. Then, it sends instructions to try to change the world's current state to be more like the desired state. Kubernetes uses this mechanism with its internal controllers, and you can also reuse the same mechanism with custom controllers. We demonstrated what is involved in writing a custom controller and how it functions and extends the Kubernetes platform.

Controllers are possible because of the highly modular and event-driven nature of the Kubernetes architecture. This architecture naturally leads to a decoupled and asynchronous approach for controllers as extension points. The significant benefit here is that we have a precise technical boundary between Kubernetes itself and any extensions. However, one issue with the asynchronous nature of controllers is that they are often hard to debug because the flow of events is not always straightforward. As a consequence, you can't easily set breakpoints in your controller to stop everything to examine a specific situation.

In Chapter 28, you'll learn about the related *Operator* pattern, which builds on this *Controller* pattern and provides an even more flexible way to configure operations.

More Information

- Controller Example
- Writing Controllers
- Writing a Kubernetes Controller
- A Deep Dive into Kubernetes Controllers
- Expose Controller
- Reloader: ConfigMap Controller
- Writing a Custom Controller: Extending the Functionality of Your Cluster
- Writing Kubernetes Custom Controllers
- Contour Ingress Controller
- Syntax and Character Set
- Kubectl-Proxy

CHAPTER 28
Operator

An operator is a controller that uses a CRD to encapsulate operational knowledge for a specific application in an algorithmic and automated form. The *Operator* pattern allows us to extend the *Controller* pattern from the preceding chapter for more flexibility and greater expressiveness.

Problem

You learned in Chapter 27, "Controller", how to extend the Kubernetes platform in a simple and decoupled way. However, for extended use cases, plain custom controllers are not powerful enough, as they are limited to watching and managing Kubernetes intrinsic resources only. Moreover, sometimes we want to add new concepts to the Kubernetes platform, which requires additional domain objects. For example, let's say we chose Prometheus as our monitoring solution and want to add it as a monitoring facility to Kubernetes in a well-defined way. Wouldn't it be wonderful to have a Prometheus resource describing our monitoring setup and all the deployment details, similar to how we define other Kubernetes resources? Moreover, could we have resources relating to services we have to monitor (e.g., with a label selector)?

These situations are precisely the kind of use cases where CustomResourceDefinition (CRD) resources are very helpful. They allow extensions of the Kubernetes API, by adding custom resources to your Kubernetes cluster and using them as if they were native resources. Custom resources, together with a controller acting on these resources, form the *Operator* pattern.

This quote by Jimmy Zelinskie probably describes the characteristics of operators best:

> An operator is a Kubernetes controller that understands two domains: Kubernetes and something else. By combining knowledge of both areas, it can automate tasks that usually require a human operator that understands both domains.

Solution

As you saw in Chapter 27, "Controller", we can efficiently react to state changes of default Kubernetes resources. Now that you understand one half of the *Operator* pattern, let's have a look at the other half—representing custom resources on Kubernetes using CRD resources.

Custom Resource Definitions

With a CRD, we can extend Kubernetes to manage our domain concepts on the Kubernetes platform. Custom resources are managed like any other resource, through the Kubernetes API, and are eventually stored in the backend store etcd.

The preceding scenario is actually implemented with these new custom resources by the CoreOS Prometheus operator to allow seamless integration of Prometheus to Kubernetes. The Prometheus CRD is defined in Example 28-1, which also explains most of the available fields for a CRD.

Example 28-1. CustomResourceDefinition

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: prometheuses.monitoring.coreos.com ❶
spec:
  group: monitoring.coreos.com             ❷
  names:
    kind: Prometheus                       ❸
    plural: prometheuses                   ❹
  scope: Namespaced                        ❺
  versions:                                ❻
  - name: v1                               ❼
    storage: true                          ❽
    served: true                           ❾
    schema:
      openAPIV3Schema: ....                ❿
```

❶ Name.

❷ API group it belongs to.

❸ Kind used to identify instances of this resource.

❹ Naming rule for creating the plural form, used for specifying a list of those objects.

❺ Scope—whether the resource can be created cluster-wide or is specific to a namespace.

❻ Versions available for this CRD.

❼ Name of a supported version.

❽ Exactly one version has to be the storage version used for storing the definition in the backend.

❾ Whether this version is served via the REST API.

❿ OpenAPI V3 schema for validation (not shown here).

An OpenAPI V3 schema can also be specified to allow Kubernetes to validate a custom resource. For simple use cases, this schema can be omitted, but for production-grade CRDs, the schema should be provided so that configuration errors can be detected early.

Additionally, Kubernetes allows us to specify two possible subresources for our CRD via the spec field `subresources`:[1]

scale
: With this property, a CRD can specify how it manages its replica count. This field can be used to declare the JSON path, where the number of desired replicas of this custom resource is specified: the path to the property that holds the actual number of running replicas and an optional path to a label selector that can be used to find copies of custom resource instances. This label selector is usually optional but is required if you want to use this custom resource with the HorizontalPodAutoscaler explained in Chapter 29, "Elastic Scale".

status
: When this property is set, a new API call becomes available that allows you to update only the `status` field of a resource. This API call can be secured individually and allows the operator to reflect the *actual* status of the resource, which might differ from the *declared* state in the `spec` field. When a custom

1 Kubernetes subresources are additional API endpoints that provide further functionality within a resource type.

resource is updated as a whole, any sent `status` section is ignored, as is the case with standard Kubernetes resources.

Example 28-2 shows a potential subresource path as is also used for a regular Pod.

Example 28-2. Subresource definition for a CustomResourceDefinition

```
kind: CustomResourceDefinition
# ...
spec:
  subresources:
    status: {}
    scale:
      specReplicasPath: .spec.replicas          ❶
      statusReplicasPath: .status.replicas      ❷
      labelSelectorPath: .status.labelSelector  ❸
```

❶ JSON path to the number of declared replicas.

❷ JSON path to the number of active replicas.

❸ JSON path to a label selector to query for the number of active replicas.

Once we define a CRD, we can easily create such a resource, as shown in Example 28-3.

Example 28-3. A Prometheus custom resource

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: prometheus
spec:
  serviceMonitorSelector:
    matchLabels:
      team: frontend
  resources:
    requests:
      memory: 400Mi
```

The `metadata` section has the same format and validation rules as any other Kubernetes resource. The `spec` contains the CRD-specific content, and Kubernetes validates against the given validation rule from the CRD.

Custom resources alone are not of much use without an active component to act on them. To give them some meaning, we need again our well-known controller, which watches the lifecycle of these resources and acts according to the declarations found within the resources.

Controller and Operator Classification

Before we dive into writing our operator, let's look at a few kinds of classifications for controllers, operators, and especially CRDs. Based on the operator's action, broadly, the classifications are as follows:

Installation CRDs
: Meant for installing and operating applications on the Kubernetes platform. Typical examples are the Prometheus CRDs, which we can use for installing and managing Prometheus itself.

Application CRDs
: In contrast, these are used to represent an application-specific domain concept. This kind of CRD allows applications deep integration with Kubernetes, which involves combining Kubernetes with an application-specific domain behavior. For example, the ServiceMonitor CRD is used by the Prometheus operator to register specific Kubernetes Services to be scraped by a Prometheus server. The Prometheus operator takes care of adapting the Prometheus server configuration accordingly.

Note that an operator can act on different kinds of CRDs as the Prometheus operator does in this case. The boundary between these two categories of CRDs is blurry.

In our categorization of controller and operator, an operator is-a controller that uses CRDs.[2] However, even this distinction is a bit fuzzy as there are variations in between.

One example is a controller, which uses a ConfigMap as a kind of replacement for a CRD. This approach makes sense in scenarios where default Kubernetes resources are not enough but creating CRDs is not feasible either. In this case, ConfigMap is an excellent middle ground, allowing encapsulation of domain logic within the content of a ConfigMap. An advantage of using a plain ConfigMap is that you don't need to have the cluster-admin rights you need when registering a CRD. In certain cluster setups, it is just not possible for you to register such a CRD (e.g., when running on public clusters like OpenShift Online).

However, you can still use the concept of *Observe-Analyze-Act* when you replace a CRD with a plain ConfigMap that you use as your domain-specific configuration. The drawback is that you don't get essential tool support like `kubectl get` for CRDs; you have no validation on the API Server level and no support for API versioning.

2 is-a emphasizes the inheritance relationship between operator and controller, that an operator has all characteristics of a controller plus a bit more.

Also, you don't have much influence on how you model the `status` field of a ConfigMap, whereas for a CRD, you are free to define your status model as you wish.[3]

Another advantage of CRDs is that you have a fine-grained permission model based on the kind of CRD, which you can tune individually, as is explained in Chapter 26, "Access Control". This kind of RBAC security is not possible when all your domain configuration is encapsulated in ConfigMaps, as all ConfigMaps in a namespace share the same permission setup.

From an implementation point of view, it matters whether we implement a controller by restricting its usage to vanilla Kubernetes objects or whether we have custom resources managed by the controller. In the former case, we already have all types available in the Kubernetes client library of our choice. For the CRD case, we don't have the type information out of the box, and we can either use a schemaless approach for managing CRD resources or define the custom types on our own, possibly based on an OpenAPI schema contained in the CRD definition. Support for typed CRDs varies by client library and framework used.

Figure 28-1 shows our controller and operator categorization starting from simpler resource definition options to more advanced with the boundary between controller and operator being the use of custom resources.

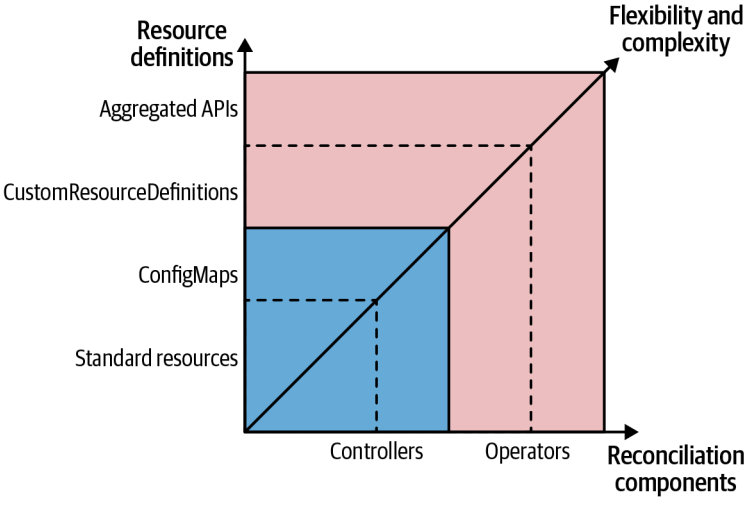


Figure 28-1. Spectrum of controllers and operators

3 However, you should be aware of common API conventions for `status` and other fields when designing your CRDs. Following common community conventions makes it easier for people and tooling to read your new API objects.

For operators, there is even a more advanced Kubernetes extension hook option. When Kubernetes-managed CRDs are not sufficient to represent a problem domain, you can extend the Kubernetes API with its own aggregation layer. We can add a custom-implemented `APIService` resource as a new URL path to the Kubernetes API.

To connect a Service that is backed by a Pod with the `APIService`, you can use a resource like that shown in Example 28-4.

Example 28-4. API aggregation with a custom APIService

```
apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  name: v1alpha1.sample-api.k8spatterns.io
spec:
  group: sample-api.k8spattterns.io
  service:
    name: custom-api-server
  version: v1alpha1
```

Besides the Service and Pod implementation, we need some additional security configuration for setting up the ServiceAccount under which the Pod is running.

After it is set up, every request to the API Server `https://<api server ip>/apis/sample-api.k8spatterns.io/v1alpha1/namespaces/<ns>/...` is directed to our custom Service implementation. It's up to this custom Service implementation to handle these requests, including persisting the resources managed via this API. This approach is different from the preceding CRD case, where Kubernetes itself completely manages the custom resources.

With a custom API Server, you have many more degrees of freedom, which allows you to go beyond watching resource lifecycle events. On the other hand, you also have to implement much more logic, so for typical use cases, an operator dealing with plain CRDs is often good enough.

A detailed exploration of the API Server capabilities is beyond the scope of this chapter. The official documentation as well as a complete sample-apiserver have more detailed information. Also, you can use the apiserver-builder library, which helps with implementing API Server aggregation.

Now, let's see how you can develop and deploy operators with CRDs.

Operator Development and Deployment

Several toolkits and frameworks are available for developing operators. The three main projects aiding in the creation of operators are as follows:

- Kubebuilder developed under the SIG API Machinery of Kubernetes itself
- Operator Framework, a CNCF project
- Metacontroller from Google Cloud Platform

We touch on each of these very briefly to give you a good starting point for developing and maintaining your own Operators.

Kubebuilder

Kubebuilder, a project by the SIG API Machinery,[4] is a framework and library for creating Kubernetes APIs via CustomResourceDefinitions.

It comes with outstanding documentation that also covers general aspects for programming Kubernetes. Kubebuilder's focus is on creating Golang-based operators by adding higher-level abstractions on top of the Kubernetes API to remove some of the overhead. It also offers scaffolding of new projects and supports multiple CRDs that can be watched by a single operator. Other projects can consume Kubebuilder as a library, and it also offers a plugin architecture to extend the support to languages and platforms beyond Golang. For programming against the Kubernetes API, Kubebuilder is an excellent starting point.

Operator framework

The Operator Framework provides extensive support for developing operators. It offers several subcomponents:

- The *Operator SDK* provides a high-level API for accessing a Kubernetes cluster and a scaffolding to start an operator project.
- The *Operator Lifecycle Manager* manages the release and updates of operators and their CRDs. You can think of it as a kind of "operator operator."
- *Operator Hub* is a publicly available catalog of operators dedicated to sharing operators built by the community.

4 Special Interest Groups (SIGs) are how the Kubernetes community organizes feature areas. You can find a list of current SIGs on the Kubernetes community site.

