Learn
KUBERNETES
IN A MONTH OF LUNCHES

ELTON STONEMAN

This ebook is the first week,
complements of the team behind

ORACLE
Linux

Learn Kubernetes in a Month of Lunches

Elton Stoneman

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Corporate Program Manager: Candace Gillhoolley, corp-sales@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN: 9781617299308

contents

foreword iv

1 Before you begin 1

2 Running containers in Kubernetes with Pods and Deployments 13

3 Connecting Pods over the network with Services 37

4 Configuring applications with ConfigMaps and Secrets 61

5 Storing data with volumes, mounts, and claims 87

6 Scaling applications across multiple Pods with controllers 115

index 139

foreword

Kubernetes is the engine that powers complex cloud native environments. It has been deemed one of the fastest growing projects in the history of open source software,. It's hard to argue, given that in the mere six years since the project started, 1.7 million developers are already using it. In a November, 2020 survey, 91% of respondents report using Kubernetes, 83% of them in production. This is an increase from 78% 2019, and 58% in 2018.

These are compelling statistics. So, do you need begin adopting Kubernetes? If enterprise cloud native applications are part of your world, the answer is yes.

Kubernetes is an open source system for automating deployment, scaling, and management of containerized applications and its development is stewarded by the Cloud Native Computing Foundation® (CNCF®), which itself is part of the Linux Foundation. Other projects endorsed by the CNCF include several de facto industry standard technologies like containerd, CoreDNS, Harbor, Helm, and Prometheus.

Combined, these projects, whether sandboxed, incubating, or graduated, define a cloud native application development and deployment framework that is the basis of most cloud native digital transformation efforts. The CNCF also provides conformance testing for Kubernetes to help ensure consistency across vendor implementations.

What does this all mean to you? Now is a good time increase your Kubernetes skills and knowledge.

That's why Oracle is offering this ebook excerpt of *Learn Kubernetes in a Month of Lunches*. Here, you can dive into the first week of lessons free. These first few chapters cover running apps on Kubernetes. Author, consultant, and trainer, Elton Stoneman travels the world helping people understand what containers, Kubernetes, and DevOps can do for them and their software delivery. In *Learn Kubernetes in a Month of Lunches,* Stoneman uses a task-focused approach, through exercises and labs, to help you quickly learn practical ways to apply the technology.

For developers who want to try Kubernetes in a real-world environment, Oracle offers two choices.

For the developer who wants to use their laptop, there is a Vagrant project published by Oracle on GitHub for Oracle Linux Cloud Native Environment, our Kubernetes implementation for Oracle Linux. It uses all open source components to deploy a production-like stack using Oracle VM VirtualBox on Windows, macOS or Linux.

For cloud-based developers, Oracle Container Engine for Kubernetes (OKE) is provided as a managed service at no additional cost over the per-instance cost for the worker nodes. With Oracle's Free Cloud Tier, you can test drive Oracle Container Engine for Kubernetes. It includes US$300 of free credit and 5TB of storage to use for up to 30 days.

Onward to navigating Kubernetes.

Sergio Leunissen,
Vice President, Development,
Oracle

Sergio Leunissen is a Vice President in Oracle's infrastructure engineering team. He joined Oracle in 1995 and has held a range of positions in sales engineering, development, and product management. Sergio currently leads initiatives to deliver solutions for developers on Oracle Linux and Oracle Cloud Infrastructure. He is also responsible for Oracle's presence on GitHub.

---

https://www.cncf.io/blog/2020/11/17/cloud-native-survey-2020-containers-in-production-jump-300-from-our-first-survey/

https://www.cncf.io/blog/2020/03/04/2019-cncf-survey-results-are-here-deployments-are-growing-in-size-and-speed-as-cloud-native-adoption-becomes-mainstream/

https://www.cncf.io/wp-content/uploads/2020/08/CNCF_Survey_Report.pdf

https://landscape.cncf.io/

https://github.com/cncf/k8s-conformance

https://www.oracle.com/cloud/free/ - free-cloud-trial

Oracle Linux Cloud Native Environment is a curated set of open source software selected from cloud native projects, integrated into a unified operating environment that includes:

- A rich set of software components for cloud native application development and deployment
- Certified Kubernetes and Kata Containers
- Enterprise-grade performance, scalability, reliability, and security

Oracle Linux Cloud Native Environment runs on-premises or in the cloud. It is tested and proven with Oracle products such as Oracle Database, Oracle WebLogic Server, MySQL and more.

To find out more see oracle.com/linux/cloudnative and oracle.com/linux

Stay connected:

- blogs.oracle.com/linux
- facebook.com/OracleLinux
- twitter.com/OracleLinux

about the author

Elton Stoneman had spent most of his career as a consultant, designing and building large enterprise applications. Then he discovered the container revolution, joined Docker, and worked with the team for three fast-and-furious years. Now he helps people break up those old enterprise apps and build new cloud-native apps—and run them all in Docker and Kubernetes. He speaks at conferences and runs workshops around the world, writes books and video courses, and helps organizations at every stage in their container journey. Elton is a 10-time Microsoft MVP and a Docker Captain.

1 Before you begin

Kubernetes is big. Really big. It was released as an open-source project on GitHub in 2014 and now it's averaging 200 changes every week from a worldwide community of 2,500 contributors. The annual KubeCon conference has grown from 1,000 attendees in 2016 to more than 12,000 at the most recent event, and it's now a global series with events in America, Europe and Asia. All the major clouds offer a managed Kubernetes service and you can run Kubernetes in the data center or on your laptop—*and they're all the same Kubernetes.*

Independence and standardization are the main reasons Kubernetes is so popular. Once you have your apps running nicely in Kubernetes you can deploy them anywhere, which is very attractive for organizations moving to the cloud, because it keeps them free to move between data centers and other clouds without a rewrite. It's also very attractive for practitioners—once you've mastered Kubernetes, you can move between projects and organizations and be very productive very quickly.

Getting to that point is hard, though, because Kubernetes is hard. Even simple apps are deployed as multiple components, described in a custom file format which can easily span many hundreds of lines. Kubernetes brings infrastructure-level concerns like load-balancing, networking, storage and compute into app configuration, which might be new concepts depending on your IT background. And Kubernetes is always expanding—there are new releases every quarter that often bring a ton of new functionality.

But it's worth it. I've spent many years helping people learn Kubernetes and there's a common pattern that repeats: the question *why is this so complicated?* gets

resolved with *you can do that? This is amazing!* Kubernetes truly is an amazing piece of technology. The more you learn about it the more you'll love it, and this book will accelerate you on your journey to Kubernetes mastery.

1.1 Understanding Kubernetes

This book is a hands-on introduction to Kubernetes. Every chapter is filled with try-it-now exercises and labs for you to get lots of experience using Kubernetes. All except this one :). We'll jump into the practical work in the next chapter, but we need just a little theory first. Let's start by understanding what Kubernetes actually is and the problems it solves.

Kubernetes is a platform for running containers. It takes care of starting your containerized applications, rolling out updates, maintaining service levels, scaling to meet demand, securing access, and much more. The two core concepts in Kubernetes are the *API* that you use to define your applications, and the *cluster* that runs your applications. A cluster is a set of individual servers that have all been configured with a container runtime like Docker, and then joined together into a single logical unit with Kubernetes. Figure 1.1 shows a high-level view of the cluster.

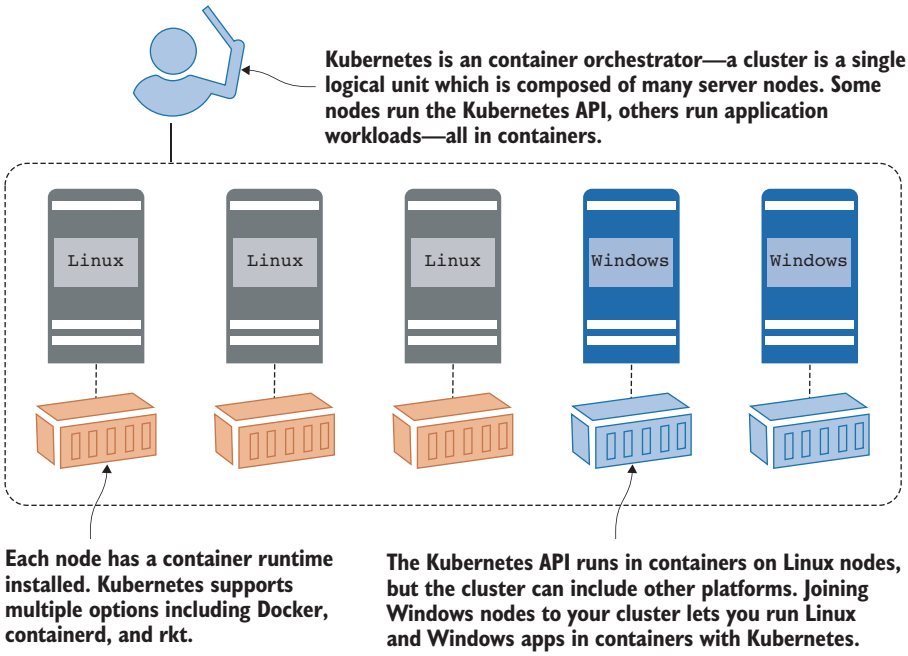

Figure 1.1 A Kubernetes cluster is just a bunch of servers that can run containers, joined into a group.

Cluster administrators manage the individual servers—called *nodes* in Kubernetes. You can add nodes to expand the capacity of the cluster, take nodes offline for servicing,

or roll out an upgrade of Kubernetes across the cluster. In a managed service like Microsoft's Azure Kubernetes Service or Amazon's Elastic Kubernetes Service, those functions are all wrapped in simple web interfaces or command lines. In normal usage you forget about the underlying nodes, and you treat the cluster as a single entity.

The Kubernetes cluster is there to run your applications. You define your apps in YAML files and send those files to the Kubernetes API. Kubernetes looks at what you're asking for in the YAML and compares it to what's already running in the cluster. It makes any changes it needs to get to the desired state, which could be updating configuration, removing containers, or creating new containers. Containers are distributed around the cluster for high availability, and they can all communicate over virtual networks managed by Kubernetes. Figure 1.2 shows the deployment process without the nodes, because we don't really care about them at this level.

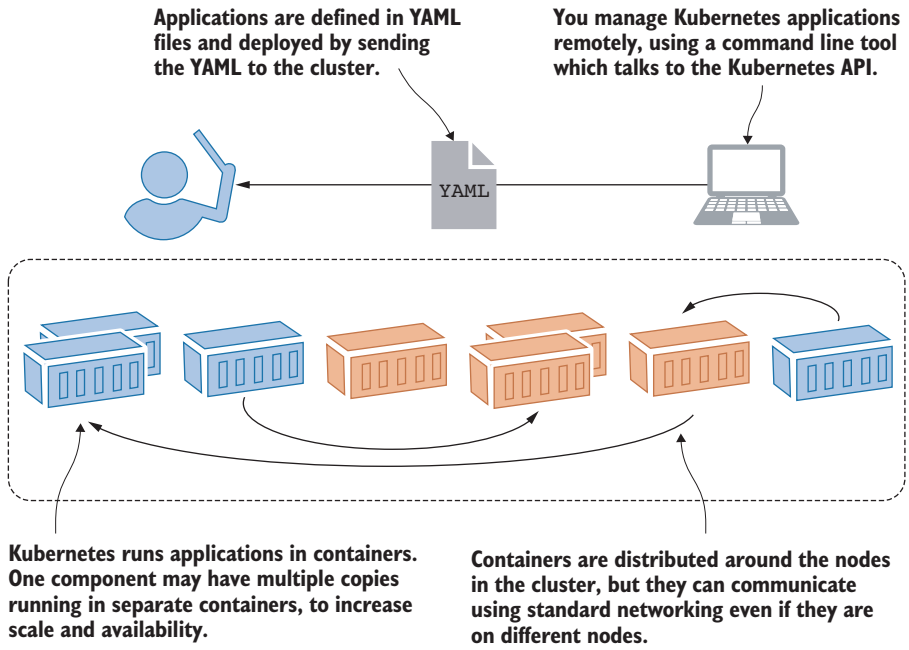

Figure 1.2 When you deploy apps to a Kubernetes cluster, you can usually ignore the actual nodes.

Defining the structure of the application is your job but running and managing everything is down to Kubernetes. If a node in the cluster goes offline and takes some containers with it, Kubernetes sees that and starts replacement containers on other nodes. If an application container becomes unhealthy, Kubernetes can restart it. If a component is under stress because of high load, Kubernetes can start extra copies of the component in new containers. You put the work into your Docker images and Kubernetes YAML files, and you'll get a self-healing app that runs in the same way on any Kubernetes cluster.

Kubernetes manages more than just containers, which is what makes it a complete application platform. The cluster has a distributed database, and you can use that to store configuration files for your applications, and to store secrets like API keys and connection credentials. Kubernetes delivers those seamlessly to your containers, which lets you use the same container images in every environment and apply the correct configuration from the cluster. Kubernetes also provides storage so your applications can maintain data outside of containers, giving you high availability for stateful apps. And Kubernetes manages network traffic coming into the cluster, sending it to the right containers for processing. Figure 1.3 shows those other resources, which are the main features of Kubernetes.

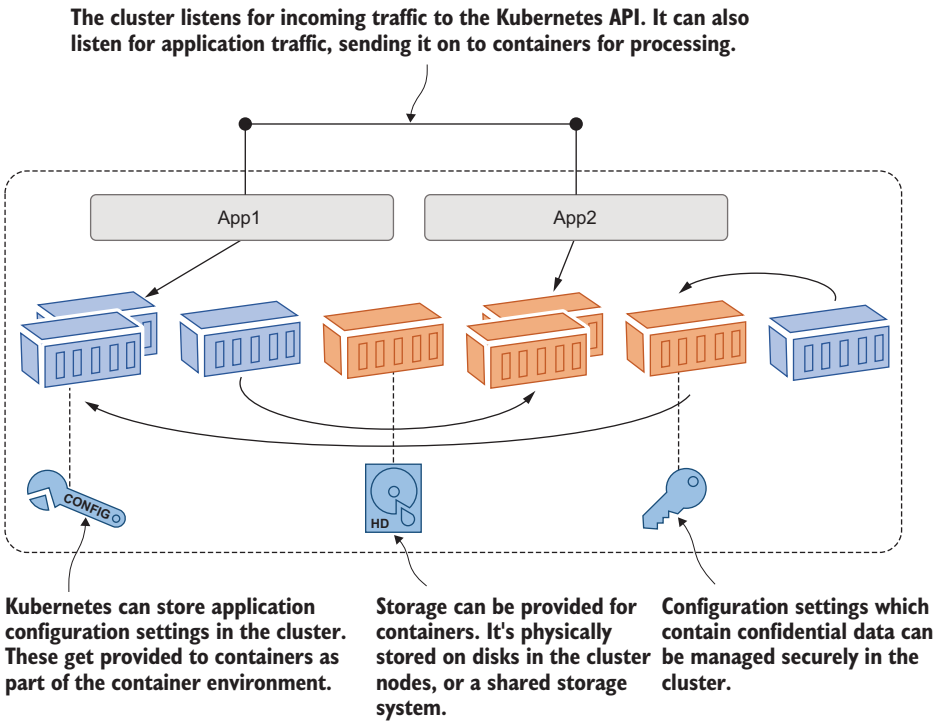

Figure 1.3 There's more to Kubernetes than just containers—the cluster manages other resources too

I haven't talked about what those applications in the containers look like, and that's because Kubernetes doesn't really care. You can run a new application built with cloud-native design across microservices in multiple containers. You can run a legacy application built as a monolith in one big container. They could be Linux apps or Windows apps. You define all types of application in YAML files using the same API, and you can run them all on a single cluster. The joy of working with Kubernetes is that it adds a layer of consistency on top of all your apps — old .NET and Java mono-

liths and new Node.js and Go microservices are all described, deployed and managed in the same way.

That's just about all the theory we need to get started with Kubernetes, but before we go any further, I want to put some proper names to the concepts I've been talking about. Those YAML files are properly called *application manifests*, because they're a list of all the components that go into shipping the app. And those components are Kubernetes *resources* — they have proper names too. Figure 1.4 takes the concepts from figure 1.3 and applies the correct Kubernetes resource names.

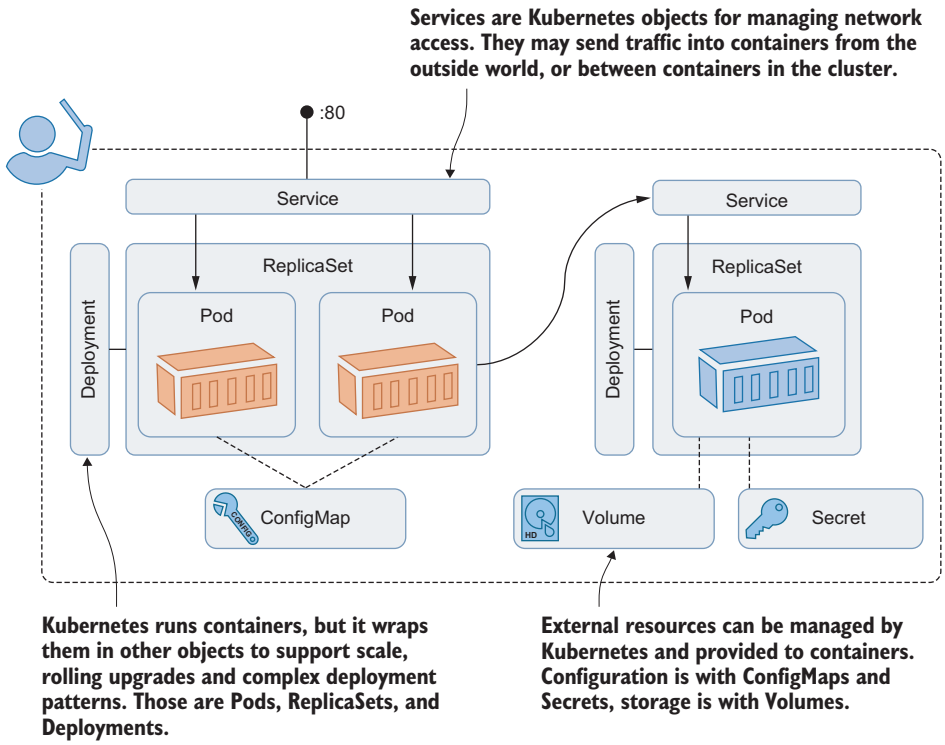

Figure 1.4 The true picture—these are the basic Kubernetes resources you need to master.

I told you Kubernetes was hard :). But we will cover all those resources one at a time over the next few chapters, layering on the understanding. By the time you've finished chapter 6 that diagram will make complete sense, and you'll have had lots of experience of defining those resources in YAML files and running them in your own Kubernetes cluster.

1.2 Is this book for you?

The goal for this book is to fast-track your Kubernetes learning to the point where you have confidence defining and running your own apps in Kubernetes, and you under-

stand what the path to production looks like. The best way to learn Kubernetes is to practice, and if you follow all the examples in the chapters and work through the labs, then you'll have a solid understanding of all the most important pieces of Kubernetes by the time you finish the book.

But Kubernetes is a huge topic, and I won't be covering everything. The biggest gaps are in administration. I won't cover cluster setup and management to any depth, because that varies across different infrastructures. If you're planning on running Kubernetes in the cloud as your production environment, then a lot of those concerns are taken care of in a managed service anyway. If you want to get your Kubernetes certification, this book is a great place to start, but it won't get you all the way. There are two Kubernetes certifications—Certified Kubernetes Application Developer (CKAD) and Certified Kubernetes Administrator (CKA). This book covers about 80% of the CKAD curriculum and about 50% of CKA.

There's also a reasonable amount of background knowledge you'll need to work with this book effectively. I'll explain lots of core principles as we encounter them in Kubernetes features, but I won't fill in any gaps about containers. If you're not familiar with ideas like images, containers, and registries, then I recommend starting with *Learn Docker in a Month of Lunches* from Manning. You don't need to use Docker with Kubernetes, but it is the easiest and most flexible way to package your apps, so you can run them in containers with Kubernetes.

If you classify yourself as a new or improving Kubernetes user with a reasonable working knowledge of containers, then this is the book for you. Your background could be in development, operations, architecture, DevOps, or SRE. Kubernetes touches all those roles, they're all welcome here, and you are going to learn an absolute ton of stuff.

1.3 How to use this book

This book follows the month of lunches principles: you should be able to work through each chapter in your lunchbreak, and work through the whole book within a month. "Work" is the key here because you should look at putting aside time to read the chapter, work through the try-it-now exercises, and have a go at the hands-on lab at the end. You should expect to have a few extended lunchbreaks because the chapters don't cut any corners or skip over key details. You need a lot of muscle memory to work effectively with Kubernetes and practicing every day will really cement the knowledge you gain in each chapter.

1.3.1 Your learning journey

Kubernetes is a vast subject, but I've taught it for years now in training sessions and workshops, in-person and virtual, and built out an incremental learning path that I know works. We'll start with the core concepts and gradually add more detail, saving the most complex topics for when you're already very familiar with Kubernetes.

Chapters 2 through 6 jump in to running apps on Kubernetes. You'll learn how to define applications in YAML that manifest which Kubernetes will run as containers, how you can configure network access for containers to talk to each other, and how to make traffic from the outside world reach your containers. You'll learn how your apps can read configuration from Kubernetes, how to write data to storage units managed by Kubernetes, and how you can scale your applications.

Chapters 7 through 11 build on the basics with subjects that deal with real-world Kubernetes usage. You'll learn how you can run containers that share a common environment, and how you can use containers to run batch jobs and scheduled jobs. You'll learn how Kubernetes supports automated rolling updates so you can release new application versions with zero downtime, and how to use Helm to provide a configurable way to deploy your apps. You'll also learn about the practicalities of building apps with Kubernetes, looking at different developer workflows and CI/CD pipelines.

Chapters 12 through 16 are all about production readiness, going beyond just running your apps in Kubernetes to running them in a way that's good enough to go live. You'll learn how to configure self-healing applications, collect and centralize all your logs, and build monitoring dashboards to visualize the health of your systems. Security is also in here and you'll learn how to secure public access to your apps, and how to secure the applications themselves.

Chapters 17 to 21 move into expert territory. Here you'll learn how to work with large Kubernetes deployments, and how to configure your applications to automatically scale up and down. You'll learn how to implement role-based access control to secure access to Kubernetes resources, and we'll cover some more interesting uses of Kubernetes—as a platform for serverless functions, and as a multi-architecture cluster that can run apps built for Linux and Windows, Intel and Arm.

By the end of the book, you should be confident about bringing Kubernetes into your daily work. The final chapter offers guidance on moving on with Kubernetes—further reading for each topic in the book and advice on choosing a Kubernetes provider.

1.3.2 Try-it-nows

Every chapter of the book has many guided exercises for you to complete. The source code for the book is all on GitHub at https://github.com/sixeyed/kiamol. You'll clone that when you set up your lab environment, and you'll use it for all the samples, which will have you running increasingly complex applications in Kubernetes.

Many chapters build on work from earlier in the book, but you do not need to follow all the chapters in order, so you can follow your own learning journey. The exercises within a chapter do often build on each other, so if you skip exercises you may find errors later on, which will help you hone your troubleshooting skills. All the exercises use container images that are publicly available on Docker Hub, and your Kubernetes cluster will download any that it needs.

There is a lot of content in this book. You'll get the most out of it if you work through the samples as you read the chapters and you'll feel a lot more comfortable

about using Kubernetes going forward. If you don't have time to work through every exercise, then it's fine to skip some; they all have a screenshot showing the output you would have seen, and every chapter finishes with a wrap-up section to make sure you're confident with the topic.

1.3.3 Hands-on labs

Each chapter also ends with a hands-on lab that invites you to go further than the try-it-now exercises. These aren't guided—you'll get some instructions and some hints, and then it will be down to you to complete the lab. There are sample answers for all the labs in the sixeyed/kiamol GitHub repo, so you can check what you've done, or see how I've done it if you don't have time for one of the labs.

1.3.4 Additional resources

Manning's *Kubernetes in Action* is a great book that covers a lot of the administration details which I don't cover here. Other than that the main resource for further reading is the official Kubernetes documentation, which you'll find in two places. The documentation site (https://kubernetes.io/docs/home/) covers everything from the architecture of the cluster, through guided walkthroughs and learning how to contribute to Kubernetes yourself. The Kubernetes API reference (https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.19) contains the detailed specification for every type of object you can create—that's one to bookmark.

Twitter is home to the Kubernetes @kubernetesio account, and you can also follow some of the founding members of the Kubernetes project and community, like Brendan Burns (@brendandburns), Tim Hockin (@thockin), Joe Beda (@jbeda) and Kelsey Hightower (@kelseyhightower).

I talk about this stuff all the time myself too. You can follow me on Twitter @EltonStoneman, my blog is https://blog.sixeyed.com, and I post YouTube videos at https://youtube.com/eltonstoneman.

1.4 Creating your lab environment

A Kubernetes cluster can have hundreds of nodes, but for the exercises in this book, a single-node cluster is fine. We'll get your lab environment set up now so you're ready to get started in the next chapter. There are dozens of Kubernetes platforms, and the exercises in this book should work with any certified Kubernetes setup. I'll describe how to create your lab on Linux, Windows, Mac, Amazon Web Services (AWS), and Azure, which covers all the major options. I'm using Kubernetes version 1.18, but earlier or later versions should be fine too.

The easiest option to run Kubernetes locally is Docker Desktop, which is a single package that gives you Docker and Kubernetes and all the command line tools. It also integrates nicely with your computer's network and has a handy Reset Kubernetes button that clears everything down. Docker Desktop is supported for Windows 10 and macOS, and if that doesn't work for you, I'll also walk through some alternatives.

One point you should know—the components of Kubernetes itself need to run as Linux containers. You can't run Kubernetes in Windows (although you can run Windows apps in containers with a multi-node Kubernetes cluster), so you'll need a Linux virtual machine if you're working on Windows. Docker Desktop sets that up and manages it for you.

And one last note for Windows users: please use PowerShell to follow along with the exercises. PowerShell supports many Linux commands, and the try-it-now exercises are built to run on Linux (and Mac) shells and PowerShell. If you try to use the classic Windows command terminal, you're going to get issues from the start.

1.4.1 Download the book's source code

Every example and exercise is in the book's source code repository on GitHub, together with sample solutions for all of the labs. If you're comfortable with Git and you have a Git client installed, you can clone the repository onto your computer with this command:

```
git clone https://github.com/sixeyed/kiamol
```

If you're not a Git user, you can browse to the GitHub page for the book at https://github.com/sixeyed/kiamol and click the Clone or Download button to download a ZIP file that you can expand.

The root of the source code is a folder called kiamol, and within that is a folder for each chapter: ch02, ch03, and so on. The first exercise in the chapter will usually ask you to open a terminal session and switch to the chXX directory, so you'll need to navigate to your kiamol folder first.

The GitHub repository is the quickest way for me to publish any corrections to the exercises, so if you do have any problems you should check for a README file with updates in the chapter folder.

1.4.2 Install Docker Desktop

Docker Desktop runs on Windows 10 or macOS Sierra (version 10.12 or higher). Browse to https://www.docker.com/products/docker-desktop and choose to install the stable version. Download the installer and run it, accepting all the defaults. On Windows, that might include a reboot to add new Windows features. When Docker Desktop is running, you'll see Docker's whale icon near the clock on the Windows taskbar or the Mac menu bar. If you're an experienced Docker Desktop user on Windows, you'll need to make sure you're in Linux container mode (which is the default for new installations).

Kubernetes isn't set up by default so you'll need to click that whale icon to open the menu and click Settings. That opens the window you can see in figure 1.5; select Kubernetes from the menu and check the Enable Kubernetes box.

Docker Desktop downloads all the container images for the Kubernetes runtime, which might take a while, and then starts everything up. When you see the two green

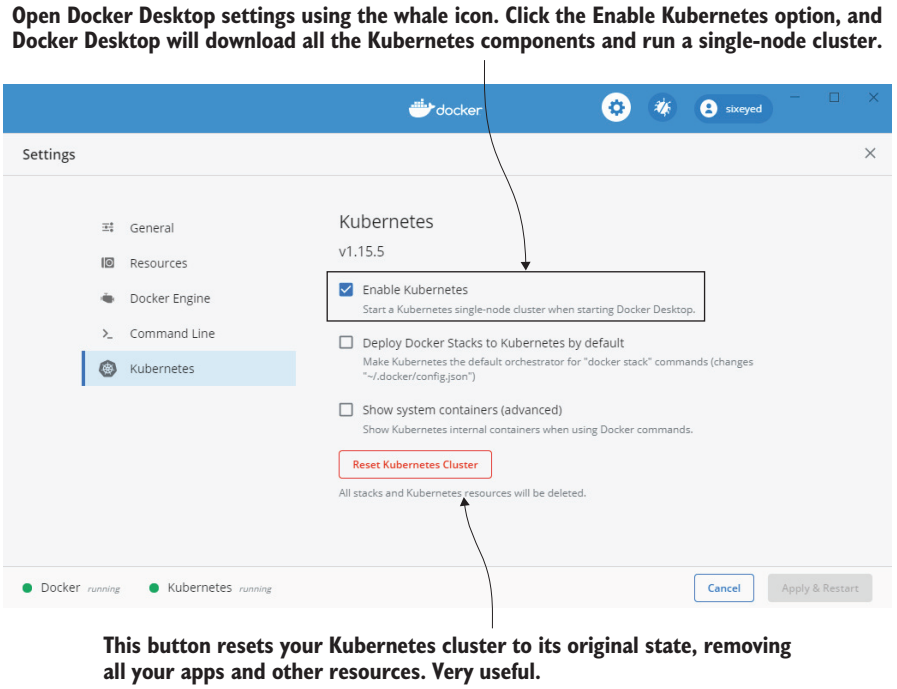

Figure 1.5 Docker Desktop creates and manages a Linux VM to run containers, and it can run Kubernetes.

dots at the bottom of the settings screen, then your Kubernetes cluster is ready to go. Docker Desktop installs everything else you need, so you can skip to section 1.4.7.

Other Kubernetes distributions can run on top of Docker Desktop but they don't integrate very well with the network setup which Docker Desktop uses, so you'll find problems running the exercises. The Kubernetes option in Docker Desktop has all the features you need for this book and is definitely the easiest option.

1.4.3 Install Docker Community Edition and K3s

If you're using a Linux machine or a Linux VM then there are several options for running a single-node cluster. Minikube and Kind are popular, but my preference is K3s, which is a minimal installation but has all the features you'll need for the exercises. (The name is a play on "K8s", which is an abbreviation of Kubernetes. K3s trims the Kubernetes codebase and the name indicates that it's half the size of K8s.)

K3s works with Docker and first you should install Docker Community Edition. You can check the full installation steps at https://rancher.com/docs/k3s/latest/en/quick-start/, but this will get you up and running:

```
# install Docker:
curl -fsSL https://get.docker.com | sh
```

```
# install K3s:
curl -sfL https://get.k3s.io | sh -s - --docker --disable=traefik --write-
     kubeconfig-mode=644
```

If you prefer to run your lab environment in a virtual machine, and you're familiar with using Vagrant to manage VMs, there is a Vagrant setup with Docker and K3s that you can use in the source repository for the book:

```
# from the root of the Kiamol repo:
cd ch01/vagrant-k3s

# provision the machine:
vagrant up

# and connect:
vagrant ssh
```

K3s installs everything else you need, so you can skip onto section 1.4.7.

1.4.4 Install the Kubernetes command-line tool

You manage Kubernetes with a tool called Kubectl (which is pronounced "cube-cuttle" as in "cuttlefish"—don't let anyone tell you different). It connects to a Kubernetes cluster and works with the Kubernetes API. If you're using Docker Desktop or K3s, they install Kubectl for you, but if you're using one of the other options here then you'll need to install it yourself.

The full installation instructions are at https://kubernetes.io/docs/tasks/tools/install-kubectl/. You can use Homebrew on the Mac and Chocolatey on Windows, and for Linux you can download the binary:

```
# macOS:
brew install kubernetes-cli

# OR Windows:
choco install kubernetes-cli

# OR Linux:
curl -Lo ./kubectl https://storage.googleapis.com/kubernetes-
     release/release/v1.18.8/bin/linux/amd64/kubectl
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin/kubectl
```

1.4.5 Verify your cluster

Now you have a running Kubernetes cluster, and whichever option you chose, they all work in the same way. Run this command to check that your cluster is up and running:

```
kubectl get nodes
```

You should see output like mine in figure 1.6. It's a list of all the nodes in your cluster, with some basic details like the status and Kubernetes version. The details of your cluster may be different, but as long as you see a node listed and in the ready state, then your cluster is good to go.

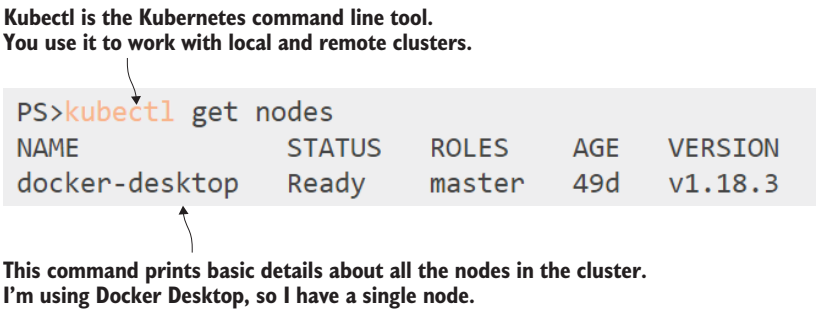

Figure 1.6 If you can run Kubectl and your nodes are ready, then you're all set to carry on.

1.5 Being immediately effective

"Immediately effective" is a core principle of the month of lunches series. In all, the focus is on learning skills and putting them into practice, in every chapter that follows.

Each chapter starts with a short introduction to the topic, followed by try-it-now exercises where you put the ideas into practice using your own Kubernetes cluster. Then there's a recap with some more detail to fill in some of the questions you may have from diving in. Last, there's a hands-on lab for you to try by yourself, to really gain confidence in your new understanding.

All the topics center around tasks which are genuinely useful in the real world. You'll learn how to be immediately effective with the topic during the chapter, and you'll finish by understanding how to apply the new skill. Let's start running some containerized apps!

2 Running containers in Kubernetes with Pods and Deployments

Kubernetes runs containers for your application workloads, but the containers themselves are not objects you need to work with. Every container belongs to a *Pod*, which is a Kubernetes object for managing one or more containers, and Pods in turn are managed by other resources. These higher-level resources abstract away the details of the container, which powers self-healing applications and lets you use a desired-state workflow—where you tell Kubernetes what you want to happen, and it decides how to make it happen.

In this chapter we'll get started with the basic building blocks of Kubernetes—Pods which run containers, and *Deployments* which manage Pods. We'll use a very simple web app for the exercises, and you'll get hands-on experience using the Kubernetes command line tool to manage applications and using the Kubernetes YAML specification to define applications.

2.1 How Kubernetes runs and manages containers

A container is a virtualized environment that typically runs a single application component. Kubernetes wraps the container in another virtualized environment: the Pod. A Pod is a unit of compute, which runs on a single node in the cluster. The Pod has its own virtual IP address which is managed by Kubernetes, and Pods in the cluster can communicate with other Pods over that virtual network, even if they're running on different nodes.

You normally run a single container in a Pod, but you can run multiple containers in one Pod, which opens up some very interesting deployment options. All the

containers in a Pod are part of the same virtual environment, so they share the same network address and can communicate using `localhost`. Figure 2.1 shows the relationship between containers and Pods.`

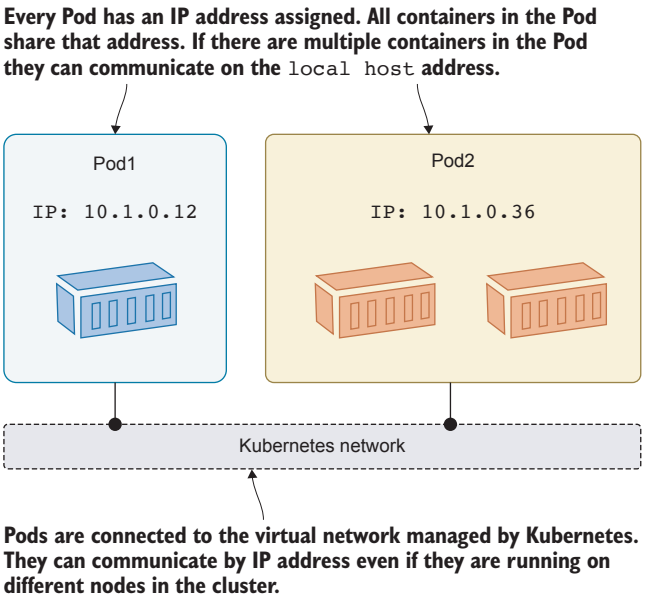

Figure 2.1 Containers run inside Pods. You manage the Pods and the Pods manage the containers.

This business of multi-container Pods is a bit much to introduce this early on. But if I glossed over it and only talked about single-container Pods you'd be rightfully asking why Kubernetes uses Pods at all instead of just containers. Let's run a Pod and see what it looks like to work with this abstraction over containers.

> **TRY IT NOW** You can run a simple Pod using the Kubernetes command line without needing a YAML specification. The syntax is similar to running a container using Docker—you state the container image you want to use, and any other parameters to configure the Pod behavior.

```
# run a Pod with a single container; the restart flag tells Kubernetes
# to create just the Pod and no other resources:
kubectl run hello-kiamol --image=kiamol/ch02-hello-kiamol --restart=Never

# wait for the pod to be ready:
kubectl wait --for=condition=Ready pod hello-kiamol

# list all the Pods in the cluster:
kubectl get pods

# show detailed information about the Pod:
kubectl describe pod hello-kiamol
```

You can see my output in figure 2.2, where I've abridged the response from the final describe Pod command. When you run it yourself, you'll see a whole lot more obscure-sounding information in there, like node selectors and tolerations. They're all part of the Pod specification, and Kubernetes has applied default values for everything that we didn't specify in the run command.

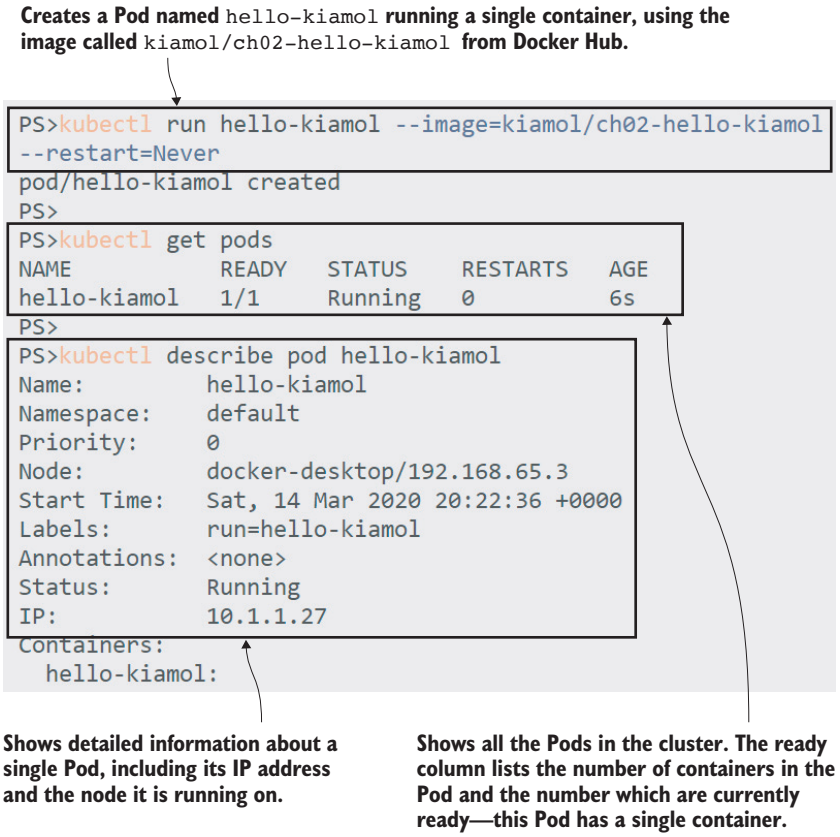

Figure 2.2 Running the simplest of Pods and checking its status using Kubectl.

Now you have a single application container in your cluster, running inside a single Pod. If you're used to working with Docker, then this is a very familiar workflow, and it turns out Pods are not as complicated as they might seem. The majority of your Pods will run single containers (until you start to explore more advanced options), and so you can effectively think of the Pod as the mechanism Kubernetes uses to run a container.

Kubernetes doesn't really run containers though—it passes the responsibility for that onto the container runtime installed on the node, which could be Docker or containerd or something more exotic. That's why the Pod is an abstraction, it's the

resource that Kubernetes manages whereas the container is managed by something outside of Kubernetes. You can get a sense of that by using Kubectl to fetch specific information about the Pod.

> **TRY IT NOW** Kubectl returns basic information from the get Pod command, but you can request more by applying an output parameter. You can name individual fields you want to see in the output parameter, and you can use the JSONPath query language or Go templates for complex output.

```
# get the basic information about the Pod:
kubectl get pod hello-kiamol

# specify custom columns in the output, selecting network details:
kubectl get pod hello-kiamol --output custom-
     columns=NAME:metadata.name,NODE_IP:status.hostIP,POD_IP:status.podIP

# specify a JSONPath query in the output,
# selecting the ID of the first container in the Pod:
kubectl get pod hello-kiamol -o
     jsonpath='{.status.containerStatuses[0].containerID}'
```

My output is in figure 2.3. I'm running a single-node Kubernetes cluster using Docker Desktop on Windows—the node IP in the second command is the IP address of my Linux VM, and the Pod IP is the virtual address of the Pod in the cluster. The container ID returned in the third command is prefixed by the name of the container runtime—mine is Docker.

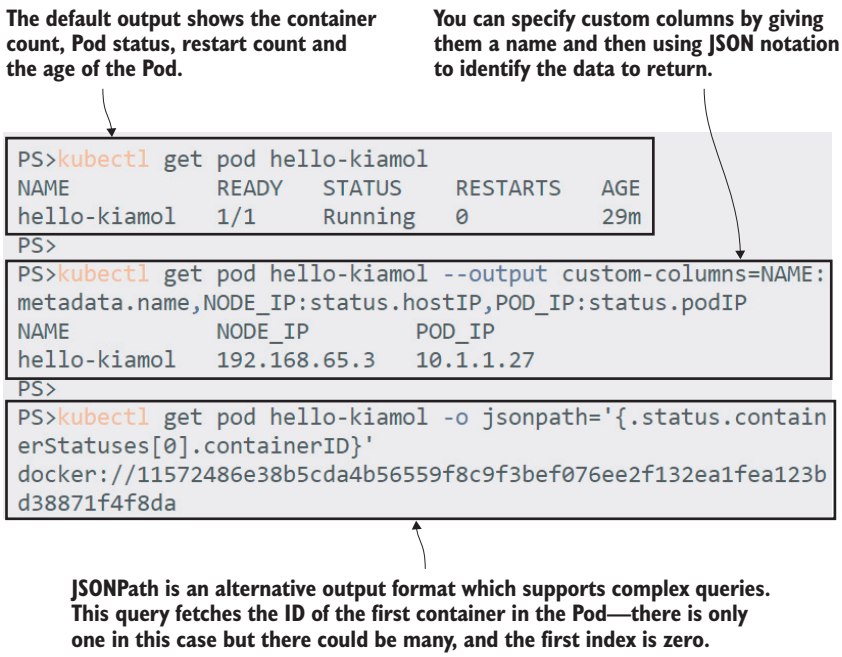

Figure 2.3 Kubectl has many options for customizing its output—for Pods and other objects.

That may have felt like a pretty dull exercise, but it comes with two important takeaways. The first is that Kubectl is a hugely powerful tool—it's your main point of contact with Kubernetes, you'll be spending a lot of time with it, so it's worth getting a solid understanding of what it can do. Querying the output from commands is a very useful way to see the information you care about, and because you can access all the details of the resource, it's great for automation too. The second takeaway is a reminder that Kubernetes does not run containers—the container ID in the Pod is a reference to another system that runs containers.

