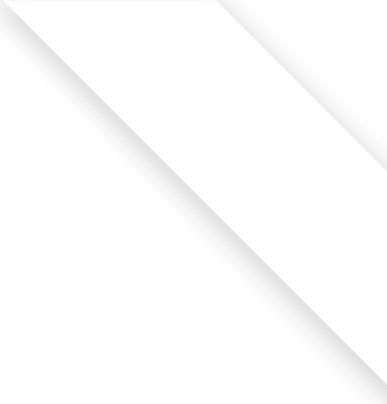
OREILLY 

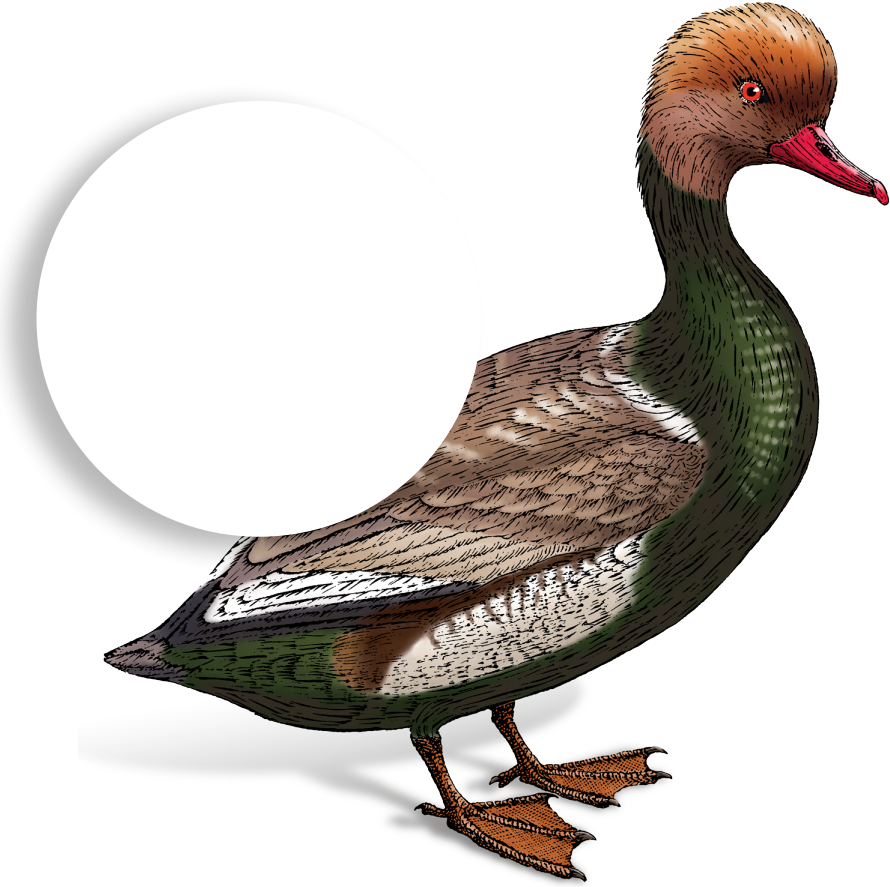
**Second**

**Edition**

**Kubernetes**

**Patterns**

Reusable Elements for Designing

Cloud Native Applications



Red Hat

Developer

**Compliments** **of**

Bilgin Ibryam &

Roland Huß Foreword by Brendan Burns

**SECOND** **EDITION**

**Kubernetes** **Patterns** ***Reusable*** ***Elements*** ***for*** ***Designing*** ***Cloud*** ***Native*** ***Applications***

***Bilgin*** ***Ibryam*** ***and*** ***Roland*** ***Huß***



Beijing · Boston · Farnham · Sebastopol · Tokyo

**Table** **of** **Contents**

**[Foreword](#bookmark1)****[xiii](#bookmark1)**

**[Preface](#bookmark2)****[xv](#bookmark2)**

**1.** **[Introduction](#bookmark3)****[1](#bookmark3)**

[The Path to Cloud Native](#bookmark4) 1

[Distributed Primitives](#bookmark5) 3

[Containers](#bookmark6) 5

[Pods](#bookmark7) 6

[Services](#bookmark8) 7

[Labels](#bookmark9) 8

[Namespaces](#bookmark10) 10

[Discussion](#bookmark11) 11

[More Information](#bookmark12) 12



**Part** **I.** **[Foundational](#bookmark13)****[Patterns](#bookmark13)**

**2.** **[Predictable](#bookmark14)****[Demands](#bookmark14)****[15](#bookmark14)**

[Problem](#bookmark15) 15

[Solution](#bookmark16) 15

[Runtime Dependencies](#bookmark17) 16

[Resource Profiles](#bookmark18) 18

[Pod Priority](#bookmark19) 21

[Project Resources](#bookmark20) 23

[Capacity Planning](#bookmark21) 25

[Discussion](#bookmark22) 26

[More Information](#bookmark23) 26

**3.** **[Declarative](#bookmark25)****[Deployment](#bookmark25)****[29](#bookmark25)**

[Problem](#bookmark26) 29

[Solution](#bookmark27) 29

[Rolling Deployment](#bookmark28) 31

[Fixed Deployment](#bookmark29) 34

[Blue-Green Release](#bookmark30) 34

[Canary Release](#bookmark31) 35

[Discussion](#bookmark32) 36

[More Information](#bookmark33) 38

**4.** **[Health](#bookmark34)****[Probe](#bookmark34)****[41](#bookmark34)**

[Problem](#bookmark35) 41

[Solution](#bookmark36) 41

[Process Health Checks](#bookmark37) 42

[Liveness Probes](#bookmark38) 42

[Readiness Probes](#bookmark39) 44

[Startup Probes](#bookmark40) 46

[Discussion](#bookmark41) 48

[More Information](#bookmark42) 49

**5.** **[Managed](#bookmark43)****[Lifecycle](#bookmark43)****[51](#bookmark43)**

[Problem](#bookmark44) 51

[Solution](#bookmark45) 51

[SIGTERM Signal](#bookmark46) 52

[SIGKILL Signal](#bookmark47) 52

[PostStart Hook](#bookmark48) 53

[PreStop Hook](#bookmark49) 54

[Other Lifecycle Controls](#bookmark50) 55

[Discussion](#bookmark51) 58

[More Information](#bookmark52) 59

**6.** **[Automated](#bookmark53)****[Placement](#bookmark53)****[61](#bookmark53)**

[Problem](#bookmark54) 61

[Solution](#bookmark55) 61

[Available Node Resources](#bookmark56) 62

[Container Resource Demands](#bookmark57) 63

[Scheduler Configurations](#bookmark58) 63

[Scheduling Process](#bookmark59) 64

[Node Affinity](#bookmark60) 66

[Pod Affinity and Anti-Affinity](#bookmark61) 67

[Topology Spread Constraints](#bookmark62) 68

[Taints and Tolerations](#bookmark63) 70

[Discussion](#bookmark64) 72 [More Information](#bookmark65) 75



**Part** **II.** **[Behavioral](#bookmark67)****[Patterns](#bookmark67)**

**7.** **[Batch](#bookmark68)****[Job](#bookmark68)****[79](#bookmark68)**

[Problem](#bookmark69) 79

[Solution](#bookmark70) 80

[Discussion](#bookmark71) 85

[More Information](#bookmark72) 86

**8.** **[Periodic](#bookmark73)****[Job](#bookmark73)****[87](#bookmark73)**

[Problem](#bookmark74) 87

[Solution](#bookmark75) 88

[Discussion](#bookmark76) 89

[More Information](#bookmark77) 90

**9.** **[Daemon](#bookmark78)****[Service](#bookmark78)****[91](#bookmark78)**

[Problem](#bookmark79) 91

[Solution](#bookmark80) 92

[Discussion](#bookmark81) 95

[More Information](#bookmark82) 95

**10.** **[Singleton](#bookmark83)****[Service](#bookmark83)****[97](#bookmark83)**

[Problem](#bookmark84) 97

[Solution](#bookmark85) 98

[Out-of-Application Locking](#bookmark86) 98

[In-Application Locking](#bookmark87) 100

[Pod Disruption Budget](#bookmark88) 103

[Discussion](#bookmark89) 104

[More Information](#bookmark90) 105

**11.** **[Stateless](#bookmark91)****[Service](#bookmark91)****[107](#bookmark91)**

[Problem](#bookmark92) 107

[Solution](#bookmark93) 108

[Instances](#bookmark94) 108

[Networking](#bookmark95) 110

[Storage](#bookmark96) 111

[Discussion](#bookmark97) 113

[More Information](#bookmark98) 114

**12.** **[Stateful](#bookmark100)****[Service](#bookmark100)****[115](#bookmark100)**

[Problem](#bookmark101) 115

[Storage](#bookmark102) 116

[Networking](#bookmark103) 116

[Identity](#bookmark104) 117

[Ordinality](#bookmark105) 117

[Other Requirements](#bookmark106) 117

[Solution](#bookmark107) 118

[Storage](#bookmark108) 119

[Networking](#bookmark109) 120

[Identity](#bookmark110) 121

[Ordinality](#bookmark111) 122

[Other Features](#bookmark112) 122

[Discussion](#bookmark113) 124

[More Information](#bookmark114) 125

**13.** **[Service](#bookmark115)****[Discovery](#bookmark115)****[127](#bookmark115)**

[Problem](#bookmark116) 127

[Solution](#bookmark117) 128

[Internal Service Discovery](#bookmark118) 129

[Manual Service Discovery](#bookmark119) 133

[Service Discovery from Outside the Cluster](#bookmark120) 135

[Application Layer Service Discovery](#bookmark121) 139

[Discussion](#bookmark122) 142

[More Information](#bookmark123) 143

**14.** **[Self](#bookmark124)****[Awareness](#bookmark124)****[145](#bookmark124)**

[Problem](#bookmark125) 145

[Solution](#bookmark126) 146

[Discussion](#bookmark127) 149

[More Information](#bookmark128) 149

**Foreword**

When Craig, Joe, and I started Kubernetes nearly eight years ago, I think we all recog ‐ nized 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 Kuber ‐ netes 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 devel ‐ opment 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 devel ‐ oper and distilled them into segments that are easy to approach and consume. As you finish this book, you will become aware not just of the components available to you in Kubernetes but also the “why” and “how” of building systems with those components.

*—* *Brendan* *Burns* *Cofounder,* *Kubernetes*

**Preface**

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

**Kubernetes**

*Kubernetes* is a container orchestration platform. The origin of Kubernetes lies some ‐ where in the Google data centers where Google’s internal container orchestration platform, [Borg](https://oreil.ly/x12HH), was born. Google used Borg for many years to run its applications. In 2014, Google decided to transfer its experience with Borg into a new open source project called “Kubernetes” (Greek for “helmsman” or “pilot”). In 2015, it became the first project donated to the newly founded Cloud Native Computing Foundation (CNCF).

From the start, Kubernetes gained a whole community of users, and the number of contributors grew incredibly fast. Today, Kubernetes is considered one of the most popular projects on GitHub. It is fair to claim that Kubernetes is the most commonly used and feature-rich container orchestration platform. Kubernetes also forms the foundation of other platforms built on top of it. The most prominent of those Platform-as-a-Service systems is Red Hat OpenShift, which provides various additional capabilities to Kubernetes. These are only some reasons we chose Kuber ‐ netes 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](#bookmark3), we recapitulate the core Kubernetes concepts and lay the foundation for the following patterns.

**Design** **Patterns**

The concept of *design* *patterns* dates back to the 1970s and is from the field of architecture.

In short, a *pattern* describes a *repeatable* *solution* *to* *a* *problem*.[1](#bookmark139) This definition works for the patterns we describe in this book, except that we probably don’t have as much variability in our solutions. A pattern is different from a recipe because instead of giving step-by-step instructions to solve a problem, it provides a blueprint for solving a whole class of similar problems. For example, the Alexandrian pattern *Beer* *Hall* describes how public drinking halls should be constructed where “strangers and friends are drinking companions” and not “anchors ofthe lonely.” All halls built after this pattern look different but share common characteristics, such as open alcoves for groups of four to eight and a place where a hundred people can meet to enjoy beverages, music, and other activities.

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



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

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

Another feature of patterns is that they follow a rigid format. However, each author defines a different form; unfortunately, there is no common standard for how pat ‐ terns should be laid out. Martin Fowler gives an excellent overview of the formats used for pattern languages at [“Writing Software Patterns”](https://oreil.ly/6IA6k).

**Who** **This** **Book** **Is** **For**

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

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

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

• This book is not an introduction to Kubernetes, nor is it a reference manual. We touch on many Kubernetes features and explain them in some detail, but we are focusing on the concepts behind those features. [Chapter 1, “Introduction”](#bookmark3), offers a brief refresher on Kubernetes basics. If you are looking for a comprehensive book on Kubernetes, we highly recommend *Kubernetes* *in* *Action* by Marko Lukša (Manning Publications).

• This book is not a step-by-step guide on how to set up a Kubernetes cluster itself. Every example assumes you have Kubernetes up and running. You have several

options for trying out the examples. If you are interested in learning how to set up a Kubernetes cluster, we recommend *[Kubernetes:](https://learning.oreilly.com/library/view/kubernetes-up-and/9781098110192)**[Up](https://learning.oreilly.com/library/view/kubernetes-up-and/9781098110192)**[and](https://learning.oreilly.com/library/view/kubernetes-up-and/9781098110192)**[Running](https://learning.oreilly.com/library/view/kubernetes-up-and/9781098110192)* by Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson (O’Reilly).

• This book is not about operating and governing a Kubernetes cluster for other teams. We deliberately skipped administrative and operational aspects of Kuber ‐ netes and took a developer-first view into Kubernetes. This book can help opera ‐ tions teams understand how a developer uses Kubernetes, but it is not sufficient for administering and automating a Kubernetes cluster. If you are interested in learning how to operate a Kubernetes cluster, we recommend *[Kubernetes](https://learning.oreilly.com/library/view/kubernetes-best-practices/9781492056461/)**[Best](https://learning.oreilly.com/library/view/kubernetes-best-practices/9781492056461/)* *[Practices](https://learning.oreilly.com/library/view/kubernetes-best-practices/9781492056461/)* by Brendan Burns, Eddie Villalba, Dave Strebel, and Lachlan Evenson (O’Reilly).

**What** **You** **Will** **Learn**

There’s a lot to discover in this book. Some patterns may read like excerpts from a Kubernetes manual at first glance, but upon closer look, you’ll see the patterns are presented from a conceptual angle not found in other books on the topic. Other patterns are explained with detailed steps to solve a concrete problem, as in Part IV, “Configuration Patterns”. In some chapters, we explain Kubernetes features that don’t fit nicely into a pattern definition. Don’t get hung up on whether it is a pattern or a feature. In all chapters, we look at the forces involved from the first principles and focus on the use cases, lessons learned, and best practices. That is the valuable part. Regardless of the pattern granularity, you will learn everything Kubernetes offers for each particular pattern, with plenty of examples to illustrate the concepts. All these examples have been tested, and we tell you how to get the complete source code in [“Using Code Examples” on page xxi](#bookmark142).

**CHAPTER** **1**

**Introduction**

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

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

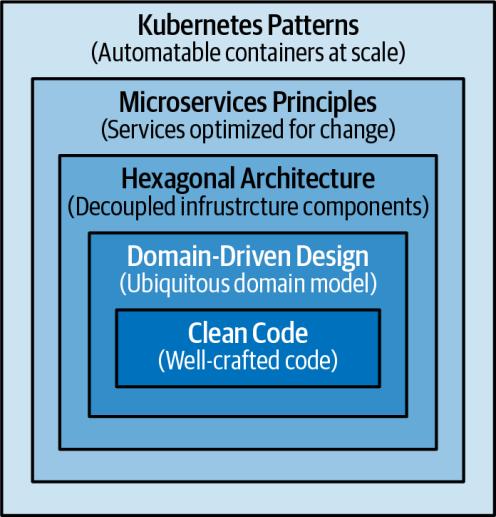
**The** **Path** **to** **Cloud** **Native**

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

As part of the microservices movement, there is a tremendous amount of theory, techniques, and supplemental tools for creating microservices from scratch or for splitting monoliths into microservices. Most of these practices are based on *[Domain-](https://oreil.ly/UoON5)* *[Driven](https://oreil.ly/UoON5)**[Design](https://oreil.ly/UoON5)* by Eric Evans (Addison-Wesley) and the concepts of bounded contexts and aggregates. *Bounded* *contexts* deal with large models by dividing them into differ ‐ ent components, and *aggregates* help to further group bounded contexts into modules with defined transaction boundaries. However, in addition to these business domain considerations, for each distributed system—whether it is based on microservices or not—there are also technical concerns around its external structure, and runtime coupling. Containers and container orchestrators such as Kubernetes bring in new

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

Throughout this book, we look at container and platform interactions by treating the containers as black boxes. However, we created this section to emphasize the importance of what goes into containers. Containers and cloud native platforms bring tremendous benefits to your distributed applications, but if all you put into containers is rubbish, you will get distributed rubbish at scale. [Figure 1-1](#bookmark147) shows the mixture of the skills required for creating good cloud native applications and where Kubernetes patterns fit in.



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

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

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

• *Domain-driven* *design* is about approaching software design from a business perspective with the intention of keeping the architecture as close to the real world as possible. This approach works best for object-oriented programming languages, but there are also other good ways to model and design software for real-world problems. A model with the right business and transaction bound ‐ aries, easy-to-consume interfaces, and rich APIs is the foundation for successful containerization and automation later.

• The *hexagonal* *architecture* and its variations, such as Onion and Clean architec ‐ tures, improve the flexibility and maintainability of applications by decoupling the application components and providing standardized interfaces for interacting with them. By decoupling the core business logic of a system from the surround ‐ ing infrastructure, hexagonal architecture makes it easier to port the system to different environments or platforms. These architectures complement domain- driven design and help arrange application code with distinct boundaries and externalized infrastructure dependencies.

• The *microservices* *architectural* *style* and the [twelve-factor app](https://12factor.net) methodology very quickly evolved to become the norm for creating distributed applications and they provide valuable principles and practices for designing changing distributed applications. Applying these principles lets you create implementations that are optimized for scale, resiliency, and pace of change, which are common require ‐ ments for any modern software today.

• *Containers* were very quickly adopted as the standard way of packaging and running distributed applications, whether these are microservices or functions. Creating modular, reusable containers that are good cloud native citizens is another fundamental prerequisite. *Cloud* *native* is a term used to describe princi ‐ ples, patterns, and tools to automate containerized applications at scale. We use *cloud* *native* interchangeably with *Kubernetes*, which is the most popular open source cloud native platform available today.

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

**Distributed** **Primitives**

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

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

We still need to use the object-oriented building blocks to create the components of the distributed application, but we can also use Kubernetes primitives for some of the application behaviors. [Table 1-1](#bookmark149) shows how various development concepts are realized differently with local and distributed primitives in the JVM and Kubernetes, respectively.