In the first edition of this book in 2019, we mentioned the high feature overlap of Kubebuilder and the Operator-SDK, and we speculated that both projects might eventually merge. It turned out that instead of a full merge, a different strategy was chosen by the community: all the overlapping parts have been moved to Kubebuilder, and the Operator-SDK uses Kubebuilder now as a dependency. This move is a good example of the power and self-healing effect of community-driven open source projects. The article "What Are the Differences Between Kubebuilder and Operator-SDK?" contains more information about the relationship between Kubebuilder and the Operator-SDK. The *Operator-SDK* offers everything needed for developing and maintaining Kubernetes operators. It is built on top of Kubebuilder and uses it directly for scaffolding and managing operators written in Golang. Beyond that, it benefits from Kubebuilder's plugin system for creating operators based on other technologies. As of 2023, the Operator-SDK provides plugins for creating operators based on Ansible playbooks or Helm Charts and Java-based operators that use a Quarkus runtime. When scaffolding a project, the SDK also adds the appropriate hooks for integration with the Operator Lifecycle Manager and the Operator Hub.

The *Operator Lifecycle Manager* (OLM) provides valuable help when using operators. One issue with CRDs is that these resources can be registered only cluster-wide and require cluster-admin permissions. While regular Kubernetes users can typically manage all aspects of the namespaces they have granted access to, they can't just use operators without interaction with a cluster administrator.

To streamline this interaction, the OLM is a cluster service running in the background under a service account with permission to install CRDs. A dedicated CRD called ClusterServiceVersion (CSV) is registered along with the OLM and allows us to specify the Deployment of an operator together with references to the CRD definitions associated with this operator. As soon as we have created such a CSV, one part of the OLM waits for that CRD and all its dependent CRDs to be registered. If this is the case, the OLM deploys the operator specified in the CSV. Then, another part of the OLM can be used to register these CRDs on behalf of a nonprivileged user. This approach is an elegant way to allow regular cluster users to install their operators.

Operators can be easily published at the Operator Hub. Operator Hub makes it easy to discover and install operators. The metadata-like name, icon, description, and more is extracted from the operator's CSV and rendered in a friendly web UI. Operator Hub also introduces the concept of *channels* that allow you to provide different streams like "stable" or "alpha," to which users can subscribe for automatic updates of various maturity levels.

Metacontroller

Metacontroller is very different from the other two operator building frameworks as it extends Kubernetes with APIs that encapsulate the common parts of writing custom controllers. It acts similarly to Kubernetes Controller Manager by running multiple controllers that are not hardcoded but are defined dynamically through Metacontroller-specific CRDs. In other words, it's a delegating controller that calls out to the service providing the actual controller logic.

Another way to describe Metacontroller is as declarative behavior. While CRDs allow us to store new types in Kubernetes APIs, Metacontroller makes it easy to define the behavior for standard or custom resources declaratively.

When we define a controller through Metacontroller, we have to provide a function that contains only the business logic specific to our controller. Metacontroller handles all interactions with the Kubernetes APIs, runs a reconciliation loop on our behalf, and calls our function through a webhook. The webhook gets called with a well-defined payload describing the CRD event. As the function returns the value, we return a definition of the Kubernetes resources that should be created (or deleted) on behalf of our controller function.

This delegation allows us to write functions in any language that can understand HTTP and JSON and that do not have any dependency on the Kubernetes API or its client libraries. The functions can be hosted on Kubernetes, or externally on a Functions-as-a-Service provider, or somewhere else.

We cannot go into many details here, but if your use case involves extending and customizing Kubernetes with simple automation or orchestration, and you don't need any extra functionality, you should have a look at Metacontroller, especially when you want to implement your business logic in a language other than Go. Some controller examples will demonstrate how to implement StatefulSet, Blue-Green Deployment, Indexed Job, and Service per Pod by using Metacontroller only.

Example

Let's look at a concrete operator example. We extend our example in Chapter 27, "Controller", and introduce a CRD of the type ConfigWatcher. An instance of this CRD then specifies a reference to the ConfigMap to watch and specifies which Pods to restart if this ConfigMap changes. With this approach, we remove the dependency of the ConfigMap on the Pods, as we don't have to modify the ConfigMap itself to add triggering annotations. Also, with our simple annotation-based approach in the Controller example, we can connect only a ConfigMap to a single application too. With a CRD, arbitrary combinations of ConfigMaps and Pods are possible.

This ConfigWatcher custom resource is shown in Example 28-5.

Example 28-5. Simple ConfigWatcher resource

```
apiVersion: k8spatterns.io/v1
kind: ConfigWatcher
metadata:
  name: webapp-config-watcher
spec:
  configMap: webapp-config ❶
  podSelector:             ❷
    app: webapp
```

❶ Reference to ConfigMap to watch.

❷ Label selector to determine Pods to restart.

In this definition, the attribute `configMap` references the name of the `ConfigMap` to watch. The field `podSelector` is a collection of labels and their values, which identify the Pods to restart.

We define the type of this custom resource with a CRD (shown in Example 28-6).

Example 28-6. ConfigWatcher CRD

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: configwatchers.k8spatterns.io
spec:
  scope: Namespaced          ❶
  group: k8spatterns.io      ❷
  names:
    kind: ConfigWatcher      ❸
    singular: configwatcher  ❹
    plural: configwatchers
  versions:
  - name: v1                 ❺
    storage: true
    served: true
    schema:
      openAPIV3Schema:       ❻
        type: object
        properties:
          configMap:
            type: string
            description: "Name of the ConfigMap"
          podSelector:
            type: object
            description: "Label selector for Pods"
            additionalProperties:
              type: string
```

❶ Connected to a namespace.

❷ Dedicated API group.

❸ Unique kind of this CRD.

❹ Labels of the resource as used in tools like `kubectl`.

❺ Initial version.

❻ OpenAPI V3 schema specification for this CRD.

For our operator to be able to manage custom resources of this type, we need to attach a ServiceAccount with the proper permissions to our operator's Deployment. For this task, we introduce a dedicated Role used later in a RoleBinding to attach it to the ServiceAccount in Example 28-7. We explain the concept and usage of ServiceAccounts, Roles, and RoleBindings in much more details in Chapter 26, "Access Control". For now, it is sufficient to know that the Role definition in Example 28-6 grants permission for all API operations to any instance of ConfigWatcher resources.

Example 28-7. Role definition allowing access to custom resource

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: config-watcher-crd
rules:
- apiGroups:
  - k8spatterns.io
  resources:
  - configwatchers
  - configwatchers/finalizers
  verbs: [ get, list, create, update, delete, deletecollection, watch ]
```

With these CRDs in place, we can now define custom resources as in Example 28-5.

To make sense of these resources, we have to implement a controller that evaluates these resources and triggers a Pod restart when the ConfigMap changes.

We expand here on our controller script in Example 27-2 and adapt the event loop in the controller script.

In the case of a ConfigMap update, instead of checking for a specific annotation, we do a query on all resources of the kind ConfigWatcher and check whether the modified ConfigMap is included as a `configMap` value. Example 28-8 shows the reconciliation loop. Refer to our Git repository for the full example, which also includes detailed instructions for installing this operator.

Example 28-8. WatchConfig controller reconciliation loop

```
curl -Ns $base/api/v1/${ns}/configmaps?watch=true | \          ❶
while read -r event
do
  type=$(echo "$event" | jq -r '.type')
  if [ $type = "MODIFIED" ]; then                              ❷

    watch_url="$base/apis/k8spatterns.io/v1/${ns}/configwatchers"
    config_map=$(echo "$event" | jq -r '.object.metadata.name')

    watcher_list=$(curl -s $watch_url | jq -r '.items[]')      ❸

    watchers=$(echo $watcher_list | \                          ❹
               jq -r "select(.spec.configMap == \"$config_map\") | .metadata.name")

    for watcher in watchers; do                                ❺
      label_selector=$(extract_label_selector $watcher)
      delete_pods_with_selector "$label_selector"
    done
  fi
done
```

❶ Start a watch stream to watch for ConfigMap changes for a given namespace.

❷ Check for a `MODIFIED` event only.

❸ Get a list of all installed ConfigWatcher custom resources.

❹ Extract from this list all ConfigWatcher elements that refer to this ConfigMap.

❺ For every ConfigWatcher found, delete the configured Pod via a selector. The logic for calculating a label selector as well as the deletion of the Pods are omitted here for clarity. Refer to the example code in our Git repository for the full implementation.

As for the controller example, this controller can be tested with a sample web application that is provided in our example Git repository. The only difference with this Deployment is that we use an unannotated ConfigMap for the application configuration.

Although our operator is quite functional, it is also clear that our shell script-based operator is still quite simple and doesn't cover edge or error cases. You can find many more interesting, production-grade examples in the wild.

The canonical place to find real-world operators is Operator Hub. The operators in this catalog are all based on the concepts covered in this chapter. We have already seen how a Prometheus operator can manage Prometheus installations. Another

Golang-based operator is the etcd operator for managing an etcd key-value store and automating operational tasks like backing up and restoring the database.

If you are looking for an operator written in the Java programming language, the *Strimzi Operator* is an excellent example of an operator that manages a complex messaging system like Apache Kafka on Kubernetes. Another good starting point for Java-based operators is the *Java Operator Plugin*, part of the Operator-SDK. As of 2023, it is still a young initiative; the best entry point for learning more about creating Java-based operators is the tutorial that explains the process to create a fully working operator.

Discussion

While we have learned how to extend the Kubernetes platform, operators are still not a silver bullet. Before using an operator, you should carefully look at your use case to determine whether it fits the Kubernetes paradigm.

In many cases, a plain controller working with standard resources is good enough. This approach has the advantage that it doesn't need any cluster-admin permission to register a CRD, but it has its limitations when it comes to security and validation.

An operator is a good fit for modeling a custom domain logic that fits nicely with the declarative Kubernetes way of handling resources with reactive controllers.

More specifically, consider using an operator with CRDs for your application domain for any of the following situations:

- You want tight integration into the already-existing Kubernetes tooling like `kubectl`.
- You are working on a greenfield project where you can design the application from the ground up.
- You benefit from Kubernetes concepts like resource paths, API groups, API versioning, and especially namespaces.
- You want to have good client support for accessing the API with watches, authentication, role-based authorization, and selectors for metadata.

If your custom use case fits these criteria, but you need more flexibility in how custom resources can be implemented and persisted, consider using a custom API Server. However, you should also not consider Kubernetes extension points as the golden hammer for everything.

If your use case is not declarative, if the data to manage does not fit into the Kubernetes resource model, or you don't need a tight integration into the platform, you

are probably better off writing your standalone API and exposing it with a classical Service or Ingress object.

The Kubernetes documentation itself also has a chapter for suggestions on when to use a controller, operator, API aggregation, or custom API implementation.

More Information

- Operator Example
- OpenAPI V3
- Kubebuilder
- Operator Framework
- Metacontroller
- Client Libraries
- Extend the Kubernetes API with CustomResourceDefinitions
- Custom Resources
- Sample-Controller
- What Are Red Hat OpenShift Operators?

CHAPTER 29

Elastic Scale

The *Elastic Scale* pattern covers application scaling in multiple dimensions: horizontal scaling by adapting the number of Pod replicas, vertical scaling by adapting resource requirements for Pods, and scaling the cluster itself by changing the number of cluster nodes. While all of these actions can be performed manually, in this chapter we explore how Kubernetes can perform scaling based on load automatically.

Problem

Kubernetes automates the orchestration and management of distributed applications composed of a large number of immutable containers by maintaining their declaratively expressed desired state. However, with the seasonal nature of many workloads that often change over time, it is not an easy task to figure out how the desired state should look. Accurately identifying how many resources a container will require and how many replicas a service will need at a given time to meet service-level agreements takes time and effort. Luckily, Kubernetes makes it easy to alter the resources of a container, the desired replicas for a service, or the number of nodes in the cluster. Such changes can happen either manually, or given specific rules, can be performed in a fully automated manner.

Kubernetes not only can preserve a fixed Pod and cluster setup but can also monitor external load and capacity-related events, analyze the current state, and scale itself for the desired performance. This kind of observation is a way for Kubernetes to adapt and gain antifragile traits based on actual usage metrics rather than anticipated factors. Let's explore the different ways we can achieve such behavior and how to combine the various scaling methods for an even greater experience.

Solution

There are two main approaches to scaling any application: horizontal and vertical. *Horizontally* in the Kubernetes world equates to creating more replicas of a Pod. *Vertically* scaling implies giving more resources to running containers managed by Pods. While it may seem straightforward on paper, creating an application configuration for autoscaling on a shared cloud platform without affecting other services and the cluster itself requires significant trial and error. As always, Kubernetes provides a variety of features and techniques to find the best setup for our applications, and we explore them briefly here.

Manual Horizontal Scaling

The manual scaling approach, as the name suggests, is based on a human operator issuing commands to Kubernetes. This approach can be used in the absence of autoscaling or for gradual discovery and tuning of the optimal configuration of an application matching the slow-changing load over long periods. An advantage of the manual approach is that it also allows anticipatory rather than reactive-only changes: knowing the seasonality and the expected application load, you can scale it out in advance, rather than reacting to an already-increased load through autoscaling, for example. We can perform manual scaling in two styles.

Imperative scaling

A controller such as ReplicaSet is responsible for making sure a specific number of Pod instances are always up and running. Thus, scaling a Pod is as trivially simple as changing the number of desired replicas. Given a Deployment named `random-generator`, scaling it to four instances can be done in one command, as shown in Example 29-1.

Example 29-1. Scaling a Deployment's replicas on the command line

```
kubectl scale random-generator --replicas=4
```

After such a change, the ReplicaSet could either create additional Pods to scale up or, if there are more Pods than desired, delete them to scale down.

Declarative scaling

While using the scale command is trivially simple and good for quick reactions to emergencies, it does not preserve this configuration outside the cluster. Typically, all Kubernetes applications would have their resource definitions stored in a source control system that also includes the number of replicas. Recreating the ReplicaSet from its original definition would change the number of replicas back to its previous

number. To avoid such a configuration drift and to introduce operational processes for backporting changes, it is a better practice to change the desired number of replicas declaratively in the ReplicaSet or some other definition and apply the changes to Kubernetes, as shown in Example 29-2.

Example 29-2. Using a Deployment for declaratively setting the number of replicas

```
kubectl apply -f random-generator-deployment.yaml
```

We can scale resources managing multiple Pods such as ReplicaSets, Deployments, and StatefulSets. Notice the asymmetric behavior in scaling a StatefulSet with persistent storage. As described in Chapter 12, "Stateful Service", if the StatefulSet has a `.spec.volumeClaimTemplates` element, it will create PVCs while scaling, but it won't delete them when scaling down to preserve the storage from deletion.