Pods are allocated to one node when they're created, and it's that node's responsibility to manage the Pod and its containers. It does that by working with the container runtime using a known API called the Container Runtime Interface (CRI). The CRI lets the node manage containers in the same way for all the different container runtimes—it uses a standard API to create and delete containers and to query their state. While the Pod is running, the node works with the container runtime to ensure the Pod has all the containers it needs.

TRY IT NOW All Kubernetes environments use the same CRI mechanism to manage containers, but not all container runtimes allow you to access containers outside of Kubernetes. This exercise shows you how a Kubernetes node keeps its Pod containers running, but you'll only be able to follow it if you're using Docker as your container runtime.

```
# find the Pod's container:
docker container ls -q --filter label=io.kubernetes.container.name=hello-
     kiamol

# now delete that container:
docker container rm -f $(docker container ls -q --filter
     label=io.kubernetes.container.name=hello-kiamol)

# check the Pod status:
kubectl get pod hello-kiamol

# and find the container again:
docker container ls -q --filter label=io.kubernetes.container.name=hello-
     kiamol
```

You can see from figure 2.4 that Kubernetes reacted when I deleted my Docker container. For an instant, the Pod had zero containers, but Kubernetes immediately created a replacement to repair the Pod and bring it back to the correct state.

It's the abstraction from containers to Pods that lets Kubernetes repair issues like this—a failed container is a temporary fault; the Pod still exists and the Pod can be brought back up to spec with a new container. This is just one level of self-healing that Kubernetes provides, with further abstractions on top of Pods giving your apps even more resilience.

One of those abstractions is the Deployment, which we'll look at in the next section—but before we move on let's see what's actually running in that Pod. It's a web application

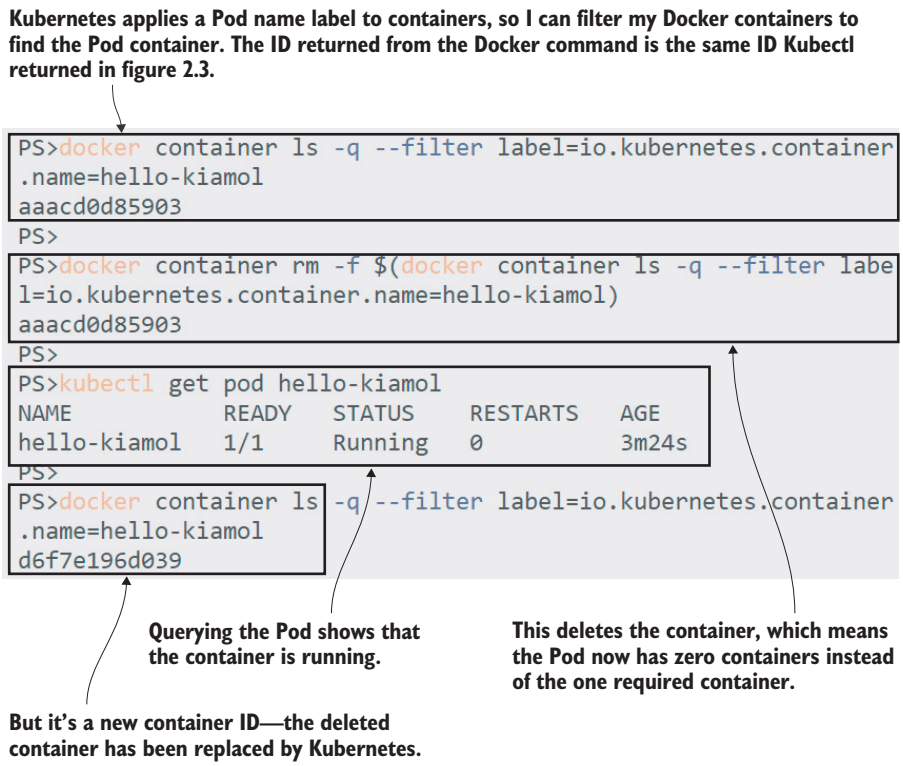

Figure 2.4 **Kubernetes makes sure Pods have all the containers they need.**

but you can't browse to it because we haven't configured Kubernetes to route network traffic into the Pod. We can get around that using another feature of Kubectl.

> **TRY IT NOW** Kubectl can forward traffic from a node into a Pod, which is a quick way to communicate with a Pod from outside the cluster. You can listen on a specific port on your machine—which is the single node in your cluster—and forward traffic into the application running in the Pod.

```
# listen on port 8080 on your machine and send traffic
# to the Pod on port 80:
kubectl port-forward pod/hello-kiamol 8080:80

# now browse to http://localhost:8080

# when you're done hit ctrl-c to end the port-forward
```

My output is in figure 2.5, and you can see it's a pretty basic website (don't contact me for web design consultancy). The web server and all the content is packaged into a container image on Docker Hub that is publicly available. All the CRI-compatible container runtimes can pull that image and run a container from it, so I know whichever Kubernetes environment you're using, when you run the app, it will work in the same way for you as it does for me.

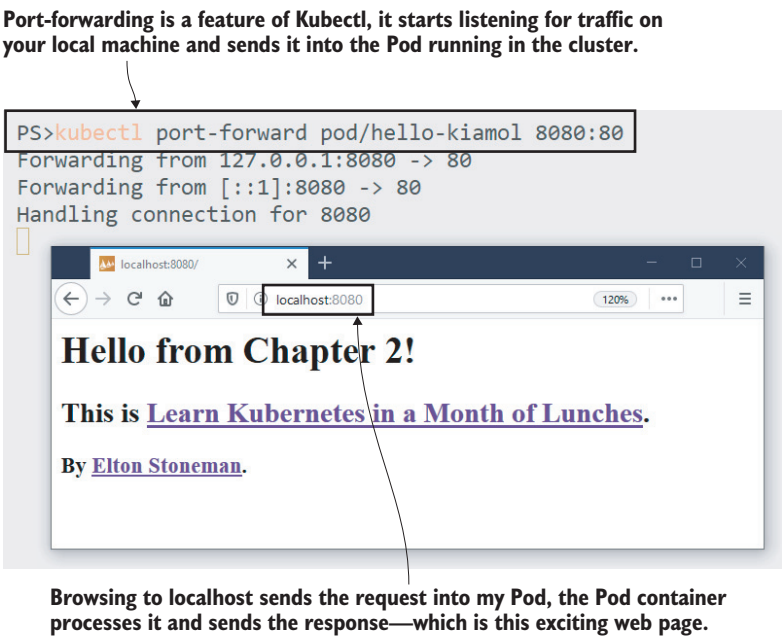

Figure 2.5 This app isn't configured to receive network traffic, but Kubectl can forward it.

Now we have a good handle on the Pod, which is the smallest unit of compute in Kubernetes. You need to understand how that all works, but the Pod is a primitive resource and in normal use you'd never run a Pod directly, you'd always create a controller object to manage the Pod for you.

2.2 Running Pods with controllers

So it's only the second section of the second chapter, and we're already onto a new Kubernetes object that is an abstraction over other objects. Kubernetes does get complicated quickly, but that complexity is a necessary part of such a powerful and configurable system. The learning curve is the entrance fee for access to a world-class container platform.

Pods are too simple to be useful on their own—they are isolated instances of an application, and each Pod is allocated to one node. If that node goes offline then the Pod is lost and Kubernetes does not replace it. You could try to get high availability by running several Pods, but there's no guarantee Kubernetes won't run them all on the same node. Even if you do get Pods spread across several nodes, you need to manage them yourself—and why do that when you have an orchestrator that can manage them for you?

That's where controllers come in. A *controller* is a Kubernetes resource that manages other resources—it works with the Kubernetes API to watch the current state of the system, compares that to the desired state of its resources, and makes any changes

it needs. There are many controllers, but the main one for managing Pods is the Deployment, which solves the problems I've just described. If a node goes offline and you lose the Pod, the Deployment will create a replacement Pod on another node. If you want to scale your Deployment, you can specify how many Pods you want and the Deployment controller will run them across many nodes. Figure 2.6 shows the relationship between Deployments, Pods, and containers.

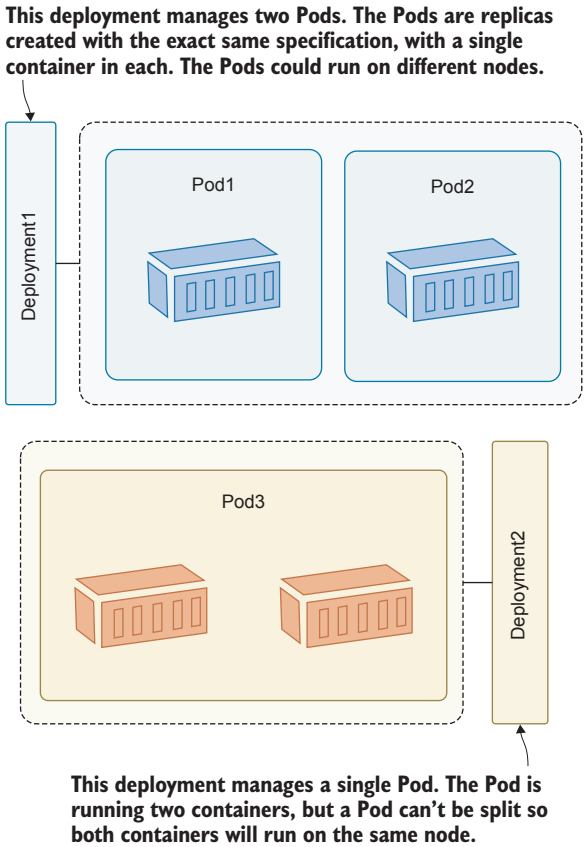

Figure 2.6 Deployment controllers manage Pods and Pods manage containers.

You can create Deployment resources with Kubectl, specifying the container image you want to run and any other configuration for the Pod. Kubernetes creates the Deployment, and the Deployment creates the Pod.

TRY IT NOW Create another instance of the web application, this time using a Deployment. The only required parameters are the name for the Deployment and the image to run.

```
# create a Deployment called "hello-kiamol-2", running the same web app:
kubectl create deployment hello-kiamol-2 --image=kiamol/ch02-hello-kiamol
```

```
# list all the Pods:
kubectl get pods
```

You can see my output in figure 2.7. Now you have two Pods in your cluster—the original one you created with the Kubectl run command, and the new one created by the Deployment. The Deployment-managed Pod has a name generated by Kubernetes, which is the name of the Deployment followed by a random suffix.

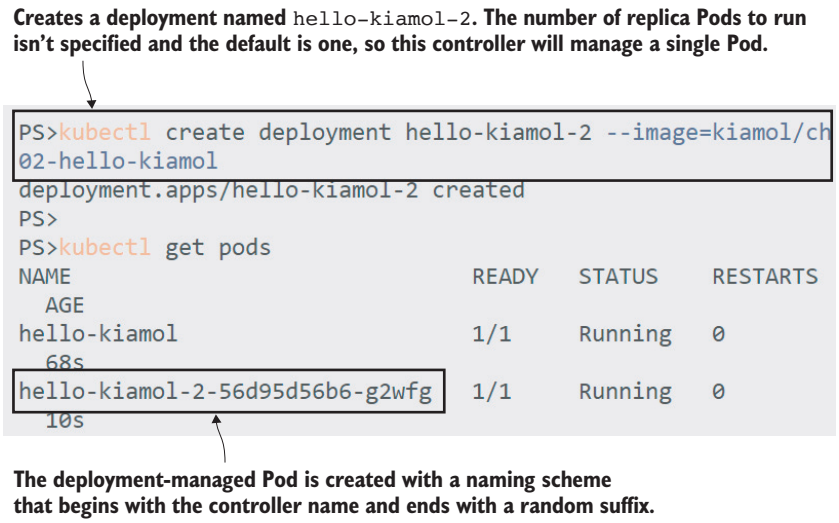

Figure 2.7 **Create a controller resource and it creates its own resources—Deployments create Pods.**

One important thing to realize from this exercise—you created the Deployment but you did not directly create the Pod. The Deployment specification described the Pod you wanted, and the Deployment created that Pod. The Deployment is a controller that checks with the Kubernetes API to see which resources are running, realizes the Pod it should be managing doesn't exist, and uses the Kubernetes API to create it. The exact mechanism doesn't really matter, you can just work with the Deployment and rely on it to create your Pod.

How the Deployment keeps track of its resources does matter though, because it's a pattern that Kubernetes uses a lot. Any Kubernetes resource can have labels applied which are simple key-value pairs. You can add labels to record your own data—you might add a label to a Deployment with the name release and the value 20.04 to indicate this Deployment is from the 20.04 release cycle. Kubernetes also uses labels to loosely couple resources, mapping the relationship between objects like a Deployment and its Pods.

TRY IT NOW The Deployment adds labels to Pods it manages. Use Kubectl to print out the labels the Deployment adds, and then list the Pods which match that label:

```
# print the labels which the Deployment adds to the Pod:
kubectl get deploy hello-kiamol-2 -o
     jsonpath='{.spec.template.metadata.labels}'

# list Pods which have that matching label:
kubectl get pods -l app=hello-kiamol-2
```

My output is in figure 2.8, where you can see some internals of how the resources are configured. Deployments use a template to create Pods, and part of that template is a metadata field which includes the labels for the Pod(s). In this case, the Deployment adds a label called app with the value `hello-kiamol-2` to the Pod. Querying Pods that have a matching label returns the single Pod managed by the Deployment.

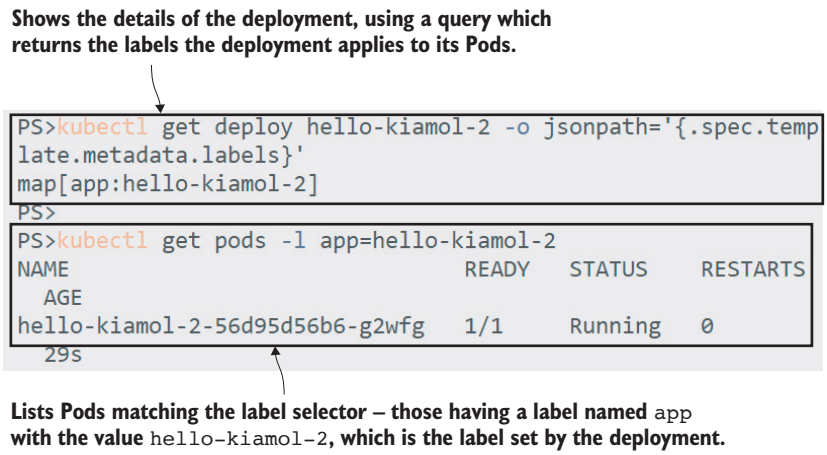

Figure 2.8 Deployments add labels when they create Pods, and you can use those labels as filters.

Using labels to identify the relationship between resources is such a core pattern in Kubernetes that it's worth a diagram to make sure it's clear. Resources can have labels applied at creation, and then added, removed, or edited during their lifetime. Controllers use a label selector to identify the resources they manage—and that can be a simple query matching resources with a particular label, as you see in figure 2.9.

This is very flexible because it means controllers don't need to maintain a list of all the resources they manage; the label selector is part of the controller specification and controllers can find matching resources at any time by querying the Kubernetes API. It's also something you need to be careful with because you can edit the labels for a resource and end up breaking the relationship between it and its controller.

TRY IT NOW The Deployment doesn't have a direct relationship with the Pod it created, it only knows there needs to be one Pod with labels that match its label selector. Edit the labels on the Pod and the Deployment no longer recognizes it.

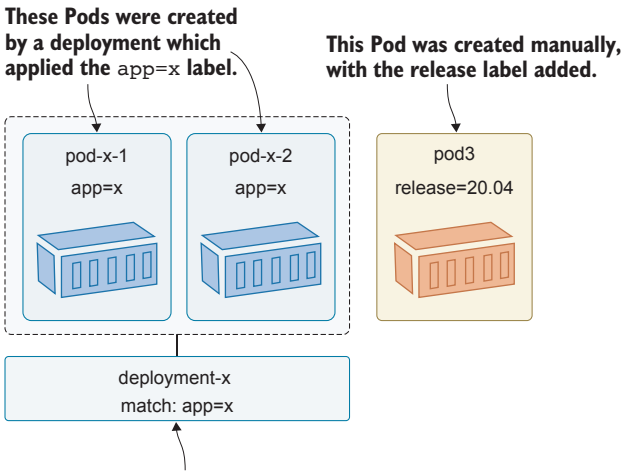

Figure 2.9 Controllers identify the resources they manage using labels and selectors.

```
# list all Pods, showing the Pod name and labels:
kubectl get pods -o custom-columns=NAME:metadata.name,LABELS:metadata.labels

# update the "app" label for the Deployment's Pod:
kubectl label pods -l app=hello-kiamol-2 --overwrite app=hello-kiamol-x

# fetch Pods again:
kubectl get pods -o custom-columns=NAME:metadata.name,LABELS:metadata.labels
```

What did you expect to happen? You can see from my output in figure 2.10 that changing the Pod label effectively removes the Pod from the Deployment. At that point, the Deployment sees that no Pods exist which match its label selector, so it creates a new one. The Deployment has done its job, but by editing the Pod directly you now have an unmanaged Pod.

This can be a useful technique in debugging, which is removing a Pod from a controller so you can connect and investigate a problem, while the controller starts a replacement Pod that keeps your app running at the desired scale. You can also do the opposite, editing the labels on a Pod to fool a controller into acquiring that Pod as part of the set it manages.

TRY IT NOW Return the original Pod to the control of the Deployment by setting its app label back so it matches the label selector.

```
# list all Pods with a label called "app", showing the Pod name and labels:
kubectl get pods -l app -o custom-
    columns=NAME:metadata.name,LABELS:metadata.labels

# update the "app" label for the the unmanaged Pod:
kubectl label pods -l app=hello-kiamol-x --overwrite app=hello-kiamol-2
```

```
# fetch Pods again:
kubectl get pods -l app -o custom-
     columns=NAME:metadata.name,LABELS:metadata.labels
```

Shows Pods with the Pod name and all the labels—labels are shown with names and values separated by colons.

```
PS>kubectl get pods -o custom-columns=NAME:metadata.name,LABE
LS:metadata.labels
NAME                              LABELS
hello-kiamol                      map[run:hello-kiamol]
hello-kiamol-2-56d95d56b6-g2wfg   map[app:hello-kiamol-2 pod-
template-hash:56d95d56b6]
PS>
PS>kubectl label pods -l app=hello-kiamol-2 --overwrite app=h
ello-kiamol-x
pod/hello-kiamol-2-56d95d56b6-g2wfg labeled
PS>
PS>kubectl get pods -o custom-columns=NAME:metadata.name,LABE
LS:metadata.labels
NAME                              LABELS
hello-kiamol                      map[run:hello-kiamol]
hello-kiamol-2-56d95d56b6-84v67   map[app:hello-kiamol-2 pod-
template-hash:56d95d56b6]
hello-kiamol-2-56d95d56b6-g2wfg   map[app:hello-kiamol-x pod-
template-hash:56d95d56b6]
```

Finds all the Pods which have the `app` label set with the value `hello-kiamol-2` and overwrites that label with the value `hello-kiamol-x`. This effectively breaks the link between the Pod and the deployment.

Listing Pods again shows that the deployment has created a new Pod to replace the one it lost when the label was changed.

Figure 2.10 If you meddle with the labels on a Pod, you can remove it from the control of the Deployment.

This exercise effectively reverses the previous exercise, setting the app label back to hello-kiamol-2 for the original Pod in the Deployment. Now when the Deployment controller checks with the API, it sees two Pods that match its label selector—but it's only supposed to manage a single Pod, so it deletes one (using a set of deletion rules to decide which one). You can see in figure 2.11 that the Deployment removed the second Pod and retained the original:

Pods run your application containers, but just like containers, Pods are meant to be short-lived. You will almost always use a higher-level resource like a Deployment to manage Pods for you. That gives Kubernetes a better chance of keeping your app running if there are issues with containers or nodes, but ultimately the Pods are running

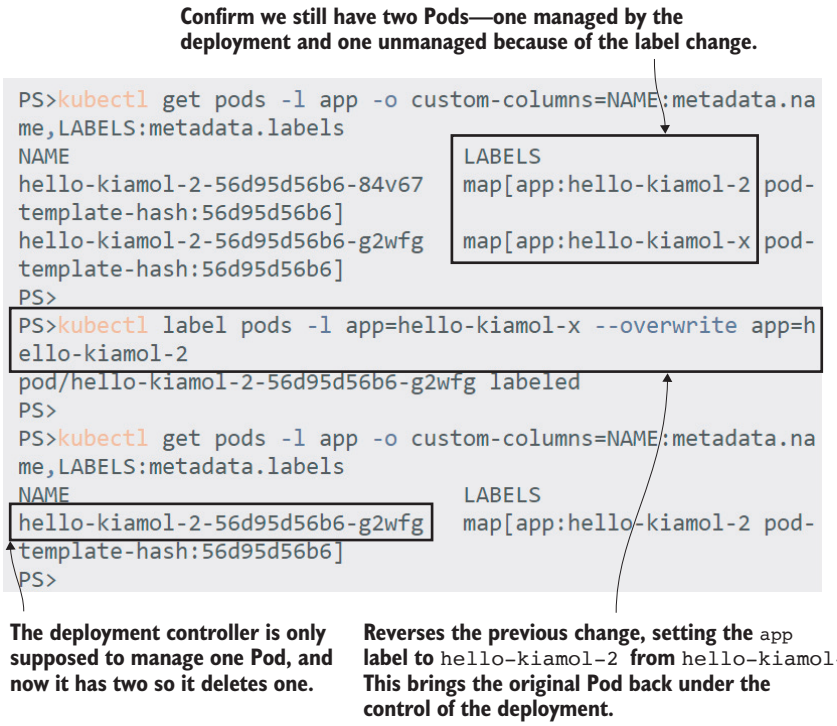

Figure 2.11 More label meddling—you can force a Deployment to adopt a Pod if the labels match.

the same containers you would run yourself, and the end-user experience for your apps will be the same.

> **TRY IT NOW** Kubectl's port forward command sends traffic into a Pod, but you don't have to find the random Pod name for a Deployment—you can configure the port forward on the Deployment resource, and the Deployment selects one of its Pods as the target.

```
# run a port forward from your local machine to the Deployment:
kubectl port-forward deploy/hello-kiamol-2 8080:80

# browse to http://localhost:8080

# when you're done, exit with ctrl-c
```

You can see my output in figure 2.12—the same app running in a container from the same Docker image, but this time in a Pod managed by a Deployment.

Pods and Deployments are the only resources we'll cover in this chapter. You can deploy very simple apps by using the Kubectl run and create commands, but more complex apps need lots more configuration, and those commands won't do. It's time to enter the world of Kubernetes YAML.

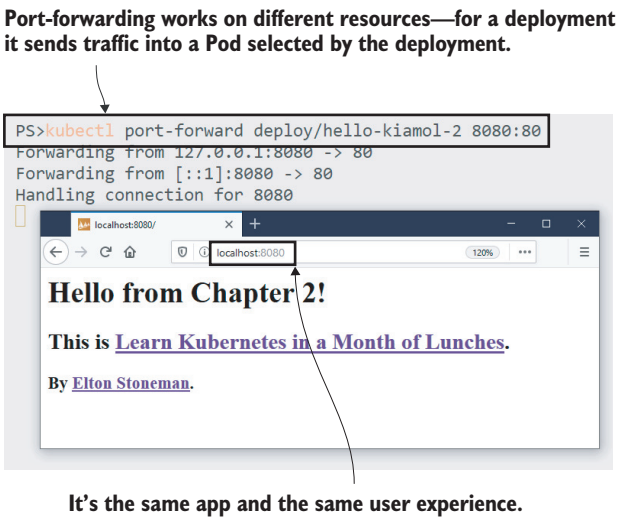

Figure 2.12 Pods and Deployments are layers on top of containers, but the app still runs in a container.

2.3 Defining Deployments in application manifests

Application manifests are one of the most attractive aspects of Kubernetes, but also one of the most frustrating. When you're wading through hundreds of lines of YAML trying to find the small misconfiguration that has broken your app, it can seem like the API was deliberately written to confuse and irritate you. At those times, you need to remember that Kubernetes manifests are a complete description of your app, which can be versioned and tracked in source control and will result in the same Deployment on any Kubernetes cluster.

Manifests can be written in JSON or YAML. JSON is the native language of the Kubernetes API, but YAML is preferred for manifests because it's easier to read, lets you define multiple resources in a single file, and—most importantly—can record comments in the specification. Listing 2.1 is the simplest app manifest you can write. It defines a single Pod using the same container image we've already used in this chapter.

Listing 2.1 pod.yaml: a single Pod to run a single container

```
# manifests always specify the version of the Kubernetes API
# and the type of resource
apiVersion: v1
kind: Pod

# metadata for the resource includes the name (mandatory)
# and labels (optional)
metadata:
  name: hello-kiamol-3

# the spec is the actual specification for the resource
# for a Pod the minimum is the container(s) to run,
# with container name and image
```

```
spec:
  containers:
   –name: web
      image: kiamol/ch02-hello-kiamol
```

That's a lot more information than you need for a Kubectl `run` command, but the big advantage of the application manifest is that it's declarative. Kubectl `run` and Kubectl `create` are imperative operations, it's you telling Kubernetes to do something. Manifests are declarative—you tell Kubernetes what you want the end result to be, and it goes off and decides what it needs to do to make that happen.

TRY IT NOW You still use Kubectl to deploy apps from manifest files, but you use the `apply` command which tells Kubernetes to apply the configuration in the file to the cluster. Run another Pod for this chapter's sample app using a YAML file with the same contents as listing 2.1.

```
# switch from the root of the kiamol repository to the chapter 2 folder:
cd ch02

# deploy the application from the manifest file:
kubectl apply -f pod.yaml

# list running Pods:
kubectl get pods
```

The new Pod works in the same way as a Pod created with the Kubectl run command—it's allocated to a node and it runs a container. My output in figure 2.13 shows that when I applied the manifest, Kubernetes decided it needed to create a Pod to get the current state of the cluster up to my desired state. That's because the manifest specifies a Pod named hello-kiamol-3, and no such Pod existed.

The apply command tells Kubernetes to apply the state described in the YAML file to the cluster. Kubectl shows the actions taken, in this case creating a single Pod.

```
PS>cd ch02
PS>
PS>kubectl apply -f pod.yaml
pod/hello-kiamol-3 created
PS>
PS>kubectl get pods
NAME                                READY   STATUS    RESTARTS
  AGE
hello-kiamol                        1/1     Running   0
  123m
hello-kiamol-2-56d95d56b6-g2wfg     1/1     Running   0
  122m
hello-kiamol-3                      1/1     Running   0
  8s
```

These three Pods all have the same specification and are running the same app, but they were all created in different ways.

Figure 2.13 Applying a manifest sends the YAML file to the Kubernetes API, which applies changes.

Now that the Pod is running, you can manage it in the same way with Kubectl—listing the details of the Pod and running a port-forward to send traffic into the Pod. The big difference is that the manifest is easy to share, and manifest-based deployment is repeatable. I can run the same Kubectl apply command with the same manifest any number of times and the result will always be the same—a Pod named hello-kiamol-3 running my web container.

TRY IT NOW Kubectl doesn't even need a local copy of a manifest file—it can read the contents from any public URL. Deploy the same Pod definition direct from the file on GitHub.

```
# deploy the application from the manifest file:
kubectl apply -f
    https://raw.githubusercontent.com/sixeyed/kiamol/master/ch02/pod.yaml
```

You'll see the very simple output in figure 2.14—the resource definition matches the Pod running in the cluster, so Kubernetes doesn't need to do anything, and Kubectl shows that the matching resource is unchanged.

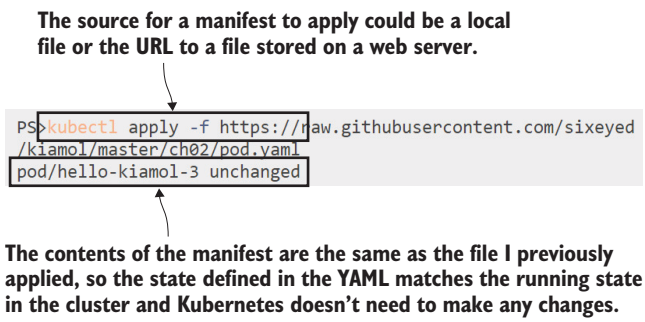

Figure 2.14 Kubectl can download manifest files from a web server and send them to the Kubernetes API.

Application manifests start to get more interesting when you work with higher-level resources. You can define a Deployment in a YAML file, and one of the required fields is the specification of the Pod which the Deployment should run. That Pod specification is the same API for defining a Pod on its own, so the Deployment definition is a composite that includes the Pod spec. Listing 2.2 shows the minimal definition for a Deployment resource, running yet another version of the same web app.

Listing 2.2 deployment.yaml: a Deployment and Pod specification

```
# Deployments are part of the "apps" version 1 API spec
apiVersion: apps/v1
kind: Deployment

# the Deployment needs a name
metadata:
  name: hello-kiamol-4

# the spec includes the label selector the Deployment uses
```

```
# to find its own managed resources—I'm using the "app" label,
# but this could be any combination of key-value pairs
spec:
  selector:
    matchLabels:
      app: hello-kiamol-4

  # the template is used when the Deployment creates Pods
  template:

    # Pods in a Deployment don't  have a name,
    # but they need to specify labels which match the selector
    metadata:
      labels:
        app: hello-kiamol-4

    # the Pod spec lists the container name and image
    spec:
      containers:
       —name: web
          image: kiamol/ch02-hello-kiamol
```

This manifest is for a completely different resource (which just happens to run the same application), but all Kubernetes manifests are deployed in the same way using Kubectl apply. That gives you a very nice layer of consistency across all your apps—no matter how complex they are, you'll define them in one or more YAML files, and deploy them using the same Kubectl command.

TRY IT NOW Apply the Deployment manifest which will create a new Deployment, which in turn will create a new Pod.

```
# run the app using the Deployment manifest:
kubectl apply -f deployment.yaml

# find Pods managed by the new Deployment:
kubectl get pods -l app=hello-kiamol-4
```

You can see my output in figure 2.15, it's the same end result as creating a Deployment with Kubectl create, but my whole app specification is clearly defined in a single YAML file.

As the app grows in complexity and I need to specify how many replicas I want, what CPU and memory limits should apply, how Kubernetes can check whether the app is healthy, where the application configuration settings come from, and where it writes data. I do all that just by adding to the YAML.

2.4 Working with applications in Pods

Pods and Deployments are there to keep your app running, but all the real work is happening in the container. Your container runtime may not give you access to work with containers directly—a managed Kubernetes cluster won't give you control of Docker or containerd—but you can still work with containers in Pods using Kubectl.

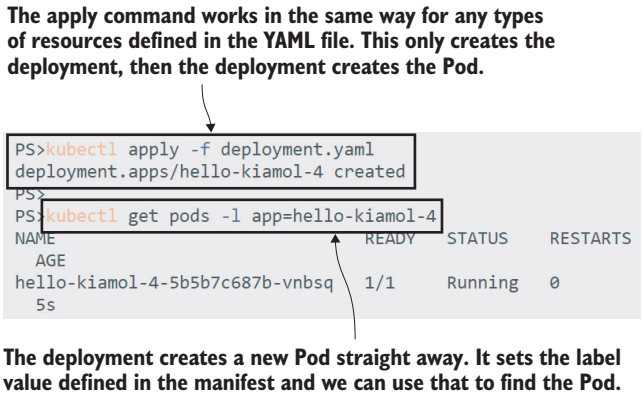

Figure 2.15 Applying a manifest creates the Deployment because no matching resource existed.

The Kubernetes command line lets you run commands in containers, view application logs, and copy files.

TRY IT NOW You can run commands inside containers with Kubectl, and connect a terminal session, so you can connect into a Pod's container as though you were connecting to a remote machine.

```
# check the internal IP address of the first Pod we ran:
kubectl get pod hello-kiamol -o custom-
     columns=NAME:metadata.name,POD_IP:status.podIP

# run an interactive shell command in the Pod:
kubectl exec -it hello-kiamol -- sh

# inside the Pod check the IP address:
hostname -i

# and test the web app:
wget -O—http://localhost | head -n 4

# leave the shell:
exit
```

My output is in figure 2.16, where you can see that the IP address in the container environment is the one set by Kubernetes, and the web server running in the container is available at the localhost address.

Running an interactive shell inside a Pod container is a useful way of seeing how the world looks to that Pod. You can read file contents to check that configuration settings are being applied correctly, run DNS queries to check services are resolving as expected, and ping endpoints to test the network. Those are all good troubleshooting techniques, but for ongoing administration a simpler option is to read the application logs, and Kubectl has a dedicated command just for that.

TRY IT NOW Kubernetes fetches application logs from the container runtime. You can always read logs with Kubectl, and if you have access to the container runtime you can verify that they are the same as the container logs.

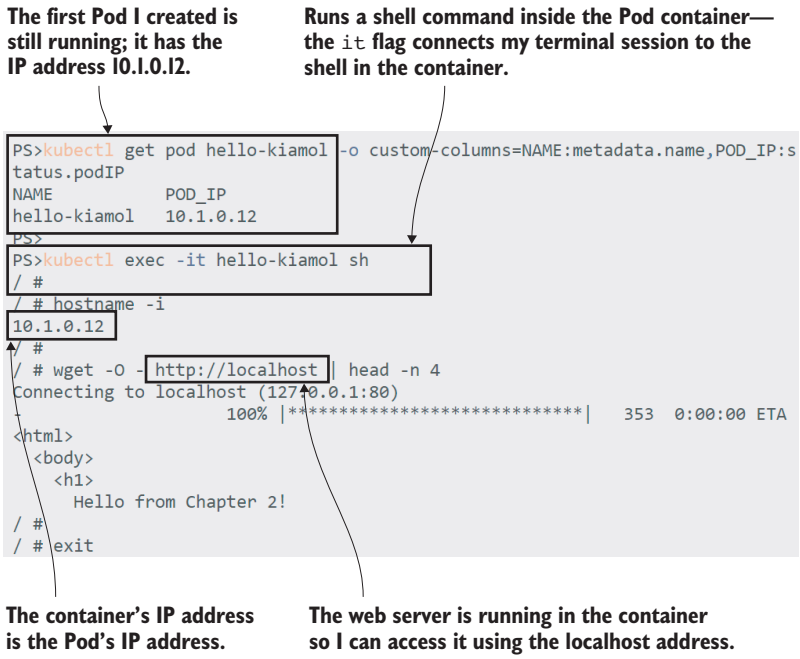

Figure 2.16 **You can use Kubectl to run commands inside Pod containers, including interactive shells.**

```
# print latest container logs from Kubernetes:
kubectl logs --tail=2 hello-kiamol

# and compare the actual container logs - if you're using Docker:
docker container logs --tail=2 $(docker container ls -q --filter
     label=io.kubernetes.container.name=hello-kiamol)
```

You can see from my output in figure 2.17 that Kubernetes just relays log entries exactly as they come from the container runtime.

The same features are available for all Pods, no matter how they were created. Pods that are managed by controllers have random names, so you don't refer to them directly; instead, you can access them by their controller or by their labels.

> **TRY IT NOW** You can run commands in Pods which are managed by a Deployment without knowing the Pod name, and you can view the logs of all Pods that match a label selector.

```
# make a call to the web app inside the container for the
# Pod we created from the Deployment YAML file:
kubectl exec deploy/hello-kiamol-4 -- sh -c 'wget -O—http://localhost >
     /dev/null'

# and check that Pod's logs:
kubectl logs --tail=1 -l app=hello-kiamol-4
```

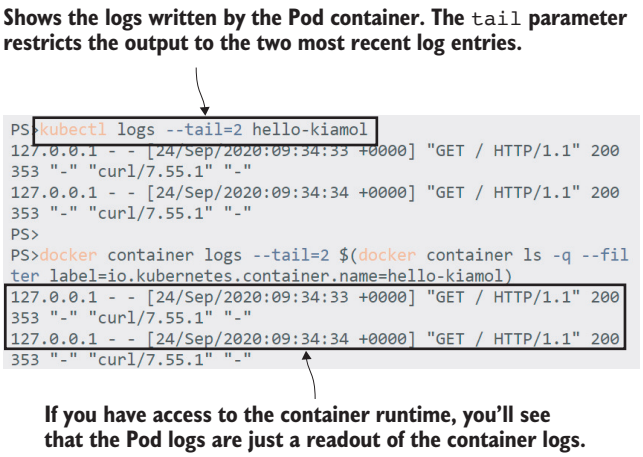

Figure 2.17 Kubernetes reads logs from the container so you don't need access to the container runtime.

Figure 2.18 shows the command running inside the Pod container, which causes the application to write a log entry and we see that in the Pod logs.

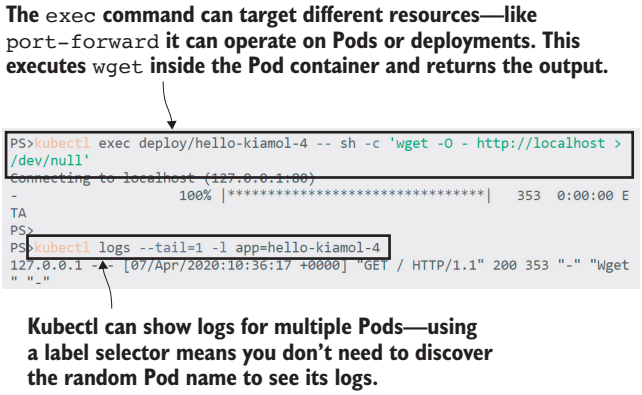

Figure 2.18 You can work with Pods using Kubectl without knowing the Pod's name.

In a production environment you can have all the logs from all of your Pods collected and sent to a central storage system, but until you get there this is a useful and easy way to read out application logs. You also saw in that exercise that there are different ways to get to Pods that are managed by a controller. Kubectl lets you supply a label selector to most commands, and some commands—like `exec`—can be run against different targets.

The last function you're likely to use with Pods is to interact with the filesystem. Kubectl lets you copy files between your local machine and containers in Pods.

TRY IT NOW Create a temporary directory on your machine and copy a file into it from the Pod container.

```
# create the local directory:
mkdir -p /tmp/kiamol/ch02
```

```
# copy the web page from the Pod:
kubectl cp hello-kiamol:/usr/share/nginx/html/index.html
     /tmp/kiamol/ch02/index.html

# check the local file contents:
cat /tmp/kiamol/ch02/index.html
```

In figure 2.19 you can see that Kubectl copies the file from the Pod container onto my local machine. This works whether your Kubernetes cluster is running locally or on remote servers, and it's bi-directional so you can use the same command to copy a local file into a Pod. That can be a useful—if hacky—way to work around an application problem.

Figure 2.19 Copying files between Pod containers and the local machine is useful for troubleshooting.

That's about all we're going to cover in this chapter, but before we move on, we need to delete the Pods we have running, and that is a little bit more involved than you might think.

2.5 Understanding Kubernetes resource management

You can easily delete a Kubernetes resource using Kubectl, but the resource might not stay deleted. If you created a resource with a controller then it's the controller's job to manage that resource—it owns the resource lifecycle and it doesn't expect any external interference. If you delete a managed resource then its controller will create a replacement.

> **TRY IT NOW** Use the Kubectl delete command to remove all Pods, and check that they're really gone.

```
# list all running Pods:
kubectl get pods

# delete all Pods:
kubectl delete pods --all
```

```
# check again:
kubectl get pods
```

You can see my output in figure 20.20. Is it what you expected?

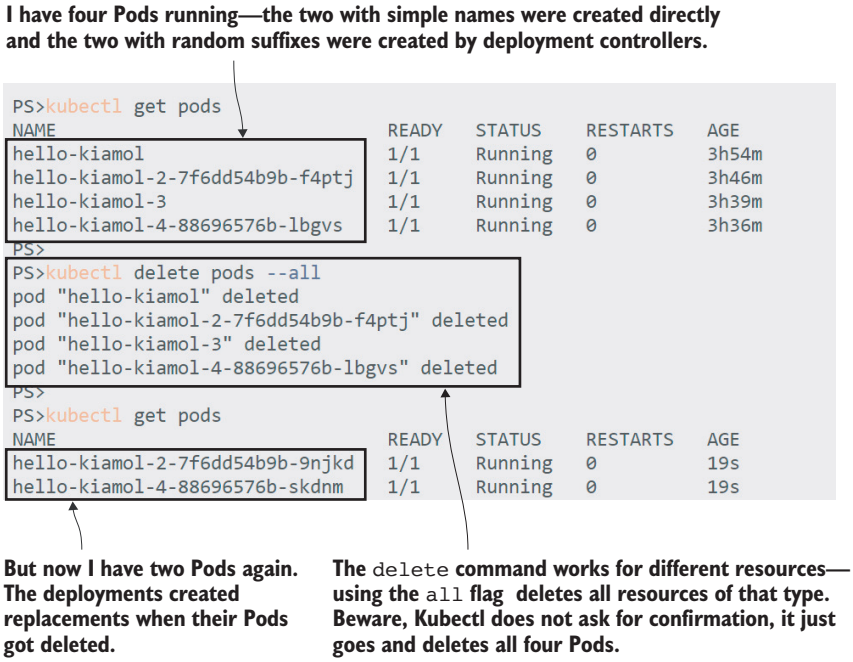

Figure 2.20 Controllers own their resources. If something else deletes them, the controller replaces them.

Two of those Pods were created directly: with the run command and with a YAML Pod specification. They don't have a controller managing them, so when you delete them, they stay deleted. The other two were created by Deployments, and when you delete the Pod the Deployment controllers still exist. They see there are no Pods that match their label selectors, so they create new ones.

It seems obvious when you know about it, but it's a gotcha that will probably keep cropping up through all your days with Kubernetes. If you want to delete a resource that is managed by a controller, then you need to delete the controller instead. Controllers clean up their resources when they get deleted, so removing a Deployment is like a cascading delete that removes all the Deployment's Pods too.

TRY IT NOW Check the Deployments you have running, then delete them and check the remaining Pods have been deleted.

```
# view Deployments:
kubectl get deploy

# delete all Deployments:
```

```
kubectl delete deploy --all

# view Pods:
kubectl get pods

# check all resources:
kubectl get all
```

Figure 2.21 shows my output—I was fast enough to see the Pods being removed, so they're shown in the terminating state. A few seconds later the Pods and the Deployment have been removed, so the only resource I have running is the Kubernetes API itself.

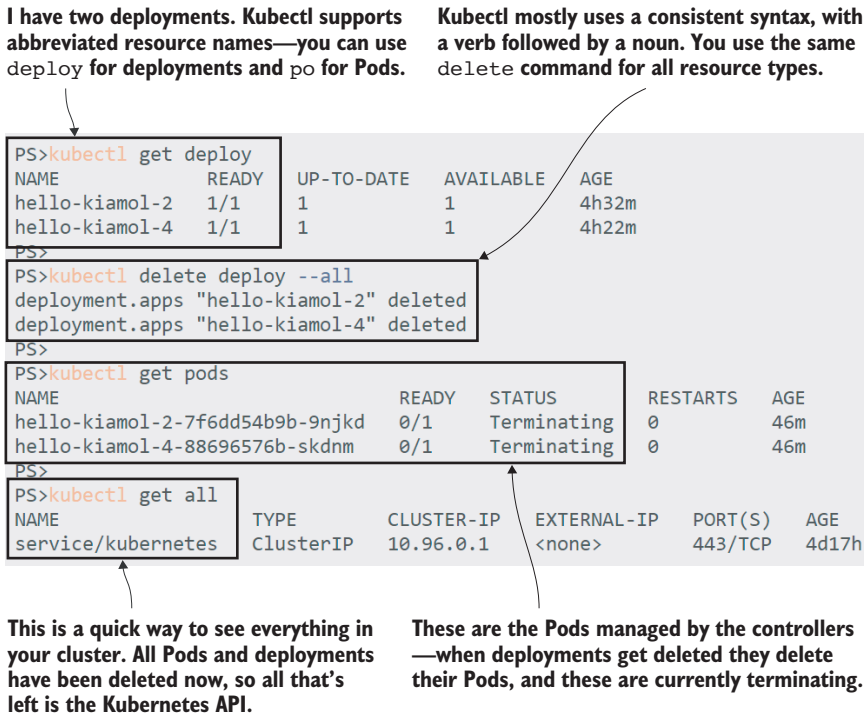

Figure 2.21 Deleting controllers starts a cascade effect, where the controller deletes all its resources.

