O'REILLY®

# Kubernetes Patterns

## Reusable Elements for Designing Cloud-Native Applications


Bilgin Ibryam &
Roland Huß

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 cloud-ready apps. See how at redhat.com/cloud-native-development.



Hybrid Cloud | Cloud-Native Development
Automation | IT Optimization | Agile Integration

# 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:** Virginia Wilson
**Production Editor:** Katherine Tozer
**Copyeditor:** Christine Edwards
**Proofreader:** Sharon Wilkey

**Indexer:** Judith McConville
**Interior Designer:** David Futato
**Cover Designer:** Karen Montgomery
**Illustrator:** Rebecca Demarest

May 2019: First Edition

**Revision History for the First Edition**
2019-04-04: First Release
2021-04-02: Second Release

See *https://www.oreilly.com/catalog/errata.csp?isbn=9781492050285* for release details.



978-1-492-07665-0

[LSI]

# Table of Contents

## Part V. Advanced Patterns

# Foreword

When Craig, Joe, and I started Kubernetes nearly five 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 evolution of microservices and containers in recent years, the way we design, develop, and run software has changed significantly. Today's applications are optimized for scalability, elasticity, failure, and speed of change. Driven by new principles, these modern architectures require a different set of patterns and practices. This book aims to help developers create cloud-native applications with Kubernetes as a runtime platform. 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"), and in 2015, it became the first project donated to the newly founded Cloud Native Computing Foundation (CNCF).

Right from the start, Kubernetes gained a whole community of users, and the number of contributors grew at an incredibly fast pace. Today, Kubernetes is considered one of the most active projects on GitHub. It is probably fair to claim that at the time of this writing, 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, including ways to build applications within the platform. These are only some of the 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 out the foundation for the following patterns.

## Design Patterns

The concept of *design patterns* dates back to the 1970s and 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] It is different from a recipe because instead of giving step-by-step instructions to solving a problem, a pattern 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 with beverages, music, and other activities.

However, a pattern does more than provide a solution. It is also about forming a language. The unique pattern names 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 we talk about 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 Christopher 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, p. x). We think this definition works for the patterns we describe in this book, except that we probably don't have as much variability in our solutions.

There are also other characteristics of a pattern language. Patterns are interconnected and can overlap so that together, they cover most of the problem space. Also, as already laid out in the original *A Pattern Language*, patterns do not have the same 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 not as widely applicable. This book contains all sort 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 format, and unfortunately there is no common standard for the way patterns should be laid out. Martin Fowler gives an excellent overview of the formats used for pattern languages in *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 is about 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*
: A discussion about the advantages and disadvantages of the solution for the given context follows.

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

We organized the patterns of 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 sit on top of the foundational patterns and add finer-grained concepts for managing various types of container and platform interactions.
- 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 very granular patterns, including concrete recipes for connecting applications to their configuration.
- Part V, *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.

A pattern might not always fit into one category alone. Depending on the context, the same pattern might fit into several categories. Every pattern chapter is self-contained, and 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 for the Kubernetes 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, technical consultants, and developers will all benefit from the repeatable patterns described here.

This book is based on use cases and lessons learned from real-world projects. We want to help you create better cloud-native applications—not reinvent the wheel.

## What You Will Learn

There's a lot to discover in this book. Some of the 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 a different approach, with detailed guidelines for very concrete problems, as in "Configuration Patterns" in Part IV.

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 full source code in "Using Code Examples".

Before we start to dive in, let's briefly look at what this book is *not*:

- This book is not a guide on how to set up a Kubernetes cluster itself. Every pattern and 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 *Managing Kubernetes* by Brendan Burns and Craig Tracey (O'Reilly). Also, the *Kubernetes Cookbook* by Michael Hausenblas and Sébastien Goasguen (O'Reilly) has recipes for setting up a Kubernetes cluster from scratch.
- This book is not an introduction to Kubernetes, nor 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 how to use Kubernetes, we highly recommend *Kubernetes in Action* by Marko Lukša (Manning Publications).

The book is written in a relaxed style, and is similar to a series of essays that can be read independently.

## Conventions

As mentioned, patterns form a kind of simple, interconnected language. To emphasize this web of patterns, each pattern is capitalized in *italics*, (e.g., *Sidecar*). When a pattern is named like a Kubernetes core concept (like *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 `fixed font width`.
- 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.

## 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 the "More Information" section of each chapter.

The "More Information" section also contains many links to further information related to the pattern. We keep these lists updated in the example repository. Changes to the link collections will also be posted on Twitter.

The source code for all examples in this book is available at GitHub. The repository and the website also have pointers and instructions on how to get a Kubernetes cluster to try out the examples. When you go through the examples, please also have a look into the provided resource files. They contain many useful comments that help further in understanding the example code.

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

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.

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



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 almost 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, conferences, 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
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)


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*.

To comment or ask technical questions about this book, email *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

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

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

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

Creating this book was a long journey spanning over two years, and we want to thank all of our reviewers who kept us on the right track. Special kudos go out to Paolo Antinori and Andrea Tarocchi for helping us through the whole journey. Big thanks also 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 advices. 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.

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 container-based cloud-native applications. Understanding these new abstractions, and the related principles and patterns from this book, are key to building distributed applications made automatable by cloud-native platforms.

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

The most popular application architecture on the cloud-native platforms such as Kubernetes is the microservices style. This software development technique tackles software complexity through modularization of business capabilities and trading development complexity for operational complexity.

As part of the microservices movement, there is a significant amount of theory and supplemental techniques for creating microservices from scratch or for splitting monoliths into microservices. Most of these practices are based on the *Domain-Driven Design* book 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 further to group bounded contexts into modules with defined transaction boundaries. However, in addition to these business domain considerations, for every distributed system—whether it is based on microservices or not—there are also numerous technical concerns around its organization, structure, and runtime behavior.

Containers and container orchestrators such as Kubernetes provide many 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.

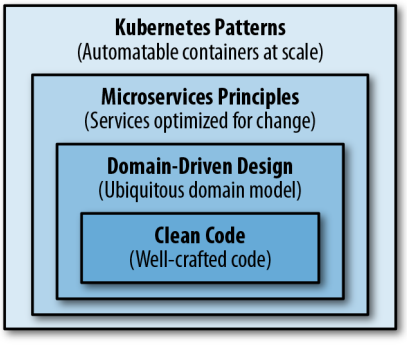


*Figure 1-1. The path to cloud native*

At a high level, there are multiple abstraction levels in a cloud-native application that require different design considerations:

- 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 amount of automated tests, constantly refactor to improve code quality, and are software craftsmen 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 *microservices architectural style* very quickly evolved to become the norm, and it provides 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. Creating modular, reusable containers that are good cloud-native citizens is another fundamental prerequisite. With a growing number of containers in every organization comes the need to manage them using more effective methods and tools. *Cloud native* is a relatively new term used to describe principles, patterns, and tools to automate containerized microservices 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, 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 applying clean code practices, domain-driven design, microservices patterns, 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.

*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 | Defer container[a] |
| Asynchronous & parallel execution | `ThreadPoolExecutor`, `ForkJoinPool` | Job |
| Periodic task | `Timer`, `ScheduledExecutorService` | CronJob |
| Background task | Daemon thread | DaemonSet |
| Configuration management | `System.getenv()`, `Properties` | ConfigMap, Secret |

[a] Defer (or de-init) containers are not yet implemented, but there is a proposal on the way to include this feature in future versions of Kubernetes. We discuss lifecycle hooks in Chapter 5, *Managed Lifecycle*.

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 replace the `ExecutorService` behavior in Java completely.

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 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 each other 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 a 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 has defined runtime dependencies and resource requirements.
- A container image has well-defined APIs to expose its functionality.
- A container runs typically 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 it is going to run. But it is also parameterized for its various use cases. 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, deployed together whether for scaling or host migration purposes, and can also share filesystem, networking, and process namespaces. This joint lifecycle allows the containers in a Pod to interact with each other over the filesystem or through networking via localhost or host interprocess communication mechanisms if desired (for performance reasons, for example).

As you can see in Figure 1-2, at the 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. One such example is when a containerized microservice uses a helper container at runtime, as Chapter 15, *Sidecar* demonstrates later.

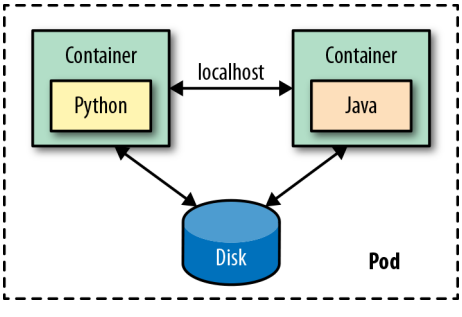


*Figure 1-2. A Pod as the deployment and management unit*

Containers and Pods and their unique characteristics offer a new set of patterns and principles for designing microservices-based applications. We looked at some of the characteristics of well-designed containers; now let's look at some of the 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 (there are some specifics around init containers, which we cover in Chapter 14, *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 each other. The most common ways for communication include using a shared local filesystem for

exchanging data or using the localhost network interface, or 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 can come and go at any time for all sort of reasons such as scaling up and down, failing container health checks, and 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. All that means is 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 allows altering implementations and scaling without affecting Service consumers. We explain Services in detail in Chapter 12, *Service Discovery*.

## Labels

We have seen that a microservice is a container at build time but 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. We cover namespaces in detail in "Namespaces" on page 9.

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 the form of a *.war*, or *.ear* or some other packaging format. But then, applications got split into microservices, which are independently developed, released, run, restarted, or scaled. With microservices, the notion of an application diminishes, and there are no longer key artifacts or activities that we have to perform at the application

level. However, 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 application 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 each other. 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.

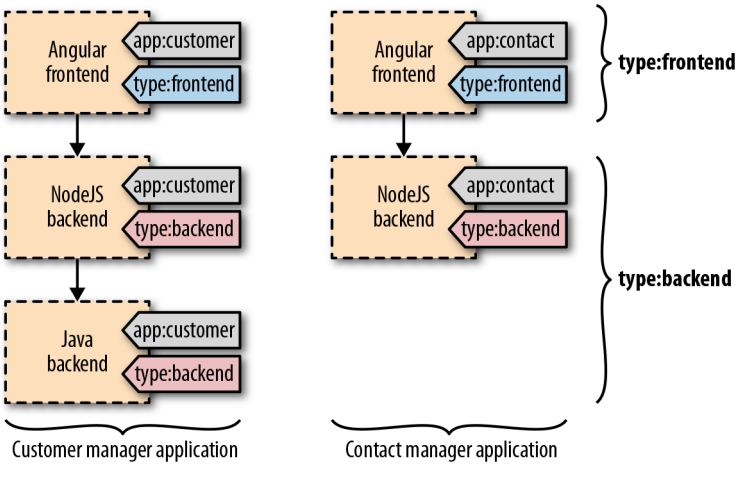


*Figure 1-3. Labels used as an application identity for Pods*

Here are a few examples where labels can be useful:

- Labels are used by the 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 used heavily by the scheduler. The scheduler uses labels for colocating or spreading Pods to place Pods on the nodes that satisfy the Pods' requirements.

- A Label can indicate a logical grouping of 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 straight-forward 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 called *annotations*. 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 in the management of 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 purpose.

Kubernetes namespaces allow dividing 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 isola-

tion 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 see 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. However, there are Kubernetes plugins that provide networking isolation to achieve true multitenancy across namespaces if desired.
- Some other resources such as namespaces themselves, 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 DNS address that has the namespace in the form of `<service-name>.<namespace-name>.svc.cluster.local`. So the namespace name is in the URI 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 half of the resources—16 GB RAM and 8 cores—for the production namespace, 8 GB RAM and 4 cores for staging environment, 4 GB RAM and 2 cores for development, and the same amount for testing namespaces. The ability of imposing resource constraints on a group of objects by using namespaces and ResourceQuotas 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 collections of Kubernetes objects you can use to reap all the benefits of Kubernetes. Keep in mind, these are only the objects used by application developers to integrate a containerized service into Kubernetes. There are also other concepts used by administrators to enable developers to manage the platform effectively. Figure 1-4 gives an overview of the multitude of Kubernetes resources that are useful for developers.

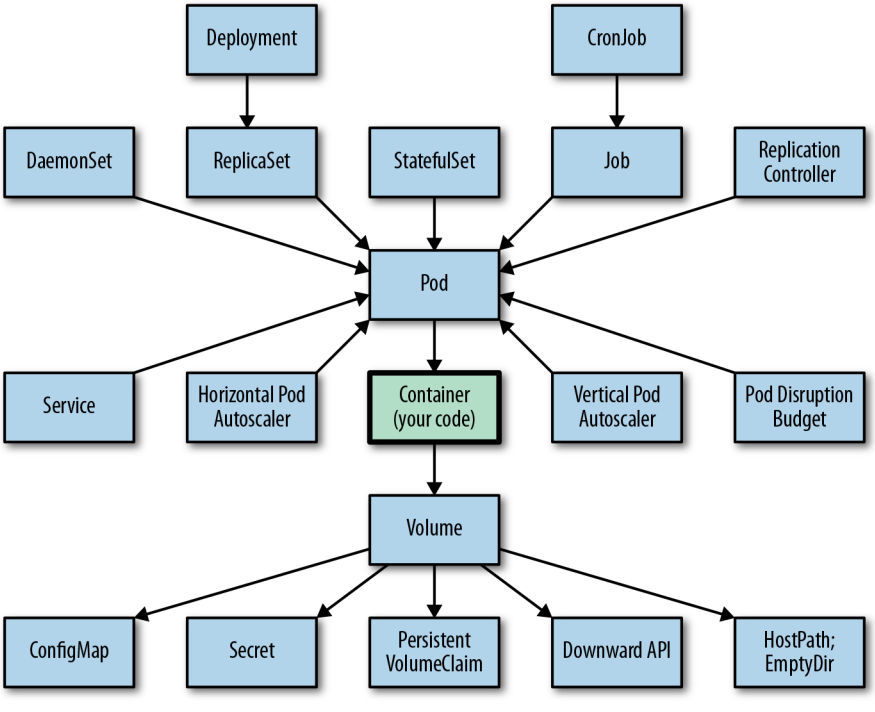


*Figure 1-4. Kubernetes concepts for developers*

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 a Kubernetes resource in detail, we will focus on Kubernetes aspects that are proven as patterns.

## More Information

- Principles of Container-Based Application Design
- The Twelve-Factor App
- Domain-Driven Design: Tackling Complexity in the Heart of Software
- Container Best Practices
- Best Practices for Writing Dockerfiles
- Container Patterns
- General Container Image Guidelines
- Pods

# 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 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 profile and stay confined to the indicated resource requirements.
- Chapter 3, *Declarative Deployment*, shows the different application deployment strategies that can be performed in a declarative way.
- Chapter 4, *Health Probe*, dictates that every container should implement specific APIs to help the platform observe and manage the application in the healthiest way possible.
- Chapter 5, *Managed Lifecycle*, describes 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 a pattern for distributing containers in a Kubernetes multinode cluster.

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 is about 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.

It is difficult to predict the amount of resources a container may need to function optimally, and it is the developer who knows the resource expectations of a service implementation (discovered through testing). Some services have a fixed CPU and memory consumption profile, and some are spiky. Some services need persistent storage to store data; some legacy services require a fixed port number on the host system to work correctly. Defining all these application characteristics and passing them to the managing platform is a fundamental prerequisite for cloud-native applications.

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 for where to place a container on the cluster for most efficient hardware utilization. In an environment with shared resources among a large number of processes with different priorities, the only way for a successful coexistence is to know the demands of every process in advance. However, intelligent placement is only one side of the coin.

The second reason container resource profiles are essential is capacity planning. Based on the particular service demands and the total number of services, we can do some capacity planning for the 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-to-hand for successful cluster management in the long term.

