

Learn KUBERNETES IN A MONTH OF LUNCHES

SUNDAY	TUESDAY	WEDNESDAY	THURSDAY	
1 Before You Begin	2 Running containers in Kubernetes with Deployments and Pods	3 Communicating with containers using Services	4 Configuring applications with ConfigMaps and Secrets	5 Writing persistent data with volumes and claims
8 Scaling containers with ReplicaSets and DaemonSets <small>✓ radical</small>	9 Patterns for multi-container pods	10 Scheduling occasional work with Jobs and CronJobs <small>✓ interesting</small>	11 Managing app deployment with upgrades and rollbacks	12 Packaging and installing apps with Helm <small>✓ liked this</small>
15 Packaging and installing apps with Helm <small>✓</small>	16 App development - developer workflow and CI/CD <small>✓</small>	17 Empowering self-healing apps with Liveness and Readiness checks	18 Centralizing application logs with Fluentd and Elasticsearch	19 Monitoring applications and Kubernetes with Prometheus <small>✓</small>
22 Managing traffic coming into your apps with Ingress	23 Securing resources with role-based access controlx	24 Kubernetes at scale-multi-node clusters and multiple clusters	25 Advanced scheduling patterns and access control	26 Securing resources with role-based <small>✓</small>
29 Serverless functions as containers in your Kubernetes cluster	30 Running a multi-architectural cluster with Windows and Arm nodes	31 Never the end next steps in your Kubernetes journey		20

ELTON STONEMAN



MANNING



MEAP Edition
Manning Early Access Program

Learn Kubernetes in a Month of Lunches

Version 7

Copyright 2020 Manning Publications

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

welcome

Hello there! Thanks for signing up to the MEAP for *Learn Kubernetes in a Month of Lunches*. This is my second book for Manning, and *Learn Docker in a Month of Lunches* was much improved from MEAP feedback, so I look forward to hearing what you have to say on this one. Kubernetes is fantastic technology, but it is complex and can be challenging to learn — you need to go into a lot of detail even for the basics. I've tried to present this book at a level that lets you get to grips with how Kubernetes works and what you can do with it without getting lost in the detail.

My name's Elton and by day I'm an independent consultant and trainer, helping organizations at every stage in their container journey. By night I write books and video courses, or do just-in-time preparation for presentations about my project experiences. I've been fortunate to travel the world, speaking at conferences and running workshops on Docker and Kubernetes, and I'm distilling many years of using and teaching containers into this book.

Learn Kubernetes in a Month of Lunches is aimed at people who are new (or new-ish) to Kubernetes but have some understanding of containers. It takes a practical approach — I don't spend the first X chapters talking about the history of Kubernetes and its relation to Borg, or about the internal components of the cluster. Other books do that. This book gets your Kubernetes environment set up in chapter 1, and from chapter 2 you're running apps in Kubernetes — adding more features with every exercise and layering up your understanding.

This *Month of Lunches* book follows the series format, but if you take the title at face value, you might need to schedule some longer-than-usual lunchtimes. Kubernetes is a huge technology, and my goal is to give you an authentic experience of each topic. I haven't cut any important details, and because of that you may find that the chapters take a little more time to work through — but I hope you'll find it to be a good investment.

The book is split into four sections, which roughly align to four working weeks if you follow the title and work through a chapter each weekday. Week 1 gets you familiar with Kubernetes — running apps, managing network traffic, setting up application configuration and working with data. Week 2 builds on that info with more advanced application architectures and a focus on the app development and management lifecycle. Week 3 shows you how to prepare your apps for production, and week 4 covers the difficult material — securing access, running hybrid workloads and extending Kubernetes.

Kubernetes is fast becoming a must-have technology for all sorts of roles — developers, architects, sysadmins, dev ops and SRE teams. *Learn Kubernetes in a Month of Lunches* isn't aimed at any single type of practitioner. I've tried to keep prerequisite knowledge to a minimum, so hopefully this book will work for you whatever your background in IT. That's where you and the other MEAPers can truly help: If you find gaps in your understanding in a chapter because I've made a bunch of assumptions about what you already know, then I want to hear that in the feedback.

Above all, this book tries to take you on a structured and enjoyable path to learn Kubernetes. A wealth of content is already out there, but it's difficult and frustrating to pick your way through outdated walkthroughs and overly technical documentation. This book is your friendly guide to Kubernetes, and the goal is to give you enough knowledge of what Kubernetes can do and enough experience in using it that, after reading, you'll feel comfortable migrating your own apps onto Kubernetes.

Thanks again for joining me, and let's get this book out there.

—Elton Stoneman

brief contents

WEEK 1: FAST TRACK TO KUBERNETES

- 1 Before You Begin*
- 2 Running containers in Kubernetes with Pods and Deployments*
- 3 Connecting Pods over the Network with Services*
- 4 Configuring applications with ConfigMaps and Secrets*
- 5 Storing data with volumes, mounts and claims*
- 6 Scaling applications across multiple Pods with Controllers*

WEEK 2: KUBERNETES IN THE REAL WORLD

- 7 Extending applications with multi-container Pods*
- 8 Running data-heavy apps with StatefulSets and Jobs*
- 9 Managing app releases with rollouts and rollbacks*
- 10 Packaging and managing apps with Helm*
- 11 App development - developer workflows and CI/CD*

WEEK 3: PREPARING FOR PRODUCTION

- 12 Empowering self-healing apps*
- 13 Centralizing logs with Fluentd and Elasticsearch*
- 14 Monitoring applications and Kubernetes with Prometheus*
- 15 Managing incoming traffic with Ingress*
- 16 Securing applications with Policies, Contexts and Admission Control*

WEEK 4: PURE AND APPLIED KUBERNETES

- 17 Securing resources with role-based access control*
- 18 Deploying Kubernetes: multi-node and multi-architecture clusters*
- 19 Controlling workload placement and automatic scaling*
- 20 Extending Kubernetes with Custom Resources and Operators*
- 21 Running serverless functions in Kubernetes*
- 22 Never the end*

APPENDICES

- A Packaging applications from source code as Docker Images*
- B Adding observability with containerized monitoring*
- C Reading app configuration from the Docker platform*
- D Writing application logs to the Docker platform*

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 over 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 which 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 which 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* which you use to define your applications, and the *cluster* which runs your applications. A cluster is a set of individual servers which 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 very high level view of the cluster.

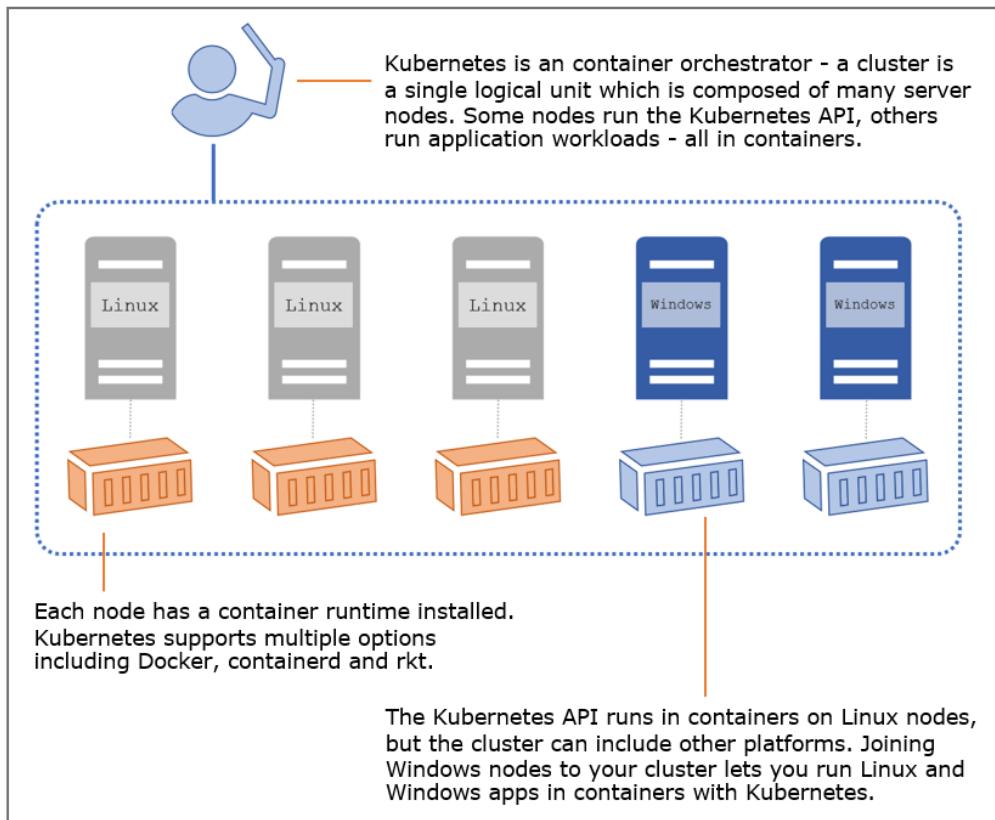


Figure 1.1 A Kubernetes cluster is just a bunch of servers which can run containers, joined into a group

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.

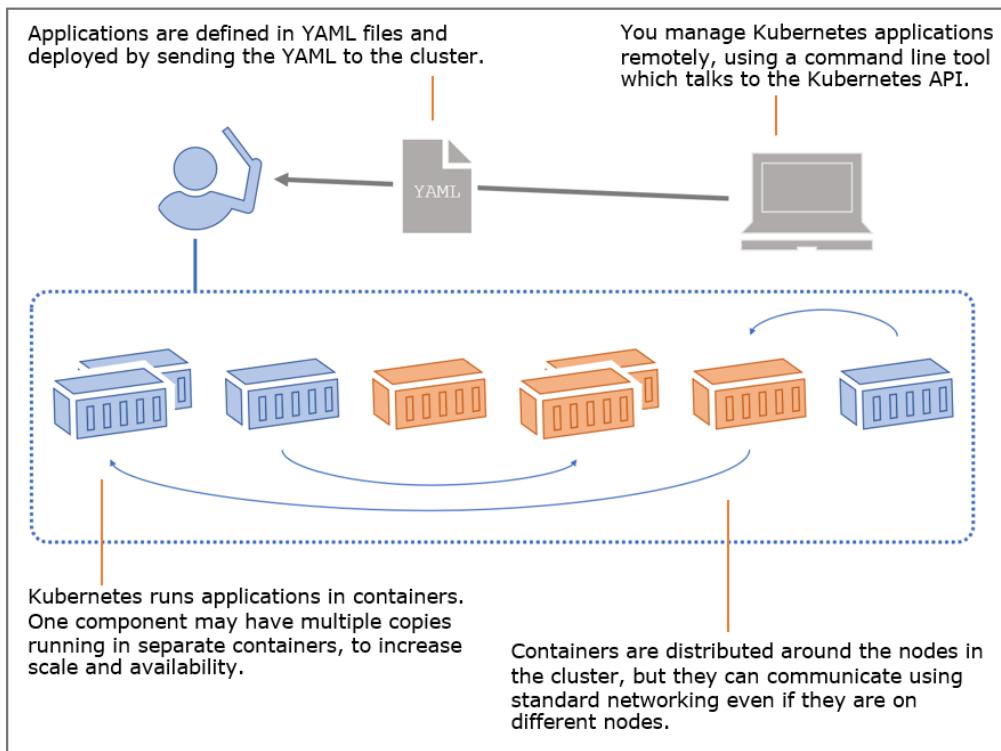


Figure 1.2 When you deploy apps to a Kubernetes cluster, you can usually ignore the actual nodes

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 which 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.

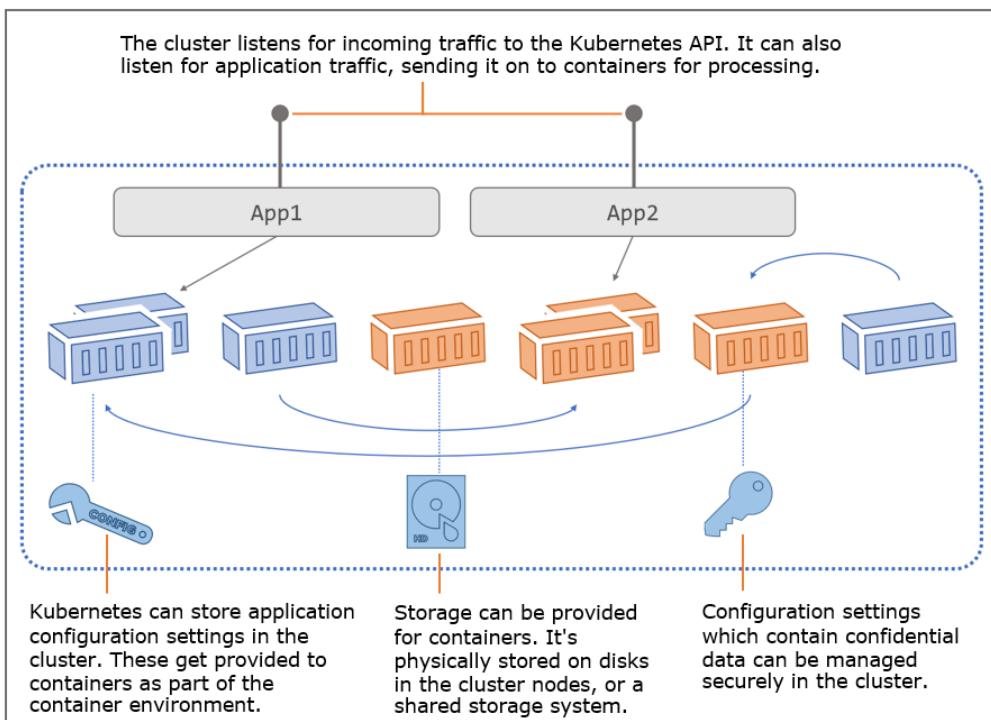


Figure 1.3 There's more to Kubernetes than just containers - the cluster manages other resources too

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 monoliths 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.

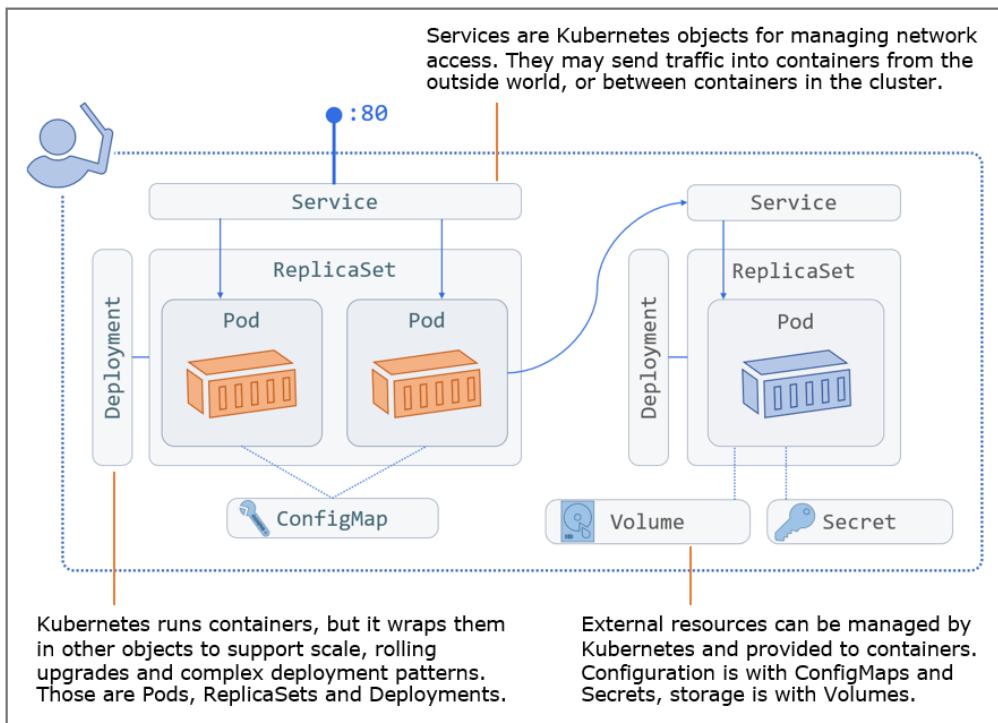


Figure 1.4 The true picture - these are the most basic Kubernetes resources you need to master

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 understand 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 very 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'd 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 which 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 manifests which Kubernetes will run as containers, how you can configure network access for containers to talk to each other, and for traffic from the outside world to reach your containers. You'll learn how your apps can read configuration from Kubernetes and 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 which 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 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 which 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 which 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 which 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, 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 which 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 which 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 setup 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.

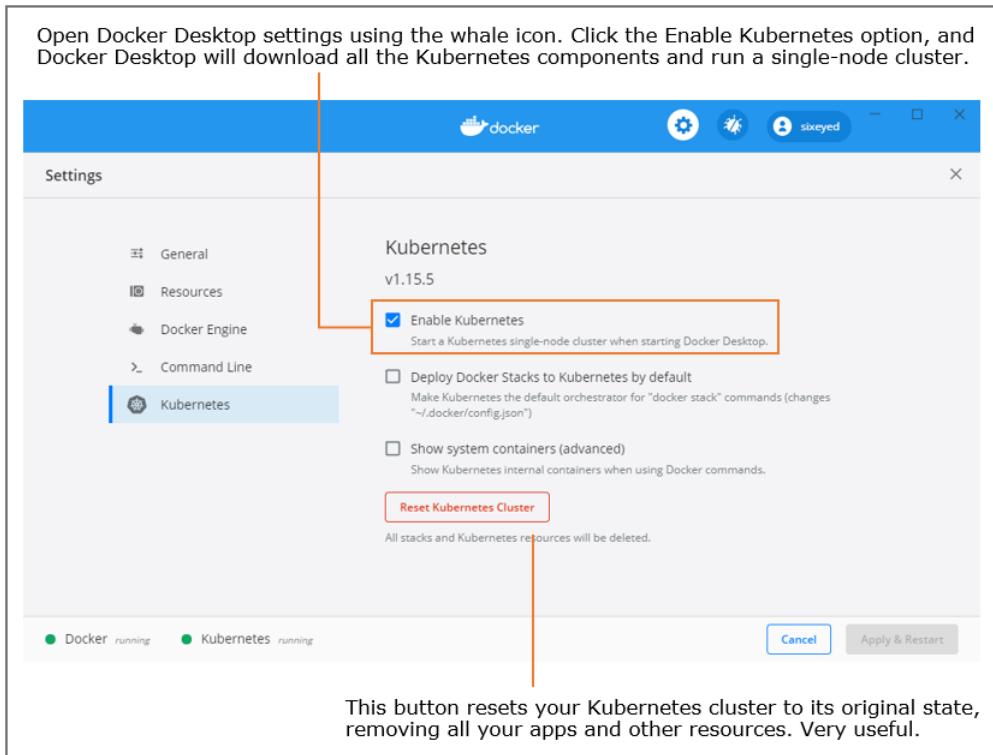


Figure 1.5 Docker Desktop creates and manages a Linux VM to run containers, and it can run Kubernetes

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 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 onto 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 below 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 Run a single-node Kubernetes Cluster in Azure

You can run a managed Kubernetes cluster in Microsoft Azure using the Azure Kubernetes Service (AKS). This might be a good option if you want to access your cluster from multiple

machines, or if you have an MSDN subscription with Azure credits. You can run a minimal single-node cluster which won't cost a huge amount, but bear in mind that there's no way to stop the cluster and you'll be paying for it 24x7 until you remove it.

The Azure portal has a nice user interface for creating an AKS cluster, but it's much easier to use the az command. You can check the latest docs at <https://docs.microsoft.com/en-us/azure/aks/kubernetes-walkthrough>, but you can get started by downloading the az command line tool and running a few commands.

```
# login to your Azure subscription:
az login

# create a resource group for the cluster:
az group create --name kiamol --location eastus

# create a single-node cluster with 2 CPU cores and 8GB RAM:
az aks create --resource-group kiamol --name kiamol-aks --node-count 1 --node-vm-size Standard_DS2_v2 --kubernetes-version 1.18.8 --generate-ssh-keys

# download certificates to use the cluster with kubectl:
az aks get-credentials --resource-group kiamol --name kiamol-aks
```

That final command downloads the credentials to connect to the Kubernetes API from your local kubectl command line.

1.4.6 Run a single-node Kubernetes Cluster in AWS

The managed Kubernetes service in AWS is called the Elastic Kubernetes Service (EKS). You can create a single-node EKS cluster with the same caveat as Azure - that you'll be paying for that node and associated resource all the time it's running.

You can use the AWS portal to create an EKS cluster, but the recommended way is with a dedicated tool called eksctl. The latest documentation for the tool is at <https://eksctl.io>, but it's pretty simple to use. First install the latest version of the tool for your operating system:

```
# install on macOS:
brew tap weaveworks/tap
brew install weaveworks/tap/eksctl

# OR on Windows:
choco install eksctl

# OR on Linux:
curl --silent --location
  "https://github.com/weaveworks/eksctl/releases/download/latest/eksctl_$(uname
  -s)_amd64.tar.gz" | tar xz -C /tmp
sudo mv /tmp/eksctl /usr/local/bin
```

Assuming you already have the AWS CLI installed, eksctl will use the credentials from the CLI (if not then check the installation guide for authenticating eksctl). Then create a simple 1-node cluster:

```
# create a single node cluster with 2 CPU cores and 8GB RAM:
```

```
eksctl create cluster --name=kiamol --nodes=1 --node-type=t3.large
```

The tool sets up the connection from your local kubectl to the EKS cluster.

1.4.7 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.

Kubectl is the Kubernetes command line tool. You use it to work with local and remote clusters.

```
PS>kubectl get nodes
NAME           STATUS    ROLES      AGE     VERSION
docker-desktop  Ready     master    49d    v1.18.3
```

This command prints basic details about all the nodes in the cluster. I'm using Docker Desktop, so I have a single node.

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. Lastly 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 which 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.

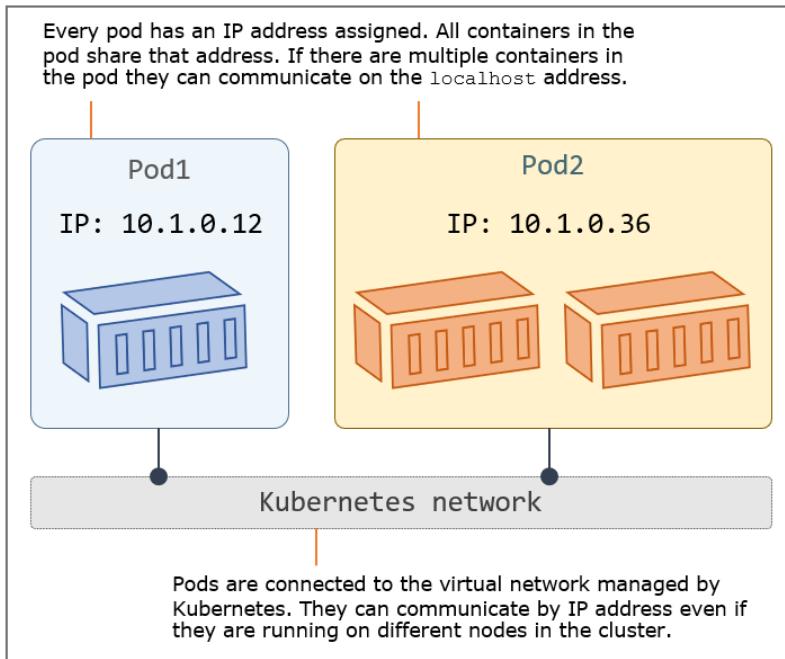


Figure 2.1 Containers run inside pods. You manage the pods and the pods manage the containers.

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

# 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.

```

Creates a pod named hello-kiamol running a single container, using
the image called kiamol/ch02-hello-kiamol from Docker Hub.

PS>kubectl run hello-kiamol --image=kiamol/ch02-hello-kiamol
--restart=Never
pod/hello-kiamol created
PS>
PS>kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
hello-kiamol   1/1     Running   0          6s
PS>
PS>kubectl describe pod hello-kiamol
Name:           hello-kiamol
Namespace:      default
Priority:       0
Node:           docker-desktop/192.168.65.3
Start Time:    Sat, 14 Mar 2020 20:22:36 +0000
Labels:         run=hello-kiamol
Annotations:   <none>
Status:        Running
IP:            10.1.1.27
Containers:
  hello-kiamol:

```

Shows detailed information about a single pod, including its IP address and the node it is running on.

Shows all the pods in the cluster. The ready column lists the number of containers in the pod and the number which are currently ready – this pod has a single container.

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 which

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.

The default output shows the container count, pod status, restart count and the age of the pod.

```
PS>kubectl get pod hello-kiamol
NAME          READY   STATUS    RESTARTS   AGE
hello-kiamol  1/1     Running   0          29m
```

You can specify custom columns by giving them a name and then using JSON notation to identify the data to return.

```
PS>kubectl get pod hello-kiamol --output custom-columns=NAME:metadata.name,NODE_IP:status.hostIP,POD_IP:status.podIP
NAME          NODE_IP          POD_IP
hello-kiamol  192.168.65.3   10.1.1.27
```

```
PS>
PS>kubectl get pod hello-kiamol -o jsonpath='{.status.containerStatuses[0].containerID}'
docker://11572486e38b5cda4b56559f8c9f3bef076ee2f132ea1fea123bd38871f4f8da
```

JSONPath is an alternative output format which supports complex queries. This query fetches the ID of the first container in the pod – there is only one in this case but there could be many, and the first index is zero.

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, and 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 which 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.pod.name=hello-kiamol

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

Kubernetes applies a pod name label to containers, so I can filter my Docker containers to find the pod container. The ID returned from the Docker command is the same ID Kubectl returned in figure 2.3.

```
PS>docker container ls -q --filter label=io.kubernetes.pod.name=hello-kiamol
11572486e38b
PS>
PS>docker container rm -f $(docker container ls -q --filter label=io.kubernetes.pod.name=hello-kiamol)
11572486e38b
PS>
PS>kubectl get pod hello-kiamol
NAME      READY   STATUS    RESTARTS   AGE
hello-kiamol  1/1     Running   0          75m
PS>
PS>docker container ls -q --filter label=io.kubernetes.pod.name=hello-kiamol
87fd7b828456
```

This deletes the container, which means the pod now has zero containers instead of the one required container.

Querying the pod shows that the container is running.

But it's a new container ID – the deleted container has been replaced by Kubernetes.

Figure 2.4 Kubernetes makes sure pods have all the containers they need

It's the abstraction from containers to pods which 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 which 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 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 which 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.

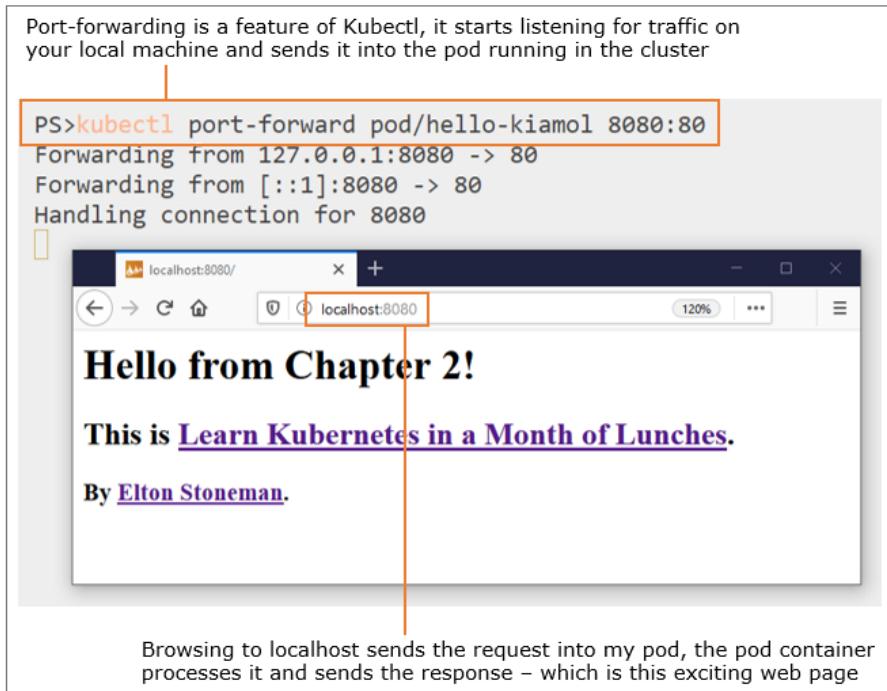


Figure 2.5 This app isn't configured to receive network traffic, but Kubectl can forward it

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 which 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 which can manage them for you?

That's where controllers come in. A *controller* is a Kubernetes resource which 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.

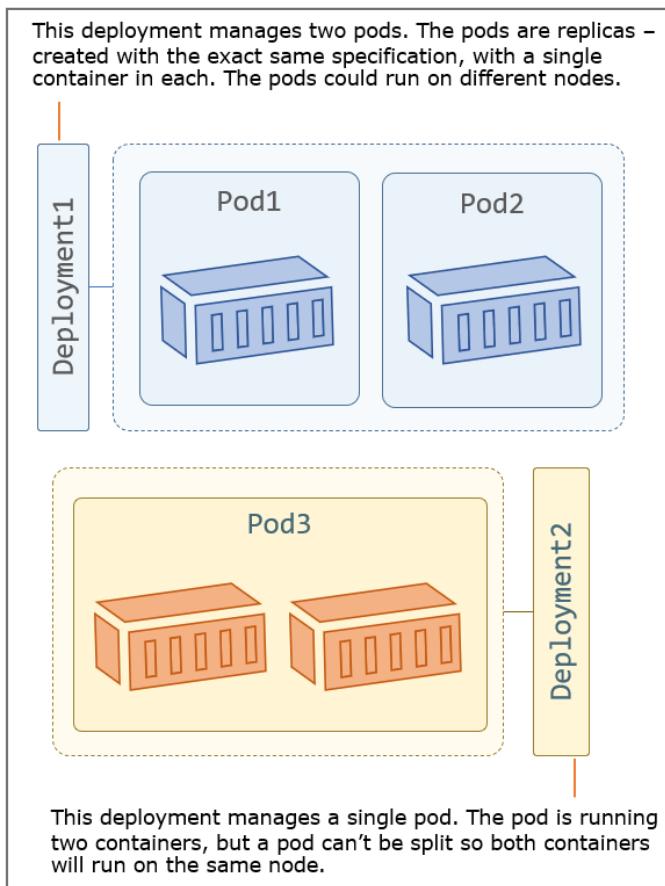


Figure 2.6 Deployment controllers manage pods and pods manage containers

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.

Creates a deployment named `hello-kiamol-2`. The number of replica pods to run isn't specified and the default is one, so this controller will manage a single pod.

```
PS>kubectl create deployment hello-kiamol-2 --image=kiamol/ch02-hello-kiamol  
deployment.apps/hello-kiamol-2 created  
PS>  
PS>kubectl get pods  
NAME                                     READY   STATUS    RESTARTS  
AGE  
hello-kiamol                               1/1     Running   0  
  68s  
hello-kiamol-2-56d95d56b6-g2wfg        1/1     Running   0  
  10s
```

The deployment-managed pod is created with a naming scheme that begins with the controller name and ends with 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 which 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 which 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 which have a matching label returns the single pod managed by the deployment.

Shows the details of the deployment, using a query which returns the labels the deployment applies to its pods.

```
PS>kubectl get deploy hello-kiamol-2 -o jsonpath='{.spec.template.metadata.labels}'  
map[app:hello-kiamol-2]
```

PS>

```
PS>kubectl get pods -l app=hello-kiamol-2  
NAME                  READY   STATUS    RESTARTS  
AGE  
hello-kiamol-2-56d95d56b6-g2wfg   1/1     Running   0  
29s
```

Lists pods matching the label selector – those having a label named `app` with the value `hello-kiamol-2`, which is the label set 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.

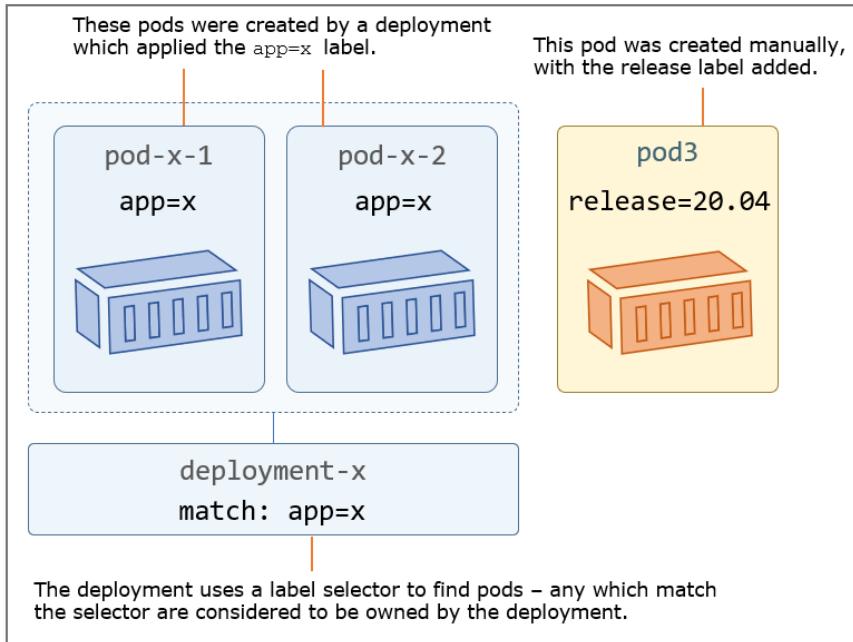


Figure 2.9 Controllers identify the resources they manage using labels and selectors

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 which match its label selector. Edit the labels on the pod and the deployment no longer recognizes it.

```
# 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.

```

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 can be a useful technique in debugging – removing a pod from a controller so you can connect and investigate a problem, while the controller starts a replacement pod which 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
```

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 which 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:

Confirm we still have two pods – one managed by the deployment and one unmanaged because of the label change.

PS> <code>kubectl get pods -l app -o custom-columns=NAME:metadata.name,LABELS:metadata.labels</code>	LABELS map[app:hello-kiamol-2 pod-template-hash:56d95d56b6] map[app:hello-kiamol-x pod-template-hash:56d95d56b6]
PS>	
PS> <code>kubectl label pods -l app=hello-kiamol-x --overwrite app=hello-kiamol-2</code>	
PS>	
PS> <code>kubectl get pods -l app -o custom-columns=NAME:metadata.name,LABELS:metadata.labels</code>	LABELS map[app:hello-kiamol-2 pod-template-hash:56d95d56b6]
PS>	

Reverses the previous change, setting the app label to hello-kiamol-2 from hello-kiamol-x. This brings the original pod back under the control of the deployment.

The deployment controller is only supposed to manage one pod, and now it has two – so it deletes one.

Figure 2.11 More label meddling - you can force a deployment to adopt a pod if the labels match

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 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.

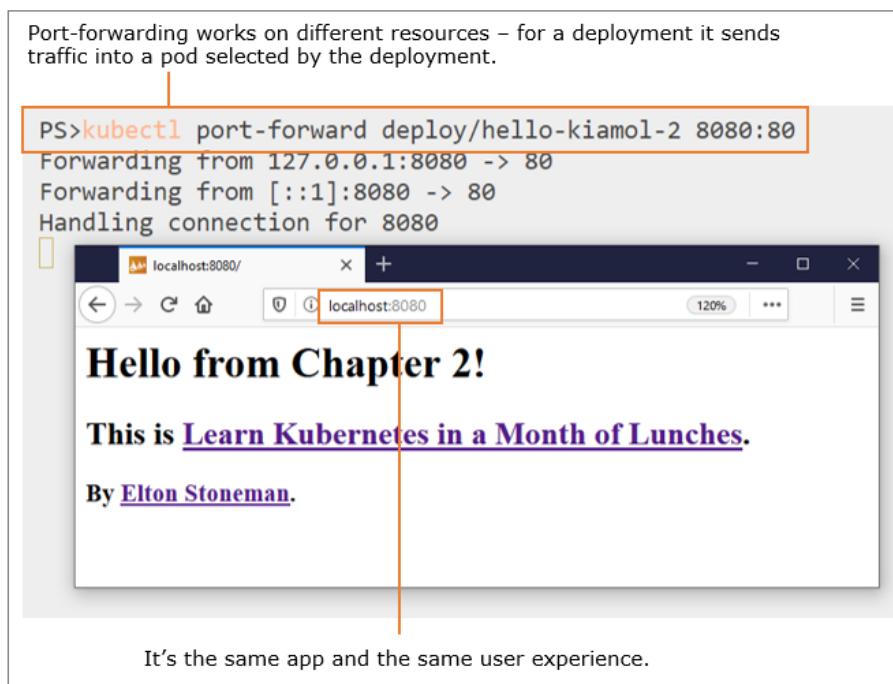


Figure 2.12 Pods and deployments are layers on top of containers, but the app still runs in a container

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.

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 which 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, let's you define multiple resources in a single file, and - most importantly - can record comments in the specification. Code 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:

Code 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 code 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>PS>kubectl apply -f pod.yaml  
pod/hello-kiamol-3 created  
PS>  
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.

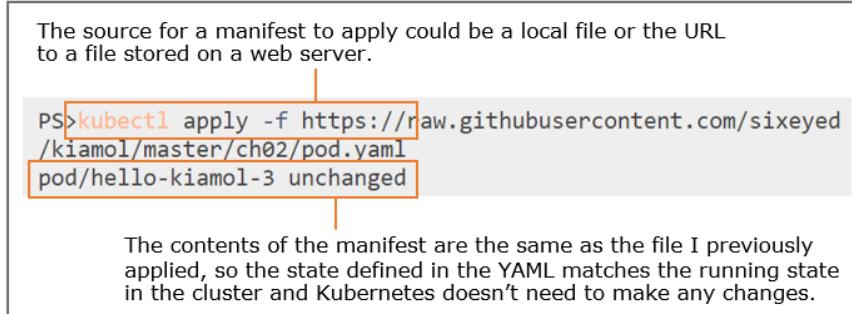


Figure 2.14 Kubectl can download manifest files from a web server and send them to the Kubernetes API

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 which includes the pod spec. Code listing 2.2 shows the minimal definition for a deployment resource, running yet another version of the same web app.

Code 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.

The apply command works in the same way for any types of resources defined in the YAML file. This only creates the deployment – then the deployment creates the pod.

```

PS>kubectl apply -f deployment.yaml
deployment.apps/hello-kiamol-4 created
PS>
PS>kubectl get pods -l app=hello-kiamol-4
NAME          READY   STATUS    RESTARTS
AGE
hello-kiamol-4-5b5b7c687b-vnbsq   1/1     Running   0
5s

```

The deployment creates a new pod straight away. It sets the label value defined in the manifest and we can use that to find the pod.

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

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. 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.

The first pod I created is still running; it has the IP address 10.1.0.12.

```

PS>kubectl get pod hello-kiamol -o custom-columns=NAME:metadata.name,POD_IP:status.podIP
NAME          POD_IP
hello-kiamol  10.1.0.12
PS>
PS>kubectl exec -it hello-kiamol sh
/ #
/ # hostname -i
10.1.0.12
/ #
/ # wget -O - http://localhost | head -n 4
Connecting to localhost (127.0.0.1:80)
-                               100% |*****| 353  0:00:00 ETA
<html>
<body>
<h1>
    Hello from Chapter 2!
/ #
/ # exit

```

The container's IP address is the pod's IP address.

Runs a shell command inside the pod container – the `-it` flag connects my terminal session to the shell in the container.

The web server is running in the container so I can access it using the localhost address.

Figure 2.16 You can use Kubectl to run commands inside pod containers, including interactive shells

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.

```

# 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.pod.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.

Shows the logs written by the pod container. The `tail` parameter restricts the output to the two most recent log entries.

```
PS>kubectl logs --tail=2 hello-kiamol
127.0.0.1 - - [07/Apr/2020:09:42:46 +0000] "GET / HTTP/1.1" 200 353 "-" "Wget
" "_"
127.0.0.1 - - [07/Apr/2020:09:43:42 +0000] "GET / HTTP/1.1" 200 353 "-" "Wget
" "_"
PS>
PS>docker container logs --tail=2 $(docker container ls -q --filter label=io.kubernetes.pod.name=hello-kiamol)
127.0.0.1 - - [07/Apr/2020:09:42:46 +0000] "GET / HTTP/1.1" 200 353 "-" "Wget
" "_"
127.0.0.1 - - [07/Apr/2020:09:43:42 +0000] "GET / HTTP/1.1" 200 353 "-" "Wget
" "_"
```

If you have access to the container runtime, you'll see that the pod logs are just a readout of the container logs.

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

The same features are available for all pods, no matter how they were created. Pods which 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 which 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 2.18 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.

The exec command can target different resources – like port-forward it can operate on pods or deployments. This executes wget inside the pod container and returns the output.

```
PS>kubectl exec deploy/hello-kiamol-4 -- sh -c 'wget -O - http://localhost > /dev/null'
Connecting to localhost (127.0.0.1:80)
-          100% |*****| 353 0:00:00 E
TA
PS>
PS>kubectl logs --tail=1 -l app=hello-kiamol-4
127.0.0.1 - - [07/Apr/2020:10:36:17 +0000] "GET / HTTP/1.1" 200 353 "-" "Wget
" "-"
```

Kubectl can show logs for multiple pods – using a label selector means you don't need to discover the random pod name to see its 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 which 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.

The `cp` command copies files between pod containers and your local filesystem. Here the source is a path in the `hello-kiamol` pod and the target is a local file path.

```
PS>mkdir -p /tmp/kiamol/ch02 | Out-Null
PS>
PS>kubectl cp hello-kiamol:/usr/share/nginx/html/index.html /tmp/kiamol/ch02/index.html
tar: removing leading '/' from member names
PS>
PS>cat /tmp/kiamol/ch02/index.html
<html>
<body>
<h1>
Hello from Chapter 2!
```

Internally Kubectl uses `tar` to compress and package files. This is an information message, not an error – but if my container image didn't have the `Tar` utility installed then I would get an error.

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?

I have four pods running – the two with simple names were created directly and the two with random suffixes were created by deployment controllers.

```
PS>kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
hello-kiamol                         1/1     Running   0          3h54m
hello-kiamol-2-7f6dd54b9b-f4ptj      1/1     Running   0          3h46m
hello-kiamol-3                         1/1     Running   0          3h39m
hello-kiamol-4-88696576b-lbgvs       1/1     Running   0          3h36m
PS>
PS>kubectl delete pods --all
pod "hello-kiamol" deleted
pod "hello-kiamol-2-7f6dd54b9b-f4ptj" deleted
pod "hello-kiamol-3" deleted
pod "hello-kiamol-4-88696576b-lbgvs" deleted
PS>
PS>kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
hello-kiamol-2-7f6dd54b9b-9njkd     1/1     Running   0          19s
hello-kiamol-4-88696576b-skdnm      1/1     Running   0          19s
```

The delete command works for different resources – using the all flag deletes all resources of that type. Beware, Kubectl does not ask for confirmation, it just goes and deletes all four pods.

But now I have two pods again. The deployments created replacements when their pods got deleted.

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 which will probably keep cropping up through all your days with Kubernetes. If you want to delete a resource which 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 which 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.

I have two deployments. Kubectl supports abbreviated resource names - you can use `deploy` for deployments and `po` for pods.

Kubectl mostly uses a consistent syntax, with a verb followed by a noun. You use the same delete command for all resource types.

PS> <code>kubectl get deploy</code>	NAME	READY	UP-TO-DATE	AVAILABLE	AGE
	hello-kiamol-2	1/1	1	1	4h32m
	hello-kiamol-4	1/1	1	1	4h22m

PS>
`kubectl delete deploy --all`
 deployment.apps "hello-kiamol-2" deleted
 deployment.apps "hello-kiamol-4" deleted

PS> <code>kubectl get pods</code>	NAME	READY	STATUS	RESTARTS	AGE
	hello-kiamol-2-7f6dd54b9b-9njkd	0/1	Terminating	0	46m
	hello-kiamol-4-88696576b-skdnm	0/1	Terminating	0	46m

PS>
`kubectl get all`
 NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
 service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 4d17h

These are the pods managed by the controllers
 - when deployments get deleted they delete their pods, and these are currently terminating.

This is a quick way to see everything in your cluster. All pods and deployments have been deleted now, so all that's left is the Kubernetes API.

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 which 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 which 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 - firstly because the IP address will change if the pod gets replaced, and secondly 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

# give Kubernetes time to download the Docker image and start the pods:
sleep 30

# 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
```

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

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

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 KinD 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 which 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:

```
Pods have a static IP address for their lifetime – this sleep-2 pod will always be accessible at 10.1.0.76  
  
PS>kubectl get pod -l app=sleep-2 --output jsonpath='{.items[0].status.podIP}'  
10.1.0.76  
PS>  
PS>kubectl delete pods -l app=sleep-2  
pod "sleep-2-766bb674b8-dnmnj" deleted  
PS>  
PS>kubectl get pod -l app=sleep-2 --output jsonpath='{.items[0].status.podIP}'  
10.1.0.78
```

Delete the pod and the deployment will create a replacement

The replacement pod is the same spec, but it has its own IP address – the new sleep-2 pod has the address 10.1.0.78

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), mapping 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.

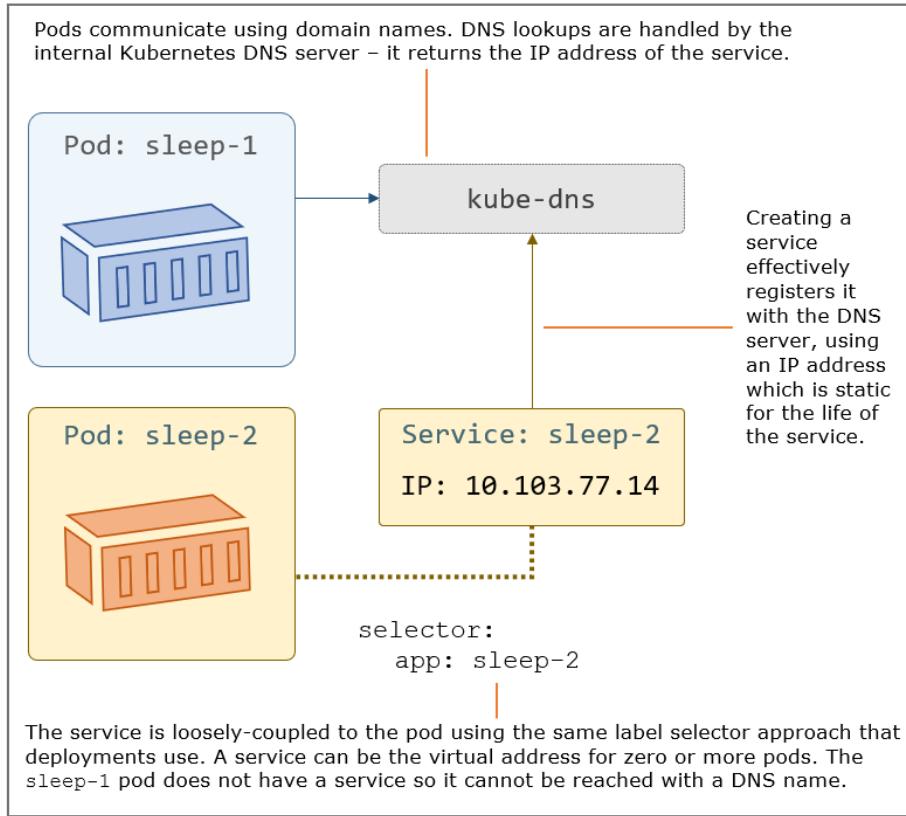


Figure 3.3 Services allow pods to communicate using a fixed domain name

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.

Code 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.

Code 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 code 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:
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:

The screenshot shows a terminal session with the following steps:

- Deploying the service: PS>`kubectl apply -f sleep/sleep2-service.yaml` service/sleep-2 created
- Verifying the service: PS>`kubectl get svc sleep-2`

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
sleep-2	ClusterIP	10.97.241.204	<none>	80/TCP	10s

- Running a ping command from a sleep-1 pod: PS>`kubectl exec deploy/sleep-1 -- ping -c 1 sleep-2`

Output: PING sleep-2 (10.97.241.204): 56 data bytes

--- sleep-2 ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
command terminated with exit code 1

Annotations in the screenshot:

 - "Deploys the service which links the domain name sleep-2 to the active sleep-2 pod." points to the first command.
 - "The service has its own IP address, which is accessible throughout the cluster." points to the table output.
 - "The sleep-1 pod does a domain name lookup and receives the service IP address from the Kubernetes DNS server." points to the ping command output.
 - "The ping command fails because it uses the ICMP protocol which Kubernetes services don't support. Services do support standard TCP and UDP traffic." points to the "command terminated with exit code 1" part of the ping output.

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 which 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 which 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

# give Kubernetes time to download the Docker image and start the pods:
sleep 30

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

Deploys a web application pod and an API, but no services so the components aren't able to communicate.