Now your Kubernetes cluster isn't running any applications and it's back to its original state. We've covered a lot in this chapter—you've got a good understanding of how Kubernetes manages containers with Pods and Deployments, had an introduction to YAML specifications and had lots of experience using Kubectl to work with the Kubernetes API. We've built on the core concepts gradually, but you probably have a fair idea now that Kubernetes is a complex system. If you have time to go through the lab which follows, that will certainly help cement what you've learned.

2.6 Lab

This is your first lab—it's a guided challenge for you to complete yourself. The goal is to write a Kubernetes YAML spec for a Deployment which will run an application in a Pod, then test the app and make sure it runs as expected. Here are a few hints to get you started:

- In the ch02/lab folder, there's a file called pod.yaml which you can try out—it runs the app but it defines a Pod rather than a Deployment.
- The application container runs a website that listens on port 80.
- When you forward traffic to the port, the web app responds with the hostname of the machine it's running on.
- That hostname is actually the Pod name, which you can verify using Kubectl.

If you find this a bit tricky, I have a sample solution on GitHub which you can use for reference:

https://github.com/sixeyed/kiamol/blob/master/ch02/lab/README.md

3 Connecting Pods over the network with Services

Pods are the basic building blocks of an application running in Kubernetes. Most applications are distributed across multiple components, and you model those in Kubernetes using a Pod for each component, so you may have a website Pod and an API Pod, or you might have dozens of Pods in a microservice architecture. They all need to communicate, and Kubernetes supports the standard networking protocols, TCP and UDP. Both protocols use IP addresses to route traffic, but IP addresses change when Pods are replaced, so Kubernetes provides a network address discovery mechanism with *Services*.

Services are flexible resources which support routing traffic between Pods, into Pods from the world outside the cluster, and from Pods to external systems. In this chapter you'll learn all the different Service configurations Kubernetes provides to glue systems together and you'll understand how they work transparently for your apps.

3.1 How Kubernetes routes network traffic

You learned two important things about Pods in the last chapter: a Pod is a virtual environment that has an IP address assigned by Kubernetes, and Pods are disposable resources whose lifetime is controlled by another resource. If one Pod wants to communicate with another, it can use the IP address but that's problematic for two reasons—first because the IP address will change if the Pod gets replaced, and second because there's no easy way to find a Pod's IP address; it can only be discovered using the Kubernetes API.

TRY IT NOW You can see that if you deploy two Pods—you can ping one Pod from the other, but you first need to find its IP address.

```
# start up your lab environment—run Docker Desktop if it's not running -
# and switch to this chapter's directory in your copy of the source code:
cd ch03

# create two deployments, which each run one Pod:
kubectl apply -f sleep/sleep1.yaml -f sleep/sleep2.yaml

# wait for the Pod to be ready:
kubectl wait --for=condition=Ready pod -l app=sleep-2

# check the IP address of the second Pod:
kubectl get pod -l app=sleep-2 --output jsonpath='{.items[0].status.podIP}'

# use that address to ping the second Pod from the first:
kubectl exec deploy/sleep-1 -- ping -c 2 $(kubectl get pod -l app=sleep-2 --
     output jsonpath='{.items[0].status.podIP}')
```

You can see my output in figure 3.1—the ping inside the container works fine and the first Pod is able to successfully reach the second Pod—but I had to find the IP address using Kubectl and pass it into the ping command.

You can pass multiple files to the Kubectl `apply` command. This deploys two Pods which don't do anything.

The JSONPath query here returns just the IP address of the `sleep-2` Pod

```
PS>cd ch03
PS>
PS>kubectl apply -f sleep/sleep1.yaml -f sleep/sleep2.yaml
deployment.apps/sleep-1 created
deployment.apps/sleep-2 created
PS>
PS>kubectl get pod -l app=sleep-2 --output jsonpath='{.items[0].status.podIP}'
10.1.0.76
PS>
PS>kubectl exec deploy/sleep-1 -- ping -c 2 $(kubectl get pod -l app=sleep-2 --
output jsonpath='{.items[0].status.podIP}')
PING 10.1.0.76 (10.1.0.76): 56 data bytes
64 bytes from 10.1.0.76: seq=0 ttl=64 time=0.076 ms
64 bytes from 10.1.0.76: seq=1 ttl=64 time=0.122 ms

--- 10.1.0.76 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.076/0.099/0.122 ms
```

Pods can reach each other over the network using IP address, so the ping command works correctly.

This uses the previous command as input to the Kubectl `exec` command, passing the `sleep-2` Pod IP address to the ping command.

Figure 3.1 Pod networking with IP addresses—you can only discover an address from the Kubernetes API.

The virtual network in Kubernetes spans the whole cluster, so Pods can communicate by IP address even if they're running on different nodes—this example works in the same way on a single-node K3s cluster and a 100-node AKS cluster. It's a useful exercise to help you see that Kubernetes doesn't do any special networking magic, it just uses the standard protocols your apps already use. But you wouldn't normally do this, because the IP address is specific to one Pod, and when the Pod gets replaced, the replacement will have a new IP address.

> **TRY IT NOW** These Pods are managed by deployment controllers. If you delete the second Pod, its controller will start a replacement that has a new IP address.

```
# check the current Pod's IP address:
kubectl get pod -l app=sleep-2 --output jsonpath='{.items[0].status.podIP}'

# delete the Pod so the deployment replaces it:
kubectl delete pods -l app=sleep-2

# check the IP address of the replacement Pod:
kubectl get pod -l app=sleep-2 --output jsonpath='{.items[0].status.podIP}'
```

In figure 3.2 you can see my output—the replacement Pod has a different IP address, and if I tried to ping the old address the command would fail.

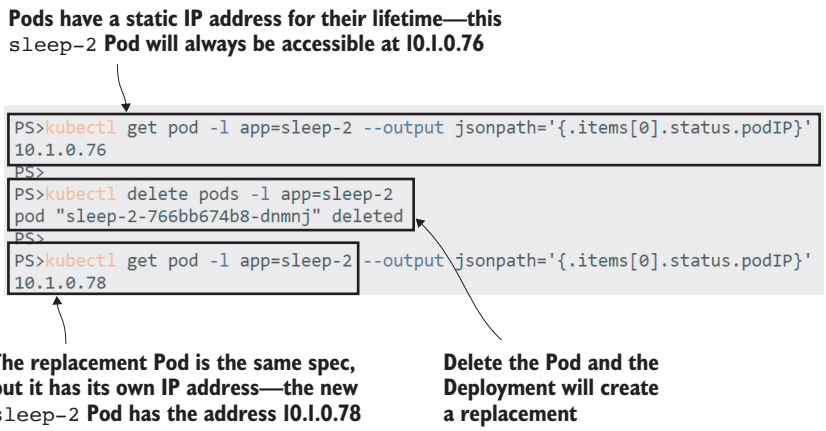

Figure 3.2 The Pod IP address is not part of its specification; a replacement Pod has a new address.

The problem of needing a permanent address for resources which can change is an old one—the Internet solved it using DNS (the Domain Name System), which maps friendly names to IP addresses, and Kubernetes uses the same system. A Kubernetes cluster has a DNS server built in, which maps Service names to IP addresses. Figure 3.3 shows how a domain name lookup works for Pod-to-Pod communication.

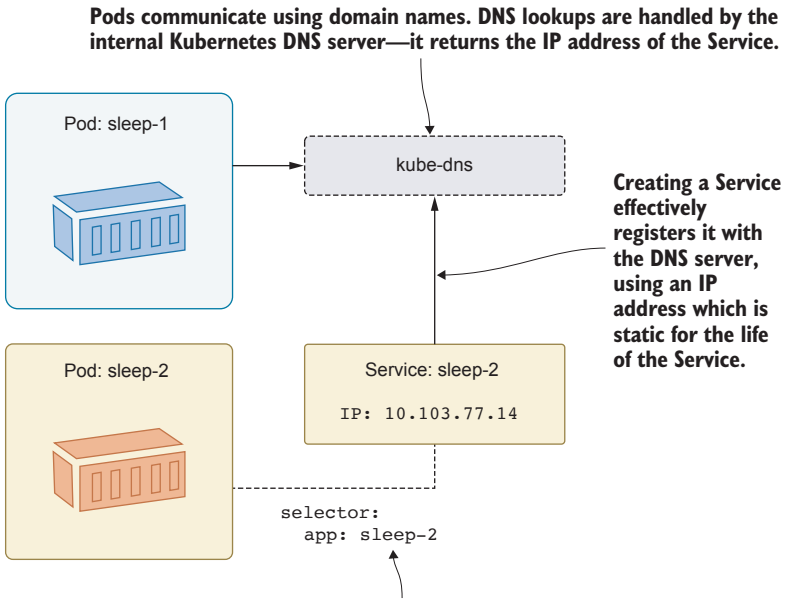

Figure 3.3 Services allow Pods to communicate using a fixed domain name.

This type of Service is an abstraction over a Pod and its network address, just like a deployment is an abstraction over a Pod and its container. The Service has its own IP address which is static, and when consumers make a network request to that address, Kubernetes routes it to the actual IP address of the Pod. The link between the Service and its Pods is set up with a label selector, just like the link between Deployments and Pods.

Listing 3.1 shows the minimal YAML specification for a Service, using the app label to identify the Pod which is the ultimate target of the network traffic.

Listing 3.1 sleep2-service.yaml, the simplest Service definition

```
apiVersion: v1    # Services use the core v1 API
kind: Service

metadata:
  name: sleep-2   # the name of a Service is used as the DNS domain name

# the specification requires a selector and a list of ports
spec:
  selector:
    app: sleep-2  # matches all Pods with an 'app' label set to 'sleep-2'
  ports:
   –port: 80    # listen on port 80 and send to port 80 on the Pod
```

This Service definition works with one of the deployments we have running from the last exercise. When you deploy it, Kubernetes creates a DNS entry called sleep-2, which routes traffic into the Pod created by the sleep-2 deployment. Other Pods can send traffic into that Pod using the Service name as the domain name.

> **TRY IT NOW** You deploy a Service using a YAML file and the usual Kubectl apply command. Deploy the Service and verify the network traffic is routed to the Pod.

```
# deploy the Service defined in listing 3.1:
kubectl apply -f sleep/sleep2-service.yaml

# show the basic details of the Service:
kubectl get svc sleep-2

# run a ping command to check connectivity—this will fail:
kubectl exec deploy/sleep-1 -- ping -c 1 sleep-2
```

My output is in figure 3.4 where you can see the name resolution worked correctly, although the ping command didn't work as expected because ping uses a network protocol that isn't supported in Kubernetes Services.

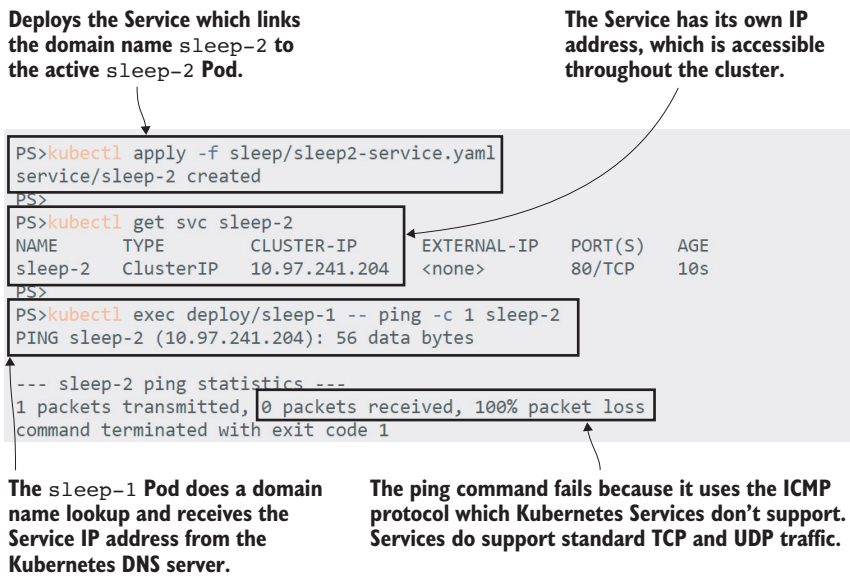

Figure 3.4 Deploying a Service creates a DNS entry, giving the Service name a fixed IP address.

So that's the basic concept behind service discovery in Kubernetes: deploy a Service resource and use the name of the Service as the domain name for components to communicate. There are different types of Service which support different networking patterns, but you work with them all in the same way. Next, we'll look more closely at Pod-to-Pod networking, with a working example of a simple distributed app.

3.2 Routing traffic between Pods

The default type of Service in Kubernetes is called ClusterIP—it creates a cluster-wide IP address that Pods on any node can access. The IP address only works within the cluster, so ClusterIP Services are only useful for communicating between Pods. That's exactly what you want for a distributed system where some components are internal and shouldn't be accessible outside of the cluster. We'll use a simple website that uses an internal API component to demonstrate that.

> **TRY IT NOW** Run two deployments, one for the web application and one for the API. There are no Services for this app yet, and it won't work correctly because the website can't find the API.

```
# run the website and API as separate deployments:
kubectl apply -f numbers/api.yaml -f numbers/web.yaml

# wait for the Pod to be ready:
kubectl wait --for=condition=Ready pod -l app=numbers-web

# forward a port to the web app:
kubectl port-forward deploy/numbers-web 8080:80

# browse to the site at http://localhost:8080 and click the Go button
#—you'll see an error message

# exit the port forward:
ctrl-c
```

You can see from my output in figure 3.5 that the app fails with a message stating the API is unavailable.

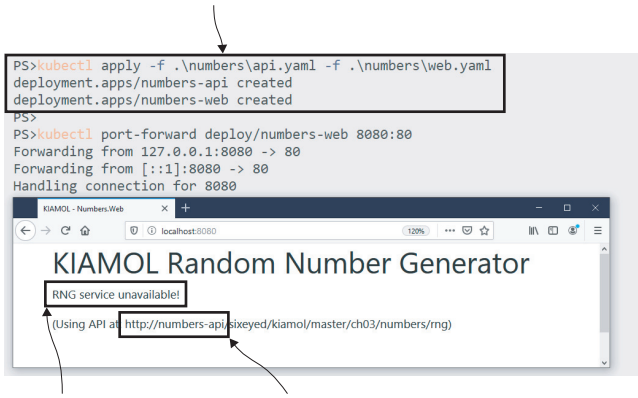

Figure 3.5 The web app runs but doesn't function correctly because the network call to the API fails.

The error page also shows the domain name where the site is expecting to find the API: http://numbers-api. That's not a fully-qualified domain name (like blog.six-eyed.com), it's an address that should be resolved by the local network, but the DNS server in Kubernetes doesn't resolve it because there is no Service with the name numbers-api. The specification in listing 3.2 shows a Service with the correct name and a label selector which matches the API Pod.

Listing 3.2 api-service.yaml—a Service for the random number API

```
apiVersion: v1
kind: Service

metadata:
  name: numbers-api     # the Service uses the domain name 'numbers-api'

spec:
  ports:
   —port: 80
  selector:
    app: numbers-api    #  traffic gets routed to Pods with this label
  type: ClusterIP       #  this Service is only available to other Pods
```

This is similar to the Service in listing 3.1, except that the names have changed and the Service type of ClusterIP is explicitly stated. That can be omitted because it's the default Service type, but I think it makes the spec clearer if you include it. Deploying the Service will route the traffic between the web Pod and the API Pod, fixing the app without any changes to the deployments or Pods.

TRY IT NOW Create the Service for the API so the domain lookup works and traffic gets sent from the web Pod to the API Pod.

```
# deploy the Service from listing 3.2:
kubectl apply -f numbers/api-service.yaml

# check the Service details:
kubectl get svc numbers-api

# forward a port to the web app:
kubectl port-forward deploy/numbers-web 8080:80

# browse to the site at http://localhost:8080 and click the Go button

# exit the port forward:
ctrl-c
```

My output in figure 3.6 shows the app working correctly, with the website displaying a random number generated by the API.

There's an important lesson here, beyond Services, Deployments, and Pods: your YAML specifications describe your whole application in Kubernetes, so that's all the components and the networking between them. Kubernetes doesn't make assumptions about your application architecture; you need to spec it all out in the YAML.

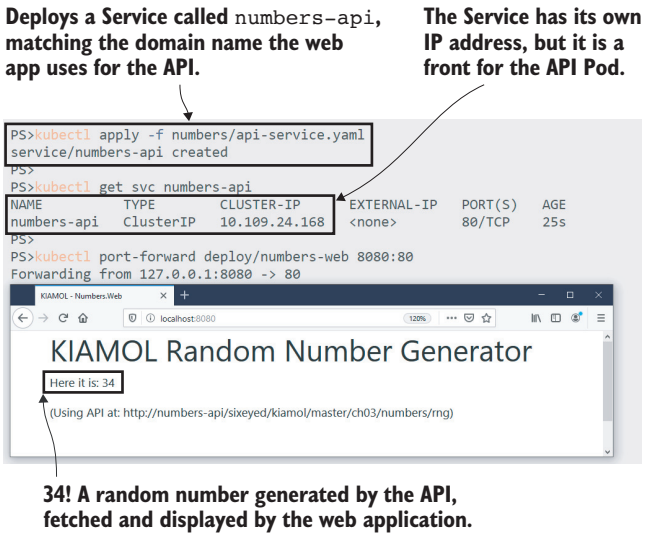

Figure 3.6 Deploying a Service fixes the broken link between the web app and the API.

This simple web app needs three Kubernetes resources defined for it to work in its current state—two Deployments and a Service, but the advantage of having all these moving parts is increased resilience.

> **TRY IT NOW** The API Pod is managed by a Deployment controller, so you can delete the Pod and a replacement will be created. The replacement is also a match for the label selector in the API Service, so traffic gets routed to the new Pod and the app keeps working.

```
# check the name and IP address of the API Pod:
kubectl get pod -l app=numbers-api -o custom-
     columns=NAME:metadata.name,POD_IP:status.podIP

# delete that Pod:
kubectl delete pod -l app=numbers-api

# check the replacement Pod:
kubectl get pod -l app=numbers-api -o custom-
     columns=NAME:metadata.name,POD_IP:status.podIP

# forward a port to the web app:
kubectl port-forward deploy/numbers-web 8080:80

# browse to the site at http://localhost:8080 and click the Go button

# exit the port forward:
ctrl-c
```

You can see in figure 3.7 that a replacement Pod gets created by the deployment controller—it's the same API Pod spec but running in a new Pod with a new IP address. The IP address of the API Service hasn't changed though, and the web Pod can reach the new API Pod at the same network address.

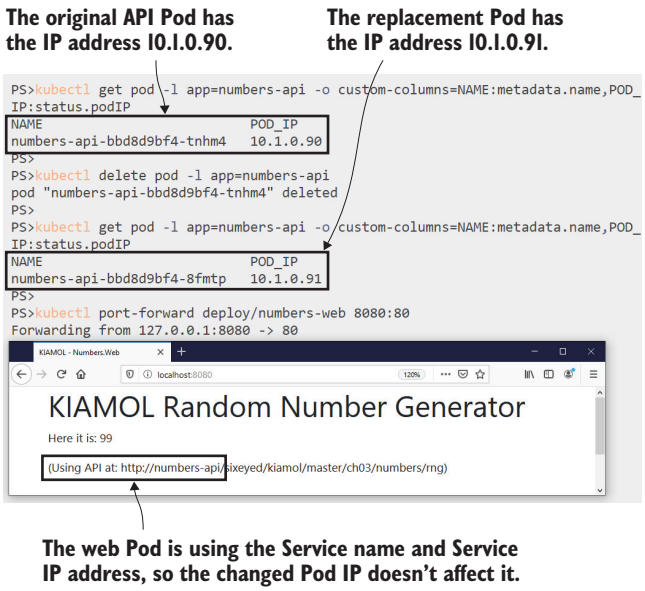

Figure 3.7 The Service isolates the web Pod from the API Pod, so it doesn't matter if the API Pod changes.

We're manually deleting Pods in these exercises to trigger the controller to create a replacement, but in the normal lifecycle of a Kubernetes application Pod replacement happens all the time. Any time you update a component of your app—to add features, fix bugs or release an update to a dependency—you'll be replacing Pods. Any time a node goes down, its Pods are replaced on other nodes. The Service abstraction keeps apps communicating through these replacements.

This demo app isn't complete yet because it doesn't have anything configured to receive traffic from outside the cluster and send it in to the web Pod. We've used port-forwarding so far, but that's really a trick for debugging. The real solution is to deploy a Service for the web Pod too.

3.3 Routing external traffic into Pods

You have lots of options to configure Kubernetes to listen for traffic coming into the cluster and forward it to a Pod—we'll start with a simple and flexible approach that is fine for everything from local development to production. It's a type of Service called LoadBalancer, which solves the problem of getting traffic to a Pod which might be running on a different node from the one which received the traffic. Figure 3.8 shows how that looks.

It looks like a tricky problem, especially because you might have many Pods that match the label selector for the Service, so the cluster needs to choose a node to send the traffic onto, and then choose a Pod on that node. All that trickiness is taken care of by Kubernetes—that's world-class orchestration for you—so all you need to do is deploy a LoadBalancer Service. Listing 3.3 shows the Service specification for the web application.

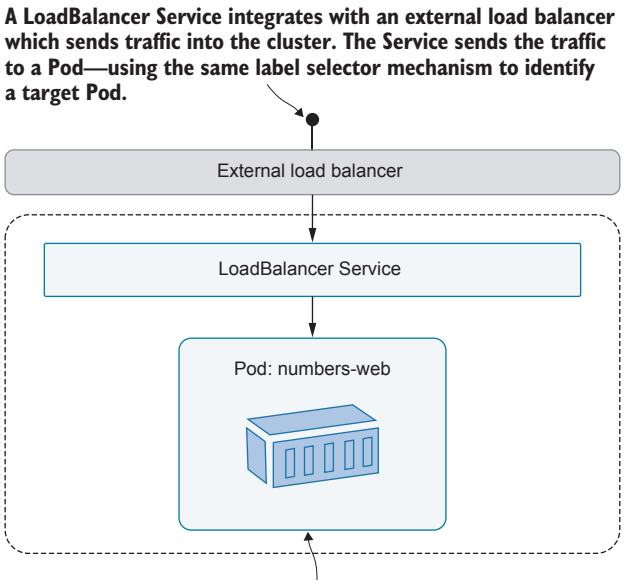

Figure 3.8 LoadBalancer Services route external traffic from any node into a matching Pod.

Listing 3.3 web-service.yaml—a LoadBalancer Service for external traffic

```
apiVersion: v1
kind: Service
metadata:
  name: numbers-web
spec:
  ports:
   —port: 8080          # the port the Service listens on
      targetPort: 80      # the port the traffic is sent to on the Pod
  selector:
    app: numbers-web
  type: LoadBalancer      # this Service is available for external traffic
```

This Service listens on port 8080 and sends traffic to the web Pod on port 80. When you deploy it, you'll be able to use the web app without setting up a port-forward in Kubectl, but the exact details of how you reach the app will depend on how you're running Kubernetes.

TRY IT NOW Deploy the Service, and then use Kubectl to find the address of the Service.

```
# deploy the LoadBalancer Service for the website—if your firewall checks #
     that you want to allow traffic, then its OK to say Yes:
kubectl apply -f numbers/web-service.yaml

# check the details of the Service:
```

```
kubectl get svc numbers-web

# use formatting to get the app URL from the EXTERNAL-IP field
kubectl get svc numbers-web -o
     jsonpath='http://{.status.loadBalancer.ingress[0].*}:8080'
```

Figure 3.9 shows the output I get from running the exercise on my Docker Desktop Kubernetes cluster, where I can browse to the website at the address http://localhost:8080.

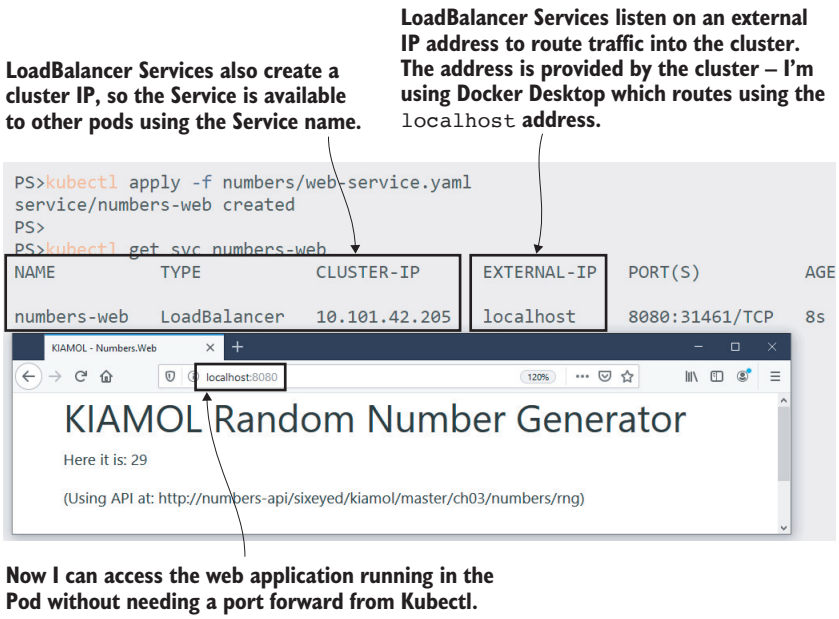

Figure 3.9 Kubernetes requests an IP address for LoadBalancer Services from the platform it's running on.

The output is different using K3s or a managed Kubernetes cluster in the cloud, where the Service deployment creates a dedicated external IP address for the load balancer. Figure 3.10 shows the output of the same exercise (using the exact same YAML specifications) using the K3s cluster on my Linux VM. Here the website is at http://172.28.132.127:8080.

How can the results be different with the same application manifests? I said in chapter 1 that you can deploy Kubernetes in different ways and *it's all the same Kubernetes* (my emphasis), but that's not strictly true. There are lots of extension points in Kubernetes and distributions have flexibility on how they implement certain features. Load balancers are a good example of where implementations differ, suited to the goals of the distribution:

- Docker Desktop is a local development environment, it runs on a single machine and integrates with the network stack so load balancer Services are

Here I'm running the same exercise on a K3s cluster, which I created using the setup described in chapter 1.

```
vagrant@kiamol:~$ kubectl get nodes
NAME      STATUS   ROLES    AGE   VERSION
kiamol    Ready    master   55m   v1.18.8+k3s1
vagrant@kiamol:~$
vagrant@kiamol:~$ kubectl get svc numbers-web
NAME          TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
numbers-web   LoadBalancer   10.43.220.197   172.28.132.127   8080:30048/TCP   3m19s
```

The LoadBalancer Service is created with a real IP address. This is a local cluster so it's not a public IP address, but if I ran this same exercise in an AKS or EKS cluster then the Service would have a public address assigned by the cloud provider.

Figure 3.10 Different Kubernetes platforms use different addresses for LoadBalancer Services.

available at the localhost address. Every load balancer Service publishes to localhost, so you'll need to use different ports if you deploy many load balancers.

- K3s supports load balancer Services with a custom component which sets up routing tables on your machine. Every load balancer Service publishes to the IP address of your machine (or VM), so you can access services with localhost or from a remote machine on your network. Like Docker Desktop, you'll need to use different ports for each load balancer.
- Cloud Kubernetes platforms like AKS and EKS are highly-available multi-node clusters. Deploying a Kubernetes load balancer Service creates an actual load balancer in your cloud, which spans all the nodes in your cluster—the cloud load balancer sends incoming traffic to one of the nodes and then Kubernetes routes it to a Pod. You'll get a different IP address for each load balancer Service, and it will be a public address, accessible from the Internet.

This is a pattern we'll see again in other Kubernetes features where distributions have different resources available and different aims. Ultimately the YAML manifests are the same and the end results are consistent, but Kubernetes allows distributions to diverge in how they get there.

Back in the world of standard Kubernetes, there's another Service type you can use that listens for network traffic coming into the cluster and directs it to a Pod, and that's the NodePort. NodePort Services don't require an external load balancer—every node in the cluster listens on the port specified in the Service and sends traffic to the target port on the Pod. Figure 3.11 shows how it works.

NodePort Services don't have the flexibility of LoadBalancer Services because you need a different port for each Service, your nodes need to be publicly accessible, and you don't get load-balancing across a multi-node cluster. They also have different levels of support in the distributions, so they work as expected in K3s and Docker Desktop, but not so well in KinD. Listing 3.4 shows a NodePort spec for reference.

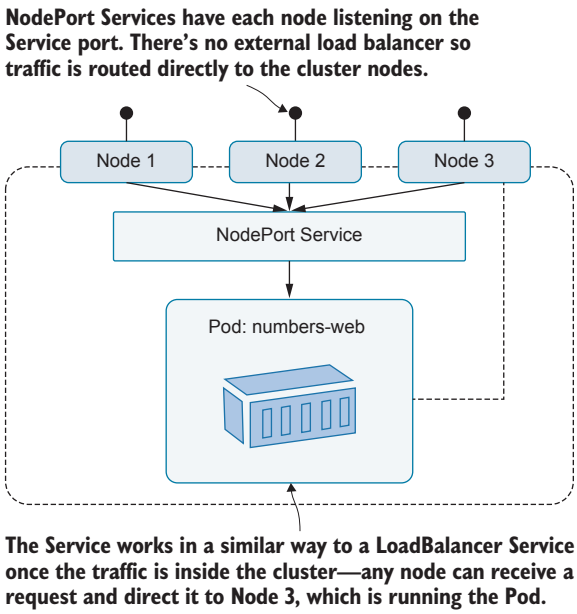

Figure 3.11 NodePort Services also route external traffic to Pods, but they don't require a load balancer.

Listing 3.4 web-service-nodePort.yaml, a NodePort Service specification

```
apiVersion: v1
kind: Service
metadata:
  name: numbers-web-node
spec:
  ports:
   –port: 8080          # the port the Service is available to other Pods
      targetPort: 80      # the port the traffic is sent to on the Pod
      nodePort: 30080     # the port the Service is available externally
  selector:
    app: numbers-web
  type: NodePort          # this Service is available on node IP addresses
```

There isn't an exercise to deploy this NodePort Service (although the YAML file is in the chapter's folder if you want to try it out). That's partly because it doesn't work in the same way on every distribution, so this section would end with lots of if branches that you'd need to try and make sense of. But there's a more important reason—you don't typically use NodePorts in production, and it's good to keep your manifests as consistent as possible across different environments. Sticking with LoadBalancer Services means you have the same specs from dev up to production, which means fewer YAML files to maintain and keep in sync.

We'll finish this chapter by digging into how Services work under the hood, but before that we'll look at one more way you can use Services, which is to communicate from Pods to components outside of the cluster.

3.4 Routing traffic outside Kubernetes

You can run pretty much any server software in Kubernetes, but that doesn't mean you should. Storage components like databases are typical candidates for running outside of Kubernetes, especially if you're deploying to the cloud and you can use a managed database service instead. Or you may be running in the datacenter and need to integrate with existing systems which won't be migrating to Kubernetes. Whatever architecture you're using, you can still use Kubernetes Services for domain name resolution to components outside the cluster.

The first option for that is to use an ExternalName Service, which is like an alias from one domain to another. ExternalName Services let you use local names in your application Pods, and the DNS server in Kubernetes resolves the local name to a fully-qualified external name when the Pod makes a lookup request. Figure 3.12 shows how that works, with a Pod using a local name that resolves to an external system address.

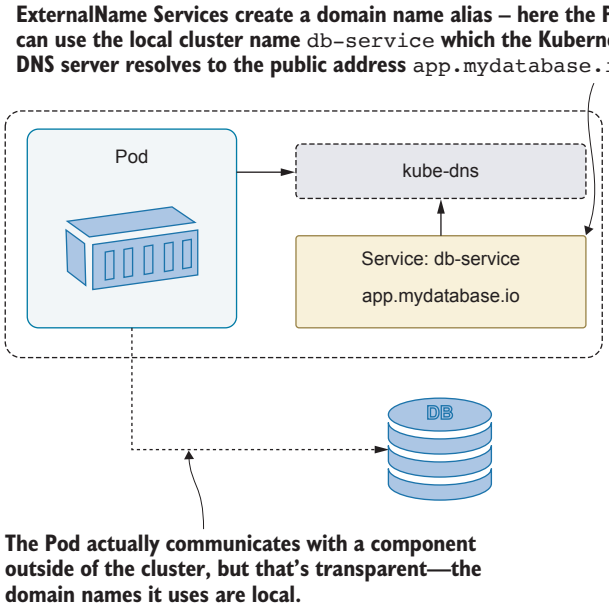

Figure 3.12 Using an ExternalName Service lets you use local cluster addresses for remote components.

The demo app for this chapter expects to use a local API to generate random numbers, but it can be switched to read a static number from a text file on GitHub just by deploying an ExternalName Service.

> **TRY IT NOW** You can't switch a Service from one type to another, so you'll need to delete the original ClusterIP Service for the API before you can deploy the ExternalName Service.

```
# delete the current API Service:
kubectl delete svc numbers-api
```

```
#deploy a new ExternalName Service:
kubectl apply -f numbers-services/api-service-externalName.yaml

# check the Service configuration:
kubectl get svc numbers-api

# refresh the website in your browser and test with the Go button
```

My output is in figure 3.13—you can see the app works in the same way, and it's using the same URL for the API. But if you refresh the page, you'll find that it always returns the same number because it's not using the random number API anymore.

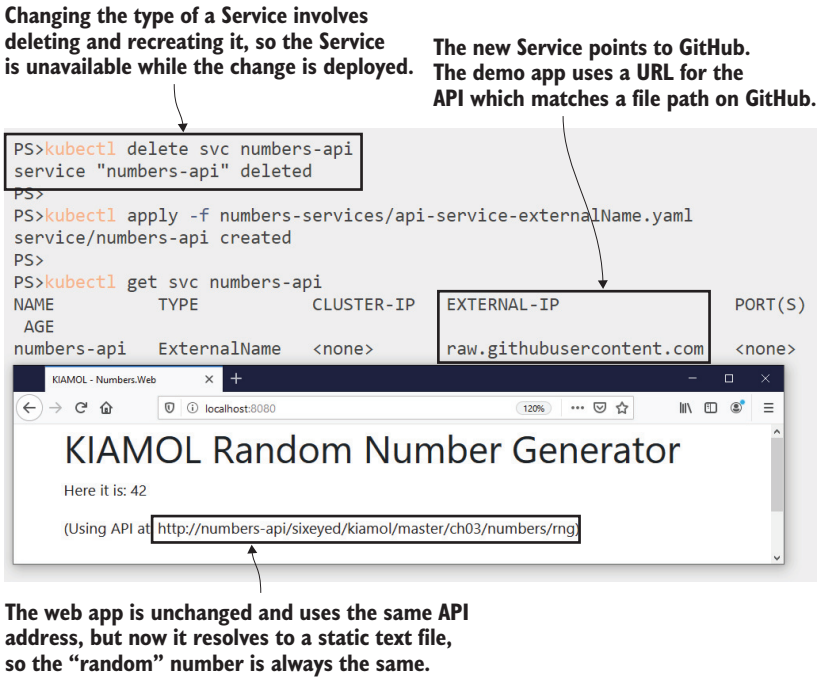

Figure 3.13 **ExternalName Services can be used as a redirect to send requests outside the cluster.**

ExternalName Services can be a useful way to deal with differences between environments that you can't workaround in your app configuration. Maybe you have an app component that uses a hardcoded string for the name of the database server—in development environments you could create a ClusterIP Service with the expected domain name that resolves to a test database running in a Pod. In production, you can use an ExternalName Service that resolves to the real domain name of the database server. Listing 3.5 shows the YAML spec for the API external name.

Listing 3.5 api-service-externalName.yaml, an ExternalName Service

```
apiVersion: v1
kind: Service
metadata:
  name: numbers-api    # the local domain name of the Service in the cluster
spec:
  type: ExternalName
  externalName: raw.githubusercontent.com    # the domain to resolve
```

Kubernetes implements ExternalName Services using a standard feature of DNS: canonical names (CNAMEs). When the web Pod makes a DNS lookup for the numbers-api domain name, the Kubernetes DNS server returns with the canonical name, which is raw.githubusercontent.com. Then the DNS resolution continues using the DNS server on the node, so it will reach out to the Internet to find the IP address of the GitHub servers.

> **TRY IT NOW** Services are part of the cluster-wide Kubernetes Pod network, so any Pod can use a Service. The sleep Pods from the first exercise in this chapter have a DNS lookup command in the container image, which you can use to check the resolution of the API Service.

```
# run the DNS lookup tool to resolve the Service name:
kubectl exec deploy/sleep-1 -- sh -c 'nslookup numbers-api | tail -n 5'
```

When you try this you may well get scrambled results that look like errors, because the Nslookup tool returns a lot of information, and not in the same order every time you run it. The data you want is in there though—I repeated the command a few times to get the fit-for-print output you can see in figure 3.14.

The Nslookup tool is installed in the sleep container image—it makes DNS queries and shows the results. The output is quite lengthy so this command shows only the final five lines.

```
PS>kubectl exec deploy/sleep-1 -- sh -c 'nslookup numbers-api | tail -n 5'
Address: 151.101.0.133

numbers-api.default.svc.cluster.local   canonical name = raw.githubusercontent.com
raw.githubusercontent.com    canonical name = github.map.fastly.net
```

The local cluster name `numbers-api` resolves to the GitHub address (which in turn resolves to a Fastly address). The sleep Pod has no logical use for the random number API, but the service is cluster-wide so any Pod can access any Service.

Figure 3.14 Apps aren't isolated by default in Kubernetes so any Pod can do a lookup for any Service.

There's one important thing to understand about ExternalName Services that you can see from this exercise—it ultimately just gives your app an address to use, but it doesn't actually change the requests your application makes. That's fine for components like databases that communicate over TCP, but it's not so simple for HTTP ser-

vices. HTTP requests include the target host name in a header field, and that won't match the actual domain from the ExternalName response, so the client call will probably fail. The random number app in this chapter has some hacky code to get around this issue, manually setting the host header, but this approach is best for non-HTTP services.

There's one other option for routing local domain names in the cluster to external systems—it doesn't fix the HTTP header issue but it does let you use a similar approach to ExternalName Services, when you want to route to an IP address rather than a domain name. These are *headless Services,* which are defined as a ClusterIP Service type, but without a label selector so they will never match any Pods. Instead the Service is deployed with an *endpoint* resource that explicitly lists the IP addresses the Service should resolve.

Listing 3.6 shows a headless Service with a single IP address in the endpoint, and it also shows a new use of YAML, with multiple resources defined, separated by three dashes.

Listing 3.6 api-service-headless.yaml, a Service with explicit addresses

```
apiVersion: v1
kind: Service
metadata:
  name: numbers-api
spec:
  type: ClusterIP      # no 'selector' field makes this a headless Service
  ports:
   -port: 80
---
kind: Endpoints        # the endpoint is a separate resource
apiVersion: v1
metadata:
  name: numbers-api
subsets:
 -addresses:         # it has a static list of IP  addresses
     -ip: 192.168.123.234
    ports:
     -port: 80       # and the ports they listen on
```

The IP address in that endpoint specification is a fake one, but Kubernetes doesn't validate that the address is reachable, so this will deploy without errors.

TRY IT NOW Replace the ExternalName Service with this headless Service—it will cause the app to fail because the API domain name now resolves to an inaccessible IP address.

```
# remove the existing Service:
kubectl delete svc numbers-api

# deploy the headless Service:
kubectl apply -f numbers-services/api-service-headless.yaml

# check the Service:
```

```
kubectl get svc numbers-api

# check the endpoint:
kubectl get endpoints numbers-api

# verify the DNS lookup:
kubectl exec deploy/sleep-1 -- sh -c 'nslookup numbers-api | grep "^[^*]"'

# browse to the app—it will fail when you try to get a number
```

My output in figure 3.15 confirms that Kubernetes will happily let you deploy a Service change that breaks your application. The domain name resolves the internal cluster IP address, but any network calls to that address fail because they get routed to the actual IP address in the endpoint, which doesn't exist.

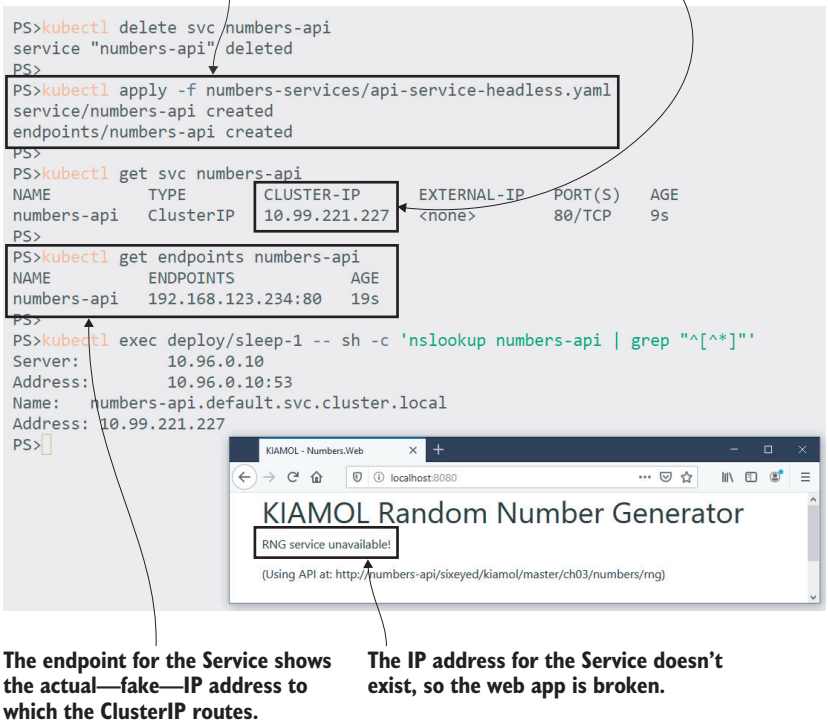

Figure 3.15 A misconfiguration in a Service can break your apps, even without deploying an app change.

The output from that exercise raises a couple of interesting questions. How come the DNS lookup returns the cluster IP address instead of the endpoint address? And why

does the domain name end with ".default.svc.cluster.local"? You don't need a background in network engineering to work with Kubernetes Services, but it will help you track down issues if you understand how Service resolution actually works, and that's how we'll finish the chapter.

3.5 Understanding Kubernetes Service resolution

Kubernetes supports all the network configurations your app is likely to need using Services, which build on established networking technologies. Application components run in Pods and communicate with other Pods using standard transfer protocols and DNS names for discovery. You don't need any special code or libraries; your apps work in the same way in Kubernetes as if you deployed them on physical servers or VMs.

We've covered all the Service types and their typical use-cases in this chapter, so now you have a good understanding of the patterns you can use. If you're feeling that there's an awful lot of detail here, be assured that the majority of times you'll be deploying ClusterIP Services, which require very little configuration. They work pretty much seamlessly, but it is useful to go one level deeper to understand the stack. Figure 3.16 shows that next level of detail.

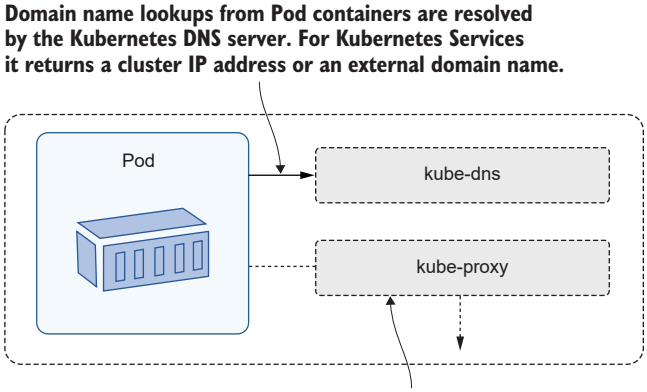

Figure 3.16 Kubernetes runs a DNS server and a proxy and uses them with standard network tools.