*Table* *1-1.* *Local* *and* *distributed* *primitives*

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

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

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

**Containers**

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

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

The parallel doesn’t go much further, but the point is that containers play a fun ‐ damental 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 con ‐ tainer image that provide packaging and isolation, what does a container represent, and what is its purpose in the context of a distributed application? Here are a few suggestions on how to look at containers:

• A container image is the unit of functionality that addresses a single concern.

• A container image is owned by one team and has its own release cycle.

• A container image is self-contained and defines and carries its runtime dependencies.

• A container image is immutable, and once it is built, it does not change; it is configured.

• A container image defines its resource requirements and external dependencies.

• A container image has well-defined APIs to expose its functionality.

• A container typically runs as a single Unix process.

• A container is disposable and safe to scale up or down at any moment.

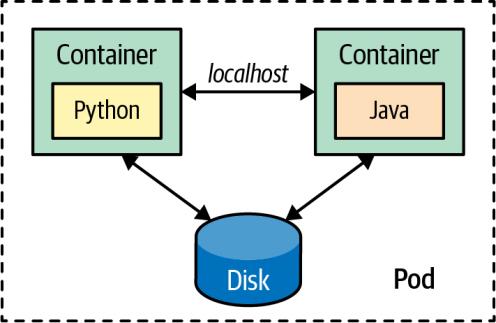
In addition to all these characteristics, a proper container image is modular. It is parameterized and created for reuse in the different environments in which it is going to run. Having small, modular, and reusable container images leads to the creation of more specialized and stable container images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

**Pods**

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

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

As you can see in [Figure 1-2](#bookmark151), at development and build time, a microservice corre ‐ sponds to a container image that one team develops and releases. But at runtime, a microservice is represented by a Pod, which is the unit of deployment, placement, and scaling. The only way to run a container—whether for scale or migration—is through the Pod abstraction. Sometimes a Pod contains more than one container. In one such example, a containerized microservice uses a helper container at runtime, as Chapter 16, “Sidecar”, demonstrates.



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

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

• A Pod is the atomic unit of scheduling. That means the scheduler tries to find a host that satisfies the requirements of all containers that belong to the Pod (we cover some specifics around init containers in Chapter 15, “Init Container”). If you create a Pod with many containers, the scheduler needs to find a host that has enough resources to satisfy all container demands combined. This scheduling process is described in [Chapter 6, “Automated Placement”](#bookmark53).

• A Pod ensures colocation of containers. Thanks to the colocation, containers in the same Pod have additional means to interact with one another. The most common ways of communicating include using a shared local filesystem for exchanging data, using the localhost network interface, or using some host inter ‐ process communication (IPC) mechanism for high-performance interactions.

• A Pod has an IP address, name, and port range that are shared by all containers belonging to it. That means containers in the same Pod have to be carefully configured to avoid port clashes, in the same way that parallel, running Unix processes have to take care when sharing the networking space on a host.

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

**Services**

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

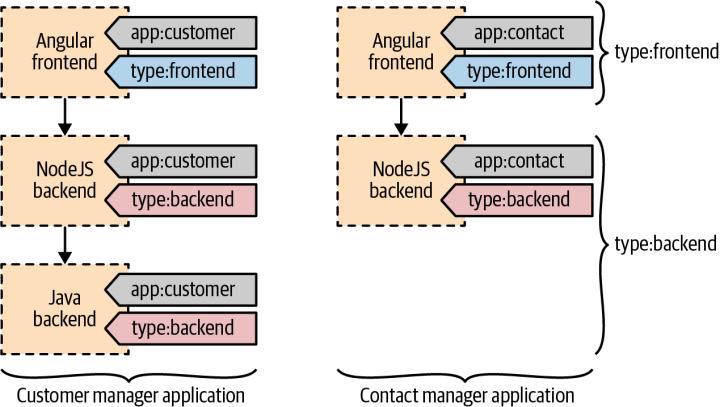
That’s where the Kubernetes Services come into play. The Service is another simple but powerful Kubernetes abstraction that binds the Service name to an IP address and port number permanently. So a Service represents a named entry point for accessing an application. In the most common scenario, the Service serves as the entry point for a set of Pods, but that might not always be the case. The Service is a generic primitive, and it may also point to functionality provided outside the Kuber ‐ netes cluster. As such, the Service primitive can be used for Service discovery and load balancing, and it allows altering implementations and scaling without affecting Service consumers. We explain Services in detail in [Chapter 13, “Service Discovery”](#bookmark115) .

**Labels**

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

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

We now have five Pod definitions (and maybe many more Pod instances) that are independent of the development and runtime points of view. However, we may still need to indicate that the first three Pods represent an application and the other two Pods represent another application. Even the Pods may be independent, to provide a business value, but they may depend on one another. For example, one Pod may contain the containers responsible for the frontend, and the other two Pods are responsible for providing the backend functionality. If either of these Pods is down, the application is useless from a business point of view. Using label selectors gives us the ability to query and identify a set of Pods and manage it as one logical unit. [Figure 1-3](#bookmark153) shows how you can use labels to group the parts of a distributed application into specific subsystems.



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

Here are a few examples where labels can be useful:

• Labels are used by ReplicaSets to keep some instances of a specific Pod running. That means every Pod definition needs to have a unique combination of labels used for scheduling.

• Labels are also heavily used by the scheduler. The scheduler uses labels for colocating or spreading Pods to the nodes that satisfy the Pods’ requirements.

• A label can indicate a logical grouping of a set of Pods and give an application identity to them.

• In addition to the preceding typical use cases, labels can be used to store meta ‐ data. 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 exam ‐ ple, having labels to indicate the logical group of an application, the business characteristics and criticality, the specific runtime platform dependencies such as hardware architecture, or location preferences are all useful.

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

|  |
| --- |
| **Annotations**  Another primitive very similar to labels is the *annotation*. Like labels, annotations are organized as a map, but they are intended for specifying nonsearchable metadata and for machine usage rather than human.  The information on the annotations is not intended for querying and matching objects. Instead, it is intended for attaching additional metadata to objects from various tools and libraries we want to use. Some examples of using annotations include build IDs, release IDs, image information, timestamps, Git branch names, pull request numbers, image hashes, registry addresses, author names, tooling infor ‐ mation, and more. So while labels are used primarily for query matching and per ‐ forming actions on the matching resources, annotations are used to attach metadata that can be consumed by a machine. |

**Namespaces**

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

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

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

• A namespace is managed as a Kubernetes resource.

• A namespace provides scope for resources such as containers, Pods, Services, or ReplicaSets. The names of resources need to be unique within a namespace but not across them.

• By default, namespaces provide scope for resources, but nothing isolates those resources and prevents access from one resource to another. For example, a Pod from a development namespace can access another Pod from a production namespace as long as the Pod IP address is known. “Network isolation across namespaces for creating a lightweight multitenancy solution is described in Chapter 24, “Network Segmentation”.

• Some other resources, such as namespaces, nodes, and PersistentVolumes, do not belong to namespaces and should have unique cluster-wide names.

• Each Kubernetes Service belongs to a namespace and gets a corresponding Domain Name Service (DNS) record that has the namespace in the form of <service-name>.<namespace-name>.svc.cluster.local. So the namespace name is in the URL of every Service belonging to the given namespace. That’s one reason it is vital to name namespaces wisely.

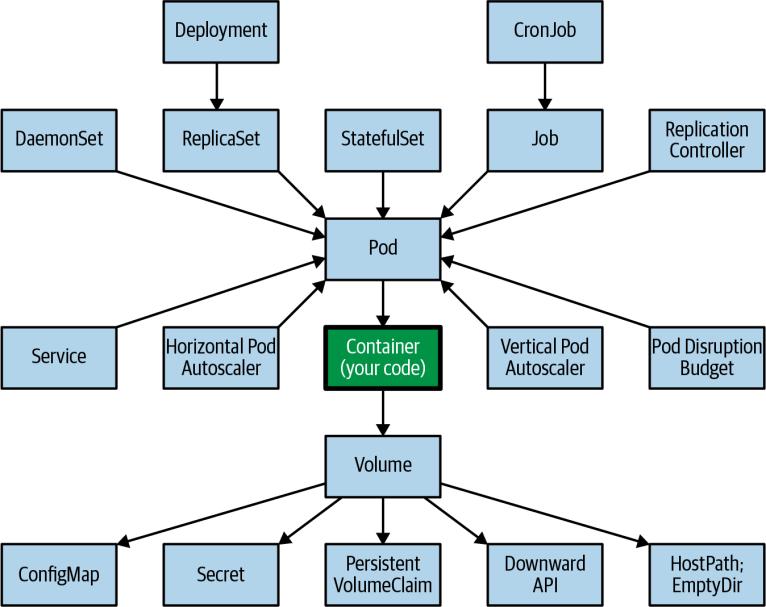
• ResourceQuotas provide constraints that limit the aggregated resource consump ‐ tion per namespace. With ResourceQuotas, a cluster administrator can control the number of objects per type that are allowed in a namespace. For example, a

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

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

**Discussion**

We’ve only briefly covered a few of the main Kubernetes concepts we use in this book. However, there are more primitives used by developers on a day-by-day basis. For example, if you create a containerized service, there are plenty of Kubernetes abstrac ‐ tions you can use to reap all the benefits of Kubernetes. Keep in mind, these are only a few of the objects used by application developers to integrate a containerized service into Kubernetes. There are plenty of other concepts used primarily by cluster administrators for managing Kubernetes. [Figure 1-4](#bookmark11) gives an overview of the main Kubernetes resources that are useful for developers.



*Figure* *1-4.* *Kubernetes* *concepts* *for* *developers*

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

**More** **Information**

• [The Twelve-Factor App](https://oreil.ly/ad0al)

• [CNCF Cloud Native Definition v1.0](https://oreil.ly/NUiXM)

• [Hexagonal Architecture](https://oreil.ly/rvcDB)

• *[Domain-Driven](https://oreil.ly/8IHI4)**[Design:](https://oreil.ly/8IHI4)**[Tackling](https://oreil.ly/8IHI4)**[Complexity](https://oreil.ly/8IHI4)**[in](https://oreil.ly/8IHI4)**[the](https://oreil.ly/8IHI4)**[Heart](https://oreil.ly/8IHI4)**[of](https://oreil.ly/8IHI4)**[Software](https://oreil.ly/8IHI4)*

• [Best Practices for Writing Dockerfiles](https://oreil.ly/Be0g6)

• [Principles of Container-Based Application Design](https://oreil.ly/-x16l)

• [General Container Image Guidelines](https://oreil.ly/yyItc)

**PART** **I**

**Foundational** **Patterns**

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

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

• [Chapter 2, “Predictable Demands”](#bookmark14), explains why every container should declare its resource requirements and stay confined to the indicated resource boundaries.

• [Chapter 3, “Declarative Deployment”](#bookmark25), describes the different application deploy‐ ment strategies that can be expressed in a declarative way.

• [Chapter 4, “Health Probe”](#bookmark34), dictates that every container should implement spe ‐ cific APIs to help the platform observe and maintain the application healthily.

• [Chapter 5, “Managed Lifecycle”](#bookmark43), explains why a container should have a way to read the events coming from the platform and conform by reacting to those events.

• [Chapter 6, “Automated Placement”](#bookmark53), introduces the Kubernetes scheduling algo ‐ rithm and the ways to influence the placement decisions from the outside.

**Predictable** **Demands**

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

**Problem**

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

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

**Solution**

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

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

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

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

**Runtime** **Dependencies**

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

The most straightforward type of volume is emptyDir, which lives as long as the Pod lives. When the Pod is removed, its content is also lost. The volume needs to be backed by another kind of storage mechanism to survive Pod restarts. If your application needs to read or write files to such long-lived storage, you must declare that dependency explicitly in the container definition using volumes, as shown in [Example 2-1](#bookmark157).

*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** : [](#bookmark159)

**claimName** : random-generator-log

[](#bookmark158) Dependency of a PersistentVolumeClaim (PVC) to be present and bound.

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

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

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

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

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

ConfigMaps and Secrets are explained in more detail in Chapter 20, “Configura‐ tion Resource”, and [Example 2-2](#bookmark161) 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**: [0](#bookmark163)

**name** : random-generator-config

**key** : pattern

[](#bookmark162) Mandatory dependency on the ConfigMap random-generator-config.

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

**Resource** **Profiles**

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

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

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

-Xmx command-line options.

The requests amount (but not limits) is used by the scheduler when placing Pods to nodes. For a given Pod, the scheduler considers only nodes that still have enough capacity to accommodate the Pod and all of its containers by summing up the requested resource amounts. In that sense, the requests field of each container affects where a Pod can be scheduled or not. [Example 2-3](#bookmark164) 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** : [o](#bookmark167)

**cpu** : 100m

**memory** : 200Mi

**limits** : [](#bookmark169)

**memory** : 200Mi

[](#bookmark166) Initial resource request for CPU and memory.

[](#bookmark168) Upper limit until we want our application to grow at max. We don’t specify CPU limits by intention.

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

memory

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

cpu

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

ephemeral-storage

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

hugepage-<size>

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

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

*Best-Effort*

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

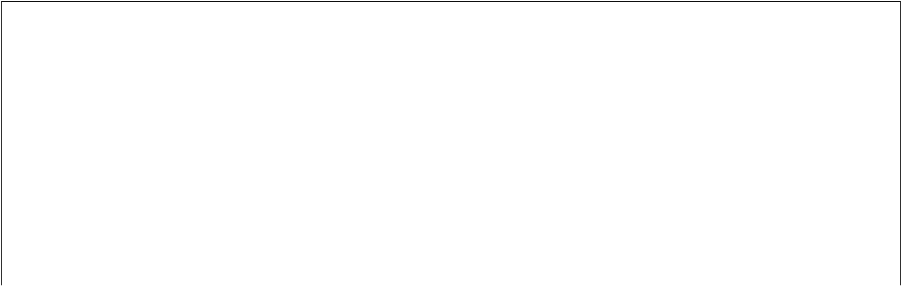
*Burstable*

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

*Guaranteed*

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

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

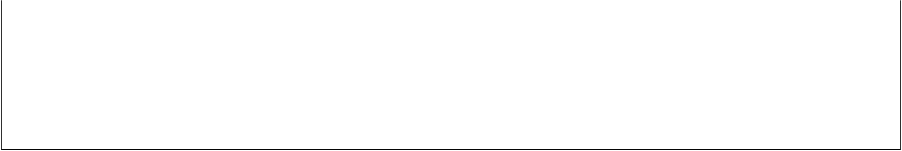


**Recommendations** **for** **CPU** **and** **Memory** **Resources**

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

• For memory, always set requests equal to limits.

• For CPU, set requests but no limits.



See the blog post [“For the Love of God, Stop Using CPU Limits on Kubernetes”](https://oreil.ly/HcMw5) for a more in-depth explanation of why you should not use limits for the CPU, and see the blog post [“What Everyone Should Know About Kubernetes Memory Limits”](https://oreil.ly/Lb_N9) for more details about the recommended memory settings.

**Pod** **Priority**

We explained how container resource declarations also define Pods’ QoS and affect the order in which the Kubelet kills the container in a Pod in case of resource starvation. Two other related concepts are Pod priority and preemption. *Pod* *priority* allows you to indicate the importance of a Pod relative to other Pods, which affects the order in which Pods are scheduled. Let’s see that in action in [Example 2-4](#bookmark170).

*Example* *2-4.* *Pod* *priority*

**apiVersion** : scheduling.k8s.io/v1

**kind** : PriorityClass

**metadata** :

**name** : high-priority [o](#bookmark172)

**value** : 1000 [](#bookmark174)

**globalDefault** : false [](#bookmark176)

**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 [](#bookmark178)

[](#bookmark171) The name of the priority class object.

[](#bookmark173) The priority value of the object.

[](#bookmark175) globalDefault set to true is used for Pods that do not specify a priorityClass Name. Only one PriorityClass can have globalDefault set to true.

[](#bookmark177) The priority class to use with this Pod, as defined in PriorityClass resource.

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

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

Pod priority affects the order in which the scheduler places Pods on nodes. First, the priority admission controller uses the priorityClassName field to populate the priority value for new Pods. When multiple Pods are waiting to be placed, the scheduler sorts the queue of pending Pods by highest priority first. Any pending Pod is picked before any other pending Pod with lower priority in the scheduling queue, and if there are no constraints preventing it from scheduling, the Pod gets scheduled. Here comes the critical part. If there are no nodes with enough capacity to place a Pod, the scheduler can preempt (remove) lower-priority Pods from nodes to free up resources and place Pods with higher priority. As a result, the higher-priority Pod might be scheduled sooner than Pods with a lower priority if all other scheduling requirements are met. This algorithm effectively enables cluster administrators to control which Pods are more critical workloads and place them first by allowing the scheduler to evict Pods with lower priority to make room on a worker node for higher-priority Pods. If a Pod cannot be scheduled, the scheduler continues with the placement of other lower-priority Pods.

Suppose you want your Pod to be scheduled with a particular priority but don’t want to evict any existing Pods. In that case, you can mark a PriorityClass with the field preemptionPolicy: Never. Pods assigned to this priority class will not trigger any eviction of running Pods but will still get scheduled according to their priority value. Pod QoS (discussed previously) and Pod priority are two orthogonal features that are not connected and have only a little overlap. QoS is used primarily by the Kubelet to preserve node stability when available compute resources are low. The Kubelet first considers QoS and then the PriorityClass of Pods before eviction. On the other hand, the scheduler eviction logic ignores the QoS of Pods entirely when choosing preemption targets. The scheduler attempts to pick a set of Pods with the lowest priority possible that satisfies the needs of higher-priority Pods waiting to be placed. When Pods have a priority specified, it can have an undesired effect on other Pods that are evicted. For example, while a Pod’s graceful termination policies are respec ‐ ted, the PodDisruptionBudget as discussed in [Chapter 10, “Singleton Service”](#bookmark83), is not guaranteed, which could break a lower-priority clustered application that relies on a quorum of Pods.

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

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

**Project** **Resources**

Kubernetes is a self-service platform that enables developers to run applications as they see suitable on the designated isolated environments. However, working in a shared multitenanted platform also requires the presence of specific boundaries and control units to prevent some users from consuming all the platform’s resources. One such tool is ResourceQuota, which provides constraints for limiting the aggregated resource consumption in a namespace. With ResourceQuotas, the cluster administra ‐ tors can limit the total sum of computing resources (CPU, memory) and storage consumed. It can also limit the total number of objects (such as ConfigMaps, Secrets, Pods, or Services) created in a namespace. [Example 2-5](#bookmark181) shows an instance that limits the usage of certain resources. See the official Kubernetes documentation on [Resource Quotas](https://oreil.ly/TLRMe) for the full list of supported resources for which you can restrict usage with ResourceQuotas.

*Example* *2-5.* *Definition* *of* *resource* *constraints*

**apiVersion** : v1

**kind** : ResourceQuota

**metadata** :

**name** : object-counts

**namespace** : default [o](#bookmark183)

**spec** :

**hard** :

**pods** : 4 [](#bookmark185)

**limits.memory** : 5Gi [](#bookmark187)

[](#bookmark182) Namespace to which resource constraints are applied.