```
PS>kubectl apply -f .\numbers\api.yaml -f .\numbers\web.yaml
deployment.apps/numbers-api created
deployment.apps/numbers-web created

PS>
PS>kubectl port-forward deploy/numbers-web 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
Handling connection for 8080
```

The error message from the web app shows that it can't reach the API.

The domain name its using for the API is numbers-api, a local domain that doesn't exist in the Kubernetes DNS server.

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.sixeyed.com`), it's an address which 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 code listing 3.2 shows a service with the correct name and a label selector which matches the API pod.

Code 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 very similar to the service in code 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 code 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.

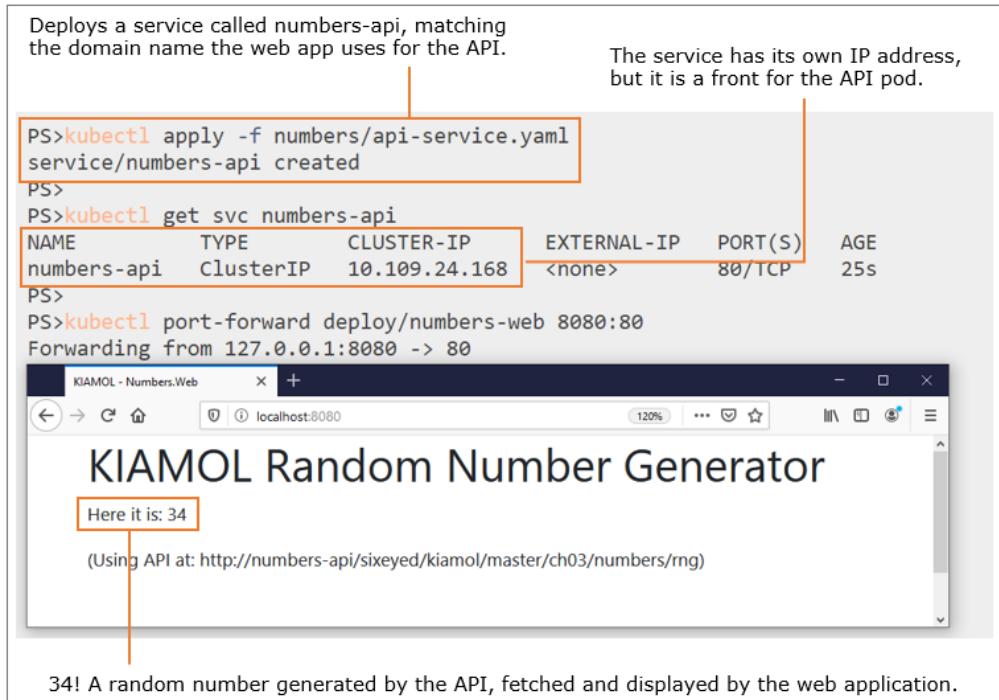


Figure 3.6 Deploying a service fixes the broken link between the web app and the API

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. 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.

The original API pod has the IP address 10.1.0.90.

NAME	POD_IP
numbers-api-bbd8d9bf4-tnhm4	10.1.0.90

The replacement pod has the IP address 10.1.0.91.

NAME	POD_IP
numbers-api-bbd8d9bf4-8fmptr	10.1.0.91

```
PS>kubectl get pod -l app=numbers-api -o custom-columns=NAME:metadata.name,POD_IP:status.podIP
NAME                               POD_IP
numbers-api-bbd8d9bf4-tnhm4      10.1.0.90
PS>
PS>kubectl delete pod -l app=numbers-api
pod "numbers-api-bbd8d9bf4-tnhm4" deleted
PS>
PS>kubectl get pod -l app=numbers-api -o custom-columns=NAME:metadata.name,POD_IP:status.podIP
NAME                               POD_IP
numbers-api-bbd8d9bf4-8fmptr     10.1.0.91
PS>
PS>kubectl port-forward deploy/numbers-web 8080:80
Forwarding from 127.0.0.1:8080 -> 80
```

The web pod is using the service name and service IP address, so the changed pod IP doesn't affect it.

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 which is fine for everything from local development to production. It's a type of a 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.

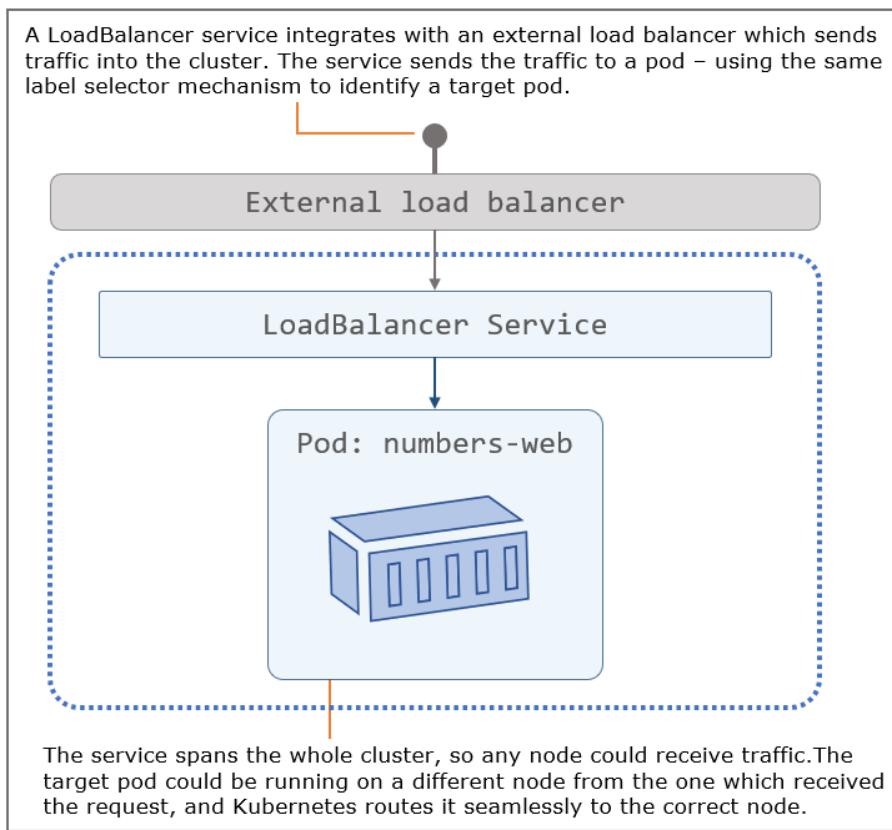


Figure 3.8 LoadBalancer services route external traffic from any node into a matching pod

It looks like a tricky problem, especially as 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. Code listing 3.3 shows the service specification for the web application.

Code 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

# browse to the address shown in the EXTERNAL-IP field, on port 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>.

LoadBalancer services listen on an external IP address to route traffic into the cluster. The address is provided by the cluster – I’m using Docker Desktop which routes using the localhost address.

LoadBalancer services also create a cluster IP, so the service is available to other pods using the service name.

```
PS>kubectl apply -f numbers/web-service.yaml
service/numbers-web created
PS>
PS>kubectl get svc numbers-web
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
numbers-web	LoadBalancer	10.101.42.205	localhost	8080:31461/TCP	8s

Now I can access the web application running in the pod without needing a port forward from Kubectl.

Figure 3.9 Kubernetes requests an IP address for LoadBalancer services from the platform it’s running on

The output is different using KinD 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 KinD cluster on my Linux desktop - here the website is at <http://172.17.255.1:8080>.

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

```
elton@sc-ub-brix:~$ kind get clusters
kiamol
elton@sc-ub-brix:~$
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
numbers-web	LoadBalancer	10.100.167.90	172.17.255.1	8080:31699/TCP	13h

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

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 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;
- KinD doesn't support load balancer services out of the box, so in the setup steps from chapter 1 you deployed an open-source product called MetalLB, which provides load balancer capabilities to bare-metal Kubernetes clusters (and KinD). You'll get a different IP address for each load balancer service in this setup, but they're virtual IPs which are only accessible from the machine running KinD;
- 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 which 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.

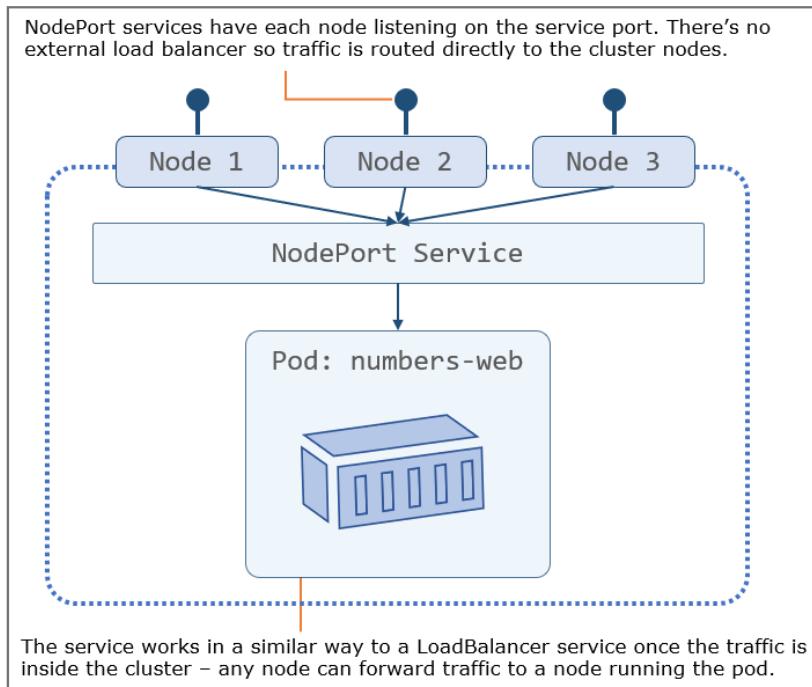


Figure 3.11 NodePort services also route external traffic to pods, but they don't require a load balancer

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 Minikube and Docker Desktop, but not so well in KinD. Code listing 3.4 shows a NodePort spec for reference.

Code 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.

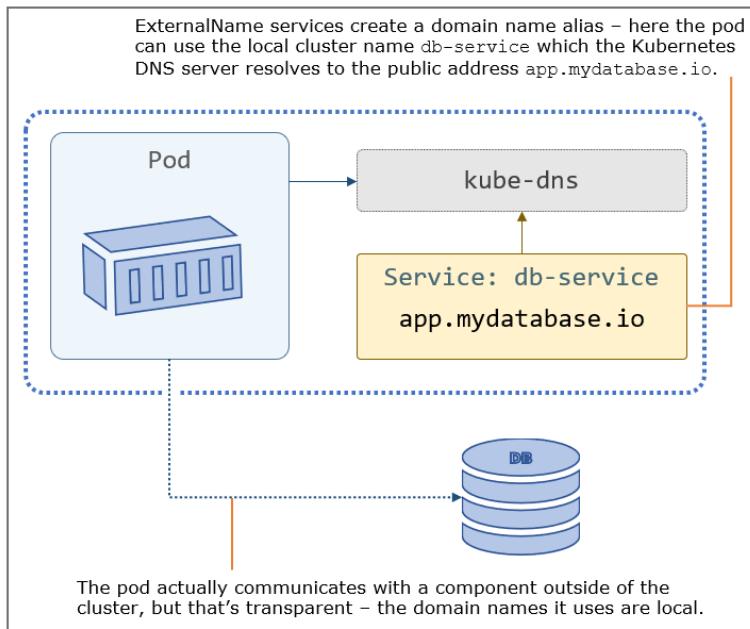


Figure 3.12 Using an ExternalName service lets you use local cluster addresses for remote components

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  
  
# browse to the website at http://localhost:8080 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 any more.

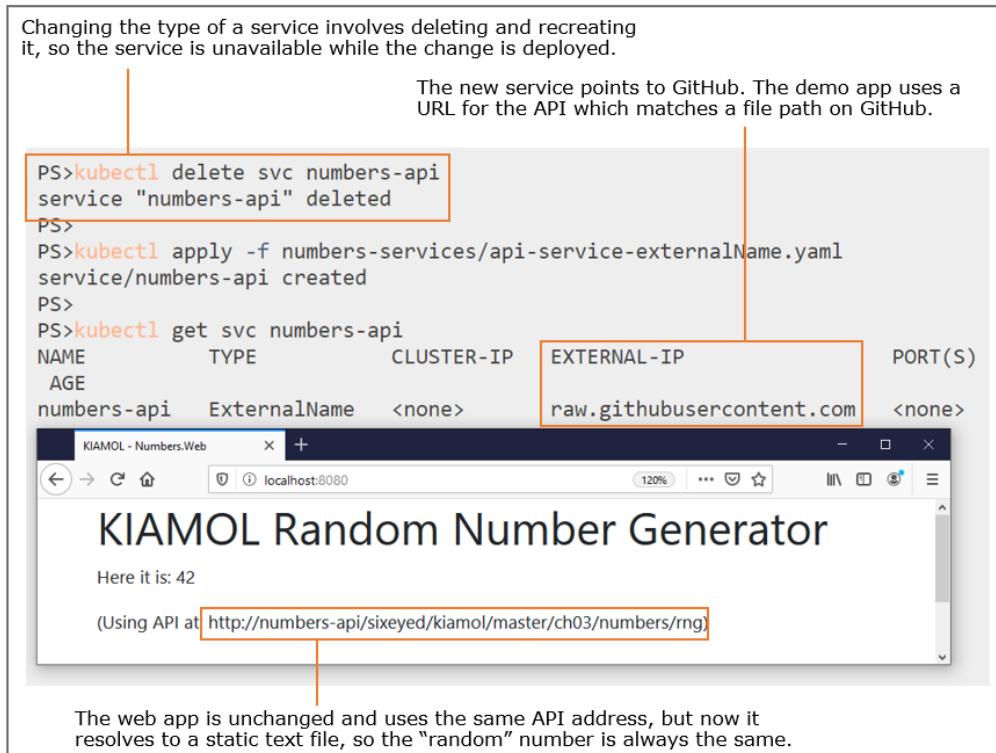


Figure 3.13 ExternalName services can be used as a redirect to send requests outside of the cluster

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 which 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 which resolves to a test database running in a pod; in production you can use an ExternalName service which resolves to the real domain name of the database server. Code listing 3.5 shows the YAML spec for the API external name.

Code 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.tastily.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 which you can see from this exercise - it ultimately just gives your app an address to use, it doesn't actually change the requests your application makes. That's fine for components like databases which communicate over TCP, but it's not so simple for HTTP services. 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 which explicitly lists the IP addresses the service should resolve.

Code 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.

Code 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 "^[^*]"'

# try the app at http://localhost:8080
```

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.

A headless service is deployed as a service with no label selector, and an endpoint specifying IP addresses.

The service itself is a ClusterIP type, which resolves to a virtual IP address in the cluster.

```

PS>kubectl delete svc numbers-api
service "numbers-api" deleted
PS>
PS>kubectl apply -f numbers-services/api-service-headless.yaml
service/numbers-api created
endpoints/numbers-api created
PS>
PS>kubectl get svc numbers-api
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
numbers-api   ClusterIP   10.99.221.227   <none>        80/TCP      9s
PS>
PS>kubectl get endpoints numbers-api
NAME      ENDPOINTS      AGE
numbers-api   192.168.123.234:80   19s
PS>
PS>kubectl exec deploy/sleep-1 -- sh -c 'nslookup numbers-api | 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>


```

The endpoint for the service shows the actual – fake – IP address to which the ClusterIP routes.

The IP address for the service doesn't exist, so the web app is broken.

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.

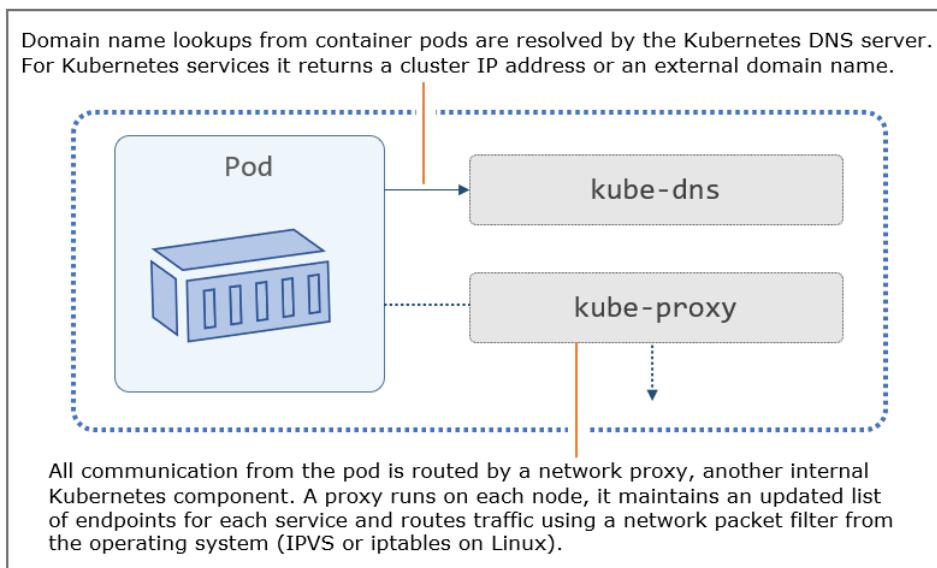


Figure 3.16 Kubernetes runs a DNS server and a proxy, and uses them with standard network tools

The key takeaway is that the ClusterIP is a virtual IP address which 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.

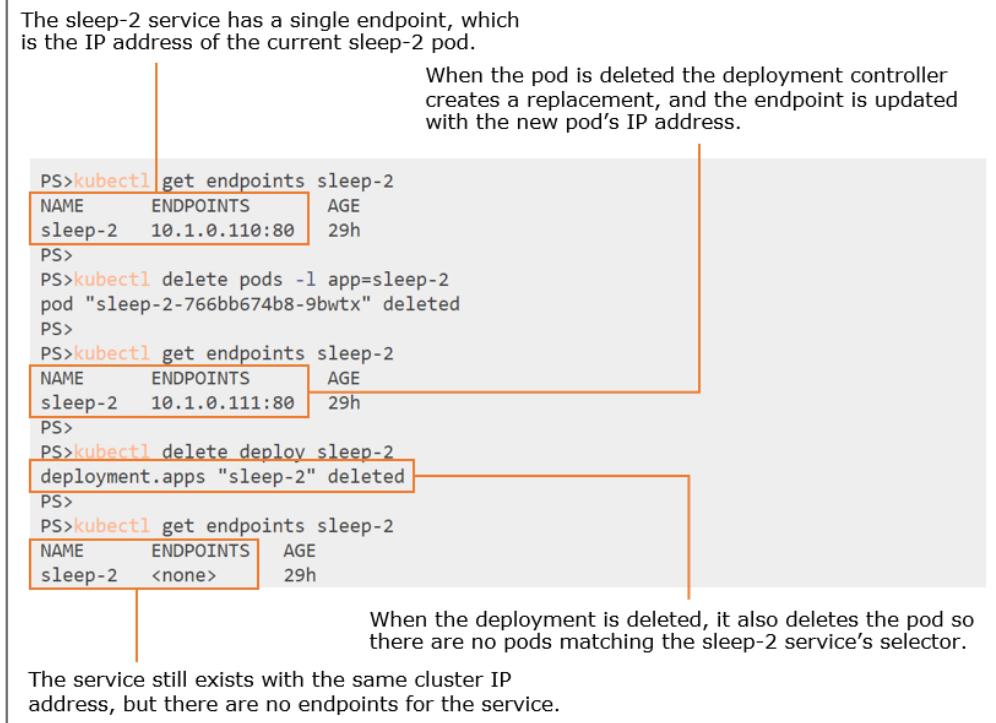


Figure 3.17 The cluster IP address for a service doesn't change but the endpoint list is always updating

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.

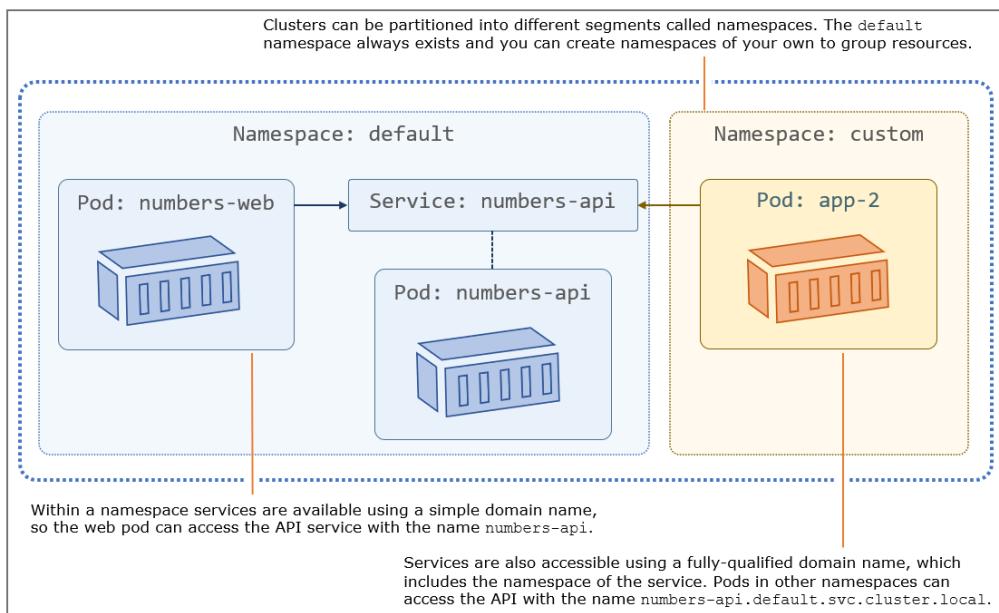


Figure 3.18 Namespaces logically partition a cluster, but services are accessible across namespaces

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 which 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> <code>kubectl get svc --namespace default</code>	EXTERNAL-IP	PORT(S)	AGE
NAME TYPE CLUSTER-IP	<none>	443/TCP	32h
kubernetes ClusterIP 10.96.0.1	<none>	80/TCP	6h57m
numbers-api ClusterIP 10.99.221.227	localhost	8080:31461/TCP	12h
numbers-web LoadBalancer 10.101.42.205	<none>	80/TCP	31h
sleep-2 ClusterIP 10.97.241.204			

PS> <code>kubectl get svc -n kube-system</code>	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> <code>kubectl exec deploy/sleep-1 -- sh -c 'nslookup numbers-api.default.svc.cluster.local grep "[^*]"'</code>	Server:	Address:	Name:	Address:
	10.96.0.10	10.96.0.10:53	numbers-api.default.svc.cluster.local	10.99.221.227

PS> PS> <code>kubectl exec deploy/sleep-1 -- sh -c 'nslookup kube-dns.kube-system.svc.cluster.local grep "[^*]"'</code>	Server:	Address:	Name:	Address:
	10.96.0.10	10.96.0.10:53	kube-dns.kube-system.svc.cluster.local	10.96.0.10

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.

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.

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 out from pods. 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.

Deleting deployments also deletes pods. There's no namespace parameter so this only deletes resources in the default namespace.

Services need to be explicitly deleted. Using the all parameter means I've accidentally deleted the Kubernetes API, which lives in the default namespace. Oops.

```
PS>kubectl delete deploy --all
deployment.apps "numbers-api" deleted
deployment.apps "numbers-web" deleted
deployment.apps "sleep-1" deleted
```

```
PS>kubectl delete svc --all
service "kubernetes" deleted
service "numbers-api" deleted
service "numbers-web" deleted
service "sleep-2" deleted
```

```
PS>
PS>kubectl get all
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP    PORT(S)      AGE
service/kubernetes  ClusterIP   10.96.0.1    <none>        443/TCP     46s
```

Luckily there's a controller in the kube-system namespace which manages the API service and recreates it.

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 which will make the API available to other pods at the domain name `numbers-api`
- write a service which will make version 2 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 which 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 - with `kubectl` you create commands 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.

Deploys a simple pod with just a container image specified and no additional config settings.

```
PS>cd ch04  
PS>  
PS>kubectl apply -f sleep/sleep.yaml  
deployment.apps/sleep created  
PS>  
PS>kubectl exec deploy/sleep -- printenv HOSTNAME KIAMOL CHAPTER  
sleep-b8f5f69-ncz57  
command terminated with exit code 1
```

Printenv is a Linux command which shows the value of environment variables. The `HOSTNAME` variable exists in all pod containers and is set by Kubernetes to be the pod name. The `KIAMOL CHAPTER` variable doesn't exist, so the command exits with an error code.

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.

Code listing 4.1 - sleep-with-env.yaml, a pod spec with environment variables

```
spec:  
  containers:  
    - name: sleep  
      image: kiamol/ch03-sleep  
      env:  
        - name: KIAMOL CHAPTER      # 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.

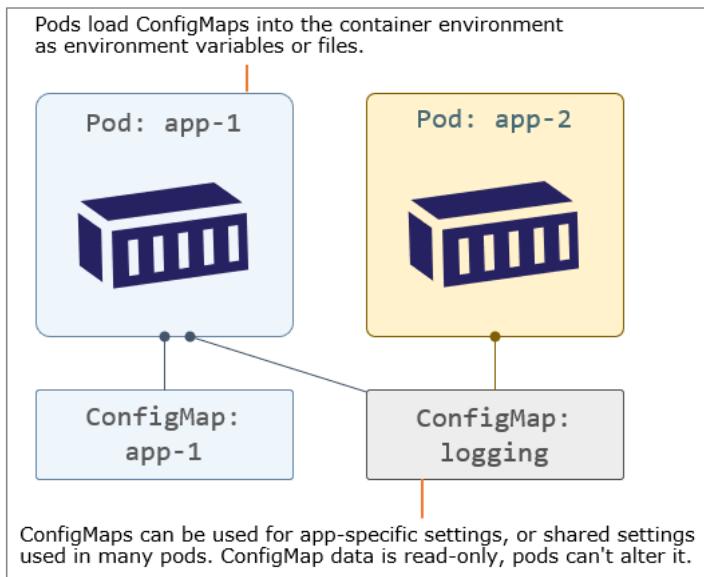


Figure 4.3 ConfigMaps are separate resources, which can be attached to zero or more pods.

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.

Code listing 4.2 - sleep-with-configMap-env.yaml, loading a ConfigMap into a pod

```
env:
- name: KIAMOL CHAPTER
  value: "04"
- name: KIAMOL SECTION
  valueFrom:
    configMapKeyRef:
      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 - 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.

The details of the ConfigMap (using the alias cm) show it has a single data item.

```

PS> kubectl create configmap sleep-config-literal --from-literal=kiamol.section='4.1'
configmap/sleep-config-literal created
PS>
PS> kubectl get cm sleep-config-literal
NAME          DATA   AGE
sleep-config-literal  1    7s
PS>
PS> kubectl describe cm sleep-config-literal
Name:           sleep-config-literal
Namespace:      default
Labels:         <none>
Annotations:    <none>

Data
=====
kiamol.section:
-----
4.1
Events:        <none>
PS>
PS> kubectl apply -f sleep/sleep-with-configMap-env.yaml
deployment.apps/sleep configured
PS>
PS> kubectl exec deploy/sleep -- sh -c 'printenv | grep "^\$KIAMOL"'
KIAMOL_SECTION=4.1
KIAMOL CHAPTER=04

```

Creating a ConfigMap from a literal value - the key kiamol.section is set with the value 4.1 in the command parameters.

Describing the ConfigMap shows the object metadata and all the config data items.

This pod loads the ConfigMap data as an environment variable - renaming the key to KIAMOL_SECTION. The other environment variable value is set in the deployment YAML.

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.

Code 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.

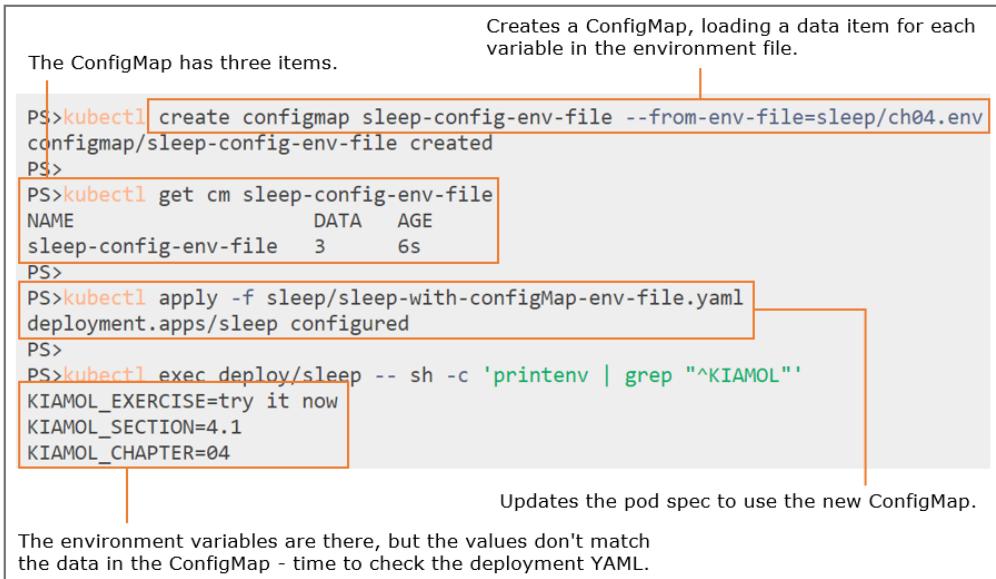


Figure 4.5 A ConfigMap can have multiple data items, and the pod can load them all

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, it loads all environment variables from the ConfigMap, but it also specifies explicit environment values with some of the same keys.

Code 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 which 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.

Code 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 which 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

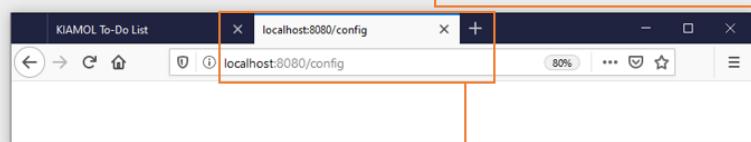
# get the address of the app:
kubectl get svc todo-web -o
jsonpath='http://{.status.loadBalancer.ingress[0].*}:8080'

# 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 we 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.

Deploys a pod spec where the app uses default config settings in the container image, plus one custom setting in an environment variable.

```
PS>kubectl apply -f todo-list/todo-web.yaml
service/todo-web created
deployment.apps/todo-web created
PS>
PS>kubectl get svc todo-web
NAME      TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
todo-web  LoadBalancer  10.102.23.110  localhost       8080:30542/TCP  3s
PS>
PS> 
PS>
PS>
PS> kubectl logs -l app=todo-web
[warn]: ToDoList.Pages.ConfigModel[0]
Attempt to view config settings
I should be able to check the config settings here, but
in the default configuration the app returns a 404.
And logs a warning that someone
tried to view the page.
```

The app uses a LoadBalancer service - on Docker Desktop I can browse to it on localhost.

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

The config hierarchy in use here is a very common approach. This is a .NET Core app which 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 in a known location, which the app checks and merges with the content from the default file.
- any settings which 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.

Code 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 its 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 which blocked access to that page.

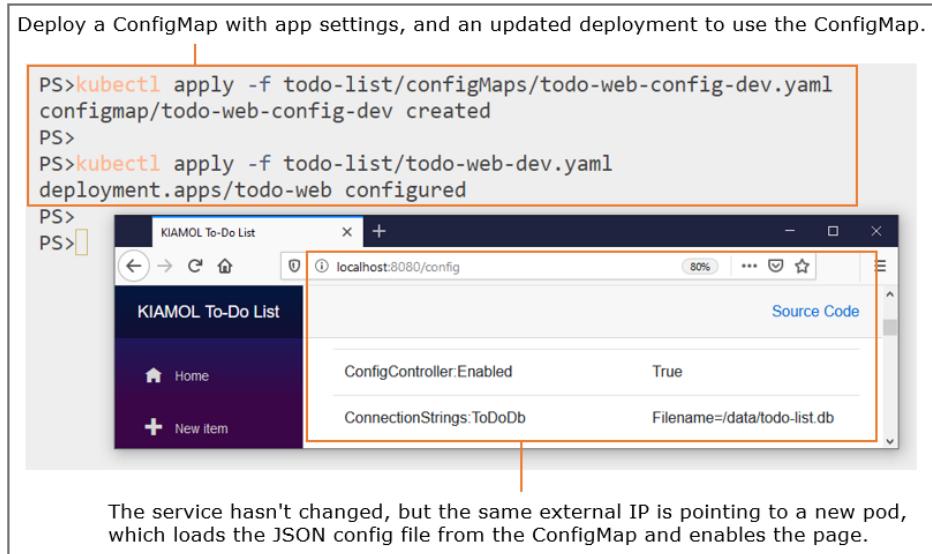


Figure 4.7 Loading ConfigMap data into the container filesystem, where the app loads config files

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 ConfigMap 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.

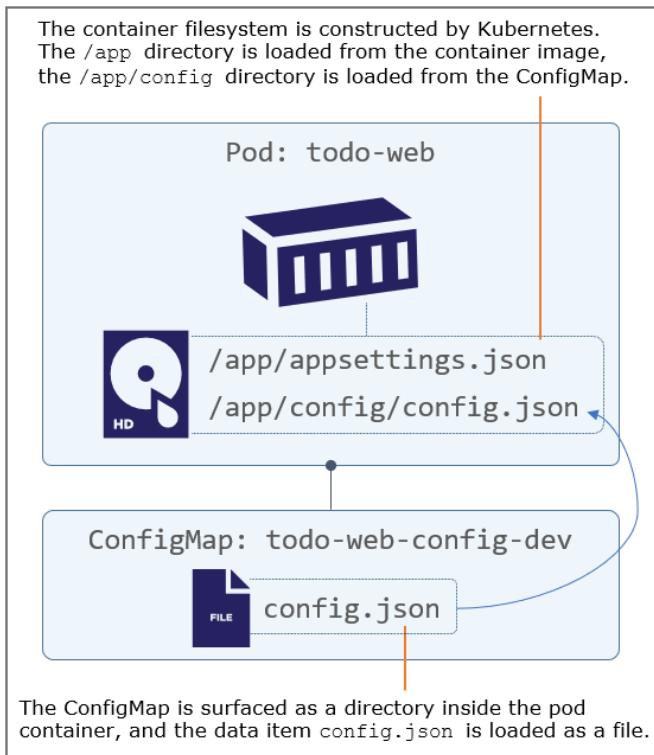


Figure 4.8 ConfigMaps can be loaded as directories in the container filesystem

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* which 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.

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

```

spec:
  containers:
    - name: web
      image: kiamol/ch04-todo-list
      volumeMounts:
        - name: config
          mountPath: "/app/config"
          readOnly: true
      # mounts a volume into the container
      # name of the volume
      # directory path to mount the volume
      # optionally flag the volume read-only

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

This is the default app configuration file, loaded from the container image. It has the file permissions which are set in the image.

```
PS>kubectl exec deploy/todo-web -- sh -c 'ls -l /app/app*.json'  
-rw-r--r-- 1 root root 333 Apr 16 14:55 /app/appsettings.json  
PS>  
PS>kubectl exec deploy/todo-web -- sh -c 'ls -l /app/config/*.json'  
lrwxrwxrwx 1 root root 18 Apr 16 20:13 /app/config/config.json -> ..data/config.json  
PS>  
PS>kubectl exec deploy/todo-web -- sh -c 'echo ch04 >> /app/config/config.json'  
sh: 1: cannot create /app/config/config.json: Read-only file system  
command terminated with exit code 2
```

This is the environment config file, loaded from the ConfigMap. The file appears to have read-write permissions, but the path is actually a link to a read-only file.

As you can see when you try to edit the file.

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.

Code 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 which 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.

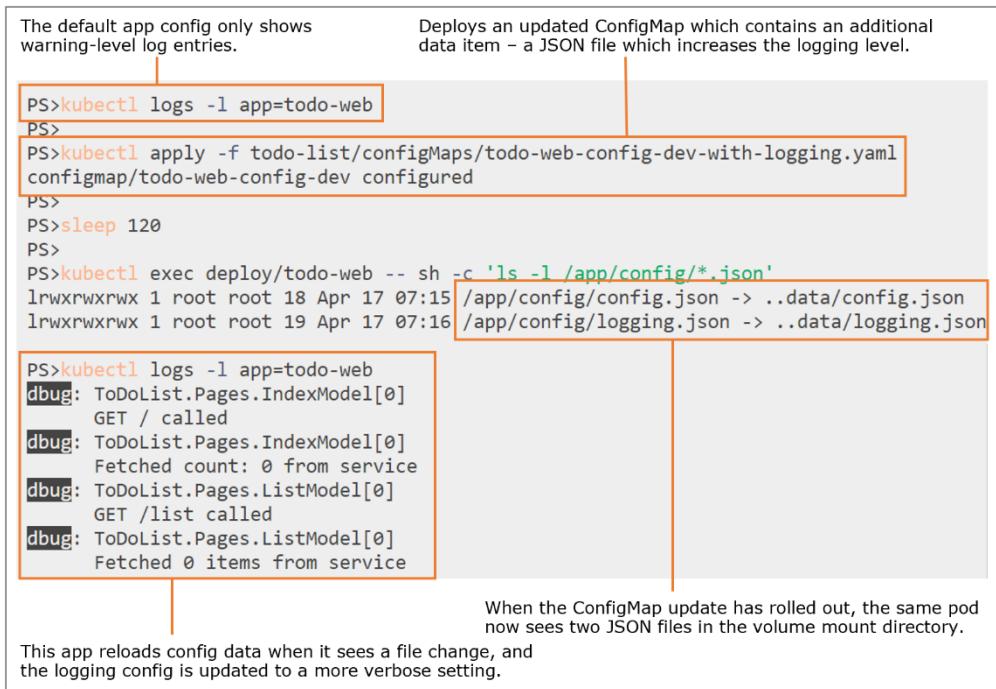


Figure 4.10 ConfigMap data is cached, so it takes couple of minutes for updates to reach pods

Volumes are a powerful option for loading config, especially with apps like this which 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.

Code 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.

Deploy the change – this will roll out a pod with a broken app container.

There are log entries from two pods here – the debug log entries are from the original pod, and the SDK failure message is from the new pod.

```
PS>kubectl apply -f todo-list/todo-web-dev-broken.yaml
deployment.apps/todo-web configured
PS>
PS>kubectl logs -l app=todo-web
It was not possible to find any installed .NET Core SDKs
Did you mean to run .NET Core SDK commands? Install a .NET Core SDK from:
  https://aka.ms/dotnet-download
dbug: ToDoList.Pages.IndexModel[0]
  GET / called
dbug: ToDoList.Pages.IndexModel[0]
  Fetched count: 0 from service
dbug: ToDoList.Pages.ListModel[0]
  GET /list called
dbug: ToDoList.Pages.ListModel[0]
  Fetched 0 items from service
PS>
PS>kubectl get pods -l app=todo-web
NAME          READY   STATUS      RESTARTS   AGE
todo-web-66944dc6db-dv6bq   0/1     CrashLoopBackOff   3          88s
todo-web-74fb9c994f-ntnw2   1/1     Running      0          111m
```

Two pods match the deployment label. The Running pod is the original, and the new pod has had three restarts. Kubernetes will wait before restarting the container again, which is the CrashLoopBackOff status.

Figure 4.11 If an updated deployment fails then the original pod doesn't get replaced

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.

Code 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 directory
```

```

        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

```

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.

Deploys an updated pod spec which only loads the app config file from the ConfigMap, and not the logging config. The ConfigMap itself is not changed.

```
PS>kubectl apply -f todo-list/todo-web-dev-no-logging.yaml
deployment.apps/todo-web configured
```

The container only sees the single JSON file explicitly set in the volume items.

```
PS>
PS>kubectl exec deploy/todo-web -- sh -c 'ls /app/config'
config.json
```

PS>

```
PS>kubectl logs -l app=todo-web
PS>
PS>kubectl get pods -l app=todo-web
NAME          READY   STATUS    RESTARTS   AGE
todo-web-7f64c56bd9-9hx55   1/1     Running   0          45s
```

After using the app there are no logs because the new pod is using the default logging configuration.

Only one pod is running. The new deployment was successful, so the previous failed pod and the original pods are removed.

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.

<p>Adds a base64 command - only needed for Windows users, as Linux and macOS have their own commands.</p> <pre>PS> .\base64.ps1 PS></pre>	<p>Creates a secret with the key <code>secret</code> from a literal value in the command.</p> <pre>PS> kubectl create secret generic sleep-secret-literal --from-literal=secret=shh... secret/sleep-secret-literal created PS> PS> kubectl describe secret sleep-secret-literal Name: sleep-secret-literal Namespace: default Labels: <none> Annotations: <none> Type: Opaque Data ==== secret: 6 bytes PS></pre>
<p>Describing the secret only shows the key and the data length, not the value.</p>	<p>Fetching the data field shows the base64-encoded string. It's obfuscated from the original, but this is not encryption - the value can easily be decoded.</p>
<pre>PS> kubectl get secret sleep-secret-literal -o jsonpath='{.data.secret}' c2h0Li4u PS> PS> kubectl get secret sleep-secret-literal -o jsonpath='{.data.secret}' base64 -d shh...</pre>	

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

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.

Code listing 4.11 - sleep-with-secret.yaml, a pod spec loading a Secret

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

TRY IT NOW Run a simple sleep pod which 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.

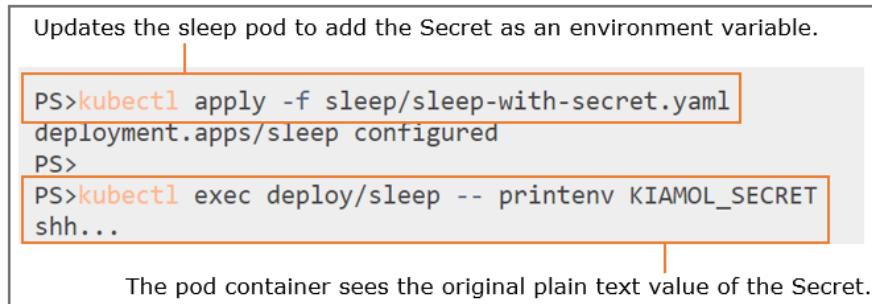


Figure 4.14 Secrets loaded into pods are not Base64 encoded

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.

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

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

```

Creates a Secret from a manifest which
stores the data item in plain text.

PS> kubectl apply -f todo-list/secrets/todo-db-secret-test.yaml
secret/todo-db-secret-test created

The data is Base64 encoded in the same
way as for secrets created as literals.

PS> kubectl get secret todo-db-secret-test -o jsonpath='{.data.POSTGRES_PASSWORD}'
a21hbW9sLTIqMio=
PS>
PS> kubectl get secret todo-db-secret-test -o jsonpath='{.metadata.annotations}'
map[kubectl.kubernetes.io/last-applied-configuration:{"apiVersion":"v1","kind":"Secret",
"metadata": {"annotations":{}, "name": "todo-db-secret-test", "namespace": "default"}, "stringData": {"POSTGRES_PASSWORD": "kiamol-2*2*"}, "type": "Opaque"}]
]

But the original plain text is also stored in an annotation. Annotations are
used for notes about the deployment, and this stores the whole contents.

```

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` - 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.

Code 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:
          - name: secret                  # mounts a Secret volume
            mountPath: "/secrets"         # the name of the volume
  volumes:
    - name: secret
      secret:
        secretName: todo-db-secret-test # volume loaded from a Secret
        defaultMode: 0400               # Secret name
        items:                          # permissions to set for files
          - key: POSTGRES_PASSWORD      # optionally name the data items
            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 environment 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.

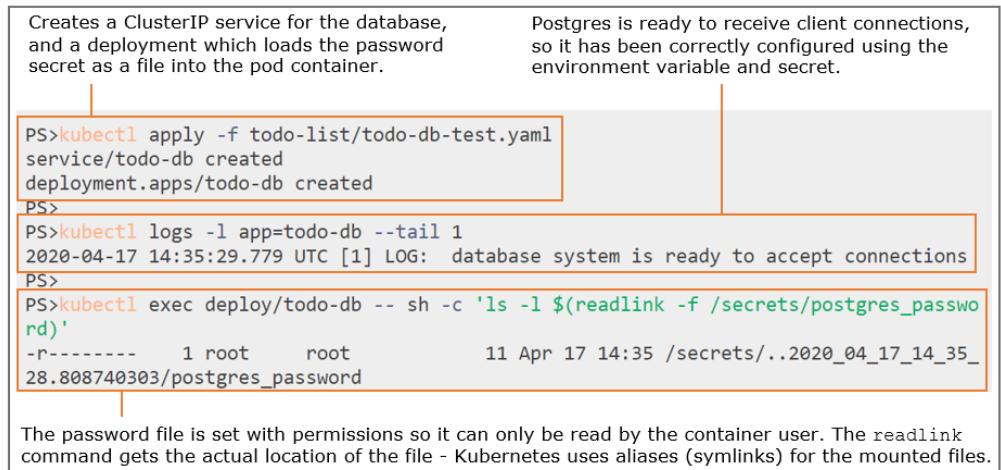


Figure 4.16 If the app supports it, configuration settings can be read by files populated from Secrets

All that's left is to run the app itself in its 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.

The full app configuration is supplied from the ConfigMap and the Secret - one stores app settings and the other stores sensitive database credentials.

```
PS>kubectl apply -f todo-list/configMaps/todo-web-config-test.yaml
configmap/todo-web-config-test created
PS>
PS>kubectl apply -f todo-list/secrets/todo-web-secret-test.yaml
secret/todo-web-secret-test created
PS>
PS>kubectl apply -f todo-list/todo-web-test.yaml
service/todo-web-test created
deployment.apps/todo-web-test created
PS>
PS>kubectl exec deploy/todo-web-test -- cat /app/secrets/secrets.json
{
  "ConnectionStrings": {
    "ToDoDb": "Server=todo-db;Database=todo;User Id=postgres;Password=kiamol-2*2*;"
  }
}
```

Inside the container filesystem the Secret is presented as plain text JSON, with the database connection details.

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, and 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.

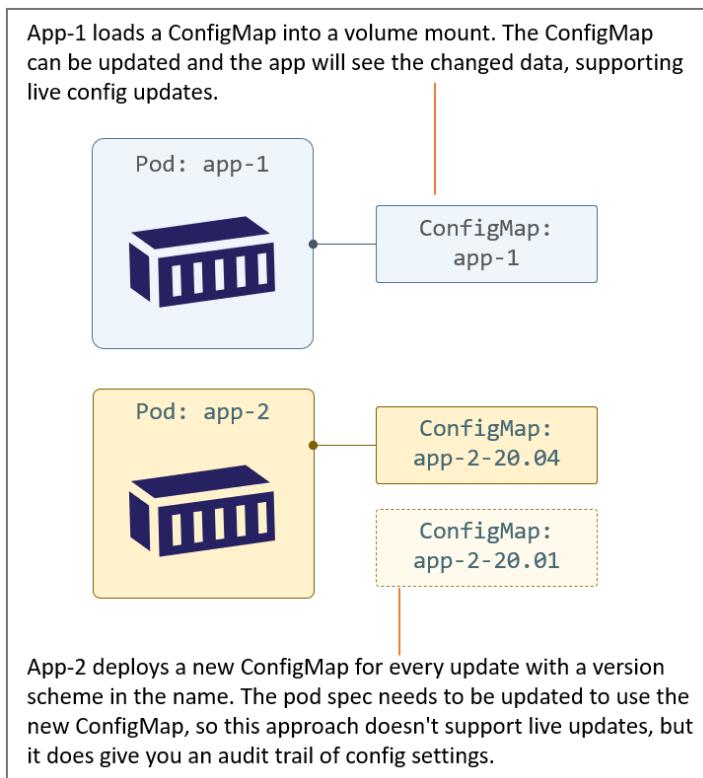


Figure 4.18 You can choose your own approach to config management, supported by Kubernetes

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

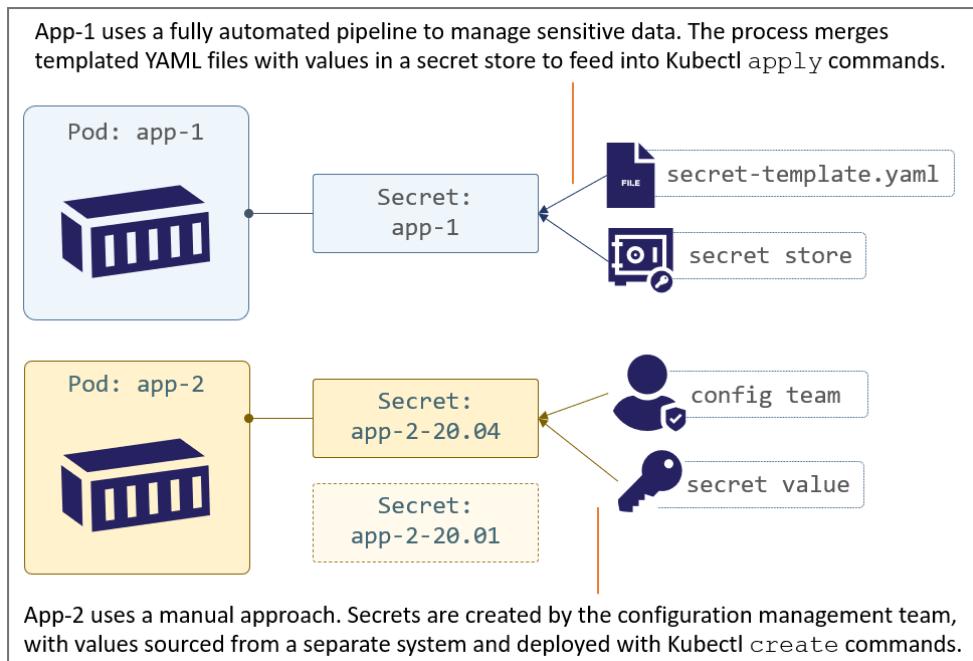


Figure 4.19 Secret management can be automated in deployment or strictly controlled by a separate team

You can use any approach which 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
- 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 which works for every scenario. Apps have different storage requirements, and the platforms which 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 writable storage layer which 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.

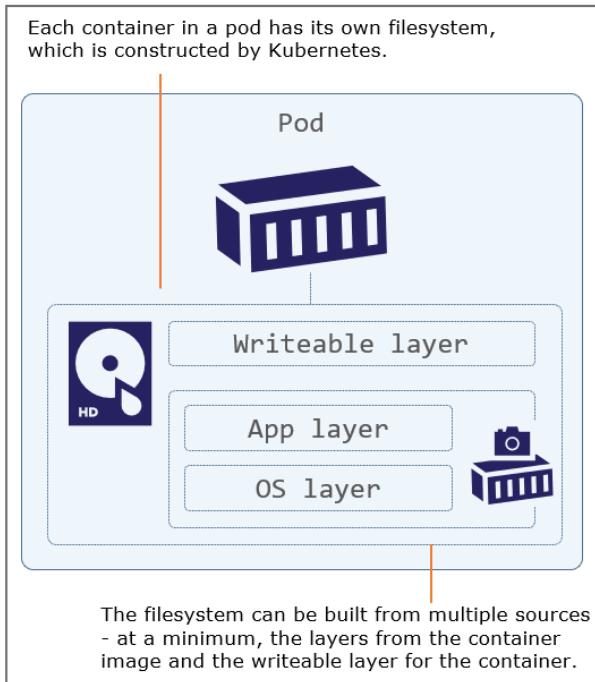


Figure 5.1 Containers don't know it but their filesystem is a virtual construct, built by Kubernetes

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 replacement 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.

Write a file inside the root directory of the pod container.

Check the container ID and then kill all the processes, which causes the container to exit.

```
PS>cd ch05
PS>
PS>kubectl apply -f sleep/sleep.yaml
deployment.apps/sleep created
PS>
PS>kubectl exec deploy/sleep -- sh -c 'echo ch05 > /file.txt; ls /*.txt'
/file.txt
PS>
PS>kubectl get pod -l app=sleep -o jsonpath='{.items[0].status.containerStatuses[0].c
ontainerID}'
docker://3cfb20aba67d25d3e84e09d9ad80513ed9c3f4325865f23c2e67eacb1f0781a4
PS>
PS>kubectl exec -it deploy/sleep -- killall5
command terminated with exit code 137
PS>
PS>kubectl get pod -l app=sleep -o jsonpath='{.items[0].status.containerStatuses[0].c
ontainerID}'
docker://e7d7b8762d2f3ed302c1e9bc0cd62ce8e18a3d817aa445723ffa44866b2f3c86
PS>
PS>kubectl exec deploy/sleep -- ls /*.txt
ls: /*.txt: No such file or directory
command terminated with exit code 1
```

Confirm that the pod has started a new container to replace the exited one.

The file you wrote in the original container doesn't exist in the replacement.

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 which are writeable. Figure 5.3 shows how you can design a pod which uses a volume to store data which persists between restarts and could even be accessible cluster-wide.