The key takeaway is that the ClusterIP is a virtual IP address that doesn't exist on the network. Pods access the network through the kube-proxy running on the node, and that uses packet filtering to send the virtual IP to the real endpoint. Kubernetes Services keep their IP addresses as long as they exist, and Services can exist independently of any other parts of your app. Services have a controller that keeps the endpoint list updated whenever there are changes to Pods—so clients always use the static virtual IP address and the kube-proxy always has the up-to-date endpoint list.

TRY IT NOW You can see how Kubernetes keeps the endpoint list immediately updated when Pods change by listing the endpoints for a Service between Pod changes. Endpoints use the same name as Services, and you can view endpoint details using Kubectl.

```
# show the endpoints for the sleep-2 Service:
kubectl get endpoints sleep-2

# delete the Pod:
kubectl delete pods -l app=sleep-2

# check the endpoint is updated with the IP of the replacement Pod:
kubectl get endpoints sleep-2

# delete the whole deployment:
kubectl delete deploy sleep-2

# check the endpoint still exists, with no IP addresses:
kubectl get endpoints sleep-2
```

You can see my output in figure 3.17 and it's the answer to the first question—Kubernetes DNS returns the cluster IP address and not the endpoint, because endpoint addresses change.

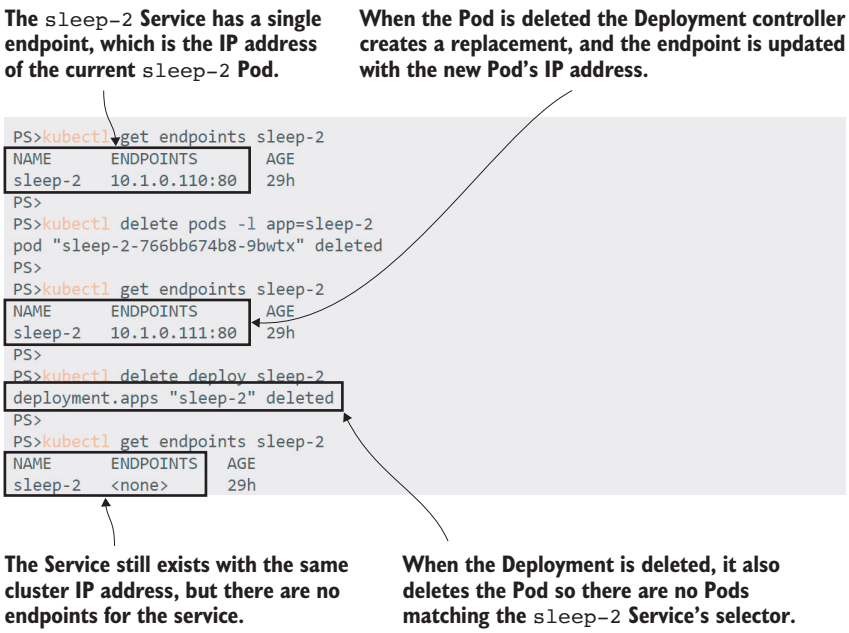

Figure 3.17 The cluster IP address for a Service doesn't change, but the endpoint list is always updating.

Using a static virtual IP means clients can cache the DNS lookup response indefinitely (which many apps do as a misguided performance saving), and that IP address will

continue to work no matter how many Pod replacements there are over time. The second question—about the domain name suffix—needs to be answered with a sideways step to look at Kubernetes *namespaces*.

Every Kubernetes resource lives inside a namespace, which is a resource you can use to group other resources. They're a way to logically partition a Kubernetes cluster—you could have one namespace per product, one per team or a single shared namespace. We won't use namespaces for a while yet, but I'm introducing them here because they have a part to play in DNS resolution. Figure 3.18 shows where the namespace comes into the Service name.

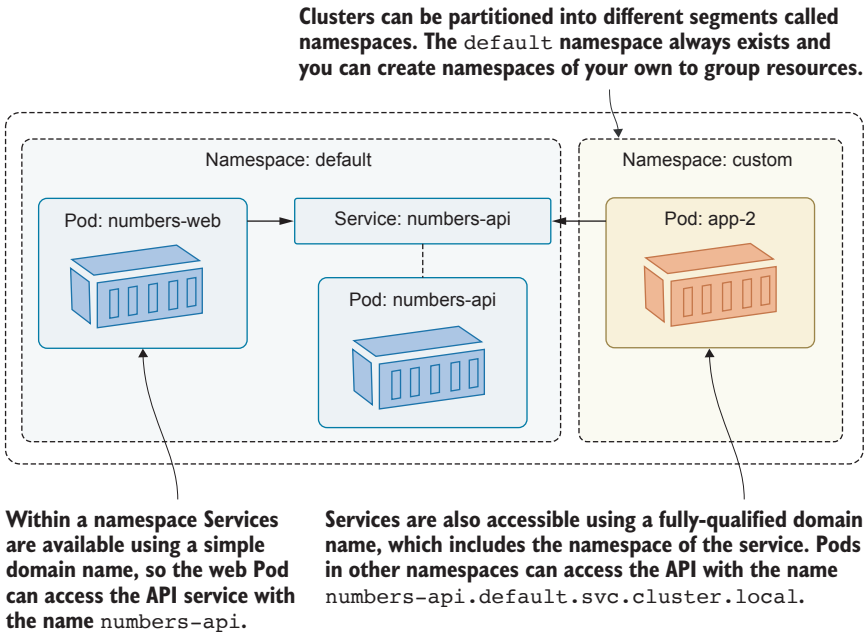

Figure 3.18 Namespaces logically partition a cluster, but Services are accessible across namespaces.

You already have several namespaces in your cluster—all the resources we've deployed so far have been created in the `default` namespace (which is the default. . . . That's why we haven't needed to specify a namespace in our YAML files). Internal Kubernetes components like the DNS server and the Kubernetes API also run in Pods, in the `kube-system` namespace.

TRY IT NOW Kubectl is namespace-aware; you can use the namespace flag to work with resources outside of the default namespace.

```
# check the Services in the default namespace:
kubectl get svc --namespace default

# and check Services in the system namespace:
kubectl get svc -n kube-system
```

```
# try a DNS lookup to a fully-qualified Service name:
kubectl exec deploy/sleep-1 -- sh -c 'nslookup numbers-
     api.default.svc.cluster.local | grep "^[^*]"'

# and for a Service in the system namespace:
kubectl exec deploy/sleep-1 -- sh -c 'nslookup kube-dns.kube-
     system.svc.cluster.local | grep "^[^*]"'
```

My output is in figure 3.19 and it answers the second question—the local domain name for a Service is just the Service name, but that's an alias for the fully qualified domain name that includes the Kubernetes namespace.

Kubectl commands support a `namespace` parameter, which restricts queries to the specified namespace. These are the Services in the `default` namespace.

And in the `kube-system` namespace.

```
PS>kubectl get svc --namespace default
NAME          TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
kubernetes    ClusterIP      10.96.0.1       <none>        443/TCP          32h
numbers-api   ClusterIP      10.99.221.227   <none>        80/TCP           6h57m
numbers-web   LoadBalancer   10.101.42.205   localhost     8080:31461/TCP   12h
sleep-2       ClusterIP      10.97.241.204   <none>        80/TCP           31h
PS>
PS>kubectl get svc -n kube-system
NAME       TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE
kube-dns   ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP   32h
PS>
PS>kubectl exec deploy/sleep-1 -- sh -c 'nslookup numbers-api.default.svc.cluster.loca
l | grep "^[^*]"'
Server:         10.96.0.10
Address:        10.96.0.10:53
Name:   numbers-api.default.svc.cluster.local
Address: 10.99.221.227
PS>
PS>kubectl exec deploy/sleep-1 -- sh -c 'nslookup kube-dns.kube-system.svc.cluster.loc
al | grep "^[^*]"'
Server:         10.96.0.10
Address:        10.96.0.10:53
Name:   kube-dns.kube-system.svc.cluster.local
Address: 10.96.0.10
```

A query for the `kube-dns` Service in the `kube-system` namespace. Hey look at that, it's the same IP address as the DNS server—the `kube-dns` service is the DNS server address for the cluster.

A DNS query in the `sleep-1` Pod resolves a fully-qualified Service name. The lookup tool also prints the address of the DNS server –`10.96.0.10`.

Figure 3.19 You can use the same Kubectl commands to view resources in different namespaces.

It's important to know about namespaces early in your Kubernetes journey, if only because it helps you see that core Kubernetes features run as Kubernetes applications too, but you don't see them in Kubectl unless you explicitly set the namespace. Namespaces are a powerful way to subdivide your cluster to increase utilization without compromising security, and we'll return to them later in this book.

For now we're done with namespaces and with Services. You've learned in this chapter that every Pod has its own IP address, and Pod communication ultimately uses that address with standard TCP and UDP protocols. You never use the Pod IP address directly though; you always create a Service resource which Kubernetes uses to provide service discovery with DNS. Services support multiple networking patterns, with different Service types configuring network traffic between Pods, into Pods from the outside world and from Pods to the world outside. You also learned that Services have their own lifecycle, independent of Pods and Deployments, so the last thing to do is clear up before we move on.

TRY IT NOW Deleting a Deployment also deletes all its Pods, but there's no cascading delete for Services, they're independent objects which need to be removed separately.

```
# delete deployments:
kubectl delete deploy --all

# and Services:
kubectl delete svc --all

# check what's running:
kubectl get all
```

Now your cluster is clear again, although as you can see in figure 3.20, you need to be careful with some of these Kubectl commands.

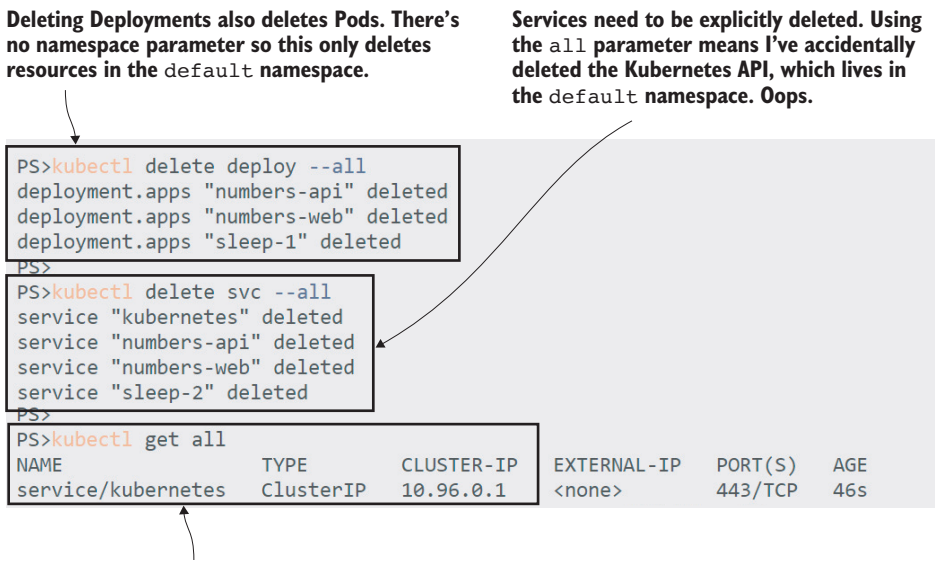

Figure 3.20 **You need to explicitly delete any Services you create—watch out with the "all" parameter.**

3.6 Lab

This lab is going to give you some practice creating Services, but it's also going to get you thinking about labels and selectors, which are powerful features of Kubernetes. The goal is to deploy Services for an updated version of the random number app which has had a UI makeover. Here are your hints:

- The lab folder for this chapter has a `deployments.yaml` file. Use that to deploy the app with Kubectl.
- Check the Pods. There are two versions of the web application running.
- Write a Service that will make the API available to other Pods at the domain name numbers-api.
- Write a Service which will make version two of the website available externally, on port 8088.
- You'll need to look closely at the Pod labels to get the correct result.

This is an extension of the exercises in the chapter, and if you want to check my solution it's up on GitHub in the repo for the book:

https://github.com/sixeyed/kiamol/blob/master/ch03/lab/README.md

4 Configuring applications with ConfigMaps and Secrets

One of the great advantages of running apps in containers is that you eliminate the gaps between environments. The deployment process is to promote the same container image through all your test environments up to production—so each deployment uses the exact same set of binaries as the previous environment. You'll never again see a production deployment fail because the servers are missing a dependency that someone manually installed on the test servers and forgot to document. Of course, there are differences between environments, and you provide for that by injecting configuration settings into containers.

Kubernetes supports configuration injection with two resource types: ConfigMaps and Secrets. Both types can store data in any reasonable format, and that data lives in the cluster independent of any other resources. Pods can be defined with access to the data in ConfigMaps and Secrets, with different options for how that data gets surfaced. In this chapter, you'll learn all the ways to manage configuration in Kubernetes, which are flexible enough to meet the requirements for pretty much any application.

4.1 How Kubernetes supplies configuration to apps

You create ConfigMap and Secret objects like other resources in Kubernetes—by creating commands with Kubectl or from a YAML specification. Unlike other resources, they don't do anything; they're just storage units intended for small amounts of data. Those storage units can be loaded into a Pod, becoming part of the container environment, so the application in the container can read the data.

Before we even get to those objects, we'll look at the simplest way to provide configuration settings, using environment variables.

> **TRY IT NOW** Environment variables are a core operating system feature in Linux and Windows, and they can be set at the machine level so any app can read them. They're commonly used, and all containers have some set by the operating system inside the container and by Kubernetes. Make sure your Kubernetes lab is up and running.

```
# switch to the exercise directory for this chapter:
cd ch04

# deploy a Pod using the sleep image with no extra configuration:
kubectl apply -f sleep/sleep.yaml

# wait for the Pod to be ready:
kubectl wait --for=condition=Ready pod -l app=sleep

# check some of the environment variables in the Pod container:
kubectl exec deploy/sleep -- printenv HOSTNAME KIAMOL_CHAPTER
```

You can see from my output in figure 4.1 that the hostname variable exists in the container and is populated by Kubernetes, but the custom "Kiamol" variable doesn't exist.

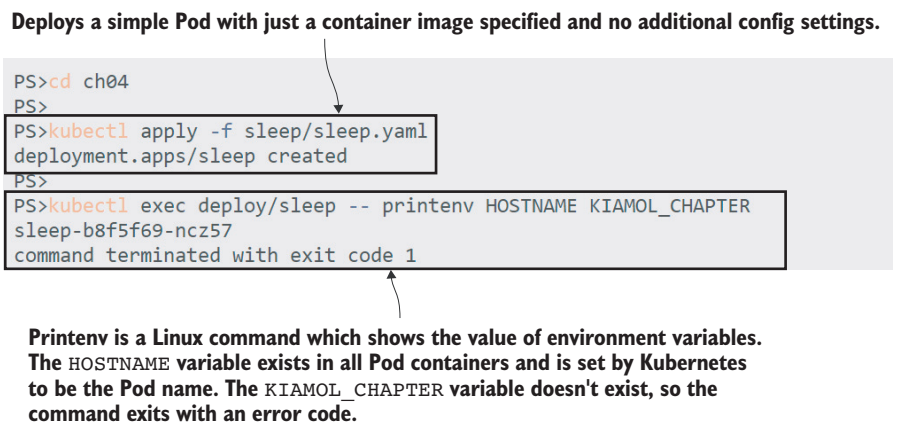

Figure 4.1 All Pod containers have some environment variables set by Kubernetes and the container OS.

In this exercise, the application is just the Linux Printenv tool, but the principle is the same for any application. Many technology stacks use environment variables as a basic configuration system—and the simplest way to provide those settings in Kubernetes is by adding environment variables in the Pod specification. Code listing 4.1 shows an updated Pod spec for the sleep Deployment which adds the Kiamol environment variable.

Listing 4.1 sleep-with-env.yaml, a Pod spec with environment variables

```
spec:
  containers:
   –name: sleep
      image: kiamol/ch03-sleep
      env:                              # sets environment variables
     –name: KIAMOL_CHAPTER              # name of the variable to create
        value: "04"                     # value to set for the variable
```

Environment variables are static for the life of the Pod—you can't update any values while the Pod is running, so if you need to make configuration changes, that means an update with a replacement Pod. You should get used to the idea that deployments aren't just for new feature releases, you'll also use them for config changes and software patches, and your apps need to be designed to handle frequent Pod replacements.

> **TRY IT NOW** Update the sleep Deployment with the new Pod spec from code listing 4.1, adding an environment variable which is visible inside the Pod container.

```
# update to the Deployment:
kubectl apply -f sleep/sleep-with-env.yaml

# check the same environment variables in the new Pod:
kubectl exec deploy/sleep -- printenv HOSTNAME KIAMOL_CHAPTER
```

My output in figure 4.2 shows the result—a new container with the Kiamol environment variable set, running in a new Pod.

Updates the existing Deployment—adding an environment variable changes the Pod spec, so this will replace the original Pod.

```
PS>kubectl apply -f sleep/sleep-with-env.yaml
deployment.apps/sleep configured
PS>
PS>kubectl exec deploy/sleep -- printenv HOSTNAME KIAMOL_CHAPTER
sleep-65f8fb555d-c62nf
04
```

The new Pod name is set in the `HOSTNAME` variable, and now the `KIAMOL_CHAPTER` variable is set.

Figure 4.2 Adding environment variables to a Pod spec makes the values available in the Pod container.

The important thing about the last exercise is that the new app is using the same Docker image: it's the same application with all the same binaries, and only the configuration settings have changed between deployments. Setting environment values inline in the Pod specification is fine for simple settings, but real applications usually have more complex configuration requirements, which is when you use ConfigMaps.

A ConfigMap is just a resource which stores some data that can be loaded into a Pod. The data can be a set of key-value pairs, a blob of text, or even a binary file. You can use key-value pairs to load Pods with environment variables, text to load any type of config file—JSON, XML, YAML, TOML, INI—and you could use binary files to load license keys. One Pod can use many ConfigMaps and each ConfigMap can be used by many Pods. Figure 4.3 shows some of those options.

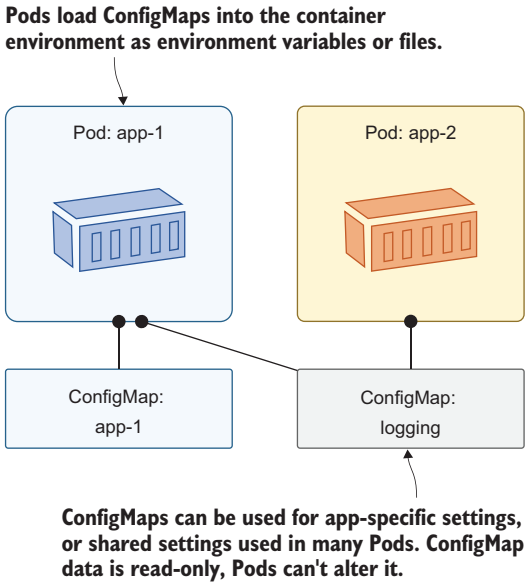

Figure 4.3 ConfigMaps are separate resources, which can be attached to zero or more Pods.

We'll stick with the simple sleep Deployment to show the basics of creating and using ConfigMaps. Code listing 4.2 shows the environment section of an updated Pod specification which uses one environment variable defined in the YAML, and a second loaded from a ConfigMap.

Listing 4.2 sleep-with-configMap-env.yaml, loading a ConfigMap into a Pod

```
  env:                                # the env section of the container spec
—name: KIAMOL_CHAPTER
    value: "04"                       # this is the variable value
—name: KIAMOL_SECTION
    valueFrom:
      configMapKeyRef:                # this value comes from a ConfigMap
        name: sleep-config-literal    # name of the ConfigMap
        key: kiamol.section           # name of the data item to load
```

If you reference a ConfigMap in a Pod specification, the ConfigMap needs to exist before you deploy the Pod. This spec expects to find a ConfigMap called `sleep-config-literal` with key-value pairs in the data, and the easiest way to create that is by passing the key and value to a Kubectl command.

TRY IT NOW Create a ConfigMap by specifying the data in the command, then check the data and deploy the updated sleep app to use the ConfigMap.

```
# create a ConfigMap with data from the command line:
kubectl create configmap sleep-config-literal --from-
     literal=kiamol.section='4.1'

# check the ConfigMap details:
kubectl get cm sleep-config-literal

# show the friendly description of the ConfigMap:
kubectl describe cm sleep-config-literal

# deploy the updated Pod spec from code listing 4.2:
kubectl apply -f sleep/sleep-with-configMap-env.yaml

# check the Kiamol environment variables:
kubectl exec deploy/sleep -- sh -c 'printenv | grep "^KIAMOL"'
```

We won't use Kubectl to describe commands much in this book because the output is usually pretty verbose and would use up most of a chapter. But it's definitely something to experiment with, because describing services and Pods gives you a whole lot of useful information in a readable format. You can see my output in figure 4.4, which includes the key-value data shown from describing the ConfigMap.

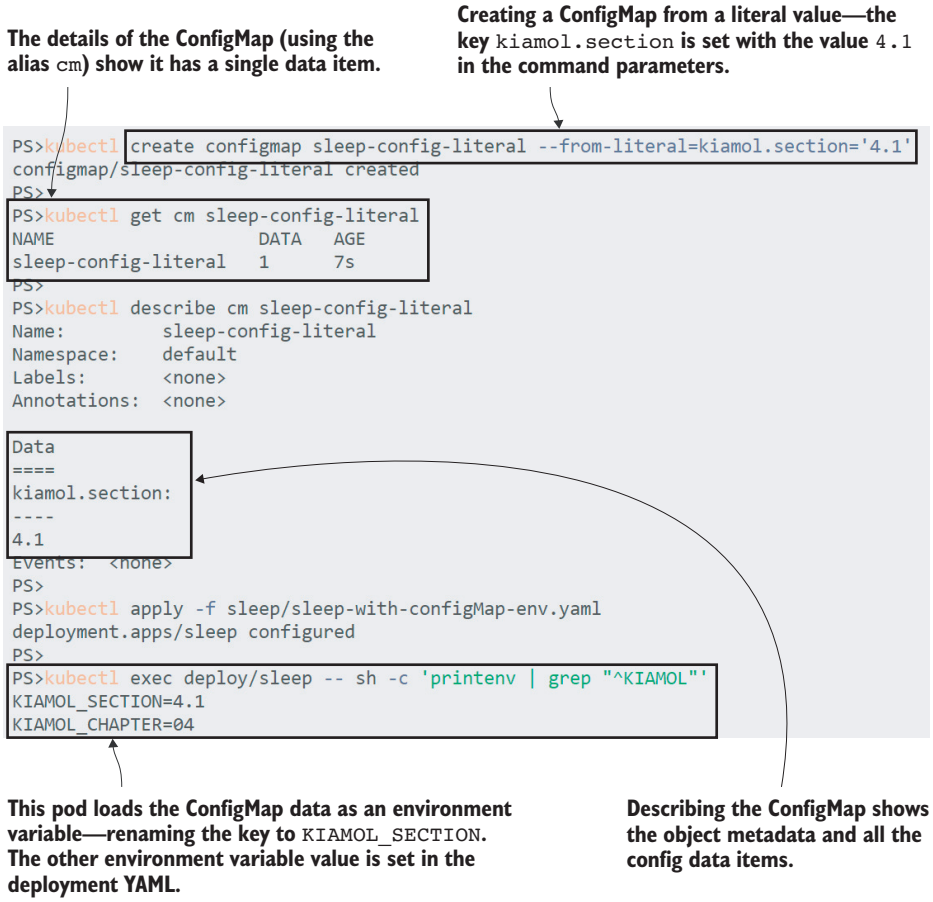

Figure 4.4 Pods can load individual data items from ConfigMaps and rename the key.

Creating ConfigMaps from literal values is fine for individual settings, but it gets cumbersome fast if you have lots of configuration data. As well as specifying literal values on the command line, Kubernetes lets you load ConfigMaps from files.

4.2 Storing and using configuration files in ConfigMaps

Options for creating and using ConfigMaps have evolved over many Kubernetes releases, so they now support practically every configuration variant you can think of. These sleep Pod exercises are a good way to show the variations, but they're getting a bit boring, so we'll just have one more before we move onto something more interesting. Code listing 4.3 shows an environment file—a text file with key-value pairs which can be loaded to create one ConfigMap with multiple data items.

Listing 4.3 ch04.env, a file of environment variables

```
# environment files use a new line for each variable
KIAMOL_CHAPTER=ch04
KIAMOL_SECTION=ch04-4.1
KIAMOL_EXERCISE=try it now
```

Environment files are a useful way to group together multiple settings, and Kubernetes has explicit support for loading them into ConfigMaps and surfacing all the settings as environment variables in a Pod container.

TRY IT NOW Create a new ConfigMap populated from the environment file in code listing 4.3, then deploy an update to the sleep app to use the new settings.

```
# load an environment variable into a new ConfigMap:
kubectl create configmap sleep-config-env-file --from-env-file=sleep/ch04.env

# check the details of the ConfigMap:
kubectl get cm sleep-config-env-file

# update the Pod to use the new ConfigMap:
kubectl apply -f sleep/sleep-with-configMap-env-file.yaml

# and check the values in the container:
kubectl exec deploy/sleep -- sh -c `printenv | grep "^KIAMOL"`
```

My output in figure 4.5 shows the Printenv command reading all the environment variables and showing the ones with Kiamol names, but it might not be the result you expect.

That exercise showed you how to create a ConfigMap from a file and it also showed you that Kubernetes has rules of precedence for applying environment variables. The Pod spec you just deployed is in code listing 4.4, and it loads all environment variables from the ConfigMap, but it also specifies explicit environment values with some of the same keys.

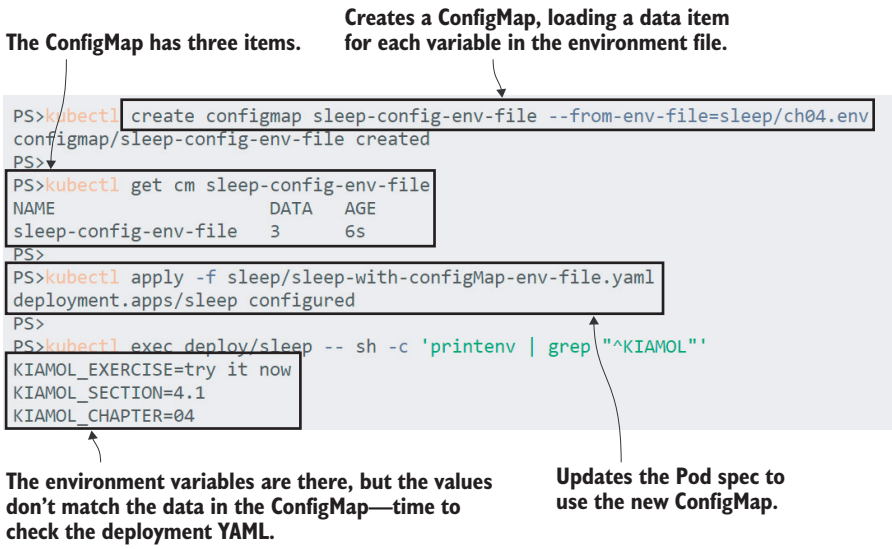

Figure 4.5 A ConfigMap can have multiple data items, and the Pod can load them all.

Listing 4.4 sleep-with-configMap-env-file.yaml, multiple ConfigMaps in a Pod

```
  env:                                  # the existing env section
—name: KIAMOL_CHAPTER
    value: "04"
—name: KIAMOL_SECTION
    valueFrom:
      configMapKeyRef:
        name: sleep-config-literal
        key: kiamol.section
  envFrom:                              # envFrom loads multiple variables
—configMapRef:                          # from a ConfigMap
      name: sleep-config-env-file
```

So the environment variables defined with `env` in the Pod spec override the values defined with `envFrom` if there are duplicate keys. It's useful to remember that you can always override any environment variables set in the container image or in ConfigMaps by explicitly setting them in the Pod spec, it can be a quick way to change a config setting when you're tracking down problems.

Environment variables are well supported but they only get you so far, and most application platforms prefer a more structured approach. We'll use a web application in the rest of the exercises for this chapter that supports a hierarchy of configuration sources. There are default settings packaged in a JSON file in the Docker image, and the app looks in other locations at runtime for JSON files with settings which override the defaults—and all the JSON settings can be overridden with environment variables. Code listing 4.5 shows the Pod spec for the first deployment we'll use.

Listing 4.5 todo-web.yaml, a web app with config settings

```
spec:
  containers:
 –name: web
    image: kiamol/ch04-todo-list
    env:
   –name: Logging__LogLevel__Default
      value: Warning
```

This run of the app will use all the default settings from the JSON configuration file in the image, except for the default logging level that is set as an environment variable.

TRY IT NOW Run the app without any additional configuration and check its behavior.

```
# deploy the app with a service to access it:
kubectl apply -f todo-list/todo-web.yaml

# wait for the Pod to be ready:
kubectl wait --for=condition=Ready pod -l app=todo-web

# get the address of the app:
kubectl get svc todo-web -o
     jsonpath='http://{.status.loadBalancer.ingress[0].*}:8080'

# browse to the app and have a play around
# then try browsing to /config

# check the application logs:
kubectl logs -l app=todo-web
```

The demo app is a simple to-do list (which will be distressingly familiar to readers of *Learn Docker in a Month of Lunches*). In its current setup, it lets you add and view items, but there should also be a `/config` page you can use in non-production environments to view all the config settings. As you can see in figure 4.6 that page is empty and the app logs a warning that someone tried to access it.

The config hierarchy in use here is a very common approach if you're not familiar with it, Appendix 3 is the chapter *Reading App Configuration from the Docker Platform* from *Learn Docker in a Month of Lunches* and that explains it in detail. This example is a .NET Core app that uses JSON but you see similar configuration systems using a variety of file formats in Java Spring apps, Node.js, Go, Python and more. In Kubernetes, you use the same app configuration approach with them all:

- Default app settings are baked into the container image—this could be just the settings which apply in every environment or it could be a full set of config, so without any extra setup the app runs in development mode (that's helpful for developers who can quickly start the app with a simple Docker `run` command).
- The actual settings for each environment are stored in a ConfigMap and surfaced into the container filesystem. Kubernetes presents the config data as a file

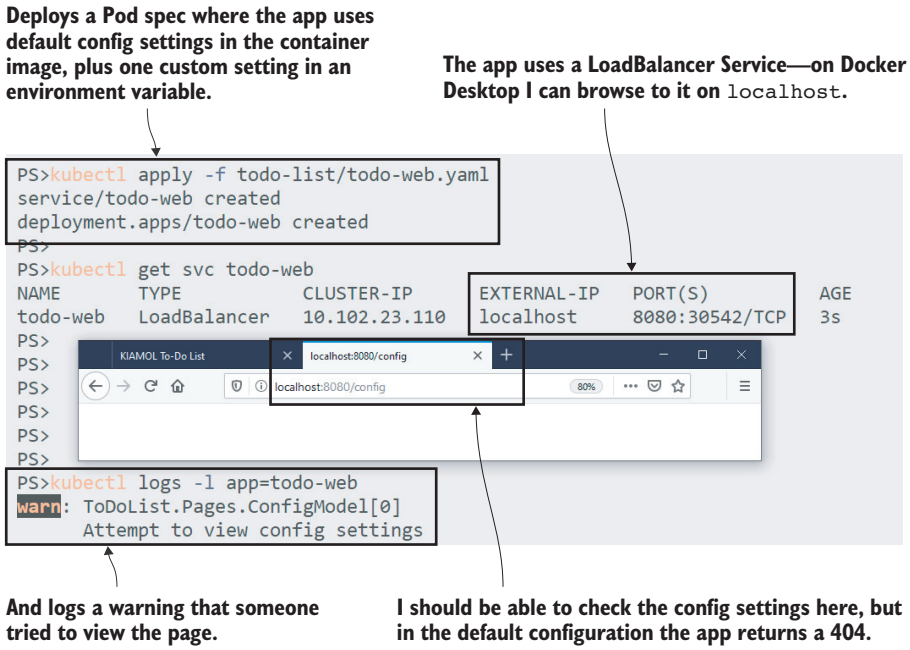

Figure 4.6 The app mostly works, but we need to set some more config values.

in a known location, which the app checks and merges with the content from the default file.

- Any settings that need to be tweaked can be applied as environment variables in the Pod specification for the Deployment.

Code listing 4.6 shows the YAML specification for the development configuration of the to-do app—it contains the contents of a JSON file which the app will merge with the default JSON config file in the container image, with a setting to make the config page visible.

Listing 4.6 todo-web-config-dev.yaml, a ConfigMap specification

```
apiVersion: v1
kind: ConfigMap                    # ConfigMap is the resource type
metadata:
  name: todo-web-config-dev        # the name of the ConfigMap
data:
  config.json: |-                  # the data key is the file name
    {                              # the file contents can be any format
      "ConfigController": {
        "Enabled" : true
      }
    }
```

You can embed any kind of text config file into a YAML spec, as long as you're careful with the whitespace. I prefer this to loading ConfigMaps directly from config files because it means you can consistently use Kubectl `apply` to deploy every part of your app. If I wanted to load the JSON file directly, I'd need to use Kubectl `create` for configuration resources and `apply` for everything else.

The ConfigMap definition in code listing 4.6 just contains a single setting, but it's stored in the native configuration format for the app. When we deploy an updated Pod spec, the setting will be applied and the config page will be visible.

TRY IT NOW The new Pod spec references the ConfigMap, so that needs to be created first by applying the YAML, then update the to-do app Deployment.

```
# create the JSON ConfigMap:
kubectl apply -f todo-list/configMaps/todo-web-config-dev.yaml

# update the app to use the ConfigMap:
kubectl apply -f todo-list/todo-web-dev.yaml

# refresh your web browser at the /config page for your service
```

You can see my output in figure 4.7. The config page loads correctly now, so the new Deployment configuration is merging in the settings from the ConfigMap to override the default setting in the image that blocked access to that page.

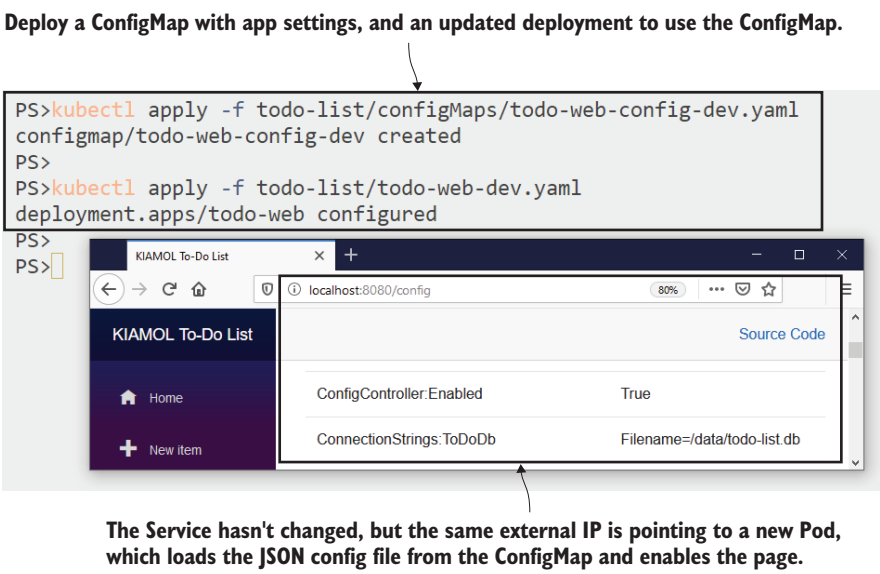

Figure 4.7 Loading ConfigMap data into the container filesystem, where the app loads config files.

This approach needs two things: your application needs to be able to merge in the ConfigMap data, and your Pod specification needs to load the data from the Config-

Map into the expected file path in the container filesystem. We'll see how that works in the next section.

4.3 Surfacing configuration data from ConfigMaps

The alternative to loading config items into environment variables is to present them as files inside directories in the container. The container filesystem is a virtual construct, built from the container image and other sources. Kubernetes can use ConfigMaps as a filesystem source, and they get mounted as a directory, with a file for each data item. Figure 4.8 shows the setup you've just deployed, where the data item in the ConfigMap is surfaced as a file.

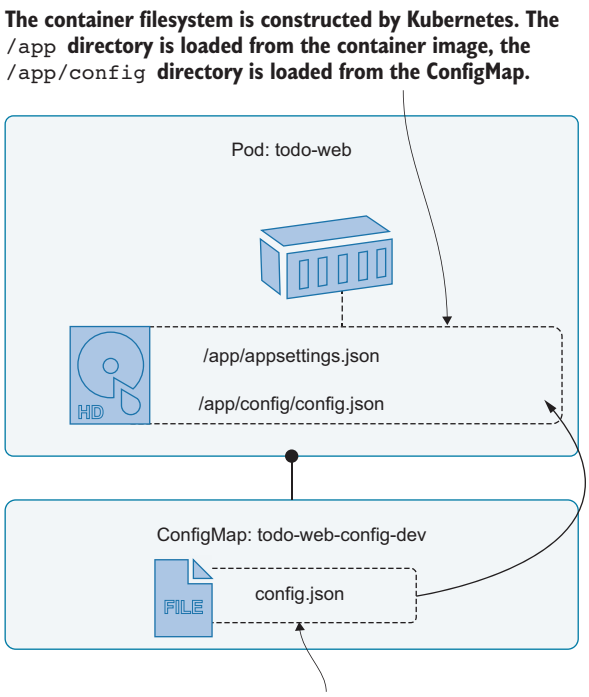

Figure 4.8 ConfigMaps can be loaded as directories in the container filesystem.

Kubernetes manages this strange magic with two features of the Pod spec: *volumes*, which make the contents of the ConfigMap available to the Pod, and *volume mounts* that load the contents of the ConfigMap volume into a specified path in the Pod container. Code listing 4.7 shows the volumes and mounts you deployed in the last exercise.

Listing 4.7 todo-web-dev.yaml, loading a ConfigMap as a volume mount

```
spec:
  containers:
   —name: web
      image: kiamol/ch04-todo-list
      volumeMounts:                  # mounts a volume into the container
```

```
        -name: config                    # name of the volume
           mountPath: "/app/config"      # directory path to mount the volume
           readOnly: true                # optionally flag the volume read-only

  volumes:                               # volumes are defined at the Pod level
   -name: config                         # name matches the volume mount
      configMap:                         # volume source is a ConfigMap
        name: todo-web-config-dev        # ConfigMap name
```

The important thing to realize here is that the ConfigMap is treated like a directory, with multiple data items which each become files in the container filesystem. In this example, the application loads its default settings from the file at `/app/appsettings.json`, and then it looks for a file at `/app/config/config.json` that can contain settings to override the defaults. The `/app/config` directory doesn't exist in the container image; it is created and populated by Kubernetes.

> **TRY IT NOW** The container filesystem appears as a single storage unit to the application, but it has been built from the image and the ConfigMap. Those sources have different behaviors.

```
# show the default config file:
kubectl exec deploy/todo-web -- sh -c 'ls -l /app/app*.json'

# show the config file in the volume mount:
kubectl exec deploy/todo-web -- sh -c 'ls -l /app/config/*.json'

# check it really is read-only:
kubectl exec deploy/todo-web -- sh -c 'echo ch04 >> /app/config/config.json'
```

My output in figure 4.9 shows that the JSON config files exist in the expected locations for the app, but the ConfigMap files are managed by Kubernetes and delivered as read-only files.

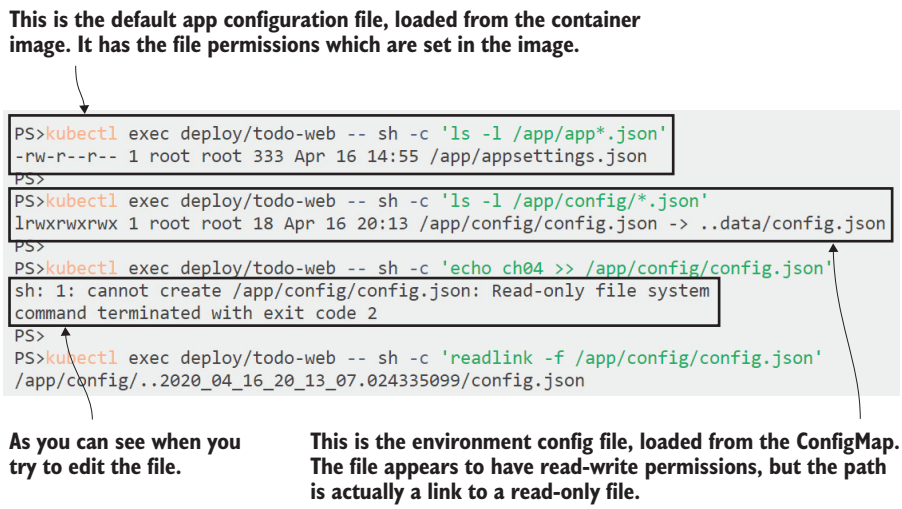

Figure 4.9 The container filesystem is built by Kubernetes from the image and the ConfigMap.

Loading ConfigMaps as directories is very flexible, and you can use it to support different approaches to app configuration. If your config is split across multiple files, you can store them all in a single ConfigMap and load them all into the container. Code listing 4.8 shows the data items for an update to the to-do ConfigMap with two JSON files—separating out the settings for application behavior and logging.

Listing 4.8 todo-web-config-dev-with-logging.yaml, a ConfigMap with two files

```
data:
  config.json: |-                       # the original app config file
    {
      "ConfigController": {
        "Enabled" : true
      }
    }
  logging.json: |-                      # a second JSON file which will be
    {                                   # surfaced in the volume mount
      "Logging": {
        "LogLevel": {
          "ToDoList.Pages" : "Debug"
        }
      }
    }
```

What happens when you deploy an update to a ConfigMap that a live Pod is using? Kubernetes delivers the updated files to the container, but what happens next very much depends on the application. Some apps load config files into memory when they start and then ignore any changes in the config directory, so changing the ConfigMap won't actually change the app configuration until the Pods are replaced. This application is more thoughtful—it watches the config directory and reloads any file changes, so deploying an update to the ConfigMap will update the application config.

TRY IT NOW Update the app configuration with the ConfigMap from code listing 4.9. That increases the logging level, so the same Pod will now start writing more log entries.

```
# check the current app logs:
kubectl logs -l app=todo-web

# deploy the updated ConfigMap:
kubectl apply -f todo-list/configMaps/todo-web-config-dev-with-logging.yaml

# wait for the config change to make it to the Pod:
sleep 120

# check the new setting:
kubectl exec deploy/todo-web -- sh -c 'ls -l /app/config/*.json'

# load a few pages from the site at your service IP address

# check the logs again:
kubectl logs -l app=todo-web
```

You can see my output in figure 4.10—the sleep is there to give the Kubernetes API time to roll out the new config files to the Pod; after a couple of minutes the new configuration is loaded and the app is operating with enhanced logging.

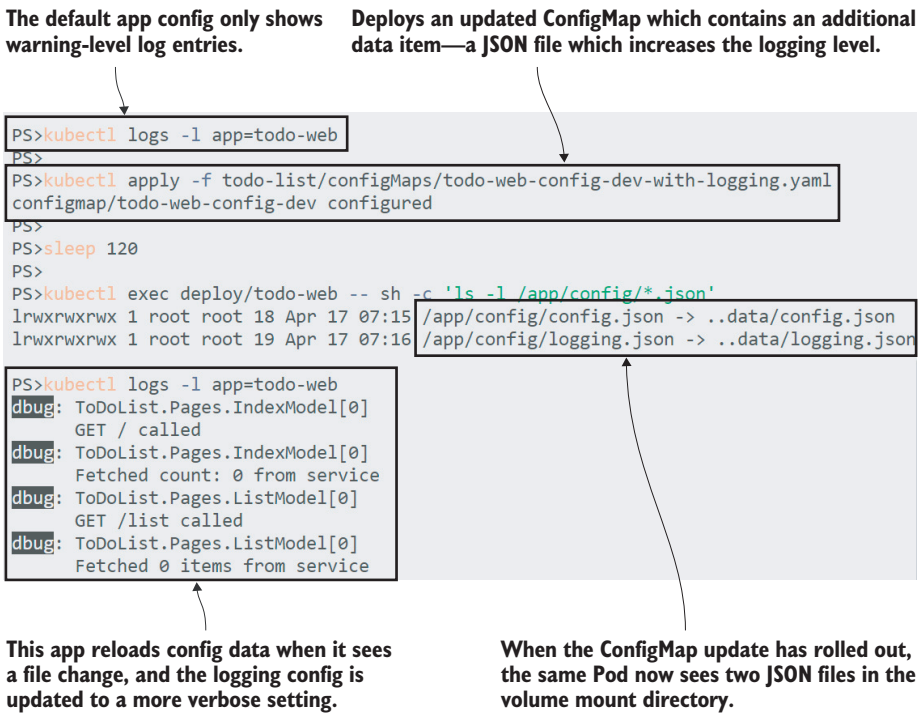

Figure 4.10 ConfigMap data is cached, so it takes couple of minutes for updates to reach Pods.

