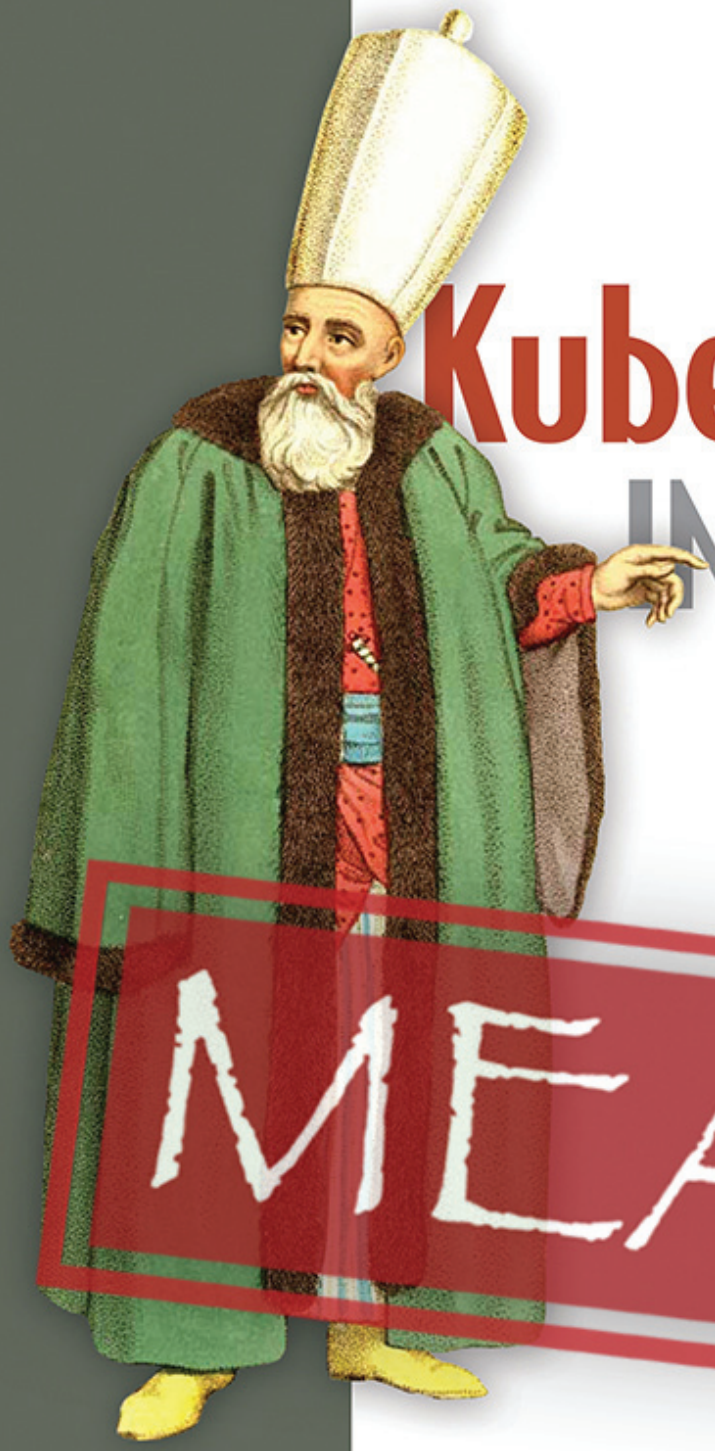

Kubernetes IN ACTION

SECOND EDITION

Marko Lukša

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Kubernetes in Action
Second edition
Version 9

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP for *Kubernetes in Action, Second Edition*.

As part of my work at Red Hat, I started using Kubernetes in 2014, even before version 1.0 was released. Those were interesting times. Not many people working in the software industry knew about Kubernetes, and there was no real community yet. There were hardly any blog posts about it and the documentation was still very basic. Kubernetes itself was ridden with bugs. When you combine all these facts, you can imagine that working with Kubernetes was extremely difficult.

In 2015 I was asked by Manning to write the first edition of this book. The originally planned 300-page book grew to over 600 pages full of information. The writing forced me to also research those parts of Kubernetes that I wouldn't have looked at more closely otherwise. I put most of what I learned into the book. Judging by their reviews and comments, readers love a detailed book like this.

The plan for the second edition of the book is to add even more information and to rearrange some of the existing content. The exercises in this book will take you from deploying a trivial application that initially uses only the basic features of Kubernetes to a full-fledged application that incorporates additional features as the book introduces them.

The book is divided into five parts. In the first part, after the introduction of Kubernetes and containers, you'll deploy the application in the simplest way. In the second part you'll learn the main concepts used to describe and deploy your application. After that you'll explore the inner workings of Kubernetes components. This will give you a good foundation to learn the difficult part - how to manage Kubernetes in production. In the last part of the book you'll learn about best practices and how to extend Kubernetes.

I hope you all like this second edition even better than the first, and if you're reading the book for the first time, your feedback will be even more valuable. If any part of the book is difficult to understand, please post your questions, comments or suggestions in the liveBook forum.

Thank you for helping me write the best book possible.

—Marko Lukša

brief contents

1 Introducing Kubernetes

This chapter covers

- Introductory information about Kubernetes and its origins
- Why Kubernetes has seen such wide adoption
- How Kubernetes transforms your data center
- An overview of its architecture and operation
- How and if you should integrate Kubernetes into your own organization

Before you can learn about the ins and outs of running applications with Kubernetes, you must first gain a basic understanding of the problems Kubernetes is designed to solve, how it came about, and its impact on application development and deployment. This first chapter is intended to give a general overview of these topics.

1.1 Introducing Kubernetes

The word *Kubernetes* is Greek for pilot or helmsman, the person who steers the ship - the person standing at the helm (the ship's wheel). A helmsman is not necessarily the same as a captain. A captain is responsible for the ship, while the helmsman is the one who steers it.

After learning more about what Kubernetes does, you'll find that the name hits the spot perfectly. A helmsman maintains the course of the ship, carries out the orders given by the captain and reports back the ship's heading. Kubernetes steers your applications and reports on their status while you - the captain - decide where you want the system to go.

How to pronounce Kubernetes and what is k8s?

The correct Greek pronunciation of Kubernetes, which is *Kie-ver-nee-tees*, is different from the English pronunciation you normally hear in technical conversations. Most often it's *Koo-ber-netties* or *Koo-ber-nay'-tace*, but you may also hear *Koo-ber-nets*, although rarely.

In both written and oral conversations, it's also referred to as *Kube* or *K8s*, pronounced *Kates*, where the 8 signifies the number of letters omitted between the first and last letter.

1.1.1 Kubernetes in a nutshell

Kubernetes is a software system for automating the deployment and management of complex, large-scale application systems composed of computer processes running in containers. Let's learn what it does and how it does it.

ABSTRACTING AWAY THE INFRASTRUCTURE

When software developers or operators decide to deploy an application, they do this through Kubernetes instead of deploying the application to individual computers. Kubernetes provides an abstraction layer over the underlying hardware to both users and applications.

As you can see in the following figure, the underlying infrastructure, meaning the computers, the network and other components, is hidden from the applications, making it easier to develop and configure them.

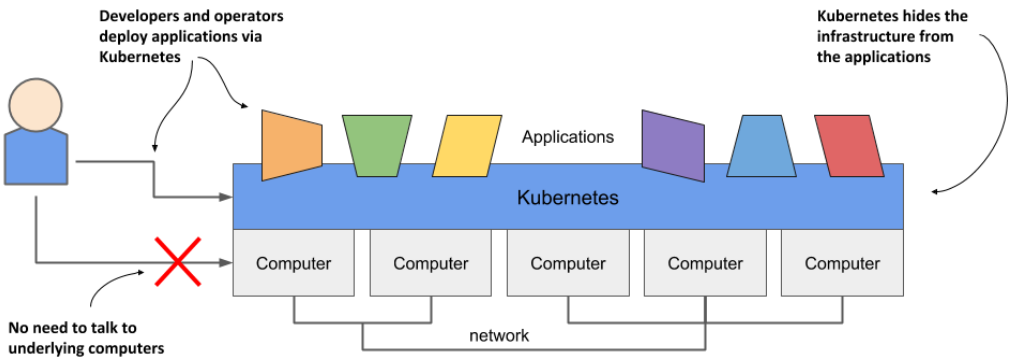

Figure 1.1 Infrastructure abstraction using Kubernetes

STANDARDIZING HOW WE DEPLOY APPLICATIONS

Because the details of the underlying infrastructure no longer affect the deployment of applications, you deploy applications to your corporate data center in the same way as you do in the cloud. A single manifest that describes the application can be used for local deployment and for deploying on any cloud provider. All differences in the underlying infrastructure are handled by Kubernetes, so you can focus on the application and the business logic it contains.

DEPLOYING APPLICATIONS DECLARATIVELY

Kubernetes uses a declarative model to define an application, as shown in the next figure. You describe the components that make up your application and Kubernetes turns this description into a running application. It then keeps the application healthy by restarting or recreating parts of it as needed.

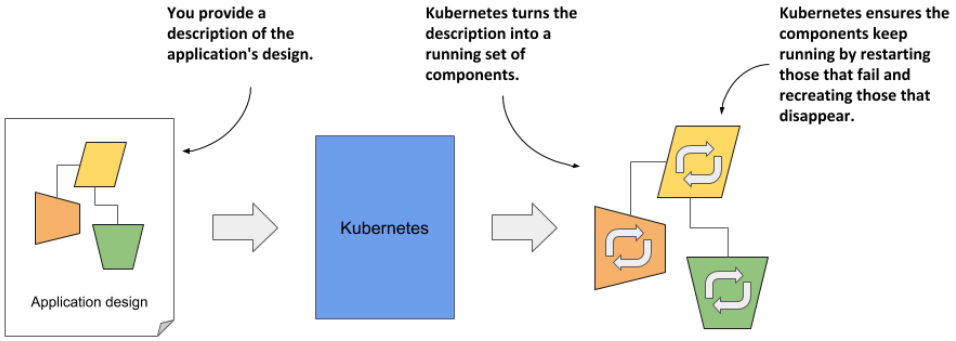

Figure 1.2 The declarative model of application deployment

Whenever you change the description, Kubernetes will take the necessary steps to reconfigure the running application to match the new description, as shown in the next figure.

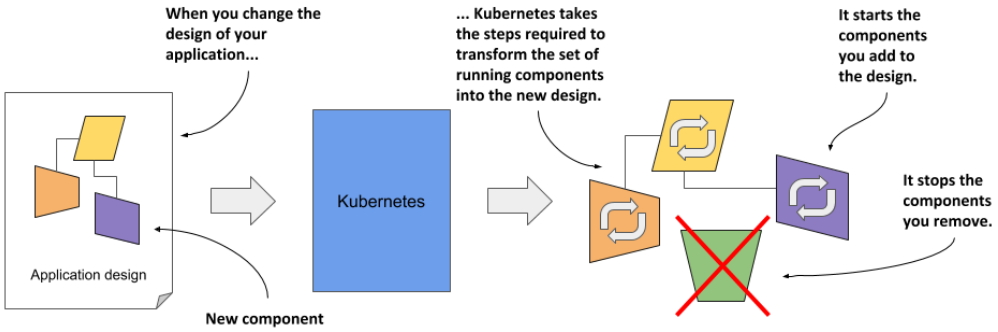

Figure 1.3 Changes in the description are reflected in the running application

TAKING ON THE DAILY MANAGEMENT OF APPLICATIONS

As soon as you deploy an application to Kubernetes, it takes over the daily management of the application. If the application fails, Kubernetes will automatically restart it. If the hardware fails or the infrastructure topology changes so that the application needs to be moved to other machines, Kubernetes does this all by itself. The engineers responsible for operating the system can focus on the big picture instead of wasting time on the details.

To circle back to the sailing analogy: the development and operations engineers are the ship's officers who make high-level decisions while sitting comfortably in their armchairs, and Kubernetes is the helmsman who takes care of the low-level tasks of steering the system through the rough waters your applications and infrastructure sail through.

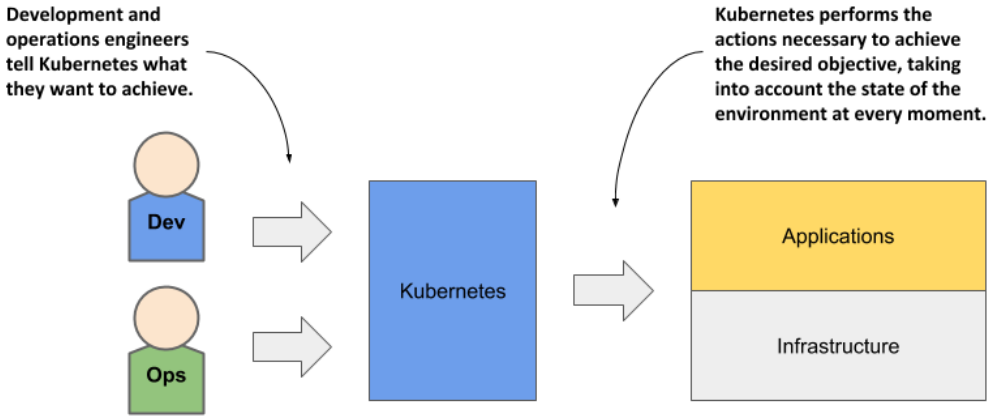

Figure 1.4 Kubernetes takes over the management of applications

Everything that Kubernetes does and all the advantages it brings requires a longer explanation, which we'll discuss later. Before we do that, it might help you to know how it all began and where the Kubernetes project currently stands.

1.1.2 About the Kubernetes project

Kubernetes was originally developed by Google. Google has practically always run applications in containers. As early as 2014, it was reported that they start two billion containers every week. That's over 3,000 containers per second, and the figure is much higher today. They run these containers on thousands of computers distributed across dozens of data centers around the world. Now imagine doing all this manually. It's clear that you need automation, and at this massive scale, it better be perfect.

ABOUT BORG AND OMEGA - THE PREDECESSORS OF KUBERNETES

The sheer scale of Google's workload has forced them to develop solutions to make the development and management of thousands of software components manageable and cost-effective. Over the years, Google developed an internal system called *Borg* (and later a new system called Omega) that helped both application developers and operators manage these thousands of applications and services.

In addition to simplifying development and management, these systems have also helped them to achieve better utilization of their infrastructure. This is important in any organization, but when you operate hundreds of thousands of machines, even tiny improvements in

utilization mean savings in the millions, so the incentives for developing such a system are clear.

NOTE Data on Google's energy use suggests that they run around 900,000 servers.

Over time, your infrastructure grows and evolves. Every new data center is state-of-the-art. Its infrastructure differs from those built in the past. Despite the differences, the deployment of applications in one data center should not differ from deployment in another data center. This is especially important when you deploy your application across multiple zones or regions to reduce the likelihood that a regional failure will cause application downtime. To do this effectively, it's worth having a consistent method for deploying your applications.

ABOUT KUBERNETES - THE OPEN-SOURCE PROJECT - AND COMMERCIAL PRODUCTS DERIVED FROM IT

Based on the experience they gained while developing Borg, Omega and other internal systems, in 2014 Google introduced Kubernetes, an open-source project that can now be used and further improved by everyone.

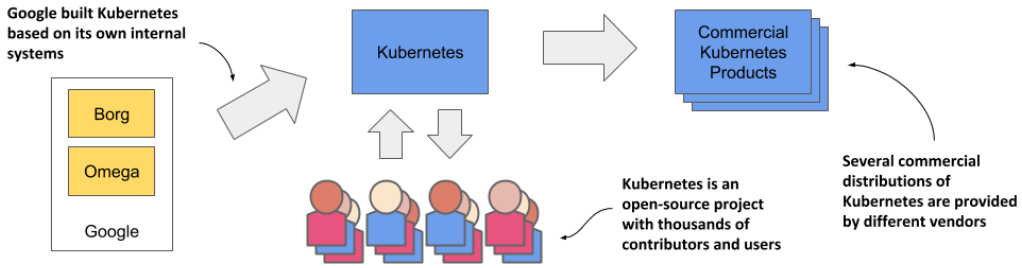

Figure 1.5 The origins and state of the Kubernetes open-source project

As soon as Kubernetes was announced, long before version 1.0 was officially released, other companies, such as Red Hat, who has always been at the forefront of open-source software, quickly stepped on board and helped develop the project. It eventually grew far beyond the expectations of its founders, and today is arguably one of the world's leading open-source projects, with dozens of organizations and thousands of individuals contributing to it.

Several companies are now offering enterprise-quality Kubernetes products that are built from the open-source project. These include Red Hat OpenShift, Pivotal Container Service, Rancher and many others.

HOW KUBERNETES GREW A WHOLE NEW CLOUD-NATIVE ECO-SYSTEM

Kubernetes has also spawned many other related open-source projects, most of which are now under the umbrella of the *Cloud Native Computing Foundation* (CNCF), which is part of the *Linux Foundation*.

CNCF organizes several KubeCon - CloudNativeCon conferences per year - in North America, Europe and China. In 2019, the total number of attendees exceeded 23,000, with KubeCon North America reaching an overwhelming number of 12,000 participants. These

figures show that Kubernetes has had an incredibly positive impact on the way companies around the world deploy applications today. It wouldn't have been so widely adopted if that wasn't the case.

1.1.3 Understanding why Kubernetes is so popular

In recent years, the way we develop applications has changed considerably. This has led to the development of new tools like Kubernetes, which in turn have fed back and fuelled further changes in application architecture and the way we develop them. Let's look at concrete examples of this.

AUTOMATING THE MANAGEMENT OF MICROSERVICES

In the past, most applications were large monoliths. The components of the application were tightly coupled, and they all ran in a single computer process. The application was developed as a unit by a large team of developers and the deployment of the application was straightforward. You installed it on a powerful computer and provided the little configuration it required. Scaling the application horizontally was rarely possible, so whenever you needed to increase the capacity of the application, you had to upgrade the hardware - in other words, scale the application vertically.

Then came the microservices paradigm. The monoliths were divided into dozens, sometimes hundreds, of separate processes, as shown in the following figure. This allowed organizations to divide their development departments into smaller teams where each team developed only a part of the entire system - just some of the microservices.

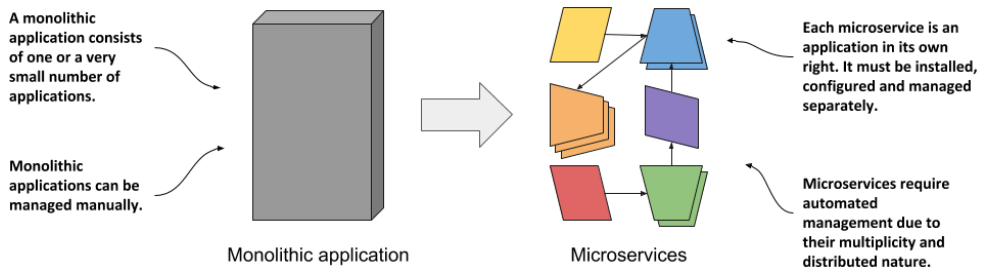

Figure 1.6 Comparing monolithic applications with microservices

Each microservice is now a separate application with its own development and release cycle. The dependencies of different microservices will inevitably diverge over time. One microservice requires one version of a library, while another microservice requires another, possibly incompatible, version of the same library. Running the two applications in the same operating system becomes difficult.

Fortunately, containers alone solve this problem where each microservice requires a different environment, but each microservice is now a separate application that must be managed individually. The increased number of applications makes this much more difficult.

Individual parts of the entire application no longer need to run on the same computer, which makes it easier to scale the entire system, but also means that the applications need to

be configured to communicate with each other. For systems with only a handful of components, this can usually be done manually, but it's now common to see deployments with well over a hundred microservices.

When the system consists of many microservices, automated management is crucial. Kubernetes provides this automation. The features it offers make the task of managing hundreds of microservices almost trivial.

BRIDGING THE DEV AND OPS DIVIDE

Along with these changes in application architecture, we've also seen changes in the way teams develop and run software. It used to be normal for a development team to build the software in isolation and then throw the finished product over the wall to the operations team, who would then deploy it and manage it from there.

With the advent of the Dev-ops paradigm, the two teams now work much more closely together throughout the entire life of the software product. The development team is now much more involved in the daily management of the deployed software. But that means that they now need to know about the infrastructure on which it's running.

As a software developer, your primary focus is on implementing the business logic. You don't want to deal with the details of the underlying servers. Fortunately, Kubernetes hides these details.

STANDARDIZING THE CLOUD

Over the past decade or two, many organizations have moved their software from local servers to the cloud. The benefits of this seem to have outweighed the fear of being locked-in to a particular cloud provider, which is caused by relying on the provider's proprietary APIs to deploy and manage applications.

Any company that wants to be able to move its applications from one provider to another will have to make additional, initially unnecessary efforts to abstract the infrastructure and APIs of the underlying cloud provider from the applications. This requires resources that could otherwise be focused on building the primary business logic.

Kubernetes has also helped in this respect. The popularity of Kubernetes has forced all major cloud providers to integrate Kubernetes into their offerings. Customers can now deploy applications to any cloud provider through a standard set of APIs provided by Kubernetes.

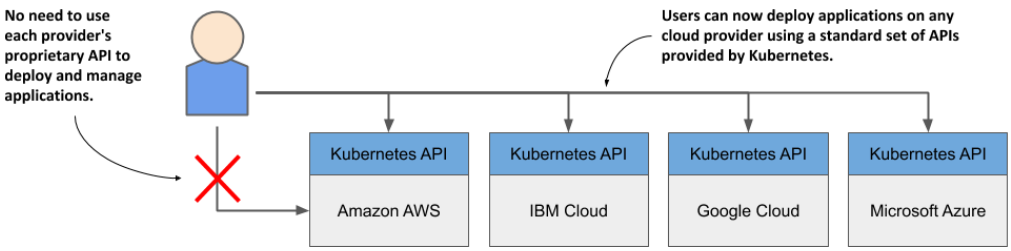

Figure 1.7 Kubernetes has standardized how you deploy applications on cloud providers

If the application is built on the APIs of Kubernetes instead of directly on the proprietary APIs of a specific cloud provider, it can be transferred relatively easily to any other provider.

1.2 Understanding Kubernetes

The previous section explained the origins of Kubernetes and the reasons for its wide adoption. In this section we'll take a closer look at what exactly Kubernetes is.

1.2.1 Understanding how Kubernetes transforms a computer cluster

Let's take a closer look at how the perception of the data center changes when you deploy Kubernetes on your servers.

KUBERNETES IS LIKE AN OPERATING SYSTEM FOR COMPUTER CLUSTERS

One can imagine Kubernetes**Error! Bookmark not defined.** as an operating system for the cluster. The next figure illustrates the analogies between an operating system running on a computer and Kubernetes running on a cluster of computers.

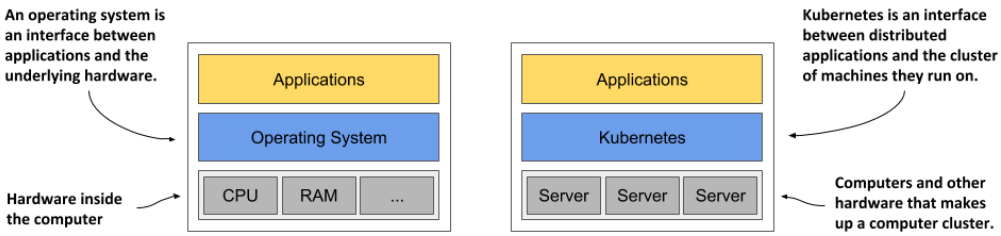

Figure 1.8 Kubernetes is to a computer cluster what an Operating System is to a computer

Just as an operating system supports the basic functions of a computer, such as scheduling processes onto its CPUs and acting as an interface between the application and the computer's hardware, Kubernetes schedules the components of a distributed application onto individual computers in the underlying computer cluster and acts as an interface between the application and the cluster.

It frees application developers from the need to implement infrastructure-related mechanisms in their applications; instead, they rely on Kubernetes to provide them. This includes things like:

- *service discovery* - a mechanism that allows applications to find other applications and use the services they provide,
- *horizontal scaling* - replicating your application to adjust to fluctuations in load,
- *load-balancing* - distributing load across all the application replicas,
- *self-healing* - keeping the system healthy by automatically restarting failed applications and moving them to healthy nodes after their nodes fail,
- *leader election* - a mechanism that decides which instance of the application should be active while the others remain idle but ready to take over if the active instance fails.

By relying on Kubernetes to provide these features, application developers can focus on implementing the core business logic instead of wasting time integrating applications with the**Error! Bookmark not defined.** infrastructure.

HOW KUBERNETES FITS INTO A COMPUTER CLUSTER

To get a concrete example of how Kubernetes is deployed onto a cluster of computers, look at the following figure.

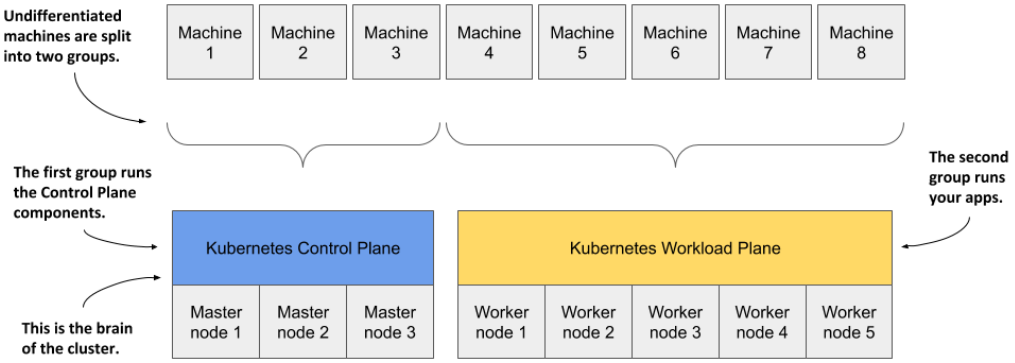

Figure 1.9 Computers in a Kubernetes cluster are divided into the Control Plane and the Workload Plane

You start with a fleet of machines that you divide into two groups - the master and the worker nodes. The master nodes will run the Kubernetes Control Plane, which represents the brain of your system and controls the cluster, while the rest will run your applications - your workloads - and will therefore represent the Workload Plane.

> **NOTE** The Workload Plane is sometimes referred to as the Data Plane, but this term could be confusing because the plane doesn't host data but applications. Don't be confused by the term "plane" either - in this context you can think of it as the "surface" the applications run on.

Non-production clusters can use a single master node, but highly available clusters use at least three physical master nodes to host the Control Plane. The number of worker nodes depends on the number of applications you'll deploy.

HOW ALL CLUSTER NODES BECOME ONE LARGE DEPLOYMENT AREA

After Kubernetes is installed on the computers, you no longer need to think about individual computers when deploying applications. Regardless of the number of worker nodes in your cluster, they all become a single space where you deploy your applications. You do this using the Kubernetes API, which is provided by the Kubernetes Control Plane.

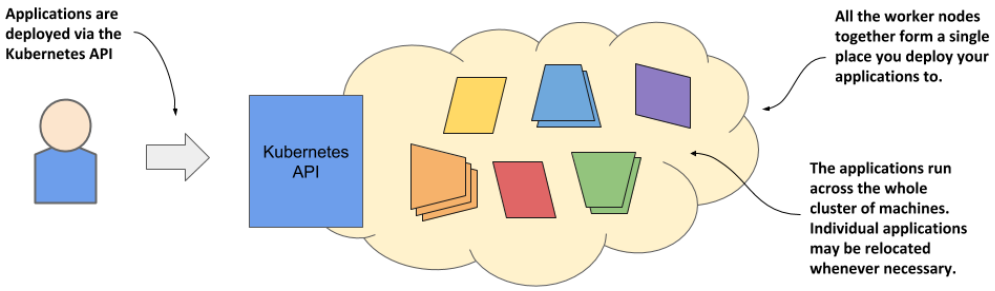

Figure 1.10 Kubernetes exposes the cluster as a uniform deployment area

When I say that all worker nodes become one space, I don't want you to think that you can deploy an extremely large application that is spread across several small machines. Kubernetes doesn't do magic tricks like this. Each application must be small enough to fit on one of the worker nodes.

What I meant was that when deploying applications, it doesn't matter which worker node they end up on. Kubernetes may later even move the application from one node to another. You may not even notice when that happens, and you shouldn't care.

1.2.2 The benefits of using Kubernetes

You've already learned why many organizations across the world have welcomed Kubernetes into their data centers. Now, let's take a closer look at the specific benefits it brings to both development and IT operations teams.

SELF-SERVICE DEPLOYMENT OF APPLICATIONS

Because**Error! Bookmark not defined.Error! Bookmark not defined.** Kubernetes presents all its worker nodes as a single deployment surface, it no longer matters which node you deploy your application to. This means that developers can now deploy applications on their own, even if they don't know anything about the number of nodes or the characteristics of each node.

In the past, the system administrators were the ones who decided where each application should be placed. This task is now left to Kubernetes. This allows a developer to deploy applications without having to rely on other people to do so. When a developer deploys an application, Kubernetes chooses the best node on which to run the application based on the resource requirements of the application and the resources available on each node.

REDUCING COSTS VIA BETTER INFRASTRUCTURE UTILIZATION

If you don't care which node your application lands on, it also means that it can be moved to any other node at any time without you having to worry about it. Kubernetes may need to do this to make room for a larger application that someone wants to deploy. This ability to move applications allows the applications to be packed tightly together so that the resources of the nodes can be utilized in the best possible way.

> **NOTE** In chapter 17 you'll learn more about how Kubernetes decides where to place each application and how you can influence the decision.

Finding optimal combinations can be challenging and time consuming, especially when the number of all possible options is huge, such as when you have many application components and many server nodes on which they can be deployed. Computers can perform this task much better and faster than**Error! Bookmark not defined.Error! Bookmark not defined.** humans. Kubernetes does it very well. By combining different applications on the same machines, Kubernetes improves the utilization of your hardware infrastructure so you can run more applications on fewer servers.

AUTOMATICALLY ADJUSTING TO CHANGING LOAD

Using**Error! Bookmark not defined.Error! Bookmark not defined.Error! Bookmark not defined.** Kubernetes to manage your deployed applications also means that the operations team doesn't have to constantly monitor the load of each application to respond to sudden load peaks. Kubernetes takes care of this also. It can monitor the resources consumed by each application and other metrics and adjust the number of running instances of each application to cope with increased load or resource usage.

When you run Kubernetes on cloud infrastructure, it can even increase the size of your cluster by provisioning additional nodes through the cloud provider's API. This way, you never run out of space to run additional instances of your applications.

KEEPING APPLICATIONS RUNNING SMOOTHLY

Kubernetes also makes every effort to ensure that your applications run smoothly. If your application crashes, Kubernetes will restart it automatically. So even if you have a broken application that runs out of memory after running for more than a few hours, Kubernetes will ensure that your application continues to provide the service to its users by automatically restarting it in this case.

Kubernetes is a self-healing system in that it deals with software errors like the one just described, but it also handles hardware failures. As clusters grow in size, the frequency of node failure also increases. For example, in a cluster with one hundred nodes and a MTBF (mean-time-between-failure) of 100 days for each node, you can expect one node to fail every day.

When a node fails, Kubernetes automatically moves applications to the remaining healthy nodes. The operations team no longer needs to manually move the application and can instead focus on repairing the node itself and returning it to the pool of available hardware resources.

If your infrastructure has enough free resources to allow normal system operation without the failed node, the operations team doesn't even have to react immediately to the failure. If it occurs in the middle of the night, no one from the operations team even has to wake up. They can sleep peacefully and deal with the failed node during regular working **Error! Bookmark not defined.Error! Bookmark not defined.Error! Bookmark not defined.Error! Bookmark not defined.**hours.

SIMPLIFYING APPLICATION DEVELOPMENT

The**Error! Bookmark not defined.** improvements described in the previous section mainly concern application deployment. But what about the process of application development? Does Kubernetes bring anything to their table? It definitely does.

As mentioned previously, Kubernetes offers infrastructure-related services that would otherwise have to be implemented in your applications. This includes the discovery of services and/or peers in a distributed application, leader election, centralized application configuration and others. Kubernetes provides this while keeping the application Kubernetes-agnostic, but when required, applications can also query the Kubernetes API to obtain detailed information about their environment. They can also use the API to change the environment.

1.2.3 The architecture of a Kubernetes cluster

As you've already learned, a Kubernetes cluster consists of nodes divided into two groups:

- A set of *master nodes***Error! Bookmark not defined.** that host the *Control Plane* components, which are the brains of the system, since they control the entire cluster.
- A set of *worker nodes****Error! Bookmark not defined.*** that form the *Workload Plane*, which is where your workloads (or applications) run.

The following figure shows the two planes and the different nodes they consist of.

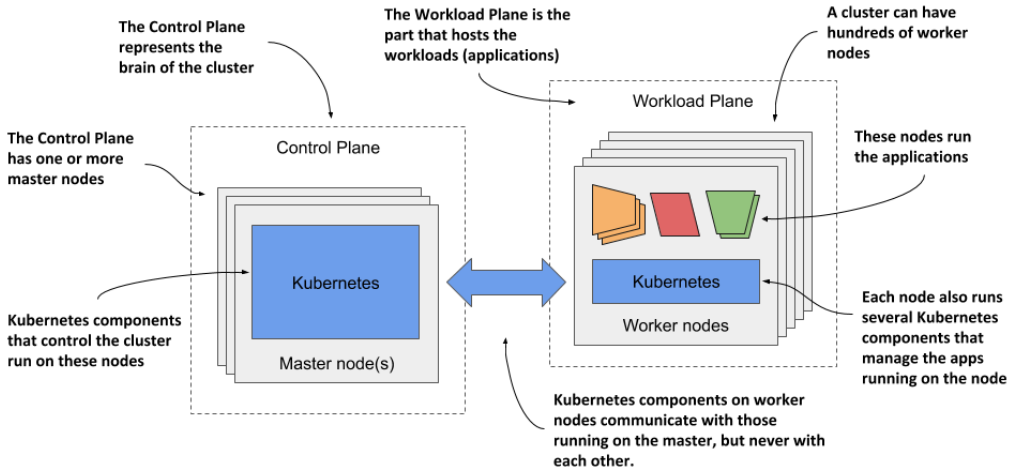

Figure 1.11 The two planes that make up a Kubernetes cluster

The two planes, and hence the two types of nodes, run different Kubernetes components. The next two sections of the book introduce them and summarize their functions without going into details. These components will be mentioned several times in the next part of the book where I explain the fundamental concepts of Kubernetes. An in-depth look at the components and their internals follows in the third part of the book.

CONTROL PLANE COMPONENTS

The**Error! Bookmark not defined.** Control Plane is what controls the cluster. It consists of several components that run on a single master node or are replicated across multiple master nodes to ensure high availability. The Control Plane's components are shown in the following figure.

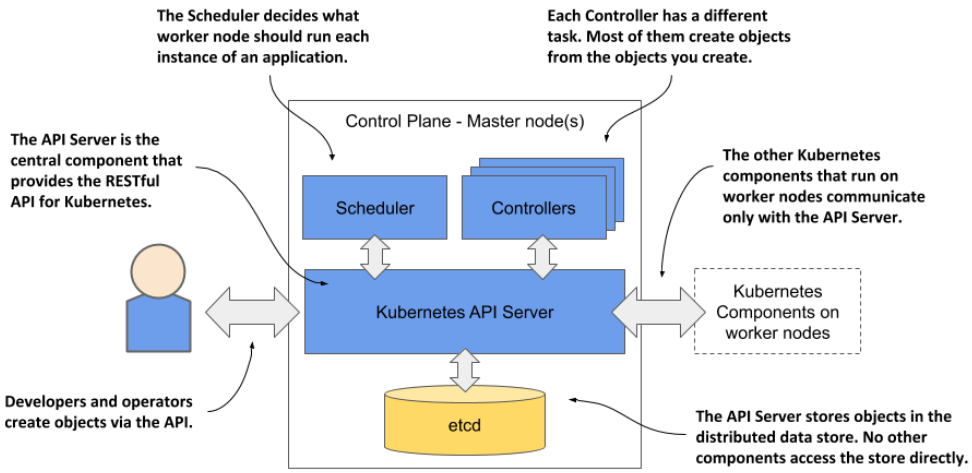

Figure 1.12 The components of the Kubernetes Control Plane

These are the components and their functions:

- The *Kubernetes API Server* exposes the RESTful Kubernetes API. Engineers using the cluster and other Kubernetes components create objects via this API.
- The *etcd* distributed datastore***Error! Bookmark not defined.*** persists the objects you create through the API, since the API Server itself is stateless. The Server is the only component that talks to etcd.
- The *Scheduler* decides on which worker node each application instance should run.
- *Controllers* bring to life the objects you create through the API. Most of them simply create other objects, but some also communicate with external systems (for example, the cloud provider via its API).

The components of the Control Plane hold and control the state of the cluster, but they don't run your applications. This is done by the (worker**Error! Bookmark not defined.**) nodes.

WORKER NODE COMPONENTS

The worker nodes are the computers on which your applications run. They form the cluster's Workload Plane. In addition to applications, several Kubernetes components also run on these nodes. They perform the task of running, monitoring and providing connectivity between your applications. They are shown in the following figure.

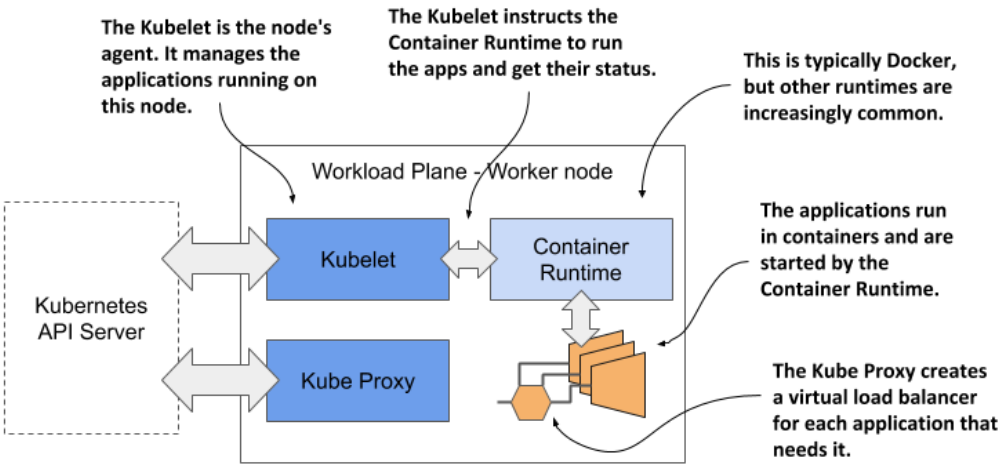

Figure 1.13 The Kubernetes components that run on each node

Each node runs the following set of components:

- The *Kubelet*, an agent that talks to the API server and manages the applications running on its node. It reports the status of these applications and the node via the API.
- The *Container Runtime*, which can be Docker or any other runtime compatible with Kubernetes. It runs your applications in containers as instructed by the Kubelet.
- The *Kubernetes Service Proxy (Kube Proxy)**Error! Bookmark not defined.* load-balances network traffic between applications. Its name suggests that traffic flows through it, but that's no longer the case. You'll learn why in chapter 14.

ADD-ON COMPONENTS

Most Kubernetes clusters also contain several other components. This includes a DNS server, network plugins, logging agents and many others. They typically run on the worker nodes but can also be configured to run on the master.

GAINING A DEEPER UNDERSTANDING OF THE ARCHITECTURE

For now, I only expect you to be vaguely familiar with the names of these components and their function, as I'll mention them many times throughout the following chapters. You'll learn snippets about them in these chapters, but I'll explain them in more detail in chapter 14.

I'm not a fan of explaining how things work until I first explain *what* something does and teach you how to use it. It's like learning to drive. You don't want to know what's under the hood. At first, you just want to learn how to get from point A to B. Only then will you be interested in how the car makes this possible. Knowing what's under the hood may one day help you get your car moving again after it has broken down and you are stranded on the side of the**Error! Bookmark not defined.Error! Bookmark not defined.** road. I hate to say it, but you'll have many moments like this when dealing with Kubernetes due to its sheer complexity.

1.2.4 How Kubernetes runs an application

With a general overview of the components that make up Kubernetes, I can finally explain how to deploy an application in Kubernetes.

DEFINING YOUR APPLICATION

Everything in Kubernetes is represented by an object. You create and retrieve these objects via the Kubernetes API. Your application consists of several types of these objects - one type represents the application deployment as a whole, another represents a running instance of your application, another represents the service provided by a set of these instances and allows reaching them at a single IP address, and there are many others.

All these types are explained in detail in the second part of the book. At the moment, it's enough to know that you define your application through several types of objects. These objects are usually defined in one or more manifest files in either YAML or JSON format.

> **DEFINITION** YAML was initially said to mean "Yet Another Markup Language", but it was latter changed to the recursive acronym "YAML Ain't Markup Language". It's one of the ways to serialize an object into a human-readable text file.

> **DEFINITION** JSON is short for JavaScript Object Notation. It's a different way of serializing an object, but more suitable for exchanging data between applications.

The following figure shows an example of deploying an application by creating a manifest with two deployments exposed using two services.

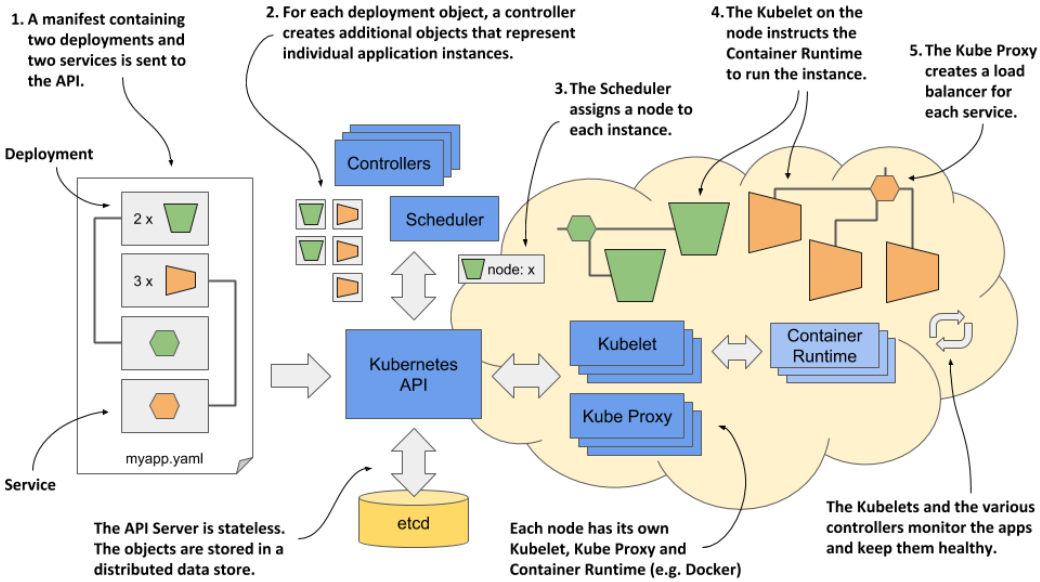

Figure 1.14 Deploying an application to Kubernetes

These actions take place when you deploy the application:

1. You submit the application manifest to the Kubernetes API. The API Server writes the objects defined in the manifest to etcd.
2. A controller notices the newly created objects and creates several new objects - one for each application instance.
3. The Scheduler assigns a node to each instance.
4. The Kubelet notices that an instance is assigned to the Kubelet's node. It runs the application instance via the Container Runtime.
5. The Kube Proxy notices that the application instances are ready to accept connections from clients and configures a load balancer for them.
6. The Kubelets and the Controllers monitor the system and keep the applications running.

The procedure is explained in more detail in the following sections, but the complete explanation is given in chapter 14, after you have familiarized yourself with all the objects and controllers involved.

SUBMITTING THE APPLICATION TO THE API

After you've created your YAML or JSON file(s), you submit the file to the API, usually via the Kubernetes command-line tool called *kubectl*.

NOTE Kubectl is pronounced *kube-control*, but the softer souls in the community prefer to call it *kube-cuddle*. Some refer to it as *kube-C-T-L*.

Kubectl splits the file into individual objects and creates each of them by sending an HTTP PUT or POST request to the API, as is usually the case with RESTful APIs. The API Server validates the objects and stores them in the etcd datastore. In addition, it notifies all interested components that these objects have been created. Controllers, which are explained next, are one of these components.

ABOUT THE CONTROLLERS

Most object types have an associated controller. A controller is interested in a particular object type. It waits for the API server to notify it that a new object has been created, and then performs operations to bring that object to life. Typically, the controller just creates other objects via the same Kubernetes API. For example, the controller responsible for application deployments creates one or more objects that represent individual instances of the application. The number of objects created by the controller depends on the number of replicas specified in the application deployment object.

ABOUT THE SCHEDULER

The scheduler is a special type of controller, whose only task is to schedule application instances onto worker nodes. It selects the best worker node for each new application instance object and assigns it to the instance - by modifying the object via the API.

ABOUT THE KUBELET AND THE CONTAINER RUNTIME

The Kubelet that runs on each worker node is also a type of controller. Its task is to wait for application instances to be assigned to the node on which it is located and run the application. This is done by instructing the Container Runtime to start the application's container.

ABOUT THE KUBE PROXY

Because an application deployment can consist of multiple application instances, a load balancer is required to expose them at a single IP address. The Kube Proxy, another controller running alongside the Kubelet, is responsible for setting up the load balancer.

KEEPING THE APPLICATIONS HEALTHY

Once the application is up and running, the Kubelet keeps the application healthy by restarting it when it terminates. It also reports the status of the application by updating the object that represents the application instance. The other controllers monitor these objects and ensure that applications are moved to healthy nodes if their nodes fail.

You're now roughly familiar with the architecture and functionality of Kubernetes. You don't need to understand or remember all the details at this moment, because internalizing this information will be easier when you learn about each individual object types and the controllers that bring them to life in the second part of the book.

1.3 Introducing Kubernetes into your organization

To close this chapter, let's see what options are available to you if you decide to introduce Kubernetes in your own IT environment.

1.3.1 Running Kubernetes on-premises and in the cloud

If you want to run your applications on Kubernetes, you have to decide whether you want to run them locally, in your organization's own infrastructure (on-premises) or with one of the major cloud providers, or perhaps both - in a hybrid cloud solution.

RUNNING KUBERNETES ON-PREMISES

Running Kubernetes on your own infrastructure may be your only option if regulations require you to run applications on site. This usually means that you'll have to manage Kubernetes yourself, but we'll come to that later.

Kubernetes can run directly on your bare-metal machines or in virtual machines running in your data center. In either case, you won't be able to scale your cluster as easily as when you run it in virtual machines provided by a cloud provider.

DEPLOYING KUBERNETES IN THE CLOUD

If you have no on-premises infrastructure, you have no choice but to run Kubernetes in the cloud. This has the advantage that you can scale your cluster at any time at short notice if required. As mentioned earlier, Kubernetes itself can ask the cloud provider to provision additional virtual machines when the current size of the cluster is no longer sufficient to run all the applications you want to deploy.

When the number of workloads decreases and some worker nodes are left without running workloads, Kubernetes can ask the cloud provider to destroy the virtual machines of these nodes to reduce your operational costs. This elasticity of the cluster is certainly one of the main benefits of running Kubernetes in the cloud.

USING A HYBRID CLOUD SOLUTION

A more complex option is to run Kubernetes on-premises, but also allow it to spill over into the cloud. It's possible to configure Kubernetes to provision additional nodes in the cloud if you exceed the capacity of your own data center. This way, you get the best of both worlds. Most of the time, your applications run locally without the cost of virtual machine rental, but in short periods of peak load that may occur only a few times a year, your applications can handle the extra load by using the additional resources in the cloud.

If your use-case requires it, you can also run a Kubernetes cluster across multiple cloud providers or a combination of any of the options mentioned. This can be done using a single control plane or one control plane in each location.

1.3.2 To manage or not to manage Kubernetes yourself

If you are considering introducing Kubernetes in your organization, the most important question you need to answer is whether you'll manage Kubernetes yourself or use a Kubernetes-as-a-Service type offering where someone else manages it for you.

MANAGING KUBERNETES YOURSELF

If you already run applications on-premises and have enough hardware to run a production-ready Kubernetes cluster, your first instinct is probably to deploy and manage it yourself. If you ask anyone in the Kubernetes community if this is a good idea, you'll usually get a very definite "no".

Figure 1.14 was a very simplified representation of what happens in a Kubernetes cluster when you deploy an application. Even that figure should have scared you. Kubernetes brings with it an enormous amount of additional complexity. Anyone who wants to run a Kubernetes cluster must be intimately familiar with its inner workings.

The management of production-ready Kubernetes clusters is a multi-billion-dollar industry. Before you decide to manage one yourself, it's essential that you consult with engineers who have already done it to learn about the issues most teams run into. If you don't, you may be setting yourself up for failure. On the other hand, trying out Kubernetes for non-production use-cases or using a managed Kubernetes cluster is much less problematic.

USING A MANAGED KUBERNETES CLUSTER IN THE CLOUD

Using Kubernetes is ten times easier than managing it. Most major cloud providers now offer Kubernetes-as-a-Service. They take care of managing Kubernetes and its components while you simply use the Kubernetes API like any of the other APIs the cloud provider offers.

The top managed Kubernetes offerings include the following:

- Google Kubernetes Engine (GKE)
- Azure Kubernetes Service (AKS)
- Amazon Elastic Kubernetes Service (EKS)
- IBM Cloud Kubernetes Service
- Red Hat OpenShift Online and Dedicated
- VMware Cloud PKS
- Alibaba Cloud Container Service for Kubernetes (ACK)

The first half of this book focuses on just using Kubernetes. You'll run the exercises in a local development cluster and on a managed GKE cluster, as I find it's the easiest to use and offers the best user experience. The second part of the book gives you a solid foundation for managing Kubernetes, but to truly master it, you'll need to gain additional experience.

1.3.3 Using vanilla or extended Kubernetes

The final question is whether to use a vanilla open-source version of Kubernetes or an extended, enterprise-quality Kubernetes product.

USING A VANILLA VERSION OF KUBERNETES

The open-source version of Kubernetes is maintained by the community and represents the cutting edge of Kubernetes development. This also means that it may not be as stable as the other options. It may also lack good security defaults. Deploying the vanilla version requires a lot of fine tuning to set everything up for production use.

USING ENTERPRISE-GRADE KUBERNETES DISTRIBUTIONS

A better option for using Kubernetes in production is to use an enterprise-quality Kubernetes distribution such as OpenShift or Rancher. In addition to the increased security and performance provided by better defaults, they offer additional object types in addition to those provided in the upstream Kubernetes API. For example, vanilla Kubernetes does not contain object types that represent cluster users, whereas commercial distributions do. They also provide additional software tools for deploying and managing well-known third-party applications on Kubernetes.

Of course, extending and hardening Kubernetes takes time, so these commercial Kubernetes distributions usually lag one or two versions behind the upstream version of Kubernetes. It's not as bad as it sounds. The benefits usually outweigh the disadvantages.

1.3.4 Should you even use Kubernetes?

I hope this chapter has made you excited about Kubernetes and you can't wait to squeeze it into your IT stack. But to close this chapter properly, we need to say a word or two about when introducing Kubernetes is not a good idea.

DO YOUR WORKLOADS REQUIRE AUTOMATED MANAGEMENT?

The first thing you need to be honest about is whether you need to automate the management of your applications at all. If your application is a large monolith, you definitely don't need Kubernetes.

Even if you deploy microservices, using Kubernetes may not be the best option, especially if the number of your microservices is very small. It's difficult to provide an exact number when the scales tip over, since other factors also influence the decision. But if your system consists of less than five microservices, throwing Kubernetes into the mix is probably not a good idea. If your system has more than twenty microservices, you will most likely benefit from the integration of Kubernetes. If the number of your microservices falls somewhere in between, other factors, such as the ones described next, should be considered.

CAN YOU AFFORD TO INVEST YOUR ENGINEERS' TIME INTO LEARNING KUBERNETES?

Kubernetes is designed to allow applications to run without them knowing that they are running in Kubernetes. While the applications themselves don't need to be modified to run in Kubernetes, development engineers will inevitably spend a lot of time learning how to use Kubernetes, even though the operators are the only ones that actually need that knowledge.

It would be hard to tell your teams that you're switching to Kubernetes and expect only the operations team to start exploring it. Developers like shiny new things. At the time of writing, Kubernetes is still a very shiny thing.

ARE YOU PREPARED FOR INCREASED COSTS IN THE INTERIM?

While Kubernetes reduces long-term operational costs, introducing Kubernetes in your organization initially involves increased costs for training, hiring new engineers, building and purchasing new tools and possibly additional hardware. Kubernetes requires additional computing resources in addition to the resources that the applications use.

DON'T BELIEVE THE HYPE

Although Kubernetes has been around for several years at the time of writing this book, I can't say that the hype phase is over. The initial excitement has just begun to calm down, but many engineers may still be unable to make rational decisions about whether the integration of Kubernetes is as necessary as it seems.

1.4 Summary

In this introductory chapter, you've learned that:

- Kubernetes is Greek for helmsman. As a ship's captain oversees the ship while the helmsman steers it, you oversee your computer cluster, while Kubernetes performs the day-to-day management tasks.
- Kubernetes is pronounced *koo-ber-netties*. Kubectl, the Kubernetes command-line tool, is pronounced *kube-control*.
- Kubernetes is an open-source project built upon Google's vast experience in running applications on a global scale. Thousands of individuals now contribute to it.
- Kubernetes uses a declarative model to describe application deployments. After you provide a description of your application to Kubernetes, it brings it to life.
- Kubernetes is like an operating system for the cluster. It abstracts the infrastructure and presents all computers in a data center as one large, contiguous deployment area.
- Microservice-based applications are more difficult to manage than monolithic applications. The more microservices you have, the more you need to automate their management with a system like Kubernetes.
- Kubernetes helps both development and operations teams to do what they do best. It frees them from mundane tasks and introduces a standard way of deploying applications both on-premises and in any cloud.
- Using Kubernetes allows developers to deploy applications without the help of system administrators. It reduces operational costs through better utilization of existing hardware, automatically adjusts your system to load fluctuations, and heals itself and the applications running on it.
- A Kubernetes cluster consists of master and worker nodes. The master nodes run the *Control Plane*, which controls the entire cluster, while the worker nodes run the deployed applications or workloads, and therefore represent the *Workload Plane*.
- Using Kubernetes is simple, but managing it is hard. An inexperienced team should use a Kubernetes-as-a-Service offering instead of deploying Kubernetes by itself.

So far, you've only observed the ship from the pier. It's time to come aboard. But before you leave the docks, you should inspect the shipping containers it's carrying. You'll do this next.

2

Understanding containers

This chapter covers

- Understanding what a container is
- Differences between containers and virtual machines
- Creating, running, and sharing a container image with Docker
- Linux kernel features that make containers possible

Kubernetes primarily manages applications that run in containers - so before you start exploring Kubernetes, you need to have a good understanding of what a container is. This chapter explains the basics of Linux containers that a typical Kubernetes user needs to know.

2.1 Introducing containers

In Chapter 1 you learned how different microservices running in the same operating system may require different, potentially conflicting versions of dynamically linked libraries or have different environment requirements.

When a system consists of a small number of applications, it's okay to assign a dedicated virtual machine to each application and run each in its own operating system. But as the microservices become smaller and their numbers start to grow, you may not be able to afford to give each one its own VM if you want to keep your hardware costs low and not waste resources.

It's not just a matter of wasting hardware resources - each VM typically needs to be individually configured and managed, which means that running higher numbers of VMs also results in higher staffing requirements and the need for a better, often more complicated automation system. Due to the shift to microservice architectures, where systems consist of hundreds of deployed application instances, an alternative to VMs was needed. Containers are that alternative.

2.1.1 Comparing containers to virtual machines

Instead of using virtual machines to isolate the environments of individual microservices (or software processes in general), most development and operations teams now prefer to use containers. They allow you to run multiple services on the same host computer, while keeping them isolated from each other. Like VMs, but with much less overhead.

Unlike VMs, which each run a separate operating system with several system processes, a process running in a container runs within the existing host operating system. Because there is only one operating system, no duplicate system processes exist. Although all the application processes run in the same operating system, their environments are isolated, though not as well as when you run them in separate VMs. To the process in the container, this isolation makes it look like no other processes exist on the computer. You'll learn how this is possible in the next few sections, but first let's dive deeper into the differences between containers and virtual machines.

COMPARING THE OVERHEAD OF CONTAINERS AND VIRTUAL MACHINES

Compared to VMs, containers are much lighter, because they don't require a separate resource pool or any additional OS-level processes. While each VM usually runs its own set of system processes, which requires additional computing resources in addition to those consumed by the user application's own process, a container is nothing more than an isolated process running in the existing host OS that consumes only the resources the app consumes. They have virtually no overhead.

Figure 2.1 shows two bare metal computers, one running two virtual machines, and the other running containers instead. The latter has space for additional containers, as it runs only one operating system, while the first runs three – one host and two guest OSes.

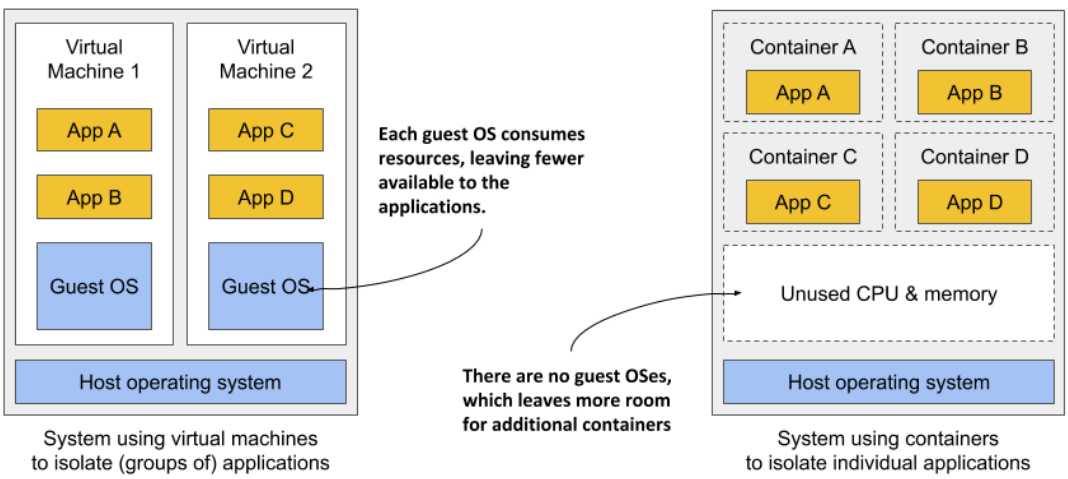

Figure 2.1 Using VMs to isolate groups of applications vs. isolating individual apps with containers

Because of the resource overhead of VMs, you often group multiple applications into each VM. You can't afford to dedicate a whole VM to each app. But containers introduce no overhead, which means you can afford to create a separate container for each application. In fact, you should never run multiple applications in the same container, as this makes managing the processes in the container much more difficult. Moreover, all existing software dealing with containers, including Kubernetes itself, is designed under the premise that there's only one application in a container.

COMPARING THE START-UP TIME OF CONTAINERS AND VIRTUAL MACHINES

In addition to the lower runtime overhead, containers also start the application faster, because only the application process itself needs to be started. No additional system processes need to be started first, as is the case when booting up a new virtual machine.

COMPARING THE ISOLATION OF CONTAINERS AND VIRTUAL MACHINES

You'll agree that containers are clearly better when it comes to the use of resources, but there's also a disadvantage. When you run applications in virtual machines, each VM runs its own operating system and kernel. Underneath those VMs is the hypervisor (and possibly an additional operating system), which splits the physical hardware resources into smaller sets of virtual resources that the operating system in each VM can use. As figure 2.2 shows, applications running in these VMs make system calls (*sys-calls*) to the guest OS kernel in the VM, and the machine instructions that the kernel then executes on the virtual CPUs are then forwarded to the host's physical CPU via the hypervisor.

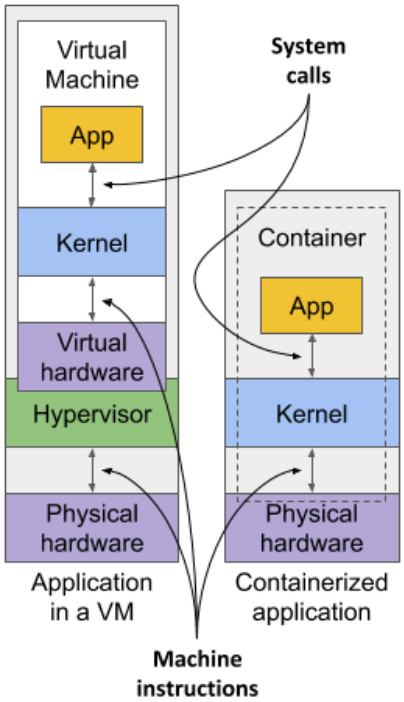

Figure 2.2 How apps use the hardware when running in a VM vs. in a container

> **NOTE** Two types of hypervisors exist. Type 1 hypervisors don't require running a host OS, while type 2 hypervisors do.

Containers, on the other hand, all make system calls on the single kernel running in the host OS. This single kernel is the only one that executes instructions on the host's CPU. The CPU doesn't need to handle any kind of virtualization the way it does with VMs.

Examine the following figure to see the difference between running three applications on bare metal, running them in two separate virtual machines, or running them in three containers.

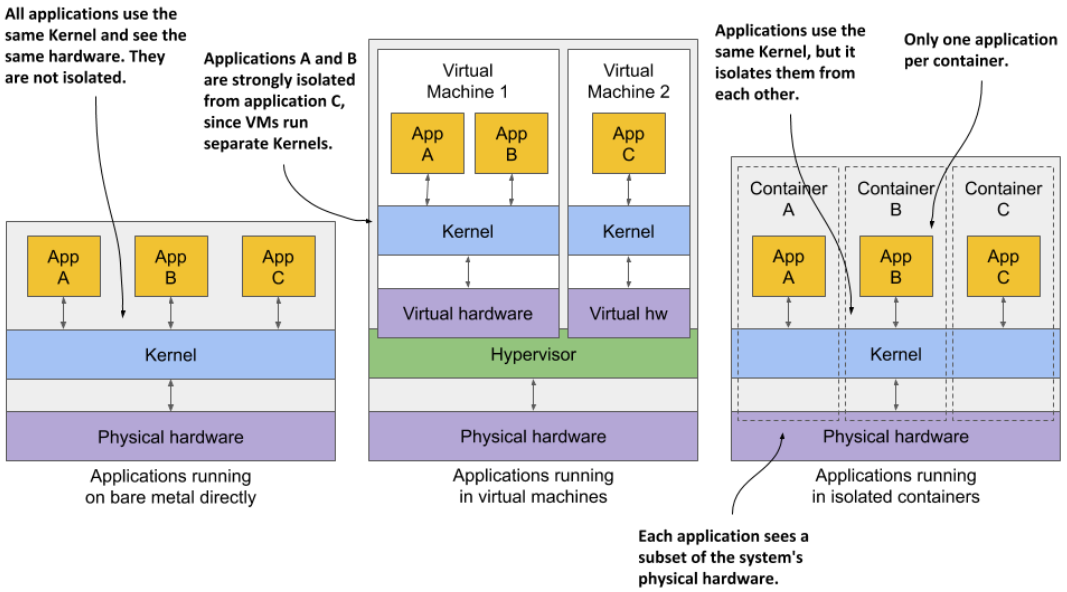

Figure 2.3 The difference between running applications on bare metal, in virtual machines, and in containers

In the first case, all three applications use the same kernel and aren't isolated at all. In the second case, applications A and B run in the same VM and thus share the kernel, while application C is completely isolated from the other two, since it uses its own kernel. It only shares the hardware with the first two.

The third case shows the same three applications running in containers. Although they all use the same kernel, they are isolated from each other and completely unaware of the others' existence. The isolation is provided by the kernel itself. Each application sees only a part of the physical hardware and sees itself as the only process running in the OS, although they all run in the same OS.

UNDERSTANDING THE SECURITY-IMPLICATIONS OF CONTAINER ISOLATION

The main advantage of using virtual machines over containers is the complete isolation they provide, since each VM has its own Linux kernel, while containers all use the same kernel. This can clearly pose a security risk. If there's a bug in the kernel, an application in one container might use it to read the memory of applications in other containers. If the apps run in different VMs and therefore share only the hardware, the probability of such attacks is much lower. Of course, complete isolation is only achieved by running applications on separate physical machines.

Additionally, containers share memory space, whereas each VM uses its own chunk of memory. Therefore, if you don't limit the amount of memory that a container can use, this could cause other containers to run out of memory or cause their data to be swapped out to disk.

NOTE This can't happen in Kubernetes, because it requires that swap is disabled on all the nodes.

UNDERSTANDING WHAT ENABLES CONTAINERS AND WHAT ENABLES VIRTUAL MACHINES

While virtual machines are enabled through virtualization support in the CPU and by virtualization software on the host, containers are enabled by the Linux kernel itself. You'll learn about container technologies later when you can try them out for yourself. You'll need to have Docker installed for that, so let's learn how it fits into the container story.

2.1.2 Introducing the Docker container platform

While container technologies have existed for a long time, they only became widely known with the rise of Docker. Docker was the first container system that made them easily portable across different computers. It simplified the process of packaging up the application and all its libraries and other dependencies - even the entire OS file system - into a simple, portable package that can be used to deploy the application on any computer running Docker.

INTRODUCING CONTAINERS, IMAGES AND REGISTRIES

Docker is a platform for packaging, distributing and running applications. As mentioned earlier, it allows you to package your application along with its entire environment. This can be just a few dynamically linked libraries required by the app, or all the files that are usually shipped with an operating system. Docker allows you to distribute this package via a public repository to any other Docker-enabled computer.

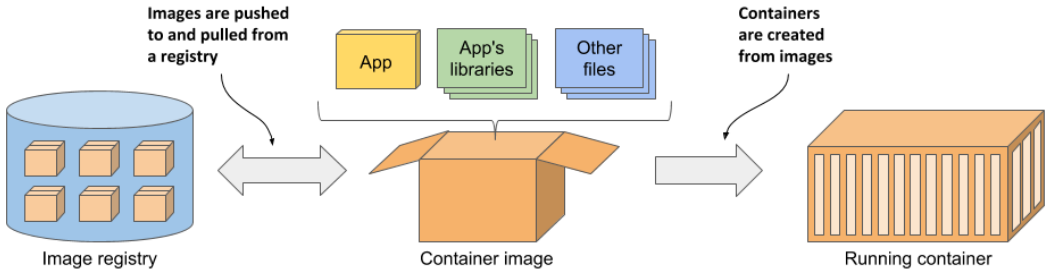

Figure 2.4 The three main Docker concepts are images, registries and containers

Figure 2.4 shows three main Docker concepts that appear in the process I've just described. Here's what each of them is:

- *Images*—A container image is something you package your application and its environment into. Like a zip file or a tarball. It contains the whole filesystem that the application will use and additional metadata, such as the path to the executable file to run when the image is executed, the ports the application listens on, and other information about the image.
- *Registries*—A registry is a repository of container images that enables the exchange of images between different people and computers. After you build your image, you can either run it on the same computer, or *push* (upload) the image to a registry and then *pull* (download) it to another computer. Certain registries are public, allowing anyone to pull images from it, while others are private and only accessible to individuals, organizations or computers that have the required authentication credentials.
- *Containers*—A container is instantiated from a container image. A running container is a normal process running in the host operating system, but its environment is isolated from that of the host and the environments of other processes. The file system of the container originates from the container image, but additional file systems can also be mounted into the container. A container is usually resource-restricted, meaning it can only access and use the amount of resources such as CPU and memory that have been allocated to it.

BUILDING, DISTRIBUTING, AND RUNNING A CONTAINER IMAGE

To understand how containers, images and registries relate to each other, let's look at how to build a container image, distribute it through a registry and create a running container from the image. These three processes are shown in figures 2.5 to 2.7.

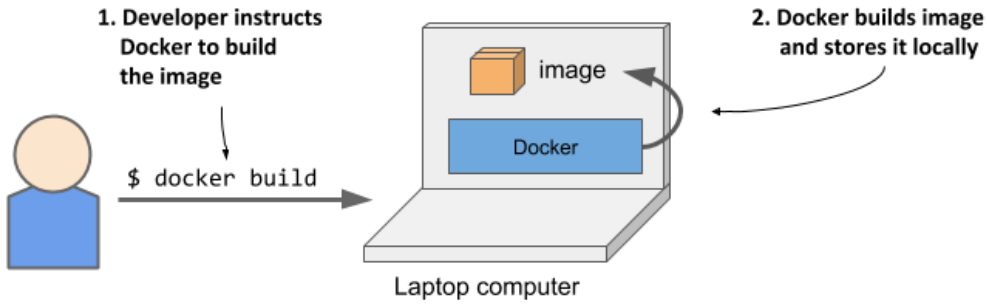

Figure 2.5 Building a container image

As shown in figure 2.5, the developer first builds an image, and then pushes it to a registry, as shown in figure 2.6. The image is now available to anyone who can access the registry.

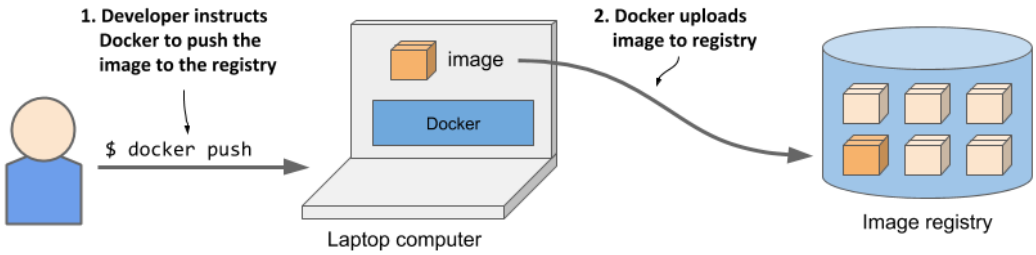

Figure 2.6 Uploading a container image to a registry

As the next figure shows, another person can now pull the image to any other computer running Docker and run it. Docker creates an isolated container based on the image and invokes the executable file specified in the image.

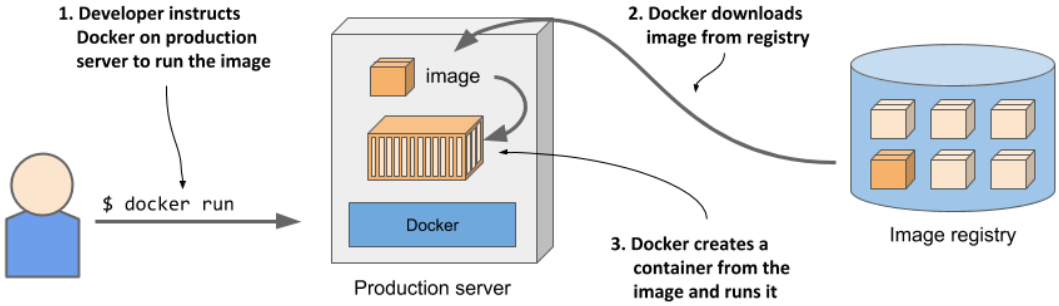

Figure 2.7 Running a container on a different computer

Running the application on any computer is made possible by the fact that the environment of the application is decoupled from the environment of the host.

UNDERSTANDING THE ENVIRONMENT THAT THE APPLICATION SEES

When you run an application in a container, it sees exactly the file system content you bundled into the container image, as well as any additional file systems you mount into the container. The application sees the same files whether it's running on your laptop or a full-fledged production server, even if the production server uses a completely different Linux distribution. The application typically has no access to the files in the host's operating system, so it doesn't matter if the server has a completely different set of installed libraries than your development computer.

For example, if you package your application with the files of the entire Red Hat Enterprise Linux (RHEL) operating system and then run it, the application will think it's running inside RHEL, whether you run it on your Fedora-based or a Debian-based computer. The Linux

distribution installed on the host is irrelevant. The only thing that might be important is the kernel version and the kernel modules it loads. Later, I'll explain why.

This is similar to creating a VM image by creating a new VM, installing an operating system and your app in it, and then distributing the whole VM image so that other people can run it on different hosts. Docker achieves the same effect, but instead of using VMs for app isolation, it uses Linux container technologies to achieve (almost) the same level of isolation.

UNDERSTANDING IMAGE LAYERS

Unlike virtual machine images, which are big blobs of the entire filesystem required by the operating system installed in the VM, container images consist of layers that are usually much smaller. These layers can be shared and reused across multiple images. This means that only certain layers of an image need to be downloaded if the rest were already downloaded to the host as part of another image containing the same layers.

Layers make image distribution very efficient but also help to reduce the storage footprint of images. Docker stores each layer only once. As you can see in the following figure, two containers created from two images that contain the same layers use the same files.

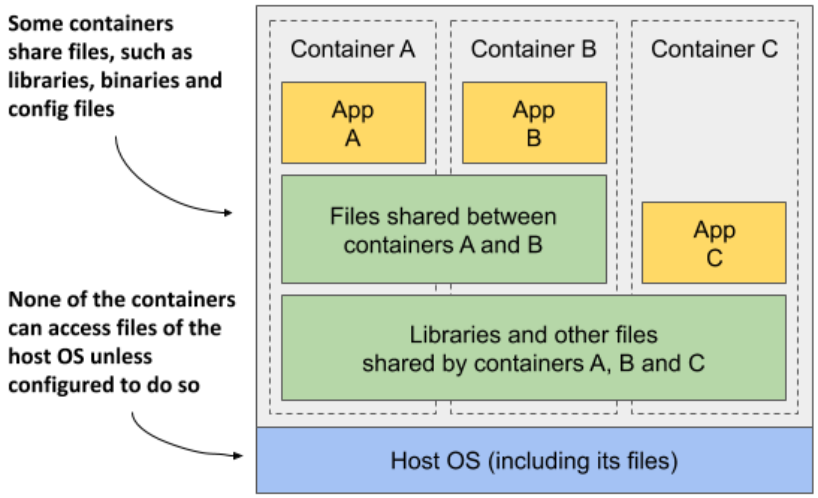

Figure 2.8 Containers can share image layers

The figure shows that containers A and B share an image layer, which means that applications A and B read some of the same files. In addition, they also share the underlying layer with container C. But if all three containers have access to the same files, how can they be completely isolated from each other? Are changes that application A makes to a file stored in the shared layer not visible to application B? They aren't. Here's why.

The filesystems are isolated by the Copy-on-Write (CoW) mechanism. The filesystem of a container consists of read-only layers from the container image and an additional read/write layer stacked on top. When an application running in container A changes a file in one of the

read-only layers, the entire file is copied into the container's read/write layer and the file contents are changed there. Since each container has its own writable layer, changes to shared files are not visible in any other container.

When you delete a file, it is only marked as deleted in the read/write layer, but it's still present in one or more of the layers below. What follows is that deleting files never reduces the size of the image.

> **WARNING** Even seemingly harmless operations such as changing permissions or ownership of a file result in a new copy of the entire file being created in the read/write layer. If you perform this type of operation on a large file or many files, the image size may swell significantly.

UNDERSTANDING THE PORTABILITY LIMITATIONS OF CONTAINER IMAGES

In theory, a Docker-based container image can be run on any Linux computer running Docker, but one small caveat exists, because containers don't have their own kernel. If a containerized application requires a particular kernel version, it may not work on every computer. If a computer is running a different version of the Linux kernel or doesn't load the required kernel modules, the app can't run on it. This scenario is illustrated in the following figure.

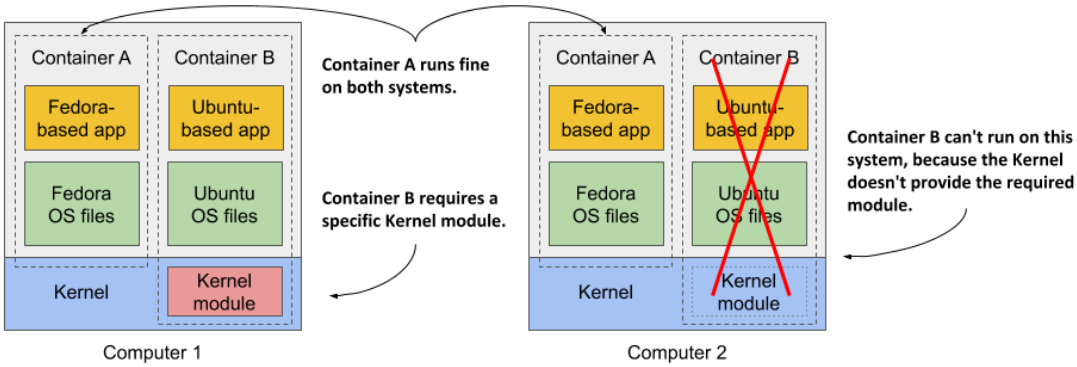

Figure 2.9 If a container requires specific kernel features or modules, it may not work everywhere

Container B requires a specific kernel module to run properly. This module is loaded in the kernel in the first computer, but not in the second. You can run the container image on the second computer, but it will break when it tries to use the missing module.

And it's not just about the kernel and its modules. It should also be clear that a containerized app built for a specific hardware architecture can only run on computers with the same architecture. You can't put an application compiled for the x86 CPU architecture into a container and expect to run it on an ARM-based computer just because Docker is available there. For this you would need a VM to emulate the x86 architecture.

2.1.3 Installing Docker and running a Hello World container

You should now have a basic understanding of what a container is, so let's use Docker to run one. You'll install Docker and run a Hello World container.

INSTALLING DOCKER

Ideally, you'll install Docker directly on a Linux computer, so you won't have to deal with the additional complexity of running containers inside a VM running within your host OS. But, if you're using macOS or Windows and don't know how to set up a Linux VM, the Docker Desktop application will set it up for you. The Docker command-line (CLI) tool that you'll use to run containers will be installed in your host OS, but the Docker daemon will run inside the VM, as will all the containers it creates.

The Docker Platform consists of many components, but you only need to install Docker Engine to run containers. If you use macOS or Windows, install Docker Desktop. For details, follow the instructions at http://docs.docker.com/install.

> **NOTE** Docker Desktop for Windows can run either Windows or Linux containers. Make sure that you configure it to use Linux containers, as all the examples in this book assume that's the case.

RUNNING A HELLO WORLD CONTAINER

After the installation is complete, you use the `docker` CLI tool to run Docker commands. Let's try pulling and running an existing image from Docker Hub, the public image registry that contains ready-to-use container images for many well-known software packages. One of them is the `busybox` image, which you'll use to run a simple `echo "Hello world"` command in your first container.

If you're unfamiliar with `busybox`, it's a single executable file that combines many of the standard UNIX command-line tools, such as `echo`, `ls`, `gzip`, and so on. Instead of the `busybox` image, you could also use any other full-fledged OS container image like Fedora, Ubuntu, or any other image that contains the `echo` executable file.

Once you've got Docker installed, you don't need to download or install anything else to run the `busybox` image. You can do everything with a single `docker run` command, by specifying the image to download and the command to run in it. To run the Hello World container, the command and its output are as follows:

```
$ docker run busybox echo "Hello World"
Unable to find image 'busybox:latest' locally    #A
latest: Pulling from library/busybox    #A
7c9d20b9b6cd: Pull complete    #A
Digest: sha256:fe301db49df08c384001ed752dff6d52b4...    #A
Status: Downloaded newer image for busybox:latest    #A
Hello World    #B
```

#A Docker downloads the container image
#B The output produced by the echo command

With this single command, you told Docker what image to create the container from and what command to run in the container. This may not look so impressive, but keep in mind that the

entire "application" was downloaded and executed with a single command, without you having to install the application or any of its dependencies.

In this example, the application was just a single executable file, but it could also have been a complex application with dozens of libraries and additional files. The entire process of setting up and running the application would be the same. What isn't obvious is that it ran in a container, isolated from the other processes on the computer. You'll see that this is true in the remaining exercises in this chapter.

UNDERSTANDING WHAT HAPPENS WHEN YOU RUN A CONTAINER

Figure 2.10 shows exactly what happens when you execute the `docker run` command.

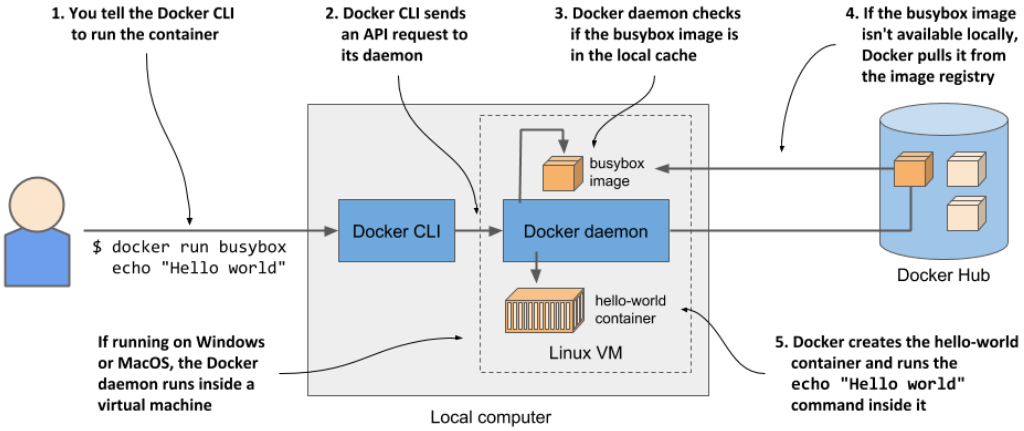

Figure 2.10 Running echo "Hello world" in a container based on the busybox container image

The `docker` CLI tool sends an instruction to run the container to the Docker daemon, which checks whether the `busybox` image is already present in its local image cache. If it isn't, the daemon pulls it from the Docker Hub registry.

After downloading the image to your computer, the Docker daemon creates a container from that image and executes the `echo` command in it. The command prints the text to the standard output, the process then terminates and the container stops.

If your local computer runs a Linux OS, the Docker CLI tool and the daemon both run in this OS. If it runs macOS or Windows, the daemon and the containers run in the Linux VM.

RUNNING OTHER IMAGES

Running other existing container images is much the same as running the `busybox` image. In fact, it's often even simpler, since you don't normally need to specify what command to execute, as with the echo command in the previous example. The command that should be executed is usually written in the image itself, but you can override it when you run it.

For example, if you want to run the Redis datastore, you can find the image name on http://hub.docker.com or another public registry. In the case of Redis, one of the images is called `redis:alpine`, so you'd run it like this:

```
$ docker run redis:alpine
```

To stop and exit the container, press Control-C (or Command-C on a Mac).

> **NOTE** If you want to run an image from a different registry, you must specify the registry along with the image name. For example, if you want to run an image from the Quay.io registry, which is another publicly accessible image registry, run it as follows: `docker run quay.io/some/image`.

UNDERSTANDING IMAGE TAGS

If you've searched for the Redis image on Docker Hub, you've noticed that there are many image *tags* you can choose from. For Redis, the tags are `latest`, `buster`, `alpine`, but also `5.0.7-buster`, `5.0.7-alpine`, and so on.

Docker allows you to have multiple versions or variants of the same image under the same name. Each variant has a unique tag. If you refer to images without explicitly specifying the tag, Docker assumes that you're referring to the special `latest` tag. When uploading a new version of an image, image authors usually tag it with both the actual version number and with `latest`. When you want to run the latest version of an image, use the `latest` tag instead of specifying the version.

> **NOTE** The `docker run` command only pulls the image if it hasn't already pulled it before. Using the `latest` tag ensures that you get the latest version when you first run the image. The locally cached image is used from that point on.

Even for a single version, there are usually several variants of an image. For Redis I mentioned `5.0.7-buster` and `5.0.7-alpine`. They both contain the same version of Redis, but differ in the base image they are built on. `5.0.7-buster` is based on Debian version "Buster", while `5.0.7-alpine` is based on the Alpine Linux base image, a very stripped-down image that is only 5MB in total – it contains only a small set of the installed binaries you see in a typical Linux distribution.

To run a specific version and/or variant of the image, specify the tag in the image name. For example, to run the `5.0.7-alpine` tag, you'd execute the following command:

```
$ docker run redis:5.0.7-alpine
```

These days, you can find container images for virtually all popular applications. You can use Docker to run those images using the simple `docker run` single-line command.

2.1.4 Introducing the Open Container Initiative and Docker alternatives

Docker was the first container platform to make containers mainstream. I hope I've made it clear that Docker itself is not what provides the process isolation. The actual isolation of containers takes place at the Linux kernel level using the mechanisms it provides. Docker is

the tool using these mechanisms to make running container almost trivial. But it's by no means the only one.

INTRODUCING THE OPEN CONTAINER INITIATIVE (OCI)

After the success of Docker, the Open Container Initiative (OCI) was born to create open industry standards around container formats and runtime. Docker is part of this initiative, as are other container runtimes and a number of organizations with interest in container technologies.

OCI members created the *OCI Image Format Specification*, which prescribes a standard format for container images, and the *OCI Runtime Specification*, which defines a standard interface for container runtimes with the aim of standardizing the creation, configuration and execution of containers.

INTRODUCING THE CONTAINER RUNTIME INTERFACE (CRI) AND ITS IMPLEMENTATION (CRI-O)

This book focuses on using Docker as the container runtime for Kubernetes, as it was initially the only one supported by Kubernetes and is still the most widely used. But Kubernetes now supports many other container runtimes through the Container Runtime Interface (CRI).

One implementation of CRI is CRI-O, a lightweight alternative to Docker that allows you to leverage any OCI-compliant container runtime with Kubernetes. Examples of OCI-compliant runtimes include *rkt* (pronounced Rocket), *runC*, and *Kata Containers*.

2.2 Deploying Kiada—the Kubernetes in Action Demo Application

Now that you've got a working Docker setup, you can start building a more complex application. You'll build a microservices-based application called *Kiada* - the Kubernetes in Action Demo Application.

In this chapter, you'll use Docker to run this application. In the next and remaining chapters, you'll run the application in Kubernetes. Over the course of this book, you'll iteratively expand it and learn about individual Kubernetes features that help you solve the typical problems you face when running applications.

2.2.1 Introducing the Kiada Suite

The Kubernetes in Action Demo Application is a web-based application that shows quotes from this book, asks you Kubernetes-related questions to help you check how your knowledge is progressing, and provides a list of hyperlinks to external websites related to Kubernetes or this book. It also prints out the information about the container that served processed the browser's request. You'll soon see why this is important.

THE LOOK AND OPERATION OF THE APPLICATION

A screenshot of the web application is presented in the following figure.

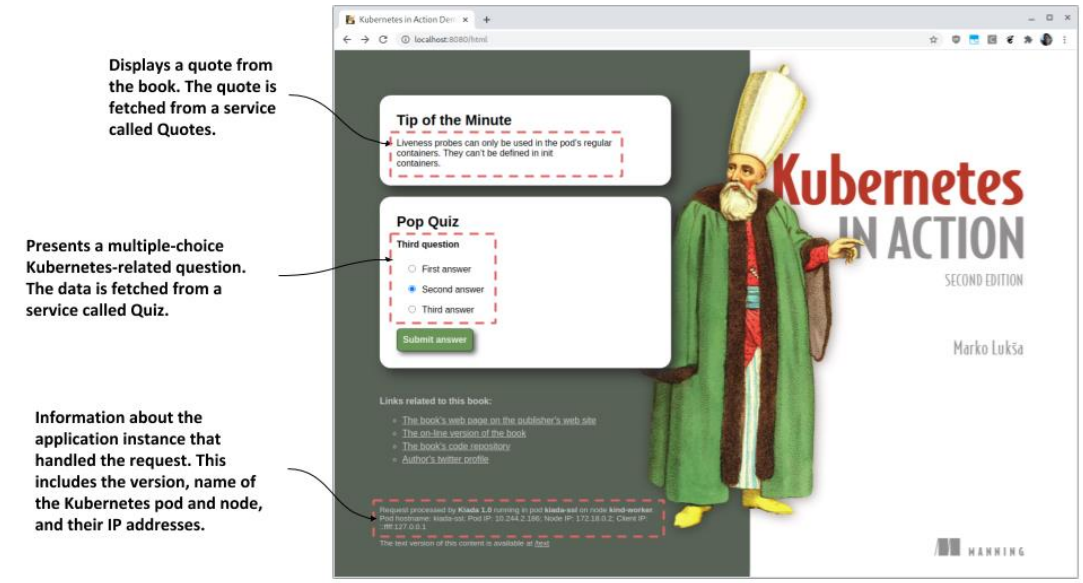

Figure 2.11 A screenshot of the Kubernetes in Action Demo Application (Kiada)

The architecture of the Kiada application is shown in the next figure. The HTML is served by a web application running in a Node.js server. The client-side JavaScript code then retrieves the quote and question from the Quote and the Quiz RESTful services. The Node.js application and the services comprise the complete *Kiada Suite*.