[](#bookmark184) Allow four active Pods in this namespace.

[](#bookmark186) The sum of all memory limits of all Pods in this namespace must not be more than 5 GB.

Another helpful tool in this area is LimitRange, which allows you to set resource usage limits for each type of resource. In addition to specifying the minimum and maximum permitted amounts for different resource types and the default values for these resources, it also allows you to control the ratio between the requests and limits, also known as the *overcommit* *level*. [Example 2-6](#bookmark188) shows a LimitRange and the possible configuration options.

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

**apiVersion** : v1

**kind** : LimitRange

**metadata** :

**name** : limits

**namespace** : default

**spec** :

**limits** :

- **min** : [o](#bookmark190)

**memory** : 250Mi

**cpu** : 500m

**max** : [](#bookmark192)

**memory** : 2Gi

**cpu** : 2

**default** : [](#bookmark194)

**memory** : 500Mi

**cpu** : 500m

**defaultRequest** : [](#bookmark196)

**memory** : 250Mi

**cpu** : 250m

**maxLimitRequestRatio** : [](#bookmark198)

**memory** : 2

**cpu** : 4

**type** : Container [](#bookmark200)

[o](#bookmark189) Minimum values for requests and limits.

[](#bookmark191) Maximum values for requests and limits.

[](#bookmark193) Default values for limits when no limits are specified.

[](#bookmark195) Default values for requests when no requests are specified.

[](#bookmark197) Maximum ratio limit/request, used to specify the allowed overcommit level.

Here, the memory limit must not be larger than twice the memory request, and the CPU limit can be as high as four times the CPU request.

[](#bookmark199) Type can be Container, Pod, (for all containers combined), or

PersistentVolumeClaim (to specify the range for a request persistent volume).

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

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

**Capacity** **Planning**

Considering that containers may have different resource profiles in different environ ‐ ments, 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 *Bursta-* *ble* containers. In such a dynamic environment, many containers are starting up and shutting down at the same time, and even if a container gets killed by the platform during resource starvation, it is not fatal. On the production cluster, where we want things to be more stable and predictable, the containers may be mainly of the *Guaranteed* type, and some may be *Burstable*. If a container gets killed, that is most likely a sign that the capacity of the cluster should be increased.

[Table 2-1](#bookmark202) presents a few services with CPU and memory demands.

*Table* *2-1.* *Capacity* *planning* *example*

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

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

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

**Discussion**

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

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

However, more importantly, resource profiles are the way an application commu ‐ nicates 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 Deploy‐](#bookmark25) [ment”](#bookmark25), you will learn multiple strategies to install and update our applications on Kubernetes.

**More** **Information**

• [Predictable Demands Example](https://oreil.ly/HYIqJ)

• [Configure a Pod to Use a ConfigMap](https://oreil.ly/c54Gh)

• [Kubernetes Best Practices: Resource Requests and Limits](https://oreil.ly/8bKD5)

• [Resource Management for Pods and Containers](https://oreil.ly/a37eO)

• [Manage HugePages](https://oreil.ly/RXQD1)

• [Configure Default Memory Requests and Limits for a Namespace](https://oreil.ly/ozlU1)

• [Node-Pressure Eviction](https://oreil.ly/fxRvs)

• [Pod Priority and Preemption](https://oreil.ly/FpUoH)

• [Configure Quality of Service for Pods](https://oreil.ly/x07OT)

• [Resource Quality of Service in Kubernetes](https://oreil.ly/yORlL)

• [Resource Quotas](https://oreil.ly/rFSLa)

• [Limit Ranges](https://oreil.ly/1bXfO)

• [Process ID Limits and Reservations](https://oreil.ly/lkmMK)

• [For the Love of God, Stop Using CPU Limits on Kubernetes](https://oreil.ly/Yk-Ag)

• [What Everyone Should Know About Kubernetes Memory Limits](https://oreil.ly/cdJkP)

**Declarative** **Deployment**

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

**Problem**

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

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

**Solution**

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

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

In [Chapter 2, “Predictable Demands”](#bookmark14), we saw that, to do its job effectively, the scheduler requires sufficient resources on the host system, appropriate placement policies, and containers with adequately defined resource profiles. Similarly, for a Deployment to do its job correctly, it expects the containers to be good cloud native citizens. At the very core of a Deployment is the ability to start and stop a set of Pods predictably. For this to work as expected, the containers themselves usually listen and honor lifecycle events (such as SIGTERM; see [Chapter 5, “Managed Lifecycle”](#bookmark43)) and also provide health-check endpoints as described in [Chapter 4, “Health Probe”](#bookmark34), 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 remain ‐ ing aspects of an update process can be defined in a declarative way and executed as one atomic action with predefined steps and an expected outcome. Let’s see the options for a container update behavior.

|  |
| --- |
| **Deployment** **Updates** **with** **kubectl** **rollout**  In previous versions of Kubernetes, rolling updates were implemented on the client side with the kubectl rolling-update command. In Kubernetes 1.18, rolling- update was removed in favor of a rollout command for kubectl. The difference is that kubectl rollout manages an application update on the server side by updating the Deployment *declaration* and leaving it to Kubernetes to perform the update. The kubectl rolling-update command, in contrast, was *imperative*: the client kubectl told the server what to do for each update step.  A Deployment can be fully managed by updating the Kubernetes resources files. However, kubectl rollout comes in very handy for everyday rollout tasks:  kubectl rollout status  Shows the current status of a Deployment’s rollout.  kubectl rollout pause  Pauses a rolling update so that multiple changes can be applied to a Deployment without retriggering another rollout.  kubectl rollout resume  Resumes a previously paused rollout.  kubectl rollout undo  Performs a rollback to a prevision revision of a Deployment. A rollback is helpful in case of an error during the update. |

|  |
| --- |
| kubectl rollout history  Shows the available revisions of a Deployment.  kubectl rollout restart  Does not perform an update but restarts the current set of Pods belonging to a Deployment using the configured rollout strategy.  You can find usage examples for kubectl rollout commands in the [examples](https://oreil.ly/IrZR3). |

**Rolling** **Deployment**

The declarative way of updating applications in Kubernetes is through the concept of Deployment. Behind the scenes, the Deployment creates a ReplicaSet that supports set-based label selectors. Also, the Deployment abstraction allows you to shape the update process behavior with strategies such as RollingUpdate (default) and Recreate. [Example 3-1](#bookmark205) 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 [o](#bookmark207)

**strategy** :

**type** : RollingUpdate

**rollingUpdate** :

**maxSurge** : 1 [](#bookmark209)

**maxUnavailable** : 1 [](#bookmark211)

**minReadySeconds** : 60 [](#bookmark213)

**selector** :

**matchLabels** :

**app** : random-generator

**template** :

**metadata** :

**labels** :

**app** : random-generator

**spec** :

**containers** :

- **image** : k8spatterns/random-generator:1.0

**name** : random-generator

**readinessProbe** : [](#bookmark215)

**exec** :

**command** : [ "stat", "/random-generator-ready" ]

[](#bookmark206) Declaration of three replicas. You need more than one replica for a rolling update to make sense.

[](#bookmark208) Number of Pods that can be run temporarily in addition to the replicas specified

during an update. In this example, it could be a maximum of four replicas.

[](#bookmark210) Number of Pods that may be unavailable during the update. Here it could be that only two Pods are available at a time during the update.

[](#bookmark212) Duration in seconds of all readiness probes for a rolled-out Pod needs to be healthy until the rollout continues.

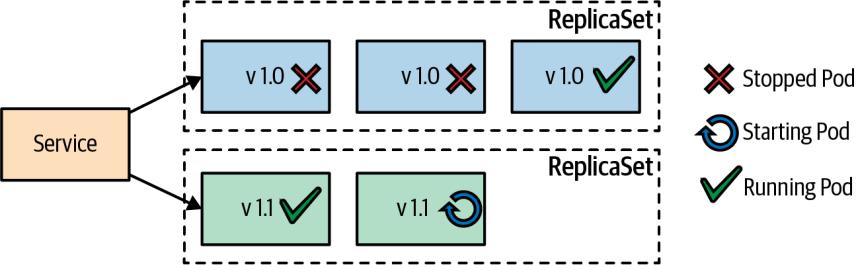
[](#bookmark214) Readiness probes that are very important for a rolling deployment to ensure zero downtime—don’t forget them (see [Chapter 4, “Health Probe”](#bookmark34)).

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

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

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

[Figure 3-1](#bookmark217) 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](https://oreil.ly/xSsID) in our repository, which demonstrates the usage of these commands and shows you how to monitor or roll back an upgrade with kubectl rollout.

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

• Deployment is a Kubernetes resource object whose status is entirely managed by Kubernetes internally. The whole update process is performed on the server side without client interaction.

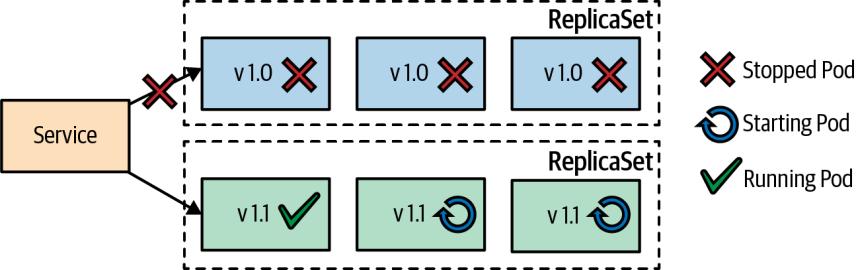
• The declarative nature of Deployment specifies how the deployed state should look rather than the steps necessary to get there.

• The Deployment definition is an executable object and more than just documen ‐ tation. It can be tried and tested on multiple environments before reaching production.

• The update process is also wholly recorded and versioned with options to pause, continue, and roll back to previous versions.

**Fixed** **Deployment**

A RollingUpdate strategy is useful for ensuring zero downtime during the update process. However, the side effect of this approach is that during the update process, two versions of the container are running at the same time. That may cause issues for the service consumers, especially when the update process has introduced backward- incompatible changes in the service APIs and the client is not capable of dealing with them. For this kind of scenario, you can use the Recreate strategy, which is illustrated in [Figure 3-2](#bookmark29).



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

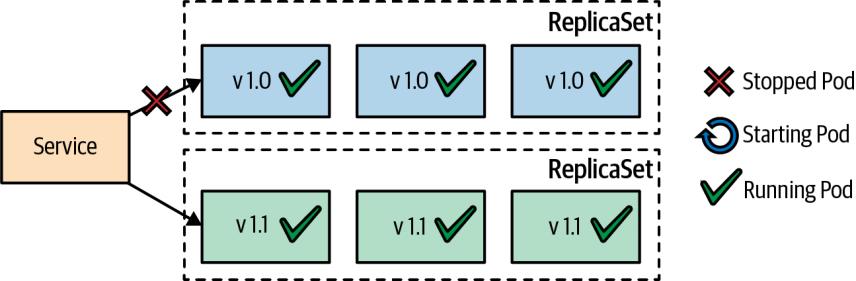
The Recreate strategy has the effect of setting maxUnavailable to the number of declared replicas. This means it first kills all containers from the current version and then starts all new containers simultaneously when the old containers are evicted. The result of this sequence is that downtime occurs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

**Blue-Green** **Release**

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

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

Once we are confident that the new version of the Pods is healthy and ready to handle live requests, we switch the traffic from old Pod replicas to the new replicas. You can do this in Kubernetes by updating the Service selector to match the new containers (labeled with green). As demonstrated in [Figure 3-3](#bookmark221), once the green (v1.1) containers handle all the traffic, the blue (v1.0) containers can be deleted and the resources freed for future Blue-Green deployments.



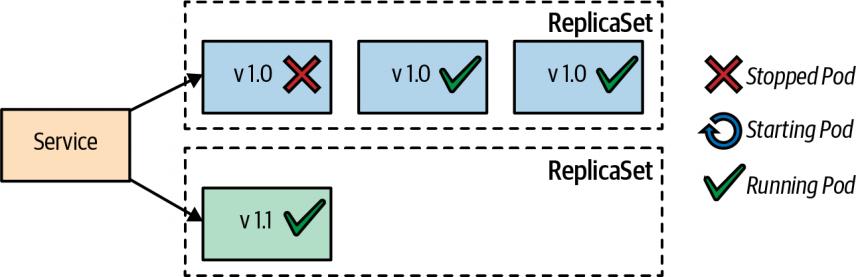
*Figure* *3-3.* *Blue-Green* *release*

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

**Canary** **Release**

*Canary* *release* is a way to softly deploy a new version of an application into produc ‐ tion by replacing only a small subset of old instances with new ones. This technique reduces the risk of introducing a new version into production by letting only some of the consumers reach the updated version. When we’re happy with the new version of our service and how it performed with a small sample of users, we can replace all the old instances with the new version in an additional step after this canary release. [Figure 3-4](#bookmark222) shows a canary release in action.

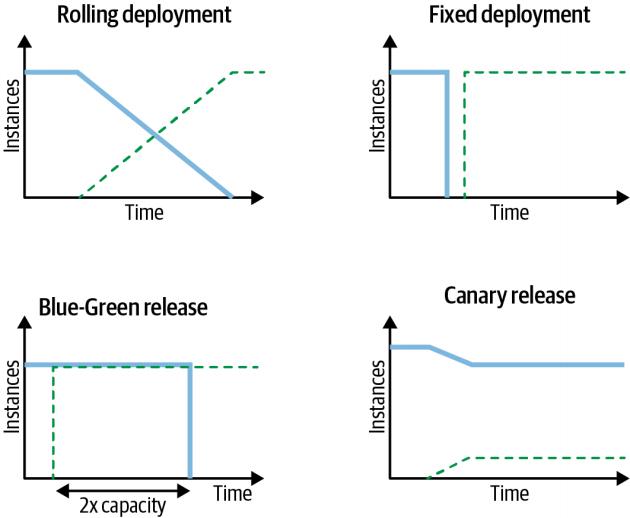
In Kubernetes, this technique can be implemented by creating a new Deployment with a small replica count that can be used as the canary instance. At this stage, the Service should direct some of the consumers to the updated Pod instances. After the canary release and once we are confident that everything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



*Figure* *3-4.* *Canary* *release*

**Discussion**

The Deployment primitive is an example of Kubernetes turning the tedious process of manually updating applications into a declarative activity that can be repeated and automated. The out-of-the-box deployment strategies (rolling and recreate) control the replacement of old containers by new ones, and the advanced release strategies (Blue-Green and canary) control how the new version becomes available to service consumers. The latter two release strategies are based on a human decision for the transition trigger and as a consequence are not fully automated by Kubernetes but require human interaction. [Figure 3-5](#bookmark222) summarizes of the deployment and release strategies, showing instance counts during transitions.



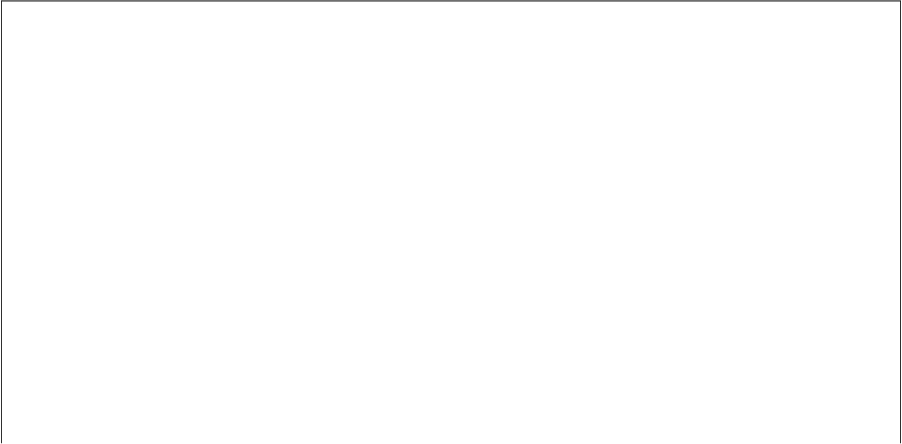
*Figure* *3-5.* *Deployment* *and* *release* *strategies*

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

|  |
| --- |
| **Pre** **and** **Post** **Deployment** **Hooks**  In the past, there has been a proposal for Kubernetes to allow [hooks in the ‐](https://oreil.ly/iGC-2)deploy [ment process](https://oreil.ly/iGC-2). Pre and Post hooks would allow the execution of custom commands before and after Kubernetes has executed a deployment strategy. Such commands could perform additional actions while the deployment is in progress and would additionally be able to abort, retry, or continue a deployment. Those hooks are a good step toward new automated deployment and release strategies. Unfortunately, this effort has been stalled for some years (as of 2023), so it is unclear whether this feature will ever come to Kubernetes. |

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

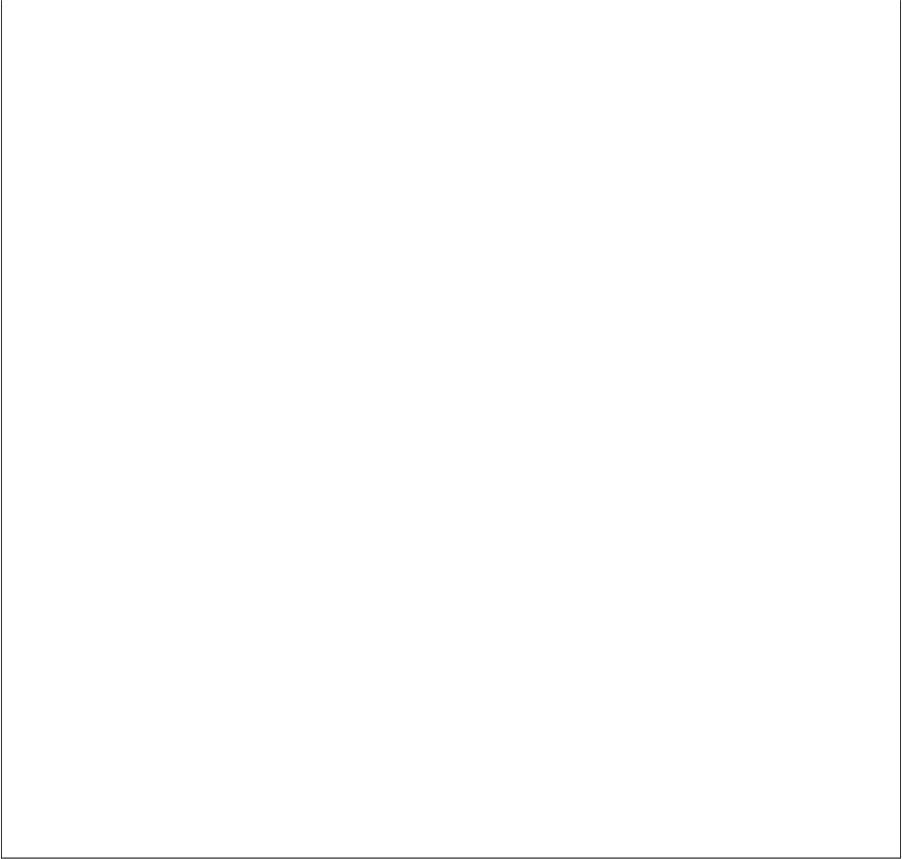
As an alternative, higher-level declarative approaches have emerged on top of Kuber ‐ netes. The most important platforms are described in the sidebar that follows. Those techniques work with operators (see Chapter 28, “Operator”) that take a declarative description of the rollout process and perform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



**Higher-Level** **Deployments**

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

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



*Flagger*

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

*Argo* *Rollouts*

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

*Knative*

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

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

Regardless of the deployment strategy you are using, it is essential for Kubernetes to know when your application Pods are up and running to perform the required sequence of steps to reach the defined target deployment state. The next pattern, *Health* *Probe*, in [Chapter 4](#bookmark34) describes how your application can communicate its health state to Kubernetes.