Volumes are a powerful option for loading config, especially with apps like this that react to changes and update settings on the fly. Bumping up the logging level without having to restart your app is a great help in tracking down issues. But you need to be careful with your configuration because volume mounts don't necessarily work the way you expect. If the mount path for a volume already exists in the container image, then the ConfigMap directory overwrites it, replacing all the contents—which can cause your app to fail in exciting ways. Code listing 4.9 shows an example.

Listing 4.9 todo-web-dev-broken.yaml, a Pod spec with a misconfigured mount

```
spec:
  containers:
   –name: web
      image: kiamol/ch04-todo-list
      volumeMounts:
       –name: config                    # mounts the ConfigMap volume
          mountPath: "/app"             # this will overwrite the directory
```

This is a broken Pod spec, where the ConfigMap is loaded into the `/app` directory rather than the `/app/config` directory. The author probably intended this to merge the directories, adding the JSON config files to the existing app directory. Instead it's going to wipe out the application binaries.

TRY IT NOW The Pod spec from code listing 4.9 removes all the app binaries, so the replacement Pod won't start. See what happens next.

```
# deploy the badly-configured Pod:
kubectl apply -f todo-list/todo-web-dev-broken.yaml

# browse back to the app and see how it looks

# check the app logs:
kubectl logs -l app=todo-web

# and check the Pod status:
kubectl get pods -l app=todo-web
```

The results here are interesting—the deployment breaks the app, and yet the app carries on working. That's Kubernetes watching out for you. Applying the change creates a new Pod, and the container in that Pod immediately exits with an error, because the binary it tries to load no longer exists in the app directory. Kubernetes restarts the container a few times to give it a chance, but it keeps failing—after three tries Kubernetes takes a rest, as you can see in figure 4.11.

Now we have two Pods, but Kubernetes doesn't remove the old Pod until the replacement is running successfully, which it never will in this case because we've broken the container setup. The old Pod doesn't get removed and still happily serves requests; the new Pod is in a failed state, but Kubernetes periodically keeps restarting the container in the hope that it might have fixed itself. This is a situation to watch out for—the `apply` command seems to work, and the app carries on working, but it's not using the manifest you've applied.

We'll fix that now and show one final option for surfacing ConfigMaps in the container filesystem. You can selectively load data items into the target directory, rather than loading every data item as its own file. Code listing 4.10 shows the updated Pod spec—the mount path has been fixed, but the volume is set to deliver only one item.

Listing 4.10 todo-web-dev-no-logging.yaml, mounting a single ConfigMap item

```
spec:
  containers:
   –name: web
      image: kiamol/ch04-todo-list
      volumeMounts:
       –name: config                  # mounts the ConfigMap volume
          mountPath: "/app/config"    # to the correct direcory
          readOnly: true
  volumes:
   –name: config
```

```
      configMap:
        name: todo-web-config-dev      # loads the ConfigMap volume
        items:                         # specifies the data items to load
       —key: config.json               # loads the config.json item
          path: config.json            # surfaces it as the file config.json
```

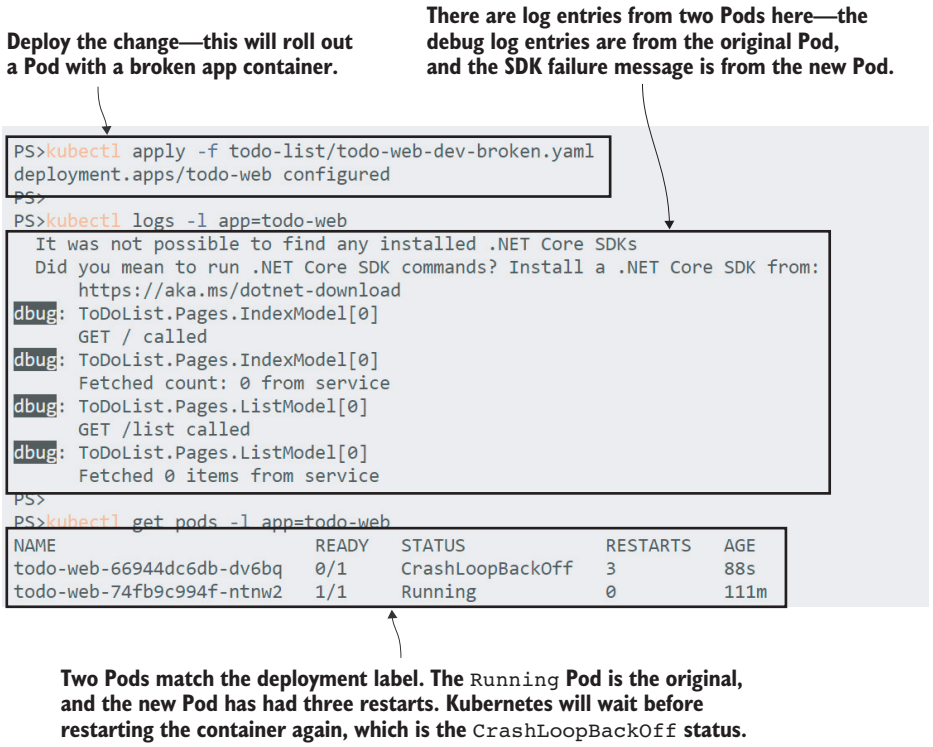

Figure 4.11 **If an updated deployment fails then the original Pod doesn't get replaced.**

This specification uses the same ConfigMap, so it is just an update to the Deployment. Actually this will be a cascading update—it will create a new Pod which will start correctly, and then Kubernetes will remove the two previous Pods.

TRY IT NOW Deploy the spec from code listing 4.10 which rolls out the updated volume mount to fix the app but also ignores the logging JSON file in the ConfigMap.

```
# apply the change:
kubectl apply -f todo-list/todo-web-dev-no-logging.yaml

# list the config folder contents:
kubectl exec deploy/todo-web -- sh -c 'ls /app/config'

# now browse to a few pages on the app

# check the logs:
```

```
kubectl logs -l app=todo-web

# and check the Pods:
kubectl get pods -l app=todo-web
```

You can see my output in figure 4.12—the app is working again, but it only sees a single configuration file, so the enhanced logging settings don't get applied.

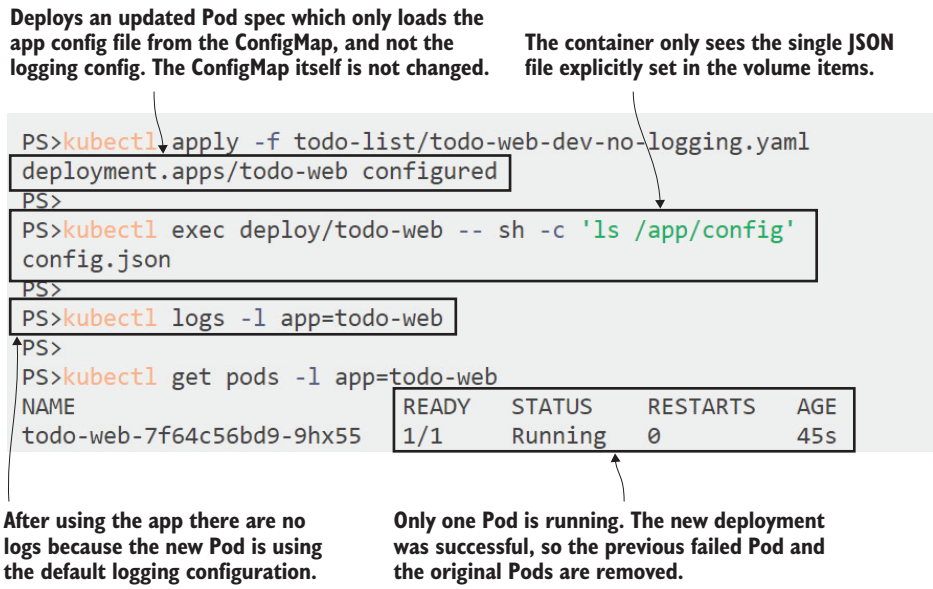

Figure 4.12 Volumes can surface selected items from a ConfigMap into a mount directory.

ConfigMaps support a wide range of configuration systems. Between environment variables and volume mounts you should be able to store app settings in ConfigMaps and apply them however your app expects. The separation between the config spec and the app spec also supports different release workflows, allowing different teams to own different parts of the process. But one thing you shouldn't use ConfigMaps for is any sensitive data; they're effectively wrappers for text files with no additional security semantics. For configuration data that you need to keep secure, Kubernetes provides Secrets.

4.4 Configuring sensitive data with Secrets

Secrets are a separate type of resource, but they have a very similar API to ConfigMaps. You work with them in the same way, but because they're meant to store sensitive information, Kubernetes manages them differently. The main differences are all around minimizing exposure—Secrets are only sent to nodes which need to use them and are stored in memory rather than on disk; Kubernetes also supports encryption in transit and at rest for Secrets.

Secrets are not encrypted 100% of the time though, anyone who has access to Secret objects in your cluster can read the unencrypted values. There is an obfuscation layer—Kubernetes can read and write Secret data with Base64 encoding—which isn't really a security feature but does prevent accidental exposure of secrets to someone looking over your shoulder.

TRY IT NOW You can create Secrets from a literal value, passing the key and data into the Kubectl command. The retrieved data is Base64 encoded.

```
# FOR WINDOWS USERS—this script adds a base64 command to your session:
. .\base64.ps1

# now create a secret from a plain text literal:
kubectl create secret generic sleep-secret-literal --from-
     literal=secret=shh...

# show the friendly details of the secret:
kubectl describe secret sleep-secret-literal

# retrieve the encoded secret value:
kubectl get secret sleep-secret-literal -o jsonpath='{.data.secret}'

# and decode the data:
kubectl get secret sleep-secret-literal -o jsonpath='{.data.secret}' | base64
     -d
```

You can see from the output in figure 4.13 that Kubernetes treats Secrets differently from ConfigMaps—the data values aren't shown in the Kubectl describe command, only the names of the item keys, and when you do fetch the data it's shown encoded so you need to pipe it into a decoder to read the actual data.

That precaution doesn't apply when Secrets are surfaced inside Pod containers—the container environment sees the original plain text data. Code listing 4.11 shows a return to the sleep app, configured to load the new Secret as an environment variable.

Listing 4.11 sleep-with-secret.yaml, a Pod spec loading a Secret

```
spec:
  containers:
   -name: sleep
      image: kiamol/ch03-sleep
      env:                                  # environment variables
     -name: KIAMOL_SECRET                   # variable name in the container
        valueFrom:                          # loaded from an external source
          secretKeyRef:                     # which is a Secret
            name: sleep-secret-literal      # name of the Secret
            key: secret                     # key of the Secret data item
```

The specification to consume Secrets is almost the same as for ConfigMaps—a named environment variable can be loaded from a named item in a Secret. This Pod spec delivers the Secret item in its original form to the container.

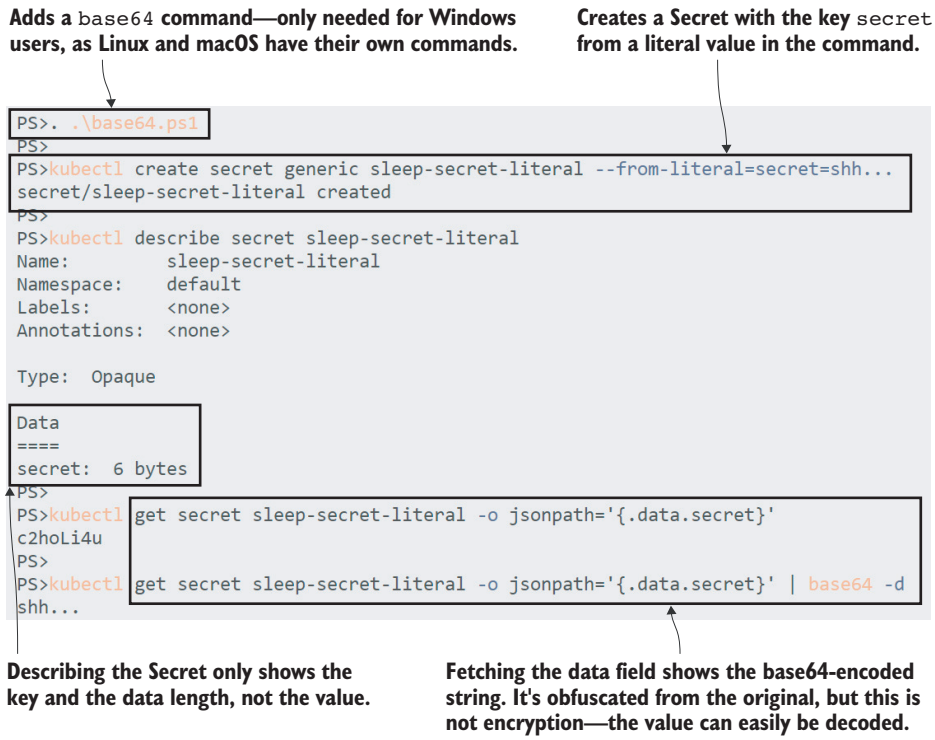

Figure 4.13 Secrets have a similar API to ConfigMaps, but Kubernetes tries to avoid accidental exposure.

TRY IT NOW Run a simple sleep Pod that uses the Secret as an environment variable.

```
# update the sleep Deployment:
kubectl apply -f sleep/sleep-with-secret.yaml

# check the environment variable in the Pod:
kubectl exec deploy/sleep -- printenv KIAMOL_SECRET
```

Figure 4.14 shows the output. In this case the Pod is only using a Secret, but Secrets and ConfigMaps can be mixed in the same Pod spec, populating environment variables or files or both.

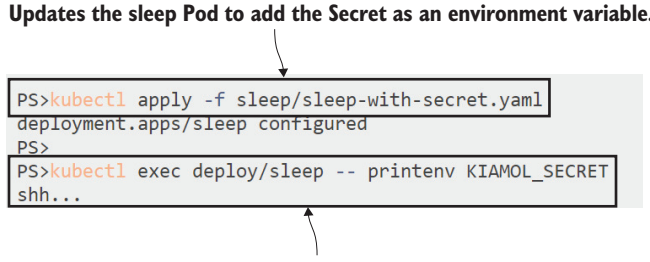

Figure 4.14 Secrets loaded into Pods are not Base64 encoded.

Loading Secrets into environment variables is something you should be wary of. Securing sensitive data is all about minimizing its exposure, and environment variables can be read from any process in the Pod container—and some application platforms log all environment variable values if they hit a critical error. The alternative is to surface Secrets as files, if the application supports it, which gives you the option of securing access with file permissions.

To round off this chapter we'll run the to-do app in a different configuration where it uses a separate database to store items, running in its own Pod. The database server is Postgres using the official image on Docker Hub, which reads logon credentials from configuration values in the environment. Code listing 4.12 shows a YAML spec for creating the database password as a Secret.

Listing 4.12 todo-db-secret-test.yaml, a Secret for the database user

```
apiVersion: v1
kind: Secret                              # Secret is the resource type
metadata:
  name: todo-db-secret-test               # the name of the Secret
type: Opaque                              # Opaque secrets are for text data
stringData:                               # stringData is for plain text
  POSTGRES_PASSWORD: "kiamol-2*2*"        # the secret key and value
```

This approach states the password in plain text in the `stringData` field, which gets encoded to Base64 when you create the secret. Using YAML files for secrets poses a tricky problem: it gives you a nice consistent deployment approach, at the cost of having all your sensitive data visible in source control . . .

In a production scenario you would keep the real data out of the YAML file, using placeholders instead, and do some additional processing as part of your deployment—something like injecting the data into the placeholder from a GitHub secret. Whichever approach you take, you need to remember that once the secret exists in Kubernetes, it's easy for anyone who has access to read the value.

TRY IT NOW Create a secret from the manifest in code listing 4.12 and check the data.

```
# deploy the Secret:
kubectl apply -f todo-list/secrets/todo-db-secret-test.yaml

# check the data is encoded:
kubectl get secret todo-db-secret-test -o
     jsonpath='{.data.POSTGRES_PASSWORD}'

# and see what annotations are stored:
kubectl get secret todo-db-secret-test -o jsonpath='{.metadata.annotations}'
```

You can see in figure 4.15 that the string is encoded to Base64—the outcome is the same as it would be if the specification used the normal data field and set the password value in Base64 directly in the YAML.

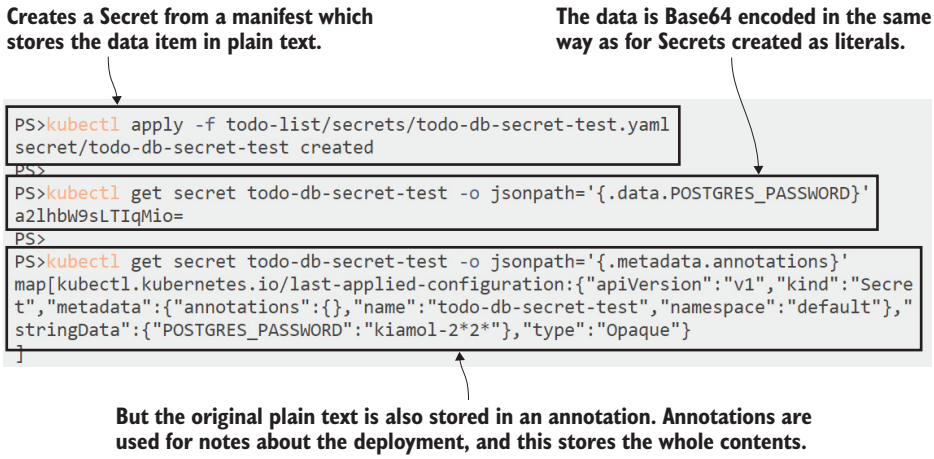

Figure 4.15 **Secrets created from string data are encoded, but the original data is also stored in the object.**

To use that Secret as the Postgres password, the image gives us a couple of options. We can load the value into an environment variable called `POSTGRES_PASSWORD`—which is not ideal—or we can load it into a file and tell Postgres where to load the file by setting the `POSTGRES_PASSWORD_FILE` environment variable. Using a file means we can control access permissions at the volume level, which is how the database is configured in code listing 4.13.

Listing 4.13 todo-db-test.yaml, a Pod spec mounting a volume from a secret

```
spec:
  containers:
   -name: db
      image: postgres:11.6-alpine
      env:
     -name: POSTGRES_PASSWORD_FILE          # sets the path to the file
        value: /secrets/postgres_password
      volumeMounts:                         # mounts a Secret volume
       -name: secret                        # the name of the volume
          mountPath: "/secrets"
  volumes:
   -name: secret
      secret:                               # volume loaded from a Secret
        secretName: todo-db-secret-test     # Secret name
        defaultMode: 0400                   # permissions to set for files
        items:                              # optionally name the data items
       -key: POSTGRES_PASSWORD
          path: postgres_password
```

When this Pod is deployed, Kubernetes will load the value of the Secret item into a file at the path /secrets/postgres_password. That file will be set with 0400 permissions, which means it can be read by the container user but not by any other users. The envi-

ronment variable is set for Postgres to load the password from that file, which the Postgres user has access to, so the database will start with credentials set from the Secret.

TRY IT NOW Deploy the database Pod, and verify the database starts correctly.

```
# deploy the YAML from code listing 4.13
kubectl apply -f todo-list/todo-db-test.yaml

# check the database logs:
kubectl logs -l app=todo-db --tail 1

# verify the password file permissions:
kubectl exec deploy/todo-db -- sh -c 'ls -l $(readlink -f
     /secrets/postgres_password)'
```

Figure 4.16 shows the database starting up and waiting for connections—so it has been configured correctly—and the final output verifies that the file permissions are set as expected.

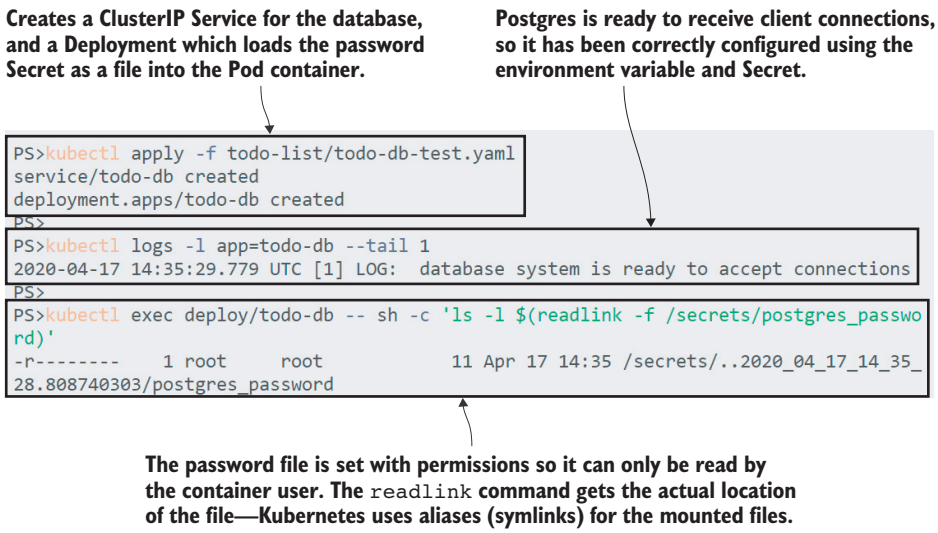

Figure 4.16 If the app supports it, configuration settings can be read by files populated from Secrets.

All that's left is to run the app itself in the test configuration, so it connects to the Postgres database rather than using a local database file for storage. There's lots more YAML for that, to create a ConfigMap, Secret, Deployment and Service, but it's all using features we've covered already, so we'll just go ahead and deploy.

TRY IT NOW Run the to-do app so it uses the Postgres database for storage.

```
# the ConfigMap configures the app to use Postgres:
kubectl apply -f todo-list/configMaps/todo-web-config-test.yaml

# the Secret contains the credentials to connect to Postgres:
kubectl apply -f todo-list/secrets/todo-web-secret-test.yaml

# the Deployment Pod spec uses the ConfigMap and Secret:
kubectl apply -f todo-list/todo-web-test.yaml

# check the database credentials are set in the app:
kubectl exec deploy/todo-web-test -- cat /app/secrets/secrets.json

# browse to the app and add some items
```

My output is in figure 4.17, where the plain text contents of the Secret JSON file are shown inside the web Pod container.

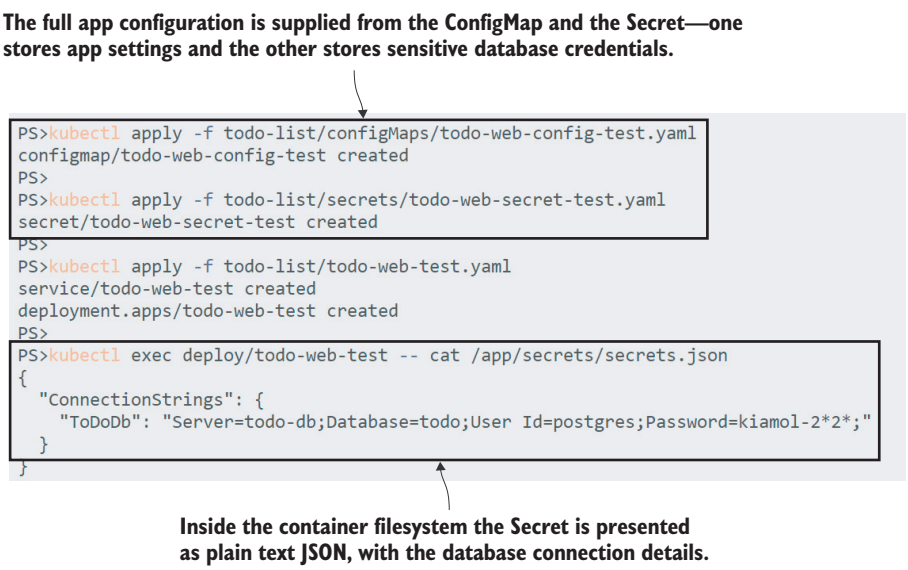

Figure 4.17 Loading app configuration into Pods—surfacing ConfigMaps and Secrets as JSON files.

Now when you add to-do items in the app they get stored in the Postgres database, so storage is separated from the application runtime. You can delete the web Pod and its controller will start a replacement with the same configuration which connects to the same database Pod, so all the data from the original web Pod is still available.

That was a pretty exhaustive look at configuration options in Kubernetes. The principles are quite simple—loading ConfigMaps or Secrets into environment variables or files—but there are a lot of variations. You need a good understanding of the nuances so you can manage app configuration in a consistent way, even if your apps all have different configuration models.

4.5 Managing app configuration in Kubernetes

Kubernetes gives you the tools to manage app configuration using whatever workflow fits for your organization. The core requirement is for your applications to read configuration settings from the environment, ideally with a hierarchy of files and environment variables. Then you have the flexibility to use ConfigMaps and Secrets to support your deployment process. There are two factors you need to consider in your design: do you need your apps to respond to live config updates, and how will you manage secrets?

If live updates without a Pod replacement are important to you, then your options are limited—you can't use environment variables for settings, because any changes to those will mean a Pod replacement. You can use a volume mount and load config from files, but you need to deploy changes by updating the existing ConfigMap or Secret objects—you can't change the volume to point to a new config object, because that's a Pod replacement too.

The alternative to updating the same config object is to deploy a new object every time with some versioning scheme in the object name, and updating the app Deployment to reference the new object. You lose live updates but gain an audit trail of configuration changes and have an easy option to revert back to previous settings. Figure 4.18 shows those options.

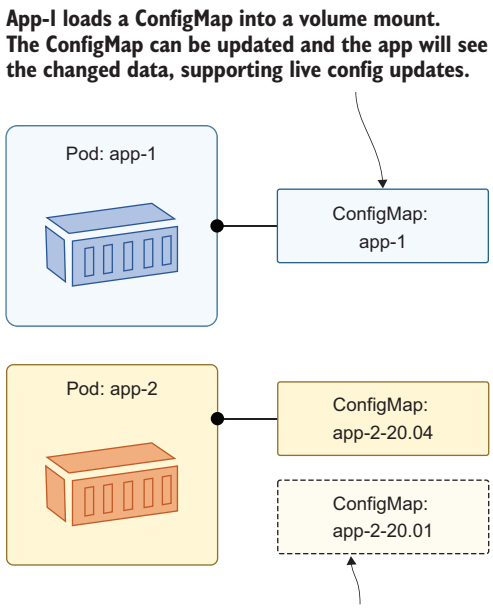

Figure 4.18 You can choose your own approach to config management, supported by Kubernetes.

The other question is how you manage sensitive data. Large organizations might have dedicated configuration management teams who own the process of deploying con-

figuration files. That fits nicely with a versioned approach to ConfigMaps and Secrets, where the config management team deploy new objects from literals or controlled files in advance of the deployment.

An alternative is a fully automated deployment, where ConfigMaps and Secrets are created from YAML templates in source control. The YAML files contain placeholders instead of sensitive data, and the deployment process replaces them with real values from a secure store like Azure KeyVault before applying them. Figure 4.19 compares those options.

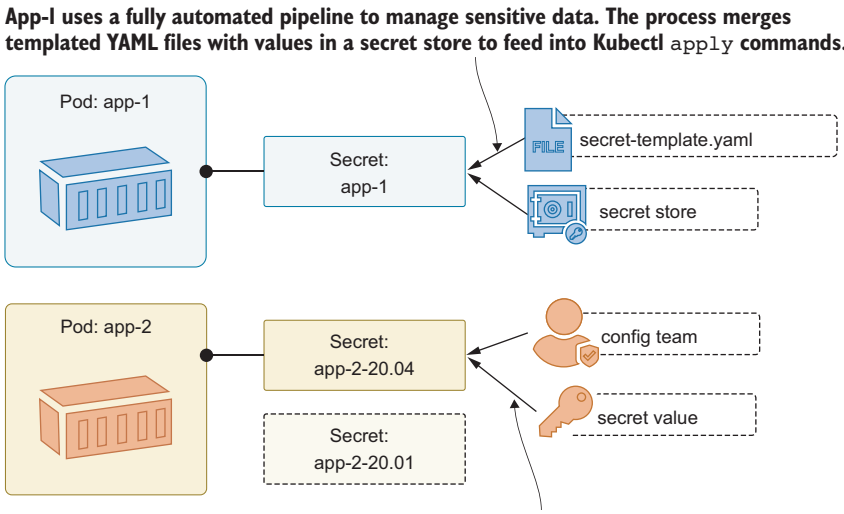

Figure 4.19 Secret management can be automated in deployment or strictly controlled by a separate team.

You can use any approach that works for your teams and your application stacks, remembering that the goal is for all configuration settings to be loaded from the platform, so the same container image is deployed in every environment.

It's time to clean up your cluster. If you've followed along with all the exercises (and of course you have . . .), you'll have a couple of dozen resources to remove. I'll introduce some very useful features of Kubectl to help clear everything out.

TRY IT NOW The Kubectl `delete` command can read a YAML file and delete the resources defined in the file. And if you have multiple YAML files in a directory, you can use the directory name as the argument to `delete` (or `apply`) and it will run over all the files.

```
# delete all the resources in all the files in all the directories:
kubectl delete -f sleep/
kubectl delete -f todo-list/
```

```
kubectl delete -f todo-list/configMaps/
kubectl delete -f todo-list/secrets/
```

4.6 Lab

If you're reeling from all the options Kubernetes gives you to configure apps, this lab is going to help. In practice, your apps will have their own ideas about config management, and you'll need to model your Kubernetes Deployments to suit the way your apps expect to be configured. That's what you need to do in this lab with a simple app called Adminer. Here we go:

- Adminer is a web UI for administering SQL databases, and it can be a handy tool to run in Kubernetes when you're troubleshooting database issues.
- Start by deploying the YAML files in the `ch04/lab/postgres` folder, then deploy the ch04/lab/adminer.yaml file to run Adminer in its basic state.
- Find the external IP for your Adminer service and browse to port 8082. Note that you need to specify a database server, and that the UI design is stuck in the 1990s. You can confirm the connection to Postgres by using `postgres` as the database name, username and password.
- Your job is to create and use some config objects in the Adminer Deployment so that the database server name defaults to the lab's Postgres service, and the UI uses the much nicer design called `price`.
- You can set the default database server in an environment variable called `ADMINER_DEFAULT_SERVER`. Let's call this sensitive data, so it should use a Secret.
- The UI design is set in the environment variable `ADMINER_DESIGN`—that's not sensitive so a ConfigMap will do nicely.

This will take a little bit of investigation and some thought on how to surface the configuration settings, so it's good practice for real application config. My solution is posted on GitHub for you to check your approach:

https://github.com/sixeyed/kiamol/blob/master/ch04/lab/README.md

5 Storing data with volumes, mounts, and claims

Data access in a clustered environment is difficult. Moving compute around is the easy part—the Kubernetes API is in constant contact with the nodes, and if a node stops responding, then Kubernetes can assume it's offline and start replacements for all of its Pods on other nodes. But if an application in one of those Pods was storing data on the node, then the replacement won't have access to that data when it starts on a different node—and it would be disappointing if that data contained a very large order which a customer hadn't completed. You really need cluster-wide storage, so Pods can access the same data from any node.

Kubernetes doesn't have built-in cluster-wide storage because there isn't a single solution that works for every scenario. Apps have different storage requirements, and the platforms that can run Kubernetes have different storage capabilities. Data is always a balance between speed of access and durability, and Kubernetes supports that by allowing you to define the different classes of storage which your cluster provides, and to request a specific storage class for your application. In this chapter you'll learn how to work with different types of storage and how Kubernetes abstracts away storage implementation details.

5.1 How Kubernetes builds the container filesystem

Containers in Pods have their filesystem constructed by Kubernetes, using multiple sources. The container image provides the initial contents of the filesystem, and every container has a writeable storage layer that it uses to write new files or to update any files from the image (Docker images are read-only, so when a container

updates a file from the image, it's actually updating a copy of the file in its own writeable layer). Figure 5.1 shows how that looks inside the Pod.

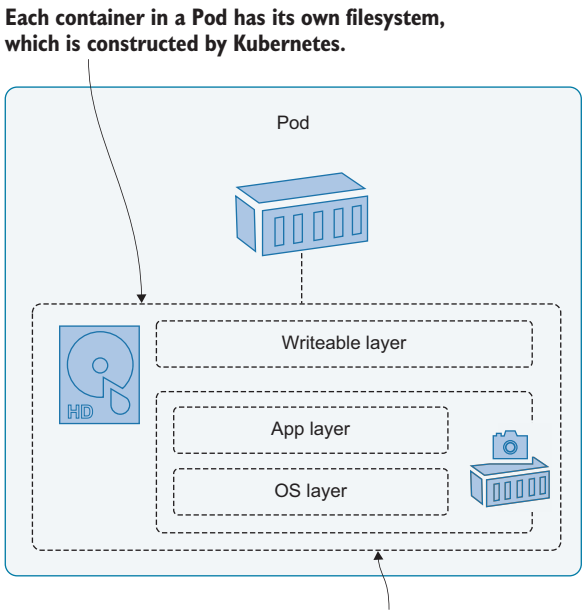

Figure 5.1 Containers don't know it, but their filesystem is a virtual construct, built by Kubernetes.

The application running in the container just sees a single filesystem to which it has read and write access, and all those layer details are hidden. That's great for moving apps to Kubernetes because they don't need to change to run in a Pod. But if your apps do write data, you will need to understand how they use storage and design your Pods to support their requirements. Otherwise your apps will seem to be running fine but you're setting yourself up for data loss when anything unexpected happens—like a container restarting inside a Pod.

> **TRY IT NOW** If the app inside a container crashes and the container exits, then the Pod will start a replacement. The new container will start with the filesystem from the container image and a new writeable layer, and any data written by the previous container in its writeable layer is gone.

```
#switch to this chapter's exercise directory:
cd ch05

# deploy a sleep Pod:
kubectl apply -f sleep/sleep.yaml

# write a file inside the container:
kubectl exec deploy/sleep -- sh -c 'echo ch05 > /file.txt; ls /*.txt'

# check the container ID:
```

```
kubectl get pod -l app=sleep -o
     jsonpath='{.items[0].status.containerStatuses[0].containerID}'

# kill all processes in the container—causing a Pod restart:
kubectl exec -it deploy/sleep -- killall5

# check the replacment container ID:
kubectl get pod -l app=sleep -o
     jsonpath='{.items[0].status.containerStatuses[0].containerID}'

# look for the file you wrote—it won't be there:
kubectl exec deploy/sleep -- ls /*.txt
```

There's two important things to remember from this exercise: the filesystem of a Pod container has the lifecycle of the container rather than the Pod, and when Kubernetes talks about a Pod restart, it's actually a replacement container. So if your apps are merrily writing data inside containers, it doesn't get stored at the Pod level, and if the Pod restarts with a new container then all the data is gone. My output in figure 5.2 shows that.

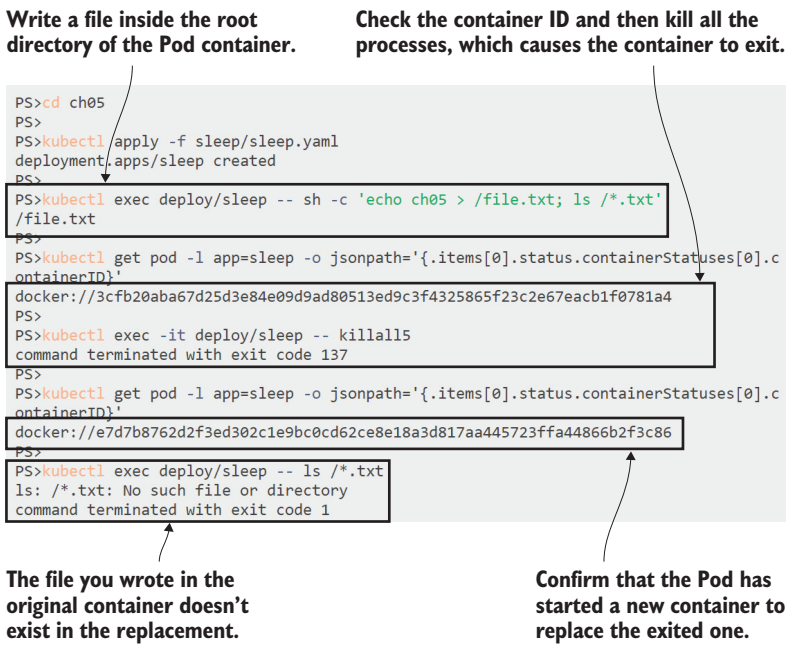

Figure 5.2 The writeable layer has the lifecycle of the container, not the Pod.

We already know that Kubernetes can build the container filesystem from other sources—we surfaced ConfigMaps and Secrets into filesystem directories in chapter 4. The mechanism for that is to define a volume at the Pod level which makes another storage source available, and then to mount it into the container filesystem at a specified path. ConfigMaps and Secrets are read-only storage units, but Kubernetes supports many other types of volume that are writeable. Figure 5.3 shows how you can design a Pod that uses a volume to store data that persists between restarts and could even be accessible cluster-wide.

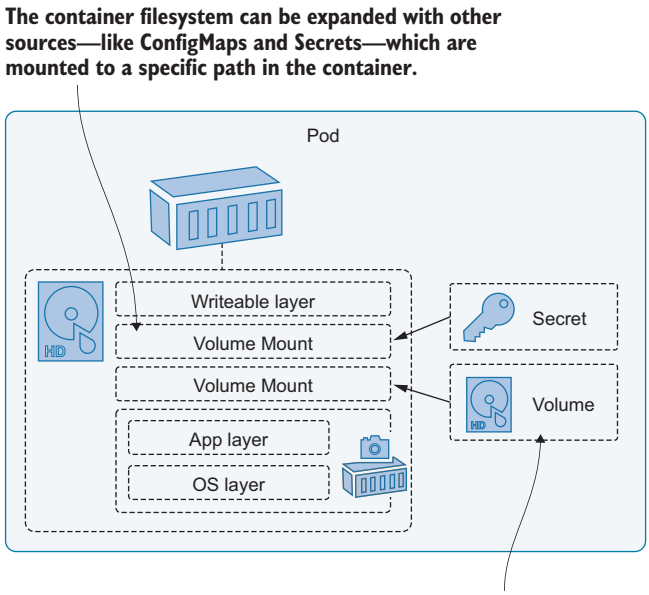

Figure 5.3 The virtual filesystem can be built from volumes which refer to external pieces of storage.

We'll come to cluster-wide volumes later in the chapter, but we'll start with a much simpler volume type that is still useful for many scenarios. Listing 5.1 shows a Pod spec using a type of volume called EmptyDir, which is just an empty directory—but it's stored at the Pod level rather than the container level. It gets mounted as a volume into the container, so it's visible as a directory, but it's not one of the image or container layers.

Listing 5.1 sleep-with-emptyDir.yaml, a simple volume spec

```
spec:
  containers:
   –name: sleep
      image: kiamol/ch03-sleep
      volumeMounts:
      –name: data                    # mounts a volume called data
          mountPath: /data           # into the /data directory
  volumes:
   –name: data                       # this is the data volume spec
      emptyDir: {}                   # which is the EmptyDir type
```

An empty directory sounds like the least useful piece of storage you can imagine, but it actually has a lot of uses because it has the lifecycle of the Pod. Any data stored in an EmptyDir volume remains in the Pod between restarts, so replacement containers can access data written by their predecessors.

TRY IT NOW Update the sleep deployment using the spec from listing 5.1, adding an EmptyDir volume. Now you can write data and kill the container and the replacement can read the data.

```
# update the sleep Pod to use an EmptyDir volume:
kubectl apply -f sleep/sleep-with-emptyDir.yaml

# list the contents of the volume mount:
kubectl exec deploy/sleep -- ls /data

# create a file in the empty directory:
kubectl exec deploy/sleep -- sh -c 'echo ch05 > /data/file.txt; ls /data'

# check the container ID:
kubectl get pod -l app=sleep -o
     jsonpath='{.items[0].status.containerStatuses[0].containerID}'

# kill the container processes:
kubectl exec deploy/sleep -- killall5

# check replacement container ID:
kubectl get pod -l app=sleep -o
     jsonpath='{.items[0].status.containerStatuses[0].containerID}'

# read the file in the volume:
kubectl exec deploy/sleep -- cat /data/file.txt
```

You can see my output in figure 5.4—the containers just see a directory in the filesystem, but that points to a storage unit that is part of the Pod.

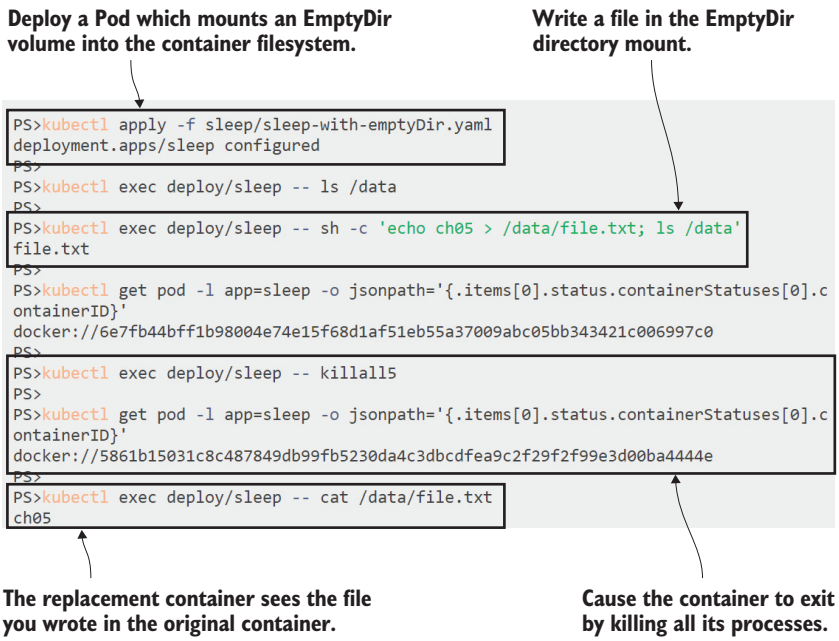

Figure 5.4 Something as basic as an empty directory is still useful because it can be shared by containers.

You can use EmptyDir volumes for any applications which use the filesystem for temporary storage—maybe your app calls an API that takes a few seconds to respond, and the

response is valid for a long time. The app might save the API response in a local file because reading from disk is faster than repeating the API call. An EmptyDir volume is a reasonable source for a local cache because if the app crashes then the replacement container will still have the cached files and still benefit from the speed boost.

EmptyDir volumes only share the lifecycle of the Pod, so if the Pod gets replaced then the new Pod starts with—well, an empty directory. If you want your data to persist between Pods, then you can mount other types of volume which have their own lifecycles.

5.2 Storing data on a node with volumes and mounts

This is where working with data gets trickier than working with compute, because we need to think about whether data will be tied to a particular node—so any replacement Pods will need to run on that node to see the data—or whether the data has cluster-wide access and the Pod can run on any node. Kubernetes supports many variations but you need to know what you want and what your cluster supports, and specify that for the Pod.

The simplest storage option is to use a volume which maps to a directory on the node, so when the container writes to the volume mount, the data is actually stored in a known directory on the node's disk. We'll demonstrate that by running a real app which uses an EmptyDir volume for cache data, understanding the limitations and then upgrading it to use node-level storage.