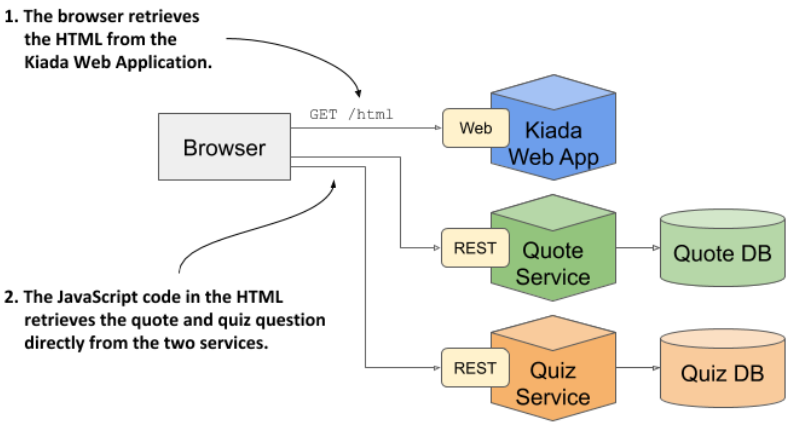

Figure 2.12 The architecture and operation of the Kiada Suite

The web browser talks directly to three different services. If you're familiar with microservice architectures, you might wonder why no API gateway exists in the system. This is so that I can demonstrate the issues and solutions associated with cases where many different services are deployed in Kubernetes (services that may not belong behind the same API gateway). But chapter 11 will also explain how to introduce Kubernetes-native API gateways into the system.

THE LOOK AND OPERATION OF THE PLAIN-TEXT VERSION

You'll spend a lot of time interacting with Kubernetes via a terminal, so you may not want to go back and forth between it and a web browser when you perform the exercises. For this reason, the application can also be used in plain-text mode.

The plain-text mode allows you to use the application directly from the terminal using a tool such as `curl`. In that case, the response sent by the application looks like the following:

```
==== TIP OF THE MINUTE
Liveness probes can only be used in the pod's regular containers.
They can't be defined in init containers.

==== POP QUIZ
Third question
0) First answer
1) Second answer
2) Third answer

Submit your answer to /question/0/answers/<index of answer> using the POST method.

==== REQUEST INFO
Request processed by Kubia 1.0 running in pod "kiada-ssl" on node "kind-worker".
Pod hostname: kiada-ssl; Pod IP: 10.244.2.188; Node IP: 172.18.0.2; Client IP: 127.0.0.1
```

The HTML version is accessible at the request URI `/html`, whereas the text version is at `/text`. If the client requests the root URI path `/`, the application inspects the `Accept` request header to guess whether the client is a graphical web browser, in which case it redirects it to `/html`, or a text-based tool like `curl`, in which case it sends the plain-text response.

In this mode of operation, it's the Node.js application that calls the Quote and the Quiz services, as shown in the next figure.

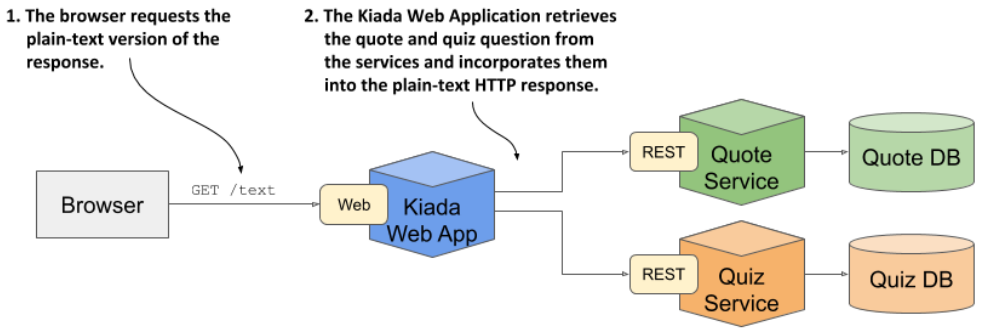

Figure 2.13 The operation when the client requests the text version

From a networking standpoint, this mode of operation is much different than the one described previously. In this case, the Quote and the Quiz service are invoked within the cluster, whereas previously, they were invoked directly from the browser. To support both operation modes, the services must therefore be exposed both internally and externally.

> **NOTE** The initial version of the application will not connect to any services. You'll build and incorporate the services in later chapters.

2.2.2 Building the application

With the general overview of the application behind us, it's time to start building the application. Instead of going straight to the full-blown version of the application, we'll take things slow and build the application iteratively throughout the book.

INTRODUCING THE INITIAL VERSION OF THE APPLICATION

The initial version of the application that you'll run in this chapter, while supporting both HTML and plain-text modes, will not display the quote and pop quiz, but merely the information about the application and the request. This includes the version of the application, the network hostname of the server that processed the client's request, and the IP of the client. Here's the plain-text response that it sends:

```
Kiada version 0.1. Request processed by "<server-hostname>". Client IP: <client-IP>
```

The application source code is available in the book's code repository on GitHub. You'll find the code of the initial version in the directory `Chapter02/kiada-0.1`. The JavaScript code is in the `app.js` file and the HTML and other resources are in the `html` subdirectory. The template for the HTML response is in `index.html`. For the plain-text response it's in `index.txt`.

You could now download and install Node.js locally and test the application directly on your computer, but that's not necessary. Since you already have Docker installed, it's easier to package the application into a container image and run it in a container. This way, you don't need to install anything, and you'll be able to run the same image on any other Docker-enabled host without installing anything there either.

CREATING THE DOCKERFILE FOR THE CONTAINER IMAGE

To package your app into an image, you need a file called `Dockerfile`, which contains a list of instructions that Docker performs when building the image. The following listing shows the contents of the file, which you'll find in Chapter02/kiada-0.1/Dockerfile.

Listing 2.1 A minimal Dockerfile for building a container image for your app

```
FROM node:12    #A
COPY app.js /app.js    #B
COPY html/ /html    #C
ENTRYPOINT ["node", "app.js"]    #D
```

#A The base image to build upon
#B Adds the app.js file into the container image
#C Copies the files in the html/ directory into the container image at /html/
#D Specifies the command to execute when the image is run

The `FROM` line defines the container image that you'll use as the starting point (the base image you're building on top of). The base image used in the listing is the `node` container image with the tag `12`. In the second line, the `app.js` file is copied from your local directory into the root directory of the image. Likewise, the third line copies the `html` directory into the image. Finally, the last line specifies the command that Docker should run when you start the container. In the listing, the command is `node app.js`.

Choosing a base image

You may wonder why use this specific image as your base. Because your app is a Node.js app, you need your image to contain the `node` binary file to run the app. You could have used any image containing this binary, or you could have even used a Linux distribution base image such as `fedora` or `ubuntu` and installed Node.js into the container when building the image. But since the `node` image already contains everything needed to run Node.js apps, it doesn't make sense to build the image from scratch. In some organizations, however, the use of a specific base image and adding software to it at build-time may be mandatory.

BUILDING THE CONTAINER IMAGE

The `Dockerfile`, the `app.js` file, and the files in the `html` directory is all you need to build your image. With the following command, you'll build the image and tag it as `kiada:latest`:

```
$ docker build -t kiada:latest .
Sending build context to Docker daemon  3.072kB
Step 1/4 : FROM node:12    #A
12: Pulling from library/node
092586df9206: Pull complete    #B
ef599477fae0: Pull complete    #B
...     #B
89e674ac3af7: Pull complete    #B
08df71ec9bb0: Pull complete    #B
Digest: sha256:a919d679dd773a56acce15afa0f436055c9b9f20e1f28b4469a4bee69e0...
Status: Downloaded newer image for node:12
 ---> e498dabfee1c    #C
Step 2/4 : COPY app.js /app.js    #D
 ---> 28d67701d6d9    #D
Step 3/4 : COPY html/ /html    #E
 ---> 1d4de446f0f0    #E
Step 4/4 : ENTRYPOINT ["node", "app.js"]    #F
 ---> Running in a01d42eda116    #F
Removing intermediate container a01d42eda116    #F
 ---> b0ecc49d7a1d    #F
Successfully built b0ecc49d7a1d    #G
Successfully tagged kiada:latest    #G
```

#A This corresponds to the first line of your Dockerfile
#B Docker downloads the individual layers of the node:12 image
#C This is the ID of image after the first build step is complete
#D The app.js is copied into the image
#E The html directory is copied into the image
#F The final step of the build
#G The final image ID and its tag

The `-t` option specifies the desired image name and tag, and the dot at the end specifies that Dockerfile and the artefacts needed to build the image are in the current directory. This is the so-called build context.

When the build process is complete, the newly created image is available in your computer's local image store. You can see it by listing local images with the following command:

```
$ docker images
REPOSITORY   TAG      IMAGE ID           CREATED             VIRTUAL SIZE
kiada        latest   b0ecc49d7a1d       1 minute ago        908 MB
...
```

UNDERSTANDING HOW THE IMAGE IS BUILT

The following figure shows what happens during the build process. You tell Docker to build an image called `kiada` based on the contents of the current directory. Docker reads the `Dockerfile` in the directory and builds the image based on the directives in the file.

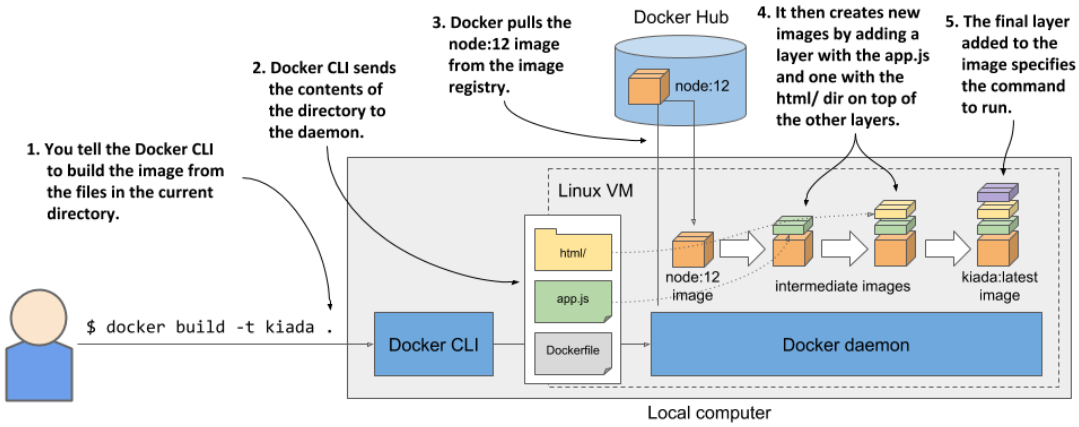

Figure 2.14 Building a new container image using a Dockerfile

The build itself isn't performed by the `docker` CLI tool. Instead, the contents of the entire directory are uploaded to the Docker daemon and the image is built by it. You've already learned that the CLI tool and the daemon aren't necessarily on the same computer. If you're using Docker on a non-Linux system such as macOS or Windows, the client is in your host OS, but the daemon runs inside a Linux VM. But it could also run on a remote computer.

> **TIP** Don't add unnecessary files to the build directory, as they will slow down the build process—especially if the Docker daemon is located on a remote system.

To build the image, Docker first pulls the base image (`node:12`) from the public image repository (Docker Hub in this case), unless the image is already stored locally. It then creates a new container from the image and executes the next directive from the Dockerfile. The container's final state yields a new image with its own ID. The build process continues by processing the remaining directives in the Dockerfile. Each one creates a new image. The final image is then tagged with the tag you specified with the `-t` flag in the `docker build` command.

UNDERSTANDING THE IMAGE LAYERS

Some pages ago, you learned that images consist of several layers. One might think that each image consists of only the layers of the base image and a single new layer on top, but that's not the case. When building an image, a new layer is created for each individual directive in the Dockerfile.

During the build of the `kiada` image, after it pulls all the layers of the base image, Docker creates a new layer and adds the `app.js` file into it. It then adds another layer with the files from the `html` directory and finally creates the last layer, which specifies the command to run when the container is started. This last layer is then tagged as `kiada:latest`.

You can see the layers of an image and their size by running `docker history`. The command and its output are shown next (note that the top-most layers are printed first):

```
$ docker history kiada:latest
IMAGE          CREATED     CREATED BY                                      SIZE
b0ecc49d7a1d   7 min ago   /bin/sh -c #(nop) ENTRYPOINT ["n...   0B    #A
1d4de446f0f0   7 min ago   /bin/sh -c #(nop) COPY dir:6ecee...   534kB    #A
28d67701d6d9   7 min ago   /bin/sh -c #(nop) COPY file:2ed5...   2.8kB    #A
e498dabfee1c   2 days ago  /bin/sh -c #(nop) CMD ["node"]        0B    #B
<missing>      2 days ago  /bin/sh -c #(nop) ENTRYPOINT ["d...   0B    #B
<missing>      2 days ago  /bin/sh -c #(nop) COPY file:2387...   116B    #B
<missing>      2 days ago  /bin/sh -c set -ex && for key in...   5.4MB    #B
<missing>      2 days ago  /bin/sh -c #(nop)  ENV YARN_VERS...   0B    #B
<missing>      2 days ago  /bin/sh -c ARCH= && dpkgArch="$(...   67MB    #B
<missing>      2 days ago  /bin/sh -c #(nop)  ENV NODE_VERS...   0B    #B
<missing>      3 weeks ago /bin/sh -c groupadd --gid 1000 n...   333kB    #B
<missing>      3 weeks ago /bin/sh -c set -ex;  apt-get upd...   562MB    #B
<missing>      3 weeks ago /bin/sh -c apt-get update && apt...   142MB    #B
<missing>      3 weeks ago /bin/sh -c set -ex;  if ! comman...   7.8MB    #B
<missing>      3 weeks ago /bin/sh -c apt-get update && apt...   23.2MB    #B
<missing>      3 weeks ago /bin/sh -c #(nop)  CMD ["bash"]       0B    #B
<missing>      3 weeks ago /bin/sh -c #(nop) ADD file:9788b...   101MB    #B
```

#A The three layers that you added
#B The layers of the node:12 image and its base image(s)

Most of the layers you see come from the `node:12` image (they also include layers of that image's own base image). The three uppermost layers correspond to the `COPY` and `ENTRYPOINT` directives in the Dockerfile.

As you can see in the `CREATED BY` column, each layer is created by executing a command in the container. In addition to adding files with the `COPY` directive, you can also use other directives in the Dockerfile. For example, the `RUN` directive executes a command in the container during the build. In the listing above, you'll find a layer where the `apt-get update` and some additional `apt-get` commands were executed. `apt-get` is part of the Ubuntu package manager used to install software packages. The command shown in the listing installs some packages onto the image's filesystem.

To learn about `RUN` and other directives you can use in a Dockerfile, refer to the Dockerfile reference at https://docs.docker.com/engine/reference/builder/.

> **TIP** Each directive creates a new layer. I have already mentioned that when you delete a file, it is only marked as deleted in the new layer and is not removed from the layers below. Therefore, deleting a file with a subsequent directive won't reduce the size of the image. If you use the `RUN` directive, make sure that the command it executes deletes all temporary files it creates before it terminates.

2.2.3 Running the container

With the image built and ready, you can now run the container with the following command:

```
$ docker run --name kiada-container -p 1234:8080 -d kiada
9d62e8a9c37e056a82bb1efad57789e947df58669f94adc2006c087a03c54e02
```

This tells Docker to run a new container called `kiada-container` from the `kiada` image. The container is detached from the console (`-d` flag) and runs in the background. Port `1234` on the

host computer is mapped to port `8080` in the container (specified by the `-p 1234:8080` option), so you can access the app at http://localhost:1234.

The following figure should help you visualize how everything fits together. Note that the Linux VM exists only if you use macOS or Windows. If you use Linux directly, there is no VM and the box depicting port `1234` is at the edge of the local computer.

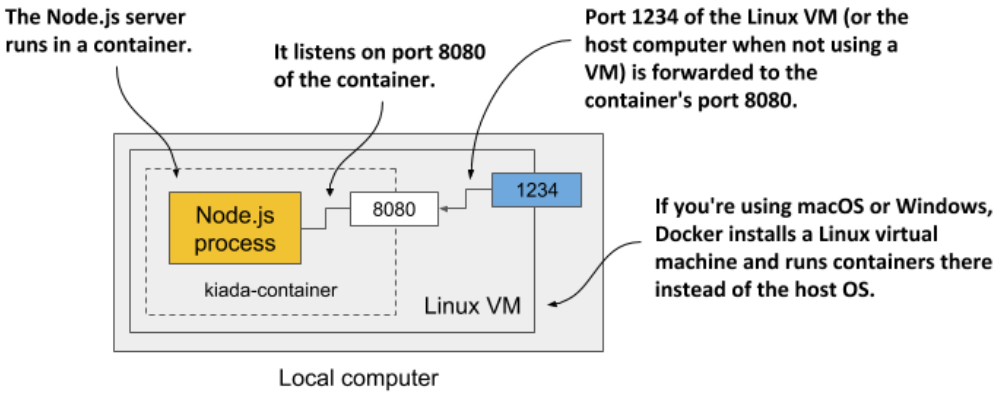

Figure 2.15 Visualizing your running container

ACCESSING YOUR APP

Now access the application at http://localhost:1234 using `curl` or your internet browser:

```
$ curl localhost:1234
Kiada version 0.1. Request processed by "44d76963e8e1". Client IP: ::ffff:172.17.0.1
```

> **NOTE** If the Docker Daemon runs on a different machine, you must replace `localhost` with the IP of that machine. You can look it up in the `DOCKER_HOST` environment variable.

If all went well, you should see the response sent by the application. In my case, it returns `44d76963e8e1` as its hostname. In your case, you'll see a different hexadecimal number. That's the ID of the container. You'll also see it displayed when you list the running containers next.

LISTING ALL RUNNING CONTAINERS

To list all the containers that are running on your computer, run the following command. Its output has been edited to make it more readable—the last two lines of the output are the continuation of the first two.

```
$ docker ps
CONTAINER ID    IMAGE           COMMAND          CREATED        ...
44d76963e8e1    kiada:latest    "node app.js"    6 minutes ago  ...

...  STATUS          PORTS                     NAMES
...  Up 6 minutes    0.0.0.0:1234->8080/tcp    kiada-container
```

For each container, Docker prints its ID and name, the image it uses, and the command it executes. It also shows when the container was created, what status it has, and which host ports are mapped to the container.

GETTING ADDITIONAL INFORMATION ABOUT A CONTAINER

The `docker ps` command shows the most basic information about the containers. To see additional information, you can use `docker inspect`:

```
$ docker inspect kiada-container
```

Docker prints a long JSON-formatted document containing a lot of information about the container, such as its state, config, and network settings, including its IP address.

INSPECTING THE APPLICATION LOG

Docker captures and stores everything the application writes to the standard output and error streams. This is typically the place where applications write their logs. You can use the `docker logs` command to see the output:

```
$ docker logs kiada-container
Kiada - Kubernetes in Action Demo Application
---------------------------------------------
Kiada 0.1 starting...
Local hostname is 44d76963e8e1
Listening on port 8080
Received request for / from ::ffff:172.17.0.1
```

You now know the basic commands for executing and inspecting an application in a container. Next, you'll learn how to distribute it.

2.2.4 Distributing the container image

The image you've built is only available locally. To run it on other computers, you must first push it to an external image registry. Let's push it to the public Docker Hub registry, so that you don't need to set up a private one. You can also use other registries, such as Quay.io, which I've already mentioned, or the Google Container Registry.

Before you push the image, you must re-tag it according to Docker Hub's image naming schema. The image name must include your Docker Hub ID, which you choose when you register at http://hub.docker.com. I'll use my own ID (`luksa`) in the following examples, so remember to replace it with your ID when trying the commands yourself.

TAGGING AN IMAGE UNDER AN ADDITIONAL TAG

Once you have your ID, you're ready to add an additional tag `for` your image. Its current name is `kiada` and you'll now tag it also as `yourid/kiada:0.1` (replace `yourid` with your actual Docker Hub ID). This is the command I used:

```
$ docker tag kiada luksa/kiada:0.1
```

Run `docker images` again to confirm that your image now has two names :

```
$ docker images
REPOSITORY     TAG       IMAGE ID        CREATED              VIRTUAL SIZE
luksa/kiada    1.0       b0ecc49d7a1d    About an hour ago    908 MB
kiada          latest    b0ecc49d7a1d    About an hour ago    908 MB
node           12        e498dabfee1c    3 days ago           908 MB
...
```

As you can see, both `kiada` and `luksa/kiada:0.1` point to the same image ID, meaning that these aren't two images, but a single image with two names.

PUSHING THE IMAGE TO DOCKER HUB

Before you can push the image to Docker Hub, you must log in with your user ID using the `docker login` command as follows:

```
$ docker login -u yourid -p yourpassword docker.io
```

Once logged in, push the `yourid/kiada:0.1` image to Docker Hub with the following command:

```
$ docker push yourid/kiada:0.1
```

RUNNING THE IMAGE ON OTHER HOSTS

When the push to Docker Hub is complete, the image is available to all. You can now run the image on any Docker-enabled host by running the following command:

```
$ docker run -p 1234:8080 -d luksa/kiada:0.1
```

If the container runs correctly on your computer, it should run on any other Linux computer, provided that the Node.js binary doesn't need any special Kernel features (it doesn't).

2.2.5 Stopping and deleting the container

If you've run the container on the other host, you can now terminate it, as you'll only need the one on your local computer for the exercises that follow.

STOPPING A CONTAINER

Instruct Docker to stop the container with this command:

```
$ docker stop kiada-container
```

This sends a termination signal to the main process in the container so that it can shut down gracefully. If the process doesn't respond to the termination signal or doesn't shut down in time, Docker kills it. When the top-level process in the container terminates, no other process runs in the container, so the container is stopped.

DELETING A CONTAINER

The container is no longer running, but it still exists. Docker keeps it around in case you decide to start it again. You can see stopped containers by running `docker ps -a`. The `-a` option prints all the containers - those running and those that have been stopped. As an exercise, you can start the container again by running `docker start kiada-container`.

You can safely delete the container on the other host, because you no longer need it. To delete it, run the following `docker rm` command:

```
$ docker rm kiada-container
```

This deletes the container. All its contents are removed and it can no longer be started. The image is still there, though. If you decide to create the container again, the image won't need to be downloaded again. If you also want to delete the image, use the `docker rmi` command:

```
$ docker rmi kiada:latest
```

To remove all dangling images, you can also use the `docker image prune` command.

2.3 Understanding containers

You should keep the container running on your local computer so that you can use it in the following exercises, in which you'll examine how containers allow process isolation without using virtual machines. Several features of the Linux kernel make this possible and it's time to get to know them.

2.3.1 Using Namespaces to customize the environment of a process

The first feature called *Linux Namespaces* ensures that each process has its own view of the system. This means that a process running in a container will only see some of the files, processes and network interfaces on the system, as well as a different system hostname, just as if it were running in a separate virtual machine.

Initially, all the system resources available in a Linux OS, such as filesystems, process IDs, user IDs, network interfaces, and others, are all in the same bucket that all processes see and use. But the Kernel allows you to create additional buckets known as namespaces and move resources into them so that they are organized in smaller sets. This allows you to make each set visible only to one process or a group of processes. When you create a new process, you can specify which namespace it should use. The process only sees resources that are in this namespace and none in the other namespaces.

INTRODUCING THE AVAILABLE NAMESPACE TYPES

More specifically, there isn't just a single type of namespace. There are in fact several types – one for each resource type. A process thus uses not only one namespace, but one namespace for each type.

The following types of namespaces exist:

- The Mount namespace (mnt) isolates mount points (file systems).
- The Process ID namespace (pid) isolates process IDs.
- The Network namespace (net) isolates network devices, stacks, ports, etc.
- The Inter-process communication namespace (ipc) isolates the communication between processes (this includes isolating message queues, shared memory, and others).
- The UNIX Time-sharing System (UTS) namespace isolates the system hostname and the Network Information Service (NIS) domain name.
- The User ID namespace (user) isolates user and group IDs.

- The Cgroup namespace isolates the Control Groups root directory. You'll learn about cgroups later in this chapter.

USING NETWORK NAMESPACES TO GIVE A PROCESS A DEDICATED SET OF NETWORK INTERFACES

The network namespace in which a process runs determines what network interfaces the process can see. Each network interface belongs to exactly one namespace but can be moved from one namespace to another. If each container uses its own network namespace, each container sees its own set of network interfaces.

Examine the following figure for a better overview of how network namespaces are used to create a container. Imagine you want to run a containerized process and provide it with a dedicated set of network interfaces that only that process can use.

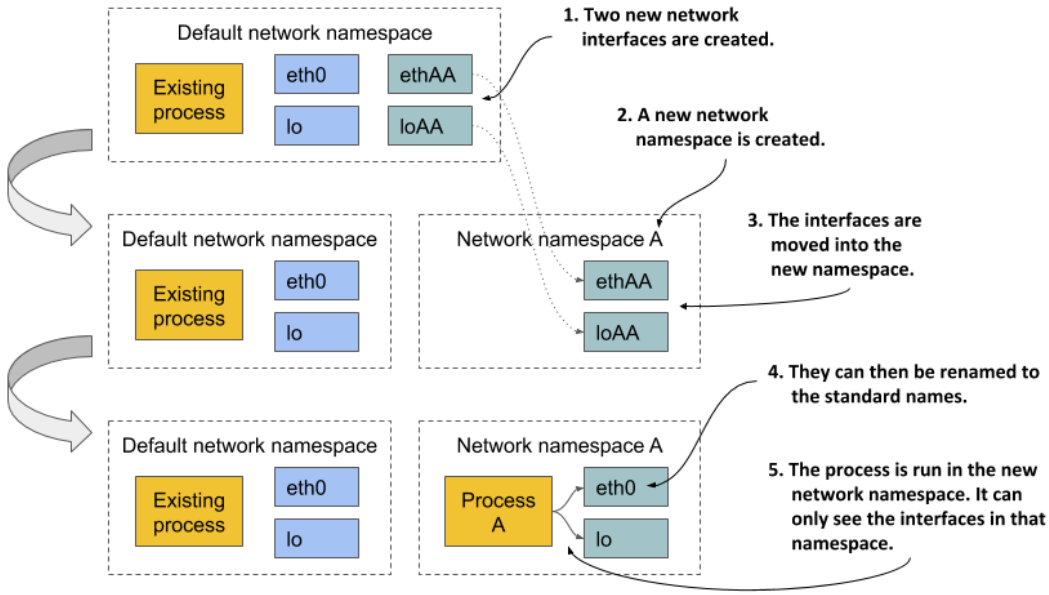

Figure 2.16 The network namespace limits which network interfaces a process uses

Initially, only the default network namespace exists. You then create two new network interfaces for the container and a new network namespace. The interfaces can then be moved from the default namespace to the new namespace. Once there, they can be renamed, because names must only be unique in each namespace. Finally, the process can be started in this network namespace, which allows it to only see the two interfaces that were created for it.

By looking solely at the available network interfaces, the process can't tell whether it's in a container or a VM or an OS running directly on a bare-metal machine.

USING THE UTS NAMESPACE TO GIVE A PROCESS A DEDICATED HOSTNAME

Another example of how to make it look like the process is running on its own host is to use the UTS namespace. It determines what hostname and domain name the process running inside this namespace sees. By assigning two different UTS namespaces to two different processes, you can make them see different system hostnames. To the two processes, it looks as if they run on two different computers.

UNDERSTANDING HOW NAMESPACES ISOLATE PROCESSES FROM EACH OTHER

By creating a dedicated namespace instance for all available namespace types and assigning it to a process, you can make the process believe that it's running in its own OS. The main reason for this is that each process has its own environment. A process can only see and use the resources in its own namespaces. It can't use any in other namespaces. Likewise, other processes can't use its resources either. This is how containers isolate the environments of the processes that run within them.

SHARING NAMESPACES BETWEEN MULTIPLE PROCESSES

In the next chapter you'll learn that you don't always want to isolate the containers completely from each other. Related containers may want to share certain resources. The following figure shows an example of two processes that share the same network interfaces and the host and domain name of the system, but not the file system.

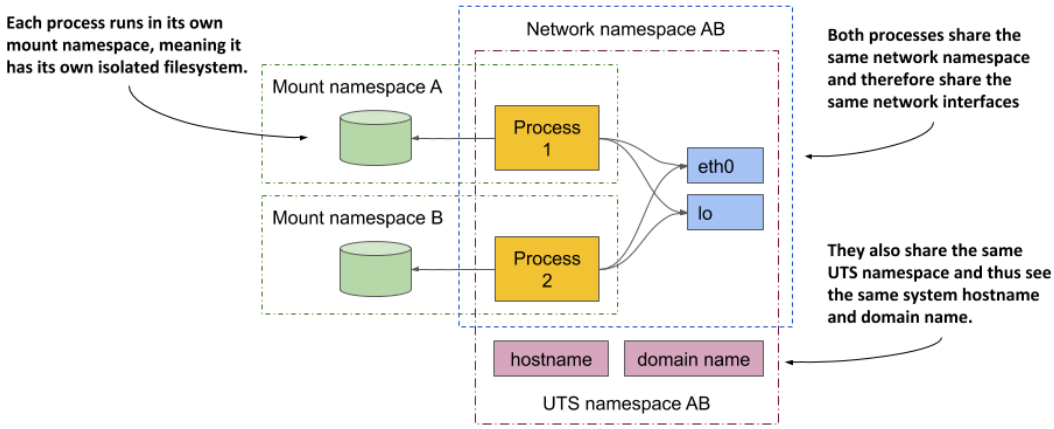

Figure 2.17 Each process is associated with multiple namespace types, some of which can be shared.

Concentrate on the shared network devices first. The two processes see and use the same two devices (`eth0` and `lo`) because they use the same network namespace. This allows them to bind to the same IP address and communicate through the loopback device, just as they could if they were running on a machine that doesn't use containers. The two processes also use the same UTS namespace and therefore see the same system host name. In contrast, they each use their own mount namespace, which means they have separate file systems.

In summary, processes may want to share some resources but not others. This is possible because separate namespace *types* exist. A process has an associated namespace for each type.

In view of all this, one might ask what is a container at all? A process that runs "in a container" doesn't run in something that resembles a real enclosure like a VM. It's only a process to which seven namespaces (one for each type) are assigned. Some are shared with other processes, while others are not. This means that the boundaries between the processes do not all fall on the same line.

In a later chapter, you'll learn how to debug a container by running a new process directly on the host OS, but using the network namespace of an existing container, while using the host's default namespaces for everything else. This will allow you to debug the container's networking system with tools available on the host that may not be available in the container.

2.3.2 Exploring the environment of a running container

What if you want to see what the environment inside the container looks like? What is the system host name, what is the local IP address, what binaries and libraries are available on the file system, and so on?

To explore these features in the case of a VM, you typically connect to it remotely via ssh and use a shell to execute commands. With containers, you run a shell in the container.

> **NOTE** The shell's executable file must be present in the container's file system. This isn't always the case with containers running in production.

RUNNING A SHELL INSIDE AN EXISTING CONTAINER

The Node.js image on which your image is based provides the bash shell, meaning you can run it in the container with the following command:

```
$ docker exec -it kiada-container bash
root@44d76963e8e1:/#    #A
```

#A This is the shell's command prompt

This command runs `bash` as an additional process in the existing `kiada-container` container. The process has the same Linux namespaces as the main container process (the running Node.js server). This way you can explore the container from within and see how Node.js and your app see the system when running in the container. The `-it` option is shorthand for two options:

- `-i` tells Docker to run the command in interactive mode.
- `-t` tells it to allocate a pseudo terminal (TTY) so you can use the shell properly.

You need both if you want to use the shell the way you're used to. If you omit the first, you can't execute any commands, and if you omit the second, the command prompt doesn't appear and some commands may complain that the `TERM` variable is not set.

LISTING RUNNING PROCESSES IN A CONTAINER

Let's**Error! Bookmark not defined.** list the processes running in the container by executing the `ps aux` command inside the shell that you ran in the container:

```
root@44d76963e8e1:/# ps aux
USER  PID %CPU %MEM    VSZ   RSS TTY STAT START TIME COMMAND
root    1  0.0  0.1 676380 16504 ?   Sl   12:31 0:00 node app.js
root   10  0.0  0.0  20216  1924 ?   Ss   12:31 0:00 bash
root   19  0.0  0.0  17492  1136 ?   R+   12:38 0:00 ps aux
```

The list shows only three processes. These are the only ones that run in the container. You can't see the other processes that run in the host OS or in other containers because the container runs in its own Process ID namespace.

SEEING CONTAINER PROCESSES IN THE HOST'S LIST OF PROCESSES

If you now open another terminal and list the processes in the host OS itself, you *will* also see the processes that run in the container. This will confirm that the processes in the container are in fact regular processes that run in the host OS. Here's the command and its output:

```
$ ps aux | grep app.js
USER  PID %CPU %MEM    VSZ   RSS TTY STAT START TIME COMMAND
root  382  0.0  0.1 676380 16504 ?   Sl   12:31 0:00 node app.js
```

> **NOTE** If you use macOS or Windows, you must list the processes in the VM that hosts the Docker daemon, as that's where your containers run. In Docker Desktop, you can enter the VM using the following command: `docker run --net=host --ipc=host --uts=host --pid=host -it --security-opt=seccomp=unconfined --privileged --rm -v /:/host alpine chroot /host`

If you have a sharp eye, you may notice that the process IDs in the container are different from those on the host. Because the container uses its own Process ID namespace it has its own process tree with its own ID number sequence. As the next figure shows, the tree is a subtree of the host's full process tree. Each process thus has two IDs.

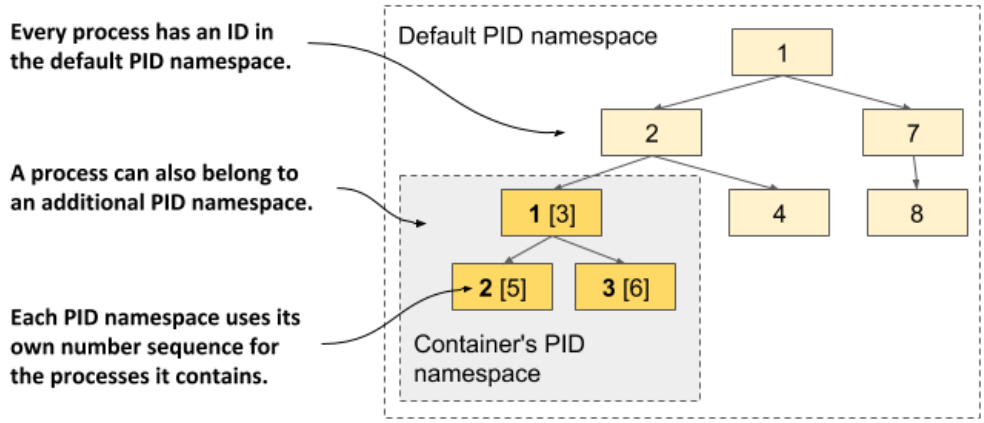

Figure 2.18 The PID namespace makes a process sub-tree appear as a separate process tree with its own numbering sequence

THE CONTAINER'S FILESYSTEM IS ISOLATED FROM THE HOST AND OTHER CONTAINERS

As with an isolated process tree, each container also has an isolated filesystem. If you list the contents of the container's root directory, only the files in the container are displayed. This includes files from the container image and files created during container operation, such as log files. The next listing shows the files in the `kiada` container's root file directory:

```
root@44d76963e8e1:/# ls /
app.js  boot  etc   lib    media  opt   root  sbin  sys  usr
bin     dev   home  lib64  mnt    proc  run   srv   tmp  var
```

It contains the `app.js` file and other system directories that are part of the `node:12` base image. You are welcome to browse the container's filesystem. You'll see that there is no way to view files from the host's filesystem. This is great, because it prevents a potential attacker from gaining access to them through vulnerabilities in the Node.js server.

To leave the container, leave the shell by running the `exit` command or pressing Control-D and you'll be returned to your host computer (similar to logging out from an `ssh` session).

> **TIP** Entering a running container like this is useful when debugging an app running in a container. When something breaks, the first thing you'll want to investigate is the actual state of the system your application sees.

2.3.3 Limiting a process' resource usage with Linux Control Groups

Linux Namespaces make it possible for processes to access only some of the host's resources, but they don't limit how much of a single resource each process can consume. For example, you can use namespaces to allow a process to access only a particular network interface, but you can't limit the network bandwidth the process consumes. Likewise, you can't use

namespaces to limit the CPU time or memory available to a process. You may want to do that to prevent one process from consuming all the CPU time and preventing critical system processes from running properly. For that, we need another feature of the Linux kernel.

INTRODUCING CGROUPS

The second Linux kernel feature that makes containers possible is called *Linux Control Groups (cgroups)*. It limits, accounts for and isolates system resources such as CPU, memory and disk or network bandwidth. When using cgroups, a process or group of processes can only use the allotted CPU time, memory, and network bandwidth for example. This way, processes cannot occupy resources that are reserved for other processes.

At this point, you don't need to know how Control Groups do all this, but it may be worth seeing how you can ask Docker to limit the amount of CPU and memory a container can use.

LIMITING A CONTAINER'S USE OF THE CPU

If you don't impose any restrictions on the container's use of the CPU, it has unrestricted access to all CPU cores on the host. You can explicitly specify which cores a container can use with Docker's `--cpuset-cpus` option. For example, to allow the container to only use cores one and two, you can run the container with the following option:

```
$ docker run --cpuset-cpus="1,2" ...
```

You can also limit the available CPU time using options `--cpus`, `--cpu-period`, `--cpu-quota` and `--cpu-shares`. For example, to allow the container to use only half of a CPU core, run the container as follows:

```
$ docker run --cpus="0.5" ...
```

LIMITING A CONTAINER'S USE OF MEMORY

As with CPU, a container can use all the available system memory, just like any regular OS process, but you may want to limit this. Docker provides the following options to limit container memory and swap usage: `--memory`, `--memory-reservation`, `--kernel-memory`, `--memory-swap`, and `--memory-swappiness`.

For example, to set the maximum memory size available in the container to 100MB, run the container as follows (`m` stands for megabyte):

```
$ docker run --memory="100m" ...
```

Behind the scenes, all these Docker options merely configure the cgroups of the process. It's the Kernel that takes care of limiting the resources available to the process. See the Docker documentation for more information about the other memory and CPU limit options.

2.3.4 Strengthening isolation between containers

Linux Namespaces and Cgroups separate the containers' environments and prevent one container from starving the other containers of compute resources. But the processes in these containers use the same system kernel, so we can't say that they are really isolated. A rogue container could make malicious system calls that would affect its neighbours.

Imagine a Kubernetes node on which several containers run. Each has its own network devices and files and can only consume a limited amount of CPU and memory. At first glance, a rogue program in one of these containers can't cause damage to the other containers. But what if the rogue program modifies the system clock that is shared by all containers?

Depending on the application, changing the time may not be too much of a problem, but allowing programs to make any system call to the kernel allows them to do virtually anything. Sys-calls allow them to modify the kernel memory, add or remove kernel modules, and many other things that regular containers aren't supposed to do.

This brings us to the third set of technologies that make containers possible. Explaining them fully is outside the scope of this book, so please refer to other resources that focus specifically on containers or the technologies used to secure them. This section provides a brief introduction to these technologies.

GIVING CONTAINERS FULL PRIVILEGES TO THE SYSTEM

The operating system kernel provides a set of *sys-calls* that programs use to interact with the operating system and underlying hardware. These includes calls to create processes, manipulate files and devices, establish communication channels between applications, and so on.

Some of these sys-calls are safe and available to any process, but others are reserved for processes with elevated privileges only. If you look at the example presented earlier, applications running on the Kubernetes node should be allowed to open their local files, but not change the system clock or modify the kernel in a way that breaks the other containers.

Most containers should run without elevated privileges. Only those programs that you trust and that actually need the additional privileges should run in privileged containers.

> **NOTE** With Docker you create a privileged container by using the `--privileged` flag.

USING CAPABILITIES TO GIVE CONTAINERS A SUBSET OF ALL PRIVILEGES

If an application only needs to invoke some of the sys-calls that require elevated privileges, creating a container with full privileges is not ideal. Fortunately, the Linux kernel also divides privileges into units called *capabilities*. Examples of capabilities are:

- `CAP_NET_ADMIN` allows the process to perform network-related operations,
- `CAP_NET_BIND_SERVICE` allows it to bind to port numbers less than 1024,
- `CAP_SYS_TIME` allows it to modify the system clock, and so on.

Capabilities can be added or removed (*dropped*) from a container when you create it. Each capability represents a set of privileges available to the processes in the container. Docker and Kubernetes drop all capabilities except those required by typical applications, but users can add or drop other capabilities if authorized to do so.

> **NOTE** Always follow the *principle of least privilege* when running containers. Don't give them any capabilities that they don't need. This prevents attackers from using them to gain access to your operating system.

USING SECCOMP PROFILES TO FILTER INDIVIDUAL SYS-CALLS

If you need even finer control over what sys-calls a program can make, you can use *seccomp* (Secure Computing Mode). You can create a custom seccomp profile by creating a JSON file that lists the sys-calls that the container using the profile is allowed to make. You then provide the file to Docker when you create the container.

HARDENING CONTAINERS USING APPARMOR AND SELINUX

And as if the technologies discussed so far weren't enough, containers can also be secured with two additional mandatory access control (MAC) mechanisms: SELinux (Security-Enhanced Linux) and AppArmor (Application Armor).

With SELinux, you attach labels to files and system resources, as well as to users and processes. A user or process can only access a file or resource if the labels of all subjects and objects involved match a set of policies. AppArmor is similar but uses file paths instead of labels and focuses on processes rather than users.

Both SELinux and AppArmor considerably improve the security of an operating system, but don't worry if you are overwhelmed by all these security-related mechanisms. The aim of this section was to shed light on everything involved in the proper isolation of containers, but a basic understanding of namespaces should be more than sufficient for the moment.

2.4 Summary

If you were new to containers before reading this chapter, you should now understand what they are, why we use them, and what features of the Linux kernel make them possible. If you have previously used containers, I hope this chapter has helped to clarify your uncertainties about how containers work, and you now understand that they're nothing more than regular OS processes that the Linux kernel isolates from other processes.

After reading this chapter, you should know that:

- Containers are regular processes, but isolated from each other and the other processes running in the host OS.
- Containers are much lighter than virtual machines, but because they use the same Linux kernel, they are not as isolated as VMs.
- Docker was the first container platform to make containers popular and the first container runtime supported by Kubernetes. Now, others are supported through the Container Runtime Interface (CRI).
- A container image contains the user application and all its dependencies. It is distributed through a container registry and used to create running containers.
- Containers can be downloaded and executed with a single `docker run` command.
- Docker builds an image from a `Dockerfile` that contains commands that Docker should execute during the build process. Images consist of layers that can be shared between multiple images. Each layer only needs to be transmitted and stored once.
- Containers are isolated by Linux kernel features called Namespaces, Control groups, Capabilities, seccomp, AppArmor and/or SELinux. Namespaces ensure that a container sees only a part of the resources available on the host, Control groups limit the amount of a resource it can use, while other features strengthen the isolation between containers.

After inspecting the containers on this ship, you're now ready to raise the anchor and sail into the next chapter, where you'll learn about running containers with Kubernetes.

3

Deploying your first application

This chapter covers

- Running a single-node Kubernetes cluster on your laptop
- Setting up a Kubernetes cluster on Google Kubernetes Engine
- Setting up and using the `kubectl` command-line tool
- Deploying an application in Kubernetes and making it available across the globe
- Horizontally scaling the application

The goal of this chapter is to show you how to run a local single-node development Kubernetes cluster or set up a proper, managed multi-node cluster in the cloud. Once your cluster is running, you'll use it to run the container you created in the previous chapter.

> **NOTE** You'll find the code files for this chapter at https://github.com/luksa/kubernetes-in-action-2nd-edition/tree/master/Chapter03

3.1 Deploying a Kubernetes cluster

Setting up a full-fledged, multi-node Kubernetes cluster isn't a simple task, especially if you're not familiar with Linux and network administration. A proper Kubernetes installation spans multiple physical or virtual machines and requires proper network setup to allow all containers in the cluster to communicate with each other.

You can install Kubernetes on your laptop computer, on your organization's infrastructure, or on virtual machines provided by cloud providers (Google Compute Engine, Amazon EC2, Microsoft Azure, and so on). Alternatively, most cloud providers now offer managed Kubernetes services, saving you from the hassle of installation and management. Here's a short overview of what the largest cloud providers offer:

- Google offers GKE - Google Kubernetes Engine,
- Amazon has EKS - Amazon Elastic Kubernetes Service,
- Microsoft has AKS – Azure Kubernetes Service,
- IBM has IBM Cloud Kubernetes Service,
- Alibaba provides the Alibaba Cloud Container Service.

Installing and managing Kubernetes is much more difficult than just using it, especially until you're intimately familiar with its architecture and operation. For this reason, we'll start with the easiest ways to obtain a working Kubernetes cluster. You'll learn several ways to run a single-node Kubernetes cluster on your local computer and how to use a hosted cluster running on Google Kubernetes Engine (GKE).

A third option, which involves installing a cluster using the `kubeadm` tool, is explained in Appendix B. The tutorial there will show you how to set up a three-node Kubernetes cluster using virtual machines. But you may want to try that only after you've become familiar with using Kubernetes. Many other options also exist, but they are beyond the scope of this book. Refer to the kubernetes.io website to learn more.

If you've been granted access to an existing cluster deployed by someone else, you can skip this section and go on to section 3.2 where you'll learn how to interact with Kubernetes clusters.

3.1.1 Using the built-in Kubernetes cluster in Docker Desktop

If you use macOS or Windows, you've most likely installed Docker Desktop to run the exercises in the previous chapter. It contains a single-node Kubernetes cluster that you can enable via its Settings dialog box. This may be the easiest way for you to start your Kubernetes journey, but keep in mind that the version of Kubernetes may not be as recent as when using the alternative options described in the next sections.

> **NOTE** Although technically not a cluster, the single-node Kubernetes system provided by Docker Desktop should be enough to explore most of the topics discussed in this book. When an exercise requires a multi-node cluster, I will point this out.

ENABLING KUBERNETES IN DOCKER DESKTOP

Assuming Docker Desktop is already installed on your computer, you can start the Kubernetes cluster by clicking the whale icon in the system tray and opening the Settings dialog box. Click the *Kubernetes* tab and make sure the *Enable Kubernetes* checkbox is selected. The components that make up the Control Plane run as Docker containers, but they aren't displayed in the list of running containers when you invoke the `docker ps` command. To display them, select the *Show system containers* checkbox.

> **NOTE** The initial installation of the cluster takes several minutes, as all container images for the Kubernetes components must be downloaded.

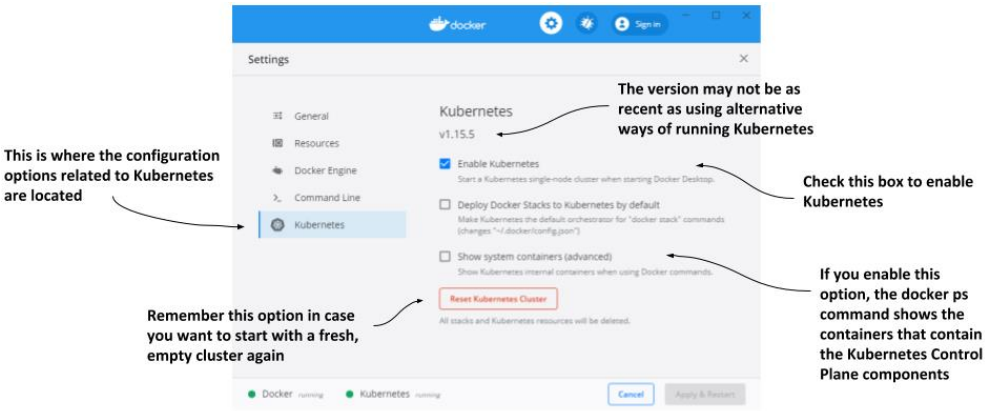

Figure 3.1 The Settings dialog box in Docker Desktop for Windows

Remember the *Reset Kubernetes Cluster* button if you ever want to reset the cluster to remove all the objects you've deployed in it.

VISUALIZING THE SYSTEM

To understand where the various components that make up the Kubernetes cluster run in Docker Desktop, look at the following figure.

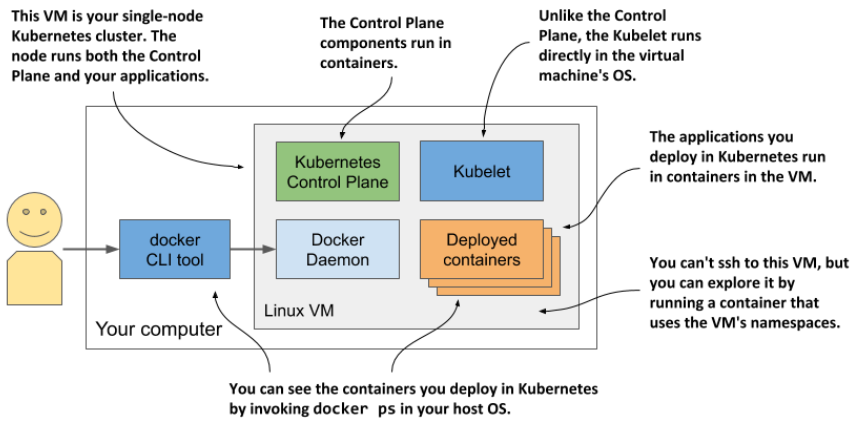

Figure 3.2 Kubernetes running in Docker Desktop

Docker Desktop sets up a Linux virtual machine that hosts the Docker Daemon and all the containers. This VM also runs the Kubelet - the Kubernetes agent that manages the node. The components of the Control Plane run in containers, as do all the applications you deploy.

To list the running containers, you don't need to log on to the VM because the docker CLI tool available in your host OS displays them.

EXPLORING THE VIRTUAL MACHINE FROM THE INSIDE

At the time of writing, Docker Desktop provides no command to log into the VM if you want to explore it from the inside. However, you can run a special container configured to use the VM's namespaces to run a remote shell, which is virtually identical to using SSH to access a remote server. To run the container, execute the following command:

```
$ docker run --net=host --ipc=host --uts=host --pid=host --privileged \
  --security-opt=seccomp=unconfined -it --rm -v /:/host alpine chroot /host
```

This long command requires explanation:

- The container is created from the `alpine` image.
- The `--net`, `--ipc`, `--uts` and `--pid` flags make the container use the host's namespaces instead of being sandboxed, and the `--privileged` and `--security-opt` flags give the container unrestricted access to all sys-calls.
- The `-it` flag runs the container interactive mode and the `--rm` flags ensures the container is deleted when it terminates.
- The `-v` flag mounts the host's root directory to the `/host` directory in the container. The `chroot /host` command then makes this directory the root directory in the container.

After you run the command, you are in a shell that's effectively the same as if you had used SSH to enter the VM. Use this shell to explore the VM - try listing processes by executing the `ps aux` command or explore the network interfaces by running `ip addr`.

3.1.2 Running a local cluster using Minikube

Another way to create a Kubernetes cluster is to use *Minikube*, a tool maintained by the Kubernetes community. The version of Kubernetes that Minikube deploys is usually more recent than the version deployed by Docker Desktop. The cluster consists of a single node and is suitable for both testing Kubernetes and developing applications locally. It normally runs Kubernetes in a Linux VM, but if your computer is Linux-based, it can also deploy Kubernetes directly in your host OS via Docker.

> **NOTE** If you configure Minikube to use a VM, you don't need Docker, but you do need a hypervisor like VirtualBox. In the other case you need Docker, but not the hypervisor.

INSTALLING MINIKUBE

Minikube supports macOS, Linux, and Windows. It has a single binary executable file, which you'll find in the Minikube repository on GitHub (http://github.com/kubernetes/minikube). It's best to follow the current installation instructions published there, but roughly speaking, you install it as follows.

On macOS you can install it using the Brew Package Manager, on Windows there's an installer that you can download, and on Linux you can either download a .deb or .rpm package or simply download the binary file and make it executable with the following command:

```
$ curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube
[CA] -linux-amd64 && sudo install minikube-linux-amd64 /usr/local/bin/mini
[CA] kube
```

For details on your specific OS, please refer to the installation guide online.

STARTING A KUBERNETES CLUSTER WITH MINIKUBE

After Minikube is installed, start the Kubernetes cluster as shown next:

```
$ minikube start
minikube v1.11.0 on Fedora 31
Using the virtualbox driver based on user configuration
Downloading VM boot image ...
> minikube-v1.11.0.iso.sha256: 65 B / 65 B [-------------] 100.00% ? p/s 0s
> minikube-v1.11.0.iso: 174.99 MiB / 174.99 MiB [] 100.00% 50.16 MiB p/s 4s
Starting control plane node minikube in cluster minikube
Downloading Kubernetes v1.18.3 preload ...
> preloaded-images-k8s-v3-v1.18.3-docker-overlay2-amd64.tar.lz4: 526.01 MiB
Creating virtualbox VM (CPUs=2, Memory=6000MB, Disk=20000MB) ...
Preparing Kubernetes v1.18.3 on Docker 19.03.8 ...
Verifying Kubernetes components...
Enabled addons: default-storageclass, storage-provisioner
Done! kubectl is now configured to use "minikube"
```

The process may take several minutes, because the VM image and the container images of the Kubernetes components must be downloaded.

> **TIP** If you use Linux, you can reduce the resources required by Minikube by creating the cluster without a VM. Use this command: `minikube start --vm-driver none`

CHECKING MINIKUBE'S STATUS

When the `minikube start` command is complete, you can check the status of the cluster by running the `minikube status` command:

```
$ minikube status
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

The output of the command shows that the Kubernetes host (the VM that hosts Kubernetes) is running, and so are the Kubelet – the agent responsible for managing the node – and the Kubernetes API server. The last line shows that the kubectl command-line tool (CLI) is configured to use the Kubernetes cluster that Minikube has provided. Minikube doesn't install the CLI tool, but it does create its configuration file. Installation of the CLI tool is explained in section 3.2.

VISUALIZING THE SYSTEM

The architecture of the system, which is shown in the next figure, is practically identical to the one in Docker Desktop.

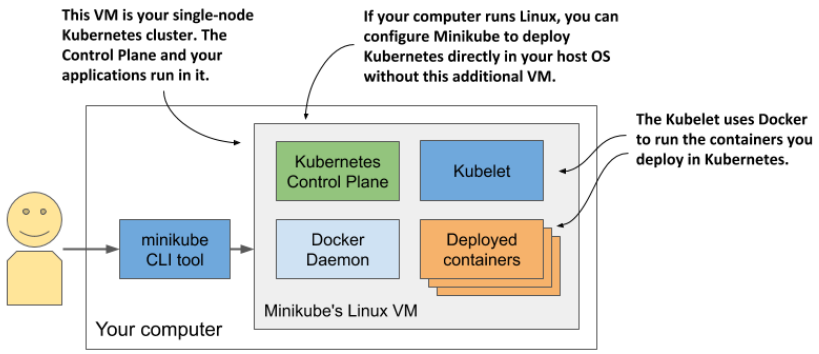

Figure 3.3 Running a single-node Kubernetes cluster using Minikube

The Control Plane components run in containers in the VM or directly in your host OS if you used the `--vm-driver none` option to create the cluster. The Kubelet runs directly in the VM's or your host's operating system. It runs the applications you deploy in the cluster via the Docker Daemon.

You can run `minikube ssh` to log into the Minikube VM and explore it from inside. For example, you can see what's running in the VM by running `ps aux` to list running processes or `docker ps` to list running containers.

> **TIP** If you want to list containers using your local docker CLI instance, as in the case of Docker Desktop, run the following command: `eval $(minikube docker-env)`

3.1.3 Running a local cluster using kind (Kubernetes in Docker)

An alternative to Minikube, although not as mature, is *kind* (Kubernetes-in-Docker). Instead of running Kubernetes in a virtual machine or directly on the host, kind runs each Kubernetes cluster node inside a container. Unlike Minikube, this allows it to create multi-node clusters by starting several containers. The actual application containers that you deploy to Kubernetes then run within these node containers. The system is shown in the next figure.

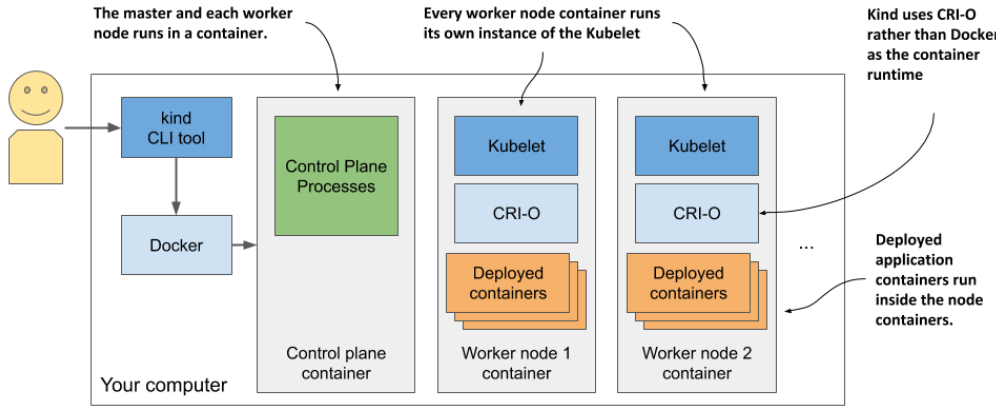

Figure 3.4 Running a multi-node Kubernetes cluster using kind

In the previous chapter I mentioned that a process that runs in a container actually runs in the host OS. This means that when you run Kubernetes using kind, all Kubernetes components run in your host OS. The applications you deploy to the Kubernetes cluster also run in your host OS.

This makes kind the perfect tool for development and testing, as everything runs locally and you can debug running processes as easily as when you run them outside of a container. I prefer to use this approach when I develop apps on Kubernetes, as it allows me to do magical things like run network traffic analysis tools such as Wireshark or even my web browser inside the containers that run my applications. I use a tool called `nsenter` that allows me to run these tools in the network or other namespaces of the container.

If you're new to Kubernetes, the safest bet is to start with Minikube, but if you're feeling adventurous, here's how to get started with kind.

INSTALLING KIND

Just like Minikube, kind consists of a single binary executable file. To install it, refer to the installation instructions at https://kind.sigs.k8s.io/docs/user/quick-start/. On macOS and Linux, the command to install it is as follows:

```
$ curl -Lo ./kind https://github.com/kubernetes-sigs/kind/releases/
[CA] download/v0.7.0/kind-$(uname)-amd64 && \
[CA] chmod +x ./kind && \
[CA] mv ./kind /some-dir-in-your-PATH/kind
```

Check the documentation to see what the latest version is and use it instead of v0.7.0 in the above example. Also, replace `/some-dir-in-your-PATH/` with an actual directory in your path.

NOTE Docker must be installed on your system to use kind.

STARTING A KUBERNETES CLUSTER WITH KIND

Starting a new cluster is as easy as it is with Minikube. Execute the following command:

```
$ kind create cluster
```

Like Minikube, kind configures kubectl to use the cluster that it creates.

STARTING A MULTI-NODE CLUSTER WITH KIND

Kind runs a single-node cluster by default. If you want to run a cluster with multiple worker nodes, you must first create a configuration file. The following listing shows the contents of this file (Chapter03/kind-multi-node.yaml).

Listing 3.1 Config file for running a three-node cluster with the kind tool

```
kind: Cluster
apiVersion: kind.sigs.k8s.io/v1alpha3
nodes:
- role: control-plane
- role: worker
- role: worker
```

With the file in place, create the cluster using the following command:

```
$ kind create cluster --config kind-multi-node.yaml
```

LISTING WORKER NODES

At the time of this writing, kind doesn't provide a command to check the status of the cluster, but you can list cluster nodes using `kind get nodes`:

```
$ kind get nodes
kind-worker2
kind-worker
kind-control-plane
```

Since each node runs as a container, you can also see the nodes by listing the running containers using `docker ps`:

```
$ docker ps
CONTAINER ID    IMAGE                   ...    NAMES
45d0f712eac0    kindest/node:v1.18.2    ...    kind-worker2
d1e88e98e3ae    kindest/node:v1.18.2    ...    kind-worker
4b7751144ca4    kindest/node:v1.18.2    ...    kind-control-plane
```

LOGGING INTO CLUSTER NODES PROVISIONED BY KIND

Unlike Minikube, where you use `minikube ssh` to log into the node if you want to explore the processes running inside it, with kind you use `docker exec`. For example, to enter the node called `kind-control-plane`, run:

```
$ docker exec -it kind-control-plane bash
```

Instead of using Docker to run containers, nodes created by kind use the CRI-O container runtime, which I mentioned in the previous chapter as a lightweight alternative to Docker. The `crictl` CLI tool is used to interact with CRI-O. Its use is very similar to that of the `docker` tool. After logging into the node, list the containers running in it by running `crictl ps` instead of `docker ps`. Here's an example of the command and its output:

```
root@kind-control-plane:/# crictl ps
CONTAINER ID    IMAGE           CREATED      STATE     NAME
c7f44d171fb72   eb516548c180f   15 min ago   Running   coredns         ...
cce9c0261854c   eb516548c180f   15 min ago   Running   coredns         ...
e6522aae66fcc   d428039608992   16 min ago   Running   kube-proxy      ...
6b2dc4bbfee0c   ef97cccdfdb50   16 min ago   Running   kindnet-cni     ...
c3e66dfe44deb   be321f2ded3f3   16 min ago   Running   kube-apiserver ...
```

3.1.4 Creating a managed cluster with Google Kubernetes Engine

If you want to use a full-fledged multi-node Kubernetes cluster instead of a local one, you can use a managed cluster, such as the one provided by Google Kubernetes Engine (GKE). This way, you don't have to manually set up all the cluster nodes and networking, which is usually too hard for someone taking their first steps with Kubernetes. Using a managed solution such as GKE ensures that you don't end up with an incorrectly configured cluster.

SETTING UP GOOGLE CLOUD AND INSTALLING THE GCLOUD CLIENT BINARY

Before you can set up a new Kubernetes cluster, you must set up your GKE environment. The process may change in the future, so I'll only give you a few general instructions here. For complete instructions, refer to https://cloud.google.com/container-engine/docs/before-you-begin.

Roughly, the whole procedure includes

1. Signing up for a Google account if you don't have one already.
2. Creating a project in the Google Cloud Platform Console.
3. Enabling billing. This does require your credit card info, but Google provides a 12-month free trial with a free $300 credit. And they don't start charging automatically after the free trial is over.
4. Downloading and installing the Google Cloud SDK, which includes the `gcloud` tool.
5. Creating the cluster using the `gcloud` command-line tool.

NOTE Certain operations (the one in step 2, for example) may take a few minutes to complete, so relax and grab a coffee in the meantime.

CREATING A GKE KUBERNETES CLUSTER WITH THREE NODES

Before you create your cluster, you must decide in which geographical region and zone it should be created. Refer to https://cloud.google.com/compute/docs/regions-zones to see the list of available locations. In the following examples, I use the europe-west3 region based in Frankfurt, Germany. It has three different zones - I'll use the zone europe-west3-c. The default zone for all gcloud operations can be set with the following command:

```
$ gcloud config set compute/zone europe-west3-c
```

Create the Kubernetes cluster like this:

```
$ gcloud container clusters create kiada --num-nodes 3
Creating cluster kiada in europe-west3-c...
...
kubeconfig entry generated for kiada.
NAME   LOCAT.   MASTER_VER  MASTER_IP   MACH_TYPE     ... NODES STATUS
kiada  eu-w3-c  1.13.11...  5.24.21.22  n1-standard-1 ... 3     RUNNING
```

NOTE I'm creating all three worker nodes in the same zone, but you can also spread them across all zones in the region by setting the `compute/zone` config value to an entire region instead of a single zone. If you do so, note that `--num-nodes` indicates the number of nodes *per zone*. If the region contains three zones and you only want three nodes, you must set `--num-nodes` to `1`.

You should now have a running Kubernetes cluster with three worker nodes. Each node is a virtual machine provided by the Google Compute Engine (GCE) infrastructure-as-a-service platform. You can list GCE virtual machines using the following command:

```
$ gcloud compute instances list
NAME      ZONE         MACHINE_TYPE   INTERNAL_IP  EXTERNAL_IP     STATUS
...-ctlk  eu-west3-c   n1-standard-1  10.156.0.16  34.89.238.55    RUNNING
...-gj1f  eu-west3-c   n1-standard-1  10.156.0.14  35.242.223.97   RUNNING
...-r01z  eu-west3-c   n1-standard-1  10.156.0.15  35.198.191.189  RUNNING
```

TIP Each VM incurs costs. To reduce the cost of your cluster, you can reduce the number of nodes to one, or even to zero while not using it. See next section for details.

The system is shown in the next figure. Note that only your worker nodes run in GCE virtual machines. The control plane runs elsewhere - you can't access the machines hosting it.

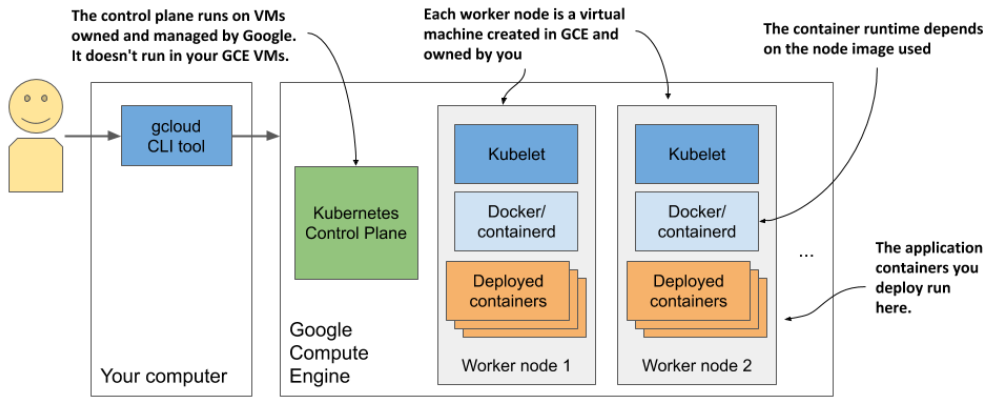

Figure 3.5 Your Kubernetes cluster in Google Kubernetes Engine

SCALING THE NUMBER OF NODES

Google allows you to easily increase or decrease the number of nodes in your cluster. For most exercises in this book you can scale it down to just one node if you want to save money. You can even scale it down to zero so that your cluster doesn't incur any costs.

To scale the cluster to zero, use the following command:

```
$ gcloud container clusters resize kiada --size 0
```

The nice thing about scaling to zero is that none of the objects you create in your Kubernetes cluster, including the applications you deploy, are deleted. Granted, if you scale down to zero, the applications will have no nodes to run on, so they won't run. But as soon as you scale the cluster back up, they will be redeployed. And even with no worker nodes you can still interact with the Kubernetes API (you can create, update, and delete objects).

INSPECTING A GKE WORKER NODE

If you're interested in what's running on your nodes, you can log into them with the following command (use one of the node names from the output of the previous command):

```
$ gcloud compute ssh gke-kiada-default-pool-9bba9b18-4glf
```

While logged into the node, you can try to list all running containers with `docker ps`. You haven't run any applications yet, so you'll only see Kubernetes system containers. What they are isn't important right now, but you'll learn about them in later chapters.

3.1.5 Creating a cluster using Amazon Elastic Kubernetes Service

If you prefer to use Amazon instead of Google to deploy your Kubernetes cluster in the cloud, you can try the Amazon Elastic Kubernetes Service (EKS). Let's go over the basics.

First, you have to install the `eksctl` command-line tool by following the instructions at https://docs.aws.amazon.com/eks/latest/userguide/getting-started-eksctl.html.

CREATING AN EKS KUBERNETES CLUSTER

Creating an EKS Kubernetes cluster using `eksctl` does not differ significantly from how you create a cluster in GKE. All you must do is run the following command:

```
$ eksctl create cluster --name kiada --region eu-central-1
[CA] --nodes 3 --ssh-access
```

This command creates a three-node cluster in the `eu-central-1` region. The regions are listed at https://aws.amazon.com/about-aws/global-infrastructure/regional-product-services/.

INSPECTING AN EKS WORKER NODE

If you're interested in what's running on those nodes, you can use SSH to connect to them. The `--ssh-access` flag used in the command that creates the cluster ensures that your SSH public key is imported to the node.

As with GKE and Minikube, once you've logged into the node, you can try to list all running containers with `docker ps`. You can expect to see similar containers as in the clusters we covered earlier.

3.1.6 Deploying a multi-node cluster from scratch

Until you get a deeper understanding of Kubernetes, I strongly recommend that you don't try to install a multi-node cluster from scratch. If you are an experienced systems administrator, you may be able to do it without much pain and suffering, but most people may want to try one of the methods described in the previous sections first. Proper management of Kubernetes clusters is incredibly difficult. The installation alone is a task not to be underestimated.

If you still feel adventurous, you can start with the instructions in Appendix B, which explain how to create VMs with VirtualBox and install Kubernetes using the kubeadm tool. You can also use those instructions to install Kubernetes on your bare-metal machines or in VMs running in the cloud.

Once you've successfully deployed one or two clusters using kubeadm, you can then try to deploy it completely manually, by following Kelsey Hightower's *Kubernetes the Hard Way* tutorial at github.com/kelseyhightower/Kubernetes-the-hard-way. Though you may run into several problems, figuring out how to solve them can be a great learning experience.

3.2 Interacting with Kubernetes

You've now learned about several possible methods to deploy a Kubernetes cluster. Now's the time to learn how to use the cluster. To interact with Kubernetes, you use a command-line tool called `kubectl`, pronounced *kube-control*, *kube-C-T-L* or *kube-cuddle*.

As the next figure shows, the tool communicates with the Kubernetes API server, which is part of the Kubernetes Control Plane. The control plane then triggers the other components to do whatever needs to be done based on the changes you made via the API.

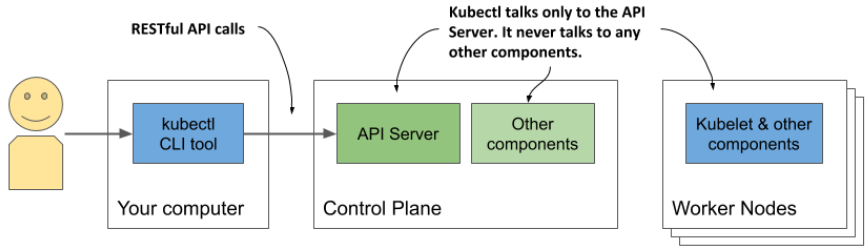

Figure 3.6 How you interact with a Kubernetes cluster

3.2.1 Setting up kubectl - the Kubernetes command-line client

Kubectl is a single executable file that you must download to your computer and place into your path. It loads its configuration from a configuration file called *kubeconfig*. To use kubectl, you must both install it and prepare the kubeconfig file so kubectl knows what cluster to talk to.

DOWNLOADING AND INSTALLING KUBECTL

The latest stable release for Linux can be downloaded and installed with the following command:

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release
[CA] /$(curl -s https://storage.googleapis.com/kubernetes-release/release
[CA] /stable.txt)/bin/linux/amd64/kubectl
[CA] && chmod +x kubectl
[CA] && sudo mv kubectl /usr/local/bin/
```

To install `kubectl` on macOS, you can either run the same command, but replace `linux` in the URL with `darwin`, or install the tool via Homebrew by running `brew install kubectl`.

On Windows, download `kubectl.exe` from https://storage.googleapis.com/kubernetes-release/release/v1.18.2/bin/windows/amd64/kubectl.exe. To download the latest version, first go to https://storage.googleapis.com/kubernetes-release/release/stable.txt to see what the latest stable version is and then replace the version number in the first URL with this version. To check if you've installed it correctly, run `kubectl --help`. Note that kubectl may or may not yet be configured to talk to your Kubernetes cluster, which means most commands may not work yet.

> **TIP** You can always append `--help` to any `kubectl` command to get information on what it does and how to use it.

SETTING UP A SHORT ALIAS FOR KUBECTL

You'll use `kubectl` often. Having to type the full command every time is needlessly time-consuming, but you can speed things up by setting up an alias and tab completion for it.

Most users of Kubernetes use `k` as the alias for kubectl. If you haven't used aliases yet, here's how to define it in Linux and macOS. Add the following line to your `~/.bashrc` or equivalent file:

```
alias k=kubectl
```

On Windows, if you use the Command Prompt, define the alias by executing `doskey k=kubectl $*`. If you use PowerShell, execute `set-alias -name k -value kubectl`.

> **NOTE** You may not need an alias if you used `gcloud` to set up the cluster. It installs the `k` binary in addition to `kubectl`.

CONFIGURING TAB COMPLETION FOR KUBECTL

Even with a short alias like `k`, you'll still have to type a lot. Fortunately, the `kubectl` command can also output shell completion code for both the bash and the zsh shell. It enables tab completion of not only command names but also the object names. For example, later you'll learn how to view details of a particular cluster node by executing the following command:

```
$ kubectl describe node gke-kiada-default-pool-9bba9b18-4glf
```

That's a lot of typing that you'll repeat all the time. With tab completion, things are much easier. You just press TAB after typing the first few characters of each token:

```
$ kubectl desc<TAB> no<TAB> gke-ku<TAB>
```

To enable tab completion in bash, you must first install a package called `bash-completion` and then run the following command (you can also add it to `~/.bashrc` or equivalent):

```
$ source <(kubectl completion bash)
```

But there's one caveat. This will only complete your commands when you use the full `kubectl` command name. It won't work when you use the `k` alias. To enable completion for the alias, you must run the following command:

```
$ complete -o default -F __start_kubectl k
```

3.2.2 Configuring kubectl to use a specific Kubernetes cluster

The kubeconfig configuration file is located at `~/.kube/config`. If you deployed your cluster using Docker Desktop, Minikube or GKE, the file was created for you. If you've been given access to an existing cluster, you should have received the file. Other tools, such as kind, may have written the file to a different location. Instead of moving the file to the default location, you can also point kubectl to it by setting the `KUBECONFIG` environment variable as follows:

```
$ export KUBECONFIG=/path/to/custom/kubeconfig
```

To learn more about how to manage kubectl's configuration and create a config file from scratch, refer to appendix A.

> **NOTE** If you want to use several Kubernetes clusters (for example, both Minikube and GKE), see appendix A for information on switching between different `kubectl` contexts.

3.2.3 Using kubectl

Assuming you've installed and configured kubectl, you can now use it to talk to your cluster.

VERIFYING IF THE CLUSTER IS UP AND KUBECTL CAN TALK TO IT

To verify that your cluster is working, use the `kubectl cluster-info` command:

```
$ kubectl cluster-info
Kubernetes master is running at https://192.168.99.101:8443
KubeDNS is running at https://192.168.99.101:8443/api/v1/namespaces/...
```

This indicates that the API server is active and responding to requests. The output lists the URLs of the various Kubernetes cluster services running in your cluster. The above example shows that besides the API server, the KubeDNS service, which provides domain-name services within the cluster, is another service that runs in the cluster.

LISTING CLUSTER NODES

Now use the `kubectl get nodes` command to list all nodes in your cluster. Here's the output that is generated when you run the command in a cluster provisioned by kind:

```
$ kubectl get nodes
NAME           STATUS   ROLES    AGE   VERSION
control-plane  Ready    <none>   12m   v1.18.2
kind-worker    Ready    <none>   12m   v1.18.2
```

```
kind-worker2    Ready    <none>  12m    v1.18.2
```

Everything in Kubernetes is represented by an object and can be retrieved and manipulated via the RESTful API. The `kubectl get` command retrieves a list of objects of the specified type from the API. You'll use this command all the time, but it only displays summary information about the listed objects.

RETRIEVING ADDITIONAL DETAILS OF AN OBJECT

To see more detailed information about an object, you use the `kubectl describe` command, which shows much more:

```
$ kubectl describe node gke-kiada-85f6-node-0rrx
```

I omit the actual output of the `describe` command because it's quite wide and would be completely unreadable here in the book. If you run the command yourself, you'll see that it displays the status of the node, information about its CPU and memory usage, system information, containers running on the node, and much more.

If you run the `kubectl describe` command without specifying the resource name, information of all nodes will be printed.

> **TIP** Executing the `describe` command without specifying the object name is useful when only one object of a certain type exists. You don't have to type or copy/paste the object name.

You'll learn more about the numerous other kubectl commands throughout the book.

3.2.4 Interacting with Kubernetes through web dashboards

If you prefer using graphical web user interfaces, you'll be happy to hear that Kubernetes also comes with a nice web dashboard. Note, however, that the functionality of the dashboard may lag significantly behind kubectl, which is the primary tool for interacting with Kubernetes.

Nevertheless, the dashboard shows different resources in context and can be a good start to get a feel for what the main resource types in Kubernetes are and how they relate to each other. The dashboard also offers the possibility to modify the deployed objects and displays the equivalent kubectl command for each action - a feature most beginners will appreciate.

Figure 3.7 shows the dashboard with two workloads deployed in the cluster.

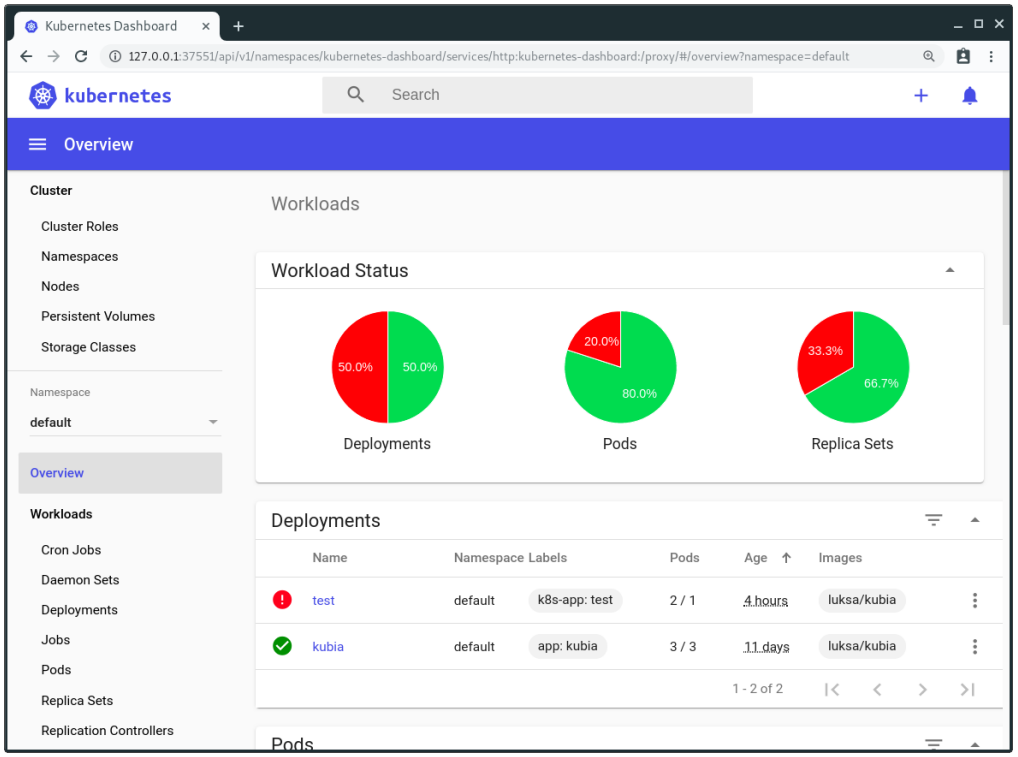

Figure 3.7 Screenshot of the Kubernetes web-based dashboard

Although you won't use the dashboard in this book, you can always open it to quickly see a graphical view of the objects deployed in your cluster after you create them via kubectl.

ACCESSING THE DASHBOARD IN DOCKER DESKTOP

Unfortunately, Docker Desktop does not install the Kubernetes dashboard by default. Accessing it is also not trivial, but here's how. First, you need to install it using the following command:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/
[CA] v2.0.0-rc5/aio/deploy/recommended.yaml
```

Refer to github.com/kubernetes/dashboard to find the latest version number. After installing the dashboard, the next command you must run is:

```
$ kubectl proxy
```

This command runs a local proxy to the API server, allowing you to access the services through it. Let the proxy process run and use the browser to open the dashboard at the following URL:

http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/

You'll be presented with an authentication page. You must then run the following command to retrieve an authentication token.

```
$ kubectl -n kubernetes-dashboard describe secret $(kubectl -n kubernetes-
[CA] dashboard get secret | sls admin-user | ForEach-Object { $_ -Split
[CA] '\s+' } | Select -First 1)
```

NOTE This command must be run in Windows PowerShell.

Find the token listed under `kubernetes-dashboard-token-xyz` and paste it into the token field on the authentication page shown in your browser. After you do this, you should be able to use the dashboard. When you're finished using it, terminate the `kubectl proxy` process using Control-C.

ACCESSING THE DASHBOARD WHEN USING MINIKUBE

If you're using Minikube, accessing the dashboard is much easier. Run the following command and the dashboard will open in your default browser:

```
$ minikube dashboard
```

ACCESSING THE DASHBOARD WHEN RUNNING KUBERNETES ELSEWHERE

The Google Kubernetes Engine no longer provides access to the open source Kubernetes Dashboard, but it offers an alternative web-based console. The same applies to other cloud providers. For information on how to access the dashboard, please refer to the documentation of the respective provider.

If your cluster runs on your own infrastructure, you can deploy the dashboard by following the guide at kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard.

3.3 Running your first application on Kubernetes

Now is the time to finally deploy something to your cluster. Usually, to deploy an application, you'd prepare a JSON or YAML file describing all the components that your application consists of and apply that file to your cluster. This would be the declarative approach.

Since this may be your first time deploying an application to Kubernetes, let's choose an easier way to do this. We'll use simple, one-line imperative commands to deploy your application.

3.3.1 Deploying your application

The imperative way to deploy an application is to use the `kubectl create deployment` command. As the command itself suggests, it creates a *Deployment* object, which represents an application deployed in the cluster. By using the imperative command, you avoid the need to know the structure of Deployment objects as when you write YAML or JSON manifests.

CREATING A DEPLOYMENT

In the previous chapter, you created a Node.js application called Kiada that you packaged into a container image and pushed to Docker Hub to make it easily distributable to any computer.

> **NOTE** If you skipped chapter two because you are already familiar with Docker and containers, you might want to go back and read section 2.2.1 that describes the application that you'll deploy here and in the rest of this book.

Let's deploy the Kiada application to your Kubernetes cluster. Here's the command that does this:

```
$ kubectl create deployment kiada --image=luksa/kiada:0.1
deployment.apps/kiada created
```

In the command, you specify three things:

- You want to create a `deployment` object.
- You want the object to be called `kiada`.
- You want the deployment to use the container image `luksa/kiada:0.1`.

By default, the image is pulled from Docker Hub, but you can also specify the image registry in the image name (for example, `quay.io/luksa/kiada:0.1`).

> **NOTE** Make sure that the image is stored in a public registry and can be pulled without access authorization. You'll learn how to provide credentials for pulling private images in chapter 8.

The Deployment object is now stored in the Kubernetes API. The existence of this object tells Kubernetes that the `luksa/kiada:0.1` container must run in your cluster. You've stated your *desired* state. Kubernetes must now ensure that the *actual* state reflects your wishes.

LISTING DEPLOYMENTS

The interaction with Kubernetes consists mainly of the creation and manipulation of objects via its API. Kubernetes stores these objects and then performs operations to bring them to life. For example, when you create a Deployment object, Kubernetes runs an application. Kubernetes then keeps you informed about the current state of the application by writing the status to the same Deployment object. You can view the status by reading back the object. One way to do this is to list all Deployment objects as follows:

```
$ kubectl get deployments
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
kiada   0/1     1            0           6s
```

The `kubectl get deployments` command lists all Deployment objects that currently exist in the cluster. You have only one Deployment in your cluster. It runs one instance of your application as shown in the `UP-TO-DATE` column, but the `AVAILABLE` column indicates that the application is not yet available. That's because the container isn't ready, as shown in the `READY` column. You can see that zero of a total of one container are ready.

You may wonder if you can ask Kubernetes to list all the running containers by running `kubectl get containers`. Let's try this.