Another Kubernetes resource that can be scaled but follows a different naming convention is the Job resource, which we described in Chapter 7, "Batch Job". A Job can be scaled to execute multiple instances of the same Pod at the same time by changing the `.spec.parallelism` field rather than `.spec.replicas`. However, the semantic effect is the same: increased capacity with more processing units that act as a single logical unit.

For describing resource fields, we use a JSON path notation. For example, `.spec.replicas` points to the `replicas` field of the resource's `spec` section.

Both manual scaling styles (imperative and declarative) expect a human to observe or anticipate a change in the application load, make a decision on how much to scale, and apply it to the cluster. They have the same effect, but they are not suitable for dynamic workload patterns that change often and require continuous adaptation. Next, let's see how we can automate scaling decisions themselves.

Horizontal Pod Autoscaling

Many workloads have a dynamic nature that varies over time and makes it hard to have a fixed scaling configuration. But cloud native technologies such as Kubernetes enable you to create applications that adapt to changing loads. Autoscaling in Kubernetes allows us to define a varying application capacity that is not fixed but instead ensures just enough capacity to handle a different load. The most straightforward approach to achieving such behavior is by using a HorizontalPodAutoscaler (HPA) to horizontally scale the number of Pods. HPA is an intrinsic part of Kubernetes and does not require any extra installation steps. One important limitation of

the HPA is that it can't scale down to zero Pods so that no resources are consumed at all if nobody is using the deployed workload. Luckily, Kubernetes add-ons offer scale-to-zero and transform Kubernetes into a true serverless platform. Knative and KEDA are the most prominent of such Kubernetes extensions. We will have a look at both in "Knative" on page 317 and "KEDA" on page 321, but let's first see how Kubernetes offers horizontal autoscaling out of the box.

Kubernetes HorizontalPodAutoscaler

The HPA is best explained with an example. An HPA for the `random-generator` Deployment can be created with the command in Example 29-3. For the HPA to have any effect, it is important that the Deployment declare a `.spec.resources.requests` limit for the CPU as described in Chapter 2, "Predictable Demands". Another requirement is enabling the metrics server, which is a cluster-wide aggregator of resource usage data.

Example 29-3. Create HPA definition on the command line

```
kubectl autoscale deployment random-generator --cpu-percent=50 --min=1 --max=5
```

The preceding command will create the HPA definition shown in Example 29-4.

Example 29-4. HPA definition

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: random-generator
spec:
  minReplicas: 1                 ❶
  maxReplicas: 5                 ❷
  scaleTargetRef:                ❸
    apiVersion: apps/v1
    kind: Deployment
    name: random-generator
  metrics:
  - resource:
      name: cpu
      target:
        averageUtilization: 50   ❹
        type: Utilization
    type: Resource
```

❶ Minimum number of Pods that should always run.

❷ Maximum number of Pods until the HPA can scale up.

❸ Reference to the object that should be associated with this HPA.

❹ Desired CPU usage as a percentage of the Pods' requested CPU resource. For example, when the Pods have a `.spec.resources.requests.cpu` of 200m, a scale-up happens when on average more than 100m CPU (= 50%) is utilized.

This definition instructs the HPA controller to keep between one and five Pod instances to retain an average Pod CPU usage of around 50% of the specified CPU resource limit in the Pod's `.spec.resources.requests` declaration. While it is possible to apply such an HPA to any resource that supports the `scale` subresource such as Deployments, ReplicaSets, and StatefulSets, you must consider the side effects. Deployments create new ReplicaSets during updates but without copying over any HPA definitions. If you apply an HPA to a ReplicaSet managed by a Deployment, it is not copied over to new ReplicaSets and will be lost. A better technique is to apply the HPA to the higher-level Deployment abstraction, which preserves and applies the HPA to the new ReplicaSet versions.

Now, let's see how an HPA can replace a human operator to ensure autoscaling. At a high level, the HPA controller performs the following steps continuously:

1. It retrieves metrics about the Pods that are subject to scaling according to the HPA definition. Metrics are not read directly from the Pods but from the Kubernetes Metrics APIs that serve aggregated metrics (and even custom and external metrics if configured to do so). Pod-level resource metrics are obtained from the Metrics API, and all other metrics are retrieved from the Custom Metrics API of Kubernetes.
2. It calculates the required number of replicas based on the current metric value and targeting the desired metric value. Here is a simplified version of the formula:

$$desiredReplicas = \left\lceil currentReplicas \times \frac{currentMetricValue}{desiredMetricValue} \right\rceil$$

For example, if there is a single Pod with a current CPU usage metric value of 90% of the specified CPU resource request value,[1] and the desired value is 50%, the number of replicas will be doubled, as $\left\lceil 1 \times \frac{90}{50} \right\rceil = 2$. The actual implementation is more complicated as it has to consider multiple running Pod instances, cover multiple metric types, and account for many corner cases and fluctuating values as well. If multiple

1 For multiple running Pods, the average CPU utilization is used as *currentMetricValue*.

metrics are specified, for example, then the HPA evaluates each metric separately and proposes a value that is the largest of all. After all the calculations, the final output is a single-integer number representing the number of desired replicas that keep the measured value below the desired threshold value.

The `replicas` field of the autoscaled resource will be updated with this calculated number, and other controllers do their bit of work in achieving and keeping the new desired state. Figure 29-1 shows how the HPA works: monitoring metrics and changing declared replicas accordingly.

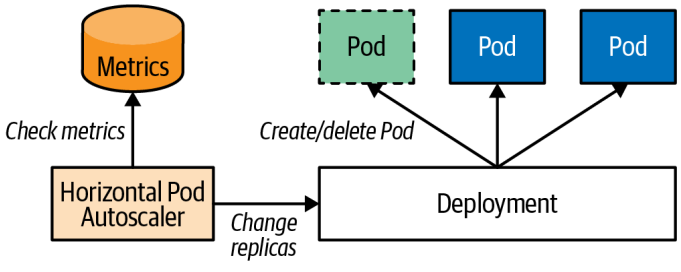


Figure 29-1. Horizontal Pod autoscaling mechanism

Autoscaling is an area of Kubernetes with many low-level details, and each one can have a significant impact on the overall behavior of autoscaling. As such, it is beyond the scope of this book to cover all the details, but "More Information" on page 333 provides the latest up-to-date information on the subject.

Broadly, there are the following metric types:

Standard metrics
: These metrics are declared with `.spec.metrics.resource[].type` equal to `Resource` and represent resource usage metrics such as CPU and memory. They are generic and available for any container on any cluster under the same name. You can specify them as a percentage, as we did in the preceding example, or as an absolute value. In both cases, the values are based on the guaranteed resource amount, which are the container resource `requests` values and not the `limits` values. These are the easiest-to-use metric types generally provided by the metrics server component, which can be launched as cluster add-ons.

Custom metrics
: These metrics with `.spec.metrics.resource[].type` equal to `Object` or `Pod` require a more advanced cluster-monitoring setup, which can vary from cluster to cluster. A custom metric with the Pod type, as the name suggests, describes a Pod-specific metric, whereas the Object type can describe any other object. The custom metrics are served in an aggregated API Server under the

`custom.metrics.k8s.io` API path and are provided by different metrics adapters, such as Prometheus, Datadog, Microsoft Azure, or Google Stackdriver.

External metrics
: This category is for metrics that describe resources that are not a part of the Kubernetes cluster. For example, you may have a Pod that consumes messages from a cloud-based queueing service. In such a scenario, you'll want to scale the number of consumer Pods based on the queue depth. Such a metric would be populated by an external metrics plugin similar to custom metrics. Only one external metrics endpoint can be hooked into the Kubernetes API server. For using metrics from many different external systems, an extra aggregation layer like KEDA is required (see "KEDA" on page 321).

Getting autoscaling right is not easy and involves a little experimenting and tuning. The following are a few of the main areas to consider when setting up an HPA:

Metric selection
: Probably one of the most critical decisions around autoscaling is which metrics to use. For an HPA to be useful, there must be a direct correlation between the metric value and the number of Pod replicas. For example, if the chosen metric is of the Queries-per-Second kind (such as HTTP requests per second), increasing the number of Pods causes the average number of queries to go down as the queries are dispatched to more Pods. The same is true if the metric is CPU usage, as there is a direct correlation between the query rate and CPU usage (an increased number of queries would result in increased CPU usage). For other metrics such as memory consumption, that is not the case. The issue with memory is that if a service consumes a certain amount of memory, starting more Pod instances most likely will not result in a memory decrease unless the application is clustered and aware of the other instances and has mechanisms to distribute and release its memory. If the memory is not released and reflected in the metrics, the HPA would create more and more Pods in an effort to decrease it, until it reaches the upper replica threshold, which is probably not the desired behavior. So choose a metric that is directly (preferably linearly) correlated to the number of Pods.

Preventing thrashing
: The HPA applies various techniques to avoid rapid execution of conflicting decisions that can lead to a fluctuating number of replicas when the load is not stable. For example, during scale-up, the HPA disregards high CPU usage samples when a Pod is initializing, ensuring a smoothing reaction to increasing load. During scale-down, to avoid scaling down in response to a short dip in usage, the controller considers all scale recommendations during a configurable time window and chooses the highest recommendation from within the window. All this makes the HPA more stable when dealing with random metric fluctuations.

Delayed reaction
: Triggering a scaling action based on a metric value is a multistep process involving multiple Kubernetes components. First, it is the cAdvisor (container advisor) agent that collects metrics at regular intervals for the Kubelet. Then the metrics server collects metrics from the Kubelet at regular intervals. The HPA controller loop also runs periodically and analyzes the collected metrics. The HPA scaling formula introduces some delayed reaction to prevent fluctuations/thrashing (as explained in the previous point). All this activity accumulates into a delay between the cause and the scaling reaction. Tuning these parameters by introducing more delay makes the HPA less responsive, but reducing the delays increases the load on the platform and increases thrashing. Configuring Kubernetes to balance resources and performance is an ongoing learning process.

Tuning the autoscale algorithm for the HPA in Kubernetes can be complex. To help with this, Kubernetes provides the `.spec.behavior` field in the HPA specification. This field allows you to customize the behavior of the HPA when scaling the number of replicas in a Deployment.

For each scaling direction (up or down), you can use the `.spec.behavior` field to specify the following parameters:

`policies`
: These describe the maximum number of replicas to scale in a given period.

`stabilizationWindowSeconds`
: This specifies when the HPA will not make any further scaling decisions. Setting this field can help to prevent thrashing effects, where the HPA rapidly scales the number of replicas up and down.

Example 29-5 shows how the behavior can be configured. All behavior parameters can also be configured on the CLI with `kubectl autoscale`.

Example 29-5. Configuration of the autoscaling algorithm

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
...
spec:
  ...
  behavior:
    scaleDown:                          ❶
      stabilizationWindowSeconds: 300   ❷
      policies:
      - type: Percent                   ❸
        value: 10
        periodSeconds: 60
```

```
  scaleUp:                          ❹
    policies:
    - type: Pods                    ❺
      value: 4
      periodSeconds: 15
```

❶ Scaling behavior when scaling down.

❷ A 5-minute minimum window for down-scaling decisions to prevent flapping.

❸ Scale down at most 10% of the current replicas in one minute.

❹ Scaling behavior when scaling up.

❺ Scale up at most four Pods within 15 seconds.

Please refer to the Kubernetes documentation on configuring the scaling behavior for all the details and usage examples.

While the HPA is very powerful and covers the basic needs for autoscaling, it lacks one crucial feature: scale-to-zero for stopping all Pods of an application if it is not used. That's important so that it does not cause any costs based on memory, CPU, or network usage. However, scaling to zero is not so hard; the tricky part is waking up again and scaling to at least one Pod by a trigger, like an incoming HTTP request or an event to process.

The following two sections introduce the two most prominent Kubernetes-based add-ons for enabling scale-to-zero: Knative and KEDA. It is essential to understand that Knative and KEDA are not alternative but complementary solutions. Both projects cover different use cases and can ideally be used together. As we will see, Knative specializes in stateless HTTP applications and offers an autoscaling algorithm that goes beyond the capabilities of the HPA. On the other hand, KEDA is a pull-based approach that can be triggered by many different sources, like messages in a Kafka topic or IBM MQ queue.

Let's have a closer look at Knative and KEDA.

Knative

Knative is a CNCF project initiated by Google in 2018, with broad industry support from vendors like IBM, VMware, and Red Hat. This Kubernetes add-on consists of three parts:

Knative Serving
: This is a simplified application deployment model with sophisticated autoscaling and traffic-splitting capabilities, including scale-to-zero.

Knative Eventing
: This provides everything needed to create an Event Mesh to connect event sources that produce CloudEvents[2] with a sink that consumes these events. Those sinks are typically Knative Serving services.

Knative Functions
: This is for scaffolding and building Knative Serving services from source code. It supports various programming languages and offers an AWS Lambda-like programming model.

In this section, we will focus on Knative Serving and its autoscaler for an application that uses HTTP to offer its services. For those workloads, CPU and memory are metrics that only indirectly correlate to actual usage. A much better metric is the number of *concurrent requests* per Pod—i.e., requests that are processed in parallel.

Another HTTP-based metric that Knative can use is *requests per second* (rps). Still, this metric does not say anything about the costs of a single request, so concurrent requests are typically the much better metric to use, as they capture the frequency of requests and the duration of those requests. You can select the scale metric individually for each application or as a global default.

Basing the autoscaling decision on concurrent requests gives a much better correlation to the latency of HTTP request processing than scaling based on CPU or memory consumption can provide.

Historically, Knative used to be implemented as a custom metric adapter for the HPA in Kubernetes. However, it later developed its own implementation in order to have more flexibility in influencing the scaling algorithm and to avoid the bottleneck of being able to register only a single custom metric adapter in a Kubernetes cluster.

While Knative still supports using the HPA for scaling based on memory or CPU usage, it now focuses on using its own autoscaling implementation, called the Knative Pod Autoscaler (KPA). This allows Knative to have more control over the scaling algorithm and to better optimize it for the needs of the application.

The architecture of the KPA is shown in Figure 29-2.