> **TRY IT NOW** Run a web application which uses a proxy component to improve performance. The web app runs in a Pod as an internal service and the proxy runs in another Pod which is publicly available on a load-balancer service.

```
# deploy the Pi application:
kubectl apply -f pi/v1/

# wait for the web Pod to be ready:
kubectl wait --for=condition=Ready pod -l app=pi-web

# find the app URL from your loadbalancer:
kubectl get svc pi-proxy -o
     jsonpath='http://{.status.loadBalancer.ingress[0].*}:8080/?dp=30000'

# browse to the URL, wait for the response then refresh the page

# check the cache in the proxy
kubectl exec deploy/pi-proxy -- ls -l /data/nginx/cache
```

This is a common setup for web applications, where the proxy boosts performance by serving responses directly from its local cache, and that also reduces load on the web app. You can see my output in figure 5.5—the first Pi calculation took over one second to respond, and the refresh was practically immediate because it came from the proxy and did not need to be calculated.

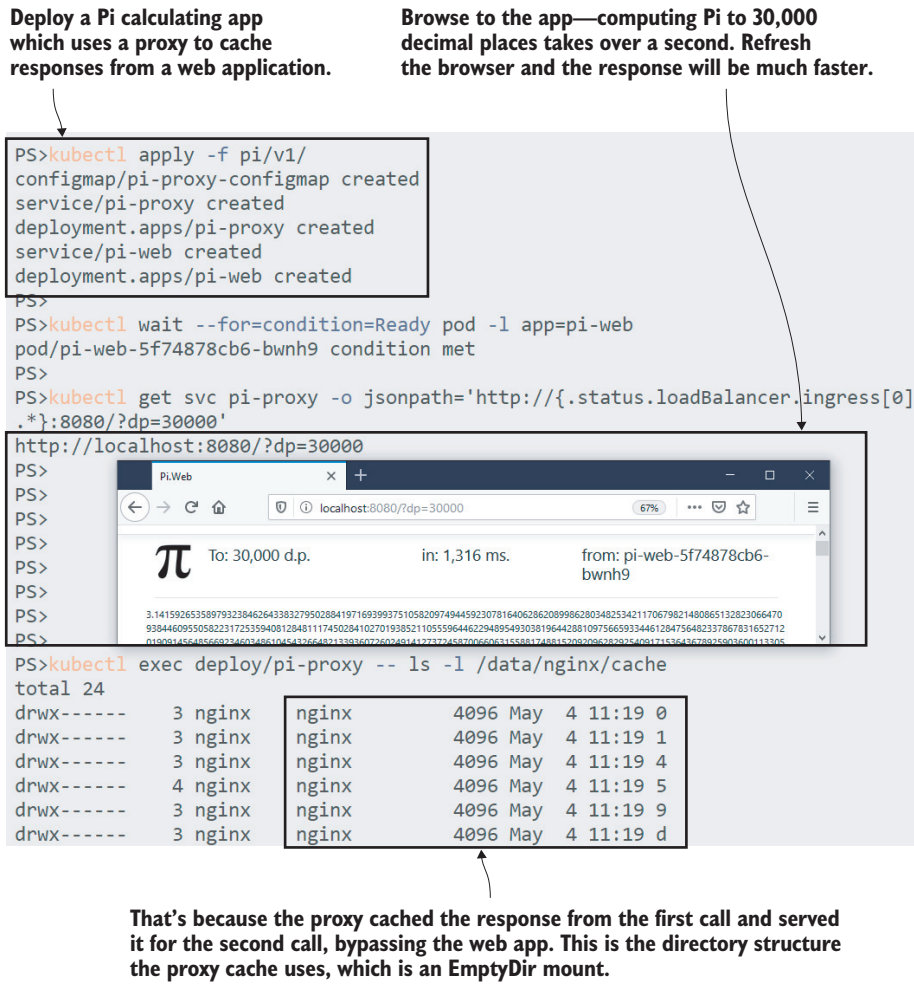

Figure 5.5 **Caching files in an EmptyDir volume means the cache survives Pod restarts.**

An EmptyDir volume could be a reasonable approach for an app like this because the data stored in the volume is not critical. If there's a Pod restart then the cache survives and the new proxy container can serve responses cached by the previous container. If the Pod gets replaced, then the cache is lost. The replacement Pod starts with an empty cache directory, but the cache isn't required—the app still functions correctly; it just starts off slow until the cache gets filled again.

TRY IT NOW Remove the proxy Pod and it will be replaced because it's managed by a deployment controller. The replacement starts with a new EmptyDir volume, which for this app means an empty proxy cache so requests get sent on to the web Pod.

```
# delete the proxy Pod:
kubectl delete pod -l app=pi-proxy

# check the cache directory of the replacement Pod:
kubectl exec deploy/pi-proxy -- ls -l /data/nginx/cache

# refresh your browser at the Pi app URL
```

You can see my output in figure 5.6—the result is the same, but I had to wait another second for it to be calculated by the web app, because the replacement proxy Pod started without a cache.

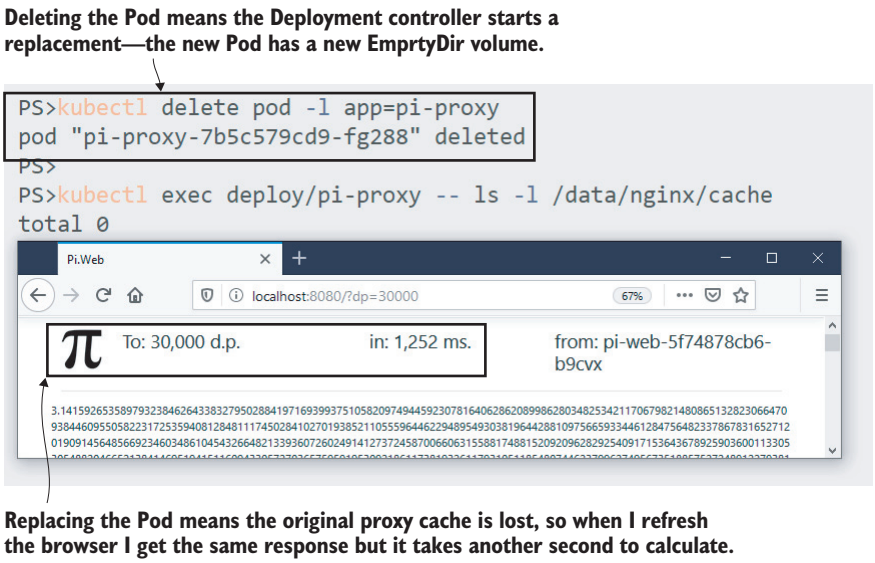

Figure 5.6 A new Pod starts with a new empty directory, which is empty . . .

The next level of durability comes from using a volume that maps to a directory on the node's disk, which Kubernetes calls a HostPath volume. HostPaths are specified as a volume in the Pod, which is mounted into the container filesystem in the usual way. When the container writes data into the mount directory it actually gets written to the disk on the node—figure 5.7 shows the relationship between node, Pod, and volume. HostPath volumes can be useful, but you need to be aware of their limitations. Data is physically stored on the node and that's that—Kubernetes doesn't magically replicate that data to all the other nodes in the cluster. Listing 5.2 shows an updated Pod spec for the web proxy which uses a HostPath volume instead of an EmptyDir—when the proxy container writes cache files to `/data/nginx/cache` they will actually be stored on the node at `/volumes/nginx/cache`.

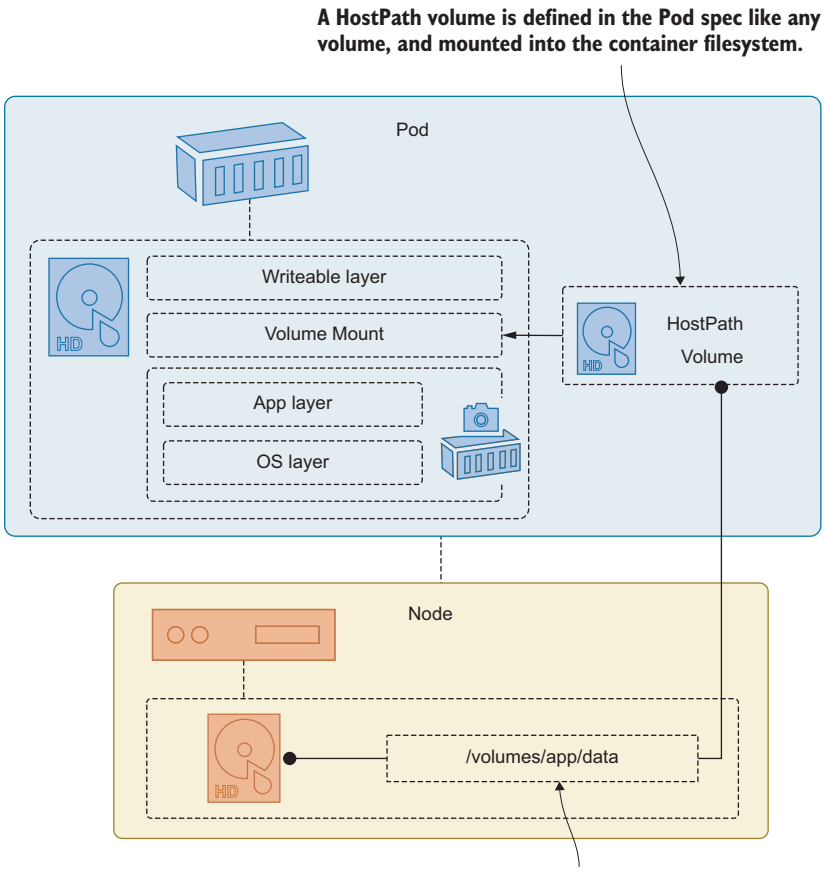

Figure 5.7 HostPath volumes keep data between Pod replacements, but only if Pods use the same node.

Listing 5.2 nginx-with-hostPath.yaml, mounting a HostPath volume

```
spec:                # this is an abridged Pod spec;
  containers:        # the full spec also contains a configMap volume mount
  –image: nginx:1.17-alpine
     name: nginx
     ports:
       –containerPort: 80
     volumeMounts:
       –name: cache-volume
          mountPath: /data/nginx/cache     # the proxy cache path
  volumes:
   –name: cache-volume
      hostPath:                            # using a directory on the node
        path: /volumes/nginx/cache         # the volume path on the node
        type: DirectoryOrCreate            # create path if it doesn't exist
```

This extends the durability of the data beyond the lifecycle of the Pod to the lifecycle of the node's disk—provided replacement Pods always run on the same node. In a single node lab cluster, that will be the case because there's only one node. Replacement Pods will load the HostPath volume when they start, and if it is populated with cache data from a previous Pod then the new proxy can start serving cached data straight away.

TRY IT NOW Update the proxy deployment to use the Pod spec from listing 5.2, then use the app and delete the Pod—the replacement responds using the existing cache.

```
# update the proxy Pod to use a HostPath volume:
kubectl apply -f pi/nginx-with-hostPath.yaml

# list the contents of the cache directory:
kubectl exec deploy/pi-proxy -- ls -l /data/nginx/cache

# browse to the app URL

# delete the proxy Pod:
kubectl delete pod -l app=pi-proxy

# check the cache directory in the replacement Pod:
kubectl exec deploy/pi-proxy -- ls -l /data/nginx/cache

# refresh your browser
```

My output is in figure 5.8. The initial request took just under a second to respond, but the refresh was pretty much instant because the new Pod inherited the cached response from the old Pod, stored on the node.

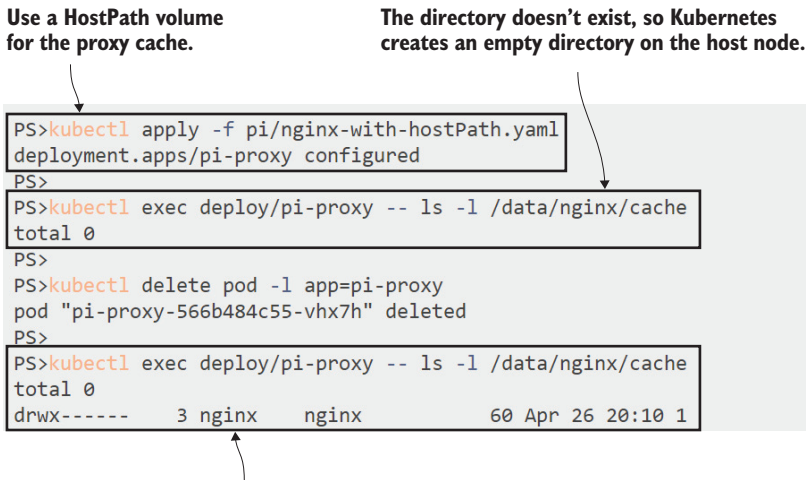

Figure 5.8 On a single-node cluster Pods always run on the same node, so they can all use the HostPath.

The obvious problem with HostPath volumes is that they don't make any sense in a cluster with more than one node, which is pretty much every cluster outside of a simple lab environment. You can include a requirement in your Pod spec to say the Pod should always run on the same node to make sure it goes where the data is, but that just limits the resilience of your solution—if the node goes offline then the Pod won't run and you lose your app.

A less obvious problem is that this is a nice security exploit. Kubernetes doesn't restrict which directories on the node are available to use for HostPath volumes. The Pod spec in listing 5.3 is perfectly valid, and it makes the entire filesystem on the node available for the Pod container to access.

Listing 5.3 sleep-with-hostPath.yaml, a Pod with full access to the node's disk

```
spec:
  containers:
   -name: sleep
      image: kiamol/ch03-sleep
      volumeMounts:
       -name: node-root
          mountPath: /node-root
  volumes:
   -name: node-root
      hostPath:
        path: /                  # the root of the node's filesystem
        type: Directory          # path needs to exist
```

Anyone who has access to create a Pod from that specification now has access to the whole filesystem of the node where the Pod is running. You might be tempted to use a volume mount like this as a quick way to read multiple paths on the host—but if your app is compromised and an attacker can execute commands in the container, then they have access to the node's disk too.

TRY IT NOW Run a Pod from the YAML in listing 5.3, then run some commands in the Pod container to explore the node's filesystem.

```
# run a Pod with a volume mount to the host:
kubectl apply -f sleep/sleep-with-hostPath.yaml

# check the log files inside the container:
kubectl exec deploy/sleep -- ls -l /var/log

# and check the logs on the node using the volume:
kubectl exec deploy/sleep -- ls -l /node-root/var/log

# check the container user:
kubectl exec deploy/sleep -- whoami
```

You can see in figure 5.9 that the Pod container can see the log files on the node—which in my case includes the Kubernetes logs. That's fairly harmless, but this container runs as the root user, which maps to the root user on the node so the container has complete access to the filesystem.

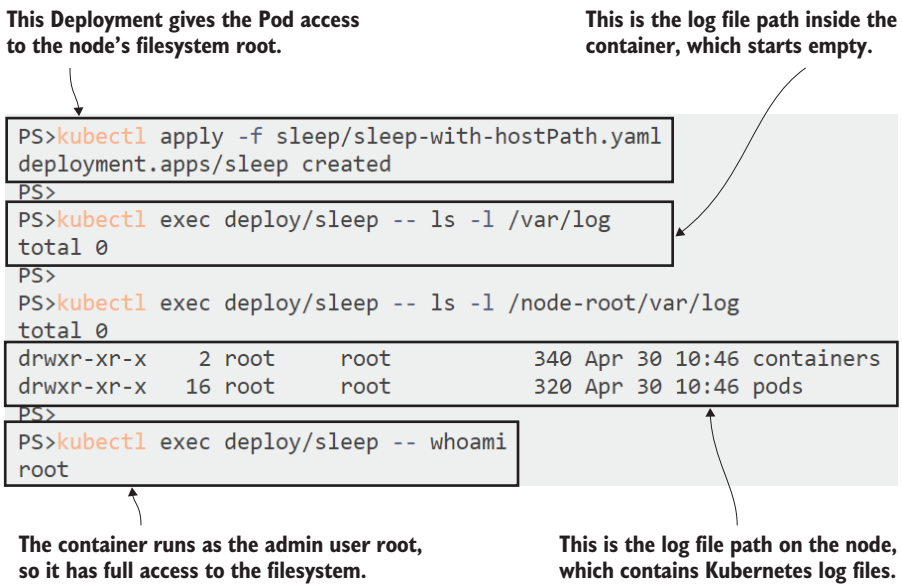

Figure 5.9 Danger! Mounting a HostPath can give you complete access to the data on the node.

If this all seems like a terrible idea, remember that Kubernetes is a platform with a wide range of features to suit many applications. You could have an older app that needs to access specific file paths on the node where it is running, and the HostPath volume lets you do that. In that scenario you can take a safer approach, using a volume that has access to one path on the node, and limiting what the container can see by declaring sub-paths for the volume mount. Listing 5.4 shows that.

Listing 5.4 sleep-with-hostPath-subPath.yaml, restricting mounts with sub-paths

```
spec:
  containers:
   -name: sleep
      image: kiamol/ch03-sleep
      volumeMounts:
       -name: node-root                  # name of the volume to mount
          mountPath: /pod-logs             # target path for the container
          subPath: var/log/pods            # source path within the volume
       -name: node-root
          mountPath: /container-logs
          subPath: var/log/containers
  volumes:
   -name: node-root
      hostPath:
        path: /
        type: Directory
```

Here the volume is still defined at the root path on the node, but the only way to access it is through the volume mounts in the container, which are restricted to defined sub-paths. Between the volume specification and the mount specification you have a lot of flexibility in building and mapping your container filesystem.

TRY IT NOW Update the sleep Pod so the container's volume mount is restricted to the sub-paths defined in listing 5.4 and check the file contents.

```
# update the Pod spec:
kubectl apply -f sleep/sleep-with-hostPath-subPath.yaml

# check the Pod logs on the node:
kubectl exec deploy/sleep -- sh -c 'ls /pod-logs | grep _pi-'

# and the container logs:
kubectl exec deploy/sleep -- sh -c 'ls /container-logs | grep nginx'
```

In this exercise there's no way to explore the node's filesystem, other than through the mounts to the log directories. My output is in figure 5.10—the container can only access files in the sub-paths.

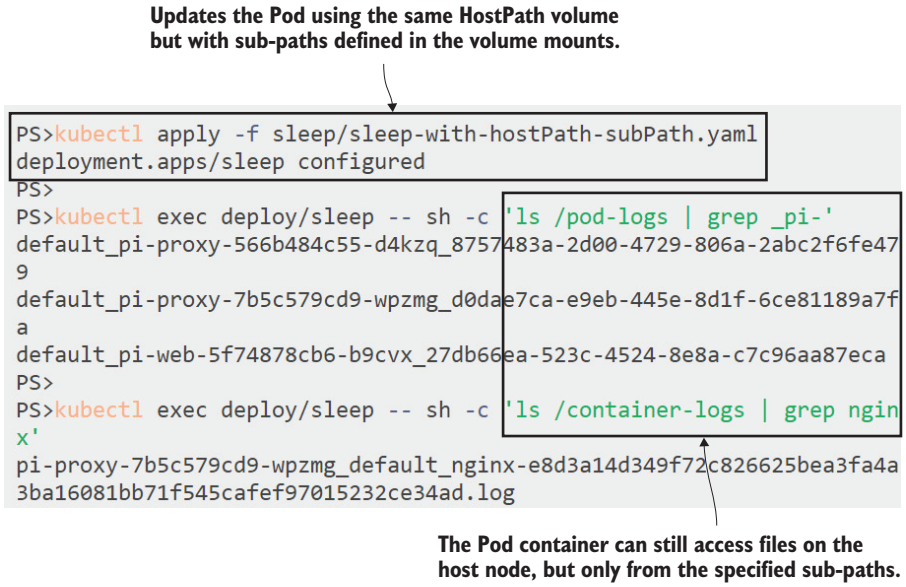

Figure 5.10 Restricting access to volumes with sub-paths limits what the container can do.

HostPath volumes are a good way to start with stateful apps—they're easy to use and they work in the same way on any cluster. They are useful in real-world applications too, but only when your apps are using state for temporary storage. For permanent storage, we need to move onto volumes which can be accessed by any node in the cluster.

5.3 Storing cluster-wide data with persistent volumes and claims

A Kubernetes cluster is like a pool of resources—it has a number of nodes that each have some CPU and memory capacity they make available to the cluster, and Kubernetes uses that to run your apps. Storage is just another resource that Kubernetes makes

available to your application, but it can only provide cluster-wide storage if the nodes can plug into a distributed storage system. Figure 5.11 shows how Pods can access volumes from any node if the volume uses distributed storage.

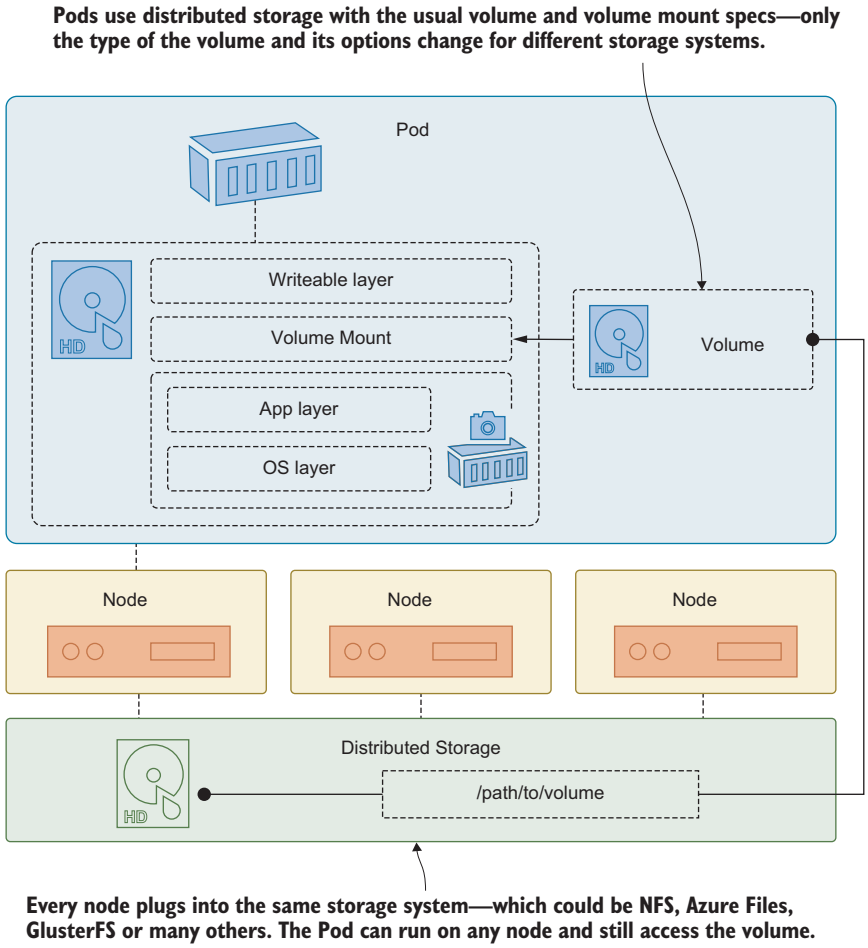

Figure 5.11 Distributed storage gives your Pod access to data from any node but needs platform support.

Kubernetes supports many volume types backed by distributed storage systems—AKS clusters can use Azure Files or Azure Disk, EKS clusters can use Elastic Block Store, and in the datacenter, you can use simple Network File Shares (NFS), or a networked filesystem like GlusterFS. All those systems have different configuration requirements, and you can specify them in the volume spec for your Pod, but that would make your application spec tightly coupled to one storage implementation, and Kubernetes provides a more flexible approach.

Pods are an abstraction over the compute layer and Services are an abstraction over the network layer. In the storage layer, the abstractions are PersistentVolumes and PersistentVolumeClaims. A PersistentVolume is a Kubernetes object which defines an available piece of storage. A cluster administrator may create a set of PersistentVolumes, which each contain the volume spec for the underlying storage system. Listing 5.5 shows a PersistentVolume spec which uses NFS storage.

Listing 5.5 persistentVolume-nfs.yaml, a volume backed by an NFS mount

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv01                    # a generic storage unit with a generic name

spec:
  capacity:
    storage: 50Mi               # the amount of storage the PV offers
  accessModes:                  # how the volume can be accessed by Pods
   –ReadWriteOnce             # it can only be used by one Pod

  nfs:                                  # this PV is backed by NFS
    server: nfs.my.network              # domain name of the NFS server
    path: "/kubernetes-volumes"         # path to the NFS share
```

You won't be able to deploy that spec in your lab environment unless you happen to have an NFS server in your network with the domain name nfs.my.network and a share called kubernetes-volumes. You could be running Kubernetes on any platform, so for the exercises that follow we'll use a local volume that will work anywhere (if I used Azure Files in the exercises they would only work on an AKS cluster, because EKS and Docker Desktop and the other Kubernetes distributions aren't configured for Azure volume types).

> **TRY IT NOW** Create a PersistentVolume (PV) which uses local storage—the PV is cluster-wide but the volume is local to one node, so we need to make sure the PV is linked to the node where the volume lives, and we'll do that with labels.

```
# apply a custom label to the first node in your cluster:
kubectl label node $(kubectl get nodes -o
     jsonpath='{.items[0].metadata.name}') kiamol=ch05

# check the nodes with a label selector:
kubectl get nodes -l kiamol=ch05

# deploy a PV which uses a local volume on the labelled node:
kubectl apply -f todo-list/persistentVolume.yaml

# check the PersistentVolume:
kubectl get pv
```

You can see my output in figure 5.12—the node labelling is only necessary because I'm not using a distributed storage system. You would normally just specify the NFS or

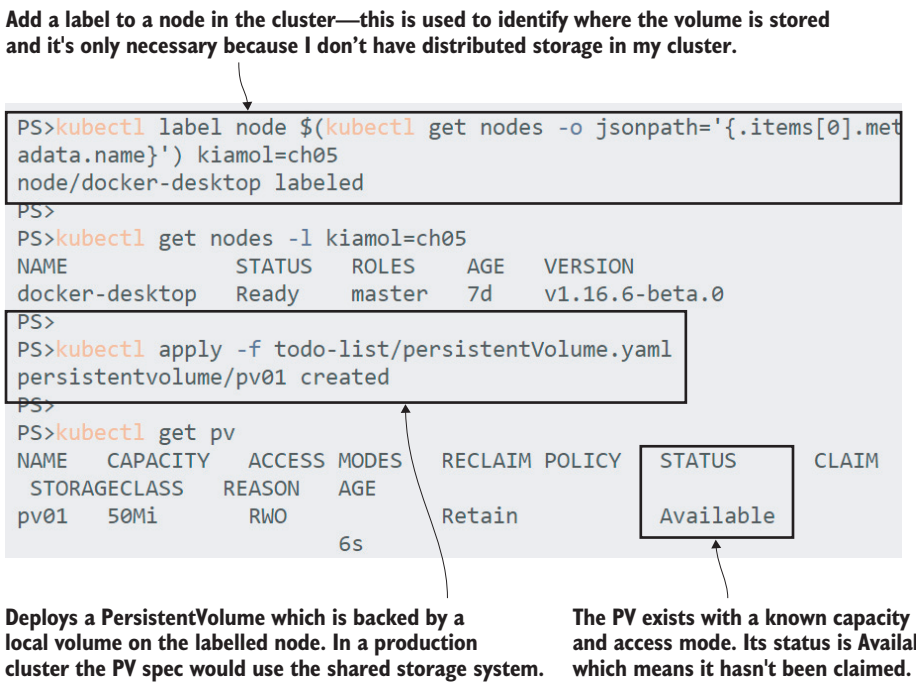

Figure 5.12 If you don't have distributed storage, you can cheat by pinning a PV to a local volume.

Azure Disk volume configuration, which is accessible from any node. But a local volume only exists on one node, and the PV identifies that node using the label.

Now the PersistentVolume exists in the cluster as an available storage unit, with a known set of features, including the size and access mode. Pods can't use that PV directly; instead, they need to claim it using a PersistentVolumeClaim (PVC). The PVC is the storage abstraction which Pods use, and it only requests some storage for an application. The PVC gets matched to a PV by Kubernetes, and it leaves the underlying volume details to the PV. Listing 5.6 shows a claim for some storage that will be matched to the PV we created.

Listing 5.6 postgres-persistentVolumeClaim.yaml, a PVC matching the PV

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pvc          # the claim will be used by a specific app
spec:
  accessModes:                # the required access mode
   -ReadWriteOnce
  resources:
    requests:
      storage: 40Mi           # and the amount of storage requested
  storageClassName: ""        # blank class means a PV needs to exist
```

The PVC spec includes an access mode, storage amount and storage class. If there is no storage class specified, Kubernetes tries to find an existing PV that matches the requirements in the claim. If there is a match then the PVC is bound to the PV, there is a one-to-one link, so once a PV is claimed it is not available for any other PVCs to use.

TRY IT NOW Deploy the PVC from listing 5.6—its requirements are met by the PV we created in the last exercise, so the claim will be bound to that volume.

```
# create a PVC which will bind to the PV:
kubectl apply -f todo-list/postgres-persistentVolumeClaim.yaml

# check PVCs:
kubectl get pvc

# check PVs:
kubectl get pv
```

My output is in figure 5.13, where you can see the one-to-one binding—the PVC is bound to the volume, and the PV is bound by the claim.

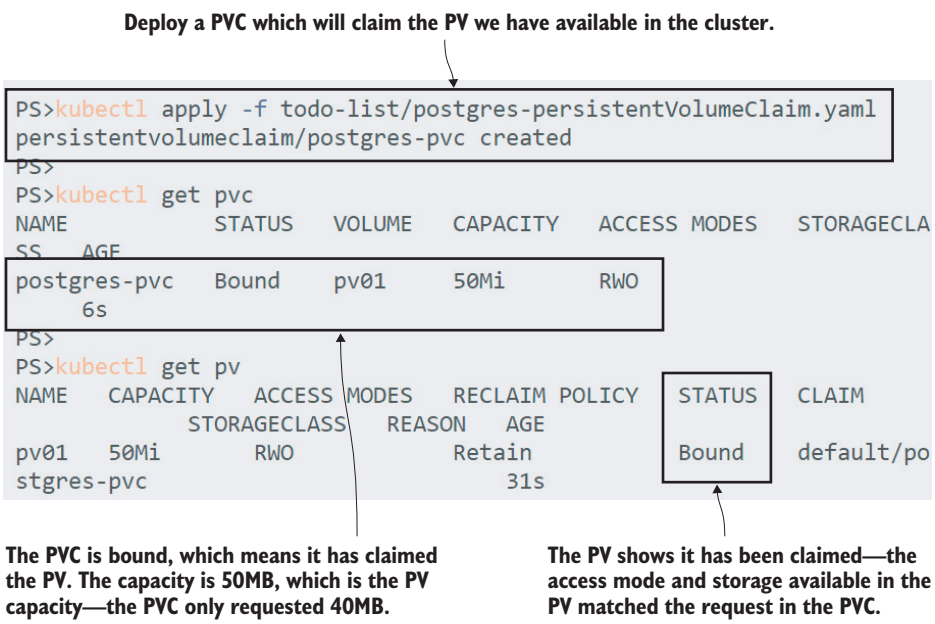

Figure 5.13 **PVs are just units of storage in the cluster—you claim them for your app with a PVC.**

This is a static provisioning approach, where the PV needs to be explicitly created so Kubernetes can bind to it. If there is no matching PersistentVolume when you create a PVC, the claim still gets created but it's not usable—it will stay in the system waiting for a PV to be created that meets its requirements.

TRY IT NOW The PV in your cluster is already bound to a claim, so it can't be used again. Create another PVC and it will remain unbound.

```
# create a PVC which doesn't match any available PVs:
kubectl apply -f todo-list/postgres-persistentVolumeClaim-too-big.yaml

# check claims:
kubectl get pvc
```

You can see in figure 5.14 that the new PVC is in the Pending status. It will remain that way until a PV appears in the cluster with at least 100MB capacity, which is the storage request in this claim.

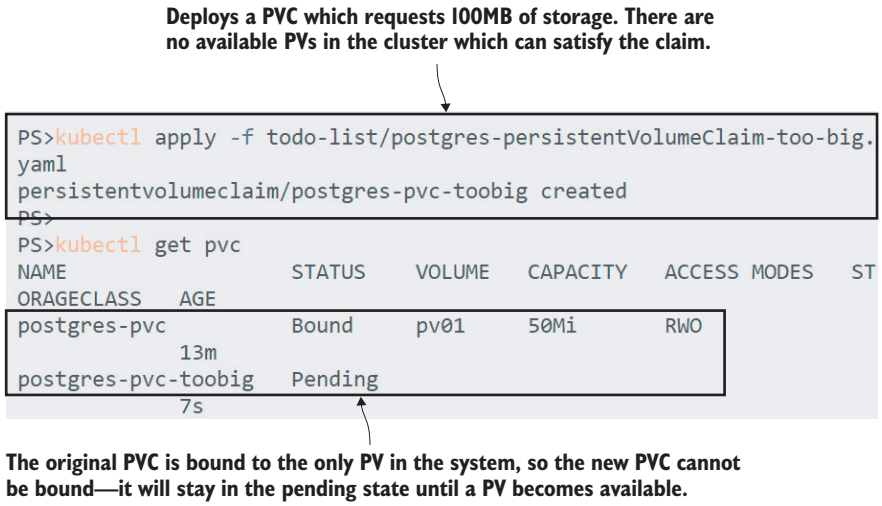

Figure 5.14 With static provisioning a PVC will be unusable until there is a PV it can bind to.

A PVC needs to be bound before it can be used by a Pod. If you deploy a Pod that references an unbound PVC, the Pod will stay in the Pending state until the PVC gets bound, and so your app will never run until it has the storage it needs. The first PVC we created has been bound, so it can be used—but only by one Pod. The access mode of the claim is ReadWriteOnce which means the volume is writeable but it can only be mounted by one Pod. Listing 5.7 shows an abbreviated Pod spec for a Postgres database, using the PVC for storage.

Listing 5.7 todo-db.yaml, a Pod spec consuming a PVC

```
spec:
  containers:
   –name: db
      image: postgres:11.6-alpine
      volumeMounts:
       –name: data
          mountPath: /var/lib/postgresql/data
  volumes:
   –name: data
      persistentVolumeClaim:              # volume uses a PVC
        claimName: postgres-pvc           # PVC to use
```

Now we have all the pieces in place to deploy a Postgres database Pod using a volume which may or may not be backed by distributed storage. The application designer owns the Pod spec and the PVC and isn't concerned about the PV—that's dependent on the infrastructure of the Kubernetes cluster and could be managed by a different team. In our lab environment we own it all, and there's one more step we need to do, which is to create the directory path on the node which the volume expects to use.

> **TRY IT NOW** You probably won't have access to log onto the nodes in a real Kubernetes cluster, so we'll cheat here by running the sleep Pod which has a HostPath mount to the node's root, and create the directory using the mount.

```
# run the sleep Pod which has access to the node's disk:
kubectl apply -f sleep/sleep-with-hostPath.yaml

# wait for the Pod to be ready:
kubectl wait --for=condition=Ready pod -l app=sleep

# create the directory path on the node which the PV expects:
kubectl exec deploy/sleep -- mkdir -p /node-root/volumes/pv01
```

You can see my output in figure 5.15, where the sleep Pod is running with root permissions, so it can create the directory on the node, even though I don't have access to the node directly.

```
PS>kubectl apply -f sleep/sleep-with-hostPath.yaml
deployment.apps/sleep configured
PS>
PS>kubectl exec deploy/sleep -- mkdir -p /node-root/volumes/pv01
```

This is the path on the host which the PV expects to use—we can create it using a Pod that has access to the node's filesystem.

Figure 5.15 **In this example, the HostPath is an alternative way to access the PV source on the node.**

Everything is in place now to run the to-do list app with persistent storage. Normally you won't need to go through as many steps as this, because you'll know the capabilities your cluster provides. But I don't know what your cluster can do, so these exercises work on any cluster, and they've been a useful introduction to all the storage resources. Figure 5.16 shows what we've deployed so far, along with the database we're about to deploy.

Let's run the database. When the Postgres container is created, it mounts the volume in the Pod which is backed by the PVC. This is a new database container connecting to an empty volume, so when it starts up it will initialize the database, creating the Write-Ahead Log (WAL), which is the main data file. The Postgres Pod doesn't know it but the PVC is backed by a local volume on the node, where we also have a sleep Pod running that we can use to look at the Postgres files.

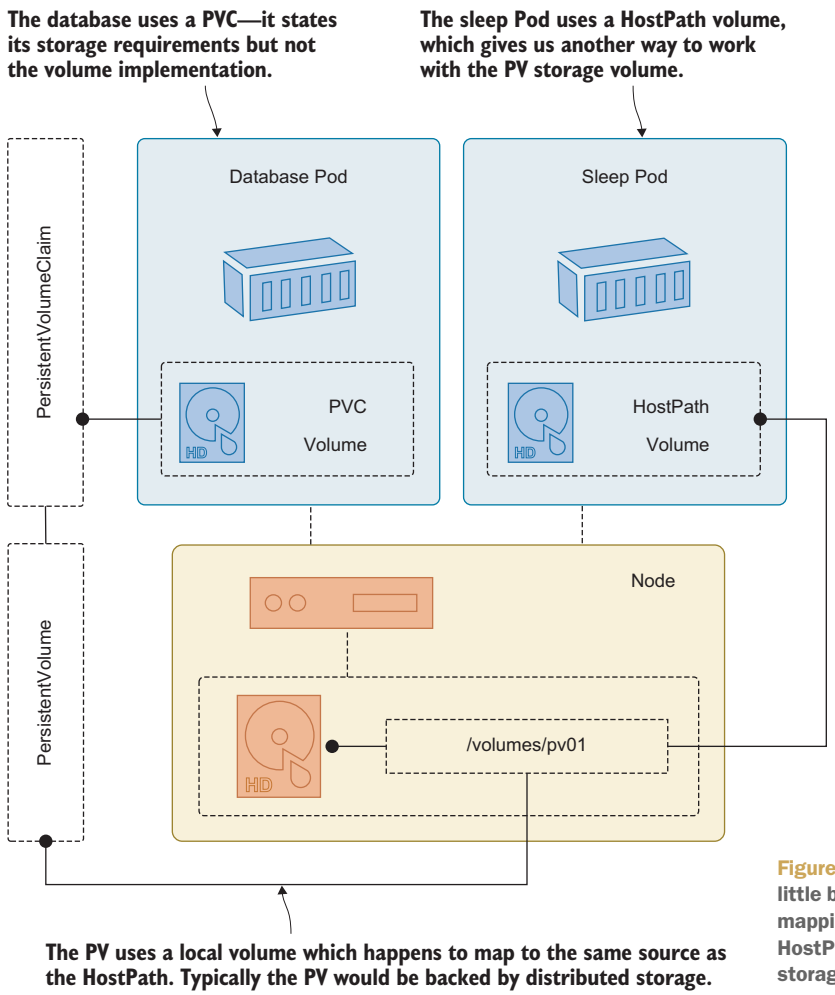

Figure 5.16 Just a little bit complicated—mapping a PV and a HostPath to the same storage location.

TRY IT NOW Deploy the database and give it time to initialize the data files, then check what's been written in the volume using the sleep Pod.

```
# deploy the database:
kubectl apply -f todo-list/postgres/

# wait for Postgre to initialize:
sleep 30

# check the database logs:
kubectl logs -l app=todo-db --tail 1

# and check the data files in the volume:
kubectl exec deploy/sleep -- sh -c 'ls -l /node-root/volumes/pv01 | grep wal'
```

My output in figure 5.17 shows the database server starting correctly and waiting for connections, having written all its data files to the volume.

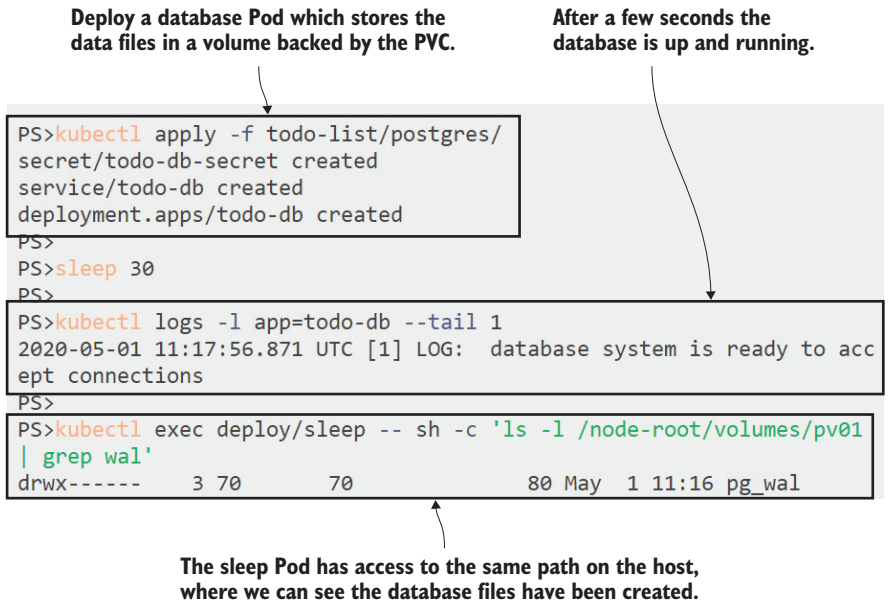

Figure 5.17 The database container writes to the local data path, but that's actually amount for the PVC.

The last thing to do is run the app, test it, and confirm the data still exists if the database Pod gets replaced.

TRY IT NOW Run the web Pod for the to-do app, which connects to the Postgres database.

```
# deploy the web app components:
kubectl apply -f todo-list/web/

# wait for the web Pod:
kubectl wait --for=condition=Ready pod -l app=todo-web

# get the app URL from the service:
kubectl get svc todo-web -o
     jsonpath='http://{.status.loadBalancer.ingress[0].*}:8081/new'

# browse to the app and add a new item

# delete the database Pod:
kubectl delete pod -l app=todo-db

# check the contents of the volume on the node:
kubectl exec deploy/sleep -- ls -l /node-root/volumes/pv01/pg_wal

# check your item is still in the to-do list
```

You can see in figure 5.18 that my to-do app is showing some data, and you'll just have to take my word for it that the data was added into the first database Pod and reloaded from the second database Pod.

Figure 5.18 The storage abstractions mean the database gets persistent storage just by mounting a PVC.

We now have a nicely decoupled app, with a web Pod that can be updated and scaled independently of the database, and a database Pod which uses persistent storage outside of the Pod lifecycle. This exercise used a local volume as the backing store for the persistent data, but the only change you'd need to make for a production deployment is to replace the volume spec in the PV with a distributed volume supported by your cluster.

Whether you really should run a relational database in Kubernetes is a question we'll address at the end of the chapter, but before we do that we'll look at the real deal with storage—having the cluster dynamically provision volumes based on an abstracted storage class.

5.4 Dynamic volume provisioning and storage classes

So far, we've used a static provisioning workflow—we explicitly created the PV and then created the PVC, which Kubernetes bound to the PV. That works for all Kubernetes clusters and might be the preferred workflow in organizations where access to storage is strictly controlled. But most Kubernetes platforms support a simpler alternative with dynamic provisioning.

In the dynamic provisioning workflow you just create the PVC and the PV which backs it is created on-demand by the cluster. Clusters can be configured with multiple storage classes which reflect the different volume capabilities on offer, and also with a default storage class. PVCs can specify the name of the storage class they want, or if they want to use the default class then they omit the storage class field in the claim spec—as in listing 5.8.

Listing 5.8 postgres-persistentVolumeClaim-dynamic.yaml, dynamic PVC

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pvc-dynamic
spec:
  accessModes:
   —ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
      # no storageClassName field, so this uses the default class
```

You can deploy this PVC to your cluster without creating a PV—and I can't tell you what will happen, because it depends on the setup of your cluster. If your Kubernetes platform supports dynamic provisioning with a default storage class, then you'll see a PV gets created and bound to the claim, and that PV will use whatever volume type your cluster has set for the default.

TRY IT NOW Deploy a PVC and see if it gets dynamically provisioned.