Let's have a look first at how to declare runtime dependencies before we dive into resource profiles.

## Runtime Dependencies

One of the most common runtime dependencies is file storage for saving application state. Container filesystems are ephemeral and 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 and when the Pod is removed, its content is also lost. The volume needs to be backed by some other kind of storage mechanism to have a volume that survives Pod restarts. If your application needs to read or write files to such long-lived storage, you have to 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 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 limit to maximum one Pod scheduled per node. Because of port conflicts, you can scale to as many Pods as there are nodes in the Kubernetes cluster.

A different type of dependency is configurations. 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. ConfigMaps and Secrets are explained in more details in Chapter 19, *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 Dependency of a ConfigMap to be present

A similar concept to ConfigMaps are Secrets, which offer a slightly more secure way of distributing environment-specific configurations to a container.[1] The way to consume a Secret is the same as it is for ConfigMap consumption, and it introduces the same kind of dependency from a container to a namespace.

While the creation of ConfigMap and Secret objects are simple admin 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 resources 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.

1 We talk more about Secret security in Chapter 19, *Configuration Resource*.

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: 100Mi
      limits:                            ❷
        cpu: 200m
        memory: 200Mi
```

❶ Initial resource request for CPU and memory

❷ Upper limit until we want our application to grow at max

Depending on whether you specify the `requests`, the `limits`, or both, the platform offers a different kind of Quality of Service (QoS).

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

*Burstable*
: Pod that has `requests` and `limits` defined, but they are not equal (and `limits` is larger than `requests` as expected). 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*
: Pod that has an equal amount of `request` and `limit` resources. These are the highest-priority Pods and guaranteed not to be killed before Best-Effort and Burstable Pods.

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.

## 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. Another related feature that is still in beta at the time of this writing is Pod Priority and Preemption. Pod priority allows indicating 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
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       ❸
```

❶ The name of the priority class object

❷ The priority value of the object

❸ 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 each other, where the higher value indicates more important Pods.

When the Pod Priority feature is enabled, it affects the order in which the scheduler places Pods on nodes. First, the priority admission controller uses the `priorityClass Name` 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.

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 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 larger 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 resources of the platform. 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.

Another useful tool in this area is LimitRange, which allows setting 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*. Table 2-1 gives an example how the possible values for `requests` and `limits` can be chosen.

*Table 2-1. Limit and request ranges*

| Type | Resource | Min | Max | Default limit | Default request | Lim/req ratio |
|---|---|---|---|---|---|---|
| Container | CPU | 500m | 2 | 500m | 250m | 4 |
| Container | Memory | 250Mi | 2Gi | 500Mi | 250Mi | 4 |

LimitRanges are useful for controlling the container resource profiles so that there are no containers that require more resources than a cluster node can provide. It can also prevent cluster users from creating containers that consume a large number of 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 how much difference there is between the `requests` and `limits` of containers. Having a big combined gap between `requests` and `limits` increases the chances for overcommitting on the node and may degrade application performance when many containers simultaneously require more resources than originally requested.

## 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 *Burstable*. If a container gets killed, that is most likely a sign that the capacity of the cluster should be increased.

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

*Table 2-2. Capacity planning example*

| Pod | CPU request | CPU limit | Memory request | Memory limit | Instances |
|---|---|---|---|---|---|
| A | 500m | 500m | 500Mi | 500Mi | 4 |
| B | 250m | 500m | 250Mi | 1000Mi | 2 |
| C | 500m | 1000m | 1000Mi | 2000Mi | 2 |
| D | 500m | 500m | 500Mi | 500Mi | 1 |
| **Total** | **4000m** | **5500m** | **5000Mi** | **8500Mi** | **9** |

Of course, in a real-life scenario, the more likely reason you are using a platform such as Kubernetes would be because there are many more services to manage, some of which are about to retire, and some are still at 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 also a different number 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.

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 get our applications installed and updated on Kubernetes.

## More Information

- Predictable Demands Example
- Using ConfigMap
- Resource Quotas
- Kubernetes Best Practices: Resource Requests and Limits
- Setting Pod CPU and Memory Limits
- Configure Out of Resource Handling
- Pod Priority and Preemption
- Resource Quality of Service in Kubernetes

CHAPTER 3

# Declarative Deployment

The heart of the *Declarative Deployment* pattern is 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 have the services placed 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 no 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 this activity 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, we have seen 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.

### Imperative Rolling Updates with kubectl Are Deprecated

Kubernetes has supported rolling updates since its very beginning. The first implementation was *imperative* in nature; the client `kubectl` tells the server what to do for each update step.

Although the `kubectl rolling-update` command is still present, it's highly deprecated because of the following drawbacks of such an imperative approach:

- Rather than describing the desired end state, `kubectl rolling-update` issues commands to get the system into the desired state.
- The whole orchestration logic for replacing the containers and the ReplicationControllers is performed by `kubectl`, which monitors and interacts with the API Server behind the scenes while the update process happens, moving an inherent server-side responsibility to the client.
- You may need more than one command to get the system into the desired state. These commands must be automated and repeatable in different environments.
- Somebody else may override your changes with time.
- The update process has to be documented and kept up-to-date while the service evolves.
- The only way to find out what we have deployed is by checking the condition of the system. Sometimes the state of the current system might not be the desired state, in which case we have to correlate it with the deployment documentation.

Instead, the Deployment resource object was introduced for supporting a *declarative update*, fully managed by the Kubernetes backend. As declarative updates have so many advantages, and imperative update support will vanish eventually, we focus exclusively on declarative updates in this pattern.

## 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 shaping 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
    selector:
      matchLabels:
        app: random-generator
    template:
      metadata:
        labels:
          app: random-generator
      spec:
        containers:
        - image: k8spatterns/random-generator:1.0
          name: random-generator
          readinessProbe:                      ❹
            exec:
              command: [ "stat", "/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 total of four replicas at maximum.

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

❹ Readiness probes are very important for a rolling deployment to provide 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. Figure 3-1 shows the rolling update process.

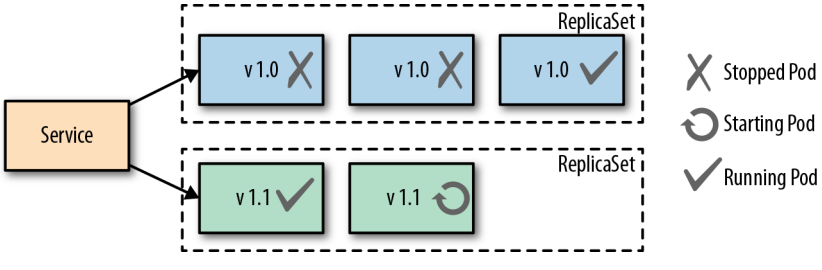


*Figure 3-1. Rolling deployment*

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 example repository, which demonstrates the usage of these commands, and shows you how you can monitor or roll back an upgrade with `kubectl rollout`.

In addition to addressing the previously mentioned drawbacks of the imperative way of deploying services, the Deployment brings 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 makes you see how the deployed state should look rather than the steps necessary to get there.
- The Deployment definition is an executable object, 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, there is the `Recreate` strategy, which is illustrated in Figure 3-2.

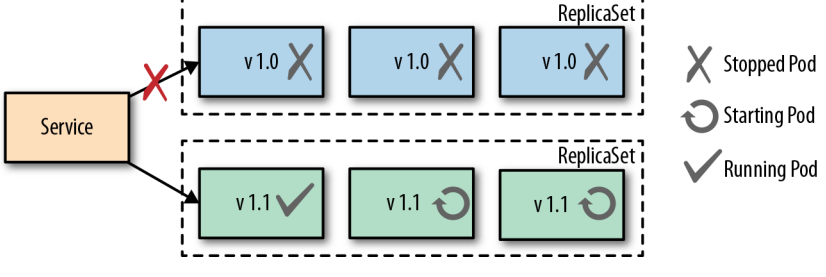


*Figure 3-2. Fixed deployment using a Recreate strategy*

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 of actions is that there is some downtime while all containers with old versions are stopped, and there are no new containers ready to handle incoming requests. On the positive side, there won't be two versions of the containers running at the same time, simplifying the life of service consumers to handle 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. 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 of a Blue-Green deployment.

A Blue-Green deployment needs to be done manually if no extensions like a Service Mesh or Knative is 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 (called *blue*) from the original Deployment 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. This activity in Kubernetes can be done by updating the Service selector to match the new containers (tagged as green). As demonstrated in Figure 3-3, once the green containers handle all the traffic, the blue containers can be deleted and the resources freed for future Blue-Green deployments.

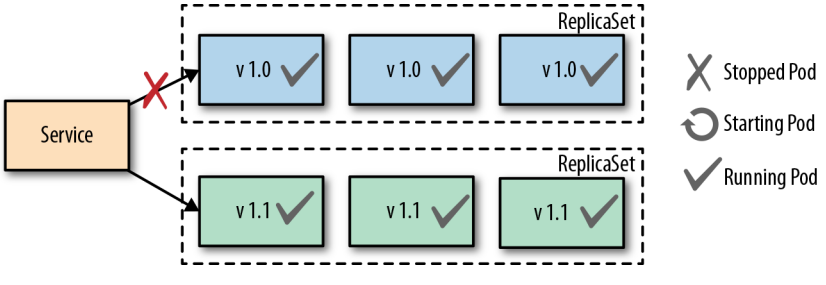


*Figure 3-3. Blue-Green release*

A benefit of the Blue-Green approach is that there's only one version of the application serving requests, 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, there can be significant complications 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 are 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.

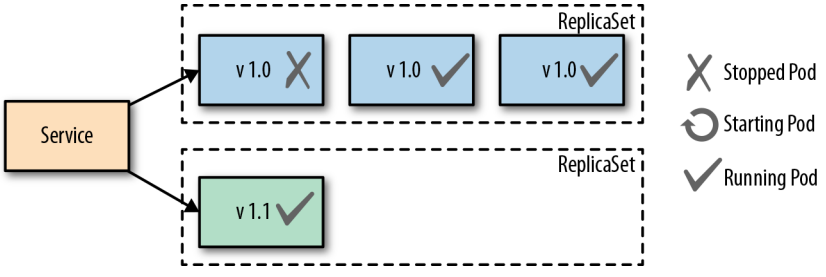


*Figure 3-4. Canary release*

In Kubernetes, this technique can be implemented by creating a new ReplicaSet for the new container version (preferably using a 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 new ReplicaSet works as expected, we scale the new ReplicaSet up, and the old ReplicaSet down to zero. In a way, we are performing a controlled and user-tested incremental rollout.

## Discussion

The Deployment primitive is an example of where Kubernetes turns 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 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 but require human interaction. Figure 3-5 shows a summary of the deployment and release strategies, showing instance counts during transitions.

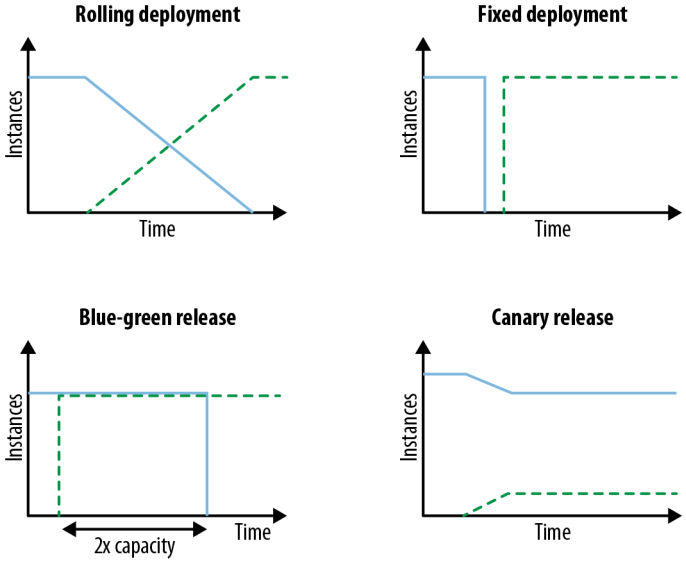


*Figure 3-5. Deployment and release strategies*

Every 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.

As of this writing, there is 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 commands are a good step toward new automated deployment and release strategies. For now, an approach that works is to script the update process at a higher level to manage the update process of services and its dependencies using the Deployment and other primitives discussed in this book.

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 reaching 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
- Rolling Update
- Deployments
- Run a Stateless Application Using a Deployment
- Blue-Green Deployment
- Canary Release
- DevOps with OpenShift

CHAPTER 4

# Health Probe

The *Health Probe* pattern is about 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 decide about 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 a check that indicates 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 all have different defini-

tions 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 see 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. 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 some 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 the in 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 what 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.
- A TCP Socket probe assumes a successful TCP connection.
- An Exec probe executes an arbitrary command in the container kernel namespace and expects a successful exit code (0).

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 implementation to decide when your application is considered healthy or not. However, keep in mind that the result of not passing a health check is restarting of your container. 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 are useful for keeping applications healthy by killing unhealthy containers and replacing them with new ones. But sometimes a container may not be healthy, and restarting it may not help either. The most common example is when a container is still starting up and not ready to handle any requests yet. Or maybe a container is overloaded, and its latency is increasing, and you want it to shield itself from additional load for a while.

For this kind of scenario, Kubernetes has *readiness probes*. The methods for performing readiness checks are the same as liveness checks (HTTP, TCP, Exec), 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.

In many cases, you have liveness and readiness probes 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.

The liveness and readiness probes are fundamental building blocks in the automation of cloud-native applications. Application frameworks such as Spring actuator, WildFly Swarm health check, Karaf health checks, 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 detecting 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. Figure 4-1 shows the possible options for how a container can communicate with the runtime platform.

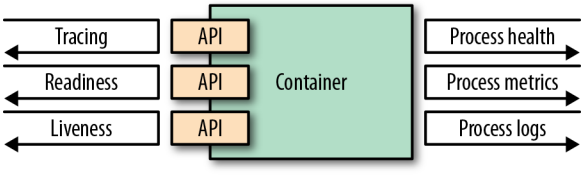


*Figure 4-1. Container observability options*

Containers provide a unified way for packaging and running applications by treating them like black boxes. 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 tracing and metrics-gathering libraries such as OpenTracing or Prometheus. Treat your application as a black box, 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
- Configuring Liveness and Readiness Probes
- Setting Up Health Checks Wth Readiness and Liveness Probes
- Resource Quality of Service
- Graceful Shutdown with Node.js and Kubernetes
- 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 on these. 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 interac-

tions 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.

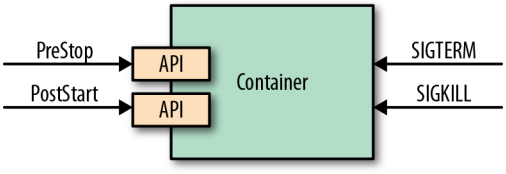


*Figure 5-1. Managed container lifecycle*

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 14, *Init Container* (and defer-containers, which is still at the proposal stage as of this writing) 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 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 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 for a grace period of 30 seconds by default after it has issued a SIGTERM signal. This grace period can be defined per Pod using the `.spec.terminationGracePeriodSeconds` field, but cannot be guaranteed as it can be overridden while issuing commands to Kubernetes. The aim here 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 there are additional lifecycle hooks such as `postStart` and `preStop` 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 here 30 seconds. `sleep` is just a simulation for any lengthy startup code that might run here. Also, it uses a trigger file here 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 many 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 giving time to 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, the main container process gets killed by Kubernetes.