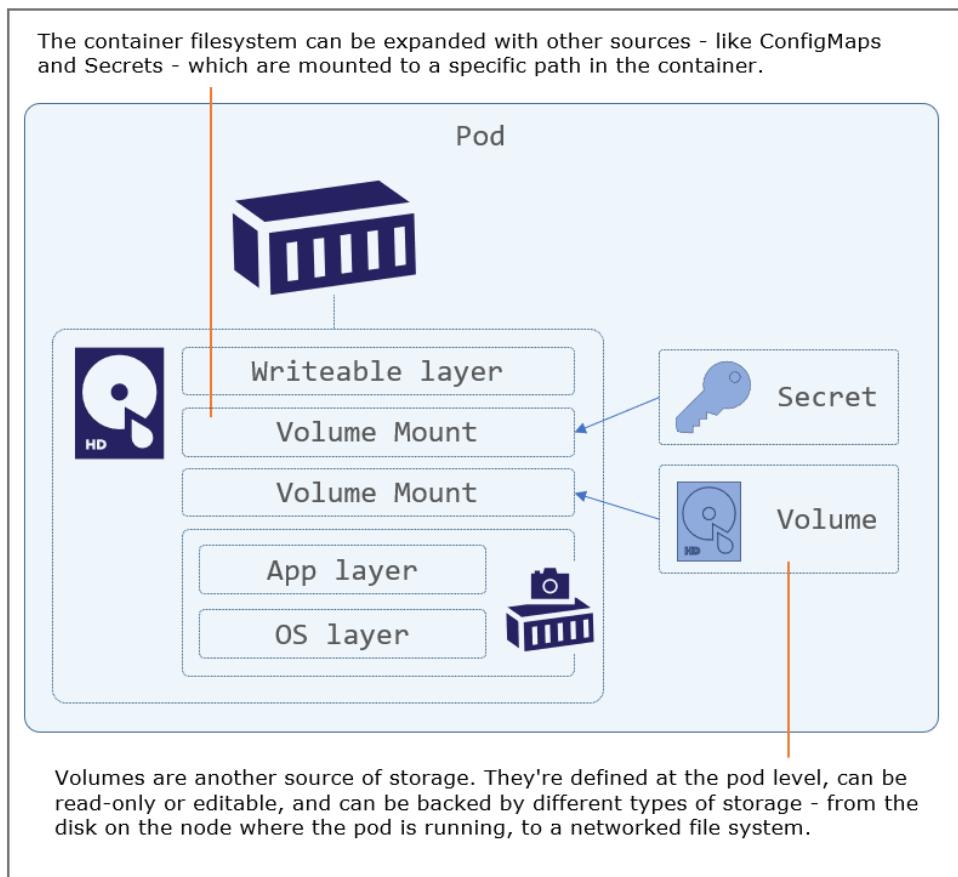


Figure 5.3 The virtual filesystem can be built from volumes which refer to external pieces of storage

We'll come to cluster-wide volumes later in the chapter, but we'll start with a much simpler volume type which 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 actually it 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 which is part of the pod.

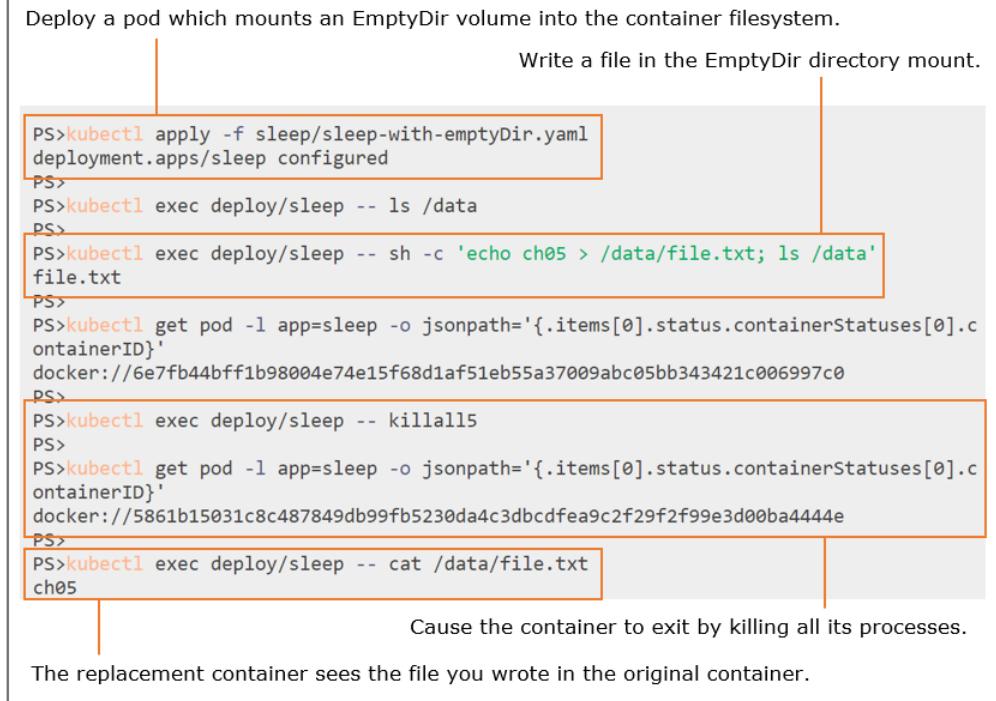


Figure 5.4 Something as basic as an empty directory is still useful because it can be shared by containers

You can use EmptyDir volumes for any applications which use the filesystem for temporary storage - maybe your app calls an API which 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.

Deploy a Pi calculating app which uses a proxy to cache responses from a web application.

Browse to the app - computing Pi to 30,000 decimal places takes over a second. Refresh the browser and the response will be much faster.

```
PS>kubectl apply -f pi/v1/
configmap/pi-proxy-configmap created
service/pi-proxy created
deployment.apps/pi-proxy created
service/pi-web created
deployment.apps/pi-web created
PS>
PS>kubectl wait --for=condition=Ready pod -l app=pi-web
pod/pi-web-5f74878cb6-bwnh9 condition met
PS>
PS>kubectl get svc pi-proxy -o jsonpath='http://{{.status.loadBalancer.ingress[0]}}:8080/?dp=30000'
http://localhost:8080/?dp=30000
PS>
PS>Pi.Web x +
PS>localhost:8080/?dp=30000
PS>
PS> π To: 30,000 d.p. in: 1,316 ms. from: pi-web-5f74878cb6-bwnh9
PS>
PS>3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825321170679821480865132823066470
938460955058223172535940812648111745028410270193852110555964462294895493038196442881097566593346128475648233787831652729
019091456485660334601485610454326648213393607260948141273245852096666631558817488152092096289254091715364367880259363600113305
PS>
PS>kubectl exec deploy/pi-proxy -- ls -l /data/nginx/cache
total 24
drwx----- 3 nginx      nginx        4096 May  4 11:19 0
drwx----- 3 nginx      nginx        4096 May  4 11:19 1
drwx----- 3 nginx      nginx        4096 May  4 11:19 4
drwx----- 4 nginx      nginx        4096 May  4 11:19 5
drwx----- 3 nginx      nginx        4096 May  4 11:19 9
drwx----- 3 nginx      nginx        4096 May  4 11:19 d
```

That's because the proxy cached the response from the first call and served it for the second call, bypassing the web app. This is the directory structure the proxy cache uses, which is an EmptyDir mount.

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.

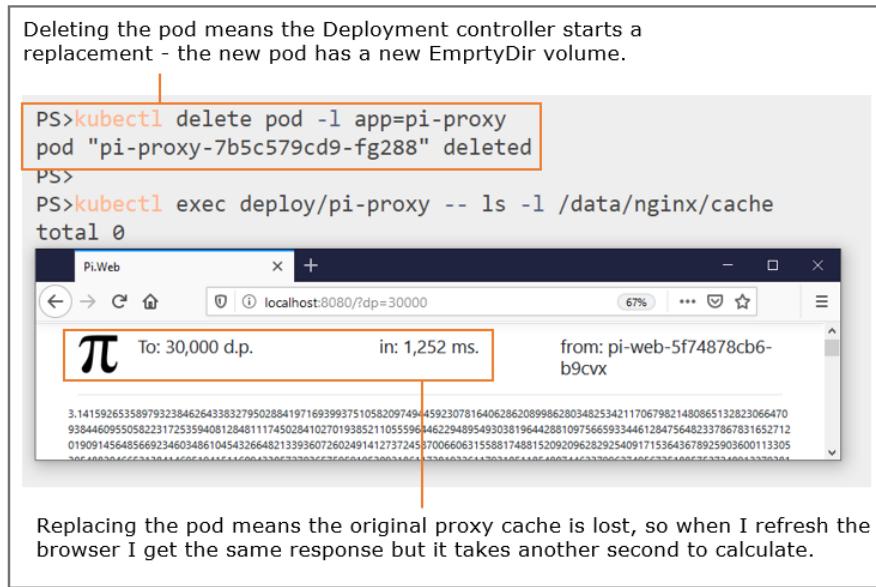


Figure 5.6 A new pod starts with a new empty directory - which is empty...

The next level of durability comes from using a volume which 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.

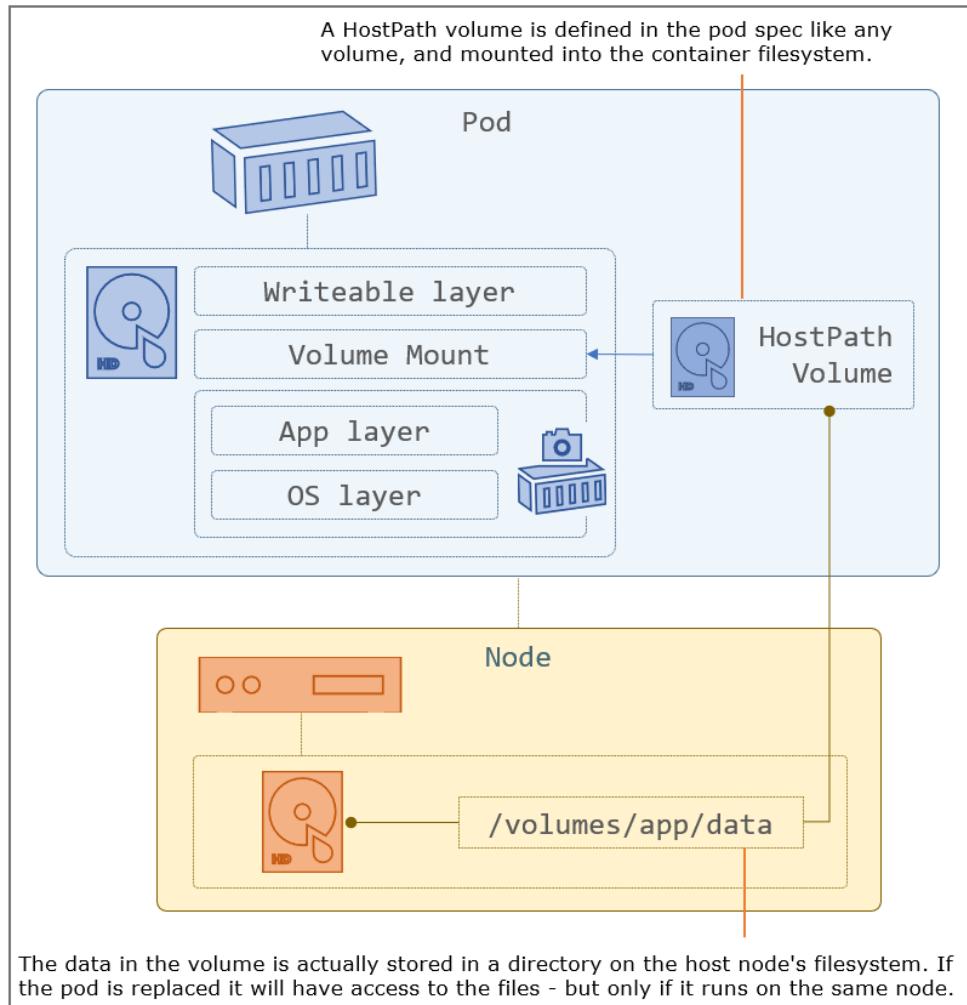


Figure 5.7 HostPath volumes keep data between pod replacements, but only if pods use the same node

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`.

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.

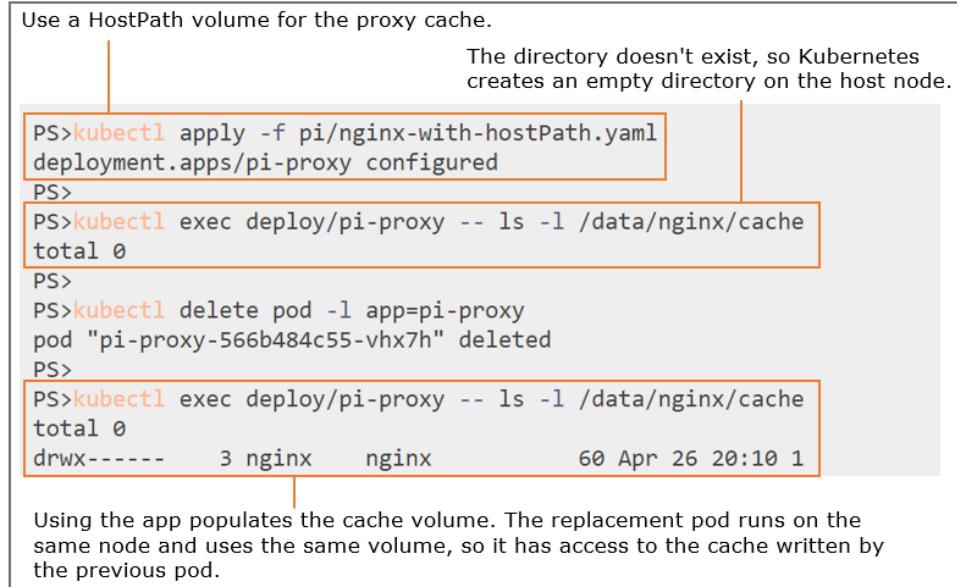


Figure 5.8 On a single-node cluster pods always run on the same node, so they can all use the HostPath

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: /
        type: Directory

```

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.

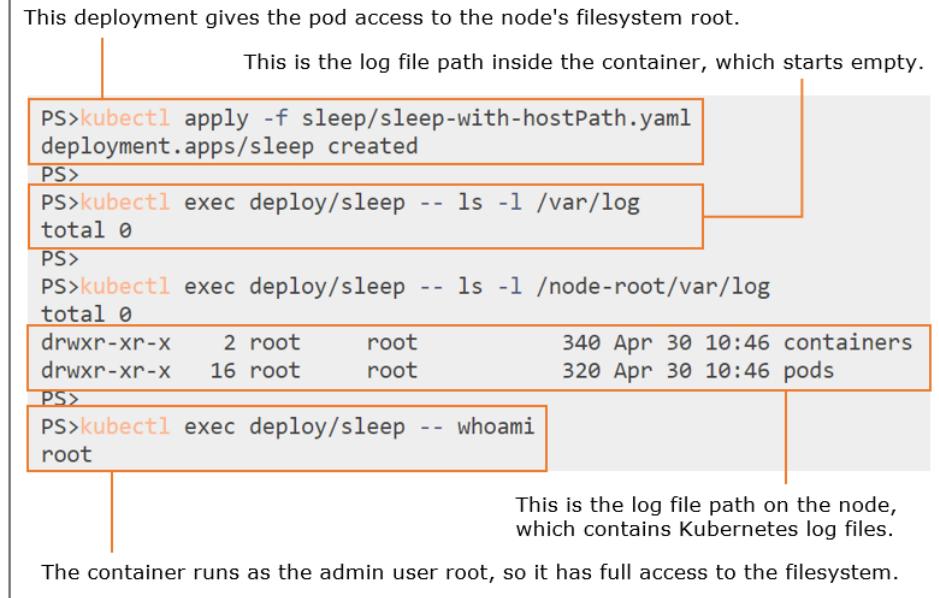


Figure 5.9 Danger! Mounting a HostPath can give you complete access to the data on the node

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 which 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 which 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.

Updates the pod using the same HostPath volume but with sub-paths defined in the volume mounts.

```
PS>kubectl apply -f sleep/sleep-with-hostPath-subPath.yaml
deployment.apps/sleep configured

PS>
PS>kubectl exec deploy/sleep -- sh -c 'ls /pod-logs | grep _pi-'
default_pi-proxy-566b484c55-d4kzq_8757483a-2d00-4729-806a-2abc2f6fe47
9
default_pi-proxy-7b5c579cd9-wpzmg_d0dae7ca-e9eb-445e-8d1f-6ce81189a7f
a
default_pi-web-5f74878cb6-b9cvx_27db66ea-523c-4524-8e8a-c7c96aa87eca
PS>
PS>kubectl exec deploy/sleep -- sh -c 'ls /container-logs | grep nginx'
pi-proxy-7b5c579cd9-wpzmg_default_nginx-e8d3a14d349f72c826625bea3fa4a
3ba16081bb71f545cafef97015232ce34ad.log
```

The pod container can still access files on the host node, but only from the specified 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 which 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 which 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.

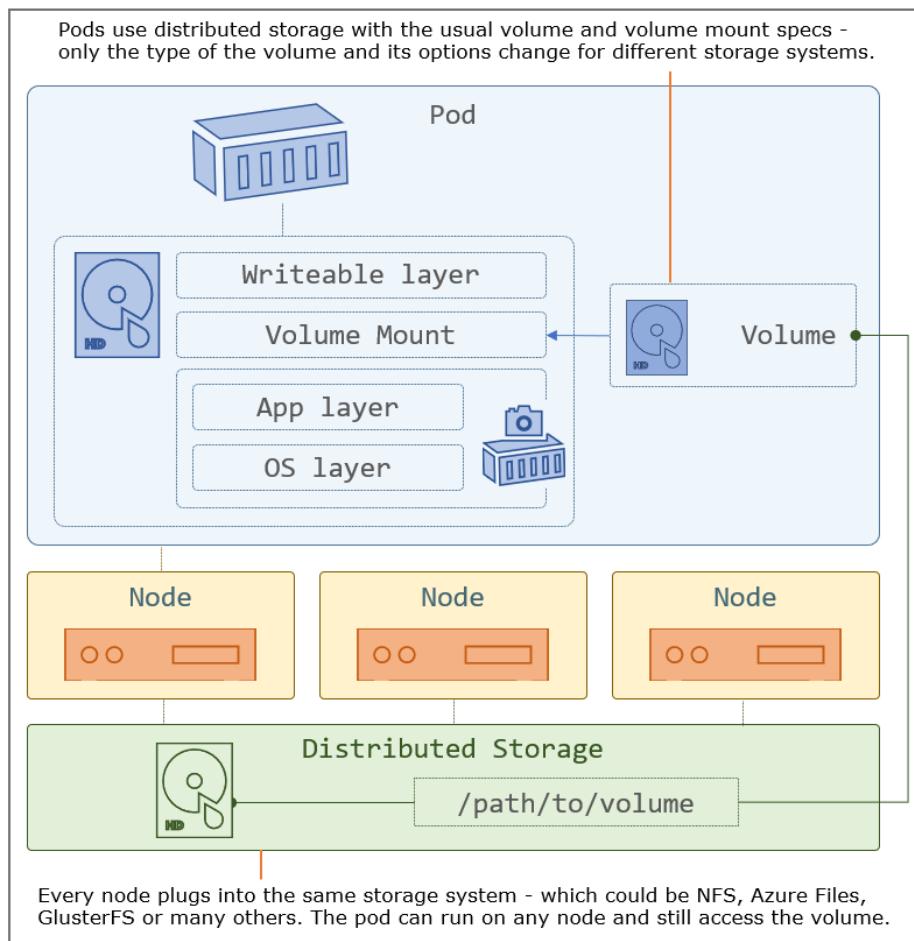


Figure 5.11 Distributed storage gives your pod access to data from any node, but needs platform support

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:
    - ReadWriteOnce    # how the volume can be accessed by pods
                           # it can only be used by one pod
  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 which 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 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.

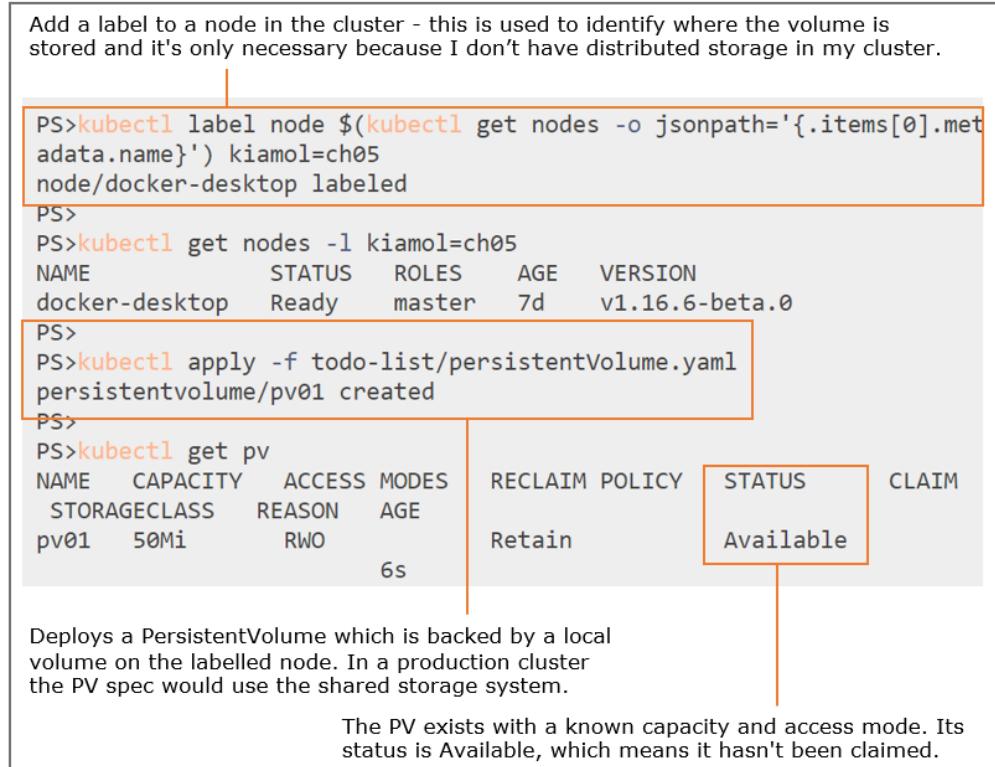


Figure 5.12 If you don't have distributed storage you can cheat by pinning a PV to a local volume

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 which 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.

Deploy a PVC which will claim the PV we have available in the cluster.

```
PS>kubectl apply -f todo-list/postgres-persistentVolumeClaim.yaml
persistentvolumeclaim/postgres-pvc created
PS>
PS>kubectl get pvc
NAME           STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLA
SS  AGE
postgres-pvc   Bound    pv01     50Mi       RWO
               6s
PS>
PS>kubectl get pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
                  STORAGECLASS   REASON   AGE
pv01     50Mi       RWO
stgres-pvc
               Retain
               31s
               Bound
               default/po
```

The PVC is bound, which means it has claimed the PV. The capacity is 50MB, which is the PV capacity - the PVC only requested 40MB.

The PV shows it has been claimed - the access mode and storage available in the PV matched the request in the PVC.

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 which 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.

Deploys a PVC which requests 100MB of storage. There are no available PVs in the cluster which can satisfy the claim.

```
PS>kubectl apply -f todo-list/postgres-persistentVolumeClaim-too-big.yaml
persistentvolumeclaim/postgres-pvc-toobig created
PS>
PS>kubectl get pvc
NAME           STATUS    VOLUME   CAPACITY   ACCESS MODES   ST
ORAGECLASS   AGE
postgres-pvc   Bound     pv01     50Mi       Rwo
               13m
postgres-pvc-toobig Pending   7s
The original PVC is bound to the only PV in the system, so the new PVC cannot be bound - it will stay in the pending state until a PV becomes available.
```

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 which 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 writable 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:
        claimName: postgres-pvc # volume uses a 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.

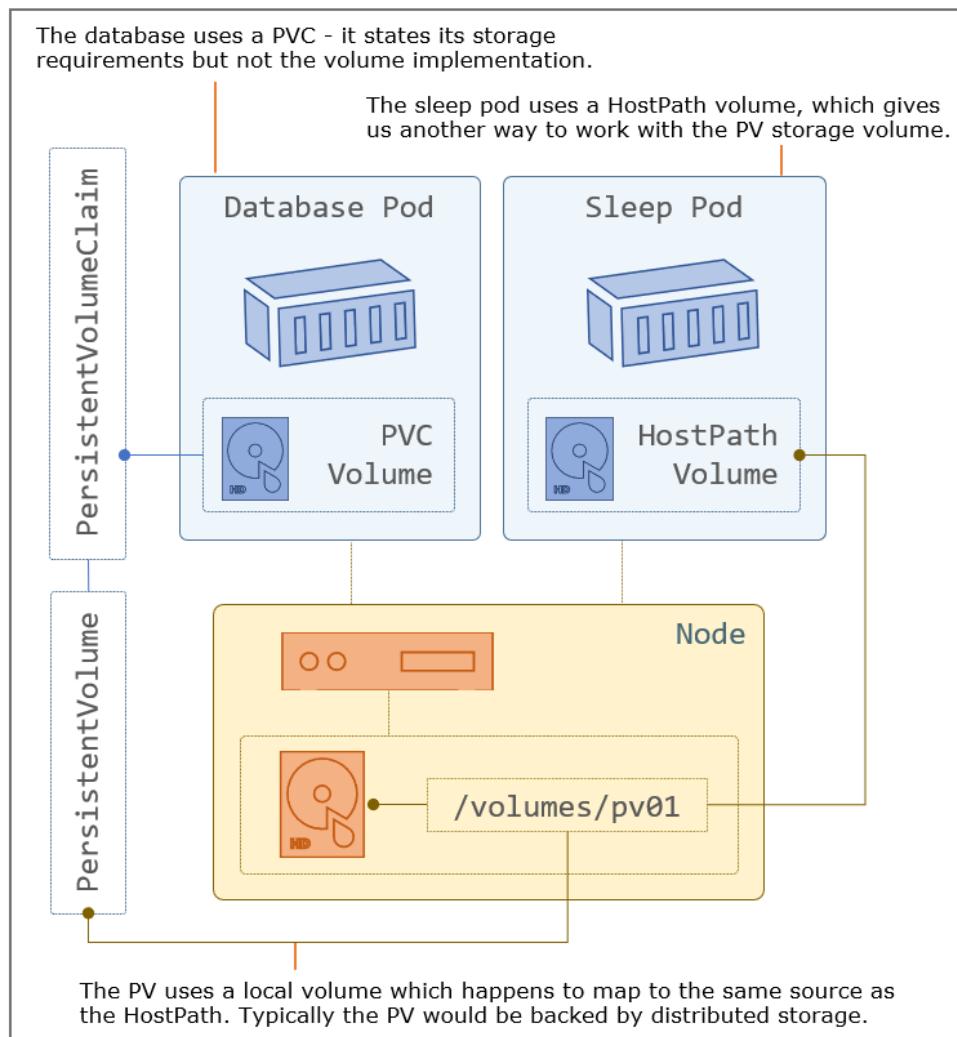


Figure 5.16 Just a little bit complicated - mapping a PV and a HostPath to the same storage location

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 which we can use to look at the Postgres files.

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.

Deploy a database pod which stores the data files in a volume backed by the PVC.

After a few seconds the database is up and running.

```
PS>kubectl apply -f todo-list/postgres/
secret/todo-db-secret created
service/todo-db created
deployment.apps/todo-db created
PS>
PS>sleep 30
PS>
PS>kubectl logs -l app=todo-db --tail 1
2020-05-01 11:17:56.871 UTC [1] LOG:  database system is ready to accept connections
PS>
PS>kubectl exec deploy/sleep -- sh -c 'ls -l /node-root/volumes/pv01
| grep wal'
drwx----- 3 70 70 80 May 1 11:16 pg_wal
```

The sleep pod has access to the same path on the host, where we can see the database files have been created.

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.

Deploy the to-do app which uses the database pod.

Delete the database pod - the replacement uses the same PVC and the same PV, so the original data is still there.

My to-do item lives in a local volume on my node, but to use distributed storage all I need to change is the PV spec.

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; Kind 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.

Create a PVC with no storage class specified, so the PV will be dynamically created using the default storage class for the cluster.

On Docker Desktop the default is for a HostPath volume.

```
PS>kubectl apply -f todo-list/postgres-persistentVolumeClaim-dynamic.yaml
persistentvolumeclaim/postgres-pvc-dynamic created
PS>
PS>kubectl get pvc
NAME           STATUS   VOLUME          CAPA
CITY  ACCESS MODES  STORAGECLASS AGE
postgres-pvc   Bound    pv01           50Mi
              RWO
postgres-pvc-dynamic   Bound    pvc-1709b65c-47ea-4187-b655-a9c54f213872 100M
i      RWO          hostpath        6s
postgres-pvc-toobig   Pending
                                         3h59m
PS>
PS>kubectl get pv
NAME          CAPACITY   ACCESS MODES  RECLAIM POL
ICY  STATUS   CLAIM          STORAGECLASS  REASON AGE
pv01         Bound      default/postgres-pvc   50Mi       RWO     Retain 4h13m
pvc-1709b65c-47ea-4187-b655-a9c54f213872 100Mi       RWO     Delete 35s
          Bound      default/postgres-pvc-dynamic  hostpath
PS>
PS>kubectl delete pvc postgres-pvc-dynamic
persistentvolumeclaim "postgres-pvc-dynamic" deleted
PS>
PS>kubectl get pv
NAME  CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS   CLAIM
          STORAGECLASS  REASON AGE
pv01  50Mi      RWO          Retain        Bound    default/postgres-pvc

```

The PV was provisioned by Kubernetes on-demand, with the access mode and storage capacity requested in the PVC.

Docker Desktop is configured to delete dynamically-provisioned PV when the PVC is deleted.

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 which 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 afterwards.

```
# 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` which has the same setup as the default storage class. My output from Docker Desktop is in figure 5.20.

Storage classes have a provisioner, which is the component that integrates your cluster with the storage systems it can use.

This script gets the details of the default storage class and creates a clone called `kiamol`.

```
PS>kubectl get storageclass
NAME           PROVISIONER      AGE
hostpath (default)  docker.io/hostpath  7d6h
PS>
PS>Set-ExecutionPolicy Bypass -Scope Process -Force; ./cloneDefaultStorageClass.ps1
configmap/clone-script created
pod/clone-sc created
pod/clone-sc condition met
storageclass.storage.k8s.io/kiamol created
configmap "clone-script" deleted
pod "clone-sc" deleted
PS>
PS>kubectl get sc
NAME           PROVISIONER      AGE
hostpath (default)  docker.io/hostpath  7d6h
kiamol          docker.io/hostpath  19s
```

Here's the new storage class. In a production cluster in the cloud you might already have multiple storage classes with different capabilities.

Figure 5.20 Cloning the default storage class to create a custom class you can use in PVC specs

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.

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.

Updates the Postgres database pod to use a new PVC that specifies the kiamol storage class.
 That uses the same provisioner as the default to create and bind a PV on-demand.

```

PS>kubectl apply -f storageClass/postgres-persistentVolumeClaim-storageClass.yaml
1
persistentvolumeclaim/postgres-pvc-kiamol created
PS>
PS>kubectl apply -f storageClass/todo-db.yaml
deployment.apps/todo-db configured
PS>
PS>kubectl get pvc
NAME           STATUS    VOLUME          CAPAC
ITY  ACCESS MODES  STORAGECLASS   AGE
postgres-pvc   Bound     pv01            50Mi
              RWO
postgres-pvc-kiamol Bound     pvc-33a99300-4267-49f8-8258-2f5df2c83e09 100Mi
              RWO      kiamol         12s
postgres-pvc-toobig Pending
              4h42m
PS>
PS>kubectl get pv
NAME           CAPACITY  ACCESS MODES  RECLAIM POL
ICY  STATUS    CLAIM          STORAGECLASS  REASON  AGE
pv01           50Mi      RWO          Retain
              Bound     default/postgres-pvc
pvc-33a99300-4267-49f8-8258-2f5df2c83e09 100Mi      RWO      Delete
              Bound     default/postgres-pvc-kiamol   kiamol      19s
PS>
PS>kubectl get pods -l app=todo-db
NAME           READY   STATUS    RESTARTS  AGE
todo-db-5dbb9bc854-wz6hz 1/1     Running   0          21s
  
```

This is a new pod using a new PVC, so the volume starts of empty and the database will initialize with new files.

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 which 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 which 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.

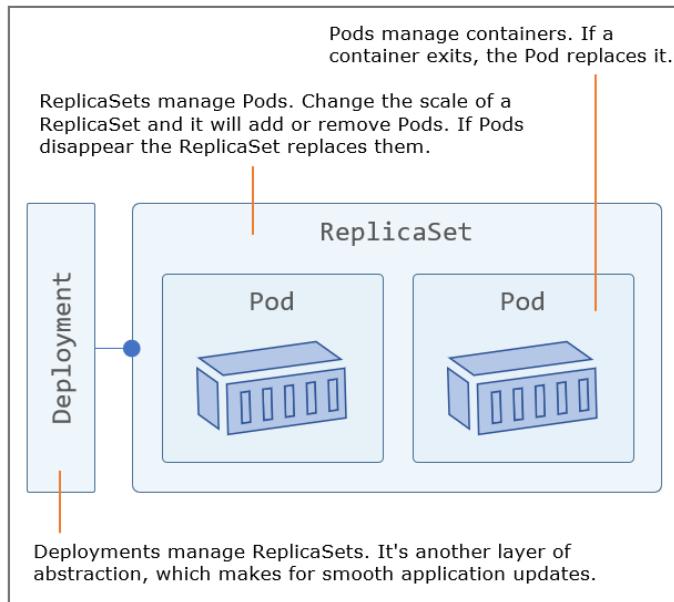


Figure 6.1 Every software problem can be solved by adding another layer of abstraction

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 for the ReplicaSet to find its Pods
  selector:
    matchLabels:
      app: whoami-web
  template:               # the usual Pod spec follows
```

The only things which 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 which 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 ends with a list of events, where the ReplicaSet writes activity logs on how it created Pods.

The ReplicaSet shows the desired and current status for the number of Pods it manages.

```

PS>cd ch06
PS>
PS>kubectl apply -f whoami/
service/whoami-web created
replicaset.apps/whoami-web created
PS>
PS>kubectl get replicaset whoami-web
NAME      DESIRED   CURRENT   READY   AGE
whoami-web 1         1         1       6s
PS>
PS>curl $(kubectl get svc whoami-web -o jsonpath='http://.status.loadBalancer.ingress[0].*:8088')
"I'm whoami-web-tv7xr running on Linux 4.19.76-linuxkit #1 SMP Fri Apr 3 15:53:26 UTC 2020"
PS>
PS>kubectl delete pods -l app=whoami-web
pod "whoami-web-tv7xr" deleted
PS>
PS>curl $(kubectl get svc whoami-web -o jsonpath='http://.status.loadBalancer.ingress[0].*:8088')
"I'm whoami-web-tmb95 running on Linux 4.19.76-linuxkit #1 SMP Fri Apr 3 15:53:26 UTC 2020"

```

This app just returns the hostname in an HTTP response - which is the Pod name in Kubernetes.

The ReplicaSet is a controller - delete the Pod and it will create a replacement.

The new Pod is online, serving its hostname.

Figure 6.2 Working with a ReplicaSet is just like working with a Deployment - it creates and manages 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?

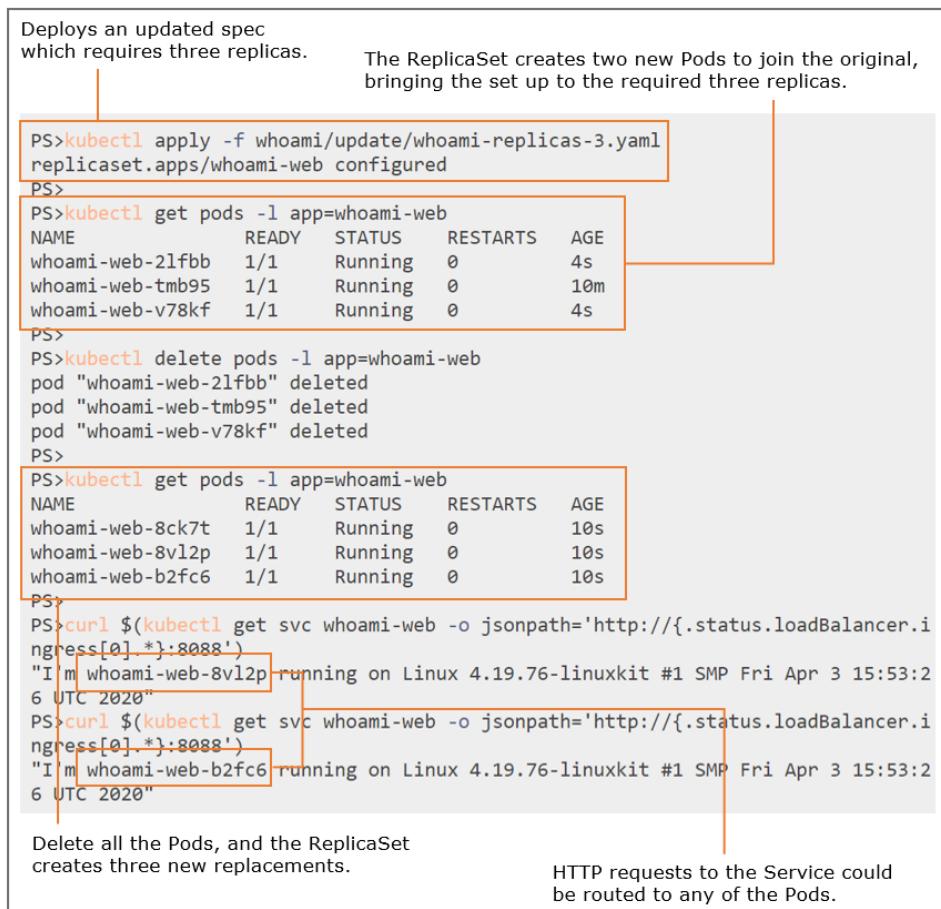


Figure 6.3 Scaling ReplicaSets is fast, and at scale a Service can distribute requests to many Pods

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 which

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.

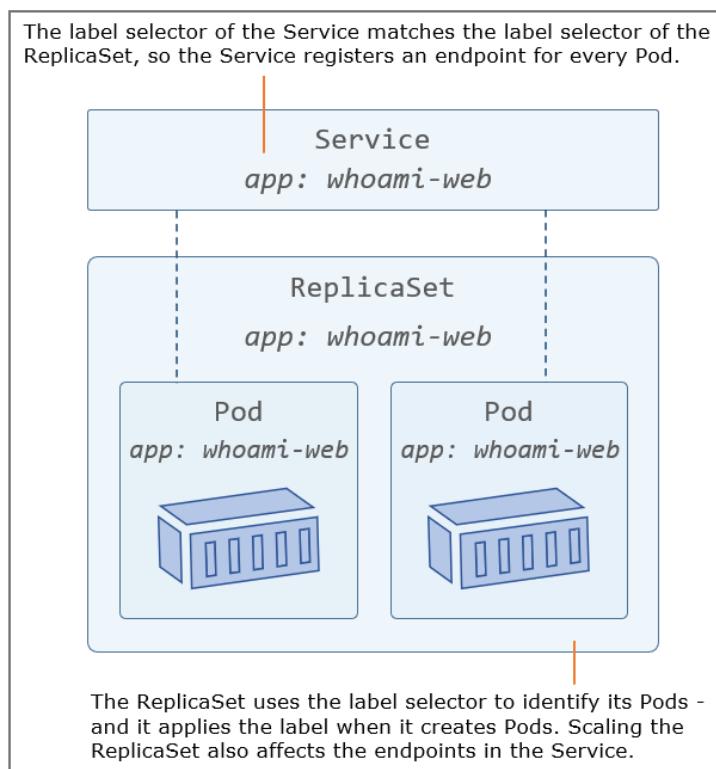


Figure 6.4 A Service with the same label selector as a ReplicaSet will use all its Pods

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.

```
# 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.

Run a sleep Pod so we can see how the Service works inside the cluster.

```
PS>kubectl apply -f sleep.yaml
deployment.apps/sleep created
```

In a DNS lookup, the IP address is the Service ClusterIP as we'd expect.

```
PS>kubectl get svc whoami-web
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
whoami-web   LoadBalancer   10.110.136.125   localhost      8088:30077/TCP   35m
PS>
PS>kubectl exec deploy/sleep -- sh -c 'nslookup whoami-web | grep "^[^"]"'
Server:    10.96.0.10
Address:   10.96.0.10:53
Name:   whoami-web.default.svc.cluster.local
Address: 10.110.136.125
```

```
PS>kubectl exec deploy/sleep -- sh -c 'for i in 1 2 3; do curl -w \\n -s http://
whoami-web:8088; done;'
```

"I'm whoami-web-b2fc6 6 UTC 2020"	running on Linux 4.19.76-linuxkit #1 SMP Fri Apr 3 15:53:2
"I'm whoami-web-8vl2p 6 UTC 2020"	running on Linux 4.19.76-linuxkit #1 SMP Fri Apr 3 15:53:2
"I'm whoami-web-8ck7t 6 UTC 2020"	running on Linux 4.19.76-linuxkit #1 SMP Fri Apr 3 15:53:2

This fancy script just runs a loop inside the sleep container, making three HTTP requests.

The responses all come from different Pods - the Service load-balances between them.

Figure 6.5 The world inside the cluster - Pod to Pod networking also benefits from Service load balancing

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 across many Pods, and the same networking layer which 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 which run in small, lean containers - and that's why applications built for Kubernetes use a distributed architecture, breaking functionality down across many pieces which 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 any more - 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.

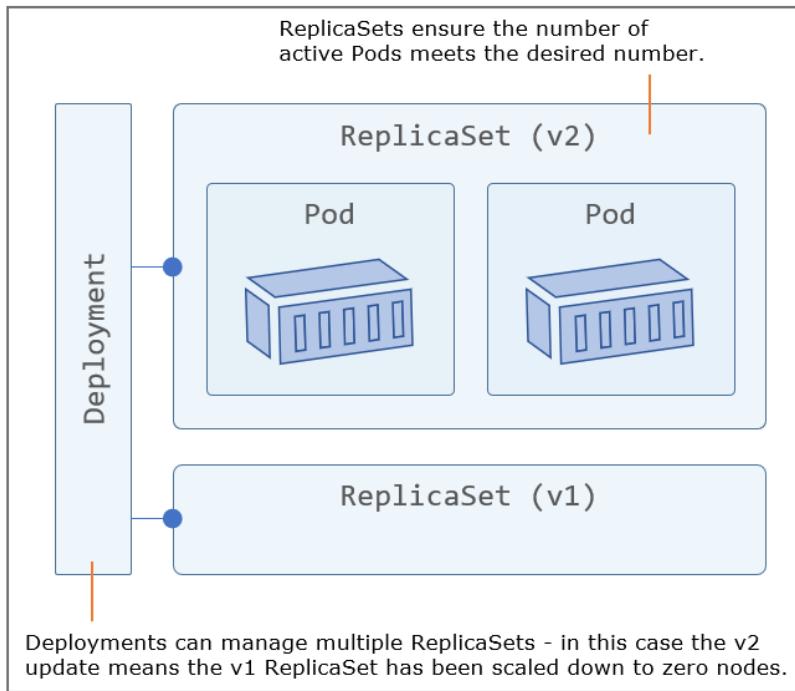


Figure 6.6 Zero is a valid number of desired replicas - Deployments scale down old ReplicaSets to zero

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.

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.

This Deployment creates a ReplicaSet with two replicas.

Updating the Deployment with an increased replica count just scales the existing ReplicaSet.

```

PS>kubectl apply -f pi/web/
service/pi-web created
deployment.apps/pi-web created
PS>
PS>kubectl get rs -l app=pi-web
NAME          DESIRED  CURRENT  READY   AGE
pi-web-5f74878cb6  2        2        2      7s
PS>
PS>kubectl apply -f pi/web/update/web-replicas-3.yaml
deployment.apps/pi-web configured
PS>
PS>kubectl get rs -l app=pi-web
NAME          DESIRED  CURRENT  READY   AGE
pi-web-5f74878cb6  3        3        2      22s
PS>
PS>kubectl apply -f pi/web/update/web-logging-level.yaml
deployment.apps/pi-web configured
PS>
PS>kubectl get rs -l app=pi-web
NAME          DESIRED  CURRENT  READY   AGE
pi-web-5f74878cb6  1        1        1      52s
pi-web-75c758cc59  3        3        2      6s

```

The next update alters the Pod spec, so the original ReplicaSet's Pods are no longer a match. The Deployment creates a new ReplicaSet and scales the old one down as the new Pods become ready.

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.

The `scale` command does the same thing as editing the number of replicas in the Deployment spec, but the change needs to be made in the YAML too.

Otherwise the next update from the spec will return the replica count to the original value.

```
PS>kubectl scale --replicas=4 deploy/pi-web
deployment.apps/pi-web scaled
PS>
PS>kubectl get rs -l app=pi-web
NAME          DESIRED   CURRENT   READY   AGE
pi-web-5f74878cb6   0         0         0      12m
pi-web-75c758cc59   4         4         4      11m
PS>
PS>kubectl apply -f pi/web/update/web-replicas-3.yaml
deployment.apps/pi-web configured
PS>
```

```
PS>kubectl get rs -l app=pi-web
NAME          DESIRED   CURRENT   READY   AGE
pi-web-5f74878cb6   3         3         3      12m
pi-web-75c758cc59   0         0         0      12m
PS>
```

```
PS>kubectl get pods -l app=pi-web
NAME          READY   STATUS    RESTARTS   AGE
pi-web-5f74878cb6-jf7m9   1/1     Running   0          18s
pi-web-5f74878cb6-lvcsq   1/1     Running   0          21s
pi-web-5f74878cb6-x2hpt   1/1     Running   0          16s
```

This update actually reverts the Deployment to the previous Pod spec, so it effects the change by scaling up the original ReplicaSet.

The ReplicaSet had been scaled down to zero, so now it creates three new Pods.

Figure 6.8 Deployments know the spec for their ReplicaSets and can rollback by scaling an old ReplicaSet

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 re-used 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 which 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
```

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.

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.					
<pre>PS>kubectl get rs -l app=pi-web --show-labels</pre>					
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>					
<pre>PS>kubectl get po -l app=pi-web --show-labels</pre>					
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

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 make use of 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.

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.

There are two layers of load-balancing in this app
- into the proxy Pods and into the web Pods.

```
PS>kubectl apply -f pi/proxy/
configmap/pi-proxy-configmap created
service/pi-proxy created
deployment.apps/pi-proxy created
PS>
PS>kubectl get svc whoami-web -o jsonpath='http://{.status.loadBalancer.ingress[0].*:8080/?dp=10000'
http://localhost:8080/?dp=10000
PS>
PS>
```

The response contains the name of the Pod which originally handled the request.

And the HTTP headers also include the name of the proxy Pod which sent the response. This was a cache hit, which means the proxy already had the response cached and didn't call a web Pod.

Figure 6.10 The Pi responses include the name of the Pod which sent them, so you can see the load-balancing at work

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.

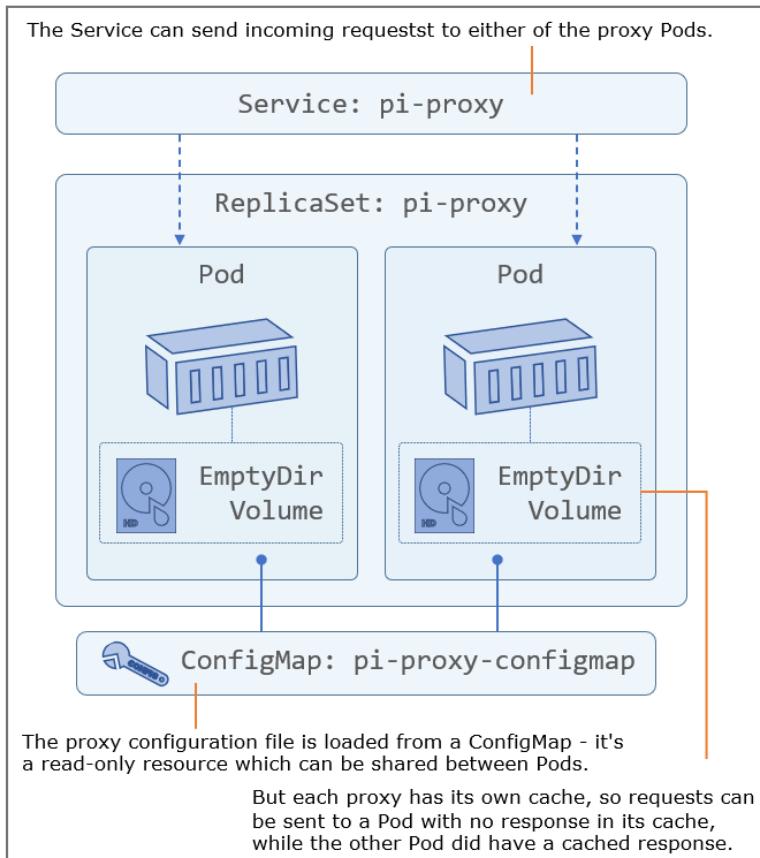


Figure 6.11 Running Pods at scale - some types of volume can be shared, others are unique to the Pod

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 which 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 which 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.

```
# 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
```

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.

This update uses a HostPath volume for the proxy cache. I'm running a single-node cluster so every Pod can share the cache files on the node.

It's a changed Pod spec so the Deployment creates a new ReplicaSet and it creates new Pods.

```
PS>kubectl apply -f pi/proxy/update/nginx-hostPath.yaml
deployment.apps/pi-proxy configured
```

NAME	READY	STATUS	RESTARTS	AGE
pi-proxy-6858657f9c-5kkkj	1/1	Running	0	4s
pi-proxy-6858657f9c-9mcvh	1/1	Running	0	6s
pi-proxy-6858657f9c-r9p8p	1/1	Running	0	5s
pi-proxy-7b5c579cd9-5jhhx	0/1	Terminating	0	26m
pi-proxy-7b5c579cd9-t6bgs	0/1	Terminating	0	6s

```
PS>
PS>kubectl logs -l app=pi-proxy --tail 1
192.168.65.3 - - [06/May/2020:14:25:00 +0000] "GET /img/pi-large.png HTTP/1.1" 3
04 0 "http://localhost:8080/?dp=5003" "Mozilla/5.0 (Windows NT 10.0; Win64; x64;
rv:75.0) Gecko/20100101 Firefox/75.0"
192.168.65.3 - - [06/May/2020:14:25:05 +0000] "GET /?dp=2003 HTTP/1.1" 200 2198
"- " "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:75.0) Gecko/20100101 Firefox/7
5.0"
192.168.65.3 - - [06/May/2020:14:25:00 +0000] "GET /lib/bootstrap/dist/js/bootst
rap.bundle.min.js HTTP/1.1" 304 0 "http://localhost:8080/?dp=5003" "Mozilla/5.0
(Windows NT 10.0; Win64; x64; rv:75.0) Gecko/20100101 Firefox/75.0"
```

You can fetch logs using a label selector - Kubectl returns logs from all matching Pods. Here we're just seeing the latest log entry from each Pod - they're all handling traffic.

Figure 6.12 At scale you can see all the Pod logs with Kubectl, using a label selector

For most stateful applications, this wouldn't work. Apps which 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 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
pi-proxy   10.1.2.46:80,10.1.2.47:80,10.1.2.48:80 + 1 more...
          AGE
          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
          AGE
          55m
          55m
          55m
          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 which 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   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 which 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 which 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.

The new DaemonSet spec includes a node selector which no nodes in my cluster match - so the set is scaled down to zero.

```
PS>kubectl apply -f pi/proxy/daemonset/nginx-ds-nodeSelector.yaml
daemonset.apps/pi-proxy configured
PS>
PS>kubectl get ds pi-proxy
NAME      DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR
AGE
pi-proxy   0          0          0       0           0           kiamol=ch06
3/m
PS>
PS>kubectl get po -l app=pi-proxy
No resources found in default namespace.
PS>
PS>kubectl label node $(kubectl get nodes -o jsonpath='{.items[0].metadata.name}')
'') kiamol=ch06 --overwrite
node/docker-desktop labeled
PS>
PS>kubectl get ds pi-proxy
NAME      DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR
AGE
pi-proxy   1          1          1       1           1           kiamol=ch06
```

When I label a node there's now one match for the node selector and the DaemonSet scales up to one.

Figure 6.15 DaemonSets actually watch nodes and their labels, as well as current Pod status

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 which 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.