```
# deploy the PVC from listing 5.8:
kubectl apply -f todo-list/postgres-persistentVolumeClaim-dynamic.yaml

# check claims and volumes:
kubectl get pvc
kubectl get pv

# delete the claim:
kubectl delete pvc postgres-pvc-dynamic

# check volumes again:
kubectl get pv
```

What happens when you run the exercise? Docker Desktop uses a HostPath volume in the default storage class for dynamically provisioned PVs; AKS uses Azure Files; K3s

uses HostPath but with a different configuration from Docker Desktop, which means you won't see the PV because it only gets created when a Pod is created which uses the PVC. My output in figure 5.19 is from Docker Desktop -it shows that the PV gets created and bound to the PVC, and when the PVC gets deleted the PV is removed too.

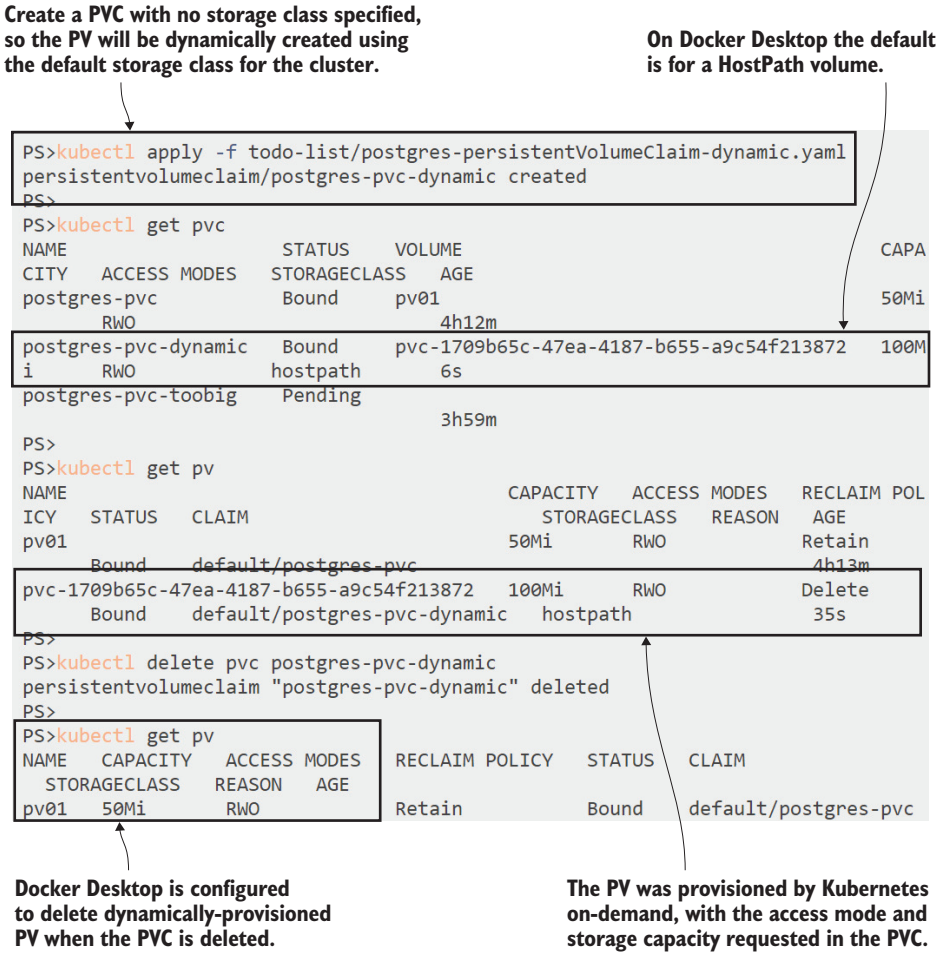

Figure 5.19 Docker Desktop has one set of behavior for the default storage class/ Other platforms differ.

Storage classes have a lot of flexibility. You create them as standard Kubernetes resources, and in the spec, you define exactly how the storage class works with three fields:

- `provisioner`—The component which creates PVs on demand. Different platforms have different provisioners; the provisioner in the default AKS storage class integrates with Azure Files to create new file shares.

- `reclaimPolicy`—What to do with dynamically created volumes when the claim is deleted. The underlying volume can be deleted too, or it can be retained.
- `volumeBindingMode`—Whether the PV gets created as soon as the PVC is created, or not until a Pod is created that uses the PVC.

Combining those properties lets you put together a choice of storage classes in your cluster, so applications can request the properties they need—everything from fast local storage to highly-available clustered storage—without ever specifying the exact details of a volume or volume type. I can't give you a storage class YAML which I can be sure will work on your cluster, because clusters don't all have the same provisioners available, so instead we'll create a new storage class by cloning your default class.

TRY IT NOW There are some nasty details about fetching the default storage class and cloning it, so I've wrapped those steps in a script. You can check the script contents if you're curious, but you may need to have a lie down afterward.

```
# list the storage classes in the cluster:
kubectl get storageclass

# clone the default—on Windows:
Set-ExecutionPolicy Bypass -Scope Process -Force;
     ./cloneDefaultStorageClass.ps1

# OR on Mac/Linux:
chmod +x cloneDefaultStorageClass.sh && ./cloneDefaultStorageClass.sh

# list storage classes:
kubectl get sc
```

The output you see from listing the storage classes shows what your cluster has configured—after running the script you should have a new class called `kiamol` that has the same setup as the default storage class. My output from Docker Desktop is in figure 5.20.

Now you have a custom storage class which your apps can request in a PVC. This is a much more intuitive and flexible way to manage storage, especially in a cloud platform where dynamic provisioning is simple and fast. Listing 5.9 shows a PVC spec requesting the new storage class.

Listing 5.9 postgres-persistentVolumeClaim-storageClass.yaml

```
spec:
  accessModes:
   —ReadWriteOnce
  storageClassName: kiamol      # the storage class is the abstraction
  resources:
    requests:
      storage: 100Mi
```

The storage classes in a production cluster will have more meaningful names, but we all now have a storage class with the same name in our clusters, so we can update the Postgres database to use that explicit class.

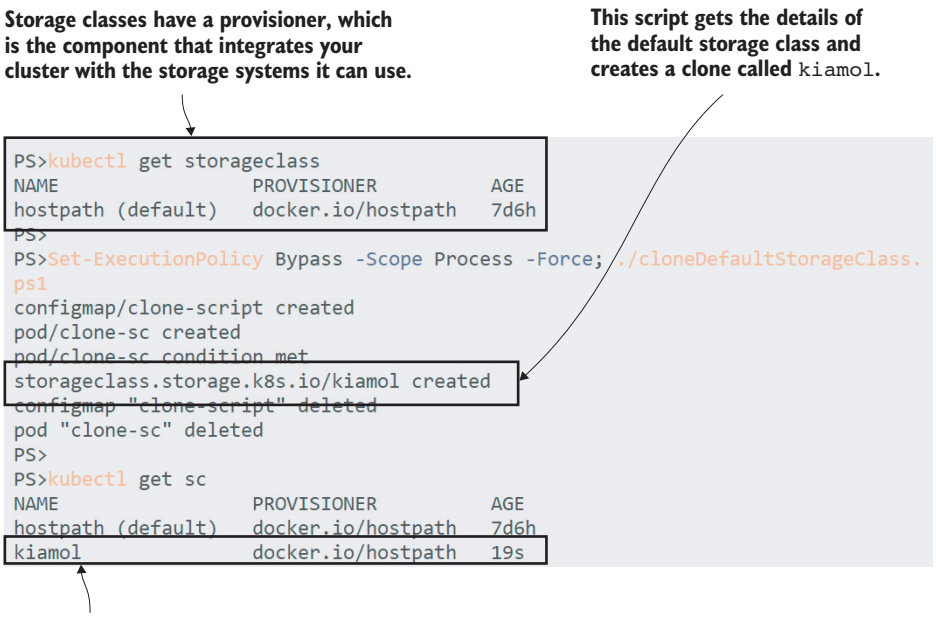

Figure 5.20 **Cloning the default storage class to create a custom class you can use in PVC specs.**

TRY IT NOW Create the new PVC and update the database Pod spec to use it.

```
# create a new PVC using the custom storage class:
kubectl apply -f storageClass/postgres-persistentVolumeClaim-
     storageClass.yaml

# update the database to use the new PVC:
kubectl apply -f storageClass/todo-db.yaml

# check the storage:
kubectl get pvc
kubectl get pv

# and the Pods:
kubectl get pods -l app=todo-db

# refresh the list in your to-do app
```

This exercise switches the database Pod to use the new dynamically provisioned PVC. You can see my output in figure 5.21. The new PVC is backed by a new volume so it will start empty and you'll lose your previous data—but the previous volume still exists, so you could deploy another update to your database Pod, revert it back to the old PVC and see your items.

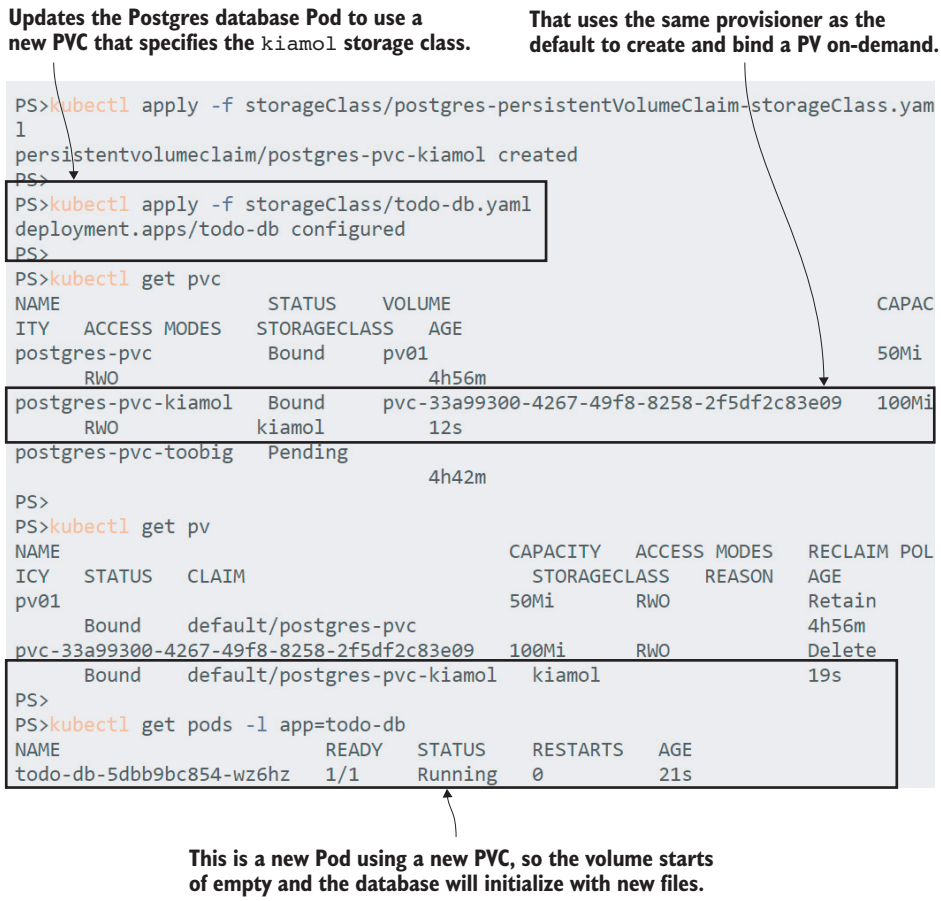

Figure 5.21 **Using storage classes greatly simplifies your app spec. You just name the class in your PVC.**

5.5 Understanding storage choices in Kubernetes

So that's storage in Kubernetes. In your usual work you'll define PersistentVolumeClaims for your Pods and specify the size and storage class you need, which could be a custom value like "FastLocal" or "Replicated". We took a long journey to get there in this chapter, because it's important to understand what actually happens when you claim storage, what other resources are involved, and how you can configure them.

We also covered volume types, and that's an area you'll need to research more to understand what options are available on your Kubernetes platform, and what capabilities they provide. If you're in a cloud environment, then you should have the luxury of multiple cluster-wide storage options—but remember that storage costs, and fast storage costs a lot. You need to understand that you can create a PVC using a fast storage class that could be configured to retain the underlying volume, and that means you'll still be paying for storage when you've deleted your deployment.

Which brings us to the big question—should you even use Kubernetes to run stateful apps like databases? The functionality is all there to give you highly-available, replicated storage (if your platform provides it), but that doesn't mean you should rush to decommission your Oracle estate and replace it with MySQL running in Kubernetes. Managing data adds a lot of complexity to your Kubernetes applications, and running stateful apps is only part of the problem. There's data backups, snapshots and rollbacks to think about—and if you're running in the cloud then a managed database service will probably give you that out of the box. But having your whole stack defined in Kubernetes manifests is pretty tempting, and there are some modern database servers which are designed to run in a container platform—TiDB and CockroachDB are options worth looking at.

All that's left now is to tidy up your lab cluster before we move onto the lab.

TRY IT NOW Delete all the objects from the manifests used in this chapter. You can ignore any errors you get, because not all the objects will exist when you run this.

```
# delete deployments, PVCs, PVs and services:
kubectl delete -f pi/v1 -f sleep/ -f storageClass/ -f todo-list/web -f todo-
     list/postgres -f todo-list/

# delete the custom storage class:
kubectl delete sc kiamol
```

5.6 Lab

These labs are meant to give you some experience in real-world Kubernetes problems, so I'm not going to ask you to replicate that exercise to clone the default storage class . . . Instead we have a new deployment of the to-do app which has a couple of issues. We're using a proxy in front of the web Pod to improve performance and a local database file inside the web Pod because this is just a dev deployment. We need some persistent storage configured at the proxy layer and the web layer, so you can remove Pods and deployments, and the data still persists.

- Start by deploying the app manifests in the ch05/lab/todo-list folder. That creates the services and deployments for the proxy and web components.
- Find the URL for the load balancer and try using the app. You'll find it doesn't respond, and you'll need to dig into the logs to find out what's wrong.
- Your task is to configure persistent storage for the proxy cache files and for the database file in the web Pod. You should be able to find the mount targets from the log entries and the Pod spec.
- When you have the app running, you should be able to add some data, delete all your Pods, refresh the browser, and see your data is still there.
- You can use any volume type or storage class that you like. This is a good opportunity to explore what your platform provides.

My solution is up on GitHub as usual for you to check if you need to:

https://github.com/sixeyed/kiamol/blob/master/ch05/lab/README.md

6 Scaling applications across multiple Pods with controllers

The basic idea for scaling applications is simple: run more Pods. Kubernetes abstracts networking and storage away from the compute layer, so you can run many Pods that are copies of the same app, and just plug them into the same abstractions. Kubernetes calls those Pods replicas, and in a multi-node cluster they'll be distributed across many nodes. That gives you all the benefits of scale—greater capacity to handle load, and high availability in case of failure—all in a platform that can scale up and down in seconds.

Kubernetes provides some alternative scaling options to meet different application requirements, and we'll work through them all in this chapter. The one you'll use most often is the Deployment controller, which is actually the simplest—but we'll spend time on the others, too, so you understand how to scale different types of applications in your cluster.

6.1 How Kubernetes runs apps at scale

The Pod is the unit of compute in Kubernetes, and you learned in chapter 2 that you don't usually run Pods directly; instead, you define another resource to manage them for you. That resource is a controller, and we've used Deployment controllers ever since. A controller spec includes a Pod template which it uses to create and replace Pods, and it can use that same template to create many replicas of a Pod.

Deployments are probably the resource you'll use most in Kubernetes, and you've already had lots of experience of them. Now it's time to dig a bit deeper and learn that Deployments don't actually manage Pods directly—that's done by

another resource called a ReplicaSet. Figure 6.1 shows the relationship between Deployment, ReplicaSet, and Pods.

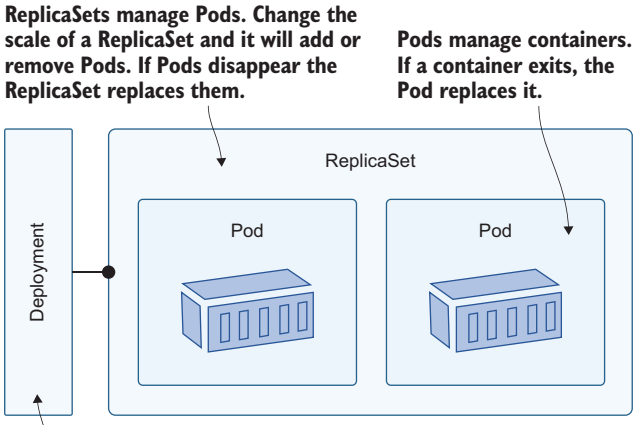

Figure 6.1 Every software problem can be solved by adding another layer of abstraction.

You'll use a Deployment to describe your app in most cases; the Deployment is a controller which manages ReplicaSets, and the ReplicaSet is a controller which manages Pods. You can create a ReplicaSet directly rather than using a Deployment, and we'll do that for the first few exercises just to see how scaling works. The YAML for a ReplicaSet is almost the same as for a deployment—it needs a selector to find the resources it owns, and a Pod template to create resources. Listing 6.1 shows an abbreviated spec.

Listing 6.1 whoami.yaml, a ReplicaSet without a Deployment

```
apiVersion: apps/v1
kind: ReplicaSet          # the spec is almost identical to a Deployment
metadata:
  name: whoami-web
spec:
  replicas: 1
  selector:               # selector for the ReplicaSet to find its Pods
    matchLabels:
      app: whoami-web
  template:               # the usual Pod spec follows
```

The only things that are different in this spec from the Deployment definitions we've used are the object type ReplicaSet and the `replicas` field, which states how many Pods to run. This spec uses a single replica, which means Kubernetes will run a single Pod.

TRY IT NOW Deploy the ReplicaSet, along with a LoadBalancer service that uses the same label selector as the ReplicaSet to send traffic to the Pods.

```
# switch to this chapter's exercises:
cd ch06
```

```
# deploy the ReplicaSet and service:
kubectl apply -f whoami/

# check the resource:
kubectl get replicaset whoami-web

# make an HTTP GET call to the service:
curl $(kubectl get svc whoami-web -o
     jsonpath='http://{.status.loadBalancer.ingress[0].*}:8088')

# delete all the Pods:
kubectl delete pods -l app=whoami-web

# repeat the HTTP call:
curl $(kubectl get svc whoami-web -o
     jsonpath='http://{.status.loadBalancer.ingress[0].*}:8088')

# show the detail about the ReplicaSet:
kubectl describe rs whoami-web
```

You can see my output in figure 6.2. There's nothing new here; the ReplicaSet owns a single Pod, and when you delete that Pod, the ReplicaSet replaces it. I've removed the `kubectl describe` output in the last command, but if you run that you'll see it

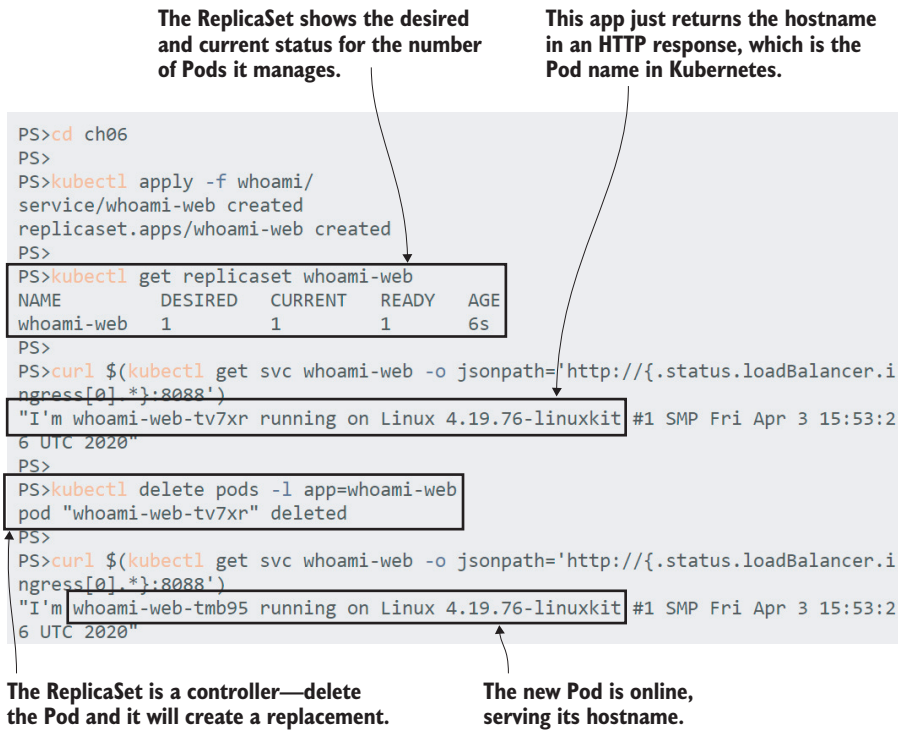

Figure 6.2 Working with a ReplicaSet is just like working with a Deployment—it creates and manages Pods.

ends with a list of events, where the ReplicaSet writes activity logs on how it created Pods.

The ReplicaSet replaces deleted Pods because it constantly runs a control loop, checking that the number of objects it owns matches the number of replicas it should have. You use the same mechanism when you scale up your application—you update the ReplicaSet spec to set a new number of replicas, and then the control loop sees that it needs more and creates them from the same Pod template.

TRY IT NOW Scale up the application by deploying an updated ReplicaSet definition that specifies three replicas.

```
# deploy the update:
kubectl apply -f whoami/update/whoami-replicas-3.yaml

#check Pods:
kubectl get pods -l app=whoami-web

# delete all the Pods:
kubectl delete pods -l app=whoami-web

# check again:
kubectl get pods -l app=whoami-web

# repeat this HTTP call a few times:
curl $(kubectl get svc whoami-web -o
     jsonpath='http://{.status.loadBalancer.ingress[0].*}:8088')
```

My output is in figure 6.3, and it raises a couple of questions—how does Kubernetes manage to scale the app so quickly, and how do the HTTP responses come from different Pods?

The first one is simple to answer—this is a single-node cluster, so every Pod will run on the same node and that node has already pulled the Docker image for the app. When you scale up in a production cluster, it's likely that new Pods will be scheduled to run on nodes that don't have the image locally, and they'll need to pull the image before they can run the Pod. So the speed at which you can scale is bounded by the speed at which your images can be pulled—which is why you need to invest time in optimizing your images.

As to how we can make an HTTP request to the same Kubernetes Service and get responses from different Pods, that's all down to the loose-coupling between Services and Pods. When you scaled up the ReplicaSet there were suddenly multiple Pods which matched the Service's label selector, and when that happens Kubernetes load-balances requests across the Pods. Figure 6.4 shows how the same label selector maintains the relationship between ReplicaSet and Pods, and between Service and Pods.

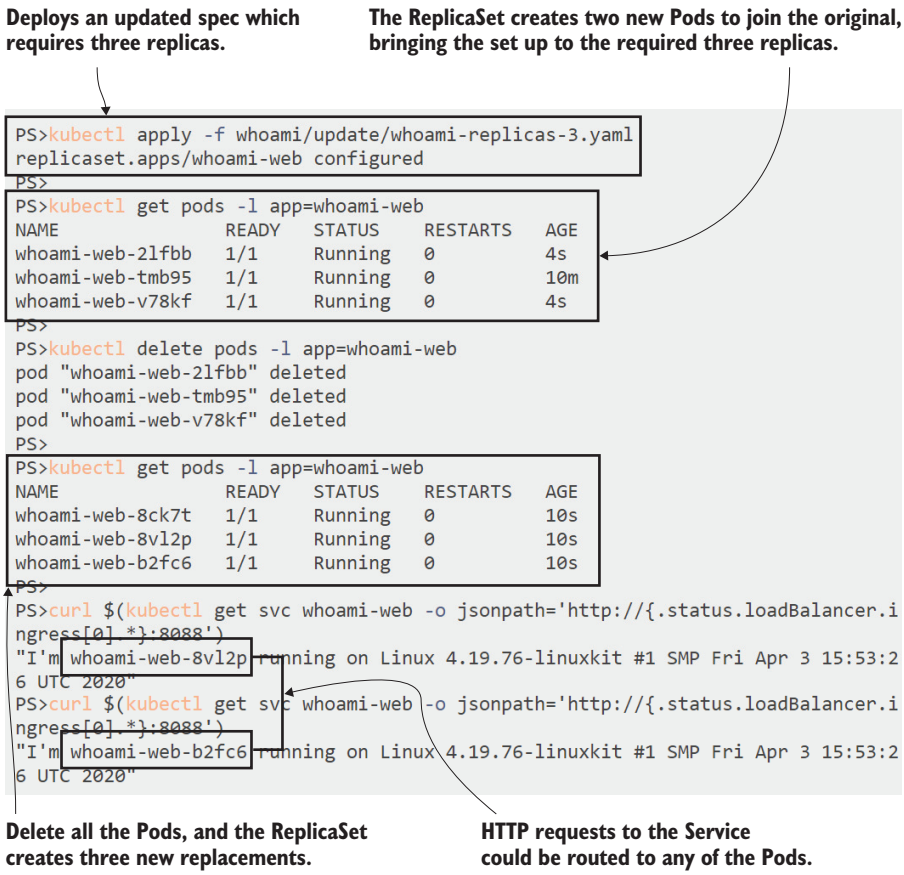

Figure 6.3 Scaling ReplicaSets is fast, and at scale a Service can distribute requests to many Pods.

The abstraction between networking and compute is what makes scaling so easy in Kubernetes. You may be experiencing a warm glow about now—suddenly all the complexity starts to fit into place, and you see how the separation between resources is the enabler for some very powerful features. This is the core of scaling—you run as many Pods as you need, and they all sit behind one Service. When consumers access the Service, Kubernetes distributes the load between Pods.

Load-balancing is a feature of all the Service types in Kubernetes. We've deployed a LoadBalancer Service in these exercises, and that receives traffic into the cluster and sends it to the Pods. It also creates a ClusterIP for other Pods to use, and when Pods communicate within the cluster, they also benefit from load-balancing.

TRY IT NOW Deploy a new Pod and use it to call the who-am-I service internally, using the ClusterIP which Kubernetes resolves from the service name.

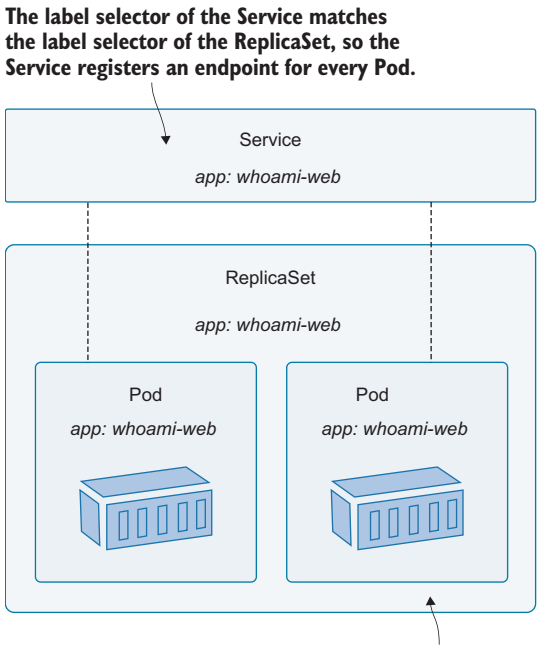

Figure 6.4 Service with the same label selector as a ReplicaSet will use all its Pods.

```
# run a sleep Pod:
kubectl apply -f sleep.yaml

# check the details of the who-am-I service:
kubectl get svc whoami-web

# run a DNS lookup for the service in the sleep Pod:
kubectl exec deploy/sleep -- sh -c `nslookup whoami-web | grep "^[^*]"`

# and make some HTTP calls:
kubectl exec deploy/sleep -- sh -c 'for i in 1 2 3; do curl -w \\n -s
     http://whoami-web:8088; done;'
```

You can see my output in figure 6.5—the behavior for a Pod consuming an internal Service is the same as for external consumers, and requests are load-balanced across the Pods. When you run this exercise you may see the requests distributed exactly equally or you may see some Pods responding more than once, depending on the vagaries of the network.

In chapter 3 we covered Services and how the ClusterIP address is an abstraction from the Pod's IP address, so when a Pod is replaced the application is still accessible using the same Service address. Now you see that the Service can be an abstraction

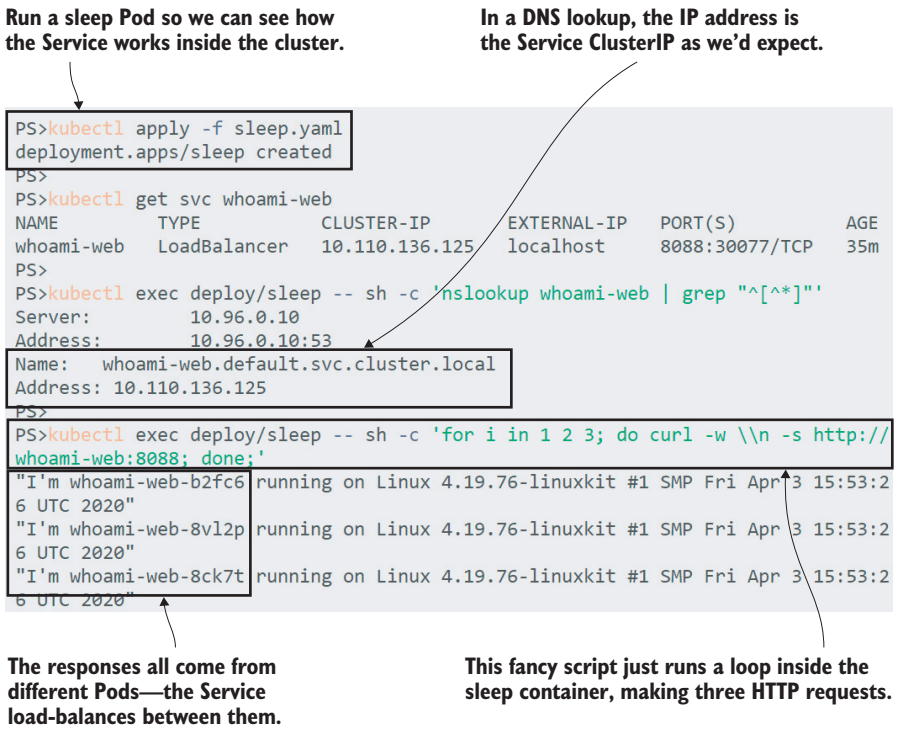

Figure 6.5 The world inside the cluster. Pod to Pod networking also benefits from Service load balancing.

across many Pods, and the same networking layer that routes traffic to a Pod on any node can load balance across multiple Pods.

6.2 Scaling for load with Deployments and ReplicaSets

ReplicaSets make it incredibly easy to scale your app; you can scale up or down in seconds just by changing the number of replicas in the spec. It's perfect for stateless components that run in small, lean containers—and that's why applications built for Kubernetes use a distributed architecture, breaking functionality down across many pieces that can be individually updated and scaled.

Deployments add a useful management layer on top of ReplicaSets, so now that we know how they work, we won't be using ReplicaSets directly anymore—Deployments should be your first choice for defining applications. We won't explore all the features of Deployments until we get to application upgrades and rollbacks in chapter 9, but it's useful to understand exactly what the extra abstraction gives you—figure 6.6 shows that. A Deployment is a controller for ReplicaSets, and to run at scale you include the same replicas field in the Deployment spec, and that gets passed on to the ReplicaSet. Listing 6.2 shows the abbreviated YAML for the Pi web application which explicitly sets two replicas.

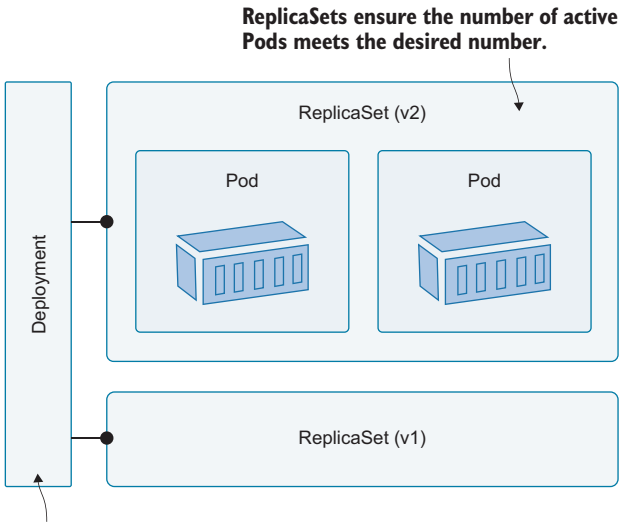

Figure 6.6 Zero is a valid number of desired replicas. Deployments scale down old ReplicaSets to zero.

Listing 6.2 web.yaml, a Deployment to run multiple replicas

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pi-web
spec:
  replicas: 2              # the replicas field is optional, it defaults to 1
selector:
  matchLabels:
    app: pi-web
  template:              # Pod spec follows
```

The label selector for the Deployment needs to match the labels defined in the Pod template, and those labels are used to express the chain of ownership from Pod to ReplicaSet to Deployment. When you scale a Deployment, it updates the existing ReplicaSet to set the new number of replicas, but if you change the Pod spec in the Deployment, it replaces the ReplicaSet and scales the previous one down to zero. That gives the Deployment a lot of control over how it manages the update and how it deals with any problems.

TRY IT NOW Create a Deployment and Service for the Pi web application and make some updates to see how the ReplicaSets are managed.

```
# deploy the Pi app:
kubectl apply -f pi/web/

# check the ReplicaSet:
kubectl get rs -l app=pi-web

# scale up to more replicas:
kubectl apply -f pi/web/update/web-replicas-3.yaml
```

```
# check the RS:
kubectl get rs -l app=pi-web

# deploy a changed Pod spec with enhanced logging:
kubectl apply -f pi/web/update/web-logging-level.yaml

# and check ReplicaSets again:
kubectl get rs -l app=pi-web
```

This exercise shows you that the ReplicaSet is still the scale mechanism—when you increase or decrease the number of replicas in your Deployment then it just updates the ReplicaSet. But the Deployment is the, well, deployment mechanism, and it manages application updates through multiple ReplicaSets. You can see my output in figure 6.7, which shows how the Deployment waits for the new ReplicaSet to be fully operational before completely scaling down the old one.

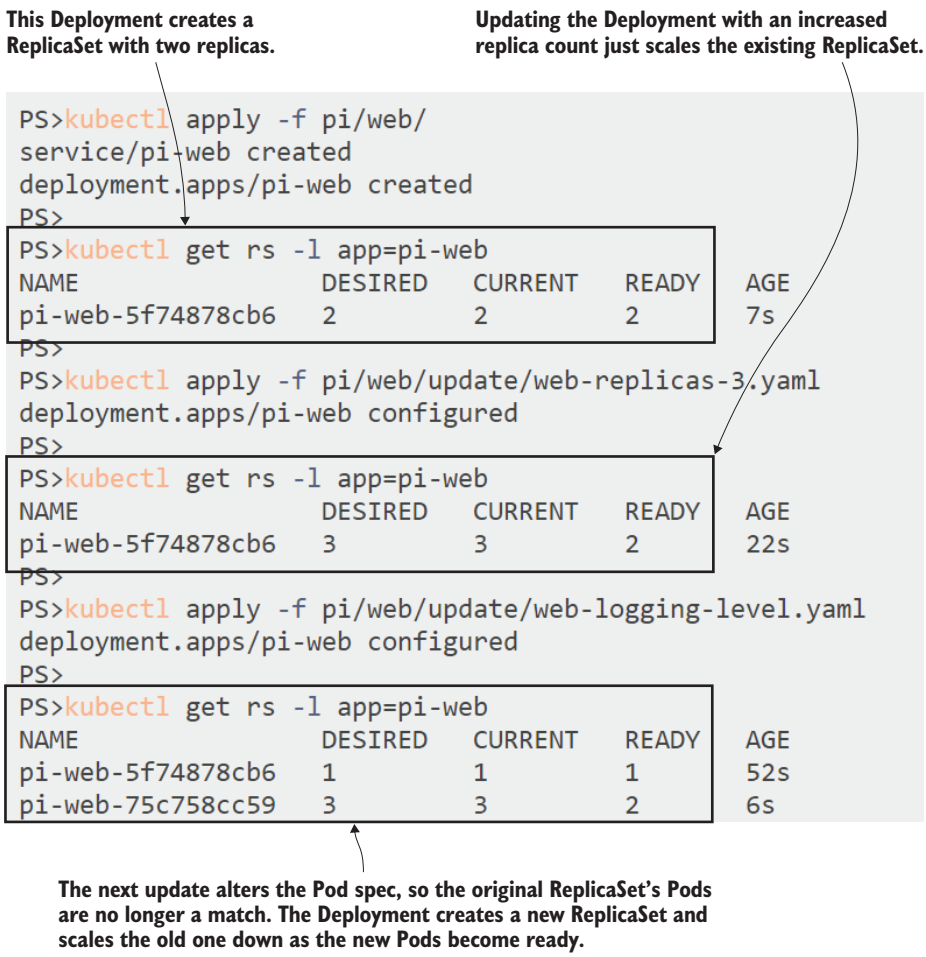

Figure 6.7 Deployments manage ReplicaSets to keep the desired number of Pods available during updates.

There is a shortcut for scaling controllers with the Kubectl `scale` command. You should use it sparingly because it's an imperative way to work, and it's much better to use declarative YAML files, so the state of your apps in production always exactly matches the spec stored in source control. But if your app is underperforming and the automated deployment takes 90 seconds, it's a quick way to scale—as long as you remember to update the YAML file too.

TRY IT NOW Scale up the Pi application using Kubectl directly, and then see what happens with the ReplicaSets when another full deployment happens.

```
# we need to scale the Pi app fast:
kubectl scale --replicas=4 deploy/pi-web

# check which ReplicaSet makes the change:
kubectl get rs -l app=pi-web

# now we can revert back to the original logging level:
kubectl apply -f pi/web/update/web-replicas-3.yaml

# but that will undo the scale we set manually:
kubectl get rs -l app=pi-web

# check the Pods:
kubectl get pods -l app=pi-web
```

You'll see two things when you apply the updated YAML: the app scales back down to three replicas, and the Deployment does that by scaling the new ReplicaSet down to zero Pods and scaling the old ReplicaSet back up to three Pods. Figure 6.8 shows that the updated Deployment results in three new Pods being created.

It shouldn't be a surprise that the Deployment update overwrote the manual scale level. The YAML definition is the desired state, and Kubernetes does not attempt to retain any part of the current spec if they differ. It might be more of a surprise that the Deployment reused the old ReplicaSet instead of creating a new one, but that's a more efficient way for Kubernetes to work, and its possible because of more labels.

Pods created from Deployments have a name generated that looks random, but actually isn't. The Pod name contains a hash of the template in the Pod spec for the deployment, so if you make a change to the spec which matches a previous deployment, then it will have the same template hash as a scaled-down ReplicaSet and the Deployment can find that ReplicaSet and scale it up again to effect the change. The Pod template hash is stored in a label.

TRY IT NOW Check out the labels for the Pi Pods and ReplicaSets to see the template hash.

```
# list ReplicaSets with labels:
kubectl get rs -l app=pi-web  --show-labels

# list Pods with labels:
kubectl get po -l app=pi-web  --show-labels
```

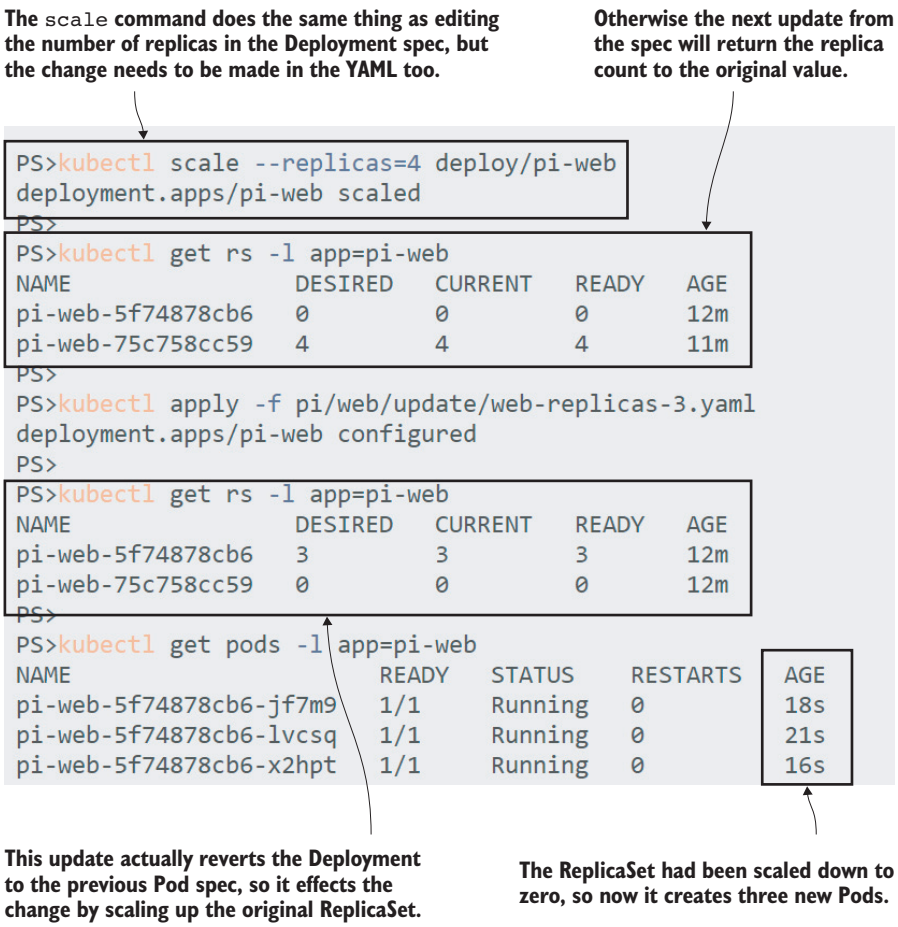

Figure 6.8 Deployments know the spec for their ReplicaSets and can roll back by scaling an old ReplicaSet.

You can see in figure 6.9 that the template hash is included in the object name, but this is just for convenience—Kubernetes uses the labels for management.

Knowing the internals of how a Deployment is related to its Pods will help you understand how changes are rolled out and clear up any confusion when you see lots of ReplicaSets with desired Pod counts of zero. But the interaction between the compute layer in the Pods and the network layer in the Services works in the same way.

In a typical distributed application, you'll have different scale requirements for each component, and you'll use Services to get multiple layers of load balancing between them. The Pi application we've deployed so far only has a ClusterIP service—it's not a public-facing component. The public component is a proxy (actually, it's a reverse proxy because it handles incoming traffic rather than outgoing traffic), and that uses a LoadBalancer service. We can run both the web component and the proxy at scale, and get load balancing from the client to the proxy Pods, and from the proxy to the application Pods.

The ReplicaSet has the app label specified in the YAML file, and also a hash for the Pod spec template, which is used by the Deployment.

```
PS>kubectl get rs -l app=pi-web  --show-labels
NAME                DESIRED   CURRENT   READY   AGE   LABELS
pi-web-5f74878cb6   3         3         3       32m   app=pi-web,pod-template-ha
sh=5f74878cb6
pi-web-75c758cc59   0         0         0       31m   app=pi-web,pod-template-ha
sh=75c758cc59
PS>
PS>kubectl get po -l app=pi-web  --show-labels
NAME                      READY   STATUS    RESTARTS   AGE   LABELS
pi-web-5f74878cb6-jf7m9   1/1     Running   0          19m   app=pi-web,pod-temp
late-hash=5f74878cb6
pi-web-5f74878cb6-lvcsq   1/1     Running   0          19m   app=pi-web,pod-temp
late-hash=5f74878cb6
pi-web-5f74878cb6-x2hpt   1/1     Running   0          19m   app=pi-web,pod-temp
late-hash=5f74878cb6
```

The Pods have the same template hash label. The hash is also used in the name for the generated objects, the ReplicaSet, and Pods.

Figure 6.9 **Object names generated by Kubernetes aren't just random—they include the template hash**

TRY IT NOW Create the proxy Deployment which runs with two replicas, along with a Service and ConfigMap which sets up the integration with the Pi web app.

```
# deploy the proxy resources:
kubectl apply -f pi/proxy/

# get the URL to the proxied app:
kubectl get svc whoami-web -o
     jsonpath='http://{.status.loadBalancer.ingress[0].*}:8080/?dp=10000'

# browse to the app and try a few different values for 'dp' in the URL
```

If you open up the developer tools in your browser and look at the network requests, you can find the response headers sent by the proxy. These include the hostname of the proxy server—which is actually the Pod name—and the web page itself includes the name of the web application Pod which generated the response. My output in figure 6.10 shows a response which came from the proxy cache.