`postStart` and `preStop` hook invocation mechanisms are similar to the *Health Probes* described in Chapter 4 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:          (1)
          port: 8080
          path: /shutdown
```

(1) Call out to a `/shutdown` endpoint running within the application

Even though `preStop` is blocking, holding on it or returning a nonsuccessful result does not prevent the container from being deleted and the process killed. `preStop` 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 executing commands when a container lifecycle event occurs. But another mechanism that is not at the container level but at a Pod level allows executing initialization instructions.

We describe in Chapter 14, *Init Container*, 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 using init containers for Pod-level initialization tasks. Both lifecycle hooks and init containers operate at a different granularity (at container level and Pod-level, respectively) and could be used interchangeably in some instances, or complement each other 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 exists yet |
| Timing guarantees | A `postStart` command is executed at the same time as the container's `ENTRYPOINT` | 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 |

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. In this sequence, the options require more effort increasingly, 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 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. 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 get into the mindset where the application lifecycle is no longer in the control of a person but 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
- Attaching Handlers to Container Lifecycle Events
- Terminating with Grace
- Graceful Shutdown of Pods with Kubernetes
- Defer Containers

CHAPTER 6

# Automated Placement

*Automated Placement* is the core function of the Kubernetes scheduler for assigning new Pods to nodes satisfying container resource requests and honoring scheduling policies. This pattern describes the principles of Kubernetes' scheduling algorithm and the way 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 that has to be shot on the move.

## Solution

In Kubernetes, assigning Pods to nodes is done by the scheduler. It is an area that is highly configurable, still evolving, and changing rapidly as of this writing. 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 funda-

mental role in the Kubernetes platform as a whole, but similarly 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 a 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 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 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 [OS system daemons like sshd, udev]
```

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. 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 for this limitation 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 that containers have their runtime dependencies and resource demands defined. 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 sensibly assigned to nodes and can run without affecting each other during peak times.

## Placement Policies

The last piece of the puzzle is having the right filtering or priority policies for your specific application needs. The scheduler has a default set of predicate and priority policies configured that is good enough for most use cases. It can be overridden during scheduler startup with a different set of policies, as shown in Example 6-2.

Scheduler policies and custom schedulers can be defined only by an administrator as part of the cluster configuration. As a regular user you just can refer to predefined schedulers.

*Example 6-2. An example scheduler policy*

```
{
    "kind" : "Policy",
    "apiVersion" : "v1",
    "predicates" : [                          ❶
        {"name" : "PodFitsHostPorts"},
        {"name" : "PodFitsResources"},
        {"name" : "NoDiskConflict"},
        {"name" : "NoVolumeZoneConflict"},
        {"name" : "MatchNodeSelector"},
        {"name" : "HostName"}
    ],
    "priorities" : [                          ❷
        {"name" : "LeastRequestedPriority", "weight" : 2},
        {"name" : "BalancedResourceAllocation", "weight" : 1},
        {"name" : "ServiceSpreadingPriority", "weight" : 2},
        {"name" : "EqualPriority", "weight" : 1}
    ]
}
```

❶ Predicates are rules that filter out unqualified nodes. For example, *PodFitsHostsPorts* schedules Pods to request certain fixed host ports only on those nodes that have this port still available.

❷ Priorities are rules that sort available nodes according to preferences. For example, *LeastRequestedPriority* gives nodes with fewer requested resources a higher priority.

Consider that in addition to configuring the policies of the default scheduler, it is also possible to run multiple schedulers and allow Pods to specify which scheduler to place them. You can start another scheduler instance that is configured differently by giving it a unique name. Then when defining a Pod, just add the field `.spec.schedulerName` with the name of your custom scheduler to the Pod specification and the Pod will be picked up by the custom scheduler only.

## 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.

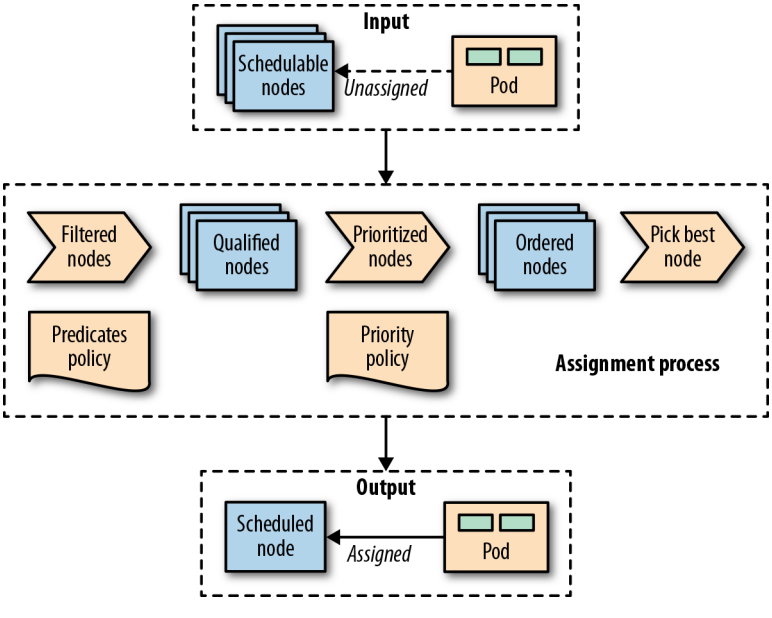


*Figure 6-1. A Pod-to-node assignment process*

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 based on the Pod's criteria. In the second stage, the remain-

ing nodes get ordered by weight. In the last stage the Pod gets a node assigned, which is the primary outcome of the scheduling process.

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 a group of nodes. This assignment can be done using a node selector. `.spec.nodeSelector` is Pod field and 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, 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 to be 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 a feature is node affinity, which is a generalization 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. Besides, 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:                                   ❹
          - key: metadata.name
            operator: NotIn
            values: [ "master" ]
  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.

❹ Match on a field (specified as jsonpath). Note that only `In` and `NotIn` are allowed as operators, and only one value is allowed to be given in the list of values.

## Pod Affinity and Antiaffinity

Node affinity is a more powerful way of scheduling and should be preferred when `nodeSelector` is not enough. This mechanism allows constraining which nodes a Pod can run based on label or field matching. It doesn't allow expressing 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 co-located together to improve latency, Pod affinity and antiaffinity can be used.

Node affinity works at node granularity, but Pod affinity is not limited to nodes and can express rules at multiple topology levels. 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 label `security-zone`. The Pod defined here is scheduled to a node with the same label and value.

❹ Antiaffinity 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 antiaffinity, called `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution`, respectively. Again, as with node affinity, there is the `IgnoredDuringExecution` suffix 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.

## Taints and Tolerations

A more advanced feature that controls where Pods can be scheduled and are 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 as an *opt-in* to allow scheduling on nodes, which 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 master node-role.kubernetes.io/master="true":NoSchedule`, which has the effect shown in Example 6-6. A matching toleration is added to a Pod as shown in Example 6-7. Notice that the values for `key` and `effect` in the `taints` section of Example 6-6 and the `tolerations:` section in Example 6-7 have the same values.

*Example 6-6. Tainted node*

```
apiVersion: v1
kind: Node
metadata:
  name: master
spec:
  taints:                                          ❶
  - effect: NoSchedule
    key: node-role.kubernetes.io/master
```

❶ Taint on a node's spec to mark this node as not available for scheduling except when a Pod tolerates this taint

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 6-7. 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/master          ❶
    operator: Exists
    effect: NoSchedule                           ❷
```

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

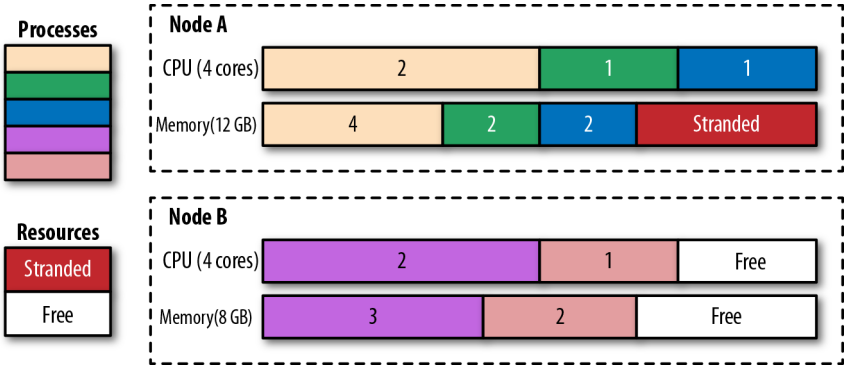


*Figure 6-2. Processes scheduled to nodes and stranded resources*

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 either.

All these are scenarios that can be addressed by the descheduler. The Kubernetes descheduler is an optional feature that typically is 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. The policies are passed as a file to the descheduler Pod, and currently, they are the following:

*RemoveDuplicates*
: This strategy ensures that only a single Pod associated with a ReplicaSet or Deployment is running on a single node. If there are more Pods than one, these excess Pods are evicted. This strategy is useful in scenarios where a node has become unhealthy, and the managing controllers started new Pods on other healthy nodes. When the unhealthy node is recovered and joins the cluster, the number of running Pods is more than desired, and the descheduler can help bring the numbers back to the desired `replicas` count. Removing duplicates on nodes can also help with the spread of Pods evenly on more nodes when scheduling policies and cluster topology have changed after the initial placement.

*LowNodeUtilization*
: This strategy finds nodes that are underutilized and evicts Pods from other overutilized nodes, hoping these Pods will be placed on the underutilized nodes, leading to better spread and use of resources. The underutilized nodes are identified as nodes with CPU, memory, or Pod count below the configured `thresholds` values. Similarly, overutilized nodes are those with values greater than the configured `targetThresholds` values. Any node between these values is appropriately utilized and not affected by this strategy.

*RemovePodsViolatingInterPodAntiAffinity*
: This strategy evicts Pods violating interpod antiaffinity rules, which could happen when the antiaffinity rules are added after the Pods have been placed on the nodes.

*RemovePodsViolatingNodeAffinity*
: This strategy is for evicting Pods violating node affinity rules.

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

- Critical Pods that are marked with `scheduler.alpha.kubernetes.io/critical-pod` annotation.
- Pods not managed by a ReplicaSet, Deployment, or Job.
- Pods managed by a DaemonSet.
- Pods that have local storage.
- Pods with PodDisruptionBudget where eviction would violate its rules.
- 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 an area where you want to have as minimal intervention as possible. 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 suitable node possible. However, when that is not enough, there are multiple ways to steer the scheduler toward the desired deployment topology. To sum up, from simpler to more complex, the following approaches control Pod scheduling (keep in mind, as of this writing, this list changes with every other release of Kubernetes):

*nodeName*
: The simplest form of hardwiring 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 limits greatly where a Pod can be scheduled. This throws us back in to the pre-Kubernetes era when we explicitly specified the nodes to run our applications.

*nodeSelector*
: Specification of a map of key-value pairs. 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 acceptable mechanisms for controlling the scheduler choices.

*Default scheduling alteration*
: The default scheduler is responsible for the placement of new Pods onto nodes within the cluster, and it does it reasonably. However, it is possible to alter the filtering and priority policies list, order, and weight of this scheduler if necessary.

*Pod affinity and antiaffinity*
: These rules allow a Pod to express dependencies on other Pods. For example, for an application's latency requirements, high availability, security constraints, and so forth.

*Node affinity*
: This rule allows a Pod to express dependency toward nodes. For example, considering nodes' hardware, location, and so forth.

*Taints and tolerations*
: Taints and tolerations allow the node to control which Pods should or should not be scheduled on them. For example, to dedicate a node for a group of Pods, or even evict Pods at runtime. 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 the Pods, but only on the Pods that should be placed on the new nodes.

*Custom scheduler*
: If none of the preceding approaches is good enough, or maybe you have complex scheduling requirements, you can also write your 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 hard-

ware 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.

As you can see, there are lots of ways to control the Pod placement and choosing the right approach or combining multiple approaches can be challenging. The takeaway from this chapter is this: size and declare container resource profiles, label Pods and nodes accordingly, and finally, do only a minimal intervention to the Kubernetes scheduler.

## More Information

- Automated Placement Example
- Assigning Pods to Nodes
- Node Placement and Scheduling Explained
- Pod Disruption Budget
- Guaranteed Scheduling for Critical Add-On Pods
- The Kubernetes Scheduler
- Scheduler Algorithm
- Configuring Multiple Schedulers
- Descheduler for Kubernetes
- Keep Your Kubernetes Cluster Balanced: The Secret to High Availability
- Everything You Ever Wanted to Know About Resource Scheduling, but Were Afraid to Ask

# PART II
# Behavioral Patterns

The patterns under this category are focused around the communication mechanisms and interactions between the Pods and the managing platform. Depending on the type of managing controller, a Pod may run until completion, or be scheduled to run periodically. It can run as a *Daemon Service*, or provide uniqueness guarantees to its replicas. There are different ways to run a Pod and picking the right Pod management primitives requires understanding their behavior. In the following chapters, we explore the patterns:

- Chapter 7, *Batch Job*, describes an isolated atomic unit of work run 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 running infrastructure-focused Pods on specific nodes, before application Pods are placed.
- Chapter 10, *Singleton Service*, ensures only one instance of a service is active at a time and still highly available.
- Chapter 11, *Stateful Service*, is all about how to create and manage distributed stateful applications with Kubernetes.
- Chapter 12, *Service Discovery*, explains how clients can access and discover the instances providing application services.
- Chapter 13, *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 Job abstraction, 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 under the names of *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.

*DaemonSet*
: A controller for running a single Pod on every node. Typically used for managing platform capabilities such as monitoring, log aggregation, storage containers, and others. See Chapter 9, *Daemon Service* for a detailed discussion on DaemonSets.

A common aspect of these Pods is the fact that they represent long-running processes that are not meant to stop after some 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
  template:
    metadata:
      name: random-generator
    spec:
      restartPolicy: OnFailure      ❸
      containers:
      - image: k8spatterns/random-generator:1.0
        name: random-generator
        command: [ "java", "-cp", "/", "RandomRunner", "/numbers.txt", "10000" ]
```

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

❷ Two Pods can run in parallel.

❸ Specifying the `restartPolicy` is mandatory for a Job.

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 either `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 kept for tracking purposes. The Pods that are created as part of the Job are also not deleted but available for examination (e.g., to check the container logs). This is also true for bare Pods, but only for a `restartPolicy: OnFailure`.
- 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 (set through `.spec.completions`), 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.
- 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 where some guarantees are required for the completion of a unit of work.

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

`.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 less (and in some corner cases, more) than the desired number (e.g., due to 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 *Batch Job* defined in Example 7-1 with a completion count of five and a parallelism of two is processed.

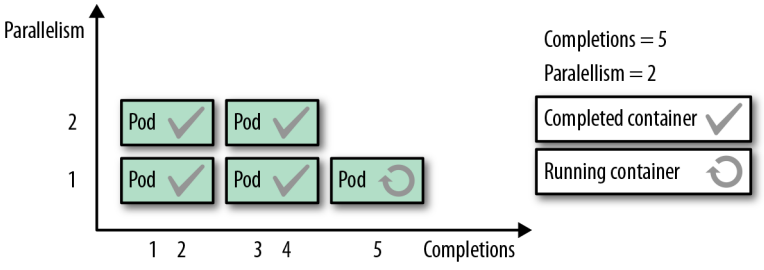


*Figure 7-1. Parallel Batch Job with a fixed completion count*

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

*Single Pod Job*
: This type is selected when you leave out both `.spec.completions` and `.spec.parallelism` or set them to their default values of one. 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*
: When you specify `.spec.completions` with a number greater than one, this many Pods must succeed. Optionally, you can set `.spec.parallelism`, or leave it at the default value of one. 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*
: You have a work queue for parallel Jobs when you leave out `.spec.completions` and set `.spec.parallelism` to an integer 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.