**More** **Information**

• [Declarative Deployment Example](https://oreil.ly/xSsID)

• [Performing a Rolling Update](https://oreil.ly/paEA0)

• [Deployments](https://oreil.ly/NKEnH)

• [Run a Stateless Application Using a Deployment](https://oreil.ly/wb7D5)

• [Blue-Green Deployment](https://oreil.ly/sbN9T)

• [Canary Release](https://oreil.ly/Z-vFT)

• [Flagger: Deployment Strategies](https://oreil.ly/JGL4C)

• [Argo Rollouts](https://oreil.ly/0lzcD)

• [Knative: Traffic Management](https://oreil.ly/PAwMQ)

**Health** **Probe**

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

**Problem**

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

**Solution**

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

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

**Process** **Health** **Checks**

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

**Liveness** **Probes**

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

*[HTTP](HTTPprobe)**[probe](HTTPprobe)*

Performs an [HTTP GET request to the container IP address and expects a](HTTPGETrequesttothecontainerIPaddressandexpectsa) successful [HTTP response code between 200 and 399](HTTPresponsecodebetween200and399).

*TCP* *Socket* *probe*

Assumes a successful TCP connection.

*Exec* *probe*

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

*gRPC* *probe*

Leverages gRPC’s intrinsic support for health checks.

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

initialDelaySeconds

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

periodSeconds

The interval in seconds between liveness probe checks.

timeoutSeconds

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

failureThreshold

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

An example [HTTP-based liveness probe is shown in](HTTP-basedlivenessprobeisshownin) [Example 4-1](#bookmark227). *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](httpGet:)**[:](httpGet:) [o](#bookmark229)

**path** : /actuator/health

**port** : 8080

**initialDelaySeconds** : 30 [](#bookmark231)

[](#bookmark228) [HTTP probe to a health-check endpoint](HTTPprobetoahealth-checkendpoint).

[](#bookmark230) Wait 30 seconds before doing the first liveness check to give the application some time to warm up.

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

**Readiness** **Probes**

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

For this kind of scenario, Kubernetes has *readiness* *probes*. The methods (<HTTP>, TCP, Exec, gRPC) and timing options for performing readiness checks are the same as for liveness checks, but the corrective action is different. Rather than restarting the con ‐ tainer, 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](#bookmark233) 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** :

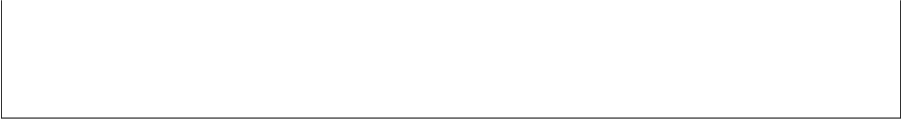
**exec** : [](#bookmark235)

**command** : [ "stat", "/var/run/random-generator-ready" ]

[](#bookmark234) 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 appli ‐ cation is ready to do its job and when it should be left alone. While process health checks and liveness checks are intended to recover from the failure by restarting the container, the readiness check buys time for your application and expects it to recover by itself. Keep in mind that Kubernetes tries to prevent your container from receiving new requests (when it is shutting down, for example), regardless of whether the readiness check still passes after having received a SIGTERM signal.

|  |
| --- |
| **Custom** **Pod** **Readiness** **Gates**  Readiness probes work on a per-container level, and a Pod is considered ready to serve requests when all containers pass their readiness probes. In some situations, this is not good enough—for example, when an external load balancer like the AWS Load ‐ Balancer needs to be reconfigured and ready too. In this case, the readinessGates field of a Pod’s specification can be used to specify extra conditions that need to be met for the Pod to become ready. [Example 4-3](#bookmark237) shows a readiness gate that will introduce an additional condition, k8spatterns.io/load-balancer-ready, to the Pod’s status sections.  *Example* *4-3.* *Readiness* *gate* *for* *indicating* *the* *status* *of* *an* *external* *load* *balancer*  **apiVersion** : v1  **kind** : Pod  ...  **spec** :  **readinessGates** :  - **conditionType** : "k8spatterns.io/load-balancer-ready"  ...  **status** :  **conditions** :  - **type** : "k8spatterns.io/load-balancer-ready" [o](#bookmark239)  **status** : "False"  ...  - **type** : Ready  **status** : "False"  ...  New condition introduced by Kubernetes and set to False by default. It needs to be switched to True externally, e.g., by a controller, as desribed in Chapter 27, “Controller”, when the load balancer is ready to serve.  The Pod is “ready” when all containers’ readiness probes are passing and the readiness gates’ conditions are True; otherwise, as here, the Pod is marked as nonready. |



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

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

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

**Startup** **Probes**

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

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

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

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

[Example 4-4](#bookmark242) is a typical example of a Jakarta EE application server that takes a long time to start.

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

**apiVersion** : v1

**kind** : Pod

**metadata** :

**name** : pod-with-startup-check

**spec** :

**containers** :

- **image** : quay.io/wildfly/wildfly [0](#bookmark245)

**name** : wildfly

**startupProbe** :

**exec** :

**command** : [ "stat", "/opt/jboss/wildfly/standalone/tmp/startup-marker" ] [](#bookmark247)

**initialDelaySeconds** : 60 [](#bookmark249)

**periodSeconds** : 60

**failureThreshold** : 15

**livenessProbe** :

**[httpGet](httpGet:)**[:](httpGet:)

**path** : /health

**port** : 9990

**periodSeconds** : 10 [](#bookmark251)

**failureThreshold** : 3

[](#bookmark244) JBoss WildFly Jakarta EE server that will take its time to start.

[](#bookmark246) Marker file that is created by WildFly after a successful startup.

[](#bookmark248) Timing parameters that specify that the container should be restarted when it has not been passing the startup probe after 15 minutes (60-second pause until the first check, then maximal 15 checks with 60-second intervals).

[](#bookmark250) Timing parameters for the liveness probes are much smaller, resulting in a restart if subsequent liveness probes fail within 20 seconds (three retries with 10-second pauses between each).

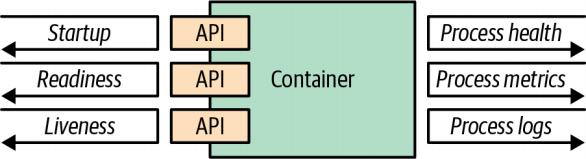
The liveness, readiness, and startup probes are fundamental building blocks of the automation of cloud native applications. Application frameworks such as Quarkus SmallRye Health, Spring Boot Actuator, WildFly Swarm health check, Apache Karaf health check, or the MicroProfile spec for Java provide implementations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

**Discussion**

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

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

Apart from logging to standard streams, it is also a good practice to log the reason for exiting a container to */dev/termination-log*. This location is the place where the con ‐ tainer can state its last will before being permanently vanished.[1](#bookmark253) [Figure 4-1](#bookmark254) 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 treat ‐ ing them like opaque systems. However, any container that is aiming to become a cloud native citizen must provide APIs for the runtime environment to observe the container health and act accordingly. This support is a fundamental prerequisite for automation of the container updates and lifecycle in a unified way, which in turn improves the system’s resilience and user experience. In practical terms, that means, as a very minimum, your containerized application must provide APIs for the different kinds of health checks (liveness and readiness).

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



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

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

The next pattern, *Managed* *Lifecycle*, is also about communication between applica ‐ tions 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](https://oreil.ly/moMx7)

• [Configure Liveness, Readiness, and Startup Probes](https://oreil.ly/h862g)

• [Kubernetes Best Practices: Setting Up Health Checks with Readiness and Liven‐](https://oreil.ly/q0wKy) [ess Probes](https://oreil.ly/q0wKy)

• [Graceful Shutdown with Node.js and Kubernetes](https://oreil.ly/kEik7)

• [Kubernetes Startup Probe—Practical Guide](https://oreil.ly/MHbup)

• [Improving Application Availability with Pod Readiness Gates](https://oreil.ly/h_W1G)

• [Customizing the Termination Message](https://oreil.ly/O2sA2)

• [SmallRye Health](https://oreil.ly/lhetJ)

• [Spring Boot Actuator: Production-Ready Features](https://oreil.ly/7kYX6)

• [Advanced Health Check Patterns in Kubernetes](https://oreil.ly/aKEGe)

**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**

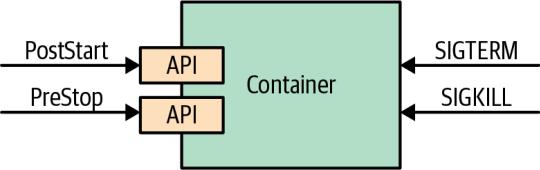
In [Chapter 4, “Health Probe”](#bookmark34), we explained why containers have to provide APIs for the different health checks. Health-check APIs are read-only endpoints the platform is continually probing to get application insight. It is a mechanism for the platform to extract information from the application.

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

**Solution**

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

interactions and lifecycle management capabilities. Some applications need help to warm up, and some applications need a gentle and clean shutdown procedure. For this and other use cases, some events, as shown in [Figure 5-1](#bookmark256), are emitted by the platform that the container can listen to and react to if desired.



*Figure* *5-1.* *Managed* *container* *lifecycle*

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

**SIGTERM** **Signal**

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

**SIGKILL** **Signal**

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

**PostStart** **Hook**

Using only process signals for managing lifecycles is somewhat limited. That is why additional lifecycle hooks such as postStart and preStop are provided by Kubernetes. A Pod manifest containing a postStart hook looks like the one in [Example 5-1](#bookmark257).

*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** : [0](#bookmark259)

- sh

- -c

- sleep 30 && echo "Wake up!" > /tmp/postStart\_done

[o](#bookmark258) The postStart command waits 30 seconds. sleep is just a simulation for any lengthy startup code that might run at this point. Also, it uses a trigger file to sync with the main application, which starts in parallel.

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

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

The postStart and preStop hook invocation mechanisms are similar to the health probes described in [Chapter 4, “Health Probe”](#bookmark34), 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](HTTPGETrequestagainstaportopenedbyonePodcontainerYouhavetobeverycarefulwhatcriticallogicyouexecuteinthepostStarthookastherearenoguaranteesforitsexecution.Sincethehookisrunninginparallelwiththecontainerprocess)

[You have to be very careful what critical logic you execute in the postStart hook](HTTPGETrequestagainstaportopenedbyonePodcontainerYouhavetobeverycarefulwhatcriticallogicyouexecuteinthepostStarthookastherearenoguaranteesforitsexecution.Sincethehookisrunninginparallelwiththecontainerprocess) [as there are no guarantees for its execution. Since the hook is running in parallel](HTTPGETrequestagainstaportopenedbyonePodcontainerYouhavetobeverycarefulwhatcriticallogicyouexecuteinthepostStarthookastherearenoguaranteesforitsexecution.Sincethehookisrunninginparallelwiththecontainerprocess) [with the container process](HTTPGETrequestagainstaportopenedbyonePodcontainerYouhavetobeverycarefulwhatcriticallogicyouexecuteinthepostStarthookastherearenoguaranteesforitsexecution.Sincethehookisrunninginparallelwiththecontainerprocess), 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](#bookmark260) 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](httpGet:)**[:](httpGet:) [o](#bookmark262)

**path** : /shutdown

**port** : 8080

[](#bookmark261) Call out to a /shutdown endpoint running within the application.

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

**Other** **Lifecycle** **Controls**

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

We describe the *Init* *Container* pattern in Chapter 15 in depth, but here we describe it briefly to compare it with lifecycle hooks. Unlike regular application containers, init containers run sequentially, run until completion, and run before any of the application containers in a Pod start up. These guarantees allow you to use init containers for Pod-level initialization tasks. Both lifecycle hooks and init containers operate at a different granularity (at the container level and Pod level, respectively) and can be used interchangeably in some instances, or complement one another in other cases. [Table 5-1](#bookmark263) summarizes the main differences between the two.

*Table* *5-1.* *Lifecycle* *hooks* *and* *init* *containers*

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

If even more control is required to manage the lifecycle of your application contain ‐ ers, there is an advanced technique for rewriting the container entrypoints, some ‐ times also referred to as the *[Commandlet](https://oreil.ly/CVZX6)* [pattern](https://oreil.ly/CVZX6). This pattern is especially useful when the main containers within a Pod have to be started in a certain order and need an extra level of control. Kubernetes-based pipeline platforms like Tekton and Argo CD require the sequential execution of containers that share data and support the inclusion of additional sidecar containers running in parallel (we talk more about sidecars in Chapter 16, “Sidecar”).

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

This concept is best explained by an example. [Example 5-3](#bookmark264) shows a typical Pod declaration that starts a single container with the given arguments.

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

**apiVersion** : v1

**kind** : Pod

**metadata** :

**name** : simple-random-generator

**spec** :

**containers** :

- **image** : k8spatterns/random-generator:1.0

**name** : random-generator

**command** :

- "random-generator-runner" [0](#bookmark266)

**args** : [](#bookmark268)

- "--seed"

- "42"

[](#bookmark265) The command executed when the container starts.

[](#bookmark267) Additional arguments provided to the entrypoint command.

The trick is now to wrap the given command random-generator-runner with a generic supervisor program that takes care of lifecycle aspects, like reacting on SIGTERM or other external signals. [Example 5-4](#bookmark269) demonstrates a Pod declaration that includes an init container for installing a supervisor, which is then started to monitor the main application.

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

**apiVersion** : v1

**kind** : Pod

**metadata** :

**name** : wrapped-random-generator

**spec** :

**volumes** :

- **name** : wrapper [o](#bookmark271)

**emptyDir** : { }

**initContainers** :

- **name** : copy-supervisor [](#bookmark273)

**image** : k8spatterns/supervisor

**volumeMounts** :

- **mountPath** : /var/run/wrapper

**name** : wrapper

**command** : [ cp ]

**args** : [ supervisor, /var/run/wrapper/supervisor ]

**containers** :

- **image** : k8spatterns/random-generator:1.0

**name** : random-generator

**volumeMounts** :

- **mountPath** : /var/run/wrapper

**name** : wrapper

[](#bookmark276)**command** :

- "/var/run/wrapper/supervisor" **args** :

- "random-generator-runner"

- "--seed"

- "42"

[](#bookmark270) A fresh emptyDir volume is created to share the supervisor daemon.

[](#bookmark272) Init container used for copying the supervisor daemon to the application containers.

[](#bookmark274) The original command randomGenerator as defined in [Example 5-3](#bookmark264) is replaced with supervisor daemon from the shared volume.

[](#bookmark275) The original command specification becomes the arguments for the supervisor commands.

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

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

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

**Discussion**

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

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

**More** **Information**

• [Managed Lifecycle Example](https://oreil.ly/2T2jc)

• [Container Lifecycle Hooks](https://oreil.ly/xzeMi)

• [Attach Handlers to Container Lifecycle Events](https://oreil.ly/NTi1h)

• [Kubernetes Best Practices: Terminating with Grace](https://oreil.ly/j-5yl)

• [Graceful Shutdown of Pods with Kubernetes](https://oreil.ly/TgjCp)

• [Argo and Tekton: Pushing the Boundaries of the Possible on Kubernetes](https://oreil.ly/CVZX6)

• [Russian Doll: Extending Containers with Nested Processes](https://oreil.ly/iBhoQ)

**Automated** **Placement**

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

**Problem**

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

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

**Solution**

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

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

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

**Available** **Node** **Resources**

First of all, the Kubernetes cluster needs to have nodes with enough resource capacity to run new Pods. Every node has capacity available for running Pods, and the schedu ‐ ler ensures that the sum of the container resources requested for a Pod is less than the available allocatable node capacity. Considering a node dedicated only to Kubernetes, its capacity is calculated using the following formula in [Example 6-1](#bookmark278).

*Example* *6-1.* *Node* *capacity*

*Alloca* *table* [capacity for application pods] =

*Node* *Capacity* [available capacity on a node]

- *Kube-Reserved* [Kubernetes daemons like kubelet, container runtime]

- *System-Reserved* [Operating System daemons like sshd, udev]

- *Eviction* *Thresholds* [Reserved memory to prevent system OOMs]

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

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

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

**Container** **Resource** **Demands**

Another important requirement for an efficient Pod placement is to define the con ‐ tainers’ runtime dependencies and resource demands. We covered that in more detail in [Chapter 2, “Predictable Demands”](#bookmark14). It boils down to having containers that declare their resource profiles (with request and limit) and environment dependencies such as storage or ports. Only then are Pods optimally assigned to nodes and can run without affecting one another and facing resource starvation during peak usage.

**Scheduler** **Configurations**

The next piece of the puzzle is having the right filtering or priority configurations for your cluster needs. The scheduler has a default set of predicate and priority policies configured that is good enough for most use cases. In Kubernetes versions before v1.23, a scheduling policy can be used to configure the predicates and priorities of a scheduler. Newer versions of Kubernetes moved to scheduling profiles to achieve the same effect. This new approach exposes the different steps of the scheduling process as an extension point and allows you to configure plugins that override the default implementations of the steps. [Example 6-2](#bookmark279) demonstrates how to override the PodTopologySpread plugin from the score step with custom plugins.

*Example* *6-2.* *A* *scheduler* *configuration*

**apiVersion** : kubescheduler.config.k8s.io/v1

**kind** : KubeSchedulerConfiguration

**profiles** :

- **plugins** :

**score** : [](#bookmark281)

**disabled** :

- **name** : PodTopologySpread [](#bookmark283)

**enabled** :

- **name** : MyCustomPlugin [](#bookmark285)

**weight** : 2

[](#bookmark280) The plugins in this phase provide a score to each node that has passed the filtering phase.

[](#bookmark282) This plugin implements topology spread constraints that we will see later in the chapter.