This configuration is a simple one which makes it easy to scale. The Pod spec for the proxy uses two volumes—a ConfigMap to load the proxy configuration file and an EmptyDir to store the cached responses. ConfigMaps are read-only, so one ConfigMap can be shared by all the proxy Pods. EmptyDir volumes are writeable, but they're unique to the Pod, so each proxy gets its own volume to use for cache files. Figure 6.11 shows the setup.

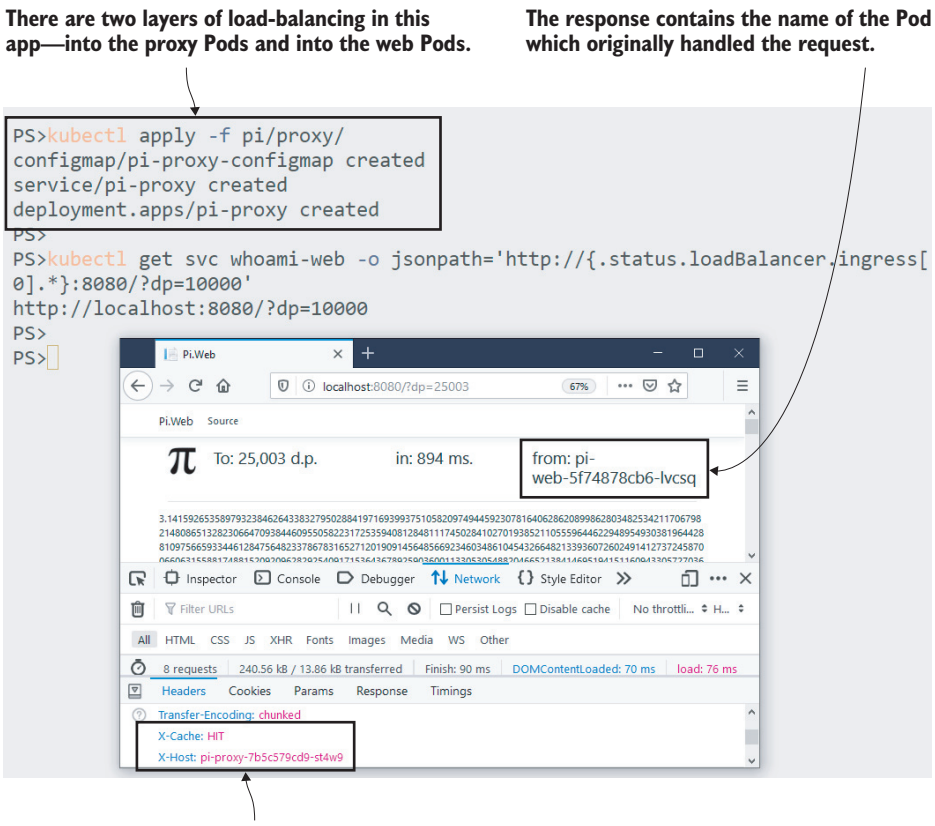

Figure 6.10 **The Pi responses include the name of the Pod which sent them, so you can see the load-balancing at work.**

There's a problem with this architecture, which you'll see if you request Pi to a high number of decimal places and keep refreshing the browser. The first request will be slow because it gets computed by the web app; subsequent responses will be fast because they come from the proxy cache, but soon your request will go to a different proxy Pod that doesn't have that response in its cache, so the page will be slow again.

It would be nice to fix this by using shared storage, so every proxy Pod had access to the same cache. That's going to bring us back to the tricky area of distributed storage that we thought we'd left behind in chapter 5, but let's start with a simple approach and see where it gets us.

TRY IT NOW Deploy an update to the proxy spec that uses a HostPath volume for cache files instead of an EmptyDir. Multiple Pods on the same node will use the same volume, which means they'll have a shared proxy cache.

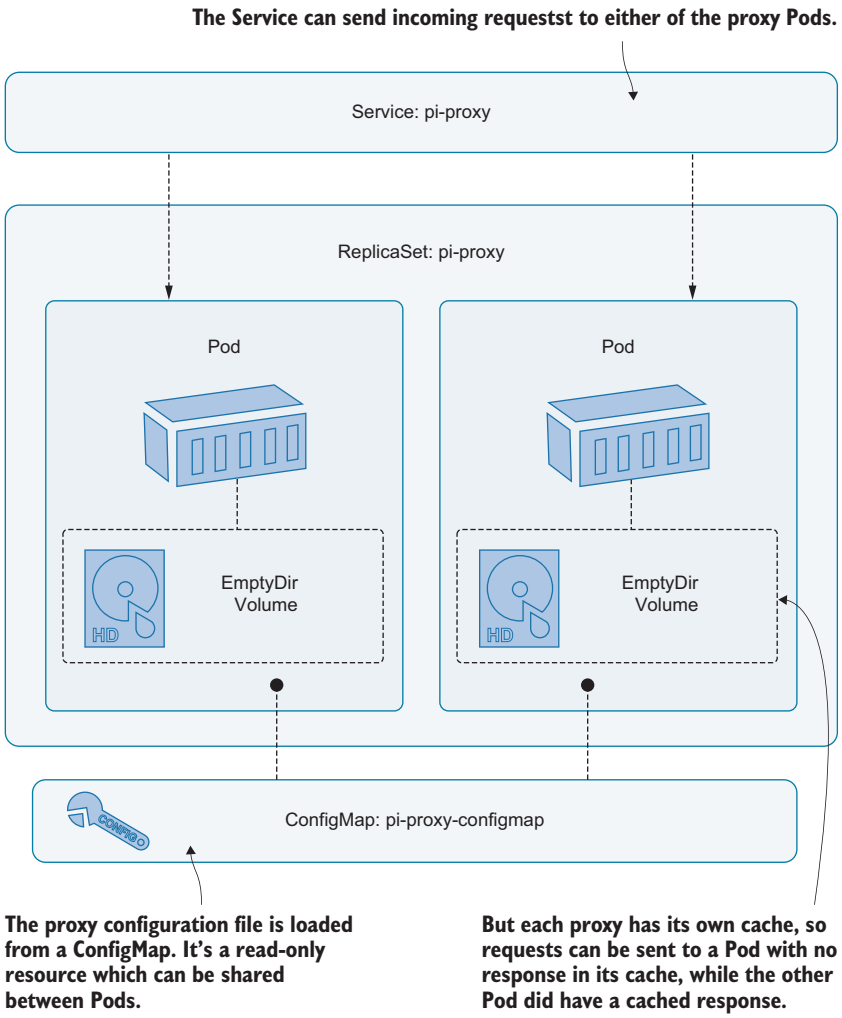

Figure 6.11 **Running Pods at scale. Some types of volume can be shared; others are unique to the Pod.**

```
# deploy the updated spec:
kubectl apply -f pi/proxy/update/nginx-hostPath.yaml

# check the Pods—the new spec adds a third replica:
kubectl get po -l app=pi-proxy

# browse back to the Pi app and refresh a few times

# check the proxy logs:
kubectl logs -l app=pi-proxy --tail 1
```

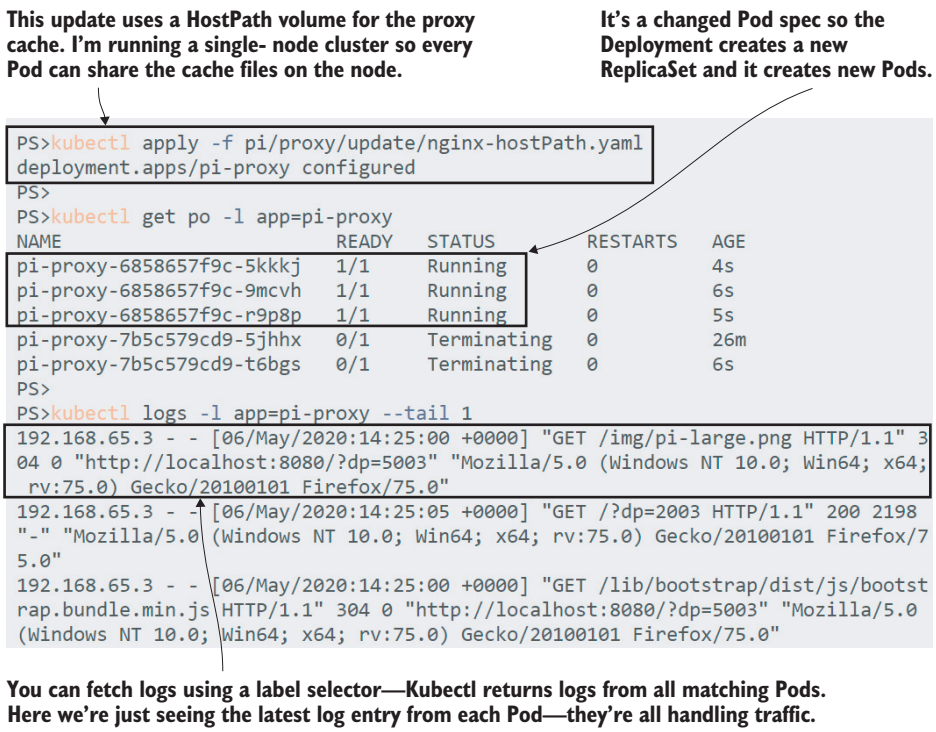

Figure 6.12 At scale, you can see all the Pod logs with Kubectl using a label selector.

Now you should be able to refresh away to your heart's content, and responses will always come from the cache, no matter which proxy Pod you get directed to. Figure 6.12 shows all my proxy Pods responding to requests, which are shared between them by the Service.

For most stateful applications, this wouldn't work. Apps that write data tend to assume they have exclusive access to the files, and if another instance of the same app tries to use the same file location, you'd get unexpected but disappointing results—like the app crashing or the data being corrupted. The reverse proxy I'm using is called Nginx. It's unusually lenient here, and it will happily share its cache directory with other instances of itself.

If your apps need scale and storage, you have a couple of other options using different types of controller. In the rest of this chapter we'll look at the DaemonSet; the final type is the StatefulSet which gets complicated quickly, and we'll come to it in chapter 8 where it gets most of the chapter to itself. DaemonSets and StatefulSets are both Pod controllers, and although you'll use them a lot less frequently than Deployments, you need to know what you can do with them because they enable some powerful patterns.

6.3 Scaling for high availability with DaemonSets

The DaemonSet takes its name from the Linux daemon, which is usually a system process that runs constantly as a single instance in the background (the equivalent of a Windows Service in the Windows world). In Kubernetes, the DaemonSet runs a single replica of a Pod on every node in the cluster, or on a subset of nodes if you add a selector in the spec.

DaemonSets are very common for infrastructure-level concerns, where you want to grab information from every node and send it on to a central collector. A Pod runs on each node, grabbing just the data for that node. You don't need to worry about any resource conflicts, because there will only be one Pod on the node. We'll use DaemonSets later in this book to collect logs from Pods or to collect metrics about the node's activity.

You can also use them in your own designs when you want high availability without the load requirements for many replicas on each node. A reverse proxy is a good example—a single Nginx Pod can handle many thousands of concurrent connections, so you don't necessarily need a lot of them. But you may want to be sure there's one running on every node, so a local Pod can respond wherever the traffic lands. Listing 6.3 shows the abbreviated YAML for a DaemonSet—it looks much like the other controllers, but without the replica count.

Listing 6.3 nginx-ds.yaml, a DaemonSet for the proxy component

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: pi-proxy
spec:
  selector:
    matchLabels:      # DaemonSets use the same label selector mechanism
   app: pi-proxy      # this finds the Pods which the set owns
template:
  metadata:
    labels:
   app: pi-proxy    # labels applied to the Pods need to match the selector
spec:
# Pod spec follows
```

This spec for the proxy still uses a HostPath volume. That means each Pod will have its own proxy cache, so we don't get ultimate performance from a shared cache, but this approach would work for stateful apps which are fussier than Nginx, because there's no issue with multiple instances using the same data files.

> **TRY IT NOW** You can't convert from one type of controller to another, but we can make the change from Deployment to DaemonSet without breaking the app.

```
# deploy the DaemonSet:
kubectl apply -f pi/proxy/daemonset/nginx-ds.yaml
```

```
# check the endpoints used in the proxy service:
kubectl get endpoints pi-proxy

# delete the Deployment:
kubectl delete deploy pi-proxy

# check the DaemonSet:
kubectl get daemonset pi-proxy

# and the Pods:
kubectl get po -l app=pi-proxy

# refresh your latest Pi calculation on the browser
```

Figure 6.13 shows my output. Creating the DaemonSet before removing the Deployment means there are always Pods available to receive requests from the Service—deleting the Deployment first would make the app unavailable until the DaemonSet started. If you check the HTTP response headers, you should also see that your request came from the proxy cache, because the new DaemonSet Pod uses the same HostPath volume as the Deployment Pods.

Creating the DaemonSet while the Deployment still exists—the DaemonSet's Pod gets added to the endpoints for the Service, so all four Pods can receive traffic.

```
PS>kubectl apply -f pi/proxy/daemonset/nginx-ds.yaml
daemonset.apps/pi-proxy created
PS>
PS>kubectl get endpoints pi-proxy
NAME       ENDPOINTS                                              AGE
pi-proxy   10.1.2.46:80,10.1.2.47:80,10.1.2.48:80 + 1 more...   81m
PS>
PS>kubectl delete deploy pi-proxy
deployment.apps "pi-proxy" deleted
PS>
PS>kubectl get daemonset pi-proxy
NAME       DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR
AGE
pi-proxy   1         1         1       1            1           <none>
32s
PS>
PS>kubectl get po -l app=pi-proxy
NAME                        READY   STATUS        RESTARTS   AGE
pi-proxy-6858657f9c-5kkkj   0/1     Terminating   0          55m
pi-proxy-6858657f9c-9mcvh   0/1     Terminating   0          55m
pi-proxy-6858657f9c-r9p8p   0/1     Terminating   0          55m
pi-proxy-rwhwt              1/1     Running       0          38s
```

Deleting the Deployment means its Pods get removed, but there have been Pods available to serve traffic throughout the change to the DaemonSet.

Figure 6.13 You need to plan the order of deployment for a big change to keep your app online.

I'm using a single-node cluster, so my DaemonSet runs a single Pod. With more nodes I'd get one Pod on each node. The control loop watches for nodes joining the cluster, and any new nodes will be scheduled to start a replica Pod as soon as they join. The controller also watches the Pod status, so if a Pod gets removed, then a replacement starts up.

TRY IT NOW Manually delete the proxy Pod, and the DaemonSet will start a replacement.

```
# check the status of the DaemonSet:
kubectl get ds pi-proxy

# delete its Pod:
kubectl delete po -l app=pi-proxy

# and check the Pods:
kubectl get po -l app=pi-proxy
```

If you refresh your browser while the Pod is deleting, you'll see it doesn't respond until the DaemonSet has started a replacement. That's because you're using a single-node lab cluster. Services only send traffic to running Pods, so in a multi-node environment the request would go to a node that still had a healthy Pod. Figure 6.14 shows my output.

```
PS>kubectl get daemonset pi-proxy
NAME       DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR
AGE
pi-proxy   1         1         1       1            1           <none>
17m
PS>
PS>kubectl delete po -l app=pi-proxy
pod "pi-proxy-wq7bt" deleted
PS>
PS>kubectl get po -l app=pi-proxy
NAME             READY   STATUS    RESTARTS   AGE
pi-proxy-tcdrx   1/1     Running   0          4s
```

DaemonSets have different rules to Deployments for creating replicas, but they are still Pod controllers and will replace any Pods which get lost.

Figure 6.14 DaemonSets watch nodes and Pods to ensure the desired replica count is always met.

Situations where you need a DaemonSet are often a bit more nuanced than just wanting to run a Pod on every node. In this proxy example, your production cluster might only have a subset of nodes which can receive traffic from the Internet, so you'd only want to run proxy Pods on those nodes. You can achieve that with labels—adding whatever arbitrary label you like to identify your nodes, and then selecting on that label in the Pod spec. Listing 6.4 shows that with a `nodeSelector` field.

Listing 6.4 nginx-ds-nodeSelector.yaml, a DaemonSet with node selection

```
# this is the Pod spec within the template field of the DaemonSet
spec:
  containers:
    # ...
  volumes:
    # ...
  nodeSelector:       # Pods will only run on certain nodes
    kiamol: ch06      # selected with the label kiamol=ch06
```

The DaemonSet controller doesn't just watch to see nodes joining the cluster, it looks at all nodes to see if they match the requirements in the Pod spec. When you deploy this change, you're telling the DaemonSet to only run on nodes that have the label `kiamol` set to the value of `ch06`, and there will be no matching nodes in your cluster so the DaemonSet will scale down to zero.

TRY IT NOW Update the DaemonSet to include the node selector from listing 6.4. Now there are no nodes which match the requirements, so the existing Pod will be removed. Then label a node and a new Pod will be scheduled.

```
# update the DaemonSet spec:
kubectl apply -f pi/proxy/daemonset/nginx-ds-nodeSelector.yaml

# check the DS:
kubectl get ds pi-proxy

# and the Pods:
kubectl get po -l app=pi-proxy

# now label a node in your cluster, so it matches the selector:
kubectl label node $(kubectl get nodes -o
     jsonpath='{.items[0].metadata.name}') kiamol=ch06 --overwrite

# check Pods again:
kubectl get ds pi-proxy
```

You can see the control loop for the DaemonSet in action in figure 6.15. When the node selector gets applied, there are no nodes that meet the selector, so the desired replica count for the DaemonSet drops to zero. The existing Pod is one too many for the desired count, so it gets removed. Then when the node is labelled, there's a match for the selector and the desired count increases to one, so a new Pod is created.

DaemonSets have a different control loop from ReplicaSets because their logic needs to watch node activity as well as Pod counts, but fundamentally they are both controllers that manage Pods. All controllers are responsible for the lifecycle of their managed objects, but the links can be broken. We'll use the DaemonSet in one more exercise to show how Pods can be set free from their controllers.

TRY IT NOW Kubectl has a `cascade` option on the `delete` command, which you can use to delete a controller without deleting its managed objects. That leaves orphaned Pods behind which can be adopted by another controller if it is a match for their previous owner.

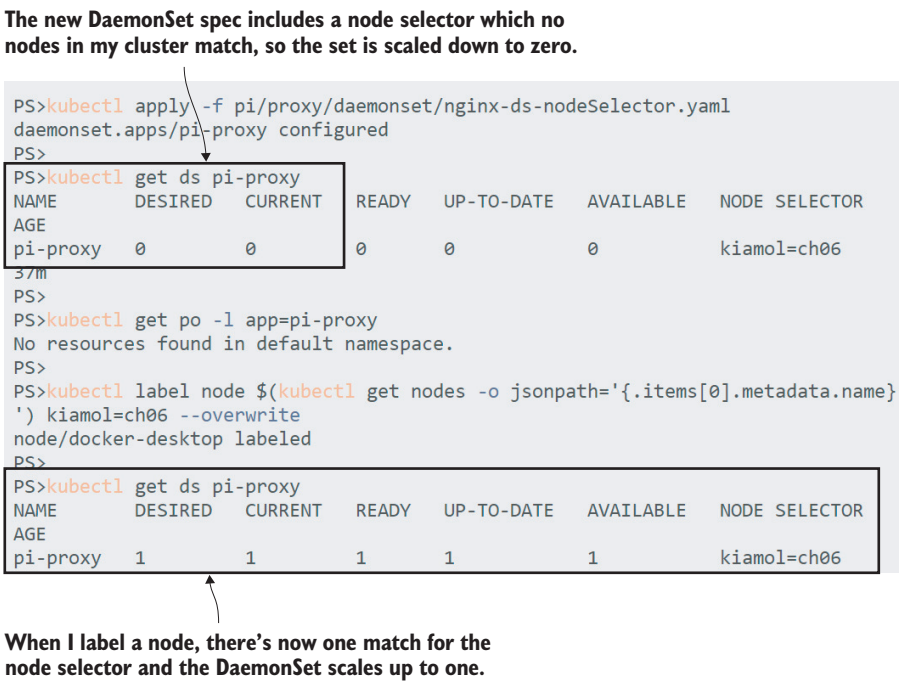

Figure 6.15 DaemonSets actually watch nodes and their labels, as well as current Pod status.

```
# delete the DaemonSet but leave the Pod alone:
kubectl delete ds pi-proxy --cascade=false

# check the Pod:
kubectl get po -l app=pi-proxy

# recreate the DS:
kubectl apply -f pi/proxy/daemonset/nginx-ds-nodeSelector.yaml

# check DS and Pod:
kubectl get ds pi-proxy

kubectl get po -l app=pi-proxy

# delete the DS again, without the cascade option:
kubectl delete ds pi-proxy

# check Pods:
kubectl get po -l app=pi-proxy
```

Figure 6.16 shows my output—the same Pod survives through the DaemonSet being deleted and recreated. The new DaemonSet requires a single Pod, and the existing Pod is a match for its template, so it becomes the manager of the Pod, and when this DaemonSet is deleted, the Pod gets removed too.

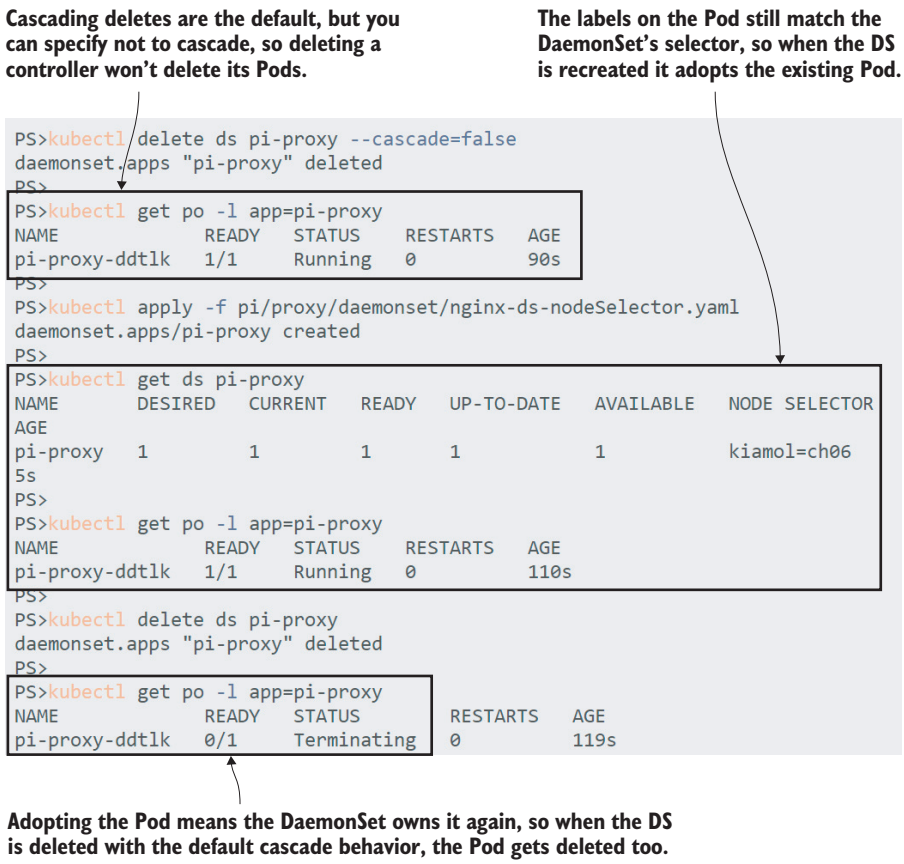

Figure 6.16 Orphaned Pods have lost their controller so they're not part of a highly available set anymore.

Putting a halt on cascading deletes is one of those features you're going to use rarely, but you'll be very glad you knew about it when you do need it. In this scenario, you might be happy with all your existing Pods but have some maintenance tasks coming up on the nodes. Rather than have the DaemonSet adding and removing Pods while you work on the nodes, you could delete it and reinstate it after the maintenance is done.

The example we've used here for DaemonSets is about high availability, but it's limited to certain types of application—where you want multiple instances and it's acceptable for each instance to have its own, independent data store. Other applications where you need high availability might need to keep data synchronized between instances, and for those you can use StatefulSets—but don't skip on to chapter 8 yet, because there are some neat patterns you'll learn in chapter 7 that help with stateful apps too.

StatefulSets, DaemonSets, ReplicaSets and Deployments are the tools you use to model your apps, and they should give you enough flexibility to run pretty much anything in Kubernetes. We'll finish this chapter with a quick look at how Kubernetes

actually manages objects which own other objects, and then we'll review how far we've come in this first section of the book.

6.4 Understanding object ownership in Kubernetes

Controllers use a label selector to find objects which they manage, and the objects themselves keep a record of their owner in a metadata field. When you delete a controller, its managed objects still exist, but not for long. Kubernetes runs a garbage collector process that looks for objects whose owner has been deleted, and it deletes them too. Object ownership can model a hierarchy—so Pods are owned by ReplicaSets, and ReplicaSets are owned by Deployments.

TRY IT NOW Look at the owner reference in the metadata fields for all Pods and ReplicaSets.

```
# check which objects own the Pods:
kubectl get po -o custom-
     columns=NAME:'{.metadata.name}',OWNER:'{.metadata.ownerReferences[0].nam
     e}',OWNER_KIND:'{.metadata.ownerReferences[0].kind}'

# and which objects own the ReplicaSets:
kubectl get rs -o custom-
     columns=NAME:'{.metadata.name}',OWNER:'{.metadata.ownerReferences[0].nam
     e}',OWNER_KIND:'{.metadata.ownerReferences[0].kind}'
```

Figure 6.17 shows my output, where all of my Pods are owned by some other object, and all but one of my ReplicaSets are owned by a deployment.

I haven't created any standalone Pods in this chapter, so all my Pods are owned by a DaemonSet or a ReplicaSet.

```
PS>kubectl get po -o custom-columns=NAME:'{.metadata.name}',OWNER:'{.metadata.ow
nerReferences[0].name}',OWNER_KIND:'{.metadata.ownerReferences[0].kind}'
NAME                    OWNER                OWNER_KIND
pi-proxy-5x2cp          pi-proxy             DaemonSet
pi-web-5f74878cb6-jf7m9 pi-web-5f74878cb6    ReplicaSet
pi-web-5f74878cb6-lvcsq pi-web-5f74878cb6    ReplicaSet
pi-web-5f74878cb6-x2hpt pi-web-5f74878cb6    ReplicaSet
sleep-b8f5f69-5dvjw     sleep-b8f5f69        ReplicaSet
whoami-web-8ck7t        whoami-web           ReplicaSet
whoami-web-8vl2p        whoami-web           ReplicaSet
whoami-web-b2fc6        whoami-web           ReplicaSet
PS>
PS>kubectl get rs -o custom-columns=NAME:'{.metadata.name}',OWNER:'{.metadata.ow
nerReferences[0].name}',OWNER_KIND:'{.metadata.ownerReferences[0].kind}'
NAME                OWNER      OWNER_KIND
pi-web-5f74878cb6   pi-web     Deployment
pi-web-75c758cc59   pi-web     Deployment
sleep-b8f5f69       sleep      Deployment
whoami-web          <none>     <none>
```

I did create one standalone ReplicaSet which has no owner, but all the others are owned by Deployments.

Figure 6.17 Objects know who their owners are, which you can find in the object metadata.

Kubernetes does a good job of managing relationships, but you need to remember that controllers keep track of their dependents using the label selector alone, so if you fiddle with labels then you could break that relationship. The default delete behavior is what you want most of the time, but you can stop cascading deletes using Kubectl and delete only the controller—that removes the owner reference in the metadata for the dependents, so they don't get picked up by the garbage collector.

Okay, we're going to finish up with a look at the architecture for the latest version of the Pi app, which we've deployed in this chapter. Figure 6.18 shows it in all its glory.

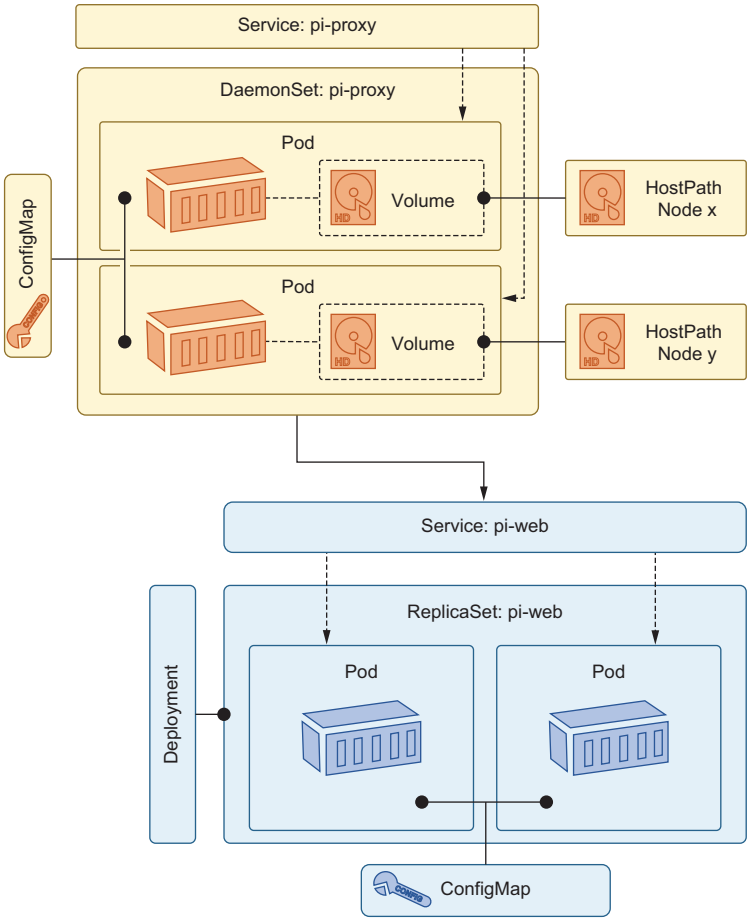

Figure 6.18
The Pi application. No annotations necessary—the diagram should be crystal clear.

There is *quite a lot* going on in that diagram. It's a simple app, but the deployment is complex because it uses lots of Kubernetes features to get high availability, scale, and flexibility. By now you should be comfortable with all those Kubernetes resources, and you should understand how they fit together, and when to use them. Around 150 lines of YAML define the application, but those YAML files are all you need to run this app

on your laptop or on a 50-node cluster in the cloud. And when someone new joins the project, if they have solid Kubernetes experience—or if they've read the first six chapters of this book—they can be productive straight away.

That's all for the first section. My apologies if you had to take a few extended lunchtimes this week, but now you have all the fundamentals of Kubernetes, and with best practices built in. All we need to do is tidy up before you attempt the lab.

> **TRY IT NOW** All the top-level objects in this chapter had a `kiamol` label applied. Now that you understand cascading deletes, you'll know that when you delete all those objects, all their dependents get deleted too.

```
# remove all the controllers and Services:
kubectl delete all -l kiamol=ch06
```

6.5 Lab

Kubernetes has changed a lot over the last few years. The controllers we've used in this chapter are the recommended ones, but there have been alternatives in the past. Your job in this lab is to take an app spec which uses some older approaches and update it to use the controllers you've learned about.

- Start by deploying the app in ch06/lab/numbers. It's the random number app from chapter 3 but with a strange configuration. And it's broken too.
- You need to update the web component to use a controller that supports high load. We'll want to run dozens of these in production.
- The API needs to be updated too. It needs to be replicated for high availability, but the app uses a hardware random number generator attached to the server, which can only be used by one Pod at a time. Nodes with the right hardware have the label `rng=hw` (you'll need to simulate that in your cluster).
- This isn't a clean upgrade, so you need to plan your deployment to make sure there's no downtime for the web app.

Sounds scary, but you shouldn't find this too bad. My solution is up on GitHub for you to check: https://github.com/sixeyed/kiamol/blob/master/ch06/lab/README.md

index

A

API
  and core concepts of Kubernetes 2
  controllers and 19
  CRI (Container Runtime Interface) 17
application manifests 26
  and higher-level resources 28
  applying deployment manifest 29
  declarative 27
  example of simple 26
  JSON and 26
  YAML and 26
application manifests. *See* YAML
applications
  and different storage requirements 87
  and reading configuration settings from environment 84
  configuring with ConfigMaps and services 61–86
  scale requirements for each component and typical distributed 125
  scaling 115–138
    across multiple pods with controllers, basic idea for 115

C

(CKA) Certified Kubernetes Administrator 6
(CKAD) Certified Kubernetes Application Developer 6
cascading delete 137–138
cloud Kubernetes platforms, load balancers and 48
cluster
  as core concepts of Kubernetes 2
  defined 2
  different platforms and 2
  distributed database 4
  illustration of 2
  options for running single-node 10
ClusterIP services
  and little configuration required by 55
  deleting before deploying another type of service 50
  described 42
  headles services 53
clusters
  and multiple storage classes 109
ConfigMaps
  and complex configuration requirements 63
  and loading data items selectively 75
  as filesystem source 71
  as read-only storage units 89
  as storage units for small data amounts 61
  attached to zero or more pods 64
  creating ConfigMap objects 61
  deploying update used by live pod 73
  example of creating and using 64–66
  loading as directories 73
  loading data into container filesystem 70
  multiple data items 67
  referencing in pod specification 64
  sensitive data and 77
  storing and using configuration files in 66–71
  storing different types of data 64
  support for wide range of configuration systems 77
  surfacing as JSON files 83

surfacing configuration data from 71–77
volume mounts 71–72
volumes 71
container filesystem
and moving apps to Kubernetes 88
as virtual construct 71, 88
ConfigMaps loaded as directories into 71
container replacement 88
initial contents 87
Kubernetes and building 87–92
multiple sources 87
container runtime
and accessing containers outside of Kubernetes 17
application logs 30
clusters and 2
nodes and 17
containers
and elimination of gaps between environments 61
and Kubernetes as platform for running 2
as parts of the same virtual environment 14
communication over virtual networks 3
configuration injection 61
defined 13
environment variables and 62
high availability 3
in pods, filesystem 87
interactive shells and 30
localhost for communication 14
pods and 13. *See also* pods
relationship between pods and 14
running and managing 13–19
single pod and running multiple 13
writeable storage layer 87
controllers
and label selector for resource identification 22
cascade effect and deleting 35
defined 19
deleted, managed objects and 136
deleting resources managed by 33–34
deployments 20
label selector 136
relationship between deployments, pods and containers 20
running pods with 19–25
CrashLoopBackOff status 76
CRI (Container Runtime Interface) 17
and managing containers 17

D

DaemonSets 130–136
creating before removing deployments 130–131
for proxy component, example of 130
high availability 130, 135
node activity and pod status 133
nodeSelector field 132
overview 130
pod replacement 132
data
cluster-wide, storing with persistent volumes and claims 99–108
storing on nodes with volumes and mounts 92–99
data storing
and the simplest storage option 92
delete command 34, 85
deployment controller 39, 115
pod replacement 44
proxy pod and 93
scaling and 20
deployments
adding labels to pods 21
and creating pods 21
and keeping track of Kubernetes resources 21
and managing ReplicaSets 116, 122–123
and pod replacement 20
application manifests and defining 26–29
as abstractions over pods 17
as controllers for ReplicaSets 121
changing pod label 22–25
defining applications and 121
manifest-based, repeatable 28
pod specification 28
reusing old ReplicaSets 124
scaling 20
scaling applications and 121–129
scaling down old ReplicaSets to zero 122
unmanaged pods 23
updating existing ReplicaSets 122
using template for creating pods 22
distributed storage systems 100
and different configuration requirements 100
DNS (Domain Name System), permanent address for resources and 39
Docker Community Edition 10
Docker Desktop
installation of 9–10
load balancers and 47
running Kubernetes locally and 8
settings 9

domain names, communication between pods and 40
dynamic provisioning workflow 109

E

Elastic Block Store 100
EmptyDir volume 90–92
  and data stored in 90
  caching files in 92–93
  lifecycle 92
  pod replacement 93
  shared by containers 91
environment files, ConfigMaps and 66
environment variables
  adding to pod specification 63
  and configuration changes 63
  as basic configuration system 62
  described 62
  Kubernetes and rules of precedence for applying 66
  loading secrets into 80
  overriding 67
environment, clustered, data access and 87
exec command 32
ExternalName services
  and app requests 52
  and differences between environments 51
  and sending requests outside clusters 51
  and using local cluster addresses for remote components 50
  canonical names (CNAMEs) 52
  described 50

G

GlusterFS networked filesystem 100

H

HostPath volume 94–99
  and keeping data between pod replacements 95
  and pods running on the same node 96
  described 94
  issues with 97
  limitations 94
  mounting and complete data access 97–98

J

JSONPath, complex queries and 16

K

K3s
  load balancers and 48
  options for running single-node cluster and 10
Kind, options for running single-node cluster and 10
KubeCon conference 1
Kubectl 11
  and creating deployment resources 20
  and example of running simple pod 14
  and forwarding network traffic 18
  and namespaces 57–58
  and output customization options 16
  and running commands inside pod containers 30–31
  apply command 27
  cascade option 133
  consistent syntax 35
  copying files between pod containers and local machines 33
  deleting Kubernetes resources 33
  manifest files 28
  printing labels 21
  run command 27
Kubernetes
  adding environment variables in pod specifications 62
  adding labels to resources 21
  and data as balance between speed of access and durability 87
  and infrastructure-level concerns 1
    DaemonSets 130
  and organizations moving to cloud 1
  and permanent address for resources 39
  and routing network traffic 37–41
  and storage 4
  app configuration approach 68–69
  application logs 30
  basic building blocks of 13, 37
  building container filesystem 87–92
  certifications 6
  cluster-wide storage 87, 99
  communication between pods running on different nodes 39
  components and high load 3
  ConfigMaps 61–71
  configuration injection 61
  consistency and 4
  CRI mechanism for container management 17
  deployments 13
  differences between implementations 47

different treatment for secrets and ConfigMaps 78
DNS server and 39
example of deployment process 3
Kubectl 11
LoadBalancer services and different IP addresses requested by 47–48
main features of 4
managing app configuration in 84–86
  choosing alternative approach to config management 84
  factors to consider 84
  live updates and limited options 84
  sensitive data and 84–85
namespaces 57–59
naming scheme for deployment-managed pods 21
network proxy 55
nodes 2
object names and template hash 126
object ownership 136–138
  garbage collector process 136
  hierarchy modelling 136
  object metadata 136
official documentation 8
platforms 8
pods 13
reasons for popularity 1
resource management 33–35
routing traffic outside 50–55
  ExternalName services 50–53
  services 50–55
  typical candidates for running outside of Kubernetes 50
running and managing containers 13–19
running applications at scale 115–121
running stateful apps 114
scaling and abstraction between networking and compute 119
scaling options 115
secret data and Base64 encoding 78
secrets 77–83
service resolution 55–59
services 37–59
standard networking protocols and 37
storage choices in 113–114
supplying configuration to apps 61–66
supports for different writeable types of volume 89
understanding 2–5
  as platform for running containers 2
  core concepts 2
  managing applications 3
  network traffic management 4
  YAML files and defining apps in 3–4
Kubernetes in Action 8

L

labels
  and identifying relationship between resources 22
  Kubernetes resources and adding 21
Learn Docker in a Month of Lunches 6
LoadBalancer
  described 45–46
  for external traffic 45–46
load-balancing 119

M

Minikube, options for running single-node cluster and 10

N

NFS (Network File Shares) 100
NodePort
  described 48
  vs. LoadBalancer 48
nodes
  and Container Runtime Interface (CRI) 17
  as individual servers 2
  cluster capacity to expand and 2
  container runtime 2, 15, 17
  distributed storage system 100
  extension of data durability 96
  going offline 19
  HostPath and complete data access on 97–98

P

performance, example of using proxy component to improve 92–93
ping command, finding pod's IP address and 38
pods
  allocation 17
  and changing labels 22–25
  and communicating from outside clusters 18
  and controllers 19–25
  and debugging 23
  and interaction with filesystem 32
  and PVC (PersistentVolumeClaim) 102–108
  as abstractions 15
  as disposable resources 37
  as primitive resources 19
  as replicas 115

as units of compute 13, 19
ClusterIP services and communication between 42
cluster-wide storage 87
controllers and scaling applications across multiple 115–138
created from deployments, names for 124
data access and distributed storage system 100
defined 13
example of running simple 14–15
filesystem's lifecycle 89
fixed domain names and communication between 40
high availability and running several 19
higher-level resources and managing 24
IP adresses and communication between 37
Kubectl and fetching information about 16
life span of 24
loading ConfigMaps into 64
loading individual data items from ConfigMaps 65
loose-coupling between services and 118
multi-container 14
orphaned 133–135
pod replacement 17
deployment controller 44
new IP address 39
pod replacement and failed state of new 75
pod restart and cache data 93
pod to pod networking and benefits from load-balancing 121
port-forwarding 28, 45
routing external traffic into 45–49
LoadBalancer and 46
routing traffic between 42–45
running at scale 128
services and communication between 37
updating app components and pod replacement 45
virtual IP address of 13
working with applications in 29–33
PowerShell 9
provisioner, storage classes and 110
proxy
Nginx 129
response headers 126
PV (PersistentVolume) 101–102
and HostPath, mapping to the same storage location 106
defined 101
PVs as cluster units of storage 103
using local storage, example of 101–102
PVC (PersistentVolumeClaim) 101–108
bound 104
creating another 104
defined 102
matching PV, example of 102
ReadWriteOnce access mode 104

R

reclaimPolicy, storage classes and 111
ReplicaSets
and managing pods 116
as another layer of abstraction 116
as scale mechanism 123
control loop run by 118
label selector and maintaining relationship between pods and 118
relationship between deployments, pods, and 116
replacing deleted pods 118
scaling application, example of 118–119
scaling applications and 121–129
stateless components 121
working with 117
resources
ConfigMaps 61
controllers 19
Kubernetes and basic 5
labels and 22
namespaces 57
pods and higher-level 13
secrets 61
run command 68

S

scale command 124
scaling 115–138
deployments and 121–129
high availability and DaemonSets 130–136
Kubernetes and 115–121
ReplicaSets and 121–129
scaling. *See* deployments
secrets
as read-only storage units 89
as storage units for small data amounts 61
avoiding accidental exposure 79
configuration setting read by files populated from 82
configuring sensitive data with 77–83
creating from literal value 78
creating secret objects 61
loading environment variables into 80

main difference between ConfigMaps and 77
overview 77–78
POSTGRES_PASSWORD environment variable 81
POSTGRES_PASSWORD_FILE environment variable 81
surfacing as files 80
surfacing as JSON files 83
using YAML files for 80
services 37–59
accessibility across namespaces 57
and ICMP protocol 41
and minimal YAML specification for 40
as parts of cluster-wide Kubernetes pod network 52
ClusterIP as default type in Kubernetes 42
ClusterIP as default type in Kubernetes. *See* ClusterIP services
defined 37
deleting one type to deploy another 50–51
deploying
and creating DNS entry 41
and fixing broken links between a web app and an API 44
and routing network traffic 43
endpoint lists 55–56
example of failed communication without 42
headless 53
LoadBalancer 45–49
load-balancing 119
NodePort 48–49
replacing ExternalName service with headless service, example of misconfiguration 53–54
routing traffic outside Kubernetes 50–55
separate removal of 59
support for multiple networking patterns 59
StatefulSets 135
static provisioning workflow 103–104, 109
storage
and applications with different requirements 87
and choices in Kubernetes 113–114
and different distributed storage systems 100
cluster-wide 87
cluster-wide data 99
persistent 99–108
storage classes and flexibility 110
writeable layer 87
storage classes
cloning default 111–112
dynamic volume provisioning and 109–113
stringData field 80

T

TCP, standard networking protocol 37
template hash, pod names and 124

U

UDP, standard networking protocol 37

V

volume mounts 71
volumeBindingMode, storage classes and 111
volumes 71
and surfacing selected items form ConfigMap into mount directory 77
as option for loading configuration 74
ConfigMap directory and overwriting mount path for 74
HostPath volume 94–99
restricting access with sub-paths 99

W

WAL (Write-Ahead Log) 105

Y

YAML
and creating ConfigMaps and secret objects 61
and specification about application architecture 43
deploying service and 41
YAML files
application manifests 5, 26