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 for the platform 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 of 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 on 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 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
- Run to Completion Finite Workloads
- Parallel Processing Using Expansions
- Coarse Parallel Processing Using a Work Queue
- Fine Parallel Processing Using a Work Queue
- Indexed Job Created with Metacontroller
- Java Batch Processing Frameworks and Libraries

# 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 the use of 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 similarly 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/v1beta1
kind: CronJob
metadata:
  name: random-generator
spec:
  # Every three minutes
  schedule: "*/3 * * * *"          ❶
  jobTemplate:
    spec:
      template:                    ❷
        spec:
          containers:
          - image: k8spatterns/random-generator:1.0
            name: random-generator
            command: [ "java", "-cp", "/", "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).

`.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 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 obsolete already.

`.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.

`.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 each other 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
- Cron Jobs
- Cron

CHAPTER 9

# Daemon Service

The *Daemon Service* pattern allows placing and running 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 they are called, the common characteristics among these programs 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 exists at the application level too. For example, in the JVM 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, there is also the concept of a DaemonSet in Kubernetes. 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:
        - sh
        - -c
        - >-
          "while true; do
          java -cp / RandomRunner /host_dev/random 100000;
          sleep 30; done"
        volumeMounts:                        ❷
        - mountPath: /host_dev
          name: devices
      volumes:
      - name: devices
```

```
      hostPath:
        path: /dev
```
❸

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 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 to every node. That can be controlled and limited to a subset of nodes by using the `nodeSelector` field.
- 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 using 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 managed by a DaemonSet are supposed to run only on targeted nodes, and as a result, are treated with higher priority and differently by many controllers. For example, the descheduler will avoid evicting such Pods, the cluster autoscaler will manage them separately, etc.

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

*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.

`NodeIP` *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 `hostIp` 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 also 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 there are no controller and no health checks 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. But DaemonSets are better integrated with the rest of the platform and recommended over static Pods.

## Discussion

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 applicability 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
- DaemonSets
- Performing a Rolling Update on a DaemonSet
- DaemonSets and Jobs
- Static Pods

CHAPTER 10

# Singleton Service

The *Singleton Service* pattern ensures only one instance of 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 24, *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 dispatching.

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 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 order 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 (usually only one is required) of a service are active at a time, 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* (or *master-slave*) 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 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 out-of-application locking can be realized with the help of a StatefulSet or ReplicaSet controller with one replica.

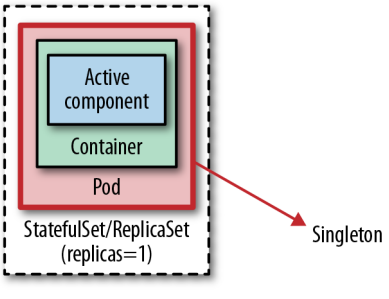


*Figure 10-1. Out-of-application locking mechanism*

The way to achieve this in Kubernetes is to start a Pod with one replica. 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 increased accidentally, as there is no platform-level mechanism to prevent a change 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, 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 11, *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 12, *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. But 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, a ReplicaSet with one replica and a regular Service would suffice. For a strict singleton and better performant service discovery, a StatefulSet and a headless Service would be preferred. You can find a complete example of this in Chapter 11, *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.

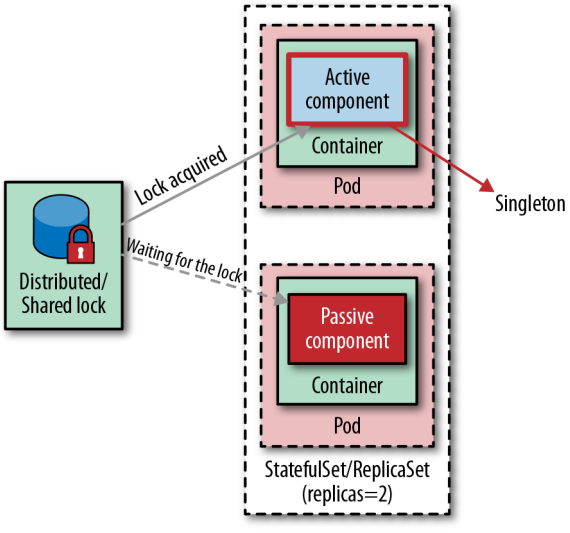


*Figure 10-2. In-application locking mechanism*

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 gets 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 a 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 master nodes. Etcd is a distributed key-value store that uses the Raft protocol to maintain its replicated state. Most importantly, it provides the necessary building blocks for implementing leader election, and a few client libraries have implemented this functionality already. For example, Apache

Camel 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.

An implementation with 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 are waiting to acquire the lock in case the master 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. *Voluntary* 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/v1beta1
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. But you cannot specify both fields, and PodDisruptionBudget typically applies only to Pods managed by a controller. For Pods not managed by a controller (also referred to as *bare* or *naked* Pods), other limitations around PodDisruptionBudget should be considered.

This functionality 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. It is another Kubernetes primitive that controls and influences the instance management at runtime, and is 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 (e.g., when a node that runs the singleton Pod is partitioned from the rest of the cluster, for example when replacing a deleted Pod instance with a new one) that have two copies of a Pod running concurrently for a short period. 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 would also prevent accidental scaling of Pods by changing the number of replicas.

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. Also, as a consequence, we either have to split the singleton component in its deployment unit to keep it a singleton (good in theory, but not always practical and 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
- Simple Leader Election with Kubernetes and Docker
- Leader Election in Go Client
- Configuring a Pod Disruption Budget
- Creating Clustered Singleton Services on Kubernetes
- Apache Camel Kubernetes Connector

CHAPTER 11

# 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

So far 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 among all these primitives is the fact 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.

While it is a significant boost to have a platform taking care of the placement, resiliency, and scaling of stateless applications, 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 some 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 and PersistentVolume 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-once semantics, and the number of replicas can vary temporarily. Such a situation can be disastrous and lead to data loss. Also, the main challenges come up 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 PersistentVolumeClaim (PVC) definition would result in all three Pods attached to the same PersistentVolume (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 here would be for the application instances to use shared storage and have an in-app mechanism for splitting the storage into subfolders and using it without conflicts. While doable, this approach creates a single point of a 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 would be 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 masters.

## 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 be always available; some are sensitive to ordinality, and some are fine with parallel Deployments; 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 creating CustomResourceDefinitions and *Operators* for managing stateful applications. Operators are explained in Chapter 23.

We have seen some of the 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 11-1 is our random-generator service as a StatefulSet. [1]

*Example 11-1. Service for accessing StatefulSet*

```
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
```

1 Let's assume we have invented a highly sophisticated way of generating random numbers in a distributed 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.

```
        name: http
        volumeMounts:
        - name: logs
          mountPath: /logs
  volumeClaimTemplates:                    ❹
  - metadata:
      name: logs
    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 11-2

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 11-1 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. Moreover, scaling down deletes the Pods, but it does not delete any PVCs (nor 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 11-2 we define a *headless* Service. In a headless Service, `clusterIP: None`, which means we don't want a kube-proxy to handle the Service, and we don't want a cluster IP allocation nor load balancing. Then why do we need a Service?

*Example 11-2. 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 each other, 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 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 additionally that load balance across your stateful Pods if that is what you want.

As shown in Figure 11-1, StatefulSets offer a set of building blocks and guaranteed behavior needed for managing stateful applications in a distributed environment. It is up to you to choose and use them in a way that is meaningful for your stateful use case.

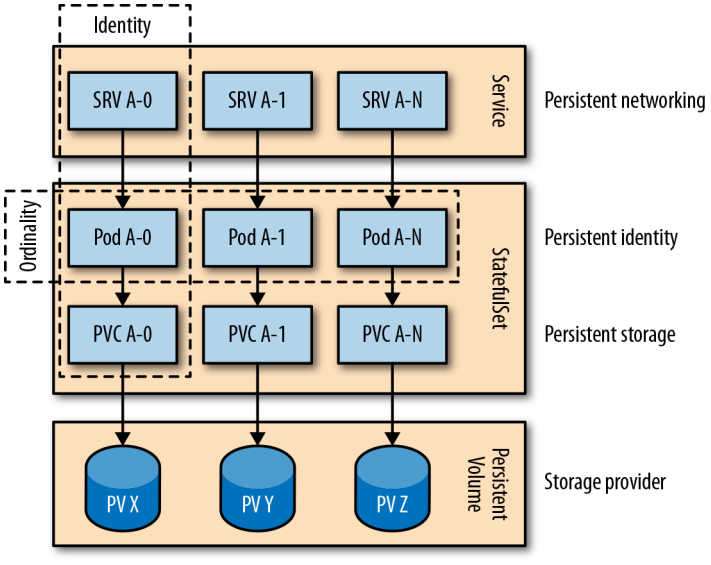


*Figure 11-1. A distributed stateful application on Kubernetes*

## Identity

*Identity* is the meta building block all other StatefulSet guarantees are built upon. Based on StatefulSet's name, we can get predictable a Pod name and identity. 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 each other 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 scaled 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 desired by 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 above the sequential ordering guarantees while 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 become running and 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 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 occasionally there is 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 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, 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 Chapter 22 and Chapter 23, 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
- Deploying Cassandra with Stateful Sets
- Running ZooKeeper, a Distributed System Coordinator
- Headless Services
- Force Delete StatefulSet Pods
- Graceful Scaledown of Stateful Apps in Kubernetes
- Configuring and Deploying Stateful Applications

CHAPTER 12

# Service Discovery

The *Service Discovery* pattern provides a stable endpoint at which clients 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 or through external stimulus. Internally initiated interactions are usually performed through a polling consumer: an application either after startup or later connects to another system and start sending and receiving data. Typical examples are an application running within a Pod that reaches a file server and starts consuming files, or connects to a message broker and starts receiving or sending messages, or connects to 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 perform tracking, registering and discovering 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 such as Prana looked up the service in a registry, as shown in Figure 12-1.

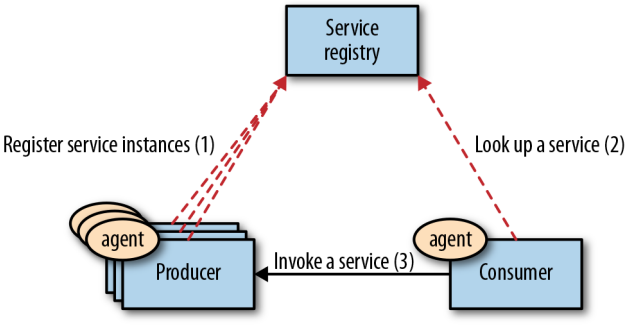


*Figure 12-1. Client-side service discovery*

In the "Post Kubernetes Era," many of the nonfunctional responsibilities of distributed systems such as placement, health check, 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 12-2 shows how registration and lookup are embraced by Kubernetes.

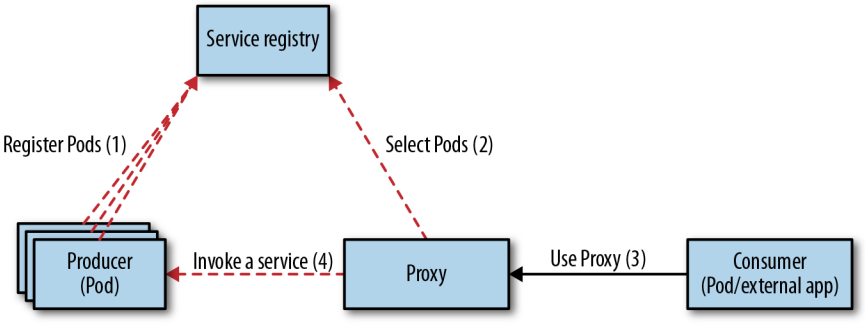


*Figure 12-2. Server-side service discovery*

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 12-1.

*Example 12-1. 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 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 12-3.

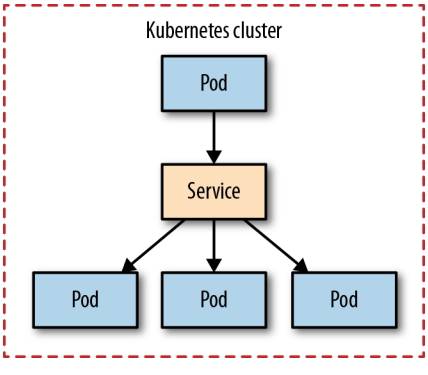


*Figure 12-3. Internal service discovery*

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 12-2 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 needs to be restarted.

*Example 12-2. 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 non-standard 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 picks a Pod randomly 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 can-

not 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 ICMP protocol. However, it is of course possible to access the Service IP address via TCP (e.g., for an HTTP request).

*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 12-1, 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 12-3.

*Example 12-3. Service without selector*

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

Next, in Example 12-4, we define an endpoints resource with the same name as the Service, and containing the target IPs and ports.

*Example 12-4. 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 here is that the list of endpoints is manually maintained, and the values there usually point to IP addresses outside the cluster, as demonstrated in Figure 12-4.

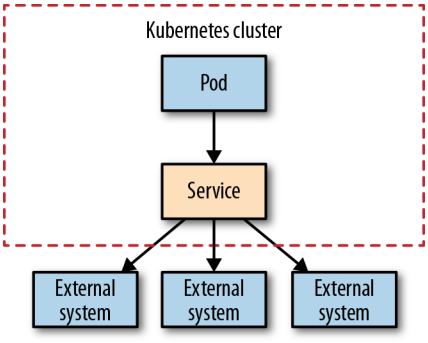


*Figure 12-4. Manual service discovery*

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 adding and removing selectors and pointing 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.

In this category of manual destination configuration, there is one more type of Service, as shown in Example 12-5.

*Example 12-5. 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. 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 of 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 12-6 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 12-6. 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 as illustrated in Figure 12-5 may seem like a good approach, it has drawbacks. 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 connecting 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 and 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.

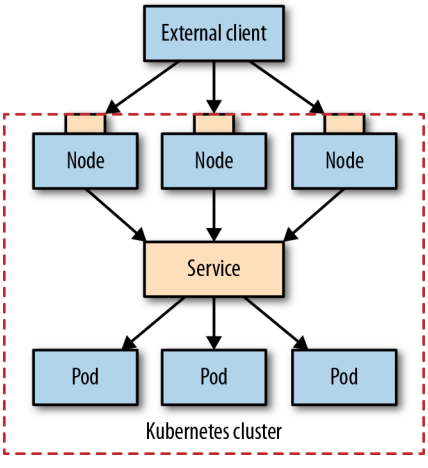


*Figure 12-5. Node port Service Discovery*

Another way of 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 12-6 shows this setup: a proprietary load balancer serves as a gateway to the Kubernetes cluster.

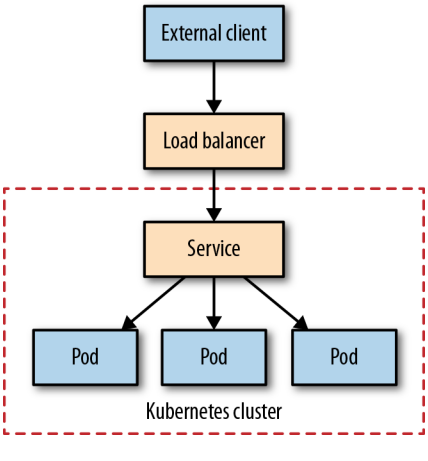


*Figure 12-6. Load balancer service discovery*

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 12-7.

*Example 12-7. 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
    protocol: TCP
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 is performed and service discovery varies among cloud providers. Some cloud providers will allow defining 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 11, *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, SSL 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 12-8.

*Example 12-8. An Ingress definition*

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: random-generator
spec:
  backend:
    serviceName: random-generator
    servicePort: 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 12-9.

*Example 12-9. An Ingress definition with mappings*

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: random-generator
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:                                  ❶
  - http:
      paths:
      - path: /                           ❷
        backend:
          serviceName: random-generator
          servicePort: 8080
```

```
  - path: /cluster-status          ❸
    backend:
      serviceName: cluster-status
      servicePort: 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 12-7.