2 CloudEvents is a CNCF standard that describes the format and metadata for events in a cloud context.

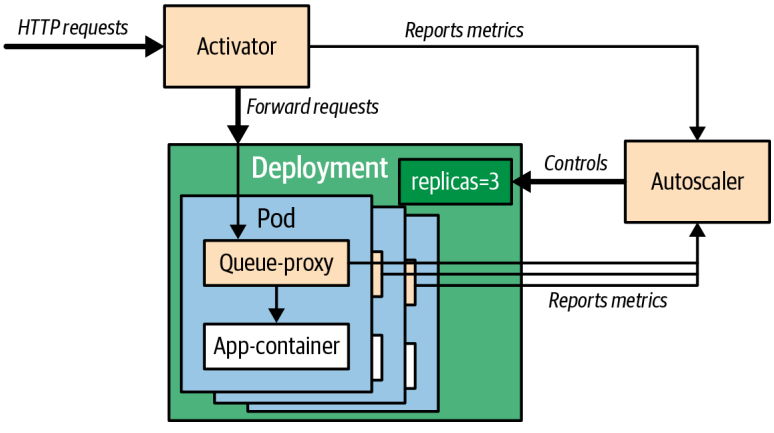


Figure 29-2. Knative Pod Autoscaler

Three components are playing together for autoscaling a service:

Activator
: This is a proxy in front of the application that is always available, even when the application is scaled down to zero Pods. When the application is scaled down to zero, and a first request comes in, the request gets buffered, and the application is scaled up to at least one Pod. It's important to note that during a *cold start*, all incoming requests will be buffered to ensure that no requests are lost.

Queue proxy
: The queue proxy is an ambassador sidecar described in Chapter 18 that is injected into the application's Pod by the Knative controller. It intercepts the request path for collecting metrics relevant to autoscaling, like concurrent requests.

Autoscaler
: This is a service running in the background that is responsible for the scaling decision based on the data it gets from the activator and queue-proxy. The autoscaler is the one that sets the replica count in the application's ReplicaSet.

The KPA algorithm can be configured in many ways to optimize the autoscaling behavior for any workload and traffic shape. Table 29-1 shows some of the configuration options for tuning the KPA for individual services via annotations. Similar configuration options also exist for global defaults that are stored in a ConfigMap. You can find the full set of all autoscaling configuration options in the Knative documentation. This documentation has more details about the Knative scaling algorithm, like dealing with bursty workloads by scaling up more aggressively when the increase in concurrent requests is over a threshold.

Table 29-1. Important Knative scaling parameters. `autoscaling.knative.dev/`, the common annotation prefix, has been omitted.

Annotation	Description	Default
`target`	Number of simultaneous requests that can be processed by each replica. This is a soft limit and might be temporarily exceeded in case of a traffic burst. `.spec.concurrencyLimit` is used as a hard limit that can't be crossed.	100
`target-utilization-percentage`	Start creating new replicas if this fraction of the concurrency limit has been reached.	70
`min-scale`	Minimum number of replicas to keep. If set to a value greater than zero, the application will never scale down to zero.	0
`max-scale`	Upper bound for the number of replicas; zero means unlimited scaling.	0
`activation-scale`	How many replicas to create when scaling up from zero.	1
`scale-down-delay`	How long scale-down conditions must hold before scaling down. Useful for keeping replicas warm before scaling zero in order to avoid cold start time.	0s
`window`	Length of the time window over which metrics are averaged to provide the input for scaling decisions.	60s

Example 29-6 shows a Knative service that deploys an example application. It looks similar to a Kubernetes Deployment. However, behind the scenes, the Knative operator creates the Kubernetes resources needed to expose your application as a web service, i.e., a ReplicaSet, Kubernetes Service, and Ingress for exposing the application to the outside of your cluster.

Example 29-6. Knative service

```
apiVersion: serving.knative.dev/v1          ❶
kind: Service
metadata:
  name: random
  annotations:
    autoscaling.knative.dev/target: "80"    ❷
    autoscaling.knative.dev/window: "120s"
spec:
  template:
    spec:
      containers:
      - image: k8spatterns/random           ❸
```

❶ Knative also uses Service for the resource name but with the API group `serving.knative.dev`, which is different from a Kubernetes Service from the `core` API group.

❷ Options for tuning the autoscaling algorithm. See Table 29-1 for the available options.

❸ The only mandatory argument for a Knative Service is a reference to a container image.

We only briefly touch on Knative here. There is much more that can help you in operating the Knative autoscaler. Please check out the online documentation for more features of Knative Serving, like traffic splitting for the complex rollout scenarios we described in Chapter 3, "Declarative Deployment". Also, if you are following an event-driven architecture (EDA) paradigm for your applications, Knative Eventing and Knative Functions have a lot to offer.

KEDA

Kubernetes Event-Driven Autoscaling (KEDA) is the other important Kubernetes-based autoscaling platform that supports scale-to-zero but has a different scope than Knative. While Knative supports autoscaling based on HTTP traffic, KEDA is a pull-based approach that scales based on external metrics from different systems. Knative and KEDA play very well together, and there is only a little overlap,[3] so nothing prevents you from using both add-ons together.

So, what is KEDA? KEDA is a CNCF project that Microsoft and Red Hat created in 2019 and consists of the following components:

- The KEDA Operator reconciles a ScaledObject custom resource that connects the scaled target (e.g., a Deployment or StatefulSet) with an autoscale trigger that connects to an external system via a so-called *scaler*. It is also responsible for configuring the HPA with the external metrics service provided by KEDA.
- KEDA's metrics service is registered as an APIService resource in the Kubernetes API aggregation layer so that the HPA can use it as an external metrics service.

Figure 29-3 illustrates the relationship between the KEDA Operator, metrics service, and the Kubernetes HPA.

3 KEDA initially did not support HTTP-triggered autoscaling, and although there is now a KEDA HTTP add-on it is still in its infancy (in 2023), requires a complex setup, and would need to catch up quite a bit to reach the maturity of the KPA that is included out of the box in Knative.

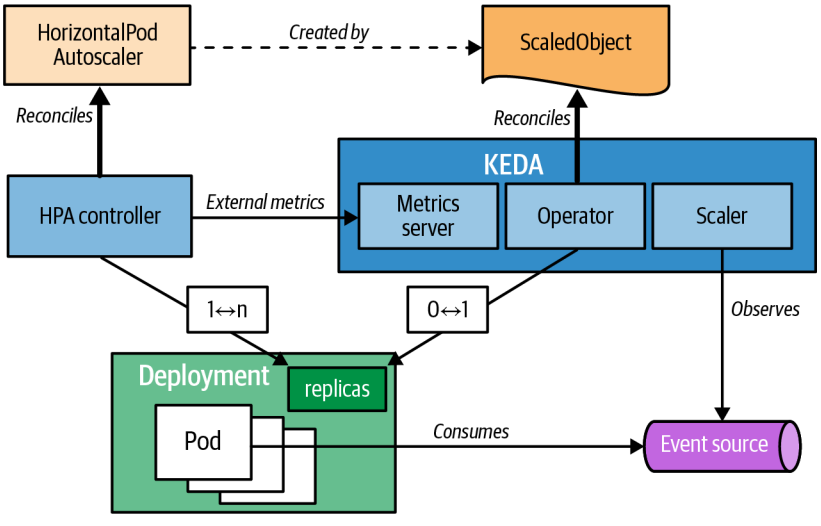


Figure 29-3. KEDA autoscaling components

While Knative is a complete solution that completely replaces HPA for a consumption-based autoscaling, KEDA is a hybrid solution. KEDA's autoscaling algorithm distinguishes between two scenarios:

- Activation by scaling from zero replicas to one ($0 \leftrightarrow 1$): This action is performed by the KEDA operator itself when it detects that a used scaler's metric exceeds a certain threshold.
- Scaling up and down when running ($1 \leftrightarrow n$): When the workload is already active, the HPA takes over and scales based on the external metric that KEDA offers.

The central element for KEDA is the custom resource ScaledObject, provided by the user to configure KEDA-based autoscaling and playing a similar role as the HorizontalPodAutoscaler resource. As soon as the KEDA operator detects a new instance of ScaledObject, it automatically creates a HorizontalPodAutoscaler resource that uses the KEDA metrics service as an external metrics provider and the scaling parameters.

Example 29-7 shows how you can scale a Deployment based on the number of messages in an Apache Kafka topic.

Example 29-7. ScaledObject definition

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: kafka-scaledobject
spec:
  scaleTargetRef:
    name: kafka-consumer                          ❶
  pollingInterval: 30                             ❷
  triggers:
    - type: kafka                                 ❸
      metadata:
        bootstrapServers: bootstrap.kafka.svc:9092 ❹
        consumerGroup: my-group
        topic: my-topic
```

❶ Reference to a Deployment with the name `kafka-consumer` that should be autoscaled. You can also specify other scalable workloads here; Deployment is the default.

❷ In the action phase (scale from zero), poll every 30 seconds for the metric value. In this example, it is the number of messages in a Kafka topic.

❸ Select the Apache Kafka scaler.

❹ Configuration options for the Apache Kafka scaler—i.e., how to connect to the Kafka cluster and which topic to monitor.

KEDA provides many out-of-the-box scalers that can be selected to connect to external systems for the autoscaling stimulus. You can obtain the complete list of directly supported scalers from the KEDA home page. In addition, you can easily integrate custom scalers by providing an external service that communicates with KEDA over a gRPC-based API.

KEDA is a great autoscaling solution when you need to scale based on work items held in external systems, like message queues that your application consumes. To some degree, this pattern shares some of the characteristics of Chapter 7, "Batch Job": the workload runs only when work is done and does not consume any resources when idle. Both can be scaled up for parallel processing of the work items. The difference here is that a KEDA ScaledObject does the up-scale automatically, whereas for a Kubernetes `Job`, you must manually determine the parallelism parameters. With KEDA, you can also automatically trigger Kubernetes Jobs based on the availability of external workloads. The ScaledJob custom resource is precisely for this purpose so that instead of scaling up replicas from 0 to 1, a Job resource is started in case a scaler's activation threshold is met. Note that the `parallelism` field in the Job is still

fixed, but the autoscaling happens on the Job resource level itself (i.e., Job resources themselves play the role of replicas).

Push Versus Pull Horizontal Autoscalers

Kubernetes knows about two main types of horizontal autoscalers: push autoscalers and pull autoscalers.

Push autoscalers operate by actively pushing metrics to the autoscaler, which then uses those metrics to decide how to scale. This technique is often used when the metrics have been directly generated by a system closely integrated with the autoscaler. For example, in Knative, the Activator pushes the metrics about concurrent requests to the Autoscaler component, as illustrated in Figure 29-2.

Pull autoscalers operate by actively pulling metrics from the application or external sources. Pulling is often used when the metrics are not directly accessible to the autoscaler or when the metrics are stored in an external system. KEDA, for example, is a pull autoscaler that scales deployments based on, for example, the number of events or messages in a queue. Figure 29-3 shows how KEDA uses a custom Kubernetes controller to pull metrics about the number of events and then uses those metrics to determine whether to scale up or down.

Push autoscalers are often used for applications that receive data, like from HTTP endpoints. In contrast, pull autoscalers are suitable for applications that actively retrieve their workload, such as pulling from a message queue.

Table 29-2 summarizes the unique features and differences between HPA, Knative, and KEDA.

Table 29-2. Horizontal autoscaling on Kubernetes

	HPA	Knative	KEDA
Scale metrics	Resource usage	HTTP requests	External metrics like message queue backlog
Scale-to-zero	No	Yes	Yes
Type	Pull	Push	Pull
Typical use cases	Stable traffic web applications, Batch processing	Serverless applications with rapid scaling, serverless functions	Message-driven microservices

Now that we have seen all the possibilities for scaling horizontally with HPA, Knative, and KEDA, let's look at a completely different kind of scaling that does not alter the number of parallel-running replicas but lets your application grow and shrink.

Vertical Pod Autoscaling

Horizontal scaling is preferred over vertical scaling because it is less disruptive, especially for stateless services. That is not the case for stateful services, where vertical scaling may be preferred. Other scenarios where vertical scaling is useful include tuning the resource needs of a service based on actual load patterns. We've discussed why identifying the correct number of Pod replicas might be difficult and even impossible when the load changes over time. Vertical scaling also has these kinds of challenges in identifying the correct `requests` and `limits` for a container. The Kubernetes Vertical Pod Autoscaler (VPA) aims to address these challenges by automating the process of adjusting and allocating resources based on real-world usage feedback.

As we saw in Chapter 2, "Predictable Demands", every container in a Pod can specify its CPU and memory `requests`, which influences where the Pods will be scheduled. In a sense, the resource `requests` and `limits` of a Pod form a contract between the Pod and the scheduler, which causes a certain amount of resources to be guaranteed or prevents the Pod from being scheduled. Setting the memory `requests` too low can cause nodes to be more tightly packed, which in turn can lead to out-of-memory errors or workload eviction due to memory pressure. If the CPU `limits` are too low, CPU starvation and underperforming workloads can occur. On the other hand, specifying resource `requests` that are too high allocates unnecessary capacity, leading to wasted resources. It is important to set resource `requests` as accurately as possible since they impact the cluster utilization and the effectiveness of horizontal scaling. Let's see how VPA helps address this.

On a cluster with VPA and the metrics server installed, we can use a VPA definition to demonstrate vertical autoscaling of Pods, as in Example 29-8.

Example 29-8. VPA

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: random-generator-vpa
spec:
  targetRef:                    ❶
    apiVersion: apps/v1
    kind: Deployment
    name: random-generator
  updatePolicy:
    updateMode: "Off"           ❷
```

❶ Reference to the higher-level resource that holds the selector to identify the Pods to manage.

❷ The update policy for how VPA will apply changes.

A VPA definition has the following main parts:

Target reference
: The target reference points to a higher-level resource that controls Pods, like a Deployment or a StatefulSet. From this resource, the VPA looks up the label selector for identifying the Pods it should handle. If the reference points to a resource that does not contain such a selector, then it will report an error in the VPA status section.