```
$ kubectl get containers
error: the server doesn't have a resource type "containers"
```

The command fails because Kubernetes doesn't have a "Container" object type. This may seem odd, since Kubernetes is all about running containers, but there's a twist. A container is not the smallest unit of deployment in Kubernetes. So, what is?

INTRODUCING PODS

In Kubernetes, instead of deploying individual containers, you deploy groups of co-located containers – so-called *pods*. You know, as in *pod of whales*, or a *pea pod*.

A pod is a group of one or more closely related containers (not unlike peas in a pod) that run together on the same worker node and need to share certain Linux namespaces, so that they can interact more closely than with other pods.

In the previous chapter I showed an example where two processes use the same namespaces. By sharing the network namespace, both processes use the same network interfaces, share the same IP address and port space. By sharing the UTS namespace, both see the same system hostname. This is exactly what happens when you run containers in the same pod. They use the same network and UTS namespaces, as well as others, depending on the pod's spec.

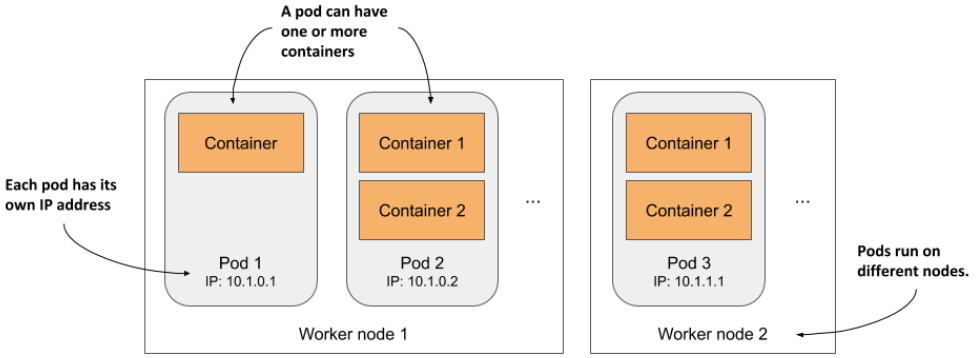

Figure 3.8 The relationship between containers, pods, and worker nodes

As illustrated in figure 3.8, you can think of each pod as a separate logical computer that contains one application. The application can consist of a single process running in a container, or a main application process and additional supporting processes, each running in a separate container. Pods are distributed across all the worker nodes of the cluster.

Each pod has its own IP, hostname, processes, network interfaces and other resources. Containers that are part of the same pod think that they're the only ones running on the computer. They don't see the processes of any other pod, even if located on the same node.

LISTING PODS

Since containers aren't a top-level Kubernetes object, you can't list them. But you can list pods. As the following output of the `kubectl get pods` command shows, by creating the Deployment object, you've deployed one pod:

```
$ kubectl get pods
NAME                      READY     STATUS    RESTARTS   AGE
kiada-9d785b578-p449x     0/1       Pending   0          1m
```

This is the pod that houses the container running your application. To be precise, since the status is still `Pending`, the application, or rather the container, isn't running yet. This is also expressed in the `READY` column, which indicates that the pod has a single container that's not ready.

The reason the pod is pending is because the worker node to which the pod has been assigned must first download the container image before it can run it. When the download is complete, the pod's container is created and the pod enters the `Running` state.

If Kubernetes can't pull the image from the registry, the `kubectl get pods` command will indicate this in the `STATUS` column. If you're using your own image, ensure it's marked as public on Docker Hub. Try pulling the image manually with the `docker pull` command on another computer.

If another issue is causing your pod not to run, or if you simply want to see more information about the pod, you can also use the `kubectl describe pod` command, as you did earlier to see the details of a worker node. If there are any issues with the pod, they should be displayed by this command. Look at the events shown at the bottom of its output. For a running pod, they should be close the following:

```
Type    Reason     Age   From                   Message
----    ------     ----  ----                   -------
Normal  Scheduled  25s   default-scheduler      Successfully assigned
                                                default/kiada-9d785b578-p449x
                                                to kind-worker2
Normal  Pulling    23s   kubelet, kind-worker2  Pulling image "luksa/kiada:0.1"
Normal  Pulled     21s   kubelet, kind-worker2  Successfully pulled image
Normal  Created    21s   kubelet, kind-worker2  Created container kiada
Normal  Started    21s   kubelet, kind-worker2  Started container kiada
```

UNDERSTANDING WHAT HAPPENS BEHIND THE SCENES

To help you visualize what happened when you created the Deployment, see figure 3.9.

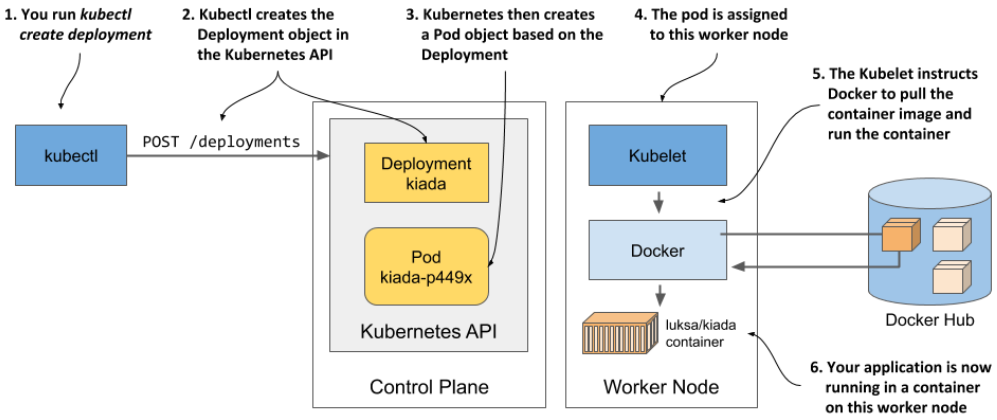

Figure 3.9 How creating a Deployment object results in a running application container

When you ran the `kubectl create` command, it created a new Deployment object in the cluster by sending an HTTP request to the Kubernetes API server. Kubernetes then created a new Pod object, which was then assigned or *scheduled* to one of the worker nodes. The Kubernetes agent on the worker node (the Kubelet) became aware of the newly created Pod object, saw that it was scheduled to its node, and instructed Docker to pull the specified image from the registry, create a container from the image, and execute it.

> **DEFINITION** The term scheduling refers to the assignment of the pod to a node. The pod runs immediately, not at some point in the future. Just like how the CPU scheduler in an operating system selects what CPU to run a process on, the scheduler in Kubernetes decides what worker node should execute each container. Unlike an OS process, once a pod is assigned to a node, it runs only on that node. Even if it fails, this instance of the pod is never moved to other nodes, as is the case with CPU processes, but a new pod instance may be created to replace it.

Depending on what you use to run your Kubernetes cluster, the number of worker nodes in your cluster may vary. The figure shows only the worker node that the pod was scheduled to. In a multi-node cluster, none of the other worker nodes are involved in the process.

3.3.2 Exposing your application to the world

Your application is now running, so the next question to answer is how to access it. I mentioned that each pod gets its own IP address, but this address is internal to the cluster and not accessible from the outside. To make the pod accessible externally, you'll *expose* it by creating a Service object.

Several types of Service objects exist. You decide what type you need. Some expose pods only within the cluster, while others expose them externally. A service with the type LoadBalancer provisions an external load balancer, which makes the service accessible via a public IP. This is the type of service you'll create now.

CREATING A SERVICE

The easiest way to create the service is to use the following imperative command:

```
$ kubectl expose deployment kiada --type=LoadBalancer --port 8080
service/kiada exposed
```

The `create deployment` command that you ran previously created a Deployment object, whereas the `expose deployment` command creates a Service object. This is what running the above command tells Kubernetes:

- You want to expose all pods that belong to the kiada Deployment as a new service.
- You want the pods to be accessible from outside the cluster via a load balancer.
- The application listens on port 8080, so you want to access it via that port.

You didn't specify a name for the Service object, so it inherits the name of the Deployment.

LISTING SERVICES

Services are API objects, just like Pods, Deployments, Nodes and virtually everything else in Kubernetes, so you can list them by executing `kubectl get services`:

```
$ kubectl get svc
NAME         TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
kubernetes   ClusterIP      10.19.240.1    <none>        443/TCP          34m
kiada        LoadBalancer   10.19.243.17   <pending>     8080:30838/TCP   4s
```

> **NOTE** Notice the use of the abbreviation `svc` instead of `services`. Most resource types have a short name that you can use instead of the full object type (for example, `po` is short for `pods`, `no` for `nodes` and `deploy` for `deployments`).

The list shows two services with their types, IPs and the ports they expose. Ignore the `kubernetes` service for now and take a close look at the `kiada` service. It doesn't yet have an external IP address. Whether it gets one depends on how you've deployed the cluster.

Listing the available object types with kubectl api-resources

You've used the `kubectl get` command to list various things in your cluster: Nodes, Deployments, Pods and now Services. These are all Kubernetes object types. You can display a list of all supported types by running `kubectl api-resources`. The list also shows the short name for each type and some other information you need to define objects in JSON/YAML files, which you'll learn in the following chapters.

UNDERSTANDING LOAD BALANCER SERVICES

While Kubernetes allows you to create so-called LoadBalancer services, it doesn't provide the load balancer itself. If your cluster is deployed in the cloud, Kubernetes can ask the cloud infrastructure to provision a load balancer and configure it to forward traffic into your cluster. The infrastructure tells Kubernetes the IP address of the load balancer and this becomes the external address of your service.

The process of creating the Service object, provisioning the load balancer and how it forwards connections into the cluster is shown in the next figure.

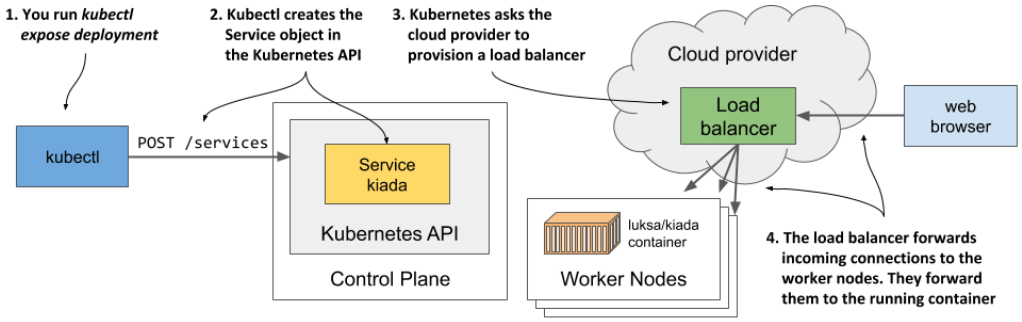

Figure 3.10 What happens when you create a Service object of type LoadBalancer

Provisioning of the load balancer takes some time, so let's wait a few more seconds and check again whether the IP address is already assigned. This time, instead of listing all services, you'll display only the `kiada` service as follows:

```
$ kubectl get svc kiada
NAME        TYPE           CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
kiada       LoadBalancer   10.19.243.17  35.246.179.22  8080:30838/TCP   82s
```

The external IP is now displayed. This means that the load balancer is ready to forward requests to your application for clients around the world.

> **NOTE** If you deployed your cluster with Docker Desktop, the load balancer's IP address is shown as `localhost`, referring to your Windows or macOS machine, not the VM where Kubernetes and the application runs. If you use Minikube to create the cluster, no load balancer is created, but you can access the service in another way. More on this later.

ACCESSING YOUR APPLICATION THROUGH THE LOAD BALANCER

You can now send requests to your application through the external IP and port of the service:

```
$ curl 35.246.179.22:8080
Kiada version 0.1. Request processed by "kiada-9d785b578-p449x". Client IP: ::ffff:1.2.3.4
```

> **NOTE** If you use Docker Desktop, the service is available at `localhost:8080` from within your host operating system. Use curl or your browser to access it.

Congratulations! If you use Google Kubernetes Engine, you've successfully published your application to users across the globe. Anyone who knows its IP and port can now access it. If you don't count the steps needed to deploy the cluster itself, only two simple commands were needed to deploy your application:

- `kubectl create deployment` and
- `kubectl expose deployment`.

ACCESSING YOUR APPLICATION WHEN A LOAD BALANCER ISN'T AVAILABLE

Not all Kubernetes clusters have mechanisms to provide a load balancer. The cluster provided by Minikube is one of them. If you create a service of type LoadBalancer, the service itself works, but there is no load balancer. Kubectl always shows the external IP as `<pending>` and you must use a different method to access the service.

Several methods of accessing services exist. You can even bypass the service and access individual pods directly, but this is mostly used for troubleshooting. You'll learn how to do this in chapter 5. For now, let's explore the next easier way to access your service if no load balancer is available.

Minikube can tell you where to access the service if you use the following command:

```
$ minikube service kiada --url
http://192.168.99.102:30838
```

The command prints out the URL of the service. You can now point `curl` or your browser to that URL to access your application:

```
$ curl http://192.168.99.102:30838
Kiada version 0.1. Request processed by "kiada-9d785b578-p449x". Client IP:
       ::ffff:172.17.0.1
```

> **TIP** If you omit the `--url` option when running the `minikube service` command, your browser opens and loads the service URL.

You may wonder where this IP address and port come from. This is the IP of the Minikube virtual machine. You can confirm this by executing the `minikube ip` command. The Minikube VM is also your single worker node. The port `30838` is the so-called *node port*. It's the port on the worker node that forwards connections to your service. You may have noticed the port in the service's port list when you ran the `kubectl get svc` command:

```
$ kubectl get svc kiada
NAME        TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)         AGE
kiada       LoadBalancer  10.19.243.17  <pending>      8080:30838/TCP  82s
```

Your service is accessible via this port number on all your worker nodes, regardless of whether you're using Minikube or any other Kubernetes cluster.

> **NOTE** If you use Docker Desktop, the VM running Kubernetes can't be reached from your host OS through the VM's IP. You can access the service through the node port only within the VM by logging into it using the special container as described in section 3.1.1.

If you know the IP of at least one of your worker nodes, you should be able to access your service through this `IP:port` combination, provided that firewall rules do not prevent you from accessing the port.

The next figure shows how external clients access the application via the node ports.

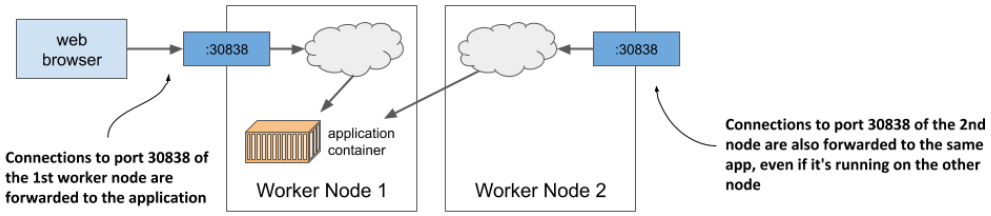

Figure 3.11 Connection routing through a service's node port

To connect this to what I mentioned earlier about the load balancer forwarding connections to the nodes and the nodes then forwarding them to the containers: the node ports are exactly where the load balancer sends incoming requests to. Kubernetes then ensures that they are forwarded to the application running in the container. You'll learn how it does this in chapter 10, as we delve deeper into services. Don't lose too much time thinking about it until then. Instead, let's play a little more with our cluster to see what else Kubernetes can do.

3.3.3 Horizontally scaling the application

You now have a running application that is represented by a Deployment and exposed to the world by a Service object. Now let's create some additional magic.

One of the major benefits of running applications in containers is the ease with which you can scale your application deployments. You're currently running a single instance of your application. Imagine you suddenly see many more users using your application. The single instance can no longer handle the load. You need to run additional instances to distribute the load and provide service to your users. This is known as *scaling out*. With Kubernetes, it's trivial to do.

INCREASING THE NUMBER OF RUNNING APPLICATION INSTANCES

To deploy your application, you've created a Deployment object. By default, it runs a single instance of your application. To run additional instances, you only need to scale the Deployment object with the following command:

```
$ kubectl scale deployment kiada --replicas=3
deployment.apps/kiada scaled
```

You've now told Kubernetes that you want to run three exact copies or *replicas* of your pod. Note that you haven't instructed Kubernetes what to do. You haven't told it to add two more pods. You just set the new desired number of replicas and let Kubernetes determine what action it must take to reach the new desired state.

This is one of the most fundamental principles in Kubernetes. Instead of telling Kubernetes what to do, you simply set a new desired state of the system and let Kubernetes achieve it. To do this, it examines the current state, compares it with the desired state, identifies the differences and determines what it must do to reconcile them.

SEEING THE RESULTS OF THE SCALE-OUT

Although it's true that the `kubectl scale deployment` command seems imperative, since it apparently tells Kubernetes to scale your application, what the command actually does is modify the specified Deployment object. As you'll see in a later chapter, you could have simply edited the object instead of giving the imperative command. Let's view the Deployment object again to see how the scale command has affected it:

```
$ kubectl get deploy
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
kiada   3/3     3            3           18m
```

Three instances are now up to date and available and three of three containers are ready. This isn't clear from the command output, but the three containers are not part of the same pod instance. There are three pods with one container each. You can confirm this by listing pods:

```
$ kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
kiada-9d785b578-58vhc    1/1     Running   0          17s
kiada-9d785b578-jmnj8    1/1     Running   0          17s
kiada-9d785b578-p449x    1/1     Running   0          18m
```

As you can see, three pods now exist. As indicated in the `READY` column, each has a single container, and all the containers are ready. All the pods are `Running`.

DISPLAYING THE PODS' HOST NODE WHEN LISTING PODS

If you use a single-node cluster, all your pods run on the same node. But in a multi-node cluster, the three pods should be distributed throughout the cluster. To see which nodes the pods were scheduled to, you can use the `-o wide` option to display a more detailed pod list:

```
$ kubectl get pods -o wide
NAME                     ...  IP           NODE
kiada-9d785b578-58vhc    ...  10.244.1.5   kind-worker    #A
kiada-9d785b578-jmnj8    ...  10.244.2.4   kind-worker2    #B
kiada-9d785b578-p449x    ...  10.244.2.3   kind-worker2    #B
```

#A Pod scheduled to one node
#B Two pods scheduled to another node

> **NOTE** You can also use the `-o wide` output option to see additional information when listing other object types.

The wide output shows that one pod was scheduled to one node, whereas the other two were both scheduled to a different node. The Scheduler usually distributes pods evenly, but it depends on how it's configured. You'll learn more about scheduling in chapter 21.

Does the host node matter?

Regardless of the node they run on, all instances of your application have an identical OS environment, because they run in containers created from the same container image. You may remember from the previous chapter that

the only thing that might be different is the OS kernel, but this only happens when different nodes use different kernel versions or load different kernel modules.

In addition, each pod gets its own IP and can communicate in the same way with any other pod - it doesn't matter if the other pod is on the same worker node, another node located in the same server rack or even a completely different data center.

So far, you've set no resource requirements for the pods, but if you had, each pod would have been allocated the requested amount of compute resources. It shouldn't matter to the pod which node provides these resources, as long as the pod's requirements are met.

Therefore, you shouldn't care where a pod is scheduled to. It's also why the default `kubectl get pods` command doesn't display information about the worker nodes for the listed pods. In the world of Kubernetes, it's just not that important.

As you can see, scaling an application is incredibly easy. Once your application is in production and there is a need to scale it, you can add additional instances with a single command without having to manually install, configure and run additional copies.

> **NOTE** The app itself must support horizontal scaling. Kubernetes doesn't magically make your app scalable; it merely makes it trivial to replicate it.

OBSERVING REQUESTS HITTING ALL THREE PODS WHEN USING THE SERVICE

Now that multiple instances of your app are running, let's see what happens when you hit the service URL again. Will the response come from the same instance every time? Here's the answer:

```
$ curl 35.246.179.22:8080
Kiada version 0.1. Request processed by "kiada-9d785b578-58vhc". Client IP: ::ffff:1.2.3.4
       #A
$ curl 35.246.179.22:8080
Kiada version 0.1. Request processed by "kiada-9d785b578-p449x". Client IP: ::ffff:1.2.3.4
       #B
$ curl 35.246.179.22:8080
Kiada version 0.1. Request processed by "kiada-9d785b578-jmnj8". Client IP: ::ffff:1.2.3.4
       #C
$ curl 35.246.179.22:8080
Kiada version 0.1. Request processed by "kiada-9d785b578-p449x". Client IP: ::ffff:1.2.3.4
       #D
```

#A Request reaches the first pod
#B Request reaches the third pod
#C Request reaches the second pod
#D Request reaches the third pod again

If you look closely at the responses, you'll see that they correspond to the names of the pods. Each request arrives at a different pod in random order. This is what services in Kubernetes

do when more than one pod instance is behind them. They act as load balancers in front of the pods. Let's visualize the system using the following figure.

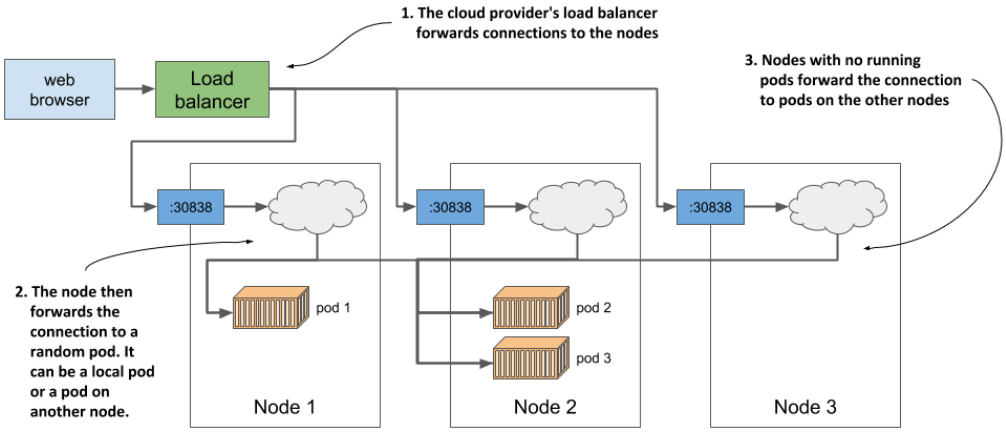

Figure 3.12 Load balancing across multiple pods backing the same service

As the figure shows, you shouldn't confuse this load balancing mechanism, which is provided by the Kubernetes service itself, with the additional load balancer provided by the infrastructure when running in GKE or another cluster running in the cloud. Even if you use Minikube and have no external load balancer, your requests are still distributed across the three pods by the service itself. If you use GKE, there are actually *two* load balancers in play. The figure shows that the load balancer provided by the infrastructure distributes requests across the nodes, and the service then distributes requests across the pods.

I know this may be very confusing right now, but it should all become clear in chapter 10.

3.3.4 Understanding the deployed application

To conclude this chapter, let's review what your system consists of. There are two ways to look at your system – the logical and the physical view. You've just seen the physical view in figure 3.12. There are three running containers that are deployed on three worker nodes (a single node when using Minikube). If you run Kubernetes in the cloud, the cloud infrastructure has also created a load balancer for you. Docker Desktop also creates a type of local load balancer. Minikube doesn't create a load balancer, but you can access your service directly through the node port.

While differences in the physical view of the system in different clusters exist, the logical view is always the same, whether you use a small development cluster or a large production cluster with thousands of nodes. If you're not the one who manages the cluster, you don't even need to worry about the physical view of the cluster. If everything works as expected, the logical view is all you need to worry about. Let's take a closer look at this view.

UNDERSTANDING THE API OBJECTS REPRESENTING YOUR APPLICATION

The logical view consists of the objects you've created in the Kubernetes API – either directly or indirectly. The following figure shows how the objects relate to each other.

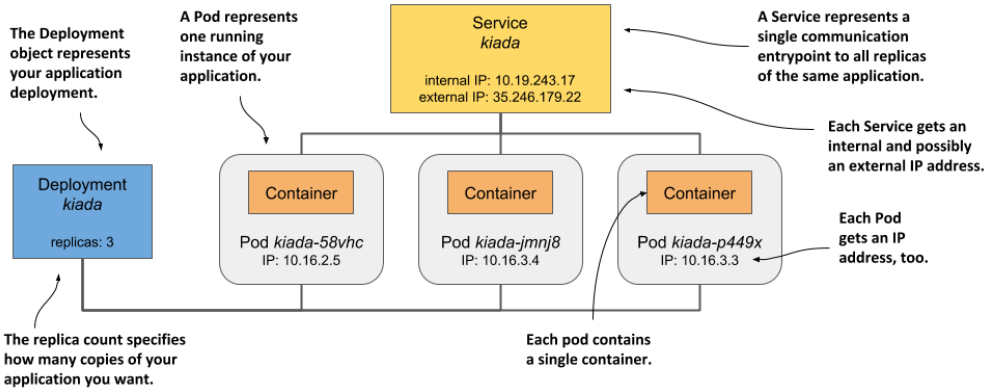

Figure 3.13 Your deployed application consists of a Deployment, several Pods, and a Service.

The objects are as follows:

- the Deployment object you created,
- the Pod objects that were automatically created based on the Deployment, and
- the Service object you created manually.

There are other objects between the three just mentioned, but you don't need to know them yet. You'll learn about them in the following chapters.

Remember when I explained in chapter 1 that Kubernetes abstracts the infrastructure? The logical view of your application is a great example of this. There are no nodes, no complex network topology, no physical load balancers. Just a simple view that only contains your applications and the supporting objects. Let's look at how these objects fit together and what role they play in your small setup.

The Deployment object represents an application deployment. It specifies which container image contains your application and how many replicas of the application Kubernetes should run. Each replica is represented by a Pod object. The Service object represents a single communication entry point to these replicas.

UNDERSTANDING THE PODS

The essential and most important part of your system are the pods. Each pod definition contains one or more containers that make up the pod. When Kubernetes brings a pod to life, it runs all the containers specified in its definition. As long as a Pod object exists, Kubernetes will do its best to ensure that its containers keep running. It only shuts them down when the Pod object is deleted.

UNDERSTANDING THE ROLE OF THE DEPLOYMENT

When you first created the Deployment object, only a single Pod object was created. But when you increased the desired number of replicas on the Deployment, Kubernetes created additional replicas. Kubernetes ensures that the actual number of pods always matches the desired number.

If one or more pods disappear or their status is unknown, Kubernetes replaces them to bring the actual number of pods back to the desired number of replicas. A pod disappears when someone or something deletes it, whereas a pod's status is unknown when the node it is running on no longer reports its status due to a network or node failure.

Strictly speaking, a Deployment results in nothing more than the creation of a certain number of Pod objects. You may wonder if you can create Pods directly instead of having the Deployment create them for you. You can certainly do this, but if you wanted to run multiple replicas, you'd have to manually create each pod individually and make sure you give each one a unique name. You'd then also have to keep a constant eye on your pods to replace them if they suddenly disappear or the node on which they run fails. And that's exactly why you almost never create pods directly but use a Deployment instead.

UNDERSTANDING WHY YOU NEED A SERVICE

The third component of your system is the Service object. By creating it, you tell Kubernetes that you need a single communication entry point to your pods. The service gives you a single IP address to talk to your pods, regardless of how many replicas are currently deployed. If the service is backed by multiple pods, it acts as a load balancer. But even if there is only one pod, you still want to expose it through a service. To understand why, you need to learn an important detail about pods.

Pods are ephemeral. A pod may disappear at any time. This can happen when its host node fails, when someone inadvertently deletes the pod, or when the pod is evicted from an otherwise healthy node to make room for other, more important pods. As explained in the previous section, when pods are created through a Deployment, a missing pod is immediately replaced with a new one. This new pod is not the same as the one it replaces. It's a completely new pod, with a new IP address.

If you weren't using a service and had configured your clients to connect directly to the IP of the original pod, you would now need to reconfigure all these clients to connect to the IP of the new pod. This is not necessary when using a service. Unlike pods, services aren't ephemeral. When you create a service, it is assigned a static IP address that never changes during lifetime of the service.

Instead of connecting directly to the pod, clients should connect to the IP of the service. This ensures that their connections are always routed to a healthy pod, even if the set of pods behind the service is constantly changing. It also ensures that the load is distributed evenly across all pods should you decide to scale the deployment horizontally.

3.4 Summary

In this hands-on chapter, you've learned:

- Virtually all cloud providers offer a managed Kubernetes option. They take on the burden

of maintaining your Kubernetes cluster, while you just use its API to deploy your applications.

- You can also install Kubernetes in the cloud yourself, but this has often proven not to be the best idea until you master all aspects of managing Kubernetes.
- You can install Kubernetes locally, even on your laptop, using tools such as Docker Desktop or Minikube, which run Kubernetes in a Linux VM, or kind, which runs the master and worker nodes as Docker containers and the application containers inside those containers.
- Kubectl, the command-line tool, is the usual way you interact with Kubernetes. A web-based dashboard also exists but is not as stable and up to date as the CLI tool.
- To work faster with kubectl, it is useful to define a short alias for it and enable shell completion.
- An application can be deployed using `kubectl create deployment`. It can then be exposed to clients by running `kubectl expose deployment`. Horizontally scaling the application is trivial: `kubectl scale deployment` instructs Kubernetes to add new replicas or removes existing ones to reach the number of replicas you specify.
- The basic unit of deployment is not a container, but a pod, which can contain one or more related containers.
- Deployments, Services, Pods and Nodes are Kubernetes objects/resources. You can list them with `kubectl get` and inspect them with `kubectl describe`.
- The Deployment object deploys the desired number of Pods, while the Service object makes them accessible under a single, stable IP address.
- Each service provides internal load balancing in the cluster, but if you set the type of service to `LoadBalancer`, Kubernetes will ask the cloud infrastructure it runs in for an additional load balancer to make your application available at a publicly accessible address.

You've now completed your first guided tour around the bay. Now it's time to start learning the ropes, so that you'll be able to sail independently. The next part of the book focuses on the different Kubernetes objects and how/when to use them. You'll start with the most important one – the Pod.

4

Introducing the Kubernetes API objects

This chapter covers

- Managing a Kubernetes cluster and the applications it hosts via its API
- Understanding the structure of Kubernetes API objects
- Retrieving and understanding an object's YAML or JSON manifest
- Inspecting the status of cluster nodes via Node objects
- Inspecting cluster events through Event objects

The previous chapter introduced three fundamental objects that make up a deployed application. You created a Deployment object that spawned multiple Pod objects representing individual instances of your application and exposed them to the world by creating a Service object that deployed a load balancer in front of them.

The chapters in the second part of this book explain these and other object types in detail. In this chapter, the common features of Kubernetes objects are presented using the example of Node and Event objects.

4.1 Getting familiar with the Kubernetes API

In a Kubernetes cluster, both users and Kubernetes components interact with the cluster by manipulating objects through the Kubernetes API, as shown in figure 4.1.

These objects represent the configuration of the entire cluster. They include the applications running in the cluster, their configuration, the load balancers through which they are exposed within the cluster or externally, the underlying servers and the storage used by these applications, the security privileges of users and applications, and many other details of the infrastructure.

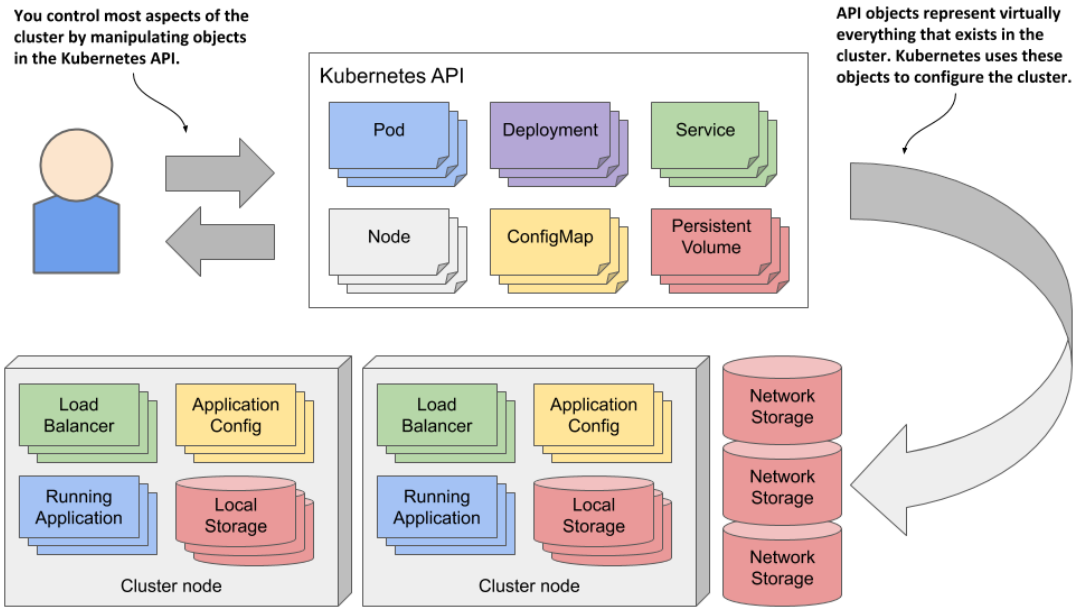

Figure 4.1 A Kubernetes cluster is configured by manipulating objects in the Kubernetes API

4.1.1 Introducing the API

The Kubernetes API is the central point of interaction with the cluster, so much of this book is dedicated to explaining this API. The most important API objects are described in the following chapters, but a basic introduction to the API is presented here.

UNDERSTANDING THE ARCHITECTURAL STYLE OF THE API

The Kubernetes API is an HTTP-based RESTful API where the state is represented by *resources* on which you perform CRUD operations (Create, Read, Update, Delete) using standard HTTP methods such as `POST`, `GET`, `PUT`/`PATCH` or `DELETE`.

> **DEFINITION** REST is Representational State Transfer, an architectural style for implementing interoperability between computer systems via web services using stateless operations, described by Roy Thomas Fielding in his doctoral dissertation. To learn more, read the dissertation at https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

It is these resources (or objects) that represent the configuration of the cluster. Cluster administrators and engineers who deploy applications into the cluster therefore influence the configuration by manipulating these objects.

In the Kubernetes community, the terms "resource" and "object" are used interchangeably, but there are subtle differences that warrant an explanation.

UNDERSTANDING THE DIFFERENCE BETWEEN RESOURCES AND OBJECTS

The essential concept in RESTful APIs is the resource, and each resource is assigned a URI or Uniform Resource Identifier that uniquely identifies it. For example, in the Kubernetes API, application deployments are represented by deployment resources.

The collection of all deployments in the cluster is a REST resource exposed at `/api/v1/deployments`. When you use the `GET` method to send an HTTP request to this URI, you receive a response that lists all deployment instances in the cluster.

Each individual deployment instance also has its own unique URI through which it can be manipulated. The individual deployment is thus exposed as another REST resource. You can retrieve information about the deployment by sending a `GET` request to the resource URI and you can modify it using a `PUT` request.

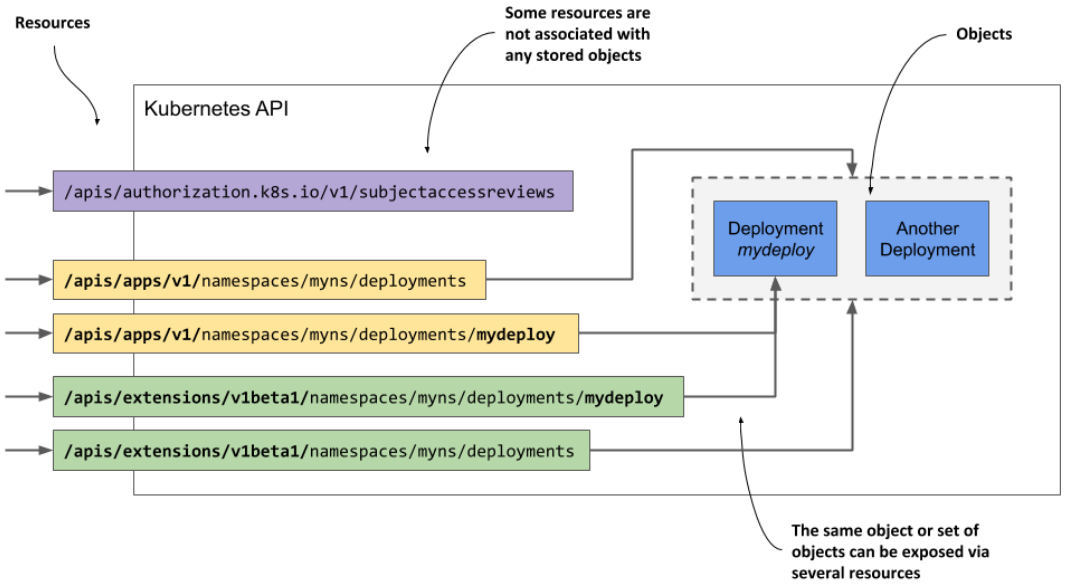

Figure 4.2 A single object can be exposed by two or more resources

An object can therefore be exposed through more than one resource. As shown in figure 4.2, the Deployment object instance named `mydeploy` is returned both as an element of a collection when you query the `deployments` resource and as a single object when you query the individual resource URI directly.

In addition, a single object instance can also be exposed via multiple resources if multiple API versions exist for an object type. Up to Kubernetes version 1.15, two different representations of Deployment objects were exposed by the API. In addition to the `apps/v1` version, exposed at `/apis/apps/v1/deployments`, an older version, `extensions/v1beta1`, exposed at `/apis/extensions/v1beta1/deployments` was available in the API. These two resources didn't represent two different sets of Deployment objects, but a single set that was

represented in two different ways - with small differences in the object schema. You could create an instance of a Deployment object via the first URI and then read it back using the second.

In some cases, a resource doesn't represent any object at all. An example of this is the way the Kubernetes API allows clients to verify whether a subject (a person or a service) is authorized to perform an API operation. This is done by submitting a `POST` request to the `/apis/authorization.k8s.io/v1/subjectaccessreviews` resource. The response indicates whether the subject is authorized to perform the operation specified in the request body. The key thing here is that no object is created by the `POST` request.

The examples described above show that a resource isn't the same as an object. If you are familiar with relational database systems, you can compare resources and object types with views and tables. Resources are views through which you interact with objects.

> **NOTE** Because the term "resource" can also refer to compute resources, such as CPU and memory, to reduce confusion, the term "objects" is used in this book to refer to API resources.

UNDERSTANDING HOW OBJECTS ARE REPRESENTED

When you make a `GET` request for a resource, the Kubernetes API server returns the object in structured text form. The default data model is JSON, but you can also tell the server to return YAML instead. When you update the object using a `POST` or `PUT` request, you also specify the new state with either JSON or YAML.

The individual fields in an object's manifest depend on the object type, but the general structure and many fields are shared by all Kubernetes API objects. You'll learn about them next.

4.1.2 Understanding the structure of an object manifest

Before you are confronted with the complete manifest of a Kubernetes object, let me first explain its major parts, because this will help you to find your way through the sometimes hundreds of lines it is composed of.

INTRODUCING THE MAIN PARTS OF AN OBJECT

The manifest of most Kubernetes API objects consists of the following four sections:

- *Type Metadata* contains information about the type of object this manifest describes. It specifies the object type, the group to which the type belongs, and the API version.
- *Object Metadata* holds the basic information about the object instance, including its name, time of creation, owner of the object, and other identifying information. The fields in the Object Metadata are the same for all object types.
- *Spec* is the part in which you specify the desired state of the object. Its fields differ between different object types. For pods, this is the part that specifies the pod's containers, storage volumes and other information related to its operation.
- *Status* contains the current actual state of the object. For a pod, it tells you the condition of the pod, the status of each of its containers, its IP address, the node it's running on, and other information that reveals what's happening to your pod.

A visual representation of an object manifest and its four sections is shown in the next figure.

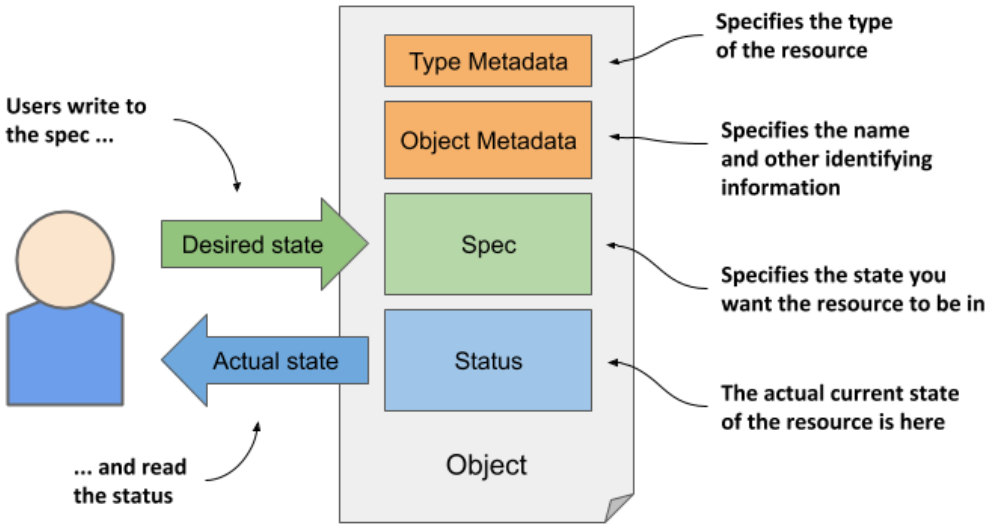

Figure 4.3 The main sections of a Kubernetes API object.

> **NOTE** Although the figure shows that users write to the object's Spec section and read its Status, the API server always returns the entire object when you perform a GET request; to update the object, you also send the entire object in the PUT request.

You'll see an example later to see which fields exist in these sections but let me first explain the Spec and Status sections, as they represent the flesh of the object.

UNDERSTANDING THE SPEC AND STATUS SECTIONS

As you may have noticed in the previous figure, the two most important parts of an object are the Spec and Status sections. You use the Spec to specify the desired state of the object and read the actual state of the object from the Status section. So, you are the one who writes the Spec and reads the Status, but who or what reads the Spec and writes the Status?

The Kubernetes Control Plane runs several components called *controllers* that manage the objects you create. Each controller is usually only responsible for one object type. For example, the *Deployment controller* manages Deployment objects.

As shown in figure 4.4, the task of a controller is to read the desired object state from the object's Spec section, perform the actions required to achieve this state, and report back the actual state of the object by writing to its Status section.

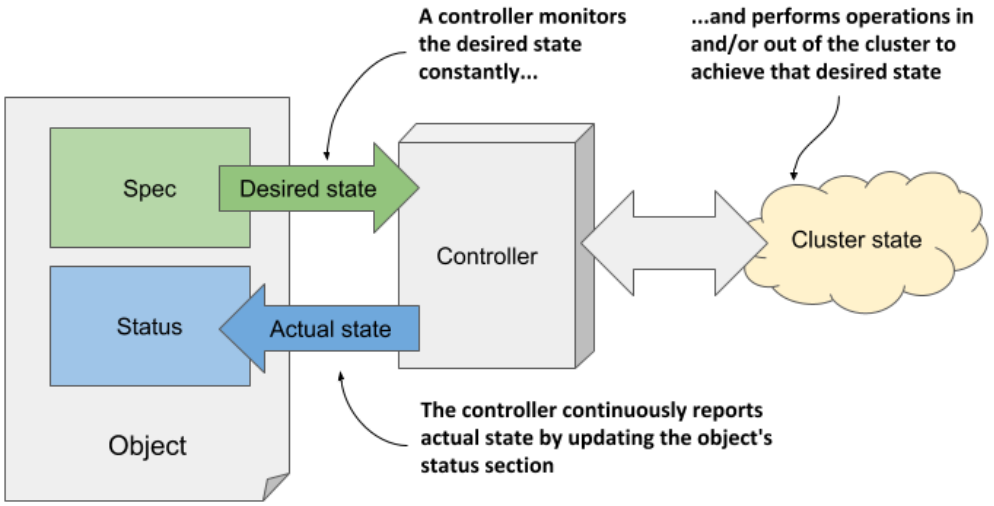

Figure 4.4 How a controller manages an object

Essentially, you tell Kubernetes what it has to do by creating and updating API objects. Kubernetes controllers use the same API objects to tell you what they have done and what the status of their work is.

You'll learn more about the individual controllers and their responsibilities in chapter 13. For now, just remember that a controller is associated with most object types and that the controller is the thing that reads the Spec and writes the Status of the object.

Not all objects have the spec and status sections

All Kubernetes API objects contain the two metadata sections, but not all have the Spec and Status sections. Those that don't, typically contain just static data and don't have a corresponding controller, so it is not necessary to distinguish between the desired and the actual state of the object.

An example of such an object is the Event object, which is created by various controllers to provide additional information about what is happening with an object that the controller is managing. The Event object is explained in section 4.3.

You now understand the general outline of an object, so the next section of this chapter can finally explore the individual fields of an object.

4.2 Examining an object's individual properties

To examine Kubernetes API objects up close, we'll need a concrete example. Let's take the Node object, which should be easy to understand because it represents something you might be relatively familiar with - a computer in the cluster.

My Kubernetes cluster provisioned by the kind tool has three nodes - one master and two workers. They are represented by three Node objects in the API. I can query the API and list these objects using `kubectl get nodes`:

```
$ kubectl get nodes
NAME                 STATUS   ROLES    AGE    VERSION
kind-control-plane   Ready    master   1h     v1.18.2
kind-worker          Ready    <none>   1h     v1.18.2
kind-worker2         Ready    <none>   1h     v1.18.2
```

The following figure shows the three Node objects and the actual cluster machines that make up the cluster. Each Node object instance represents one host. In each instance, the Spec section contains (part of) the configuration of the host, and the Status section contains the state of the host.

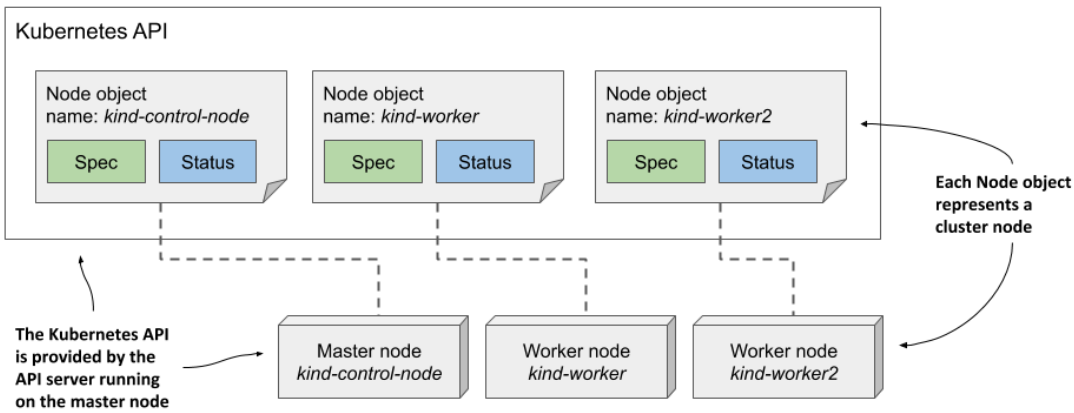

Figure 4.5 Cluster nodes are represented by Node objects

> **NOTE** Node objects are slightly different from other objects because they are usually created by the Kubelet - the node agent running on the cluster node - rather than by users. When you add a machine to the cluster, the Kubelet registers the node by creating a Node object that represents the host. Users can then edit (some of) the fields in the Spec section.

4.2.1 Exploring the full manifest of a Node object

Let's take a close look at one of the Node objects. List all Node objects in your cluster by running the `kubectl get nodes` command and select one you want to inspect. Then, execute

the `kubectl get node <node-name> -o yaml` command, where you replace `<node-name>` with the name of the node, as shown here:

```
$ kubectl get node kind-control-plane -o yaml
apiVersion: v1            #A
kind: Node                #A
metadata:                                                          #B
  annotations: ...
  creationTimestamp: "2020-05-03T15:09:17Z"
  labels: ...
  managedFields: ...
  name: kind-control-plane                                        #C
  resourceVersion: "3220054"
  selfLink: /api/v1/nodes/kind-control-plane
  uid: 16dc1e0b-8d34-4cfb-8ade-3b0e91ec838b
spec:                                                              #D
  podCIDR: 10.244.0.0/24                                           #E
  podCIDRs:                                                        #E
  - 10.244.0.0/24                                                  #E
  taints:
  - effect: NoSchedule
    key: node-role.kubernetes.io/master
status:                                                            #F
  addresses:                                                       #G
  - address: 172.18.0.2                                            #G
    type: InternalIP                                               #G
  - address: kind-control-plane                                    #G
    type: Hostname                                                 #G
  allocatable: ...
  capacity:                                                        #H
    cpu: "8"                                                       #H
    ephemeral-storage: 401520944Ki                                 #H
    hugepages-1Gi: "0"                                             #H
    hugepages-2Mi: "0"                                             #H
    memory: 32720824Ki                                             #H
    pods: "110"                                                    #H
  conditions:
  - lastHeartbeatTime: "2020-05-17T12:28:41Z"
    lastTransitionTime: "2020-05-03T15:09:17Z"
    message: kubelet has sufficient memory available
    reason: KubeletHasSufficientMemory
    status: "False"
    type: MemoryPressure
    ...
  daemonEndpoints:
    kubeletEndpoint:
      Port: 10250
  images:                                                          #I
  - names:                                                         #I
    - k8s.gcr.io/etcd:3.4.3-0                                      #I
    sizeBytes: 289997247                                           #I
    ...                                                            #I
  nodeInfo:                                                        #J
    architecture: amd64                                            #J
    bootID: 233a359f-5897-4860-863d-06546130e1ff                   #J
    containerRuntimeVersion: containerd://1.3.3-14-g449e9269       #J
    kernelVersion: 5.5.10-200.fc31.x86_64                          #J
    kubeProxyVersion: v1.18.2                                      #J
    kubeletVersion: v1.18.2                                        #J
    machineID: 74b74e389bb246e99abdf731d145142d                    #J
    operatingSystem: linux                                         #J
    osImage: Ubuntu 19.10                                          #J
```

```
    systemUUID: 8749f818-8269-4a02-bdc2-84bf5fa21700                #J
```

#A The Type Metadata specifies the type of object and the API version of this object manifest.
#B The Object Metadata section begins here
#C The object name (the node's name)
#D The node's desired state is specified in the spec section, which begins here
#E The IP range reserved for the pods on this node
#F The node's actual state is shown in the status section, which begins here and extends to the end of this listing
#G The IP(s) and hostname of the node
#H The nodes capacity (the amount of compute resources it has)
#I The list of cached container images on this node
#J Information about the node's operating system and the Kubernetes components running on it

NOTE Use the `-o json` option to display the object in JSON instead of YAML.

In the YAML manifest, the four main sections of the object definition and the more important properties of the node are annotated to help you distinguish between the more and less important fields. Some lines have been omitted to reduce the length of the manifest.

Accessing the API directly

You may be interested in trying to access the API directly instead of through kubectl. As explained earlier, the Kubernetes API is web based, so you can use a web browser or the curl command to perform API operations, but the API server uses TLS and you typically need a client certificate or token for authentication. Fortunately, kubectl provides a special proxy that takes care of this, allowing you to talk to the API through the proxy using plain HTTP.

To run the proxy, execute the command:

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

You can now access the API using HTTP at 127.0.0.1:8001. For example, to retrieve the node object, open the URL http://127.0.0.1:8001/api/v1/nodes/kind-control-plane (replace kind-control-plane with one of your nodes' names).

Now let's take a closer look at the fields in each of the four main sections.

THE TYPE METADATA FIELDS

As you can see, the manifest starts with the `apiVersion` and `kind` fields, which specify the API version and type of the object that this object manifest specifies. The API version is the schema used to describe this object. As mentioned before, an object type can be associated with more than one schema, with different fields in each schema being used to describe the object. However, usually only one schema exists for each type.

The `apiVersion` in the previous manifest is `v1`, but you'll see in the following chapters that the `apiVersion` in other object types contains more than just the version number. For Deployment objects, for example, the `apiVersion` is `apps/v1`. Whereas the field was originally used only to specify the API version, it is now also used to specify the API group to which the

resource belongs. Node objects belong to the core API group, which is conventionally omitted from the `apiVersion` field.

The type of object defined in the manifest is specified by the field `kind`. The object kind in the previous manifest is `Node`. In the previous chapters, you created objects of kind Deployment, Service, and Pod.

FIELDS IN THE OBJECT METADATA SECTION

The `metadata` section contains the metadata of this object instance. It contains the `name` of the instance, along with additional attributes such as `labels` and `annotations`, which are explained in chapter 9, and fields such as `resourceVersion`, `managedFields`, and other low-level fields, which are explained at depth in chapter 12.

FIELDS IN THE SPEC SECTION

Next comes the `spec` section, which is specific to each object kind. It is relatively short for Node objects compared to what you find for other object kinds. The `podCIDR` fields specify the pod IP range assigned to the node. Pods running on this node are assigned IPs from this range. The `taints` field is not important at this point, but you'll learn about it in chapter 18.

Typically, an object's spec section contains many more fields that you use to configure the object.

FIELDS IN THE STATUS SECTION

The `status` section also differs between the different kinds of object, but its purpose is always the same - it contains the last observed state of the thing the object represents. For Node objects, the status reveals the node's IP address(es), host name, capacity to provide compute resources, the current conditions of the node, the container images it has already downloaded and which are now cached locally, and information about its operating system and the version of Kubernetes components running on it.

4.2.2 Understanding individual object fields

To learn more about individual fields in the manifest, you can refer to the API reference documentation at http://kubernetes.io/docs/reference/ or use the `kubectl explain` command as described next.

USING KUBECTL EXPLAIN TO EXPLORE API OBJECT FIELDS

The kubectl tool has a nice feature that allows you to look up the explanation of each field for each object type (kind) from the command line. Usually, you start by asking it to provide the basic description of the object kind by running `kubectl explain <kind>`, as shown here:

```
$ kubectl explain nodes
KIND:     Node
VERSION:  v1

DESCRIPTION:
     Node is a worker node in Kubernetes. Each node will have a unique
     identifier in the cache (i.e. in etcd).

FIELDS:
   apiVersion   <string>
     APIVersion defines the versioned schema of this representation of an
     object. Servers should convert recognized schemas to the latest...

   kind <string>
     Kind is a string value representing the REST resource this object
     represents. Servers may infer this from the endpoint the client...

   metadata     <Object>
     Standard object's metadata. More info: ...

   spec <Object>
     Spec defines the behavior of a node...

   status       <Object>
     Most recently observed status of the node. Populated by the system.
     Read-only. More info: ...
```

The command prints the explanation of the object and lists the top-level fields that the object can contain.

DRILLING DEEPER INTO AN API OBJECT'S STRUCTURE

You can then drill deeper to find subfields under each specific field. For example, you can use the following command to explain the node's `spec` field:

```
$ kubectl explain node.spec
KIND:     Node
VERSION:  v1

RESOURCE: spec <Object>

DESCRIPTION:
     Spec defines the behavior of a node.

     NodeSpec describes the attributes that a node is created with.

FIELDS:
   configSource <Object>
     If specified, the source to get node configuration from The
     DynamicKubeletConfig feature gate must be enabled for the Kubelet...

   externalID   <string>
     Deprecated. Not all kubelets will set this field...

   podCIDR      <string>
     PodCIDR represents the pod IP range assigned to the node.
```

Please note the API version given at the top. As explained earlier, multiple versions of the same kind can exist. Different versions can have different fields or default values. If you want to display a different version, specify it with the `--api-version` option.

> **NOTE** If you want to see the complete structure of an object (the complete hierarchical list of fields without the descriptions), try `kubectl explain pods --recursive`.

4.2.3 Understanding an object's status conditions

The set of fields in both the `spec` and `status` sections is different for each object kind, but the `conditions` field is found in many of them. It gives a list of conditions the object is currently in. They are very useful when you need to troubleshoot an object, so let's examine them more closely. Since the Node object is used as an example, this section also teaches you how to easily identify problems with a cluster node.

INTRODUCING THE NODE'S STATUS CONDITIONS

Let's print out the YAML manifest of the one of the node objects again, but this time we'll only focus on the `conditions` field in the object's `status`. The command to run and its output are as follows:

```
$ kubectl get node kind-control-plane -o yaml
...
status:
  ...
  conditions:
  - lastHeartbeatTime: "2020-05-17T13:03:42Z"
    lastTransitionTime: "2020-05-03T15:09:17Z"
    message: kubelet has sufficient memory available
    reason: KubeletHasSufficientMemory
    status: "False"                                          #A
    type: MemoryPressure                                     #A
  - lastHeartbeatTime: "2020-05-17T13:03:42Z"
    lastTransitionTime: "2020-05-03T15:09:17Z"
    message: kubelet has no disk pressure
    reason: KubeletHasNoDiskPressure
    status: "False"                                          #B
    type: DiskPressure                                       #B
  - lastHeartbeatTime: "2020-05-17T13:03:42Z"
    lastTransitionTime: "2020-05-03T15:09:17Z"
    message: kubelet has sufficient PID available
    reason: KubeletHasSufficientPID
    status: "False"                                          #C
    type: PIDPressure                                        #C
  - lastHeartbeatTime: "2020-05-17T13:03:42Z"
    lastTransitionTime: "2020-05-03T15:10:15Z"
    message: kubelet is posting ready status
    reason: KubeletReady
    status: "True"                                #D
    type: Ready                                   #D
```

#A Node is not running out of memory
#B Node is not running out of disk space
#C Node has not run out of unused process Ids

#D Node is ready

> **TIP** The `jq` tool is very handy if you want to see only a part of the object's structure. For example, to display the node's status conditions, you can run `kubectl get node <name> -o json | jq .status.conditions`. The equivalent tool for YAML is `yq`.

There are four conditions that reveal the state of the node. Each condition has a `type` and a `status` field, which can be `True`, `False` or `Unknown`, as shown in the figure 4.6. A condition can also specify a machine-facing `reason` for the last transition of the condition and a human-facing `message` with details about the transition. The `lastTransitionTime` field indicates when the condition moved from one status to another, whereas the `lastHeartbeatTime` field reveals the last time the controller received an update on the given condition.

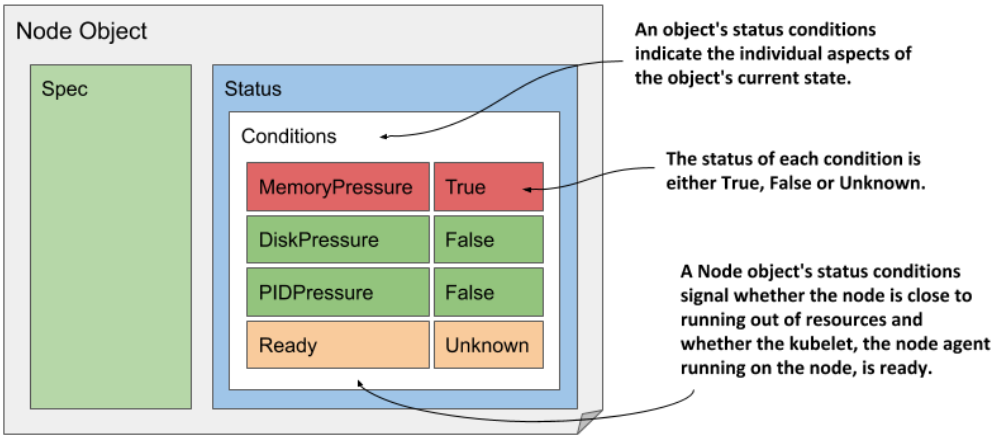

Figure 4.6 The status conditions indicating the state of a Node object

Although it's the last condition in the list, the `Ready` condition is probably the most important, as it signals whether the node is ready to accept new workloads (pods). The other conditions (`MemoryPressure`, `DiskPressure` and `PIDPressure`) signal whether the node is running out of resources. Remember to check these conditions if a node starts to behave strangely - for example, if the applications running on it start running out of resources and/or crash.

UNDERSTANDING CONDITIONS IN OTHER OBJECT KINDS

A condition list such as that in Node objects is also used in many other object kinds. The conditions explained earlier are a good example of why the state of most objects is represented by multiple conditions instead of a single field.

> **NOTE** Conditions are usually orthogonal, meaning that they represent unrelated aspects of the object.

If the state of an object were represented as a single field, it would be very difficult to subsequently extend it with new values, as this would require updating all clients that monitor the state of the object and perform actions based on it. Some object kinds originally used such a single field, and some still do, but most now use a list of conditions instead.

Since the focus of this chapter is to introduce the common features of the Kubernetes API objects, we've focused only on the `conditions` field, but it is far from being the only field in the status of the Node object. To explore the others, use the `kubectl explain` command as described in the previous sidebar. The fields that are not immediately easy for you to understand should become clear to you after reading the remaining chapters in this part of the book.

> **NOTE** As an exercise, use the command `kubectl get <kind> <name> -o yaml` to explore the other objects you've created so far (deployments, services, and pods).

4.2.4 Inspecting objects using the kubectl describe command

To give you a correct impression of the entire structure of the Kubernetes API objects, it was necessary to show you the complete YAML manifest of an object. While I personally often use this method to inspect an object, a more user-friendly way to inspect an object is the `kubectl describe` command, which typically displays the same information or sometimes even more.

UNDERSTANDING THE KUBECTL DESCRIBE OUTPUT FOR A NODE OBJECT

Let's try running the `kubectl describe` command on a Node object. To keep things interesting, let's use it to describe one of the worker nodes instead of the master. This is the command and its output:

```
$ kubectl describe node kind-worker-2
Name:               kind-worker2
Roles:              <none>
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=kind-worker2
                    kubernetes.io/os=linux
Annotations:        kubeadm.alpha.kubernetes.io/cri-socket: /run/contain...
                    node.alpha.kubernetes.io/ttl: 0
                    volumes.kubernetes.io/controller-managed-attach-deta...
CreationTimestamp:  Sun, 03 May 2020 17:09:48 +0200
Taints:             <none>
Unschedulable:      false
Lease:
  HolderIdentity:  kind-worker2
  AcquireTime:     <unset>
  RenewTime:       Sun, 17 May 2020 16:15:03 +0200
Conditions:
  Type             Status  ...  Reason                       Message
  ----             ------  ---  ------                       -------
  MemoryPressure   False   ...  KubeletHasSufficientMemory   ...
  DiskPressure     False   ...  KubeletHasNoDiskPressure     ...
  PIDPressure      False   ...  KubeletHasSufficientPID      ...
  Ready            True    ...  KubeletReady                 ...
```

```
Addresses:
  InternalIP:  172.18.0.4
  Hostname:    kind-worker2
Capacity:
  cpu:                8
  ephemeral-storage:  401520944Ki
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             32720824Ki
  pods:               110
Allocatable:
  ...
System Info:
  ...
PodCIDR:                      10.244.1.0/24
PodCIDRs:                     10.244.1.0/24
Non-terminated Pods:          (2 in total)
  Namespace     Name              CPU Requests  CPU Limits  ...  AGE
  ---------     ----              ------------  ----------  ...  ---
  kube-system   kindnet-4xmjh     100m (1%)     100m (1%)   ...  13d
  kube-system   kube-proxy-dgkfm  0 (0%)        0 (0%)      ...  13d
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests   Limits
  --------           --------   ------
  cpu                100m (1%)  100m (1%)
  memory             50Mi (0%)  50Mi (0%)
  ephemeral-storage  0 (0%)     0 (0%)
  hugepages-1Gi      0 (0%)     0 (0%)
  hugepages-2Mi      0 (0%)     0 (0%)
Events:
  Type    Reason                   Age    From                       Message
  ----    ------                   ----   ----                       -------
  Normal  Starting                 3m50s  kubelet, kind-worker2      ...
  Normal  NodeAllocatableEnforced  3m50s  kubelet, kind-worker2      ...
  Normal  NodeHasSufficientMemory  3m50s  kubelet, kind-worker2      ...
  Normal  NodeHasNoDiskPressure    3m50s  kubelet, kind-worker2      ...
  Normal  NodeHasSufficientPID     3m50s  kubelet, kind-worker2      ...
  Normal  Starting                 3m49s  kube-proxy, kind-worker2   ...
```

As you can see, the `kubectl describe` command displays all the information you previously found in the YAML manifest of the Node object, but in a more readable form. You can see the name, IP address, and hostname, as well as the conditions and available capacity of the node.

INSPECTING OTHER OBJECTS RELATED TO THE NODE

In addition to the information stored in the Node object itself, the `kubectl describe` command also displays the pods running on the node and the total amount of compute resources allocated to them. Below is also a list of events related to the node.

This additional information isn't found in the Node object itself but is collected by the kubectl tool from other API objects. For example, the list of pods running on the node is obtained by retrieving Pod objects via the `pods` resource.

If you run the `describe` command yourself, no events may be displayed. This is because only events that have occurred recently are shown. For Node objects, unless the node has resource capacity issues, you'll only see events if you've recently (re)started the node.

Virtually every API object kind has events associated with it. Since they are crucial for debugging a cluster, they warrant a closer look before you start exploring other objects.

4.3 Observing cluster events via Event objects

As controllers perform their task of reconciling the actual state of an object with the desired state, as specified in the object's `spec` field, they generate events to reveal what they have done. Two types of events exist: Normal and Warning. Events of the latter type are usually generated by controllers when something prevents them from reconciling the object. By monitoring this type of events, you can be quickly informed of any problems that the cluster encounters.

4.3.1 Introducing the Event object

Like everything else in Kubernetes, events are represented by Event objects that are created and read via the Kubernetes API. As the following figure shows, they contain information about what happened to the object and what the source of the event was. Unlike other objects, each Event object is deleted one hour after its creation to reduce the burden on etcd, the data store for Kubernetes API objects.

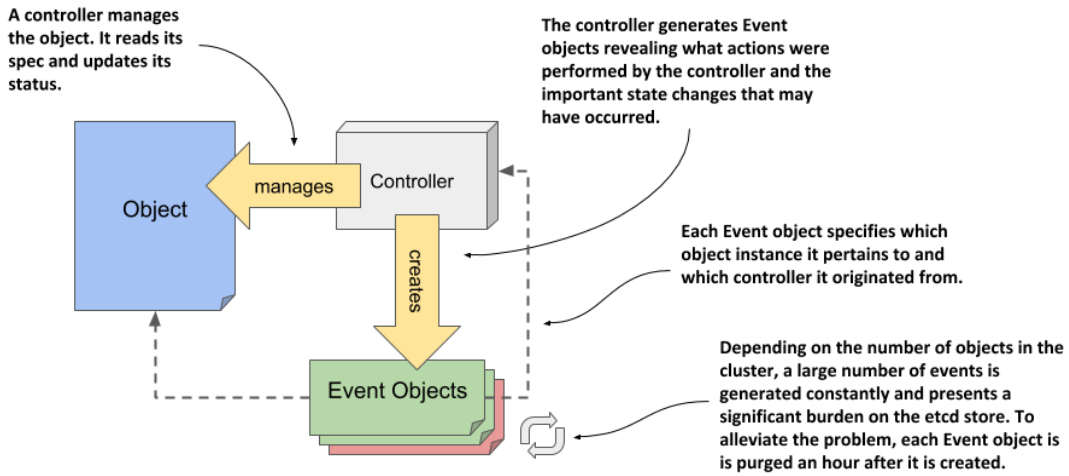

Figure 4.7 The relationship between Event objects, controllers, and other API objects.

NOTE The amount of time to retain events is configurable via the API server's command-line options.

LISTING EVENTS USING KUBECTL GET EVENTS

The events displayed by `kubectl describe` refer to the object you specify as the argument to the command. Due to their nature and the fact that many events can be created for an object

in a short time, they aren't part of the object itself. You won't find them in the object's YAML manifest, as they exist on their own, just like Nodes and the other objects you've seen so far.

> **NOTE** If you want to follow the exercises in this section in your own cluster, you may need to restart one of the nodes to ensure that the events are recent enough to still be present in etcd. If you can't do this, don't worry, and just skip doing these exercises yourself, as you'll also be generating and inspecting events in the exercises in the next chapter.

Because Events are standalone objects, you can list them using `kubectl get events`:

```
$ kubectl get ev
LAST
SEEN   TYPE    REASON                    OBJECT             MESSAGE
48s    Normal  Starting                  node/kind-worker2  Starting kubelet.
48s    Normal  NodeAllocatableEnforced   node/kind-worker2  Updated Node A...
48s    Normal  NodeHasSufficientMemory   node/kind-worker2  Node kind-work...
48s    Normal  NodeHasNoDiskPressure     node/kind-worker2  Node kind-work...
48s    Normal  NodeHasSufficientPID      node/kind-worker2  Node kind-work...
47s    Normal  Starting                  node/kind-worker2  Starting kube-...
```

> **NOTE** The previous listing uses the short name `ev` in place of `events`.

You'll notice that some events displayed in the listing match the status conditions of the Node. This is often the case, but you'll also find additional events. The two events with the reason `Starting` are two such examples. In the case at hand, they signal that the Kubelet and the Kube Proxy components have been started on the node. You don't need to worry about these components yet. They are explained in the third part of the book.

UNDERSTANDING WHAT'S IN AN EVENT OBJECT

As with other objects, the `kubectl get` command only outputs the most important object data. To display additional information, you can enable additional columns by executing the command with the `-o wide` option:

```
$ kubectl get ev -o wide
```

The output of this command is extremely wide and is not listed here in the book. Instead, the information that is displayed is explained in the following table.

Table 4.1 Properties of the Event object

Property	Description
Name	The name of this Event object instance. Useful only if you want to retrieve the given object from the API.
Type	The type of the event. Either `Normal` or `Warning`.
Reason	The machine-facing description why the event occurred.
Source	The component that reported this event. This is usually a controller.
Object	The object instance to which the event refers. For example, node/xyz.
Sub-object	The sub-object to which the event refers. For example, what container of the pod.
Message	The human-facing description of the event.
First seen	The first time this event occurred. Remember that each Event object is deleted after a while, so this may not be the first time that the event actually occurred.
Last seen	Events often occur repeatedly. This field indicates when this event last occurred.
Count	The number of times this event has occurred.

TIP As you complete the exercises throughout this book, you may find it useful to run the `kubectl get events` command each time you make changes to one of your objects. This will help you learn what happens beneath the surface.

DISPLAYING ONLY WARNING EVENTS

Unlike the `kubectl describe` command, which only displays events related to the object you're describing, the `kubectl get events` command displays all events. This is useful if you want to check if there are events that you should be concerned about. You may want to ignore events of type `Normal` and focus only on those of type `Warning`.

The API provides a way to filter objects through a mechanism called field selectors. Only objects where the specified field matches the specified selector value are returned. You can use this to display only Warning events. The `kubectl get` command allows you to specify the field selector with the `--field-selector` option. To list only events that represent warnings, you execute the following command:

```
$ kubectl get ev --field-selector type=Warning
No resources found in default namespace.
```

If the command does not print any events, as in the above case, no warnings have been recorded in your cluster recently.

You may wonder how I knew the exact name of the field to be used in the field selector and what its exact value should be (perhaps it should have been lower case, for example). Hats off if you guessed that this information is provided by the `kubectl explain events` command. Since events are regular API objects, you can use it to look up documentation on the event objects' structure. There you'll learn that the `type` field can have two values: either `Normal` or `Warning`.

4.3.2 Examining the YAML of the Event object

To inspect the events in your cluster, the commands `kubectl describe` and `kubectl get events` should be sufficient. Unlike other objects, you'll probably never have to display the complete YAML of an Event object. But I'd like to take this opportunity to show you an annoying thing about Kubernetes object manifests that the API returns.

EVENT OBJECTS HAVE NO SPEC AND STATUS SECTIONS

If you use the `kubectl explain` to explore the structure of the Event object, you'll notice that it has no `spec` or `status` sections. Unfortunately, this means that its fields are not as nicely organized as in the Node object, for example.

Inspect the following YAML and see if you can easily find the object's `kind`, `metadata`, and other fields.

```
apiVersion: v1                                                    #A
count: 1
eventTime: null
firstTimestamp: "2020-05-17T18:16:40Z"
involvedObject:
  kind: Node
  name: kind-worker2
  uid: kind-worker2
kind: Event                                                       #B
lastTimestamp: "2020-05-17T18:16:40Z"
message: Starting kubelet.
metadata:                                                         #C
  creationTimestamp: "2020-05-17T18:16:40Z"
  managedFields:
  - ...
  name: kind-worker2.160fe38fc0bc3703                             #D
  namespace: default
  resourceVersion: "3528471"
  selfLink: /api/v1/namespaces/default/events/kind-worker2.160f...
  uid: da97e812-d89e-4890-9663-091fd1ec5e2d
reason: Starting
reportingComponent: ""
reportingInstance: ""
source:
  component: kubelet
  host: kind-worker2
type: Normal
```

#A The apiVersion field is easy to spot
#B The kind field is hard to find
#C The object's metadata appears in the metadata section, which begins here
#D The object's name is hidden here

You will surely agree that the YAML manifest in the listing is disorganized. The fields are listed alphabetically instead of being organized into coherent groups. This makes it difficult for us humans to read. It looks so chaotic that it's no wonder that many people hate to deal with Kubernetes YAML or JSON manifests, since both suffer from this problem.

In contrast, the earlier YAML manifest of the Node object was relatively easy to read, because the order of the top-level fields is what one would expect: `apiVersion`, `kind`,

`metadata`, `spec`, and `status`. You'll notice that this is simply because the alphabetical order of the five fields just happens to make sense. But the fields under those fields suffer from the same problem, as they are also sorted alphabetically.

YAML is supposed to be easy for people to read, but the alphabetical field order in Kubernetes YAML breaks this. Fortunately, most objects contain the `spec` and `status` sections, so at least the top-level fields in these objects are well organized. As for the rest, you'll just have to accept this unfortunate aspect of dealing with Kubernetes manifests.

4.4 Summary

In this chapter, you've learned:

- Kubernetes provides a RESTful API for interaction with a cluster. API Objects map to actual components that make up the cluster, including applications, load balancers, nodes, storage volumes, and many others.
- An object instance can be represented by many resources. A single object type can be exposed through several resources that are just different representations of the same thing.
- Kubernetes API objects are described in YAML or JSON manifests. Objects are created by posting a manifest to the API. The status of the object is stored in the object itself and can be retrieved by requesting the object from the API with a `GET` request.
- All Kubernetes API objects contain Type and Object Metadata, and most have a `spec` and `status` sections. A few object types don't have these two sections, because they only contain static data.
- Controllers bring objects to life by constantly watching for changes in their `spec`, updating the cluster state and reporting the current state via the object's `status` field.
- As controllers manage Kubernetes API objects, they emit events to reveal what actions they have performed. Like everything else, events are represented by Event objects and can be retrieved through the API. Events signal what is happening to a Node or other object. They show what has recently happened to the object and can provide clues as to why it is broken.
- The `kubectl explain` command provides a quick way to look up documentation on a specific object kind and its fields from the command line.
- The status in a Node object contains information about the node's IP address and hostname, its resource capacity, conditions, cached container images and other information about the node. Pods running on the node are not part of the node's status, but the `kubectl describe node` commands gets this information from the `pods` resource.
- Many object types use status conditions to signal the state of the component that the object represents. For nodes, these conditions are `MemoryPressure`, `DiskPressure` and `PIDPressure`. Each condition is either `True`, `False`, or `Unknown` and has an associated `reason` and `message` that explain why the condition is in the specified state.

You should now be familiar with the general structure of the Kubernetes API objects. In the next chapter, you'll learn about the Pod object, the fundamental building block which represents one running instance of your application.

5

Running applications in Pods

This chapter covers

- Understanding how and when to group containers
- Running an application by creating a Pod object from a YAML file
- Communicating with an application, viewing its logs, and exploring its environment
- Adding a sidecar container to extend the pod's main container
- Initializing pods by running init containers at pod startup

Let me refresh your memory with a diagram that shows the three types of objects you created in chapter 3 to deploy a minimal application on Kubernetes. Figure 5.1 shows how they relate to each other and what functions they have in the system.

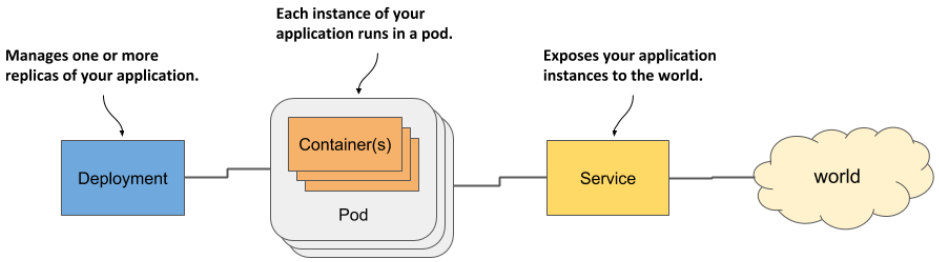

Figure 5.1 Three basic object types comprising a deployed application

You now have a basic understanding of how these objects are exposed via the Kubernetes API. In this and the following chapters, you'll learn about the specifics of each of them and many others that are typically used to deploy a full application. Let's start with the Pod object,

as it represents the central, most important concept in Kubernetes - a running instance of your application.

> **NOTE** You'll find the code files for this chapter at https://github.com/luksa/kubernetes-in-action-2nd-edition/tree/master/Chapter05

5.1 Understanding pods

You've already learned that a pod is a co-located group of containers and the basic building block in Kubernetes. Instead of deploying containers individually, you deploy and manage a group of containers as a single unit — a pod. Although pods may contain several, it's not uncommon for a pod to contain just a single container. When a pod has multiple containers, all of them run on the same worker node — a single pod instance never spans multiple nodes. Figure 5.2 will help you visualize this information.

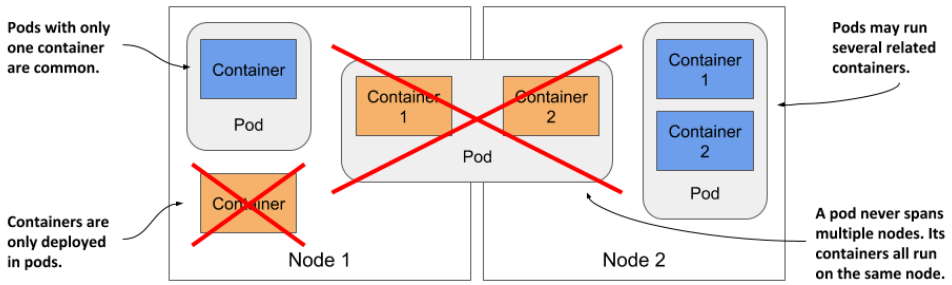

Figure 5.2 All containers of a pod run on the same node. A pod never spans multiple nodes.

5.1.1 Understanding why we need pods

Let's discuss why we need to run multiple containers together, as opposed to, for example, running multiple processes in the same container.

UNDERSTANDING WHY ONE CONTAINER SHOULDN'T CONTAIN MULTIPLE PROCESSES

Imagine an application that consists of several processes that communicate with each other via *IPC* (Inter-Process Communication) or shared files, which requires them to run on the same computer. In chapter 2, you learned that each container is like an isolated computer or virtual machine. A computer typically runs several processes; containers can also do this. You can run all the processes that make up an application in just one container, but that makes the container very difficult to manage.

Containers are *designed* to run only a single process, not counting any child processes that it spawns. Both container tooling and Kubernetes were developed around this fact. For example, a process running in a container is expected to write its logs to standard output. Docker and Kubernetes commands that you use to display the logs only show what has been captured from this output. If a single process is running in the container, it's the only writer,

but if you run multiple processes in the container, they all write to the same output. Their logs are therefore intertwined, and it's difficult to tell which process each line belongs to.

Another indication that containers should only run a single process is the fact that the container runtime only restarts the container when the container's root process dies. It doesn't care about any child processes created by this root process. If it spawns child processes, it alone is responsible for keeping all these processes running.

To take full advantage of the features provided by the container runtime, you should consider running only one process in each container.

UNDERSTANDING HOW A POD COMBINES MULTIPLE CONTAINERS

Since you shouldn't run multiple processes in a single container, it's evident you need another higher-level construct that allows you to run related processes together even when divided into multiple containers. These processes must be able to communicate with each other like processes in a normal computer. And that is why pods were introduced.

With a pod, you can run closely related processes together, giving them (almost) the same environment as if they were all running in a single container. These processes are somewhat isolated, but not completely - they share some resources. This gives you the best of both worlds. You can use all the features that containers offer, but also allow processes to work together. A pod makes these interconnected containers manageable as one unit.

In the second chapter, you learned that a container uses its own set of Linux namespaces, but it can also share some with other containers. This sharing of namespaces is exactly how Kubernetes and the container runtime combine containers into pods.

As shown in figure 5.3, all containers in a pod share the same Network namespace and thus the network interfaces, IP address(es) and port space that belong to it.

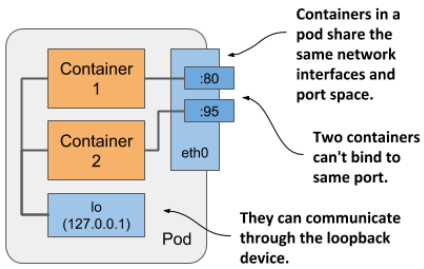

Figure 5.3 Containers in a pod share the same network interfaces

Because of the shared port space, processes running in containers of the same pod can't be bound to the same port numbers, whereas processes in other pods have their own network interfaces and port spaces, eliminating port conflicts between different pods.

All the containers in a pod also see the same system hostname, because they share the UTS namespace, and can communicate through the usual IPC mechanisms because they share the IPC namespace. A pod can also be configured to use a single PID namespace for all its containers, which makes them share a single process tree, but you must explicitly enable this for each pod individually.

NOTE When containers of the same pod use separate PID namespaces, they can't see each other or send process signals like `SIGTERM` or `SIGINT` between them.

It's this sharing of certain namespaces that gives the processes running in a pod the impression that they run together, even though they run in separate containers.

In contrast, each container always has its own Mount namespace, giving it its own file system, but when two containers must share a part of the file system, you can add a *volume* to the pod and mount it into both containers. The two containers still use two separate Mount namespaces, but the shared volume is mounted into both. You'll learn more about volumes in chapter 7.

5.1.2 Organizing containers into pods

You can think of each pod as a separate computer. Unlike virtual machines, which typically host multiple applications, you typically run only one application in each pod. You never need to combine multiple applications in a single pod, as pods have almost no resource overhead. You can have as many pods as you need, so instead of stuffing all your applications into a single pod, you should divide them so that each pod runs only closely related application processes.

Let me illustrate this with a concrete example.

SPLITTING A MULTI-TIER APPLICATION STACK INTO MULTIPLE PODS

Imagine a simple system composed of a front-end web server and a back-end database. I've already explained that the front-end server and the database shouldn't run in the same container, as all the features built into containers were designed around the expectation that not more than one process runs in a container. If not in a single container, should you then run them in separate containers that are all in the same pod?

Although nothing prevents you from running both the front-end server and the database in a single pod, this isn't the best approach. I've explained that all containers of a pod always run co-located, but do the web server and the database have to run on the same computer? The answer is obviously no, as they can easily communicate over the network. Therefore you shouldn't run them in the same pod.

If both the front-end and the back-end are in the same pod, both run on the same cluster node. If you have a two-node cluster and only create this one pod, you are using only a single worker node and aren't taking advantage of the computing resources available on the second node. This means wasted CPU, memory, disk storage and bandwidth. Splitting the containers into two pods allows Kubernetes to place the front-end pod on one node and the back-end pod on the other, thereby improving the utilization of your hardware.

SPLITTING INTO MULTIPLE PODS TO ENABLE INDIVIDUAL SCALING

Another reason not to use a single pod has to do with horizontal scaling. A pod is not only the basic unit of deployment, but also the basic unit of scaling. In chapter 2 you scaled the Deployment object and Kubernetes created additional pods – additional replicas of your application. Kubernetes doesn't replicate containers within a pod. It replicates the entire pod.

Front-end components usually have different scaling requirements than back-end components, so we typically scale them individually. When your pod contains both the front-end and back-end containers and Kubernetes replicates it, you end up with multiple instances of both the front-end and back-end containers, which isn't always what you want. Stateful back-ends, such as databases, usually can't be scaled. At least not as easily as stateless front ends. If a container has to be scaled separately from the other components, this is a clear indication that it must be deployed in a separate pod.

The following figure illustrates what was just explained.

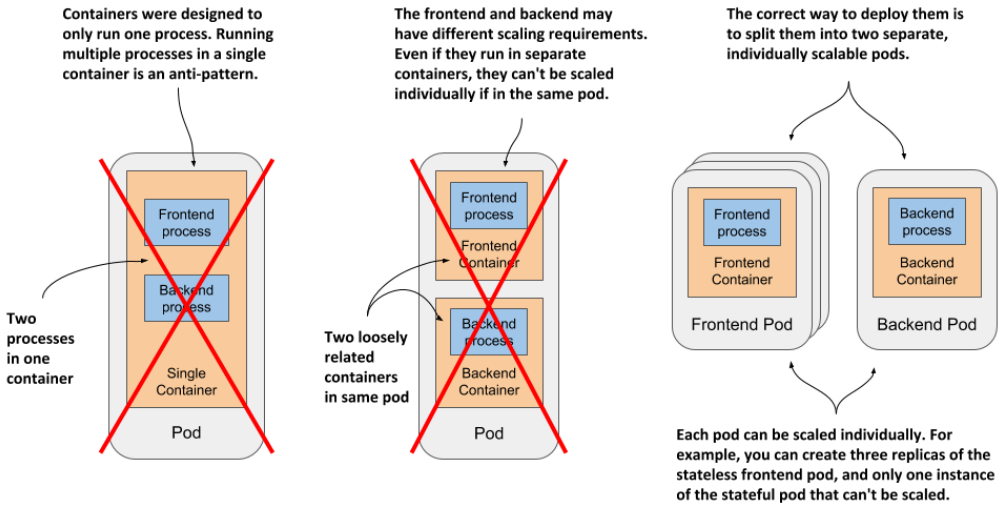

Figure 5.4 Splitting an application stack into pods

Splitting application stacks into multiple pods is the correct approach. But then, when does one run multiple containers in the same pod?

INTRODUCING SIDECAR CONTAINERS

Placing several containers in a single pod is only appropriate if the application consists of a primary process and one or more processes that complement the operation of the primary process. The container in which the complementary process runs is called a *sidecar container* because it's analogous to a motorcycle sidecar, which makes the motorcycle more stable and offers the possibility of carrying an additional passenger. But unlike motorcycles, a pod can have more than one sidecar, as shown in figure 5.5.

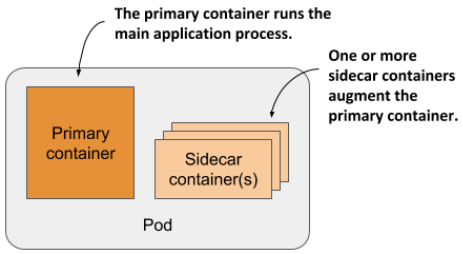

Figure 5.5 A pod with a primary and sidecar container(s)

It's difficult to imagine what constitutes a complementary process, so I'll give you some examples. In chapter 2, you deployed pods with one container that runs a Node.js application. The Node.js application only supports the HTTP protocol. To make it support HTTPS, we could add a bit more JavaScript code, but we can also do it without changing the existing application at all - by adding an additional container to the pod – a reverse proxy that converts HTTPS traffic to HTTP and forwards it to the Node.js container. The Node.js container is thus the primary container, whereas the container running the proxy is the sidecar container. Figure 5.6 shows this example.

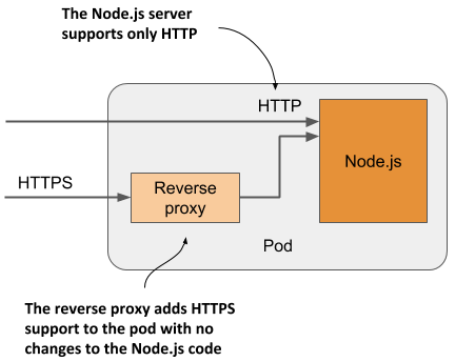

Figure 5.6 A sidecar container that converts HTTPS traffic to HTTP

NOTE You'll create this pod in section 5.4.

Another example, shown in figure 5.7, is a pod where the primary container runs a web server that serves files from its webroot directory. The other container in the pod is an agent that periodically downloads content from an external source and stores it in the web server's webroot directory. As I mentioned earlier, two containers can share files by sharing a volume. The webroot directory would be located on this volume.

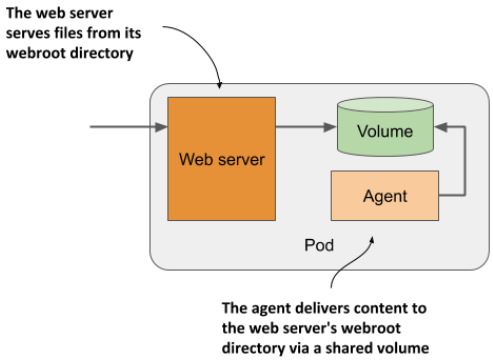

Figure 5.7 A sidecar container that delivers content to the web server container via a volume

NOTE You'll create this pod in the chapter 7.

Other examples of sidecar containers are log rotators and collectors, data processors, communication adapters, and others.

Unlike changing the application's existing code, adding a sidecar increases the pod's resources requirements because an additional process must run in the pod. But keep in mind that adding code to legacy applications can be very difficult. This could be because its code is difficult to modify, it's difficult to set up the build environment, or the source code itself is no longer available. Extending the application by adding an additional process is sometimes a cheaper and faster option.

HOW TO DECIDE WHETHER TO SPLIT CONTAINERS INTO MULTIPLE PODS

When deciding whether to use the sidecar pattern and place containers in a single pod, or to place them in separate pods, ask yourself the following questions:

- Do these containers have to run on the same host?
- Do I want to manage them as a single unit?
- Do they form a unified whole instead of being independent components?
- Do they have to be scaled together?
- Can a single node meet their combined resource needs?

If the answer to all these questions is yes, put them all in the same pod. As a rule of thumb, always place containers in separate pods unless a specific reason requires them to be part of the same pod.

5.2 Creating pods from YAML or JSON files

With the information you learned in the previous sections, you can now start creating pods. In chapter 3, you created them using the imperative command `kubectl create`, but pods and other Kubernetes objects are usually created by creating a JSON or YAML manifest file and posting it to the Kubernetes API, as you've already learned in the previous chapter.

NOTE The decision whether to use YAML or JSON to define your objects is yours. Most people prefer to use YAML because it's slightly more human-friendly and allows you to add comments to the object definition.

By using YAML files to define the structure of your application, you don't need shell scripts to make the process of deploying your applications repeatable, and you can keep a history of all changes by storing these files in a VCS (Version Control System). Just like you store code.

In fact, the application manifests of the exercises in this book are all stored in a VCS. You can find them on GitHub at github.com/luksa/kubernetes-in-action-2nd-edition.

5.2.1 Creating a YAML manifest for a pod

In the previous chapter you learned how to retrieve and examine the YAML manifests of existing API objects. Now you'll create an object manifest from scratch.

You'll start by creating a file called `pod.kiada.yaml` on your computer, in a location of your choosing. You can also find the file in the book's code archive. The following listing shows the contents of the file (Chapter05/pod.kiada.yaml).

Listing 5.1 A basic pod manifest file

```
apiVersion: v1    #A
kind: Pod    #B
metadata:
  name: kiada    #C
spec:
  containers:
  - name: kiada    #D
    image: luksa/kiada:0.1    #E
    ports:
    - containerPort: 8080    #F
```

#A This manifest uses the v1 API version to define the object
#B The object specified in this manifest is a pod
#C The name of the pod
#D The name of the container
#E Container image to create the container from
#F The port the app is listening on

I'm sure you'll agree that this pod manifest is much easier to understand than the mammoth of a manifest representing the Node object, which you saw in the previous chapter. But once you post this pod object manifest to the API and then read it back, it won't be much different.

The manifest in listing 5.1 is short only because it does not yet contain all the fields that a pod object gets after it is created through the API. For example, you'll notice that the `metadata` section contains only a single field and that the `status` section is completely missing. Once you create the object from this manifest, this will no longer be the case. But we'll get to that later.

Before you create the object, let's examine the manifest in detail. It uses version `v1` of the Kubernetes API to describe the object. The object kind is `Pod` and the name of the object is `kiada`. The pod consists of a single container also called `kiada`, based on the `luksa/kiada:0.1`

image. The pod definition also specifies that the application in the container listens on port `8080`.

> **TIP** Whenever you want to create a pod manifest from scratch, you can also use the following command to create the file and then edit it to add more fields: `kubectl run kiada --image=luksa/kiada:0.1 --dry-run=client -o yaml > mypod.yaml`. The `--dry-run=client` flag tells kubectl to output the definition instead of actually creating the object via the API.

The fields in the YAML file are self-explanatory, but if you want more information about each field or want to know what additional fields you can add, remember to use the `kubectl explain pods` command.

5.2.2 Creating the Pod object from the YAML file

After you've prepared the manifest file for your pod, you can now create the object by posting the file to the Kubernetes API.

CREATING OBJECTS BY APPLYING THE MANIFEST FILE TO THE CLUSTER

When you post the manifest to the API, you are directing Kubernetes to *apply* the manifest to the cluster. That's why the `kubectl` sub-command that does this is called `apply`. Let's use it to create the pod:

```
$ kubectl apply -f pod.kiada.yaml
pod "kiada" created
```

UPDATING OBJECTS BY MODIFYING THE MANIFEST FILE AND RE-APPLYING IT

The `kubectl apply` command is used for creating objects as well as for making changes to existing objects. If you later decide to make changes to your pod object, you can simply edit the `pod.kiada.yaml` file and run the `apply` command again. Some of the pod's fields aren't mutable, so the update may fail, but you can always delete the pod and re-create it. You'll learn how to delete pods and other objects at the end of this chapter.

Retrieving the full manifest of a running pod

The pod object is now part of the cluster configuration. You can now read it back from the API to see the full object manifest with the following command:

```
$ kubectl get po kiada -o yaml
```

If you run this command, you'll notice that the manifest has grown considerably compared to the one in the `pod.kiada.yaml` file. You'll see that the `metadata` section is now much bigger, and the object now has a `status` section. The `spec` section has also grown by several fields. You can use `kubectl explain` to learn more about these new fields, but most of them will be explained in this and the following chapters.

5.2.3 Checking the newly created pod

Let's use the basic `kubectl` commands to see how the pod is doing before we start interacting with the application running inside it.

QUICKLY CHECKING THE STATUS OF A POD

Your Pod object has been created, but how do you know if the container in the pod is actually running? You can use the `kubectl get` command to see a summary of the pod:

```
$ kubectl get pod kiada
NAME     READY   STATUS    RESTARTS   AGE
kiada    1/1     Running   0          32s
```

You can see that the pod is running, but not much else. To see more, you can try the `kubectl get pod -o wide` or the `kubectl describe` command that you learned in the previous chapter.

USING KUBECTL DESCRIBE TO SEE POD DETAILS

To display a more detailed view of the pod, use the `kubectl describe` command:

```
$ kubectl describe pod kiada
Name:         kiada
Namespace:    default
Priority:     0
Node:         worker2/172.18.0.4
Start Time:   Mon, 27 Jan 2020 12:53:28 +0100
...
```

The listing doesn't show the entire output, but if you run the command yourself, you'll see virtually all information that you'd see if you print the complete object manifest using the `kubectl get -o yaml` command.

INSPECTING EVENTS TO SEE WHAT HAPPENS BENEATH THE SURFACE

As in the previous chapter where you used the `describe node` command to inspect a Node object, the `describe pod` command should display several events related to the pod at the bottom of the output.

If you remember, these events aren't part of the object itself, but are separate objects. Let's print them to learn more about what happens when you create the pod object. These are the events that were logged after the pod was created:

```
$ kubectl get events
LAST SEEN   TYPE     REASON      OBJECT      MESSAGE
<unknown>   Normal   Scheduled   pod/kiada   Successfully assigned default/
                                             kiada to kind-worker2
5m          Normal   Pulling     pod/kiada   Pulling image luksa/kiada:0.1
5m          Normal   Pulled      pod/kiada   Successfully pulled image
5m          Normal   Created     pod/kiada   Created container kiada
5m          Normal   Started     pod/kiada   Started container kiada
```

These events are printed in chronological order. The most recent event is at the bottom. You see that the pod was first assigned to one of the worker nodes, then the container image was pulled, then the container was created and finally started.

No warning events are displayed, so everything seems to be fine. If this is not the case in your cluster, you should read section 5.4 to learn how to troubleshoot pod failures.

5.3 Interacting with the application and the pod

Your container is now running. In this section, you'll learn how to communicate with the application, inspect its logs, and execute commands in the container to explore the application's environment. Let's confirm that the application running in the container responds to your requests.

5.3.1 Sending requests to the application in the pod

In chapter 2, you used the `kubectl expose` command to create a service that provisioned a load balancer so you could talk to the application running in your pod(s). You'll now take a different approach. For development, testing and debugging purposes, you may want to communicate directly with a specific pod, rather than using a service that forwards connections to randomly selected pods.

You've learned that each pod is assigned its own IP address where it can be accessed by every other pod in the cluster. This IP address is typically internal to the cluster. You can't access it from your local computer, except when Kubernetes is deployed in a specific way – for example, when using kind or Minikube without a VM to create the cluster.

In general, to access pods, you must use one of the methods described in the following sections. First, let's determine the pod's IP address.

GETTING THE POD'S IP ADDRESS

You can get the pod's IP address by retrieving the pod's full YAML and searching for the `podIP` field in the `status` section. Alternatively, you can display the IP with `kubectl describe`, but the easiest way is to use `kubectl get` with the `wide` output option:

```
$ kubectl get pod kiada -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP           NODE     ...
kiada   1/1     Running   0          35m   10.244.2.4   worker2  ...
```

As indicated in the IP column, my pod's IP is `10.244.2.4`. Now I need to determine the port number the application is listening on.

GETTING THE PORT NUMBER USED BY THE APPLICATION

If I wasn't the author of the application, it would be difficult for me to find out which port the application listens on. I could inspect its source code or the Dockerfile of the container image, as the port is usually specified there, but I might not have access to either. If someone else had created the pod, how would I know which port it was listening on?

Fortunately, you can specify a list of ports in the pod definition itself. It isn't necessary to specify any ports, but it is a good idea to always do so. See sidebar for details.

Why specify container ports in pod definitions

Specifying ports in the pod definition is purely informative. Their omission has no effect on whether clients can connect to the pod's port. If the container accepts connections through a port bound to its IP address, anyone can connect to it, even if the port isn't explicitly specified in the pod spec or if you specify an incorrect port number.

Despite this, it's a good idea to always specify the ports so that anyone who has access to your cluster can see which ports each pod exposes. By explicitly defining ports, you can also assign a name to each port, which is very useful when you expose pods via services.

The pod manifest says that the container uses port 8080, so you now have everything you need to talk to the application.

CONNECTING TO THE POD FROM THE WORKER NODES

The Kubernetes network model dictates that each pod is accessible from any other pod and that each *node* can reach any pod on any node in the cluster.

Because of this, one way to communicate with your pod is to log into one of your worker nodes and talk to the pod from there. You've already learned that the way you log on to a node depends on what you used to deploy your cluster. If you're using kind, run `docker exec -it kind-worker bash`, or `minikube ssh` if you're using Minikube. On GKE use the `gcloud compute ssh` command. For other clusters refer to their documentation.

Once you have logged into the node, use the `curl` command with the pod's IP and port to access your application. My pod's IP is 10.244.2.4 and the port is 8080, so I run the following command:

```
$ curl 10.244.2.4:8080
Kiada version 0.1. Request processed by "kiada". Client IP: ::ffff:10.244.2.1
```

Normally you don't use this method to talk to your pods, but you may need to use it if there are communication issues and you want to find the cause by first trying the shortest possible communication route. In this case, it's best to log into the node where the pod is located and run `curl` from there. The communication between it and the pod takes place locally, so this method always has the highest chances of success.

CONNECTING FROM A ONE-OFF CLIENT POD

The second way to test the connectivity of your application is to run `curl` in another pod that you create specifically for this task. Use this method to test if other pods will be able to access your pod. Even if the network works perfectly, this may not be the case. In chapter 24, you'll learn how to lock down the network by isolating pods from each other. In such a system, a pod can only talk to the pods it's allowed to.

To run `curl` in a one-off pod, use the following command:

```
$ kubectl run --image=tutum/curl -it --restart=Never --rm client-pod curl 10.244.2.4:8080
Kiada version 0.1. Request processed by "kiada". Client IP: ::ffff:10.244.2.5
pod "client-pod" deleted
```

This command runs a pod with a single container created from the `tutum/curl` image. You can also use any other image that provides the `curl` binary executable. The `-it` option attaches your console to the container's standard input and output, the `--restart=Never` option ensures that the pod is considered Completed when the `curl` command and its container terminate, and the `--rm` options removes the pod at the end. The name of the pod is `client-pod` and the command executed in its container is `curl 10.244.2.4:8080`.

> **NOTE** You can also modify the command to run the `bash` shell in the client pod and then run `curl` from the shell.

Creating a pod just to see if it can access another pod is useful when you're specifically testing pod-to-pod connectivity. If you only want to know if your pod is responding to requests, you can also use the method explained in the next section.

CONNECTING TO PODS VIA KUBECTL PORT FORWARDING

During development, the easiest way to talk to applications running in your pods is to use the `kubectl port-forward` command, which allows you to communicate with a specific pod through a proxy bound to a network port on your local computer, as shown in the next figure.

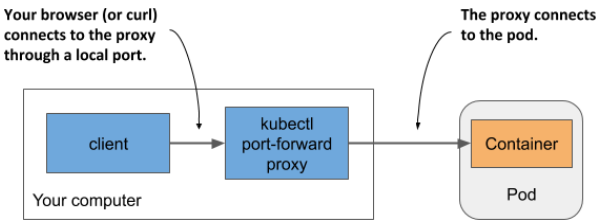

Figure 5.8 Connecting to a pod through the kubectl port-forward proxy

To open a communication path with a pod, you don't even need to look up the pod's IP, as you only need to specify its name and the port. The following command starts a proxy that forwards your computer's local port `8080` to the `kiada` pod's port `8080`:

```
$ kubectl port-forward kiada 8080
... Forwarding from 127.0.0.1:8080 -> 8080
... Forwarding from [::1]:8080 -> 8080
```

The proxy now waits for incoming connections. Run the following `curl` command in another terminal:

```
$ curl localhost:8080
Kiada version 0.1. Request processed by "kiada". Client IP: ::ffff:127.0.0.1
```

As you can see, `curl` has connected to the local proxy and received the response from the pod. While the `port-forward` command is the easiest method for communicating with a specific pod during development and troubleshooting, it's also the most complex method in terms of what happens underneath. Communication passes through several components, so if

anything is broken in the communication path, you won't be able to talk to the pod, even if the pod itself is accessible via regular communication channels.

> **NOTE** The `kubectl port-forward` command can also forward connections to services instead of pods and has several other useful features. Run `kubectl port-forward --help` to learn more.

Figure 5.9 shows how the network packets flow from the `curl` process to your application and back.

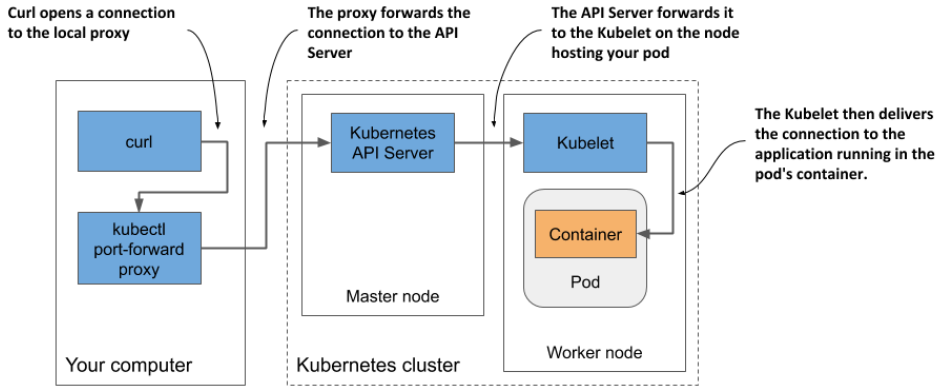

Figure 5.9 The long communication path between curl and the container when using port forwarding

As shown in the figure, the `curl` process connects to the proxy, which connects to the API server, which then connects to the Kubelet on the node that hosts the pod, and the Kubelet then connects to the container through the pod's loopback device (in other words, through the localhost address). I'm sure you'll agree that the communication path is exceptionally long.

> **NOTE** The application in the container must be bound to a port on the loopback device for the Kubelet to reach it. If it listens only on the pod's `eth0` network interface, you won't be able to reach it with the `kubectl port-forward` command.

5.3.2 Viewing application logs

Your Node.js application writes its log to the standard output stream. Instead of writing the log to a file, containerized applications usually log to the standard output (*stdout*) and standard error streams (*stderr*). This allows the container runtime to intercept the output, store it in a consistent location (usually `/var/log/containers`) and provide access to the log without having to know where each application stores its log files.

When you run an application in a container using Docker, you can display its log with `docker logs <container-id>`. When you run your application in Kubernetes, you could log into the node that hosts the pod and display its log using `docker logs`, but Kubernetes provides an easier way to do this with the `kubectl logs` command.

RETRIEVING A POD'S LOG WITH KUBECTL LOGS

To view the log of your pod (more specifically, the container's log), run the following command:

```
$ kubectl logs kiada
Kiada - Kubernetes in Action Demo Application
---------------------------------------------
Kiada 0.1 starting...
Local hostname is kiada
Listening on port 8080
Received request for / from ::ffff:10.244.2.1    #A
Received request for / from ::ffff:10.244.2.5    #B
Received request for / from ::ffff:127.0.0.1     #C
```

#A Request you sent from within the node
#B Request from the one-off client pod
#C Request sent through port forwarding

STREAMING LOGS USING KUBECTL LOGS -F

If you want to stream the application log in real-time to see each request as it comes in, you can run the command with the `--follow` option (or the shorter version `-f`):

```
$ kubectl logs kiada -f
```

Now send some additional requests to the application and have a look at the log. Press ctrl-C to stop streaming the log when you're done.

DISPLAYING THE TIMESTAMP OF EACH LOGGED LINE

You may have noticed that we forgot to include the timestamp in the log statement. Logs without timestamps have limited usability. Fortunately, the container runtime attaches the current timestamp to every line produced by the application. You can display these timestamps by using the `--timestamps=true` option as follows:

```
$ kubectl logs kiada --timestamps=true
2020-02-01T09:44:40.954641934Z Kiada - Kubernetes in Action Demo Application
2020-02-01T09:44:40.954843234Z ---------------------------------------------
2020-02-01T09:44:40.955032432Z Kiada 0.1 starting...
2020-02-01T09:44:40.955123432Z Local hostname is kiada
2020-02-01T09:44:40.956435431Z Listening on port 8080
2020-02-01T09:50:04.978043089Z Received request for / from ...
2020-02-01T09:50:33.640897378Z Received request for / from ...
2020-02-01T09:50:44.781473256Z Received request for / from ...
```

> **TIP** You can display timestamps by only typing `--timestamps` without the value. For boolean options, merely specifying the option name sets the option to `true`. This applies to all kubectl options that take a Boolean value and default to `false`.

DISPLAYING RECENT LOGS

The previous feature is great if you run third-party applications that don't include the timestamp in their log output, but the fact that each line is timestamped brings us another benefit: filtering log lines by time. Kubectl provides two ways of filtering the logs by time.

The first option is when you want to only display logs from the past several seconds, minutes or hours. For example, to see the logs produced in the last two minutes, run:

```
$ kubectl logs kiada --since=2m
```

The other option is to display logs produced after a specific date and time using the `--since-time` option. The time format to be used is RFC3339. For example, the following command is used to print logs produced after February 1st, 2020 at 9:50 a.m.:

```
$ kubectl logs kiada --since-time=2020-02-01T09:50:00Z
```

DISPLAYING THE LAST SEVERAL LINES OF THE LOG

Instead of using time to constrain the output, you can also specify how many lines from the end of the log you want to display. To display the last ten lines, try:

```
$ kubectl logs kiada --tail=10
```

> **NOTE** Kubectl options that take a value can be specified with an equal sign or with a space. Instead of `--tail=10`, you can also type `--tail 10`.

UNDERSTANDING THE AVAILABILITY OF THE POD'S LOGS

Kubernetes keeps a separate log file for each container. They are usually stored in `/var/log/containers` on the node that runs the container. A separate file is created for each container. If the container is restarted, its logs are written to a new file. Because of this, if the container is restarted while you're following its log with `kubectl logs -f`, the command will terminate, and you'll need to run it again to stream the new container's logs.

The `kubectl logs` command displays only the logs of the current container. To view the logs from the previous container, use the `--previous` (or `-p`) option.

> **NOTE** Depending on your cluster configuration, the log files may also be rotated when they reach a certain size. In this case, `kubectl logs` will only display the current log file. When streaming the logs, you must restart the command to switch to the new file when the log is rotated.

When you delete a pod, all its log files are also deleted. To make pods' logs available permanently, you need to set up a central, cluster-wide logging system. Chapter 23 explains how.

WHAT ABOUT APPLICATIONS THAT WRITE THEIR LOGS TO FILES?

If your application writes its logs to a file instead of stdout, you may be wondering how to access that file. Ideally, you'd configure the centralized logging system to collect the logs so you can view them in a central location, but sometimes you just want to keep things simple and don't mind accessing the logs manually. In the next two sections, you'll learn how to copy log and other files from the container to your computer and in the opposite direction, and how to run commands in running containers. You can use either method to display the log files or any other file inside the container.

5.3.3 Copying files to and from containers

Sometimes you may want to add a file to a running container or retrieve a file from it. Modifying files in running containers isn't something you normally do - at least not in production - but it can be useful during development.

Kubectl offers the `cp` command to copy files or directories from your local computer to a container of any pod or from the container to your computer. For example, if you'd like to modify the HTML file that the `kiada` pod serves, you can use to following command to copy it to your local file system:

```
$ kubectl cp kiada:html/index.html /tmp/index.html
```

This command copies the file `/html/index.html` file from the pod named `kiada` to the `/tmp/index.html` file on your computer. You can now edit the file locally. Once you're happy with the changes, copy the file back to the container with the following command:

```
$ kubectl cp /tmp/index.html kiada:html/
```

Hitting refresh **in** your browser should now include the changes you've made.

> **NOTE** The `kubectl cp` command requires the `tar` binary to be present in your container, but this requirement may change in the future.

5.3.4 Executing commands in running containers

When debugging an application running in a container, it may be necessary to examine the container and its environment from the inside. Kubectl provides this functionality, too. You can execute any binary file present in the container's file system using the `kubectl exec` command.

INVOKING A SINGLE COMMAND IN THE CONTAINER

For example, you can list the processes running in the container in the `kiada` pod by running the following command:

```
$ kubectl exec kiada -- ps aux
USER  PID %CPU %MEM    VSZ   RSS TTY STAT START TIME COMMAND
root    1  0.0  1.3 812860 27356 ?   Ssl  11:54 0:00 node app.js #A
root  120  0.0  0.1  17500  2128 ?   Rs   12:22 0:00 ps aux      #B
```

#A The Node.js server
#B The command you've just invoked

This is the Kubernetes equivalent of the Docker command you used to explore the processes in a running container in chapter 2. It allows you to remotely run a command in any pod without having to log in to the node that hosts the pod. If you've used `ssh` to execute commands on a remote system, you'll see that `kubectl exec` is not much different.

In section 5.3.1 you executed the `curl` command in a one-off client pod to send a request to your application, but you can also run the command inside the `kiada` pod itself:

```
$ kubectl exec kiada -- curl -s localhost:8080
Kiada version 0.1. Request processed by "kiada". Client IP: ::1
```

Why use a double dash in the kubectl exec command?

The double dash (`--`) in the command delimits kubectl arguments from the command to be executed in the container. The use of the double dash isn't necessary if the command has no arguments that begin with a dash. If you omit the double dash in the previous example, the `-s` option is interpreted as an option for `kubectl exec` and results in the following misleading error:

```
$ kubectl exec kiada curl -s localhost:8080
The connection to the server localhost:8080 was refused – did you specify the right host or port?
```

This may look like the Node.js server is refusing to accept the connection, but the issue lies elsewhere. The curl command is never executed. The error is reported by `kubectl` itself when it tries to talk to the Kubernetes API server at `localhost:8080`, which isn't where the server is. If you run the `kubectl options` command, you'll see that the `-s` option can be used to specify the address and port of the Kubernetes API server. Instead of passing that option to curl, kubectl adopted it as its own. Adding the double dash prevents this.

Fortunately, to prevent scenarios like this, newer versions of kubectl are set to return an error if you forget the double dash.

RUNNING AN INTERACTIVE SHELL IN THE CONTAINER

The two previous examples showed how a single command can be executed in the container. When the command completes, you are returned to your shell. If you want to run several commands in the container, you can run a shell in the container as follows:

```
$ kubectl exec -it kiada -- bash
root@kiada:/#         #A
```

#A The command prompt of the shell running in the container

The `-it` is short for two options: `-i` and `-t`, which indicate that you want to execute the `bash` command interactively by passing the standard input to the container and marking it as a terminal (TTY).

You can now explore the inside of the container by executing commands in the shell. For example, you can view the files in the container by running `ls -la`, view its network interfaces with `ip link`, or test its connectivity with `ping`. You can run any tool available in the container.

NOT ALL CONTAINERS ALLOW YOU TO RUN SHELLS

The container image of your application contains many important debugging tools, but this isn't the case with every container image. To keep images small and improve security in the container, most containers used in production don't contain any binary files other than those required for the container's primary process. This significantly reduces the attack surface, but also means that you can't run shells or other tools in production containers. Fortunately, a new Kubernetes feature called *ephemeral containers* allows you to debug running containers by attaching a debug container to them.

NOTE TO MEAP READERS Ephemeral containers are currently an alpha feature, which means they may change or even be removed at any time. This is also why they are currently not explained in this book. If they graduate to beta before the book goes into production, a section explaining them will be added.

5.3.5 Attaching to a running container

The `kubectl attach` command is another way to interact with a running container. It attaches itself to the standard input, output and error streams of the main process running in the container. Normally, you only use it to interact with applications that read from the standard input.

USING KUBECTL ATTACH TO SEE WHAT THE APPLICATION PRINTS TO STANDARD OUTPUT

If the application doesn't read from standard input, the `kubectl attach` command is no more than an alternative way to stream the application logs, as these are typically written to the standard output and error streams, and the `attach` command streams them just like the `kubectl logs -f` command does.

Attach to your `kiada` pod by running the following command:

```
$ kubectl attach kiada
Defaulting container name to kiada.
Use 'kubectl describe pod/kiada -n default' to see all of the containers in this pod.
If you don't see a command prompt, try pressing enter.
```

Now, when you send new HTTP requests to the application using `curl` in another terminal, you'll see the lines that the application logs to standard output also printed in the terminal where the `kubectl attach` command is executed.

USING KUBECTL ATTACH TO WRITE TO THE APPLICATION'S STANDARD INPUT

The Kiada application version 0.1 doesn't read from the standard input stream, but you'll find the source code of version 0.2 that does this in the book's code archive. This version allows you to set a status message by writing it to the standard input stream of the application. This status message will be included in the application's response. Let's deploy this version of the application in a new pod and use the `kubectl attach` command to set the status message.

You can find the artifacts required to build the image in the `kiada-0.2/` directory. You can also use the pre-built image `docker.io/luksa/kiada:0.2`. The pod manifest is shown in the following listing (Chapter05/pod.kiada-stdin.yaml). It contains one additional line compared to the previous manifest (it's highlighted in the listing).

Listing 5.2 Enabling standard input for a container

```
apiVersion: v1
kind: Pod
metadata:
  name: kiada-stdin    #A
spec:
  containers:
  - name: kiada
    image: luksa/kiada:0.2    #B
    stdin: true    #C
    ports:
    - containerPort: 8080
```

#A This pod is named kiada-stdin
#B It uses the 0.2 version of the Kiada application
#C The application needs to read from the standard input stream

As you can see in the listing, if the application running in a pod wants to read from standard input, you must indicate this in the pod manifest by setting the `stdin` field in the container definition to `true`. This tells Kubernetes to allocate a buffer for the standard input stream, otherwise the application will always receive an `EOF` when it tries to read from it.

Create the pod from this manifest file with the `kubectl apply` command:

```
$ kubectl apply -f pod.kiada-stdin.yaml
pod/kiada-stdin created
```

To enable communication with the application, use the `kubectl port-forward` command again, but because the local port `8080` is still being used by the previously executed `port-forward` command, you must either terminate it or choose a different local port to forward to the new pod. You can do this as follows:

```
$ kubectl port-forward kiada-stdin 8888:8080
Forwarding from 127.0.0.1:8888 -> 8080
Forwarding from [::1]:8888 -> 8080
```

The command-line argument `8888:8080` instructs the command to forward local port `8888` to the pod's port `8080`.

You can now reach the application at http://localhost:8888:

```
$ curl localhost:8888
Kiada version 0.2. Request processed by "kiada-stdin". Client IP: ::ffff:127.0.0.1
```

Let's set the status message by using `kubectl attach` to write to the standard input stream of the application. Run the following command:

```
$ kubectl attach -i kiada-stdin
```

Note the use of the additional option `-i` in the command. It instructs `kubectl` to pass its standard input to the container.

> **NOTE** Like the `kubectl exec` command, `kubectl attach` also supports the `--tty` or `-t` option, which indicates that the standard input is a terminal (TTY), but the container must be configured to allocate a terminal through the `tty` field in the container definition.

You can now enter the status message into the terminal and press the ENTER key. For example, type the following message:

```
This is my custom status message.
```

The application prints the new message to the standard output:

```
Status message set to: This is my custom status message.
```

To see if the application now includes the message in its responses to HTTP requests, re-execute the `curl` command or refresh the page in your web browser:

```
$ curl localhost:8888
Kiada version 0.2. Request processed by "kiada-stdin". Client IP: ::ffff:127.0.0.1
This is my custom status message.    #A
```

#A Here's the message you set via the kubectl attach command.

You can change the status message again by typing another line in the terminal running the `kubectl attach` command. To exit the `attach` command, press Control-C or the equivalent key.

> **NOTE** An additional field in the container definition, `stdinOnce`, determines whether the standard input channel is closed when the attach session ends. It's set to `false` by default, which allows you to use the standard input in every `kubectl attach` session. If you set it to `true`, standard input remains open only during the first session.

5.4 Running multiple containers in a pod

The Kiada application you deployed in section 5.2 only supports HTTP. Let's add TLS support so it can also serve clients over HTTPS. You could do this by adding code to the `app.js` file, but an easier option exists where you don't need to touch the code at all.

You can run a reverse proxy alongside the Node.js application in a sidecar container, as explained in section 5.1.2, and let it handle HTTPS requests on behalf of the application. A very popular software package that can provide this functionality is *Envoy*. The Envoy proxy is a high-performance open source service proxy originally built by Lyft that has since been contributed to the Cloud Native Computing Foundation. Let's add it to your pod.

5.4.1 Extending the Kiada Node.js application using the Envoy proxy

Let me briefly explain what the new architecture of the application will look like. As shown in the next figure, the pod will have two containers - the Node.js and the new Envoy container. The Node.js container will continue to handle HTTP requests directly, but the HTTPS requests will be handled by Envoy. For each incoming HTTPS request, Envoy will create a new HTTP

request that it will then send to the Node.js application via the local loopback device (via the localhost IP address).

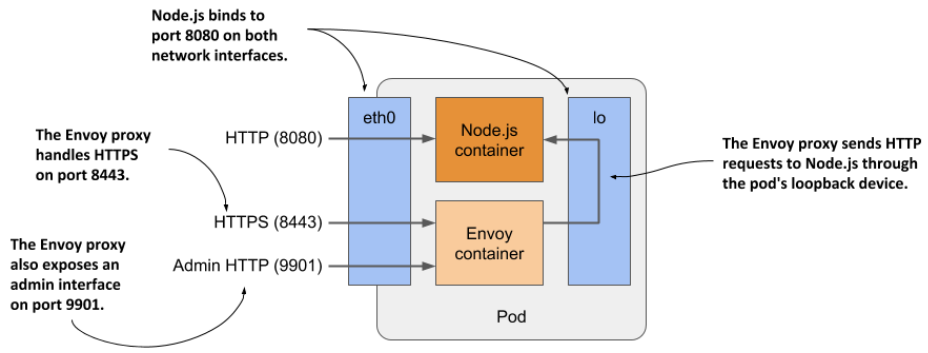

Figure 5.10 Detailed view of the pod's containers and network interfaces

Envoy also provides a web-based administration interface that will prove handy in some of the exercises in the next chapter.

It's obvious that if you implement TLS support within the Node.js application itself, the application will consume less computing resources and have lower latency because no additional network hop is required, but adding the Envoy proxy could be a faster and easier solution. It also provides a good starting point from which you can add many other features provided by Envoy that you would probably never implement in the application code itself. Refer to the Envoy proxy documentation at envoyproxy.io to learn more.

5.4.2 Adding Envoy proxy to the pod

You'll create a new pod with two containers. You've already got the Node.js container, but you also need a container that will run Envoy.

CREATING THE ENVOY CONTAINER IMAGE

The authors of the proxy have published the official Envoy proxy container image at Docker Hub. You could use this image directly, but you would need to somehow provide the configuration, certificate, and private key files to the Envoy process in the container. You'll learn how to do this in chapter 7. For now, you'll use an image that already contains all three files.

I've already created the image and made it available at `docker.io/luksa/kiada-ssl-proxy:0.1`, but if you want to build it yourself, you can find the files in the `kiada-ssl-proxy-image` directory in the book's code archive.

The directory contains the `Dockerfile`, as well as the private key and certificate that the proxy will use to serve HTTPS. It also contains the `envoy.conf` config file. In it, you'll see that the proxy is configured to listen on port `8443`, terminate TLS, and forward requests to port

`8080` on `localhost`, which is where the Node.js application is listening. The proxy is also configured to provide an administration interface on port `9901`, as explained earlier.

CREATING THE POD MANIFEST

After building the image, you must create the manifest for the new pod. The following listing shows the pod manifest file Chapter05/pod.kiada-ssl.yaml.

Listing 5.3 Manifest of pod kiada-ssl

```
apiVersion: v1
kind: Pod
metadata:
  name: kiada-ssl
spec:
  containers:
  - name: kiada                                  #A
    image: luksa/kiada:0.2                       #A
    ports:                                       #A
    - name: http                                 #A
      containerPort: 8080                        #A
  - name: envoy                                  #B
    image: luksa/kiada-ssl-proxy:0.1             #B
    ports:                                       #B
    - name: https                                #B
      containerPort: 8443                        #B
    - name: admin                                #B
      containerPort: 9901                        #B
```

#A The container running the Node.js server, which listens on port 8080.
#B The container running the Envoy proxy on ports 8443 (HTTPS) and 9901 (admin).

The name of this pod is `kiada-ssl`. It has two containers: `kiada` and `envoy`. The manifest is only slightly more complex than the manifest in section 5.2.1. The only new fields are the port names, which are included so that anyone reading the manifest can understand what each port number stands for.

CREATING THE POD

Create the pod from the manifest using the command `kubectl apply -f pod.kiada-ssl.yaml`. Then use the `kubectl get` and `kubectl describe` commands to confirm that the pod's containers were successfully launched.

5.4.3 Interacting with the two-container pod

When the pod starts, you can start using the application in the pod, inspect its logs and explore the containers from within.

COMMUNICATING WITH THE APPLICATION

As before, you can use the `kubectl port-forward` to enable communication with the application in the pod. Because it exposes three different ports, you enable forwarding to all three ports as follows:

```
$ kubectl port-forward kiada-ssl 8080 8443 9901
```

```
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
Forwarding from 127.0.0.1:8443 -> 8443
Forwarding from [::1]:8443 -> 8443
Forwarding from 127.0.0.1:9901 -> 9901
Forwarding from [::1]:9901 -> 9901
```

First, confirm that you can communicate with the application via HTTP by opening the URL http://localhost:8080 in your browser or by using `curl`:

```
$ curl localhost:8080
Kiada version 0.2. Request processed by "kiada-ssl". Client IP: ::ffff:127.0.0.1
```

If this works, you can also try to access the application over HTTPS at https://localhost:8443. With `curl` you can do this as follows:

```
$ curl https://localhost:8443 --insecure
Kiada version 0.2. Request processed by "kiada-ssl". Client IP: ::ffff:127.0.0.1
```

Success! The Envoy proxy handles the task perfectly. Your application now supports HTTPS using a sidecar container.

Why it is necessary to use the --insecure option

There are two reasons why you must use the `--insecure` option when accessing the service. The certificate used by the Envoy proxy is self-signed and was issued for the domain name `example.com`. You access the service via the local `kubectl proxy` and use `localhost` as the domain name in the URL, which means that it doesn't match the name in the server certificate. To make it match, you'd have to use the following command:

```
$ curl https://example.com:8443 --resolve example.com:8443:127.0.0.1
```

This ensures that the certificate matches the requested URL, but because the certificate is self-signed, `curl` still can't verify the legitimacy of the server. You must either replace the server's certificate with a certificate signed by a trusted authority or use the `--insecure` flag anyway; in this case, you also don't need to bother with using the `--resolve` flag.

DISPLAYING LOGS OF PODS WITH MULTIPLE CONTAINERS

The `kiada-ssl` pod contains two containers, so if you want to display the logs, you must specify the name of the container using the `--container` or `-c` option. For example, to view the logs of the `kiada` container, run the following command:

```
$ kubectl logs kiada-ssl -c kiada
```

The Envoy proxy runs in the container named `envoy`, so you display its logs as follows:

```
$ kubectl logs kiada-ssl -c envoy
```

Alternatively, you can display the logs of both containers with the `--all-containers` option:

```
$ kubectl logs kiada-ssl --all-containers
```

You can also combine these commands with the other options explained in section 5.3.2.

RUNNING COMMANDS IN CONTAINERS OF MULTI-CONTAINER PODS

If you'd like to run a shell or another command in one of the pod's containers using the `kubectl exec` command, you also specify the container name using the `--container` or `-c` option. For example, to run a shell inside the envoy container, run the following command:

```
$ kubectl exec -it kiada-ssl -c envoy -- bash
```

> **NOTE** If you don't provide the name, `kubectl exec` defaults to the first container specified in the pod manifest.

5.5 Running additional containers at pod startup

When a pod contains more than one container, all the containers are started in parallel. Kubernetes doesn't yet provide a mechanism to specify whether a container depends on another container, which would allow you to ensure that one is started before the other. However, Kubernetes allows you to run a sequence of containers to initialize the pod before its main containers start. This special type of container is explained in this section.

5.5.1 Introducing init containers

A pod manifest can specify a list of containers to run when the pod starts and before the pod's normal containers are started. These containers are intended to initialize the pod and are appropriately called *init containers*. As the following figure shows, they run one after the other and must all finish successfully before the main containers of the pod are started.

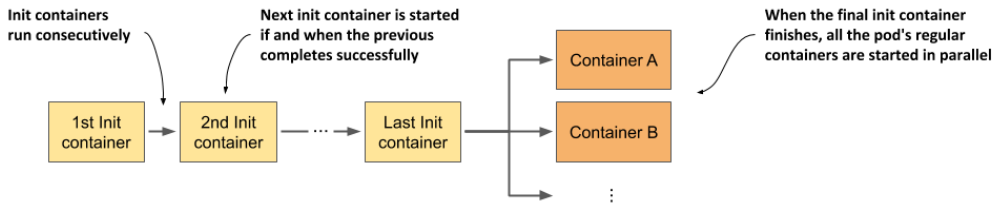

Figure 5.11 Time sequence showing how a pod's init and regular containers are started

Init containers are like the pod's regular containers, but they don't run in parallel - only one init container runs at a time.

UNDERSTANDING WHAT INIT CONTAINERS CAN DO

Init containers are typically added to pods to achieve the following:

- Initialize files in the volumes used by the pod's main containers. This includes retrieving certificates and private keys used by the main container from secure certificate stores,

generating config files, downloading data, and so on.
- Initialize the pod's networking system. Because all containers of the pod share the same network namespaces, and thus the network interfaces and configuration, any changes made to it by an init container also affect the main container.
- Delay the start of the pod's main containers until a precondition is met. For example, if the main container relies on another service being available before the container is started, an init container can block until this service is ready.
- Notify an external service that the pod is about to start running. In special cases where an external system must be notified when a new instance of the application is started, an init container can be used to deliver this notification.

You could perform these operations in the main container itself but using an init container is sometimes a better option and can have other advantages. Let's see why.

UNDERSTANDING WHEN MOVING INITIALIZATION CODE TO INIT CONTAINERS MAKES SENSE

Using an init container to perform initialization tasks doesn't require the main container image to be rebuilt and allows a single init container image to be reused with many different applications. This is especially useful if you want to inject the same infrastructure-specific initialization code into all your pods. Using an init container also ensures that this initialization is complete before any of the (possibly multiple) main containers start.

Another important reason is security. By moving tools or data that could be used by an attacker to compromise your cluster from the main container to an init container, you reduce the pod's attack surface.

For example, imagine that the pod must be registered with an external system. The pod needs some sort of secret token to authenticate against this system. If the registration procedure is performed by the main container, this secret token must be present in its filesystem. If the application running in the main container has a vulnerability that allows an attacker to read arbitrary files on the filesystem, the attacker may be able to obtain this token. By performing the registration from an init container, the token must be available only in the filesystem of the init container, which an attacker can't easily compromise.

5.5.2 Adding init containers to a pod

In a pod manifest, init containers are defined in the `initContainers` field in the spec section, just as regular containers are defined in its `containers` field.

DEFINING INIT CONTAINERS IN A POD MANIFEST

Let's look at an example of adding two init containers to the kiada pod. The first init container emulates an initialization procedure. It runs for 5 seconds, while printing a few lines of text to standard output.

The second init container performs a network connectivity test by using the `ping` command to check if a specific IP address is reachable from within the pod. The IP address is configurable via a command-line argument which defaults to 1.1.1.1.

NOTE An init container that checks if specific IP addresses are reachable could be used to block an application from starting until the services it depends on become available.

You'll find the `Dockerfile`s and other artifacts for both images in the book's code archive, if you want to build them yourself. Alternatively, you can use the images that I've built.

A pod manifest containing these two init containers is shown in the following listing (file Chapter05/pod.kiada-init.yaml).

Listing 5.4 Defining init containers in a pod manifest

```
apiVersion: v1
kind: Pod
metadata:
  name: kiada-init
spec:
  initContainers:                                    #A
  - name: init-demo                                  #B
    image: luksa/init-demo:0.1                       #B
  - name: network-check                              #C
    image: luksa/network-connectivity-checker:0.1    #C
  containers:                                        #D
  - name: kiada                                      #D
    image: luksa/kiada:0.2                           #D
    stdin: true                                      #D
    ports:                                           #D
    - name: http                                     #D
      containerPort: 8080                            #D
  - name: envoy                                      #D
    image: luksa/kiada-ssl-proxy:0.1                 #D
    ports:                                           #D
    - name: https                                    #D
      containerPort: 8443                            #D
    - name: admin                                    #D
      containerPort: 9901                            #D
```

#A Init containers are specified in the initContainers field
#B This container runs first
#C This container runs after the first one completes
#D These are the pod's regular containers. They run at the same time.

As you can see, the definition of an init container is almost trivial. It's sufficient to specify only the `name` and `image` for each container.

NOTE Container names must be unique within the union of all init and regular containers.

DEPLOYING A POD WITH INIT CONTAINERS

Before you create the pod from the manifest file, run the following command in a separate terminal so you can see how the pod's status changes as the init and regular containers start:

```
$ kubectl get pods -w
```

You'll also want to watch events in another terminal using the following command:

```
$ kubectl get events -w
```

When ready, create the pod by running the apply command:

```
$ kubectl apply -f pod.kiada-init.yaml
```

INSPECTING THE STARTUP OF A POD WITH INIT CONTAINERS

As the pod starts up, inspect the events that are shown by the `kubectl get events -w` command:

```
TYPE     REASON      MESSAGE
Normal   Scheduled   Successfully assigned pod to worker2
Normal   Pulling     Pulling image "luksa/init-demo:0.1"       #A
Normal   Pulled      Successfully pulled image                 #A
Normal   Created     Created container init-demo               #A
Normal   Started     Started container init-demo               #A
Normal   Pulling     Pulling image "luksa/network-connec...    #B
Normal   Pulled      Successfully pulled image                 #B
Normal   Created     Created container network-check           #B
Normal   Started     Started container network-check           #B
Normal   Pulled      Container image "luksa/kiada:0.1"         #C
                     already present on machine                #C
Normal   Created     Created container kiada                   #C
Normal   Started     Started container kiada                   #C
Normal   Pulled      Container image "luksa/kiada-ssl-         #C
                     proxy:0.1" already present on machine     #C
Normal   Created     Created container envoy                   #C
Normal   Started     Started container envoy                   #C
```

#A The first init container's image is pulled and the container is started
#B After the first init container completes, the second is started
#C The pod's two main containers are then started in parallel

The listing shows the order in which the containers are started. The `init-demo` container is started first. When it completes, the `network-check` container is started, and when it completes, the two main containers, `kiada` and `envoy`, are started.

Now inspect the transitions of the pod's status in the other terminal. They should look like this:

```
NAME          READY   STATUS            RESTARTS   AGE
kiada-init    0/2     Pending           0          0s
kiada-init    0/2     Pending           0          0s
kiada-init    0/2     Init:0/2          0          0s          #A
kiada-init    0/2     Init:0/2          0          1s          #A
kiada-init    0/2     Init:1/2          0          6s          #B
kiada-init    0/2     PodInitializing   0          7s          #C
kiada-init    2/2     Running           0          8s          #D
```

#A The first init container is running
#B The first init container is complete, the second is now running
#C All init containers have completed successfully
#D The pod's main containers are running

As the listing shows, when the init containers run, the pod's status shows the number of init containers that have completed and the total number. When all init containers are done, the pod's status is displayed as `PodInitializing`. At this point, the images of the main containers are pulled. When the containers start, the status changes to `Running`.

5.5.3 Inspecting init containers

As with regular containers, you can run additional commands in a running init container using `kubectl exec` and display the logs using `kubectl logs`.

DISPLAYING THE LOGS OF AN INIT CONTAINER

The standard and error output, into which each init container can write, are captured exactly as they are for regular containers. The logs of an init container can be displayed using the `kubectl logs` command by specifying the name of the container with the `-c` option either while the container runs or after it has completed. To display the logs of the `network-check` container in the `kiada-init` pod, run the next command:

```
$ kubectl logs kiada-init -c network-check
Checking network connectivity to 1.1.1.1 ...
Host appears to be reachable
```

The logs show that the `network-check` init container ran without errors. In the next chapter, you'll see what happens if an init container fails.

ENTERING A RUNNING INIT CONTAINER

You can use the `kubectl exec` command to run a shell or a different command inside an init container the same way you can with regular containers, but you can only do this before the init container terminates. If you'd like to try this yourself, create a pod from the `pod.kiada-init-slow.yaml` file, which makes the `init-demo` container run for 60 seconds. When the pod starts, run a shell in the container with the following command:

```
$ kubectl exec -it kiada-init-slow -c init-demo -- sh
```

You can use the shell to explore the container from the inside, but only for a short time. When the container's main process exits after 60 seconds, the shell process is also terminated.

You typically enter a running init container only when it fails to complete in time, and you want to find the cause. During normal operation, the init container terminates before you can run the `kubectl exec` command.

5.6 Deleting pods and other objects

If you've tried the exercises in this chapter and in chapter 2, several pods and other objects now exist in your cluster. To close this chapter, you'll learn various ways to delete them. Deleting a pod will terminate its containers and remove them from the node. Deleting a Deployment object causes the deletion of its pods, whereas deleting a LoadBalancer-typed Service deprovisions the load balancer if one was provisioned.

5.6.1 Deleting a pod by name

The easiest way to delete an object is to delete it by name.

DELETING A SINGLE POD

Use the following command to remove the `kiada` pod from your cluster:

```
$ kubectl delete po kiada
pod "kiada" deleted
```

By deleting a pod, you state that you no longer want the pod or its containers to exist. The Kubelet shuts down the pod's containers, removes all associated resources, such as log files, and notifies the API server after this process is complete. The Pod object is then removed.

> **TIP** By default, the `kubectl delete` command waits until the object no longer exists. To skip the wait, run the command with the `--wait=false` option.

While the pod is in the process of shutting down, its status changes to `Terminating`:

```
$ kubectl get po kiada
NAME    READY   STATUS        RESTARTS   AGE
kiada   1/1     Terminating   0          35m
```

Knowing exactly how containers are shut down is important if you want your application to provide a good experience for its clients. This is explained in the next chapter, where we dive deeper into the life cycle of the pod and its containers.

> **NOTE** If you're familiar with Docker, you may wonder if you can stop a pod and start it again later, as you can with Docker containers. The answer is no. With Kubernetes, you can only remove a pod completely and create it again later.

DELETING MULTIPLE PODS WITH A SINGLE COMMAND

You can also delete multiple pods with a single command. If you ran the `kiada-init` and the `kiada-init-slow` pods, you can delete them both by specifying their names separated by a space, as follows:

```
$ kubectl delete po kiada-init kiada-init-slow
pod "kiada-init" deleted
pod "kiada-init-slow" deleted
```

5.6.2 Deleting objects defined in manifest files

Whenever you create objects from a file, you can also delete them by passing the file to the `delete` command instead of specifying the name of the pod.

DELETING OBJECTS BY SPECIFYING THE MANIFEST FILE

You can delete the `kiada-ssl` pod, which you created from the `pod.kiada-ssl.yaml` file, with the following command:

```
$ kubectl delete -f pod.kiada-ssl.yaml
pod "kiada-ssl" deleted
```

In your case, the file contains only a single pod object, but you'll typically come across files that contain several objects of different types that represent a complete application. This makes deploying and removing the application as easy as executing `kubectl apply -f app.yaml` and `kubectl delete -f app.yaml`, respectively.

DELETING OBJECTS FROM MULTIPLE MANIFEST FILES

Sometimes, an application is defined in several manifest files. You can specify multiple files by separating them with a comma. For example:

```
$ kubectl delete -f pod.kiada.yaml,pod.kiada-ssl.yaml
```

> **NOTE** You can also apply several files at the same time using this syntax (for example: `kubectl apply -f pod.kiada.yaml,pod.kiada-ssl.yaml`).

I've never actually used this approach in the many years I've been using Kubernetes, but I often deploy all the manifest files from a file directory by specifying the directory name instead of the names of individual files. For example, you can deploy all the pods you created in this chapter again by running the following command in the base directory of this book's code archive:

```
$ kubectl apply -f Chapter05/
```

This applies to all files in the directory that have the correct file extension (`.yaml`, `.json`, and similar). You can then delete the pods using the same method:

```
$ kubectl delete -f Chapter05/
```

> **NOTE** Use the `--recursive` flag to also scan subdirectories.

5.6.3 Deleting all pods

You've now removed all pods except `kiada-stdin` and the pods you created in chapter 3 using the `kubectl create deployment` command. Depending on how you've scaled the deployment, some of these pods should still be running:

```
$ kubectl get pods
NAME                    READY   STATUS    RESTARTS   AGE
kiada-stdin             1/1     Running   0          10m
kiada-9d785b578-58vhc   1/1     Running   0          1d
kiada-9d785b578-jmnj8   1/1     Running   0          1d
```

Instead of deleting these pods by name, we can delete them all using the `--all` option:

```
$ kubectl delete po --all
pod "kiada-stdin" deleted
pod "kiada-9d785b578-58vhc" deleted
pod "kiada-9d785b578-jmnj8" deleted
```

Now confirm that no pods exist by executing the `kubectl get pods` command again:

```
$ kubectl get po
NAME                    READY   STATUS    RESTARTS   AGE
kiada-9d785b578-cc6tk   1/1     Running   0          13s
kiada-9d785b578-h4gml   1/1     Running   0          13s
```

That was unexpected! Two pods are still running. If you look closely at their names, you'll see that these aren't the two you've just deleted. The `AGE` column also indicates that these are

new pods. You can try to delete them as well, but you'll see that no matter how often you delete them, new pods are created to replace them.

The reason why these pods keep popping up is because of the Deployment object. The controller responsible for bringing Deployment objects to life must ensure that the number of pods always matches the desired number of replicas specified in the object. When you delete a pod associated with the Deployment, the controller immediately creates a replacement pod.

To delete these pods, you must either scale the Deployment to zero or delete the object altogether. This would indicate that you no longer want this deployment or its pods to exist in your cluster.

5.6.4 Deleting objects using the "all" keyword

You can delete everything you've created so far - including the deployment, its pods, and the service - with the following command:

```
$ kubectl delete all --all
pod "kiada-9d785b578-cc6tk" deleted
pod "kiada-9d785b578-h4gml" deleted
service "kubernetes" deleted
service "kiada" deleted
deployment.apps "kiada" deleted
replicaset.apps "kiada-9d785b578" deleted
```

The first `all` in the command indicates that you want to delete objects of all types. The `--all` option indicates that you want to delete all instances of each object type. You used this option in the previous section when you tried to delete all pods.

When deleting objects, `kubectl` prints the type and name of each deleted object. In the previous listing, you should see that it deleted the pods, the deployment, and the service, but also a so-called replica set object. You'll learn what this is in chapter 11, where we take a closer look at deployments.

You'll notice that the delete command also deletes the built-in `kubernetes` service. Don't worry about this, as the service is automatically recreated after a few moments.

Certain objects aren't deleted when using this method, because the keyword `all` does not include all object kinds. This is a precaution to prevent you from accidentally deleting objects that contain important information. The Event object kind is one example of this.

> **NOTE** You can specify multiple object types in the `delete` command. For example, you can use `kubectl delete events,all --all` to delete events along with all object kinds included in `all`.

5.7 Summary

In this chapter, you've learned:

- Pods run one or more containers as a co-located group. They are the unit of deployment and horizontal scaling. A typical container runs only one process. Sidecar containers complement the primary container in the pod.
- Containers should only be part of the same pod if they must run together. A frontend and a backend process should run in separate pods. This allows them to be scaled individually.
- When a pod starts, its init containers run one after the other. When the last init container completes, the pod's main containers are started. You can use an init container to configure the pod from within, delay startup of its main containers until a precondition is met, or notify an external service that the pod is about to start running.
- The `kubectl` tool is used to create pods, view their logs, copy files to/from their containers, execute commands in those containers and enable communication with individual pods during development.

In the next chapter, you'll learn about the lifecycle of the pod and its containers.

6

Managing the lifecycle of the Pod's containers

This chapter covers

- Inspecting the pod's status
- Keeping containers healthy using liveness probes
- Using lifecycle hooks to perform actions at container startup and shutdown
- Understanding the complete lifecycle of the pod and its containers

After reading the previous chapter, you should be able to deploy, inspect and communicate with pods containing one or more containers. In this chapter, you'll gain a much deeper understanding of how the pod and its containers operate.

> **NOTE** You'll find the code files for this chapter at https://github.com/luksa/kubernetes-in-action-2nd-edition/tree/master/Chapter06

6.1 Understanding the pod's status

After you create a pod object and it runs, you can see what's going on with the pod by reading the pod object back from the API. As you've learned in chapter 4, the pod object manifest, as well as the manifests of most other kinds of objects, contain a section, which provides the status of the object. A pod's `status` section contains the following information:

- the IP addresses of the pod and the worker node that hosts it
- when the pod was started
- the pod's quality-of-service (QoS) class
- what phase the pod is in,
- the conditions of the pod, and

- the state of its individual containers.

The IP addresses and the start time don't need any further explanation, and the QoS class isn't relevant now - you'll learn about it in chapter 19. However, the phase and conditions of the pod, as well as the states of its containers are important for you to understand the pod lifecycle.

6.1.1 Understanding the pod phase

In any moment of the pod's life, it's in one of the five phases shown in the following figure.

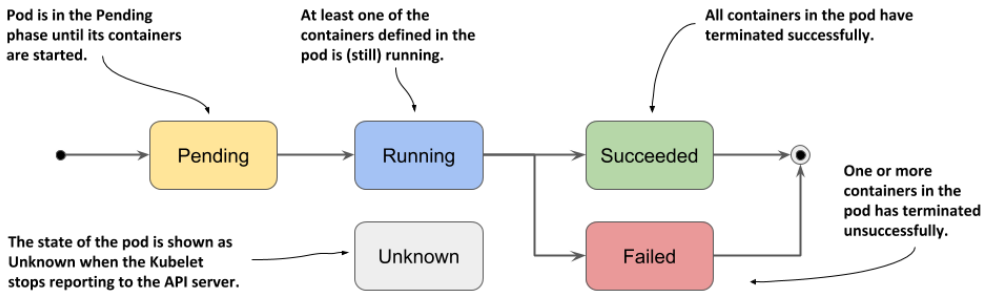

Figure 6.1 The phases of a Kubernetes pod

The meaning of each phase is explained in the following table.

Table 6.1 List of phases a pod can be in

Pod Phase	Description
`Pending`	After you create the Pod object, this is its initial phase. Until the pod is scheduled to a node and the images of its containers are pulled and started, it remains in this phase.
`Running`	At least one of the pod's containers is running.
`Succeeded`	Pods that aren't intended to run indefinitely are marked as `Succeeded` when all their containers complete successfully.
`Failed`	When a pod is not configured to run indefinitely and at least one of its containers terminates unsuccessfully, the pod is marked as Failed.
`Unknown`	The state of the pod is unknown because the Kubelet has stopped reporting communicating with the API server. Possibly the worker node has failed or has disconnected from the network.

The pod's phase provides a quick summary of what's happening with the pod. Let's deploy the `kiada` pod again and inspect its phase. Create the pod by applying the manifest file to your cluster again, as in the previous chapter (you'll find it in Chapter06/pod.kiada.yaml):

```
$ kubectl apply -f pod.kiada.yaml
```

DISPLAYING A POD'S PHASE

The pod's phase is one of the fields in the pod object's `status` section. You can see it by displaying its manifest and optionally grepping the output to search for the field:

```
$ kubectl get po kiada -o yaml | grep phase
phase: Running
```

> **TIP** Remember the `jq` tool? You can use it to print out the value of the `phase` field like this: `kubectl get po kiada -o json | jq .status.phase`

You can also see the pod's phase using `kubectl describe`. The pod's status is shown close to the top of the output.

```
$ kubectl describe po kiada
Name:         kiada
Namespace:    default
...
Status:       Running
...
```

Although it may appear that the `STATUS` column displayed by `kubectl get pods` also shows the phase, this is only true for pods that are healthy:

```
$ kubectl get po kiada
NAME    READY   STATUS    RESTARTS   AGE
kiada   1/1     Running   0          40m
```

For unhealthy pods, the `STATUS` column indicates what's wrong with the pod. You'll see this later in this chapter.

6.1.2 Understanding pod conditions

The phase of a pod says little about the condition of the pod. You can learn more by looking at the pod's list of conditions, just as you did for the node object in chapter 4. A pod's conditions indicate whether a pod has reached a certain state or not, and why that's the case.

In contrast to the phase, a pod has several conditions at the same time. Four condition *types* are known at the time of writing. They are explained in the following table.

Table 6.2 List of pod conditions

Pod Condition	Description
`PodScheduled`	Indicates whether or not the pod has been scheduled to a node.
`Initialized`	The pod's init containers have all completed successfully.
`ContainersReady`	All containers in the pod indicate that they are ready. This is a necessary but not sufficient condition for the entire pod to be ready.
`Ready`	The pod is ready to provide services to its clients. The containers in the pod and the pod's readiness gates are all reporting that they are ready. Note: this is explained in chapter 10.

Each condition is either fulfilled or not. As you can see in the following figure, the `PodScheduled` and `Initialized` conditions start as unfulfilled, but are soon fulfilled and remain so throughout the life of the pod. In contrast, the `Ready` and `ContainersReady` conditions can change many times during the pod's lifetime.

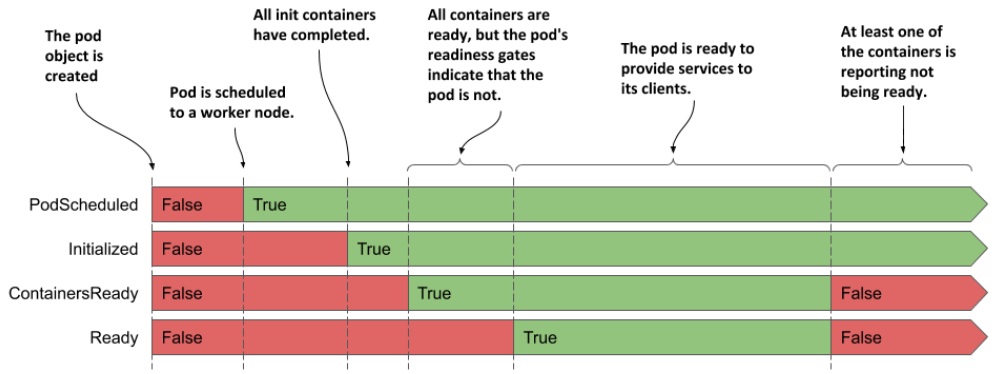

Figure 6.2 The transitions of the pod's conditions during its lifecycle

Do you remember the conditions you can find in a node object? They are `MemoryPressure`, `DiskPressure`, `PIDPressure` and `Ready`. As you can see, each object has its own set of condition types, but many contain the generic `Ready` condition, which typically indicates whether everything is fine with the object.

INSPECTING THE POD'S CONDITIONS

To see the conditions of a pod, you can use `kubectl describe` as shown here:

```
$ kubectl describe po kiada
...
Conditions:
  Type              Status
  Initialized       True    #A
  Ready             True    #B
  ContainersReady   True    #B
  PodScheduled      True    #C
...
```

#A The pod has been initialized
#B The pod and its containers are ready
#C The pod has been scheduled to a node

The `kubectl describe` command shows only whether each condition is true or not. To find out why a condition is false, you must look for the `status.conditions` field in the pod manifest as follows:

```
$ kubectl get po kiada -o json | jq .status.conditions
[
  {
    "lastProbeTime": null,
    "lastTransitionTime": "2020-02-02T11:42:59Z",
    "status": "True",
    "type": "Initialized"
  },
  ...
```

Each condition has a `status` field that indicates whether the condition is `True`, `False` or `Unknown`. In the case of the `kiada` pod, the status of all conditions is `True`, which means they are all fulfilled. The condition can also contain a `reason` field that specifies a machine-facing reason for the last change of the condition's status, and a `message` field that explains the change in detail. The `lastTransitionTime` field shows when the change occurred, while the `lastProbeTime` indicates when this condition was last checked.

6.1.3 Understanding the container status

Also contained in the status of the pod is the status of each of its containers. Inspecting the status provides better insight into the operation of each individual container.

The status contains several fields. The `state` field indicates the container's current state, whereas the `lastState` field shows the state of the previous container after it has terminated. The container status also indicates the internal ID of the container (`containerID`), the `image` and `imageID` the container is running, whether the container is `ready` or not and how often it has been restarted (`restartCount`).

UNDERSTANDING THE CONTAINER STATE

The most important part of a container's status is its `state`. A container can be in one of the states shown in the following figure.

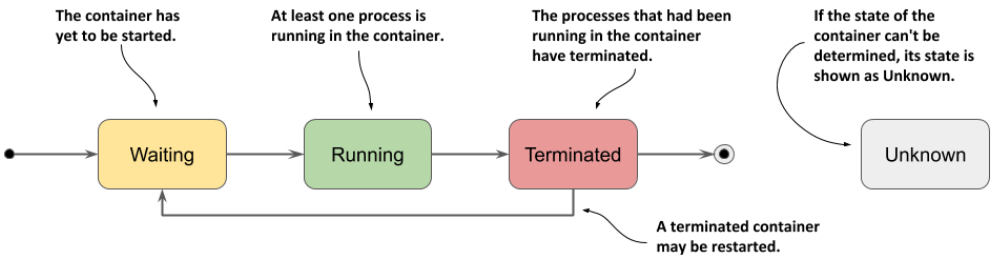

Figure 6.3 The possible states of a container

Individual states are explained in the following table.

Table 6.3 Possible container states

Container State	Description
`Waiting`	The container is waiting to be started. The `reason` and `message` fields indicate why the container is in this state.
`Running`	The container has been created and processes are running in it. The `startedAt` field indicates the time at which this container was started.
`Terminated`	The processes that had been running in the container have terminated. The `startedAt` and `finishedAt` fields indicate when the container was started and when it terminated. The exit code with which the main process terminated is in the `exitCode` field.
`Unknown`	The state of the container couldn't be determined.

DISPLAYING THE STATUS OF THE POD'S CONTAINERS

The pod list displayed by `kubectl get pods` shows only the number of containers in each pod and how many of them are ready. To see the status of individual containers, you can use `kubectl describe`:

```
$ kubectl describe po kiada
...
Containers:
  kiada:
    Container ID:   docker://c64944a684d57faacfced0be1af44686...
    Image:          luksa/kiada:0.1
    Image ID:       docker-pullable://luksa/kiada@sha256:3f28...
    Port:           8080/TCP
    Host Port:      0/TCP
    State:          Running    #A
      Started:      Sun, 02 Feb 2020 12:43:03 +0100    #A
    Ready:          True    #B
    Restart Count:  0    #C
    Environment:    <none>
...
```

#A The current state of the container and when it was started
#B Whether the container is ready to provide its services
#C How many times the container has been restarted

Focus on the annotated lines in the listing, as they indicate whether the container is healthy. The `kiada` container is `Running` and is `Ready`. It has never been restarted.

> **TIP** You can also display the container status(es) using `jq` like this: `kubectl get po kiada -o json | jq .status.containerStatuses`

INSPECTING THE STATUS OF AN INIT CONTAINER

In the previous chapter, you learned that in addition to regular containers, a pod can also have init containers that run when the pod starts. As with regular containers, the status of these

containers is available in the `status` section of the pod object manifest, but in the `initContainerStatuses` field.

Inspecting the status of the kiada-init pod

As an additional exercise you can try on your own, create the `kiada-init` pod from the previous chapter and inspect its phase, conditions and the status of its two regular and two init containers. Use the `kubectl describe` command and the `kubectl get po kiada-init -o json | jq .status` command to find the information in the object definition.

6.2 Keeping containers healthy

The pods you created in the previous chapter ran without any problems. But what if one of the containers dies? What if all the containers in a pod die? How do you keep the pods healthy and their containers running? That's the focus of this section.

6.2.1 Understanding container auto-restart

When a pod is scheduled to a node, the Kubelet on that node starts its containers and from then on keeps them running for as long as the pod object exists. If the main process in the container terminates for any reason, the Kubelet restarts the container. If an error in your application causes it to crash, Kubernetes automatically restarts it, so even without doing anything special in the application itself, running it in Kubernetes automatically gives it the ability to heal itself. Let's see this in action.

OBSERVING A CONTAINER FAILURE

In the previous chapter, you created the `kiada-ssl` pod, which contains the Node.js and the Envoy containers. Create the pod again and enable communication with the pod by running the following two commands:

```
$ kubectl apply -f pod.kiada-ssl.yaml
$ kubectl port-forward kiada-ssl 8080 8443 9901
```

You'll now cause the Envoy container to terminate to see how Kubernetes deals with the situation. Run the following command in a separate terminal so you can see how the pod's status changes when one of its containers terminates:

```
$ kubectl get pods -w
```

You'll also want to watch events in another terminal using the following command:

```
$ kubectl get events -w
```

You could emulate a crash of the container's main process by sending it the `KILL` signal, but you can't do this from inside the container because the Linux Kernel doesn't let you kill the root process (the process with PID 1). You would have to SSH to the pod's host node and kill

the process from there. Fortunately, Envoy's administration interface allows you to stop the process via its HTTP API.

To terminate the `envoy` container, open the URL http://localhost:9901 in your browser and click the *quitquitquit* button or run the following `curl` command in another terminal:

```
$ curl -X POST http://localhost:9901/quitquitquit
OK
```

To see what happens with the container and the pod it belongs to, examine the output of the `kubectl get pods -w` command you ran earlier. This is its output:

```
$ kubectl get po -w
NAME           READY   STATUS     RESTARTS   AGE
kiada-ssl      2/2     Running    0          1s
kiada-ssl      1/2     NotReady   0          9m33s
kiada-ssl      2/2     Running    1          9m34s
```

The listing shows that the pod's `STATUS` changes from `Running` to `NotReady`, while the `READY` column indicates that only one of the two containers is ready. Immediately thereafter, Kubernetes restarts the container and the pod's status returns to `Running`. The `RESTARTS` column indicates that one container has been restarted.

NOTE If one of the pod's containers fails, the other containers continue to run.

Now examine the output of the `kubectl get events -w` command you ran earlier. Here's the command and its output:

```
$ kubectl get ev -w
LAST SEEN   TYPE      REASON      OBJECT           MESSAGE
0s          Normal    Pulled      pod/kiada-ssl    Container image already
                                                   present on machine
0s          Normal    Created     pod/kiada-ssl    Created container envoy
0s          Normal    Started     pod/kiada-ssl    Started container envoy
```

The events show that the new `envoy` container has been started. You should be able to access the application via HTTPS again. Please confirm with your browser or `curl`.

The events in the listing also expose an important detail about how Kubernetes restarts containers. The second event indicates that the entire `envoy` container has been recreated. Kubernetes never restarts a container, but instead discards it and creates a new container. Regardless, we call this *restarting* a container.

NOTE Any data that the process writes to the container's filesystem is lost when the container is recreated. This behavior is sometimes undesirable. To persist data, you must add a storage volume to the pod, as explained in the next chapter.

NOTE If init containers are defined in the pod and one of the pod's regular containers is restarted, the init containers are not executed again.

CONFIGURING THE POD'S RESTART POLICY

By default, Kubernetes restarts the container regardless of whether the process in the container exits with a zero or non-zero exit code - in other words, whether the container completes successfully or fails. This behavior can be changed by setting the `restartPolicy` field in the pod's `spec`.

Three restart policies exist. They are explained in the following figure.

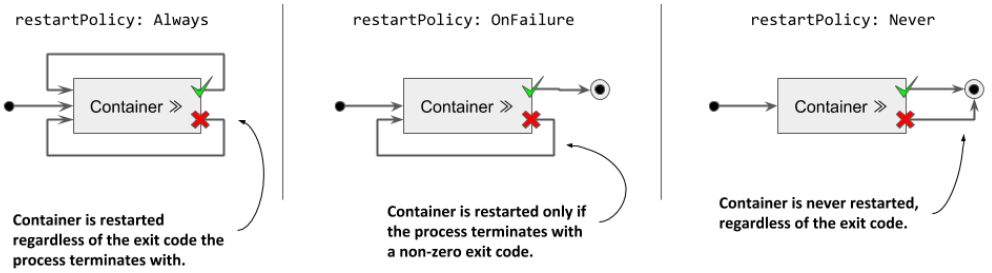

Figure 6.4 The pod's restartPolicy determines whether its containers are restarted or not

The following table describes the three restart policies.

Table 6.4 Pod restart policies

Restart Policy	Description
`Always`	Container is restarted regardless of the exit code the process in the container terminates with. This is the default restart policy.
`OnFailure`	The container is restarted only if the process terminates with a non-zero exit code, which by convention indicates failure.
`Never`	The container is never restarted - not even when it fails.

> **NOTE** Surprisingly, the restart policy is configured at the pod level and applies to all its containers. It can't be configured for each container individually.

UNDERSTANDING THE TIME DELAY INSERTED BEFORE A CONTAINER IS RESTARTED

If you call Envoy's `/quitquitquit` endpoint several times, you'll notice that each time it takes longer to restart the container after it terminates. The pod's status is displayed as either `NotReady` or `CrashLoopBackOff`. Here's what it means.

As shown in the following figure, the first time a container terminates, it is restarted immediately. The next time, however, Kubernetes waits ten seconds before restarting it again. This delay is then doubled to 20, 40, 80 and then to 160 seconds after each subsequent termination. From then on, the delay is kept at five minutes. This delay that doubles between attempts is called exponential back-off.

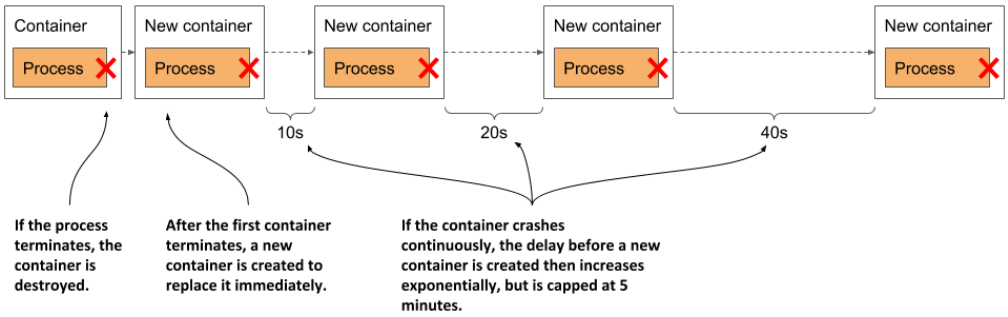

Figure 6.5 Exponential back-off between container restarts

In the worst case, a container can therefore be prevented from starting for up to five minutes.

> **NOTE** The delay is reset to zero when the container has run successfully for 10 minutes. If the container must be restarted later, it is restarted immediately.

Check the container status in the pod manifest as follows:

```
$ kubectl get po kiada-ssl -o json | jq .status.containerStatuses
...
"state": {
  "waiting": {
    "message": "back-off 40s restarting failed container=envoy ...",
    "reason": "CrashLoopBackOff"
```

As you can see in the output, while the container is waiting to be restarted, its state is `Waiting`, and the `reason` is `CrashLoopBackOff`. The `message` field tells you how long it will take for the container to be restarted.

> **NOTE** When you tell Envoy to terminate, it terminates with exit code zero, which means it hasn't crashed. The `CrashLoopBackOff` status can therefore be misleading.

6.2.2 Checking the container's health using liveness probes

In the previous section, you learned that Kubernetes keeps your application healthy by restarting it when its process terminates. But applications can also become unresponsive without terminating. For example, a Java application with a memory leak eventually starts spewing out OutOfMemoryErrors, but its JVM process continues to run. Ideally, Kubernetes should detect this kind of error and restart the container.

The application could catch these errors by itself and immediately terminate, but what about the situations where your application stops responding because it gets into an infinite loop or deadlock? What if the application can't detect this? To ensure that the application is restarted in such cases, it may be necessary to check its state from the outside.

INTRODUCING LIVENESS PROBES

Kubernetes can be configured to check whether an application is still alive by defining a *liveness probe*. You can specify a liveness probe for each container in the pod. Kubernetes runs the probe periodically to ask the application if it's still alive and well. If the application doesn't respond, an error occurs, or the response is negative, the container is considered unhealthy and is terminated. The container is then restarted if the restart policy allows it.

> **NOTE** Liveness probes can only be used in the pod's regular containers. They can't be defined in init containers.

TYPES OF LIVENESS PROBES

Kubernetes can probe a container with one of the following three mechanisms:

- An *HTTP GET* probe sends a GET request to the container's IP address, on the network port and path you specify. If the probe receives a response, and the response code doesn't represent an error (in other words, if the HTTP response code is `2xx` or `3xx`), the probe is considered successful. If the server returns an error response code, or if it doesn't respond in time, the probe is considered to have failed.
- A *TCP Socket* probe attempts to open a TCP connection to the specified port of the container. If the connection is successfully established, the probe is considered successful. If the connection can't be established in time, the probe is considered failed.
- An *Exec* probe executes a command inside the container and checks the exit code it terminates with. If the exit code is zero, the probe is successful. A non-zero exit code is considered a failure. The probe is also considered to have failed if the command fails to terminate in time.

> **NOTE** In addition to a liveness probe, a container may also have a *startup* probe, which is discussed in section 6.2.6, and a *readiness* probe, which is explained in chapter 10.

6.2.3 Creating an HTTP GET liveness probe

Let's look at how to add a liveness probe to each of the containers in the `kiada-ssl` pod. Because they both run applications that understand HTTP, it makes sense to use an HTTP GET probe in each of them. The Node.js application doesn't provide any endpoints to explicitly check the health of the application, but the Envoy proxy does. In real-world applications, you'll encounter both cases.

DEFINING LIVENESS PROBES IN THE POD MANIFEST

The following listing shows an updated manifest for the pod, which defines a liveness probe for each of the two containers, with different levels of configuration (Chapter06/pod.kiada-liveness.yaml).

Listing 6.1 Adding a liveness probe to a pod

```
apiVersion: v1
kind: Pod
metadata:
  name: kiada-liveness
spec:
  containers:
  - name: kiada
    image: luksa/kiada:0.1
    ports:
    - name: http
      containerPort: 8080
    livenessProbe:    #A
      httpGet:    #A
        path: /    #A
        port: 8080    #A
  - name: envoy
    image: luksa/kiada-ssl-proxy:0.1
    ports:
    - name: https
      containerPort: 8443
    - name: admin
      containerPort: 9901
    livenessProbe:    #B
      httpGet:    #B
        path: /ready    #B
        port: admin    #B
      initialDelaySeconds: 10    #B
      periodSeconds: 5    #B
      timeoutSeconds: 2    #B
      failureThreshold: 3    #B
```

#A The liveness probe definition for the container running Node.js
#B The liveness probe for the Envoy proxy

These liveness probes are explained in the next two sections.

DEFINING A LIVENESS PROBE USING THE MINIMUM REQUIRED CONFIGURATION

The liveness probe for the `kiada` container is the simplest version of a probe for HTTP-based applications. The probe simply sends an HTTP `GET` request for the path `/` on port `8080` to determine if the container can still serve requests. If the application responds with an HTTP status between `200` and `399`, the application is considered healthy.

The probe doesn't specify any other fields, so the default settings are used. The first request is sent 10s after the container starts and is repeated every 10s. If the application doesn't respond within one second, the probe attempt is considered failed. If it fails three times in a row, the container is considered unhealthy and is terminated.

UNDERSTANDING LIVENESS PROBE CONFIGURATION OPTIONS

The administration interface of the Envoy proxy provides the special endpoint `/ready` through which it exposes its health status. Instead of targeting port `8443`, which is the port through which Envoy forwards HTTPS requests to Node.js, the liveness probe for the `envoy` container targets this special endpoint on the `admin` port, which is port number `9901`.

NOTE As you can see in the `envoy` container's liveness probe, you can specify the probe's target port by name instead of by number.

The liveness probe for the `envoy` container also contains additional fields. These are best explained with the following figure.

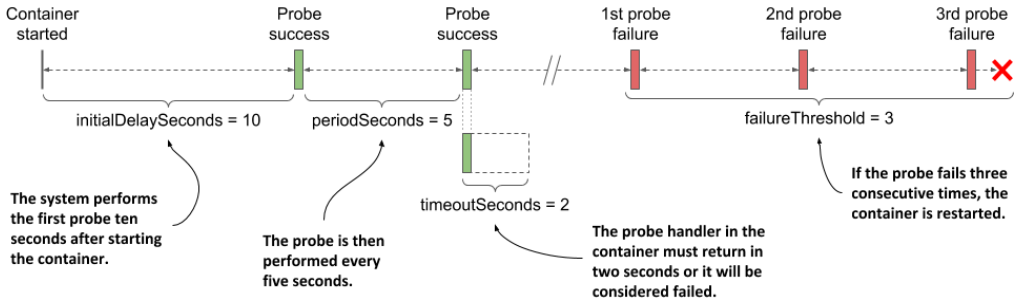

Figure 6.6 The configuration and operation of a liveness probe

The parameter `initialDelaySeconds` determines how long Kubernetes should delay the execution of the first probe after starting the container. The `periodSeconds` field specifies the amount of time between the execution of two consecutive probes, whereas the `timeoutSeconds` field specifies how long to wait for a response before the probe attempt counts as failed. The `failureThreshold` field specifies how many times the probe must fail for the container to be considered unhealthy and potentially restarted.

6.2.4 Observing the liveness probe in action

To see Kubernetes restart a container when its liveness probe fails, create the pod from the pod.kiada-liveness.yaml manifest file using `kubectl apply`, and run `kubectl port-forward` to enable communication with the pod. You'll need to stop the `kubectl port-forward` command still running from the previous exercise. Confirm that the pod is running and is responding to HTTP requests.

OBSERVING A SUCCESSFUL LIVENESS PROBE

The liveness probes for the pod's containers starts firing soon after the start of each individual container. Since the processes in both containers are healthy, the probes continuously report success. As this is the normal state, the fact that the probes are successful is not explicitly indicated anywhere in the status of the pod nor in its events.

The only indication that Kubernetes is executing the probe is found in the container logs. The Node.js application in the `kiada` container prints a line to the standard output every time it handles an HTTP request. This includes the liveness probe requests, so you can display them using the following command:

```
$ kubectl logs kiada-liveness -c kiada -f
```

The liveness probe for the `envoy` container is configured to send HTTP requests to Envoy's administration interface, which doesn't log HTTP requests to the standard output, but to the file `/var/log/envoy.admin.log` in the container's filesystem. To display the log file, you use the following command:

```
$ kubectl exec kiada-liveness -c envoy -- tail -f /var/log/envoy.admin.log
```

OBSERVING THE LIVENESS PROBE FAIL

A successful liveness probe isn't interesting, so let's cause Envoy's liveness probe to fail. To see what will happen behind the scenes, start watching events by executing the following command in a separate terminal:

```
$ kubectl get events -w
```

Using Envoy's administration interface, you can configure its health check endpoint to succeed or fail. To make it fail, open URL http://localhost:9901 in your browser and click the *healthcheck/fail* button, or use the following `curl` command:

```
$ curl -X POST localhost:9901/healthcheck/fail
```

Immediately after executing the command, observe the events that are displayed in the other terminal. When the probe fails, a `Warning` event is recorded, indicating the error and the HTTP status code returned:

```
Warning  Unhealthy  Liveness probe failed: HTTP probe failed with code 503
```

Because the probe's `failureThreshold` is set to three, a single failure is not enough to consider the container unhealthy, so it continues to run. You can make the liveness probe succeed again by clicking the *healthcheck/ok* button in Envoy's admin interface, or by using `curl` as follows:

```
$ curl -X POST localhost:9901/healthcheck/ok
```

If you are fast enough, the container won't be restarted.

OBSERVING THE LIVENESS PROBE REACH THE FAILURE THRESHOLD

If you let the liveness probe fail multiple times, the `kubectl get events -w` command should print the following events (note that some columns are omitted due to page width constraints):

```
$ kubectl get events -w
TYPE     REASON     MESSAGE
Warning  Unhealthy  Liveness probe failed: HTTP probe failed with code 503    #A
Warning  Unhealthy  Liveness probe failed: HTTP probe failed with code 503    #A
Warning  Unhealthy  Liveness probe failed: HTTP probe failed with code 503    #A
Normal   Killing    Container envoy failed liveness probe, will be restarted    #B
Normal   Pulled     Container image already present on machine
Normal   Created    Created container envoy
Normal   Started    Started container envoy
```

#A The liveness probe fails three times
#B When the failure threshold is reached, the container is restarted

Remember that the probe failure threshold is set to 3, so when the probe fails three times in a row, the container is stopped and restarted. This is indicated by the events in the listing.

The `kubectl get pods` command shows that the container has been restarted:

```
$ kubectl get po kiada-liveness
NAME             READY   STATUS    RESTARTS   AGE
kiada-liveness   2/2     Running   1          5m
```

The `RESTARTS` column shows that one container restart has taken place in the pod.

UNDERSTANDING HOW A CONTAINER THAT FAILS ITS LIVENESS PROBE IS RESTARTED

If you're wondering whether the main process in the container was gracefully stopped or killed forcibly, you can check the pod's status by retrieving the full manifest using `kubectl get` or using `kubectl describe`:

```
$ kubectl describe po kiada-liveness
Name:           kiada-liveness
...
Containers:
  ...
  envoy:
    ...
    State:          Running    #A
      Started:      Sun, 31 May 2020 21:33:13 +0200    #A
    Last State:     Terminated    #B
      Reason:       Completed    #B
      Exit Code:    0    #B
      Started:      Sun, 31 May 2020 21:16:43 +0200    #B
      Finished:     Sun, 31 May 2020 21:33:13 +0200    #B
    ...
```

#A This is the state of the new container.
#B The previous container terminated with exit code 0.

The exit code zero shown in the listing implies that the application process gracefully exited on its own. If it had been killed, the exit code would have been 137.

> **NOTE** Exit code `128+n` indicates that the process exited due to external signal `n`. Exit code `137` is `128+9`, where `9` represents the `KILL` signal. You'll see this exit code whenever the container is killed. Exit code `143` is `128+15`, where 15 is the `TERM` signal. You'll typically see this exit code when the container runs a shell that has terminated gracefully.

Examine Envoy's log to confirm that it caught the `TERM` signal and has terminated by itself. You must use the `kubectl logs` command with the `--container` or the shorter `-c` option to specify what container you're interested in.

Also, because the container has been replaced with a new one due to the restart, you must request the log of the previous container using the `--previous` or `-p` flag. Here's the command to use and the last four lines of its output:

```
$ kubectl logs kiada-liveness -c envoy -p
...
...[warning][main] [source/server/server.cc:493] caught SIGTERM
...[info][main] [source/server/server.cc:613] shutting down server instance
...[info][main] [source/server/server.cc:560] main dispatch loop exited
...[info][main] [source/server/server.cc:606] exiting
```

The log confirms that Kubernetes sent the `TERM` signal to the process, allowing it to shut down gracefully. Had it not terminated by itself, Kubernetes would have killed it forcibly.

After the container is restarted, its health check endpoint responds with HTTP status `200 OK` again, indicating that the container is healthy.

6.2.5 Using the exec and the tcpSocket liveness probe types

For applications that don't expose HTTP health-check endpoints, the `tcpSocket` or the `exec` liveness probes should be used.

ADDING A TCPSOCKET LIVENESS PROBE

For applications that accept non-HTTP TCP connections, a `tcpSocket` liveness probe can be configured. Kubernetes tries to open a socket to the TCP port and if the connection is established, the probe is considered a success, otherwise it's considered a failure.

An example of a `tcpSocket` liveness probe is shown here:

```
    livenessProbe:
      tcpSocket:    #A
        port: 1234    #A
      periodSeconds: 2    #B
      failureThreshold: 1    #C
```

#A This tcpSocket probe uses TCP port 1234
#B The probe runs every 2s
#C A single probe failure is enough to restart the container

The probe in the listing is configured to check if the container's network port `1234` is open. An attempt to establish a connection is made every two seconds and a single failed attempt is enough to consider the container as unhealthy.

ADDING AN EXEC LIVENESS PROBE

Applications that do not accept TCP connections may provide a command to check their status. For these applications, an `exec` liveness probe is used. As shown in the next figure, the command is executed inside the container and must therefore be available on the container's file system.

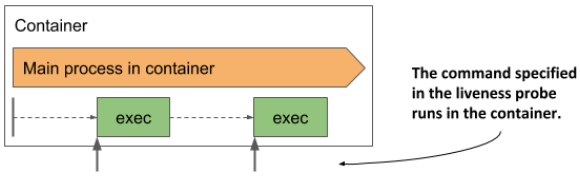

Figure 6.7 The exec liveness probe runs the command inside the container

The following is an example of a probe that runs `/usr/bin/healthcheck` every two seconds to determine if the application running in the container is still alive:

```
livenessProbe:
  exec:
    command:    #A
    - /usr/bin/healthcheck    #A
  periodSeconds: 2    #B
  timeoutSeconds: 1    #C
  failureThreshold: 1    #D
```

#A The command to run and its arguments
#B The probe runs every second
#C The command must return within one second
#D A single probe failure is enough to restart the container

If the command returns exit code zero, the container is considered healthy. If it returns a non-zero exit code or fails to complete within one second as specified in the `timeoutSeconds` field, the container is terminated immediately, as configured in the `failureThreshold` field, which indicates that a single probe failure is sufficient to consider the container as unhealthy.

6.2.6 Using a startup probe when an application is slow to start

The default liveness probe settings give the application between 20 and 30 seconds to start responding to liveness probe requests. If the application takes longer to start, it is restarted and must start again. If the second start also takes as long, it is restarted again. If this continues, the container never reaches the state where the liveness probe succeeds and gets stuck in an endless restart loop.

To prevent this, you can increase the `initialDelaySeconds`, `periodSeconds` or `failureThreshold` settings to account for the long start time, but this will have a negative effect on the normal operation of the application. The higher the result of `periodSeconds * failureThreshold`, the longer it takes to restart the application if it becomes unhealthy. For applications that take minutes to start, increasing these parameters enough to prevent the application from being restarted prematurely may not be a viable option.

INTRODUCING STARTUP PROBES

To deal with the discrepancy between the start and the steady-state operation of an application, Kubernetes also provides *startup probes*.

If a startup probe is defined for a container, only the startup probe is executed when the container is started. The startup probe can be configured to consider the slow start of the

application. When the startup probe succeeds, Kubernetes switches to using the liveness probe, which is configured to quickly detect when the application becomes unhealthy.

ADDING A STARTUP PROBE TO A POD'S MANIFEST

Imagine that the Kiada Node.js application needs more than a minute to warm up, but you want it to be restarted within 10 seconds when it becomes unhealthy during normal operation. The following listing shows how you configure the startup and liveness probes (you'll find it in the file Chapter06/pod.kiada-startup-probe.yaml).

Listing 6.2 Using a combination of a startup and a liveness probe