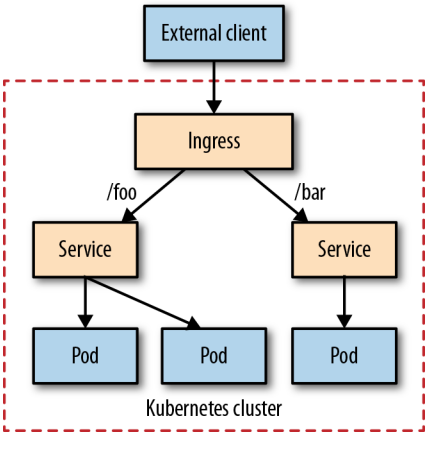


*Figure 12-7. Application layer service discovery*

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.

### 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 are 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. However, you can replace the build in OpenShift load balancer, too.
- 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 12-1 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 12-1. Service Discovery mechanisms*

| Name | Configuration | Client type | Summary |
|---|---|---|---|
| ClusterIP | `type: ClusterIP`<br>`.spec.selector` | Internal | The most common internal discovery mechanism |
| Manual IP | `type: ClusterIP`<br>`kind: Endpoints` | Internal | External IP discovery |
| Manual FQDN | `type: ExternalName`<br>`.spec.externalName` | Internal | External FQDN discovery |
| Headless Service | `type: ClusterIP`<br>`.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 in advanced serving, building, and messaging.

In the context of *Service Discovery*, 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 Build" on page 230 and "Knative Serving" on page 210, but a full discussion of Knative is beyond the scope of this book. In "More Information" on page 219, you will find links that point to detailed information about Knative.

## More Information

- Service Discovery Example
- Kubernetes Services
- DNS for Services and Pods
- Debug Services
- Using Source IP
- Create an External Load Balancer
- Kubernetes NodePort versus LoadBalancer versus Ingress?
- Ingress
- Kubernetes Ingress versus OpenShift Route

CHAPTER 13

# 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 passing 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 interests us, and Kubernetes populates the values dynamically. Figure 13-1 gives an overview of how the Downward API injects resource and runtime information into interested Pods.

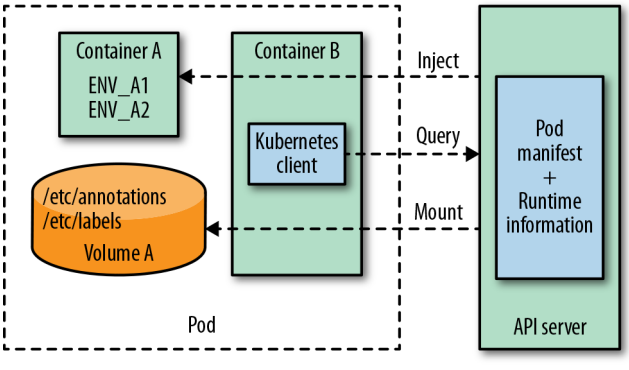


*Figure 13-1. Application introspection mechanisms*

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 13-1.

*Example 13-1. 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 come 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 13-1 are available for `fieldRef.fieldPath` both as environment variables and `downwardAPI` volumes.

*Table 13-1. 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* |

Similarly to `fieldRef`, we can use `resourceFieldRef` to access metadata specific to a container belonging to the Pod. This metadata is specific to a container which can be specified with `resourceFieldRef.container`. When used as environment variable then by default the current container is used. The possible keys for `resourceFieldRef.resource` are shown in Table 13-2.

*Table 13-2. Downward API information available in resourceFieldRef.resource*

| Name | Description |
|---|---|
| `requests.cpu` | A container's CPU request |
| `limits.cpu` | A container's CPU limit |
| `limits.memory` | A container's memory request |
| `requests.memory` | A container's memory 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 13-2 shows how such volumes can be used.

*Example 13-2. 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

On many occasions, 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 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, for example. 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
- Expose Pod Information to Containers Through Files
- Expose Pod Information to Containers Through Environment Variables
- Amazon ECS Container Agent Introspection
- Instance Metadata and User Data

# 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 such as Pods, which provide unique runtime capabilities.

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

- Chapter 14, *Init Container*, introduces a separate lifecycle for initialization-related tasks and the main application containers.
- Chapter 15, *Sidecar*, describes how to extend and enhance the functionality of a pre-existing container without changing it.
- Chapter 16, *Adapter*, takes an heterogeneous system and makes it conform to a consistent unified interface that can be consumed by the outside world.
- Chapter 17, *Ambassador*, describes a proxy that decouples access to external services.

CHAPTER 14

# Init Container

*Init Containers* enable 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 at Pod level rather than 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 setting up special permissions 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 14-1.

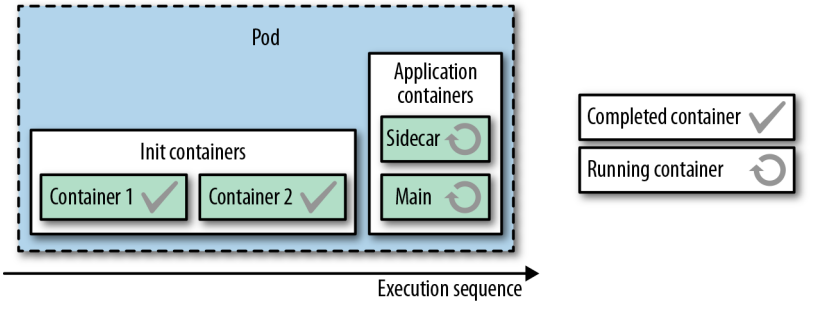


*Figure 14-1. Init and application containers in a Pod*

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 health-checking and resource-handling semantics. There is no readiness check 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. 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 keeping 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 containers and focus on configuration and initialization tasks only. We demonstrate this in Example 14-1, 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 14-1 shows an init container that copies data into an empty volume.

*Example 14-1. Init Container*

```
apiVersion: v1
kind: Pod
metadata:
  name: www
  labels:
    app: www
spec:
  initContainers:
  - name: download
    image: axeclbr/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: docker.io/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 wanted 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.

#### Keep a Pod running

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 15, 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 (as in Example 15-1, where the Git synchronizer container continuously updates the local folder). 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 worth listing here for completeness:

*Admission controllers*
: These are a set of plugins that intercept every request to the Kubernetes API Server before persistence of the object and can mutate or validate it. There are many controllers for applying checks, enforcing limits, and setting default values, but they are all 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.

*Initializers*
: Initializers are useful for admins to force policies or to inject defaults by specifying a list of pending preinitialization tasks, stored in every object's metadata. Then custom initializer controllers whose names correspond to the names of the tasks perform the tasks. Only after all initialization tasks are completed fully does the API object become visible to regular controllers.

*PodPresets*
: PodPresets are evaluated by another admission controller, which helps inject fields specified in a matching PodPreset into Pods at creation time. The fields can include volumes, volume mounts, or environment variables. Thus, PodPresets inject additional runtime requirements into a Pod at creation time using label selectors to specify the Pods to which a given PodPreset applies. PodPresets allow Pod template authors to automate adding repetitive information required for multiple Pods.

There are many techniques for initializing Kubernetes resources. However, these techniques differ from admission webhooks because they validate and mutate resources at creation time. You could use these techniques, for example, to inject an init

container into any Pod that doesn't have one already. In contrast, the *Init Container* pattern discussed in this chapter is something that activates and performs its responsibilities during startup of the Pod. In the end, the most significant difference is that init containers is for developers deploying on Kubernetes, whereas the techniques described here help administrators control and manage the container initialization process.

## Discussion

So why separate containers in a Pod into two groups? Why not just use an application container 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 the same guarantees as init containers. With this distinction in hand, we can create containers focused on a single initialization or application-focused task, and organize them in Pods.

## More Information

- Init Container Example
- Init Containers
- Configuring Pod Initialization
- The Initializer Pattern in JavaScript
- Object Initialization in Swift
- Using Admission Controllers
- Dynamic Admission Control
- How Kubernetes Initializers Work
- Pod Preset
- Inject Information into Pods Using a PodPreset
- Kubernetes Initializer Tutorial

# CHAPTER 15
# Sidecar

A *Sidecar* container extends and enhances the functionality of a preexisting container without changing it. This pattern is one of the fundamental container patterns that allows single-purpose containers to cooperate closely together. In this chapter, we learn all about the basic *Sidecar* concept. The specialized follow-up patterns, *Adapter* and *Ambassador*, are discussed in Chapter 16 and Chapter 17, respectively.

## Problem

Containers are a popular packaging technology that allows 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 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 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* (in some places 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 used to demonstrate this pattern is with an HTTP server and a Git synchronizer. The HTTP server container is focused only on serving files over HTTP and does not know how and 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 to the files once synced, and its only concern is keeping the local folder in sync with the remote Git server. Example 15-1 shows a Pod definition with these two containers configured to use a volume for file exchange.

*Example 15-1. Pod with Sidecar*

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: app
    image: docker.io/centos/httpd      ❶
    ports:
    - containerPort: 80
    volumeMounts:
    - mountPath: /var/www/html         ❸
      name: git
  - name: poll
    image: axeclbr/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"
  - "git clone $(GIT_REPO) . && watch -n 600 git pull"
  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 enhances 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 15-1, 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.

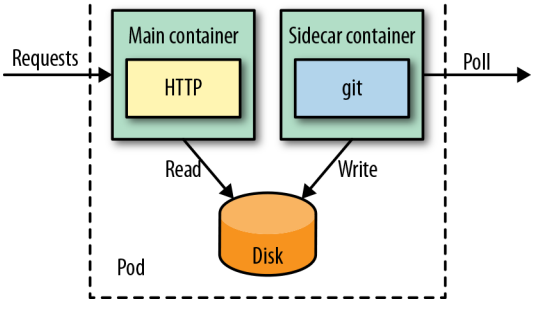


*Figure 15-1. Sidecar pattern*

## 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 of these 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 the ability to later swap containers in the Pod definition. On the other hand, compositions in Pods mean 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.

From a different point of view, container composition is similar to aspect-oriented programming, in that with additional containers we introduce orthogonal capabilities to the Pod without touching the main container. In recent months, use of the *Sidecar* pattern has become more and more common, especially for handling networking, monitoring, and tracing aspects of services, where every service ships *Sidecar* containers as well.

## More Information

- Sidecar Example
- 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
- The Almighty Pause Container

CHAPTER 16

# 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*, 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 this 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 16-1, an *Adapter* container translates locally stored metrics information into the external format the monitoring server understands.

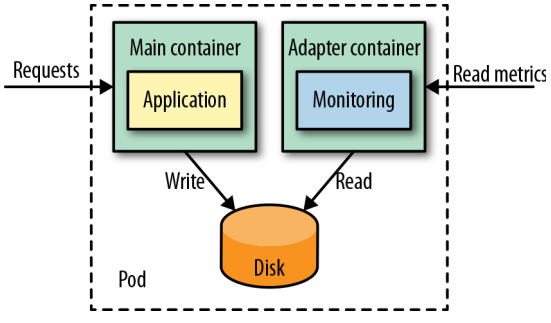


*Figure 16-1. Adapter pattern*

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 revisit our sample random generator application and create the adapter shown in Figure 16-1. 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 16-1 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 our GitHub repository demonstrates this setup together with a Prometheus installation.

*Example 16-1. 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 to

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 level 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 13, 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 15. 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 17

# Ambassador

The *Ambassador* pattern is a specialized *Sidecar* responsible for hiding complexity and providing a unified interface for accessing services outside the Pod. In this chapter, we see how the *Ambassador* pattern can act as a proxy and decouple the main Pod 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 would be consuming a service by

looking it up in a registry and performing client-side service discovery. A third example would be 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 17-1 and 17-2 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 17-1, we see how data access can be delegated to a fully distributed remote store like Etcd.

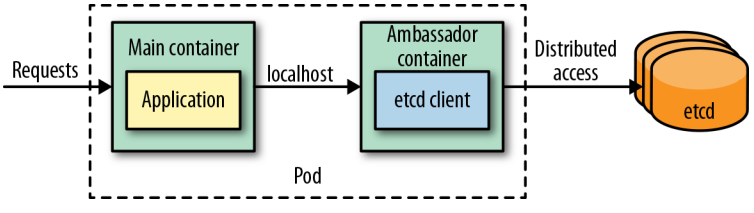


*Figure 17-1. Ambassador for accessing a remote distributed cache*

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 17-2).

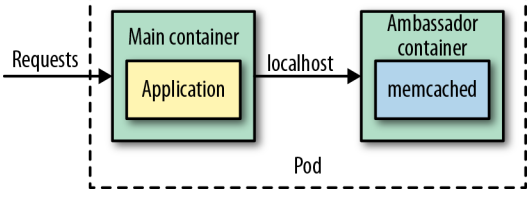


*Figure 17-2. Ambassador for accessing a local cache*

The benefits of this pattern are similar to those of the *Sidecar* pattern—both allow keeping 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 creating specialized and reusable *Ambassador* containers that can be combined with other application containers.

Example 17-1 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 17-1. 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* 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, where it gets its name from (this pattern is sometimes referred to as the *Proxy* pattern as well).

## More Information

- Ambassador Example
- How to Use the Ambassador Pattern to Dynamically Configure Services
- Dynamic Docker Links with an Ambassador Powered
- Link via an Ambassador Container
- Modifications to the CoreOS Ambassador Pattern

# PART IV
# Configuration Patterns

Every application needs to be configured, and the easiest way to do so is by storing configurations in the source code. This approach has the side effect of configuration and code living and dying together, as described by the Immutable Server concept. However, we still need the flexibility to adapt configuration without recreating the application image. In fact, this recreation would be time-consuming and 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.

In such a scenario, how would we adapt an application to the different setups of development, integration, and production environments? The answer is to use 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 18, *EnvVar Configuration*, uses environment variables to store configuration data.
- Chapter 19, *Configuration Resource*, uses Kubernetes resources like ConfigMaps or Secrets to store configuration information.
- Chapter 20, *Immutable Configuration*, brings immutability to large configuration sets by putting it into containers linked to the application at runtime.
- Chapter 21, *Configuration Template*, is useful when large configuration files need to be managed for various environments that differ only slightly.

CHAPTER 18

# 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 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 then can 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 18-1.

*Example 18-1. 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 18-2.

*Example 18-2. 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 18-3.

*Example 18-3. 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 in Example 18-4).

*Example 18-4. 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 (like for `LOG_FILE`), but also can use a delegation to Kubernetes Secrets (for sensitive data) and ConfigMaps (for non-sensitive configuration). 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 19, *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.

## 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 they 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. All that shows us that environment variables are suitable for small 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 comes from. 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
- Immutable Server
- Spring Boot Profiles for Using Sets of Configuration Values

CHAPTER 19

# Configuration Resource

Kubernetes provides native configuration resources for regular and confidential data, which allows decoupling of 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 is that it's suitable for only a handful of variables and simple configurations. Another one 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 is not overridden in another location. For example, environment variables defined within a Docker 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 19-1.

*Example 19-1. 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 19-2.

*Example 19-2. 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 19-3.

*Example 19-3. 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 exposing 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 19-4.

*Example 19-4. Environment variable set from ConfigMap*

```
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 19-1, 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 used as a volume, the complete ConfigMap is projected into this volume, with the keys used as filenames. See Example 19-5.

*Example 19-5. 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 19-1 that is mounted as a volume results in four files in the */config* folder: An *application.properties* 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 and the filename under which it should be available. Refer to the ConfigMap documentation for more details.

### How Secure Are Secrets?

Secrets hold Base64-encoded data, and decode it prior 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 storing 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 removed when the Pod is removed.
- In Etcd, Secrets are 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. 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 application level too.