```
# 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.

Cascading deletes are the default - but you can specify not to cascade, so deleting a controller won't delete its Pods.

The labels on the Pod still match the DaemonSet's selector, so when the DS is recreated it adopts the existing Pod.

```
PS>kubectl delete ds pi-proxy --cascade=false
daemonset.apps "pi-proxy" deleted
PS>
PS>kubectl get po -l app=pi-proxy
NAME      READY  STATUS   RESTARTS  AGE
pi-proxy-ddt1k  1/1    Running   0          90s
PS>
PS>kubectl apply -f pi/proxy/daemonset/nginx-ds-nodeSelector.yaml
daemonset.apps/pi-proxy created
PS>
PS>kubectl get ds pi-proxy
NAME      DESIRED  CURRENT  READY  UP-TO-DATE  AVAILABLE  NODE SELECTOR
AGE
pi-proxy  1         1         1     1           1           kiamol=ch06
5s
PS>
PS>kubectl get po -l app=pi-proxy
NAME      READY  STATUS   RESTARTS  AGE
pi-proxy-ddt1k  1/1    Running   0          110s
PS>
PS>kubectl delete ds pi-proxy
daemonset.apps "pi-proxy" deleted
PS>
PS>kubectl get po -l app=pi-proxy
NAME      READY  STATUS   RESTARTS  AGE
pi-proxy-ddt1k  0/1    Terminating   0          119s
```

Adopting the Pod means the DaemonSet owns it again, so when the DS is deleted with the default cascade behavior, the Pod gets deleted too.

Figure 6.16 Orphaned Pods have lost their controller so they're not part of a highly-available set any more

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 which 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 which 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].name}',OWNER_KI
ND:'{.metadata.ownerReferences[0].kind}'

# and which objects own the ReplicaSets:
kubectl get rs -o custom-
columns=NAME:'{.metadata.name}',OWNER:'{.metadata.ownerReferences[0].name}',OWNER_KI
ND:'{.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.

NAME	OWNER	OWNER_KIND
pi-proxy-5x2cp	pi-proxy	DaemonSet
pi-web-5f74878cb6-jf7m9	pi-web-5f74878cb6	ReplicaSet
pi-web-5f74878cb6-lvcsg	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

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.

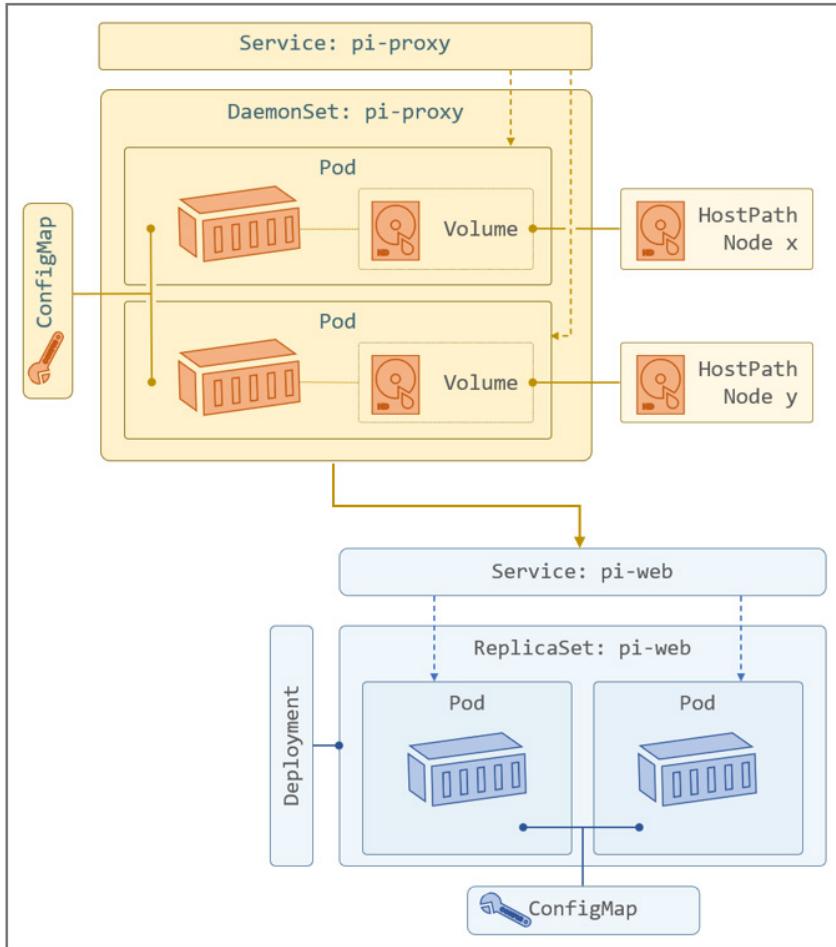


Figure 6.18 The Pi application. No annotations necessary - the diagram should be crystal clear

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 which 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>

7

Extending applications with multi-container Pods

We met Pods in chapter 2, when you learned that you can run many containers in one Pod, but you didn't actually do it. In this chapter you're going to see how it works, and understand the patterns it enables. This is the first of the more advanced topics in this part of the book, but it's not a complicated subject - it just helps to have all the background knowledge from the previous chapters. Conceptually its quite simple: one Pod runs many containers, which is typically your app container plus some helper containers. It's what you can do with those helpers which makes this such an interesting feature.

Containers in a Pod share the same virtual environment, so when one container takes an action, other containers can see it and react to it - and they can even modify the intended action without the original container knowing. That lets you model your application so that the app container is very simple — it just focuses on its work and it has helpers which take care of integrating the app with other components and with the Kubernetes platform. It's a great way to add a consistent management API to all your apps, whether new or legacy.

7.1 How containers communicate in a Pod

The Pod is a virtual environment which creates a shared networking and filesystem space for one or more containers. The containers are isolated units - they have their own processes and environment variables, and they can use different images with different technology stacks. But the Pod is a single unit, so when it is allocated to run on a node, all the Pod containers run on the same node - you can have one container running Python and another running Java, but you can't have some Linux and some Windows containers in the same Pod (yet), because Linux containers need to run on a Linux node and Windows containers on a Windows node.

Containers in a Pod share the network, so each container has the same IP address, which is the IP address of the Pod. Multiple containers can receive external traffic, but they need to listen on different ports, and containers within the Pod can communicate using the localhost address. Each container has its own filesystem, but it can mount volumes from the Pod, so containers can exchange information by sharing the same mounts. Figure 7.1 shows the layout of a Pod with two containers.

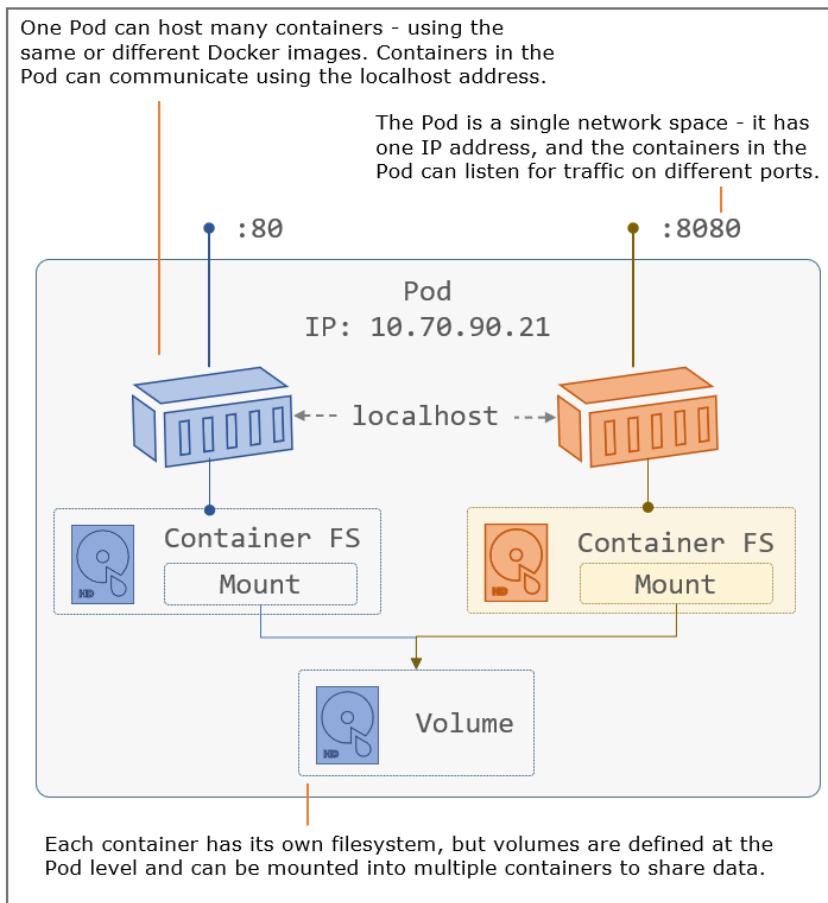


Figure 7.1 The Pod is a shared network and storage environment for many containers

That's all the theory we need for now, and as we go through the chapter you'll be surprised at some of the smart things you can do just with shared networking and disk. We'll start with some simple exercises in this section to explore the Pod environment. Listing 7.1 shows the multi-container Pod spec for a Deployment - there are two containers defined, which happen

to use the same image, and they both mount an EmptyDir volume which is defined in the Pod.

Listing 7.1 - sleep-with-file-reader.yaml, a simple multi-container Pod spec

```
spec:
  containers:
    - name: sleep
      image: kiamol/ch03-sleep
      volumeMounts:
        - name: data
          mountPath: /data-rw      # mounts a volume as writeable
    - name: file-reader
      image: kiamol/ch03-sleep
      volumeMounts:
        - name: data
          mountPath: /data-ro    # containers need different names
          readOnly: true         # but can use the same or different images
  volumes:
    - name: data             # volumes can be mounted by many containers
      emptyDir: {}
```

This is a single Pod spec which runs two containers. When you deploy it you'll see that there are some differences to how you work with multi-container Pods.

TRY IT NOW Deploy listing 7.1 and run a Pod with two containers.

```
# switch to the chapter folder:
cd ch07

# deploy the Pod spec:
kubectl apply -f sleep/sleep-with-file-reader.yaml

# get the detailed Pod information:
kubectl get pod -l app=sleep -o wide

# and show the container names:
kubectl get pod -l app=sleep -o
jsonpath='{.items[0].status.containerStatuses[*].name}'

# check the Pod logs - this will fail:
kubectl logs -l app=sleep
```

My output is in figure 7.2 - it shows the Pod has two containers with a single IP address, which both run on the same node. You can see the details of the Pod as a single unit, but you can't print the logs at a Pod level; you need to specify a container to fetch the logs from.

The ready column shows this Pod should have two containers running, and they are both up.

The wide output shows additional information - confirming that the Pod has a single IP address and runs on a single node.

```

PS>cd ch07
PS>
PS>kubectl apply -f sleep/sleep-with-file-reader.yaml
deployment.apps/sleep created
PS>
PS>kubectl get pod -l app=sleep -o wide
NAME           NOMINATED-NODE   READY   STATUS    RESTARTS   AGE   IP          NODE
sleep-5b6d599954-wnzmm   <none>     2/2     Running   0          7s   10.1.2.223   docker-
PS>
PS>kubectl get pod -l app=sleep -o jsonpath='{.items[0].status.containerStatuses[*].name}'
file-reader sleep
PS>
PS>kubectl logs -l app=sleep
Error from server (BadRequest): a container name must be specified for pod sleep-5b6d599954-wnzmm, choose one of: [sleep file-reader]

```

These are the container names defined in the Pod spec.

You can't print all the logs for a multi-container Pod, you need to specify the name of one container.

Figure 7.2 You always work with a Pod as a single unit - except when you need to specify a container

Both the containers in that exercise use the sleep image, so they're not doing anything - but the containers keep running and the Pod stays available to work with. The containers both mount the EmptyDir volume from the Pod, so that's a shared part of the filesystem and you can use it in both containers.

TRY IT NOW One container mounts the volume as read-write and the other one as read-only. You can write files in one container and read them in the other.

```

# write a file to the shared volume using one container:
kubectl exec deploy/sleep -c sleep -- sh -c 'echo ${HOSTNAME} > /data-rw/hostname.txt'

# read the file out using the same container:
kubectl exec deploy/sleep -c sleep -- cat /data-rw/hostname.txt

# and using the other container:
kubectl exec deploy/sleep -c file-reader -- cat /data-ro/hostname.txt
# try to add to the file in the read-only container - this will fail:
kubectl exec deploy/sleep -c file-reader -- sh -c 'echo more >> /data-ro/hostname.txt'

```

You'll see when you run this that the first container can write data into the shared volume, and the second container can read it but it can't write data itself. That's because the volume mount is defined as read-only for the second container in this Pod spec - it's not a generic Pod limitation; mounts can be defined as writable for multiple containers if you need that. Figure 7.3 shows my output.

```

Commands which need to target one container in a
Pod can use the -c flag to specify which container.

The container called sleep has read
and write access to the volume mount.

PS>kubectl exec deploy/sleep -c sleep -- sh -c 'echo ${HOSTNAME} > /data-rw/hostname.txt'
PS>kubectl exec deploy/sleep -c sleep -- cat /data-rw/hostname.txt
sleep-5b6d599954-wnzmm

PS>kubectl exec deploy/sleep -c file-reader -- cat /data-ro/hostname.txt
sleep-5b6d599954-wnzmm

PS>kubectl exec deploy/sleep -c file-reader -- sh -c 'echo more >> /data-ro/hostname.txt'
sh: can't create /data-ro/hostname.txt: Read-only file system
command terminated with exit code 1

The container called file-reader mounts the same
volume and can read the data written by sleep.

But the file-reader mount is defined as read-
only, so it's not allowed to modify the data.

```

Figure 7.3 Containers can mount the same Pod volume to share data, but with different access levels

A good old empty directory volume shows its worth again here - it's a simple scratchpad which all the Pod containers can access. Volumes are defined at the Pod level and mounted at the container level, which means you can use any type of volume or PVC and make it available for many containers to use. Decoupling the volume definition from the volume mount also allows selective sharing - so one container may be able to see Secrets while the others can't.

The other shared space is the network, where containers can listen on different ports and provide independent pieces of functionality. That's very useful if your app container is doing some background work but it doesn't have any features to report on progress - another container in the same Pod can provide a REST API which reports on what the app container is doing.

There's a simplified version of that in listing 7.2. This is an update to the sleep deployment which replaces the filesharing container with a new container spec that runs a simple HTTP server.

Listing 7.2 - sleep-with-server.yaml, running a web server in a second container

```

spec:
  containers:

```

```

- name: sleep
  image: kiamol/ch03-sleep  # the same container spec as listing 7.1
- name: server
  image: kiamol/ch03-sleep  # the second container is different
  command: ['sh', '-c', "while true; do echo -e 'HTTP/1.1 ...'"]
  ports:
    - containerPort: 8080  # it runs an HTTP server on port 8080

```

Now the Pod will run with the original app container - the sleep container which isn't really doing anything - and a server container which provides an HTTP endpoint on port 8080. The two containers share the same network space, so the sleep container can access the server using the localhost address.

TRY IT NOW Update the sleep deployment using the file from listing 7.2 and confirm that the server container is accessible.

```

# deploy the update:
kubectl apply -f sleep/sleep-with-server.yaml

# check the Pod status:
kubectl get pods -l app=sleep

# list the container names in the new Pod:
kubectl get pod -l app=sleep -o
jsonpath='{.items[0].status.containerStatuses[*].name}'

# make a network call between the containers:
kubectl exec deploy/sleep -c sleep -- wget -q -O - localhost:8080

# check the server container logs:
kubectl logs -l app=sleep -c server

```

You can see my output in figure 7.4 - although these are separate containers, at the network level they function as though they were different processes running on the same machine, using the local address for communication.

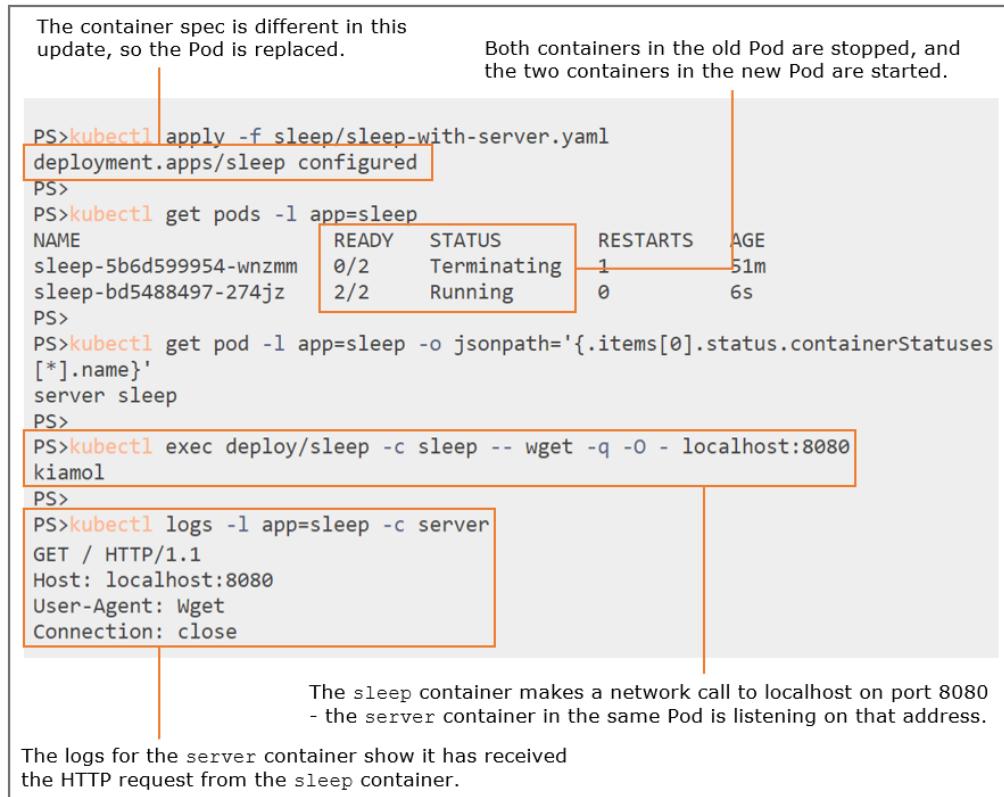


Figure 7.4 Network communication between containers in the same Pod is over localhost

It's not just within the Pod that the network is shared. The Pod has an IP address on the cluster and if any containers in the Pod are listening on ports, then they can be accessed by other Pods. You can create a Service which routes traffic to the Pod on a specific port, and whichever container is listening on that port will receive the request.

TRY IT NOW Use a Kubectl command to expose the Pod port, which is a quick way to create a Service without writing YAML. Then test that the HTTP server is accessible externally.

```

# create a Service targeting the server container port:
kubectl expose -f sleep/sleep-with-server.yaml --type LoadBalancer --port 8020 --
target-port 8080

# get the URL for your service:
kubectl get svc sleep -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8020'
# open the URL in your browser

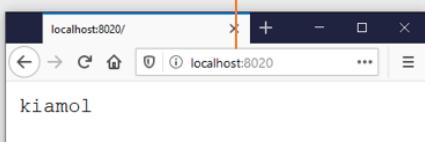
# check the server container logs:
kubectl logs -l app=sleep -c server
  
```

You can see my output in figure 7.5 - from the outside world it's just network traffic going to a Service which gets routed to a Pod. The Pod is running multiple containers, but that's a detail which is hidden from the consumer.

This is a quick way to create a Service for an existing Pod - this maps the server container's port 8080 to a public port 8020.

The Service routes network calls to the right container in the Pod.

```
PS>kubectl expose -f sleep/sleep-with-server.yaml --type LoadBalancer --port 8020 --target-port 8080
service/sleep exposed
PS>
PS>kubectl get svc sleep -o jsonpath='http://.status.loadBalancer.ingress[0].*:8020'
http://localhost:8020
PS>
PS>
PS>
PS>
PS>kubectl logs -l app=sleep -c server
```



```
GET / HTTP/1.1
Host: localhost:8020
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:76.0) Gecko/20100101 Firefox/76.0
```

The server container logs show the latest browser request.

Figure 7.5 Services can route network requests to any Pod containers which have published ports

You should be getting a feel for how powerful this is, and in the rest of the chapter we'll put the ideas to work in real-world scenarios. There's one thing that needs to be stressed though - a Pod is not a replacement for a VM, so don't think you can run all the components of an app in one Pod. You might be tempted to model an app like that, with a web server container and an API container running in the same Pod: don't. A Pod is a single unit and it should be used for a single component of your app. Additional containers can be used to support the app container, but you shouldn't be running different apps in the same Pod - if you do, it ruins your ability to update, scale and manage those components independently.

7.2 Setting up applications with init containers

So far we've run Pods with multiple containers where all the containers run in parallel - they start together and the Pod isn't considered to be ready until all the containers are ready. You'll hear that referred to as the *sidecar pattern*, which reinforces the idea that additional containers (the sidecars) play a supporting role to the application container (the bike). There's another pattern which Kubernetes supports when you need a container to run before the app container, to set up part of the environment. Those are called *init containers*.

Init containers work differently to sidecars. You can have multiple init containers defined for the Pod and they run in sequence - in the order they're written in the Pod spec. Each init container needs to complete successfully before the next one starts, and all must complete successfully before the Pod containers are started. Figure 7.6 shows the startup sequence for a Pod with init containers.

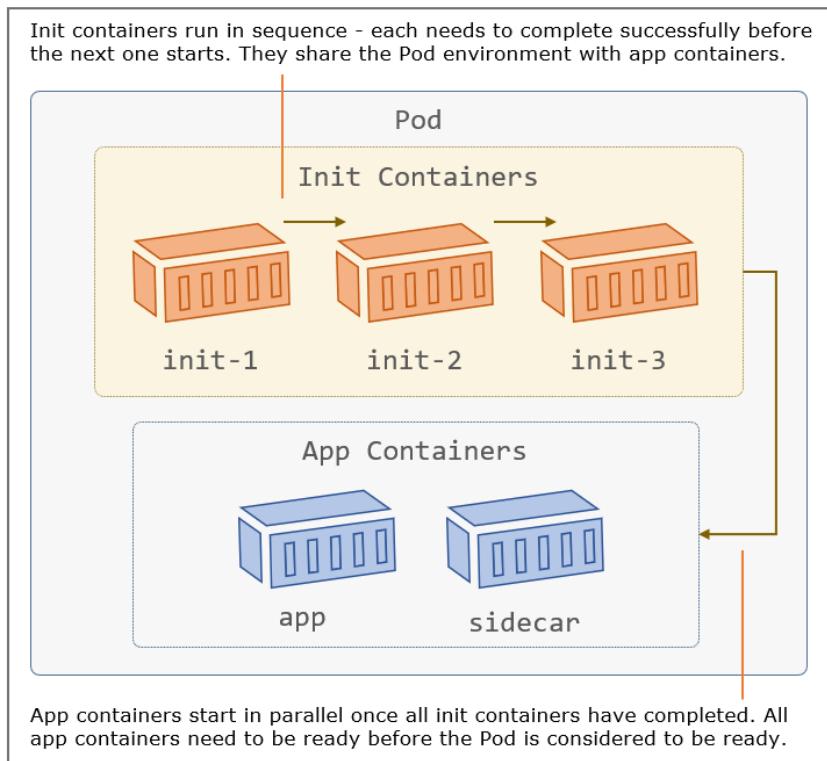


Figure 7.6 Init containers are useful for startup tasks to prep the Pod for the app containers

All containers can access volumes defined in the Pod, so the major use-case is for an init container to write data which prepares the environment for the application container. Listing 7.3 shows a simple extension to the HTTP server in the sleep Pod from the last exercise. An init container runs and generates an HTML file which it writes in a mount for an EmptyDir volume - the server container responds to HTTP requests using that file.

Listing 7.3 - sleep-with-html-server.yaml, an init container in the Pod spec

```
spec:
  initContainers:
    - name: init-html
      image: kiamol/ch03-sleep
```

Pod spec in the Deployment template
init containers have their own array
and they run in sequence

```

    command: ['sh', '-c', "echo '<!DOCTYPE html...>' > /data/index.html"]
  volumeMounts:
  - name: data
    mountPath: /data           # init containers can mount Pod volumes

```

This example uses the same sleep image for the init container - but it can be any image. You might use an init container to set up the application environment using tools that you don't want to be present in the running application. An init container can use a Docker image with the Git command line installed and clone a repository into the shared filesystem. The app container gets access to the files without you having to set up the Git client in your app image.

TRY IT NOW Deploy the update from listing 7.3 and see how init containers work.

```

# apply the updated spec with the init container:
kubectl apply -f sleep/sleep-with-html-server.yaml

# check the Pod containers:
kubectl get pod -l app=sleep -o
jsonpath='{.items[0].status.containerStatuses[*].name}'

# and the init containers:
kubectl get pod -l app=sleep -o
jsonpath='{.items[0].status.initContainerStatuses[*].name}'

# check logs from the init container - there are none:
kubectl logs -l app=sleep -c init-html

# and check the file is available in the sidecar:
kubectl exec deploy/sleep -c server -- ls -l /data-ro

```

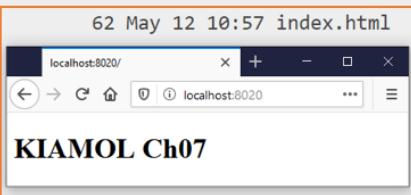
You'll pick up a few things from this exercise. App containers are guaranteed not to run until init containers complete successfully, so your app can safely make assumptions about the environment which the init container prepares - in this case the HTML file is sure to exist before the server container starts. And init containers are a different part of the Pod spec, but some management features work in the same way as app pods - you can read the logs from init containers even after they have exited. My output is in figure 7.7.

Containers and init containers are separate parts of the Pod spec.

You can view logs for init containers after they've completed - this one doesn't write any log entries.

```

PS>kubectl apply -f sleep/sleep-with-html-server.yaml
deployment.apps/sleep configured
PS>
PS>kubectl get pod -l app=sleep -o jsonpath='{.items[0].status.containerStatuses[*].name}'
server sleep
PS>
PS>kubectl get pod -l app=sleep -o jsonpath='{.items[0].status.initContainerStatuses[*].name}'
init-html
PS>
PS>kubectl logs -l app=sleep -c init-html
PS>
PS>kubectl exec deploy/sleep -c server -- ls -l /data-ro
total 4
-rw-r--r--    1 root      root
PS>
PS>
```



The server sidcar container now responds to HTTP requests with the HTML file generated by the init container.

Figure 7.7 Init containers are useful for preparing the Pod environment for app and sidcar containers

That still isn't a very real-world example though, so let's do something better. We covered app configuration in chapter 4, and saw how to use environment variables, ConfigMaps and Secrets to build up a hierarchy of config settings. That's great if your app supports it, but many older apps don't have that flexibility: they expect to find a single config file in one place and they don't go looking anywhere else. Let's look at an app like that.

TRY IT NOW There's a new demo app for this chapter, because if I'm getting bored with looking at Pi then you must be too. This one isn't much more fun but at least it's different - it just writes a timestamp to a log file every few seconds. It has an old-style config framework so we can't use any of the configuration techniques we've learned.

```

# run the app which uses a single config file:
kubectl apply -f timecheck/timecheck.yaml

# check the container logs - there won't be any:
kubectl logs -l app=timecheck

# check the log file inside the container:
```

```
kubectl exec deploy/timecheck -- cat /logs/timecheck.log
# and check the config setup:
kubectl exec deploy/timecheck -- cat /config/appsettings.json
```

You can see my output in figure 7.8. A limited config framework isn't the only reason this app isn't a good citizen in a container platform - there are no logs in the Pod either - but we can address all the problems with additional containers in the Pod.

This version of the app doesn't write log entries where Kubernetes can find them.

Instead they're written in a file in the container filesystem.

```
PS>kubectl apply -f timecheck/timecheck.yaml
deployment.apps/timecheck created
PS>
PS>kubectl logs -l app=timecheck
PS>
PS>kubectl exec deploy/timecheck -- cat /logs/timecheck.log
2020-05-12 11:13:38.546 +00:00 [INF] Environment: DEV; version: 1.0; time check: 11:13.38
PS>
PS>kubectl exec deploy/timecheck -- cat /config/appsettings.json
{
  "Application": {
    "Version": "1.0",
    "Environment": "DEV"
  },
  "Timer": {
    "IntervalSeconds": "5"
  }
}
```

The log entries use settings from the config file - including the environment name and the app version.

Figure 7.8 Older apps which use a single config source can't benefit from a config hierarchy

An init container is a perfect tool to bring this app into line with the config approach we want to use for all our apps. We can store the settings in ConfigMaps, Secrets and environment variables, and use an init container to read from all the different inputs, merge the contents and write the output to the single file location which the app uses. Listing 7.4 shows the init container in the Pod spec.

Listing 7.4 - timecheck-with-config.yaml, an init container which writes config

```
spec:
  initContainers:
    - name: init-config
      image: kiamol/ch03-sleep    # this image has the jq tool
```

```

    command: ['sh', '-c', "cat /config-in/appsettings.json | jq --arg APP_ENV
    \\\"$APP_ENVIRONMENT\\\" '.Application.Environment=$APP_ENV' > /config-
    out/appsettings.json"]           # the container uses it to write JSON
    env:
      - name: APP_ENVIRONMENT      # all containers have their own environment
        value: TEST                 # variables - they're not shared in the Pod
    volumeMounts:
      - name: config-map          # mounts a ConfigMap volume to read
        mountPath: /config-in
      - name: config-dir           # mounts an EmptyDir volume to write to
        mountPath: /config-out

```

There are a few things to note here before we update the deployment:

- the init container uses the jq tool, which the app doesn't need. The containers use different images, each with just the tools necessary to run that step.
- the command in the init container reads from a ConfigMap volume mount, merges in the environment variable values and writes out to an EmptyDir volume mount.
- the app container mounts the EmptyDir volume to the path where the config file needs to be - the file generated by the init container hides the default configuration in the app image.
- containers don't share environment variables - the settings are specified for the init container; the app container doesn't see those.
- containers map the volumes they need - both containers mount the EmptyDir volume which they share, but only the init container mounts the ConfigMap.

When we apply this update, the app's behavior will change in line with the ConfigMap and environment variables, even though the app container doesn't use them as config sources.

TRY IT NOW Update the timecheck app using listing 7.4 so the app container is configured from multiple sources.

```

# apply the ConfigMap and the new Deployment spec:
kubectl apply -f timecheck/timecheck-configMap.yaml -f timecheck/timecheck-with-
config.yaml

# wait for the containers to start:
kubectl wait --for=condition=ContainersReady pod -l app=timecheck,version=v2

# check the log file in the new app container:
kubectl exec deploy/timecheck -- cat /logs/timecheck.log

# and see the config file built by the init container:
kubectl exec deploy/timecheck -- cat /config/appsettings.json

```

You'll see when you run this that the app works with the new configuration, and the only change for the application container spec is that the config directory is mounted from the EmptyDir volume. My output is in figure 7.9.

This update adds an init container which reads a ConfigMap and environment variables and writes them to a config file.

The app reads the config file and the settings are used in the logs.

```

PS> kubectl apply -f timecheck/timecheck-configMap.yaml -f timecheck/timecheck-with-config.yaml
configmap/timecheck-config created
deployment.apps/timecheck configured
PS>
PS> kubectl exec deploy/timecheck -- cat /logs/timecheck.log
2020-05-12 12:54:24.155 +00:00 [INF] Environment: TEST; version: 1.1; time check
: 12:54.24
PS>
PS> kubectl exec deploy/timecheck -- cat /config/appsettings.json
{
  "Application": {
    "Version": "1.1",
    "Environment": "TEST"
  },
  "Timer": {
    "IntervalSeconds": "7"
  }
}

```

The config file is in the expected place, but this is now a volume mount which surfaces the file written by the init container.

Figure 7.9 Init containers can change app behavior without changes to the app code or Docker image

This approach works because the config file is loaded from a dedicated directory - remember that a volume mount overwrites a directory from the image, if it already exists. If the app loaded a config file from the same directory as the app binaries, you couldn't do this because the EmptyDir mount would overwrite the whole app folder. In that scenario you would need an additional step in the app container startup, to copy the config file from the mount into the application directory.

Applying a standard config approach to non-standard apps is a great use for init containers, but older apps still won't play nicely in a modern platform, and that's where sidecar containers can help.

7.3 Applying consistency with adapter containers

Moving apps to Kubernetes is a great opportunity to add a layer of consistency across all your apps, so you deploy and manage them the same way using the same tools, no matter what the app is doing, or what technology stack it uses, or when it was developed. My fellow Docker Captain Sune Keller has talked about the [service hotel](#) concept they use at Alm Brand - their container platform offers a set of guarantees for "customers" (like high availability and security), provided they adhere to the rules (like pulling config from the platform and writing logs out to it).

Not all apps know about the rules, and some of them can't be applied by the platform from the outside - but sidecar containers run alongside the app container so they have a privileged position. You can use them as *adapters*, which understand some aspect of how the app works and adapts it to how the platform wants it to work. Logging is a classic example.

Every app writes some output to log entries - or should, otherwise it would be entirely unmanageable and you should refuse to work with it. Modern app platforms like Node.js and .NET Core write to the standard output stream, which is where Docker fetches container logs, and where Kubernetes gets the Pod logs. Older apps have different ideas about logging and they may write to files or other targets which never get surfaced as container logs, so you never see any Pod logs. That's what the timecheck app does and we can fix it with a very simple sidecar container - the spec is in listing 7.5.

Listing 7.5 - timecheck-with-logging.yaml, using a sidecar container to expose logs

```
containers:
  - name: timecheck
    image: kiamol/ch07-timecheck
    volumeMounts:
      - name: logs-dir           # the app container writes the log file
        mountPath: /logs         # to an EmptyDir volume mount
        # abbreviated - the full spec also includes the config mount
  - name: logger
    image: kiamol/ch03-sleep   # the sidecar just watches the log file
    command: ['sh', '-c', 'tail -f /logs-ro/timecheck.log']
    volumeMounts:
      - name: logs-dir
        mountPath: /logs-ro     # using the same volume as the app
        readOnly: true
```

All the sidecar does is to mount the log volume (go EmptyDir!) and use the standard Linux `tail` command to read from the log file. The `-f` option means the command will follow the file - effectively it just sits and watches for new writes, and when any lines get written to the file they're echoed to standard out. It's a relay which adapts the app's actual logging implementation to the expectations of Kubernetes.

TRY IT NOW Apply the update from listing 7.5 and check the app logs are available.

```
# add the sidecar logging container:
kubectl apply -f timecheck/timecheck-with-logging.yaml

# wait for the containers to start:
kubectl wait --for=condition=ContainersReady pod -l app=timecheck,version=v3

# check the Pods:
kubectl get pods -l app=timecheck

# and the containers in the Pod:
kubectl get pod -l app=timecheck -o
jsonpath='{.items[0].status.containerStatuses[*].name}'

# now you can see the app logs in the Pod:
kubectl logs -l app=timecheck -c logger
```

There's some inefficiency here because the app container writes logs to a file and then the logging container reads them back out again. There will be a small time lag and potentially a lot of wasted disk - but the Pod will be replaced in the next app update and all the space used in the volume will be reclaimed. The benefit is that this Pod now behaves like every other Pod, making application logs available to Kubernetes, but without any changes needed to the app itself - as you see in figure 7.10.

The original Pod had a single app container - the init container isn't included as a running container. The replacement has two.

Those are the timecheck app container and the logging sidecar.

```

PS>kubectl apply -f timecheck/timecheck-with-logging.yaml
deployment.apps/timecheck configured
PS>
PS>kubectl get pods -l app=timecheck
NAME                  READY   STATUS    RESTARTS   AGE
timecheck-674dddc7f-h9257  2/2     Running   0          6s
timecheck-779f995658-q9cs4  1/1     Terminating   0          54m
PS>
PS>kubectl get pod -l app=timecheck -o jsonpath='{.items[0].status.containerStatuses[*].name}'
logger timecheck
PS>
PS>kubectl logs -l app=timecheck -c logger
2020-05-12 13:48:37.065 +00:00 [INF] Environment: TEST; version: 1.1; time check : 13:48:37

```

The sidecar is a log adapter - the entries written to the filesystem in the app container are now available from the Pod.

Figure 7.10 Adapters bring a layer of consistency to Pods, making old apps behave like new apps

Receiving configuration from the platform and writing logs out to the platform are pretty much the fundamentals for any application, but as your platform matures you'll have more expectations for standard behavior. You'll want to be able to test that the application inside the container is healthy, and you'll also want to be able to pull metrics out from the app to see what it's doing and how hard it's working.

Sidecars can help there too, either by running custom containers which provide information tailored to the app, or by having standard health and metrics container images which you apply to all your Pod specs. We'll round off the exercises using the timecheck app, and add those features which make it a good citizen for Kubernetes. We'll cheat though with some more static HTTP server containers, which you can see in listing 7.6.

Listing 7.6 - timecheck-good-citizen.yaml, more sidecars to extend the app

```

containers:
  - name: timecheck
  # ...

```

```

- name: logger
# ...

- name: healthz      # a new sidecar which exposes a healthcheck API
  image: kiamol/ch03-sleep    # this is just a static response
  command: ['sh', '-c', "while true; do echo -e 'HTTP/1.1 200 OK\nContent-Type: application/json\nContent-Length: 17\n\n{\"status\": \"OK\"}' | nc -l -p 8080; done"]
  ports:
    - containerPort: 8080      # available at port 8080 in the Pod

- name: metrics        # another sidecar which adds a metrics API
  image: kiamol/ch03-sleep    # the content is static again
  command: ['sh', '-c', "while true; do echo -e 'HTTP/1.1 200 OK\nContent-Type: text/plain\nContent-Length: 104\n\n# HELP timechecks_total The total number timechecks.\n# TYPE timechecks_total counter\nntimechecks_total 6' | nc -l -p 8081; done"]
  ports:
    - containerPort: 8081      # and is available on a different port

```

The full YAML file also includes a ClusterIP Service which publishes port 8080 for the health endpoint and port 8081 for the metrics endpoint - in a production cluster those would be used by other components to collect monitoring stats. The Deployment is an extension of the previous releases, so the app uses an init container for configuration and has a logging sidecar along with the new sidecars.

TRY IT NOW Deploy the update and check the new management endpoints for the health and performance of the app.

```

# apply the update:
kubectl apply -f timecheck/timecheck-good-citizen.yaml

# wait for all the containers to be ready:
kubectl wait --for=condition=ContainersReady pod -l app=timecheck,version=v4

# check the running containers:
kubectl get pod -l app=timecheck -o
jsonpath='{.items[0].status.containerStatuses[*].name}'

# use the sleep container to check the timecheck app health:
kubectl exec deploy/sleep -c sleep -- wget -q -O - http://timecheck:8080

# and its metrics:
kubectl exec deploy/sleep -c sleep -- wget -q -O - http://timecheck:8081

```

When you run the exercise you'll see everything works as expected, as in figure 7.11. But you may also see the updates weren't as speedy as you're used to, with the new Pod taking longer to start up and the old Pod taking longer to terminate. The additional startup time is from having the init container, the app container and all the sidecars - they all need to be ready before the new Pod is considered ready. The additional termination time is because the replaced Pod also had multiple containers, which are each given a grace period for the container process to shut down.

The update has to start multiple containers in the new Pod and terminate multiple containers in the old Pod - so it takes longer than for a single-container Pod.

The new Pod has four active containers - the app container and three sidecars.

```
PS>kubectl apply -f timecheck/timecheck-good-citizen.yaml
service/timecheck created
deployment.apps/timecheck configured
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=timecheck,version=v4
pod/timecheck-54b49f688-76f71 condition met
PS>
PS>kubectl get pod -l app=timecheck -o jsonpath='{.items[0].status.containerStatuses[*].name}'
healthz logger metrics timecheck
PS>
PS>kubectl exec deploy/sleep -c sleep -- wget -q -O - http://timecheck:8080
{"status": "OK"}
PS>
PS>kubectl exec deploy/sleep -c sleep -- wget -q -O - http://timecheck:8081
# HELP timechecks_total The total number timechecks.
# TYPE timechecks_total counter
timechecks_total 6
```

Port 8080 is a simple healthcheck which returns a (fake) JSON response. This can be used to check if the app is healthy.

Port 8081 returns (fake) metrics. These can be used to monitor what the app is doing.

Figure 7.11 Multiple adapter sidecars give the app a consistent management API

There is an overhead to running all these sidecar containers as adapters. You've seen that it increases deployment times, but it also increases the ongoing compute requirements of the app - even basic sidecars which just tail log files and serve simple HTTP responses all use memory and compute cycles, and storage too. But if you want to move existing apps to Kubernetes which don't have those features, it's an acceptable approach to get all your apps behaving in the same way - as you see in figure 7.12.

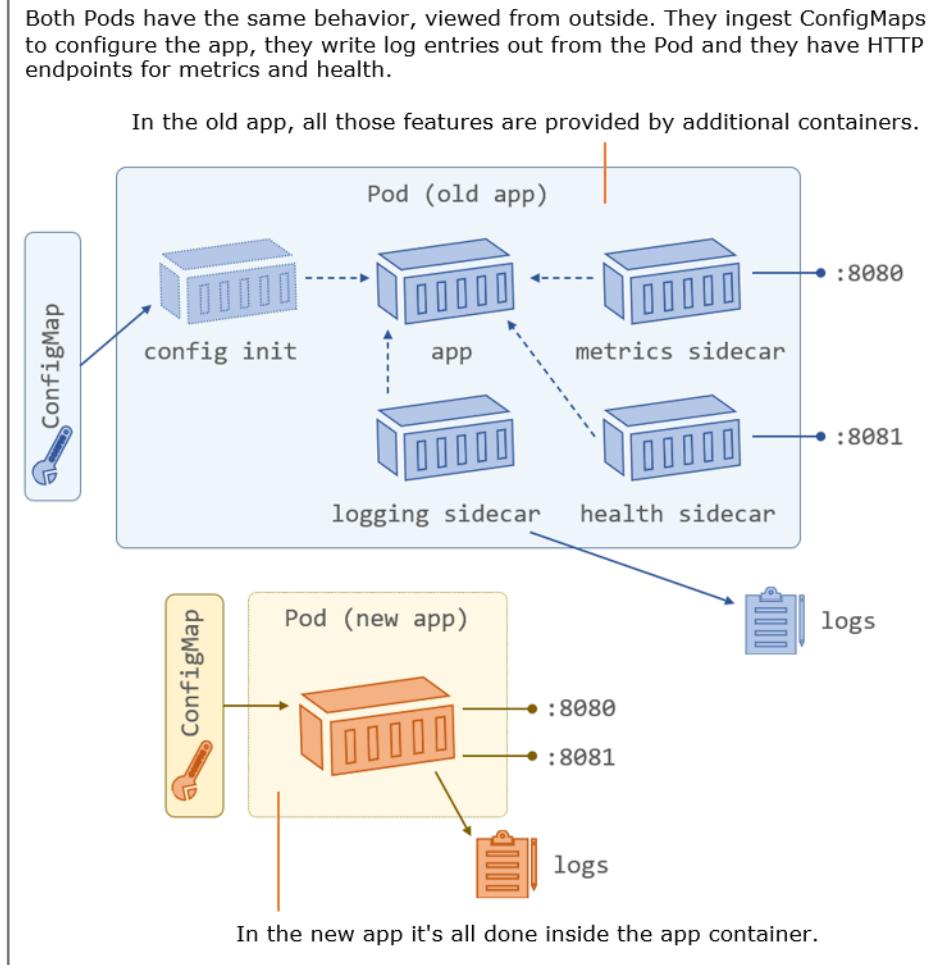


Figure 7.12 A consistent management API makes it easy to work with Pods — it doesn't matter how the API is provided inside the Pod

In the last exercise we used an old sleep Pod we had lying around to call the new HTTP endpoints for the timecheck app. Remember that Kubernetes has a flat networking model, where Pods can send traffic to any other Pods via a Service. You may want more control over the network communication in your app and you can do that with sidecars too, by running a proxy container which manages the outgoing traffic from your app container.

7.4 Abstracting connections with ambassador containers

The ambassador pattern lets you control and simplify outgoing connections from your application: your app makes network requests to the localhost address, which get picked up and actioned by the ambassador. There are lots of variations where you can make use of a generic ambassador container, or one which is specific to your application components. Figure 7.13 shows some examples - the logic in the ambassador might be geared to improving performance or increasing reliability or security.

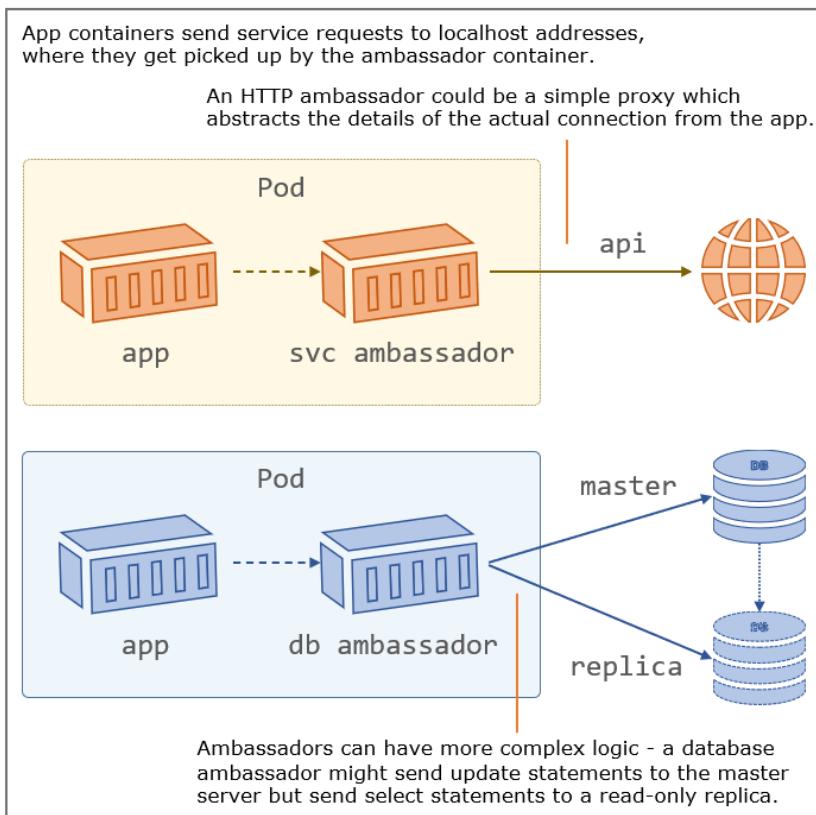


Figure 7.13 The ambassador pattern has lots of potential, from simplifying app logic to increasing performance

Taking control of the network away from the application is hugely powerful. A proxy container can do service discovery, load balancing, retries and even layer encryption onto an unencrypted channel. If you've heard of the Service Mesh architecture, using technologies like Linkerd and Istio - they're all powered by proxy sidecar containers in a variation of the ambassador pattern.

We won't use a Service Mesh here because that would take us well past lunchtime and on into the night, but we'll get a flavor of what they can do with a simplified example. The starting point is the random number app we've used before - there's a web app running in a Pod which consumes an API running in another Pod. The API is the only component the web app uses, so ideally we would restrict network calls to any other address, but in the initial deployment that doesn't happen.

TRY IT NOW Run the random number app and verify that the web app container can use any network address.

```
# deploy the app and Services:  
kubectl apply -f numbers/  
  
# find the URL for your app:  
kubectl get svc numbers-web -o  
jsonpath='http://{.status.loadBalancer.ingress[0].*}:8090'  
  
# browse and get yourself a nice random number  
  
# and check the web app has access to other endpoints:  
kubectl exec deploy/numbers-web -c web -- wget -q -O - http://timecheck:8080
```

The web Pod can reach the API using the ClusterIP Service and the domain name numbers-api, but it can also access any other address, which could be a URL on the public Internet or another ClusterIP Service. Figure 7.14 shows the app can read the health endpoint of the timecheck app - that should be a private endpoint and it might expose information which is useful to someone up to no good.

The web application gets its numbers from a call to the API, running in another Pod.

```

PS>kubectl apply -f numbers/
service/numbers-api created
deployment.apps/numbers-api created
service/numbers-web created
deployment.apps/numbers-web created
PS>
PS>kubectl get svc numbers-web -o jsonpath='http://{.status.loadBalancer.ingress[0].*:8090}'
http://localhost:8090
PS>
PS>kubectl exec deploy/numbers-web -c web -- wget -q -O - http://timecheck:8080
{"status": "OK"}
PS>

```

A screenshot of a terminal window showing Kubernetes commands being run. The commands include applying a deployment configuration, getting a service, and executing a command in a pod to download a file from another pod. To the right of the terminal is a browser window titled 'KIAMOL - Numbers.Web' showing the output of the command, which is a random number generator. The browser URL is 'localhost:8090'.

But the web container can access any network address - if it was compromised, attackers could use it to explore other apps in the cluster.

Figure 7.14 Kubernetes doesn't have any default restrictions on outgoing connections from Pod containers

There are lots of options for restricting network access besides using a proxy sidecar, but the ambassador pattern comes with some additional features which make it worth considering. Listing 7.7 shows an update to the web app spec, using a simple proxy container as an ambassador.

Listing 7.7 - web-with-proxy.yaml, using a proxy as an ambassador

```

containers:
  - name: web
    image: kiamol/ch03-numbers-web
    env:
      - name: http_proxy
        value: http://localhost:1080      # sets the container to use the proxy
                                         # so traffic goes to the ambassador
      - name: RngApi_Url
        value: http://localhost/api      # use a localhost address for the API
      - name: proxy
        image: kiamol/ch07-simple-proxy    # this is a basic HTTP proxy
        env:
          - name: Proxy_Port
            value: "1080"                # it routes network requests from the app
                                         # using the configured URI mapping
          - name: Proxy_Request_UriMap_Source
            value: http://localhost/api

```

```
- name: Proxy__Request__UriMap__Target
  value: http://numbers-api/sixeyed/kiamol/master/ch03/numbers/rng
```

This example shows the major pieces of the ambassador pattern: the app container uses localhost addresses for any services it consumes, and it's configured to route all network calls through the proxy container. The proxy is a custom app which logs network calls, maps localhost addresses to real addresses and blocks any addresses which are not listed in the map. All that becomes functionality in the Pod, but it's transparent to the application container.

TRY IT NOW Update the random number app and confirm the network is now locked down.

```
# apply the update from listing 7.5:
kubectl apply -f numbers/update/web-with-proxy.yaml

# refresh your browser and get a new number

# check the proxy container logs:
kubectl logs -l app=numbers-web -c proxy

# try to read the health of the timecheck app:
kubectl exec deploy/numbers-web -c web -- wget -q -O - http://timecheck:8080

# and check proxy logs again:
kubectl logs -l app=numbers-web -c proxy
```

Now the web app is decoupled even further from the API, because it doesn't even know the URL of the API - that's set in the ambassador, which can be configured independently of the app. The web app is also restricted to using a single address for outgoing requests, and all those calls are logged by the proxy as you see in figure 7.15.