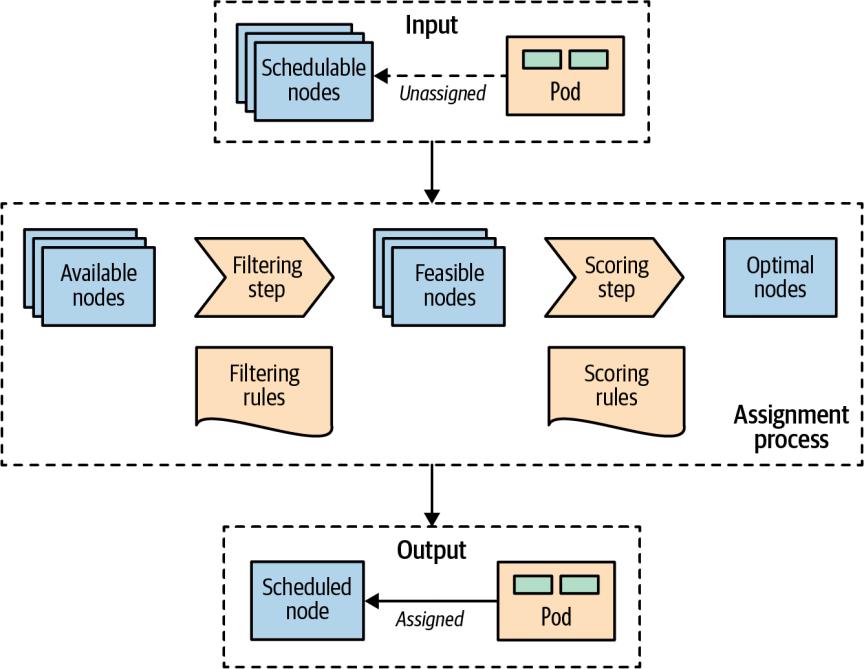
[](#bookmark284) The disabled plugin in the previous step is replaced by a new one.

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

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

**Scheduling** **Process**

Pods get assigned to nodes with certain capacities based on placement policies. For completeness, [Figure 6-1](#bookmark59) visualizes at a high level how these elements get together and the main steps a Pod goes through when being scheduled.



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

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

In most cases, it is better to let the scheduler do the Pod-to-node assignment and not micromanage the placement logic. However, on some occasions, you may want to force the assignment of a Pod to a specific node or group of nodes. This assignment can be done using a node selector. The .spec.nodeSelector Pod field specifies a map of key-value pairs that must be present as labels on the node for the node to be eligible to run the Pod. For example, let’s say you want to force a Pod to run on a specific node where you have SSD storage or GPU acceleration hardware. With the Pod definition in [Example 6-3](#bookmark286) 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 [o](#bookmark288)

[](#bookmark287) Set of node labels a node must match to be considered the node of this Pod.

In addition to specifying custom labels to your nodes, you can use some of the default labels that are present on every node. Every node has a unique kubernetes.io/host name 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 place ‐ ment too.

**Node** **Affinity**

Kubernetes supports many more flexible ways to configure the scheduling processes. One such feature is *node* *affinity*, which is a more expressive way of the node selector approach described previously that allows specifying rules as either required or preferred. *Required* *rules* must be met for a Pod to be scheduled to a node, whereas preferred rules only imply preference by increasing the weight for the matching nodes without making them mandatory. In addition, the node affinity feature greatly expands the types of constraints you can express by making the language more expressive with operators such as In, NotIn, Exists, DoesNotExist, Gt, or Lt. [Exam‐](#bookmark290)  [6-4](#bookmark290)ple 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** : [o](#bookmark292)

**nodeSelectorTerms** :

- **matchExpressions** : [](#bookmark294)

- **key** : numberCores

**operator** : Gt

**values** : [ "3" ]

**preferredDuringSchedulingIgnoredDuringExecution** : [](#bookmark296)

- **weight** : 1

**preference** :

**matchFields** :

- **key** : metadata.name

**operator** : NotIn

**values** : [ "control-plane-node" ]

**containers** :

- **image** : k8spatterns/random-generator:1.0

**name** : random-generator

[](#bookmark291) 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.

[](#bookmark293) Match on labels. In this example, all nodes are matched that have a label number Cores with a value greater than 3.

[](#bookmark295) Soft requirements, which is a list of selectors with weights. For every node, the sum of all weights for matching selectors is calculated, and the highest-valued node is chosen, as long as it matches the hard requirement.

**Pod** **Affinity** **and** **Anti-Affinity**

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

Node affinity works at node granularity, but Pod affinity is not limited to nodes and can express rules at various topology levels based on the Pods already running on a node. Using the topologyKey field, and the matching labels, it is possible to enforce more fine-grained rules, which combine rules on domains like node, rack, cloud provider zone, and region, as demonstrated in [Example 6-5](#bookmark298).

*Example* *6-5.* *Pod* *with* *Pod* *affinity*

**apiVersion** : v1

**kind** : Pod

**metadata** :

**name** : random-generator

**spec** :

**affinity** :

**podAffinity** :

**requiredDuringSchedulingIgnoredDuringExecution** : [0](#bookmark300)

- **labelSelector** : [](#bookmark301)

**matchLabels** :

**confidential** : high

**topologyKey** : security-zone [](#bookmark303)

**podAntiAffinity** : [](#bookmark305)

**preferredDuringSchedulingIgnoredDuringExecution** : [](#bookmark307)

- **weight** : 100

**podAffinityTerm** :

**labelSelector** :

**matchLabels** :

**confidential** : none

**topologyKey** : kubernetes.io/hostname

**containers** :

- **image** : k8spatterns/random-generator:1.0

**name** : random-generator

[](#bookmark299) Required rules for the Pod placement concerning other Pods running on the target node.

[](#bookmark299) Label selector to find the Pods to be colocated with.

[](#bookmark302) The nodes on which Pods with labels confidential=high are running are sup ‐ posed to carry a security-zone label. The Pod defined here is scheduled to a node with the same label and value.

[](#bookmark304) Anti-affinity rules to find nodes where a Pod would *not* be placed.

[](#bookmark306) Rule describing that the Pod should not (but could) be placed on any node where a Pod with the label confidential=none is running.

Similar to node affinity, there are hard and soft requirements for Pod affinity and anti-affinity, called requiredDuringSchedulingIgnoredDuringExecution and preferredDuringSchedulingIgnoredDuringExecution, respectively. Again, as with node affinity, the IgnoredDuringExecution suffix is in the field name, which exists for future extensibility reasons. At the moment, if the labels on the node change and affinity rules are no longer valid, the Pods continue running,[1](#bookmark309) but in the future, runtime changes may also be taken into account.

**Topology** **Spread** **Constraints**

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

Let’s look at an example to understand how topology spread constraints can help. Let’s suppose we have an application with two replicas and a two-node cluster. To avoid downtime and a single point of failure, we can use Pod anti-affinity rules to prevent the coexistence of the Pods on the same node and spread them into both nodes. While this setup makes sense, it will prevent you from performing rolling upgrades because the third replacement Pod cannot be placed on the existing nodes because of the Pod anti-affinity constraints. We will have to either add another node or change the Deployment strategy from rolling to recreate. Topology spread constraints would be a better solution in this situation as they allow you to tolerate some degree of uneven Pod distribution when the cluster is running out of resources. [Example 6-6](#bookmark310) allows the placement of the third rolling deployment Pod on one of the two nodes because it allows imbalances—i.e., a skew of one Pod.



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

*Example* *6-6.* *Pod* *with* *topology* *spread* *constraints*

**apiVersion** : v1

**kind** : Pod

**metadata** :

**name** : random-generator

**labels** :

**app** : bar

**spec** :

**topologySpreadConstraints** : [0](#bookmark313)

- **maxSkew** : 1 [](#bookmark314)

**topologyKey** : topology.kubernetes.io/zone [](#bookmark316)

**whenUnsatisfiable** : DoNotSchedule [](#bookmark318)

**labelSelector** : [](#bookmark320)

**matchLabels** :

**app** : bar

**containers** :

- **image** : k8spatterns/random-generator:1.0

**name** : random-generator

[](#bookmark312) Topology spread constraints are defined in the topologySpreadConstraints field of the Pod spec.

[](#bookmark312) maxSkew defines the maximum degree to which Pods can be unevenly distributed in the topology.

[](#bookmark315) A topology domain is a logical unit of your infrastructure. And a topologyKey is the key of the Node label where identical values are considered to be in the same topology.

[](#bookmark317) The whenUnsatisfiable field defines what action should be taken when maxSkew can’t be satisfied. DoNotSchedule is a hard constraint preventing the scheduling of Pods, whereas ScheduleAnyway is a soft constraint that gives scheduling prior ‐ ity to nodes that reduce cluster imbalance.

[](#bookmark319) labelSelector Pods that match this selector are grouped together and counted when spreading them to satisfy the constraint.

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

**Taints** **and** **Tolerations**

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

A taint is added to a node by using kubectl: kubectl taint nodes control- plane-node node-role.kubernetes.io/control-plane="true":NoSchedule, which has the effect shown in [Example 6-7](#bookmark322). A matching toleration is added to a Pod as shown in [Example 6-8](#bookmark323). Notice that the values for key and effect in the taints section of [Example 6-7](#bookmark322) and the tolerations section in [Example 6-8](#bookmark323) are the same.

*Example* *6-7.* *Tainted* *node*

**apiVersion** : v1

**kind** : Node

**metadata** :

**name** : control-plane-node

**spec** :

**taints** : [o](#bookmark325)

- **effect** : NoSchedule

**key** : node-role.kubernetes.io/control-plane

**value** : "true"

[o](#bookmark324) Mark this node as unschedulable except when a Pod tolerates this taint. *Example* *6-8.* *Pod* *tolerating* *node* *taints*

**apiVersion** : v1

**kind** : Pod

**metadata** :

**name** : random-generator

**spec** :

**containers** :

- **image** : k8spatterns/random-generator:1.0

**name** : random-generator

**tolerations** :

- **key** : node-role.kubernetes.io/control-plane [0](#bookmark327)

**operator** : Exists

**effect** : NoSchedule [](#bookmark329)

[](#bookmark326) Tolerate (i.e., consider for scheduling) nodes, which have a taint with key node- role.kubernetes.io/control-plane. On production clusters, this taint is set on the control plane node to prevent scheduling of Pods on this node. A toleration like this allows this Pod to be installed on the control plane node nevertheless.

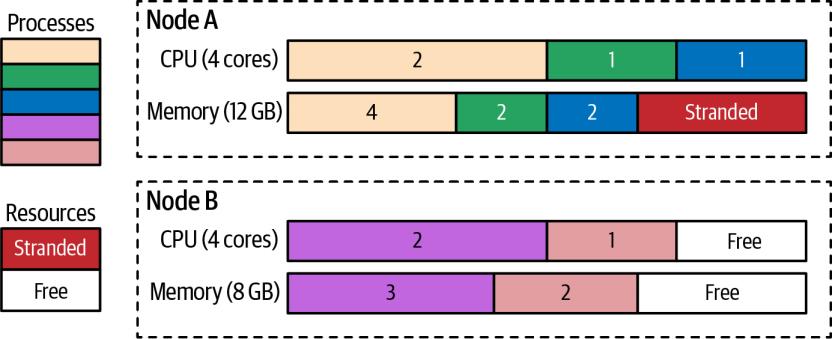
[](#bookmark328) Tolerate only when the taint specifies a NoSchedule effect. This field can be empty here, in which case the toleration applies to every effect.

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](#bookmark331), 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 fragmen ‐ tation and poor utilization of cluster resources. Another potential issue is that the scheduler decisions are based on its cluster view at the point in time when a new Pod is scheduled. If a cluster is dynamic and the resource profile of the nodes changes or new nodes are added, the scheduler will not rectify its previous Pod placements. Apart from changing the node capacity, you may also alter the labels on the nodes that affect placement, but past placements are not rectified.

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

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

• Node- or cluster-critical Pods

• Pods not managed by a ReplicaSet, Deployment, or Job, as these Pods cannot be recreated

• Pods managed by a DaemonSet

• Pods that have local storage

• Pods with PodDisruptionBudget, where eviction would violate its rules

• Pods that have a non-nil DeletionTimestamp field set

• Deschedule Pod itself (achieved by marking itself as a critical Pod)

Of course, all evictions respect Pods’ QoS levels by choosing *Best-Efforts* Pods first, then *Burstable* Pods, and finally *Guaranteed* Pods as candidates for eviction. See [Chapter 2, “Predictable Demands”](#bookmark14), for a detailed explanation of these QoS levels.

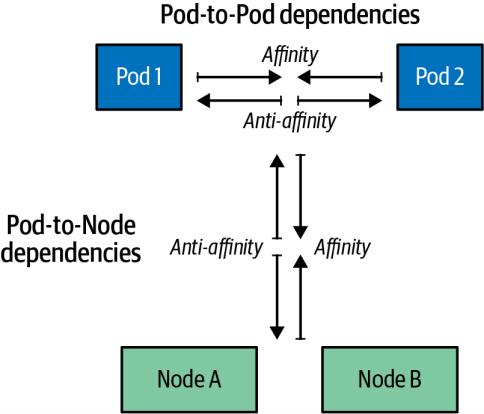
**Discussion**

Placement is the art of assigning Pods to nodes. You want to have as minimal intervention as possible, as the combination of multiple configurations can be hard to predict. In simpler scenarios, scheduling Pods based on resource constraints should be sufficient. If you follow the guidelines from [Chapter 2, “Predictable Demands”](#bookmark14), and declare all the resource needs of a container, the scheduler will do its job and place the Pod on the most feasible node possible.

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

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

[Figure 6-3](#bookmark334) shows one approach to thinking and making sense of the different sched ‐ uling techniques in Kubernetes.



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

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

*nodeName*

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

*nodeSelector*

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

*Node* *affinity*

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

*Taints* *and* *tolerations*

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

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

*Pod* *affinity* *and* *anti-affinity*

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

*Topology* *spread* *constraints*

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

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

*Scheduler* *tuning*

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

*Custom* *scheduler*

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

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

**More** **Information**

• [Automated Placement Example](https://oreil.ly/N-iAz)

• [Assigning Pods to Nodes](https://oreil.ly/QlbMB)

• [Scheduler Configuration](https://oreil.ly/iPbBT)

• [Pod Topology Spread Constraints](https://oreil.ly/qkp60)

• [Configure Multiple Schedulers](https://oreil.ly/appyT)

• [Descheduler for Kubernetes](https://oreil.ly/4lPFX)

• [Disruptions](https://oreil.ly/oNGSR)

• [Guaranteed Scheduling for Critical Add-On Pods](https://oreil.ly/w9tKY)

• [Keep Your Kubernetes Cluster Balanced: The Secret to High Availability](https://oreil.ly/_MODM)

• [Advanced Kubernetes Pod to Node Scheduling](https://oreil.ly/6Tog3)

**PART** **II**

**Behavioral** **Patterns**

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

• [Chapter 7, “Batch Job”](#bookmark68), describes how to isolate an atomic unit of work and run it until completion.

• [Chapter 8, “Periodic Job”](#bookmark73), allows the execution of a unit of work to be triggered by a temporal event.

• [Chapter 9, “Daemon Service”](#bookmark78), allows you to run infrastructure-focused Pods on specific nodes, before application Pods are placed.

• [Chapter 10, “Singleton Service”](#bookmark83), ensures that only one instance of a service is active at a time and still remains highly available.

• [Chapter 11, “Stateless Service”](#bookmark91), describes the building blocks used for managing identical application instances.

• [Chapter 12, “Stateful Service”](#bookmark100), is all about how to create and manage distributed stateful applications with Kubernetes.

• [Chapter 13, “Service Discovery”](#bookmark115), explains how client services can discover and consume the instances of providing services.

• [Chapter 14, “Self Awareness”](#bookmark124), describes mechanisms for introspection and meta ‐ data injection into applications.

**Batch** **Job**

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

**Problem**

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

*Bare* *Pod*

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

*ReplicaSet*

This controller is used for creating and managing the lifecycle of Pods expected to run continuously (e.g., to run a web server container). It maintains a stable set of replica Pods running at any given time and guarantees the availability of a specified number of identical Pods. ReplicaSets are described in detail in [Chapter 11, “Stateless Service”](#bookmark91).

*DaemonSet*

This controller runs a single Pod on every node and is used for managing platform capabilities such as monitoring, log aggregation, storage containers, and others. See [Chapter 9, “Daemon Service”](#bookmark78), for a more detailed discussion.

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

**Solution**

A Kubernetes Job is similar to a ReplicaSet as it creates one or more Pods and ensures they run successfully. However, the difference is that, once the expected number of Pods terminate successfully, the Job is considered complete, and no additional Pods are started. A Job definition looks like [Example 7-1](#bookmark337).

*Example* *7-1.* *A* *Job* *specification*

**apiVersion** : batch/v1

**kind** : Job

**metadata** :

**name** : random-generator

**spec** :

**completions** : 5 [0](#bookmark339)

**parallelism** : 2 [](#bookmark341)

**ttlSecondsAfterFinished** : 300 [](#bookmark343)

**template** :

**metadata** :

**name** : random-generator

**spec** :

**restartPolicy** : OnFailure [](#bookmark345)

**containers** :

- **image** : k8spatterns/random-generator:1.0

**name** : random-generator

**command** : [ "java", "RandomRunner", "/numbers.txt", "10000" ] [](#bookmark338) Job should run five Pods to completion, which all must succeed.

[](#bookmark340) Two Pods can run in parallel.

[](#bookmark342) Keep Pods for five minutes (300 seconds) before garbage-collecting them.

[](#bookmark344) Specifying the restartPolicy is mandatory for a Job. The possible values are OnFailure or Never.

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

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

• A Job is not an ephemeral in-memory task but a persisted one that survives cluster restarts.

• When a Job is completed, it is not deleted but is kept for tracking purposes. The Pods that are created as part of the Job are also not deleted but are available for examination (e.g., to check the container logs). This is also true for bare Pods but only for restartPolicy: OnFailure. You can still remove the Pods of a Job after a certain time by specifying .spec.ttlSecondsAfterFinished.

• A Job may need to be performed multiple times. Using the .spec.completions field, it is possible to specify how many times a Pod should complete successfully before the Job itself is done.

• When a Job has to be completed multiple times, it can also be scaled and exe ‐ cuted by starting multiple Pods at the same time. That can be done by specifying the .spec.parallelism field.

• A Job can be suspended by setting the field .spec.suspend to true. In this case, all active Pods are deleted and restarted if the Job is resumed (i.e., .spec.suspend set to false by the user).

• If the node fails or when the Pod is evicted for some reason while still running, the scheduler places the Pod on a new healthy node and reruns it. Bare Pods would remain in a failed state as existing Pods are never moved to other nodes.

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

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

.spec.completions

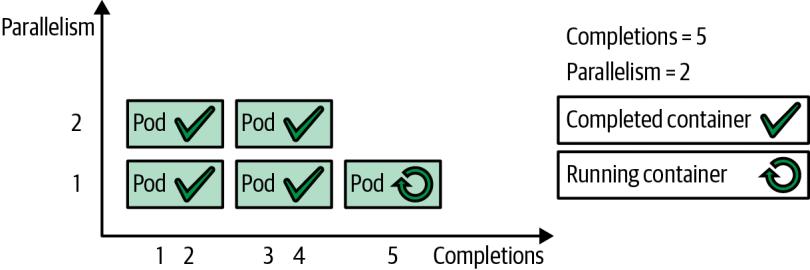
Specifies how many Pods should run to complete a Job.