Another way to store configuration with the help of Kubernetes is the usage of `gitRepo` volumes. This type of volume mounts an empty directory on the Pod and clones a Git repository into it. The advantage of keeping configuration on Git is that you get versioning and auditing for free. But `gitRepo` volumes require external access to a Git repository, which is not a Kubernetes resource, and is possibly located outside the cluster and needs to be monitored and managed separately. The cloning and mounting happens during startup of the Pod, and the local cloned repo is not updated automatically with changes. This volume works similarly to the approach described in Chapter 20, *Immutable Configuration* by using *Init Containers* to copy configuration into a shared local volume.

In fact, volumes of type `gitRepo` are deprecated now in favor of *Init Container*-based solutions, as this approach is more generally applicable and supports other sources of data and not just Git. So going forward, you can use the same approach of retrieving configuration data from external systems and storing it into a volume, but rather than using the predefined `gitRepo` volume, use the more flexible *Init Container* method. We explain this technique in detail in "Kubernetes Init Containers" on page 159.

## Discussion

ConfigMaps and Secrets allow the storage of 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 by using configurations independently from the configurations.

Another benefit of ConfigMaps and Secrets is that they are intrinsic features of the platform. No custom construct like that in Chapter 20, *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 non-configuration 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 *Immutable Configuration* and *Configuration Templates*.

## More Information

- Configuration Resource Example
- ConfigMap Documentation
- Secrets Documentation
- Encrypting Secret Data at Rest
- Distribute Credentials Securely Using Secrets
- gitRepo Volumes
- Size Restriction of a ConfigMap

CHAPTER 20

# Immutable Configuration

The *Immutable Configuration* pattern packages configuration data into an immutable container image and links the configuration container to the application at runtime. With this pattern, we are able to 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 18, *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*. However, all of these patterns do not enforce *immutability* of the configuration data itself. 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

To address the preceding concerns, 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 environ-

ments. 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 20-1, this volume linking is an excellent way to share configuration information from a dedicated configuration container with another application container.

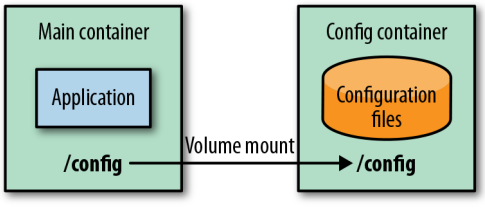


*Figure 20-1. Immutable configuration with Docker volume*

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 */config*. We can create such an image with `Dockerfile-config` as in Example 20-1.

*Example 20-1. Dockerfile for a configuration image*

```
FROM scratch

ADD app-dev.properties /config/app.properties      ❶

VOLUME /config                                     ❷
```

❶ Add the 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 20-2.

*Example 20-2. 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 20-3).

*Example 20-3. 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 within a directory */config*, 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 20-4.

*Example 20-4. 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. In order to use *Immutable Configuration* containers in Kubernetes, we can use the *Init Containers* pattern from Chapter 14 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 without any operating system files. We don't need anything more there because all we want is the configuration data shared via Docker volumes. However, for Kubernetes init containers, we need some help from the base image to copy over the configuration data to a shared Pod volume. `busybox` is a good choice

for the base image, which is still small but allows us to use 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 20-5.

*Example 20-5. 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 20-1 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 Docker image starts. This image can now be referenced in an init container within a Deployment's `.template.spec` (see Example 20-6).

*Example 20-6. 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 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 20-2 illustrates how the application container accesses the configuration data created by an init container over a shared volume.

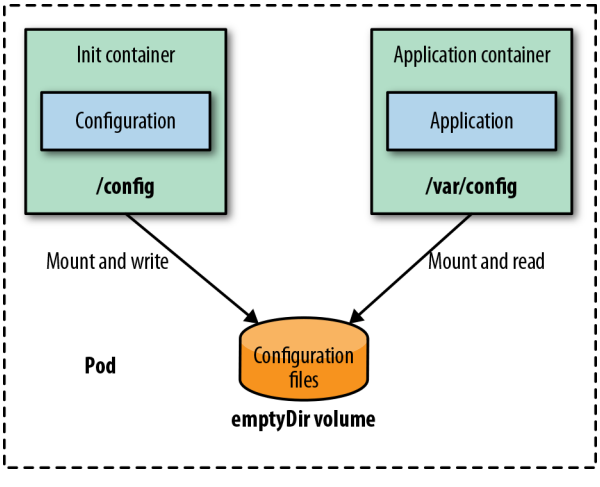


*Figure 20-2. Immutable configuration with an init container*

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

Templates are regular resource descriptors that are parameterized. As seen in Example 20-7, we can easily use the configuration image as a parameter.

*Example 20-7. 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: v1
  kind: DeploymentConfig
    // ....
    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 we 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 20-8.

*Example 20-8. 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. However, this pattern has 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 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, this 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.
- 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, we should carefully evaluate whether such an involved approach is really required. If immutability is not required, maybe a simple ConfigMap as described in Chapter 19, *Configuration Resource* is entirely sufficient.

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
- Feature Request: Image Volumes in Kubernetes
- docker-flexvol: A Kubernetes Driver that Supports Docker Volumes
- OpenShift Templates

CHAPTER 21

# Configuration Template

The *Configuration Template* pattern enables creating and processing 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 19, *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 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 the 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 JBoss WildFly

*standalone.xml*). There are many tools like *Tiller* (Ruby) or *Gomplate* (Go) for processing templates during application initialization. Figure 21-1 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. *Init Containers* are described in detail in Chapter 14.

For Kubernetes, the *Init Container* approach is the most appealing because we can use ConfigMaps directly for the template parameters. The diagram in Figure 21-1 illustrates how this pattern works.

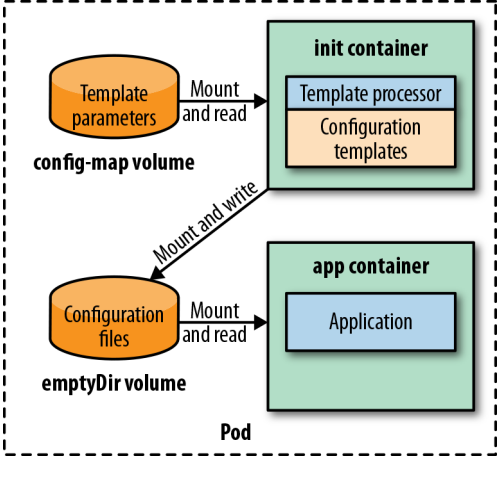


*Figure 21-1. Configuration template*

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 them-

selves. 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 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 our example GitHub repo. (We show only the main concept here; for the technical details, refer to the source repo.)

The log pattern in Example 21-1 is stored in *standalone.xml*, which we parameterize by using the Go template syntax.

*Example 21-1. 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 21-2).

*Example 21-2. Simple Dockerfile for template image*

```
FROM k8spatterns/gomplate
COPY in /in
```

The base image *k8spatterns/gomplate* contains the template processor and an entry point 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 21-3, we just use a simple file with key-value pairs.

*Example 21-3. Values for log Configuration Template*

```
logFormat: "DEVELOPMENT: %-5p %s%e%n"
```

A ConfigMap named `wildfly-parameters` contains this YAML-formatted data referenced by a key `config.yml` and is picked up by an init container.

Finally, we need the Deployment resource for the WildFly server (Example 21-4).

*Example 21-4. 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"
    ports:
    - containerPort: 8080
      name: http
      protocol: TCP
    volumeMounts:
    - mountPath: "/config"                                 ❹
      name: wildfly-config
  volumes:                                                 ❺
  - name: wildfly-parameters
    configMap:
      name: wildfly-parameters
  - name: wildfly-config
    emptyDir: {}
```

❶ Image holding the configuration templates

❷ Parameters are mounted from a ConfigMap `wildfly-parameters`

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`, contains our ConfigMap with the same name (i.e., it includes a file called *config.yml* holding our parameter value).
- 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[1] configuration without copying and maintaining duplicated large configuration files. 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.

**Volume debugging tip**

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. Just `kubectl exec` into the Pod when it is running, and examine this directory for any created files.

# Discussion

The *Configuration Template* pattern builds on top of *Configuration Resource* and is especially suited when we need to operate applications in different environments with similar complex configurations. However, the setup with *Configuration Template* 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 burden on the mainte-

1 DRY is an acronym for "Don't Repeat Yourself."

nance of that configuration because it is doomed to diverge over time. For such a situation, the template approach is perfect.

## More Information

- Configuration Template Example
- Tiller Template Engine
- Gomplate
- Go Template Syntax

# PART V
# 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* are timeless, and Kubernetes itself is built on them. However, some of the other pattern implementations are still very new (like Knative for building container images and scale-to-zero for Services) and might already be different by the time you read this book. 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 22, *Controller*, is essential to Kubernetes itself and this pattern shows how custom controllers can extend the platform.
- Chapter 23, *Operator*, combines a *Controller* with custom and domain-specific resources to encapsulate operational knowledge in an automated form.
- Chapter 24, *Elastic Scale*, describes how Kubernetes can handle dynamic loads by scaling in various dimensions.
- Chapter 25, *Image Builder*, moves the aspect of building application images into the cluster itself.

CHAPTER 22

# 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 core concept for extending the platform for our needs.

## Problem

You already have 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 question that arises here is about 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 then can 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 master 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 plug in natively other custom controllers. Custom controllers can add extra functionality to the behavior state change events, the same way as the 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 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. Kubernetes' backend is then responsible for starting up the requested Pod on a node.

Figure 22-1 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).

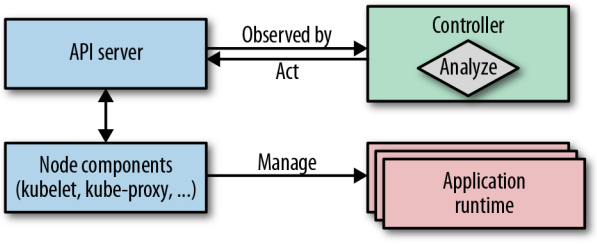


*Figure 22-1. Observe-Analyze-Act cycle*

Controllers are part of Kubernetes' control plane, and it became clear early on that they would also allow extending the platform with custom behavior. Moreover, they have turned into 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. We discuss *Operators* in depth in Chapter 23.

As stated previously, these classifications help introduce new concepts gradually. Here, we focus on the simpler *Controllers*, and in the next chapter, we introduce CRDs and build up to the *Operator* pattern.

To avoid having multiple controllers acting on the same resources simultaneously, controllers use the *SingletonService* 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 pure shell script later in Example 22-1.

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 which 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 23, *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.

*fabric8/configmapcontroller*
: This is a controller that watches ConfigMap objects for changes and performs rolling upgrades of their associated Deployments. 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. In Example 22-2, we implement such a controller with a plain shell script.

*Container Linux Update Operator*
: This is a controller that reboots a Kubernetes node when it detects a particular annotation on the node.

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 annotation value 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 22-1 is used in our web application to provide a welcome message.

*Example 22-1. 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 22-2 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 we then analyze to detect whether a changed ConfigMap carries our annotation. As events arrive, we act 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 22-2.

*Example 22-2. 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 an *Ambassador* 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 are using the Downward API described in Chapter 13, *Self Awareness* to monitor the namespace in which we have deployed the controller as configured in Example 22-3 as part of the controller Deployment.

*Example 22-3. 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 22-2. 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 master 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 17 for more details on this 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 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 22-4 is performed.

*Example 22-4. 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 key *k8spatterns.io/podDeleteSelector*. See "Some jq Fu" on page 182 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. After we have extracted the ConfigMap, 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 in "Some jq Fu" on page 182 if you are interested in how this extraction works.

## 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 a 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 an URI.

This expression converts a JSON structure like

```
{
  "k8spatterns.io/pattern": "Controller",
  "k8spatterns.io/podDeleteSelector": "app=webapp"
}
```

to a selector `app%3Dwebapp`.

If the script can extract a `selector`, we can use it now 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.

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. We use here 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 our 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 22-5 (the full version can be found in our example repository).

*Example 22-5. 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 have created previously and directly use it as the startup command for the primary container. For simplicity reasons, we omitted any liveness and readiness checks as well as resource limit declarations. Also, we need a ServiceAccount `config-`

`watcher-controller` allowed to monitor ConfigMaps. Refer to the example repository for the full security setup.

Let's see the controller in action now. 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 our example repository.

We deploy the HTTP server with a ConfigMap and Deployment as is sketched in Example 22-6.

*Example 22-6. 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 23, 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 Custom Controller in Python
- A Deep Dive into Kubernetes Controllers
- Expose Controller
- ConfigMap Controller
- Writing a Custom Controller
- Writing Kubernetes Custom Controllers
- Contour Ingress Controller
- AppController
- Characters Allowed for Labels
- Kubectl-Proxy

CHAPTER 23

# 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 22, *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. Sometimes we want to add new concepts to the Kubernetes platform, which requires additional domain objects. Let's say we chose Prometheus as our monitoring solution, and we want to add it as a monitoring facility to Kubernetes in a well-defined way. Wouldn't it be wonderful if we had a Prometheus resource describing our monitoring setup and all the deployment details, similar to the way we define other Kubernetes resources? Moreover, could we then have resources describing which services we have to monitor (e.g., with a label selector)?

These situations are precisely the kind of use cases where CustomResourceDefinitions (CRDs) 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 22, *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 now 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 eventually stored in the backend store Etcd. Historically, the predecessors of CRDs were ThirdPartyResources.

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 as in Example 23-1, which also explains most of the available fields for a CRD.

*Example 23-1. 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 the supported version

❽ Exactly one version has to be the storage version used to 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`:

*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 24, *Elastic Scale*.

*status*
: When we set this property, a new API call is available that only allows you to change the status. This API call can be secured individually and allows for status updates from outside the controller. On the other hand, when we update a custom resource as a whole, the `status` section is ignored as for standard Kubernetes resources.

Example 23-2 shows a potential subresource path as is also used for a regular Pod.

*Example 23-2. 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 23-3.

*Example 23-3. 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 for being 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.[1] 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., like 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;

1 is-a emphasizes the inheritance relationship between *Operator* and *Controller*, that an *Operator* has all characteristics of a *Controller* plus a bit more

you have no validation on the API Server level, and no support for API versioning. 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.

Another advantage of CRDs is that you have a fine-grained permission model based on the kind of CRD, which you can tune individually. 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 23-1 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.

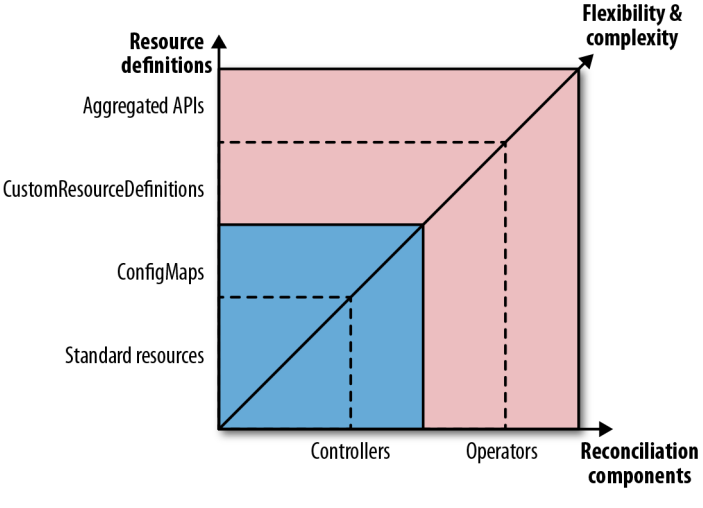


*Figure 23-1. Spectrum of Controllers and Operators*

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 an own aggregation layer. We can add a custom implemented `APIService` resource as a new URL path to the Kubernetes API.

To connect a given Service with name `custom-api-server` and backed by a Pod with your service, you can use a resource like that shown in Example 23-4.

*Example 23-4. 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.

Once 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's 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 going 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 we can develop and deploy our operators with CRDs.

## Operator Development and Deployment

At the time of this writing (2019), operator development is an area of Kubernetes that is actively evolving, with several toolkits and frameworks available for writing operators. The three main projects aiding in the creation of operators are as follows:

- CoreOS Operator Framework
- Kubebuilder developed under the SIG API Machinery of Kubernetes itself

- Metacontroller from Google Cloud Platform

We touch on them very briefly next, but be aware that all of these projects are quite young and might change over time or even get merged.

### Operator framework

The Operator Framework provides extensive support for developing Golang-based operators. It provides several subcomponents:

- The *Operator SDK* provides a high-level API for accessing a Kubernetes cluster and a scaffolding to start up an operator project
- The *Operator Lifecycle Manager* manages the release and updates of operators and their CRDs. You can think about it as a kind of "operator operator."
- *Operator Metering* enables using reporting for operators.

We won't go into much detail here about the Operator SDK, which is still evolving, but the Operator Lifecycle Manager (OLM) provides particularly valuable help when using operators. One issue with CRDs is that these resources can be registered only cluster-wide and hence require cluster-admin permissions.[2] 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 permissions 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 of its dependent CRDs to be registered. If this is the case, the OLM deploys the operator specified in the CSV. 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.

### Kubebuilder

*Kubebuilder* is a project by the SIG API Machinery with comprehensive documentation.[3] Like the Operator SDK, it supports scaffolding of Golang projects and the management of multiple CRDs within one project.

2 This restriction might be lifted in the future, as there are plans for namespace-only registration of CRDs.

3 SIGs (Special Interest Groups) is the way the Kubernetes community organizes feature areas. You can find a list of current SIGs on the Kubernetes GitHub repository.

There is a slight difference from the Operator Framework in that Kubebuilder works directly with the Kubernetes API, whereas the Operator SDK adds some extra abstraction on top of the standard API, which makes it easier to use (but lacks some bells and whistles).

The support for installing and managing the lifecycle of an operator is not as sophisticated as the OLM from the Operator Framework. However, both projects have a significant overlap, they might eventually converge in one way or another.

#### 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 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 would be 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 have a look at a concrete *Operator* example. We extend our example in Chapter 22, *Controller* and introduce a CRD of type ConfigWatcher. An instance of this

CRD specifies then a reference to the ConfigMap to watch and 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 looks like that in Example 23-5.

*Example 23-5. Simple ConfigWatcher resource*