Update policy
: The update policy controls how the VPA applies changes. The `Initial` mode allows you to assign resource requests only during Pod creation time and not later. The default `Auto` mode allows resource assignment to Pods at creation time, but additionally, it can update Pods during their lifetimes, by evicting and rescheduling the Pod. The value `Off` disables automatic changes to Pods but allows you to suggest resource values. This is a kind of dry run for discovering the right size of a container without applying it directly.

A VPA definition can also have a resource policy that influences how the VPA computes the recommended resources (e.g., by setting per-container lower and upper resource boundaries).

Depending on which `.spec.updatePolicy.updateMode` is configured, the VPA involves different system components. All three VPA components—recommender, admission plugin, and updater—are decoupled and independent and can be replaced with alternative implementations. The module with the intelligence to produce recommendations is the recommender, which is inspired by Google's Borg system. The implementation analyzes the actual resource usage of a container under load for a certain period (by default, eight days), produces a histogram, and chooses a high-percentile value for that period. In addition to metrics, it also considers resource and specifically memory-related Pod events such as evictions and `OutOfMemory` events.

In our example, we chose `.spec.updatePolicy.updateMode` equals `Off`, but there are two other options to choose from, each with a different level of potential disruption on the scaled Pods. Let's see how different values for `updateMode` work, starting from nondisruptive to a more disruptive order:

Off
: The VPA recommender gathers Pod metrics and events and then produces recommendations. The VPA recommendations are always stored in the `status` section of the VPA resource. However, this is as far as the `Off` mode goes. It analyzes and produces recommendations, but it does not apply them to the Pods. This mode is useful for getting insight on the Pod resource consumption without introducing any changes and causing disruption. That decision is left for the user to make if desired.

Initial
: In this mode, the VPA goes one step further. In addition to the activities performed by the recommender component, it also activates the VPA admission Controller, which applies the recommendations to newly created Pods only. For example, if a Pod is scaled manually, updated by a Deployment, or evicted and restarted for whatever reason, the Pod's resource request values are updated by the VPA Admission Controller.

  This controller is a *mutating admission Webhook* that overrides the `requests` of new matching Pods that are associated with the VPA resource. This mode does not restart a running Pod, but it is still partially disruptive because it changes the resource request of newly created Pods. This in turn can affect where a new Pod is scheduled. What's more, it is possible that after applying the recommended resource requests, the Pod is scheduled to a different node, which can have unexpected consequences. Or worse, the Pod might not be scheduled to any node if there is not enough capacity on the cluster.

Recreate and Auto
: In addition to the recommendation creation and its application for newly created Pods, as described previously, in this mode, the VPA also activates its updated component. The `Recreate` update mode forcibly evicts and restarts all Pods in the deployment to apply the VPA's recommendations, while the `Auto` update mode is supposed to support in-place updates of resource limits without restarting Pods in a future version of Kubernetes. As of 2023, `Auto` behaves the same as `Recreate`, so both update modes can be disruptive and may lead to the unexpected scheduling issues that have been described earlier.

Kubernetes is designed to manage immutable containers with immutable Pod `spec` definitions, as seen in Figure 29-4. While this simplifies horizontal scaling, it introduces challenges for vertical scaling, such as requiring Pod deletion and recreation, which can impact scheduling and cause service disruptions. This is true even when the Pod is scaling down and wants to release already-allocated resources with no disruption.

Another concern is the coexistence of VPA and HPA because these autoscalers are not currently aware of each other, which can lead to unwanted behavior. For example, if an HPA is using resource metrics such as CPU and memory, and the VPA is also influencing the same values, you may end up with horizontally scaled Pods that are also vertically scaled (hence double scaling).

We can't go into more details here. Although it is still evolving, it is worth keeping an eye on the VPA as it is a feature that has the potential to significantly improve resource consumption.

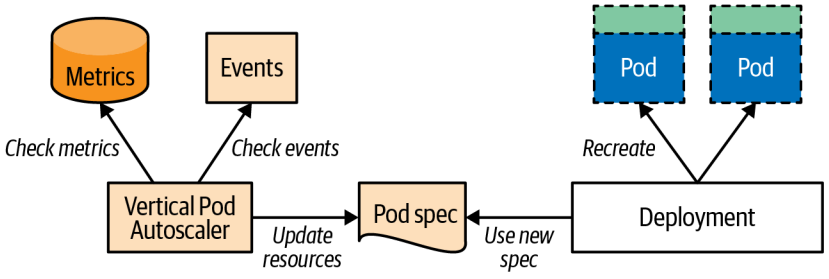


Figure 29-4. Vertical Pod autoscaling mechanism

Cluster Autoscaling

The patterns in this book primarily use Kubernetes primitives and resources targeted at developers using a Kubernetes cluster that's already set up, which is usually an operational task. Since it is a topic related to the elasticity and scaling of workloads, we will briefly cover the Kubernetes Cluster Autoscaler (CA) here.

One of the tenets of cloud computing is pay-as-you-go resource consumption. We can consume cloud services when needed, and only as much as needed. CA can interact with cloud providers where Kubernetes is running and request additional nodes during peak times or shut down idle nodes during other times, reducing infrastructure costs. While the HPA and VPA perform Pod-level scaling and ensure service-capacity elasticity within a cluster, the CA provides node scalability to ensure cluster-capacity elasticity.

Cluster API

All major cloud providers support Kubernetes CA. However, to make this happen, plugins have been written by cloud providers, leading to vendor locking and inconsistent CA support. Luckily, the Cluster API Kubernetes project aims to provide APIs for cluster creation, configuration, and management. All major public and private cloud providers like AWS, IBM Cloud, Azure, GCE, vSphere, and OpenStack support this initiative. This also allows CA to be used in on-premises Kubernetes installations. The heart of the Cluster API is a machine controller running in the background, for which several independent implementations like the Kubermatic machine-controller or the machine-api-operator by Red Hat OpenShift already exist. It is worth keeping an eye on the Cluster API as it may become the backbone for any cluster autoscaling in the future.

CA is a Kubernetes add-on that has to be turned on and configured with a minimum and maximum number of nodes. It can function only when the Kubernetes cluster is running on a cloud-computing infrastructure where nodes can be provisioned and

decommissioned on demand and that has support for Kubernetes CA, such as AWS, IBM Cloud Kubernetes Service, Microsoft Azure, or Google Compute Engine.

A CA primarily performs two operations: it add new nodes to a cluster or removes nodes from a cluster. Let's see how these actions are performed:

Adding a new node (scale-up)
: If you have an application with a variable load (busy times during the day, weekend, or holiday season and much less load during other times), you need varying capacity to meet these demands. You could buy fixed capacity from a cloud provider to cover the peak times, but paying for it during less busy periods reduces the benefits of cloud computing. This is where CA becomes truly useful.

  When a Pod is scaled horizontally or vertically, either manually or through HPA or VPA, the replicas have to be assigned to nodes with enough capacity to satisfy the requested CPU and memory. If no node in the cluster has enough capacity to satisfy all of the Pod's requirements, the Pod is marked as *unschedulable* and remains in the waiting state until such a node is found. CA monitors for such Pods to see whether adding a new node would satisfy the needs of the Pods. If the answer is yes, it resizes the cluster and accommodates the waiting Pods.

  CA cannot expand the cluster by a random node—it has to choose a node from the available node groups the cluster is running on.[4] It assumes that all the machines in a node group have the same capacity and the same labels, and that they run the same Pods specified by local manifest files or DaemonSets. This assumption is necessary for CA to estimate how much extra Pod capacity a new node will add to the cluster.

  If multiple node groups are satisfying the needs of the waiting Pods, CA can be configured to choose a node group by different strategies called *expanders*. An expander can expand a node group with an additional node by prioritizing least cost or least resource waste, accommodating most Pods, or just randomly. At the end of a successful node selection, a new machine should be provisioned by the cloud provider in a few minutes and registered in the API Server as a new Kubernetes node ready to host the waiting Pods.

Removing a node (scale-down)
: Scaling down Pods or nodes without service disruption is always more involved and requires many checks. CA performs scale-down if there is no need to scale up and a node is identified as unneeded. A node is qualified for scale-down if it satisfies the following main conditions:

4 Node groups is not an intrinsic Kubernetes concept (i.e., there is no NodeGroup resource) but is used as an abstraction in the CA and Cluster APIs to describe nodes that share certain characteristics.

- More than half of its capacity is unused—that is, the sum of all requested CPU and the memory of all Pods on the node is less than 50% of the node-allocatable resource capacity.
- All movable Pods on the node (Pods that are not run locally by manifest files or Pods created by DaemonSets) can be placed on other nodes. To prove that, CA performs a scheduling simulation and identifies the future location of every Pod that would be evicted. The final location of the Pods is still determined by the scheduler and can be different, but the simulation ensures there is spare capacity for the Pods.
- There are no other reasons to prevent node deletion, such as a node being excluded from scaling down through annotations.
- There are no Pods that cannot be moved, such as Pods with a PodDisruptionBudget that cannot be satisfied, Pods with local storage, Pods with annotations preventing eviction, Pods created without a controller, or system Pods.

All of these checks are performed to ensure no Pod is deleted that cannot be started on a different node. If all of the preceding conditions are true for a while (the default is 10 minutes), the node qualifies for deletion. The node is deleted by marking it as unschedulable and moving all Pods from it to other nodes.

Figure 29-5 summarizes how the CA interacts with cloud providers and Kubernetes for scaling out cluster nodes.

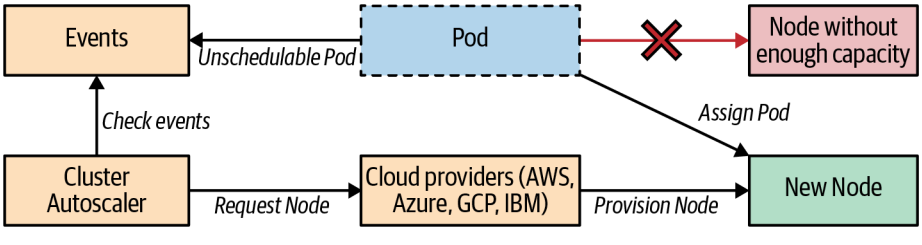


Figure 29-5. Cluster autoscaling mechanism

As you've probably figured out by now, scaling Pods and nodes are decoupled but complementary procedures. An HPA or VPA can analyze usage metrics and events, and scale Pods. If the cluster capacity is insufficient, the CA kicks in and increases the capacity. The CA is also helpful when irregularities occur in the cluster load due to batch Jobs, recurring tasks, continuous integration tests, or other peak tasks that require a temporary increase in the capacity. It can increase and reduce capacity and provide significant savings on cloud infrastructure costs.

Scaling Levels

In this chapter, we explored various techniques for scaling deployed workloads to meet their changing resource needs. While a human operator can manually perform most of the activities listed here, that doesn't align with the cloud native mindset. To enable large-scale distributed system management, automating repetitive activities is a must. The preferred approach is to automate scaling and enable human operators to focus on tasks that a Kubernetes Operator cannot automate yet.

Let's review all of the scaling techniques, from the more granular to the more coarse-grained order, as shown in Figure 29-6.

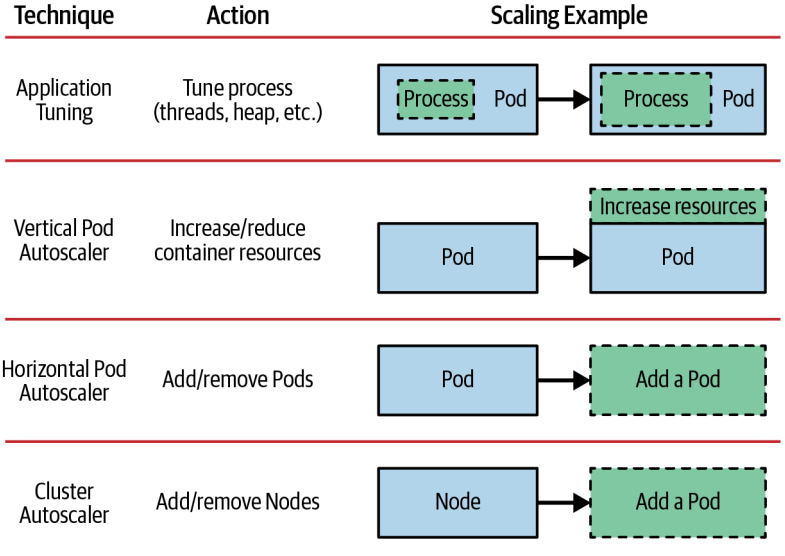


Figure 29-6. Application-scaling levels

Application tuning

At the most granular level, there is an application tuning technique we didn't cover in this chapter, as it is not a Kubernetes-related activity. However, the very first action you can take is to tune the application running in the container to best use the allocated resources. This activity is not performed every time a service is scaled, but it must be performed initially before hitting production. For example, for Java runtimes, that is right-sizing thread pools for best use of the available CPU shares the container is getting, then tuning the different memory regions such as heap, nonheap, and thread stack sizes. Adjusting these values is typically performed through configuration changes rather than code changes.

Container-native applications use start scripts that can calculate good default values for thread counts, and memory sizes for the application based on the allocated

container resources rather than the shared full-node capacity. Using such scripts is an excellent first step. You can also go one step further and use techniques and libraries such as the Netflix Adaptive Concurrency Limits library, where the application can dynamically calculate its concurrency limits by self-profiling and adapting. This is a kind of in-app autoscaling that removes the need for manually tuning services.

Tuning applications can cause regressions similar to a code change and must be followed by a degree of testing. For example, changing the heap size of an application can cause it to be killed with an `OutOfMemory` error, and horizontal scaling won't be able to help. On the other hand, scaling Pods vertically or horizontally, or provisioning more nodes, will not be as effective if your application is not consuming the resources allocated for the container properly. So tuning for scale at this level can impact all other scaling methods and can be disruptive, but it must be performed at least once for optimal application behavior.

Vertical Pod autoscaling

Assuming the application is consuming the container resources effectively, the next step is setting the right resource requests and limits in the containers. Earlier, we explored how VPA can automate the process of discovering and applying optimal values driven by real consumption. A significant concern here is that Kubernetes requires Pods to be deleted and created from scratch, which leaves the potential for short or unexpected periods of service disruption. Allocating more resources to a resource-starved container may make the Pod unschedulable and increase the load on other instances even more. Increasing container resources may also require application tuning to best use the increased resources.