.spec.parallelism

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

[Figure 7-1](#bookmark347) shows how the Job defined in [Example 7-1](#bookmark337) with a completion count of 5

and a parallelism of 2 is processed.



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

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

*Single* *Pod* *Jobs*

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

*Fixed* *completion* *count* *Jobs*

For a fixed completion count Job, you should set .spec.completions to the number of completions needed. You can set .spec.parallelism, or leave it unset and it will default to 1. Such a Job is considered completed after the .spec.com pletions number of Pods has completed successfully. [Example 7-1](#bookmark337) shows this mode in action and is the best choice when we know the number of work items in advance and the processing cost of a single work item justifies the use of a dedicated Pod.

*Work* *queue* *Jobs*

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

*Indexed* *Jobs*

Similar to *Work* *queue* *Jobs*, you can distribute work items to individual Jobs without needing an external work queue. When using a fixed completion count and setting the completion mode .spec.completionMode to Indexed, every Pod of the Job gets an associated index ranging from 0 to .spec.comple tions - 1. The assigned index is available to the containers through the Pod annotation batch.kubernetes.io/job-completion-index (see [Chapter 14, “Self](#bookmark124) [Awareness”](#bookmark124), to learn how this annotation can be accessed from your code) or directly via the environment variable JOB\_COMPLETION\_INDEX that is set to the index associated with this Pod. With this index at hand, the application can pick the associated work item without any external synchronization. [Example 7-2](#bookmark348) shows a Job that processes the lines of a single file individually by separate Pods. A more realistic example would be an indexed Job used for video processing, where parallel Pods are processing a certain frame range calculated from the index.

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

**apiVersion** : batch/v1

**kind** : Job

**metadata** :

**name** : file-split

**spec** :

**completionMode** : Indexed [o](#bookmark350)

**completions** : 5 [](#bookmark352)

**parallelism** : 5

**template** :

**metadata** :

**name** : file-split

**spec** :

**containers** :

- **image** : alpine

**name** : split

**command** : [](#bookmark354)

- "sh"

- "-c"

- |

start=$(expr $JOB\_COMPLETION\_INDEX \\* 10000) [](#bookmark356)

end=$(expr $JOB\_COMPLETION\_INDEX \\* 10000 + 10000)

awk "NR>=$start && NR<$end" /logs/random.log \ [](#bookmark358)

> /logs/random-$JOB\_COMPLETION\_INDEX.txt

**volumeMounts** :

- **mountPath** : /logs [](#bookmark360)

**name** : log-volume

**restartPolicy** : OnFailure

[](#bookmark349) Enable an indexed completion mode.

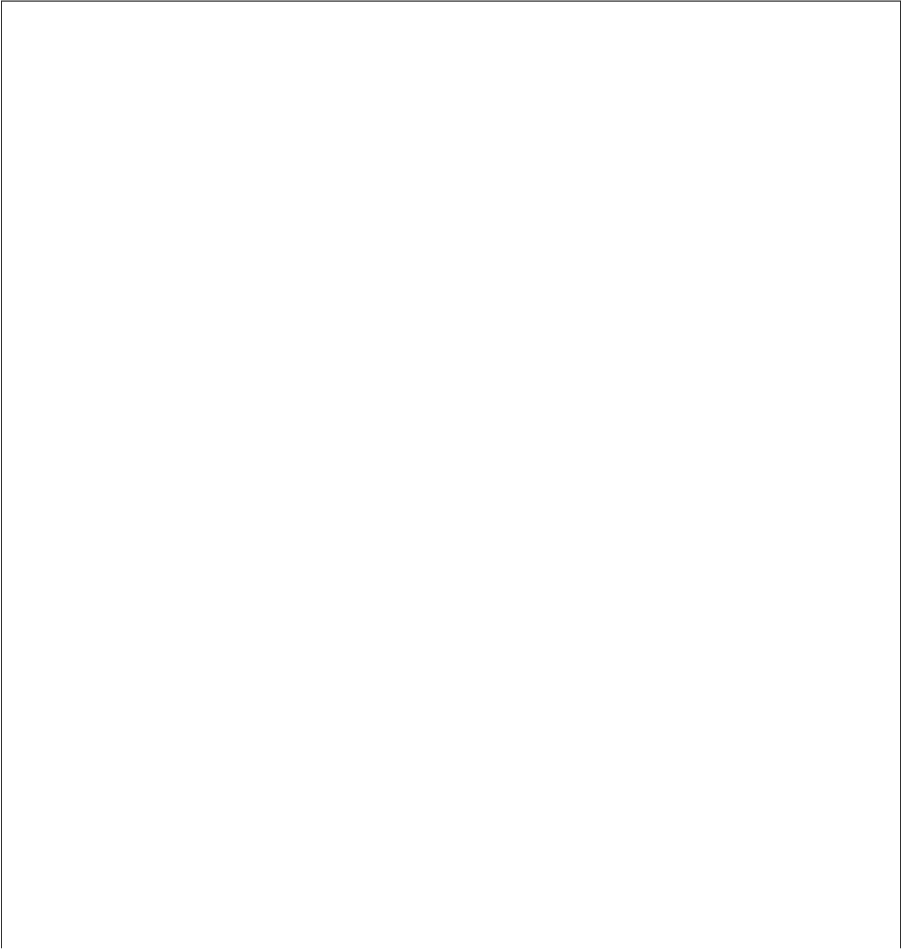
[](#bookmark351) Run five Pods in parallel to completion.

[](#bookmark353) Execute a shell script that prints out a range of lines from a given file */logs/* *random.log*. This file is expected to have 50,000 lines of data.

[](#bookmark355) Calculate start and end line numbers.

[](#bookmark357) Use awk to print out a range of line numbers (NR is the awk-internal line number when iterating over the file).

[](#bookmark359) Mount the input data from an external volume. The volume is not shown here; you can find the full working definition in the [repository](https://oreil.ly/PkVF0)example.

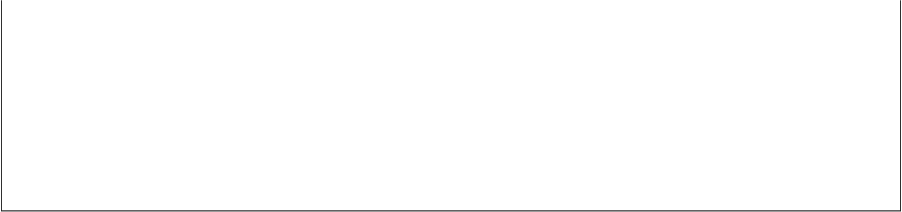


**Partitioning** **the** **Work**

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

• Hardcode the knowledge of the total number of Pods working on a Job into the application code. While this might work for simple examples like [Example 7-2](#bookmark348), it’s generally an imperfect solution as it couples the code in your container to the Kubernetes declaration. That is, if you want to change the number of comple ‐ tions in your Job definition, you would also have to create a new container image for your Job logic with an updated value.

• To access the value of .spec.completions in your application code, you can copy it to an environment variable or pass it as an argument to the container command in the Job’s template specification. But if you plan to change the num ‐ ber of completions, you will need to update two places in the Job declaration.



There has been some [discussion within the Kubernetes community](https://oreil.ly/z7XV7) about whether Kubernetes should provide the value of the .spec.completions field as an environ ‐ ment variable by default. The main concern with this approach is that environment variables cannot be modified at runtime, which could complicate support for resiza ‐ ble Jobs in the future. As a result, a JOB\_COMPLETION\_TOTAL environment variable is not provided by Kubernetes as of version 1.26.

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

**Discussion**

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

*One* *Job* *per* *work* *item*

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

*One* *Job* *for* *all* *work* *items*

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

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

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

**More** **Information**

• [Batch Job Example](https://oreil.ly/PkVF0)

• [Jobs](https://oreil.ly/I2Xum)

• [Parallel Processing Using Expansions](https://oreil.ly/mNmhN)

• [Coarse Parallel Processing Using a Work Queue](https://oreil.ly/W5aqH)

• [Fine Parallel Processing Using a Work Queue](https://oreil.ly/-8FBt)

• [Indexed Job for Parallel Processing with Static Work Assignment](https://oreil.ly/2B2Nn)

• [Spring Batch on Kubernetes: Efficient Batch Processing at Scale](https://oreil.ly/8dLDo)

• [JBeret Introduction](https://oreil.ly/YyYxy)

**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 light‐](HTTPandlight‐weightmessaging.However) [weight messaging. However](HTTPandlight‐weightmessaging.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 rele ‐ vant to business applications requiring specific tasks to be performed periodically. Typical examples here are business-to-business integration through file transfer, application integration through database polling, sending newsletter emails, and cleaning up and archiving old files.

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

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

In the end, a simple service that has to copy a few files once a day may end up requir ‐ ing 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”](#bookmark68), 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](#bookmark365) for a sample definition.

*Example* *8-1.* *A* *CronJob* *resource*

**apiVersion** : batch/v1

**kind** : CronJob

**metadata** :

**name** : random-generator

**spec** :

**schedule** : "\*/3 \* \* \* \*" [0](#bookmark367)

**jobTemplate** :

**spec** :

**template** : [](#bookmark369)

**spec** :

**containers** :

- **image** : k8spatterns/random-generator:1.0

**name** : random-generator

**command** : [ "java", "RandomRunner", "/numbers.txt", "10000" ]

**restartPolicy** : OnFailure

[](#bookmark366) Cron specification for running every three minutes.

[](#bookmark368) Job template that uses the same specification as a regular Job.

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

.spec.schedule

Crontab entry for specifying the Job’s schedule (e.g., 0 \* \* \* \* for running every hour). You can also use shortcuts like @daily or @hourly. Please refer to the [CronJob documentation](https://oreil.ly/Qc3TA) for all available options.

.spec.startingDeadlineSeconds

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

.spec.concurrencyPolicy

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

.spec.suspend

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

.spec.successfulJobsHistoryLimit *and* .spec.failedJobsHistoryLimit

Fields specifying how many completed and failed Jobs should be kept for audit ‐ ing purposes.

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

**Discussion**

As you can see, a CronJob is a pretty simple primitive that adds clustered, cron-like behavior to the existing Job definition. But when it is combined with other primitives such as Pods, container resource isolation, and other Kubernetes features such as those described in [Chapter 6, “Automated Placement”](#bookmark53), or [Chapter 4, “Health Probe”](#bookmark34), 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 per ‐ formed 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](https://oreil.ly/yINcj)

• [CronJob](https://oreil.ly/9096p)

• [Cron](https://oreil.ly/ZPavq)

• [Crontab Specification](https://oreil.ly/Oi3b5)

• [Cron Expression Generator](https://oreil.ly/xYymj)

**Daemon** **Service**

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

**Problem**

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

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

Similarly, Kubernetes also has the concept of a DaemonSet. Considering that Kuber ‐ netes 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 ofthe cluster.

**Solution**

ReplicaSet and its predecessor ReplicationController are control structures respon ‐ sible 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](#bookmark372).

*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** : [](#bookmark374)

**feature** : hw-rng

**containers** :

- **image** : k8spatterns/random-generator:1.0

**name** : random-generator

**command** : [ "java", "RandomRunner", "/numbers.txt", "10000", "30" ]

**volumeMounts** : [](#bookmark376)

- **mountPath** : /host\_dev

**name** : devices

**volumes** :

- **name** : devices

**hostPath** : [](#bookmark378)

**path** : /dev

[](#bookmark373) Use only nodes with the label feature set to value hw-rng.

[](#bookmark375) DaemonSets often mount a portion of a node’s filesystem to perform mainte ‐ nance actions.

[](#bookmark377) hostPath for accessing the node directories directly.

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

• By default, a DaemonSet places one Pod instance on every node. That can be controlled and limited to a subset of nodes by using the nodeSelector or affinity fields.

• A Pod created by a DaemonSet already has nodeName specified. As a result, the DaemonSet doesn’t require the existence of the Kubernetes scheduler to run containers. That also allows you to use a DaemonSet for running and managing the Kubernetes components.

• Pods created by a DaemonSet can run before the scheduler has started, which allows them to run before any other Pod is placed on a node.

• Since the scheduler is not used, the unschedulable field of a node is not respec ‐ ted by the DaemonSet controller.

• Pods created by a DaemonSet can have a RestartPolicy only set to Always or left unspecified, which defaults to Always. This is to ensure that when a liveness probe fails, the container will be killed and always restarted.

• Pods managed by a DaemonSet are supposed to run only on targeted nodes and, as a result, are treated with higher priority by many controllers. For example, the descheduler will avoid evicting such Pods, the cluster autoscaler will manage them separately, etc.

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

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

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

*Service*

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

*DNS*

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

*Node* *IP* *with* hostPort

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

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

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

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

**Discussion**

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

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

**More** **Information**

• [Daemon Service Example](https://oreil.ly/_YRZc)

• [DaemonSet](https://oreil.ly/62c3q)

• [Perform a Rolling Update on a DaemonSet](https://oreil.ly/nTSbi)

• [DaemonSets and Jobs](https://oreil.ly/CnHin)

• [Create Static Pods](https://oreil.ly/yYHft)

**Singleton** **Service**

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

**Problem**

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

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

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

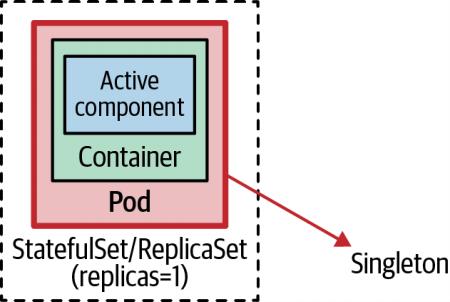
**Solution**

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

**Out-of-Application** **Locking**

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

[Figure 10-1](#bookmark385) shows how to implement out-of-application locking with the help of a StatefulSet or ReplicaSet controller with one replica.



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

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

thanks to the controller performing health checks as described in [Chapter 4, “Health](#bookmark34) [Probe”](#bookmark34), and healing the Pod in case of failures.

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

It’s not entirely true that only one instance is running at all times, especially when things go wrong. Kubernetes primitives such as ReplicaSet favor availability over con ‐ sistency—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 availa ‐ bility and avoiding disruption, as needed for stateless and scalable applications.

Singletons can be resilient and recover, but by definition, they are not highly avail ‐ able. Singletons typically favor consistency over availability. The Kubernetes resource that also favors consistency over availability and provides the desired strict singleton guarantees is the StatefulSet. If ReplicaSets do not provide the desired guarantees for your application, and you have strict singleton requirements, StatefulSets might be the answer. StatefulSets are intended for stateful applications and offer many features, including stronger singleton guarantees, but they come with increased complexity as well. We discuss concerns around singletons and cover StatefulSets in more detail in [Chapter 12, “Stateful Service”](#bookmark100).

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

We cover Kubernetes Services in depth in [Chapter 13, “Service Discovery”](#bookmark115), but let’s discuss briefly the part that applies to singletons here. A regular Service (with type: ClusterIP) creates a virtual IP and performs load balancing among all the Pod instances that its selector matches. However, a singleton Pod managed through

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

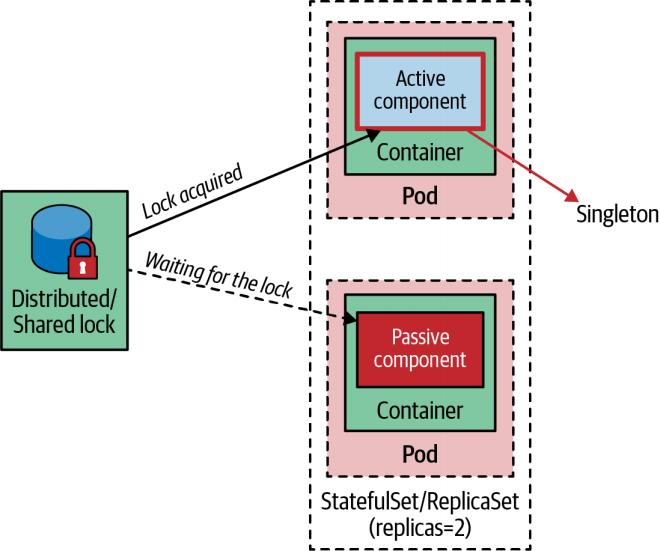
However, such a Service is still useful because a headless Service with selectors creates endpoint records in the API Server and generates DNS A records for the matching Pod(s). With that, a DNS lookup for the Service does not return its virtual IP but instead the IP address(es) of the backing Pod(s).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

To sum up, for nonstrict singletons with at least one instance requirement, defining a ReplicaSet with one replica would suffice. This configuration favors availability and ensures there is at least one available instance, and possibly more in some corner cases. For a strict singleton with an At-Most-One requirement and better performant service discovery, a StatefulSet and a headless Service would be preferred. Using StatefulSet will favor consistency and ensure there is an At-Most-One instance and occasionally none in some corner cases. You can find a complete example of this in [Chapter 12, “Stateful Service”](#bookmark100), 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](#bookmark388). 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 availa ‐ bility 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 environ ‐ ment, first we need a distributed lock implementation such as the one provided by Apache ZooKeeper, HashiCorp’s Consul, Redis, or etcd.

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

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

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

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

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

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

**Pod** **Disruption** **Budget**

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

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

The PodDisruptionBudget in [Example 10-1](#bookmark392) applies to Pods that match its selector and ensures two Pods must be available all the time.

*Example* *10-1.* *PodDisruptionBudget*

**apiVersion** : policy/v1

**kind** : PodDisruptionBudget

**metadata** :

**name** : random-generator-pdb

**spec** :

**selector** :

**matchLabels** : [](#bookmark394)

**app** : random-generator

**minAvailable** : 2 [](#bookmark396)

[](#bookmark393) Selector to count available Pods.

[](#bookmark395) 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.maxUna vailable, which specifies the number of Pods from that set that can be unavailable after the eviction. Similar to .spec.minAvailable, it can be either an absolute number or a percentage, but it has a few additional limitations. You can specify only either .spec.minAvailable or .spec.maxUnavailable in a single PodDisrup ‐ tionBudget, and then it can be used only to control the eviction of Pods that have an associated controller such as ReplicaSet or StatefulSet. For Pods not managed by a controller (also referred to as *bare* or *naked* Pods), other limitations around PodDisruptionBudget should be considered.

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

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

**Discussion**

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

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

**More** **Information**

• [Singleton Service Example](https://oreil.ly/aGoPv)

• [Leases](https://oreil.ly/tb9aX)

• [Specifying a Disruption Budget for Your Application](https://oreil.ly/W1ABD)

• [Leader Election in Go Client](https://oreil.ly/NU1aN)

• [Dapr: Distributed Lock Overview](https://oreil.ly/ES8Ve)

• [Creating Clustered Singleton Services on Kubernetes](https://oreil.ly/K8zI1)

• [Akka: Kubernetes Lease](https://oreil.ly/tho5T)

**Stateless** **Service**

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

**Problem**

The microservices architecture style is the dominant choice for implementing new greenfield cloud native applications. Among the driving principles of this architecture are things such as how it addresses a single concern, how it owns its data, how it has a well-encapsulated deployment boundary, and others. Typically, such applications also follow the [twelve-factor app](https://12factor.net)principles, which makes them easy to operate with Kubernetes on dynamic cloud environments.