```
kind: ConfigWatcher
apiVersion: k8spatterns.io/v1
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 can define the type of this custom resource with a CRD, as shown in Example 23-6.

*Example 23-6. 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 that is used later in a RoleBinding to attach it to the ServiceAccount in Example 23-7.

*Example 23-7. 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 23-5.

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 22-2 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 kind ConfigWatcher and check whether the modified ConfigMap is included as a `configMap:` value. Example 23-8 shows the reconciliation loop. Refer to our Git repository for the full example, which also includes detailed instructions for installing this operator.

*Example 23-8. 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.

Awesome Operators has a nice list of real-world operators that 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 of the database.

If you are looking for an operator written in the Java programming languages, 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 *JVM Operator Toolkit*, which provides a foundation for creating operators in Java and JVM-based languages like Groovy or Kotlin and also comes with a set of examples.

## Discussion

While we have seen 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 has its limitations when it comes to security or 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 at 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 already 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
- Operator Framework
- OpenAPI V3
- Kubebuilder
- Kubernetes Client Libraries
- Metacontroller
- JVM Operator Toolkit
- Extend the Kubernetes API with CustomResourceDefinitions
- Awesome Operators in the Wild
- Custom Resources Versus API Server Aggregations
- Comparison of Kubebuilder, Operator Framework, and Metacontroller
- TPR Is Dead! Kubernetes 1.7 Turns on CRD
- Code Generation for Custom Resources
- A Sample Operator in Go
- Prometheus Operator
- Etcd Operator
- Memhog Operator

CHAPTER 24

# 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. Identifying accurately 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 also can 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 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 24-1.

*Example 24-1. 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 num-

ber. 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 24-2.

*Example 24-2. 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 11, *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 creating 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.

An HPA for the `random-generator` Deployment can be created with the command in Example 24-3. 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 that you have the metrics server enabled, which is a cluster-wide aggregator of resource usage data.

*Example 24-3. 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 24-4.

*Example 24-4. HPA definition*

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: random-generator
spec:
  minReplicas: 1                    ❶
  maxReplicas: 5                    ❷
  scaleTargetRef:                   ❸
    apiVersion: extensions/v1beta1
    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.

In Example 24-4 we use the API version v2beta2 of the resource to configure the HPA. This version is in active development and is feature-wise a superset of version v1. Version v2 allows for much more criteria than the CPU usage, like memory consumption or application-specific custom metrics. By using `kubectl get hpa.v2beta2.autoscaling -o yaml`, you can easily convert a v1 HPA resource created by `kubectl autoscale` to a v2 resource.

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. 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. 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 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 24-1 shows how the HPA works: monitoring metrics and changing declared replicas accordingly.

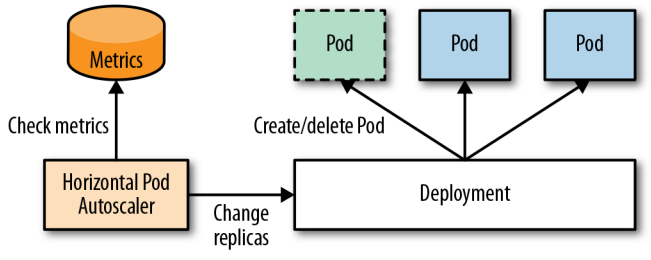


*Figure 24-1. Horizontal Pod autoscaling mechanism*

Autoscaling is an area of Kubernetes with many low-level details that are still evolving rapidly, 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 219 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` equals 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 is the container resource `requests` values and not the `limits` values. These are the easiest-to-use metric types generally provided by the metrics server or Heapster components, which can be launched as cluster addons.

*Custom metrics*
: These metrics with `.spec.metrics.resource[:].type` equals 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 `custom.met`

`rics.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. Very often in such a scenario, you may 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.

Getting autoscaling right is not easy and involves a little bit of 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 Queries-per-Second (such as HTTP requests per second) kind, 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 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 HPA more stable to 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.

### Knative Serving

Knative serving (which we introduce in "Knative Build" on page 230) allows even more advanced horizontal scaling techniques. These advanced features include "scale-to-zero" where the set of Pods backing a Service can be scaled down to 0 and scaled up only when a specific trigger happens, like an incoming request. For this to work, Knative is built on top of the service mesh Istio, which in turn provides transparent internal proxying services for Pods. Knative serving provides the foundation for a serverless framework for an even more flexible and fast horizontal scaling behavior that goes beyond the Kubernetes standard mechanisms.

A detailed discussion of Knative serving is beyond the scope of this book, as this is still a very young project and deserves its own book. We have added more links to Knative resources in "More Information" on page 219.

## 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 is for tuning the actual resource needs of a service based on actual load patterns. We have discussed why identifying the correct number of replicas of a Pod might be difficult and even impossible when 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 get 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 24-5.

*Example 24-5. VPA*

```
apiVersion: poc.autoscaling.k8s.io/v1alpha1
kind: VerticalPodAutoscaler
metadata:
  name: random-generator-vpa
spec:
  selector:
    matchLabels:          ❶
      app: random-generator
  updatePolicy:
    updateMode: "Off"     ❷
```

❶ Label 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:

*Label selector*
: Specifies what to scale by identifying the Pods it should handle.

*Update policy*
: Controls how VPA applies changes. The `Initial` mode allows assigning resource requests only during Pod creation time but 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 suggesting resource values. This is a kind of dry run for discovering the right size of a container, but without applying it directly.

A VPA definition can also have a resource policy that influences how 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, 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 current 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:

`updateMode: 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 how far 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.

`updateMode: 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 plugin, which applies the recommendations to newly created Pods only. For example, if a Pod is scaled manually through an HPA, 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 plugin* that overrides the `requests` of new Pods matching the VPA label selector. 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.

`updateMode: 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. This component evicts running Pods matching its label selector. After the eviction, the Pods get recreated by the VPA admission plugin component, which updates their resource requests. So this approach is the most disruptive as it restarts all Pods to forcefully apply the recommendations and can lead to unexpected scheduling issues as described earlier.

Kubernetes is designed to manage immutable containers with immutable Pod `spec` definitions as seen in Figure 24-2. 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 around the VPA and HPA coexistence 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 don't go into more detail here because the VPA is still in beta and may change after it's in active use. But it is a feature that has the potential to improve resource consumption significantly.

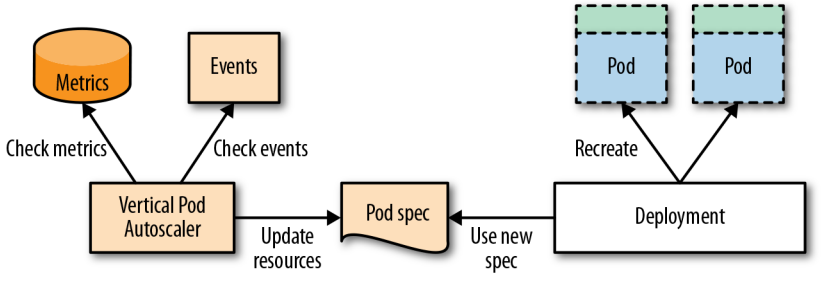


*Figure 24-2. Vertical Pod autoscaling mechanism*

## 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 elasticity and scaling of workloads, we will cover the Kubernetes Cluster Autoscaler (CA) here briefly.

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 HPA and VPA perform Pod-level scaling and ensure service capacity elasticity within a cluster, CA provides node scalability to ensure cluster capacity elasticity.

CA is a Kubernetes addon 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, Microsoft Azure, or Google Compute Engine.

### 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 there is the Cluster API Kubernetes project, which aims to provide APIs for cluster creation, configuration, and management. All major public and private cloud providers like AWS, Azure, GCE, vSphere, and OpenStack support this initiative. This also allows CA on 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.

A CA performs primarily two operations: add new nodes to a cluster or remove 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 there is no node in the cluster with 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. 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, then 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, 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:

- More than half of its capacity is unused—that is, the sum of all requested CPU and 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 still is 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 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 24-3 summarizes how the CA interacts with cloud providers and Kubernetes for scaling out cluster nodes.

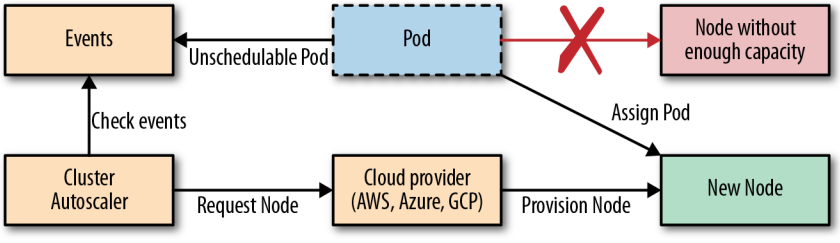


*Figure 24-3. Cluster autoscaling mechanism*

As you probably figured out by now, scaling Pods and nodes are decoupled but complementary procedures. An HPA or VPA can analyze usage metrics, events, and scale Pods. If the cluster capacity is insufficient, the CA kicks in and increases the capacity. The CA is also helpful when there are irregularities 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. In order to enable large-scale distributed system management, the automation of 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 24-4.

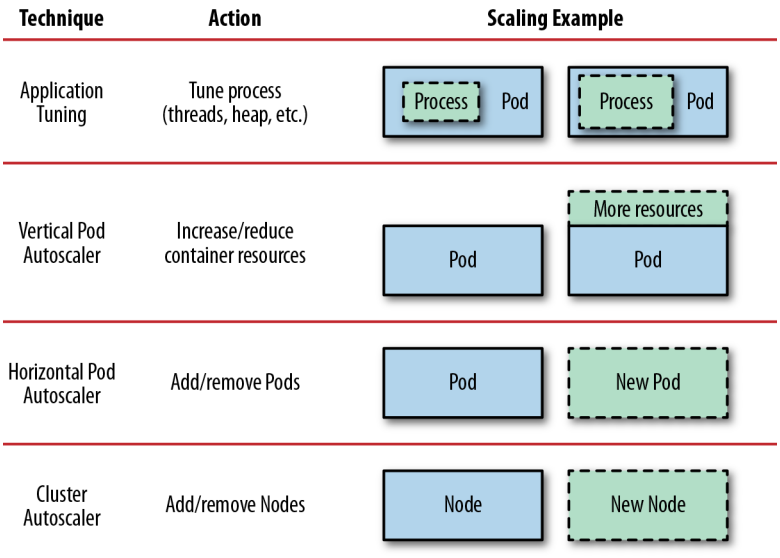


*Figure 24-4. Application-scaling levels*

### 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 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 Netflix's 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 provision-

ing 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 to get 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 for the introduction of any regression and disruption and allows more straightforward automation. HPA is currently the most popular form of scaling. While initially, it provided minimal functionality through CPU and memory metrics support only, now using custom and external metrics allow more advanced scaling use cases.

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 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. It 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 are still actively evolving. HPA recently added proper metric support, and VPA is still experimental. Also, with the popularization of the serverless programming model, scaling to zero and quick scaling has become a priority. Knative serving is a Kubernetes addon that exactly addresses this need to provide the foundation for scale-to-zero as we briefly describe in "Knative Serving" on page 210 and "Knative Build" on page 230. Knative and the underlying service meshes are progressing quickly and introduce very exciting new cloud-native primitives. We are watching this space closely and recommend you to have an eye on Knative 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 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 brittle, giving Kubernetes antifragile capabilities.

## More Information

- Elastic Scale Example
- Rightsize Your Pods with Vertical Pod Autoscaling
- Kubernetes Autoscaling 101
- Horizontal Pod Autoscaler
- HPA Algorithm
- Horizontal Pod Autoscaler Walk-Through
- Kubernetes Metrics API and Clients
- Vertical Pod Autoscaling
- Configuring Vertical Pod Autoscaling
- Vertical Pod Autoscaler Proposal
- Vertical Pod Autoscaler GitHub Repo
- Cluster Autoscaling in Kubernetes
- Adaptive Concurrency Limits
- Cluster Autoscaler FAQ
- Cluster API

- Kubermatic Machine-Controller
- OpenShift Machine API Operator
- Knative
- Knative: Serving Your Serverless Services
- Knative Tutorial

CHAPTER 25

# 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. We learned how we can develop and prepare our applications to be good cloud-native citizens. But 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 the company policies allow, having only one cluster for everything is advantageous. Building and running applications in one place can reduce maintenance costs considerably. 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 223, we will see how OpenShift implements such automation.

### Daemonless Builds

When building within Kubernetes, the cluster has full control of the build process, but needs higher security standards as a result because the build is not running in isolation anymore. To do builds in the cluster, it is essential to run the build without root privileges. Luckily, today there are many ways to achieve so-called *daemonless builds* that work without elevated privileges.

Docker was tremendously successful in bringing container technologies to the masses, thanks to its unmatched user experience. Docker is based on a client-server architecture with a Docker 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 running 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 containers because building also happens within a container when the Docker daemon executes arbitrary commands.

Many projects have been created to allow Docker builds without requiring root permissions to reduce that attack surface. Some of them don't allow running commands during building (like *Jib*), and other tools use different techniques. At the time of this writing, the most prominent daemonless image build tools are *img*, *buildah*, and *Kaniko*. Also, the S2I system explained in "Source-to-Image" on page 225 performs the image build without root permissions.