```
...
  containers:
  - name: kiada
    image: luksa/kiada:0.1
    ports:
    - name: http
      containerPort: 8080
    startupProbe:
      httpGet:
        path: /    #A
        port: http    #A
      periodSeconds: 10    #B
      failureThreshold:  12    #B
    livenessProbe:
      httpGet:
        path: /    #A
        port: http    #A
      periodSeconds: 5    #C
      failureThreshold: 2    #C
```

#A The startup and the liveness probes typically use the same endpoint
#B The application gets 120 seconds to start
#C After startup, the application's health is checked every 5 seconds, and is restarted when it fails the liveness probe twice

When the container defined in the listing starts, the application has 120 seconds to start responding to requests. Kubernetes performs the startup probe every 10 seconds and makes a maximum of 12 attempts.

As shown in the following figure, unlike liveness probes, it's perfectly normal for a startup probe to fail. A failure only indicates that the application hasn't yet been completely started. A successful startup probe indicates that the application has started successfully, and Kubernetes should switch to the liveness probe. The liveness probe is then typically executed using a shorter period of time, which allows for faster detection of non-responsive applications.

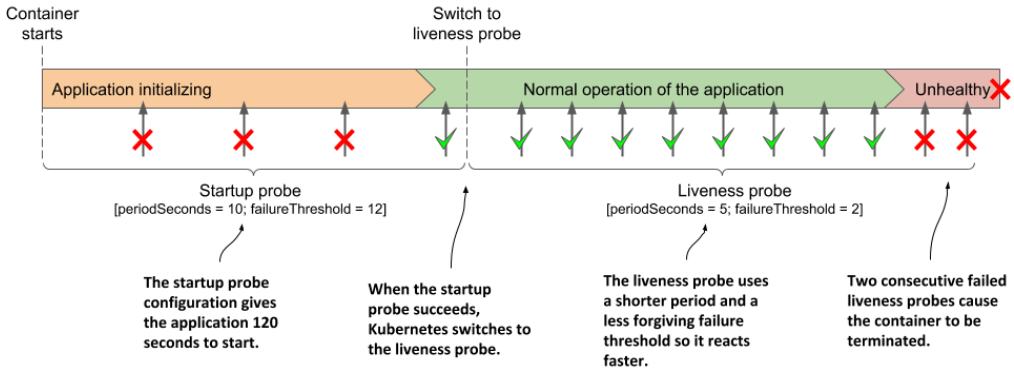

Figure 6.8 Fast detection of application health problems using a combination of startup and liveness probe

> **NOTE** If the startup probe fails often enough to reach the `failureThreshold`, the container is terminated as if the liveness probe had failed.

Usually, the startup and liveness probes are configured to use the same HTTP endpoint, but different endpoints can be used. You can also configure the startup probe as an `exec` or `tcpSocket` probe instead of an `httpGet` probe.

6.2.7 Creating effective liveness probe handlers

You should define a liveness probe for all your pods. Without one, Kubernetes has no way of knowing whether your app is still alive or not, apart from checking whether the application process has terminated.

CAUSING UNNECESSARY RESTARTS WITH BADLY IMPLEMENTED LIVENESS PROBE HANDLERS

When you implement a handler for the liveness probe, either as an HTTP endpoint in your application or as an additional executable command, be very careful to implement it correctly. If a poorly implemented probe returns a negative response even though the application is healthy, the application will be restarted unnecessarily. Many Kubernetes users learn this the hard way. If you can make sure that the application process terminates by itself when it becomes unhealthy, it may be safer not to define a liveness probe.

WHAT A LIVENESS PROBE SHOULD CHECK

The liveness probe for the `kiada` container isn't configured to call an actual health-check endpoint, but only checks that the Node.js server responds to simple HTTP requests for the root URI. This may seem overly simple, but even such a liveness probe works wonders, because it causes a restart of the container if the server no longer responds to HTTP requests, which is its main task. If no liveness probe were defined, the pod would remain in an unhealthy state where it doesn't respond to any requests and would have to be restarted manually. A simple liveness probe like this is better than nothing.

To provide a better liveness check, web applications typically expose a specific health-check endpoint, such as `/healthz`. When this endpoint is called, the application performs an internal status check of all the major components running within the application to ensure that none of them have died or are no longer doing what they should.

> **TIP** Make sure that the `/healthz` HTTP endpoint doesn't require authentication or the probe will always fail, causing your container to be restarted continuously.

Make sure that the application checks only the operation of its internal components and nothing that is influenced by an external factor. For example, the health-check endpoint of a frontend service should never respond with failure when it can't connect to a backend service. If the backend service fails, restarting the frontend will not solve the problem. Such a liveness probe will fail again after the restart, so the container will be restarted repeatedly until the backend is repaired. If many services are interdependent in this way, the failure of a single service can result in cascading failures across the entire system.

KEEPING PROBES LIGHT

The handler invoked by a liveness probe shouldn't use too much computing resources and shouldn't take too long to complete. By default, probes are executed relatively often and only given one second to complete.

Using a handler that consumes a lot of CPU or memory can seriously affect the main process of your container. Later in the book you'll learn how to limit the CPU time and total memory available to a container. The CPU and memory consumed by the probe handler invocation count towards the resource quota of the container, so using a resource-intensive handler will reduce the CPU time available to the main process of the application.

> **TIP** When running a Java application in your container, you may want to use an HTTP GET probe instead of an exec liveness probe that starts an entire JVM. The same applies to commands that require considerable computing resources.

AVOIDING RETRY LOOPS IN YOUR PROBE HANDLERS

You've learned that the failure threshold for the probe is configurable. Instead of implementing a retry loop in your probe handlers, keep it simple and instead set the `failureThreshold` field to a higher value so that the probe must fail several times before the application is considered unhealthy. Implementing your own retry mechanism in the handler is a waste of effort and represents another potential point of failure.

6.3 Executing actions at container start-up and shutdown

In the previous chapter you learned that you could use init containers to run containers at the start of the pod lifecycle. You may also want to run additional processes every time a container starts and just before it stops. You can do this by adding *lifecycle hooks* to the container. Two types of hooks are currently supported:

- *Post-start* hooks, which are executed when the container starts, and

- *Pre-stop* hooks, which are executed shortly before the container stops.

These lifecycle hooks are specified per container, as opposed to init containers, which are specified at the pod level. The next figure should help you visualize how lifecycle hooks fit into the lifecycle of a container.

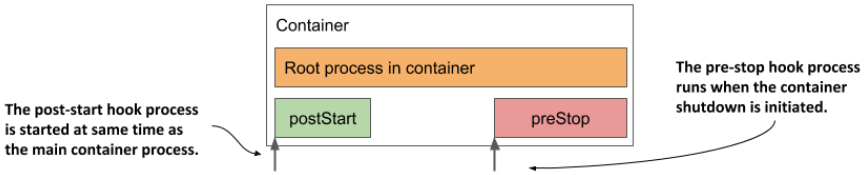

Figure 6.9 How the post-start and pre-stop hook fit into the container's lifecycle

Like liveness probes, lifecycle hooks can be used to either

- execute a command inside the container, or
- send an HTTP GET request to the application in the container.

> **NOTE** The same as with liveness probes, lifecycle hooks can only be applied to regular containers and not to init containers. Unlike probes, lifecycle hooks do not support `tcpSocket` handlers.

Let's look at the two types of hooks individually to see what you can use them for.

6.3.1 Using post-start hooks to perform actions when the container starts

The post-start lifecycle hook is invoked immediately after the container is created. You can use the `exec` type of the hook to execute an additional process as the main process starts, or you can use the `httpGet` hook to send an HTTP request to the application running in the container to perform some type of initialization or warm-up procedure.

If you're the author of the application, you could perform the same operation within the application code itself, but if you need to add it to an existing application that you didn't create yourself, you may not be able to do so. A post-start hook provides a simple alternative that doesn't require you to change the application or its container image.

Let's look at an example of how a post-start hook can be used in a new service you'll create.

INTRODUCING THE QUOTE SERVICE

You may remember from section 2.2.1 that the final version of the Kubernetes in Action Demo Application (Kiada) Suite will contain the Quote and Quiz services in addition to the Node.js application. The data from those two services will be used to show a random quote from the book and a multiple-choice pop quiz to help you test your Kubernetes knowledge. To refresh your memory, the following figure shows the three components that make up the Kiada Suite.

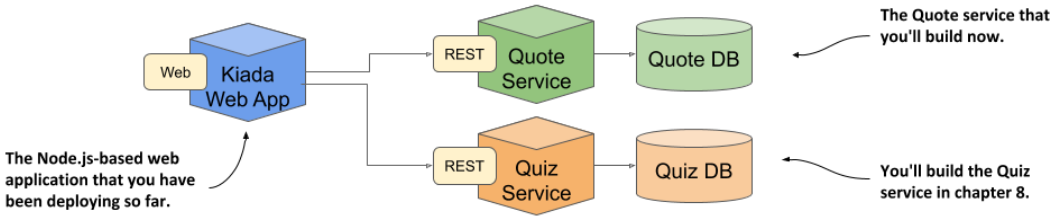

Figure 6.10 The Kubernetes in Action Demo Application Suite

During my first steps with Unix in the 1990s, one of the things I found most amusing was the random, sometimes funny message that the `fortune` command displayed every time I logged into our high school's Sun Ultra server. Nowadays, you'll rarely see the `fortune` command installed on Unix/Linux systems anymore, but you can still install it and run it whenever you're bored. Here's an example of what it may display:

```
$ fortune
Dinner is ready when the smoke alarm goes off.
```

The command gets the quotes from files that are packaged with it, but you can also use your own file(s). So why not use `fortune` to build the Quote service? Instead of using the default files, I'll provide a file with quotes from this book.

But one caveat exists. The `fortune` command prints to the standard output. It can't serve the quote over HTTP. However, this isn't a hard problem to solve. We can combine the `fortune` program with a web server such as Nginx to get the result we want.

USING A POST-START CONTAINER LIFECYCLE HOOK TO RUN A COMMAND IN THE CONTAINER

For the first version of the service, the container will run the `fortune` command when it starts up. The output will be redirected to a file in Nginx' web-root directory, so that it can serve it. Although this means that the same quote is returned in every request, this is a perfectly good start. You'll later improve the service iteratively.

The Nginx web server is available as a container image, so let's use it. Because the `fortune` command is not available in the image, you'd normally build a new image that uses that image as the base and installs the `fortune` package on top of it. But we'll keep things even simpler for now.

Instead of building a completely new image you'll use a post-start hook to install the `fortune` software package, download the file containing the quotes from this book, and finally run the `fortune` command and write its output to a file that Nginx can serve. The operation of the quote-poststart pod is presented in the following figure.

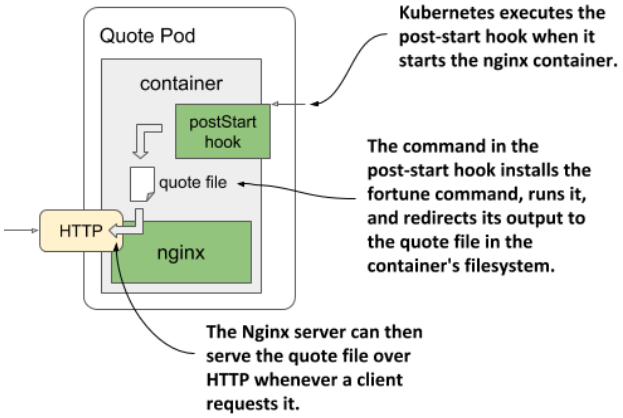

Figure 6.11 The operation of the quote-poststart pod

The following listing shows how to define the hook (Chapter06/pod.quote-poststart.yaml).

Listing 6.3 Pod with a post-start lifecycle hook

```
apiVersion: v1
kind: Pod
metadata:
  name: quote-poststart    #A
spec:
  containers:
  - name: nginx    #B
    image: nginx:alpine    #B
    ports:    #C
    - name: http    #C
      containerPort: 80    #C
    lifecycle:    #D
      postStart:    #D
        exec:    #D
          command:    #D
          - sh    #E
          - -c    #F
          - |    #G
            apk add fortune && \    #H
            curl -O https://luksa.github.io/kiada/book-quotes.txt && \    #H
            curl -O https://luksa.github.io/kiada/book-quotes.txt.dat && \    #H
            fortune book-quotes.txt > /usr/share/nginx/html/quote    #H
```

#A The name of this pod is quote-poststart.
#B The nginx:alpine container image is used in this single-container pod.
#C The Nginx server runs on port 80.
#D A post-start lifecycle hook is used to run a command when the container starts.
#E This is the command.
#F This is its first argument.
#G The second argument is the multi-line string that follows.
#H The second argument consists of these lines.

The YAML in the listing is not simple, so let me make sense of it. First, the easy parts. The pod is named `quote-poststart` and contains a single container based on the `nginx:alpine` image. A single port is defined in the container. A `postStart` lifecycle hook is also defined for the container. It specifies what command to run when the container starts. The tricky part is the definition of this command, but I'll break it down for you.

It's a list of commands that are passed to the `sh` command as an argument. The reason this needs to be so is because you can't define multiple commands in a lifecycle hook. The solution is to invoke a shell as the main command and letting it run the list of commands by specifying them in the command string:

```
sh -c "the command string"
```

In the previous listing, the third argument (the command string) is rather long, so it must be specified over multiple lines to keep the YAML legible. Multi-line string values in YAML can be defined by typing a pipeline character and following it with properly indented lines. The command string in the previous listing is therefore as follows:

```
apk add fortune && \
curl -O https://luksa.github.io/kiada/book-quotes.txt && \
curl -O https://luksa.github.io/kiada/book-quotes.txt.dat && \
fortune book-quotes.txt > /usr/share/nginx/html/quote
```

As you can see, the command string consists of four commands. Here's what they do:

6. The `apk add fortune` command runs the Alpine Linux package management tool, which is part of the image that `nginx:alpine` is based on, to install the `fortune` package in the container.
7. The first `curl` command downloads the `book-quotes.txt` file.
8. The second `curl` command downloads the `book-quotes.txt.dat` file.
9. The `fortune` command selects a random quote from the `book-quotes.txt` file and prints it to standard output. That output is redirected to the `/usr/share/nginx/html/quote` file.

The lifecycle hook command runs parallel to the main process. The `postStart` name is somewhat misleading, because the hook isn't executed after the main process is fully started, but as soon as the container is created, at around the same time the main process starts.

When the `postStart` hook in this example completes, the quote produced by the `fortune` command is stored in the `/usr/share/nginx/html/quote` file and can be served by Nginx.

Use the `kubectl apply` command to create the pod from the Chapter06/pod.quote-poststart.yaml file, and you should then be able to use `curl` or your browser to get the quote at URI `/quote` on port `80` of the `quote-poststart` pod. You've already learned how to use the `kubectl port-forward` command to open a tunnel to the container, but you may want to refer to the sidebar because a caveat exists.

Accessing the quote-poststart pod

To retrieve the quote from the `quote-poststart` pod, you must first run the `kubectl port-forward` command, which may fail as shown here:

$ **kubectl port-forward quote-poststart 80**
Unable to listen on port 80: Listeners failed to create with the following errors: [unable to create listener: Error listen tcp4 127.0.0.1:80: bind: permission denied unable to create listener: Error listen tcp6 [::1]:80: bind: permission denied]
error: unable to listen on any of the requested ports: [{80 80}]

The command fails if your operating system doesn't allow you to run processes that bind to port numbers 0-1023. To fix this, you must use a higher local port number as follows:

$ **kubectl port-forward quote-poststart 1080:80**

The last argument tells `kubectl` to use port `1080` locally and forward it to port `80` of the pod. You can now access the Quote service at http://localhost:1080/quote.

If everything works as it should, the Nginx server will return a random quote from this book as in the following example:

```
$ curl localhost:1080/quote
The same as with liveness probes, lifecycle hooks can only be applied to regular containers
       and
not to init containers. Unlike probes, lifecycle hooks do not support tcpSocket handlers.
```

The first version of the Quote service is now done, but you'll improve it in the next chapter. Now let's learn about the caveats of using post-start hooks before we move on.

UNDERSTANDING HOW A POST-START HOOK AFFECTS THE CONTAINER

Although the post-start hook runs asynchronously with the main container process, it affects the container in two ways.

First, the container remains in the `Waiting` state with the reason `ContainerCreating` until the hook invocation is completed. The phase of the pod is `Pending`. If you run the `kubectl logs` command at this point, it refuses to show the logs, even though the container is running. The `kubectl port-forward` command also refuses to forward ports to the pod.

If you want to see this for yourself, deploy the Chapter06/pod.quote-poststart-slow.yaml pod manifest file. It defines a post-start hook that takes 60 seconds to complete. Immediately after the pod is created, inspect its state, and display the logs with the following command:

```
 $ kubectl logs quote-poststart-slow
Error from server (BadRequest): container "nginx" in pod "quote-poststart-slow" is waiting
       to start: ContainerCreating
```

The error message returned implies that the container hasn't started yet, which isn't the case. To prove this, use the following command to list processes in the container:

```
$ kubectl exec quote-poststart-slow -- ps x
PID   USER     TIME  COMMAND
  1 root      0:00 nginx: master process nginx -g daemon off;          #A
  7 root      0:00 sh -c apk add fortune && \ sleep 60 && \ curl...    #B
 13 nginx     0:00 nginx: worker process                               #A
...                                                                    #A
 20 nginx     0:00 nginx: worker process                               #A
 21 root      0:00 sleep 60                                            #B
 22 root      0:00 ps x
```

#A Nginx is running
#B The processes that run as part of the post-start hook

The other way a post-start hook could affect the container is if the command used in the hook can't be executed or returns a non-zero exit code. If this happens, the entire container is restarted. To see an example of a post-start hook that fails, deploy the pod manifest Chapter06/pod.quote-poststart-fail.yaml.

If you watch the pod's status using `kubectl get pods -w`, you'll see the following status:

```
quote-poststart-fail   0/1     PostStartHookError: command 'sh -c echo 'Emulating a post-
       start hook failure'; exit 1' exited with 1:
```

It shows the command that was executed and the code with which it terminated. When you review the pod events, you'll see a `FailedPostStartHook` warning event that indicates the exit code and what the command printed to the standard or error output. This is the event:

```
Warning  FailedPostStartHook  Exec lifecycle hook ([sh -c ...]) for Container "nginx" in
       Pod "quote-poststart-fail_default(...)" failed - error: command '...' exited with 1:
       , message: "Emulating a post-start hook failure\n"
```

The same information is also contained in the `containerStatuses` field in the pod's `status` field, but only for a short time, as the container status changes to `CrashLoopBackOff` shortly afterwards.

> **TIP** Because the state of a pod can change quickly, inspecting just its status may not tell you everything you need to know. Rather than inspecting the state at a particular moment in time, reviewing the pod's events is usually a better way to get the full picture.

CAPTURING THE OUTPUT PRODUCED BY THE PROCESS INVOKED VIA A POST-START HOOK

As you've just learned, the output of the command defined in the post-start hook can be inspected if it fails. In cases where the command completes successfully, the output of the command is not logged anywhere. To see the output, the command must log to a file instead of the standard or error output. You can then view the contents of the file with a command like the following:

```
$ kubectl exec my-pod -- cat logfile.txt
```

USING AN HTTP GET POST-START HOOK

In the previous example, you configured the post-start hook to execute a command inside the container. Alternatively, you can have Kubernetes send an HTTP GET request when it starts the container by using an `httpGet` post-start hook.

> **NOTE** You can't specify both an `exec` and an `httpGet` post-start hook for a container. They are exclusive.

You can configure the lifecycle hook to send the request to a process running in the container itself, a different container in the pod, or a different host altogether.

For example, you can use an `httpGet` post-start hook to tell another service about your pod. The following listing shows an example of a post-start hook definition that does this. You'll find it in file Chapter06/pod.poststart-httpget.yaml.

Listing 6.4 Using an httpGet post-start hook to warm up a web server

```
    lifecycle:    #A
      postStart:    #A
        httpGet:    #A
          host: myservice.example.com    #B
          port: 80    #B
          path: /container-started    #C
```

#A This is a post-start lifecycle hook that sends an HTTP GET request.
#B The host and port where the request is sent.
#C The URI requested in the HTTP request.

The example in the listing shows an `httpGet` post-start hook that calls the following URL when the container starts: `http://myservice.example.com/container-started`.

In addition to the `host`, `port`, and `path` fields shown in the listing, you can also specify the `scheme` (`HTTP` or `HTTPS`) and the `httpHeaders` to be sent in the request. The `host` field defaults to the pod IP. Don't set it to `localhost` unless you want to send the request to the node hosting the pod. That's because the request is sent from the host node, not from within the container.

As with command-based post-start hooks, the HTTP GET post-start hook is executed at the same time as the container's main process. And this is what makes these types of lifecycle hooks applicable only to a limited set of use-cases.

If you configure the hook to send the request to the container its defined in, you'll be in trouble if the container's main process isn't yet ready to accept requests. In that case, the post-start hook fails, which then causes the container to be restarted. On the next run, the same thing happens. The result is a container that keeps being restarted.

To see this for yourself, try creating the pod defined in Chapter06/pod.poststart-httpget-slow.yaml. I've made the container wait one second before starting the web server. This ensures that the post-start hook never succeeds. But the same thing could also happen if the pause didn't exist. There is no guarantee that the web server will always start up fast enough. It might start fast on your own computer or a server that's not overloaded, but on a production system under considerable load, the container may never start properly.

> **WARNING** Using an HTTP GET post-start hook might cause the container to enter an endless restart loop. Never configure this type of lifecycle hook to target the same container or any other container in the same pod.

Another problem with HTTP GET post-start hooks is that Kubernetes doesn't treat the hook as failed if the HTTP server responds with status code such as `404 Not Found`. Make sure you specify the correct URI in your HTTP GET hook, otherwise you might not even notice that the post-start hook missed its mark.

6.3.2 Using pre-stop hooks to run a process just before the container terminates

Besides executing a command or sending an HTTP request at container startup, Kubernetes also allows the definition of a *pre-stop* hook in your containers.

A pre-stop hook is executed immediately before a container is terminated. To terminate a process, the `TERM` signal is usually sent to it. This tells the application to finish what it's doing and shut down. The same happens with containers. Whenever a container needs to be stopped or restarted, the `TERM` signal is sent to the main process in the container. Before this happens, however, Kubernetes first executes the pre-stop hook, if one is configured for the container. The `TERM` signal is not sent until the pre-stop hook completes unless the process has already terminated due to the invocation of the pre-stop hook handler itself.

> **NOTE** When container termination is initiated, the liveness and other probes are no longer invoked.

A pre-stop hook can be used to initiate a graceful shutdown of the container or to perform additional operations without having to implement them in the application itself. As with post-

start hooks, you can either execute a command within the container or send an HTTP request to the application running in it.

USING A PRE-STOP LIFECYCLE HOOK TO SHUT DOWN A CONTAINER GRACEFULLY

The Nginx web server used in the quote pod responds to the `TERM` signal by immediately closing all open connections and terminating the process. This is not ideal, as the client requests that are being processed at this time aren't allowed to complete.

Fortunately, you can instruct Nginx to shut down gracefully by running the command `nginx -s quit`. When you run this command, the server stops accepting new connections, waits until all in-flight requests have been processed, and then quits.

When you run Nginx in a Kubernetes pod, you can use a pre-stop lifecycle hook to run this command and ensure that the pod shuts down gracefully. The following listing shows the definition of this pre-stop hook (you'll find it in the file Chapter06/pod.quote-prestop.yaml).

Listing 6.5 Defining a pre-stop hook for Nginx

```
    lifecycle:    #A
      preStop:    #A
        exec:    #B
          command:    #B
          - nginx    #C
          - -s    #C
          - quit    #C
```

#A This is a pre-stop lifecycle hook
#B It executes a command
#C This is the command that gets executed

Whenever a container using this pre-stop hook is terminated, the command `nginx -s quit` is executed in the container before the main process of the container receives the `TERM` signal.

Unlike the post-start hook, the container is terminated regardless of the result of the pre-stop hook - a failure to execute the command or a non-zero exit code does not prevent the container from being terminated. If the pre-stop hook fails, you'll see a `FailedPreStopHook` warning event among the pod events, but you might not see any indication of the failure if you are only monitoring the status of the pod.

> **TIP** If successful completion of the pre-stop hook is critical to the proper operation of your system, make sure that it runs successfully. I've experienced situations where the pre-stop hook didn't run at all, but the engineers weren't even aware of it.

Like post-start hooks, you can also configure the pre-stop hook to send an HTTP GET request to your application instead of executing commands. The configuration of the HTTP GET pre-stop hook is the same as for a post-start hook. For more information, see section 6.3.1.

Why doesn't my application receive the TERM signal?

Many developers make the mistake of defining a pre-stop hook just to send a `TERM` signal to their applications in the pre-stop hook. They do this when they find that their application never receives the `TERM` signal. The root cause is usually not that the signal is never sent, but that it is swallowed by something inside the container. This typically happens when you use the *shell* form of the `ENTRYPOINT` or the `CMD` directive in your Dockerfile. Two forms of these directives exist.

The *exec* form is: `ENTRYPOINT ["/myexecutable", "1st-arg", "2nd-arg"]`

The *shell* form is: `ENTRYPOINT /myexecutable 1st-arg 2nd-arg`

When you use the exec form, the executable file is called directly. The process it starts becomes the root process of the container. When you use the shell form, a shell runs as the root process, and the shell runs the executable as its child process. In this case, the shell process is the one that receives the `TERM` signal. Unfortunately, it doesn't pass this signal to the child process.

In such cases, instead of adding a pre-stop hook to send the `TERM` signal to your app, the correct solution is to use the exec form of `ENTRYPOINT` or `CMD`.

Note that the same problem occurs if you use a shell script in your container to run the application. In this case, you must either intercept and pass signals to the application or use the `exec` shell command to run the application in your script.

Pre-stop hooks are only invoked when the container is requested to terminate, either because it has failed its liveness probe or because the pod has to shut down. They are not called when the process running in the container terminates by itself.

UNDERSTANDING THAT LIFECYCLE HOOKS TARGET CONTAINERS, NOT PODS

As a final consideration on the post-start and pre-stop hooks, I would like to emphasize that these lifecycle hooks apply to containers and not to pods. You shouldn't use a pre-stop hook to perform an action that needs to be performed when the entire pod is shut down, because pre-stop hooks run every time the container needs to terminate. This can happen several times during the pod's lifetime, not just when the pod shuts down.

6.4 Understanding the pod lifecycle

So far in this chapter you've learned a lot about how the containers in a pod run. Now let's take a closer look at the entire lifecycle of a pod and its containers.

When you create a pod object, Kubernetes schedules it to a worker node that then runs its containers. The pod's lifecycle is divided into the three stages shown in the next figure:

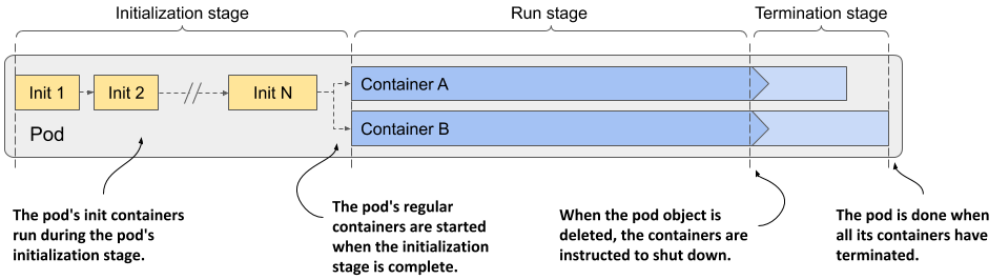

Figure 6.12 The three stages of the pod's lifecycle

The three stages of the pod's lifecycle are:

10. The initialization stage, during which the pod's init containers run.
11. The run stage, in which the regular containers of the pod run.
12. The termination stage, in which the pod's containers are terminated.

Let's see what happens in each of these stages.

6.4.1 Understanding the initialization stage

As you've already learned, the pod's init containers run first. They run in the order specified in the `initContainers` field in the pod's `spec`. Let me explain everything that unfolds.

PULLING THE CONTAINER IMAGE

Before each init container is started, its container image is pulled to the worker node. The `imagePullPolicy` field in the container definition in the pod specification determines whether the image is pulled every time, only the first time, or never.

Table 6.5 List of image-pull policies

Image pull policy	Description
Not specified	If the `imagePullPolicy` is not explicitly specified, it defaults to `Always` if the `:latest` tag is used in the image. For other image tags, it defaults to `IfNotPresent`.
`Always`	The image is pulled every time the container is (re)started. If the locally cached image matches the one in the registry, it is not downloaded again, but the registry still needs to be contacted.
`Never`	The container image is never pulled from the registry. It must exist on the worker node beforehand. Either it was stored locally when another container with the same image was deployed, or it was built on the node itself, or simply downloaded by someone or something else.

`IfNotPresent`	Image is pulled if it is not already present on the worker node. This ensures that the image is only pulled the first time it's required.

The image-pull policy is also applied every time the container is restarted, so a closer look is warranted. Examine the following figure to understand the behavior of these three policies.

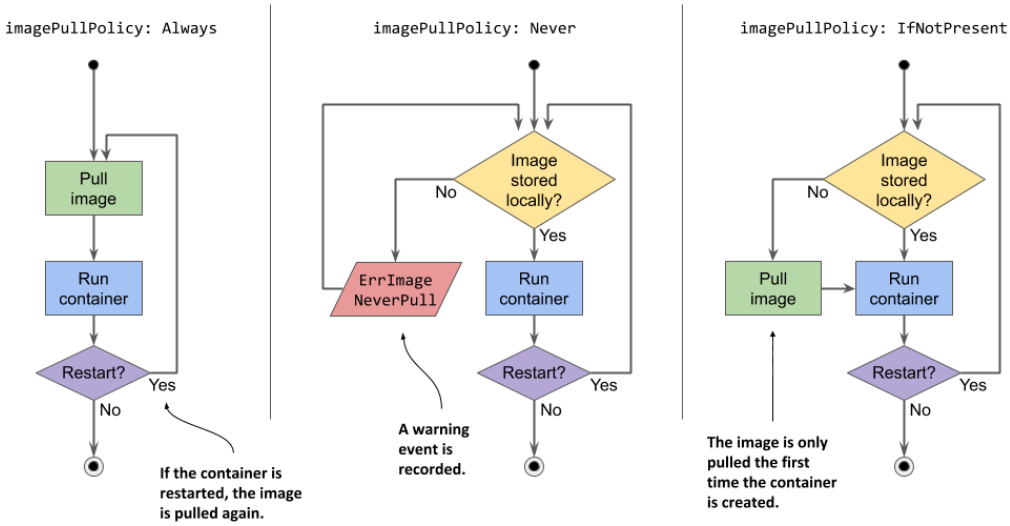

Figure 6.13 An overview of the three different image-pull policies

WARNING If the `imagePullPolicy` is set to `Always` and the image registry is offline, the container will not run even if the same image is already stored locally. A registry that is unavailable may therefore prevent your application from (re)starting.

RUNNING THE CONTAINERS

When the first container image is downloaded to the node, the container is started. When the first init container is complete, the image for the next init container is pulled and the container is started. This process is repeated until all init containers are successfully completed. Containers that fail might be restarted, as shown in the following figure.

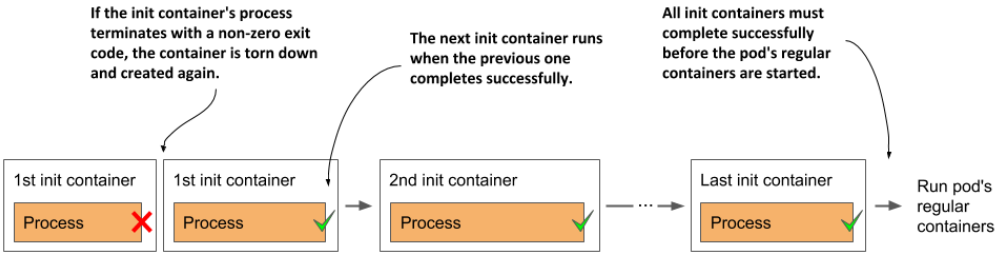

Figure 6.14 All init containers must run to completion before the regular containers can start

RESTARTING FAILED INIT CONTAINERS

If an init container terminates with an error and the pod's restart policy is set to `Always` or `OnFailure`, the failed init container is restarted. If the policy is set to `Never`, the subsequent init containers and the pod's regular containers are never started. The pod's status is displayed as `Init:Error` indefinitely. You must then delete and recreate the pod object to restart the application. To try this yourself, deploy the file Chapter06/pod.kiada-init-fail-norestart.yaml.

> **NOTE** If the container needs to be restarted and `imagePullPolicy` is set to `Always`, the container image is pulled again. If the container had terminated due to an error and you push a new image with the same tag that fixes the error, you don't need to recreate the pod, as the updated container image will be pulled before the container is restarted.

RE-EXECUTING THE POD'S INIT CONTAINERS

Init containers are normally only executed once. Even if one of the pod's main containers is terminated later, the pod's init containers are not re-executed. However, in exceptional cases, such as when Kubernetes must restart the entire pod, the pod's init containers might be executed again. This means that the operations performed by your init containers must be idempotent.

6.4.2 Understanding the run stage

When all init containers are successfully completed, the pod's regular containers are all created in parallel. In theory, the lifecycle of each container should be independent of the other containers in the pod, but this is not quite true. See sidebar for more information.

A container's post-start hook blocks the creation of the subsequent container

The Kubelet doesn't start all containers of the pod at the same time. It creates and starts the containers synchronously in the order they are defined in the pod's `spec`. If a post-start hook is defined for a container, it runs asynchronously with the main container process, but the execution of the post-start hook handler blocks the creation and start of the subsequent containers.

This is an implementation detail that might change in the future.

In contrast, the termination of containers is performed in parallel. A long-running pre-stop hook does block the shutdown of the container in which it is defined, but it does not block the shutdown of other containers. The pre-stop hooks of the containers are all invoked at the same time.

The following sequence runs independently for each container. First, the container image is pulled, and the container is started. When the container terminates, it is restarted, if this is provided for in the pod's restart policy. The container continues to run until the termination of the pod is initiated. A more detailed explanation of this sequence is presented next.

PULLING THE CONTAINER IMAGE

Before the container is created, its image is pulled from the image registry, following the pod's `imagePullPolicy`. Once the image is pulled, the container is created.

NOTE Even if a container image can't be pulled, the other containers in the pod are started nevertheless.

WARNING Containers don't necessarily start at the same moment. If pulling the image takes time, the container may start long after all the others have already started. Consider this if a containers depends on others.

RUNNING THE CONTAINER

The container starts when the main container process starts. If a post-start hook is defined in the container, it is invoked in parallel with the main container process. The post-start hook runs asynchronously and must be successful for the container to continue running.

Together with the main container and the potential post-start hook process, the startup probe, if defined for the container, is started. When the startup probe is successful, or if the startup probe is not configured, the liveness probe is started.

TERMINATING AND RESTARTING THE CONTAINER ON FAILURES

If the startup or the liveness probe fails so often that it reaches the configured failure threshold, the container is terminated. As with init containers, the pod's `restartPolicy` determines whether the container is then restarted or not.

Perhaps surprisingly, if the restart policy is set to `Never` and the startup hook fails, the pod's status is shown as `Completed` even though the post-start hook failed. You can see this for yourself by creating the pod defined in the file Chapter06/pod.quote-poststart-fail-norestart.yaml.

INTRODUCING THE TERMINATION GRACE PERIOD

If a container must be terminated, the container's pre-stop hook is called so that the application can shut down gracefully. When the pre-stop hook is completed, or if no pre-stop hook is defined, the `TERM` signal is sent to the main container process. This is another hint to the application that it should shut down.

The application is given a certain amount of time to terminate. This time can be configured using the `terminationGracePeriodSeconds` field in the pod's `spec` and defaults to 30 seconds. The timer starts when the pre-stop hook is called or when the `TERM` signal is sent if no hook is defined. If the process is still running after the termination grace period has expired, it's terminated by force via the `KILL` signal. This terminates the container.

The following figure illustrates the container termination sequence.

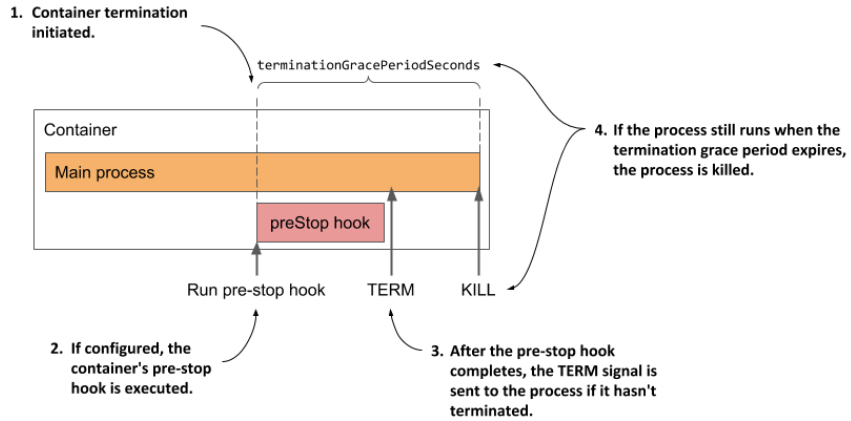

Figure 6.15 A container's termination sequence

After the container has terminated, it will be restarted if the pod's restart policy allows it. If not, the container will remain in the `Terminated` state, but the other containers will continue running until the entire pod is shut down or until they fail as well.

6.4.3 Understanding the termination stage

The pod's containers continue to run until you finally delete the pod object. When this happens, termination of all containers in the pod is initiated and its status is changed to `Terminating`.

INTRODUCING THE DELETION GRACE PERIOD

The termination of each container at pod shutdown follows the same sequence as when the container is terminated because it has failed its liveness probe, except that instead of the termination grace period, the pod's *deletion grace period* determines how much time is available to the containers to shut down on their own.

This grace period is defined in the pod's `metadata.deletionGracePeriodSeconds` field, which gets initialized when you delete the pod. By default, it gets its value from the `spec.terminationGracePeriodSeconds` field, but you can specify a different value in the `kubectl delete` command. You'll see how to do this later.

UNDERSTANDING HOW THE POD'S CONTAINERS ARE TERMINATED

As shown in the next figure, the pod's containers are terminated in parallel. For each of the pod's containers, the container's pre-stop hook is called, the `TERM` signal is then sent to the main container process, and finally the process is terminated using the `KILL` signal if the deletion grace period expires before the process stops by itself. After all the containers in the pod have stopped running, the pod object is deleted.

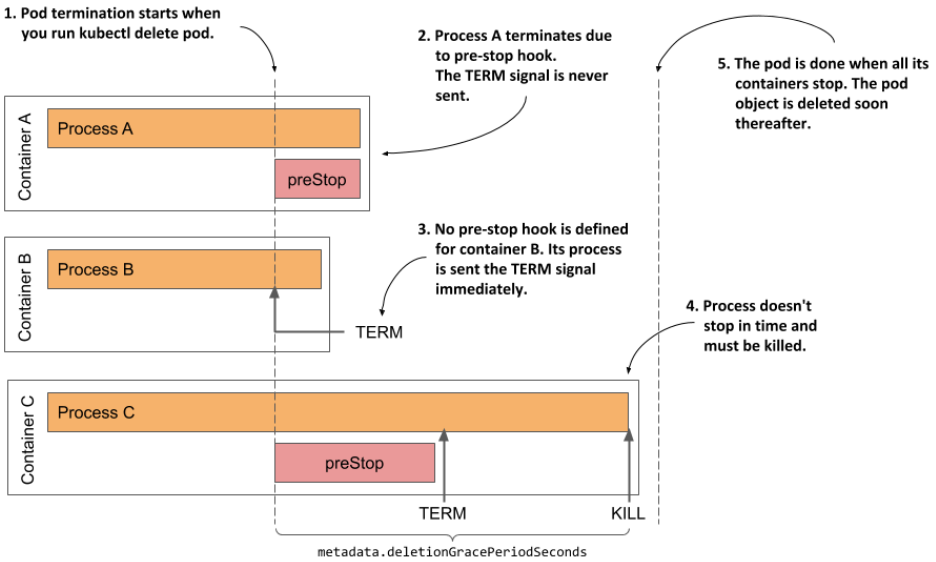

Figure 6.16 The termination sequence inside a pod

INSPECTING THE SLOW SHUTDOWN OF A POD

Let's look at this last stage of the pod's life on one of the pods you created previously. If the `kiada-ssl` pod doesn't run in your cluster, please create it again. Now delete the pod by running `kubectl delete pod kiada-ssl`.

It takes surprisingly long to delete the pod, doesn't it? I counted at least 30 seconds. This is neither normal nor acceptable, so let's fix it.

Considering what you've learned in this section, you may already know what's causing the pod to take so long to finish. If not, let me help you analyze the situation.

The `kiada-ssl` pod has two containers. Both must stop before the pod object can be deleted. Neither container has a pre-stop hook defined, so both containers should receive the `TERM` signal immediately when you delete the pod. The 30s I mentioned earlier match the default termination grace period value, so it looks like one of the containers, if not both, doesn't stop when it receives the `TERM` signal, and is killed after the grace period expires.

CHANGING THE TERMINATION GRACE PERIOD

You can try setting the pod's `terminationGracePeriodSeconds` field to a lower value to see if it terminates sooner. The following manifest shows how to the field in the pod manifest (Chapter06/pod.kiada-ssl-shortgraceperiod.yaml).

Listing 6.6 Setting a lower terminationGracePeriodSeconds for faster pod shutdown

```
apiVersion: v1
kind: Pod
metadata:
  name: kiada-ssl-shortgraceperiod
spec:
  terminationGracePeriodSeconds: 5    #A
  containers:
  ...
```

#A This pod's containers have 5 seconds to terminate after receiving the TERM signal or they will be killed

In the listing above, the pod's `terminationGracePeriodSeconds` is set to `5`. If you create and then delete this pod, you'll see that its containers are terminated within 5s of receiving the `TERM` signal.

> **TIP** A reduction of the termination grace period is rarely necessary. However, it is advisable to extend it if the application usually needs more time to shut down gracefully.

SPECIFYING THE DELETION GRACE PERIOD WHEN DELETING THE POD

Any time you delete a pod, the pod's `terminationGracePeriodSeconds` determines the amount of time the pod is given to shut down, but you can override this time when you execute the `kubectl delete` command using the `--grace-period` command line option.

For example, to give the pod 10s to shut down, you run the following command:

```
$ kubectl delete po kiada-ssl --grace-period 10
```

> **NOTE** If you set this grace period to zero, the pod's pre-stop hooks are not executed.

FIXING THE SHUTDOWN BEHAVIOR OF THE KIADA APPLICATION

Considering that the shortening of the grace period leads to a faster shutdown of the pod, it's clear that at least one of the two containers doesn't terminate by itself after it receives the `TERM` signal. To see which one, recreate the pod, then run the following commands to stream the logs of each container before deleting the pod again:

```
$ kubectl logs kiada-ssl -c kiada -f
$ kubectl logs kiada-ssl -c envoy -f
```

The logs show that the Envoy proxy catches the signal and immediately terminates, whereas the Node.js application doesn't respond to the signal. To fix this, you need to add the code in the following listing to the end of your `app.js` file. You'll find the updated file in Chapter06/kiada-0.3/app.js.

Listing 6.7 Handling the TERM signal in the kiada application

```
process.on('SIGTERM', function () {
  console.log("Received SIGTERM. Server shutting down...");
  server.close(function () {
    process.exit(0);
```

```
  });
});
```

After you make the change to the code, create a new container image with the tag `:0.3`, push it to your image registry, and deploy a new pod that uses the new image. You can also use the image `docker.io/luksa/kiada:0.3` that I've built. To create the pod, apply the manifest file Chapter06/kiada-ssl-0.3.yaml.

If you delete this new pod, you'll see that it shuts down considerably faster. From the logs of the `kiada` container, you can see that it begins to shut down as soon as it receives the `TERM` signal.

> **TIP** Don't forget to ensure that your init containers also handle the `TERM` signal so that they shut down immediately if you delete the pod object while it's still being initialized.

6.4.4 Visualizing the full lifecycle of the pod's containers

To conclude this chapter on what goes on in a pod, I present a final overview of everything that happens during the life of a pod. The following two figures summarize everything that has been explained in this chapter. The initialization of the pod is shown in the next figure.

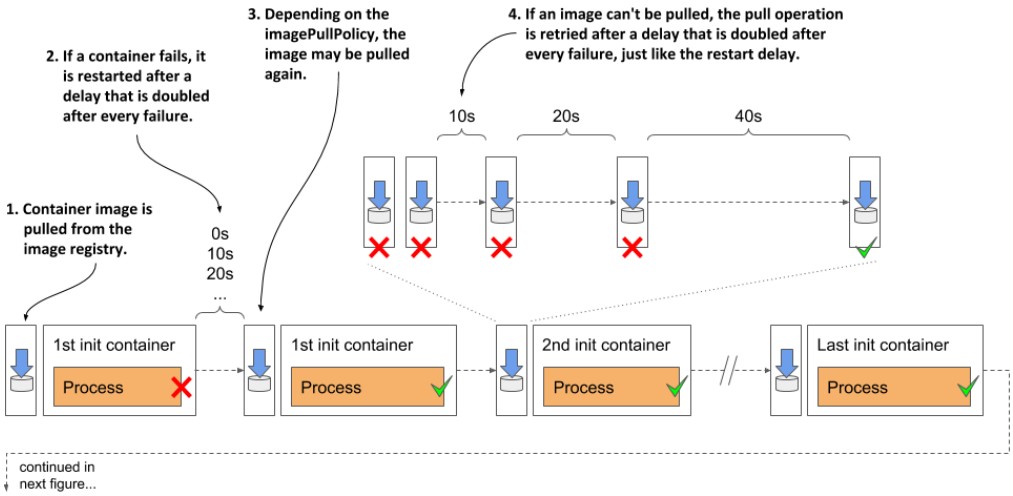

Figure 6.17 Complete overview of the pod's initialization stage

When initialization is complete, normal operation of the pod's containers begins. This is shown in the next figure.

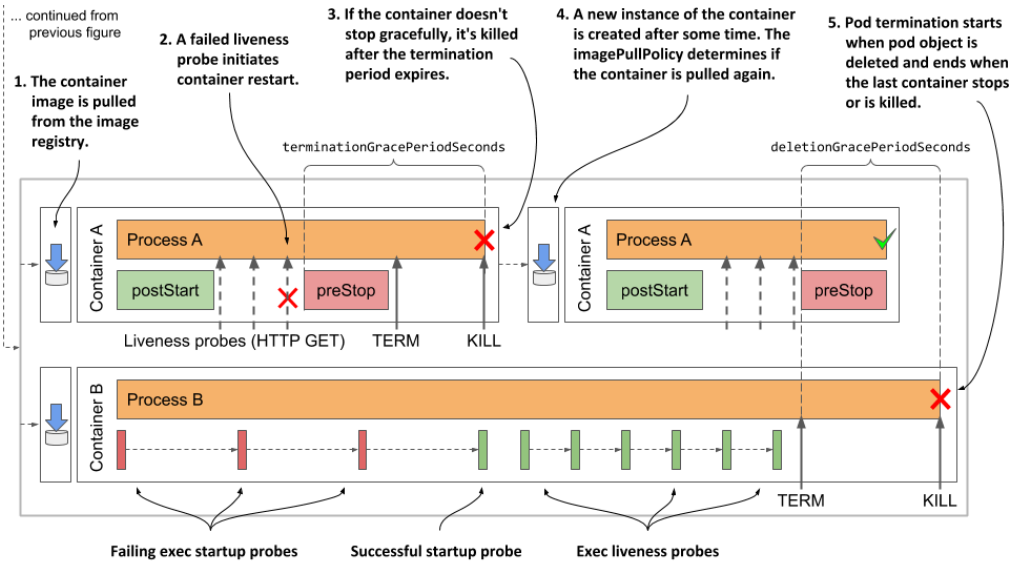

Figure 6.18 Complete overview of the pod's normal operation

6.5 Summary

In this chapter, you've learned:

- The status of the pod contains information about the phase of the pod, its conditions, and the status of each of its containers. You can view the status by running the `kubectl describe` command or by retrieving the full pod manifest using the command `kubectl get -o yaml`.
- Depending on the pod's restart policy, its containers can be restarted after they are terminated. In reality, a container is never actually restarted. Instead, the old container is destroyed, and a new container is created in its place.
- If a container is repeatedly terminated, an exponentially increasing delay is inserted before each restart. There is no delay for the first restart, then the delay is 10 seconds and then doubles before each subsequent restart. The maximum delay is 5 minutes and is reset to zero when the container has been running properly for at least twice this time.
- An exponentially increasing delay is also used after each failed attempt to download a container image.
- Adding a liveness probe to a container ensures that the container is restarted when it stops responding. The liveness probe checks the state of the application via an HTTP GET request, by executing a command in the container, or opening a TCP connection to one of the network ports of the container.
- If the application needs a long time to start, a startup probe can be defined with settings that are more forgiving than those in the liveness probe to prevent premature restarting

of the container.

- You can define lifecycle hooks for each of the pod's main containers. A post-start hook is invoked when the container starts, whereas a pre-stop hook is invoked when the container must shut down. A lifecycle hook is configured to either send an HTTP GET request or execute a command within the container.
- If a pre-stop hook is defined in the container and the container must terminate, the hook is invoked first. The `TERM` signal is then sent to the main process in the container. If the process doesn't stop within `terminationGracePeriodSeconds` after the start of the termination sequence, the process is killed.
- When you delete a pod object, all its containers are terminated in parallel. The pod's `deletionGracePeriodSeconds` is the time given to the containers to shut down. By default, it's set to the termination grace period, but can be overridden with the `kubectl delete` command.
- If shutting down a pod takes a long time, it is likely that one of the processes running in it doesn't handle the `TERM` signal. Adding a `TERM` signal handler is a better solution than shortening the termination or deletion grace period.

You now understand everything about the operation of containers in pods. In the next chapter you'll learn about the other important component of pods - storage volumes.

7

Mounting storage volumes into the Pod's containers

This chapter covers

- Persisting files across container restarts
- Sharing files between containers of the same pod
- Sharing files between pods
- Attaching network storage to pods
- Accessing the host node filesystem from within a pod

The previous two chapters focused on the pod's containers, but they are only half of what a pod typically contains. They are typically accompanied by storage volumes that allow a pod's containers to store data for the lifetime of the pod or beyond, or to share files with the other containers of the pod. This is the focus of this chapter.

> **NOTE** You'll find the code files for this chapter at https://github.com/luksa/kubernetes-in-action-2nd-edition/tree/master/Chapter07

7.1 Introducing volumes

A pod is like a small logical computer that runs a single application. This application can consist of one or more containers that run the application processes. These processes share computing resources such as CPU, RAM, network interfaces, and others. In a typical computer, the processes use the same filesystem, but this isn't the case with containers. Instead, each container has its own isolated filesystem provided by the container image.

When a container starts, the files in its filesystem are those that were added to its container image during build time. The process running in the container can then modify those files or

create new ones. When the container is terminated and restarted, all changes it made to its files are lost, because the previous container is not really restarted, but completely replaced, as explained in the previous chapter. Therefore, when a containerized application is restarted, it can't continue from the point where it was when it stopped. Although this may be okay for some types of applications, others may need the entire filesystem or at least part of it to be preserved on restart.

This is achieved by adding a *volume* to the pod and *mounting* it into the container.

> **DEFINITION** *Mounting* is the act of attaching the filesystem of some storage device or volume into a specific location in the operating system's file tree, as shown in figure 7.1. The contents of the volume are then available at that location.

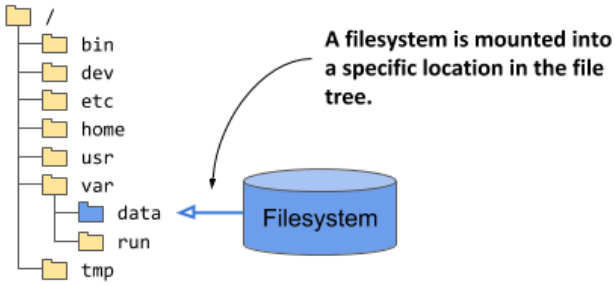

Figure 7.1 Mounting a filesystem into the file tree

7.1.1 Demonstrating the need for volumes

In this chapter, you'll build a new service that requires its data to be persisted. To do this, the pod that runs the service will need to contain a volume. But before we get to that, let me tell you about this service, and allow you to experience first-hand why it can't work without a volume.

INTRODUCING THE QUIZ SERVICE

The first 14 chapters of this book aim to teach you about the main Kubernetes concepts by showing you how to deploy the Kubernetes in Action Demo Application Suite. You already know the three components that comprise it. If not, the following figure should refresh your memory.

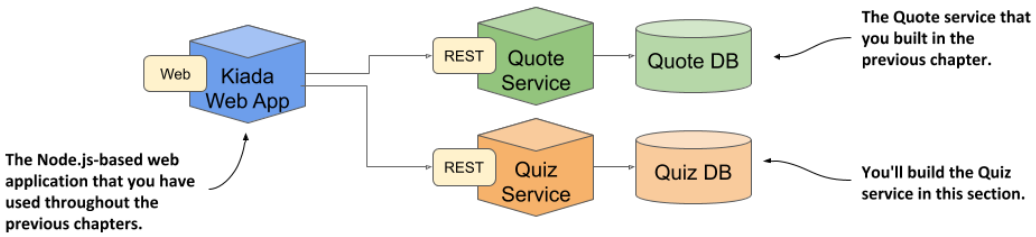

Figure 7.2 How the Quiz service fits into the architecture of the Kiada Suite

You've already built the initial version of the Kiada web application and the Quote service. Now you'll create the Quiz Service. It will provide the multiple-choice questions that the Kiada web application displays and store your answers to those questions.

The Quiz service consists of a RESTful API frontend and a MongoDB database as the backend. Initially, you'll run these two components in separate containers of the same pod, as shown in the following figure.

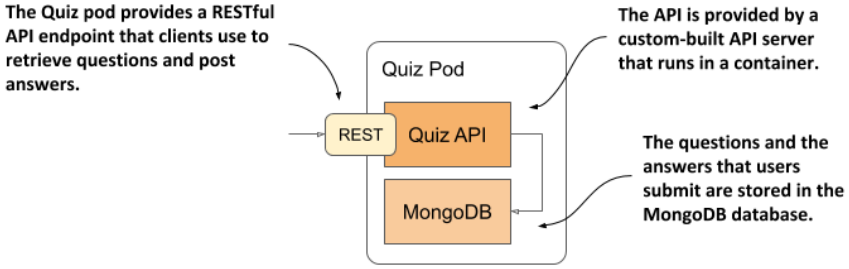

Figure 7.3 The Quiz API and the MongoDB database run in the same pod

As I explained in the pod introduction in chapter 5, creating pods like this is not the best idea, as it doesn't allow for the containers to be scaled individually. The reason we'll use a single pod is because you haven't yet learned the correct way to make pods communicate with each other. You'll learn this in chapter 11. That's when you'll split the two containers into separate pods.

BUILDING THE QUIZ API CONTAINER

The source code and the artefacts for the container image for the Quiz API component are in the Chapter07/quiz-api-0.1/ directory. The code is written in Go and built using a container. This may need further explanation for some readers. Instead of having to install the Go environment on your own computer to build the binary file from the Go source code, you build it in a container that already contains the Go environment. The result of the build is the `quiz-api` binary executable file that is written to the Chapter07/quiz-api-0.1/app/bin/ directory.

This file is then packaged into the `quiz-api:0.1` container image with a separate `docker build` command. If you wish, you can try building the binary and the container image yourself,

but you can also use the image that I've built. It's available at `docker.io/luksa/quiz-api:0.1`.

RUNNING THE QUIZ SERVICE IN A POD WITHOUT A VOLUME

The following listing shows the YAML manifest of the `quiz` pod. You can find it in the file Chapter07/pod.quiz.novolume.yaml.

Listing 7.1 The Quiz pod with no volume

```
apiVersion: v1
kind: Pod
metadata:
  name: quiz
spec:    #A
  containers:
  - name: quiz-api    #B
    image: luksa/quiz-api:0.1    #B
    ports:
    - name: http    #C
      containerPort: 8080    #C
  - name: mongo    #C
    image: mongo    #C
```

#A This pod manifest defines containers, but no volumes.
#B The quiz-api container runs the API server written in Go.
#C The mongo container runs the MongoDB database and represents the backend.

The listing shows that two containers are defined in the pod. The `quiz-api` container runs the Quiz API component explained earlier, and the `mongo` container runs the MongoDB database that the API component uses to store data.

Create the pod from the manifest and use `kubectl port-forward` to open a tunnel to the pod's port 8080 so that you can talk to the Quiz API. To get a random question, send a GET request to the `/questions/random` URI as follows:

```
$ curl localhost:8080/questions/random
ERROR: Question random not found
```

The database is still empty. You need to add questions to it.

ADDING QUESTIONS TO THE DATABASE

The Quiz API doesn't provide a way to add questions to the database, so you'll have to insert it directly. You can do this via the mongo shell that's available in the `mongo` container. Use `kubectl exec` to run the shell like this:

```
$ kubectl exec -it quiz -c mongo -- mongo
MongoDB shell version v4.4.2
connecting to: mongodb://127.0.0.1:27017/...
Implicit session: session { "id" : UUID("42671520-0cf7-...") }
MongoDB server version: 4.4.2
Welcome to the MongoDB shell.
...
```

The Quiz API reads the questions from the `questions` collection in the `kiada` database. To add a question to that collection, type the following two commands (printed in bold):

```
> use kiada
switched to db kiada
> db.questions.insert({
... id: 1,
... text: "What does k8s mean?",
... answers: ["Kates", "Kubernetes", "Kooba Dooba Doo!"],
... correctAnswerIndex: 1})
WriteResult({ "nInserted" : 1 })
```

> **NOTE** Instead of typing all these commands, you can simply run the Chapter07/insert-question.sh shell script on your local computer to insert the question.

Feel free to add additional questions. When you're done, exit the shell by pressing Control-D or typing the `exit` command.

READING QUESTIONS FROM THE DATABASE AND THE QUIZ API

To confirm that the questions that you've just inserted are now stored in the database, run the following command:

```
> db.questions.find()
{ "_id" : ObjectId("5fc249ac18d1e29fed666ab7"), "id" : 1, "text" : "What does k8s mean?",
       "answers" : [ "Kates", "Kubernetes", "Kooba Dooba Doo!" ], "correctAnswerIndex" : 1
       }
```

Now try to retrieve a random question through the Quiz API:

```
$ curl localhost:8080/questions/random
{"id":1,"text":"What does k8s mean?","correctAnswerIndex":1,
"answers":["Kates","Kubernetes","Kooba Dooba Doo!"]}
```

Good. It looks like the quiz pod provides the service we need for the Kiada Suite. But is that always the case?

RESTARTING THE MONGODB DATABASE

Because the MongoDB database writes its files to the container's filesystem, they are lost every time the container is restarted. You can confirm this by telling the database to shut down with the following command:

```
 $ kubectl exec -it quiz -c mongo -- mongo admin --eval "db.shutdownServer()"
```

When the database shuts down, the container stops, and Kubernetes starts a new one in its place. Because this is now a new container, with a fresh filesystem, it doesn't contain the questions you entered earlier. You can confirm this is true with the following command:

```
$ kubectl exec -it quiz -c mongo -- mongo kiada --quiet --eval "db.questions.count()"
0    #A
```

#A There are no questions in the database

Keep in mind that the `quiz` pod is still the same pod as before. The `quiz-api` container has been running fine this whole time. Only the `mongo` container was restarted. To be perfectly accurate, it was re-created, not restarted. You caused this by shutting down MongoDB, but it

could happen for any reason. You'll agree that it's not acceptable that a simple restart causes data to be lost.

To ensure that the data is persisted, it needs to be stored outside of the container - in a volume.

7.1.2 Understanding how volumes fit into pods

Like containers, volumes aren't top-level resources like pods or nodes, but are a component within the pod and thus share its lifecycle. As the following figure shows, a volume is defined at the pod level and then mounted at the desired location in the container.

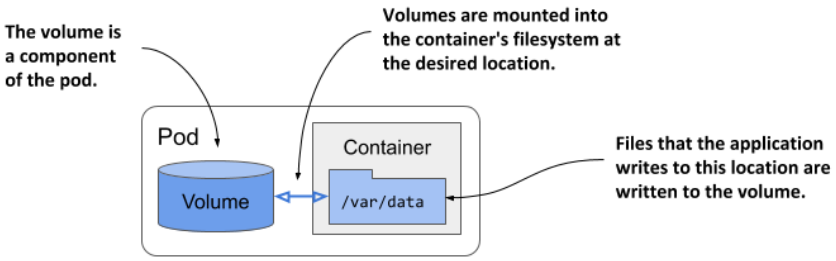

Figure 7.4 Volumes are defined at the pod level and mounted in the pod's containers

The lifecycle of a volume is tied to the lifecycle of the entire pod and is independent of the lifecycle of the container in which it is mounted. Due to this fact, volumes are also used to persist data across container restarts.

PERSISTING FILES ACROSS CONTAINER RESTARTS

All volumes in a pod are created when the pod is set up - before any of its containers are started. They are torn down when the pod is shut down.

Each time a container is (re)started, the volumes that the container is configured to use are mounted in the container's filesystem. The application running in the container can read from the volume and write to it if the volume and mount are configured to be writable.

A typical reason for adding a volume to a pod is to persist data across container restarts. If no volume is mounted in the container, the entire filesystem of the container is ephemeral. Since a container restart replaces the entire container, its filesystem is also re-created from the container image. As a result, all files written by the application are lost.

If, on the other hand, the application writes data to a volume mounted inside the container, as shown in the following figure, the application process in the new container can access the same data after the container is restarted.

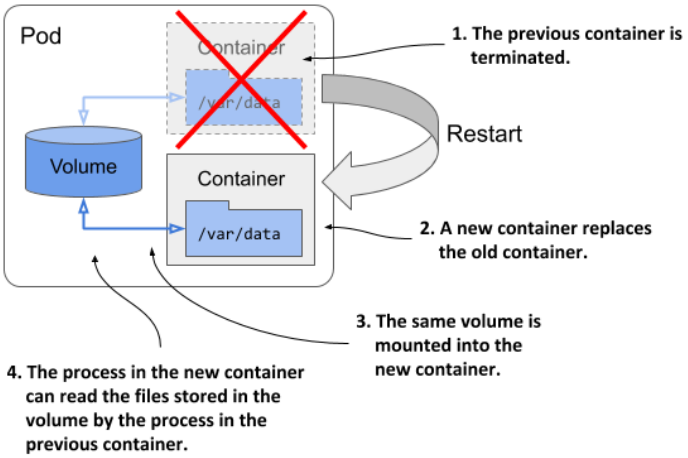

Figure 7.5 Volumes ensure that part of the container's filesystem is persisted across restarts

It is up to the author of the application to determine which files must be retained on restart. Normally you want to preserve data representing the application's state, but you may not want to preserve files that contain the application's locally cached data, as this prevents the container from starting fresh when it's restarted. Starting fresh every time may allow the application to heal itself when corruption of the local cache causes it to crash. Just restarting the container and using the same corrupted files could result in an endless crash loop.

> **TIP** Before you mount a volume in a container to preserve files across container restarts, consider how this affects the container's self-healing capability.

MOUNTING MULTIPLE VOLUMES IN A CONTAINER

A pod can have multiple volumes and each container can mount zero or more of these volumes in different locations, as shown in the following figure.

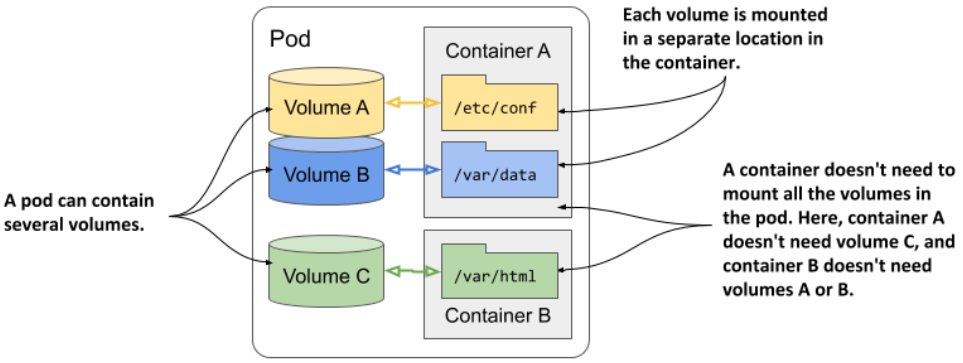

Figure 7.6 A pod can contain multiple volumes and a container can mount multiple volumes

The reason why you might want to mount multiple volumes in one container is that these volumes may serve different purposes and can be of different types with different performance characteristics.

In pods with more than one container, some volumes can be mounted in some containers but not in others. This is especially useful when a volume contains sensitive information that should only be accessible to some containers.

SHARING FILES BETWEEN MULTIPLE CONTAINERS

A volume can be mounted in more than one container so that applications running in these containers can share files. As discussed in chapter 5, a pod can combine a main application container with sidecar containers that extend the behavior of the main application. In some cases, the containers must read or write the same files.

For example, you could create a pod that combines a web server running in one container with a content-producing agent running in another container. The content agent container generates the static content that the web server then delivers to its clients. Each of the two containers performs a single task that has no real value on its own. However, as the next figure shows, if you add a volume to the pod and mount it in both containers, you enable these containers to become a complete system that provides a valuable service and is more than the sum of its parts.

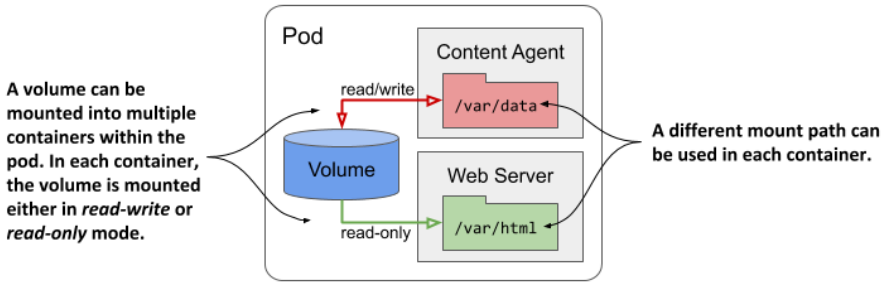

Figure 7.7 A volume can be mounted into more than one container

The same volume can be mounted at different places in each container, depending on the needs of the container itself. If the content agent writes content to `/var/data`, it makes sense to mount the volume there. Since the web server expects the content to be in `/var/html`, the container running it has the volume mounted at this location.

In the figure you'll also notice that the volume mount in each container can be configured either as read/write or as read-only. Because the content agent needs to write to the volume whereas the web server only reads from it, the two mounts are configured differently. In the interest of security, it's advisable to prevent the web server from writing to the volume, since this could allow an attacker to compromise the system if the web server software has a vulnerability that allows attackers to write arbitrary files to the filesystem and execute them.

Other examples of using a single volume in two containers are cases where a sidecar container runs a tool that processes or rotates the web server logs or when an init container creates configuration files for the main application container.

PERSISTING DATA ACROSS POD INSTANCES

A volume is tied to the lifecycle of the pod and only exists for as long as the pod exists, but depending on the volume type, the files in the volume can remain intact after the pod and volume disappear and can later be mounted into a new volume.

As the following figure shows, a pod volume can map to persistent storage outside the pod. In this case, the file directory representing the volume isn't a local file directory that persists data only for the duration of the pod, but is instead a volume mount to an existing, typically network-attached storage volume (NAS) whose lifecycle isn't tied to any pod. The data stored in the volume is thus persistent and can be used by the application even after the pod it runs in is replaced with a new pod running on a different worker node.

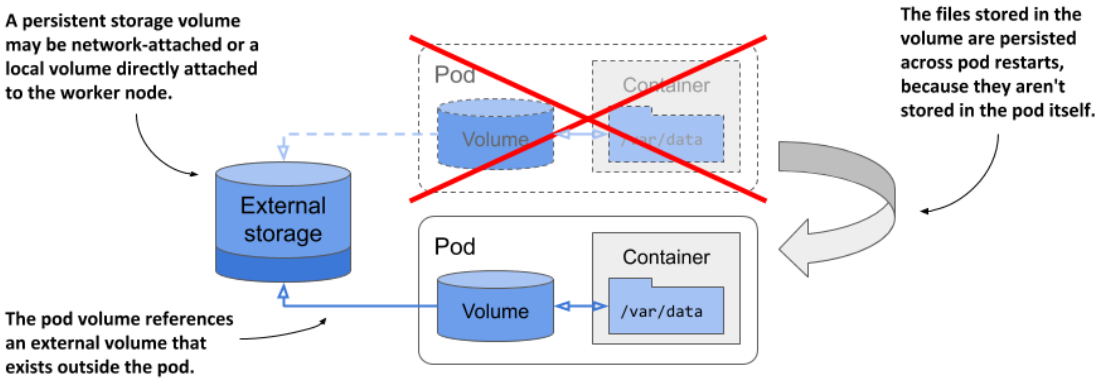

Figure 7.8 Pod volumes can also map to storage volumes that persist across pod restarts

If the pod is deleted and a new pod is created to replace it, the same network-attached storage volume can be attached to the new pod instance so that it can access the data stored there by the previous instance.

SHARING DATA BETWEEN PODS

Depending on the technology that provides the external storage volume, the same external volume can be attached to multiple pods simultaneously, allowing them to share data. The following figure shows a scenario where three pods each define a volume that is mapped to the same external persistent storage volume.

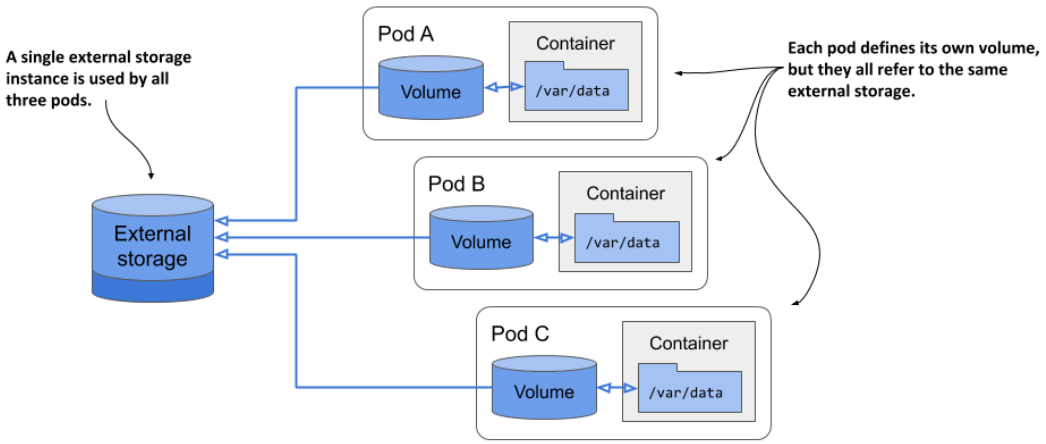

Figure 7.9 Using volumes to share data between pods

In the simplest case, the persistent storage volume could be a simple local directory on the worker node's filesystem, and the three pods have volumes that map to that directory. If all three pods are running on the same node, they can share files through this directory.

If the persistent storage is a network-attached storage volume, the pods may be able to use it even when they are deployed to different nodes. However, this depends on whether the underlying storage technology supports concurrently attaching the network volume to multiple computers.

While technologies such as Network File System (NFS) allow you to attach the volume in read/write mode on multiple computers, other technologies typically available in cloud environments, such as the Google Compute Engine Persistent Disk, allow the volume to be used either in read/write mode on a single cluster node, or in read-only mode on many nodes.

INTRODUCING THE AVAILABLE VOLUME TYPES

When you add a volume to a pod, you must specify the volume type. A wide range of volume types is available. Some are generic, while others are specific to the storage technologies used underneath. Here's a non-exhaustive list of the supported volume types:

- `emptyDir`—A simple directory that allows the pod to store data for the duration of its life cycle. The directory is created just before the pod starts and is initially empty - hence the name. The `gitRepo` volume, which is now deprecated, is similar, but is initialized by cloning a Git repository. Instead of using a `gitRepo` volume, it is recommended to use an `emptyDir` volume and initialize it using an init container.
- `hostPath`—Used for mounting files from the worker node's filesystem into the pod.
- `nfs`—An NFS share mounted into the pod.
- `gcePersistentDisk` (Google Compute Engine Persistent Disk), `awsElasticBlockStore` (Amazon Web Services Elastic Block Store), `azureFile` (Microsoft Azure File Service), `azureDisk` (Microsoft Azure Data Disk)—Used for mounting cloud provider-specific storage.
- `cephfs`, `cinder`, `fc`, `flexVolume`, `flocker`, `glusterfs`, `iscsi`, `portworxVolume`, `quobyte`, `rbd`, `scaleIO`, `storageos`, `photonPersistentDisk`, `vsphereVolume`—Used for mounting other types of network storage.
- `configMap`, `secret`, `downwardAPI`, and the `projected` volume type—Special types of volumes used to expose information about the pod and other Kubernetes objects through files. They are typically used to configure the application running in the pod. You'll learn about them in chapter 9.
- `persistentVolumeClaim`—A portable way to integrate external storage into pods. Instead of pointing directly to an external storage volume, this volume type points to a `PersistentVolumeClaim` object that points to a `PersistentVolume` object that finally references the actual storage. This volume type requires a separate explanation, which you'll find in the next chapter.
- `csi`—A pluggable way of adding storage via the Container Storage Interface. This volume type allows anyone to implement their own storage driver that is then referenced in the `csi` volume definition. During pod setup, the CSI driver is called to attach the volume to the pod.

These volume types serve different purposes. The following sections cover the most representative volume types and help you to gain a general understanding of volumes.

7.2 Using an emptyDir volume

The simplest volume type is `emptyDir`. As its name suggests, a volume of this type starts as an empty directory. When this type of volume is mounted in a container, files written by the application to the path where the volume is mounted are preserved for the duration of the pod's existence.

This volume type is used in single-container pods when data must be preserved even if the container is restarted. It's also used when the container's filesystem is marked read-only, and you want part of it to be writable. In pods with two or more containers, an `emptyDir` volume is used to share data between them.

7.2.1 Persisting files across container restarts

Let's add an `emptyDir` volume to the quiz pod from section 7.1.1 to ensure that its data isn't lost when the MongoDB container restarts.

ADDING AN EMPTYDIR VOLUME TO A POD

You'll modify the definition of the quiz pod so that the MongoDB process writes its files to the volume instead of the filesystem of the container it runs in, which is perishable. A visual representation of the pod is given in the next figure.

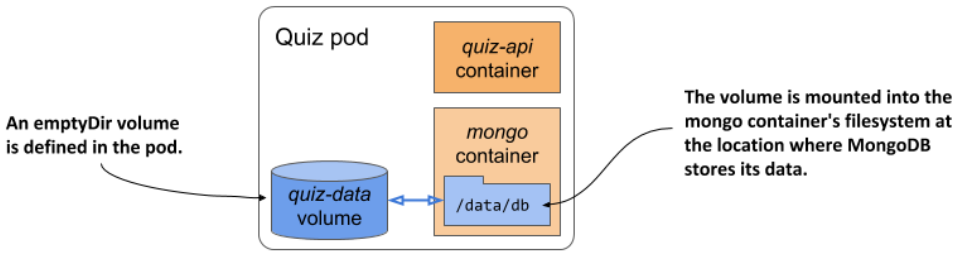

Figure 7.10 The quiz pod with an emptyDir volume for storing MongoDB data files

Two changes to the pod manifest are required to achieve this:

13. An `emptyDir` volume must be added to the pod.
14. The volume must be mounted into the container.

The following listing shows the new pod manifest with these two changes highlighted in bold. You'll find the manifest in the file Chapter07/pod.quiz.emptydir.yaml.

Listing 7.2 The quiz pod with an emptyDir volume for the mongo container

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

```
metadata:
  name: quiz
spec:
  volumes:    #A
  - name: quiz-data    #A
    emptyDir: {}    #A
  containers:
  - name: quiz-api
    image: luksa/quiz-api:0.1
    ports:
    - name: http
      containerPort: 8080
  - name: mongo
    image: mongo
    volumeMounts:    #B
    - name: quiz-data    #B
      mountPath: /data/db    #B
```

#A An emptyDir volume with the name quiz-data is defined.
#B The quiz-data volume is mounted into the mongo container's filesystem at the location /data/db.

The listing shows that an `emptyDir` volume named `quiz-data` is defined in the `spec.volumes` array of the pod manifest and that it is mounted into the `mongo` container's filesystem at the location `/data/db`. The following two sections explain more about the volume and the volume mount definitions.

CONFIGURING THE EMPTYDIR VOLUME

In general, each volume definition must include a `name` and a type, which is indicated by the name of the nested field (for example: `emptyDir`, `gcePersistentDisk`, `nfs`, and so on). This field typically contains several sub-fields that allow you to configure the volume. The set of sub-fields that you set depends on the volume type.

For example, the `emptyDir` volume type supports two fields for configuring the volume. They are explained in the following table.

Table 7.1 Configuration options for an emptyDir volume

Field	Description
`medium`	The type of storage medium to use for the directory. If left empty, the default medium of the host node is used (the directory is created on one of the node's disks). The only other supported option is `Memory`, which causes the volume to use `tmpfs`, a virtual memory filesystem where the files are kept in memory instead of on the hard disk.
`sizeLimit`	The total amount of local storage required for the directory, whether on disk or in memory. For example, to set the maximum size to ten mebibytes, you set this field to `10Mi`.

NOTE The `emptyDir` field in the volume definition defines neither of these properties. The curly braces `{}` have been added to indicate this explicitly, but they can be omitted.

MOUNTING THE VOLUME IN A CONTAINER

Defining a volume in the pod is only half of what you need to do to make it available in a container. The volume must also be mounted in the container. This is done by referencing the volume by name in the `volumeMounts` array in the container definition.

In addition to the `name`, a volume mount definition must also include the `mountPath` - the path within the container where the volume should be mounted. In listing 7.2, the volume is mounted at `/data/db` because that's where MongoDB stores its files. You want these files to be written to the volume instead of the container's filesystem, which is ephemeral.

The full list of supported fields in a volume mount definition is presented in the following table.

Table 7.2 Configuration options for a volume mount

Field	Description
`name`	The name of the volume to mount. This must match one of the volumes defined in the pod.
`mountPath`	The path within the container at which to mount the volume.
`readOnly`	Whether to mount the volume as read-only. Defaults to `false`.
`mountPropagation`	Specifies what should happen if additional filesystem volumes are mounted inside the volume. Defaults to `None`, which means that the container won't receive any mounts that are mounted by the host, and the host won't receive any mounts that are mounted by the container. `HostToContainer` means that the container will receive all mounts that are mounted into this volume by the host, but not the other way around. `Bidirectional` means that the container will receive mounts added by the host, and the host will receive mounts added by the container.
`subPath`	Defaults to "" which indicates that the entire volume is to be mounted into the container. When set to a non-empty string, only the specified `subPath` within the volume is mounted into the container.
`subPathExpr`	Just like `subPath` but can have environment variable references using the syntax `$(ENV_VAR_NAME)`. Only environment variables that are explicitly defined in the container definition are applicable. Implicit variables such as `HOSTNAME` will not be resolved. You'll learn how to specify environment variables in chapter 9.

In most cases, you only specify the `name`, `mountPath` and whether the mount should be `readOnly`. The `mountPropagation` option comes into play for advanced use-cases where additional mounts are added to the volume's file tree later, either from the host or from the container. The `subPath` and `subPathExpr` options are useful when you want to use a single volume with multiple directories that you want to mount to different containers instead of using multiple volumes.

The `subPathExpr` option is also used when a volume is shared by multiple pod replicas. In chapter 9, you'll learn how to use the Downward API to inject the name of the pod into an environment variable. By referencing this variable in `subPathExpr`, you can configure each replica to use its own subdirectory based on its name.

UNDERSTANDING THE LIFESPAN OF AN EMPTYDIR VOLUME

If you replace the quiz pod with the one in listing 7.2 and insert questions into the database, you'll notice that the questions you add to the database remain intact, regardless of how often the container is restarted. This is because the volume's lifecycle is tied to that of the pod.

To see this is the case, insert the question(s) into the MongoDB database as you did in section 7.1.1. I suggest using the shell script in the file Chapter07/insert-question.sh so that

you don’t have to type the entire question JSON again. After you add the question, count the number of questions in the database as follows:

```
$ kubectl exec -it quiz -c mongo -- mongo kiada --quiet --eval "db.questions.count()"
1    #A
```

#A The number of questions in the database

Now shut down the MongoDB server:

```
$ kubectl exec -it quiz -c mongo -- mongo admin --eval "db.shutdownServer()"
```

Check that the `mongo` container was restarted:

```
$ kubectl get po quiz
NAME   READY   STATUS    RESTARTS   AGE
quiz   2/2     Running   1          10m    #A
```

#A The restart count shows that one container was restarted

After the container restarts, recheck the number of questions in the database:

```
$ kubectl exec -it quiz -c mongo -- mongo kiada --quiet --eval "db.questions.count()"
1    #A
```

#A The data has survived the container restart

Restarting the container no longer causes the files to disappear because they no longer reside in the container's filesystem. They are stored in the volume. But where exactly is that? Let's find out.

UNDERSTANDING WHERE THE FILES IN AN EMPTYDIR VOLUME ARE STORED

As you can see in the following figure, the files in an `emptyDir` volume are stored in a directory in the host node's filesystem. It's nothing but a normal file directory. This directory is mounted into the container at the desired location.

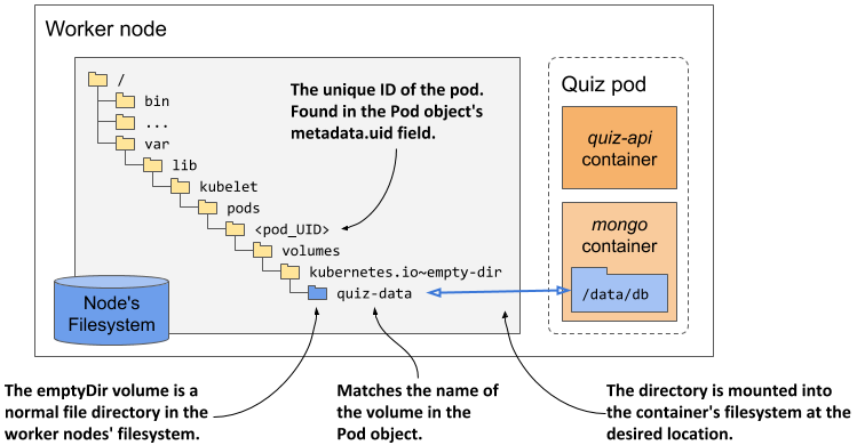

Figure 7.11 The emptyDir is a normal file directory in the node's filesystem that's mounted into the container

The directory is typically located at the following location in the node's filesystem:

```
/var/lib/kubelet/pods/<pod_UID>/volumes/kubernetes.io~empty-dir/<volume_name>
```

The `pod_UID` is the unique ID of the pod, which you'll find the Pod object's `metadata` section. If you want to see the directory for yourself, run the following command to get the `pod_UID`:

```
$ kubectl get po quiz -o json | jq .metadata.uid
"4f49f452-2a9a-4f70-8df3-31a227d020a1"
```

The `volume_name` is the name of the volume in the pod manifest - in the quiz pod, the name is `quiz-data`.

To get the name of the node that runs the pod, use `kubectl get po quiz -o wide` or the following alternative:

```
$ kubectl get po quiz -o json | jq .spec.nodeName
```

Now you have everything you need. Try to log into the node and inspect the contents of the directory. You'll notice that the files match those in the `mongo` container's `/data/db` directory.

If you delete the pod, the directory is deleted. This means that the data is lost once again. You'll learn how to persist it properly by using external storage volumes in section 7.3.

CREATING THE EMPTYDIR VOLUME IN MEMORY

The `emptyDir` volume in the previous example created a directory on the actual drive of the worker node that runs your pod, so its performance depends on the type of drive installed on the node. If you want the I/O operations on the volume to be as fast as possible, you can instruct Kubernetes to create the volume using the *tmpfs* filesystem, which keeps files in memory. To do this, set the `emptyDir's medium` field to `Memory` as in the following snippet:

```
volumes:
  - name: content
    emptyDir:
      medium: Memory    #A
```

#A This directory should be stored in memory.

Creating the `emptyDir` volume in memory is also a good idea whenever it's used to store sensitive data. Because the data is not written to disk, there is less chance that the data will be compromised and persisted longer than desired. As you'll learn in chapter 9, Kubernetes uses the same in-memory approach when it exposes the data from the *Secret* object kind in the container.

SPECIFYING THE SIZE LIMIT FOR THE EMPTYDIR VOLUME

The size of an `emptyDir` volume can be limited by setting the `sizeLimit` field. Setting this field is especially important for in-memory volumes when the overall memory usage of the pod is limited by so-called *resource limits*. You'll learn about this in chapter 20.

Next, let's see how an `emptyDir` volume is used to share files between containers of the same pod.

7.2.2 Populating an emptyDir volume with data using an init container

Every time you create the quiz pod from the previous section, the MongoDB database is empty, and you have to insert the questions manually. Let's improve the pod by automatically populating the database when the pod starts.

Many ways of doing this exist. You could run the MongoDB container locally, insert the data, commit the container state into a new image and use that image in your pod. But then you'd have to repeat the process every time a new version of the MongoDB container image is released.

Fortunately, the MongoDB container image provides a mechanism to populate the database the first time it's started. On start-up, if the database is empty, it invokes any .js and .sh files that it finds in the /docker-entrypoint-initdb.d directory. All you need to do is get the file into that location. Again, you could build a new MongoDB image with the file in that location, but you'd run into the same problem as described previously. An alternative solution is to use a volume to inject the file into that location of the MongoDB container's filesystem. But how do you get the file into the volume in the first place?

Kubernetes provides a special type of volume that is initialized by cloning a Git repository - the `gitRepo` volume. However, this type of volume is now deprecated. The proposed alternative is to use an `emptyDir` volume that you initialize with an init container that executes the `git clone` command. You could use this approach, but this would mean that the pod must make a network call to fetch the data.

Another, more generic way of populating an `emptyDir` volume, is to package the data into a container image and copy the data files from the container to the volume when the container starts. This removes the dependency on any external systems and allows the pod to run regardless of the network connectivity status.

To help you visualize the pod, look at the following figure.

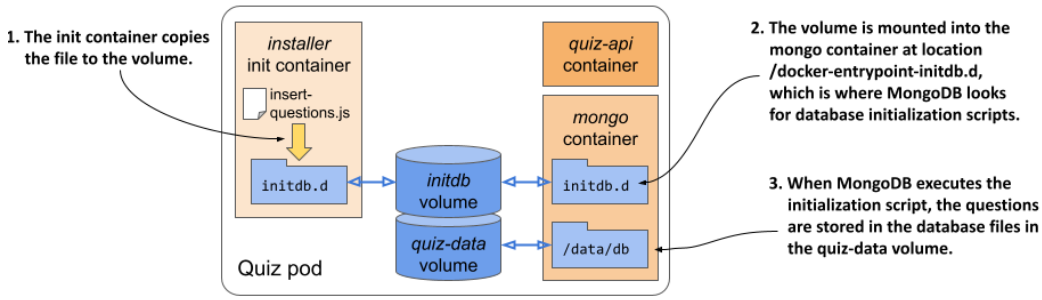

Figure 7.12 Using an init container to initialize an emptyDir volume

When the pod starts, first the volumes and then the init container is created. The initdb volume is mounted into this init container. The container image contains the insert-questions.js file, which the container copies to the volume when it runs. Then the copy operation is complete, the init container finishes and the pod's main containers are started. The initdb volume is mounted into the mongo container at the location where MongoDB looks for database

initialization scripts. On first start-up, MongoDB executes the insert-questions.js script. This inserts the questions into the database. As in the previous version of the pod, the database files are stored in the quiz-data volume to allow the data to survive container restarts.

You'll find the insert-questions.js file and the Dockerfile required to build init container image in the book's code repository. The following listing shows part of the insert-questions.js file.

Listing 7.3 The contents of the insert-questions.js file

```
db.getSiblingDB("kiada").questions.insertMany(    #A
[{    #B
    "id": 1,    #B
    "text": "What is kubectl?",    #B
    ...    #B
```

#A This command inserts documents into the questions collection of the kiada database.
#B This is the first document.

The Dockerfile for the container image is shown in the next listing.

Listing 7.4 Dockerfile for the quiz-initdb-script-installer:0.1 container image

```
FROM busybox
COPY insert-questions.js /    #A
CMD cp /insert-questions.js /initdb.d/ \    #B
    && echo "Successfully copied insert-questions.js to /initdb.d" \    #B
    || echo "Error copying insert-questions.js to /initdb.d"    #B
```

#A Adds the file to the container image
#B When the container runs, it copis the file to the /docker-entrypoint-initdb.d directory and prints a status message to the standard output.

Use these two files to build the image or use the image that I've built. You'll find it at `docker.io/luksa/quiz-initdb-script-installer:0.1`.

After you've got the container image, modify the pod manifest from the previous section so its contents match the next listing (the resulting file is Chapter07/pod.quiz.emptydir.init.yaml). The lines that you must add are highlighted in bold font.

Listing 7.5 Using an init container to initialize an emptyDir volume

```
apiVersion: v1
kind: Pod
metadata:
  name: quiz
spec:
  volumes:
  - name: initdb    #A
    emptyDir: {}    #A
  - name: quiz-data
    emptyDir: {}
  initContainers:
  - name: installer    #B
    image: luksa/quiz-initdb-script-installer:0.1    #B
    volumeMounts:    #B
    - name: initdb    #B
      mountPath: /initdb.d    #B
  containers:
  - name: quiz-api
    image: luksa/quiz-api:0.1
    ports:
    - name: http
      containerPort: 8080
  - name: mongo
    image: mongo
    volumeMounts:
    - name: quiz-data
      mountPath: /data/db
    - name: initdb    #C
      mountPath: /docker-entrypoint-initdb.d/    #C
      readOnly: true    #C
```

#A The initdb emptyDir volume is defined here.
#B The volume is mounted in the init container at the location to which the container copies the insert-questions.js file.
#C The same volume is also mounted in the mongo container at the location where MongoDB looks for initialization scripts.

The listing shows that the initdb volume is mounted into the init container. After this container copies the insert-questions.js file to the volume, it terminates and allows the mongo and quiz-api containers to start. Because the initdb volume is mounted in the /docker-entrypoint-initdb.d directory in the mongo container, MongoDB executes the .js file, which populates the database with questions.

You can delete the old quiz pod and deploy this new version of the pod. You'll see that the database gets populated every time you deploy the pod.

7.2.3 Sharing files between containers

As you saw in the previous section, an emptyDir volume can be initialized with an init container and then used by one of the pod's main containers. But a volume can also be used by multiple main containers concurrently. The quiz-api and the mongo containers that are in the quiz pod don't need to share files, so you'll use a different example to learn how volumes are shared between containers.

Remember the quote pod from the previous chapter? The one that uses a post-start hook to run the `fortune` command. The command writes a quote from this book into a file that is then served by the Nginx web server. The quote pod currently serves the same quote throughout the lifetime of the pod. This isn't that interesting. Let's build a new version of the pod, where a new quote is served every 60 seconds.

You'll retain Nginx as the web server but will replace the post-start hook with a container that periodically runs the `fortune` command to update the file where the quote is stored. Let's call this container `quote-writer`. The Nginx server will continue to be in the `nginx` container.

As visualized in the following figure, the pod now has two containers instead of one. To allow the `nginx` container to see the file that the `quote-writer` creates, a volume must be defined in the pod and mounted into both containers.

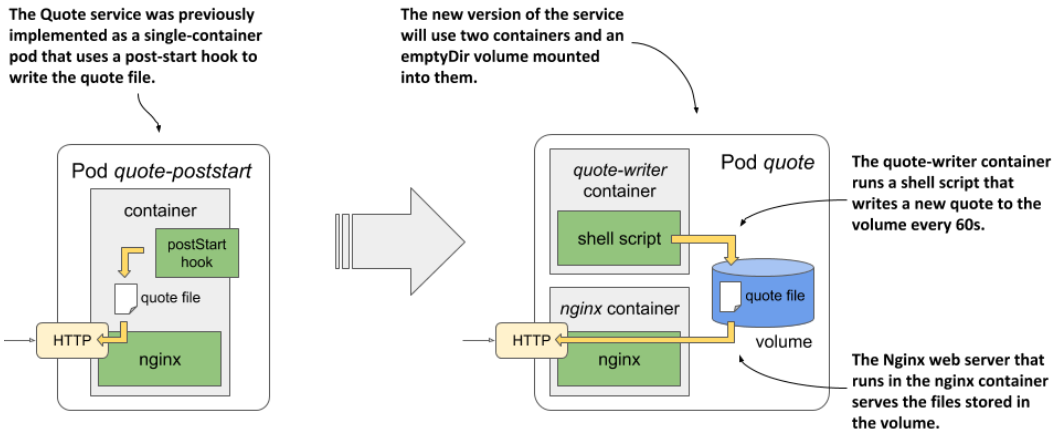

Figure 7.13 The new version of the Quote service uses two containers and a shared volume

CREATING A POD WITH TWO CONTAINERS AND A SHARED VOLUME

The image for the `quote-writer` container is available at `docker.io/luksa/quote-writer:0.1`, but you can also build it yourself from the files in the Chapter07/quote-writer-0.1 directory. The `nginx` container will continue to use the existing `nginx:alpine` image.

The pod manifest for the new quote pod is shown in the next listing. You'll find it in file Chapter07/pod.quote.yaml.

Listing 7.6 A pod with two containers that share a volume