Horizontal Pod autoscaling

The preceding two techniques are a form of vertical scaling; we hope to get better performance from existing Pods by tuning them but without changing their count. The following two techniques are a form of horizontal scaling: we don't touch the Pod specification, but we change the Pod and node count. This approach reduces the chances of introducing any regression and disruption and allows more straightforward automation. HPA, Knative, and KEDA are the most popular forms of horizontal scaling. Initially, HPA provided minimal functionality through CPU and memory metrics support only. Now it uses custom and external metrics for more advanced scaling use cases that allow scaling based on metrics that have an improved cost correlation.

Assuming that you have performed the preceding two methods once for identifying good values for the application setup itself and determined the resource consumption of the container, from there on, you can enable HPA and have the application adapt to shifting resource needs.

Cluster autoscaling

The scaling techniques described in HPA and VPA provide elasticity within the boundary of the cluster capacity only. You can apply them only if there is enough room within the Kubernetes cluster. CA introduces flexibility at the cluster capacity level. CA is complementary to the other scaling methods but is also completely decoupled. It doesn't care about the reason for extra capacity demand, or why there is unused capacity, or whether it is a human operator or an autoscaler that is changing the workload profiles. CA can extend the cluster to ensure demanded capacity or shrink it to spare some resources.

Discussion

Elasticity and the different scaling techniques are an area of Kubernetes that is still actively evolving. The VPA, for example, is still experimental. Also, with the popularization of the serverless programming model, scaling to zero and quick scaling have become a priority. Knative and KEDA are Kubernetes add-ons that exactly address this need to provide the foundation for scale-to-zero, as we briefly described in "Knative" on page 317 and "KEDA" on page 321. Those projects are progressing quickly and are introducing very exciting new cloud native primitives. We are watching this space closely and recommend you keep an eye on Knative and KEDA too.

Given a desired state specification of a distributed system, Kubernetes can create and maintain it. It also makes it reliable and resilient to failures, by continuously monitoring and self-healing and ensuring its current state matches the desired one. While a resilient and reliable system is good enough for many applications today, Kubernetes goes a step further. A small but properly configured Kubernetes system would not break under a heavy load but instead would scale the Pods and nodes. So in the face of these external stressors, the system would get bigger and stronger rather than weaker and more brittle, giving Kubernetes antifragile capabilities.

More Information

- Elastic Scale Example
- Rightsize Your Pods with Vertical Pod Autoscaling
- Kubernetes Autoscaling 101
- Horizontal Pod Autoscaling
- HPA Algorithm Details
- Horizontal Pod Autoscaler Walk-Through
- Knative
- Knative Autoscaling

- Knative: Serving Your Serverless Services
- KEDA
- Application Autoscaling Made Easy with Kubernetes Event-Driven Autoscaling (KEDA)
- Kubernetes Metrics API and Clients
- Vertical Pod Autoscaling
- Configuring Vertical Pod Autoscaling
- Vertical Pod Autoscaler Proposal
- Vertical Pod Autoscaler GitHub Repo
- Kubernetes VPA: Guide to Kubernetes Autoscaling
- Cluster Autoscaler
- Performance Under Load: Adaptive Concurrency Limits at Netflix
- Cluster Autoscaler FAQ
- Cluster API
- Kubermatic Machine-Controller
- OpenShift Machine API Operator
- Adaptive Concurrency Limits Library (Java)
- Knative Tutorial

CHAPTER 30

Image Builder

Kubernetes is a general-purpose orchestration engine, suitable not only for running applications but also for building container images. The *Image Builder* pattern explains why it makes sense to build the container images within the cluster and what techniques exist today for creating images within Kubernetes.

Problem

All the patterns in this book so far have been about operating applications on Kubernetes. You've learned how to develop and prepare applications to be good cloud native citizens. However, what about *building* the application itself? The classic approach is to build container images outside the cluster, push them to a registry, and refer to them in the Kubernetes Deployment descriptors. However, building within the cluster has several advantages.

If your company policies allow, having only one cluster for everything is advantageous. Building and running applications in one place can considerably reduce maintenance costs. It also simplifies capacity planning and reduces platform resource overhead.

Typically, continuous integration (CI) systems like Jenkins are used to build images. Building with a CI system is a scheduling problem for efficiently finding free computing resources for build jobs. At the heart of Kubernetes is a highly sophisticated scheduler that is a perfect fit for this kind of scheduling challenge.

Once we move to continuous delivery (CD), where we transition from *building* images to *running* containers, if the build happens within the same cluster, both phases share the same infrastructure and ease transition. For example, let's assume that a new security vulnerability is discovered in a base image used for all applications. As soon as your team has fixed this issue, you have to rebuild all the application images

that depend on this base image and update your running applications with the new image. When implementing this *Image Builder* pattern, the cluster knows both—the build of an image and its deployment—and can automatically do a redeployment if a base image changes. In "OpenShift Build" on page 346, we'll see how OpenShift implements such automation.

Having seen the benefits of building images on the platform, let's look at what techniques exist for creating images in a Kubernetes cluster.

Solution

As of 2023, a whole zoo of in-cluster container image-build techniques exists. While all target the same goal of building images, each tool adds a twist, making it unique and suitable for specific situations.

Figure 30-1 contains the essential image-building techniques as of 2023 for building container images within a Kubernetes cluster.

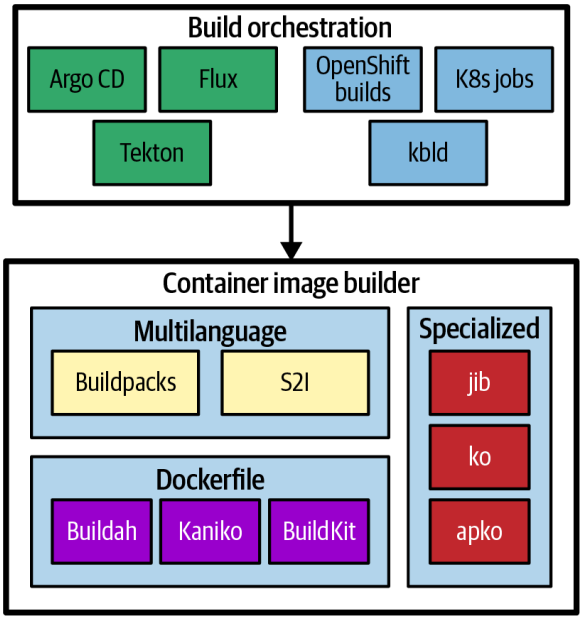


Figure 30-1. Container image builds within Kubernetes

This chapter contains a brief overview of most of these techniques. You can find more details about these tools by following the links in "More Information" on page 353. Please note that while many of the tools described here are matured and used in production projects, there are no guarantees that some of those projects still exist

when you read these lines. Before using one, you should check whether the project is still alive and supported.

Categorizing these tools is not straightforward as they are partly overlapping or dependent on one another. Each of these tools has a unique focus, but for in-cluster builds, we can identify these high-level categories:

Container image builder
: These tools create container images within the cluster. There is some overlap of these tools, and they vary, but all of them can run without privileged access. You can also run these tools outside the cluster as CLI programs. The sole purpose of these builders is to create a container image, but they don't care about application redeployments.

Build orchestration
: These tools operate on a higher level of abstraction and eventually trigger the container image builder for creating images. They also support build-related tasks like updating the deployment descriptors after the image has been built. CI/CD systems, as described previously, are typical examples of orchestrators.

Container Image Builder

One of the essential prerequisites for building images from within a cluster is creating images without having privileged access to the node host. Various tools exists that fulfill this prerequisite, and they can be roughly categorized according to how the container image is specified and built.

Rootless Builds

When building within Kubernetes, the cluster has complete control over the build process. Because of this, the cluster needs higher security standards to protect against potential vulnerabilities. One way to improve security during builds is to run them without root privileges, a practice known as *rootless builds*. There are many ways to achieve rootless builds in Kubernetes that allow you to build without elevated privileges.

Docker successfully brought container technologies to the masses thanks to its unmatched user experience. Docker is based on a client-server architecture with a daemon running in the background and taking instructions via a REST API from its client. This daemon needs root privileges mainly for network and volume management reasons. Unfortunately, this imposes a security risk, as untrusted processes can escape their container, and an intruder could get control of the whole host. This concern applies not only when running containers but also when building container images because building also happens within a container when the Docker daemon executes arbitrary commands.

Most of the in-cluster build techniques described in this chapter allow container images to be built in a nonprivileged mode to reduce that attack surface, which is very useful for locked-down Kubernetes clusters.

Dockerfile-Based builders

The following builders are based on the well-known Dockerfile format for defining the build instructions. All of them are compatible on a Dockerfile level, and they either work completely without talking to a background daemon or talk via a REST API remotely with a build process that is running in a nonprivileged mode:

Buildah and Podman
: Buildah and its sister Podman are potent tools for building OCI-compliant images without a Docker daemon. They create images locally within the container before pushing them to an image registry. Buildah and Podman overlap in functionality, with Buildah focusing on building container images (though Podman can also create container images by wrapping the Buildah API). The difference is shaped more clearly in this README.

Kaniko
: Kaniko is one backbone of the Google Cloud Build service and is deliberately targeted for running as a build container in Kubernetes. Within the build container, Kaniko still runs with UID 0, but the Pod holding the container itself is nonprivileged. This requirement prevents the usage of Kaniko in clusters that disallow running as a root user in a container, like in OpenShift. We see Kaniko in action in "Build Pod" on page 342.

BuildKit
: Docker extracted its build engine into a separate project, BuildKit, which can be used independently of Docker. It inherits from Docker its client-server architecture with a BuildKit daemon running in the background, waiting for build jobs. Usually, this daemon runs directly in the container that triggers the build, but it can also run in a Kubernetes cluster to allow distributed rootless builds. BuildKit introduces a Low-Level Build (LLB) definition format supported by multiple frontends. LLB allows complex build graphs and can be used for arbitrary complex build definitions. BuildKit also supports features that go beyond the original Dockerfile specification. In addition to Dockerfiles, BuildKit can use other frontends to define the container image's content via LLB.

Multilanguage builders

Many developers care only that their application gets packaged as container images and not so much about how this is done. To cover this use case, multilanguage builders exist to support many programming platforms. They detect an existing project,

like a Spring Boot application or generic Python build, and select an opinionated image build flow accordingly.

Buildpacks have been around since 2012 and were initially introduced by Heroku to allow you to push developer's code directly to their platform. Cloud Foundry picked up that idea and created a fork of buildpacks that eventually led to the infamous `cf push` idiom that many considered the gold standard of Platform as a Service (PaaS). In 2018, the different forks of Buildpacks united under the umbrella of the CNCF and are now known as *Cloud Native Buildpacks* (CNB). Besides individual buildpacks for different programming languages, CNB introduce a lifecycle for transforming source code to executable container images.

The lifecycle can roughly be divided into three main phases:[1]

- In the *detect* phase, CNB iterate over a list of configured buildpacks. Each buildpack can decide whether it fits for the given source code. For example, a Java-based buildpack will raise its hand when it detects a Maven *pom.xml*.
- All buildpacks that survived the detect phase will be called in the *build* phase to provide their part for the final, possibly compiled artifact. For example, a buildpack for a Node.js application calls `npm install` to fetch all required dependencies.
- The last step in the CNB lifecycle is an *export* to the final OCI image that gets pushed to a registry.

CNB target two personas. The primary audience includes *Developers* who want to deploy their code onto Kubernetes or any other container-based platform. The other is *Buildpack Authors*, who create individual buildpacks and group them into so-called *builders*. You can choose from a list of prefactored buildpacks and builders or create your own for you and your team. Developers can then pick up those buildpacks by referencing them when running the CNB lifecycle on their source code. Several tools are available for executing this lifecycle; you'll find a complete list at the Cloud Native Buildpacks site.

For using CNB within a Kubernetes cluster, the following tasks are helpful:

- `pack` is a CLI command to configure and execute the CNB lifecycle locally. It requires access to an OCI container runtime engine like Docker or Podman to run Builder images that hold the list of buildpacks to use.
- CI steps like Tekton build tasks or GitHub actions that call the lifecycle directly from a configured Builder image.

1 CNB cover more phases. The entire lifecycle is explained on the Buildpacks site.

- `kpack` comes with an Operator that allows you to configure and run buildpacks within a Kubernetes cluster. All the core concepts of CNB, like Builder or Buildpacks, are reflected directly as CustomResourceDefinitions. `kpack` is not yet part of the CNB project itself, but as of 2023 is about to be absorbed.

Many other platforms and projects have adopted CNB as their build platform of choice. For example, Knative Functions use CNB under the hood to transform Function code to container images before they get deployed as Knative services.

OpenShift's Source-to-Image (S2I) is another opinionated building method with builder images. S2I takes you directly from your application's source code to executable container images. We will look closely at S2I in "OpenShift Build" on page 346.

Specialized builders

Finally, specialized builders with an opinionated way of creating images exist for specific situations. While their scope is narrow, their strong opinion allows for a highly optimized build flow that increases flexibility and decreases build times. All these builders perform a rootless build. They create the container image without running arbitrary commands as with a Dockerfile `RUN` directive. They create the image layers locally with the application artifacts and push them directly to a container image registry:

Jib
: Jib is a pure Java library and build extension that integrates nicely with Java build tools like Maven or Gradle. It creates separate image layers directly for the Java build artifacts, its dependencies, and other static resources to optimize image rebuild times. Like the other builders, it speaks directly with a container image registry for the resulting images.

ko
: For creating images from Golang sources, ko is a great tool. It can directly create images from remote Git repositories and update Pod specifications to point to the image after it has been built and pushed to a registry.

Apko
: Apko is a unique builder that uses Alpine's Apk packages as building blocks instead of Dockerfile scripts. This strategy allows for the easy reuse of building blocks when creating multiple similar images.

This list is only a selection of the many specialized build techniques. All of them have a very narrow scope of what they can build. The advantage of this opinionated approach is that they can optimize build time and image size because they know precisely about the domain in which they operate and can make strong assumptions.

Now that we have seen some ways to build container images, let's jump one abstraction level higher and see how we can embed the actual build in a broader context.