The web container routes all traffic through the proxy - which maps the requested address to the actual address and logs the connection request.

```

PS>kubectl apply -f numbers/update/web-with-proxy.yaml
deployment.apps/numbers-web configured
PS>
PS>kubectl logs -l app=numbers-web -c proxy
** Logging proxy listening on port: 1080 ***
GET http://localhost/api -> http://numbers-api/sixeyed/kiamol/master/ch03/numbers/rng
PS>
PS>kubectl exec deploy/numbers-web -c web -- wget -q -O - http://timecheck:8080
PS>
PS>kubectl logs -l app=numbers-web -c proxy
** Logging proxy listening on port: 1080 ***
GET http://localhost/api -> http://numbers-api/sixeyed/kiamol/master/ch03/numbers/rng
GET http://timecheck:8080/ [BLOCKED]
PS>
PS>
```

The web app uses a localhost address for the API.

This proxy is configured to block requests for any other addresses; the web app can't access the health endpoint in the timecheck app.

Figure 7.15 All network access is via the ambassador, which can implement its own access rules

The ambassador for this web app proxies HTTP calls outside of the Pod, but the ambassador pattern is wider than that - it plugs into the network at the transport layer, so it can work on any kind of traffic. A database ambassador can make some smart choices - like sending queries to a read-only database replica, and only using the master database for writes. That's going to improve performance and scale, while keeping complex logic out of the application.

We'll round out the chapter by taking a closer look at what it means to use the Pod as a shared environment for many containers.

7.5 Understanding the Pod environment

The Pod is a boundary around one or more containers, just like the container is a boundary around one or more processes. They create layers of virtualization without adding overhead,

so they're flexible and efficient. The cost of that flexibility is - as always - complexity, and there are some nuances to working with multi-container Pods which you need to be aware of.

The main thing to understand is that the Pod is still the single unit of compute, even if there are lots of containers running inside it. Pods aren't ready until all the containers in the Pod are ready - and Services only send traffic to Pods which are ready. Adding sidecars and init containers adds to the failure modes for your application.

TRY IT NOW You can break your application if an init container fails - this update to the numbers app won't be successful because the init container is misconfigured.

```
# apply the update:  
kubectl apply -f numbers/update/web-v2-broken-init-container.yaml  
  
# check the new Pod:  
kubectl get po -l app=numbers-web,version=v2  
  
# and the logs for the new init container:  
kubectl logs -l app=numbers-web,version=v2 -c init-version  
  
# check the status of the Deployment:  
kubectl get deploy numbers-web  
  
# and the ReplicaSets:  
kubectl get rs -l app=numbers-web
```

You can see in that exercise that the failed init container effectively prevents the application from updating - the new Pod never enters the running state and won't receive traffic from the Service. The Deployment never scales down the old ReplicaSet because the new one doesn't reach the required level of availability, but the basic details of the Deployment look like the update has worked - as you see in figure 7.16.

This update has a broken init container spec. Kubernetes restarts the container but it will never run successfully, so the app containers won't start and the Pod will never be ready.

The logs from the failed init container are still available.

```

PS>kubectl apply -f numbers/update/web-v2-broken-init-container.yaml
deployment.apps/numbers-web configured
PS>
PS>kubectl get po -l app=numbers-web,version=v2
NAME           READY   STATUS      RESTARTS   AGE
numbers-web-5554bf49f8-k5l9v   0/2     Init:CrashLoopBackOff   1          12s
PS>
PS>kubectl logs -l app=numbers-web,version=v2 -c init-version
sh: can't create /config-out/version.txt: Read-only file system
PS>
PS>kubectl get deploy numbers-web
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
numbers-web   1/1     1           1           44m
PS>
PS>kubectl get rs -l app=numbers-web
NAME            DESIRED   CURRENT   READY   AGE
numbers-web-5554bf49f8   1         1         0       87s
numbers-web-6446c7f79f   1         1         1       28m
numbers-web-bb95c88d8   0         0         0       44m

```

The status of the Deployment is misleading - it's reporting on the ReplicaSets, and it's true that one is available.

But it's the wrong one - the old ReplicaSet is still running because the new one isn't ready, having no ready Pods.

Figure 7.16 Adding more containers to your Pod spec adds more opportunities for the Pod to fail

The same situation will happen if a sidecar container fails on startup - the Pod doesn't have all its containers running so the Pod itself isn't ready. Any deployment checks you have in place need to be extended for multi-container Pods to ensure all init containers run to completion and all Pod containers are running. You need to be aware of restart conditions too:

- if a Pod with init containers is replaced then the new Pod runs all the init containers again - you need to make sure your init logic can be run repeatedly;
- if you deploy a change to the init container image(s) for a Pod, that restarts the Pod - init containers all execute again and app containers are replaced;
- if you deploy a Pod spec change to the app container image(s), the app containers are replaced but the init containers are not executed again;
- if an application container exits then the Pod re-creates it - until the container is replaced the Pod is not fully running and won't receive Service traffic.

The Pod is a single compute environment but when you add multiple moving parts inside that environment you need to test all the failure scenarios and make sure your app behaves as you expect.

There's one last part of the Pod environment which we haven't covered, and that's the compute layer. Pod containers have a shared network and can share parts of the filesystem, but they can't access each other's processes - the container boundary still provides compute isolation. That's the default behavior but in some cases you want your sidecar to have access to the processes in the application container, either for inter-process communication or so the sidecar can fetch metrics about the app process.

You can enable this with a simple setting in the Pod spec - `shareProcessNamespace: true`. That means every container in the Pod shares the same compute space and can see each other's processes.

TRY IT NOW Deploy an update to the sleep Pod so the containers use a shared compute space and can access each other's processes.

```
# check the processes in the current container:  
kubectl exec deploy/sleep -c sleep -- ps  
  
# apply the update:  
kubectl apply -f sleep/sleep-with-server-shared.yaml  
  
# wait for the new containers:  
kubectl wait --for=condition=ContainersReady pod -l app=sleep,version=shared  
  
# and check the processes again:  
kubectl exec deploy/sleep -c sleep -- ps
```

You can see my output in figure 7.17 - the sleep container can see all the server container's processes - and it could happily kill them all and leave the Pod in a confused state.

The sleep container can only see its own processes, not the processes in the server container.

This update uses a shared compute space for all the containers in the Pod.

```

PS>kubectl exec deploy/sleep -c sleep -- ps
PID  USER      TIME  COMMAND
 1 root      0:00 /bin/sh -c trap : TERM INT; (while true; do sleep 1000; don
e) & wait
 44 root      0:00 sleep 1000
 45 root      0:00 ps

PS>
PS>kubectl apply -f sleep/sleep-with-server-shared.yaml
deployment.apps/sleep configured

PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=sleep,version=shared
pod/sleep-7594f96c7c-xqgn5 condition met
PS>
PS>kubectl exec deploy/sleep -c sleep -- ps
PID  USER      TIME  COMMAND
 1 root      0:00 /pause
 6 root      0:00 /bin/sh -c trap : TERM INT; (while true; do sleep 1000; don
e) & wait
 18 root      0:00 sleep 1000
 19 root      0:00 sh -c while true; do echo -e 'HTTP/1.1 200 OK Content-Type:
text/plain Content-Length: 7 kiamol' | nc -l -p 8080; done
 25 root      0:00 nc -l -p 8080
 31 root      0:00 ps

```

Now the sleep container can see the nc process, which is the HTTP server running in the sidecar container.

Figure 7.17 You can configure a Pod so all containers can see all processes. Use with care.

That's all for multi-container Pods. You've seen in this chapter that you can use init containers to prepare the environment for your application container, and run sidecar containers to add features to your app - all without changing the app code or the Docker image. There are some caveats to using multiple containers, but it's a pattern you'll use often to extend your applications. Just remember that the Pod should be one logical component: I don't want to see you running Nginx, WordPress and MySQL in a single Pod just because you can. Let's tidy up now and get ready for the lab.

TRY IT NOW Remove everything matching this chapter's label.

```
kubectl delete all -l kiamol=ch07
```

7.6 Lab

It's back to the Pi app for this lab. The Docker image `kiamol/ch05-pi` can actually be used in different ways, and to run it as a web app you need to override the startup command in the container spec. We've done that in the YAML files in previous chapters, but now we've been asked to use a standard approach to setting up the pod. Here are the requirements and some hints:

- the app container needs to use a standard startup command that all Pods in our platform are using - it should run `/init/startup.sh`
- the Pod should use port 80 for the app container
- the Pod should also publish port 8080 for an HTTP server which returns the version number of the app
- there is no startup script in the app container image, so you'll need to use something which can create that script and make it executable for the app container to run
- the app doesn't publish a version API on port 8080 (or anywhere else), so you'll need something which can provide that (it can just be any static text).

The starting point is the YAML in `ch07/lab/pi`, which is broken at the moment. You'll need to do some investigation into how the app ran in previous chapters and apply the techniques we've learned in this chapter. There are plenty of ways to approach this one, and you'll find my sample solution in the usual place:

<https://github.com/sixeyed/kiamol/blob/master/ch07/lab/README.md>

8

Running data-heavy apps with StatefulSets and Jobs

"Data-heavy" isn't a very scientific term, but this chapter is about running a class of application which isn't just stateful, but is very demanding about how it uses state. Databases are one example. They need to run across multiple instances for high availability, each instance needs a local data store for fast access, and those independent data stores need to be kept in sync. The data has its own availability requirements and you'll need to run backups periodically to guard against terminal failure or corruption. There are similar requirements for other data-intensive applications like message queues and distributed caches.

You can run those kinds of app in Kubernetes, but you need to design around an inherent conflict: Kubernetes is a dynamic environment, and data-heavy apps typically expect to run in a stable environment. Clustered applications which expect to find peers at a known network address won't work nicely in a ReplicaSet, and backup jobs which expect to read from a disk drive won't work well with PersistentVolumeClaims. You need to model your app differently if it has strict data requirements, and we'll cover how to do that in this chapter with some more advanced controllers: StatefulSets, Jobs and CronJobs.

8.1 How Kubernetes models stability with StatefulSets

A StatefulSet is a Pod controller with predictable management features - it lets you run applications at scale within a stable framework. When you deploy a ReplicaSet it creates Pods with random names, which are not individually addressable over DNS, and it starts them all in parallel. When you deploy a StatefulSet it creates Pods with predictable names, which can be individually accessed over DNS, and starts them in order - the first pod needs to be up and running before the second pod gets created.

Clustered applications are a great candidate for StatefulSets - typically they're designed with a primary instance and one or more secondaries, which gives them high availability. You might be able to scale the secondaries, but they all need to reach the primary and then use it to synchronize their own data. You can't model that with a Deployment because in the ReplicaSet there is no way to identify a single Pod as the primary, so you'd end up with bizarre and unpredictable conditions with multiple primaries or zero primaries.

Figure 8.1 shows an example of that, which could be used to run the Postgres database we've used in previous chapters for the to-do list application, but using a StatefulSet to get replicated data and high availability.

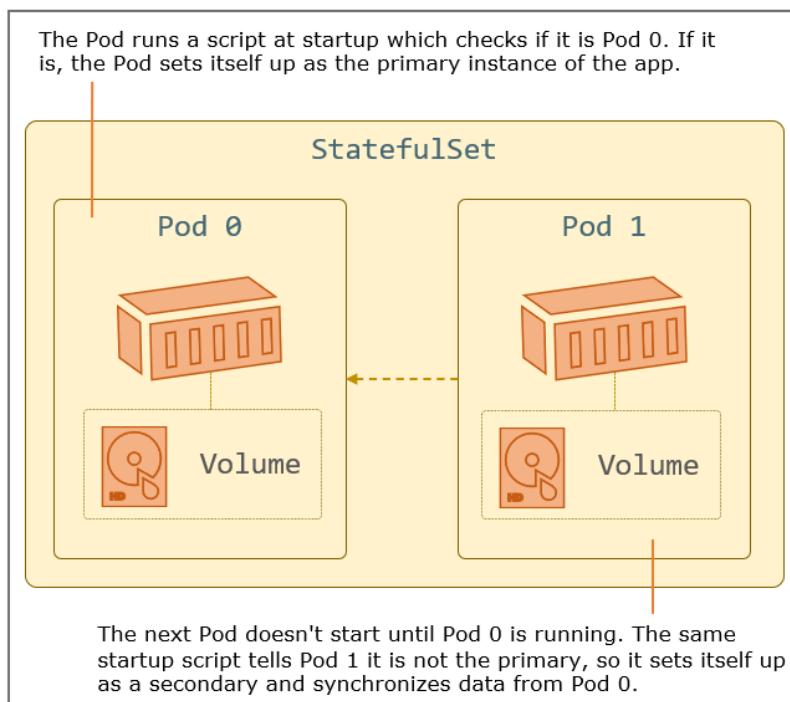


Figure 8.1 In a StatefulSet each Pod can have its own copy of data replicated from the first Pod

The setup for this is quite involved, and we'll spend a couple of sections getting there in stages, so you learn how all the pieces of a working StatefulSet fit together. It's a pattern which is useful for more than just databases - many older applications were designed for a static runtime environment and made assumptions about stability which don't hold true in Kubernetes. StatefulSets allow you to model that stability, and if your goal is to move your existing apps to Kubernetes then they may be something you use early in that journey.

Let's start with a simple StatefulSet that shows the basics. Listing 8.1 shows that StatefulSets have pretty much the same specs as other Pod controllers, except that they also need to include the name of a Service.

Listing 8.1 - todo-db.yaml, a simple StatefulSet

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: todo-db
spec:
  selector:           # StatefulSets use the same selector mechanism
  matchLabels:
    app: todo-db
  serviceName: todo-db      # StatefulSets must be linked to a Service
  replicas: 2
  template:
    # pod spec...
```

When you deploy that YAML file, you'll get a StatefulSet running two Postgres pods - but don't get too excited — they're just two separate database servers which happen to be managed by the same controller. There's more work needed to get two Pods to be a replicated database cluster, and we'll get there over the next few sections.

TRY IT NOW Deploy the StatefulSet from listing 8.1 and see how the Pods it creates compare to Pods managed by a ReplicaSet.

```
# deploy the StatefulSet, Service and a Secret for the Postgres password:
kubectl apply -f todo-list/db/

# check the StatefulSet:
kubectl get statefulset todo-db

# check the pods:
kubectl get pods -l app=todo-db

# find the hostname of Pod 0:
kubectl exec pod/todo-db-0 -- hostname

# and check the logs of Pod 1:
kubectl logs todo-db-1 --tail 1
```

You can see from figure 8.2 that a StatefulSet works in a very different way from a ReplicaSet or a DaemonSet. The Pods have a predictable name, which is the StatefulSet name followed by the index of the Pod, so you can manage the Pods using their names instead of having to use a label selector.

StatefulSets are Pod controllers, but they work in a predictable way - the Pods are started in order and the Pod name contains the index of the Pod in the set.

```

PS>kubectl apply -f todo-list/db/
secret/todo-db-secret created
service/todo-db created
statefulset.apps/todo-db created
PS>
PS>kubectl get statefulset todo-db
NAME      READY   AGE
todo-db   2/2     8s
PS>
PS>kubectl get pods -l app=todo-db
NAME      READY   STATUS    RESTARTS   AGE
todo-db-0  1/1     Running   0          14s
todo-db-1  1/1     Running   0          12s
PS>
PS>kubectl exec pod/todo-db-0 -- hostname
todo-db-0
PS>
PS>kubectl logs todo-db-1 --tail 1
2020-05-06 19:45:40.191 UTC [1] LOG: database system is ready to accept connections

```

This is a stable environment - the logic inside the Pod can check if it's the first Pod in the set, and operators can use the predictable Pod names to work with them.

Figure 8.2 A StatefulSet can create the environment for a clustered application, but the app needs to configure itself

The Pods are still managed by the controller, but in a more predictable way than with a ReplicaSet. Pods are created in order from zero up to n; you can scale down the set and it will remove them in the reverse order, starting from n and working down. If you delete a Pod the controller will create a replacement - it will have the same name and configuration as the original, but it will be a new Pod.

TRY IT NOW Delete Pod 0 of the StatefulSet and see that Pod 0 comes back again.

```

# check the internal ID of Pod 0:
kubectl get pod todo-db-0 -o jsonpath='{.metadata.uid}'

# delete the Pod:
kubectl delete pod todo-db-0

# check Pods:
kubectl get pods -l app=todo-db

# and check that the new Pod is a new Pod:
kubectl get pod todo-db-0 -o jsonpath='{.metadata.uid}'

```

You can see in figure 8.3 that a StatefulSet provides a stable environment for the app - Pod 0 is replaced with an identical Pod 0 - but that doesn't trigger a whole new set; the original Pod 1 remains. Ordering is only applied for creation and scaling, and not for replacing missing Pods.

Every Kubernetes object has an internal ID - the UID. You don't often use them, but they're useful when objects get replaced by different objects with the same name.

```
PS>kubectl get pod todo-db-0 -o jsonpath='{.metadata.uid}'  
071ec8fd-5056-49e8-9feb-d83dc91412b6
```

```
PS>
```

```
PS>kubectl delete pod todo-db-0
```

```
pod "todo-db-0" deleted
```

```
PS>
```

```
PS>kubectl get pods -l app=todo-db
```

NAME	READY	STATUS	RESTARTS	AGE
todo-db-0	1/1	Running	0	4s
todo-db-1	1/1	Running	0	11m

```
PS>
```

```
PS>kubectl get pod todo-db-0 -o jsonpath='{.metadata.uid}'  
cb813fad-0082-4623-a7b1-0b2c844ae1d8
```

Deleting Pod 0 causes a new Pod 0 to be created - it has the configuration of the original and will use any volumes the original used, but it is a new object.

Figure 8.3 StatefulSets replace missing replicas exactly as they were

The StatefulSet is only the first part of modelling a stable environment. You can get DNS names for each Pod linking the StatefulSet to a service, and that means you can configure Pods to initialize themselves by working with other replicas at known addresses.

8.2 Bootstrapping Pods with init containers in StatefulSets

The Kubernetes API composes objects from other objects - the Pod template in a StatefulSet definition is the same object type you use in the template for a Deployment, and in a bare Pod definition. That means all the Pod features are available for StatefulSets even though the Pods themselves are managed in a very different way. We learned all about init containers in chapter 7 and they're a perfect tool for the complicated initialization steps you often need in clustered applications.

Listing 8.2 shows the first init container for an update to the Postgres deployment. There are multiple init containers in this Pod spec that run in sequence - and because the Pods also start in sequence, you can guarantee that the first init container in Pod 1 won't run until Pod 0 is fully initialized and ready.

listing 8.2 - todo-db.yaml, the replicated Postgres setup with initialization

```

initContainers:
  - name: wait-service
    image: kiamol/ch03-sleep
    envFrom:
      - configMapRef:
          name: todo-db-env
    command: ['/scripts/wait-service.sh']
    volumeMounts:
      - name: scripts           # volume loads scripts from configmap
        mountPath: "/scripts"

```

The script which runs in this init container has two functions - if it's running in Pod 0 then it just prints a log to confirm that this is the database primary and then the container exits; if it's running in any other Pod, then it makes a DNS lookup call to the primary, to make sure its accessible before continuing. The next init container will start the replication process, so this one makes sure everything is in place.

The exact steps in this example are specific to Postgres, but the pattern is the same for many clustered and replicated applications - MySQL, Elasticsearch, RabbitMQ and NATS all have broadly similar requirements. Figure 8.4 shows how you can model that pattern using init containers in a StatefulSet.

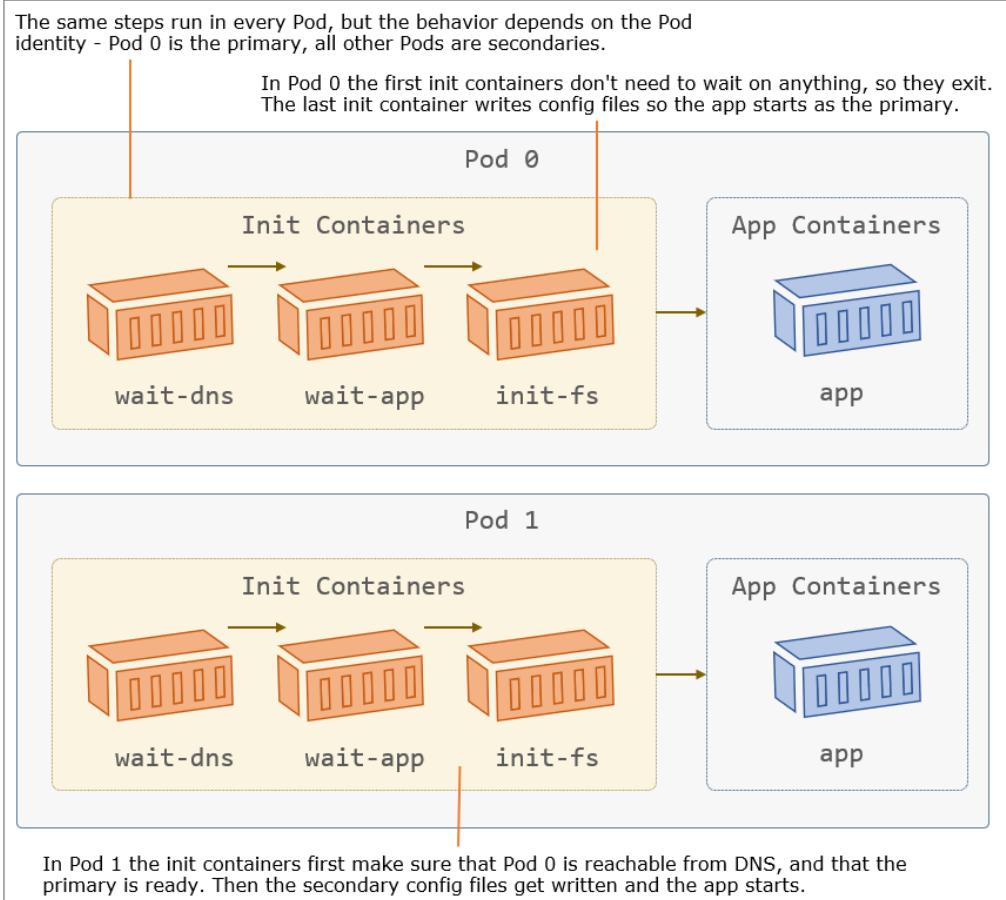


Figure 8.4 The stable environment of a StatefulSet gives guarantees you can use in initialization

You get DNS names for the individual Pods in a StatefulSet by identifying a Service in the spec, but it needs to be a special configuration of headless Service. Listing 8.3 shows how the database Service is configured with no ClusterIP address and with a selector for the Pods.

Listing 8.3 - todo-db-service.yaml, a headless Service for a StatefulSet

```
apiVersion: v1
kind: Service
metadata:
  name: todo-db
spec:
  selector:
    app: todo-db      # the Pod selector matches the StatefulSet
```

```
clusterIP: None      # the service will not get its own IP address
ports:
  # ports follow
```

A Service with no ClusterIP is still available as a DNS entry in the cluster, but it doesn't use a fixed IP address for the Service. There's no virtual IP which gets routed to the real destination by the networking layer - instead the DNS entry for the service returns an IP address for each Pod in the StatefulSet, and each Pod additionally gets its own DNS entry.

TRY IT NOW We've already deployed the headless Service - we can use a sleep deployment to query DNS for the StatefulSet and see how it compares to a typical ClusterIP service.

```
# show the Service details:
kubectl get svc todo-db

# run a sleep Pod to use for network lookups:
kubectl apply -f sleep/sleep.yaml

# run a DNS query for the Service name:
kubectl exec deploy/sleep -- sh -c 'nslookup todo-db | grep "^[^*]"'

# and a DNS lookup for Pod 0:
kubectl exec deploy/sleep -- sh -c 'nslookup todo-db-0.todo-
db.default.svc.cluster.local | grep "^[^*]"'
```

You'll see in this exercise that the DNS lookup for the service returns two IP addresses, which are the internal Pod IPs. The Pods themselves have their own DNS entry in the format `pod-name.service-name` with the usual cluster domain suffix. Figure 8.5 shows my output.

This Service doesn't have its own virtual IP address like a typical ClusterIP service.

```
PS>kubectl get svc todo-db
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
todo-db   ClusterIP  None           <none>        5432/TCP     26m
PS>
PS>kubectl exec deploy/sleep -- sh -c 'nslookup todo-db | grep "^[^*]"'
Server: 10.96.0.10
Address: 10.96.0.10:53
Name: todo-db.default.svc.cluster.local
Address: 10.1.2.56
Name: todo-db.default.svc.cluster.local
Address: 10.1.2.55
PS>
PS>kubectl exec deploy/sleep -- sh -c 'nslookup todo-db-0.todo-db.default.svc.cluster.local | grep "^[^*]"'
Server: 10.96.0.10
Address: 10.96.0.10:53
Name: todo-db-0.todo-db.default.svc.cluster.local
Address: 10.1.2.56
```

It still has a DNS entry, but the lookup returns the IP addresses for all the Pods.

For a StatefulSet linked to a Service, every Pod has its own DNS entry based on the Pod name.

Figure 8.5 StatefulSets give each Pod its own DNS entry, so they are individually addressable

Predictable startup order and individually addressable Pods is the foundation for initializing a clustered app in a StatefulSet. The details will differ wildly between applications, but very broadly the startup logic for the Pod will be something like this: if I am Pod 0 then I'm the primary, so I do all the primary setup stuff; otherwise I'm a secondary, so I'll give the primary some time to get set up, check that it's working and then synchronize using the Pod 0 address.

The actual setup for Postgres is quite involved so I'll skip over it here - it uses scripts in ConfigMaps with init containers to set up the primary and secondaries. I use various techniques we've covered in the book so far in the spec for the StatefulSet which is worth exploring, but the details of the scripts are all specific to Postgres.

TRY IT NOW Update the database to make it a replicated setup - there are configuration files and startup scripts in ConfigMaps, and the StatefulSet is updated to use them in init containers.

```
# deploy the replicated StatefulSet setup:
kubectl apply -f todo-list/db/replicated/

# wait for the pods to spin up
kubectl wait --for=condition=Ready pod -l app=todo-db

# check the logs of Pod 0 - the primary:
kubectl logs todo-db-0 --tail 1
```

```
# and of Pod 1 - the secondary:  
kubectl logs todo-db-1 --tail 2
```

Postgres uses an active-passive model for replication, so the primary is used for database reads and writes, the secondaries sync data from the primary and they can be used by clients, but only for read access. Figure 8.6 shows how the init containers recognize the role for each Pod and initialize them.

This deployment uses init containers, running scripts to establish if the Pod is the primary or a secondary.

Pod 0 has set itself up as the primary, and its available for clients to use for read and write access.

```
PS>kubectl apply -f todo-list/db/replicated/
configmap/todo-db-config created
configmap/todo-db-env created
configmap/todo-db-scripts created
statefulset.apps/todo-db configured
PS>
PS>kubectl wait --for=condition=Ready pod -l app=todo-db
pod/todo-db-0 condition met
pod/todo-db-1 condition met
PS>
PS>kubectl logs todo-db-0 --tail 1
2020-05-19 09:50:49.850 UTC [6] LOG: database system is ready to accept connect
ions
PS>
PS>kubectl logs todo-db-1 --tail 2
2020-05-19 09:56:29.745 UTC [10] LOG: database system is ready to accept read c
nly connections
2020-05-19 09:56:29.748 UTC [30] LOG: started streaming WAL from primary at 0/3
000000 on timeline 1
```

Pod 1 starts as a secondary - it synchronizes data from Pod 0 and is available for read-only access.

Figure 8.6 Pods are replicas, but they can have different behavior, using init containers to choose a role

Most of the complexity in initializing replicated apps like this is around modelling the workflow, which is specific to the app - the init container scripts here use the pg_isready tool to check the primary is ready to receive connections and the pb_basebackup tool to start the replication. Those implementation details are abstracted away from operators managing the system - they can add more replicas by scaling up the StatefulSet, like any other replication controller.

TRY IT NOW Scale up the database to add another replica, and confirm that that the new Pod also starts as a secondary.

```
# add another replica:
kubectl scale --replicas=3 statefulset/todo-db

# wait for Pod 2 to spin up
kubectl wait --for=condition=Ready pod -l app=todo-db

# check the new Pod sets itself up as another secondary:
kubectl logs todo-db-2 --tail 2
```

I wouldn't call this an enterprise-grade production setup, but it's a good starting point where a real Postgres expert could take over. You now have a functional, replicated Postgres database cluster with a primary and two secondaries - Postgres calls them *standbys*. As you can see in figure 8.7 all the standbys start in the same way, syncing data from the primary, and they can all be used by clients for read-only access.

The StatefulSet spec hides all the complexity of the underlying app, so scaling up is simple.

```
PS>kubectl scale --replicas=3 statefulset/todo-db
statefulset.apps/todo-db scaled
PS>
PS>kubectl wait --for=condition=Ready pod -l app=todo-db
pod/todo-db-0 condition met
pod/todo-db-1 condition met
pod/todo-db-2 condition met
PS>
PS>kubectl logs todo-db-2 --tail 2
2020-05-19 09:59:53.022 UTC [10] LOG:  database system is ready to accept read o
nly connections
2020-05-19 09:59:53.025 UTC [30] LOG:  started streaming WAL from primary at 0/5
000000 on timeline 1
```

Any Pods which aren't Pod 0 will initialize themselves as secondaries.
We can scale up to add as many secondaries as we need.

Figure 8.7 Individually addressable Pods means secondaries can always find the primary

There's one obvious part missing here - the actual storage of the data. The setup we have isn't really usable because it doesn't have any volumes for storage, so each database container writes data in its own writeable layer, and not in a persistent volume. StatefulSets have a very neat way of defining volume requirements - you can include a set of Persistent Volume Claim *templates* in the spec.

8.3 Requesting storage with volume claim templates

Volumes are part of the standard Pod spec, and you can load ConfigMaps and Secrets into the Pods for a StatefulSet - you can even include a PVC and mount it into the app container - but that gives you volumes which are shared between all the Pods. That's fine for read-only config settings where you want every Pod to have the same data, but if you mount a standard PVC for data storage, then every Pod will try to write to the same volume.

You actually want each Pod to have its own PVC, and Kubernetes provides that for StatefulSets with the `volumeClaimTemplates` field in the spec. Volume claim templates can include a storage class as well as capacity and access mode requirements. When you deploy a StatefulSet with volume claim templates it creates a PVC for each Pod, and they're linked - so if Pod 0 gets replaced, the new Pod 0 will attach to the PVC used by the previous Pod 0.

Listing 8.4 shows a simple sleep spec which uses volume claim templates. As we learned in chapter 5, different Kubernetes platforms offer different storage classes, and I can't be sure what your cluster provides. This spec omits the storage class, which means the volume will be dynamically provisioned using your cluster's default storage class.

Listing 8.4 - sleep-with-pvc.yaml, a StatefulSet with volume claim templates

```
spec:
  selector:
    # pod selector...
  serviceName:
    # headless service name...
  replicas: 2
  template:
    # pod template...

  volumeClaimTemplates:
    - metadata:
        name: data          # the name to use for volume mounts in the Pod
      spec:                 # this is a standard PVC spec
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 5Mi
```

We'll use this exercise to see how volume claim templates in StatefulSets work in a simple environment, before adding them as the storage layer for our database cluster.

TRY IT NOW Deploy the StatefulSet from listing 8.4 and explore the PVCs it creates.

```
# deploy the StatefulSet with volume claim templates:
kubectl apply -f sleep/sleep-with-pvc.yaml

# check the PVCs get created:
kubectl get pvc

# write some data to the PVC mount in Pod 0:
kubectl exec sleep-with-pvc-0 -- sh -c 'echo Pod 0 > /data/pod.txt'

# confirm Pod 0 can read the data:
kubectl exec sleep-with-pvc-0 -- cat /data/pod.txt

# and Pod 1 can't - this will fail:
kubectl exec sleep-with-pvc-1 -- cat /data/pod.txt
```

You'll see that each Pod in the set gets a PVC dynamically created, which in turn creates a PersistentVolume using the default storage class (or the requested storage class if I had included one in the spec). The PVCs all have the same configuration, and they use the same stable approach as Pods in the StatefulSet - they have a predictable name, and as you see in figure 8.8 each Pod has its own PVC giving the replicas independent storage.

Deploying a StatefulSet with volume claim templates creates a PVC for each Pod in the set; the PVC is dynamic so it creates a PersistentVolume.

```

PS>kubectl apply -f sleep/sleep-with-pvc.yaml
service/sleep-with-pvc created
statefulset.apps/sleep-with-pvc created
PS>
PS>kubectl get pvc
NAME           STATUS   VOLUME                                     CAPA
CITY  ACCESS MODES   STORAGECLASS   AGE
data-sleep-with-pvc-0 Bound    pvc-15f602b0-88e8-4a36-9d15-16c199983b02  5Mi
          RWO      hostpath        5s
data-sleep-with-pvc-1 Bound    pvc-cc969eba-e7b6-4ed6-9d22-bee6fd5cd617  5Mi
          RWO      hostpath        1s
PS>
PS>kubectl exec sleep-with-pvc-0 -- sh -c 'echo Pod 0 > /data/pod.txt'
PS>
PS>kubectl exec sleep-with-pvc-0 -- cat /data/pod.txt
Pod 0
PS>
PS>kubectl exec sleep-with-pvc-1 -- cat /data/pod.txt
cat: can't open '/data/pod.txt': No such file or directory
command terminated with exit code 1

```

Pods have the same volume mount in the spec, but each PVC is linked to one Pod, so they mount their own volumes.

Figure 8.8 Volume claim templates dynamically create storage for Pods in StatefulSets

The link between the Pod and its PVC is maintained when Pods get replaced, which is what really gives StatefulSets the power to run data-heavy applications. When you rollout an update to your app, the new Pod 0 will attach to the PVC from the previous Pod 0 and the new app container will have access to the exact same state as the replaced app container.

TRY IT NOW Trigger a Pod replacement by removing Pod 0 - it will be replaced with another Pod 0 which attaches to the same PVC.

```

# delete the Pod:
kubectl delete pod sleep-with-pvc-0

# check the replacement gets created:
kubectl get pods -l app=sleep-with-pvc

# and that the new Pod 0 can see the old data:

```

```
kubectl exec sleep-with-pvc-0 -- cat /data/pod.txt
```

This simple example makes that clear - you can see in figure 8.9 that the new Pod 0 has access to all the data from the original Pod. In a production cluster you would specify a storage class that uses a volume type which any node can access, so replacement Pods can run on any node and the app container can still mount the PVC.

The new Pod 0 is an exact replacement for the original Pod 0.

```

PS>kubectl delete pod sleep-with-pvc-0
pod "sleep-with-pvc-0" deleted
PS>
PS>kubectl get pods -l app=sleep-with-pvc
NAME          READY   STATUS    RESTARTS   AGE
sleep-with-pvc-0   1/1     Running   0          5s
sleep-with-pvc-1   1/1     Running   0          2m3s
PS>
PS>kubectl exec sleep-with-pvc-0 -- cat /data/pod.txt
Pod 0

```

It mounts the same PVC and the new app container can read the data written by the previous container.

Figure 8.9 Stability in a StatefulSet extends to preserving the PVC link between Pod replacements

Volume claim templates are the final piece we need to add to the Postgres deployment to model a fully reliable database. StatefulSets are intended to present a stable environment for your app, so they're less flexible than other controllers when it comes to updates - you can't update an existing StatefulSet and make a fundamental change, like adding volume claims. You need to make sure your design meets the app requirements for a StatefulSet because it's hard to maintain service levels during big changes.

TRY IT NOW We'll update the Postgres deployment, but first we need to remove the existing StatefulSet.

```

# apply the update with volume claim templates - this will fail:
kubectl apply -f todo-list/db/replicated/update/todo-db-pvc.yaml

# so delete the existing set:
kubectl delete statefulset todo-db

# and create a new one with volume claims:
kubectl apply -f todo-list/db/replicated/update/todo-db-pvc.yaml

# check the volume claims:
kubectl get pvc -l app=todo-db

```

When you run this exercise you should see clearly how the StatefulSet preserves order and waits for each Pod to be running before it starts the next Pod. PVCs get created for each Pod in sequence too - as you can see from my output in figure 8.10.

You can't make significant changes to an existing StatefulSet - it's meant to be a stable environment.

```

PS>kubectl apply -f todo-list/db/replicated/update/todo-db-pvc.yaml
The StatefulSet "todo-db" is invalid: spec: Forbidden: updates to statefulset spec for fields other than 'replicas', 'template', and 'updateStrategy' are forbidden
PS>
PS>kubectl delete statefulset todo-db
statefulset.apps "todo-db" deleted
PS>
PS>kubectl apply -f todo-list/db/replicated/update/todo-db-pvc.yaml
statefulset.apps/todo-db created
PS>
PS>kubectl get pvc -l app=todo-db
NAME           STATUS   VOLUME                                     CAPACITY
ACCESS MODES   STORAGECLASS   AGE
data-todo-db-0 Bound    pvc-8421f65a-a8cf-4007-8226-e725d69de206  50Mi
RWO          hostpath      59s
data-todo-db-1 Bound    pvc-3415556e-ea74-4421-acf2-44abbd1c5039  50Mi
RWO          hostpath      16s

```

Deploying a StatefulSet with volume claim templates creates a PVC for each Pod. They're created in order like the Pods, so Pod 1's PVC doesn't get created until Pod 0 is up and running.

Figure 8.10 PVCs are created and allocated to the Postgres Pods

It feels like we've spent a long time on StatefulSets, but it's a topic you should understand well so you're not taken by surprise when someone asks you to move their database to Kubernetes (which they will). StatefulSets come with a good deal of complexity and you'll avoid using them most of the time. But if you are looking to migrate existing apps to Kubernetes, StatefulSets could be the difference between being able to run everything on the same platform, or having to keep a handful of VMs just to run one or two apps.

We'll finish the section with an exercise to show the power of our clustered database. The Postgres secondary replicates all the data from the primary, and it can be used by clients for read-only access. If we had a serious production issue with our to-do list app which was causing it to lose data, we have the option to switch to read-only mode and use the secondary while we investigate the problem. That keeps the app running safely with minimal functionality - which is definitely preferable to taking it offline.

TRY IT NOW Run the to-do web app and enter some items. In the default configuration it connects to the Postgres primary in Pod 0 of the StatefulSet. Then we'll switch the app configuration to put it into read-only mode - which makes it connect to the read-only Postgres standby in Pod 1, which has replicated all the data from Pod 0.

```
# deploy the web app:
```

```
kubectl apply -f todo-list/web/  
  
# get the URL for the app:  
kubectl get svc todo-web -o  
jsonpath='http://{.status.loadBalancer.ingress[0].*}:8081/new'  
  
# browse and add a new item  
  
# switch to read-only mode, using the database secondary:  
kubectl apply -f todo-list/web/update/todo-web_READONLY.yaml  
  
# refresh the app - the /new page is read-only;  
# browse to /list and you'll see your original data  
  
# check there are no clients using the primary in Pod 0:  
kubectl exec -it todo-db-0 -- sh -c "psql -U postgres -t -c 'SELECT datname, query  
FROM pg_stat_activity WHERE datid > 0'"  
  
# and that the web app really is using the secondary in Pod 1:  
kubectl exec -it todo-db-1 -- sh -c "psql -U postgres -t -c 'SELECT datname, query  
FROM pg_stat_activity WHERE datid > 0'"
```

You can see my output in figure 8.11, with some tiny screenshots to show the app running in read-only mode, but still with access to all the data.

Starts the web app - this configuration uses the Pod 0 address for the database StatefulSet for read/write access.

Browse to the new item page and you can add data.

```

PS>kubectl apply -f todo-list/web/
configmap/todo-web-config created
secret/todo-web-secret created
service/todo-web created
deployment.apps/todo-web created
PS>
PS>kubectl get svc todo-web -o jsonpath='http://{{.status.loadBalancer.ingress[0]}}:8081/new'
http://localhost:8081/new
PS>
PS>kubectl apply -f todo-list/web/update/todo-web-readonly.yaml
service/todo-web unchanged
deployment.apps/todo-web configured
PS>
PS>kubectl exec -it todo-db-0 -- sh -c "psql -U postgres -t -c 'SELECT datname,
query FROM pg_stat_activity WHERE datid > 0'"
postgres | SELECT datname, query FROM pg_stat_activity WHERE datid > 0

PS>
PS>kubectl exec -it todo-db-1 -- sh -c "psql -U postgres -t -c 'SELECT datname,
query FROM pg_stat_activity WHERE datid > 0'"
todo | SELECT t."ToDoId", t."DateAdded", t."Item"
      | FROM "ToDos" AS t
postgres | SELECT datname, query FROM pg_stat_activity WHERE datid > 0

```

In read-only mode the web app connects to Pod 1, the database standby.

Both database Pods are running, but this confirms only the secondary has any clients.

Figure 8.11 Switching an app to read-only mode is a useful option if there's a data issue

Postgres has existed as a SQL database engine since 1996 - it pre-dates Kubernetes by 20 years. Using a StatefulSet you can model an application environment which suits Postgres and other clustered applications like it, providing stable networking, storage and initialization in the dynamic world of containers.

8.4 Running maintenance tasks with Jobs and CronJobs

Data-intensive apps need replicated data with storage aligned to compute, and they usually also need some independent nurturing of the storage layer. Data backups and reconciliation are well suited to another type of Pod controller: the Job. Kubernetes Jobs are defined with a Pod spec, and they run the Pod as a batch job, ensuring it runs to completion.

Jobs aren't just for stateful apps, they're a great way to bring a standard approach to any batch processing problems, where you can hand off all the scheduling, monitoring and retry logic to the cluster. You can run any container image in the Pod for a Job - but it should start a process which ends, otherwise your jobs will keep running forever. Listing 8.5 shows a Job spec which runs the Pi application in batch mode.

Listing 8.5 - pi-job.yaml, a simple Job to calculate Pi

```
apiVersion: batch/v1
kind: Job                                # Job is the object type
metadata:
  name: pi-job
spec:
  template:
    spec:                                     # the standard Pod spec
      containers:
        - name: pi                         # container should run and exit
          image: kiamol/ch05-pi
          command: ["dotnet", "Pi.Web.dll", "-m", "console", "-dp", "50"]
    restartPolicy: Never                  # if the container fails, replace the Pod
```

The Job template contains a standard Pod spec, with the addition that the `restartPolicy` field is required. That controls the behavior of the Job in response to failure. You can choose to have Kubernetes restart the same Pod with a new container if the run fails, or always create a replacement Pod - potentially on a different node. In a normal run of the Job where the Pod completes successfully, the Job and the Pod are retained so the container logs are available.

TRY IT NOW Run the Pi job from listing 8.5 and check the output from the Pod.

```
# deploy the Job:
kubectl apply -f pi/pi-job.yaml

# check the logs for the Pod:
kubectl logs -l job-name=pi-job

# and the status of the Job:
kubectl get job pi-job
```

Jobs add their own labels to the Pods they create - the `job-name` label is always added, so you can navigate to Pods from the Job. My output in figure 8.12 shows that the Job has had one successful completion and the calculation result is available in the logs.

Jobs are Pod controllers - this spec creates a Job which creates a single Pod to run a Pi calculation.

The Pod is retained after the Job completes, so you can see its logs and status.

```
PS>kubectl apply -f pi/pi-job.yaml
job.batch/pi-job created
PS>
PS>kubectl logs -l job-name=pi-job
3.14159265358979323846264338327950288419716939937510
PS>
PS>kubectl get job pi-job
NAME      COMPLETIONS      DURATION      AGE
pi-job    1/1              3s            16s
```

Jobs can be configured to run multiple Pods - this is set for a single run, which has completed.

Figure 8.12 Jobs create Pods and make sure they complete and then leave them in the cluster

It's always useful to have different options for computing Pi, but this is just a simple example - you can use any container image in the Pod spec so you can run any kind of batch process with a Job. You might have a set of input items which need the same work done on them; you can create one Job for the whole set, which creates a Pod for each item, and Kubernetes distributes the work all around the cluster. The Job spec supports this with two optional fields:

- `completions` - specifies how many times the Job should run - if your Job is processing a work queue then the app container needs to understand how to fetch the next item to work on; the Job itself just ensures that it runs a number of Pods equal to the desired number of completions.
- `parallelism` - specifies how many Pods to run in parallel for a Job with multiple completions set. This lets you tweak the speed of running the Job, balancing that with the compute requirements on the cluster.

One last Pi example for this chapter - a new Job spec which runs multiple Pods in parallel, each computing Pi to a random number of decimal places. This spec uses an init container to generate the number of decimal places to use, and the app container reads that input using a shared `EmptyDir` mount. That's a nice approach which means the app container doesn't need to be modified to work in a parallel environment - you could extend that with an init container that fetched a work item from a queue, so the app itself wouldn't need to be aware of the queue.

TRY IT NOW Run an alternative Pi job which uses parallelism and shows that multiple Pods from the same spec can process different workloads.

```
# deploy the new Job:
kubectl apply -f pi/pi-job-random.yaml

# check the Pod status:
kubectl get pods -l job-name=pi-job-random

# and the Job status:
kubectl get job pi-job-random

# list the Pod output:
kubectl logs -l job-name=pi-job-random
```

This exercise may take a while to run, depending on your hardware and the number of decimal places it generates. You'll see all the Pods running in parallel, working on their own calculations. The final output will be three sets of Pi, probably to thousands of decimal places - I've abbreviated my results in figure 8.13.

This Job manages multiple Pods - it's configured to run three completions, with a parallelism value of three.

All three Pods start simultaneously. On a multi-node cluster these would be distributed across nodes.

```
PS>kubectl apply -f pi/pi-job-random.yaml
job.batch/pi-job-random created
PS>
PS>kubectl get pods -l job-name=pi-job-random
NAME          READY   STATUS      RESTARTS   AGE
pi-job-random-bmxdd  0/1     Completed   0          6s
pi-job-random-qb9wg  0/1     PodInitializing   0          6s
pi-job-random-wnn8r  0/1     PodInitializing   0          6s
PS>
PS>kubectl get job pi-job-random
NAME          COMPLETIONS   DURATION   AGE
pi-job-random  3/3           9s         19s
PS>
PS>kubectl logs -l job-name=pi-job-random
3.141592653589793238462643383279502884197169399375105820974944592307816406286208
99862803482534211706798214808651328230664709384460955058223172535940812848111745
```

When the Job completes you can see the output from all the Pods in their logs - these are calculating Pi to a random number of decimal places, so the output is quite large.

Figure 8.13 Jobs can run multiple Pods from the same spec which each process different workloads

Jobs are a great tool to have in your pocket. They're perfect for anything compute-intensive or IO-intensive, where you want to make sure a process completes but you don't really mind when. You can even submit Jobs from your own application - a web app running in Kubernetes has access to the Kubernetes API server, and it can create Jobs to run work for users.

The real power of Jobs is that they run in the context of the cluster, so they have all the cluster resources available to them. Back to the Postgres example, we can run a database

backup process in a Job and the Pod it runs can access the Pods in the StatefulSet or the PVCs, depending on what it needs to do. That takes care of the nurturing aspect of these data-intensive apps, but those Jobs need to be run regularly, which is where the CronJob comes in. The CronJob is a Job controller, which creates Jobs on a regular schedule - figure 8.14 shows the workflow.

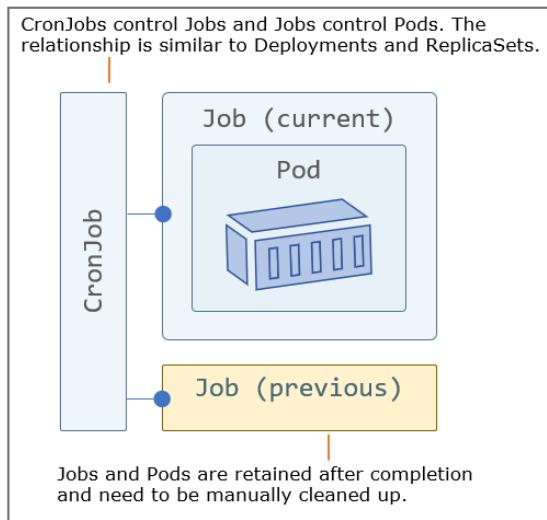


Figure 8.14 CronJobs are the ultimate owner of the Job Pods, so everything can be removed with cascading deletes

CronJob specs include a Job spec - so you can do anything in a CronJob that you can do in a Job, including running multiple completions in parallel. The schedule for running the Job uses the Linux Cron format - which lets you express everything from simple "every minute" or "every day" schedules, to more complex "at 4 a.m. and 6 a.m. every Sunday" routines. Listing 8.6 shows part of the CronJob spec for running database backups.

Listing 8.6 - todo-db-backup-cronjob.yaml, a CronJob for database backups

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: todo-db-backup
spec:
  schedule: "*/2 * * * *"          # this creates a Job every 2 minutes
  concurrencyPolicy: Forbid        # prevents overlap so new Job won't be
  jobTemplate:                     # created if the previous one is running
    spec:
      # job template...
```

The full spec uses the Postgres Docker image, with a command to run the `pg_dump` backup tool. The Pod loads environment variables and passwords from the same ConfigMaps and Secrets that the StatefulSet uses, so there's no duplication in the config. It also uses its own PVC as the storage location to write the backup files.

TRY IT NOW Create a CronJob from the spec in listing 8.6 to run a database backup Job every two minutes.

```
# deploy the CronJob and target PVC for backup files:  
kubectl apply -f todo-list/db/backup/  
  
# wait for the Job to run - this is a good time to make tea:  
sleep 150  
  
# check the CronJob status:  
kubectl get cronjob todo-db-backup  
  
# now run a sleep Pod which mounts the backup PVC:  
kubectl apply -f sleep/sleep-with-db-backup-mount.yaml  
  
# and check if the CronJob Pod created the backup:  
kubectl exec deploy/sleep -- ls -l /backup
```

The CronJob is set to run every two minutes, so you'll need to give it time to fire up during this exercise. On schedule the CronJob creates a Job which creates a Pod which runs the backup command. The Job ensures the Pod completes successfully. You can check the backup file gets written by mounting the same PVC in another Pod, and you can see it all works correctly in figure 8.15.

This CronJob is set to run every two minutes - the most recent Job was created 74 seconds ago.

```

PS>kubectl apply -f todo-list/db/backup/
cronjob.batch/todo-db-backup created
persistentvolumeclaim/todo-db-backup-pvc created
PS>
PS>sleep 150
PS>
PS>kubectl get cronjob todo-db-backup
NAME          SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
todo-db-backup  * /2 * * * *   False     0        74s  2m35s
PS>
PS>kubectl apply -f sleep/sleep-with-db-backup-mount.yaml
deployment.apps/sleep configured
PS>
PS>kubectl exec deploy/sleep -- ls -l /backup
total 8
-rw-r--r--    1 root      root         8192 May 19 20:06 200519-2006.tar

```

Running a Pod which mounts the same PVC the backup Job uses - the tar file was created by the backup Pod, by connecting to one of the Postgres read-only replicas.

Figure 8.15 CronJobs run Pods which can access other Kubernetes objects - this one connects to a database Pod

There's no automatic cleanup for Pods and Jobs created by CronJobs. There is actually a controller which can do that, called the time-to-live (TTL) controller, but it's an alpha-grade feature which isn't available in many Kubernetes platforms. Without it you need to manually delete the child objects when you're sure you no longer need them. You can also move CronJobs to a suspended state, which means the object spec still exists in the cluster, but it doesn't run until the CronJob is activated again.