```
apiVersion: v1
kind: Pod
metadata:
  name: quote
spec:
  volumes:    #A
  - name: shared    #A
    emptyDir: {}    #A
  containers:
  - name: quote-writer    #B
    image: luksa/quote-writer:0.1    #B
    volumeMounts:    #C
    - name: shared    #C
      mountPath: /var/local/output    #C
  - name: nginx    #D
    image: nginx:alpine    #D
    volumeMounts:    #E
    - name: shared    #E
      mountPath: /usr/share/nginx/html    #E
      readOnly: true    #E
    ports:
    - name: http
      containerPort: 80
```

#A An emptyDir volume with the name shared is defined.
#B The quote-writer container writes the quote to a file.
#C The shared volume is mounted into the quote-writer container.
#D The nginx container serves the quote file.
#E The shared volume is mounted into the nginx container.

The pod consists of two containers and a single volume, which is mounted in both containers, but at a different location in each container. The reason for this is that the `quote-writer` container writes the `quote` file to the `/var/local/output` directory, whereas the `nginx` container serves files from the `/usr/share/nginx/html` directory.

> **NOTE** Since the two containers start at the same time, there can be a short period where nginx is already running, but the quote hasn't been generated yet. One way of making sure this doesn't happen is to generate the initial quote using an init container, as explained in section 7.2.3.

RUNNING THE POD

When you create the pod from the manifest, the two containers start and continue running until the pod is deleted. The `quote-writer` container writes a new quote to the file every 60 seconds, and the `nginx` container serves this file. After you create the pod, use the `kubectl port-forward` command to open a communication tunnel to the pod:

```
$ kubectl port-forward quote 1080:80
```

In another terminal, verify that the server responds with a different quote every 60 seconds by running the following command several times:

```
$ curl localhost:1080/quote
```

Alternatively, you can also display the contents of the file using either of the following two commands:

```
$ kubectl exec quote -c quote-writer -- cat /var/local/output/quote
$ kubectl exec quote -c nginx -- cat /usr/share/nginx/html/quote
```

As you can see, one of them prints the contents of the file from within the `quote-writer` container, whereas the other command prints the contents from within the `nginx` container. Because the two paths point to the same `quote` file on the shared volume, the output of the commands is identical.

7.3 Using external storage in pods

An `emptyDir` volume is a dedicated directory created specifically for and used exclusively by the pod in which the volume is defined. When the pod is deleted, the volume and its contents are deleted. However, other types of volumes don't create a new directory, but instead mount an existing external directory in the filesystem of the container. The contents of this volume can survive multiple instantiations of the same pod and can even be shared by multiple pods. These are the types of volumes we'll explore next.

To learn how external storage is used in a pod, you'll create a pod that runs the document-oriented database MongoDB. To ensure that the data stored in the database is persisted, you'll add a volume to the pod and mount it in the container at the location where MongoDB writes its data files.

The tricky part of this exercise is that the type of persistent volumes available in your cluster depends on the environment in which the cluster is running. At the beginning of this book, you learned that Kubernetes could reschedule a pod to another node at any time. To ensure that the quiz pod can still access its data, it should use network-attached storage instead of the worker node's local drive.

Ideally, you should use a proper Kubernetes cluster, such as GKE, for the following exercises. Unfortunately, clusters provisioned with Minikube or kind don't provide any kind of network storage volume out of the box. So, if you're using either of these tools, you'll need to resort to using node-local storage provided by the so-called `hostPath` volume type, but this volume type is not explained until section 7.4.

7.3.1 Using a Google Compute Engine Persistent Disk as a volume

If you use Google Kubernetes Engine to run the exercises in this book, your cluster nodes run on Google Compute Engine (GCE). In GCE, persistent storage is provided via GCE Persistent Disks. Kubernetes supports adding them to your pods via the `gcePersistentDisk` volume type.

> **NOTE** To adapt this exercise for use with other cloud providers, use the appropriate volume type supported by the cloud provider. Consult the documentation provided by the cloud provider to determine how to create the storage volume and how to mount it into the pod.

CREATING A GCE PERSISTENT DISK

Before you can use the GCE Persistent Disk volume in your pod, you must create the disk itself. It must reside in the same zone as your Kubernetes cluster. If you don't remember in which zone you created the cluster, you can see it by listing your Kubernetes clusters using the `gcloud` command as follows:

```
$ gcloud container clusters list
NAME    ZONE            MASTER_VERSION   MASTER_IP        ...
kiada   europe-west3-c  1.14.10-gke.42   104.155.84.137   ...
```

In my case, the command output indicates that the cluster is in zone `europe-west3-c`, so I have to create the GCE Persistent Disk there. Create the disk in the correct zone as follows:

```
$ gcloud compute disks create --size=10GiB --zone=europe-west3-c quiz-data
WARNING: You have selected a disk size of under [200GB]. This may result in poor I/O
       performance.
For more information, see: https://developers.google.com/compute/docs/disks#pdperformance.
Created [https://www.googleapis.com/.../zones/europe-west3-c/disks/quiz-data].
NAME       ZONE           SIZE_GB  TYPE         STATUS
quiz-data  europe-west3-c  10       pd-standard  READY
```

This command creates a GCE Persistent Disk called `quiz-data` with 10GiB of space. You can freely ignore the disk size warning, because it doesn't affect the exercises you're about to run. You may also see an additional warning that the disk is not yet formatted. You can ignore that, too, because formatting is done automatically when you use the disk in your pod.

CREATING A POD WITH A GCEPERSISTENTDISK VOLUME

Now that you have set up your physical storage, you can use it in a volume inside your quiz pod. You'll create the pod from the YAML in the following listing (Chapter07/pod.quiz.gcepd.yaml). The highlighted lines are the only difference from the pod.quiz.emptydir.yaml file that you deployed in section 7.2.1.

Listing 7.7 Using a gcePersistentDisk volume in the quiz pod

```
apiVersion: v1
kind: Pod
metadata:
  name: quiz
spec:
  volumes:
  - name: quiz-data
    gcePersistentDisk:    #A
      pdName: quiz-data     #B
      fsType: ext4     #C
  containers:
  - name: quiz-api
    image: luksa/quiz-api:0.1
    ports:
    - name: http
      containerPort: 8080
  - name: mongo
    image: mongo
    volumeMounts:
    - name: quiz-data
```

```
      mountPath: /data/db
```

#A The volume points to a GCE Persistent Disk.you created earlier.
#B The name of the GCE Persistent Disk
#C The filesystem type

> **NOTE** If you created your cluster with Minikube or kind, you can't use a GCE Persistent Disk. Use the file `pod.mongodb.hostpath.yaml`, which uses a `hostPath` volume in place of the GCE PD. This type of volume uses node-local instead of network storage, so you must ensure that the pod is always deployed to the same node. This is always true in Minikube because it creates a single node cluster. However, if you're using kind, create the pod from the file `pod.mongodb.hostpath.kind.yaml` to ensure that the pod is always deployed to the same node.

The pod is visualized in the following figure. It contains a single volume that refers to the GCE Persistent Disk you created earlier. The volume is mounted in the `mongo` container at `/data/db`. This ensures that MongoDB writes its files to the persistent disk.

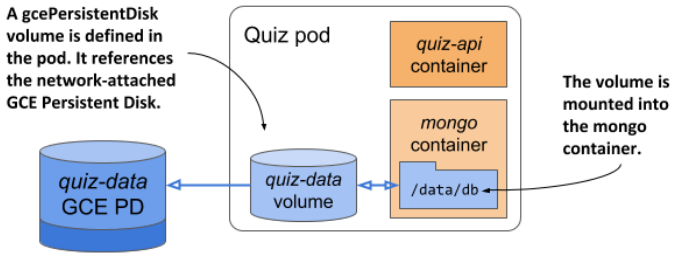

Figure 7.14 A GCE Persistent Disk referenced in a pod volume and mounted into the mongo container

VERIFYING THAT THE GCE PERSISTENT DISK PERSISTS DATA

Use the shell script in the file Chapter07/insert-question.sh to add a question to the MongoDB database. Confirm that the question is stored by using the following command:

```
$ kubectl exec -it quiz -c mongo -- mongo kiada --quiet --eval "db.questions.count()"
1    #A
```

#A The number of questions in the database

Okay, the database has the data. MongoDB's data files are stored in the `/data/db` directory, which is where you mounted the GCE Persistent Disk. Therefore, the data files should be stored on the GCE PD.

You can now safely delete the quiz pod and recreate it:

```
$ kubectl delete pod quiz
pod "quiz" deleted
$ kubectl apply -f pod.quiz.gcepd.yaml
pod "quiz" created
```

Since the new pod is an exact replica of the previous, it points to the same GCE Persistent Disk as the previous pod did. The mongo container should see the files that it wrote earlier, even if the new pod is scheduled to another node.

TIP You can see what node a pod is scheduled to by running `kubectl get po -o wide`.

NOTE If you use a kind-provisioned cluster, the pod is always scheduled to the same node.

After the pod starts, recheck the number of questions in the database:

```
$ kubectl exec -it quiz -c mongo -- mongo kiada --quiet --eval "db.questions.count()"
1    #A
```

#A The data is still present

As expected, the data still exists even though you deleted and recreated the pod. This confirms that you can use a GCE Persistent Disk to persist data across multiple instantiations of the same pod. To be perfectly precise, it isn't the same pod. These are two pods whose volumes point to the same underlying persistent storage volume.

You might wonder if you can use the same persistent disk in two or more pods at the same time. The answer to this question is not straightforward, because it requires the understanding of how external volumes are mounted in pods. I'll explain this in section 7.3.3. Before I do that, I need to explain how use external storage when your cluster doesn't run on Google's infrastructure.

7.3.2 Using other persistent volume types

In the previous exercise, I explained how to add persistent storage to a pod running in Google Kubernetes Engine. If you run your cluster elsewhere, you should use whatever volume type is supported by the underlying infrastructure.

For example, if your Kubernetes cluster runs on Amazon's AWS EC2, you can use an `awsElasticBlockStore` volume. If your cluster runs on Microsoft Azure, you can use the `azureFile` or the `azureDisk` volume. I won't go into detail about how to do this, but it's practically the same as in the previous example. You first need to create the actual underlying storage and then set the right fields in the volume definition.

USING AN AWS ELASTIC BLOCK STORE VOLUME

For example, if you want to use an AWS Elastic Block Store volume instead of the GCE Persistent Disk, you only need to change the volume definition as shown in the following listing (Chapter07/pod.quiz.aws.yaml).

Listing 7.8 Using an awsElasticBlockStore volume in the quiz pod

```
apiVersion: v1
kind: Pod
metadata:
  name: quiz
spec:
  volumes:
```

```
  - name: quiz-data
    awsElasticBlockStore:    #A
      volumeId: quiz-data    #B
      fsType: ext4    #C
  containers:
  - ...
```

#A This volume refers to an awsElasticBlockStore.
#B The ID of the EBS volume
#C The filesystem type

USING AN NFS VOLUME

If your cluster runs on your own servers, you have a range of other supported options for adding external storage to your pods. For example, to mount an NFS share, you only need to specify the NFS server and the path exported by the server, as shown in the following listing (Chapter07/pod.quiz.nfs.yaml).

Listing 7.9 Using an nfs volume in the quiz pod

```
...
  volumes:
  - name: quiz-data
    nfs:    #A
      server: 1.2.3.4    #B
      path: /some/path    #C
...
```

#A This volume refers to an NFS share.
#B IP address of the NFS server
#C File path exported by the server

> **NOTE** Although Kubernetes supports nfs volumes, the operating system running on the worker nodes provisioned by Minikube or kind might not support mounting nfs volumes.

USING OTHER STORAGE TECHNOLOGIES

Other supported options are `iscsi` for mounting an iSCSI disk resource, `glusterfs` for a GlusterFS mount, `rbd` for a RADOS Block Device, `flexVolume`, `cinder`, `cephfs`, `flocker`, `fc` (Fibre Channel), and others. You don't need to understand all these technologies. They're mentioned here to show you that Kubernetes supports a wide range of these technologies, and you can use the technologies that are available in your environment or that you prefer.

For details on the properties that you need to set for each of these volume types, you can either refer to the Kubernetes API definitions in the Kubernetes API reference or look up the information by running `kubectl explain pod.spec.volumes`. If you're already familiar with a particular storage technology, you should be able to use the `explain` command to easily find out how to configure the correct volume type (for example, for iSCSI you can see the configuration options by running `kubectl explain pod.spec.volumes.iscsi`).

WHY DOES KUBERNETES FORCE SOFTWARE DEVELOPERS TO UNDERSTAND LOW-LEVEL STORAGE?

If you're a software developer and not a system administrator, you might wonder if you really need to know all this low-level information about storage volumes? As a developer, should you

have to deal with infrastructure-related storage details when writing the pod definition, or should this be left to the cluster administrator?

At the beginning of this book, I explained that Kubernetes abstracts away the underlying infrastructure. The configuration of storage volumes explained earlier clearly contradicts this. Furthermore, including infrastructure-related information, such as the NFS server hostname directly in a pod manifest means that this manifest is tied to this specific Kubernetes cluster. You can't use the same manifest without modification to deploy the pod in another cluster.

Fortunately, Kubernetes offers another way to add external storage to your pods. One that divides the responsibility for configuring and using the external storage volume into two parts. The low-level part is managed by cluster administrators, while software developers only specify the high-level storage requirements for their applications. Kubernetes then connects the two parts.

You'll learn about this in the next chapter, but first you need a basic understanding of pod volumes. You've already learned most of it, but I still need to explain some details.

7.3.3 Understanding how external volumes are mounted

To understand the limitations of using external volumes in your pods, whether a pod references the volume directly or indirectly, as explained in the next chapter, you must be aware of the caveats associated with the way network storage volumes are actually attached to the pods.

Let's return to the issue of using the same network storage volume in multiple pods at the same time. What happens if you create a second pod and point it to the same GCE Persistent Disk?

I've prepared a manifest for a second MongoDB pod that uses the same GCE Persistent Disk. The manifest can be found in the file pod.quiz2.gcepd.yaml. If you use it to create the second pod, you'll notice that it never runs. Even after several minutes, it never gets past `ContainerCreating`:

```
$ kubectl get po
NAME       READY   STATUS              RESTARTS   AGE
quiz       2/2     Running             0          10m
quiz2      0/2     ContainerCreating   0          2m
```

> **NOTE** If your GKE cluster has a single worker node and the pod's status is `Pending`, the reason could be that there isn't enough unallocated CPU for the pod to fit on the node. Resize the cluster to at least two nodes with the command `gcloud container clusters resize <cluster-name> --size <number-of-nodes>`.

You can see why this is the case with the `kubectl describe pod quiz2` command. At the very bottom, you see a `FailedAttachVolume` event generated by the `attachdetach-controller`. The event has the following message:

```
AttachVolume.Attach failed for volume "quiz-data" : googleapi: Error 400:
RESOURCE_IN_USE_BY_ANOTHER_RESOURCE -    #A
The disk resource
'projects/kiada/zones/europe-west3-c/disks/quiz-data' is already being used by
'projects/kiada/zones/europe-west3-c/instances/gke-kiada-default-pool-xyz-1b27'    #B
```

```
#A The GCE Persistent Disk is already being used by another node.
#B The worker node that the GCE PD is attached to
```

The message indicates that the node hosting the `quiz2` pod can't attach the external volume because it's already in use by another node. If you check where the two pods are scheduled, you'll see that they are not on the same node:

```
$ kubectl get po -o wide
NAME    READY   STATUS              ... NODE
quiz    2/2     Running             ... gke-kiada-default-pool-xyz-1b27
quiz2   0/2     ContainerCreating   ... gke-kiada-default-pool-xyz-gqbj
```

The `quiz` pod runs on node `xyz-1b27`, whereas `quiz2` is on node `xyz-gqbj`. As is typically the case in cloud environments, you can't mount the same GCE Persistent Disk on multiple hosts simultaneously in read/write mode. You can only mount it on multiple hosts if you use the read-only mode.

Interestingly, the error message doesn't say that the disk is being used by the `quiz` pod, but by the node hosting the pod. This is an often overlooked detail about how external volumes are mounted into pods.

> **TIP** Use the following command to see which network volumes that are attached to a node: `kubectl get node <node-name> -o json | jq .status.volumesAttached`.

As the following figure shows, a network volume is mounted by the host node, and then the pod is given access to the mount point. The underlying storage technology may not allow a volume to be attached to more than one node at a time in read/write mode, but multiple pods on the same node *can* all use the volume in read/write mode.

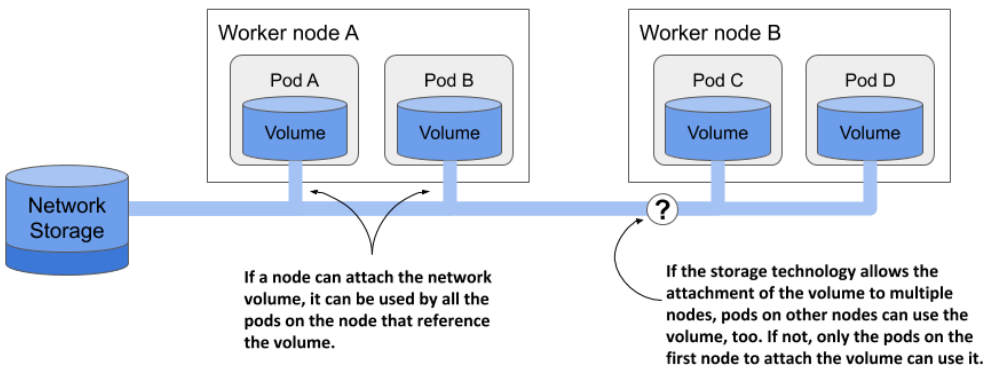

Figure 7.15 Network volumes are mounted by the host node and then exposed in pods

For most storage technologies available in the cloud, you can typically use the same network volume on multiple nodes simultaneously if you mount them in read-only mode. For example, pods scheduled to different nodes can use the same GCE Persistent Disk if it is mounted in read-only mode, as shown in the next listing.

Listing 7.10 Mounting a GCE Persistent Disk in read-only mode

```
kind: Pod
spec:
  volumes:
  - name: my-volume
    gcePersistentDisk:
      pdName: my-volume
      fsType: ext4
      readOnly: true     #A
```

#A This GCE Persistent Disk is mounted in read-only mode

It is important to consider this network storage limitation when designing the architecture of your distributed application. Replicas of the same pod typically can't use the same network volume in read/write mode. Fortunately, Kubernetes takes care of this, too. In chapter 13, you'll learn how to deploy stateful applications, where each pod instance gets its own network storage volume.

You're now done playing with these two quiz pods, so you can delete them. But don't delete the underlying GCE Persistent Disk yet. You'll use it again in the next chapter.

7.4 Accessing files on the worker node's filesystem

Most pods shouldn't care which host node they are running on, and they shouldn't access any files on the node's filesystem. System-level pods are the exception. They may need to read the node's files or use the node's filesystem to access the node's devices or other components via the filesystem. Kubernetes makes this possible through the `hostPath` volume type. I already mentioned it in the previous section, but this is where you'll learn when to actually use it.

7.4.1 Introducing the hostPath volume

A `hostPath` volume points to a specific file or directory in the filesystem of the host node, as shown in the next figure. Pods running on the same node and using the same path in their `hostPath` volume have access to the same files, whereas pods on other nodes do not.

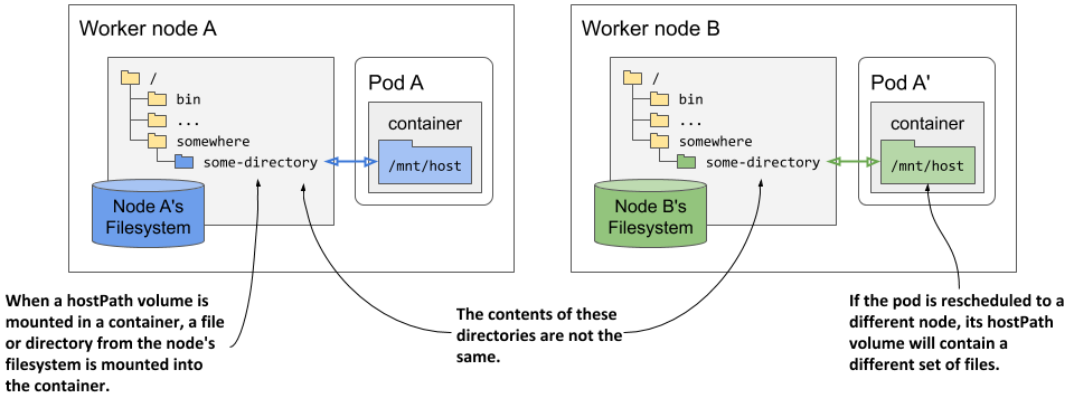

Figure 7.16 A `hostPath` volume mounts a file or directory from the worker node's filesystem into the container.

A `hostPath` volume is not a good place to store the data of a database unless you ensure that the pod running the database always runs on the same node. Because the contents of the volume are stored on the filesystem of a specific node, the database pod will not be able to access the data if it gets rescheduled to another node.

Typically, a `hostPath` volume is used in cases where the pod needs to read or write files in the node's filesystem that the processes running on the node read or generate, such as system-level logs.

The `hostPath` volume type is one of the most dangerous volume types in Kubernetes and is usually reserved for use in privileged pods only. If you allow unrestricted use of the `hostPath` volume, users of the cluster can do anything they want on the node. For example, they can use it to mount the Docker socket file (typically `/var/run/docker.sock`) in their container and then run the Docker client within the container to run any command on the host node as the root user. You'll learn how to prevent this in chapter 24.

7.4.2 Using a hostPath volume

To demonstrate how dangerous `hostPath` volumes are, let's deploy a pod that allows you to explore the entire filesystem of the host node from within the pod. The pod manifest is shown in the following listing.

Listing 7.11 Using a hostPath volume to gain access to the host node's filesystem

```
apiVersion: v1
kind: Pod
metadata:
  name: node-explorer
spec:
  volumes:
  - name: host-root                    #A
    hostPath:                          #A
      path: /                          #A
```

```
  containers:
  - name: node-explorer
    image: alpine
    command: ["sleep", "9999999999"]
    volumeMounts:                          #B
    - name: host-root                      #B
      mountPath: /host                     #B
```

#A The hostPath volume points to the root directory on the node's filesystem.
#B The volume is mounted in the container at /host.

As you can see in the listing, a `hostPath` volume must specify the `path` on the host that it wants to mount. The volume in the listing will point to the root directory on the node's filesystem, providing access to the entire filesystem of the node the pod is scheduled to.

After creating the pod from this manifest using `kubectl apply`, run a shell in the pod with the following command:

```
$ kubectl exec -it node-explorer -- sh
```

You can now navigate to the root directory of the node's filesystem by running the following command:

```
/ # cd /host
```

From here, you can explore the files on the host node. Since the container and the shell command are running as root, you can modify any file on the worker node. Be careful not to break anything.

> **NOTE** **If your cluster has more than one worker node, the pod runs on a randomly selected one. If you'd like to deploy the pod on a specific node, edit the file `node-explorer.specific-node.pod.yaml`, which you'll find in the book's code archive, and set the `.spec.nodeName` field to the name of the node you'd like to run the pod on. You'll learn about scheduling pods to a specific node or a set of nodes in later chapters.**

Now imagine you're an attacker that has gained access to the Kubernetes API and are able to deploy this type of pod in a production cluster. Unfortunately, at the time of writing, Kubernetes doesn't prevent regular users from using `hostPath` volumes in their pods and is therefore totally unsecure. As already mentioned, you'll learn how to secure the cluster from this type of attack in chapter 24.

SPECIFYING THE TYPE FOR A HOSTPATH VOLUME

In the previous example, you only specified the path for the `hostPath` volume, but you can also specify the `type` to ensure that the path represents what the process in the container expects (a file, a directory, or something else).

The following table explains the supported `hostPath` types:

Table 7.3 Supported hostPath volume types

Type	Description
`<empty>`	Kubernetes performs no checks before it mounts the volume.
`Directory`	Kubernetes checks if a directory exists at the specified path. You use this type if you want to mount a pre-existing directory into the pod and want to prevent the pod from running if the directory doesn't exist.
`DirectoryOrCreate`	Same as `Directory`, but if nothing exists at the specified path, an empty directory is created.
`File`	The specified path must be a file.
`FileOrCreate`	Same as `File`, but if nothing exists at the specified path, an empty file is created.
`BlockDevice`	The specified path must be a block device.
`CharDevice`	The specified path must be a character device.
`Socket`	The specified path must be a UNIX socket.

If the specified path doesn't match the type, the pod's containers don't run. The pod's events explain why the hostPath type check failed.

> **NOTE** When the type is `FileOrCreate` or `DirectoryOrCreate` and Kubernetes needs to create the file/directory, its file permissions are set to `644` (`rw-r--r--`) and `755` (`rwxr-xr-x`), respectively. In either case, the file/directory is owned by the user and group used to run the Kubelet.

7.5 Summary

This chapter has explained the basics of adding volumes to pods, but this was only the beginning. You'll learn more about this topic in the next chapter. So far, you've learned the following:

- Pods consist of containers and volumes. Each volume can be mounted at the desired location in the container's filesystem.
- Volumes are used to persist data across container restarts, share data between the containers in the pod, and even share data between the pods.
- Many volume types exist. Some are generic and can be used in any cluster regardless of the cluster environment, while others, such as the `gcePersistentDisk`, can only be used if the cluster runs on a specific cloud provider's infrastructure.
- An `emptyDir` volume is used to store data for the duration of the pod. It starts as an empty directory just before the pod's containers are started and is deleted when the pod terminates.
- The `gitRepo` volume is a deprecated volume type that is initialized by cloning a Git repository. Alternatively, an `emptyDir` volume can be used in combination with an init container that initializes the volume from Git or any other source.
- Network volumes are typically mounted by the host node and then exposed to the pod(s) on that node.
- Depending on the underlying storage technology, you may or may not be able to mount a network storage volume in read/write mode on multiple nodes simultaneously.
- By using a proprietary volume type in a pod manifest, the pod manifest is tied to a specific Kubernetes cluster. The manifest must be modified before it can be used in another cluster. Chapter 8 explains how to avoid this issue.
- The `hostPath` volume allows a pod to access any path in filesystem of the worker node. This volume type is dangerous because it allows users to make changes to the configuration of the worker node and run any process they want on the node.

In the next chapter, you'll learn how to abstract the underlying storage technology away from the pod manifest and make the manifest portable to any other Kubernetes cluster.

8

Persisting application data with PersistentVolumes

This chapter covers

- Using PersistentVolume objects to represent persistent storage
- Claiming persistent volumes with PersistentVolumeClaim objects
- Dynamic provisioning of persistent volumes
- Using node-local persistent storage

The previous chapter taught you how to mount a network storage volume into your pods. However, the experience was not ideal because you needed to understand the environment your cluster was running in to know what type of volume to add to your pod. For example, if your cluster runs on Google's infrastructure, you must define a `gcePersistentDisk` volume in your pod manifest. You can't use the same manifest to run your application on Amazon because GCE Persistent Disks aren't supported in their environment. To make the manifest compatible with Amazon, one must modify the volume definition in the manifest before deploying the pod.

You may remember from chapter 1 that Kubernetes is supposed to standardize application deployment between cloud providers. Using proprietary storage volume types in pod manifests goes against this premise.

Fortunately, there is a better way to add persistent storage to your pods. One where you don't refer to a specific storage technology within the pod. This chapter explains this improved approach.

> **NOTE** You'll find the code files for this chapter at https://github.com/luksa/kubernetes-in-action-2nd-edition/tree/master/Chapter08

8.1 Decoupling pods from the underlying storage technology

Ideally, a developer who deploys their applications on Kubernetes shouldn't need to know what storage technology the cluster provides, just as they don't need to know the characteristics of the physical servers used to run the pods. Details of the infrastructure should be handled by the people who run the cluster.

For this reason, when you deploy an application to Kubernetes, you typically don't refer directly to the external storage in the pod manifest, as you did in the previous chapter. Instead, you use an indirect approach that is explained in the following section.

One of the examples in the previous chapter shows how to use an NFS file share in a pod. The volume definition in the pod manifest contains the IP address of the NFS server and the file path exported by that server. This ties the pod definition to a specific cluster and prevents it from being used elsewhere.

As illustrated in the following figure, if you were to deploy this pod to a different cluster, you would typically need to change at least the NFS server IP. This means that the pod definition isn't portable across clusters. It must be modified each time you deploy it in a new Kubernetes cluster.

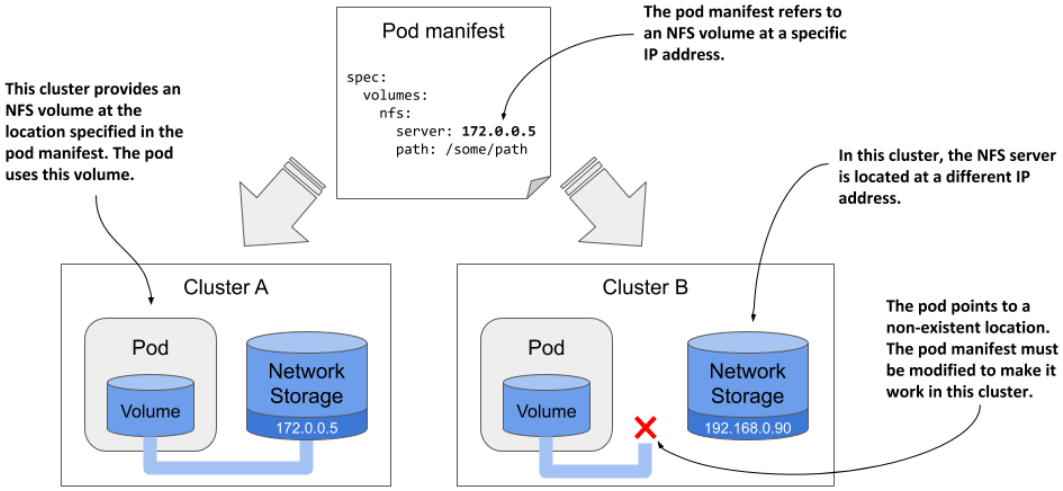

Figure 8.1 A pod manifest with infrastructure-specific volume information is not portable to other clusters

8.1.1 Introducing persistent volumes and claims

To make pod manifests portable across different cluster environments, the environment-specific information about the actual storage volume is moved to a PersistentVolume object, as shown in the next figure. A PersistentVolumeClaim object connects the pod to this PersistentVolume object.

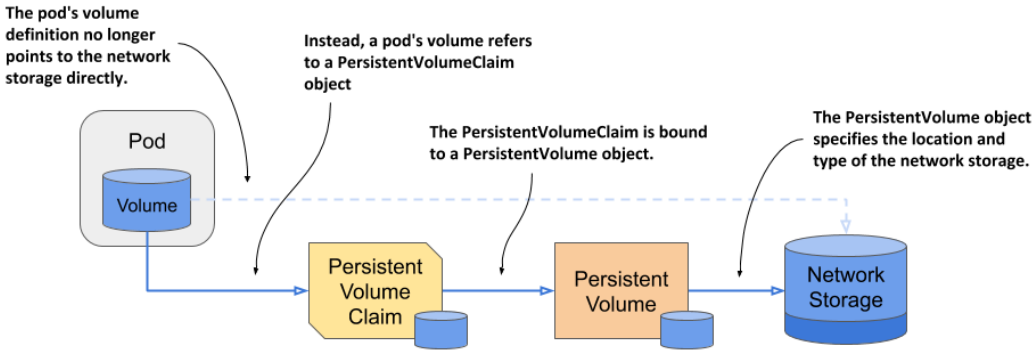

Figure 8.2 Using persistent volumes and persistent volume claims to attach network storage to pods

These two objects are explained next.

INTRODUCING PERSISTENT VOLUMES

As the name suggests, a PersistentVolume object represents a storage volume that is used to persist application data. As shown in the previous figure, the PersistentVolume object stores the information about the underlying storage and decouples this information from the pod.

When this infrastructure-specific information isn't in the pod manifest, the same manifest can be used to deploy pods in different clusters. Of course, each cluster must now contain a PersistentVolume object with this information. I agree that this approach doesn't seem to solve anything, since we've only moved information into a different object, but you'll see later that this new approach enables things that weren't possible before.

INTRODUCING PERSISTENT VOLUME CLAIMS

A pod doesn't refer directly to the PersistentVolume object. Instead, it points to a PersistentVolumeClaim object, which then points to the PersistentVolume.

As its name suggests, a PersistentVolumeClaim object represents a user's claim on the persistent volume. Because its lifecycle is not tied to that of the pod, it allows the ownership of the persistent volume to be decoupled from the pod. Before a user can use a persistent volume in their pods, they must first claim the volume by creating a PersistentVolumeClaim object. After claiming the volume, the user has exclusive rights to it and can use it in their pods. They can delete the pod at any time, and they won't lose ownership of the persistent volume. When the volume is no longer needed, the user releases it by deleting the PersistentVolumeClaim object.

USING A PERSISTENT VOLUME CLAIM IN A POD

To use the persistent volume in a pod, in its manifest you simply refer to the name of the persistent volume claim that the volume is bound to.

For example, if you create a persistent volume claim that gets bound to a persistent volume that represents an NFS file share, you can attach the NFS file share to your pod by adding a volume definition that points to the PersistentVolumeClaim object. The volume definition in the

pod manifest only needs to contain the name of the persistent volume claim and no infrastructure-specific information, such as the IP address of the NFS server.

As the following figure shows, when this pod is scheduled to a worker node, Kubernetes finds the persistent volume that is bound to the claim referenced in the pod, and uses the information in the PersistentVolume object to mount the network storage volume in the pod's container.

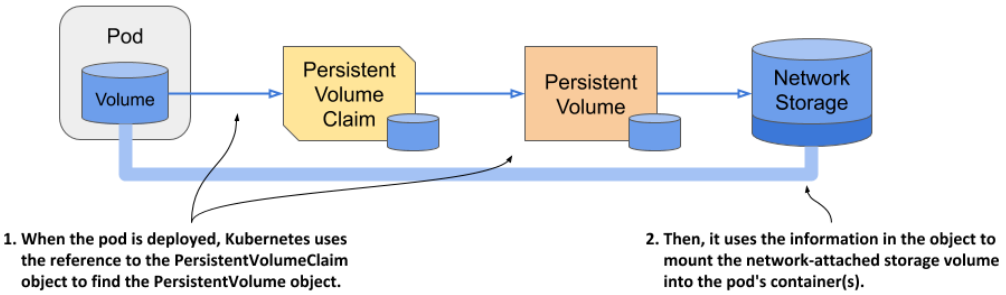

Figure 8.3 Mounting a persistent volume into the pod's container(s)

USING A CLAIM IN MULTIPLE PODS

Multiple pods can use the same storage volume if they refer to the same persistent volume claim and therefore transitively to the same persistent volume, as shown in the following figure.

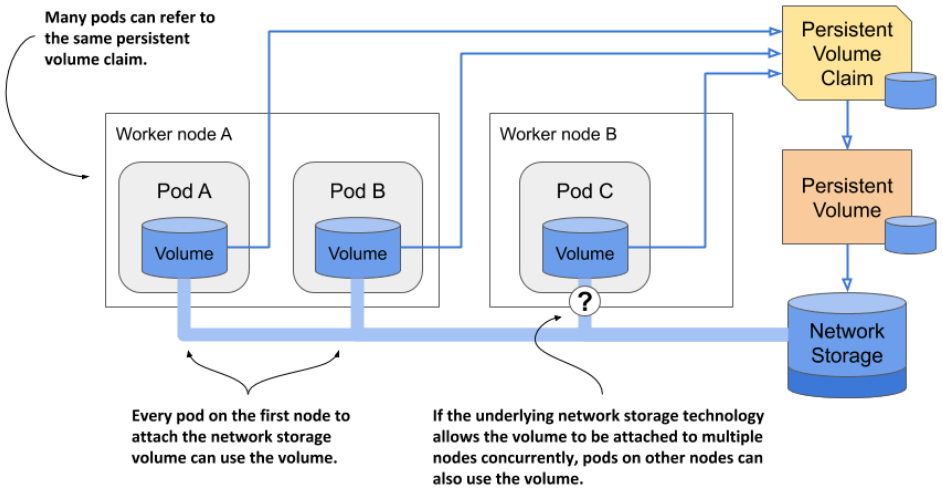

Figure 8.4 Using the same persistent volume claim in multiple pods

Whether these pods must all run on the same cluster node or can access the underlying storage from different nodes depends on the technology that provides that storage. If the underlying storage technology supports attaching the storage to many nodes concurrently, it can be used by pods on different nodes. If not, the pods must all be scheduled to the node that attached the storage volume first.

8.1.2 Understanding the benefits of using persistent volumes and claims

A system where you must use two additional objects to let a pod use a storage volume is more complex than the simple approach explained in the previous chapter, where the pod simply referred to the storage volume directly. Why is this new approach better?

The biggest advantage of using persistent volumes and claims is that the infrastructure-specific details are now decoupled from the application represented by the pod. Cluster administrators, who know the data center better than anyone else, can create the PersistentVolume objects with all their infrastructure-related low-level details, while software developers focus solely on describing the applications and their needs via the Pod and PersistentVolumeClaim objects.

The following figure shows how the two user roles and the objects they create fit together.

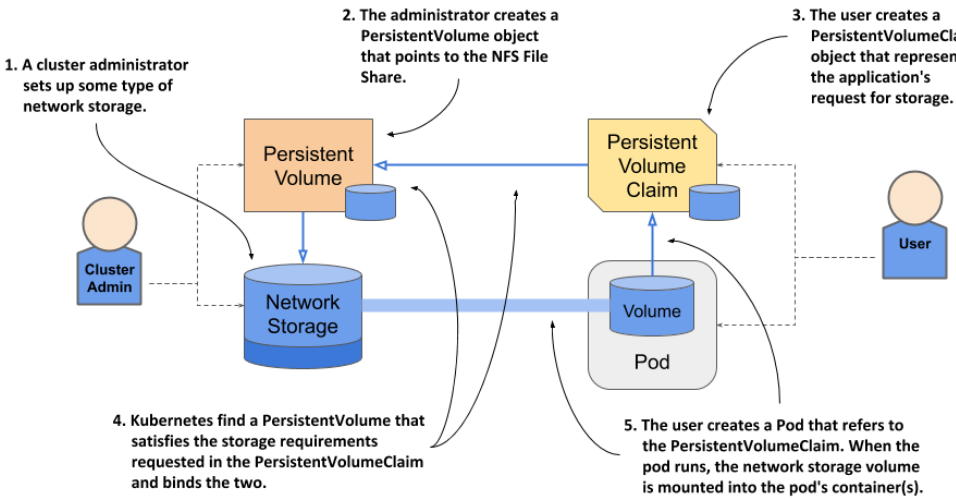

Figure 8.5 Persistent volumes are provisioned by cluster admins and consumed by pods through persistent volume claims.

Instead of the developer adding a technology-specific volume to their pod, the cluster administrator sets up the underlying storage and then registers it in Kubernetes by creating a PersistentVolume object through the Kubernetes API.

When a cluster user needs persistent storage in one of their pods, they first create a PersistentVolumeClaim object in which they either refer to a specific persistent volume by name, or specify the minimum volume size and access mode required by the application, and

let Kubernetes find a persistent volume that meets these requirements. In both cases, the persistent volume is then bound to the claim and is given exclusive access. The claim can then be referenced in a volume definition within one or more pods. When the pod runs, the storage volume configured in the PersistentVolume object is attached to the worker node and mounted into the pod's containers.

It's important to understand that the application developer can create the manifests for the Pod and the PersistentVolumeClaim objects without knowing anything about the infrastructure on which the application will run. Similarly, the cluster administrator can provision a set of storage volumes of varying sizes in advance without knowing much about the applications that will use them.

Furthermore, by using dynamic provisioning of persistent volumes, as discussed later in this chapter, administrators don't need to pre-provision volumes at all. If an automated volume provisioner is installed in the cluster, the physical storage volume and the PersistentVolume object are created on demand for each PersistentVolumeClaim object that users create.

8.2 Creating persistent volumes and claims

Now that you have a basic understanding of persistent volumes and claims and their relationship to the pods, let's revisit the quiz pod from the previous chapter. You may remember that this pod contains a `gcePersistentDisk` volume. You'll modify that pod's manifest to make it use the GCE Persistent Disk via a PersistentVolume object.

As explained earlier, there are usually two different types of Kubernetes users involved in the provisioning and use of persistent volumes. In the following exercises, you will first take on the role of the cluster administrator and create some persistent volumes. One of them will point to the existing GCE Persistent Disk. Then you'll take on the role of a regular user to create a persistent volume claim to get ownership of that volume and use it in the quiz pod.

8.2.1 Creating a PersistentVolume object

Imagine being the cluster administrator. The development team has asked you to provide two persistent volumes for their applications. One will be used to store the data files used by MongoDB in the quiz pod, and the other will be used for something else.

If you use Google Kubernetes Engine to run these examples, you'll create persistent volumes that point to GCE Persistent Disks (GCE PD). For the quiz data files, you can use the GCE PD that you provisioned in the previous chapter.

> **NOTE** If you use a different cloud provider, consult the provider's documentation to learn how to create the physical volume in their environment. If you use Minikube, kind, or any other type of cluster, you don't need to create volumes because you'll use a persistent volume that refers to a local directory on the worker node.

CREATING A PERSISTENT VOLUME WITH GCE PERSISTENT DISK AS THE UNDERLYING STORAGE

If you don't have the `quiz-data` GCE Persistent Disk set up from the previous chapter, create it again using the `gcloud compute disks create quiz-data` command. After the disk is created, you must create a manifest file for the PersistentVolume object, as shown in the following listing. You'll find the file in Chapter09/pv.quiz-data.gcepd.yaml.

Listing 8.1 A persistent volume manifest referring to a GCE Persistent Disk

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: quiz-data    #A
spec:
  capacity:    #B
    storage: 1Gi    #B
  accessModes:    #C
  - ReadWriteOnce    #C
  - ReadOnlyMany    #C
  gcePersistentDisk:    #D
    pdName: quiz-data    #D
    fsType: ext4    #D
```

#A The name of this persistent volume
#B The storage capacity of this volume
#C Whether a single node or many nodes can access this volume in read/write or read-only mode.
#D This persistent volume uses the GCE Persistent Disk created in the previous chapter

The `spec` section in a PersistentVolume object specifies the storage capacity of the volume, the access modes it supports, and the underlying storage technology it uses, along with all the information required to use the underlying storage. In the case of GCE Persistent Disks, this includes the name of the PD resource in Google Compute Engine, the filesystem type, the name of the partition in the volume, and more.

Now create another GCE Persistent Disk named `other-data` and an accompanying PersistentVolume object. Create a new file from the manifest in listing 8.1 and make the necessary changes. You'll find the resulting manifest in the file Chapter09/pv.other-data.gcepd.yaml.

CREATING PERSISTENT VOLUMES BACKED BY OTHER STORAGE TECHNOLOGIES

If your Kubernetes cluster runs on a different cloud provider, you should be able to easily change the persistent volume manifest to use something other than a GCE Persistent Disk, as you did in the previous chapter when you directly referenced the volume within the pod manifest.

If you used Minikube or the kind tool to provision your cluster, you can create a persistent volume that uses a local directory on the worker node instead of network storage by using the `hostPath` field in the PersistentVolume manifest. The manifest for the `quiz-data` persistent volume is shown in the next listing (Chapter09/pv.quiz-data.hostpath.yaml). The manifest for the `other-data` persistent volume is in Chapter09/pv.other-data.hostpath.yaml.

Listing 8.2 A persistent volume using a local directory: quiz-data.hostpath.pv.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: quiz-data
spec:
  capacity:
    storage: 1Gi
  accessModes:
  - ReadWriteOnce
  - ReadOnlyMany
  hostPath:    #A
    path: /var/quiz-data    #A
```

#A Instead of a GCE Persistent Disk, this persistent volume refers to a local directory on the host node

You'll notice that the two persistent volume manifests in this and the previous listing differ only in the part that specifies which underlying storage method to use. The hostPath-backed persistent volume stores data in the `/var/quiz-data` directory in the worker node's filesystem.

> **NOTE** To list all other supported technologies that you can use in a persistent volume, run `kubectl explain pv.spec`. You can then drill further down to see the individual configuration options for each technology. For example, for GCE Persistent Disks, run `kubectl explain pv.spec.gcePersistentDisk`.

I will not bore you with the details of how to configure the persistent volume for each available storage technology, but I do need to explain the `capacity` and `accessModes` fields that you must set in each persistent volume.

SPECIFYING THE VOLUME CAPACITY

The `capacity` of the volume indicates the size of the underlying volume. Each persistent volume must specify its capacity so that Kubernetes can determine whether a particular persistent volume can meet the requirements specified in the persistent volume claim before it can bind them.

SPECIFYING VOLUME ACCESS MODES

Each persistent volume must specify a list of `accessModes` it supports. Depending on the underlying technology, a persistent volume may or may not be mounted by multiple worker nodes simultaneously in read/write or read-only mode. Kubernetes inspects the persistent volume's access modes to determine if it meets the requirements of the claim.

> **NOTE** The access mode determines how many *nodes*, not pods, can attach the volume at a time. Even if a volume can only be attached to a single node, it can be mounted in many pods if they all run on that single node.

Three access modes exist. They are explained in the following table along with their abbreviated form displayed by `kubectl`.

Table 8.1 Persistent volume access modes

Access Mode	Abbr.	Description
`ReadWriteOnce`	`RWO`	The volume can be mounted by a single worker node in read/write mode. While it's mounted to the node, other nodes can't mount the volume.
`ReadOnlyMany`	`ROX`	The volume can be mounted on multiple worker nodes simultaneously in read-only mode.
`ReadWriteMany`	`RWX`	The volume can be mounted in read/write mode on multiple worker nodes at the same time.

NOTE The `ReadOnlyOnce` option doesn't exist. If you use a `ReadWriteOnce` volume in a pod that doesn't need to write to it, you can mount the volume in read-only mode.

USING PERSISTENT VOLUMES AS BLOCK DEVICES

A typical application uses persistent volumes with a formatted filesystem. However, a persistent volume can also be configured so that the application can directly access the underlying block device without using a filesystem. This is configured on the PersistentVolume object using the `spec.volumeMode` field. The supported values for the field are explained in the next table.

Table 8.2 Configuring the volume mode for the persistent volume

Volume Mode	Description
`Filesystem`	When the persistent volume is mounted in a container, it is mounted to a directory in the file tree of the container. If the underlying storage is an unformatted block device, Kubernetes formats the device using the filesystem specified in the volume definition (for example, in the field `gcePersistentDisk.fsType`) before it is mounted in the container. This is the default volume mode.
`Block`	When a pod uses a persistent volume with this mode, the volume is made available to the application in the container as a raw block device (without a filesystem). This allows the application to read and write data without any filesystem overhead. This mode is typically used by special types of applications, such as database systems.

The manifests for the `quiz-data` and `other-data` persistent volumes do not specify a `volumeMode` field, which means that the default mode is used, namely `Filesystem`.

CREATING AND INSPECTING THE PERSISTENT VOLUME

You can now create the PersistentVolume objects by posting the manifests to the Kubernetes API using the now well-known command `kubectl apply`. Then use the `kubectl get` command to list the persistent volumes in your cluster:

```
$ kubectl get pv
NAME         CAPACITY    ACCESS MODES   ...   STATUS      CLAIM    ...   AGE
```

```
other-data   10Gi        RWO,ROX        ...   Available           ...   3m
quiz-data    10Gi        RWO,ROX        ...   Available           ...   3m
```

TIP Use `pv` as the shorthand for PersistentVolume.

The `STATUS` column indicates that both persistent volumes are `Available`. This is expected because they aren't yet bound to any persistent volume claim, as indicated by the empty `CLAIM` column. Also displayed are the volume capacity and access modes, which are shown in abbreviated form, as explained in table 8.1.

The underlying storage technology used by the persistent volume isn't displayed by the `kubectl get pv` command because it's less important. What is important is that each persistent volume represents a certain amount of storage space available in the cluster that applications can access with the specified modes. The technology and the other parameters configured in each persistent volume are implementation details that typically don't interest users who deploy applications. If someone needs to see these details, they can use `kubectl describe` or print the full definition of the PersistentVolume object as in the following command:

```
$ kubectl get pv quiz-data -o yaml
```

8.2.2 Claiming a persistent volume

Your cluster now contains two persistent volumes. Before you can use the `quiz-data` volume in the quiz pod, you need to claim it. This section explains how to do this.

CREATING A PERSISTENTVOLUMECLAIM OBJECT

To claim a persistent volume, you create a PersistentVolumeClaim object in which you specify the requirements that the persistent volume must meet. These include the minimum capacity of the volume and the required access modes, which are usually dictated by the application that will use the volume. For this reason, persistent volume claims should be created by the author of the application and not by cluster administrators, so take off your administrator hat now and put on your developer hat.

TIP As an application developer, you should never include persistent volume definitions in your application manifests. You should include persistent volume claims because they specify the storage requirements of your application.

To create a PersistentVolumeClaim object, create a manifest file with the contents shown in the following listing. You'll also find the file in Chapter08/pvc.quiz-data.static.yaml.

Listing 8.3 A PersistentVolumeClaim object manifest

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: quiz-data    #A
spec:
  resources:
    requests:    #B
      storage: 1Gi    #B
  accessModes:    #C
  - ReadWriteOnce    #C
  storageClassName: ""    #D
  volumeName: quiz-data    #E
```

#A The name of this claim. The pod will refer to this claim using by this name.
#B The volume must provide at least 1 GiB of storage space.
#C The volume must support mounting by a single node for both reading and writing.
#D This must be set to an empty string to disable dynamic provisioning.
#E You want to claim the quiz-data persistent volume.

The persistent volume claim defined in the listing requests that the volume is at least `1GiB` in size and can be mounted on a single node in read/write mode. The field `storageClassName` is used for dynamic provisioning of persistent volumes, which you'll learn about later in the chapter. The field must be set to an empty string if you want Kubernetes to bind a pre-provisioned persistent volume to this claim instead of provisioning a new one.

In this exercise, you want to claim the `quiz-data` persistent volume, so you must indicate this with the `volumeName` field. In your cluster, two matching persistent volumes exist. If you don't specify this field, Kubernetes could bind your claim to the `other-data` persistent volume.

If the cluster administrator creates a bunch of persistent volumes with non-descript names, and you don't care which one you get, you can skip the `volumeName` field. In that case, Kubernetes will randomly choose one of the persistent volumes whose capacity and access modes match the claim.

> **NOTE** **Like persistent volumes, claims can also specify the required `volumeMode`. As you learned in section 8.2.1, this can be either `Filesystem` or `Block`. If left unspecified, it defaults to `Filesystem`. When Kubernetes checks whether a volume can satisfy the claim, the `volumeMode` of the claim and the volume is also considered.**

To create the PersistentVolumeClaim object, apply its manifest file with `kubectl apply`. After the object is created, Kubernetes soon binds a volume to the claim. If the claim requests a specific persistent volume by name, that's the volume that is bound, if it also matches the other requirements. Your claim requires 1GiB of disk space and the `ReadWriteOnce` access mode. The persistent volume `quiz-data` that you created earlier meets both requirements and this allows it to be bound to the claim.

LISTING PERSISTENT VOLUME CLAIMS

If all goes well, your claim should now be bound to the `quiz-data` persistent volume. Use the `kubectl get` command to see if this is the case:

```
$ kubectl get pvc
NAME        STATUS   VOLUME      CAPACITY   ACCESS MODES   STORAGECLASS   AGE
quiz-data   Bound    quiz-data   10Gi       RWO,ROX                       2m    #A
```

#A The claim is bound to the quiz-data persistent volume

> **TIP** Use `pvc` as a shorthand for `persistentvolumeclaim`.

The output of the `kubectl` command shows that the claim is now bound to your persistent volume. It also shows the capacity and access modes of this volume. Even though the claim requested only 1GiB, it has 10GiB of storage space available, because that's the capacity of the volume. Similarly, although the claim requested only the `ReadWriteOnce` access mode, it is bound to a volume that supports both the `ReadWriteOnce` (`RWO`) and the `ReadOnlyMany` (`ROX`) access modes.

If you put your cluster admin hat back on for a moment and list the persistent volumes in your cluster, you'll see that it too is now displayed as `Bound`:

```
$ kubectl get pv
NAME        CAPACITY   ACCESS MODES   ...   STATUS   CLAIM                 ...
quiz-data   10Gi       RWO,ROX        ...   Bound    default/quiz-data     ...
```

Any cluster admin can see which claim each persistent volume is bound to. In your case, the volume is bound to the claim `default/quiz-data`.

> **NOTE** You may wonder what the word `default` means in the claim name. This is the *namespace* in which the PersistentVolumeClaim object is located. Namespaces allow objects to be organized into disjoint sets. You'll learn about them in chapter 10.

By claiming the persistent volume, you and your pods now have the exclusive right to use the volume. No one else can claim it until you release it by deleting the PersistentVolumeClaim object.

8.2.3 Using a claim and volume in a single pod

In this section, you'll learn the ins and outs of using a persistent volume in a single pod at a time.

USING A PERSISTENT VOLUME IN POD

To use a persistent volume in a pod, you define a volume within the pod in which you refer to the PersistentVolumeClaim object. To try this, modify the quiz pod from the previous chapter and make it use the `quiz-data` claim. The changes to the pod manifest are highlighted in the next listing. You'll find the file in Chapter09/pod.quiz.pvc.yaml.

Listing 8.4 A pod using a persistentVolumeClaim volume

```
apiVersion: v1
kind: Pod
metadata:
  name: quiz
spec:
  volumes:
  - name: quiz-data
    persistentVolumeClaim:    #A
      claimName: quiz-data    #A
  containers:
  - name: quiz-api
    image: luksa/quiz-api:0.1
    ports:
    - name: http
      containerPort: 8080
  - name: mongo
    image: mongo
    volumeMounts:    #B
    - name: quiz-data    #B
      mountPath: /data/db    #B
```

#A The volume refers to a persistent volume claim with the name quiz-data.
#B The volume is mounted the same way that other volumes types are mounted.

As you can see in the listing, you don't define the volume as a `gcePersistentDisk`, `awsElasticBlockStore`, `nfs` or `hostPath` volume, but as a `persistentVolumeClaim` volume. The pod will use whatever persistent volume is bound to the `quiz-data` claim. In your case, that should be the `quiz-data` persistent volume.

Create and test this pod now. Before the pod starts, the GCE PD volume is attached to the node and mounted into the pod's container(s). If you use GKE and have configured the persistent volume to use the GCE Persistent Disk from the previous chapter, which already contains data, you should be able to retrieve the quiz questions you stored earlier by running the following command:

```
$ kubectl exec -it quiz -c mongo -- mongo kiada --quiet --eval "db.questions.find()"
{ "_id" : ObjectId("5fc3a4890bc9170520b22452"), "id" : 1, "text" : "What does k8s mean?",
"answers" : [ "Kates", "Kubernetes", "Kooba Dooba Doo!" ], "correctAnswerIndex" : 1 }
```

If your GCE PD has no data, add it now by running the shell script Chapter08/insert-question.sh.

RE-USING THE CLAIM IN A NEW POD INSTANCE

When you delete a pod that uses a persistent volume via a persistent volume claim, the underlying storage volume is detached from the worker node (assuming that it was the only pod that was using it on that node). The persistent volume object remains bound to the claim. If you create another pod that refers to this claim, this new pod gets access to the volume and its files.

Try deleting the quiz pod and recreating it. If you run the `db.questions.find()` query in this new pod instance, you'll see that it returns the same data as the previous one. If the persistent volume uses network-attached storage such as GCE Persistent Disks, the pod sees

the same data regardless of what node it's scheduled to. If you use a kind-provisioned cluster and had to resort to using a hostPath-based persistent volume, this isn't the case. To access the same data, you must ensure that the new pod instance is scheduled to the node to which the original instance was scheduled, as the data is stored in that node's filesystem.

RELEASING A PERSISTENT VOLUME

When you no longer plan to deploy pods that will use this claim, you can delete it. This releases the persistent volume. You might wonder if you can then recreate the claim and access the same volume and data. Let's find out. Delete the pod and the claim as follows to see what happens:

```
$ kubectl delete pod quiz
pod "quiz" deleted

$ kubectl delete pvc quiz-data
persistentvolumeclaim "quiz-data" deleted
```

Now check the status of the persistent volume:

```
$ kubectl get pv quiz-data
NAME        ...   RECLAIM POLICY   STATUS     CLAIM               ...
quiz-data   ...   Retain           Released   default/quiz-data   ...
```

The `STATUS` column shows the volume as `Released` rather than `Available`, as was the case initially. The `CLAIM` column still shows the `quiz-data` claim to which it was previously bound, even if the claim no longer exists. You'll understand why in a minute.

BINDING TO A RELEASED PERSISTENT VOLUME

What happens if you create the claim again? Is the persistent volume bound to the claim so that it can be reused in a pod? Run the following commands to see if this is the case.

```
$ kubectl apply -f pvc.quiz-data.static.yaml
persistentvolumeclaim/quiz-data created

$ kubectl get pvc
NAME        STATUS    VOLUME    CAPACITY    ACCESSMODES    STORAGECLASS    AGE
quiz-data   Pending                                                        13s    #A
```

#A The claim's status is Pending.

The claim isn't bound to the volume and its status is `Pending`. When you created the claim earlier, it was immediately bound to the persistent volume, so why not now?

The reason behind this is that the volume has already been used and might contain data that should be erased before another user claims the volume. This is also the reason why the status of the volume is `Released` instead of `Available` and why the claim name is still shown on the persistent volume, as this helps the cluster administrator to know if the data can be safely deleted.

MAKING A RELEASED PERSISTENT VOLUME AVAILABLE FOR RE-USE

To make the volume available again, you must delete and recreate the PersistentVolume object. But will this cause the data stored in the volume to be lost?

Imagine if you had accidentally deleted the pod and the claim and caused a loss of service to the Kiada application. You need to restore the service as soon as possible, with all data intact. If you think that deleting the PersistentVolume object would delete the data, that sounds like the last thing you should do but is actually completely safe.

With a pre-provisioned persistent volume like the one at hand, deleting the object is equivalent to deleting a data pointer. The PersistentVolume object merely *points* to a GCE Persistent Disk. It doesn't store the data. If you delete and recreate the object, you end up with a new pointer to the same GCE PD and thus the same data. You'll confirm this is the case in the next exercise.

```
$ kubectl delete pv quiz-data
persistentvolume "quiz-data" deleted

$ kubectl apply -f pv.quiz-data.gcepd.yaml
persistentvolume/quiz-data created

$ kubectl get pv quiz-data
NAME        ...   RECLAIM POLICY   STATUS      CLAIM   ...
quiz-data   ...   Retain           Available           ...
```

NOTE An alternative way of making a persistent volume available again is to edit the PersistentVolume object and remove the `claimRef` from the `spec` section.

The persistent volume is displayed as `Available` again. Let me remind you that you created a claim for the volume earlier. Kubernetes has been waiting for a volume to bind to the claim. As you might expect, the volume you've just created will be bound to this claim in a few seconds. List the volumes again to confirm:

```
$ kubectl get pv quiz-data
NAME        ...   RECLAIM POLICY   STATUS      CLAIM                ...
quiz-data   ...   Retain           Bound       default/quiz-data    ...    #A
```

#A The persistent volume is again bound to the claim.

The output shows that the persistent volume is again bound to the claim. If you now deploy the quiz pod and query the database again with the following command, you'll see that the data in underlying GCE Persistent Disk has not been lost:

```
$ kubectl exec -it quiz -c mongo -- mongo kiada --quiet --eval "db.questions.find()"
{ "_id" : ObjectId("5fc3a4890bc9170520b22452"), "id" : 1, "text" : "What does k8s mean?",
"answers" : [ "Kates", "Kubernetes", "Kooba Dooba Doo!" ], "correctAnswerIndex" : 1 }
```

CONFIGURING THE RECLAIM POLICY ON PERSISTENT VOLUMES

What happens to a persistent volume when it is released is determined by the volume's reclaim policy. When you used the `kubectl get pv` command to list persistent volumes, you may have noticed that the `quiz-data` volume's policy is `Retain`. This policy is configured using the field `.spec.persistentVolumeReclaimPolicy` in the PersistentVolume object.

The field can have one of the three values explained in the following table.

Table 8.3 Persistent volume reclaim policies

Reclaim policy	Description
`Retain`	When the persistent volume is released (this happens when you delete the claim that's bound to it), Kubernetes *retains* the volume. The cluster administrator must manually reclaim the volume. This is the default policy for manually created persistent volumes.
`Delete`	The PersistentVolume object and the underlying storage are automatically deleted upon release. This is the default policy for dynamically provisioned persistent volumes, which are discussed in the next section.
`Recycle`	This option is deprecated and shouldn't be used as it may not be supported by the underlying volume plugin. This policy typically causes all files on the volume to be deleted and makes the persistent volume available again without the need to delete and recreate it.

TIP You can change the reclaim policy of an existing PersistentVolume at any time. If it's initially set to `Delete`, but you don't want to lose your data when deleting the claim, change the volume's policy to `Retain` before doing so.

WARNING If a persistent volume is `Released` and you subsequently change its reclaim policy from `Retain` to `Delete`, the PersistentVolume object and the underlying storage will be deleted immediately. However, if you instead delete the object manually, the underlying storage remains intact.

DELETING A PERSISTENT VOLUME WHILE IT'S BOUND

You're done playing with the `quiz` pod, the `quiz-data` persistent volume claim, and the `quiz-data` persistent volume, so you'll now delete them. You'll learn one more thing in the process.

Have you wondered what happens if a cluster administrator deletes a persistent volume while it's in use (while it's bound to a claim)? Let's find out. Delete the persistent volume like so:

```
$ kubectl delete pv quiz-data
persistentvolume "quiz-data" deleted    #A
```

#A The command blocks after printing this message

This command tells the Kubernetes API to delete the PersistentVolume object and then waits for Kubernetes controllers to complete the process. But this can't happen until you release the persistent volume from the claim by deleting the PersistentVolumeClaim object.

You can cancel the wait by pressing Control-C. However, this doesn't cancel the deletion, as its already underway. You can confirm this as follows:

```
$ kubectl get pv quiz-data
NAME        CAPACITY   ACCESS MODES   STATUS        CLAIM               ...
quiz-data   10Gi       RWO,ROX        Terminating   default/quiz-data   ...    #A
```

#A The persistent volume is terminating

As you can see, the persistent volume's status shows that it's being terminated. But it's still bound to the persistent volume claim. You need to delete the claim for the volume deletion to complete.

DELETING A PERSISTENT VOLUME CLAIM WHILE A POD IS USING IT

The claim is still being used by the quiz pod, but let's try deleting it anyway:

```
$ kubectl delete pvc quiz-data
persistentvolumeclaim "quiz-data" deleted    #A
```

#A The command blocks after printing this message

Like the `kubectl delete pv` command, this command also doesn't complete immediately. As before, the command waits for the claim deletion to complete. You can interrupt the execution of the command, but this won't cancel the deletion, as you can see with the following command:

```
$ kubectl get pvc quiz-data
NAME        STATUS        VOLUME      CAPACITY   ACCESS MODES   STORAGECLASS   AGE
quiz-data   Terminating   quiz-data   10Gi       RWO,ROX                       15m    #A
```

#A The persistent volume claim is being terminated

The deletion of the claim is blocked by the pod. Unsurprisingly, deleting a persistent volume or a persistent volume claim has no immediate effect on the pod that's using it. The application running in the pod continues to run unaffected. Kubernetes never kills pods just because the cluster administrator wants their disk space back.

To allow the termination of the persistent volume claim and the persistent volume to complete, delete the quiz pod with `kubectl delete po quiz`.

DELETING THE UNDERLYING STORAGE

As you learned in the previous section, deleting the persistent volume does not delete the underlying storage, such as the `quiz-data` GCE Persistent Disk if you use Google Kubernetes Engine to perform these exercises, or the `/var/quiz-data` directory on the worker node if you use Minikube or kind.

You no longer need the data files and can safely delete them. If you use Minikube or kind, you don't need to delete the data directory, as it doesn't cost you anything. However, a GCE Persistent Disk does. You can delete it with the following command:

```
$ gcloud compute disks delete quiz-data
```

You might remember that you also created another GCE Persistent Disk called `other-data`. Don't delete that one just yet. You'll use it in the next section's exercise.

8.2.4 Using a claim and volume in multiple pods

So far, you used a persistent volume in only one pod instance at a time. You used the persistent volume in the so-called ReadWriteOnce `(RWO)` access mode because it was attached to a single node and allowed both read and write operations. You may remember that two other modes exist, namely ReadWriteMany (`RWX`) and ReadOnlyMany (`ROX`). The volume's access modes

indicate whether it can concurrently be attached to one or many cluster nodes and whether it can only be read from or also written to.

The ReadWriteOnce mode doesn't mean that only a single pod can use it, but that a single *node* can attach the volume. As this is something that confuses a lot of users, it warrants a closer look.

BINDING A CLAIM TO A RANDOMLY SELECTED PERSISTENT VOLUME

This exercise requires the use of a GKE cluster. Make sure it has at least two nodes. First, create a persistent volume claim for the `other-data` persistent volume that you created earlier. You'll find the manifest in the file Chapter08/pvc.other-data.yaml. It's shown in the following listing.

Listing 8.5 A persistent volume claim requesting both ReadWriteOnce and ReadOnlyMany access

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: other-data
spec:
  resources:
    requests:
      storage: 1Gi
  accessModes:    #A
  - ReadWriteOnce    #A
  - ReadOnlyMany    #A
  storageClassName: ""    #B
```

#A This claim requires the volume to support both access modes
#B The storage class name is empty to force the claim to be bound to an existing persistent volume

You'll notice that unlike in the previous section, this persistent volume claim does not specify the `volumeName`. This means that the persistent volume for this claim will be selected at random among all the volumes that can provide at least 1Gi of space and support both the `ReadWriteOnce` and the `ReadOnlyMany` access modes.

Your cluster should currently contain only the `other-data` persistent volume. Because it matches the requirements in the claim, this is the volume that will be bound to it.

USING A READWRITEONCE VOLUME IN MULTIPLE PODS

The persistent volume bound to the claim supports both ReadWriteOnce and ReadOnlyMany access modes. First, you'll use it in ReadWriteOnce mode, as you'll deploy pods that write to it.

You'll create several replicas of a data-writer pod from a single pod manifest. The manifest is shown in the following listing. You'll find it in Chapter08/pod.data-writer.yaml.

Listing 8.6 A pod that writes a file to a shared persistent volume

```
apiVersion: v1
kind: Pod
metadata:
  generateName: data-writer-    #A
spec:
  volumes:
  - name: other-data
    persistentVolumeClaim:    #B
      claimName: other-data    #B
  containers:
  - name: writer
    image: busybox
    command:
    - sh
    - -c
    - |
      echo "A writer pod wrote this." > /other-data/${HOSTNAME} &&    #C
      echo "I can write to /other-data/${HOSTNAME}." ;    #C
      sleep 9999    #C
    volumeMounts:
    - name: other-data
      mountPath: /other-data
    resources:    #D
      requests:    #D
        cpu: 1m    #D
```

#A This pod manifest doesn't set a name for the pod. The generateName field allows a random name with this prefix to be generated for each pod you create from this manifest.
#B All pods created from this manifest will use the other-data persistent volume claim.
#C The pod writes a short message to a file in the persistent volume. The filename is the pod's hostname. If the file creation succeeds, a message is printed to the standard output of the container. The container then waits for 9999 seconds.
#D Ignore these lines. You'll learn about them in chapter 20.

Use the following command to create the pod from this manifest:

```
$ kubectl create -f pod.data-writer.yaml    #A
pod/data-writer-6mbjg created    #B
```

#A The command kubectl create is used instead of kubectl apply
#B The pod gets a randomly generated name

Notice that you aren't using the `kubectl apply` this time. Because the pod manifest uses the `generateName` field instead of specifying the pod name, `kubectl apply` won't work. You must use `kubectl create`, which is similar, but is only used to create and not update objects.

Repeat the command several times so that you create two to three times as many writer pods as there are cluster nodes to ensure that at least two pods are scheduled to each node. Confirm that this is the case by listing the pods with the `-o wide` option and inspecting the `NODE` column:

```
$ kubectl get pods -o wide
NAME                READY   STATUS              RESTARTS   AGE   IP           NODE
data-writer-6mbjg   1/1     Running             0          5m    10.0.10.21   gkdp-r6j4
       #A
data-writer-97t9j   0/1     ContainerCreating   0          5m    <none>       gkdp-mcbg
       #B
data-writer-d9f2f   1/1     Running             0          5m    10.0.10.23   gkdp-r6j4
       #A
data-writer-dfd8h   0/1     ContainerCreating   0          5m    <none>       gkdp-mcbg
       #B
data-writer-f867j   1/1     Running             0          5m    10.0.10.17   gkdp-r6j4
       #A
```

#A These pods run on the first node.
#B These pods are scheduled to the second node, but don't run.

> **NOTE** I've shortened the node names for clarity.

If all your pods are located on the same node, create a few more. Then look at the `STATUS` of these pods. You'll notice that all the pods scheduled to the first node run fine, whereas the pods on the other node are all stuck in the status `ContainerCreating`. Even waiting for several minutes doesn't change anything. Those pods will never run.

If you use `kubectl describe` to display the events related to one of these pods, you'll see that it doesn't run because the persistent volume can't be attached to the node that the pod is on:

```
$ kubectl describe po data-writer-97t9j
...
  Warning  FailedAttachVolume   ...   attachdetach-controller  AttachVolume.Attach failed
for volume "other-data" : googleapi: Error 400: RESOURCE_IN_USE_BY_ANOTHER_RESOURCE -    #A
The disk resource 'projects/.../disks/other-data' is already being used by    #A
'projects/.../instances/gkdp-r6j4'    #A
```

#A The disk is being used by node gkdp-r6j4

The reason the volume can't be attached is because it's already attached to the first node in read-write mode. The volume supports ReadWriteOnce and ReadOnlyMany but doesn't support ReadWriteMany. This means that only a single node can attach the volume in read-write mode. When the second node tries to do the same, the operation fails.

All the pods on the first node run fine. Check their logs to confirm that they were all able to write a file to the volume. Here's the log of one of them:

```
$ kubectl logs other-data-writer-6mbjg
I can write to /other-data/other-data-writer-6mbjg.
```

You'll find that all the pods on the first node successfully wrote their files to the volume. You don't need ReadWriteMany for multiple pods to write to the volume if they are on the same node. As explained before, the word "Once" in ReadWriteOnce refers to nodes, not pods.

USING A COMBINATION OF READ-WRITE AND READ-ONLY PODS WITH A READWRITEONCE AND READONLYMANY VOLUME

You'll now deploy a group of reader pods alongside the data-writer pods. They will use the persistent volume in read-only mode. The following listing shows the pod manifest for these data-reader pods. You'll find it in Chapter08/pod.data-reader.yaml.

Listing 8.7 A pod that mounts a shared persistent volume in read-only mode

```
apiVersion: v1
kind: Pod
metadata:
  generateName: data-reader-
spec:
  volumes:
  - name: other-data
    persistentVolumeClaim:
      claimName: other-data    #A
      readOnly: true    #B
  containers:
  - name: reader
    image: busybox
    imagePullPolicy: Always
    command:
    - sh
    - -c
    - |
      echo "The files in the persistent volume and their contents:" ;    #C
      grep ^ /other-data/* ;    #C
      sleep 9999    #C
    volumeMounts:
    - name: other-data
      mountPath: /other-data
    ...
```

#A This pod also uses the other-data persistent volume claim.
#B Unlike the writer pod, this pod uses the persistent volume in read-only mode.
#C When it runs, it prints the name and contents of each file stored in the persistent volume.

Use the `kubectl create` command to create as many of these reader pods as necessary to ensure that each node runs at least two instances. Use the `kubectl get po -o wide` command to see how many pods are on each node.

As before, you'll notice that only those reader pods that are scheduled to the first node are running. The pods on the second node are stuck in `ContainerCreating`, just like the writer pods. Here's a list of just the reader pods (the writer pods are still there, but aren't shown):

```
$ kubectl get pods -o wide | grep reader
NAME                READY   STATUS              RESTARTS   AGE   IP           NODE
data-reader-6594s   1/1     Running             0          2m    10.0.10.25   gkdp-r6j4
       #A
data-reader-lqwkv   1/1     Running             0          2m    10.0.10.24   gkdp-r6j4
       #A
data-reader-mr5mk   0/1     ContainerCreating   0          2m    <none>       gkdp-mcbg
       #B
data-reader-npk24   1/1     Running             0          2m    10.0.10.27   gkdp-r6j4
       #A
data-reader-qbpt5   0/1     ContainerCreating   0          2m    <none>       gkdp-mcbg
       #B
```

#A These run on the first node
#B These are scheduled to the second node, but don't run

These pods use the volume in read-only mode. The claim's (and volume's) access modes are both ReadWriteOnce (`RWO`) and ReadOnlyMany (`ROX`), as you can see by running `kubectl get pvc`:

```
$ kubectl get pvc other-data
NAME         STATUS   VOLUME       CAPACITY   ACCESS MODES   STORAGECLASS   AGE
other-data   Bound    other-data   10Gi       RWO,ROX                       23h
```

If the claim supports access mode ReadOnlyMany, why can't both nodes attach the volume and run the reader pods? This is caused by the writer pods. The first node attached the persistent volume in read-write mode. This prevents other nodes from attaching the volume, even in read-only mode.

Wonder what happens if you delete all the writer pods? Does that allow the second node to attach the volume in read-only mode and run its pods? Delete the writer pods one by one or use the following command to delete them all if you use a shell that supports the following syntax:

```
$ kubectl delete $(kubectl get po -o name | grep writer)
```

Now list the pods again. The status of the reader pods that are on the second node is still ContainerCreating. Even if you give it enough time, the pods on that node never run. Can you figure out why that is so?

It's because the volume is still being used by the reader pods on the first node. The volume is attached in read-write mode because that was the mode requested by the writer pods, which you deployed first. Kubernetes can't detach the volume or change the mode in which it is attached while it's being used by pods.

In the next section, you'll see what happens if you deploy reader pods without first deploying the writers. Before moving on, delete all the pods as follows:

```
$ kubectl delete po --all
```

Give Kubernetes some time to detach the volume from the node. Then go to the next exercise.

USING A READONLYMANY VOLUME IN MULTIPLE PODS

Create several reader pods again by repeating the `kubectl create -f pod.data-reader.yaml` command several times. This time, all the pods run, even if they are on different nodes:

```
$ kubectl get pods -o wide
NAME                READY   STATUS    RESTARTS   AGE   IP           NODE
data-reader-9xs5q   1/1     Running   0          27s   10.0.10.34   gkdp-r6j4
data-reader-b9b25   1/1     Running   0          29s   10.0.10.32   gkdp-r6j4
data-reader-cbnp2   1/1     Running   0          16s   10.0.9.12    gkdp-mcbg
data-reader-fjx6t   1/1     Running   0          21s   10.0.9.11    gkdp-mcbg
```

All these pods specify the `readOnly: true` field in the `persistentVolumeClaim` volume definition. This causes the node that runs the first pod to attach the persistent volume in read-only mode. The same thing happens on the second node. They can both attach the volume because they both attach it in read-only mode and the persistent volume supports ReadOnlyMany.

The ReadOnlyMany access mode doesn't need further explanation. If no pod mounts the volume in read-write mode, any number of pods can use the volume, even on many different nodes.

Can you guess what happens if you deploy a writer pod now? Can it write to the volume? Create the pod and check its status. This is what you'll see:

```
$ kubectl get po -o wide
NAME                READY   STATUS    RESTARTS   AGE     IP           NODE
...
data-writer-dj6w5   1/1     Running   0          3m33s   10.0.10.38   gkdp-r6j4
```

This pod is shown as `Running`. Does that surprise you? It did surprise me. I thought it would be stuck in `ContainerCreating` because the node couldn't mount the volume in read-write mode because it's already mounted in read-only mode. Does that mean that the node was able to upgrade the mount point from read-only to read-write without detaching the volume?

Let's check the pod's log to confirm that it could write to the volume:

```
$ kubectl logs data-writer-dj6w5
sh: can't create /other-data/data-writer-dj6w5: Read-only file system
```

Ahh, there's your answer. The pod is unable to write to the volume because it's read-only. The pod was started even though the volume isn't mounted in read-write mode as the pod requests. This might be a bug. If you try this yourself and the pod doesn't run, you'll know that the bug was fixed after the book was published.

You can now delete all the pods, the persistent volume claim and the underlying GCE Persistent Disk, as you're done using them.

USING A READWRITEMANY VOLUME IN MULTIPLE PODS

GCE Persistent Disks don't support the ReadWriteMany access mode. However, network-attached volumes available in other cloud environments do support it. As the name of the ReadWriteMany access mode indicates, volumes that support this mode can be attached to many cluster nodes concurrently, yet still allow both read and write operations to be performed on the volume.

As this mode has no restrictions on the number of nodes or pods that can use the persistent volume in either read-write or read-only mode, it doesn't need any further explanation. If you'd like to play with them anyhow, I suggest you deploy the writer and the reader pods as in the previous exercise, but this time use the ReadWriteMany access mode in both the persistent volume and the persistent volume claim definitions.

8.2.5 Understanding the lifecycle of manually provisioned persistent volumes

You used the same GCE Persistent Disk throughout several exercises in this chapter, but you created multiple volumes, claims, and pods that used the same GCE PD. To understand the lifecycles of these four objects, take a look at the following figure.

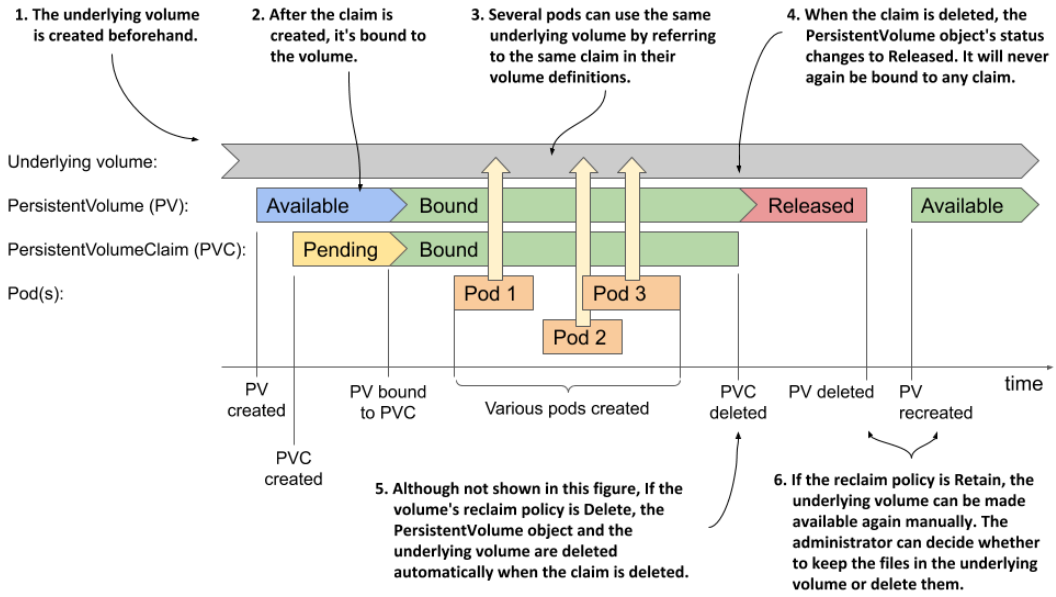

Figure 8.6 The lifecycle of statically provisioned persistent volumes, claims and the pods that use them

When using manually provisioned persistent volumes, the lifecycle of the underlying storage volume is not coupled to the lifecycle of the PersistentVolume object. Each time you create the object, its initial status is `Available`. When a PersistentVolumeClaim object appears, the persistent volume is bound to it, if it meets the requirements set forth in the claim. Until the claim is bound to the volume, it has the status `Pending`; then both the volume and the claim are displayed as `Bound`.

At this point, one or many pods may use the volume by referring to the claim. When each pod runs, the underlying volume is mounted in the pod's containers. After all the pods are finished with the claim, the PersistentVolumeClaim object can be deleted.

When the claim is deleted, the volume's reclaim policy determines what happens to the PersistentVolume object and the underlying volume. If the policy is `Delete`, both the object and the underlying volume are deleted. If it's `Retain`, the PersistentVolume object and the underlying volume are preserved. The object's status changes to `Released` and the object can't be bound until additional steps are taken to make it `Available` again.

If you delete the PersistentVolume object manually, the underlying volume and its files remain intact. They can be accessed again by creating a new PersistentVolume object that references the same underlying volume.

> **NOTE** The sequence of events described in this section applies to the use of statically provisioned volumes that exist before the claims are created. When persistent volumes are dynamically provisioned, as described in the next section, the situation is different. Look for a similar diagram at the end of the next section.

8.3 Dynamic provisioning of persistent volumes

So far in this chapter you've seen how developers can claim pre-provisioned persistent volumes as a place for their pods to store data persistently without having to deal with the details of the underlying storage technology. However, a cluster administrator must pre-provision the physical volumes and create a PersistentVolume object for each of these volumes. Then each time the volume is bound and released, the administrator must manually delete the data on the volume and recreate the object.

To keep the cluster running smoothly, the administrator may need to pre-provision dozens, if not hundreds, of persistent volumes, and constantly keep track of the number of available volumes to ensure the cluster never runs out. All this manual work contradicts the basic idea of Kubernetes, which is to automate the management of large clusters. As one might expect, a better way to manage volumes exists. It's called *dynamic provisioning of persistent volumes*.

With dynamic provisioning, instead of provisioning persistent volumes in advance (and manually), the cluster admin deploys a persistent volume provisioner to automate the just-in-time provisioning process, as shown in the following figure.

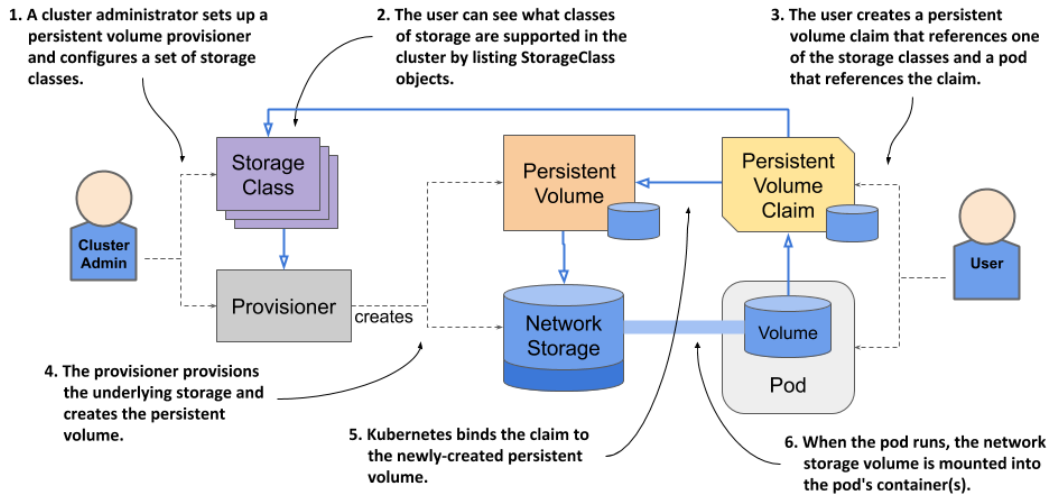

Figure 8.7 Dynamic provisioning of persistent volumes

In contrast to static provisioning, the order in which the claim and the volume arise is reversed. When a user creates a persistent volume claim, the dynamic provisioner provisions the underlying storage and creates the PersistentVolume object for that particular claim. The two objects are then bound.

If your Kubernetes cluster is managed by a cloud provider, it probably already has a persistent volume provisioner configured. If you are running Kubernetes on-premises, you'll need to deploy a custom provisioner, but this is outside the scope of this chapter. Clusters that are provisioned with Minikube or kind usually also come with a provisioner out of the box.

8.3.1 Introducing the StorageClass object

The persistent volume claim definition you created in the previous section specifies the minimum size and the required access modes of the volume, but it also contains a field named `storageClassName`, which wasn't discussed yet.

A Kubernetes cluster can run multiple persistent volume provisioners, and a single provisioner may support several different types of storage volumes. When creating a claim, you use the `storageClassName` field to specify which storage class you want.

LISTING STORAGE CLASSES

The storage classes available in the cluster are represented by *StorageClass* API objects. You can list them with the `kubectl get sc` command. In a GKE cluster, this is the result:

```
$ kubectl get sc
NAME                 PROVISIONER            AGE
standard (default)   kubernetes.io/gce-pd   1d    #A
```

#A The standard storage class in a GKE cluster

> **NOTE** The shorthand for `storageclass` is `sc`.

In a kind-provisioned cluster, the result is similar:

```
$ kubectl get sc
NAME                 PROVISIONER             RECLAIMPOLICY   ...
standard (default)   rancher.io/local-path   Delete          ...    #A
```

#A The standard storage class in a cluster created with the kind tool

Clusters created with Minikube also provide a storage class with the same name:

```
$ kubectl get sc
NAME                 PROVISIONER                RECLAIMPOLICY   VOLUMEBINDINGMODE   ...
standard (default)   k8s.io/minikube-hostpath   Delete          Immediate           ...
```

#A The standard storage class in a cluster created with the kind tool

In many clusters, as in these three examples, only one storage class called `standard` is configured. It's also marked as the default, which means that this is the class that is used to provision the persistent volume when the persistent volume claim doesn't specify the storage class.

> **NOTE** Remember that omitting the `storageClassName` field causes the default storage class to be used, whereas explicitly setting the field to "" disables dynamic provisioning and causes an existing persistent volume to be selected and bound to the claim.

INSPECTING THE DEFAULT STORAGE CLASS

Let's get to know the StorageClass object kind by inspecting the YAML definition of the `standard` storage class with the `kubectl get` command. In GKE, you'll find the following definition:

```
$ kubectl get sc standard -o yaml    #A
allowVolumeExpansion: true
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"    #B
  name: standard
  ...
parameters:    #C
  type: pd-standard    #C
provisioner: kubernetes.io/gce-pd    #D
reclaimPolicy: Delete    #E
volumeBindingMode: Immediate    #F
```

#A This command was run against a GKE cluster.
#B This marks the storage class as default.
#C The parameters for the provisioner
#D The name of the provisioner that gets called to provision persistent volumes of this class
#E The reclaim policy for persistent volumes of this class
#F When persistent volumes of this class are provisioned and bound

The storage class definition in a kind-provisioned cluster is not much different. The main differences are highlighted in bold:

```
$ kubectl get sc standard -o yaml    #A
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"    #B
  name: standard
  ...
provisioner: rancher.io/local-path                         #C
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer                    #D
```

#A This command was run against a kind-provisioned cluster.
#B Again, this is storage class is the default.
#C Kind uses a different provisioner than GKE. There are no parameters defined for the provisioner.
#D Kind uses a different volume binding mode than GKE.

In clusters created with Minikube, the standard storage class looks as follows:

```
$ kubectl get sc standard -o yaml    #A
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"    #B
  name: standard    #A
  ...
provisioner: k8s.io/minikube-hostpath    #C
reclaimPolicy: Delete    #D
volumeBindingMode: Immediate    #E
```

#A This command was run against a Minikube cluster.
#B This storage class is the default.
#C Minikube uses its own provisioner.
#D The volume binding mode is the same as in GKE.

> **NOTE** You'll notice that StorageClass objects have no `spec` or `status` sections. This is because the object only contains static information. Since the object's fields aren't organized in the two sections, the YAML manifest may be more difficult to read. This is also confounded by the fact that fields in YAML are typically sorted in alphabetical order, which means that some fields may appear above the `apiVersion`, `kind` or `metadata` fields. Be careful not to overlook these.

If you look closely at the top of the storage class definitions, you'll see that they all include an annotation that marks the storage class as default.

> **NOTE** You'll learn what an object annotation is in chapter 10.

As specified in GKE's storage class definition, when you create a persistent volume claim that references the `standard` class in GKE, the provisioner `kubernetes.io/gce-pd` is called to provision the persistent volume. In kind-provisioned clusters, the provisioner is `rancher.io/local-path`, whereas in Minikube it's `k8s.io/minikube-hostpath`. GKE's default storage class also specifies a parameter that is provided to the provisioner.

Regardless of what provisioner is used, the volume's reclaim policy is set to whatever is specified in the storage class, which in all of the previous examples is `Delete`. As you have already learned, this means that the volume is deleted when you release it by deleting the claim.

The last field in the storage class definition is `volumeBindingMode`. Both GKE and Minikube use the volume binding mode `Immediate`, whereas kind uses `WaitForFirstConsumer`. You'll learn what the difference is later in this chapter.

StorageClass objects also support several other fields that are not shown in the above listing. You can use `kubectl explain` to see what they are. You'll learn about some of them in the following sections.

In summary, a StorageClass object represents a class of storage that can be dynamically provisioned. As shown in the following figure, each storage class specifies what provisioner to use and the parameters that should be passed to it when provisioning the volume. The user decides which storage class to use for each of their persistent volume claims.

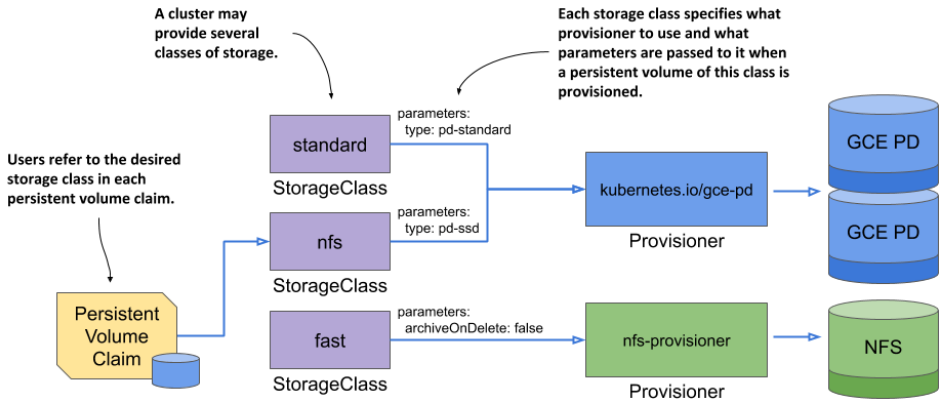

Figure 8.8 The relationship between storage classes, persistent volume claims and dynamic volume provisioners

8.3.2 Dynamic provisioning using the default storage class

You've previously used a statically provisioned persistent volume for the quiz pod. Now you'll use dynamic provisioning to achieve the same result, but with much less manual work. And most importantly, you can use the same pod manifest, regardless of whether you use GKE, Minikube, kind, or any other tool to run your cluster, assuming that a default storage class exists in the cluster.

CREATING A CLAIM WITH DYNAMIC PROVISIONING

To dynamically provision a persistent volume using the storage class from the previous section, you can create a PersistentVolumeClaim object with the `storageClassName` field set to `standard` or with the field omitted altogether.

Let's use the latter approach, as this makes the manifest as minimal as possible. You can find the manifest in the Chapter09/pvc.quiz-data-default.yaml file. Its contents are shown in the following listing.

Listing 8.8 A minimal PVC definition that uses the default storage class

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: quiz-data-default
spec:
  resources:
    requests:
      storage: 1Gi    #A
  accessModes:
    - ReadWriteOnce    #B
#C
```

#A The minimum size
#B The desired access mode

#C No storageClassName field

This PersistentVolumeClaim manifest contains only the storage size request and the desired access mode, but no `storageClassName` field, so the default storage class is used.