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

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

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

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

**Solution**

In [Chapter 3, “Declarative Deployment”](#bookmark25), you learned how to use the concept of Deployment to control how an application should be updated to the next version, using the RollingUpdate and Recreate strategies. But this is only the upgrading aspect of Deployment. At a broader level, a Deployment represents an application deployed in the cluster. Kubernetes doesn’t have the notion of an Application or a Container as top-level entities. Instead, an application is typically composed of a collection of Pods managed by a controller such as ReplicaSet, Deployment, or StatefulSet, combined with ConfigMap, Secret, Service, PersistentVolumeClaim, etc. The controller that is used for managing stateless Pods is ReplicaSet, but that is a lower-level internal control structure used by a Deployment. Deployment is the rec ‐ ommended user-facing abstraction for creating and updating stateless applications, which creates and manages the ReplicaSets behind the scene. A ReplicaSet should be used when the update strategies provided by Deployment are not suitable, or a custom mechanism is required, or no control over the update process is needed at all.

**Instances**

The primary purpose of a ReplicaSet is to ensure a specified number of identical Pod replicas running at any given time. The main sections of a ReplicaSet definition include the number of replicas indicating how many Pods it should maintain, a selector that specifies how to identify the Pods it manages, and a Pod template for creating new Pod replicas. Then, a ReplicaSet creates and deletes Pods as needed to maintain the desired replica count using the given Pod template, as demonstrated in [Example 11-1](#bookmark399).

*Example* *11-1.* *ReplicaSet* *definition* *for* *a* *stateless* *Pod*

**apiVersion** : apps/v1

**kind** : ReplicaSet

**metadata** :

**name** : rg

**labels** :

**app** : random-generator

**spec** :

**replicas** : 3 [](#bookmark402)

**selector** : [](#bookmark404)

**matchLabels** :

**app** : random-generator

**template** : [](#bookmark406)

**metadata** :

**labels** :

**app** : random-generator

**spec** :

**containers** :

- **name** : random-generator

**image** : k8spatterns/random-generator:1.0

[](#bookmark401) Desired number of Pod replicas to maintain running.

[](#bookmark403) Label selector used to identify the Pods to manage.

[](#bookmark405) Template specifying the data for creating new Pods.

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

Regardless of whether you create a ReplicaSet directly or through a Deployment, the end result will be that the desired number of identical Pod replicas are created and maintained. The added benefit of using Deployment is that we can control how the replicas are upgraded and rolled back, which we described in detail in [Chapter 3,](#bookmark25) [“Declarative Deployment”](#bookmark25). Next, the replicas are scheduled to the available nodes as per the policies we covered in [Chapter 6, “Automated Placement”](#bookmark53). The ReplicaSet’s job is to restart the containers if needed and scale out or in when the number of replicas is increased or decreased, respectively. With this behavior, Deployment and ReplicaSet can automate the lifecycle management of stateless applications.

**Networking**

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

Depending on how the application within the container connects to the other systems to accept requests or poll for messages, for example, you may require a Kubernetes Service. If the application is starting an egress connection to a message broker or database, and that is the only way it exchanges data, then there is no need for a Kubernetes Service. But more often, stateless services are contacted by other services over synchronous request/response-driven protocols such as [HTTP and gRPC. Since](HTTPandgRPC.SincethePodIPaddresschangeswitheveryPodrestart) [the Pod IP address changes with every Pod restart](HTTPandgRPC.SincethePodIPaddresschangeswitheveryPodrestart), it is better to use a permanent IP address based on a Kubernetes Service that service consumers can use. A Kubernetes Service has a fixed IP address that doesn’t change during the lifetime of the Service, and it ensures the client requests are always load-balanced across instances and routed to the healthy and ready-to-accept-requests Pods. We cover different types of Kubernetes Services in [Chapter 13, “Service Discovery”](#bookmark115) . In [Example 11-2](#bookmark408), we use a simple Service to expose the Pods internally within the cluster to other Pods.

*Example* *11-2.* *Exposing* *a* *stateless* *service*

**apiVersion** : v1

**kind** : Service

**metadata** :

**name** : random-generator [0](#bookmark410)

**spec** :

**selector** : [](#bookmark412)

**app** : random-generator

**ports** :

- **port** : 80

**targetPort** : 8080

**protocol** : TCP

[](#bookmark409) Name of the service that can be used to reach the matching Pods.

[](#bookmark411) Selector matching the Pod labels from the ReplicaSet.

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

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

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

**Storage**

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

A PersistentVolume (PV) represents a storage resource abstraction in a Kubernetes cluster that has a lifecycle independent of any Pod lifecycle that is using it. A Pod cannot directly refer to a PV; however, a Pod uses PersistentVolumeClaim (PVC) to request and bind to the PV, which points to the actual durable storage. This indirect connection allows for a separation of concerns and Pod lifecycle decoupling from PV. A cluster administrator can configure storage provisioning and define PVs. The developer creating Pod definitions can use PVC to use the storage. With this indirection, even if the Pod is deleted, the ownership of the PV remains attached to the PVC and continues to exist. [Example 11-3](#bookmark413) shows a storage claim that can be used in a Pod template.

*Example* *11-3.* *A* *claim* *for* *a* *PersistentVolume*

**apiVersion** : v1

**kind** : PersistentVolumeClaim

**metadata** :

**name** : random-generator-log [](#bookmark415)

**spec** :

**storageClassName** : "manual"

**accessModes** :

- ReadWriteOnce [](#bookmark417)

**resources** :

**requests** :

**storage** : 1Gi [](#bookmark419)

[](#bookmark414) Name of the claim that can be referenced from a Pod template.

[](#bookmark416) Indicates that only a single node can mount the volume for reading and writing. [](#bookmark418) Requesting 1 GiB of storage.

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

*ReadWriteOnce*

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

*ReadOnlyMany*

The volume can be mounted to multiple nodes, but it allows read-only opera ‐ tions to all Pods.

*ReadWriteMany*

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

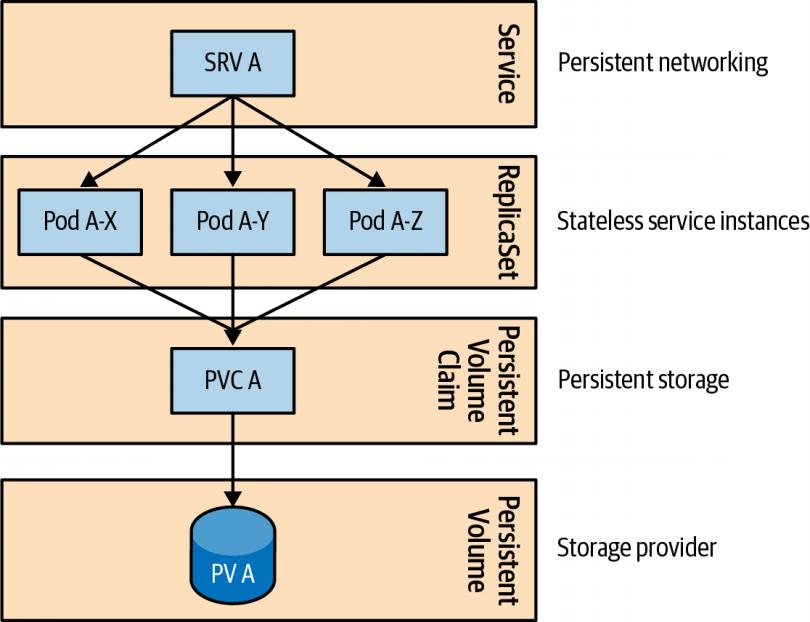
*ReadWriteOncePod*

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

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

**Discussion**

A complex distributed system is usually composed of multiple services, some of which will be stateful and perform some form of distributed coordination, some of which might be short-lived jobs, and some of which might be highly scalable stateless services. Stateless services are composed of identical, swappable, ephemeral, and replaceable instances. They are ideal for handling short-lived requests and can scale up and down rapidly without having any dependencies among the instances. As shown in [Figure 11-1](#bookmark97), Kubernetes offers a number of useful primitives to manage such applications.



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

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

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

**More** **Information**

• [Stateless Service Example](https://oreil.ly/h0Ytj)

• [ReplicaSet](https://oreil.ly/XugMo)

• [Persistent Volumes](https://oreil.ly/HvApe)

• [Storage Classes](https://oreil.ly/qxFrz)

• [Access Modes](https://oreil.ly/iovaa)

**Stateful** **Service**

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

**Problem**

We have seen many Kubernetes primitives for creating distributed applications: containers with health checks and resource limits, Pods with multiple containers, dynamic cluster-wide placements, batch jobs, scheduled jobs, singletons, and more. The common characteristic of these primitives is that they treat the managed appli ‐ cation as a stateless application composed of identical, swappable, and replaceable containers and comply with the [twelve-factor app](https://12factor.net)principles.

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

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

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

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

**Storage**

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

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

Another workaround is to have a separate ReplicaSet (with replicas=1) for every instance of the distributed stateful application. In this scenario, every ReplicaSet would get its PVC and dedicated storage. The downside of this approach is that it is intensive in manual labor: scaling up requires creating a new set of ReplicaSet, PVC, or Service definitions. This approach lacks a single abstraction for managing all instances ofthe 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, manag ‐ ing 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 typi ‐ cally impacts the sequence in which the instances are scaled up and down. However, it can also be used for data distribution or access and in-cluster behavior positioning such as locks, singletons, or leaders.

**Other** **Requirements**

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

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

**Solution**

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

Let’s explore how StatefulSets work and how they address the needs of stateful

applications. [Example 12-1](#bookmark422) is our random-generator service as a StatefulSet.[1](#bookmark423) *Example* *12-1.* *StatefulSet* *definition* *for* *a* *stateful* *application*

**apiVersion** : apps/v1

**kind** : StatefulSet

**metadata** :

**name** : rg [o](#bookmark425)

**spec** :

**serviceName** : random-generator [](#bookmark427)

**replicas** : 2 [](#bookmark429)

**selector** :

**matchLabels** :

**app** : random-generator

**template** :

**metadata** :

**labels** :

**app** : random-generator

**spec** :

**containers** :

- **image** : k8spatterns/random-generator:1.0

**name** : random-generator

**ports** :

- **containerPort** : 8080

**name**: <http>

**volumeMounts** :

- **name** : logs

**mountPath** : /logs

**volumeClaimTemplates** : [](#bookmark431)

- **metadata** :

**name** : logs



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

Random Number Generator (RNG) cluster with several instances of our service as nodes. Of course, that’s not true, but for this example’s sake, it’s a good enough story.

**spec** :

**accessModes** : [ "ReadWriteOnce" ]

**resources** :

**requests** :

**storage** : 10Mi

[](#bookmark424) Name of the StatefulSet is used as prefix for the generated node names.

[](#bookmark426) References the mandatory Service defined in [Example 12-2](#bookmark432).

[](#bookmark428) Two Pod members in the StatefulSet named *rg-0* and *rg-1*.

[](#bookmark430) Template for creating a PVC for each Pod (similar to the Pod’s template).

Rather than going through the definition in [Example 12-1](#bookmark422) 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 volume ClaimTemplates on the fly during Pod creation. This mechanism allows every Pod to get its own dedicated PVC during initial creation as well as during scaling up by changing the replicas count of the StatefulSets.

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

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

**Networking**

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

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

In [Example 12-2](#bookmark432), we define a *headless* Service. In a headless Service, clusterIP is set to None, which means we don’t want a kube-proxy to handle the Service, and we don’t want a cluster IP allocation or load balancing. Then why do we need a Service?

*Example* *12-2.* *Service* *for* *accessing* *StatefulSet*

**apiVersion** : v1

**kind** : Service

**metadata** :

**name** : random-generator

**spec** :

**clusterIP** : None [](#bookmark435)

**selector** :

**app** : random-generator

**ports** :

- **name**: [http](httpport:8080)

**[port](httpport:8080)**[: 8080](httpport:8080)

[](#bookmark434) Declares this Service as headless.

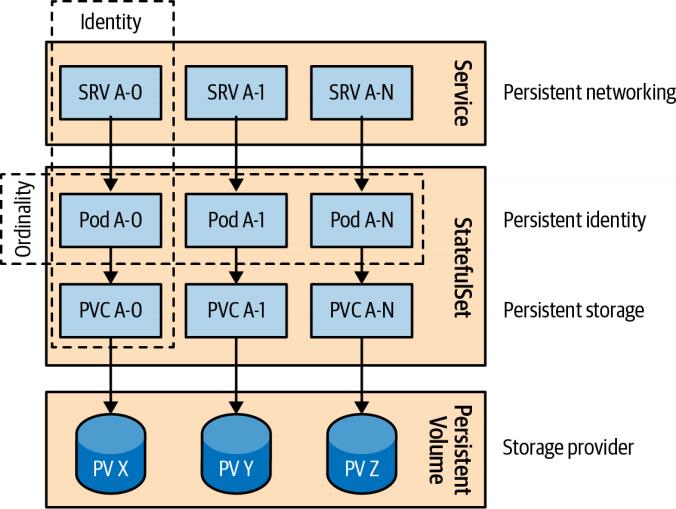
Stateless Pods created through a ReplicaSet are assumed to be identical, and it doesn’t matter on which one a request lands (hence the load balancing with a regular Ser ‐ vice). But stateful Pods differ from one another, and we may need to reach a specific Pod by its coordinates.

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

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

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

As [Figure 12-1](#bookmark436) shows, StatefulSets offer a set of building blocks and guaranteed behavior needed for managing stateful applications in a distributed environment. Your job is to choose and use them in a meaningful way for your stateful use case.



*Figure* *12-1.* *A* *distributed* *stateful* *application* *on* *Kubernetes*

**Identity**

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

**Ordinality**

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

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

If we create a ReplicaSet with multiple replicas, Pods are scheduled and started together without waiting for the first one to start successfully (running and ready status, as described in [Chapter 4, “Health Probe”](#bookmark34)). The order in which Pods are starting and are ready is not guaranteed. It is the same when we scale down a Replica ‐ Set (either by changing the replicas count or deleting it). All Pods belonging to a ReplicaSet start shutting down simultaneously without any ordering and dependency among them. This behavior may be faster to complete but is not preferred for stateful applications, especially if data partitioning and distribution are involved among the instances.

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

**Other** **Features**

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

*Partitioned* *updates*

We described earlier the sequential ordering guarantees when scaling a Stateful ‐ Set. 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 ofthe instances.

By using the default rolling update strategy, you can partition instances by speci ‐ fying a .spec.updateStrategy.rollingUpdate.partition number. The param ‐ eter (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 ofthe cluster by setting the partition back to 0.

*Parallel* *deployments*

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

*At-Most-One* *Guarantee*

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

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

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

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

We discuss other approaches to achieving singletons in more depth in [Chapter 10,](#bookmark83) [“Singleton Service”](#bookmark83).

**Discussion**

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

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

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

**More** **Information**

• [Stateful Service Example](https://oreil.ly/FXeca)

• [StatefulSet Basics](https://oreil.ly/NdHnS)

• [StatefulSets](https://oreil.ly/WyxHN)

• [Example: Deploying Cassandra with a Stateful Set](https://oreil.ly/YECff)

• [Running ZooKeeper, a Distributed System Coordinator](https://oreil.ly/WzQXP)

• [Headless Services](https://oreil.ly/7GPda)

• [Force Delete StatefulSet Pods](https://oreil.ly/ZRTlO)

• [Graceful Scaledown of Stateful Apps in Kubernetes](https://oreil.ly/7Zw-5)

**Service** **Discovery**

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

**Problem**

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

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

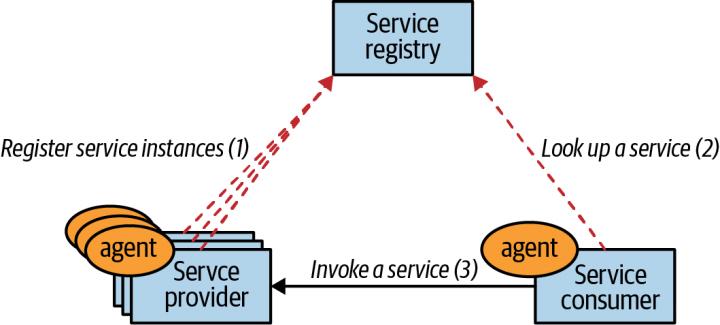
To implement the patterns described in [Chapter 7, “Batch Job”](#bookmark68), or [Chapter 8, “Peri‐](#bookmark73) [odic Job”](#bookmark73), we often use this technique. In addition, long-running Pods in DaemonSets or ReplicaSets sometimes actively connect to other systems over the network. The more common use case for Kubernetes workloads occurs when we have long-running services expecting external stimulus, most commonly in the form of incoming <HTTP> connections from other Pods within the cluster or external systems. In these cases,

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

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

**Solution**

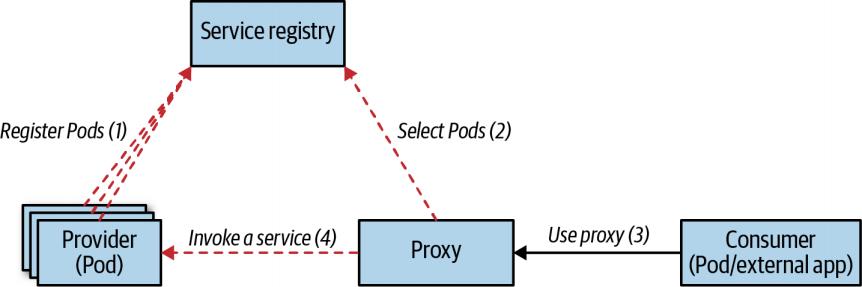
If we look at the “Before Kubernetes Era,” the most common mechanism of service discovery was through client-side discovery. In this architecture, when a service consumer had to call another service that might be scaled to multiple instances, the service consumer would have a discovery agent capable of looking at a registry for service instances and then choosing one to call. Classically, that would be done, for example, either with an embedded agent within the consumer service (such as a ZooKeeper client, Consul client, or Ribbon) or with another colocated process looking up the service in a registry, as shown in [Figure 13-1](#bookmark117).



*Figure* *13-1.* *Client-side* *service* *discovery*

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

In the Kubernetes world, all that happens behind the scenes so that a service con ‐ sumer calls a fixed virtual Service endpoint that can dynamically discover service instances implemented as Pods. [Figure 13-2](#bookmark440) shows how registration and lookup are embraced by Kubernetes.



*Figure* *13-2.* *Server-side* *service* *discovery*

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

**Internal** **Service** **Discovery**

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

This challenge is what the Kubernetes Service resource addresses. It provides a con ‐ stant 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 Ser ‐ vice for a Pod or multiple Pods of a Deployment or ReplicaSet. The command creates a virtual IP address referred to as the clusterIP, and it pulls both Pod selectors and port numbers from the resources to create the Service definition. However, to have full control over the definition, we create the Service manually, as shown in [Example 13-1](#bookmark441).