TRY IT NOW Suspend the CronJob so it doesn't keep creating backup Jobs and then explore the status of the CronJob and its Jobs.

```

# update the CronJob and set it to suspend:
kubectl apply -f todo-list/db/backup/update/todo-db-backup-cronjob-suspend.yaml

# check the CronJob:
kubectl get cronjob todo-db-backup

# and find the Jobs owned by the CronJob:
kubectl get jobs -o jsonpath="{.items[?(@.metadata.ownerReferences[0].name=='todo-db-backup')].metadata.name}"

```

If you explore the object hierarchy, you'll see that CronJobs don't follow the standard controller model, with a label selector to identify the Jobs it owns. You can add your own

labels in the Job template for the CronJob, but if you don't do that then you need to identify Jobs where the owner reference is the CronJob, as you see in figure 8.16.

Setting a CronJob to suspended retains the spec and any previous jobs, but it won't run any new jobs.

```
PS>kubectl apply -f todo-list/db/backup/update/todo-db-backup-cronjob-suspend.yaml
cronjob.batch/todo-db-backup configured
PS>
PS>kubectl get cronjob todo-db-backup
NAME          SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
todo-db-backup  */2 * * * *  True        0        40s           4m1s
PS>
PS>kubectl get jobs -o jsonpath=".items[?(@.metadata.ownerReferences[0].name=='todo-db-backup')].metadata.name"
todo-db-backup-1589918760 todo-db-backup-1589918880
PS>
PS>kubectl get pod -l job-name=todo-db-backup-1589918760
NAME                           READY   STATUS    RESTARTS   AGE
todo-db-backup-1589918760-s8gg2  0/1     Completed  0          4m26s
```

CronJobs don't create a label on Jobs the way Jobs do for Pods, so you need to find them through the owner reference.

Completed Jobs and their Pods are retained.

Figure 8.16 CronJobs don't use a label selector to model ownership, because they don't keep track of Jobs

As you start to make more use of Jobs and CronJobs you'll realize that the simplicity of the spec masks some complexity in the process, and presents some interesting failure modes. Kubernetes does its best to make sure your batch jobs start when you want them to, and run to completion - which means your containers need to be resilient. Completing a Job might mean restarting a Pod with a new container or replacing the Pod on a new node, and for CronJobs there could be multiple Pods running if the process takes longer to run than the schedule interval. Your container logic needs to allow for all those scenarios.

So now you know how to run data-heavy apps in Kubernetes, with StatefulSets to model a stable runtime environment and initialize the app, and CronJobs to process data backups and other regular maintenance work. We'll close out the chapter thinking about whether this is really a good idea.

8.5 Choosing your platform for stateful apps

The great promise of Kubernetes is that it gives you a single platform that can run all your apps, on any infrastructure. It's hugely appealing to think that you can model all the aspects of any application in a chunk of YAML, deploy it with some Kubectl commands and know it will run in the same way on any cluster, taking advantage of all the extended features the

platform offers. But data is precious and usually irreplaceable, so you need to think carefully before you decide that Kubernetes is the place to run data-heavy apps.

Figure 8.17 shows the full setup we've built out in this chapter to run an almost-production grade SQL database in Kubernetes. Just look at all the moving parts there are - do you really want to manage all that? And how much time will you need to invest just testing this setup with your own data sizes - validating that the replicas are syncing correctly, verifying the backups can be restored, running chaos experiments to be sure that failures are handled in the way you expect?

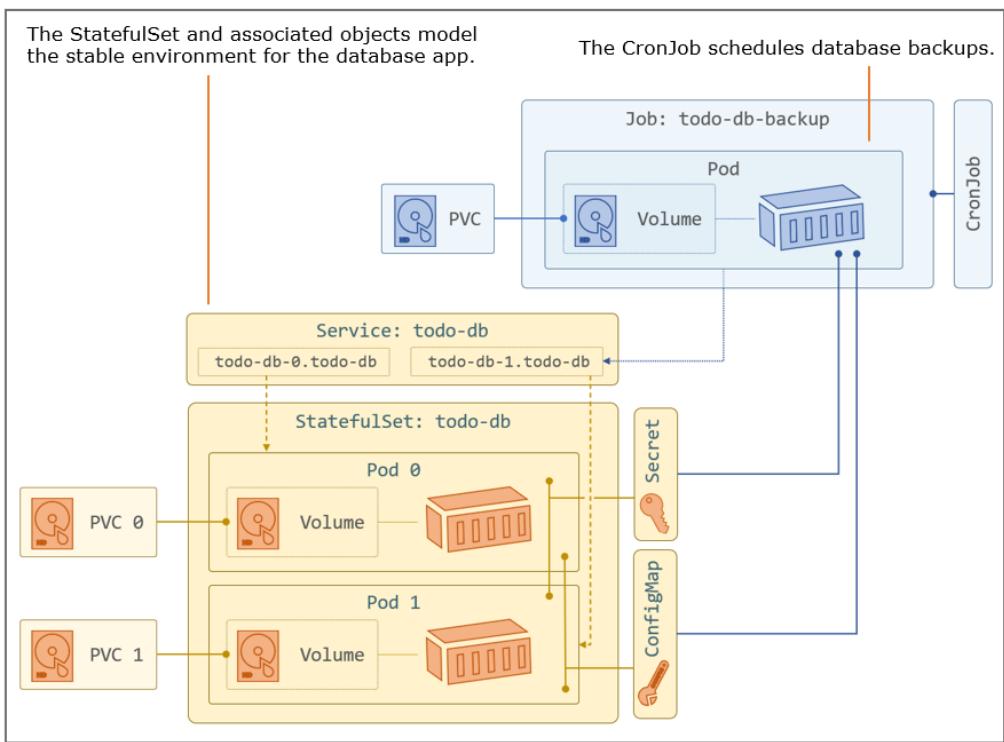


Figure 8.17 Yikes. And this is a simplification which doesn't show volumes or init containers.

Compare that to a managed database in the cloud - Azure, AWS and GCP all offer managed services for Postgres, MySQL and SQL Server, as well as their own custom cloud-scale databases. The cloud provider takes care of scale and high availability, including features for backups to cloud storage, and more advanced options like threat detection. An alternative architecture just uses Kubernetes for compute, and plugs in to managed cloud services for data and communication.

Which is the better option? Well, I'm a consultant by day and I know the only real answer is: "it depends". If you're running in the cloud then I think you need a very good reason *not*

to use managed services in production, where data is critical. In non-production environments it often makes sense to run equivalent services in Kubernetes instead - so you run a containerized database and message queue in your dev and test environments for lower costs and ease of deployment, and swap out to managed versions in production. Kubernetes makes a swap like that very simple with all the Pod and Service configuration options.

In the data center the picture is a little different. If you're already invested in running Kubernetes on your own infrastructure, then you're taking on a lot of management and it might make sense to maximize utilization of your clusters and use them for everything. If you choose to go that way, Kubernetes gives you the tools to migrate data-heavy apps into the cluster and run them with the levels of availability and scale that you need. Just don't underestimate the complexity of getting there.

We're done with StatefulSets and Jobs now, so we can clean up before going on to the lab.

TRY IT NOW All the top-level objects are labelled, so we can remove everything with cascading deletes.

```
# delete all the owning objects:  
kubectl delete all -l kiamol=ch08  
  
# and the PVCs  
kubectl delete pvc -l kiamol=ch08
```

8.6 Lab

So, how much of your lunchtime do you have left for this lab? Can you model a MySQL database from scratch, with backups? Probably not, but don't worry - this lab isn't so involved as that. The goal is just to give you some experience working with StatefulSets and PVCs, so we'll use a much simpler app. You're going to run the Nginx web server in a StatefulSet, where each Pod writes log files to a PVC of its own; and then to run a job which prints the size of each Pod's log file. The basic pieces are there for you, so it's about applying some of the techniques from the chapter.

- the starting point is the Nginx spec in `ch08/lab/nginx` - that runs a single Pod writing logs to an EmptyDir volume
- the Pod spec needs to be migrated to a StatefulSet definition, which is configured to run with three Pods and provide separate storage for each of them
- when you have the StatefulSet working, you should be able to make calls to your Service and see the log files being written in the Pods
- then you can complete the Job spec in the file `disk-calc-job.yaml`, adding the volume mounts so it can read the log files from the Nginx Pods.

It's not as bad as it looks, and it will get you thinking about storage and Jobs. My solution is up on GitHub for you to check in the usual place:

<https://github.com/sixeyed/kiamol/blob/master/ch08/lab/README.md>

9

Managing app releases with rollouts and rollbacks

You'll update existing apps far more often than you'll deploy something new. Containerized apps inherit multiple release cadences from the base images they use - official images on Docker Hub for operating systems, platform SDKs and runtimes typically have a new release every month. You should have a process to rebuild your images and release updates whenever those dependencies get updated, because they could contain critical security patches. Key to that process is being able to roll out an update safely, and give yourself options to pause and rollback the update if it goes wrong. Kubernetes has those scenarios covered for Deployments, DaemonSets and StatefulSets.

There isn't a single update approach which works for every type of application, so Kubernetes provides different update strategies for the controllers, and options to tune how the strategies work. We'll explore all those options in this chapter. If you're thinking of skipping this one because you're not excited by the thought of 6,000 words on application updates - I'd recommend sticking with it. Updates are the biggest cause of application downtime, but you can reduce the risk significantly if you understand the tools Kubernetes gives you. And I'll try to inject a little excitement along the way.

9.1 How Kubernetes manages rollouts

We'll start with Deployments, and actually you've already done plenty of Deployment updates. Every time we've applied a change to an existing Deployment (something we do 10 times a chapter), Kubernetes has implemented that with a rollout. In a rollout the Deployment creates a new ReplicaSet and scales it up to the desired number of replicas, while scaling down the previous ReplicaSet to zero replicas. Figure 9.1 shows an update in progress.

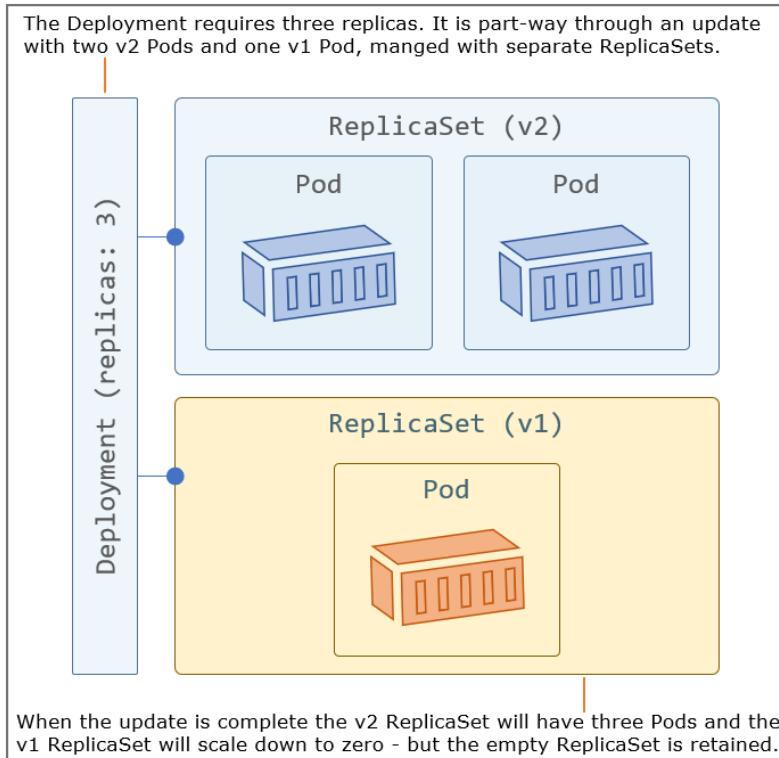


Figure 9.1 Deployments control multiple ReplicaSets so they can manage rolling updates

Rollouts aren't triggered from every change to a Deployment - only from a change to the Pod spec. If you make a change which the Deployment can manage with the current ReplicaSet, like updating the number of replicas, that's done without a rollout.

TRY IT NOW Deploy a simple app with two replicas, then update it to increase scale and see how the ReplicaSet is managed.

```
# change to the exercise directory:  
cd ch09  
  
# deploy a simple web app:  
kubectl apply -f vweb/  
  
# check the ReplicaSets:  
kubectl get rs -l app=vweb  
  
# now increase the scale:  
kubectl apply -f vweb/update/vweb-v1-scale.yaml
```

```
# check the ReplicaSets:  
kubectl get rs -l app=vweb  
  
# and the deployment history:  
kubectl rollout history deploy/vweb
```

The Kubectl `rollout` command has options to view and manage rollouts. You can see from my output in figure 9.2 that there's only one rollout in this exercise, which was the initial deployment that created the ReplicaSet. The scale update only changed the existing ReplicaSet so there was no second rollout.

This Deployment requires two Pods which it manages through the ReplicaSet.

```
PS>cd ch09
PS>
PS>kubectl apply -f vweb/
service/vweb created
deployment.apps/vweb created
PS>
PS>kubectl get rs -l app=vweb
NAME      DESIRED   CURRENT   READY   AGE
vweb-668f7b746f   2         2         2       7s
PS>
PS>kubectl apply -f vweb/update/vweb-v1-scale.yaml
deployment.apps/vweb configured
PS>
PS>kubectl get rs -l app=vweb
NAME      DESIRED   CURRENT   READY   AGE
vweb-668f7b746f   3         3         3      20s
PS>
PS>kubectl rollout history deploy/vweb
deployment.apps/vweb
REVISION  CHANGE-CAUSE
1        <none>
```

Updating the scale is done with the existing ReplicaSet.

The rollout history shows updates to the Deployment - scaling the Pods doesn't cause a rollout, so only the initial deployment is shown.

Figure 9.2 Deployments manage changes through rollouts, but only if the Pod spec changes

Your ongoing application updates will center around deploying new Pods running an updated version of your container image. You should manage that with an update to your YAML specs, but Kubectl provides a quick alternative with the `set` command. It's an imperative way to update an existing Deployment and you should view it in the same way as the `scale` command - it's a

useful hack to get out of a sticky situation, but it needs to be followed up with an update to the YAML files.

TRY IT NOW Use Kubectl to update the image version for the Deployment - this is a change to the Pod spec so it will trigger a new rollout.

```
# update the image for the web app:  
kubectl set image deployment/vweb web=kiamol/ch09-vweb:v2  
  
# check the ReplicaSets again:  
kubectl get rs -l app=vweb  
  
# and the rollouts:  
kubectl rollout history deploy/vweb
```

The Kubectl `set` command changes the spec of an existing object - you can use it to change the image or environment variables for a Pod, or the selector for a Service. It's a short-cut to applying a new YAML spec but it gets implemented in the same way. In this exercise the change caused a rollout, with a new ReplicaSet created to run the new Pod spec and the old ReplicaSet scaled down to zero. You can see that in figure 9.3.

Updating the container image with the `set` command changes the Pod spec.

```
PS>kubectl set image deployment/vweb web=kiamol/ch09-vweb:v2  
deployment.apps/vweb image updated
```

PS>

```
PS>kubectl get rs -l app=vweb  
NAME      DESIRED   CURRENT   READY   AGE  
vweb-668f7b746f  0          0          0    9m25s  
vweb-f9f66fb    3          3          3    8s
```

PS>

```
PS>kubectl rollout history deploy/vweb
```

deployment.apps/vweb	REVISION	CHANGE-CAUSE
	1	<none>
	2	<none>

Changing the Pod spec for a Deployment creates a new ReplicaSet.

The new ReplicaSet is scaled up and the old one is scaled down, so this is the second rollout.

Figure 9.3 Imperative updates go through the same rollout process, but now your YAML is out of sync

Kubernetes uses the same concept of rollouts for other Pod controllers: DaemonSets and StatefulSets. They're an odd part of the API because they don't map directly to an object (you

don't create a resource with the kind "rollout"), but they're an important management tool to work with your releases. You can use rollouts to track release history and to revert back to previous releases.

9.2 Updating Deployments with rollouts and rollbacks

If you look again at figure 9.3 you'll see the rollout history is pretty unhelpful - there's a revision number recorded for each rollout, but nothing else. It's not clear what caused the change, or which ReplicaSet relates to which revision. It's good to include a version number (or a changeset ID) as a label for the Pods, and then the Deployment adds that label to the ReplicaSet too, which makes it easier to trace updates.

TRY IT NOW Apply an update to the Deployment, which uses the same Docker image but changes the version label for the Pod. That's a change to the Pod spec so it will create a new rollout.

```
# apply the change using the record flag:
kubectl apply -f vweb/update/vweb-v11.yaml --record

# check the ReplicaSets and their labels:
kubectl get rs -l app=vweb --show-labels

# see the current rollout status:
kubectl rollout status deploy/vweb

# and the rollout history:
kubectl rollout history deploy/vweb

# show the rollout revision for the ReplicaSets:
kubectl get rs -l app=vweb -o=custom-
  columns=NAME:.metadata.name,REPLICAS:.status.replicas,REVISION:.metadata.annotations.deployment\.kubernetes\.io/revision
```

You can see my output in figure 9.4. Adding the `record` flag saves the Kubectl command as a detail to the rollout, which can be helpful if your YAML files have identifying names. Often they won't because you'll be deploying a whole folder, so the version number label in the Pod spec is a useful addition - but then you need some awkward JSONPath to find the link between a rollout revision and a ReplicaSet.

The record flag saves the details of the Kubectl command for the rollout.

The Pod specs include a version label, which the Deployment adds to the ReplicaSet labels.

```

PS>kubectl apply -f vweb/update/vweb-v11.yaml --record
deployment.apps/vweb configured
PS>
PS>kubectl get rs -l app=vweb --show-labels
NAME      DESIRED   CURRENT   READY   AGE     LABELS
vweb-656cbc5cc4  3          3          3    4s     app=vweb,pod-template-hash=656cbc5cc4,version=v1.1
vweb-668f7b746f  0          0          0    47m    app=vweb,pod-template-hash=68f7b746f,version=v1
vweb-f9f66fb     0          0          0    38m    app=vweb,pod-template-hash=f9f66fb,version=v1
PS>
PS>kubectl rollout status deploy/vweb
deployment "vweb" successfully rolled out
PS>
PS>kubectl rollout history deploy/vweb
deployment.apps/vweb
REVISION  CHANGE-CAUSE
1          <none>
2          <none>
3          kubectl.exe apply --filename=vweb/update/vweb-v11.yaml --record=true
PS>kubectl get rs -l app=vweb -o=custom-columns=NAME:.metadata.name,REPLICAS:.status.replicas,REVISION:.metadata.annotations.deployment\\.kubernetes\\.io/revision
NAME      REPLICAS  REVISION
vweb-656cbc5cc4  3          3
vweb-668f7b746f  0          1
vweb-f9f66fb     0          2

```

Revision 3 includes the Kubectl command which triggered it, because of the record flag.

The revision number is stored as an annotation on the ReplicaSet, which needs some tricky JSONPath to see.

Figure 9.4 Kubernetes uses labels for key information, and extra detail is stored in annotations

As your Kubernetes maturity increases, you'll want to have a standard set of labels which you include in all your object specs. Labels and selectors are a core feature and you'll use them all the time to find and manage objects. Application name, component name and version are good labels to start with - but it's important to distinguish between the labels you include for your convenience, and the labels which Kubernetes uses to map object relationships.

Listing 9.1 shows the Pod labels and the selector for the Deployment in the last exercise. The `app` label is used in the selector, which the Deployment uses to find its Pods. The Pod also contains a `version` label for our convenience - but that's not part of the selector. If it was, then the Deployment would be linked to one version, because you can't change the selector once a Deployment is created.

Listing 9.1 - vweb-v11.yaml, a Deployment with additional labels in the Pod spec

```
spec:
  replicas: 3
  selector:
    matchLabels:
      app: vweb          # the app name is used as the selector
  template:
    metadata:
      labels:
        app: vweb
        version: v1.1     # the Pod spec also includes a version label
```

So you need to plan your selectors carefully up-front, but you should add whatever labels you need to your Pod spec to make your updates manageable. Deployments retain multiple ReplicaSets (10 is the default), and the Pod template hash in the name makes them hard to work with directly, even after just a few updates. Let's see what the app we've deployed actually does and then look at the ReplicaSets in another rollout.

TRY IT NOW Make an HTTP call to the web Service to see the response, then start another update and check the response again.

```
# we'll use the app URL a lot, so save it to a local file:
kubectl get svc vweb -o
  jsonpath='http://{.status.loadBalancer.ingress[0].*:8090/v.txt' > url.txt

# then use the contents of the file to make an HTTP request:
curl $(cat url.txt)

# deploy the v2 update:
kubectl apply -f vweb/update/vweb-v2.yaml --record

# check the response again:
curl $(cat url.txt)

# and the ReplicaSet details:
kubectl get rs -l app=vweb --show-labels
```

You'll see in this exercise that ReplicaSets aren't easy objects to manage - which is where standardised labels come in. It's easy to see which version of the app is active by checking the labels for the ReplicaSet which has all the desired replicas - as you see in figure 9.5 - but labels are just text fields so you need process safeguards to make sure they're reliable.

Using some Kubectl magic to write the app URL to a file, and then calling the app by reading that URL from the file. v1 of the app returns the response "v1".

```

PS>kubectl get svc vweb -o jsonpath='http://{.status.loadBalancer.ingress[0].*}:8090/v.txt' > url.txt
PS>
PS>curl $(cat url.txt)
v1
PS>
PS>kubectl apply -f vweb/update/vweb-v2.yaml --record
deployment.apps/vweb configured
PS>
PS>curl $(cat url.txt)
v2
PS>
PS>kubectl get rs -l app=vweb --show-labels
NAME          DESIRED   CURRENT   READY   AGE    LABELS
vweb-656cbc5cc4   0         0         0     39m   app=vweb,pod-template-hash
=656cbc5cc4,version=v1.1
vweb-668f7b746f   0         0         0     86m   app=vweb,pod-template-hash
=668f7b746f,version=v1
vweb-7659d9d47f   3         3         3     2m28s  app=vweb,pod-template-hash
=7659d9d47f,version=v2
vweb-f9f66fb      0         0         0     77m   app=vweb,pod-template-hash
=f9f66fb,version=v1

```

This updates the app to v2, which returns the response "v2".

Which is which? ReplicaSets are shown in name order, which is by the Pod template hash. In this case the active ReplicaSet is third in the list, and without the version label we wouldn't know what those Pods are running.

Figure 9.5 Kubernetes manages rollouts for you, but it helps if you add labels to see what's what

Rollouts do help to abstract away the details of the ReplicaSets, but their main use is to manage releases. We've seen the rollout history from Kubectl, and you can also run commands to pause an ongoing rollout, or roll back a deployment to an earlier revision. A simple command will roll back to the previous deployment, but if you want to roll back to a specific version you need some more JSONPath trickery to find the revision you want. We'll see that now and use a very handy feature of Kubectl which tells you what will happen when you run a command, without actually executing it.

TRY IT NOW Check the rollout history and try rolling back to v1 of the app.

```

# look at the revisions:
kubectl rollout history deploy/vweb

# list ReplicaSets with their revisions:
kubectl get rs -l app=vweb -o=custom-
  columns=NAME:.metadata.name,REPLICAS:.status.replicas,VERSION:.metadata.labels

```

```
.version,REVISION:.metadata.annotations.deployment\.kubernetes\.io/revision  
# see what would happen with a rollback:  
kubectl rollout undo deploy/vweb --dry-run  
  
# and then start a rollback to revision 2:  
kubectl rollout undo deploy/vweb --to-revision=2  
  
# check the app - this should surprise you:  
curl $(cat url.txt)
```

Hands up if you ran that exercise and got confused when you hit the final output you see in figure 9.6 (this is the exciting part of the chapter). My hand is up and I already knew what was going to happen. This is why you need a consistent release process, preferably one which is fully automated, because as soon as you start mixing approaches you get confusing results. I rolled back to revision 2, and that should have reverted back to v1 of the app judging by the labels on the ReplicaSets. But revision 2 was actually from the kubectl set image exercise in section 9.1, so the container image is v2 but the ReplicaSet label is v1.

There are four revisions of the app, which are the initial rollout and three updates.

You can print custom columns from the ReplicaSet to see the revisions matched to the version labels.

```
PS>kubectl rollout history deploy/vweb
deployment.apps/vweb
REVISION  CHANGE-CAUSE
1          <none>
2          <none>
3          kubectl.exe apply --filename=vweb/update/vweb-v11.yaml --record=true
4          kubectl.exe apply --filename=vweb/update/vweb-v2.yaml --record=true

PS>kubectl get rs -l app=vweb -o=custom-columns=NAME:.metadata.name,REPLICAS:.status.replicas,VERSION:.metadata.labels.version,REVISION:.metadata.annotations.deployment\.kubernetes\.io/revision
NAME        REPLICAS  VERSION  REVISION
vweb-656cbc5cc4  0         v1.1     3
vweb-668f7b746f  0         v1       1
vweb-7659d9d47f  3         v2       4
vweb-f9f66fb    0         v1       2
PS>
PS>kubectl rollout undo deploy/vweb --dry-run
deployment.apps/vweb Pod Template:
  Labels:      app=vweb
                pod-template-hash=656cbc5cc4
                version=v1.1

PS>kubectl rollout undo deploy/vweb --to-revision=2
deployment.apps/vweb rolled back
PS>
PS>curl $(cat url.txt)
v2
```

Undoing the rollout rolls back to the previous revision and the dry-run flag shows the result without actually doing it. This would roll back to v1.1.

Rolling back to revision 2 should reinstate v1, but we're actually getting v2...

Figure 9.6 Labels are a key management feature, but they're set by humans so they're fallible

You see that the moving parts of the release process are fairly simple - Deployments create and re-use ReplicaSets, scaling them up and down as required, and changes to ReplicaSets are recorded as rollouts. Kubernetes gives you control of the key factors in the rollout strategy, but before we move on to that we're going to look at releases which also involve a configuration change, because that adds another complicating factor.

In chapter 4 I talked about different approaches to updating the content of ConfigMaps and Secrets, and the choice you make impacts your ability to roll back cleanly. The first approach is to say that config is mutable, so a release might include a ConfigMap change, which is an update

to an existing ConfigMap object. But if your release is *only* a config change, then you have no record of that as a rollout and no option to roll back.

TRY IT NOW Remove the existing Deployment so we have a clean history, then deploy a new version which uses a ConfigMap and see what happens when you update the same ConfigMap.

```
# remove the existing app:  
kubectl delete deploy vweb  
  
# deploy a new version which stores content in config:  
kubectl apply -f vweb/update/vweb-v3-with-configMap.yaml --record  
  
# check the response:  
curl $(cat url.txt)  
  
# update the ConfigMap and wait for the change to propagate:  
kubectl apply -f vweb/update/vweb-configMap-v31.yaml --record  
sleep 120  
  
# check the app again:  
curl $(cat url.txt)  
  
# and the rollout history:  
kubectl rollout history deploy/vweb
```

As you see in figure 9.7, the update to the ConfigMap changes the behavior of the app but it's not a change to the Deployment, so there is no revision to roll back to if the config change causes an issue.

The new Deployment uses a ConfigMap for the response content to the HTTP request.

```

PS>kubectl delete deploy vweb
deployment.apps "vweb" deleted
PS>
PS>kubectl apply -f vweb/update/vweb-v3-with-configMap.yaml --record
configmap/vweb-config configured
deployment.apps/vweb created
PS>
PS>curl $(cat url.txt)
v3-from-config
PS>
PS>kubectl apply -f vweb/update/vweb-configMap-v31.yaml --record
configmap/vweb-config configured
PS>
PS>sleep 120
PS>
PS>curl $(cat url.txt)
v3.1
PS>
PS>kubectl rollout history deploy/vweb
deployment.apps/vweb
REVISION  CHANGE-CAUSE
1        kubectl.exe apply --filename=vweb/update/vweb-v3-with-configMap.yaml -
-record=true

```

The app reloads configuration when the ConfigMap file changes, but Kubernetes caches the content so it can take two minutes to take effect.

The app's behavior is different, but there was no change to the Deployment so there's no option to rollback.

Figure 9.7 Config updates might change app behavior but without recording a rollout

This is the hot reload approach, which works nicely if your apps support it, precisely because a config-only change doesn't require a rollout. The existing Pods and containers keep running, so there's no risk of service interruption. The cost is the loss of the rollback option, and you'll have to decide whether that's more important than a hot reload.

Your alternative is to consider all ConfigMaps and Secrets as immutable, so you include some versioning scheme in the object name and you never update a config object once it's created. Instead you create a new config object with a new name and release it along with an update to your Deployment, which references the new config object.

TRY IT NOW Deploy a new version of the app with immutable config, so you can compare the release process.

```

# remove the old Deployment:
kubectl delete deploy vweb
# create a new Deployment using immutable config:
kubectl apply -f vweb/update/vweb-v4-with-configMap.yaml --record

```

```
# check the output:  
curl $(cat url.txt)  
  
# release a new ConfigMap and updated Deployment:  
kubectl apply -f ./vweb/update/vweb-v41-with-configMap.yaml --record  
  
# check the output again:  
curl $(cat url.txt)  
  
# the update is a full rollout:  
kubectl rollout history deploy/vweb  
  
# so you can rollback:  
kubectl rollout undo deploy/vweb  
curl $(cat url.txt)
```

Figure 9.8 shows my output, where the config update is accompanied by a Deployment update, which preserves the rollout history and enables the roll back.

This Deployment uses a ConfigMap with a versioned name for the response content.

Updating config is done with a new ConfigMap and a change to the Pod spec to use the new config object name.

```
PS>kubectl delete deploy vweb
deployment.apps "vweb" deleted
PS>
PS>kubectl apply -f vweb/update/vweb-v4-with-configMap.yaml --record
configmap/vweb-config-v4 configured
deployment.apps/vweb created
PS>
PS>curl $(cat url.txt)
v4-from-config
PS>
PS>kubectl apply -f .\vweb\update\vweb-v41-with-configMap.yaml --record
configmap/vweb-config-v41 unchanged
deployment.apps/vweb configured
PS>
PS>curl $(cat url.txt)
v4.1-from-config
PS>
PS>kubectl rollout history deploy/vweb
deployment.apps/vweb
REVISION  CHANGE-CAUSE
1          kubectl.exe apply --filename=vweb/update/vweb-v4-with-configMap.yaml -
-record=true
2          kubectl.exe apply --filename=.\\vweb\\update\\vweb-v41-with-configMap.yaml
1 --record=true
```

```
PS>kubectl rollout undo deploy/vweb
deployment.apps/vweb rolled back
PS>
PS>curl $(cat url.txt)
v4-from-config
```

This approach preserves the rollout history for config changes.

So you can rollback to a previous configuration.

Figure 9.8 Immutable config preserves rollout history - but it means a rollout for every config change

Kubernetes doesn't really care which approach you take, and your choice will partly depend on who owns configuration in your organization. If the project team also owns deployment and configuration then you might prefer mutable config objects to simplify the release process and the number of objects to manage. If a separate team owns config then the immutable approach will be better as they can deploy new config objects ahead of the release. The scale of your apps will affect the decision too - at high scale you may prefer to reduce the number of app deployments and rely on mutable config.

There's a cultural impact to this decision, because it frames how application releases are perceived - as everyday events that are no big deal, or as something slightly scary which is to

be avoided as much as possible. In the container world releases should be trivial events which you're happy to do with minimal ceremony as soon as they're needed. Testing and tweaking your release strategy will go a long way to giving you that confidence.

9.3 Configuring rolling updates for Deployments

Deployments support two update strategies: RollingUpdate is the default and the one we've used so far, and the other is Recreate. You know how rolling updates work by scaling down the old ReplicaSet while scaling up the new ReplicaSet, which provides service continuity and the ability to stagger the update over a longer period. The recreate strategy gives you neither of those - it still uses ReplicaSets to implement changes, but it scales the previous set down to zero before scaling up the replacement. Listing 9.2 shows the recreate strategy in a Deployment spec -it's just one setting but it has a significant impact.

Listing 9.2 - vweb-recreate-v2.yaml, a Deployment using the recreate update strategy

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vweb
spec:
  replicas: 3
  strategy: # this is the update strategy
    type: Recreate # recreate is the alternative to the
                    # default strategy, RollingUpdate
  # selector & Pod spec follow
```

When you deploy this you'll see it's just a normal app with a Deployment, a ReplicaSet and some Pods. If you look at the details of the Deployment you'll see it uses the recreate update strategy, but that only has an effect when the Deployment is updated.

TRY IT NOW Deploy the app from listing 9.2 and explore the objects - this is just like a normal Deployment.

```
# delete the existing app:
kubectl delete deploy vweb

# deploy with recreate strategy:
kubectl apply -f vweb-strategies/vweb-recreate-v2.yaml

# check the ReplicaSets:
kubectl get rs -l app=vweb

# test the app:
curl $(cat url.txt)

# and look at the details of the Deployment:
kubectl describe deploy vweb
```

Figure 9.9 shows this is a new deployment of the same old web app, using v2 of the container image. There are three Pods, they're all running and the app works as expected - so far so good.

This Deployment uses the Recreate update strategy - but until you update it, the behavior is the same as a standard Deployment.

```
PS>kubectl delete deploy vweb
deployment.apps "vweb" deleted
PS>
PS>kubectl apply -f vweb-strategies/vweb-recreate-v2.yaml
deployment.apps/vweb created
PS>
PS>kubectl get rs -l app=vweb
NAME          DESIRED   CURRENT   READY   AGE
vweb-7659d9d47f   3         3         3      6s
PS>
PS>curl $(cat url.txt)
v2
PS>kubectl describe deploy vweb
Name:           vweb
Namespace:      default
CreationTimestamp: Fri, 22 May 2020 20:32:15 +0100
Labels:          kiamol=ch09
Annotations:    deployment.kubernetes.io/revision: 1
Replicas:       3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:   Recreate
```

Same old app, running v2 of the container image.

This abbreviated output shows the current revision and the update strategy of the Deployment.

Figure 9.9 The recreate update strategy doesn't affect behavior until you release an update

This is a very dangerous configuration though, and one you should only use if different versions of your app can't coexist - something like a database schema update, where you need to be sure that only one version of your app connects to the database. Even in that case there are better options, but if you have a scenario which definitely needs this approach, then you'd better be sure you test all your updates before you go live. If you deploy an update where the new Pods fail, you won't know that until your old Pods have all been terminated and your app will be completely unavailable.

TRY IT NOW v3 of the web app is ready to deploy. It's broken, as you'll see when the app goes offline because there are no Pods running.

```
# deploy the updated Pod spec:
kubectl apply -f vweb-strategies/vweb-recreate-v3.yaml

# check the status, with a time limit for updates:
kubectl rollout status deploy/vweb --timeout=2s
```

```
# check the ReplicaSets:  
kubectl get rs -l app=vweb  
  
# and the Pods:  
kubectl get pods -l app=vweb  
  
# test the app - this will fail:  
curl $(cat url.txt)
```

You'll see in this exercise that Kubernetes happily takes your app offline, because that's what you've requested. The recreate strategy creates a new ReplicaSet with the updated Pod template, then scales the previous ReplicaSet to zero and scales up the new ReplicaSet to three. The new image is broken, so the new Pods fail and there's nothing to respond to requests, as you see in figure 9.10.

This update will fail because the new container image is broken.

Checking the status will wait for the rollout to complete, but it never will in this case, so the timeout just waits for two seconds.

```
PS>kubectl apply -f vweb-strategies/vweb-recreate-v3.yaml  
deployment.apps/vweb configured  
PS>  
PS>kubectl rollout status deploy/vweb --timeout=2s  
Waiting for deployment "vweb" rollout to finish: 0 of 3 updated replicas are available...  
error: timed out waiting for the condition  
PS>  
PS>kubectl get rs -l app=vweb  
NAME          DESIRED   CURRENT   READY   AGE  
vweb-7659d9d47f  0         0         0      117s  
vweb-d9955b85  3         3         0      12s
```

PS>
PS>kubectl get pods -l app=vweb
NAME READY STATUS RESTARTS AGE
vweb-d9955b85-p9vlt 0/1 Error 2 21s
vweb-d9955b85-t54vw 0/1 Error 2 21s
vweb-d9955b85-thz5k 0/1 Error 2 21s

PS>
PS>curl \$(cat url.txt)
curl: (52) Empty reply from server

The old ReplicaSet has scaled down to zero, and the new ReplicaSet has scaled up to three, but none of the Pods are ready.

All the Pods are errored, so there is nowhere for the Service to send traffic and the request fails.

Figure 9.10 The recreate strategy will happily take down your app if the new Pod spec is broken

Now that you've seen it, you should probably try to forget about the recreate strategy. There are some scenarios where it will seem attractive, but when it does it will be worth considering

alternative options - even if it means looking again at your architecture. The wholesale takedown of your application is going to cause downtime, and probably more downtime than you plan for.

Rolling updates are the default because they guard against downtime, but even then the default behavior is quite aggressive. For a production release there are a few settings you'll want to tune, which set the speed of the release and how it gets monitored. As part of the rolling update spec you can add options which control how quickly the new ReplicaSet is scaled up and how quickly the old ReplicaSet is scaled down, using two values:

- `maxUnavailable` is the accelerator for scaling down the old ReplicaSet. It defines how many Pods can be unavailable during the update, relative to the desired Pod count. You can think of it as the batch size for terminating Pods in the old ReplicaSet - in a Deployment of 10 Pods, setting this to 30% means three Pods will be terminated immediately.
- `maxSurge` is the accelerator for scaling up the new ReplicaSet. It defines how many extra Pods can exist, over the desired replica count. This is like the batch size for creating Pods in the new ReplicaSet. In a Deployment of 10, setting this to 40% will create four new Pods.

Nice and simple - except that both settings are used during a rollout, so you have a see-saw effect. The new ReplicaSet is scaled up until the Pod count is the desired replica count plus the `maxSurge` value, and then the Deployment waits for old Pods to be removed. The old ReplicaSet is scaled down to the desired count minus the `maxUnavailable` count, then the Deployment waits for new Pods to reach the ready state. You can't set both values to zero because that means nothing will change. Figure 9.11 shows how you can combine the settings to prefer a create-then-remove, or a remove-then-create, or a remove-and-create approach to new releases.

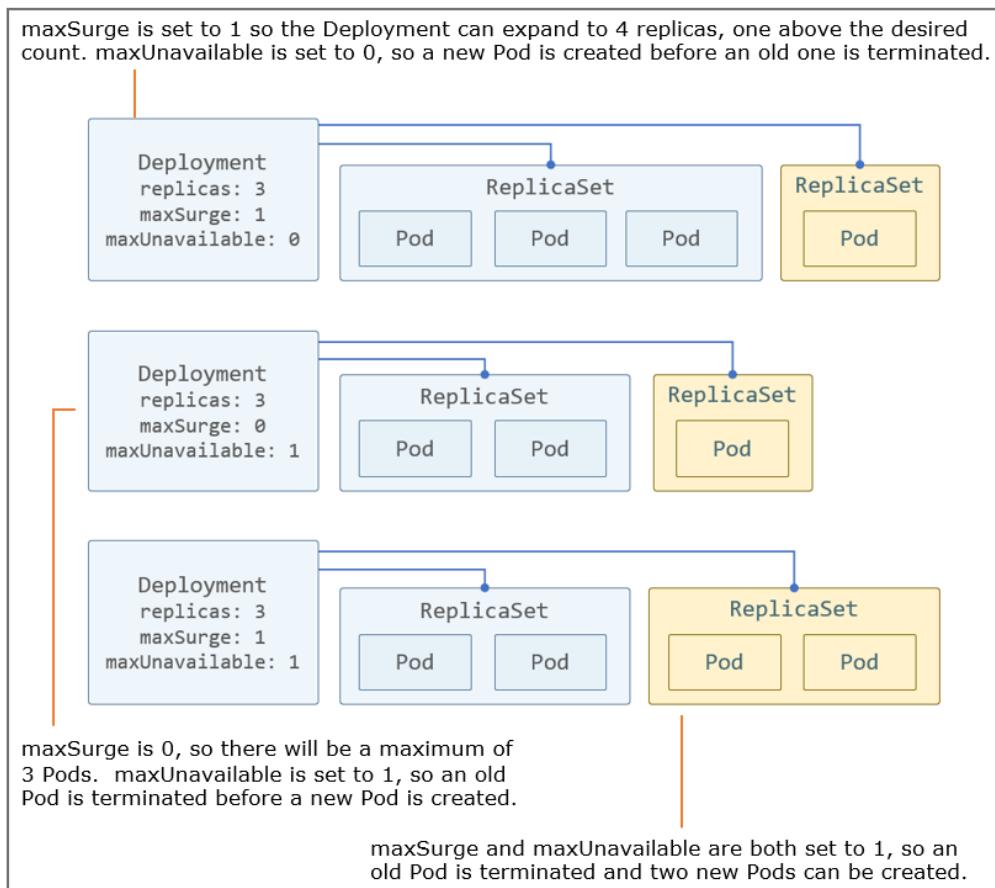


Figure 9.11 Deployment updates in progress, using different rollout options

You can tweak these settings for a faster rollout if you have spare compute power in your cluster, and you can create additional Pods over your scale setting - but that's riskier if there's a problem with the new release. A slower rollout is more conservative: it uses less compute and gives you more opportunity to discover any issues, but it reduces the overall capacity of your app during the release. Let's see how these look, first by fixing our broken app with a conservative rollout.

TRY IT NOW Revert back to the working v2 image, using `maxSurge=1` and `maxUnavailable=0` in the rolling update strategy.

```
# update the Deployment to use rolling updates and the v2 image:
kubectl apply -f vweb-strategies/vweb-rollingUpdate-v2.yaml
```

```
# check the Pods for the app:  
kubectl get po -l app=vweb  
  
# and the rollout status:  
kubectl rollout status deploy/vweb  
  
# check the ReplicaSets:  
kubectl get rs -l app=vweb  
  
# and test the app:  
curl $(cat url.txt)
```

In this exercise the new Deployment spec changed the Pod image back to version 2, and it also changed the update strategy to rolling update. You can see in figure 9.12 that the strategy change is made first, and then the Pod update is made in line with the new strategy, creating one new Pod at a time.

This spec changes the update strategy and the container image - the strategy change is made first, and the Pods are updated using the new strategy.

maxSurge is set to 1, so the Deployment can have 4 Pods - one old Pod is terminating, so there are four active Pods, two from each ReplicaSet.

```
PS>kubectl apply -f vweb-strategies/vweb-rollingUpdate-v2.yaml  
deployment.apps/vweb configured
```

```
PS>kubectl get po -l app=vweb  
NAME READY STATUS RESTARTS AGE  
vweb-7659d9d47f-5hc72 1/1 Running 0 2s  
vweb-7659d9d47f-fkzn4 0/1 ContainerCreating 0 1s  
vweb-d9955b85-p9vlt 0/1 CrashLoopBackOff 30 47h  
vweb-d9955b85-t54vw 0/1 CrashLoopBackOff 30 47h  
vweb-d9955b85-thz5k 0/1 Terminating 30 47h
```

```
PS>  
PS>kubectl rollout status deploy/vweb  
deployment "vweb" successfully rolled out  
PS>
```

```
PS>kubectl get rs -l app=vweb
```

NAME	DESIRED	CURRENT	READY	AGE
vweb-7659d9d47f	3	3	3	47h
vweb-d9955b85	0	0	0	47h

```
PS>
```

```
PS>curl $(cat url.txt)  
v2
```

The new Pod spec is the same as the original Pod spec, so the Deployment reuses the first ReplicaSet - scaling it up to 3.

The Pods are working correctly, so the app is back online.

Figure 9.12 Deployment updates will use an existing ReplicaSet if the Pod template matches the new spec

You'll need to work fast to see the rollout in progress in the last exercise, because this is a simple app which starts quickly, and as soon as one new Pod is running then the rollout continues with another new Pod. You can control the pace of the rollout with two other fields in the Deployment spec:

- `minReadySeconds` adds a delay where the Deployment waits to make sure new Pods are stable. It specifies the number of seconds the Pod should be up with no containers crashing before it's considered to be successful. The default is zero, which is why new Pods are created quickly during rollouts.
- `progressDeadlineSeconds` specifies the amount of time a Deployment update can run before it's considered as failing to progress. The default is 600 seconds, so if an update is not completed within 10 minutes it's flagged as not progressing.

Monitoring how long the release takes sounds useful, but as of Kubernetes 1.18 exceeding the deadline doesn't actually affect the rollout - it just sets a flag on the Deployment. Kubernetes doesn't have an automatic rollback feature for failed rollouts, but when that feature does come it will be triggered by this flag. Waiting and checking a Pod for failed containers is a fairly blunt tool, but it's better than having no checks at all and you should consider having `minReadySeconds` specified in all your Deployments.

These are useful safety measures to add to your Deployment, but they don't really help with our web app because the new Pods always fail. We can make this Deployment safe and keep the app online using a rolling update - the next v3 update sets both `maxUnavailable` and `maxSurge` to one. That has the same effect as the default values (each 25%), but it's clearer to use exact values in the spec, and Pod counts are easier to work with than percentages in small deployments.

TRY IT NOW Deploy the v3 update again - it will still fail, but with a rolling update strategy it doesn't take the app offline.

```
# update to the failing container image:
kubectl apply -f vweb-strategies/vweb-rollingUpdate-v3.yaml

# check the Pods:
kubectl get po -l app=vweb

# and the rollout:
kubectl rollout status deploy/vweb

# see the scale in the ReplicaSets:
kubectl get rs -l app=vweb

# and test the app:
curl $(cat url.txt)
```

When you run this exercise you'll see the update never completes and the Deployment is stuck with two ReplicaSets having a desired Pod count of two - as you see in figure 9.13. The old ReplicaSet won't scale down any further because the Deployment has `maxUnavailable` set

to one; it has already been scaled down by one and no new Pods will become ready to continue the rollout. The new ReplicaSet won't scale up any more because maxSurge is set to one, and the total Pod count for the Deployment has been reached.

This rolling update has maxSurge and maxUnavailable both set to one, so the initial batch terminates one old v2 Pod and creates two new v3 Pods.

The update is set with a progress deadline of 120 seconds.
The status command waits until the rollout hits that deadline.

```
PS>kubectl apply -f vweb-strategies/vweb-rollingUpdate-v3.yaml
deployment.apps/vweb configured
PS>
PS>kubectl get po -l app=vweb
NAME           READY   STATUS    RESTARTS   AGE
vweb-7659d9d47f-5t78g  1/1     Running   0          5m58s
vweb-7659d9d47f-k986n  0/1     Terminating   0          5m58s
vweb-7659d9d47f-tslh7  1/1     Running   0          5m58s
vweb-d9955b85-769g2    0/1     Error     0          2s
vweb-d9955b85-s86bj    0/1     Error     0          2s
PS>
PS>kubectl rollout status deploy/vweb
Waiting for deployment "vweb" rollout to finish: 2 out of 3 new replicas have been updated...
error: deployment "vweb" exceeded its progress deadline
PS>
PS>kubectl get rs -l app=vweb
NAME      DESIRED   CURRENT   READY   AGE
vweb-7659d9d47f   2         2         2       2d
vweb-d9955b85    2         2         0       2d
PS>
PS>curl $(cat url.txt)
v2
```

The update won't progress because the new Pods have failed.
Kubernetes doesn't have automatic rollbacks, so this is the final state.

The app is still running at reduced capacity, using the v2 Pods.

Figure 9.13 Failed updates don't automatically roll back or pause; they just keep trying

If you check back on the new Pods in a few minutes you'll see they're in the state CrashLoopBackoff. Kubernetes keeps restarting failed Pods by creating replacement containers, but it adds a pause between each restart so it doesn't choke the CPU on the node. That pause is the backoff time and it increases exponentially - 10 seconds for the first restart, then 20 seconds and then 40 seconds, up to a maximum of 5 minutes. These v3 Pods will never restart successfully, but Kubernetes will keep trying.

Deployments will be the controllers you use the most, and it's worth spending time working through the update strategy and timing settings to be sure you understand the impact for your

apps. DaemonSets and StatefulSets also have rolling update functionality, and because they have different ways of managing their Pods, they have different approaches to rollouts too.

9.4 Rolling updates in DaemonSets and StatefulSets

DaemonSets and StatefulSets have two update strategies available. The default is RollingUpdate, which we'll explore in this section. The alternative is OnDelete, which is for situations when you need close control over when each Pod is updated. You deploy the update and the controller watches Pods, but it doesn't terminate any existing Pods - it waits until they are deleted by another process, and then it replaces them with Pods from the new spec.

This isn't quite as pointless as it sounds, when you think about the use-cases for these controllers. You may have a StatefulSet where each Pod needs to have flushed data to disk before it's removed, and you can have an automated process to do that. You may have a DaemonSet where each Pod needs to be disconnected from a hardware component, so it's free for the next Pod to use. They're rare cases, but the OnDelete strategy lets you take ownership of when Pods are deleted, and still have Kubernetes automatically create replacements.

We'll focus on rolling updates in this section, and for that we'll deploy a version of the to-do list app which runs the database in a StatefulSet, the web app in a Deployment and a reverse proxy for the web app in a DaemonSet.

TRY IT NOW The to-do app runs across six Pods, so start by clearing down the existing apps to make room.

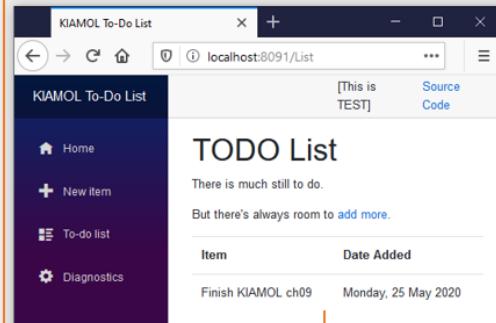
Then deploy the app and test that it works correctly.

```
# remove all this chapter's apps:  
kubectl delete all -l kiamol=ch09  
  
# deploy the to-do app, database and proxy:  
kubectl apply -f todo-list/db/ -f todo-list/web/ -f todo-list/proxy/  
  
# get the app URL:  
kubectl get svc todo-proxy -o  
  jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8091'  
  
# browse to the app, add an item and check it's in the list
```

This is just setting us up for the updates - you should now have a working app where you can add items and see the list. My output is in figure 9.14.

The current state of the to-do app - a StatefulSet for the database, Deployment for the website, DaemonSet for the proxy, and associated ConfigMaps and Secrets.

```
PS>kubectl delete all -l kiamol=ch09
service "vweb" deleted
deployment.apps "vweb" deleted
PS>
PS>kubectl apply -f todo-list/db/ -f todo-list/web/ -f todo-list/proxy/
configmap/todo-db-config created
configmap/todo-db-env created
configmap/todo-db-scripts created
secret/todo-db-secret configured
service/todo-db created
statefulset.apps/todo-db created
configmap/todo-web-config created
secret/todo-web-secret configured
service/todo-web created
deployment.apps/todo-web created
configmap/todo-proxy-configmap created
service/todo-proxy created
daemonset.apps/todo-proxy created
PS>
PS>kubectl get svc todo-proxy -o jsonpath='http://.status.loadBalancer.ingress[0].*:8091'
http://localhost:8091
```



The app works in the same way, and now we're ready to make some updates.

Figure 9.14 Running the to-do app with a gratuitous variety of controllers

The first update is for the DaemonSet where we'll be rolling out a new version of the Nginx proxy image. DaemonSets run a single Pod on all (or some) of the nodes in the cluster, and with a rolling update there is no surge option. During the update, nodes will never run two Pods, so this is always a delete-then-remove strategy. You can add the maxUnavailable setting to control how many nodes are updated in parallel, but if you take down multiple Pods you'll be running at reduced capacity until the replacements are ready.

We'll update the proxy using a maxUnavailable setting of one, and a minReadySeconds setting of 90. On a single-node lab cluster the delay won't have any affect - there's only one Pod on one node to replace. On a larger cluster it would mean replacing one Pod at a time, and waiting 90 seconds for the Pod to prove it's stable before moving on to the next.

TRY IT NOW Start the rolling update of the DaemonSet. On a single node cluster there will be a short outage while the replacement Pod starts.