After you create the claim with `kubectl apply`, you can see which storage class it's using by inspecting the claim with `kubectl get`. This is what you'll see if you use GKE:

```
$ kubectl get pvc quiz-data-default
NAME                STATUS   VOLUME             CAPACITY   ACCESS MODES   STORAGECLASS
       AGE
quiz-data-default   Bound    pvc-ab623265-...   1Gi        RWO            standard       3m
```

As expected, and as indicated in the `STORAGECLASS` column, the claim you just created uses the `standard` storage class.

In GKE and Minikube, the persistent volume is created immediately and bound to the claim. However, if you create the same claim in a kind-provisioned cluster, that's not the case:

```
$ kubectl get pvc quiz-data-default
NAME                STATUS    VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
quiz-data-default   Pending                                      standard       3m
```

In a kind-provisioned cluster, and possibly other clusters, too, the persistent volume claim you just created is not bound immediately and its status is `Pending`.

In one of the previous sections, you learned that this happens when no persistent volume matches the claim, either because it doesn't exist or because it's not available for binding. However, you are now using dynamic provisioning, where the volume should be created after you create the claim, and specifically for this claim. Is your claim pending because the cluster needs more time to provision the volume?

No, the reason for the pending status lies elsewhere. Your claim will remain in the `Pending` state until you create a pod that uses this claim. I'll explain why later. For now, let's just create the pod.

USING THE PERSISTENT VOLUME CLAIM IN A POD

Create a new pod manifest file from the pod.quiz.pvc.yaml file that you created earlier. Change the name of the pod to `quiz-default` and the value of the `claimName` field to `quiz-data-default`. You can find the resulting manifest in the file Chapter09/pod.quiz-default.yaml. Use this file to create the pod.

If you use a kind-provisioned cluster, the status of the persistent volume claim should change to `Bound` within moments of creating the pod:

```
$ kubectl get pvc quiz-data-default
NAME                STATUS   VOLUME             CAPACITY   ACCESS   ...
quiz-data-default   Bound    pvc-c71fb2c2-...   1Gi        RWO      ...
```

This implies that the persistent volume has been created. List persistent volumes to confirm (the following output has been reformatted to make it easier to read):

```
$ kubectl get pv
NAME              CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   ...
pvc-c71fb2c2...   1Gi        RWO            Delete           Bound    ...
```

```
...   STATUS   CLAIM                        STORAGECLASS   REASON   AGE
...   Bound    default/quiz-data-default    standard                3s
```

As you can see, because the volume was created on demand, its properties perfectly match the requirements specified in the claim and the storage class it references. The volume capacity is `1Gi` and the access mode is `RWO`.

UNDERSTANDING WHEN A DYNAMICALLY PROVISIONED VOLUME IS ACTUALLY PROVISIONED

Why is the volume in a kind-provisioned cluster created and bound to the claim only after you deploy the pod? In an earlier example that used a manually pre-provisioned persistent volume, the volume was bound to the claim as soon as you created the claim. Is this a difference between static and dynamic provisioning? Because in both GKE and Minikube, the volume was dynamically provisioned and bound to the claim immediately, it's clear that dynamic provisioning alone is not responsible for this behavior.

The system behaves this way because of how the storage class in a kind-provisioned cluster is configured. You may remember that this storage class was the only one that has `volumeBindingMode` set to `WaitForFirstConsumer`. This causes the system to wait until the first pod, or the *consumer* of the claim, exists before the claim is bound. The persistent volume is also not provisioned before that.

Some types of volumes require this type of behavior, because the system needs to know *where* the pod is scheduled *before* it can provision the volume. This is the case with provisioners that create node-local volumes, such as the one you find in clusters created with the kind tool. You may remember that the provisioner referenced in the storage class had the word "local" in its name (`rancher.io/local-path`). Minikube also provisions a local volume (the provisioner it uses is called `k8s.io/minikube-hostpath`), but because there's only one node in the cluster, there's no need to wait for the pod to be created in order to know which node the persistent volume needs to be created on.

> **NOTE** Refer to the documentation of your chosen provisioner to determine whether it requires the volume binding mode to be set to `WaitForFirstConsumer`.

The alternative to `WaitForFirstConsumer` is the `Immediate` volume binding mode. The two modes are explained in the following table.

Table 8.4 Supported volume binding modes

Volume binding mode	Description
`Immediate`	The provision and binding of the persistent volume takes place immediately after the claim is created. Because the consumer of the claim is unknown at this point, this mode is only applicable to volumes that are can be accessed from any cluster node.
`WaitForFirstConsumer`	The volume is provisioned and bound to the claim when the first pod that uses this claim is created. This mode is used for topology-constrained volume types.

8.3.3 Creating a storage class and provisioning volumes of that class

As you saw in the previous sections, most Kubernetes clusters contain a single storage class named `standard`, but use different provisioners. A full-blown cluster such as the one you find in GKE can surely provide more than just a single type of persistent volume. So how does one create other types of volumes?

INSPECTING THE DEFAULT STORAGE CLASS IN GKE

Let's look at the default storage class in GKE more closely. I've rearranged the fields since the original alphabetical ordering makes the YAML definition more difficult to understand. The storage class definition follows:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
    ...
provisioner: kubernetes.io/gce-pd    #A
parameters:    #B
  type: pd-standard    #B
volumeBindingMode: Immediate
allowVolumeExpansion: true
reclaimPolicy: Delete
```

#A The provisioner used to provision volumes of this storage class
#B This type parameter is passed to the provisioner

If you create a persistent volume claim that references this storage class, the provisioner `kubernetes.io/gce-pd` is called to create the volume. In this call, the provisioner receives the parameters defined in the storage class. In the case of the default storage class in GKE, the parameter `type: pd-standard` is passed to the provisioner. This tells the provisioner what type of GCE Persistent Disk to create.

You can create additional storage class objects and specify a different value for the `type` parameter. You'll do this next.

> **NOTE** The availability of GCE Persistent Disk types depends on the zone in which your cluster is deployed. To view the list of types for each availability zone, run `gcloud compute disk-types list`.

CREATING A NEW STORAGE CLASS TO ENABLE THE USE OF SSD PERSISTENT DISKS IN GKE

One of the disk types supported in most GCE zones is the `pd-ssd` type, which provisions a network-attached SSD. Let's create a storage class called `fast` and configure it so that the provisioner creates a disk of type `pd-ssd` when you request this storage class in your claim. The storage class manifest is shown in the next listing (Chapter09/sc.fast.gcepd.yaml).

Listing 8.9 A custom storage class definition

```
apiVersion: storage.k8s.io/v1           #A
kind: StorageClass                      #A
metadata:
  name: fast                            #B
provisioner: kubernetes.io/gce-pd       #C
parameters:
  type: pd-ssd                          #D
```

#A This manifest defines a StorageClass object
#B The name of this storage class
#C The provisioner to use
#D Tells the provisioner to provision an SSD disk

> **NOTE** If you're using another cloud provider, check their documentation to find the name of the provisioner and the parameters you need to pass in. If you're using Minikube or kind, and you'd like to run this example, set the provisioner and parameters to the same values as in the default storage class. For this exercise, it doesn't matter if the provisioned volume doesn't actually use an SSD.

Create the StorageClass object by applying this manifest to your cluster and list the available storage classes to confirm that more than one is now available. You can now use this storage class in your claims. Let's conclude this section on dynamic provisioning by creating a persistent volume claim that will allow your Quiz pod to use an SSD disk.

CLAIMING A VOLUME OF A SPECIFIC STORAGE CLASS

The following listing shows the updated YAML definition of the `quiz-data` claim, which requests the storage class `fast` that you've just created instead of using the default class. You'll find the manifest in the file Chapter09/pvc.quiz-data-fast.yaml.

Listing 8.10 A persistent volume claim requesting a specific storage class

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: quiz-data-fast
spec:
  storageClassName: fast     #A
  resources:
    requests:
      storage: 1Gi
  accessModes:
    - ReadWriteOnce
```

#A This claim requests that this specific storage class be used to provision the volume.

Rather than just specify the size and access modes and let the system use the default storage class to provision the persistent volume, this claim specifies that the storage class `fast` be used for the volume. When you create the claim, the persistent volume is created by the provisioner referenced in this storage class, using the specified parameters.

You can now use this claim in a new instance of the Quiz pod. Apply the file Chapter09/pod.quiz-fast.yaml. If you run this example on GKE, the pod will use an SSD volume.

> **NOTE** If a persistent volume claim refers to a non-existent storage class, the claim remains `Pending` until the storage class is created. Kubernetes attempts to bind the claim at regular intervals, generating a `ProvisioningFailed` event each time. You can see the event if you execute the `kubectl describe` command on the claim.

8.3.4 Resizing persistent volumes

If the cluster supports dynamic provisioning, a cluster user can self-provision a storage volume with the properties and size specified in the claim and referenced storage class. If the user later needs a different storage class for their volume, they must, as you might expect, create a new persistent volume claim that references the other storage class. Kubernetes does not support changing the storage class name in an existing claim. If you try to do so, you receive the following error message:

```
* spec: Forbidden: is immutable after creation except resources.requests for bound claims
```

The error indicates that the majority of the claim's specification is immutable. The part that is mutable is `spec.resources.requests`, which is where you indicate the desired size of the volume.

In the previous MongoDB examples you requested 1GiB of storage space. Now imagine that the database grows near this size. Can the volume be resized without restarting the pod and application? Let's find out.

REQUESTING A LARGER VOLUME IN AN EXISTING PERSISTENT VOLUME CLAIM

If you use dynamic provisioning, you can generally change the size of a persistent volume simply by requesting a larger capacity in the associated claim. For the next exercise, you'll increase the size of the volume by modifying the `quiz-data-default` claim, which should still exist in your cluster.

To modify the claim, either edit the manifest file or create a copy and then edit it. Set the `spec.resources.requests.storage` field to `10Gi` as shown in the following listing. You can find this manifest in the book's GitHub repository (file quiz-data-default.dynamic.10gib.pvc.yaml).

Listing 8.11 Requesting a larger volume

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: quiz-data-default     #A
spec:
  resources:                  #B
    requests:                 #B
      storage: 10Gi           #B
  accessModes:
    - ReadWriteOnce
```

#A Ensure that the name matches the name of the existing claim.
#B Request a larger amount of storage.

When you apply this file with the `kubectl apply` command, the existing PersistentVolumeClaim object is updated. Use the `kubectl get pvc` command to see if the volume's capacity has increased:

```
$ kubectl get pvc quiz-data-default
NAME                STATUS   VOLUME         CAPACITY   ACCESS MODES   ...
quiz-data-default   Bound    pvc-ed36b...   1Gi        RWO            ...
```

You may recall that when claims are listed, the `CAPACITY` column displays the size of the bound volume and not the size requirement specified in the claim. According to the output, this means that the size of the volume hasn't changed. Let's find out why.

DETERMINING WHY THE VOLUME HASN'T BEEN RESIZED

To find out why the size of the volume has remained the same regardless of the change you made to the claim, the first thing you might do is inspect the claim using `kubectl describe`. If this is the case, you've already got the hang of debugging objects in Kubernetes. You'll find that one of the claim's conditions clearly explains why the volume was not resized:

```
$ kubectl describe pvc quiz-data-default
...
Conditions:
  Type                      Status  ... Message
  ----                      ------  ... -------
  FileSystemResizePending   True        Waiting for user to (re-)start a
                                        pod to finish file system resize of
                                        volume on node.
```

To resize the persistent volume, you may need to delete and recreate the pod that uses the claim. After you do this, the claim and the volume will display the new size:

```
$ kubectl get pvc quiz-data-default
NAME                STATUS   VOLUME         CAPACITY   ACCESS MODES   ...
quiz-data-default   Bound    pvc-ed36b...   10Gi       RWO            ...
```

ALLOWING AND DISALLOWING VOLUME EXPANSION IN THE STORAGE CLASS

The previous example shows that cluster users can increase the size of the bound persistent volume by changing the storage requirement in the persistent volume claim. However, this is only possible if it's supported by the provisioner and the storage class.

When the cluster administrator creates a storage class, they can use the `spec.allowVolumeExpansion` field to indicate whether volumes of this class can be resized. If you attempt to expand a volume that you're not supposed to expand, the API server immediately rejects the update operation on the claim.

8.3.5 Understanding the benefits of dynamic provisioning

This section on dynamic provisioning should convince you that automating the provisioning of persistent volumes benefits both the cluster administrator and anyone who uses the cluster to deploy applications. By setting up the dynamic volume provisioner and configuring several

storage classes with different performance or other features, the administrator gives cluster users the ability to provision as many persistent volumes of any type as they want. Each developer decides which storage class is best suited for each claim they create.

UNDERSTANDING HOW STORAGE CLASSES ALLOW CLAIMS TO BE PORTABLE

Another great thing about storage classes is that claims refer to them by name. If the storage classes are named appropriately, such as `standard`, `fast`, and so on, the persistent volume claim manifests are portable across different clusters.

> **NOTE** Remember that persistent volume claims are usually part of the application manifest and are written by application developers.

If you used GKE to run the previous examples, you can now try to deploy the same claim and pod manifests in a non-GKE cluster, such as a cluster created with Minikube or kind. In this way, you can see this portability for yourself. The only thing you need to ensure is that all your clusters use the storage class names.

8.3.6 Understanding the lifecycle of dynamically provisioned persistent volumes

To conclude this section on dynamic provisions, let's take one final look at the lifecycles of the underlying storage volume, the PersistentVolume object, the associated PersistentVolumeClaim object, and the pods that use them, like we did in the previous section on statically provisioned volumes.

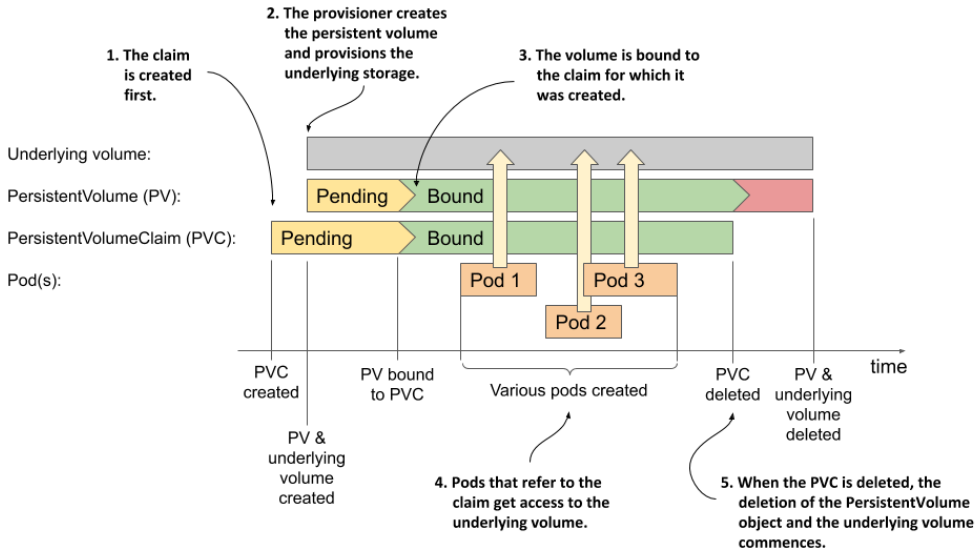

Figure 8.9 The lifecycle of dynamically provisioned persistent volumes, claims and the pods using them

Unlike statically provisioned persistent volumes, the sequence of events when using dynamic provisioning begins with the creation of the PersistentVolumeClaim object. As soon as one such object appears, Kubernetes instructs the dynamic provisioner configured in the storage class referenced in this claim to provision a volume for it. The provisioner creates both the underlying storage, typically through the cloud provider's API, and the PersistentVolume object that references the underlying volume.

The underlying volume is typically provisioned asynchronously. When the process completes, the status of the PersistentVolume object changes to Available; at this point, the volume is bound to the claim.

Users can then deploy pods that refer to the claim to gain access to the underlying storage volume. When the volume is no longer needed, the user deletes the claim. This typically triggers the deletion of both the PersistentVolume object and the underlying storage volume.

This entire process is repeated for each new claim that the user creates. A new PersistentVolume object is created for each claim, which means that the cluster can never run out of them. Obviously, the datacentre itself can run out of available disk space, but at least there is no need for the administrator to keep recycling old PersistentVolume objects.

8.4 Node-local persistent volumes

In the previous sections of this chapter, you've used persistent volumes and claims to provide network-attached storage volumes to your pods, but this type of storage is too slow for some applications. To run a production-grade database, you should probably use an SSD connected directly to the node where the database is running.

In the previous chapter, you learned that you can use a `hostPath` volume in a pod if you want the pod to access part of the host's filesystem. Now you'll learn how to do the same with persistent volumes. You might wonder why I need to teach you another way to do the same thing, but it's really not the same.

You might remember that when you add a `hostPath` volume to a pod, the data that the pod sees depends on which node the pod is scheduled to. In other words, if the pod is deleted and recreated, it might end up on another node and no longer have access to the same data.

If you use a local persistent volume instead, this problem is resolved. The Kubernetes scheduler ensures that the pod is always scheduled on the node to which the local volume is attached.

> **NOTE** Local persistent volumes are also better than `hostPath` volumes because they offer much better security. As explained in the previous chapter, you don't want to allow regular users to use `hostPath` volumes at all. Because persistent volumes are managed by the cluster administrator, regular users can't use them to access arbitrary paths on the host node.

8.4.1 Creating local persistent volumes

Imagine you are a cluster administrator and you have just connected a fast SSD directly to one of the worker nodes. Because this is a new class of storage in the cluster, it makes sense to create a new StorageClass object that represents it.

CREATING A STORAGE CLASS TO REPRESENT LOCAL STORAGE

Create a new storage class manifest as shown in the following listing.

Listing 8.12 Defining the local storage class

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local                                  #A
provisioner: kubernetes.io/no-provisioner      #B
volumeBindingMode: WaitForFirstConsumer        #C
```

#A Let's call this storage class local
#B Persistent volumes of this class are provisioned manually
#C The persistent volume claim should be bound only when the first pod that uses the claim is deployed.

As I write this, locally attached persistent volumes need to be provisioned manually, so you need to set the provisioner as shown in the listing. Because this storage class represents locally attached volumes that can only be accessed within the nodes to which they are physically connected, the `volumeBindingMode` is set to `WaitForFirstConsumer`, so the binding of the claim is delayed until the pod is scheduled.

ATTACHING A DISK TO A CLUSTER NODE

I assume that you're using a Kubernetes cluster created with the kind tool to run this exercise. Let's emulate the installation of the SSD in the node called `kind-worker`. Run the following command to create an empty directory at the location `/mnt/ssd1` in the node's filesystem:

```
$ docker exec kind-worker mkdir /mnt/ssd1
```

CREATING A PERSISTENTVOLUME OBJECT FOR THE NEW DISK

After attaching the disk to one of the nodes, you must tell Kubernetes that this node now provides a local persistent volume by creating a PersistentVolume object. The manifest for the persistent volume is shown in the following listing.

Listing 8.13 Defining a local persistent volume

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: local-ssd-on-kind-worker       #A
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: local              #B
  capacity:
    storage: 10Gi
  local:                               #C
    path: /mnt/ssd1                    #C
  nodeAffinity:                        #D
    required:                          #D
      nodeSelectorTerms:               #D
      - matchExpressions:              #D
        - key: kubernetes.io/hostname  #D
          operator: In                 #D
          values:                      #D
          - kind-worker                #D
```

#A This persistent volume represents the local SSD installed in the kind-worker node, hence the name.
#B This volume belongs to the local storage class.
#C This volume is mounted in the node's filesystem at the specified path.
#D This section tells Kubernetes which nodes can access this volume. Since the SSD is attached only to the node kind-worker, it is only accessible on this node.

Because this persistent volume represents a local disk attached to the `kind-worker` node, you give it a name that conveys this information. It refers to the `local` storage class that you created previously. Unlike previous persistent volumes, this volume represents storage space that is directly attached to the node. You therefore specify that it is a `local` volume. Within the `local` volume configuration, you also specify the path where the SSD is mounted (`/mnt/ssd1`).

At the bottom of the manifest, you'll find several lines that indicate the volume's node affinity. A volume's node affinity defines which nodes can access this volume.

> **NOTE** You'll learn more about node affinity and selectors in later chapters. Although it looks complicated, the node affinity definition in the listing simply defines that the volume is accessible from nodes whose `hostname` is `kind-worker`. This is obviously exactly one node.

Okay, as a cluster administrator, you've now done everything you needed to do to enable cluster users to deploy applications that use locally attached persistent volumes. Now it's time to put your application developer hat back on again.

8.4.2 Claiming and using local persistent volumes

As an application developer, you can now deploy your pod and its associated persistent volume claim.

CREATING THE POD

The pod definition is shown in the following listing.

Listing 8.14 Pod using a locally attached persistent volume

```
apiVersion: v1
kind: Pod
metadata:
  name: mongodb-local
spec:
  volumes:
  - name: mongodb-data
    persistentVolumeClaim:
      claimName: quiz-data-local      #A
  containers:
  - image: mongo
    name: mongodb
    volumeMounts:
    - name: mongodb-data
      mountPath: /data/db
```

#A The pod uses the quiz-data-local claim

There should be no surprises in the pod manifest. You already know all this.

CREATING THE PERSISTENT VOLUME CLAIM FOR A LOCAL VOLUME

As with the pod, creating the claim for a local persistent volume is no different than creating any other persistent volume claim. The manifest is shown in the next listing.

Listing 8.15 Persistent volume claim using the local storage class

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: quiz-data-local
spec:
  storageClassName: local               #A
  resources:
    requests:
      storage: 1Gi
  accessModes:
    - ReadWriteOnce
```

#A The claim requests a persistent volume from the local storage class

No surprises here either. Now on to creating these two objects.

CREATING THE POD AND THE CLAIM

After you write the pod and claim manifests, you can create the two objects by applying the manifests in any order you want. If you create the pod first, since the pod requires the claim to exist, it simply remains in the `Pending` state until you create the claim.

After both the pod and the claim are created, the following events take place:

15. The claim is bound to the persistent volume.
16. The scheduler determines that the volume bound to the claim that is used in the pod can only be accessed from the kind-worker node, so it schedules the pod to this node.
17. The pod's container is started on this node, and the volume is mounted in it.

You can now use the MongoDB shell again to add documents to it. Then check the `/mnt/ssd1` directory on the kind-worker node to see if the files are stored there.

RECREATING THE POD

If you delete and recreate the pod, you'll see that it's always scheduled on the kind-worker node. The same happens if multiple nodes can provide a local persistent volume when you deploy the pod for the first time. At this point, the scheduler selects one of them to run your MongoDB pod. When the pod runs, the claim is bound to the persistent volume on that particular node. If you then delete and recreate the pod, it is always scheduled on the same node, since that is where the volume that is bound to the claim referenced in the pod is located.

8.5 Summary

This chapter explained the details of adding persistent storage for your applications. You've learned that:

- Infrastructure-specific information about storage volumes doesn't belong in pod manifests. Instead, it should be specified in the PersistentVolume object.
- A PersistentVolume object represents a portion of the disk space that is available to applications within the cluster.
- Before an application can use a PersistentVolume, the user who deploys the application must claim the PersistentVolume by creating a PersistentVolumeClaim object.
- A PersistentVolumeClaim object specifies the minimum size and other requirements that the PersistentVolume must meet.
- When using statically provisioned volumes, Kubernetes finds an existing persistent volume that meets the requirements set forth in the claim and binds it to the claim.
- When the cluster provides dynamic provisioning, a new persistent volume is created for each claim. The volume is created based on the requirements specified in the claim.
- A cluster administrator creates StorageClass objects to specify the storage classes that users can request in their claims.
- A user can change the size of the persistent volume used by their application by modifying the minimum volume size requested in the claim.
- Local persistent volumes are used when applications need to access disks that are directly attached to nodes. This affects the scheduling of the pods, since the pod must be scheduled to one of the nodes that can provide a local persistent volume. If the pod is subsequently deleted and recreated, it will always be scheduled to the same node.

In the next chapter, you'll learn how to pass configuration data to your applications using command-line arguments, environment variables, and files. You'll learn how to specify this data directly in the pod manifest and other Kubernetes API objects.

9

Configuring applications using ConfigMaps, Secrets, and the Downward API

This chapter covers

- Setting the command and arguments for the container's main process
- Setting environment variables
- Storing configuration in config maps
- Storing sensitive information in secrets
- Using the Downward API to expose pod metadata to the application
- Using configMap, secret, downwardAPI and projected volumes

You've now learned how to use Kubernetes to run an application process and attach file volumes to it. In this chapter, you'll learn how to configure the application - either in the pod manifest itself, or by referencing other API objects within it. You'll also learn how to inject information about the pod itself into the application running inside it.

> **NOTE** You'll find the code files for this chapter at https://github.com/luksa/kubernetes-in-action-2nd-edition/tree/master/Chapter09

9.1 Setting the command, arguments, and environment variables

Like regular applications, containerized applications can be configured using command-line arguments, environment variables, and files.

You learned that the command that is executed when a container starts is typically defined in the container image. The command is configured in the container's Dockerfile using the `ENTRYPOINT` directive, while the arguments are typically specified using the `CMD` directive. Environment variables can also be specified using the the `ENV` directive in the Dockerfile. If the application is configured using configuration files, these can be added to the container image using the `COPY` directive. You've seen several examples of this in the previous chapters.

Let's take the kiada application and make it configurable via command-line arguments and environment variables. The previous versions of the application all listen on port 8080. This will now be configurable via the `--listen-port` command line argument. Also, the application will read the initial status message from the environment variable `INITIAL_STATUS_MESSAGE`. Instead of just returning the hostname, the application now also returns the pod name and IP address, as well as the name of the cluster node on which it is running. The application obtains this information through environment variables. You can find the updated code in the book's code repository. The container image for this new version is available at docker.io/luksa/kiada:0.4.

The updated Dockerfile, which you can also find in the code repository, is shown in the following listing.

Listing 9.1 A sample Dockerfile using several application configuration methods

```
FROM node:12
COPY app.js /app.js
COPY html/ /html

ENV INITIAL_STATUS_MESSAGE="This is the default status message"    #A

ENTRYPOINT ["node", "app.js"]                                      #B
CMD ["--listen-port", "8080"]                                      #C
```

#A Set an environment variable
#B Set the command to run when the container is started
#C Set the default command-line arguments

Hardcoding the configuration into the container image is the same as hardcoding it into the application source code. This is not ideal because you must rebuild the image every time you change the configuration. Also, you should never include sensitive configuration data such as security credentials or encryption keys in the container image because anyone who has access to it can easily extract them.

Instead, it's much safer to store these files in a volume that you mount in the container. As you learned in the previous chapter, one way to do this is to store the files in a persistent volume. Another way is to use an `emptyDir` volume and an init container that fetches the files from secure storage and writes them to the volume. You should know how to do this if you've read the previous chapters, but there's a better way. In this chapter, you'll learn how to use special volume types to achieve the same result without using init containers. But first, let's learn how to change the command, arguments, and environment variables without recreating the container image.

9.1.1 Setting the command and arguments

When creating a container image, the command and its arguments are specified using the `ENTRYPOINT` and `CMD` directives in the Dockerfile. Since both directives accept array values, you can specify both the command and its arguments with one of these directives or split them between the two. When the container is executed, the two arrays are concatenated to produce the full command.

Kubernetes provides two fields that are analogous to Docker's `ENTRYPOINT` and `CMD` directives. The two fields are called `command` and `args`, respectively. You specify these fields in the container definition in your pod manifest. As with Docker, the two fields accept array values, and the resulting command executed in the container is derived by concatenating the two arrays.

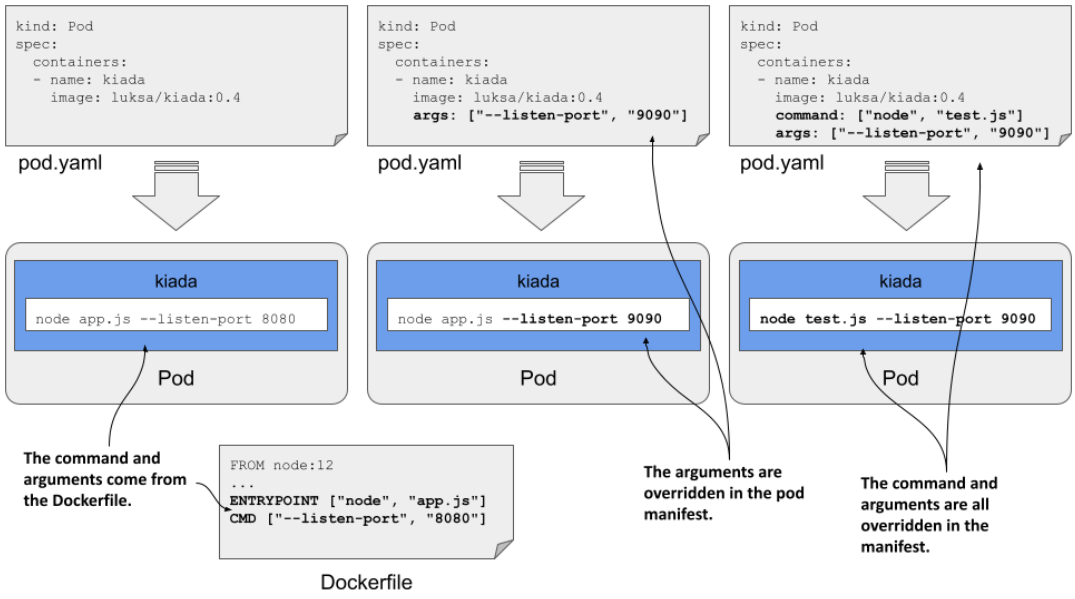

Figure 9.1 Overriding the command and arguments in the pod manifest

Normally, you use the `ENTRYPOINT` directive to specify the bare command, and the `CMD` directive to specify the arguments. This allows you to override the arguments in the pod manifest without having to specify the command again. If you want to override the command, you can still do so. And you can do it without overriding the arguments.

The following table shows the equivalent pod manifest field for each of the two Dockerfile directives.

Table 9.1 Specifying the command and arguments in the Dockerfile vs the pod manifest

Dockerfile	Pod manifest	Description
`ENTRYPOINT`	`command`	The executable file that runs in the container. This may contain arguments in addition to the executable.
`CMD`	`args`	Additional arguments passed to the command specified with the `ENTRYPOINT` directive or the `command` field.

Let's look at two examples of setting the `command` and `args` fields.

SETTING THE COMMAND

Imagine you want to run the Kiada application with CPU and heap profiling enabled. With Node.JS, you can enable profiling by passing the `--cpu-prof` and `--heap-prof` arguments to the `node` command. Instead of modifying the Dockerfile and rebuilding the image, you can do this by modifying the pod manifest, as shown in the following listing.

Listing 9.2 A container definition with the command specified

```
kind: Pod
spec:
  containers:
  - name: kiada
    image: luksa/kiada:0.4
    command: ["node", "--cpu-prof", "--heap-prof", "app.js"]    #A
```

#A When the container is started, this command is executed instead of the one defined in the container image

When you deploy the pod in the listing, the `node --cpu-prof --heap-prof app.js` command is run instead of the default command specified in the Dockerfile, which is `node app.js`.

As you can see in the listing, the `command` field, just like its Dockerfile counterpart, accepts an array of strings representing the command to be executed. The array notation used in the listing is great when the array contains only a few elements, but becomes difficult to read as the number of elements increases. In this case, you're better off using the following notation:

```
    command:
    - node
    - --cpu-prof
    - --heap-prof
    - app.js
```

> **TIP** Values that the YAML parser might interpret as something other than a string must be enclosed in quotes. This includes numeric values such as `1234`, and Boolean values such as `true` and `false`. Some other special strings must also be quoted, otherwise they would also be interpreted as Boolean or other types. These include the values `true`, `false`, `yes`, `no`, `on`, `off`, `y`, `n`, `t`, `f`, `null`, and others.

SETTING COMMAND ARGUMENTS

Command line arguments can be overridden with the `args` field, as shown in the following listing.

Listing 9.3 A container definition with the args fields set

```
kind: Pod
spec:
  containers:
  - name: kiada
    image: luksa/kiada:0.4
    args: ["--listen-port", "9090"]    #A
```

#A This overrides the arguments set in the container image

The pod manifest in the listing overrides the default `--listen-port 8080` arguments set in the Dockerfile with `--listen-port 9090`. When you deploy this pod, the full command that runs in the container is `node app.js --listen-port 9090`. The command is a concatenation of the `ENTRYPOINT` in the Dockerfile and the `args` field in the pod manifest.

9.1.2 Setting environment variables in a container

Containerized applications are often configured using environment variables. Just like the command and arguments, you can set environment variables for each of the pod's containers, as shown in figure 9.2.

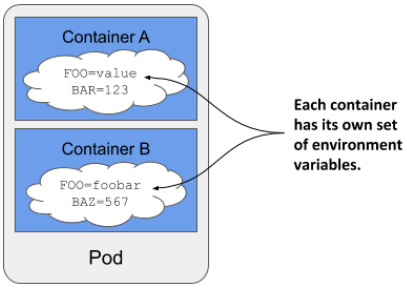

Figure 9.2 Environment variables are set per container.

> **NOTE** As I write this, environment variables can only be set for each container individually. It isn't possible to set a global set of environment variables for the entire pod and have them inherited by all its containers.

You can set an environment variable to a literal value, have it reference another environment variable, or obtain the value from an external source. Let's see how.

SETTING A LITERAL VALUE TO AN ENVIRONMENT VARIABLE

Version 0.4 of the Kiada application displays the name of the pod, which it reads from the environment variable `POD_NAME`. It also allows you to set the status message using the environment variable `INITIAL_STATUS_MESSAGE`. Let's set these two variables in the pod manifest.

To set the environment variable, you could add the `ENV` directive to the Dockerfile and rebuild the image, but the faster way is to add the `env` field to the container definition in the pod manifest, as I've done in the file pod.kiada.env-value.yaml shown in the following listing.

Listing 9.4 Setting environment variables in the pod manifest

```
kind: Pod
metadata:
  name: kiada
spec:
  containers:
  - name: kiada
    image: luksa/kiada:0.4
    env:                                                    #A
    - name: POD_NAME                                        #B
      value: kiada                                          #B
    - name: INITIAL_STATUS_MESSAGE                          #C
      value: This status message is set in the pod spec.    #C
    ...
```

#A The env field contains a list of environment variables for the container
#B The environment variable POD_NAME is set to "kiada"
#C Another environment variable is set here.

As you can see in the listing, the `env` field takes an array of values. Each entry in the array specifies the name of the environment variable and its value.

> **NOTE** Since environment variables values must be strings, you must enclose values that aren't strings in quotes to prevent the YAML parser from treating them as anything other than a string. As explained in section 9.1.1, this also applies to strings such as `yes`, `no`, `true`, `false`, and so on.

When you deploy the pod in the listing and send an HTTP request to the application, you should see the pod name and status message that you specified using environment variables. You can also run the following command to examine the environment variables in the container. You'll find the two environment variables in the following output:

```
$ kubectl exec kiada -- env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin    #A
HOSTNAME=kiada                                                       #A
NODE_VERSION=12.19.1                                                 #B
YARN_VERSION=1.22.5                                                  #B
POD_NAME=kiada                                                       #C
INITIAL_STATUS_MESSAGE=This status message is set in the pod spec.   #C
KUBERNETES_SERVICE_HOST=10.96.0.1                                    #D
...                                                                  #D
KUBERNETES_SERVICE_PORT=443                                          #D
```

#A Set by the system
#B Set in the container image
#C Set in the pod manifest
#D Set by Kubernetes

As you can see, there are a few other variables set in the container. They come from different sources - some are defined in the container image, some are added by Kubernetes, and the rest come from elsewhere. While there is no way to know where each of the variables comes from, you'll learn to recognize some of them. For example, the ones added by Kubernetes relate to the Service object, which is covered in chapter 11. To determine where the rest come from, you can inspect the pod manifest and the Dockerfile of the container image.

USING VARIABLE REFERENCES IN ENVIRONMENT VARIABLE VALUES

In the previous example, you set a fixed value for the environment variable `INITIAL_STATUS_MESSAGE`, but you can also reference other environment variables in the value by using the syntax `$(VAR_NAME)`.

For example, you can reference the variable `POD_NAME` within the status message variable as shown in the following listing, which shows part of the file pod.kiada.env-value-ref.yaml.

Listing 9.5 Referring to an environment variable in another variable

```
env:
- name: POD_NAME
  value: kiada
- name: INITIAL_STATUS_MESSAGE
  value: My name is $(POD_NAME). I run NodeJS version $(NODE_VERSION).   #A
```

#A The value includes a reference to the POD_NAME and NODE_VERSION environment variables

Notice that one of the references points to the environment variable `POD_NAME` defined above, whereas the other points to the variable `NODE_VERSION` set in the container image. You saw this variable when you ran the `env` command in the container earlier. When you deploy the pod, the status message it returns is the following:

```
My name is kiada. I run NodeJS version $(NODE_VERSION).
```

As you can see, the reference to `NODE_VERSION` isn't resolved. This is because you can only use the `$(VAR_NAME)` syntax to refer to variables defined in the same manifest. The referenced variable must be defined *before* the variable that references it. Since `NODE_VERSION` is defined in the NodeJS image's Dockerfile and not in the pod manifest, it can't be resolved.

> **NOTE** If a variable reference can't be resolved, the reference string remains unchanged.

> **NOTE** When you want a variable to contain the literal string `$(VAR_NAME)` and don't want Kubernetes to resolve it, use a double dollar sign as in `$$(VAR_NAME)`. Kubernetes will remove one of the dollar signs and skip resolving the variable.

USING VARIABLE REFERENCES IN THE COMMAND AND ARGUMENTS

You can refer to environment variables defined in the manifest not only in other variables, but also in the `command` and `args` fields you learned about in the previous section. For example, the file pod.kiada.env-value-ref-in-args.yaml defines an environment variable named `LISTEN_PORT` and references it in the `args` field. The following listing shows the relevant part of this file.

Listing 9.6 Referring to an environment variable in the args field

```
spec:
  containers:
  - name: kiada
    image: luksa/kiada:0.4
    args:
    - --listen-port
    - $(LISTEN_PORT)           #A
    env:
    - name: LISTEN_PORT
      value: "8080"
```

#A Resolved to the LISTEN_PORT variable set below

This isn't the best example, since there's no good reason to use a variable reference instead of just specifying the port number directly. But later you'll learn how to get the environment variable value from an external source. You can then use a reference as shown in the listing to inject that value into the container's command or arguments.

REFERRING TO ENVIRONMENT VARIABLES THAT AREN'T IN THE MANIFEST

Just like using references in environment variables, you can only use the `$(VAR_NAME)` syntax in the `command` and `args` fields to reference variables that are defined in the pod manifest. You can't reference environment variables defined in the container image, for example.

However, you can use a different approach. If you run the command through a shell, you can have the shell resolve the variable. If you are using the bash shell, you can do this by referring to the variable using the syntax `$VAR_NAME` or `${VAR_NAME}` instead of `$(VAR_NAME)`.

For example, the command in the following listing correctly prints the value of the `HOSTNAME` environment variable even though it's not defined in the pod manifest but is initialized by the operating system. You can find this example in the file pod.env-var-references-in-shell.yaml.

Listing 9.7 Referring to environment variables in a shell command

```
containers:
- name: main
  image: alpine
  command:
  - sh   #A
  - -c   #A
  - 'echo "Hostname is $HOSTNAME."; sleep infinity'   #B
```

#A The top command executed in this container is the shell.
#B The shell resolves the reference to the HOSTNAME environment variable before executing the commands echo and sleep.

Setting the pod's fully qualified domain name

While we're on the subject of the pod's hostname, this is a good time to explain that the pod's hostname and subdomain are configurable in the pod manifest. By default, the hostname is the same as the pod's name, but you can override it using the `hostname` field in the pod's `spec`. You can also set the `subdomain` field so that the fully qualified domain name (FQDN) of the pod is as follows: `<hostname>.<subdomain>.<pod namespace>.svc.<cluster domain>`

This is only the internal FQDN of the pod. It isn't resolvable via DNS without additional steps, which are explained in chapter 11. You can find a sample pod that specifies a custom hostname for the pod in the file pod.kiada.hostname.yaml.

9.2 Using a config map to decouple configuration from the pod

In the previous section, you learned how to hardcode configuration directly into your pod manifests. While this is much better than hard-coding in the container image, it's still not ideal because it means you might need a separate version of the pod manifest for each environment you deploy the pod to, such as your development, staging, or production cluster.

To reuse the same pod definition in multiple environments, it's better to decouple the configuration from the pod manifest. One way to do this is to move the configuration into a ConfigMap object, which you then reference in the pod manifest. This is what you'll do next.

9.2.1 Introducing ConfigMaps

A ConfigMap is a Kubernetes API object that simply contains a list of key/value pairs. The values can range from short strings to large blocks of structured text that you typically find in an application configuration file. Pods can reference one or more of these key/value entries in the config map. A pod can refer to multiple config maps, and multiple pods can use the same config map.

To keep applications Kubernetes-agnostic, they typically don't read the ConfigMap object via the Kubernetes REST API. Instead, the key/value pairs in the config map are passed to containers as environment variables or mounted as files in the container's filesystem via a `configMap` volume, as shown in the following figure.

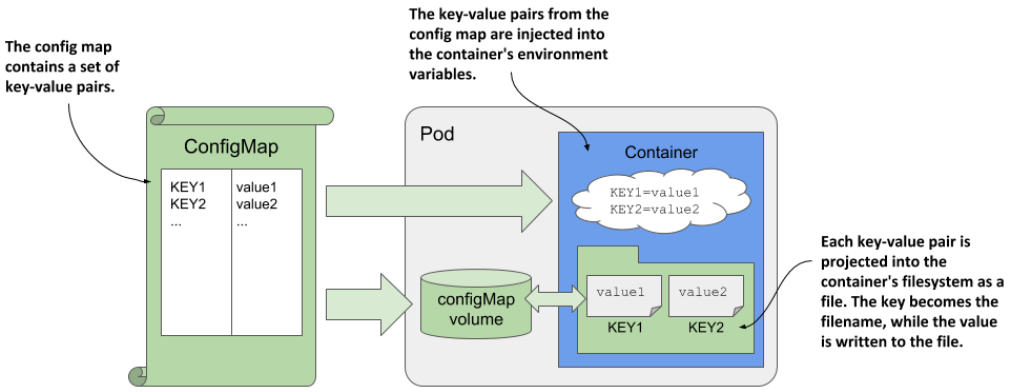

Figure 9.3 Pods use config maps through environment variables and configMap volumes.

In the previous section you learned how to reference environment variables in command-line arguments. You can use this technique to pass a config map entry that you've exposed as an environment variable into a command-line argument.

Regardless of how an application consumes config maps, storing the configuration in a separate object instead of the pod allows you to keep the configuration separate for different environments by simply keeping separate config map manifests and applying each to the environment for which it is intended. Because pods reference the config map by name, you can deploy the same pod manifest across all your environments and still have a different configuration for each environment by using the same config map name, as shown in the following figure.

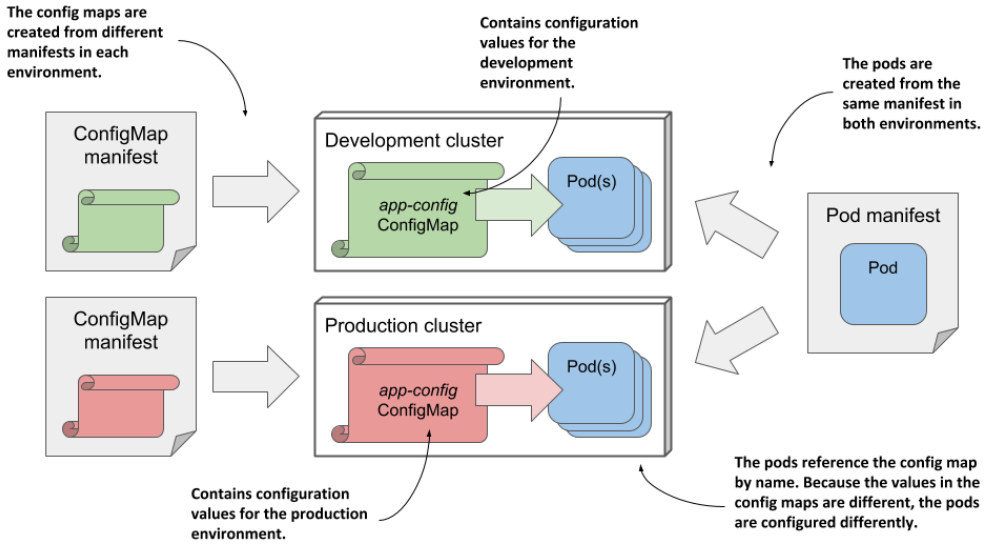

Figure 9.4 Deploying the same pod manifest and different config map manifests in different environments

9.2.2 Creating a ConfigMap object

Let's create a config map and use it in a pod. The following is a simple example where the config map contains a single entry used to initialize the environment variable `INITIAL_STATUS_MESSAGE` for the kiada pod.

CREATING A CONFIG MAP WITH THE KUBECTL CREATE CONFIGMAP COMMAND

As with pods, you can create the ConfigMap object from a YAML manifest, but a faster way is to use the `kubectl create configmap` command as follows:

```
$ kubectl create configmap kiada-config --from-literal status-message="This status message
      is set in the kiada-config config map"
configmap "kiada-config" created
```

> **NOTE** Keys in a config map may only consist of alphanumeric characters, dashes, underscores, or dots. Other characters are not allowed.

Running this command creates a config map called `kiada-config` with a single entry. The key and value are specified with the `--from-literal` argument.

In addition to `--from-literal`, the `kubectl create configmap` command also supports sourcing the key/value pairs from files. The following table explains the available methods.

Table 9.2 Options for creating config map entries using kubectl create configmap

Option	Description
`--from-literal`	Inserts a key and a literal value into the config map. Example: `--from-literal mykey=myvalue`.
`--from-file`	Inserts the contents of a file into the config map. The behavior depends on the argument that comes after `--from-file`: If only the filename is specified (example: `--from-file myfile.txt`), the base name of the file is used as the key and the entire contents of the file are used as the value. If `key=file` is specified (example: `--from-file mykey=myfile.txt`), the contents of the file are stored under the specified key. If the filename represents a directory, each file contained in the directory is included as a separate entry. The base name of the file is used as the key, and the contents of the file are used as the value. Subdirectories, symbolic links, devices, pipes, and files whose base name isn't a valid config map key are ignored.
`--from-env-file`	Inserts each line of the specified file as a separate entry (example: `--from-env-file myfile.env`). The file must contain lines with the following format: `key=value`

Config maps usually contain more than one entry. To create a config map with multiple entries, you can use multiple arguments `--from-literal`, `--from-file`, and `--from-env-file`, or a combination thereof.

CREATING A CONFIG MAP FROM A YAML MANIFEST

Alternatively, you can create the config map from a YAML manifest file. The following listing shows the contents of an equivalent manifest file named cm.kiada-config.yaml, which is available in the code repository You can create the config map by applying this file using `kubectl apply`.

Listing 9.8 A config map manifest file

```
apiVersion: v1   #A
kind: ConfigMap   #A
metadata:
  name: kiada-config   #B
data:                              #C
  status-message: This status message is set in the kiada-config config map   #C
```

#A This manifest defines a ConfigMap object.
#B The name of this config map
#C Key/value pairs are specified in the data field

LISTING CONFIG MAPS AND DISPLAYING THEIR CONTENTS

Config maps are Kubernetes API objects that live alongside pods, nodes, persistent volumes, and the others you've learned about so far. You can use various kubectl commands to perform CRUD operations on them. For example, you can list config maps with:

```
$ kubectl get cm
```

> **NOTE** The shorthand for `configmaps` is `cm`.

You can display the entries in the config map by instructing kubectl to print its YAML manifest:

```
$ kubectl get cm kiada-config -o yaml
```

> **NOTE** Because YAML fields are output in alphabetical order, you'll find the `data` field at the top of the output.

> **TIP** To display only the key/value pairs, combine `kubectl` with `jq`. For example: `kubectl get cm kiada-config -o json | jq .data`. Display the value of a given entry as follows: `kubectl... | jq '.data["status-message"]'`.

9.2.3 Injecting config map values into environment variables

In the previous section, you created the `kiada-config` config map. Let's use it in the kiada pod.

INJECTING A SINGLE CONFIG MAP ENTRY

To inject the single config map entry into an environment variable, you just need to replace the `value` field in the environment variable definition with the `valueFrom` field and refer to the config map entry. The following listing shows the relevant part of the pod manifest. The full manifest can be found in the file pod.kiada.env-valueFrom.yaml.

Listing 9.9 Setting an environment variable from a config map entry

```
kind: Pod
...
spec:
  containers:
  - name: kiada
    env:   #A
    - name: INITIAL_STATUS_MESSAGE   #A
      valueFrom:   #B
        configMapKeyRef:   #B
          name: kiada-config   #C
          key: status-message   #D
          optional: true   #E
    volumeMounts:
    - ...
```

#A You're setting the environment variable INITIAL_STATUS_MESSAGE.
#B Instead of using a fixed value, the value is obtained from a config map key
#C The name of the config map that contains the value
#D The config map key you're referencing
#E The container may run even If the config map or key is not found

Let me break down the definition of the environment variable that you see in the listing. Instead of specifying a fixed value for the variable, you declare that the value should be obtained from a config map. The name of the config map is specified using the `name` field, whereas the `key` field specifies the key within that map.

Create the pod from this manifest and inspect its environment variables using the following command:

```
$ kubectl exec kiada -- env
...
INITIAL_STATUS_MESSAGE=This status message is set in the kiada-config config map
...
```

The status message should also appear in the pod's response when you access it via `curl` or your browser.

MARKING A REFERENCE OPTIONAL

In the previous listing, the reference to the config map key is marked as `optional` so that the container can be executed even if the config map or key is missing. If that's the case, the environment variable isn't set. You can mark the reference as optional because the Kiada application will run fine without it. You can delete the config map and deploy the pod again to confirm this.

> **NOTE** If a config map or key referenced in the container definition is missing and not marked as optional, the pod will still be scheduled normally. The other containers in the pod are started normally. The container that references the missing config map key is started as soon as you create the config map with the referenced key.

INJECTING THE ENTIRE CONFIG MAP

The `env` field in a container definition takes an array of values, so you can set as many environment variables as you need. However, if you want to set more than a few variables, it can become tedious and error prone to specify them one at a time. Fortunately, by using the `envFrom` instead of the `env` field, you can inject all the entries that are in the config map without having to specify each key individually.

The downside to this approach is that you lose the ability to transform the key to the environment variable name, so the keys must already have the proper form. The only transformation that you can do is to prepend a prefix to each key.

For example, the Kiada application reads the environment variable `INITIAL_STATUS_MESSAGE`, but the key you used in the config map is `status-message`. You must change the config map key to match the expected environment variable name if you want it to be read by the application when you use the `envFrom` field to inject the entire config map into the pod. I've already done this in the cm.kiada-config.envFrom.yaml file. In addition to the INITIAL_STATUS_MESSAGE key, it contains two other keys to demonstrate that they will all be injected into the container's environment.

Replace the config map with the one in the file by running the following command:

```
$ kubectl replace -f cm.kiada-config.envFrom.yaml
```

The pod manifest in the pod.kiada.envFrom.yaml file uses the `envFrom` field to inject the entire config map into the pod. The following listing shows the relevant part of the manifest.

Listing 9.10 Using envFrom to inject the entire config map into environment variables

```
kind: Pod
...
spec:
  containers:
  - name: kiada
    envFrom:   #A
    - configMapRef:   #B
        name: kiada-config   #B
        optional: true   #C
```

#A Using envFrom instead of env to inject the entire config map
#B The name of the config map to inject. Unlike before, no key is specified.
#C The container should run even if the config map does not exist

Instead of specifying both the config map name and the key as in the previous example, only the config map name is specified. If you create the pod from this manifest and inspect its environment variables, you'll see that it contains the environment variable `INITIAL_STATUS_MESSAGE` as well as the other two keys defined in the config map.

As before, you can mark the config map reference as `optional`, in which case the container will run even if the config map doesn't exist. By default, this isn't the case. Containers that reference config maps are prevented from starting until the referenced config maps exist.

INJECTING MULTIPLE CONFIG MAPS

Listing 9.10 shows that the `envFrom` field takes an array of values, which means you can combine entries from multiple config maps. If two config maps contain the same key, the last one takes precedence. You can also combine the `envFrom` field with the `env` field if you wish to inject all entries of one config map and particular entries of another.

> **NOTE** When an environment variable is configured in the `env` field, it takes precedence over environment variables set in the `envFrom` field.

PREFIXING KEYS

Regardless of whether you inject a single config map or multiple config maps, you can set an optional `prefix` for each config map. When their entries are injected into the container's environment, the prefix is prepended to each key to yield the environment variable name.

9.2.4 Injecting config map entries into containers as files

Environment variables are typically used to pass small single-line values to the application, while multiline values are usually passed as files. Config map entries can also contain larger blocks of data that can be projected into the container using the special `configMap` volume type.

> **NOTE** The amount of information that can fit in a config map is dictated by etcd, the underlying data store used to store API objects. At this point, the maximum size is on the order of one megabyte.

A `configMap` volume makes the config map entries available as individual files. The process running in the container gets the entry's value by reading the contents of the file. This mechanism is most often used to pass large config files to the container, but can also be used for smaller values, or combined with the `env` or `envFrom` fields to pass large entries as files and others as environment variables.

CREATING CONFIG MAP ENTRIES FROM FILES

In chapter 4, you deployed the kiada pod with an Envoy sidecar that handles TLS traffic for the pod. Because volumes weren't explained at that point, the configuration file, TLS certificate, and private key that Envoy uses were built into the container image. It would be more convenient if these files were stored in a config map and injected into the container. That way you could update them without having to rebuild the image. But since the security considerations of these files are different, we must handle them differently. Let's focus on the config file first.

You've already learned how to create a config map from a literal value using the `kubectl create configmap` command. This time, instead of creating the config map directly in the cluster, you'll create a YAML manifest for the config map so that you can store it in a version control system alongside your pod manifest.

Instead of writing the manifest file by hand, you can create it using the same `kubectl create` command that you used to create the object directly. The following command creates the YAML file for a config map named `kiada-envoy-config`:

```
$ kubectl create configmap kiada-envoy-config \
      --from-file=envoy.yaml \
      --from-file=dummy.bin \
      --dry-run=client -o yaml > cm.kiada-envoy-config.yaml
```

The config map will contain two entries that come from the files specified in the command. One is the envoy.yaml configuration file, while the other is just some random data to demonstrate that binary data can also be stored in a config map.

When using the `--dry-run` option, the command doesn't create the object in the Kubernetes API server, but only generates the object definition. The `-o yaml` option prints the YAML definition of the object to standard output, which is then redirected to the `cm.kiada-envoy-config.yaml` file. The following listing shows the contents of this file.

Listing 9.11 A config map manifest containing a multi-line value

```
apiVersion: v1
binaryData:
  dummy.bin: n2VW39IEkyQ6Jxo+rdo5J06Vi7cz5...   #A
data:
  envoy.yaml: |   #B
    admin:   #B
      access_log_path: /var/log/envoy.admin.log   #B
      address:   #B
        socket_address:   #B
          protocol: TCP   #B
          address: 0.0.0.0    #B
    ...   #B
kind: ConfigMap
metadata:
  creationTimestamp: null
  name: kiada-envoy-config   #C
```

#A Base64-encoded content of the dummy.bin file.
#B Contents of the envoy.yaml file.
#C The name of this config map.

As you can see in the listing, the binary file ends up in the `binaryData` field, whereas the envoy config file is in the `data` field, which you know from the previous sections. If a config map entry contains non-UTF-8 byte sequences, it must be defined in the `binaryData` field. The `kubectl create configmap` command automatically determines where to put the entry. The values in this field are Base64 encoded, which is how binary values are represented in YAML.

In contrast, the contents of the envoy.yaml file are clearly visible in the `data` field. In YAML, you can specify multi-line values using a pipeline character and appropriate indentation. See the YAML specification on YAML.org for more ways to do this.

Mind your whitespace hygiene when creating config maps

When creating config maps from files, make sure that none of the lines in the file contain trailing whitespace. If any line ends with whitespace, the value of the entry in the manifest is formatted as a quoted string with the newline character escaped. This makes the manifest incredibly hard to read and edit.

Compare the formatting of the two values in the following config map:

```
$ kubectl create configmap whitespace-demo \
  --from-file=envoy.yaml \
  --from-file=envoy-trailingspace.yaml \
  --dry-run=client -o yaml
apiVersion: v1
data:
 envoy-trailingspace.yaml: "admin: \n  access_log_path: /var/log/envoy.admin.log\n    #A
  \ address:\n    socket_address:\n      protocol: TCP\n      address: 0.0.0.0\n    #A
  \     port_value: 9901\nstatic_resources:\n  listeners:\n  - name: listener_0\n...    #A
 envoy.yaml: |    #B
  admin:    #B
    access_log_path: /var/log/envoy.admin.log    #B
    address:    #B
      socket_address:...    #B
```

#A Entry created from a file with trailing whitespace
#B Entry created from a clean file with no trailing whitespace

Notice that the `envoy-trailingspace.yaml` file contains a space at the end of the first line. This causes the config map entry to be presented in a not very human-friendly format. In contrast, the `envoy.yaml` file contains no trailing whitespace and is presented as an unescaped multi-line string, which makes it easy to read and modify in place.

Don't apply the config map manifest file to the Kubernetes cluster yet. You'll first create the pod that refers to the config map. This way you can see what happens when a pod points to a config map that doesn't exist.

USING A CONFIGMAP VOLUME IN A POD

To make config map entries available as files in the container's filesystem, you define a `configMap` volume in the pod and mount it in the container, as shown in the following listing.

Listing 9.12 Defining a configMap volume in a pod: pod.kiada-ssl.configmap-volume.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kiada-ssl
spec:
  volumes:
  - name: envoy-config   #A
    configMap:   #A
      name: kiada-envoy-config   #A
  ...
  containers:
  ...
  - name: envoy
    image: luksa/kiada-ssl-proxy:0.1
    volumeMounts:   #B
    - name: envoy-config   #B
      mountPath: /etc/envoy   #B
  ...
```

#A The definition of the configMap volume
#B The volume is mounted into the container

If you've read the previous two chapters, the definitions of the `volume` and `volumeMount` in this listing should be clear. As you can see, the volume is a `configMap` volume that points to the `kiada-envoy-config` config map, and it's mounted in the `envoy` container under `/etc/envoy`. The volume contains the `envoy.yaml` and `dummy.bin` files that match the keys in the config map.

Create the pod from the manifest file and check its status. Here's what you'll see:

```
$ kubectl get po
NAME        READY   STATUS              RESTARTS   AGE
Kiada-ssl   0/2     ContainerCreating   0          2m
```

Because the pod's `configMap` volume references a config map that doesn't exist, and the reference isn't marked as optional, the container can't run.

MARKING A CONFIGMAP VOLUME AS OPTIONAL

Previously, you learned that if a container contains an environment variable definition that refers to a config map that doesn't exist, the container is prevented from starting until you create that config map. You also learned that this doesn't prevent the other containers from starting. What about the case at hand where the missing config map is referenced in a volume?

Because all of the pod's volumes must be set up before the pod's containers can be started, referencing a missing config map in a volume prevents all the containers in the pod from starting, not just the container in which the volume is mounted. An event is generated indicating the problem. You can display it with the `kubectl describe pod` or `kubectl get events` command, as explained in the previous chapters.

NOTE A `configMap` volume can be marked as optional by adding the line `optional: true` to the volume definition. If a volume is optional and the config map doesn't exist, the volume is not created, and the container is started without mounting the volume.

To enable the pod's containers to start, create the config map by applying the `cm.kiada-envoy-config.yaml` file you created earlier. Use the `kubectl apply` command. After doing this, the pod should start, and you should be able to confirm that both config map entries are exposed as files in the container by listing the contents of the `/etc/envoy` directory as follows:

```
$ kubectl exec kiada-ssl -c envoy -- ls /etc/envoy
dummy.bin
envoy.yaml
```

PROJECTING ONLY SPECIFIC CONFIG MAP ENTRIES

Envoy doesn't need the dummy.bin file, but imagine that it's needed by another container or pod and you can't remove it from the config map. But having this file appear in `/etc/envoy` is not ideal, so let's do something about it.

Fortunately, configMap volumes let you specify which config map entries to project into files. The following listing shows how.

Listing 9.13 Specifying which config map entries to include into a configMap volume

```
  volumes:
  - name: envoy-config
    configMap:
      name: kiada-envoy-config
      items:   #A
      - key: envoy.yaml   #B
        path: envoy.yaml   #B
```

#A Only the following config map entry should be included in the volume.
#B The config map entry value stored under the key "envoy.yaml" should be included in the volume as file "envoy.yaml".

The `items` field specifies the list of config map entries to include in the volume. Each item must specify the `key` and the file name in the `path` field. Entries not listed here aren't included in the volume. In this way, you can have a single config map for a pod with some entries showing up as environment variables and others as files.

SETTING FILE PERMISSIONS IN A CONFIGMAP VOLUME

By default, the file permissions in a `configMap` volume are set to `rw-r--r--` or `0644` in octal form.

> **NOTE** If you aren't familiar with Unix file permissions, `0644` in the octal number system is equivalent to `110-100-100` in the binary system. This is equivalent to `rw-,r--,r--`, meaning that the user who owns the file can read and write it but not execute it (`rw-`), while the group that owns the file and other users can only read it (first and second `r--` sequence, respectively).

You can set the default permissions for the files in a `configMap` volume by setting the `defaultMode` field in the volume definition. In YAML, the field takes either an octal or decimal value. For example, to set permissions to `rwxr-----`, add `defaultMode: 0740` to the `configMap` volume definition. To set permissions for individual files, set the `mode` field next to the item's `key` and `path`.

When specifying file permissions in YAML manifests, make sure you never forget the leading zero, which indicates that the value is in octal form. If you omit the zero, the value will be treated as decimal, which may cause the file to have permissions that you didn't intend.

> **IMPORTANT** When you use `kubectl get -o yaml` to display the YAML definition of a pod, note that the file permissions are represented as decimal values. For example, you'll regularly see the value `420`. This is the decimal equivalent of the octal value `0644`, which is the default file permissions.

Before you move on to setting file permissions and checking them in the container, you should know that the files you find in the `configMap` volume are symbolic links (section 9.2.6 explains why). To see the permissions of the actual file, you must follow these links, because they themselves have no permissions and are always shown as `rwxrwxrwx`.

9.2.5 Updating and deleting config maps

As with most Kubernetes API objects, you can update a config map at any time by modifying the manifest file and reapplying it to the cluster using `kubectl apply`. There's also a quicker way, which you'll mostly use during development.

IN-PLACE EDITING OF API OBJECTS USING KUBECTL EDIT

When you want to make a quick change to an API object, such as a ConfigMap, you can use the `kubectl edit` command. For example, to edit the `kiada-envoy-config` config map, run the following command:

```
$ kubectl edit configmap kiada-envoy-config
```

This opens the object manifest in your default text editor, allowing you to change the object directly. When you close the editor, kubectl posts your changes to the Kubernetes API server.

Configuring kubectl edit to use a different text editor

You can tell `kubectl` to use a text editor of your choice by setting the `KUBE_EDITOR` environment variable. For example, if you'd like to use `nano` for editing Kubernetes resources, execute the following command (or put it into your `~/.bashrc` or an equivalent file):

```
export KUBE_EDITOR="/usr/bin/nano"
```

If the `KUBE_EDITOR` environment variable isn't set, `kubectl edit` falls back to using the default editor, usually configured through the `EDITOR` environment variable.

WHAT HAPPENS WHEN YOU MODIFY A CONFIG MAP

When you update a config map, the files in the `configMap` volume are automatically updated.

> **NOTE** It can take up to a minute for the files in a `configMap` volume to be updated after you change the config map.

Unlike files, environment variables can't be updated while the container is running. However, if the container is restarted for some reason (because it crashed or because it was terminated externally due to a failed liveness probe), Kubernetes will use the new config map values when it sets up the environment variables for the new container. The question is whether you want it to do that at all.

UNDERSTANDING THE CONSEQUENCES OF UPDATING A CONFIG MAP

One of the most important properties of containers is their immutability, which allows you to be sure that there are no differences between multiple instances of the same container (or pod). Shouldn't the config maps from which these instances get their configuration also be immutable?

Let's think about this for a moment. What happens if you change a config map used to inject environment variables into an an application? What if the application is configured via config files, but it doesn't automatically reload them when they are modified? The changes you make to the config map don't affect any of these running application instances. However, if some of these instances are restarted or if you create additional instances, they *will* use the new configuration.

A similar scenario occurs even with applications that can reload their configuration. Kubernetes updates `configMap` volumes asynchronously. Some application instances may see the changes sooner than others. And because the update process may take dozens of seconds, the files in individual pod instances can be out of sync for a considerable amount of time.

In both scenarios, you get instances that are configured differently. This may cause parts of your system to behave differently than the rest. You need to take this into account when deciding whether to allow changes to a config map while it's being used by running pods.

PREVENTING A CONFIG MAP FROM BEING UPDATED

To prevent users from changing the values in a config map, you can mark the config map as immutable, as shown in the following listing.

Listing 9.14 Creating an immutable config map

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-immutable-configmap
data:
  mykey: myvalue
  another-key: another-value
immutable: true                    #A
```

#A This prevents this config map's values from being updated

If someone tries to change the `data` or `binaryData` fields in an immutable config map, the API server will prevent it. This ensures that all pods using this config map use the same configuration values. If you want to run a set of pods with a different configuration, you typically create a new config map and point them to it.

Immutable config maps prevent users from accidentally changing application configuration, but also help improve the performance of your Kubernetes cluster. When a config map is marked as immutable, the Kubelets on the worker nodes that use it don't have to be notified of changes to the ConfigMap object. This reduces the load on the API server.

DELETING A CONFIG MAP

ConfigMap objects can be deleted with the `kubectl delete` command. The running pods that reference the config map continue to run unaffected, but only until their containers must be restarted. If the config map reference in the container definition isn't marked as optional, the container will fail to run.

9.2.6 Understanding how configMap volumes work

Before you start using `configMap` volumes in your own pods, it's important that you understand how they work, or you'll spend a lot of time fighting them.

You might think that when you mount a configMap volume in a directory in the container, Kubernetes merely creates some files in that directory, but things are more complicated than that. There are two caveats that you need to keep in mind. One is how volumes are mounted in general, and the other is how Kubernetes uses symbolic links to ensure that files are updated atomically.

MOUNTING A VOLUME HIDES EXISTING FILES IN THE FILE DIRECTORY

If you mount any volume to a directory in the container's filesystem, the files that are in the container image in that directory can no longer be accessed. For example, if you mount a `configMap` volume into the `/etc` directory, which in a Unix system contains important configuration files, the applications running in the container will only see the files defined in the config map. This means that all other files that should be in `/etc` are no longer present

and the application may not run. However, this problem can be mitigated by using the `subPath` field when mounting the volume.

Imagine you have a `configMap` volume that contains the file `my-app.conf`, and you want to add it to the `/etc` directory without losing any existing files in that directory. Instead of mounting the entire volume in `/etc`, you mount only the specific file using a combination of the `mountPath` and `subPath` fields, as shown in the following listing.

Listing 9.15 Mounting an individual file into a container

```
spec:
  containers:
  - name: my-container
    volumeMounts:
    - name: my-volume
      subPath: my-app.conf                #A
      mountPath: /etc/my-app.conf         #B
```

#A Instead of mounting the entire volume, you mount only the my-app.conf file.
#B You're mounting a single file instead of the entire directory.

To make it easier to understand how all this works, inspect the following figure.

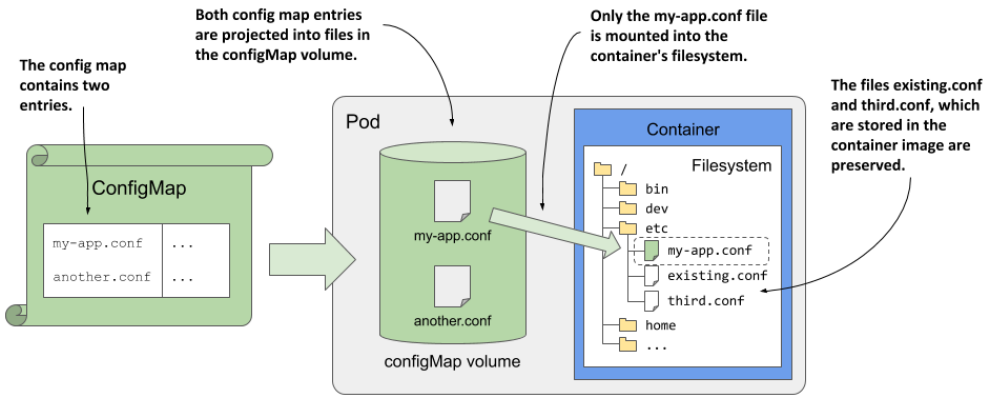

Figure 9.5 Using subPath to mount a single file from the volume

The `subPath` property can be used when mounting any type of volume, but when you use it with a `configMap` volume, please note the following warning:

> **WARNING** If you use the `subPath` field to mount individual files instead of the entire `configMap` volume, the file won't be updated when you modify the config map.

To get around this problem, you can mount the entire volume in another directory and create a symbolic link in the desired location pointing to the file in the other directory. You can create this symbolic link beforehand in the container image itself.

CONFIGMAP VOLUMES USE SYMBOLIC LINKS TO PROVIDE ATOMIC UPDATES

Some applications watch for changes to their configuration files and reload them when this happens. However, if the application is using a large file or multiple files, the application may detect that a file has changed before all file updates are complete. If the application reads the partially updated files, it may not function properly.

To prevent this, Kubernetes ensures that all files in a `configMap` volume are updated atomically, meaning that all updates are done instantaneously. This is achieved with the use of symbolic file links, as you can see if you list all the files in the `/etc/envoy` directory:

```
$ kubectl exec kiada-ssl -c envoy -- ls -lA /etc/envoy
total 4
drwxr-xr-x   ...  ..2020_11_14_11_47_45.728287366   #A
lrwxrwxrwx   ...  ..data -> ..2020_11_14_11_47_45.728287366   #B
lrwxrwxrwx   ...  envoy.yaml -> ..data/envoy.yaml   #C
```

#A Sub-directory that contains the actual files
#B A symbolic link to the subdirectory
#C A symbolic link for each config map entry

As you can see in the listing, the config map entries that are projected as files into the volume are symbolic links that point to file paths within the directory named `..data`, which is also a symbolic link. It points to a directory whose name clearly represents a timestamp. So the file paths that the application reads point to actual files via two successive symbolic links.

This may look unnecessary, but it allows you to update all files atomically. Every time you change the config map, Kubernetes creates a new timestamped directory, writes the files to it, and then associates the `..data` symbolic link with this new directory, replacing all files instantaneously.

> **NOTE** If you use `subPath` in your volume mount definition, this mechanism isn't used. Instead, the file is written directly to the target directory and the file isn't updated when you modify the config map.

9.3 Using Secrets to pass sensitive data to containers

In the previous section, you learned how to store configuration data in ConfigMap objects and make it available to the application via environment variables or files. You may think that you can also use config maps to also store sensitive data such as credentials and encryption keys, but this isn't the best option. For any data that needs to be kept secure, Kubernetes provides another type of object - *Secrets*. They will be covered next.

9.3.1 Introducing Secrets

Secrets are remarkably similar to config maps. Just like config maps, they contain key-value pairs and can be used to inject environment variables and files into containers. So why do we need secrets at all?

Kubernetes supported secrets even before config maps were added. Originally, secrets were not user-friendly when it came to storing plain-text data. For this reason, config maps were then introduced. Over time, both the secrets and config maps evolved to support both types of values. The functions provided by these two types of object converged. If they were

added now, they would certainly be introduced as a single object type. However, because they each evolved gradually, there are some differences between them.

DIFFERENCES IN FIELDS BETWEEN CONFIG MAPS AND SECRETS

The structure of a secret is slightly different from that of a config map. The following table shows the fields in each of the two object types.

Table 9.3 Differences in the structure of secrets and config maps

Secret	ConfigMap	Description
`data`	`binaryData`	A map of key-value pairs. The values are Base64-encoded strings.
`stringData`	`data`	A map of key-value pairs. The values are plain text strings. The stringData field in secrets is write-only.
`immutable`	`immutable`	A boolean value indicating whether the data stored in the object can be updated or not.
`type`	`N/A`	A string indicating the type of secret. Can be any string value, but several built-in types have special requirements.

As you can see in the table, the `data` field in secrets corresponds to the `binaryData` field in config maps. It can contain binary values as Base64-encoded strings. The `stringData` field in secrets is equivalent to the `data` field in config maps and is used to store plain text values. This `stringData` field in secrets is write-only. You can use it to add plaintext values to the secret without having to encode them manually. When you read back the Secret object, any values you added to `stringData` will be included in the `data` field as Base64-encoded strings.

This is different from the behavior of the `data` and `binaryData` fields in config maps. Whatever key-value pair you add to one of these fields will remain in that field when you read the ConfigMap object back from the API.

Like config maps, secrets can be marked immutable by setting the `immutable` field to `true`.

Secrets have a field that config maps do not. The `type` field specifies the type of the secret and is mainly used for programmatic handling of the secret. You can set the type to any value you want, but there are several built-in types with specific semantics.

UNDERSTANDING BUILT-IN SECRET TYPES

When you create a secret and set its type to one of the built-in types, it must meet the requirements defined for that type, because they are used by various Kubernetes components that expect them to contain values in specific formats under specific keys. The following table explains the built-in secret types that exist at the time of writing this.

Table 9.4 Types of secrets

Built-in secret type	Description
`Opaque`	This type of secret can contain secret data stored under arbitrary keys. If you create a secret with no `type` field, an Opaque secret is created.
`bootstrap.kubernetes.io/token`	This type of secret is used for tokens that are used when bootstrapping new cluster nodes.
`kubernetes.io/basic-auth`	This type of secret stores the credentials required for basic authentication. It must contain the `username` and `password` keys.
`kubernetes.io/dockercfg`	This type of secret stores the credentials required for accessing a Docker image registry. It must contain a key called `.dockercfg`, where the value is the contents of the `~/.dockercfg` configuration file used by legacy versions of Docker.
`kubernetes.io/dockerconfigjson`	Like above, this type of secret stores the credentials for accessing a Docker registry, but uses the newer Docker configuration file format. The secret must contain a key called `.dockerconfigjson`. The value must be the contents of the `~/.docker/config.json` file used by Docker.
`kubernetes.io/service-account-token`	This type of secret stores a token that identifies a Kubernetes service account. You'll learn about service accounts and this token in chapter 23.
`kubernetes.io/ssh-auth`	This type of secret stores the private key used for SSH authentication. The private key must be stored under the key `ssh-privatekey` in the secret.
`kubernetes.io/tls`	This type of secrets stores a TLS certificate and the associated private key. They must be stored in the secret under the key `tls.crt` and `tls.key`, respectively.

UNDERSTANDING HOW KUBERNETES STORES SECRETS AND CONFIG MAPS

In addition to the small differences in the names of the fields supported by config maps or secrets, Kubernetes treats them differently. When it comes to secrets, you need to remember that they are handled in specific ways in all Kubernetes components to ensure their security. For example, Kubernetes ensures that the data in a secret is distributed only to the node that runs the pod that needs the secret. Also, secrets on the worker nodes themselves are always stored in memory and never written to physical storage. This makes it less likely that sensitive data will leak out.

For this reason, it's important that you store sensitive data only in secrets and not config maps.

9.3.2 Creating a secret

In section 9.2, you used a config map to inject the configuration file into the Envoy sidecar container. In addition to the file, Envoy also requires a TLS certificate and private key. Because the key represents sensitive data, it should be stored in a secret.

In this section, you'll create a secret to store the certificate and key, and project it into the container's filesystem. With the config, certificate and key files all sourced from outside the container image, you can replace the custom `kiada-ssl-proxy` image with the generic `envoyproxy/envoy` image. This is a considerable improvement, as removing custom images from the system is always a good thing, since you no longer need to maintain them.

First, you'll create the secret. The files for the certificate and private key are provided in the book's code repository, but you can also create them yourself.

CREATING A TLS SECRET

Like for config maps, `kubectl` also provides a command for creating different types of secrets. Since you are creating a secret that will be used by your own application rather than Kubernetes, it doesn't matter whether the secret you create is of type `Opaque` or `kubernetes.io/tls`, as described in table 9.4. However, since you are creating a secret with a TLS certificate and a private key, you should use the built-in secret type `kubernetes.io/tls` to standardize things.

To create the secret, run the following command:

```
$ kubectl create secret tls kiada-tls \       #A
  --cert example-com.crt \                    #B
  --key example-com.key                       #C
```

#A Creating a TLS secret called kiada-tls
#B The path to the certificate file
#C The path to the private key file

This command instructs `kubectl` to create a `tls` secret named `kiada-tls`. The certificate and private key are read from the file `example-com.crt` and `example-com.key`, respectively.

CREATING A GENERIC (OPAQUE) SECRET

Alternatively, you could use kubectl to create a generic secret. The items in the resulting secret would be the same, the only difference would be its type. Here's the command to create the secret:

```
$ kubectl create secret generic kiada-tls \    #A
    --from-file tls.crt=example-com.crt \      #B
    --from-file tls.key=example-com.key        #C
```

#A Creating a generic secret called kiada-tls
#B The contents of the example-com.crt file should be stored under the key tls.crt
#C The contents of the example-com.key file should be stored under the key tls.key

In this case, `kubectl` creates a generic secret. The contents of the `example-com.crt` file are stored under the `tls.crt` key, while the contents of the `example-com.key` file are stored under `tls.key`.

NOTE Like config maps, the maximum size of a secret is approximately 1MB.

CREATING SECRETS FROM YAML MANIFESTS

The kubectl create secret command creates the secret directly in the cluster. Previously, you learned how to create a YAML manifest for a config map. What about secrets?

For obvious reasons, it's not the best idea to create YAML manifests for your secrets and store them in your version control system, as you do with config maps. However, if you need to create a YAML manifest instead of creating the secret directly, you can again use the `kubectl create --dry-run=client -o yaml` trick.

Suppose you want to create a secret YAML manifest containing user credentials under the keys `user` and `pass`. You can use the following command to create the YAML manifest:

```
$ kubectl create secret generic my-credentials \    #A
   --from-literal user=my-username \                 #B
   --from-literal pass=my-password \                 #B
   --dry-run=client -o yaml                          #C
apiVersion: v1
data:
  pass: bXktcGFzc3dvcmQ=                             #D
  user: bXktdXNlcm5hbWU=                             #D
kind: Secret
metadata:
  creationTimestamp: null
  name: my-credentials
```

#A Create a generic secret
#B Store the credentials in keys user and pass
#C Print the YAML manifest instead of posting the secret to the API server
#D Base64-encoded credentials

Creating the manifest using the `kubectl create` trick as shown here is much easier than creating it from scratch and manually entering the Base64-encoded credentials. Alternatively, you could avoid encoding the entries by using the `stringData` field as explained next.

USING THE STRINGDATA FIELD

Since not all sensitive data is in binary form, Kubernetes also allows you to specify plain text values in secrets by using `stringData` instead of the `data` field. The following listing shows how you'd create the same secret that you created in the previous example.

Listing 9.16 Adding plain text entries to a secret using the `stringData` field

```
apiVersion: v1
kind: Secret
stringData:                  #A
  user: my-username          #B
  pass: my-password          #B
```

#A The stringData is used to enter plain-text values without encoding them
#B These credentials aren't encoded using Base64 encoding

The `stringData` field is write-only and can only be used to set values. If you create this secret and read it back with `kubectl get -o yaml`, the `stringData` field is no longer present. Instead, any entries you specified in it will be displayed in the `data` field as Base64-encoded values.

> **TIP** Since entries in a secret are always represented as Base64-encoded values, working with secrets (especially reading them) is not as human-friendly as working with config maps, so use config maps wherever possible. But never sacrifice security for the sake of comfort.

Okay, let's return to the TLS secret you created earlier. Let's use it in a pod.

9.3.3 Using secrets in containers

As explained earlier, you can use secrets in containers the same way you use config maps - you can use them to set environment variables or create files in the container's filesystem. Let's look at the latter first.

USING A SECRET VOLUME TO PROJECT SECRET ENTRIES INTO FILES

In one of the previous sections, you created a secret called kiada-tls. Now you will project the two entries it contains into files using a `secret` volume. A `secret` volume is analogous to the `configMap` volume used before, but points to a secret instead of a config map.

To project the TLS certificate and private key into the `envoy` container of the `kiada-ssl` pod, you need to define a new `volume` and a new `volumeMount`, as shown in the next listing.