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

One of the oldest and most mature ways of building images in a Kubernetes cluster is the *OpenShift build* subsystem. It allows several ways of building images. One supported technique is *Source-to-Image* (S2I), an opinionated way of doing builds with so-called builder images. We take a closer look at S2I and the OpenShift way of building images in "OpenShift Build" on page 223.

Another mechanism for doing intracluster builds is through *Knative Build*. It works on top of Kubernetes, and the service mesh Istio and is one of the main parts of Knative, a platform for building, deploying, and managing serverless workloads. At the

time of this writing, Knative is still a very young project and is moving fast. The section "Knative Build" on page 230 gives an overview of Knative and provides examples for building images in a Kubernetes cluster with the help of Knative build.

Let's have a look at OpenShift build first.

## 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 also adds a native image building capability to Kubernetes. OKD (formerly known as OpenShift Origin) 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 a build to build jobs of an internally managed Jenkins server by allowing the user to configure a Jenkins 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 we see in Example 25-2.

*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 standalone. We will see later in Example 25-1 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 command 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, build or deployment 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, if you don't plan to use ImageStreams.

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, with two mandatory commands we have to provide:

`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 25-1. 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 compile and prepared artifacts, and the image's `run` script is set as the entry point.

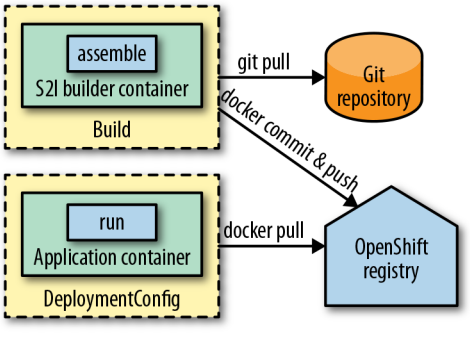


*Figure 25-1. S2I Build with Git Source as input*

Example 25-1 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 images changes.

*Example 25-1. S2I Build using a Java builder image*

```
apiVersion: 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: ImageChange
```

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 builder image is updated.

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 the internet again and again, it is recommended to 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 reusing 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 con-

tainer and the performance benefits are typically not much better than using a cluster-local proxy holding the dependencies.

Another drawback of S2I is that the generated image also contains the whole build environment. This fact not only increases the size of the application image but also increases 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 227.

### Docker builds

OpenShift also supports Docker builds directly within this 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 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 25-2. A chained build consists of an initial S2I build, which creates the runtime artifact like a binary executable. This artifact is then picked up from the generated image by a second build, typically a Docker kind of build.

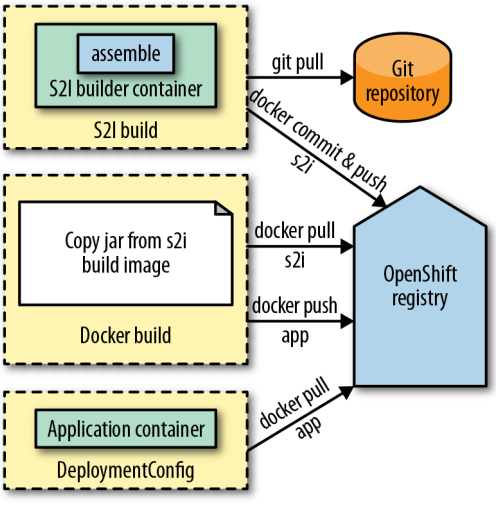


*Figure 25-2. Chained build with S2I for compiling and Docker build for application image*

Example 25-2 shows the setup of this second build config, which uses the JAR file generated in Example 25-1. The image that eventually is pushed to the ImageStream `random-generator-runtime` can be used in a DeploymentConfig to run the application.

Note that the trigger used in Example 25-2 that 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 25-2. Docker build for creating the application image.*

```
apiVersion: 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:8-alpine
      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 a Kubernetes cluster with OpenShift flavor.

Let's look at another way to build container images within a vanilla Kubernetes cluster.

## Knative Build

Google started the *Knative* project in 2018 with the aim of bringing advanced application-related functionality to Kubernetes.

The basis of Knative is a *service mesh* like Istio, which provides infrastructure services for traffic management, observability, and security out of the box. Service meshes use *Sidecars* to instrument applications with infrastructure-related functionality.

On top of the service mesh, Knative provides additional services, primarily targeted at application developers:

*Knative serving*
: For scale-to-zero support for application services, that can be leveraged by Function-as-a-Service platforms, for example. Together with the pattern described in Chapter 24, *Elastic Scale* and the underlying service mesh support, Knative serving enables scaling from zero to arbitrary many replicas.

*Knative eventing*
: A mechanism for delivering events from sources to sinks through channels. Events can trigger Services used as sinks to scale up from zero.

*Knative build*
: For compiling an application's source code to container images within a Kubernetes cluster. A follow-up project is Tekton Pipelines, which will eventually replace Knative build.

Both Istio and Knative are implemented with the *Operator* pattern and use CRDs to declare their managed domain resources.

In the remaining part of this section, we focus on Knative build as this is Knative's implementation of the *Image Builder* pattern.

Knative build is primarily targeted at tool developers to provide a user interface and seamless build experience to the end user. Here we give a high-level overview of the building blocks of Knative build. The project is rapidly moving and might even be replaced by a follow-up project like Tekton Pipelines but the principle mechanics are very likely to stay the same. Nevertheless, some details might change in the future, so refer to the example code, which we keep up-to-date with the latest Knative versions and projects.

Knative is designed to provide building blocks to integrate with your existing CI/CD solutions. It is not a CI/CD solution for building container images by itself. We expect

more and more such solutions to emerge over time, but for now let's have a look at its building blocks.

### Simple build

The Build CRD is the central element of the Knative build. The Build defines the concrete steps that the Knative build operator needs to perform for a build. Example 25-3 demonstrates the main ingredients:

- A `source` specification for pointing to the location of the application's source code. The source can be a Git repository as in this example or other remote locations like Google Cloud Storage, or even an arbitrary container from where the build operator can extract the source.
- The `steps` required for turning the source code into a runnable container image. Each step refers to a builder image that is used to perform the step. Every step has access to a volume mounted on */workspace* that contains the source code but is also used to share data among the steps.

In this example, the source is again our sample Java project hosted on GitHub and built with Maven. The builder image is a container image that contains Java and Maven. Jib is used to build the image without a Docker daemon and push it to a registry.

*Example 25-3. Knative Java build using Maven with Jib*

```
apiVersion: build.knative.dev/v1alpha1
kind: Build
metadata:
  name: random-generator-build-jib      ❶
spec:
  source:                               ❷
    git:
      url: https://github.com/k8spatterns/random-generator.git
      revision: master
  steps:                                ❸
  - name: build-and-push
    image: gcr.io/cloud-builders/mvn    ❹
    args:                               ❺
    - compile
    - com.google.cloud.tools:jib-maven-plugin:build
    - -Djib.to.image=registry/k8spatterns/random-generator
    workingDir: /workspace              ❻
```

❶ Name of the build object

❷ Source specification with a GitHub URL

❸ One or more build steps

❹ Image with Java and Maven included that is used for this build step

❺ Arguments given to the builder container that will trigger Maven to compile, create, and push a container image via the jib-maven-plugin

❻ The directory */workspace* is shared and mounted for every build step.

It's also interesting to take a brief look under the hood to learn how the Knative build *Operator* performs the build.

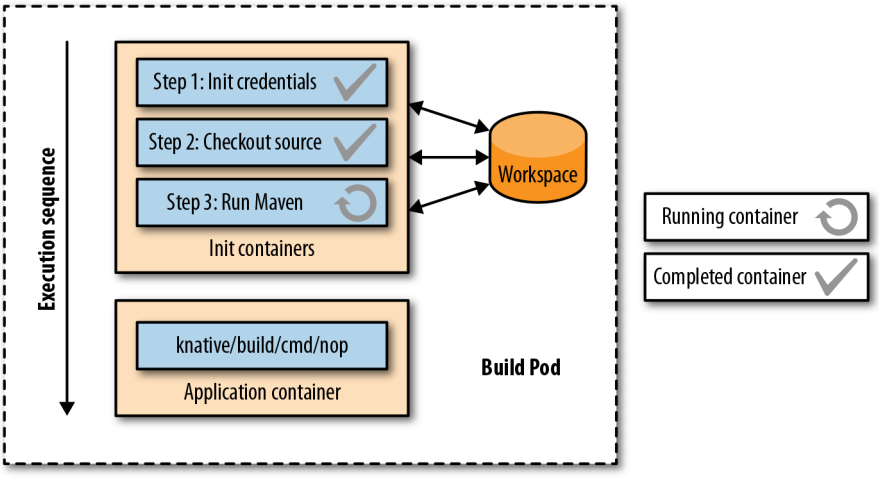


*Figure 25-3. Knative build using init containers*

Figure 25-3 shows how a custom Build resource is transformed into plain Kubernetes resources. A Build becomes a Pod with the build steps translated into a chain of *Init Containers* that are called one after another. The first init containers are implicitly created. In our example, there is one for the initializing credentials used for interacting with external repositories and a second init container for checking out the source from GitHub. The remaining init container are just the given steps with the declared builder images. When all init containers are finished, the primary container is just a no-operation that does nothing, so that the Pod stops after the initialization steps.

### Build templates

Example 25-3 contains only a single build step, but your typical build consists of multiple steps. The BuildTemplate custom resource can be used to reuse the same steps for similar builds.

Example 25-4 demonstrates such a template for a build with three steps:

1. Create a Java JAR file with `mvn package`.
2. Create a Dockerfile that copies this JAR file into a container image and starts it with `java -jar`.
3. Create and push a container image with a builder image using Kaniko. Kaniko is a tool created by Google for building container images from a Dockerfile inside a container with a local Docker daemon running in user space.

A template is more or less the same as a Build with the exception that it supports parameters such as placeholders, which are filled in when the template is used. In this example, `IMAGE` is the single parameter required to specify the target image to create.

*Example 25-4. Knative build template using Maven and Kaniko*

```
apiVersion: build.knative.dev/v1alpha1
kind: BuildTemplate
metadata:
  name: maven-kaniko
spec:
  parameters:
  - name: IMAGE                          ❶
    description: The name of the image to create and push
  steps:
  - name: maven-build                    ❷
    image: gcr.io/cloud-builders/mvn
    args:
    - package
    workingDir: /workspace
  - name: prepare-docker-context         ❸
    image: alpine
    command: [ .... ]
  - name: image-build-and-push           ❹
    image: gcr.io/kaniko-project/executor
    args:
    - --context=/workspace
    - --destination=${IMAGE}             ❺
```

❶ A list of parameters to be provided when the template is used

❷ Step for compiling and packaging a Java application by using Maven

❸ Step for creating a Dockerfile for copying and starting the generated JAR file. The details are left out here but can be found in our example GitHub repository.

❹ Step for calling Kaniko to build and push the Docker image

❺ The destination uses the provided template parameter `${IMAGE}`.

This template can then be used by a Build, which specifies the name of the template instead of a list of steps as seen in Example 25-5. As you can see, you have to specify only the name of the application container image to create, and you can easily reuse this multistep build to create different applications.

*Example 25-5. Knative build using a build template*

```
apiVersion: build.knative.dev/v1alpha1
kind: Build
metadata:
  name: random-generator-build-chained
spec:
  source:                                  ❶
    git:
      url: https://github.com/k8spatterns/random-generator.git
      revision: master
  template:                                ❷
    name: maven-kaniko
    arguments:
    - name: IMAGE                          ❸
      value: registry:80/k8spatterns/random-generator
```

❶ Source specification for where to pick up the source code

❷ Reference to template we defined in Example 25-4

❸ Image specification filled into the template as parameter

You can find many predefined templates in the Knative build-templates repository.

This example ends our quick tour through Knative build. As mentioned, this project is still very young and the details are likely to change, but the main mechanics described here should stay the same.

# Discussion

You have seen two ways to build container images within a cluster. The OpenShift build system demonstrates nicely one of the main benefits of building and running an application in the same cluster. With OpenShift's ImageStream triggers, you can not only connect multiple builds, but also redeploy your application if a build updates your application's container image. This is especially useful for lower environments where a build step is usually followed by a deploy step. 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 currently usable only when using the OpenShift distribution of Kubernetes.

Knative build is another implementation of this *Image Builder* pattern. Knative build's primary purpose is to transform source code into a runnable container image and push it to a registry so that it can be picked up by Deployments. These steps are performed by builder images, which can be provided for different technologies. Knative build is agnostic about the concrete steps of the build but cares about how to manage the build's lifecycle and how to schedule the build.

Knative build is still a young project (as of 2019) that provides the building blocks for cluster builds. It's not meant so much for the end user, but more for tool builders. You can expect new and existing tools to support Knative build or its successor projects soon, so we will see more implementations of the *Image Builder* pattern emerging.

## More Information

- ImageBuilder Examples
- Jib
- Img
- Buildah
- Kaniko
- OpenShift Builds Design Document
- Multistage Dockerfile
- Chaining S2I Builds
- Build Triggers
- Source-to-Image Specification
- Incremental S2I Builds
- Knative
- Building Container Images on Your Kubernetes Cluster with Knative Build
- Knative Build
- Tekton Pipelines
- Knative: Building Your Serverless Service
- Introducing Knctl: A Simpler Way to Work with Knative
- Interactive Knative Build Tutorial
- Knative Build Templates

# Afterword

## Ubiquitous Platform

Today, Kubernetes is the most popular container orchestration platform. It is jointly developed and supported by all major software companies and offered as a service by all the major cloud providers. It supports Linux and Windows systems and all major programming languages. Kubernetes can orchestrate and automate stateless and stateful applications, batch jobs, periodic tasks, and serverless workloads. It is the new application portability layer and the common denominator among everybody on the cloud. If you are a software developer targeting the cloud, the odds are that Kubernetes will become part of your everyday life sooner or later.

## Hybrid Responsibilities

In recent years, more and more of the application's nonfunctional requirements are provided as capabilities by cloud-native platforms. Distributed application responsibilities such as provisioning, deployment, service discovery, configuration management, job management, resource isolation, and health checks are all implemented by Kubernetes. With the popularization of microservices architecture, implementing even a simple service will require a good understanding of distributed technology stacks and container orchestration fundamentals. As a consequence, a developer has to be fluent in a modern programming language to implement the business functionality, and equally fluent in cloud-native technologies to address the nonfunctional requirements.

## What We Covered

In this book, we covered 24 of 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 the workload, a Pod may run until completion as a batch job, or be scheduled to run periodically. It can run 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 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.
- *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 new 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 for your Kubernetes journey; it is so for us.

Happy kubectl-ing.

# Index

## D



## E



## F

## Q



## R



## S

## T



## U



## V



## Z

## About the Authors

**Bilgin Ibryam** (@bibryam) is a principal architect at Red Hat, a member of Apache Software Foundation, and committer to multiple open source projects. He is a regular blogger, open source evangelist, blockchain enthusiast, speaker, and the author of *Camel Design Patterns*. He has over a decade of experience building and designing highly scalable, resilient, distributed systems.

In his day-to-day job, Bilgin enjoys mentoring, coding, and leading enterprise companies to be successful with building open source solutions. His current work focuses on application integration, enterprise blockchains, distributed system design, microservices, and cloud-native applications in general.

**Dr. Roland Huß** (@ro14nd) is a principal software engineer at Red Hat who worked as tech lead on Fuse Online and landed recently in the serverless team for coding on Knative. He has been developing in Java for over 20 years now and recently found another love with Golang. However, he has never forgotten his roots as a system administrator. Roland is an active open source contributor, lead developer of the JMX-HTTP bridge Jolokia and some popular Java build tools for creating container images and deploying them on Kubernetes and OpenShift. Besides coding, he enjoys spreading the word about his work at conferences and through his writing.

## 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."

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

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.