Build Orchestrators

Build orchestrators are CI and CD platforms like Tekton, Argo CD, or Flux. Those platforms cover your application's entire automated management lifecycle, including building, testing, releasing, deploying, security scanning, and much more. There are excellent books that cover those platforms and bring it all together, so we won't go into the details here.

In addition to general-purpose CI and CD platforms, we can use more specialized orchestrators to create container images:

OpenShift builds
: One of the oldest and most mature ways of building images in a Kubernetes cluster is the *OpenShift build* subsystem. It allows you to build images in several ways. We take a closer look at the OpenShift way of building images in "OpenShift Build" on page 346.

kbld
: kbld is part of Carvel, a toolset for building, configuring, and deploying on Kubernetes. kbld is responsible for building containers with one of the builder technologies we described in "Container Image Builder" on page 337 and updating resource descriptors with a reference to the images that have been built. The technique for updating the YAML files is very similar to how ko works: kbld looks for `image` fields and sets their values to the coordinates of the freshly built image.

Kubernetes Job
: You can also use standard Kubernetes Jobs for triggering builds with any of the image builders from "Container Image Builder" on page 337. Jobs are described in detail in Chapter 7, "Batch Job". Such a Job wraps a build Pod specification for defining the runtime parts. The build Pod picks up the source code from a remote source repository and uses one of the in-cluster builders to create the appropriate image. We'll see such a Pod in action in "Build Pod" on page 342.

What Happened to Knative Build?

In the first edition of this book, we described Knative Build as one possibility for creating container images from within the cluster. As time has shown, Knative as an umbrella project was too small for the community, so Knative Build was split off from Knative and transformed into a new project, Tekton, with a much larger scope than only building container images. Tekton a is a full-featured CI solution that

fully integrates into Kubernetes and uses CustomResourceDefinitions as described in Chapter 28 as the basis for the description of the CI pipelines.

While Knative Build is history now, it was an excellent lesson about how open source communities evolve and can transform in unexpected ways. Keep this in mind, as it might happen to other popular projects too.

Build Pod

To carve out the essential ingredients of typical in-cluster builds, let's start minimally and use a Kubernetes Pod for performing a complete build and deploy cycle. These build steps are illustrated in Figure 30-2.

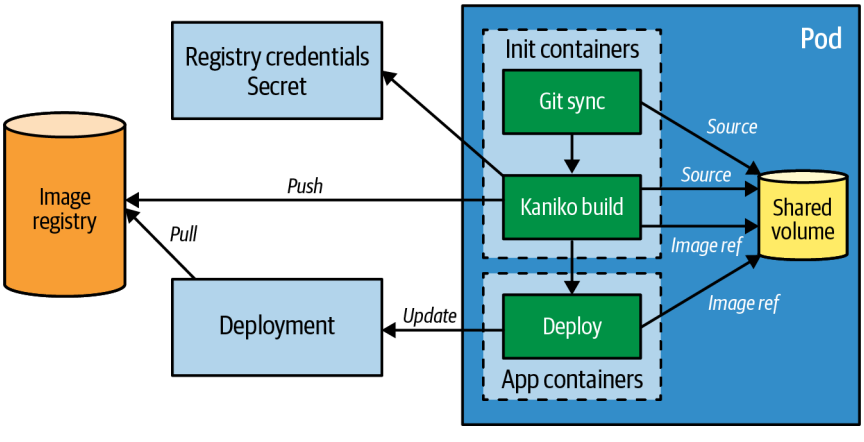


Figure 30-2. In-cluster container image build with a build Pod

The following tasks are representative of all build orchestrators and cover all aspects of creating container images:

- Check out the source code from a given remote Git repository.
- For a compiled language, perform a local build within the container.
- Build the application with one of the techniques described in "Container Image Builder" on page 337.
- Push the image to a remote image registry.
- Optionally, update a deployment with the new image reference, which will trigger a redeployment of the application following the strategies described in Chapter 3, "Declarative Deployment".

The build Pod in our example uses init containers as described in Chapter 15, "Init Container", to ensure that the build steps are running one after the other. In a

real-world scenario, you would use a CI system like Tekton to specify and execute these tasks sequentially.

The complete build Pod definition is shown in Example 30-1.

Example 30-1. Build Pod using Kaniko

```
apiVersion: v1
kind: Pod
metadata:
  name: build
spec:
  initContainers:
  - name: git-sync                    ❶
    image: k8s.gcr.io/git-sync/git-sync
    args: [
      "--one-time",
      "--depth", "1",
      "--root", "/workspace",
      "--repo", "https://github.com/k8spatterns/random-generator.git",
      "--dest", "main",
      "--branch", "main"]
    volumeMounts:                     ❷
    - name: source
      mountPath: /workspace
  - name: build                       ❸
    image: gcr.io/kaniko-project/executor
    args:
    - "--context=dir:///workspace/main/"
    - "--destination=index.docker.io/k8spatterns/random-generator-kaniko"
    - "--image-name-with-digest-file=/workspace/image-name"
    securityContext:
      privileged: false               ❹
    volumeMounts:
    - name: kaniko-secret             ❺
      mountPath: /kaniko/.docker
    - name: source                    ❻
      mountPath: /workspace
  containers:
  - name: image-update                ❼
    image: k8spatterns/image-updater
    args:
    - "random"
    - "/opt/image-name"
    volumeMounts:
    - name: source
      mountPath: /opt
  volumes:
  - name: kaniko-secret               ❽
    secret:
      secretName: registry-creds
```

```
    items:
    - key: .dockerconfigjson
      path: config.json
- name: source                ❾
  emptyDir: {}
serviceAccountName: build-pod  ❿
restartPolicy: Never           ⓫
```

❶ Init container for fetching the source code from a remote Git repository.

❷ Volume in which to store the source code.

❸ Kaniko as build container, storing the created image as a reference in the shared workspace.

❹ Build is running unprivileged.

❺ Secret for pushing to Docker Hub registry mounted at a well-known path so that Kaniko can find it.

❻ Mounting shared workspace for getting the source code.

❼ Container for updating the deployment `random` with the image reference from the Kaniko build.

❽ Secret volume with the Docker Hub credentials.

❾ Definition of a shared volume as an empty directory on the node's local filesystem.

❿ ServiceAccount that is allowed to patch a Deployment resource.

⓫ Never restart this Pod.

This example is quite involved, so let's break it down into three main parts.

First, before being able to build a container image, the application code needs to be fetched. In most cases, the source code is picked up from a remote Git repository, but other techniques are available. For development purposes, it is convenient to get the source code from your local machine so that you don't have to go over a remote source repository and mess up your commit history with triggering commits. Because the build happens within a cluster, that source code must be uploaded somehow to your build container. Another possibility is to distribute the source code packaged in a container image and distribute it via a container image registry.

In Example 30-1, we use an init container to fetch the source code from our source Git repository and store it in a shared Pod volume `source` of type `emptyDir` so that it can later be picked up by the build process container.

Second, after the application code is retrieved, the actual build happens. In our example, we use Kaniko, which uses a regular Dockerfile and can run entirely unprivileged. We again use an init container to ensure that the build starts only after the source code has been fully fetched. The container image is created locally on disk, and we also configure Kaniko to push the resulting image to a remote Docker registry.

The credentials for pushing to the registry are picked up from a Kubernetes Secret. We describe Secrets in detail in Chapter 20, "Configuration Resource".

Luckily, for the particular case of authentication against a Docker registry, we have direct support from `kubectl` for creating such a secret that stores this configuration in a well-known format:

```
kubectl create secret docker-registry registry-creds \
    --docker-username=k8spatterns \
    --docker-password********* \
    --docker-server=https://index.docker.io/
```

For Example 30-1, such a secret is mounted into the build container under a given path so that Kaniko can pick it up when pushing the created image. In Chapter 25, "Secure Configuration", we explain how such a secret can be stored securely so that it can't be forged.

The final step is to update an existing Deployment with the newly created image. This task is now performed in the actual application container of the Pod.[2] The referenced image is from our example repository and contains just a `kubectl` binary that patches the specified Deployment with the new image name with the following call, shown in Example 30-2.

Example 30-2. Update image field in Deployment

```
IMAGE=$(cat $1)                    ❶
PATCH=<<EOT                        ❷
[{
  "op":    "replace",
  "path":  "/spec/template/spec/containers/0/image",
  "value": "$IMAGE"
}]
EOT
```

2 We could have also chosen an init container again here and used a no-op application container, but since the application containers start only after all init containers have been finished, it doesn't matter much where we put the container. In all cases, the three specified containers run after one another.

```
kubectl patch deployment $2 \ ❸
   --type="json" \
   --patch=$PATCH
```

❶ Pickup image name stored by the previous build step in the file */opt/image-name*. This file is provided as the first argument to this script.

❷ JSON path to update the Pod spec with the new image reference.

❸ Patch the deployment given as the second argument (`random` in our example) and trigger a new rollout.

The Pod's assigned ServiceAccount `build-pod` is set up so it can write to this Deployment. Assigning permissions to a ServiceAccount is described fully in Chapter 26, "Access Control". When the image reference is updated in the Deployment, a rollout as described in Chapter 3, "Declarative Deployment", is performed.

You can find the fully working setup in the book's example repository. The build Pod is the simplest way to orchestrate an in-cluster build and redeployment. As mentioned, it is meant for illustrative purposes only.

For real-world use cases, you should use a CI/CD solution like Tekton or a whole build orchestration platform like OpenShift Build, which we describe now.

OpenShift Build

Red Hat OpenShift is an enterprise distribution of Kubernetes. Besides supporting everything Kubernetes supports, it adds a few enterprise-related features like an integrated container image registry, single sign-on support, and a new user interface, and it also adds a native image building capability to Kubernetes. OKD is the upstream open source community edition distribution that contains all the OpenShift features.

OpenShift build was the first cluster-integrated way of directly building images managed by Kubernetes. It supports multiple strategies for building images:

Source-to-Image (S2I)
: Takes the source code of an application and creates the runnable artifact with the help of a language-specific S2I builder image and then pushes the images to the integrated registry.

Docker builds
: Use a Dockerfile plus a context directory and creates an image as a Docker daemon would do.

Pipeline builds
: Map build-to-build jobs of an internally managed Tekton by allowing the user to configure a Tekton pipeline.

Custom builds
: Give you full control over how you create your image. Within a custom build, you have to create the image on your own within the build container and push it to a registry.

The input for doing the builds can come from different sources:

Git
: Repository specified via a remote URL from where the source is fetched.

Dockerfile
: A Dockerfile that is directly stored as part of the build configuration resource.

Image
: Another container image from which files are extracted for the current build. This source type allows for *chained builds*, as shown in Example 30-4.

Secret
: Resource for providing confidential information for the build.

Binary
: Source to provide all input from the outside. This input has to be provided when starting the build.

The choice of which input sources we can use in which way depends on the build strategy. *Binary* and *Git* are mutually exclusive source types. All other sources can be combined or used on a standalone basis. We will see later in Example 30-3 how this works.

All the build information is defined in a central resource object called BuildConfig. We can create this resource either by directly applying it to the cluster or by using the CLI tool `oc`, which is the OpenShift equivalent of `kubectl`. `oc` supports build-specific commands for defining and triggering a build.

Before we look at BuildConfig, we need to understand two additional concepts specific to OpenShift.

An ImageStream is an OpenShift resource that references one or more container images. It is a bit similar to a Docker repository, which also contains multiple images with different tags. OpenShift maps an actual tagged image to an ImageStreamTag resource so that an ImageStream (repository) has a list of references to ImageStreamTags (tagged images). Why is this extra abstraction required? Because it allows OpenShift to emit events when an image is updated in the registry for an ImageStreamTag. Images are created during builds or when an image is pushed to the OpenShift internal registry. That way, the build or deployment controllers can listen to these events and trigger a new build or start a deployment.

To connect an ImageStream to a deployment, OpenShift uses the DeploymentConfig resource instead of the Kubernetes Deployment resource, which can only use container image references directly. However, you can still use vanilla Deployment resources in OpenShift with ImageStreams by adding some OpenShift-specific annotations.

The other concept is a *trigger*, which we can consider as a kind of listener to events. One possible trigger is `imageChange`, which reacts to the event published because of an ImageStreamTag change. As a reaction, such a trigger can, for example, cause the rebuild of another image or redeployment of the Pods using this image. You can read more about triggers and the kinds of triggers available in addition to the `imageChange` trigger in the OpenShift documentation.

Source-to-Image

Let's have a quick look at what an S2I builder image looks like. We won't go into too many details here, but an S2I builder image is a standard container image that contains a set of S2I scripts. It is very similar to Cloud Native Buildpacks but with a much simpler lifecycle that knows two mandatory commands:

`assemble`
: The script that gets called when the build starts. Its task is to take the source given by one of the configured inputs, compile it if necessary, and copy the final artifacts to the proper locations.

`run`
: Used as an entry point for this image. OpenShift calls this script when it deploys the image. This run script uses the generated artifacts to deliver the application services.

Optionally, you can also script to provide a usage message, saving the generated artifacts for so-called *incremental builds* that are accessible by the `assemble` script in a subsequent build run, or add some sanity checks.

Let's have a closer look at an S2I build in Figure 30-3. An S2I build has two ingredients: a builder image and a source input. Both are brought together by the S2I build system when a build is started—either because a trigger event was received or because we started it manually. When the build image has finished by, for example, compiling the source code, the container is committed to an image and pushed to the configured ImageStreamTag. This image contains the compiled and prepared artifacts, and the image's `run` script is set as the entry point.

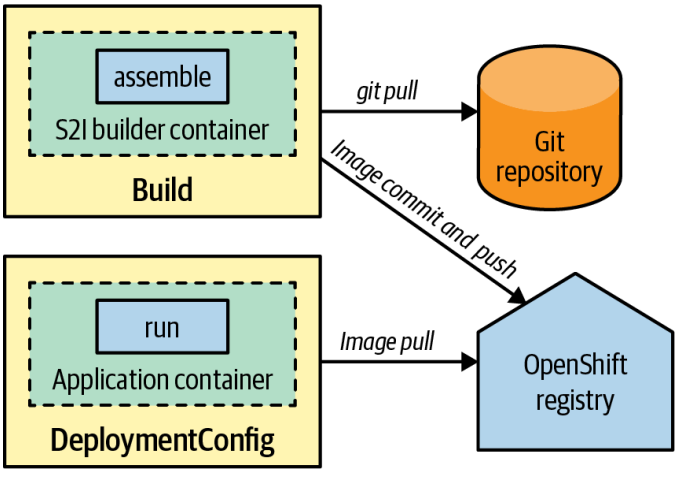


Figure 30-3. S2I build with Git source as input

Example 30-3 shows a simple Java S2I build with a Java S2I image. This build takes a source, the builder image, and produces an output image that is pushed to an ImageStreamTag. It can be started manually via `oc start-build` or automatically when the builder image changes.