Listing 9.17 Using a secret volume in a pod: pod.kiada-ssl.secret-volume.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kiada-ssl
spec:
  volumes:
  - name: cert-and-key   #A
    secret:   #A
      secretName: kiada-tls   #A
      items:   #B
      - key: tls.crt   #B
        path: example-com.crt   #B
      - key: tls.key   #B
        path: example-com.key   #B
        mode: 0600   #C
  ...
  containers:
  - name: kiada
    ...
  - name: envoy
    image: envoyproxy/envoy:v1.14.1
    volumeMounts:   #D
    - name: cert-and-key   #D
      mountPath: /etc/certs   #D
      readOnly: true   #D
    ...
    ports:
    ...
```

#A This defines a secret volume that projects the entries of the kiada-tls secret into files.
#B The keys in the secret need to be mapped to the correct filenames that are configured in the Envoy configuration file.
#C The example-com.key file's permissions are set to 0600 or rw——.
#D The secret volume is mounted in /etc/certs

If you've read section 9.2 on config maps, the definitions of the `volume` and `volumeMount` in this listing should be straightforward since they contain the same fields as you'd find if you were using a config map. The only two differences are that the volume type is `secret` instead of `configMap`, and that the name of the referenced secret is specified in the `secretName` field, whereas in a `configMap` volume definition the config map is specified in the `name` field.

> **NOTE** As with `configMap` volumes, you can set the file permissions on `secret` volumes with the `defaultMode` and `mode` fields. Also, you can set the `optional` field to `true` if you want the pod to start even if the referenced secret doesn't exist. If you omit the field, the pod won't start until you create the secret.

Given the sensitive nature of the example-com.key file, the `mode` field is used to set the file permissions to `0600` or `rw-------`. The file example-com.crt is given the default permissions.

To illustrate the pod, its secret volume and the referenced secret and its entries, see the following figure.

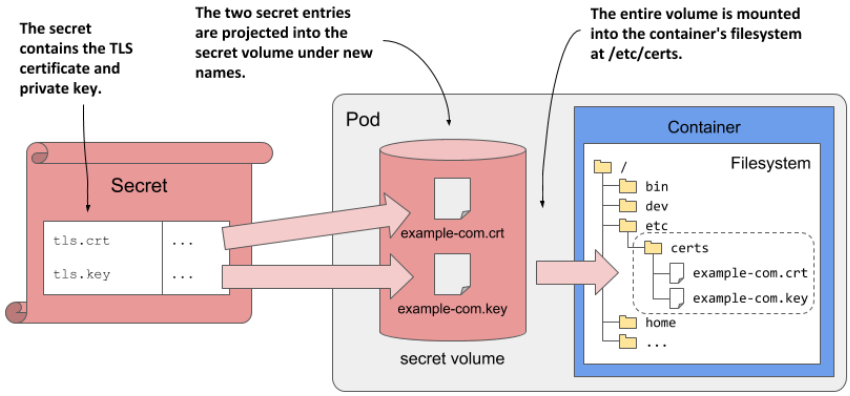

Figure 9.6 Projecting a secret's entries into the container's filesystem via a secret volume

READING THE FILES IN THE SECRET VOLUME

After you deploy the pod from the previous listing, you can use the following command to inspect the certificate file in the secret volume:

```
$ kubectl exec kiada-ssl -c envoy -- cat /etc/certs/example-com.crt
-----BEGIN CERTIFICATE-----
MIIFkzCCA3ugAwIBAgIUQhQiuFP7vEplCBG167ICGxg4q0EwDQYJKoZIhvcNAQEL
BQAwWDELMAkGA1UEBhMCWFgxFTATBgNVBAcMDERlZmF1bHQgQ2l0eTEcMBoGA1UE
...
```

As you can see, when you project the entries of a secret into a container via a `secret` volume, the projected file is not Base64-encoded. The application doesn't need to decode the file. The same is true if the secret entries are injected into environment variables.

> **NOTE** The files in a secret volume are stored in an in-memory filesystem (tmpfs), so they are less likely to be compromised.

INJECTING SECRETS INTO ENVIRONMENT VARIABLES

As with config maps, you can also inject secrets into the container's environment variables. For example, you can inject the TLS certificate into the `TLS_CERT` environment variable as if the certificate were stored in a config map. The following listing shows how you'd do this.

Listing 9.18 Exposing a secret's key-value pair as an environment variable

```
  containers:
  - name: my-container
    env:
    - name: TLS_CERT
      valueFrom:                  #A
        secretKeyRef:             #A
          name: kiada-tls         #B
          key: tls.crt            #C
```

#A The value is obtained from a secret.
#B The name of the secret that contains the key.
#C The key that contains the value.

This is not unlike setting the `INITIAL_STATUS_MESSAGE` environment variable, except that you're now referring to a secret by using `secretKeyRef` instead of `configMapKeyRef`.

Instead of using `env.valueFrom`, you could also use `envFrom` to inject the entire secret instead of injecting its entries individually, as you did in section 9.2.3. Instead of `configMapRef`, you'd use the `secretRef` field.

SHOULD YOU INJECT SECRETS INTO ENVIRONMENT VARIABLES?

As you can see, injecting secrets into environment variables is no different from injecting config maps. But even if Kubernetes allows you to expose secrets in this way, it may not be the best idea, as it can pose a security risk. Applications typically output environment variables in error reports or even write them to the application log at startup, which can inadvertently expose secrets if you inject them into environment variables. Also, child processes inherit all environment variables from the parent process. So, if your application calls a third-party child process, you don't know where your secrets end up.

> **TIP** Instead of injecting secrets into environment variables, project them into the container as files in a secret volume. This reduces the likelihood that the secrets will be inadvertently exposed to attackers.

9.4 Passing pod metadata to the application via the Downward API

So far in this chapter, you've learned how to pass configuration data to your application. But that data was always static. The values were known before you deployed the pod, and if you deployed multiple pod instances, they would all use the same values.

But what about data that isn't known until the pod is created and scheduled to a cluster node, such as the IP of the pod, the name of the cluster node, or even the name of the pod itself? And what about data that is already specified elsewhere in the pod manifest, such as the amount of CPU and memory that is allocated to the container? Good engineers never want to repeat themselves.

> **NOTE** You'll learn how to specify the container's CPU and memory limits in chapter 20.

9.4.1 Introducing the Downward API

In the remaining chapters of the book, you'll learn about many additional configuration options that you can set in your pods. There are cases where you need to pass the same information to your application. You could repeat this information when defining the container's environment variable, but a better option is to use what's called the Kubernetes *Downward API*, which allows you to expose pod and container metadata via environment variables or files.

UNDERSTANDING WHAT THE DOWNWARD API IS

The Downward API isn't a REST endpoint that your app needs to call to get the data. It's simply a way to inject values from the pod's `metadata`, `spec`, or `status` fields down into the container. Hence the name. An illustration of the Downward API is shown in the following figure.

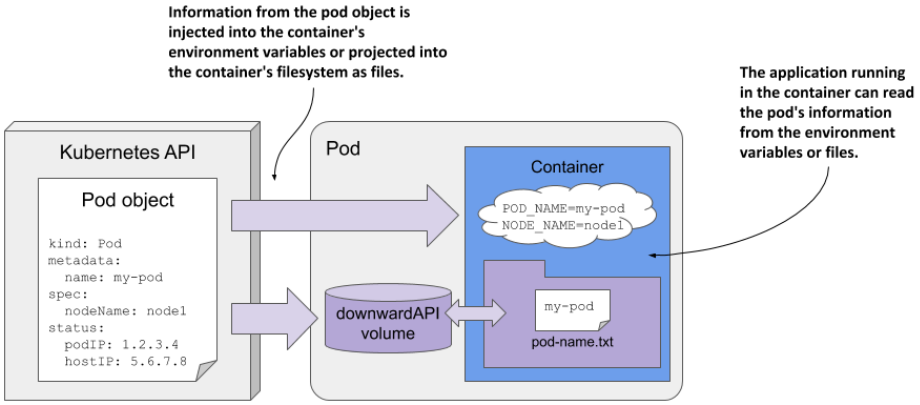

Figure 9.7 The Downward API exposes pod metadata through environment variables or files.

As you can see, this is no different from setting environment variables or projecting files from config maps and secrets, except that the values come from the pod object itself.

UNDERSTANDING HOW THE METADATA IS INJECTED

Earlier in the chapter, you learned that you initialize environment variables from external sources using the `valueFrom` field. To get the value from a config map, use the `configMapKeyRef` field, and to get it from a secret, use `secretKeyRef`. To instead use the Downward API to source the value from the pod object itself, use either the `fieldRef` or the `resourceFieldRef` field, depending on what information you want to inject. The former is used to refer to the pod's general metadata, whereas the latter is used to refer to the container's compute resource constraints.

Alternatively, you can project the pod's metadata as files into the container's filesystem by adding a `downwardAPI` volume to the pod, just as you'd add a `configMap` or `secret` volume. You'll learn how to do this soon, but first let's see what information you can inject.

UNDERSTANDING WHAT METADATA CAN BE INJECTED

You can't use the Downward API to inject any field from the pod object. Only certain fields are supported. The following table shows the fields you can inject via `fieldRef`, and whether they can only be exposed via environment variables, files, or both.

Table 9.5 Downward API fields injected via the fieldRef field

Field	Description	Allowed in env	Allowed in volume
`metadata.name`	The pod's name.	Yes	Yes
`metadata.namespace`	The pod's namespace.	Yes	Yes
`metadata.uid`	The pod's UID.	Yes	Yes
`metadata.labels`	All the pod's labels, one label per line, formatted as `key="value"`.	No	Yes
`metadata.labels['key']`	The value of the specified label.	Yes	Yes
`metadata.annotations`	All the pod's annotations, one per line, formatted as `key="value"`.	No	Yes
`metadata.annotations['key']`	The value of the specified annotation.	Yes	Yes
`spec.nodeName`	The name of the worker node the pod runs on.	Yes	No
`spec.serviceAccountName`	The name of the pod's service account.	Yes	No
`status.podIP`	The pod's IP address.	Yes	No
`status.hostIP`	The worker node's IP address.	Yes	No

You may not know most of these fields yet, but you will in the remaining chapters of this book. As you can see, some fields can only be injected into environment variables, whereas others can only be projected into files. Some allow doing both.

Information about the container's computational resource constraints is injected via the `resourceFieldRef` field. They can all be injected into environment variables and via a downwardAPI volume. The following table lists them.

Table 9.6 Downward API resource fields injected via the resourceFieldRef field

Resource field	Description	Allowed in env	Allowed in vol
`requests.cpu`	The container's CPU request.	Yes	Yes
`requests.memory`	The container's memory request.	Yes	Yes
`requests.ephemeral-storage`	The container's ephemeral storage request.	Yes	Yes
`limits.cpu`	The container's CPU limit.	Yes	Yes
`limits.memory`	The container's memory limit.	Yes	Yes
`limits.ephemeral-storage`	The container's ephemeral storage limit.	Yes	Yes

You'll learn what resource requests and limits are in chapter 20, which explains how to constrain the compute resources available to a container.

The book's code repository contains the file `pod.downward-api-test.yaml`, which defines a pod that uses the Downward API to inject each supported field into both environment variables and files. You can deploy the pod and then look in its container log to see what was injected.

A practical example of using the Downward API in the Kiada application is presented next.

9.4.2 Injecting pod metadata into environment variables

At the beginning of this chapter, a new version of the Kiada application was introduced. The application now includes the pod and node names and their IP addresses in the HTTP response. You'll make this information available to the application through the Downward API.

INJECTING POD OBJECT FIELDS

The application expects the pod's name and IP, as well as the node name and IP, to be passed to it via the environment variables `POD_NAME`, `POD_IP`, `NODE_NAME`, and `NODE_IP`, respectively. The following listing uses the Downward API to set them.

Listing 9.19 Using the Downward API in environment variables: kiada-1.2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kiada-ssl
spec:
  ...
  containers:
  - name: kiada
    image: luksa/kiada:0.4
    env:                                #A
    - name: POD_NAME                    #B
      valueFrom:                        #B
        fieldRef:                       #B
          fieldPath: metadata.name      #B
    - name: POD_IP                      #C
      valueFrom:                        #C
        fieldRef:                       #C
          fieldPath: status.podIP       #C
    - name: NODE_NAME                   #D
      valueFrom:                        #D
        fieldRef:                       #D
          fieldPath: spec.nodeName      #D
    - name: NODE_IP                     #E
      valueFrom:                        #E
        fieldRef:                       #E
          fieldPath: status.hostIP      #E
    ports:
    ...
```

#A These are the environment variables for this container.
#B The POD_NAME environment variable gets its value from the Pod object's metadata.name field.
#C The POD_IP environment variable gets the value from the Pod object's status.podIP field.
#D The NODE_NAME variable gets the value from the spec.nodeName field.
#E The NODE_IP variable is initialized from the status.hostIP field.

After you create this pod, you can examine its log using `kubectl logs`. The application prints the values of the three environment variables at startup. You can also send a request to the application and you should get a response like the following:

```
Request processed by Kiada 0.4 running in pod "kiada-ssl" on node "kind-worker".
Pod hostname: kiada-ssl; Pod IP: 10.244.2.15; Node IP: 172.18.0.4. Client IP:
       ::ffff:127.0.0.1.
```

Compare the values in the response with the field values in the YAML definition of the Pod object by running the command `kubectl get po kiada-ssl -o yaml`. Alternatively, you can compare them with the output of the following commands:

```
$ kubectl get po kiada-ssl -o wide
NAME    READY   STATUS    RESTARTS   AGE     IP            NODE          ...
kiada   1/1     Running   0          7m41s   10.244.2.15   kind-worker   ...

$ kubectl get node kind-worker -o wide
NAME          STATUS   ROLES    AGE   VERSION   INTERNAL-IP    ...
kind-worker   Ready    <none>   26h   v1.19.1   172.18.0.4     ...
```

You can also inspect the container's environment by running `kubectl exec kiada-ssl -- env`.

INJECTING CONTAINER RESOURCE FIELDS

Even if you haven't yet learned how to constrain the compute resources available to a container, let's take a quick look at how to pass those constraints to the application when it needs them.

Chapter 20 explains how to set the number of CPU cores and the amount of memory a container may consume. These settings are called CPU and memory resource *limits*. Kubernetes ensures that the container can't use more than the allocated amount.

Some applications need to know how much CPU time and memory they have been given to run optimally within the given constraints. That's another thing the Downward API is for. The following listing shows how to expose the CPU and memory limits in environment variables.

Listing 9.20 Pod with a `downwardAPI` volume: downward-api-volume.yaml

```
  env:
    - name: MAX_CPU_CORES              #A
      valueFrom:                       #A
        resourceFieldRef:              #A
          resource: limits.cpu         #A
    - name: MAX_MEMORY_KB              #B
      valueFrom:                       #B
        resourceFieldRef:              #B
          resource: limits.memory      #B
          divisor: 1k                  #B
```

#A The MAX_CPU_CORES environment variable will contain the CPU resource limit.
#B The MAX_MEMORY_KB variable will contain the memory limit in kilobytes.

To inject container resource fields, the field `resourceFieldRef` is used. The `resource` field specifies the resource value that is injected.

Each `resourceFieldRef` can also specify a `divisor`. It specifies which unit to use for the value. In the listing, the `divisor` is set to `1k`. This means that the memory limit value is divided by 1000 and the result is then stored in the environment variable. So, the memory limit value in the environment variable will use kilobytes as the unit. If you don't specify a divisor, as is the case in the `MAX_CPU_CORES` variable definition in the listing, the value defaults to 1.

The divisor for memory limits/requests can be `1` (byte), `1k` (kilobyte) or `1Ki` (kibibyte), `1M` (megabyte) or `1Mi` (mebibyte), and so on. The default divisor for CPU is `1`, which is a whole core, but you can also set it to `1m`, which is a milli core or a thousandth of a core.

Because environment variables are defined within a container definition, the resource constraints of the enclosing container are used by default. In cases where a container needs to know the resource limits of another container in the pod, you can specify the name of the other container using the `containerName` field within `resourceFieldRef`.

9.4.3 Using a downwardAPI volume to expose pod metadata as files

As with config maps and secrets, pod metadata can also be projected as files into the container's filesystem using the `downwardAPI` volume type.

Suppose you want to expose the name of the pod in the `/pod-metadata/pod-name` file inside the container. The following listing shows the `volume` and `volumeMount` definitions you'd add to the pod.

Listing 9.21 Injecting pod metadata into the container's filesystem

```
...
  volumes:                              #A
  - name: pod-meta                      #A
    downwardAPI:                        #A
      items:                            #B
      - path: pod-name.txt              #B
        fieldRef:                       #B
          fieldPath: metadata.name      #B
  containers:
  - name: foo
    ...
    volumeMounts:                       #C
    - name: pod-meta                    #C
      mountPath: /pod-metadata          #C
```

#A This defines a downwardAPI volume with the name pod-meta.
#B A single file will appear in the volume. The name of the file is pod-name.txt and it contains the name of the pod.
#C The volume is mounted into the /pod-metadata path in the container.

The pod manifest in the listing contains a single volume of type `downwardAPI`. The volume definition contains a single file named `pod-name.txt`, which contains the name of the pod read from the `metadata.name` field of the pod object. This volume is mounted in the container's filesystem at `/pod-metadata`.

As with environment variables, each item in a `downwardAPI` volume definition uses either `fieldRef` to refer to the pod object's fields, or `resourceFieldRef` to refer to the container's resource fields. For resource fields, the `containerName` field must be specified because volumes are defined at the pod level and it isn't obvious which container's resources are being referenced. As with environment variables, a `divisor` can be specified to convert the value into the expected unit.

As with `configMap` and `secret` volumes, you can set the default file permissions using the `defaultMode` field or per-file using the `mode` field, as explained earlier.

9.5 Using projected volumes to combine volumes into one

In this chapter, you learned how to use three special volume types to inject values from config maps, secrets, and the Pod object itself. Unless you use the `subPath` field in your `volumeMount` definition, you can't inject the files from these different sources, or even multiple sources of the same type, into the same file directory. For example, you can't combine the keys from different secrets into a single volume and mount them into a single file directory. While the `subPath` field allows you to inject individual files from multiple volumes, it isn't the final solution because it prevents the files from being updated when the source values change.

If you need to populate a single volume from multiple sources, you can use the `projected` volume type.

9.5.1 Introducing the projected volume type

Projected volumes allow you to combine information from multiple config maps, secrets, and the Downward API into a single pod volume that you can then mount in the pod's containers. They behave exactly like the `configMap`, `secret`, and `downwardAPI` volumes you learned about in the previous sections of this chapter. They provide the same features and are configured in almost the same way as the other volume types.

The following figure shows a visualization of a projected volume.

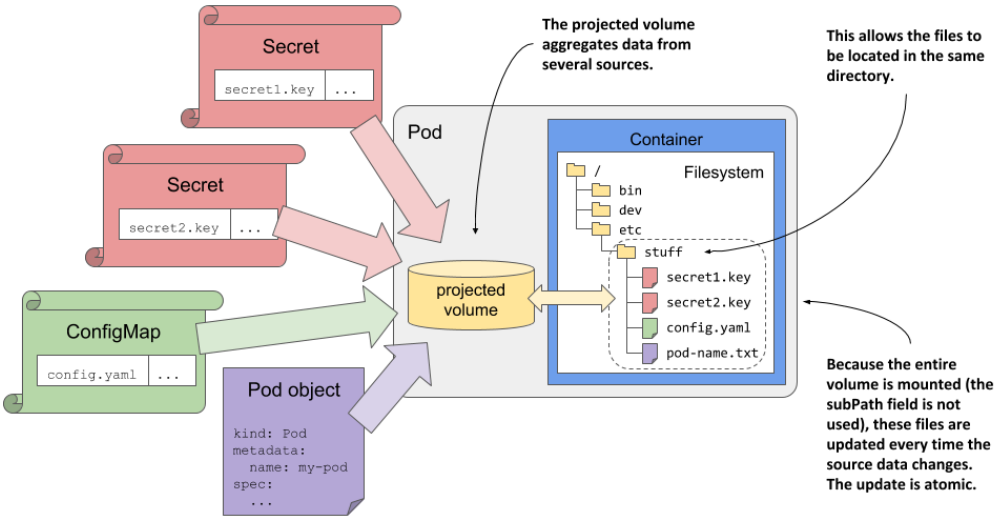

Figure 9.8 Using a projected volume with several sources

In addition to the three volume types described earlier, you can also use projected volumes to expose the service account token to your application. You'll learn what those are in chapter 23.

9.5.2 Using a projected volume in a pod

In the final exercise of this chapter, you'll modify the `kiada-ssl` pod to use a single `projected` volume in the `envoy` container. The previous version of the pod used a `configMap` volume mounted in `/etc/envoy` to inject the `envoy.yaml` config file and a `secret` volume mounted in `/etc/certs` to inject the TLS certificate and key files. You'll now replace these two volumes with a single `projected` volume. This will allow you to keep all three files in the same directory (`/etc/envoy`).

First, you need to change the TLS certificate paths in the `envoy.yaml` configuration file inside the `kiada-envoy-config` config map so that the certificate and key are read from the same directory. After editing, the lines in the config map should look like this:

```
                tls_certificates:
                - certificate_chain:
                    filename: "/etc/envoy/example-com.crt"   #A
                  private_key:
                    filename: "/etc/envoy/example-com.key"   #B
```

#A This used to be "/etc/certs/example-com.crt"
#B This used to be "/etc/certs/example-com.key"

You can find the pod manifest with the projected volume in the file pod.kiada-ssl.projected-volume.yaml. The relevant parts are shown in the next listing.

Listing 9.22 Using a projected volume instead of a configMap and secret volume

```
apiVersion: v1
kind: Pod
metadata:
  name: kiada-ssl
spec:
  volumes:
  - name: etc-envoy   #A
    projected:   #A
      sources:   #A
      - configMap:   #B
          name: kiada-envoy-config   B
      - secret:   #C
          name: kiada-tls   #C
          items:   #C
          - key: tls.crt   #C
            path: example-com.crt   #C
          - key: tls.key   #C
            path: example-com.key   #C
            mode: 0600   #D
  containers:
  - name: kiada
    image: luksa/kiada:1.2
    env:
    ...
  - name: envoy
    image: envoyproxy/envoy:v1.14.1
    volumeMounts:   #E
    - name: etc-envoy   #E
      mountPath: /etc/envoy   #E
      readOnly: true   #E
    ports:
    ...
```

#A A single projected volume is defined.
#B The first volume source is the config map.
#C The second source is the secret.
#D Set restricted file permissions for the private key file.
#E The volume is mounted into the envoy container at /etc/envoy.

The listing shows that a single `projected` volume named `etc-envoy` is defined in the pod. Two sources are used for this volume. The first is the `kiada-envoy-config` config map. All entries in this config map become files in the projected volume. The second source is the `kiada-tls` secret. Two of its entries become files in the projected volume - the value of the `tls.crt` key

becomes file `example-com.crt`, whereas the value of the `tls.key` key becomes file `example-com.key` in the volume. The volume is mounted in read-only mode in the `envoy` container at `/etc/envoy`.

As you can see, the source definitions in the `projected` volume are not much different from the `configMap` and `secret` volumes you created in the previous sections. Therefore, further explanation of the projected volumes is unnecessary. Everything you learned about the other volumes also applies to this new volume type.

9.6 Summary

This wraps up this chapter on how to pass configuration data to containers. You've learned that:

- The default command and arguments defined in the container image can be overridden in the pod manifest.
- Environment variables for each container can also be set in the pod manifest. Their values can be hardcoded in the manifest or can come from other Kubernetes API objects.
- Config maps are Kubernetes API objects used to store configuration data in the form of key/value pairs. Secrets are another similar type of object used to store sensitive data such as credentials, certificates, and authentication keys.
- Entries of both config maps and secrets can be exposed within a container as environment variables or as files via the `configMap` and `secret` volumes.
- Config maps and other API objects can be edited in place using the `kubectl edit` command.
- The Downward API provides a way to expose the pod metadata to the application running within it. Like config maps and secrets, this data can be injected into environment variables or files.
- Projected volumes can be used to combine multiple volumes of possibly different types into a composite volume that is mounted into a single directory, rather than being forced to mount each individual volume into its own directory.

You've now seen that an application deployed in Kubernetes may require many additional objects. If you are deploying many applications in the same cluster, you need organize them so that everyone can see what fits where. In the next chapter, you'll learn how to do just that.

10

Organizing objects using Namespaces, labels, and selectors

This chapter covers

- Using namespaces to split a physical cluster into virtual clusters
- Organizing objects using labels
- Using label selectors to perform operations on subsets of objects
- Using label selectors to schedule pods onto specific nodes
- Using field selectors to filter objects based on their properties
- Annotating objects with additional non-identifying information

A Kubernetes cluster is usually used by many teams. How should these teams deploy objects to the same cluster and organize them so that one team doesn't accidentally modify the objects created by other teams?

And how can a large team deploying hundreds of microservices organize them so that each team member, even if new to the team, can quickly see where each object belongs and what its role in the system is? For example, to which application does a config map or a secret belong.

These are two different problems. Kubernetes solves the first with the Namespace resource, and the other with object labels. In this chapter, you will learn about both.

10.1 Organizing objects into Namespaces

Imagine that your organization is running a single Kubernetes cluster that's used by multiple engineering teams. Each of these teams deploys the entire Kiada application suite to develop and test it. You want each team to only deal with their own instance of the application suite -

each team only wants to see the objects they've created and not those created by the other teams. This is achieved by creating objects in separate Kubernetes namespaces.

NOTE Namespaces in Kubernetes help organize Kubernetes API objects into non-overlapping groups. They have nothing to do with Linux namespaces, which help isolate processes running in one container from those in another, as you learned in chapter 2.

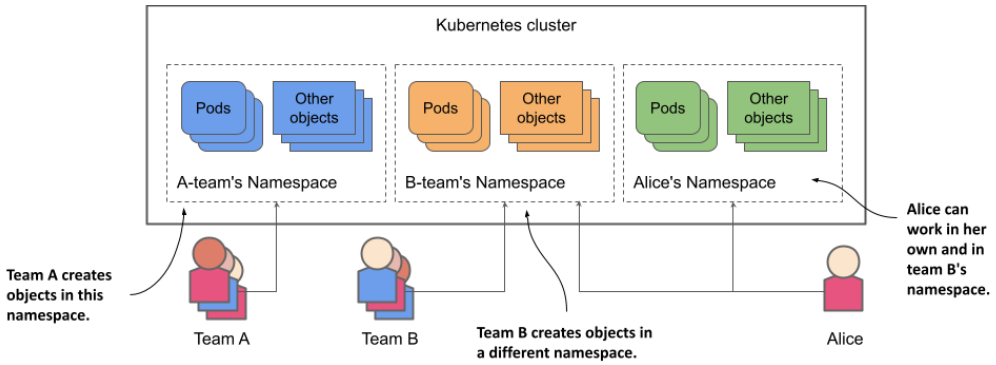

Figure 10.1 Splitting a physical cluster into several virtual clusters by utilizing Kubernetes Namespaces

As shown in the previous figure, you can use namespaces to divide a single physical Kubernetes cluster into many virtual clusters. Instead of everyone creating their objects in a single location, each team gets access to one or more namespaces in which to create their objects. Because namespaces provide a scope for object names, different teams can use the same names for their objects when they create them in their respective namespaces. Some namespaces can be shared between different teams or individual users.

UNDERSTANDING WHEN TO ORGANIZE OBJECTS INTO NAMESPACES

Using multiple namespaces allows you to divide complex systems with numerous components into smaller groups that are managed by different teams. They can also be used to separate objects in a multitenant environment (for example, you can create a separate namespace for each client and deploy the entire application suite for that client in that namespace). As explained earlier, you can also have each team (or team member) deploy in its own namespace.

NOTE Most Kubernetes API object types are namespaced, but a few are not. Pods, ConfigMaps, Secrets, PersistentVolumeClaims, and Events are all namespaced. Nodes, PersistentVolumes, StorageClasses, and Namespaces themselves are not. To see if a resource is namespaced or cluster-scoped, check the `NAMESPACED` column when running `kubectl api-resources`.

Without namespaces, each user of the cluster would have to prefix their object names with a unique prefix or each user would have to use their own Kubernetes cluster.

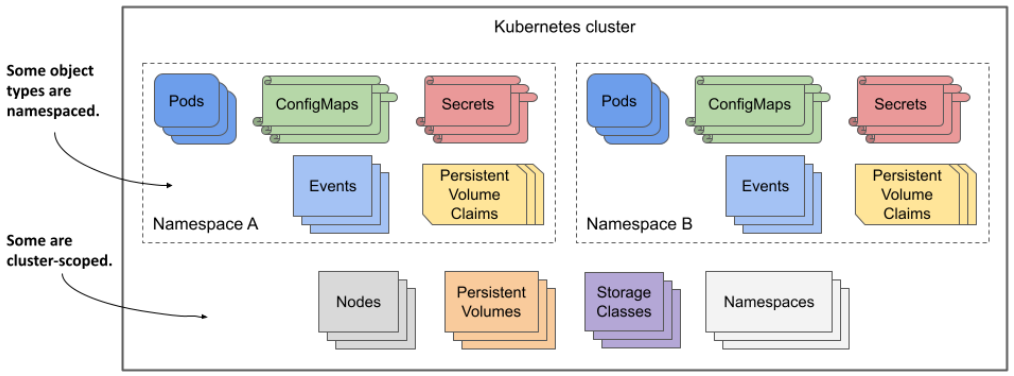

Figure 10.2 Some Kubernetes API types are namespaced, whereas others are cluster-scoped.

As you'll learn in chapter 23, namespaces also provide a scope for user privileges. A user may have permission to manage objects in one namespace but not in others. Similarly, resource quotas, which are also tied to namespaces, are explained in chapter 20.

10.1.1 Listing namespaces and the objects they contain

Every Kubernetes cluster you create contains a few common namespaces. Let's see what they are.

LISTING NAMESPACES

Since each namespace is represented by the *Namespace* object, you can display them with the `kubectl get` command, as you would any other Kubernetes API object. To see the namespaces in your cluster, run the following command:

```
$ kubectl get namespaces
NAME                 STATUS   AGE
default              Active   1h
kube-node-lease      Active   1h
kube-public          Active   1h
kube-system          Active   1h
local-path-storage   Active   1h
```

NOTE The short form for `namespace` is `ns`. You can also list namespaces with `kubectl get ns`.

Up to this point, you've been working in the `default` namespace. Every time you created an object, it was created in that namespace. Similarly, when you listed objects, such as pods, with the `kubectl get` command, the command only displayed the objects in that namespace. You may be wondering if there are pods in the other namespaces. Let's take a look.

NOTE Namespaces prefixed with `kube-` are reserved for Kubernetes system namespaces.

LISTING OBJECTS IN A SPECIFIC NAMESPACE

To list the pods in the `kube-system` namespace, run `kubectl get` with the `--namespace` option as follows:

```
$ kubectl get po --namespace kube-system
NAME                          READY   STATUS    RESTARTS   AGE
coredns-558bd4d5db-4n5zg      1/1     Running   0          1h
coredns-558bd4d5db-tnfws      1/1     Running   0          1h
etcd-kind-control-plane       1/1     Running   0          1h
kindnet-54ks9                 1/1     Running   0          1h
...
```

TIP **You can also use `-n` instead of `--namespace`.**

You'll learn more about these pods later in this book. Don't worry if the pods shown here don't exactly match the ones in your cluster. As the namespace name implies, these are the Kubernetes system pods. By having them in this separate namespace, everything stays neatly nice and clear. If they were all in the default namespace, mixed in with the pods you create yourself, it would be hard to tell what belongs where, and you could accidentally delete system objects.

LISTING OBJECTS ACROSS ALL NAMESPACES

Instead of listing objects in each namespace individually, you can also tell kubectl to list objects in all namespaces. This time, instead of listing pods, let's list all config maps in the cluster:

```
$ kubectl get cm --all-namespaces
NAMESPACE            NAME                                 DATA   AGE
default              kiada-envoy-config                   2      1h
default              kube-root-ca.crt                     1      1h
kube-node-lease      kube-root-ca.crt                     1      1h
kube-public          cluster-info                         1      1h
kube-public          kube-root-ca.crt                     1      1h
...
```

TIP **You can also type `-A` instead of `--all-namespaces`.**

The `--all-namespaces` option is handy when you want to see all objects in the cluster, regardless of namespace, or when you can't remember which namespace an object is in.

10.1.2 Creating namespaces

Now that you know the other namespaces in your cluster, you'll create two new namespaces.

CREATING A NAMESPACE WITH KUBECTL CREATE NAMESPACE

The fastest way to create a namespace is to use the `kubectl create namespace` command. Create a namespace named `kiada-test` as follows:

```
$ kubectl create namespace kiada-test1
namespace/kiada-test1 created
```

> **NOTE** The names of most objects must conform to the naming conventions for DNS subdomain names, as specified in RFC 1123, that is, they may contain only lowercase alphanumeric characters, hyphens, and dots, and must start and end with an alphanumeric character. The same applies to namespaces, but they may not contain dots.

You've just created the namespace `kiada-test1`. You'll now create another one using a different method.

CREATING A NAMESPACE FROM A MANIFEST FILE

As mentioned earlier, Kubernetes namespaces are represented by Namespace objects. As such, you can list them with the `kubectl get` command, as you've already done, but you can also create them from a YAML or JSON manifest file that you post to the Kubernetes API.

Use this method to create another namespace called `kiada-test2`. First, create a file named ns.kiada-test.yaml with the contents of the following listing.

Listing 10.1 A YAML definition of a namespace: ns.kiada-test2.yaml

```
apiVersion: v1
kind: Namespace   #A
metadata:
  name: kiada-test2   #B
```

#A This manifest contains a Namespace object.
#B This is the name of the namespace.

Now, use `kubectl apply` to post the file to the Kubernetes API:

```
$ kubectl apply -f ns.kiada-test.yaml
namespace/kiada-test2 created
```

You will not normally create namespaces this way, but I wanted to remind you that everything in Kubernetes has a corresponding API object that you can create, read, update, and delete by posting a YAML manifest to the API. This is also true for namespaces.

Before you continue, you should run `kubectl get ns` to list all namespaces again to see that your cluster now contains the two namespaces you created.

10.1.3 Managing objects in other namespaces

You've now created two new namespaces: `kiada-test1` and `kiada-test2`, but as mentioned earlier, you're still in the `default` namespace. If you create an object such as a pod without explicitly specifying the namespace, the object is created in the `default` namespace.

CREATING OBJECTS IN A SPECIFIC NAMESPACE

In section 10.1.1, you learned that you can specify the `--namespace` argument (or the shorter `-n` option) to list objects in a particular namespace. You can use the same argument when applying an object manifest to the API.

To create the kiada-ssl pod and its associated config map and secret in the `kiada-test1` namespace, run the following command:

```
$ kubectl apply -f kiada-ssl.yaml -n kiada-test1
pod/kiada-ssl created
configmap/kiada-envoy-config created
secret/kiada-tls created
```

You can now list pods, config maps and secrets in the `kiada-test1` namespace to confirm that these objects were created there and not in the `default` namespace:

```
$ kubectl -n kiada-test1 get po
NAME        READY   STATUS    RESTARTS   AGE
kiada-ssl   2/2     Running   0          1m
```

SPECIFYING THE NAMESPACE IN THE OBJECT MANIFEST

The object manifest can specify the namespace of the object in the `namespace` field in the manifest's `metadata` section. When you apply the manifest with the `kubectl apply` command, the object is created in the specified namespace. You don't need to specify the namespace with the `--namespace` option.

The manifest shown in the following listing contains the same three objects as before, but with the namespace specified in the manifest.

Listing 10.2 Specifying the namespace in the object manifest

```
apiVersion: v1
kind: Pod
metadata:
  name: kiada-ssl
  namespace: kiada-test2    #A
spec:
  volumes: ...
...
```

#A This Pod object specifies the namespace. When you apply the manifest, this Pod is created in the kiada-test2 namespace.

When you apply this manifest with the following command, the pod, config map, and secret are created in the `kiada-test2` namespace:

```
$ kubectl apply -f pod.kiada-ssl.kiada-test2-namespace.yaml
pod/kiada-ssl created
configmap/kiada-envoy-config created
secret/kiada-tls created
```

Notice that you didn't specify the `--namespace` option this time. If you did, the namespace would have to match the namespace specified in the object manifest, or kubectl would display an error like in the following example:

```
$ kubectl apply -f kiada-ssl.kiada-test2-namespace.yaml -n kiada-test1
the namespace from the provided object "kiada-test2" does not match the namespace
      "kiada-test1". You must pass '--namespace=kiada-test2' to perform this
      operation.
```

MAKING KUBECTL DEFAULT TO A DIFFERENT NAMESPACE

In the previous two examples you learned how to create and manage objects in namespaces other than the namespace that kubectl is currently using as the default. You'll use the --namespace option frequently - especially when you want to quickly check what's in another namespace. However, you'll do most of your work in the current namespace.

After you create a new namespace, you'll usually run many commands in it. To make your life easier, you can tell kubectl to switch to that namespace. The current namespace is a property of the current kubectl context, which is configured in the kubeconfig file.

> **NOTE** You learned about the kubeconfig file in chapter 3.

To switch to a different namespace, update the current context. For example, to switch to the `kiada-test1` namespace, run the following command:

```
$ kubectl config set-context --current --namespace kiada-test1
Context "kind-kind" modified.
```

Every kubectl command you run from now on will use the `kiada-test1` namespace. For example, you can now list the pods in this namespace by simply typing `kubectl get po`.

> **TIP** To quickly switch to a different namespace, you can set up the following alias: `alias kns='kubectl config set-context --current --namespace '`. You can then switch between namespaces with `kns some-namespace`. Alternatively, you can install a kubectl plugin that does the same thing. You can find it at https://github.com/ahmetb/kubectx

There's not much more to learn about creating and managing objects in different namespaces. But before you wrap up this section, I need to explain how well Kubernetes isolates workloads running in different namespaces.

10.1.4 Understanding the (lack of) isolation between namespaces

You created several pods in different namespaces so far. You already know how to use the `--all-namespaces` option (or `-A` for short) to list pods across all namespaces, so please do so now:

```
$ kubectl get po -A
NAMESPACE     NAME         READY   STATUS    RESTARTS   AGE
default       kiada-ssl    2/2     Running   0          8h    #A
default       quiz         2/2     Running   0          8h
default       quote        2/2     Running   0          8h
kiada-test1   kiada-ssl    2/2     Running   0          2m    #A
kiada-test2   kiada-ssl    2/2     Running   0          1m    #A
...
```

#A Three pods named kiada-ssl exist in different namespaces

In the output of the command, you should see at least two pods named `kiada-ssl`. One in the `kiada-test1` namespace and the other in the `kiada-test2` namespace. You may also have another pod named `kiada-ssl` in the `default` namespace from the exercises in the previous chapters. In this case, there are three pods in your cluster with the same name, all of which you were able to create without issue thanks to namespaces. Other users of the same cluster could deploy many more of these pods without stepping on each other's toes.

UNDERSTANDING THE RUNTIME ISOLATION BETWEEN PODS IN DIFFERENT NAMESPACES

When users use namespaces in a single physical cluster, it's as if they each use their own virtual cluster. But this is only true up to the point of being able to create objects without running into naming conflicts. The physical cluster nodes are shared by all users in the cluster. This means that the isolation between the their pods is not the same as if they were running on different physical clusters and therefore on different physical nodes.

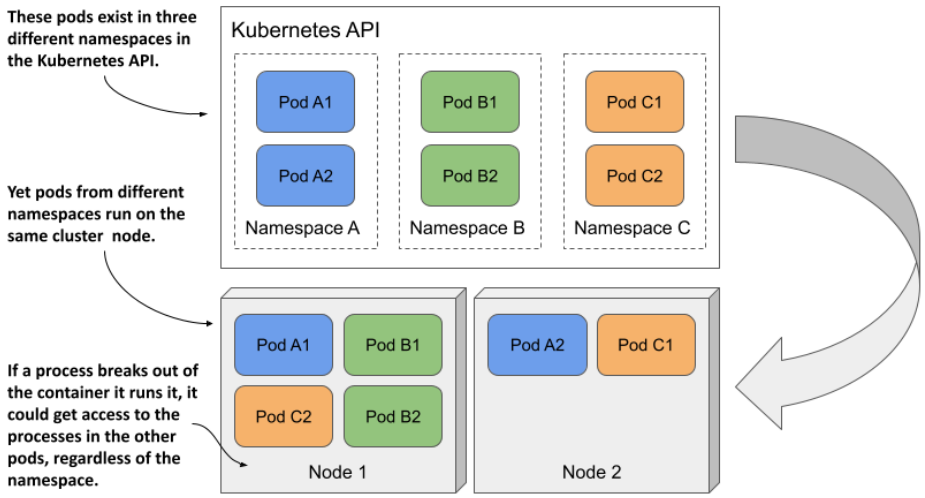

Figure 10.3 Pods from different namespaces may run on the same cluster node.

When two pods created in different namespaces are scheduled to the same cluster node, they both run in the same OS kernel. Although they are isolated from each other with container technologies, an application that breaks out of its container or consumes too much of the

node's resources can affect the operation of the other application. Kubernetes namespaces play no role here.

UNDERSTANDING NETWORK ISOLATION BETWEEN NAMESPACES

Unless explicitly configured to do so, Kubernetes doesn't provide network isolation between applications running in pods in different namespaces. An application running in one namespace can communicate with applications running in other namespaces. By default, there is no network isolation between namespaces. However, you can use the NetworkPolicy object to configure which applications in which namespaces can connect to which applications in other namespaces. You'll learn about this in chapter 25.

USING NAMESPACES TO SEPARATE PRODUCTION, STAGING AND DEVELOPMENT ENVIRONMENTS?

Because namespaces don't provide true isolation, you should not use them to split a single physical Kubernetes cluster into the production, staging, and development environments. Hosting each environment on a separate physical cluster is a much safer approach.

10.1.5 Deleting namespaces

Let's conclude this section on namespaces by deleting the two namespaces you created. When you delete the Namespace object, all the objects you created in that namespace are automatically deleted. You don't need to delete them first.

Delete the `kiada-test2` namespaces as follows:

```
$ kubectl delete ns kiada-test2
namespace "kiada-test2" deleted
```

The command blocks until everything in the namespace and the namespace itself are deleted. But, if you interrupt the command and list the namespaces before the deletion is complete, you'll see that the namespace's status is `Terminating`:

```
$ kubectl get ns
NAME                 STATUS        AGE
default              Active        2h
kiada-test1          Active        2h
kiada-test2          Terminating   2h
...
```

The reason I show this is because you will eventually run the delete command and it will never finish. You'll probably interrupt the command and check the namespace list, as I show here. Then you'll wonder why the namespace termination doesn't complete.

DIAGNOSING WHY NAMESPACE TERMINATION IS STUCK

In short, the reason a namespace can't be deleted is because one or more objects created in it can't be deleted. You may think to yourself, "Oh, I'll list the objects in the namespace with `kubectl get all` to see which object is still there," but that usually doesn't get you any further because kubectl doesn't return any results.

NOTE Remember that the `kubectl get all` command lists only some types of objects. For example, it doesn't list secrets. Even though the command doesn't return anything, this doesn't mean that the namespace is empty.

In most, if not all, cases where I've seen a namespace get stuck this way, the problem was caused by a custom object and its custom controller not processing the object's deletion and removing a finalizer from the object. You'll learn more about finalizers in chapter 15, and about custom objects and controllers in chapter 29.

Here I just want to show you how to figure out which object is causing the namespace to be stuck. Here's a hint: Namespace objects also have a `status` field. While the `kubectl describe` command normally also displays the status of the object, at the time of writing this is not the case for Namespaces. I consider this to be a bug that will likely be fixed at some point. Until then, you can check the status of the namespace as follows:

```
$ kubectl get ns kiada-test2 -o yaml
...
status:
  conditions:
  - lastTransitionTime: "2021-10-10T08:35:11Z"
    message: All resources successfully discovered
    reason: ResourcesDiscovered
    status: "False"
    type: NamespaceDeletionDiscoveryFailure
  - lastTransitionTime: "2021-10-10T08:35:11Z"
    message: All legacy kube types successfully parsed
    reason: ParsedGroupVersions
    status: "False"
    type: NamespaceDeletionGroupVersionParsingFailure
  - lastTransitionTime: "2021-10-10T08:35:11Z"    #A
    message: All content successfully deleted, may be waiting on finalization    #A
    reason: ContentDeleted    #A
    status: "False"    #A
    type: NamespaceDeletionContentFailure    #A
  - lastTransitionTime: "2021-10-10T08:35:11Z"    #B
    message: 'Some resources are remaining: pods. has 1 resource instances'    #B
    reason: SomeResourcesRemain    #B
    status: "True"    #B
    type: NamespaceContentRemaining    #B
  - lastTransitionTime: "2021-10-10T08:35:11Z"    #C
    message: 'Some content in the namespace has finalizers remaining:    #C
              xyz.xyz/xyz-finalizer in 1 resource instances'    #C
    reason: SomeFinalizersRemain    #C
    status: "True"    #C
    type: NamespaceFinalizersRemaining    #C
  phase: Terminating
```

#A All objects in the namespace were marked for deletion, but some haven't been fully deleted yet.
#B One pod remains in the namespace.
#C The pod hasn't been fully deleted because a controller has not removed the specified finalizer from the object.

When you delete the kiada-test2 namespace, you won't see the output in this example. The command output in this example is hypothetical. I forced Kubernetes to produce it to demonstrate what happens when the delete process gets stuck. If you look at the output, you'll see that the objects in the namespace were all successfully marked for deletion, but one pod

remains in the namespace due to a finalizer that was not removed from the pod. Don't worry about finalizers for now. You'll learn about them soon enough.

Before proceeding to the next section, please also delete the `kiada-test1` namespace.

10.2 Organizing pods with labels

In this book, you will build and deploy the full Kiada application suite, which is composed of several services. So far, you've implemented the Kiada, the Quote service, and the Quiz service. These services run in three different pods. Accompanying the pods are other types of objects, like config maps, secrets, persistent volumes, and claims.

As you can imagine, the number of these objects will increase as the book progresses. Before things get out of hand, you need to start organizing these objects so that you and all the other users in your cluster can easily figure out which objects belong to which service.

In other systems that use a microservices architecture, the number of services can exceed 100 or more. Some of these services are replicated, which means that multiple copies of the same pod are deployed. Also, at certain points in time, multiple versions of a service are running simultaneously. This results in hundreds or even thousands of pods in the system.

Imagine you, too, start replicating and running multiple releases of the pods in your Kiada suite. For example, suppose you are running both the stable and canary release of the Kiada service.

> **DEFINITION** A canary release is a deployment pattern where you deploy a new version of an application alongside the stable version, and direct only a small portion of requests to the new version to see how it behaves before rolling it out to all users. This prevents a bad release from being made available to too many users.

You run three replicas of the stable Kiada version, and one canary instance. Similarly, you run three instances of the stable release of the Quote service, along with a canary release of the Quote service. You run a single, stable release of the Quiz service. All these pods are shown in the following figure.

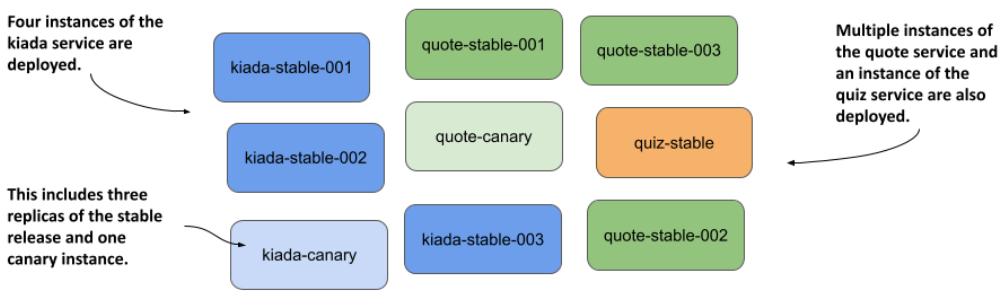

Figure 10.4 Unorganized pods of the Kiada application suite

Even with only nine pods in the system, the system diagram is challenging to understand. And it doesn't even show any of the other API objects required by the pods. It's obvious that you need to organize them into smaller groups. You could split these three services into three namespaces, but that's not the real purpose of namespaces. A more appropriate mechanism for this case is object *labels*.

10.2.1 Introducing labels

Labels are an incredibly powerful yet simple feature for organizing Kubernetes API objects. A label is a key-value pair you attach to an object that allows any user of the cluster to identify the object's role in the system. Both the key and the value are simple strings that you can specify as you wish. An object can have more than one label, but the label keys must be unique within that object. You normally add labels to objects when you create them, but you can also change an object's labels later.

USING LABELS TO PROVIDE ADDITIONAL INFORMATION ABOUT AN OBJECT

To illustrate the benefits of adding labels to objects, let's take the pods shown in figure 10.4. These pods run three different services - the Kiada service, the Quote, and the Quiz service. The Kiada and Quote services are each divided into stable and canary releases. While it is true that you can tell which service and release type each pod belongs to by its name, that's only because I made sure to name the pods that way. However, it's not always practical to include all this information in the name of the object.

Instead, we can store this information in pod labels. Kubernetes does not care what labels you add to your objects. You can choose the keys and values however you want. In the case at hand, the following two labels make sense:

- The "app" label indicates to which application the pod belongs.
- The "rel" label indicates whether the pod is running the stable or canary release of the application.

As you can see in the following figure, the value of the "app" label is set to "kiada" in all three kiada-stable-xxx and the kiada-canary pod, since all these pods are running the Kiada application. The "rel" label differs between the pods running the stable release and the pod running the canary release.

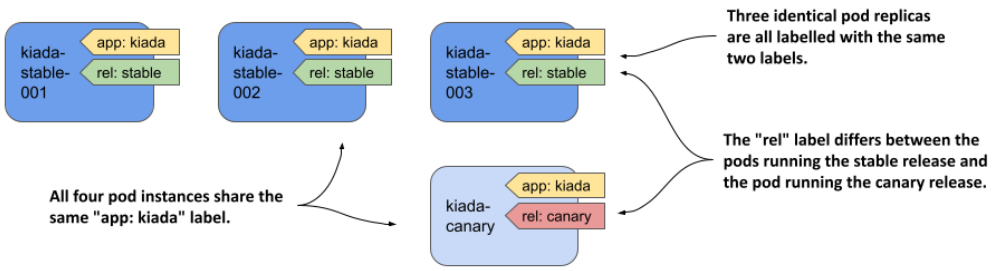

Figure 10.5 Labelling pods with the app and rel label

The illustration shows only the kiada pods, but imagine adding the same two labels to the other pods as well. With these labels, users that come across these pods can easily tell what application and what kind of release is running in the pod.

UNDERSTANDING HOW LABELS KEEP OBJECTS ORGANIZED

If you haven't yet realized the value of adding labels to an object, consider that by adding the "app" and "rel" labels, you've essentially organized your pods in two dimensions (horizontally by application and vertically by release), as shown in the next figure.

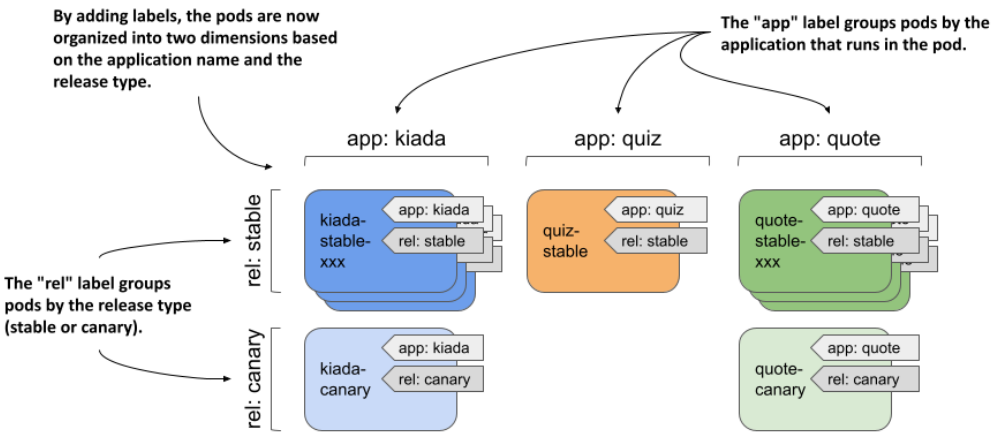

Figure 10.6 All the pods of the Kiada suite organized by two criteria

This may seem abstract until you see how these labels make it easier to manage these pods with kubectl, so let's get practical.

10.2.2 Attaching labels to pods

The book's code archive contains an object manifest file with all the pods from the previous example. All the stable pods are already labelled, but the canary pods aren't. You'll label them manually.

SETTING UP THE EXERCISE

To get started, create a new namespace called kiada, switch to that namespace, and deploy the objects as follows. To create the namespace, run:

```
$ kubectl create namespace kiada
namespace/kiada created
```

To switch to the namespace, run:

```
$ kubectl config set-context --current --namespace kiada
Context "kind-kind" modified.
```

To deploy the pods and related objects, run:

```
$ kubectl apply -f many-pods.yaml
pod/kiada-stable-001 created
pod/kiada-stable-002 created
pod/kiada-stable-003 created
pod/kiada-canary created
configmap/kiada-envoy-config created
secret/kiada-tls created
pod/quote-stable-001 created
pod/quote-stable-002 created
pod/quote-stable-003 created
pod/quote-canary created
pod/quiz-stable created
persistentvolumeclaim/quiz-data created
```

As you can see, the manifest contains several objects of different kinds. Their labels will be crucial if you want to make sense of them.

DEFINING LABELS IN OBJECT MANIFESTS

Examine the manifest file. At the top, you'll see a pod named `kiada-stable-001`. In the `metadata` section, you'll find the `labels` field with the labels `app` and `rel`, as shown in the following listing.

Listing 10.3 A pod with labels

```
apiVersion: v1
kind: Pod
metadata:
  name: kiada-stable-001
  labels:    #A
    app: kiada    #B
    rel: stable    #C
spec:
  ...
```

#A The object's labels are defined in the metadata.labels field.
#B The "app" label is set to "kiada".
#C The "rel" label is set to "stable".

THE LISTING SHOWS HOW TO ADD LABELS TO AN OBJECT DEFINITION. LABELS ARE SUPPORTED BY ALL OBJECT KINDS. REGARDLESS OF THE KIND, YOU ADD LABELS TO THE OBJECT IN THIS WAY.

DISPLAYING OBJECT LABELS

You can see the labels of a particular object by running the `kubectl describe` command. View the labels of the pod `kiada-stable-001` as follows:

```
$ kubectl describe po kiada-stable-001
Name:         kiada-stable-001
Namespace:    kiada
Priority:     0
Node:         kind-worker2/172.18.0.2
Start Time:   Sun, 10 Oct 2021 21:58:25 +0200
Labels:       app=kiada    #A
              rel=stable    #A
Annotations:  <none>
...
```

#A These are the two labels that are defined in this pod's manifest file.

The `kubectl get pods` command doesn't display labels by default, but you can display them with the `--show-labels` option. Check the labels of all pods in the namespace as follows:

```
$ kubectl get po --show-labels
NAME                READY   STATUS    RESTARTS   AGE   LABELS     #A
kiada-canary        2/2     Running   0          12m   <none>     #B
kiada-stable-001    2/2     Running   0          12m   app=kiada,rel=stable    #C
kiada-stable-002    2/2     Running   0          12m   app=kiada,rel=stable    #C
kiada-stable-003    2/2     Running   0          12m   app=kiada,rel=stable    #C
quiz-stable         2/2     Running   0          12m   app=quiz,rel=stable    #D
quote-canary        2/2     Running   0          12m   <none>     #B
quote-stable-001    2/2     Running   0          12m   app=quote,rel=stable    #E
quote-stable-002    2/2     Running   0          12m   app=quote,rel=stable    #E
quote-stable-003    2/2     Running   0          12m   app=quote,rel=stable    #E
```

#A The pod labels are shown in the LABELS column
#B These pods have no labels.
#C These are the stable kiada pods.
#D This is the stable quiz pod.
#E These are the stable quote pods.

Instead of showing all labels with `--show-labels`, you can also show specific labels with the `--label-columns` option (or the shorter variant `-L`). Each label is displayed in its own column. List all pods along with their `app` and `rel` labels as follows:

```
$ kubectl get po -L app,rel
NAME                READY   STATUS    RESTARTS   AGE   APP     REL
kiada-canary        2/2     Running   0          14m
kiada-stable-001    2/2     Running   0          14m   kiada   stable
kiada-stable-002    2/2     Running   0          14m   kiada   stable
kiada-stable-003    2/2     Running   0          14m   kiada   stable
quiz-stable         2/2     Running   0          14m   quiz    stable
quote-canary        2/2     Running   0          14m
quote-stable-001    2/2     Running   0          14m   quote   stable
quote-stable-002    2/2     Running   0          14m   quote   stable
quote-stable-003    2/2     Running   0          14m   quote   stable
```

You can see that the two canary pods have no labels. Let's add them.

ADDING LABELS TO AN EXISTING OBJECT

To add labels to an existing object, you can edit the object's manifest file, add labels to the `metadata` section, and reapply the manifest using `kubectl apply`. You can also edit the object

definition directly in the API using `kubectl edit`. However, the simplest method is to use the `kubectl set label` command.

Add the labels `app` and `rel` to the `kiada-canary` pod using the following command:

```
$ kubectl label pod kiada-canary app=kiada rel=canary
pod/kiada-canary labeled
```

Now do the same for the pod `quote-canary`:

```
$ kubectl label pod quote-canary app=kiada rel=canary
pod/quote-canary labeled
```

List the pods and display their labels to confirm that all pods are now labelled. If you didn't notice the error when you entered the previous command, you probably caught it when you listed the pods. The `app` label of the pod `quote-canary` is set to the wrong value (`kiada` instead of `quote`). Let's fix this.

CHANGING LABELS OF AN EXISTING OBJECT

You can use the same command to update object labels. To change the label you set incorrectly, run the following command:

```
$ kubectl label pod quote-canary app=quote
error: 'app' already has a value (kiada), and --overwrite is false
```

To prevent accidentally changing the value of an existing label, you must explicitly tell kubectl to overwrite the label with `--overwrite`. Here's the correct command:

```
$ kubectl label pod quote-canary app=quote --overwrite
pod/quote-canary labeled
```

List the pods again to check that all the labels are now correct.

LABELLING ALL OBJECTS OF A KIND

Now imagine that you want to deploy another application suite in the same namespace. Before doing this, it is useful to add the `suite` label to all existing pods so that you can distinguish which pods belong to one suite and which belong to the other. Run the following command to add the label to all pods in the namespace:

```
$ kubectl label po --all suite=kiada-suite
pod/kiada-canary labeled
pod/kiada-stable-001 labeled
...
pod/quote-stable-003 labeled
```

List the pods again with the `--show-labels` or the `-L suite` option to confirm that all pods now contain this new label.

REMOVING A LABEL FROM AN OBJECT

Okay, I lied. You will not be setting up another application suite. Therefore, the `suite` label is redundant. To remove the label from an object, run the `kubectl label` command with a minus sign after the label key as follows:

```
$ kubectl label pod kiada-canary suite-    #A
pod/kiada-canary labeled
```

#A The minus sign signifies the removal of a label

To remove the label from all other pods, specify `--all` instead of the pod name:

```
$ kubectl label pod --all suite-
label "suite" not found.    #A
pod/kiada-canary not labeled    #A
pod/kiada-stable-001 labeled
...
pod/quote-stable-003 labeled
```

#A The kiada-canary pod doesn't have the suite label

> **NOTE** If you set the label value to an empty string, the label key is not removed. To remove it, you must use the minus sign after the label key.

10.2.3 Label syntax rules

While you can label your objects however you like, there are some restrictions on both the label keys and the values.

VALID LABEL KEYS

In the examples, you used the label keys `app`, `rel`, and `suite`. These keys have no prefix and are considered private to the user. Common label keys that Kubernetes itself applies or reads always start with a prefix. This also applies to labels used by Kubernetes components outside of the core, as well as other commonly accepted label keys.

An example of a prefixed label key used by Kubernetes is `kubernetes.io/arch`. You can find it on Node objects to identify the architecture type used by the node.

```
$ kubectl get node -L kubernetes.io/arch
NAME                 STATUS   ROLES                  AGE   VERSION   ARCH
kind-control-plane   Ready    control-plane,master   31d   v1.21.1   amd64    #A
kind-worker          Ready    <none>                 31d   v1.21.1   amd64    #A
kind-worker2         Ready    <none>                 31d   v1.21.1   amd64    #A
```

#A The kubernetes.io/arch label is set to amd64 on all three nodes.

The label prefixes `kubernetes.io/` and `k8s.io/` are reserved for Kubernetes components. If you want to use a prefix for your labels, use your organization's domain name to avoid conflicts.

When choosing a key for your labels, some syntax restrictions apply to both the prefix and the name part. The following table provides examples of valid and invalid label keys.

Table 10.1 Examples of valid and invalid label keys

Valid label keys	Invalid label keys
`foo`	`_foo`
`foo-bar_baz`	`foo%bar*baz`
`example/foo`	`/foo`
`example/FOO`	`EXAMPLE/foo`
`example.com/foo`	`example..com/foo`
`my_example.com/foo`	`my@example.com/foo`
`example.com/foo-bar`	`example.com/-foo-bar`
`my.example.com/foo`	`a.very.long.prefix.over.253.characters/foo`

The following syntax rules apply to the prefix:

- Must be a DNS subdomain (must contain only lowercase alphanumeric characters, hyphens, underscores, and dots).
- Must be no more than 253 characters long (not including the slash character).
- Must end with a forward slash.

The prefix must be followed by the label name, which:

- Must begin and end with an alphanumeric character.
- May contain hyphens, underscores, and dots.
- May contain uppercase letters.
- May not be longer than 63 characters.

VALID LABEL VALUES

Remember that labels are used to add identifying information to your objects. As with label keys, there are certain rules you must follow for label values. For example, label values can't contain spaces or special characters. The following table provides examples of valid and invalid label values.

Table 10.2 Examples of valid and invalid label values

Valid label values	Invalid label values
`foo`	`_foo`
`foo-bar_baz`	`foo%bar*baz`
`FOO`	`value.longer.than.63.characters`
(empty)	`value with spaces`

A label value:

- May be empty.
- Must begin with an alphanumeric character if not empty.
- May contain only alphanumeric characters, hyphens, underscores, and dots.
- Must not contain spaces or other whitespace.
- Must be no more than 63 characters long.

If you need to add values that don't follow these rules, you can add them as annotations instead of labels. You'll learn more about annotations later in this chapter.

10.2.4 Using standard label keys

While you can always choose your own label keys, there are some standard keys you should know. Some of these are used by Kubernetes itself to label system objects, while others have become common for use in user-created objects.

WELL-KNOWN LABELS USED BY KUBERNETES

Kubernetes doesn't usually add labels to the objects you create. However, it does use various labels for system objects such as Nodes and PersistentVolumes, especially if the cluster is running in a cloud environment. The following table lists some well-known labels you might find on these objects.

Table 10.3 Well-known labels on Nodes and PersistentVolumes

Label key	Example value	Applied to	Description
`kubernetes.io/arch`	`amd64`	Node	The architecture of the node.
`kubernetes.io/os`	`linux`	Node	The operating system running on the node.
`kubernetes.io/hostname`	`worker-node2`	Node	The node's hostname.
`topology.kubernetes.io/region`	`eu-west3`	Node PersistentVolume	The region in which the node or persistent volume is located.
`topology.kubernetes.io/zone`	`eu-west3-c`	Node PersistentVolume	The zone in which the node or persistent volume is located.
`node.kubernetes.io/instance-type`	`micro-1`	Node	The node instance type. Set when using cloud-provided infrastructure.

NOTE You can also find some of these labels under the older prefix `beta.kubernetes.io`, in addition to `kubernetes.io`.

Cloud providers can provide additional labels for nodes and other objects. For example, Google Kubernetes Engine adds the labels `cloud.google.com/gke-nodepool` and `cloud.google.com/gke-os-distribution` to provide further information about each node. You can also find more standard labels on other objects.

RECOMMENDED LABELS FOR DEPLOYED APPLICATION COMPONENTS

The Kubernetes community has agreed on a set of standard labels that you can add to your objects so that other users and tools can understand them. The following table lists these standard labels.

Table 10.4 Recommended labels used in the Kubernetes community

Label	Example	Description
app.kubernetes.io/name	quotes	The name of the application. If the application consists of multiple components, this is the name of the entire application, not the individual components.
app.kubernetes.io/instance	quotes-foo	The name of this application instance. If you create multiple instances of the same application for different purposes, this label helps you distinguish between them.
app.kubernetes.io/component	database	The role that this component plays in the application architecture.
app.kubernetes.io/part-of	kubia-demo	The name of the application suite to which this application belongs.
app.kubernetes.io/version	1.0.0	The version of the application.
app.kubernetes.io/managed-by	quotes-operator	The tool that manages the deployment and update of this application.

All objects belonging to the same application instance should have the same set of labels. For example, the pod and the persistent volume claim used by that pod should have the same values for the labels listed in the previous table. This way, anyone using the Kubernetes cluster can see which components belong together and which do not. Also, you can manage these components using bulk operations by using label selectors, which are explained in the next section.

10.3 Filtering objects with label selectors

The labels you added to the pods in the previous exercises allow you to identify each object and understand its place in the system. So far, these labels have only provided additional information when you list objects. But the real power of labels comes when you use *label selectors* to filter objects based on their labels.

Label selectors allow you to select a subset of pods or other objects that contain a particular label and perform an operation on those objects. A label selector is a criterion that filters objects based on whether they contain a particular label key with a particular value.

There are two types of label selectors:

- *equality-based* selectors, and
- *set-based* selectors.

INTRODUCING EQUALITY-BASED SELECTORS

An equality-based selector can filter objects based on whether the value of a particular label is equal to or not equal to a particular value. For example, applying the label selector `app=quote` to all pods in our previous example selects all quote pods (all stable instances plus the canary instance), as shown in the following figure.

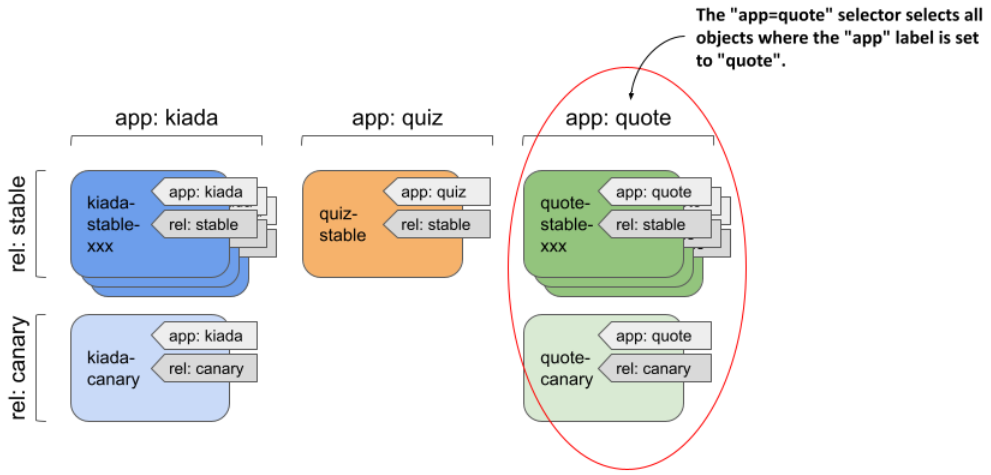

Figure 10.7 Selecting objects using an equality-based selector

Similarly, the label selector `app!=quote` selects all pods except the quote pods.

INTRODUCING SET-BASED SELECTORS

Set-based selectors are more powerful and allow you to specify:

- a set of values that a particular label must have; for example: `app in (quiz, quote)`,
- a set of values that a particular label must not have; for example: `app notin (kiada)`,
- a particular label key that should be present in the object's labels; for example, to select objects that have the `app` label, the selector is simply `app`,
- a particular label key that should not be present in the object's labels; for example, to select objects that do not have the `app` label, the selector is `!app`.

COMBINING MULTIPLE SELECTORS

When you filter objects, you can combine multiple selectors. To be selected, an object must match all of the specified selectors. As shown in the following figure, the selector `app=quote,rel=canary` selects the pod `quote-canary`.

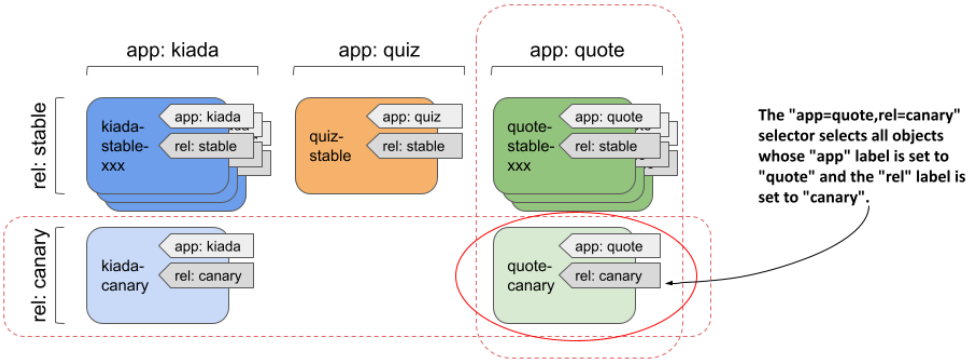

Figure 10.8 Combining two label selectors

You use label selectors when managing objects with kubectl, but they are also used internally by Kubernetes when an object references a subset of other objects. These scenarios are covered in the next two sections.

10.3.1 Using label selectors for object management with kubectl

If you've been following the exercises in this book, you've used the `kubectl get` command many times to list objects in your cluster. When you run this command without specifying a label selector, it prints all the objects of a particular kind. Fortunately, you never had more than a few objects in the namespace, so the list was never too long. In real-world environments, however, you can have hundreds of objects of a particular kind in the namespace. That's when label selectors come in.

FILTERING THE LIST OF OBJECTS USING LABEL SELECTORS

You'll use a label selector to list the pods you created in the `kiada` namespace in the previous section. Let's try the example in figure 10.7, where the selector `app=quote` was used to select only the pods running the quote application. To apply a label selector to `kubectl get`, specify it with the `--selector` argument (or the short equivalent `-l`) as follows:

```
$ kubectl get po -l app=quote
NAME                READY   STATUS    RESTARTS   AGE
quote-canary        2/2     Running   0          2h
quote-stable-001    2/2     Running   0          2h
quote-stable-002    2/2     Running   0          2h
quote-stable-003    2/2     Running   0          2h
```

Only the quote pods are shown. Other pods are ignored. Now, as another example, try listing all the canary pods:

```
$ kubectl get po -l rel=canary
NAME           READY   STATUS    RESTARTS   AGE
kiada-canary   2/2     Running   0          2h
quote-canary   2/2     Running   0          2h
```

Let's also try the example from figure 10.8, combining the two selectors `app=quote` and `rel=canary`:

```
$ kubectl get po -l app=quote,rel=canary
NAME           READY   STATUS    RESTARTS   AGE
quote-canary   2/2     Running   0          2h
```

Only the labels of the `quote-canary` pod match both label selectors, so only this pod is shown.

As the next example, try using a set-based selector. To display all quiz and quote pods, use the selector `'app in (quiz, quote)'` as follows:

```
$ kubectl get po -l 'app in (quiz, quote)' -L app
NAME               READY   STATUS    RESTARTS   AGE   APP
quiz-stable        2/2     Running   0          2h    quiz
quote-canary       2/2     Running   0          2h    quote
quote-stable-001   2/2     Running   0          2h    quote
quote-stable-002   2/2     Running   0          2h    quote
quote-stable-003   2/2     Running   0          2h    quote
```

You'd get the same result if you used the equality-based selector `'app!=kiada'` or the set-based selector `'app notin (kiada)'`. The `-L app` option in the command displays the value of the `app` label for each pod (see the `APP` column in the output).

The only two selectors you haven't tried yet are the ones that only test for the presence (or absence) of a particular label key. If you want to try them, first remove the `rel` label from the `quiz-stable` pod with the following command:

```
$ kubectl label po quiz-stable rel-
pod/quiz-stable labeled
```

You can now list pods that do not have the `rel` label like so:

```
$ kubectl get po -l '!rel'
NAME          READY   STATUS    RESTARTS   AGE
quiz-stable   2/2     Running   0          2h
```

NOTE **Make sure to use single quotes around `!rel`, so your shell doesn't evaluate the exclamation mark.**

And to list all pods that *do* have the `rel` label, run the following command:

```
$ kubectl get po -l rel
```

The command should show all pods except the `quiz-stable` pod.

If your Kubernetes cluster is running in the cloud and distributed across multiple regions or zones, you can also try to list nodes of a particular type or list nodes and persistent volumes in a particular region or zone. In table 10.3, you can see which label key to specify in the selector.

You've now mastered the use of label selectors when listing objects. Do you have the confidence to use them for deleting objects as well?

DELETING OBJECTS USING A LABEL SELECTOR

There are currently two canary releases in use in your system. It turns out that they aren't behaving as expected and need to be terminated. You could list all canaries in your system and remove them one by one. A faster method is to use a label selector to delete them in a single operation, as illustrated in the following figure.

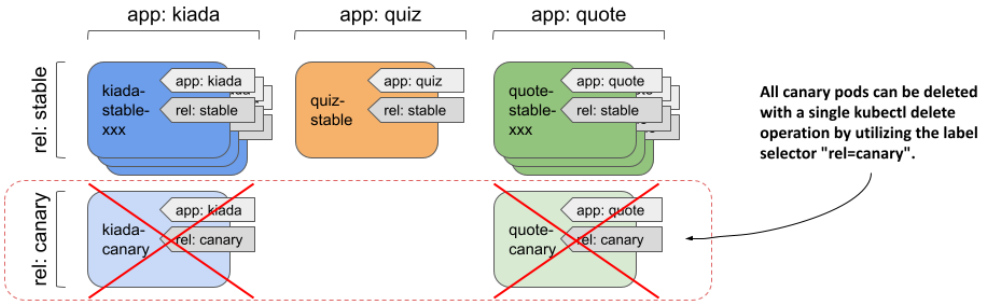

Figure 10.9 Selecting and deleting all canary pods using the `rel=canary` label selector

Delete the canary pods with the following command:

```
$ kubectl delete po -l rel=canary
pod "kiada-canary" deleted
pod "quote-canary" deleted
```

The output of the command shows that both the `kiada-canary` and `quote-canary` pods have been deleted. However, because the `kubectl delete` command does not ask for confirmation, you should be very careful when using label selectors to delete objects. Especially in a production environment.

10.3.2 Utilizing label selectors within Kubernetes API objects

You've learned how to use labels and selectors with kubectl to organize your objects and filter them, but selectors are also used within Kubernetes API objects.

For example, you can specify a node selector in each Pod object to specify which nodes the pod can be scheduled to. In the next chapter, which explains the Service object, you'll learn that you need to define a pod selector in this object to specify a pod subset to which the service will forward traffic. In the following chapters, you'll see how pod selectors are used by objects such as Deployment, ReplicaSet, DaemonSet, and StatefulSet to define the set of pods that belong to these objects.

USING LABEL SELECTORS TO SCHEDULE PODS TO SPECIFIC NODES

All the pods you've created so far have been randomly distributed across your entire cluster. Normally, it doesn't matter which node a pod is scheduled to, because each pod gets exactly

the amount of compute resources it requests (CPU, memory, and so on). Also, other pods can access this pod regardless of which node this and the other pods are running on. However, there are scenarios where you may want to deploy certain pods only on a specific subset of nodes.

A good example is when your hardware infrastructure isn't homogenous. If some of your worker nodes use spinning disks while others use SSDs, you may want to schedule pods that require low-latency storage only to the nodes that can provide it.

Another example is if you want to schedule front-end pods to some nodes and back-end pods to others. Or if you want to deploy a separate set of application instances for each customer and want each set to run on its own set of nodes for security reasons.

In all of these cases, rather than scheduling a pod to a particular node, allow Kubernetes to select a node out from a set of nodes that meet the required criteria. Typically, you'll have more than one node that meets the specified criteria, so that if one node fails, the pods running on it can be moved to the other nodes.

The mechanisms you can use to do this are labels and selectors.

ATTACHING LABELS TO NODES

The Kiada application suite consists of the Kiada, Quiz, and Quote services. Let's consider the Kiada service as the front-end and the Quiz and Quote services as the back-end services. Imagine that you want the Kiada pods to be scheduled only to the cluster nodes that you reserve for front-end workloads. To do this, you first label some of the nodes as such.

First, list all the nodes in your cluster and select one of the worker nodes. If your cluster consists of only one node, use that one.

```
$ kubectl get node
NAME                 STATUS   ROLES                  AGE   VERSION
kind-control-plane   Ready    control-plane,master   1d    v1.21.1
kind-worker          Ready    <none>                 1d    v1.21.1
kind-worker2         Ready    <none>                 1d    v1.21.1
```

In this example, I choose the `kind-worker` node as the node for the front-end workloads. After selecting your node, add the `node-role: front-end` label to it as follows:

```
$ kubectl label node kind-worker node-role=front-end
node/kind-worker labeled
```

Now list the nodes with a label selector to confirm that this is the only front-end node:

```
$ kubectl get node -l node-role=front-end
NAME          STATUS   ROLES    AGE   VERSION
kind-worker   Ready    <none>   1d    v1.21.1
```

If your cluster has many nodes, you can label multiple nodes this way.

SCHEDULING PODS TO NODES WITH SPECIFIC LABELS

To schedule a pod to the node(s) you designated as front-end nodes, you must add a node selector to the pod's manifest before you create the pod. The following listing shows the contents of the pod.kiada-front-end.yaml manifest file. The node selector is specified in the `spec.nodeSelector` field.

Listing 3.4 Using a node selector to schedule a pod to a specific node

```
apiVersion: v1
kind: Pod
metadata:
  name: kiada-front-end
spec:
  nodeSelector:    #A
    node-role: front-end    #A
  volumes:
```

#A This pod may only be scheduled to nodes with the node-role=front-end label.

In the `nodeSelector` field, you can specify one or more label keys and values that the node must match to be eligible to run the pod. Note that this field only supports specifying an equality-based label selector. The label value must match the value in the selector. You can't use a not-equal or set-based selector in the `nodeSelector` field. However, set-based selectors are supported in other objects.

When you create the pod from the previous listing by applying the manifest with `kubectl apply`, you'll see that the pod is scheduled to the node(s) that you have labelled with the label `node-role: front-end`. You can confirm this by displaying the pod with the `-o wide` option to show the pod's node as follows:

```
$ kubectl get po kiada-front-end -o wide
NAME              READY   STATUS    RESTARTS   AGE   IP            NODE
kiada-front-end   2/2     Running   0          1m    10.244.2.20   kind-worker   #A
```

#A The pod is running on the kind-worker node.

You can delete and recreate the pod several times to make sure that it always lands on the front-end node(s).

NOTE Other mechanisms for affecting pod scheduling are covered in chapter 21.

USING LABEL SELECTORS IN PERSISTENT VOLUME CLAIMS

In chapter 8, you learned about persistent volumes and persistent volume claims. A persistent volume usually represents a network storage volume, and the persistent volume claim allows you to reserve one of the persistent volumes so that you can use it in your pods.

I didn't mention this at the time, but you can specify a label selector in the PersistentVolumeClaim object definition to indicate which persistent volumes Kubernetes should consider for binding. Without the label selector, any available persistent volume that matches the capacity and access modes specified in the claim will be bound. If the claim specifies a label selector, Kubernetes also checks the labels of the available persistent volumes and binds the claim to a volume only if its labels match the label selector in the claim.

Unlike the node selector in the Pod object, the label selector in the PersistentVolumeClaim object supports both equality-based and set-based selectors and uses a slightly different syntax.

The following listing shows a PersistentVolumeClaim object definition that uses an equality-based selector to ensure that the bound volume has the label `condition: new`.

Listing 10.4 A PersistentVolumeClaim definition with an equality-based selector

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ssd-claim
spec:
  selector:    #A
    matchLabels:    #B
      type: ssd    #C
```

#A The label selector that the eligible PersistentVolume object must match.
#B The selector is equality-based.
#C Only PersistentVolume with the "type: ssd" label are eligible.

The `matchLabels` field behaves just like the `nodeSelector` field in the Pod object you learned about in the previous section.

Alternatively, you can use the `matchExpressions` field to specify a more expressive set-based label selector. The following listing shows a selector that selects only those PersistentVolumes that either don't have the `type` label set or have a value other than `ssd` and have the `age` label set to either `old` or `very-old`.

Listing 10.5 Using a set-based selector in a PersistentVolumeClaim

```
spec:
  selector:
    matchExpressions:    #A
    - key: type    #B
      operator: NotIn    #B
      values:    #B
      - ssd    #B
    - key: age    #C
      operator: In    #C
      values:    #C
      - old    #C
      - very-old    #C
```

#A A set-based selector is defined. The PersistentVolume's labels must match the following expressions.
#B The "type" label must not match the value "ssd".
#C The "age" label must be either "old" or "very-old".

As you can see in the listing, you can specify multiple expressions. The PersistentVolume's labels must match all of the specified expressions to be selected. You must specify the `key`, `operator`, and `values` for each expression.

The `key` is the label key to which the selector is applied. The `operator` must be one of `In`, `NotIn`, `Exists`, and `DoesNotExist`. When you use the `In` or `NotIn` operators, the `values` array must not be empty, but it must be empty when you use the `Exists` and `DoesNotExist` operators.

To see these selectors in action, first create the persistent volumes found in the manifest file persistent-volumes.yaml. Then create the two claims in the manifest files pvc.ssd-claim.yaml and pvc.old-non-ssd-claim.yaml. You can find these files in the Chapter10/ directory in the book's code archive.

Filtering objects with field selectors

Kubernetes initially only allowed you to filter objects with label selectors. Then it became clear that users want to filter objects by other properties as well. One such example is filtering pods based on the cluster node they are running on. This can now be accomplished with *field selectors*. Unlike label selectors, you only use field selectors with kubectl or other Kubernetes API clients. No object uses field selectors internally.

The set of fields you can use in a field selector depends on the object kind. The `metadata.name` and `metadata.namespace` fields are always supported. Like equality-based label selectors, field selectors support the equal (`=` or `==`) and not equal (`!=`) operator, and you can combine multiple field selectors by separating them with a comma.

Listing pods running on a specific node

As an example of using field selectors, run the following command to list pods on the `kind-worker` node (if your cluster wasn't provisioned with the kind tool, you must specify a different node name):

```
$ kubectl get pods --field-selector spec.nodeName=kind-worker
NAME           READY   STATUS    RESTARTS   AGE
kiada-front-end   2/2     Running   0          15m
kiada-stable-002  2/2     Running   0          3h
quote-stable-002  2/2     Running   0          3h
```

Instead of displaying all pods in the current namespace, the filter selected only those pods whose `spec.nodeName` field is set to `kind-worker`.

How do you know which field to use in the selector? By looking up the field names with `kubectl explain`, of course. You learned this in chapter 4. For example: `kubectl explain pod.spec` shows the fields in the `spec` section of Pod objects. It doesn't show which fields are supported in field selectors, but you can try to use a field and kubectl will tell you if it isn't supported.

Finding pods that aren't running

Another example of using field selectors is to find pods that aren't currently running. You accomplish this by using the `status.phase!=Running` field selector as follows:

```
$ kubectl get po --field-selector status.phase!=Running
```

Since all pods in your namespace are running, this command won't produce any results, but it can be useful in practice, especially if you combine it with the `--all-namespaces` option to list non-running pods in all namespaces. The full command is as follows:

```
$ kubectl get po --field-selector status.phase!=Running --all-namespaces
```

The `--all-namespaces` option is also useful when you use the `metadata.name` or `metadata.namespace` fields in the field selector.

10.4 Annotating objects

Adding labels to your objects makes them easier to manage. In some cases, objects must have labels because Kubernetes uses them to identify which objects belong to the same set. But as you learned in this chapter, you can't just store anything you want in the label value. For example, the maximum length of a label value is only 63 characters, and the value can't contain whitespace at all.

For this reason, Kubernetes provides a feature similar to labels–object *annotations*.

10.4.1 Introducing object annotations

Like labels, annotations are also key-value pairs, but they don't store identifying information and can't be used to filter objects. Unlike labels, an annotation value can be much longer (up to 256 KB at the time of this writing) and can contain any character.

UNDERSTANDING ANNOTATIONS ADDED BY KUBERNETES

Tools like kubectl and the various controllers that run in Kubernetes may add annotations to your objects if the information can't be stored in one of the object's fields.

Annotations are often used when new features are introduced to Kubernetes. If a feature requires a change to the Kubernetes API (for example, a new field needs to be added to an object's schema), that change is usually deferred for a few Kubernetes releases until it's clear that the change makes sense. After all, changes to any API should always be made with great care, because after you add a field to the API, you can't just remove it or you'll break everyone that use the API.

Changing the Kubernetes API requires careful consideration, and each change must first be proven in practice. For this reason, instead of adding new fields to the schema, usually a new object annotation is introduced first. The Kubernetes community is given the opportunity to use the feature in practice. After a few releases, when everyone's happy with the feature, a new field is introduced and the annotation is deprecated. Then a few releases later, the annotation is removed.

ADDING YOUR OWN ANNOTATIONS

As with labels, you can add your own annotations to objects. A great use of annotations is to add a description to each pod or other object so that all users of the cluster can quickly see information about an object without having to look it up elsewhere.

For example, storing the name of the person who created the object and their contact information in the object's annotations can greatly facilitate collaboration between cluster users.

Similarly, you can use annotations to provide more details about the application running in a pod. For example, you can attach the URL of the Git repository, the Git commit hash, the build timestamp, and similar information to your pods.

You can also use annotations to add the information that certain tools need to manage or augment your objects. For example, a particular annotation value set to `true` could signal to the tool whether it should process and modify the object.

UNDERSTANDING ANNOTATION KEYS AND VALUES

The same rules that apply to label keys also apply to annotations keys. For more information, see section 10.2.3. Annotation values, on the other hand, have no special rules. An annotation value can contain any character and can be up to 256 KB long. It must be a string, but can contain plain text, YAML, JSON, or even a Base64-Encoded value.

10.4.2 Adding annotations to objects

Like labels, annotations can be added to existing objects or included in the object manifest file you use to create the object. Let's look at how to add an annotation to an existing object.

SETTING OBJECT ANNOTATIONS

The simplest way to add an annotation to an existing object is to use the `kubectl annotate` command. Let's add an annotation to one of the pods. You should still have a pod named `kiada-front-end` from one of the previous exercises in this chapter. If not, you can use any other pod or object in your current namespace. Run the following command:

```
$ kubectl annotate pod kiada-front-end created-by='Marko Luksa <marko.luksa@xyz.com>'
pod/kiada-front-end annotated
```

This command adds the annotation `created-by` with the value `'Marko Luksa <marko.luksa@xyz.com>'` to the `kiada-front-end` pod.

SPECIFYING ANNOTATIONS IN THE OBJECT MANIFEST

You can also add annotations to your object manifest file before you create the object. The following listing shows an example. You can find the manifest in the pod.pod-with-annotations.yaml file.

Listing 10.6 Annotations in an object manifest

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-annotations
  annotations:
    created-by: Marko Luksa <marko.luksa@xyz.com>    #A
    contact-phone: +1 234 567 890   #B
    managed: 'yes'    #C
    revision: '3'    #D
spec:
  ...
```

#A Here's one annotation.
#B Here's another one.
#C Third annotation. Value must be quoted. See next warning for explanation.
#D Another annotation value that must be quoted or an error would occur.

WARNING Make sure you enclose the annotation value in quotes if the YAML parser would otherwise treat it as something other than a string. If you don't, a cryptic error will occur when you apply the manifest. For example, if the annotation value is a number like 123 or a value that could be interpreted as a boolean (true, false, but also words like yes and no), enclose the value in quotes (examples: "123", "true", "yes") to avoid the following error: "unable to decode yaml ... ReadString: expects " or n, but found t".

Apply the manifest from the previous listing by executing the following command:

```
$ kubectl apply -f pod.pod-with-annotations.yaml
```

10.4.3 Inspecting an object's annotations

Unlike labels, the `kubectl get` command does not provide an option to display annotations in the object list. To see the annotations of an object, you should use `kubectl describe` or find the annotation in the object's YAML or JSON definition.

VIEWING AN OBJECT'S ANNOTATIONS WITH KUBECTL DESCRIBE

To see the annotations of the `pod-with-annotations` pod you created, use `kubectl describe`:

```
$ kubectl describe po pod-with-annotations
Name:         pod-with-annotations
Namespace:    kiada
Priority:     0
Node:         kind-worker/172.18.0.4
Start Time:   Tue, 12 Oct 2021 16:37:50 +0200
Labels:       <none>
Annotations:  contact-phone: +1 234 567 890    #A
              created-by: Marko Luksa <marko.luksa@xyz.com>    #A
              managed: yes    #A
              revision: 3    #A
Status:       Running

...
```

#A These are the four annotations that were defined in the manifest file.

DISPLAYING THE OBJECT'S ANNOTATIONS IN THE OBJECT'S JSON DEFINITION

Alternatively, you can use the `jq` command to extract the annotations from the JSON definition of the pod:

```
$ kubectl get po pod-with-annotations -o json | jq .metadata.annotations
{
  "contact-phone": "+1 234 567 890",
  "created-by": "Marko Luksa <marko.luksa@xyz.com>",
  "kubectl.kubernetes.io/last-applied-configuration": "..."    #A
  "managed": "yes",
  "revision": "3"
}
```

#A This annotation is added by kubectl. It could be deprecated and removed in the future.

You'll notice that there's an additional annotation in the object with the key `kubectl.kubernetes.io/last-applied-configuration`. It isn't shown by the `kubectl`

`describe` command, because it's only used internally by kubectl and would also make the output too long. In the future, this annotation may become deprecated and then be removed. Don't worry if you don't see it when you run the command yourself.

10.4.4 Updating and removing annotations

If you want to use the `kubectl annotate` command to change an existing annotation, you must also specify the `--overwrite` option, just as you would when changing an existing object label. For example, to change the annotation `created-by`, the full command is as follows:

```
$ kubectl annotate pod kiada-front-end created-by='Humpty Dumpty' --overwrite
```

To remove an annotation from an object, add the minus sign to the end of the annotation key you want to remove:

```
$ kubectl annotate pod kiada-front-end created-by-
```

10.5 Summary

The Kubernetes features described in this chapter will help you keep your cluster organized and make your system easier to understand. In this chapter, you learned that:

- Objects in a Kubernetes cluster are typically divided into many namespaces. Within a namespace, object names must be unique, but you can give two objects the same name if you create them in different namespaces.
- Namespaces allow different users and teams to use the same cluster as if they were using separate Kubernetes clusters.
- Each object can have several labels. Labels are key-value pairs that help identify the object. By adding labels to objects, you can effectively organize objects into groups.
- Label selectors allow you to filter objects based on their labels. You can easily filter pods that belong to a specific application, or by any other criteria if you've previously added the appropriate labels to those pods.
- Field selectors are like label selectors, but they allow you to filter objects based on specific fields in the object manifest. A field selector is used to list pods that run on a particular node.
- Instead of performing an operation on each pod individually, you can use a label selector to perform the same operation on a set of objects that match the label selector.
- Labels and selectors are also used internally by some object types. You can add labels to Node objects and define a node selector in a pod to schedule that pod only to those nodes that meet the specified criteria.
- In addition to labels, you can also add annotations to objects. An annotation can contain a much larger amount of data and can include whitespace and other special characters that aren't allowed in labels. Annotations are typically used to add additional information used by tools and cluster users. They are also used to defer changes to the Kubernetes API.

In the next chapter, you'll learn how to forward traffic to a set of pods using the Service object.