```
# deploy the DaemonSet update:
kubectl apply -f todo-list/proxy/update/nginx-rollingUpdate.yaml
# watch the Pod update:
```

```
kubectl get po -l app=todo-proxy --watch
# ctrl-c when the update completes
```

The `watch` flag in Kubectl is very useful for monitoring changes - it just keeps looking at an object and prints an update line whenever the state changes. In this exercise you'll see that the old Pod is terminated before the new one is created, which means there's downtime for the app while the new Pod starts up. Figure 9.15 shows I had one second of downtime in my release.

Updating a DaemonSet is the same process of applying an updated Pod spec.

```
PS>kubectl apply -f todo-list/proxy/update/nginx-rollingUpdate.yaml
daemonset.apps/todo-proxy configured
```

PS>

```
PS>kubectl get po -l app=todo-proxy --watch
```

NAME	READY	STATUS	RESTARTS	AGE
todo-proxy-wvqwp	0/1	Terminating	0	13m
todo-proxy-wvqwp	0/1	Terminating	0	13m
todo-proxy-wvqwp	0/1	Terminating	0	13m
todo-proxy-c6rsrv	0/1	Pending	0	0s
todo-proxy-c6rsrv	0/1	Pending	0	0s
todo-proxy-c6rsrv	0/1	ContainerCreating	0	0s
todo-proxy-c6rsrv	1/1	Running	0	1s

Viewing an object with the `watch` flag shows all the updates - the original Pod terminates, then the replacement is scheduled and created.

Figure 9.15 DaemonSets update by removing the existing Pod before creating a replacement

In a multi-node cluster there wouldn't be any downtime because the Service only sends traffic to Pods which are ready, and only one Pod at a time gets updated so there will always be other Pods available. There will be reduced capacity though, and if you tune a faster rollout with a higher `maxUnavailable` setting, that means a greater reduction in capacity as more Pods are updated in parallel. That's the only setting you have for DaemonSets, so it's a simple choice between manually controlling the update by deleting Pods, or having Kubernetes roll out the update by a specified number of Pods in parallel.

StatefulSets are more interesting, although they only have one option to configure the rollout. Pods are managed in order by the StatefulSet and that also applies to updates - the rollout proceeds backwards from the last Pod in the set down to the first. That's especially useful for clustered applications where Pod 0 is the primary, because it validates the update on the secondaries first.

There is no `maxSurge` or `maxUnavailable` setting for StatefulSets - the update is always by one Pod at a time. Your configuration option is to define how many Pods should be updated in total, using the `partition` setting. This defines the cut-off point where the rollout stops, and

it's useful for performing a staged rollout of a stateful app. If you have five replicas in your set and your spec includes partition=3, then only Pod 4 and Pod 3 will be updated - Pods 0, 1 and 2 are left running the previous spec.

TRY IT NOW Deploy a partitioned update to the database image in the StatefulSet, which stops after Pod 1, so Pod 0 doesn't get updated.

```
# deploy the update:
kubectl apply -f todo-list/db/update/todo-db-rollingUpdate-partition.yaml

# check the rollout status:
kubectl rollout status statefulset/todo-db

# list the Pods, showing the image name and start time:
kubectl get pods -l app=todo-db -o=custom-
    columns=NAME:.metadata.name,IMAGE:.spec.containers[0].image,START_TIME:.status
        .startTime

# switch the web app to read-only mode, so it uses the secondary db:
kubectl apply -f todo-list/web/update/todo-web-readonly.yaml

# test app - the data is there, but now it's read-only
```

This exercise is a partitioned update which rolls out a new version of the Postgres container image, but only to the secondary Pods - which is a single Pod in this case, as you see in figure 9.16. When you use the app in read-only mode you'll see that it connects to the updated secondary, which still contains the replicated data from the previous Pod.

Rollouts are available for StatefulSets and DaemonSets as well as Deployments.

```
PS>kubectl apply -f todo-list/db/update/todo-db-rollingUpdate-partition.yaml
statefulset.apps/todo-db configured
PS>
PS>kubectl rollout status statefulset/todo-db
Waiting for partitioned roll out to finish: 0 out of 1 new pods have been updated...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
partitioned roll out complete: 1 new pods have been updated...
PS>
PS>kubectl get pods -l app=todo-db -o=custom-columns=NAME:.metadata.name,IMAGE:.spec.containers[0].image,START TIME:.status.startTime
NAME           IMAGE          START_TIME
todo-db-0      postgres:11.6-alpine  2020-05-25T06:56:25Z
todo-db-1      postgres:11.8-alpine  2020-05-25T07:33:20Z
PS>
PS>kubectl apply -f todo-list/web/update/todo-web-readonly.yaml
deployment.apps/todo-web configured
```

This is a partitioned update - the rollout is complete, and Pod 1 has been updated but Pod 0 has not.

Switching the web app to read-only mode tests that the update to the database Pod has worked correctly.

Figure 9.16 Partitioned updates to StatefulSets let you update secondaries and leave the primary unchanged

This rollout is complete, even though the Pods in the set are running from different specs. For a data-heavy application in a StatefulSet you may have a suite of verification jobs that you need to run on each updated Pod before you're happy to continue the rollout, and a partitioned update lets you do that. You can manually control the pace of the release by running successive updates with decreasing partition values, until you remove the partition altogether in the final update to finish the set.

TRY IT NOW Deploy the update to the database primary - this spec is the same as the previous exercise, but with the partition setting removed.

```
# apply the update:
kubectl apply -f todo-list/db/update/todo-db-rollingUpdate.yaml

# check its progress:
kubectl rollout status statefulset/todo-db

# Pods should now all have the same spec:
kubectl get pods -l app=todo-db -o=custom-
columns=NAME:.metadata.name,IMAGE:.spec.containers[0].image,START_TIME:.status
.startTime

# reset the web app back to read-write mode:
```

```
kubectl apply -f todo-list/web/todo-web.yaml
# test the app works, connected to the updated primary Pod
```

You can see my output in figure 9.17, where the full update has completed and the primary is using the same updated version of Postgres as the secondary. If you've done updates to replicated databases before, you'll know that this is about as simple as it gets - unless you're using a managed database service, of course.

The final stage in the StatefulSet update - with no partition setting the rollout will update Pod 0.

```
PS>kubectl apply -f todo-list/db/update/todo-db-rollingUpdate.yaml
statefulset.apps/todo-db configured
PS>
PS>kubectl rollout status statefulset/todo-db
waiting for statefulset rolling update to complete 1 pods at revision todo-db-d7
7d4cdc5...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
statefulset rolling update complete 2 pods at revision todo-db-d77d4cdc5...
PS>
PS>kubectl get pods -l app=todo-db -o=custom-columns=NAME:.metadata.name,IMAGE:.
spec.containers[0].image,START_TIME:.status.startTime
NAME           IMAGE          START_TIME
todo-db-0      postgres:11.8-alpine  2020-05-25T07:41:41Z
todo-db-1      postgres:11.8-alpine  2020-05-25T07:33:20Z
PS>
PS>kubectl apply -f todo-list/web/todo-web.yaml
deployment.apps/todo-web configured
```

Pod 0 is replaced, but Pod 1 is not updated because it was updated in the last rollout and already matches the Pod spec.

The web app can come out of read-only mode, and it will connect to Pod 0 for the database.

Figure 9.17 Completing the StatefulSet rollout, with an update that is not partitioned

Rolling updates are the default for Deployments, DaemonSets and StatefulSets, and they all work in broadly the same way - gradually replacing Pods running the previous application spec with Pods running the new spec. The actual details are different because the controllers work in different ways and have different goals, but they impose the same requirement on your app - it needs to work correctly when there are multiple versions live. That's not always possible and there are alternative ways to deploy app updates in Kubernetes.

9.5 Understanding release strategies

Take the example of a web application - a rolling update is great because it lets each Pod close gracefully when all its client requests are dealt with, and the rollout can be as fast or as

conservative as you like. The practical side of the rollout is simple, but there's the user experience (UX) side to consider too.

Application updates might well change the UX - with a different design, new features or an updated workflow. Any changes like that will be pretty strange for the user if they see the new version during a rollout, then refresh and find themselves with the old version, because the requests have been served by Pods running different versions of the app.

The strategies to deal with that go beyond the rolling update spec in your controllers. You can set cookies in your web app to link a client to a particular UX, and then use a more advanced traffic routing system to ensure users keep seeing the new version. We'll cover that in chapter 15, and you'll see it introduces several more moving parts. For cases where that's too complex or it doesn't solve the problem of dual-running multiple versions, you can manage the release yourself with a blue-green deployment.

Blue-green deployments are a simple concept: you have both the old and new versions of your app deployed at the same time, but only one version is active. You can flip a switch to choose which version is the active one. In Kubernetes that can be done by updating the label selector in a Service, to send traffic to the Pods in a different Deployment - as you see in figure 9.18.

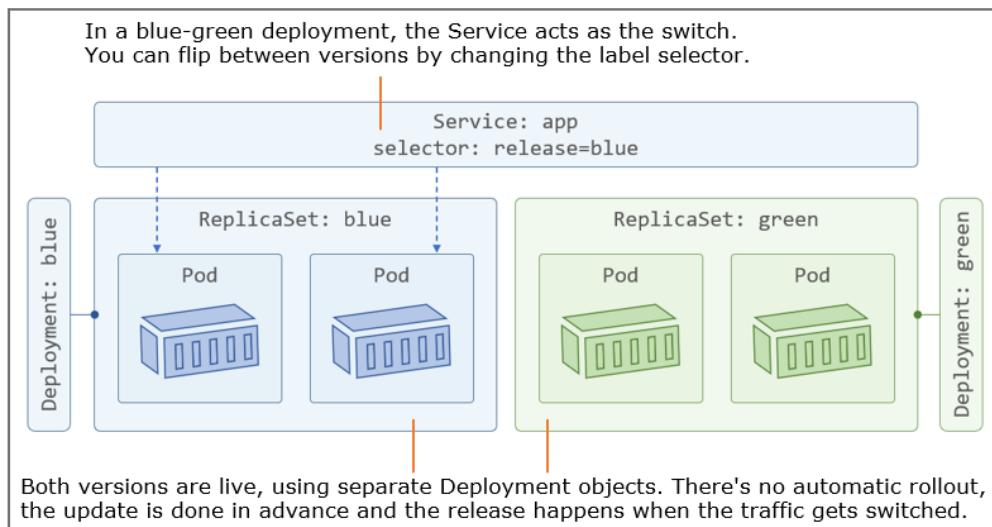


Figure 9.18 You run multiple versions of the app in a blue-green deployment, but only one is live

You need to have the capacity in your cluster to run two complete copies of your app - but if it's a web or API component then the new version should be using minimal memory and CPU because it's not receiving any traffic. You switch between versions by updating the label selector for the Service, so the update is practically instant because all the Pods are running and ready

to receive traffic. You can flip back and forward easily, so you can rollback a problem release without waiting for ReplicaSets to scale up and down.

Blue-green deployments are less sophisticated than rolling updates, but they're simpler because of that. They can be a better fit for organizations moving to Kubernetes who have a history of big-bang deployments, but they're a compute-intensive approach which require multiple steps and don't preserve the rollout history of your app. You should look to rolling updates as your preferred deployment strategy, but blue-green deployments are a good stepping-stone to use while you gain confidence.

That's all on rolling updates for now, but we will return to the concepts when we cover topics in production readiness, network ingress and monitoring. We just need to tidy up the cluster now before going on to the lab.

TRY IT NOW Remove all the objects created for this chapter.

```
kubectl delete all -l kiamol=ch09
kubectl delete cm -l kiamol=ch09
kubectl delete pvc -l kiamol=ch09
```

9.6 Lab

We had the theory of blue-green deployments in the last section, and now in the lab you're going to make it happen. Working through this lab will help make it clear how selectors relate Pods to other objects, and give you experience working with the alternative to rolling updates.

- the starting point is v1 of the web app which you can deploy from the `lab/v1` folder
- you need to create a blue-green deployment for v2 of the app - the spec will be similar to the v1 spec but using the `:v2` container image
- when you deploy your update you should be able to flip between the v1 and v2 release just by changing the Service and without any updates to Pods.

This is good practice in copying YAML files and trying to work out which fields you need to change. You can find my solution on GitHub:

<https://github.com/sixeyed/kiamol/blob/master/ch09/lab/README.md>

10

Packaging and managing apps with Helm

Vast as Kubernetes is, it doesn't solve every problem by itself and a large ecosystem exists to fill the gaps. One of the gaps is packaging and distributing apps, and the solution is Helm. You can use Helm to group together a set of Kubernetes YAML files into one artifact and share that on a public or private repository. Anyone with access to the repository can install the app with a single Helm command. That might deploy a whole set of related Kubernetes resources including ConfigMaps, Deployments and Services, and you can customize the configuration as part of the installation.

People use Helm in different ways. Some teams only use Helm to install and manage third-party apps from public repositories. Other teams use Helm for their own applications, packaging and publishing them to private repositories. In this chapter you'll learn how to do both and you'll leave with your own idea of how Helm might fit in your organization. You don't need to learn Helm to be effective with Kubernetes, but it's very widely used so it's something you should be familiar with. The project is governed by the Cloud Native Computing Foundation (CNCF) - the same foundation which stewards Kubernetes - and that's a reliable indicator of maturity and longevity.

10.1 What Helm adds to Kubernetes

Kubernetes apps are modelled in a sprawl of YAML files at design time, and managed using sets of labels at runtime. There's no native concept of an "application" in Kubernetes which clearly groups together a set of related resources, and that's one of the problems Helm solves. It's a command-line tool which interacts with repository servers to find and download application packages, and with your Kubernetes cluster to install and manage applications.

Helm is another layer of abstraction, this time at the application level. When you install an application with Helm it just creates a set of resources in your Kubernetes cluster - and they're

standard Kubernetes resources. The Helm packaging format extends Kubernetes YAML files, so a Helm package is really just a set of Kubernetes manifests stored along with some metadata. We'll start by using Helm to deploy one of the sample apps from previous chapters, but first we need to install Helm.

TRY IT NOW Helm is a cross-platform tool which runs on Windows, macOS and Linux. You can find the latest install instructions here - <https://helm.sh/docs/intro/install>. This exercise assumes you already have a package manager like Homebrew or Chocolatey installed. If not, you'll need to refer to the Helm site for the full install instructions.

```
# on Windows, using Chocolatey:  
choco install -y kubernetes-helm  
  
# on Mac, using Homebrew:  
brew install helm  
  
# on Linux, using the Helm install script:  
curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 | bash  
  
# and check Helm is working:  
helm version
```

The install steps in that exercise may not work on your system, in which case you'll need to stop here and head to the Helm install docs - we can't go much further until you have Helm installed, and you see successful output from the version command, like mine in figure 10.1.

Helm is an open-source project published on GitHub. It's available through package managers - I'm using Chocolatey on Windows.

```
PS> choco install -y --limit-output kubernetes-helm
Installing the following packages:
kubernetes-helm
By installing you accept licenses for the packages.
Progress: Downloading kubernetes-helm 3.2.3... 100%

The install of kubernetes-helm was successful.
Software installed to 'C:\ProgramData\chocolatey\lib\kubernetes-helm\tools'

Chocolatey installed 1/1 packages.
See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).

PS>
PS>
PS> helm version
version.BuildInfo{Version:"v3.2.3", GitCommit:"8f832046e258e2cb800894579b1b3b50c
2d83492", GitTreeState:"clean", GoVersion:"go1.13.12"}
```

Helm is a command-line utility. Make sure you're running version 3 - it's a significant upgrade from previous versions.

Figure 10.1 There are lots of options to install Helm; using a package manager is easiest

Helm is a client-side tool - previous versions of Helm required a server component to be deployed in your Kubernetes cluster, but that was changed in the major update to Helm 3. It uses the same connection information that Kubectl uses to connect to your Kubernetes cluster, so you don't need any extra configuration to install an app. You do need to configure a package repository. Helm repositories are similar to container image registries like Docker Hub, but the server publishes an index of all available packages; Helm caches a local copy of your repository indexes, which you can use to search packages.

TRY IT NOW Add the Helm repository for the book, sync it and search for an app.

```
# add a repository, using a local name to refer to a remote server:  
helm repo add kiamol https://kiamol.net  
  
# update the local repository cache:  
helm repo update  
  
# and search for an app in the repo cache:  
helm search repo vweb --versions
```

The Kiamol repository is a public server, and you can see in this exercise that there are two versions of the package called vweb - my output is in figure 10.2.

There are lots of public Helm repositories, and you can run your own repository server. This is the repository for the book.

Updating the repository downloads the package index to a local cache.

```
PS>helm repo add kiamol https://kiamol.net  
"kiamol" has been added to your repositories  
PS>  
PS>helm repo update  
Hang tight while we grab the latest from your chart repositories...  
...Successfully got an update from the "kiamol" chart repository  
Update Complete. * Happy Helming!*
```

NAME	CHART VERSION	APP VERSION	DESCRIPTION
kiamol/vweb	2.0.0	2.0.0	Simple versioned web app
kiamol/vweb	1.0.0	1.0.0	Simple versioned web app

There are two packages named vweb - they have the same name and description, but different version numbers.

Figure 10.2 Syncing a local copy of the Kiamol Helm repository and searching for packages

You're getting a feel for Helm, but it's time for some theory so we can use the correct concepts and their correct names before we go any further. An application package in Helm is called a *chart*; charts can be developed and deployed locally, or published to a *repository*.

When you install a chart that's called a *release*; every release has a name and you can install multiple instances of the same chart in your cluster as separate, named releases.

Charts contain Kubernetes YAML manifests, and the manifests can contain parameterized values so users can install the same chart with different configuration settings - the number of replicas to run, or the application logging level could be parameter values. Each chart also contains a default set of values, and they can be inspected using the command line. Figure 10.3 shows the file structure of a Helm chart.

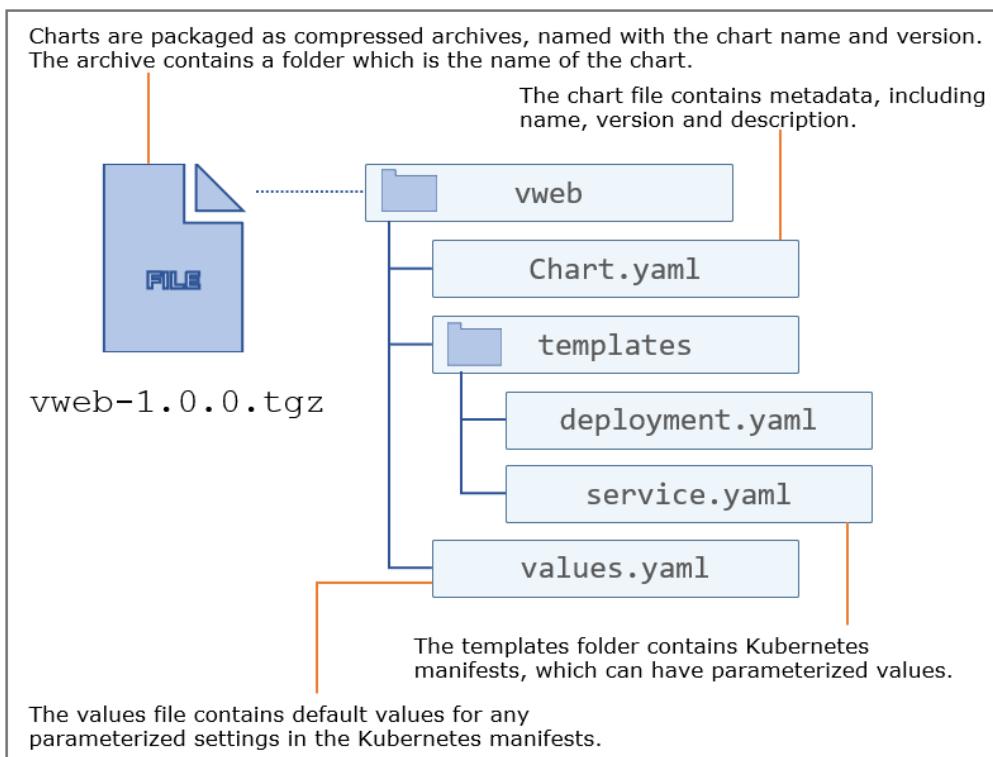


Figure 10.3 Helm charts contain all the Kubernetes YAML for the app, plus some metadata

The `vweb` charts package the simple web app we used to demonstrate updates and rollbacks in chapter 9. Each chart contains a spec for a Service and a Deployment, with some parameterized values and default settings. You can use the Helm command line to inspect all the available values before installing the chart, and then override the defaults with custom values when you install a release.

TRY IT NOW Check the values available in version 1 of the `vweb` chart, and then install a release using custom values.

```
# inspect the default values stored in the chart:
helm show values kiamol/vweb --version 1.0.0

# install the chart, overriding the default values:
helm install --set servicePort=8010 --set replicaCount=1 ch10-vweb kiamol/vweb --version
1.0.0

# check the releases you have installed:
helm ls
```

In that exercise you can see that the chart has default values for the Service port and the number of replicas in the Deployment - my output is in figure 10.4. You use the `set` argument with `helm install` to specify your own values, and when the install completes you have an app running in Kubernetes without using Kubectl and without any direct use of YAML manifests.

Shows the default values for a specific version of a chart. The chart name is prefixed with the local repo name, kiamol.

These are the settings in the `values.yaml` file in the chart.

```
PS>helm show values kiamol/vweb --version 1.0.0
# port for the Service to listen on
servicePort: 8090
# number of replicas for the web Pod
replicaCount: 2
PS>
PS>helm install --set servicePort=8010 --set replicaCount=1 ch10-vweb kiamol/vwe
b --version 1.0.0
NAME: ch10-vweb
LAST DEPLOYED: Thu Jun 11 07:21:04 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
PS>
PS>helm ls
NAME          NAMESPACE      REVISION      UPDATED
           STATUS        CHART        APP VERSION
ch10-vweb    default       1            2020-06-11 07:21:04.9162549 +0100
0/BST        deployed     vweb-1.0.0   1.0.0
```

You can override the default values using the `set` argument in the `install` command, and specify the exact version to install.

This lists all installed releases, showing the status, chart name and application version.

Figure 10.4 Installing an app with Helm - this creates Kubernetes resources, without using Kubectl

Helm has a set of features for working with repositories and charts, and for installing, updating and rolling back releases, but it's not for ongoing management of applications. The Helm command line isn't a replacement for Kubectl - you use both together. Now that the

release is installed, you can work with the Kubernetes resources in the usual way and you can also return to Helm if you need to modify settings.

TRY IT NOW Check the resources Helm has deployed using Kubectl, then return to Helm to scale up the Deployment and check the app is working correctly.

```
# show the details of the Deployment:  
kubectl get deploy -l app.kubernetes.io/instance=ch10-vweb --show-labels  
  
# update the release to increase the replica count:  
helm upgrade --set servicePort=8010 --set replicaCount=3 ch10-vweb kiamol/vweb --version  
1.0.0  
  
# check the ReplicaSet:  
kubectl get rs -l app.kubernetes.io/instance=ch10-vweb  
  
# get the URL for the app:  
kubectl get svc ch10-vweb -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8010'  
  
# browse to the app URL
```

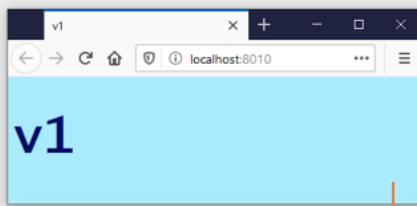
There are a couple of things to call out from that exercise. The first is that the labels are a lot more verbose than the standard "app" and "version" labels you've seen so far. That's because this is a public chart on a public repo, so I'm using the recommended label names from the Kubernetes best practice guide - that's my choice, not a requirement of Helm. The second is that the Helm upgrade command specifies the Service port again, although it's only the replica count I want to modify. That's because Helm uses the default values unless you specify them, so if the port isn't included in the upgrade command it would be changed to the default value. You can see my output in figure 10.5.

The manifests in the template add best-practice labels, which are prefixed `app.kubernetes.io`.

You can use Helm to update the settings of a release using the same version of the chart.

```
PS>kubectl get deploy -l app.kubernetes.io/instance=ch10-vweb --show-labels
NAME      READY   UP-TO-DATE   AVAILABLE   AGE   LABELS
ch10-vweb 1/1     1           1           18s   app.kubernetes.io/instance=ch10-vweb,app.kubernetes.io/managed-by=Helm,app.kubernetes.io/name=vweb,kiamol=ch10
PS>
PS>helm upgrade --set servicePort=8010 --set replicaCount=3 ch10-vweb kiamol/vwe
b --version 1.0.0
Release "ch10-vweb" has been upgraded. Happy Helming!
NAME: ch10-vweb
LAST DEPLOYED: Thu Jun 11 07:34:27 2020
NAMESPACE: default
STATUS: deployed
REVISION: 2
TEST SUITE: None
PS>
PS>kubectl get rs -l app.kubernetes.io/instance=ch10-vweb
NAME      DESIRED   CURRENT   READY   AGE
ch10-vweb-565bf9748d  3         3         3       34s
PS>
```

```
PS>kubectl get svc ch10-vweb -o jsonpath='http://.status.loadBalancer.ingress[0].*:8010'
http://localhost:8010
PS>
```



The chart value `replicaCount` sets the replicas for the Deployment - now there are three Pods.

The Service is still using port 8010, set in the original release.

Figure 10.5 You don't use Helm to manage apps, but you can use it to update the configuration

This is the consumer side of the Helm workflow - you can search repositories for applications, discover the configuration values available for an app, and then install and upgrade the application all from the Helm command line. It's package management for apps built to run in Kubernetes, and in the next section you'll learn how to package and publish your own apps, which is the producer side of the workflow.

10.2 Packaging your own apps with Helm

Helm charts are folders or zipped archives which contain Kubernetes manifests. You create your own charts by taking your application manifests, identifying any values you want to be parameterized, and replacing the actual value with a templated variable. Listing 10.1 shows the beginning of a templated Deployment spec, which uses values set by Helm for the resource name and the label value.

Listing 10.1 - web-ping-deployment.yaml, a templated Kubernetes manifest

```
apiVersion: apps/v1
kind: Deployment # this much is standard Kubernetes YAML

metadata:
  name: {{ .Release.Name }} # set to the name of the release
  labels:
    kiamol: {{ .Values.kiamolChapter }} # set to the "kiamolChapter" value
```

The double brace syntax is for templated values - everything from the opening {{ to the closing }}} gets replaced at install time, and Helm sends the processed YAML to Kubernetes. Multiple sources can replace templated values - the snippet in listing 10.1 uses the Release object to get the name of the release and the Values object to get a parameter value called kiamolChapter. The Release object is populated with information from the install or upgrade command and the Values object is populated from defaults in the chart and any settings the user has overridden. Templates can also access static details about the chart and runtime details about the capabilities of the Kubernetes cluster.

Helm is very particular about the file structure in a chart - you can use the `helm create` command to generate the boilerplate structure for a new chart. The top level is a folder whose name has to match the chart name you want to use, and that folder must have at least these three items:

- a `Chart.yaml` file which specifies the chart metadata, including the name and version
- a `values.yaml` file which sets the default values for parameters
- a `templates` folder which contains the templated Kubernetes manifests

Listing 10.1 is from a file called `web-ping-deployment.yaml` in the `web-ping/templates` folder in this chapter's source. The `web-ping` folder contains all the files needed for a valid chart, and Helm can validate the chart contents and install a release from the chart folder.

TRY IT NOW When you're developing charts you don't need to package them in zip archives; you can work with the chart folder.

```
# switch to this chapter's source:
cd ch10

# validate the chart contents:
helm lint web-ping

# and install a release from the chart folder:
```

```
helm install wp1 web-ping/
# check installed releases:
helm ls
```

The lint command is only for working with local charts, but the install command is the same for local charts and for charts stored in a repository. Local charts can be folders or zipped archives, and you'll see in this exercise that installing a release from a local chart is the same experience as installing from a repository. My output in figure 10.6 shows I now have two releases installed, one from the vweb chart and one from the web-ping chart.

The lint command checks that a chart is valid. Any issues which will cause installation to fail will be printed as errors.

You can install charts from a local folder, which is how you develop a chart before packaging it for a repository.

```
PS>cd ch10
PS>
PS>helm lint web-ping
==> Linting web-ping
[INFO] Chart.yaml: icon is recommended

1 chart(s) linted, 0 chart(s) failed
PS>
PS>helm install wp1 web-ping/
NAME: wp1
LAST DEPLOYED: Thu Jun 11 09:21:53 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
PS>
PS>helm ls
NAME          NAMESPACE   REVISION        UPDATED           STATUS      CHART
              STATUS        APP VERSION
ch10-vweb     default     2            2020-06-11 07:34:27.7181051 +0100
0/BST         deployed    vweb-1.0.0    1.0.0
wp1           default     1            2020-06-11 09:21:53.8346124 +0100
0/BST         deployed    web-ping-0.1.0 1.0.0
```

The chart metadata in the folder contains a version number, so all charts are versioned in the same way whether the source is a local folder or a remote repository.

Figure 10.6 Installing and upgrading from a local folder lets you iterate quickly on chart development

The web-ping application is a basic utility which checks if a website is up - it makes HTTP requests to a domain name on a regular schedule. Right now you have a Pod running which is sending requests to my blog every 30 seconds - my blog runs on Kubernetes, so I'm sure it will be able to handle that. The app uses environment variables to configure the URL to use

and the schedule interval, and those are templated in the manifest for Helm. Listing 10.2 shows the Pod spec with the templated variables.

Listing 10.2 - web-ping-deployment.yaml, templated container environment

```
spec:
  containers:
    - name: app
      image: kiamol/ch10-web-ping
      env:
        - name: TARGET
          value: {{ .Values.targetUrl }}
        - name: INTERVAL
          value: {{ .Values.pingIntervalMilliseconds | quote }}
```

Helm has a rich set of templating functions you can use to manipulate the values that get set in the YAML. The quote function in listing 10.2 wraps the provided value in quote marks, if it doesn't already have them. You can include looping and branching logic in your templates, calculate strings and numbers, and even query the Kubernetes API to find details from other objects. We won't get into that much detail, but it's important to remember that Helm lets you generate sophisticated templates that can do pretty much anything.

You need to think carefully about the parts of your spec that need to be templated. One of the big benefits of Helm over standard manifest deployments is that you can run multiple instances of the same app from a single chart. You can't do that with Kubectl because the manifests contain resource names which need to be unique, and if you deploy the same set of YAML multiple times Kubernetes will just update the same resources. If you template all the unique parts of the spec - like resource names and label selectors - then you can run many copies of the same app with Helm.

TRY IT NOW Deploy a second release of the web-ping app, using the same chart folder but specifying a different URL to ping.

```
# check the available settings for the chart:
helm show values web-ping/

# install a new release named wp2, using a different target:
helm install --set targetUrl=kiamol.net wp2 web-ping/

# wait a minute or so for the pings to fire, then check the logs:
kubectl logs -l app=web-ping --tail 1
```

You'll see in that exercise that I need to do some optimization on my blog - it returns in around 500ms whereas the Kiamol website returns in 100ms. But more importantly you can see two instances of the app running - two Deployments managing two sets of Pods with different container specs. My output is in figure 10.7.

These values in the chart are used in environment variables in the Deployment template, so they set the configuration of the app.

```

PS>helm show values web-ping/
# targetUrl - URL of the website to ping
targetUrl: blog.sixeyed.com
# httpMethod - HTTP method to use for pings
httpMethod: HEAD
# pingIntervalMilliseconds - interval between pings in ms
pingIntervalMilliseconds: 30000
# chapter where this exercise is used
kiamolChapter: ch10
PS>
PS>helm install --set targetUrl=kiamol.net wp2 web-ping/
NAME: wp2
LAST DEPLOYED: Thu Jun 11 10:46:58 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
PS>
PS>kubectl logs -l app=web-ping --tail 1
Got response status: 200 at 1591869361199; duration: 507ms
Got response status: 200 at 1591869360941; duration: 113ms

```

Installing the same chart with a different name deploys a completely separate release.

Both instances of the app are busy pinging different websites and logging the response times - the Pods in different Deployments have the same app label.

Figure 10.7 You can't install multiple instances of an app with plain manifests, but you can with Helm

It should be clear now that the Helm workflow for installing and managing apps is different from the Kubectl workflow - but you also need to understand that the two are *incompatible*. You can't deploy the app by running `kubectl apply` in the templates folder for the chart, because the templated variables are not valid YAML and the command will fail. If you adopt Helm you need to choose between using Helm for every environment - which will slow down the developer workflow, or using plain Kubernetes manifests for development and Helm for other environments - which means multiple copies of your YAML.

Remember that Helm is about distribution and discovery as much as its about installation. The additional friction that Helm brings is the price of being able to simplify complex applications down to a few variables and share them on a repository. And a repository is really just an index file with a list of chart versions that can be stored on any web server (the Kiamol

repository uses GitHub pages and you can see the whole contents at <https://kiamol.net/index.yaml>).

You can use any server technology to host your repository, but for the rest of this section we'll use a dedicated repository server called ChartMuseum, which is a popular open-source option. You can run ChartMuseum as a private Helm repository in your own organization, and it's easy to set up because you can install it with a Helm chart.

TRY IT NOW The ChartMuseum chart is on the official Helm repo, conventionally called "stable". Add that repo and you can install a release to run your own repo locally.

```
# add the official Helm repo:
helm repo add stable https://kubernetes-charts.storage.googleapis.com

# install ChartMuseum - the repo flag fetches details from
# the repo so you don't need to update your local cache:
helm install --set service.type=LoadBalancer --set service.externalPort=8008 --set
  env.open.DISABLE_API=false repo stable/chartmuseum --version 2.13.0 --wait

# get the URL for ChartMuseum:
kubectl get svc repo-chartmuseum -o
  jsonpath='http://{{.status.loadBalancer.ingress[0].*.}}:8008'

# and add it as a repository called local:
helm repo add local $(kubectl get svc repo-chartmuseum -o
  jsonpath='http://{{.status.loadBalancer.ingress[0].*.}}:8008')
```

Now you have three repositories registered with Helm - the Kiamol repository, the stable Kubernetes repository (which is a curated set of charts, similar to the official images in Docker Hub) and your own local repository. You can see my output in figure 10.8, which is abridged to reduce the output from the Helm install command.

Add the official Helm chart repo, which hosts the ChartMuseum chart.

Install ChartMuseum with settings to use a LoadBalancer Service on port 8008, and enable the application API.

```
PS>helm repo add stable https://kubernetes-charts.storage.googleapis.com
"stable" has been added to your repositories
```

```
PS>helm install --set service.type=LoadBalancer --set service.externalPort=8008
--set env.open.DISABLE_API=false repo stable/chartmuseum --version 2.13.0
```

```
NAME: repo
LAST DEPLOYED: Thu Jun 11 12:04:31 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
** Please be patient while the chart is being deployed **
```

```
PS>kubectl get svc repo-chartmuseum -o jsonpath='http://{.status.loadBalancer.ingress[0].*:8008}'
http://localhost:8008
```

```
PS>helm repo add local $(kubectl get svc repo-chartmuseum -o jsonpath='http://{.status.loadBalancer.ingress[0].*:8008}')
"local" has been added to your repositories
```

When the install completes, check the URL for the Service.

Use the URL to add your local ChartMuseum repository to Helm.

Figure 10.8 Running your own Helm repository is as simple as installing a chart from a Helm repository

Charts need to be packaged before they can be published to a repository, and publishing is usually a three stage process: package the chart into a zip archive; upload the archive to a server; update the repository index to add the new chart. ChartMuseum takes care of the last step for you, so you just need to package and upload the chart, and the repository index is automatically updated.

TRY IT NOW Use Helm to create the zip archive for the chart, and use curl to upload it to your ChartMuseum repository. Check the repository and you'll see your chart has been indexed.

```
# package the local chart:
helm package web-ping

# *on Windows 10* remove the PowerShell alias to use the real curl:
Remove-Item Alias:curl -ErrorAction Ignore

# upload the chart zip archive to ChartMuseum:
curl --data-binary "@web-ping-0.1.0.tgz" $(kubectl get svc repo-chartmuseum -o
jsonpath='http://{.status.loadBalancer.ingress[0].*:8008/api/charts')
```

```
# check that ChartMuseum has updated its index:
curl $(kubectl get svc repo-chartmuseum -o
      jsonpath='http://{.status.loadBalancer.ingress[0].*}:8008/index.yaml')
```

Helm uses compressed archives to make charts easy to distribute, and the files are tiny - they contain the Kubernetes manifests and the metadata and values, but they don't contain any large binaries. Pod specs in the chart will specify container images to use, but the images themselves are not part of the chart - they're pulled from Docker Hub or your own image registry when you install a release. You can see in figure 10.9 that ChartMusem generates the repository index when you upload a chart and adds the new chart details.

Packaging a chart produces the artifact - a zipped archive which can be uploaded to a server.

ChartMuseum has an API for uploading chart archives, this sends the zip file using curl.

```
PS>helm package web-ping
Successfully packaged chart and saved it to: D:\scm\github\sixeyed\kiamol\ch10\web-ping-0.1.0.tgz
PS>
PS>curl --data-binary "@web-ping-0.1.0.tgz" http://localhost:8008/api/charts
{"saved":true}
PS>
PS>curl http://localhost:8008/index.yaml
apiVersion: v1
entries:
  web-ping:
    - apiVersion: v2
      appVersion: 1.0.0
      created: "2020-06-11T11:14:09.269651Z"
      description: A simple web pinger
      digest: 5d2c58004c5166c49dad4f6bdbcd28759ce9642a6608b9c87186a37cec7ddb18
      name: web-ping
      type: application
      urls:
        - charts/web-ping-0.1.0.tgz
      version: 0.1.0
```

The new chart is shown in the repository index. The metadata is taken from the Chart.yaml file, and a URL for the archive download is added, together with a digest for the file so Helm can validate the download.

Figure 10.9 You can run ChartMuseum as a private repo to easily share charts between teams

You can use ChartMuseum or another repository server in your organization to share internal applications, or to push charts as part of your Continuous Integration process before making release candidates available on your public repository. The local repo you have is only running in your lab environment, but it's published using a LoadBalancer Service so anyone with network access can install the web-ping app from your repo.

TRY IT NOW Install yet another version of the web-ping app, this time using the chart from your local repo, and providing a values file instead of specifying each setting in the install command.

```
# update your repo cache:  
helm repo update  
  
# verify that Helm can find your chart:  
helm search repo web-ping  
  
# check the local values file:  
cat web-ping-values.yaml  
  
# install from the repo using the values file:  
helm install -f web-ping-values.yaml wp3 local/web-ping  
  
# list all the Pods running the web-ping apps:  
kubectl get pod -l app=web-ping -o custom-  
columns='NAME:.metadata.name,ENV:.spec.containers[0].env[*].value'
```

In that exercise you saw another way to install a Helm release with custom settings - using a local values file. That's a good practice, because you can store the settings for different environments in different files, and you mitigate the risk that an update reverts back to a default value when a setting isn't provided. My output is in figure 10.10.

Updating your local repo cache lets you search for charts.

The new web-ping chart is listed in the local repo.

```

PS>helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "local" chart repository
...Successfully got an update from the "kiamol" chart repository
...Successfully got an update from the "stable" chart repository
Update Complete. * Happy Helming!*
PS>
PS>helm search repo web-ping
NAME          CHART VERSION   APP VERSION   DESCRIPTION
local/web-ping  0.1.0          1.0.0          A simple web pinger
PS>
PS>cat web-ping-values.yaml
pingIntervalMilliseconds: 15000
PS>
PS>helm install -f web-ping-values.yaml wp3 local/web-ping
NAME: wp3
LAST DEPLOYED: Thu Jun 11 13:49:51 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
PS>
PS>kubectl get pod -l app=web-ping -o custom-columns='NAME:.metadata.name,ENV:.spec.containers[0].env[*].value'
NAME          ENV
wp1-658b698fb-qc285  blog.sixeyed.com,HEAD,30000
wp2-77b6545bf5-krsqn  kiamol.net,HEAD,30000
wp3-58998db649-7p7z5  blog.sixeyed.com,HEAD,15000

```

Installation with a values file is a better practice than using set arguments, because you can use the same file for upgrades.

Each release of the app uses the same chart with different values - and those values set environment variables in the Pod container.

Figure 10.10 Installing charts from your local repository is the same as installing from any remote repo

You also saw in the last exercise that you can install a chart from a repository without specifying a version - that's not such a good practice, because that installs the latest version, which is a moving target. It's better to always explicitly state the chart version. Helm requires you to use semantic versioning so chart consumers know whether the package they're about to upgrade is a beta release, or if it has breaking changes.

You can do far more with charts than I'm going to cover here - they can include tests, which are Kubernetes Job specs that run after installation to verify the deployment; they can have hooks which let you run Jobs at specific points in the installation workflow; and they can be signed and shipped with a signature for provenance. In the next section I'm going to cover

just one more feature you use in authoring templates, but it's an important one - building charts which are dependent on other charts.

10.3 Modelling dependencies in charts

Helm lets you design your app so it works in different environments, and that raises an interesting problem for dependencies. A dependency might be required in some environments but not in others - maybe you have a web app which really needs a caching reverse proxy to improve performance. In some environments you'll want to deploy the proxy along with the app, and in others you'll already have a shared proxy so you just want to deploy the web app itself. Helm supports that with conditional dependencies.

Listing 10.3 shows a chart manifest for the Pi web application we've been using since chapter 5. It has two dependencies - one from the Kiamol repository and one from the local filesystem - and those are both separate charts.

Listing 10.3 - Chart.yaml, a chart which includes optional dependencies

```
apiVersion: v2      # the version of the Helm spec
name: pi           # chart name
version: 0.1.0      # chart version
dependencies:      # other charts this chart is dependent on
  - name: vweb
    version: 2.0.0
    repository: https://kiamol.net    # a dependency from a repo
    condition: vweb.enabled         # only installed if required
  - name: proxy
    version: 0.1.0
    repository: file:///proxy      # a dependency from a local folder
    condition: proxy.enabled       # only installed if required
```

You need to keep your charts flexible when you model dependencies - the *parent chart* (the Pi app in this case) may require the *subchart* (the proxy and vweb charts), but subcharts themselves need to be stand-alone. You should template the Kubernetes manifests in a subchart to make it generically useful - if it's something which is only ever useful in one application, then it should be part of that chart and not a subchart.

My proxy is generically useful, it's just a caching reverse proxy which can use any HTTP server as the content source. The chart uses a templated value for the name of the server to proxy, so although it's primarily intended for the Pi app it can be used to proxy any Kubernetes Service. We can verify that by installing a release which proxies an existing app in the cluster.

TRY IT NOW Install the proxy chart on its own, using it as a reverse proxy for the vweb app we installed earlier in the chapter.

```
# install a release from the local chart folder:
helm install --set upstreamToProxy=ch10-vweb:8010 vweb-proxy proxy/

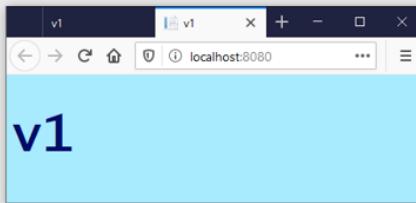
# get the URL for the new proxy service:
kubectl get svc vweb-proxy-proxy -o
  jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8080'
```

```
# browse to the URL
```

The proxy chart in that exercise is completely independent of the Pi app; it's being used to proxy the web app I deployed with Helm from the Kiamol repo. You can see in figure 10.11 that it just works as a caching proxy for any HTTP server.

The proxy chart is used as a subchart for the Pi application, but it can also be installed as a standalone chart. The value here filters into a ConfigMap and sets the proxy to load content from the vweb app.

```
PS>helm install --set upstreamToProxy=ch10-vweb:8010 vweb-proxy proxy/
NAME: vweb-proxy
LAST DEPLOYED: Thu Jun 11 14:45:45 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
PS>
PS>kubectl get svc vweb-proxy-proxy -o jsonpath='http://.status.loadBalancer.in
gress[0].*:8080'
http://localhost:8080
PS>
PS>
```



The proxy uses a LoadBalancer service listening on port 8080. When you browse to that URL you'll see the vweb app, with the content served by the proxy.

Figure 10.11 The proxy subchart is built to be useful as a chart in its own right - it can proxy any app

To use the proxy as a dependency, you need to add it in the dependency list in a parent chart, so it becomes a subchart. Then you can specify values for the subchart settings in the parent chart, by prefixing the setting name with the dependency name - so the setting `upstreamToProxy` in the proxy chart is referenced as `proxy.upstreamToProxy` in the Pi chart. Listing 10.4 shows the default values file for the Pi app, which includes settings for the app itself and for the proxy dependency.

Listing 10.4 - values.yaml, the default settings for the Pi chart

```
replicaCount: 2          # number of app Pods to run
serviceType: LoadBalancer # type of the Pi Service

proxy:                  # settings for the reverse proxy
  enabled: false         # whether to deploy the proxy
  upstreamToProxy: "{{ .Release.Name }}-web"    # server to proxy
  servicePort: 8030        # port of the proxy Service
  replicaCount: 2         # number of proxy Pods to run
```

These values deploy the app itself without the proxy, using a LoadBalancer Service for the Pi Pods. The setting `proxy.enabled` is specified as the condition for the proxy dependency in the Pi chart, so the entire subchart gets skipped unless the install settings override the default. The full values file also sets the `vweb.enabled` value to false - that dependency is only there to demonstrate that subcharts can be sourced from repositories, so the default is not to deploy that chart either.

There's one extra detail to call out here. The name of the Service for the Pi app is templated in the chart, using the release name. That's important to enable multiple installs of the same chart, but it adds complexity to the default values for the proxy subchart. The name of the server to proxy needs to match the Pi Service name, so the values file uses the same templated value as the Service name and that links the proxy to the Service in the same release.

Charts need to have their dependencies available before you can install or package them, and you use the Helm command line to do that. Building dependencies will populate them into the chart's charts folder - either by downloading the archive from a repository, or packaging a local folder into an archive.

TRY IT NOW Build the dependencies for the Pi chart, which downloads the remote chart and packages the local chart, and adds them to the chart folder.

```
# build dependencies:
helm dependency build pi

# check the dependencies have been downloaded:
ls ./pi/charts
```

Figure 10.12 shows why versioning is so important for Helm charts. Chart packages are versioned using the version number in the chart metadata. Parent charts are packaged with their dependencies, at the specified version. If I update the proxy chart without updating the version number, my Pi chart will be out of sync because version 0.1.0 of the proxy chart in the Pi package is different from the latest version 0.1.0. You should consider Helm charts to be immutable and always publish changes by publishing a new package version.

Dependencies need to be added to the local chart folder - Helm downloads remote charts and builds local charts for all the subcharts listed in the parent chart's manifest.

```
PS>helm dependency build pi
```

```
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "local" chart repository
...Successfully got an update from the "kiamol" chart repository
...Successfully got an update from the "stable" chart repository
Update Complete. *Happy Helming!*
Saving 2 charts
Downloading vweb from repo https://kiamol.net
Deleting outdated charts
PS>
PS>ls pi/charts
```

Directory: D:\scm\github\sixeyed\kiamol\ch10\pi\charts			
Mode	LastWriteTime	Length	Name
-----	-----	-----	-----
-a---	11/06/2020	15:31	1485 proxy-0.1.0.tgz
-a---	11/06/2020	15:31	919 vweb-2.0.0.tgz

The `charts` subdirectory contains the subchart archives - these get expanded and saved with the parent chart when that chart is packaged.

Figure 10.12 Helm bundles dependencies into the parent chart and they are distributed as one package

This principle of conditional dependencies is how you could manage a much more complex application like the to-do app from chapter 8. The Postgres database deployment would be a subchart which users could skip altogether for environments where they want to use an external database. Or you could even have multiple conditional dependencies - allowing users to deploy a simple Postgres Deployment for dev environments, use a highly-available StatefulSet for test environments, and plug into a managed Postgres service in production.

The Pi app is simpler than that, and we can just choose whether to deploy it on its own or with a proxy. This chart uses a templated value for the type of the Pi Service, but that could be computed in the template instead - setting it to LoadBalancer if the proxy is not deployed, and ClusterIP if the proxy is deployed.

TRY IT NOW Deploy the Pi app with the proxy subchart enabled. Use Helm's dry run feature to check the default deployment, and then use custom settings for the actual install.

```
# print the YAML Helm would deploy with default values:
helm install pi1 ./pi --dry-run

# and install with custom settings to add the proxy:
helm install --set serviceType=ClusterIP --set proxy.enabled=true pi2 ./pi
```

```
# get the URL for the proxied app:  
kubectl get svc pi2-proxy -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8030'  
  
# and browse to it
```

You'll see in that exercise that the dry-run flag is very useful - it applies values to the templates and writes out all the YAML for the resources it would install, without deploying anything. Then in the actual installation, setting a couple of flags deploys a whole additional chart which is integrated with the main chart, so the application works as a single unit - you can see my Pi calculation in figure 10.13.

The default chart settings just deploy a Service and a Deployment for the Pi app, the proxy subchart is not enabled (I've abbreviated the full YAML output).

Enabling the proxy subchart and making the Pi Service internal means the app is only available via the proxy.

```
PS>helm install pi1 ./pi --dry-run  
NAME: pi1  
LAST DEPLOYED: Thu Jun 11 15:49:45 2020  
NAMESPACE: default  
STATUS: pending-install  
REVISION: 1  
TEST SUITE: None  
HOOKS:  
MANIFEST:  
---  
# Source: pi/templates/web-service.yaml  
apiVersion: v1  
kind: Service  
metadata:  
  name: pi1-web  
  
PS>helm install --set serviceType=ClusterIP --set proxy.enabled=true pi2 ./pi  
NAME: pi2  
LAST DEPLOYED: Thu Jun 11 15:50:57  
NAMESPACE: default  
STATUS: deployed  
REVISION: 1  
TEST SUITE: None  
PS>  
PS>kubectl get svc pi2-proxy -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8030'  
http://localhost:8030
```

A browser window titled "Pi.Web" shows the output of the command above. The address bar says "localhost:8030". The page content displays "from: pi2-web-85d7c49846-jvlp" and "3.141592".

And Pi is still the same.

Figure 10.13 Installing a chart with an optional subchart by overriding default settings

There's a whole lot of Helm which I haven't made room for in this chapter, because there's a level of complexity you only need to dive into if you bet big on Helm and plan to use it

extensively. If that's you then you'll find Helm has the power to cover you. How's this for an example: you can generate a hash from the contents of a ConfigMap template and use that as a label in a Deployment template, so every time the configuration changes the Deployment label changes too, and upgrading your config triggers a Pod rollout.

That's neat, but it's not for everybody so in the next section we'll return to a simple demo app and look at how Helm smoothes the upgrade and rollback process.

10.4 Upgrading and rolling back Helm releases

Upgrading an app with Helm doesn't do anything special, it just sends the updated specs to Kubernetes which rolls out changes in the usual way. If you want to configure the specifics of the rollout, you still do that in the YAML files in the chart, using the settings we explored in chapter 9. What Helm brings to upgrades is a consistent approach for all types of resources, and the ability to easily roll back to previous versions.

One other advantage you get with Helm is the ability to safely try out a new version by deploying an additional instance to your cluster. I started this chapter by deploying version 1.0.0 of the vweb app in my cluster, and it's still running happily. Version 2.0.0 is available now, but before I upgrade the running app I can use Helm to install a separate release and test the new functionality.

TRY IT NOW Check the original vweb release is still there and then install a v2 release alongside, specifying settings to keep the app private.