*Example* *13-1.* *A* *simple* *Service*

**apiVersion** : v1

**kind** : Service

**metadata** :

**name** : random-generator

**spec** :

**selector** : [0](#bookmark444)

**app** : random-generator

**ports** :

- **port** : 80  [](#bookmark446)

**targetPort** : 8080 [](#bookmark448)

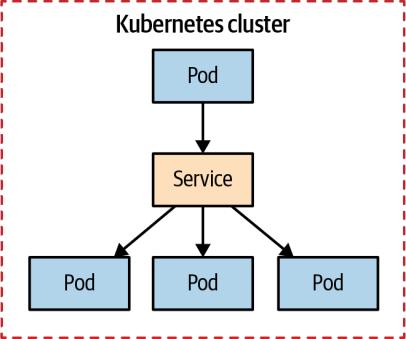
**protocol** : TCP

[](#bookmark443) Selector matching Pod labels.

[](#bookmark445) Port over which this Service can be contacted.

[](#bookmark447) Port on which the Pods are listening.

The definition in this example will create a Service named random-generator (the name is important for discovery later) and type: ClusterIP (which is the default) that accepts TCP connections on port 80 and routes them to port 8080 on all the matching Pods with the selector app: random-generator. It doesn’t matter when or how the Pods are created—any matching Pod becomes a routing target, as illustrated in [Figure 13-3](#bookmark449).



*Figure* *13-3.* *Internal* *service* *discovery*

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

*Discovery* *through* *environment* *variables*

When Kubernetes starts a Pod, its environment variables get populated with the details of all Services that exist up to that moment. For example, our random- generator Service listening on port 80 gets injected into any newly starting Pod, as the environment variables shown in [Example 13-2](#bookmark450) 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 depend ‐ ency on Service creation. Since environment variables cannot be injected into already-running Pods, the Service coordinates are available only for Pods started after the Service is created in Kubernetes. That requires the Service to be defined before starting the Pods that depend on the Service—or if this is not the case, the Pods need to be restarted.

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

RANDOM\_GENERATOR\_SERVICE\_HOST=<10.109.72.32>

RANDOM\_GENERATOR\_SERVICE\_PORT=80

*Discovery* *through* *DNS* *lookup*

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

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

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

*Multiple* *ports*

A single Service definition can support multiple source and target ports. For example, if your Pod supports both [HTTP on port 8080 and HTTPS on port](HTTPonport8080andHTTPSonport8443) [8443](HTTPonport8080andHTTPSonport8443), there is no need to define two Services. A single Service can expose both ports on 80 and 443, for example.

*Session* *affinity*

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

*Readiness* *probes*

In [Chapter 4, “Health Probe”](#bookmark34), you learned how to define a readinessProbe for a container. If a Pod has defined readiness checks, and they are failing, the Pod is removed from the list of Service endpoints to call even if the label selector matches the Pod.

*Virtual* *IP*

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

*Choosing* *ClusterIP*

During Service creation, we can specify an IP to use with the field .spec.clus terIP. 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 clus ‐ ter; 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 discov‐ ery of endpoints that are manually specified.

**Manual** **Service** **Discovery**

When we create a Service with selector, Kubernetes tracks the list of matching and ready-to-serve Pods in the list of endpoint resources. For [Example 13-1](#bookmark441), you can check all endpoints created on behalf of the Service with kubectl get endpoints random-generator. Instead of redirecting connections to Pods within the cluster, we could also redirect connections to external IP addresses and ports. We can do that by omitting the selector definition of a Service and manually creating endpoint resources, as shown in [Example 13-3](#bookmark451).

*Example* *13-3.* *Service* *without* *selector*

**apiVersion** : v1

**kind** : Service

**metadata** :

**name** : external-service

**spec** :

**type** : ClusterIP

**ports** :

- **protocol** : TCP

**port** : 80

Next, in [Example 13-4](#bookmark452), we define an endpoint resource with the same name as the Service and containing the target IPs and ports.

*Example* *13-4.* *Endpoints* *for* *an* *external* *service*

**apiVersion** : v1

**kind** : Endpoints

**metadata** :

**name** : external-service [0](#bookmark454)

**subsets** :

- **addresses** :

- **ip** : <1.1.1.1>

- **ip** : <2.2.2.2>

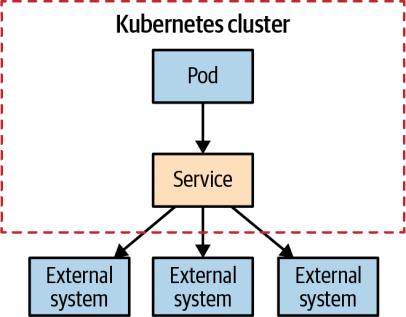
**ports** :

- **port** : 8080

[](#bookmark453) Name must match the Service that accesses these endpoints.

This Service is also accessible only within the cluster and can be consumed in the same way as the previous ones, through environment variables or DNS lookup. The difference is that the list of endpoints is manually maintained and those values usually point to IP addresses outside the cluster, as demonstrated in [Figure 13-4](#bookmark456).

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



*Figure* *13-4.* *Manual* *service* *discovery*

In this category of manual destination configuration, there is one more type of Service, as shown in [Example 13-5](#bookmark457).

*Example* *13-5.* *Service* *with* *an* *external* *destination*

**apiVersion** : v1

**kind** : Service

**metadata** :

**name** : database-service

**spec** :

**type** : ExternalName

**externalName** : my.database.example.com

**ports** :

- **port** : 80

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

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

**Service** **Discovery** **from** **Outside** **the** **Cluster**

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

The first method to create a Service and expose it outside of the cluster is through type: NodePort. The definition in [Example 13-6](#bookmark459) creates a Service as earlier, serving Pods that match the selector app: random-generator, accepting connections on port 80 on the virtual IP address and routing each to port 8080 of the selected Pod. However, in addition to all of that, this definition also reserves port 30036 on all the nodes and forwards incoming connections to the Service. This reservation makes the Service accessible internally through the virtual IP address, as well as externally through a dedicated port on every node.

*Example* *13-6.* *Service* *with* *type* *NodePort*

**apiVersion** : v1

**kind** : Service

**metadata** :

**name** : random-generator

**spec** :

**type** : NodePort [0](#bookmark461)

**selector** :

**app** : random-generator

**ports** :

- **port** : 80

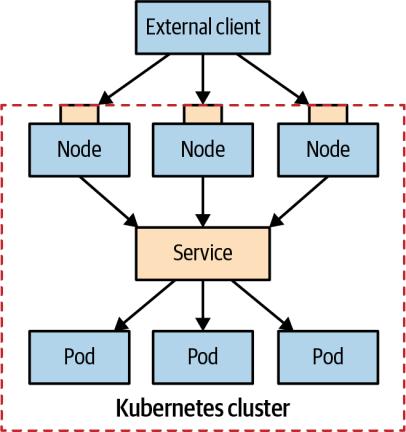
**targetPort** : 8080

**nodePort** : 30036 [](#bookmark463)

**protocol** : TCP

[](#bookmark460) Open port on all nodes.

[](#bookmark462) Specify a fixed port (which needs to be available) or leave this out to get a randomly selected port assigned.



*Figure* *13-5.* *Node* *port* *service* *discovery*

Let’s see some of its distinguishing characteristics:

*Port* *number*

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

*Firewall* *rules*

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

*Node* *selection*

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

*Pods* *selection*

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

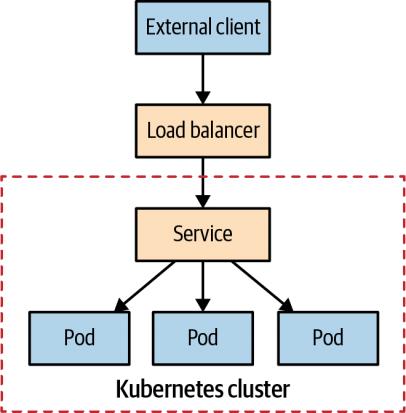
are Pods placed on every node (e.g., by using daemon services) or make sure the client knows which nodes have healthy Pods placed on them.

*Source* *addresses*

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

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

In addition to creating a regular Service, and opening a port on every node, as with type: NodePort, it also exposes the service externally using a cloud provider’s load balancer. [Figure 13-6](#bookmark465) shows this setup: a proprietary load balancer serves as a gateway to the Kubernetes cluster.



*Figure* *13-6.* *Load* *balancer* *service* *discovery*

So this type of Service works only when the cloud provider has Kubernetes support and provisions a load balancer. We can create a Service with a load balancer by specifying the type LoadBalancer. Kubernetes then will add IP addresses to the .spec and .status fields, as shown in [Example 13-7](#bookmark466).

*Example* *13-7.* *Service* *of* *type* *LoadBalancer*

**apiVersion** : v1

**kind** : Service

**metadata** :

**name** : random-generator

**spec** :

**type** : LoadBalancer

**clusterIP** : <10.0.171.239> [o](#bookmark468)

**loadBalancerIP** : <78.11.24.19>

**selector** :

**app** : random-generator

**ports** :

- **port** : 80

**targetPort** : 8080

**status** : [](#bookmark470)

**loadBalancer** :

**ingress** :

- **ip** : <146.148.47.155>

[](#bookmark467) Kubernetes assigns clusterIP and loadBalancerIP when they are available. [](#bookmark469) The status field is managed by Kubernetes and adds the Ingress IP.

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



Yet another type of Service is available: *headless* services, for which you don’t request a dedicated IP address. You create a headless service by specifying clusterIP None within the Service’s spec sec ‐ tion. For headless services, the backing Pods are added to the inter ‐ nal DNS server and are most useful for implementing Services to StatefulSets, as described in detail in [Chapter 12, “Stateful Service”](#bookmark100).

**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](HTTP-basedaccesstoServicesthroughexternallyreachableURLs) [externally reachable URLs](HTTP-basedaccesstoServicesthroughexternallyreachableURLs), load balancing, TLS termination, and name-based virtual hosting, but there are also other specialized Ingress implementations. For Ingress to work, the cluster must have one or more Ingress controllers running. A simple Ingress that exposes a single Service is shown in [Example 13-8](#bookmark472).

*Example* *13-8.* *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 control ‐ ler 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](HTTPURIpathslookslike) [Example 13-9](#bookmark473).

*Example* *13-9.* *A* *definition* *for* *Nginx* *Ingress* *controller*

**apiVersion** : networking.k8s.io/v1beta1

**kind** : Ingress

**metadata** :

**name** : random-generator

**annotations** :

**nginx.ingress.kubernetes.io/rewrite-target** : /

**spec** :

**rules** : [o](#bookmark476)

- **[http](http:paths:)**[:](http:paths:)

**[paths](http:paths:)**[:](http:paths:)

- **path** : / [](#bookmark478)

**backend** :

**serviceName** : random-generator

**servicePort** : 8080

- **path** : /cluster-status [](#bookmark480)

**backend** :

**serviceName** : cluster-status

**servicePort** : 80

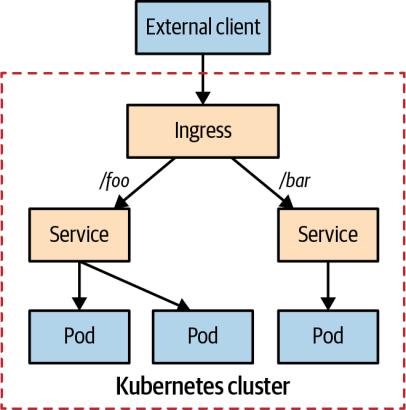
[](#bookmark475) Dedicated rules for the Ingress controller for dispatching requests based on the request path.

[](#bookmark477) Redirect every request to Service random-generator …

[](#bookmark479) … 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 def‐ inition would provision a load balancer and get an external IP address that services two Services under two different paths, as shown in [Figure 13-7](#bookmark481).

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



*Figure* *13-7.* *Application* *layer* *service* *discovery*

|  |
| --- |
| **OpenShift** **Routes**  Red Hat OpenShift is a popular enterprise distribution of Kubernetes. Besides being fully compliant with Kubernetes, OpenShift provides some additional features. One of these features is Routes, which are very similar to Ingress. They are so similar, in fact, the differences might be difficult to spot. First of all, Routes predates the introduction of the Ingress object in Kubernetes, so Routes can be considered a kind of predecessor of Ingress.  However, some technical differences still exist between Routes and Ingress objects:  • A Route is picked up automatically by the OpenShift-integrated HAProxy load balancer, so there is no requirement for an extra Ingress controller to be installed.  • You can use additional TLS termination modes like re-encryption or pass- through for the leg to the Service.  • Multiple weighted backends for splitting traffic can be used.  • Wildcard domains are supported.  Having said all that, you can use Ingress on OpenShift too. So you have the choice when using OpenShift. |

**Discussion**

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

[Table 13-1](#bookmark482) summarizes the various ways service discovery is implemented in Kuber ‐ netes. This table aims to organize the various service discovery mechanisms in this chapter from more straightforward to more complex. We hope it can help you build a mental model and understand them better.

*Table* *13-1.* *Service* *Discovery* *mechanisms*

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

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

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

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

**More** **Information**

• [Service Discovery Example](https://oreil.ly/nagmD)

• [Kubernetes Service](https://oreil.ly/AEDi5)

• [DNS for Services and Pods](https://oreil.ly/WRT5H)

• [Debug Services](https://oreil.ly/voVbw)

• [Using Source IP](https://oreil.ly/mGjzg)

• [Create an External Load Balancer](https://oreil.ly/pzOiM)

• [Ingress](https://oreil.ly/Idv2c)

• [Kubernetes NodePort Versus LoadBalancer Versus Ingress? When Should I Use](https://oreil.ly/W4i8U) [What?](https://oreil.ly/W4i8U)

• [Kubernetes Ingress Versus OpenShift Route](https://oreil.ly/fXicP)

**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**

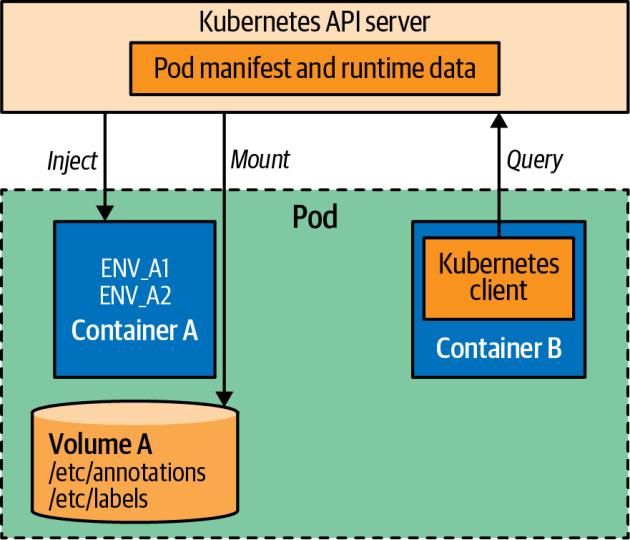
For the majority of use cases, cloud native applications are stateless and disposable without an identity relevant to other applications. However, sometimes even these kinds of applications need to have information about themselves and the environ ‐ ment they are running in. That may include information 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 algo ‐ rithm or memory allocation. You may want to use the Pod name and the hostname while logging information, or while sending metrics to a central server. You may want to discover other Pods in the same namespace with a specific label and join them into a clustered application. For these and other use cases, Kubernetes provides the downward API.

**Solution**

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

The Kubernetes approach is even more elegant and easier to use. The *downward* *API* allows you to pass metadata about the Pod to the containers and the cluster through environment variables and files. These are the same mechanisms we used for passing application-related data from ConfigMaps and Secrets. But in this case, the data is not created by us. Instead, we specify the keys that interest us, and Kubernetes populates the values dynamically. [Figure 14-1](#bookmark126) gives an overview of how the downward API injects resource and runtime information into interested Pods.



*Figure* *14-1.* *Application* *introspection* *mechanisms*

The main point here is that with the downward API, the metadata is injected into your Pod and made available locally. The application does not need to use a client and interact with the Kubernetes API and can remain Kubernetes-agnostic. Let’s see how easy it is to request metadata through environment variables in [Example 14-1](#bookmark483).

*Example* *14-1.* *Environment* *variables* *from* *downward* *API*

**apiVersion** : v1

**kind** : Pod

**metadata** :

**name** : random-generator

**spec** :

**containers** :

- **image** : k8spatterns/random-generator:1.0

**name** : random-generator

**env** :

- **name** : POD\_IP

**valueFrom** :

**fieldRef**: [](#bookmark485)

**fieldPath** : status.podIP

- **name** : MEMORY\_LIMIT

**valueFrom** :

**resourceFieldRef**:

**containerName** : random-generator [](#bookmark487)

**resource** : limits.memory

[o](#bookmark484) The environment variable POD\_IP is set from the properties of this Pod and comes into existence at Pod startup time.

[](#bookmark486) The environment variable MEMORY\_LIMIT is set to the value of the memory

resource limit of this container; the actual limit declaration is not shown here.

In this example, we use fieldRef to access Pod-level metadata. The keys shown in [Table 14-1](#bookmark488) are available for fieldRef.fieldPath both as environment variables and downwardAPI volumes.

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

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

As with fieldRef, we use resourceFieldRef to access metadata specific to a con ‐ tainer’s resource specification belonging to the Pod. This metadata is specific to a container and is specified with resourceFieldRef.container. When used as an environment variable, by default the current container is used. Possible keys for resourceFieldRef.resource are shown in [Table 14-2](#bookmark489). Resource declarations are explained in [Chapter 2, “Predictable Demands”](#bookmark14).

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

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

A user can change certain metadata such as labels and annotations while a Pod is run ‐ ning. 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.annota tions references. [Example 14-2](#bookmark490) shows how such volumes can be used.

*Example* *14-2.* *Downward* *API* *through* *volumes*

**apiVersion** : v1

**kind** : Pod

**metadata** :

**name** : random-generator

**spec** :

**containers** :

- **image** : k8spatterns/random-generator:1.0

**name** : random-generator

**volumeMounts** :

- **name** : pod-info [o](#bookmark492)

**mountPath** : /pod-info

**volumes** :

- **name** : pod-info

**downwardAPI** :

**items** :

- **path** : labels [](#bookmark494)

**fieldRef**:

**fieldPath** : metadata.labels

- **path** : annotations [](#bookmark495)

**fieldRef**:

**fieldPath** : metadata.annotations

[](#bookmark491) Values from the downward API can be mounted as files into the Pod.

[](#bookmark493) The file labels contain all labels, line by line, in the format name=value. This file gets updated when labels are changing.

[](#bookmark496) The annotations file holds all annotations in the same format as the labels.

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

**Discussion**

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

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

**More** **Information**

• [Self Awareness Example](https://oreil.ly/fHu1O)

• [AWS EC2: Instance Metadata and User Data](https://oreil.ly/iCwPr)

• [Expose Pod Information to Containers Through Files](https://oreil.ly/qe2Gc)

• [Expose Pod Information to Containers Through Environment Variables](https://oreil.ly/bZrtR)

• [Downward API: Available Fields](https://oreil.ly/Jh4zf)