Example 30-3. S2I Build using a Java builder image

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: random-generator-build
spec:
  source:    ❶
    git:
      uri: https://github.com/k8spatterns/random-generator
  strategy:  ❷
    sourceStrategy:
      from:
        kind: DockerImage
        name: fabric8/s2i-java
  output:    ❸
    to:
      kind: ImageStreamTag
      name: random-generator-build:latest
  triggers:  ❹
  - type: GitHub
    github:
      secretReference: my-secret
```

❶ Reference to the source code to fetch; in this case, pick it up from GitHub.

❷ `sourceStrategy` switches to S2I mode, and the builder image is picked up directly from Docker Hub.

❸ The ImageStreamTag to update with the generated image. It's the committed builder container after the `assemble` script has run.

❹ Rebuild automatically when the source code in the repository changes.

S2I is a robust mechanism for creating application images, and it is more secure than plain Docker builds because the build process is under full control of trusted builder images. However, this approach still has some drawbacks.

For complex applications, S2I can be slow, especially when the build needs to load many dependencies. Without any optimization, S2I loads all dependencies afresh for every build. In the case of a Java application built with Maven, there is no caching as when doing local builds. To avoid downloading half of the internet again and again, it is recommended that you set up a cluster-internal Maven repository that serves as a cache. The builder image then has to be configured to access this common repository instead of downloading the artifacts from remote repositories.

Another way to decrease the build time is to use *incremental builds* with S2I, which allows you to reuse artifacts created or downloaded in a previous S2I build. However, a lot of data is copied over from the previously generated image to the current build container, and the performance benefits are typically not much better than using a cluster-local proxy that holds the dependencies.

Another drawback of S2I is that the generated image also contains the whole build environment.[3] This fact increases not only the size of the application image but also the surface for a potential attack, as builder tools can become vulnerable too.

To get rid of unneeded builder tools like Maven, OpenShift offers *chained builds*, which take the result of an S2I build and create a slim runtime image. We look at chained builds in "Chained builds" on page 351.

Docker builds

OpenShift also supports Docker builds directly within the cluster. Docker builds work by mounting the Docker daemon's socket directly in the build container, which is then used for a `docker build`. The source for a Docker build is a Dockerfile and a directory holding the context. You can also use an `Image` source that refers an

3 This is different from Cloud Native Buildpacks, which use a separate runtime image in its stack for carrying the final artifact.

arbitrary image and from which files can be copied into the Docker build context directory. As mentioned in the next section, this technique, together with triggers, can be used for chained builds.

Alternatively, you can use a standard multistage Dockerfile to separate the build and runtime parts. Our example repository contains a fully working multistage Docker build example that results in the same image as the chained build described in the next section.

Chained builds

The mechanics of a chained build are shown in Figure 30-4. A chained build consists of an initial S2I build, which creates the runtime artifact such as a binary executable. This artifact is then picked up from the generated image by a second build, typically a Docker build.

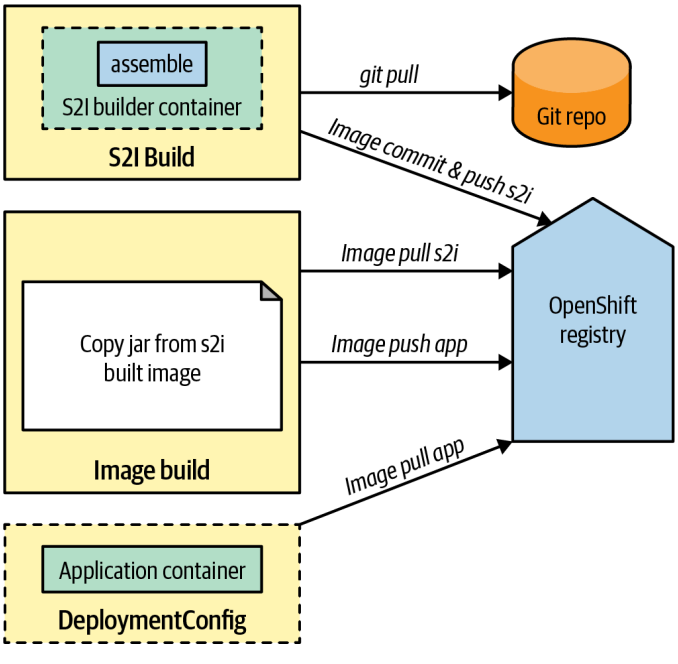


Figure 30-4. Chained build with S2I for compiling and Docker build for application image

Example 30-4 shows the setup of this second build config, which uses the JAR file generated in Example 30-3. The image that is eventually pushed to the ImageStream `random-generator-runtime` can be used in a DeploymentConfig to run the application.

The trigger used in Example 30-4 monitors the result of the S2I build. This trigger causes a rebuild of this runtime image whenever we run an S2I build so that both ImageStreams are always in sync.

Example 30-4. Docker build for creating the application image

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: runtime
spec:
  source:
    images:
    - from:                    ❶
        kind: ImageStreamTag
        name: random-generator-build:latest
      paths:
      - sourcePath: /deployments/.
        destinationDir: "."
    dockerfile: |-             ❷
      FROM openjdk:17
      COPY *.jar /
      CMD java -jar /*.jar
  strategy:                    ❸
    type: Docker
  output:                      ❹
    to:
      kind: ImageStreamTag
      name: random-generator:latest
  triggers:                    ❺
  - imageChange:
      automatic: true
      from:
        kind: ImageStreamTag
        name: random-generator-build:latest
    type: ImageChange
```

❶ Image source references the ImageStream that contains the result of the S2I build run and selects a directory within the image that contains the compiled JAR archive.

❷ Dockerfile source for the Docker build that copies the JAR archive from the ImageStream generated by the S2I build.

❸ The `strategy` selects a Docker build.

❹ Rebuild automatically when the S2I result ImageStream changes—after a successful S2I run to compile the JAR archive.

❺ Register listener for image updates, and do a redeploy when a new image has been added to the ImageStream.

You can find the full example with installation instructions in our example repository.

As mentioned, OpenShift build, along with its most prominent S2I mode, is one of the oldest and most mature ways to safely build container images within an OpenShift cluster.

Discussion

You have seen two ways to build container images within a cluster. The plain build Pod illustrates the most crucial tasks that every build system needs to execute: fetching the source code, creating a runnable artifact from your source code, creating a container image containing the application's artifacts, pushing this image to an image registry, and finally updating any deployments so that it picks up the newly created image from that registry. This example is not meant for direct production use as it contains too many manual steps that existing build orchestrators cover more effectively.

The OpenShift build system nicely demonstrates one of the main benefits of building and running an application in the same cluster. With OpenShift's ImageStream triggers, you can connect multiple builds and redeploy your application if a build updates your application's container image. Better integration between build and deployment is a step forward to the holy grail of CD. OpenShift builds with S2I are a proven and established technology, but S2I is usable only when using the OpenShift distribution of Kubernetes.

The landscape of in-cluster build tools as of 2023 is rich and contains many exciting techniques that partly overlap. As a result, you can expect some consolidation, but new tooling will arise over time, so we'll see more implementations of the *Image Builder* pattern emerge.

More Information

- Image Builder Example
- Image Builders:
  - Buildah
  - Kaniko

  - What Is BuildKit?
  - Building Multi-Architecture Images with Buildpacks
  - Jib
  - Pack
  - Kpack
  - Ko
  - Apko: A Better Way to Build Containers?
- Build Orchestrators:
  - OpenShift Builds
  - Kbld
- Multistage Build
- Chaining S2I Builds
- Build Triggers Overview
- Source-to-Image Specification
- Incremental S2I Builds
- Building Container Images in Kubernetes: It's Been a Journey!
- Build Multi-Architecture Container Images Using Kubernetes
- Best Practices for Running Buildah in a Container

Afterword

Kubernetes is the leading platform for deploying and managing containerized distributed applications at scale. However, these on-cluster applications rely on off-cluster resources, including databases, document stores, message queues, and other cloud services. Kubernetes is not limited to managing applications within a single cluster. Kubernetes can also orchestrate off-cluster resources through various cloud services' operators. This allows Kubernetes APIs to be the single "source of truth" for a resource's desired state, not only for on-cluster containers but also for off-cluster resources. If you are already familiar with Kubernetes patterns and practices for operating applications, you can leverage this knowledge for managing and using external resources too.

The physical boundaries of a Kubernetes cluster don't always conform to the desired application boundaries. Organizations often need to deploy applications across multiple data centers, clouds, and Kubernetes clusters for a variety of reasons, such as scaling, data locality, isolation, and more. Often, the same application or a fleet of applications has to be deployed into multiple clusters, which requires multicluster deployments and orchestration. Kubernetes is frequently embedded in various third-party services and used for operating applications across multiple clusters. These services utilize the Kubernetes API as the control plane, with each cluster serving as a data plane, allowing Kubernetes to extend its reach across multiple clusters.

Today, Kubernetes has evolved beyond just a container orchestrator. It is capable of managing on-cluster, off-cluster, and multicluster resources, making it a versatile and extensible operational model for managing many kinds of resources. Its declarative YAML API and asynchronous reconciliation process have become synonymous with the resource orchestration paradigm. Its CRDs and Operators have become common extension mechanisms for merging domain knowledge with distributed systems. We believe that the majority of modern applications will be running on platforms that are offering Kubernetes APIs, or on runtimes that are heavily influenced by Kubernetes abstractions and patterns. If you are a software developer creating such applications, you must be proficient in modern programming languages to implement business

functionality, as well as cloud native technologies. Kubernetes patterns will become mandatory common knowledge for integrating applications with the runtime platform. Familiarizing yourself with the Kubernetes patterns will enable you to create and run applications in any environment.

What We Covered

In this book, we covered the most popular patterns from Kubernetes, grouped as the following:

- *Foundational patterns* represent the principles that containerized applications must comply with in order to become good cloud native citizens. Regardless of the application nature, and the constraints you may face, you should aim to follow these guidelines. Adhering to these principles will help ensure that your applications are suitable for automation on Kubernetes.
- *Behavioral patterns* describe the communication mechanisms and interactions between the Pods and the managing platform. Depending on the type of workload, a Pod may run until completion as a batch job or be scheduled to run periodically. It can run as a stateless or stateful service and as a daemon service or singleton. Picking the right management primitive will help you run a Pod with the desired guarantees.
- *Structural patterns* focus on structuring and organizing containers in a Pod to satisfy different use cases. Having good cloud native containers is the first step but is not enough. Reusing containers and combining them into Pods to achieve a desired outcome is the next step.
- *Configuration patterns* cover customizing and adapting applications for different configuration needs on the cloud. Every application needs to be configured, and no one way works for all. We explore patterns from the most common to the most specialized.
- *Security patterns* describe how to constrain an application while intersecting with Kubernetes. Containerized applications have security dimensions too, and we cover application interactions with the nodes, interactions with other Pods, the Kubernetes API server, and secure configurations.
- *Advanced patterns* explore more complex topics that do not fit in any of the other categories. Some of the patterns, such as *Controller*, are mature—Kubernetes itself is built on it—and some are still evolving and might change by the time you read this book. But these patterns cover fundamental ideas that cloud native developers should be familiar with.

Final Words

Like all good things, this book has come to an end. We hope you have enjoyed reading this book and that it has changed the way you think about Kubernetes. We truly believe Kubernetes and the concepts originating from it will be as fundamental as object-oriented programming concepts are. This book is our attempt to create the Gang of Four Design Patterns but for container orchestration. We hope this is not the end but the beginning of your Kubernetes journey; it has been so for us.

Happy kubectl-ing.

Index

D

I

J

K

L

M

N

Q

R

S

T

About the Authors

Bilgin Ibryam (@bibryam) is a product manager at Diagrid, where he drives the company's product strategy in bringing the Dapr project to the enterprise. Previously, he was a consultant and architect at Red Hat, where he mentored and led teams to be successful in building highly scalable and resilient solutions. Bilgin is an open source evangelist and member of the Apache Software Foundation, a regular blogger, speaker, and author of *Camel Design Patterns* (Leanpub). Bilgin's mission is to make creating distributed systems a no-brainer for all developers. Find out more about his work on Twitter: *https://twitter.com/bibryam*.

Dr. Roland Huß (@ro14nd@hachyderm.io) is a seasoned software engineer with over 25 years of experience in the field. Currently working at Red Hat, he is the architect of OpenShift Serverless and a former member of the Knative TOC. Roland is a passionate Java and Golang coder and a sought-after speaker at tech conferences. An advocate of open source, he is an active contributor and enjoys growing chili peppers in his free time.

Colophon

The animal on the cover of *Kubernetes Patterns* is a red-crested whistling duck (*Netta rufina*). The species name *rufina* means "red-haired" in Latin. Another common name for them is "red-crested pochard," with *pochard* meaning "diving duck." The red-crested whistling duck is native to the wetlands of Europe and central Asia. Its population has also spread throughout northern African and south Asian wetlands.

Red-crested whistling ducks reach 1.5–2 feet in height and weigh 2–3 pounds when fully grown. Their wingspan is nearly 3 feet. Females have varying shades of brown feathers with a light face and are less colorful than males. A male red-crested whistling duck has a red bill, rusty orange head, black tail and breast, and white sides.

The red-crested whistling duck's diet primarily consists of roots, seeds, and aquatic plants. They build nests in the vegetation beside marshes and lakes and lay eggs in the spring and summer. A normal brood is 8–12 ducklings. Red-crested whistling ducks are most vocal during mating. The call of the male sounds more like a wheeze than a whistle, and the female's is a shorter "vrah, vrah, vrah."

The conservation status of the red-crested whistling duck is Least Concern. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black-and-white engraving from *British Birds*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.