```
# list all releases:
helm ls -q

# check the values for the new chart version:
helm show values kiamol/vweb --version 2.0.0

# deploy a new release using an internal Service type:
helm install --set servicePort=8020 --set replicaCount=1 --set serviceType=ClusterIP ch10-
vweb-v2 kiamol/vweb --version 2.0.0

# use a port-forward so you can test the app:
kubectl port-forward svc/ch10-vweb-v2 8020:8020

# browse to localhost:8020 then exit the port-forward with Ctrl-C or Cmd-C
```

That exercise makes use of the values the chart supports to install the app without making it publicly available - using a ClusterIP Service type and a port-forward so the app is only accessible to the current user. The original app is unchanged and I have a chance to smoke test the new deployment in the target cluster - figure 10.14 shows the new version running.

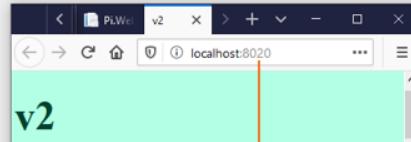
These are all the releases managed by Helm, with the output just showing the release name.

Version 2.0.0 of the vweb chart adds a `serviceType` value so the type of the Service can be set as well as the port.

```

PS>helm ls -q
ch10-vweb
pi2
repo
vweb-proxy
wp1
wp2
wp3
PS>
PS>helm show values kiamol/vweb --version 2.0.0
# port for the Service to listen on
servicePort: 8090
# type of the Service:
serviceType: LoadBalancer
# number of replicas for the web Pod
replicaCount: 2
PS>
PS>helm install --set servicePort=8020 --set replicaCount=1 --set serviceType=ClusterIP ch10-vweb-v2 kiamol/vweb --version 2.0.0
NAME: ch10-vweb-v2
LAST DEPLOYED: Thu Jun 11 19:49:59 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
PS>
PS>kubectl port-forward svc/ch10-vweb-v2 8020:8020

```



Installing the release with a ClusterIP type keeps the Service private.

You can access the website using a port-forward in Kubernetes.

Figure 10.14 Charts which deploy Services typically let you set the type, so you can keep them private

Now I'm happy that the 2.0.0 version is good, I can use the Helm upgrade command to upgrade my actual release. I want to make sure I deploy with the same values I set in the previous release, and Helm has features to show the current values, and to reuse custom values in the upgrade.

TRY IT NOW Remove the temporary v2 release and upgrade the v1 release to the v2 chart re-using the same values set on the current release.

```

# remove the test release:
helm uninstall ch10-vweb-v2

# check the values used in the current v1 release:

```

```
helm get values ch10-vweb
# upgrade to v2 using the same values - this will fail:
helm upgrade --reuse-values --atomic ch10-vweb kiamol/vweb --version 2.0.0
```

Oh dear. This is a particularly nasty issue that will take some tracking down to understand. The `reuse-values` flag tells Helm to reuse all the values set for the current release on the new release - but the version 2.0.0 chart includes another value, the type of the Service, which wasn't set in the current release because it didn't exist. The net result is that the Service type is blank, which defaults to ClusterIP in Kubernetes and the update fails because that clashes with the existing Service spec. You can see this hinted at in the output in figure 10.15.

Uninstalling a release removes all of the Kubernetes resources.

This shows the values set for the current v1 release.

```
PS>helm uninstall ch10-vweb-v2
release "ch10-vweb-v2" uninstalled
PS>
PS>helm get values ch10-vweb
USER-SUPPLIED VALUES:
replicaCount: 3
servicePort: 8010
PS>
PS>helm upgrade --reuse-values --atomic ch10-vweb kiamol/vweb --version 2.0.0
Error: UPGRADE FAILED: release ch10-vweb failed, and has been rolled back due to
atomic being set: cannot patch "ch10-vweb" with kind Service: Service "ch10-vwe
b" is invalid: spec.ports[0].nodePort: Forbidden: may not be used when `type` is
'ClusterIP'
```

Trying to re-use the same values for the v2 upgrade fails because v2 introduces a new setting. The defaults in the chart are ignored when you re-use values so the `serviceType` is left blank and the update fails.

Figure 10.15 An invalid upgrade fails and Helm can automatically roll back to the previous release

This sort of problem is where Helm's abstraction layer really helps. You can get the same issue with a standard Kubectl deployment, but if one resource update fails you need to check through all the other resources and manually roll them back. Helm does that automatically with the `atomic` flag - it waits for all the resource updates to complete, and if any of them fails it rolls back every other resource to the previous state. Check the history of the release and you can see that Helm has automatically rolled back to version 1.0.0.

TRY IT NOW Recall from chapter 9 that Kubernetes doesn't give you much information on the history of a rollout - compare that to the detail you get from Helm.

```
# show the history of the vweb release:
helm history ch10-vweb
```

That command gets an exercise all to itself, because there's a wealth of information there which you just don't get in the history for a standard Kubernetes rollout. Figure 10.16 shows all four revisions of the release - the first install, a successful upgrade, a failed upgrade and an automatic rollback.

Helm's history shows what action was taken and what the status was.				
REVISION	APP VERSION	UPDATED	STATUS	CHART
1	1.0.0	Thu Jun 11 20:17:54 2020	superseded	vweb-1.0.0
2	1.0.0	Thu Jun 11 20:18:03 2020	superseded	vweb-1.0.0
3	2.0.0	Thu Jun 11 20:18:43 2020	failed	vweb-2.0.0
		Upgrade "ch10-vweb" failed: cannot patch "ch10-vweb" with kind Serv ice: Service "ch10-vweb" is invalid: spec.ports[0].nodePort: Forbidden: may not be used when `type` is 'ClusterIP'		
4	1.0.0	Thu Jun 11 20:18:43 2020	deployed	vweb-1.0.0
		Rollback to 2		

It also shows the revision of the release, and the versions of the chart and the application. The app version is from the chart metadata and it may be different from the chart version.

Figure 10.16 The release history clearly links application and chart versions to revisions

To fix the failed update I can manually set all the values in the upgrade command, or use a values file with the same settings that are currently deployed. I don't have that values file, but I can save the output of the `get values` command to a file and use that in the upgrade, which gives me all my previous settings plus the defaults in the chart for any new settings.

TRY IT NOW Upgrade to v2 again, this time saving the current v1 values to a file and using that in the upgrade command.

```
# save the values of the current release to a YAML file:  
helm get values ch10-vweb -o yaml > vweb-values.yaml  
# upgrade to v2 using the values file and the atomic flag:  
helm upgrade -f vweb-values.yaml --atomic ch10-vweb kiamol/vweb --version 2.0.0  
  
# check the Service and ReplicaSet configuration:  
kubectl get svc,rs -l app.kubernetes.io/instance=ch10-vweb
```

This upgrade succeeds so the atomic rollback doesn't kick in. The upgrade is actually effected by the Deployment, which scales up the replacement ReplicaSet and scales down the current ReplicaSet in the usual way. You can see in figure 10.17 that the configuration values set in the previous release have been retained, the Service is listening on port 8010 and there are three Pods running.

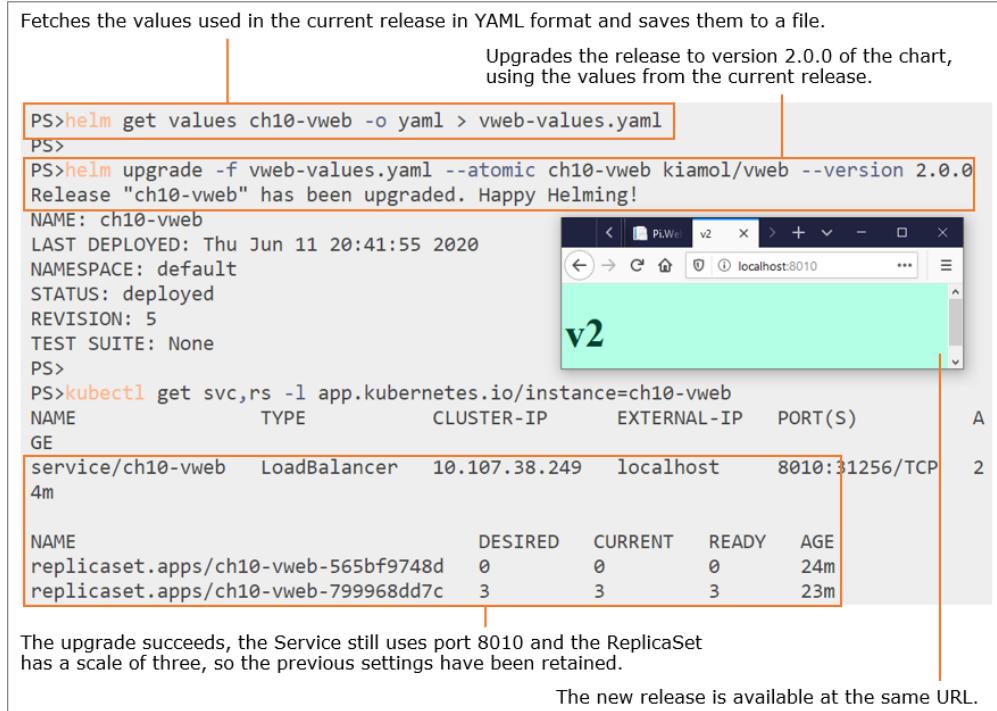


Figure 10.17 The upgrade succeeds by exporting the release settings to a file and using them again

All that's left is to try out a rollback, which is syntactically similar to a rollback in Kubectl, but Helm makes it much easier to track down the revision you want to use. You've already seen the meaningful release history in figure 10.16, and you can also use Helm to check the values set for a particular revision. If I want to roll back the web application to version 1.0.0 but preserve the values I set in revision 2, I can check those values first.

TRY IT NOW Rollback to the second revision, which was version 1.0.0 of the app upgraded to use three replicas.

```
# confirm the values used in revision 2:
helm get values ch10-vweb --revision 2

# roll back to that revision:
helm rollback ch10-vweb 2

# check the latest two revisions:
helm history ch10-vweb --max 2 -o yaml
```

You can see my output in figure 10.18, where the rollback is successful and the history shows that the latest revision is 6, which is actually a rollback to revision 2.

Before rolling back you can check the values set for a revision.

```

PS>helm get values ch10-vweb --revision 2
USER-SUPPLIED VALUES:
replicaCount: 3
servicePort: 8010

PS>
PS>helm rollback ch10-vweb 2
Rollback was a success! Happy Helming!
PS>
PS>helm history ch10-vweb --max 2 -o yaml
- app_version: 2.0.0
  chart: vweb-2.0.0
  description: Upgrade complete
  revision: 5
  status: superseded
  updated: "2020-06-11T20:41:55.6812879+01:00"
- app_version: 1.0.0
  chart: vweb-1.0.0
  description: Rollback to 2
  revision: 6
  status: deployed
  updated: "2020-06-11T20:58:23.7932335+01:00"

```

After the rollback the status shows that revision 6 is the same as 2.

Figure 10.18 Helm makes it easy to check exactly what you're rolling back to

The simplicity of this example is good for focusing on the upgrade and rollback workflow, and highlighting some of the quirks, but it hides the power of Helm for major upgrades. A Helm release is an abstraction of an application, and different versions of the application might be modelled in different ways. A chart might use a ReplicationController in an early release, then change to a ReplicaSet and then a Deployment; as long as the user-facing parts remain the same, the internal workings become an implementation detail.

10.5 Understanding where Helm fits in

Helm adds a lot of value to Kubernetes but it's invasive - once you template your manifests there's no going back. Everyone on the team has to switch to Helm or you have to commit to having multiple sets of manifests: pure Kubernetes for the dev team and Helm for every other environment. You really don't want two sets of manifests getting out of sync, but equally Kubernetes itself is plenty to learn without adding Helm on top.

Whether Helm fits in for you depends very much on the type of applications you're packaging and the way your teams work. If your app is composed of 50+ microservices then dev teams might only work on a subset of the full app, running it natively or with Docker and Docker Compose, and a separate team owns the full Kubernetes deployment. In that

environment a move to Helm will reduce friction rather than increasing it, centralizing hundreds of YAML files into manageable charts.

There are a couple of other indicators that Helm is a good fit - a fully automated Continuous Deployment process can be easier to build with Helm, running test environments from the same chart version with custom values files, and running verification jobs as part of the deployment. And when you find yourself needing to template your Kubernetes manifests - which you will sooner or later - Helm gives you a standard approach, which is better than writing and maintaining your own tools.

That's all for Helm in this chapter, so it's time to tidy up the cluster before moving on to the lab.

TRY IT NOW Everything in this chapter was deployed with Helm, so we can use Helm to uninstall it all.

```
# uninstall all the releases:  
helm uninstall $(helm ls -q)
```

10.6 Lab

It's back to the to-do app again for the lab, and you're going to take a working set of Kubernetes manifests and package them up into a Helm chart. Don't worry, it's not the full-on app from chapter 8 with StatefulSets and backup Jobs, it's a much simpler version. Here are the goals:

- use the manifests in the `lab/todo-list` folder as the starting point (there are hints in the YAML for what needs templating)
- create the Helm chart structure
- template the resource names and any other values which need to be templated so the app can run as multiple releases
- add parameters for configuration settings to support running the app as different environments
- your chart should run as the Test configuration when installed with default values
- your chart should run as the Dev configuration when installed using the `lab/dev-values.yaml` values file

If you're planning on making use of Helm, this is one lab you should really find time for, because these are the exact set of tasks you'll need to do when you package apps in Helm. My solution is up on GitHub for you to check in the usual place:

<https://github.com/sixeyed/kiamol/blob/master/ch10/lab/README.md>

Happy Helming!

11

App development - developer workflows and CI/CD

This is the final chapter on Kubernetes in the real world, and the focus here is the practicality of developing and delivering software to run on Kubernetes. Whether you identify as a developer or you're on the ops side supporting developers, the move to containers impacts the way you work, the tools you use, and the amount of time and effort from making a code change to seeing it running in dev and test environments. In this chapter we'll examine how Kubernetes affects both the *inner loop* - the developer workflow on the local machine, and the *outer loop* - the CI/CD workflow which pushes changes to test and production.

How you use Kubernetes in your organization will be quite different from how you've used it so far in this book, because you'll use shared resources like clusters and image registries. As we explore delivery workflows in this chapter we'll also cover lots of small details which can trip you up as you make the change to the real world - things like using private registries and maintaining isolation on a shared cluster. The main focus of the chapter is to help you understand the choice between a Docker-centric workflow and something more like a Platform-as-a-Service (PaaS) running on Kubernetes.

11.1 The Docker developer workflow

Developers love Docker - it was voted the number one most wanted platform and the number two "most loved" in StackOverflow's annual survey two years in a row. Docker makes some parts of the developer workflow incredibly easy but at a cost: the Docker artifacts become central to the project, and that has an impact on the inner loop. You can run the app in a local environment using the exact same technologies as production, but only if you accept a different way of working.

In this section we'll walk through the developer workflow where Docker and Kubernetes are used in every environment, and where developers have their own dedicated cluster. You'll

need to have Docker running if you want to follow along with the exercises, so if your lab environment is Docker Desktop or Kind then you're good to go. The first thing we'll look at is developer on-boarding - joining a new project and getting up to speed as quickly as possible.

TRY IT NOW There's a whole new demo app for this chapter - a simple bulletin board where you can post details of upcoming events. It's written in Node.js, but you don't need to have Node.js installed to get up and running with the Docker workflow.

```
# switch to this chapter's folder in the source code:  
cd ch11  
  
# build the app:  
docker-compose -f bulletin-board/docker-compose.yml build  
  
# run the app:  
docker-compose -f bulletin-board/docker-compose.yml up -d  
  
# check the running containers:  
docker ps  
  
# and browse to the app at http://localhost:8010/
```

This is just about the simplest way you can get started as a developer on a new project. The only software you need installed is Docker, then you grab a copy of the code and off you go. You can see my output in figure 11.1 - I don't have Node.js installed on my machine and it doesn't matter whether you do or not, or what version you have; your results will be the same.

The Docker Compose file contains the path to the Dockerfile, so you can build the app with a Compose command - I'm hiding the output, you'll see a lot more.

```
PS>cd ch11
PS>
PS>docker-compose -f bulletin-board/docker-compose.yml build | Out-Null
Building bulletin-board
PS>
PS>docker-compose -f bulletin-board/docker-compose.yml up -d
Creating network "bulletin-board_default" with the default driver
Creating bulletin-board_bulletin-board_1 ... done
PS>
PS>docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
ED                 kiamol/ch11-bulletin-board:dev   "docker-entrypoint.s..."   25 seconds ago   Up 23 seconds      0.0.0.0:8010->8080/tcp   bulletin-board_bulletin
PS>
PS>
```

IMAGE	COMMAND	CREATED
kiamol/ch11-bulletin-board:dev	"docker-entrypoint.s..."	25 seconds
Up 23 seconds	0.0.0.0:8010->8080/tcp	bulletin-board_bulletin

Compose uses a desired-state approach like Kubernetes. This runs the app, starting containers in the background with the detached `-d` flag.

The app is up and running from source code - you don't need any compilers or runtime platforms installed.

Figure 11.1 Developer onboarding is a breeze with Docker and Compose - if there are no problems

Behind the magic there are two things: a Dockerfile which has all the steps to build and package the Node.js component, and a Docker Compose file which specifies all the components and the path to their Dockerfiles. There's only one component in this app, but there could be a dozen - all using different technologies - and the workflow would be the same. But this isn't how we're going to run the app in production, so if we want to use the same technology stack we can switch to running the app in Kubernetes locally, just using Docker for the build.

TRY IT NOW There are simple Kubernetes manifests for running the app using the local image. Remove the Compose version of the app and deploy it to Kubernetes instead.

```
# stop the app in Compose:
docker-compose -f bulletin-board/docker-compose.yml down
```

```
# deploy in Kubernetes:
kubectl apply -f bulletin-board/kubernetes/

# get the new URL:
kubectl get svc bulletin-board -o
  jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8011'

# and browse
```

That workflow is still pretty simple, although we now have three container artifacts to work with - the Dockerfile, the Compose file and the Kubernetes manifest. I have my own Kubernetes cluster, and with that I can run the app exactly as it will run in production. My output in figure 11.2 shows it's the same app, using the same local image, built with Docker Compose in the previous exercise.

Compose is a good way to get started but we're going to run on Kubernetes instead - this removes the Docker containers and network.

Run the app in Kubernetes instead, with a Deployment and a service.

```
PS>docker-compose -f bulletin-board/docker-compose.yml down
Stopping bulletin-board_bulletin-board_1 ... done
Removing bulletin-board_bulletin-board_1 ... done
Removing network bulletin-board_default
PS>
PS>kubectl apply -f bulletin-board/kubernetes/
deployment.apps/bulletin-board created
service/bulletin-board created
PS>
PS>kubectl get svc bulletin-board -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8011'
http://localhost:8011
PS>
PS>
```

It's the same app, running from the same Docker image you just built.

Figure 11.2 You can mix Docker and Kubernetes - using Compose to build images to run in Pods

Kubernetes is happy to use a local image which you've created or pulled with Docker, but there are some rules around whether it uses the local image or pulls it from a registry. If the image doesn't have an explicit tag in the name (so it uses the default :latest tag) then Kubernetes will always try to pull the image first - otherwise Kubernetes will use the local

image if it exists. You can override the rules by specifying an image pull policy - listing 11.1 shows the Pod spec for the bulletin board app, which includes an explicit policy.

Listing 11.1 - bb-deployment.yaml, specifying image pull policies

```
spec:                                     # this is the Pod spec within the Deployment
  containers:
    - name: bulletin-board
      image: kiamol/ch11-bulletin-board:dev
      imagePullPolicy: IfNotPresent  # prefer the local image if it exists
```

That's the sort of detail which can be a nasty stumbling block in the developer workflow. The Pod spec might be configured so the registry image is preferred, and then you can build your own local image as much as you like and never see the changes, because Kubernetes will always use the remote image. There are similar complications around image versions, because an image can be replaced with another version using the same name. That doesn't play well with Kubernetes' desired state approach, because if you deploy an update with an unchanged Pod spec, nothing happens, even if the image contents have changed.

Your first task on the project is to add some more detail to the events list - which is an easy code change for you. Testing your change is more challenging, because you can repeat the Docker Compose command to rebuild the image, but if you repeat the Kubectl command to deploy the changes, you'll see that nothing changes. If you're up on containers you can do some investigation to understand the problem, and delete the Pod to force a replacement - but if you're not, then your workflow is already broken.

TRY IT NOW You don't really need to make a code change - there's a new file which has the changes in it. Just replace the code file and rebuild the image, then delete the Pod to see the new app version running in the replacement Pod.

```
# remove the original code file:
rm bulletin-board/src/backend/events.js

# and replace it with an updated version:
cp bulletin-board/src/backend/events-update.js bulletin-board/src/backend/events.js

# rebuild the image using Compose:
docker-compose -f bulletin-board/docker-compose.yml build

# try to redeploy using Kubectl:
kubectl apply -f bulletin-board/kubernetes/

# delete the existing Pod to recreate it:
kubectl delete pod -l app=bulletin-board
```

You can see my output in figure 11.3. The updated application is running in the screenshot, but only after the Pod was manually deleted and then recreated by the Deployment controller, using the latest image version.

This updates the app to include a description in the event display.

Building the app replaces the container image, but uses the same image name.

```

PS>rm bulletin-board/src/backend/events.js
PS>
PS>cp bulletin-board/src/backend/events-update.js bulletin-board/src/backend/events.js
PS>
PS>docker-compose -f bulletin-board/docker-compose.yml build | out-Null
Building bulletin-board
PS>
PS>kubectl apply -f bulletin-board/kubernetes/deployment.apps/bulletin-board unchanged
service/bulletin-board unchanged
PS>
PS>kubectl delete pod -l app=bulletin-board
pod "bulletin-board-67ff99ddd4-5bbkq" deleted
PS>
```

The Pod spec hasn't changed, so Kubernetes doesn't do anything - it doesn't check the ID or hash of the image, only the image name.

Deleting the Pod forces a new one to be created which uses the latest image version, but it's not an intuitive workflow.

Figure 11.3 Docker images are mutable - but renaming images doesn't trigger an update in Kubernetes

If you chose a Docker-centric workflow then this is just one of the complications which will slow down and frustrate the dev teams (debugging and making live app updates are the next ones they'll hit). Container technologies are not easy topics to learn as you go, they really need some dedicated time to understand the principles, and not everyone on every team will want to make that investment.

The alternative is to centralize all the container technologies in a single team who provide a CI/CD pipeline which dev teams can plug into to deploy their apps. The pipeline takes care of packaging container images and deploying to the cluster, so the dev teams don't need to bring Docker and Kubernetes into their own work.

11.2 The Kubernetes-as-a-Service developer workflow

A Platform-as-a-Service experience running on top of Kubernetes is an attractive option for a lot of organizations. You can run a single cluster for all your test environments which also hosts the CI/CD service to take care of the messy details about running in containers. All the Docker artifacts are removed from the developer workflow so devs work on components directly, running Node.js and everything else they need on their machines, and they don't use containers locally.

This moves containers to the outer loop - when developers push changes to source control, that triggers a build which creates the container images, pushes them to a registry and deploys the new version to a test environment in the cluster. You get all the benefits of running in a container platform, without the friction containers bring to development. Figure 11.4 shows how that looks with one set of technology options.

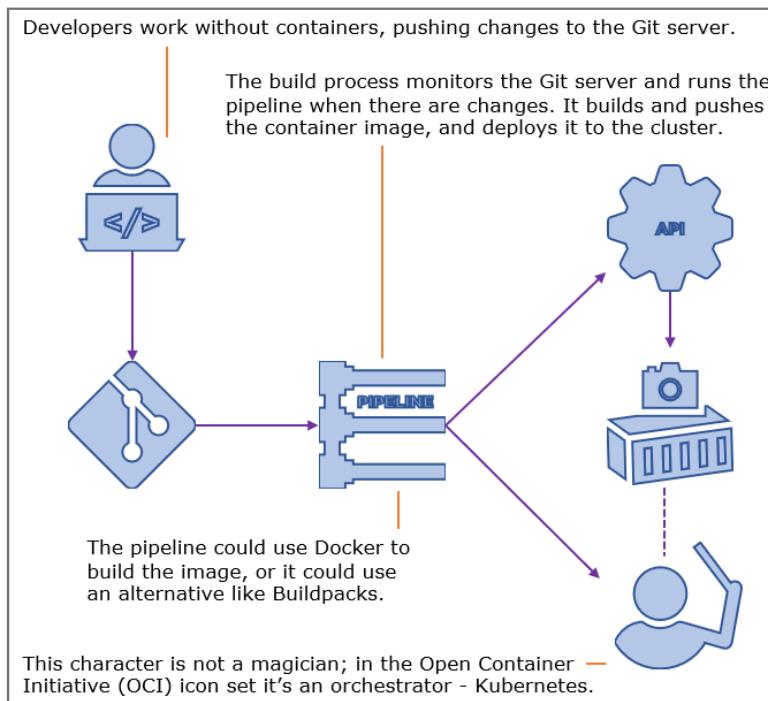


Figure 11.4 Using containers in the outer loop lets developers focus on code

The promise of this approach is that you get to run your app on Kubernetes without affecting the developer workflow or requiring every team member to skill up on Docker and Compose. It can work well in organizations where dev teams work on small components and a separate team assembles all the pieces into a working system, because only that team needs the container skills. You can also remove Docker entirely, which is useful if your cluster uses a different container runtime - but if you want to build container images without Docker you need to replace it with a lot of other moving pieces. You'll end up with more complexity overall, but centralized in the delivery pipelines and not the projects.

We'll walk through an example of that in this chapter, but to manage the complexity we'll do it in stages, starting with the view from inside the build service. To keep it simple we'll run our own Git server so we can push changes and trigger builds all from our lab cluster.

TRY IT NOW Gogs is a simple but powerful Git server which is published as an image on Docker Hub. It's a great way to run a private Git server in your organization, or to quickly spin up a backup if your online service goes offline. Run Gogs in your cluster and you can push a local copy of the book's source code.

```
# deploy the Git server:  
kubectl apply -f infrastructure/gogs.yaml  
  
# wait for it to spin up:  
kubectl wait --for=condition=ContainersReady pod -l app=gogs  
  
# add your local Git server to the book's repository -  
# this grabs the URL from the Service to use as the target:  
git remote add gogs $(kubectl get svc gogs -o  
    jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:3000/kiamol/kiamol.git')  
  
# and push the code to your server:  
git push gogs  
  
# find the server URL:  
kubectl get svc gogs -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:3000'  
  
# browse and sign in with username kiamol and password kiamol
```

Figure 11.5 shows my output. You don't need to run your own Git server for this workflow; it works in the same way using GitHub or any other source control system, but this makes for an easily reproducible environment - the Gogs setup for the chapter is preconfigured with a user account, so you can get up and running quickly.

Gogs is a powerful, lightweight open-source Git server.

Adding the local Git server as a "remote" means you can push the book's source code to your own server.

```

PS>kubectl apply -f infrastructure/gogs.yaml
service/gogs created
deployment.apps/gogs created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=gogs
pod/gogs-696779f5cc-4qwz9 condition met
PS>
PS>git remote add gogs $(kubectl get svc gogs -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:3000/kiamol/kiamol.git')
PS>
PS>git push gogs
Enumerating objects: 1267,
Counting objects: 100% (1267)
Delta compression using up to 83% space
Compressing objects: 100% (1267)
Writing objects: 100% (1267)
Total 1267 (delta 516), reused 0 objects, 303.22 KiB
remote: Resolving deltas: 100% (1267/1267)
To http://localhost:3000/kiamol/kiamol
 * [new branch]      master
PS>
PS>kubectl get svc gogs -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:3000'
http://localhost:3000

```

Browse to the server and sign in to see the source code - the username and password are set to `kiamol`.

Figure 11.5 Running your own Git server in Kubernetes is easy with Gogs

Now we have a local source control server which we can plug the other components into. The next is a system which can build container images, and to make this portable so it runs on any cluster, we need something which doesn't require Docker. There are a few options for that but one of the best is BuildKit - an open-source project from the Docker team. BuildKit started as a replacement for the image building component inside the Docker Engine, and it has a pluggable architecture, so you can build images with or without Dockerfiles. You can run BuildKit as a server, so other components in the toolchain can use it to build images.

TRY IT NOW Run BuildKit as a server inside the cluster, and confirm it has all the tools it needs to build container images without Docker.

```

# deploy BuildKit:
kubectl apply -f infrastructure/buildkitd.yaml

# wait for it to spin up:
kubectl wait --for=condition=ContainersReady pod -l app=buildkitd

```

```
# verify that Git and BuildKit are available:  
kubectl exec deploy/buildkitd -- sh -c 'git version && buildctl --version'  
  
# and check Docker isn't installed - this command will fail:  
kubectl exec deploy/buildkitd -- sh -c 'docker version'
```

You can see my output in figure 11.6, where the BuildKit Pod is running from an image with BuildKit and the Git client installed, but not Docker. It's important to realize that BuildKit is completely standalone - it doesn't connect to the container runtime in Kubernetes to build images; that's all going to happen within the Pod.

BuildKit is an open source tool for building container images from Dockerfiles or other inputs.

```
PS>kubectl apply -f infrastructure/buildkitd.yaml  
service/buildkitd created  
deployment.apps/buildkitd created  
PS>  
PS>kubectl wait --for=condition=ContainersReady pod -l app=buildkitd  
pod/buildkitd-7d75c7c97-72b17 condition met  
PS>  
PS>kubectl exec deploy/buildkitd -- sh -c 'git version && buildctl --version'  
git version 2.24.1  
buildctl github.com/moby/buildkit v0.7.1 ddd175c5a2cc24530ea8ff427887c22939ca4289  
PS>  
PS>kubectl exec deploy/buildkitd -- sh -c 'docker version'  
sh: docker: not found  
command terminated with exit code 127
```

The Pod is running from an image which has the Git client and BuildKit installed.
But not Docker - this will be a Docker-free pipeline.

Figure 11.6 BuildKit running as a container image building service, without requiring Docker

There are a few more pieces to set up before we can see the full PaaS workflow, but we have enough in place now to see how the build part of it works. We're targeting a Docker-free approach here so we're going to ignore the Dockerfile we used in the last section, and build the app into a container image directly from source code. How so? Using a CNCF project called Buildpacks, a technology pioneered by Heroku to power their PaaS product.

Buildpacks use the same concept as multistage Dockerfiles: running the build tools inside a container to compile the app and then packaging the compiled app on top of another container image which has the application runtime. You can do that with a tool called Pack, which you run over the source code for your app. Pack works out what language you're using, matches it to a Buildpack and then packages your app into an image - no Dockerfile required. Right now Pack only runs with Docker, but we're not using Docker so we can use an alternative to integrate Buildpacks with BuildKit.

TRY IT NOW We're going to step inside the build process, manually running a build which we'll go on to automate later in the chapter. Connect to the BuildKit Pod, pull the book's code from your local Git server and build it using Buildpacks instead of the Dockerfile.

```
# connect to a session on the BuildKit Pod:  
kubectl exec -it deploy/buildkitd -- sh  
  
# clone the source code from your Gogs server:  
cd ~  
git clone http://gogs:3000/kiamol/kiamol.git  
  
# switch to the app directory:  
cd kiamol/ch11/bulletin-board/  
  
# and build the app using BuildKit; the options tell BuildKit  
# to use Buildpacks instead of a Dockerfile as input, and to  
# produce an image as the output:  
buildctl build --frontend=gateway.v0 --opt source=kiamol/buildkit-buildpacks --local  
    context=src --output type=image,name=kiamol/ch11-bulletin-board:buildkit  
  
# leave the session when the build completes  
exit
```

It will take a while for that exercise to run, but keep an eye on the output from BuildKit and you'll see what's happening - first it downloads the component which provides the Buildpacks integration, then that runs and finds this is a Node.js app; it packages the app into a compressed archive, and then exports the archive into a container image which has the Node.js runtime installed. My output is in figure 11.7.

Connecting to a session on the BuildKit Pod to manually run the build and see how it all works.

This clones the source code from the Gogs server - we're inside the cluster here so the URL is the Kubernetes Service name.

```

PS>kubectl exec -it deploy/buildkitd -- sh
/ #
/ # cd ~
~ #
~ # git clone http://gogs:3000/kiamol/kiamol.git
Cloning into 'kiamol'...
remote: Enumerating objects: 1198, done.
remote: Counting objects: 100% (1198/1198), done.
remote: Compressing objects: 100% (645/645), done.
remote: Total 1198 (delta 470), reused 1198 (delta 470)
Receiving objects: 100% (1198/1198), 9.55 MiB | 45.07 MiB/s, done.
Resolving deltas: 100% (470/470), done.

~ #
~ # cd kiamol/ch11/bulletin-board/
~/kiamol/ch11/bulletin-board #
~/kiamol/ch11/bulletin-board # buildctl build --frontend=gateway.v0 --opt source=kiamol/buildkit-buildpacks --local context=src --output type=image,name=kiamol/ch11-bulletin-board:buildkit
[+] Building 2.9s (0/1)
=> resolve image config for docker.io/kiamol/buildkit-buildpacks:latest 2.9s
=> exporting to image 5.9s
=> => exporting layers 5.8s
=> => exporting manifest sha256:2577da0cc0bd405b05ab04e99130921b2c427464d48 0.0s
=> => exporting config sha256:84c66f86989bf98e2f86beff856f2e89b43b339129a82 0.0s
~/kiamol/ch11/bulletin-board #
~/kiamol/ch11/bulletin-board # exit

```

This complicated BuildKit command is the alternative to building with Docker. It uses a custom component called `buildkit-buildpacks` to use Buildpacks instead of a Dockerfile.

I've snipped the output - the final logs show BuildKit creating a container image in the local image cache in the Pod.

Figure 11.7 Building container images without Docker and Dockerfiles adds a lot of complexity

You can't run a container from that image on the BuildKit Pod, because it doesn't have a container runtime configured - but BuildKit is able to push images to a registry after building, and that's what we'll do in the complete workflow. So far we've seen that you can build and package your apps to run in containers without Dockerfiles or Docker, which is pretty impressive - but it comes at a cost.

The biggest issue is the complexity of the build process and the maturity of all the pieces. BuildKit is a stable tool, but it isn't anywhere near as well used as the standard Docker build engine. Buildpacks are a promising approach, but the dependency on Docker means they don't work well in a Docker-free environment like a managed Kubernetes cluster in the cloud. The component we're using to bridge them is a tool written by Tonis Tigi, a maintainer on the BuildKit project. It's really just a proof-of-concept to plug Buildpacks into Buildkit; it works

well enough to demonstrate the workflow, but it's not something you would want to rely on to build apps for production.

There are alternatives - GitLab is a product which combines a Git server with a build pipeline that uses Buildpacks, and Jenkins X is a native build server for Kubernetes. But they are complex products themselves, and you need to be aware that if you want to remove Docker from your developer workflow, you'll be trading it for more complexity in the build process. You'll be able to decide whether the result is worth it by the end of this chapter, and next we'll look at how you can isolate workloads in Kubernetes, so a single cluster can run your delivery pipelines and all your test environments.

11.3 Isolating workloads with contexts and namespaces

Way back in chapter 3 I introduced Kubernetes namespaces - and very quickly moved on. You need to be aware of them to make sense of the fully-qualified DNS names Kubernetes uses for Services, but you don't need to use them until you start dividing up your cluster. Namespaces are a grouping mechanism - every Kubernetes object belongs to a namespace - and multiple namespaces can be used to create virtual clusters from one real cluster.

Namespaces are very flexible and organizations use them in different ways. You might use them in a production cluster to divide it up for different products. Or to divide up a non-production cluster for different environments - integration test, system test and user testing. You might even have a development cluster where each developer has their own namespace, so they don't need to run their own cluster. Namespaces are a boundary where you can apply security and resource restrictions, so they support all these scenarios. We'll use a dedicated namespace in our CI/CD deployment, but we'll start with a simple walkthrough.

TRY IT NOW Kubectl is namespace aware. You can explicitly create a namespace, then deploy and query resources using the `namespace` flag - this creates a simple sleep Deployment.

```
# create a new namespace:
kubectl create namespace kiamol-ch11-test

# deploy a sleep Pod in the new namespace:
kubectl apply -f sleep.yaml --namespace kiamol-ch11-test

# list sleep Pods - this won't return anything:
kubectl get pods -l app=sleep

# now list the Pods in the namespace:
kubectl get pods -l app=sleep -n kiamol-ch11-test
```

My output is in figure 11.8 where you can see that namespaces are an essential part of resource metadata - you need to explicitly specify the namespace to work with an object in Kubectl. The only reason we've avoided this for the first 10 chapters is that every cluster has a namespace called `default` which is used if you don't specify a namespace, and that's where we've created and used everything so far.

Namespaces are Kubernetes objects which group other objects.

You can specify the namespace where manifests should be deployed.

```
PS>kubectl create namespace kiamol-ch11-test
namespace/kiamol-ch11-test created
```

```
PS>
PS>kubectl apply -f sleep.yaml --namespace kiamol-ch11-test
deployment.apps/sleep created
```

```
PS>
PS>kubectl get pods -l app=sleep
No resources found in default namespace.
```

```
PS>
PS>kubectl get pods -l app=sleep -n kiamol-ch11-test
NAME           READY   STATUS    RESTARTS   AGE
sleep-b8f5f69-rp9fz   1/1     Running   0          16s
```

Kubectl queries only look in the default namespace, so this doesn't find the new Pod.

To see the Pod you need to explicitly target the namespace.

Figure 11.8 Namespaces isolate workloads - you can use them to represent different environments

Objects within a namespace are isolated, so you can deploy the same apps with the same object names in different namespaces. Resources can't see resources in other namespaces - Kubernetes networking is flat, so Pods in different namespaces can communicate through Services, but a controller only looks for Pods in its own namespace. Namespaces are ordinary Kubernetes resources too. Listing 11.2 shows a namespace spec in YAML, along with the metadata for another sleep Deployment which uses the new namespace.

Listing 11.2 - sleep-uat.yaml, a manifest that creates and targets a namespace

```
apiVersion: v1
kind: Namespace      # namespace specs only need a name
metadata:
  name: kiamol-ch11-uat
---
apiVersion: apps/v1
kind: Deployment
metadata:           # the target namespace is part of the
  name: sleep        # object metadata. The namespace needs
  namespace: kiamol-ch11-uat  # to exist or the deployment fails.
# Pod spec follows
```

The Deployment and Pod spec in that YAML file use the same names as the objects you deployed in the last exercise, but because the controller is set to use a different namespace,

all the objects it creates will be in that namespace too. When you deploy this manifest you'll see the new objects created without any naming collisions.

TRY IT NOW Create a new "UAT" namespace and Deployment from the YAML in listing 11.2 - the controller uses the same name, and you can see objects across namespaces using Kubectl. Deleting a namespace deletes all its resources.

```
# create the namespace and Deployment:  
kubectl apply -f sleep-uat.yaml  
  
# list the sleep Deployments in all namespaces:  
kubectl get deploy -l app=sleep --all-namespaces  
  
# delete the new UAT namespace:  
kubectl delete namespace kiamol-ch11-uat  
  
# and list Deployments again:  
kubectl get deploy -l app=sleep --all-namespaces
```

You can see my output in figure 11.9. The original sleep Deployment didn't specify a namespace in the YAML file and we created it in the `kiamol-ch11-test` namespace by specifying that in the Kubectl command. The second sleep Deployment specified the `kiamol-ch11-uat` namespace in the YAML, so it was created there without needing a Kubectl namespace flag.

This manifest contains a namespace and a Deployment configured to use the new namespace.

Querying objects with the `--all-namespaces` flag shows results across all namespaces. There are two Deployments with the same name running in different namespaces.

```

PS> kubectl apply -f sleep-uat.yaml
namespace/kiamol-ch11-uat created
deployment.apps/sleep created
PS>
PS> kubectl get deploy -l app=sleep --all-namespaces
NAMESPACE      NAME    READY   UP-TO-DATE   AVAILABLE   AGE
kiamol-ch11-test  sleep  1/1     1           1           13m
kiamol-ch11-uat  sleep  1/1     1           1           21s
PS>
PS> kubectl delete namespace kiamol-ch11-uat
namespace "kiamol-ch11-uat" deleted
PS>
PS> kubectl get deploy -l app=sleep --all-namespaces
NAMESPACE      NAME    READY   UP-TO-DATE   AVAILABLE   AGE
kiamol-ch11-test  sleep  1/1     1           1           13m

```

Deleting a namespace deletes all the objects in the namespace.

Now the only sleep Deployment is the one in the test namespace.

Figure 11.9 Namespaces are a useful abstraction for managing groups of objects

In a shared cluster environment you might regularly use different namespaces - deploying apps in your own dev namespace and then looking at logs in the test namespace. Switching between them using Kubectl flags is time consuming and error prone, and Kubectl provides an easier way with contexts. A context defines the connection details for a Kubernetes cluster and sets the default namespace to use in Kubectl commands. Your lab environment will already have a context set up, and you can modify that to switch namespaces.

TRY IT NOW Show your configured contexts and update the current one to set the default namespace to the test namespace.

```

# list all contexts:
kubectl config get-contexts

# update the default namespace for the current context:
kubectl config set-context --current --namespace=kiamol-ch11-test

# list Pods in the default namespace:
kubectl get pods

```

You can see in figure 11.10 that setting the namespace for the context sets the default namespace for all Kubectl commands - any queries which don't specify a namespace and any

create commands where the YAML doesn't specify a namespace will now all use the test namespace. You can create multiple contexts, all using the same cluster but different namespaces and switch between them with Kubectl's `use-context` command.

Contexts show the cluster name and the default namespace. A blank namespace uses the default, called `default`. The asterisk shows the currently active context.

```
PS>kubectl config get-contexts
CURRENT  NAME          CLUSTER      AUTHINFO      NAMESPACE
*        docker-desktop docker-desktop docker-desktop docker-desktop
PS>
PS>kubectl config set-context --current --namespace=kiamol-ch11-test
Context "docker-desktop" modified.
PS>
PS>kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
sleep-b8f5f69-rp9fz   1/1     Running   0          39m
```

Updating the context lets you set a different default namespace.
Now queries are restricted to the `kiamol-ch11-test` namespace, so the sleep Pod is shown.

Figure 11.10 Contexts are an easy way to switch between namespaces and clusters

The other important use for contexts is to switch between clusters. When you set up Docker Desktop or Kind they create a context for your local cluster - the details all live in a configuration file which is stored in the `.kube` directory in your home folder. Managed Kubernetes services usually have a feature to add a cluster to your config file, so you can work with remote clusters from your local machine. The remote API server will be secured using TLS, and your Kubectl configuration will use a client certificate to identify you as the user - you can see those security details by viewing the configuration.

TRY IT NOW Reset your context to use the default namespace, and then print out the details of the client configuration.

```
# setting the namespace to blank resets the default:
kubectl config set-context --current --namespace=

# printing out config shows your cluster connection:
kubectl config view
```

Figure 11.11 shows my output, with a local connection to my Docker Desktop cluster using TLS certificates - which aren't shown by Kubectl - to authenticate the connection.

This resets the namespace to the default for the current context.

```
PS>kubectl config set-context --current --namespace=
Context "docker-desktop" modified.
PS>
PS>kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://kubernetes.docker.internal:6443
    name: docker-desktop
contexts:
- context:
    cluster: docker-desktop
    user: docker-desktop
    name: docker-desktop
current-context: docker-desktop
kind: Config
preferences: {}
users:
- name: docker-desktop
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

Kubectl connects to the cluster API using the connection details in the context, which also contains a TLS client certificate for authentication.

Figure 11.11 Contexts contain the connection details for the cluster, which could be local or remote

Kubectl can also use a token to authenticate with the Kubernetes API server, and Pods are provided with a token they can use as a Secret, so apps running in Kubernetes can connect to the Kubernetes API to query or deploy objects. And that's a long way to getting where we want to go next: we'll run a build server in a Pod which triggers a build when the source code changes in Git, builds the image using BuildKit and deploys it to Kubernetes in the test namespace.

11.4 Continuous Delivery in Kubernetes without Docker

Actually we're not quite there yet, because the build process needs to push the image to a registry, so Kubernetes can pull it to run Pod containers. Real clusters have multiple nodes and each of them needs to be able to access the image registry. That's been easy so far because we've used public images on Docker Hub, but in your own builds you'll push to a private repository first. Kubernetes supports pulling private images by storing registry credentials in a special type of Secret object.

You'll need to have an account set up on an image registry to follow along with this section
- Docker Hub is fine, or you can create a private registry on the cloud using Azure Container

Registry (ACR) or Amazon Elastic Container Registry (ECR). If you're running your cluster in the cloud then it makes sense to use that cloud's registry to reduce download times, but all registries use the same API as Docker Hub, so they're interchangeable.

TRY IT NOW Create a Secret to store registry credentials - to make it easier to follow along, there's a script to collect the credentials into local variables. Don't worry, the scripts don't email your credentials to me...

```
# collect the details - on Windows:  
. .\set-registry-variables.ps1  
  
# OR on Linux:  
. ./set-registry-variables.sh  
  
# create the secret using the details from the script:  
kubectl create secret docker-registry registry-creds --docker-server=$REGISTRY_SERVER --  
  docker-username=$REGISTRY_USER --docker-password=$REGISTRY_PASSWORD  
  
# show the secret details:  
kubectl get secret registry-creds
```

My output is in figure 11.12. I'm using Docker Hub which lets you create temporary access tokens which you can use in the same way as a password for your account. When I'm done with this chapter I'll revoke the access token - that's a nice security feature in Hub.

This script collects registry authentication details into variables which you can use in later commands.

```
PS> .\set-registry-variables.ps1
Registry server (blank for Docker Hub):
Username: sixeyed
Password: *****
PS>
PS>kubectl create secret docker-registry registry-creds --docker-server=$REGISTRY_S
ERVER --docker-username=$REGISTRY_USER --docker-password=$REGISTRY_PASSWORD
secret/registry-creds created
PS>
PS>kubectl get secret registry-creds
NAME          TYPE           DATA   AGE
registry-creds  kubernetes.io/dockerconfigjson  1      8s
```

The docker-registry is a special type of Secret which Kubernetes can use to authenticate and pull private images.
It's stored as JSON which contains your registry password in plain text.

Figure 11.12 Your organization may use a private image registry - you need a Secret to authenticate

Okay: now we're ready. We have a Docker-less build server running in the BuildKit Pod, a local Git server we can use to quickly iterate over the build process, and a registry Secret stored in the cluster. We can use all those pieces with an automation server to run the build pipeline, and we'll be using Jenkins for that. Jenkins has a long legacy as a build server and

it's very popular - but you don't need to be a Jenkins guru to set up this build, because I have it already configured in a custom Docker Hub image.

The Jenkins image for this chapter has the BuildKit and Kubectl command lines installed, and the Pod is set up to surface credentials in the right places. The registry Secret you created in the last exercise is mounted in the Pod container, so BuildKit can use it to authenticate to the registry when it pushes the image. Kubectl is configured to connect to the local API server in the cluster using the token Kubernetes provides in another Secret. Deploy the Jenkins server and you can check everything is correctly configured.

TRY IT NOW Jenkins gets everything it needs from Kubernetes Secrets, using a startup script in the container image. Start by deploying Jenkins and confirming it can connect to Kubernetes.

```
# deploy Jenkins:  
kubectl apply -f infrastructure/jenkins.yaml  
  
# wait for the Pod to spin up:  
kubectl wait --for=condition=ContainersReady pod -l app=jenkins  
  
# check that Kubectl can connect to the cluster:  
kubectl exec deploy/jenkins -- sh -c 'kubectl version --short'  
# and that the registry secret is mounted:  
kubectl exec deploy/jenkins -- sh -c 'ls -l /root/.docker'
```

In that exercise you'll see Kubectl report the version of your own Kubernetes lab cluster - that confirms the Jenkins Pod container is set up correctly to authenticate to Kubernetes, so it can deploy applications to the same cluster where it is running. My output is in figure 11.13.

The Jenkins manifest mounts the Docker registry Secret, and Kubernetes provides its own API token Secret. A startup script in the image configures all the authentication.

Kubectl shows the version of the command line client, and the version of the cluster it's connected to. Running this in the Jenkins Pod shows it can connect to the Kubernetes API.

```
PS>kubectl apply -f infrastructure/jenkins.yaml
service/jenkins created
deployment.apps/jenkins created

PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=jenkins
pod/jenkins-888fb995-6g48p condition met
PS>
PS>kubectl exec deploy/jenkins -- sh -c 'kubectl version --short'
Client Version: v1.18.4
Server Version: v1.16.6-beta.0
PS>
PS>kubectl exec deploy/jenkins -- sh -c 'ls -l /root/.docker'
total 0
lrwxrwxrwx 1 root root 18 Jun 22 13:46 config.json -> ..data/config.json
```

The Docker registry secret is mounted to the path where BuildKit looks for authentication. If you print the file contents you'll see your registry password in plain text, which is why I haven't.

Figure 11.13 Jenkins runs the pipeline, so it needs authentication details for Kubernetes and the registry

Everything is in place now for Jenkins to fetch application code from the Gogs Git server, connect to the BuildKit server to build the container image using Buildpacks and push it to the registry, and deploy the latest application version to the test namespace. That work is already set up using a Jenkins pipeline, but the pipeline steps just use simple build scripts in the application folder. Listing 11.3 shows the build stage which packages and pushes the image.

Listing 11.3 - build.sh, the build script using BuildKit

```
buildctl --addr tcp://buildkitd:1234 \
    build \
    --frontend=gateway.v0 \
    --opt source=kiamol/buildkit-buildpacks \
    --local context=src \
    --output type=image,name=${REGISTRY_SERVER}/${REGISTRY_USER}/bulletin-
    board:${BUILD_NUMBER}-kiamol,push=true # push the output to the registry
```

The script is an extension of the simpler BuildKit command you ran in section 11.2, when you were pretending to be the build server. The Buildctl command uses the same integration component for Buildpacks, so there's no Dockerfile in here. This command runs inside the Jenkins Pod so it specifies an address for the BuildKit server - which is running in a separate Pod behind the Service called `buildkitd`. No Docker here either. The variables in the image name are all set by Jenkins, but they're standard environment variables so there's no dependency on Jenkins in the build scripts.

When this stage of the pipeline completes, the image will have been built and pushed to the registry. The next stage is to deploy the updated application, which is in a separate script shown in listing 11.4.

Listing 11.4 - run.sh, the deployment script using Helm

```
helm upgrade --install --atomic \
    --set registryServer=${REGISTRY_SERVER}, \
        registryUser=${REGISTRY_USER}, \
            imageBuildNumber=${BUILD_NUMBER} \
                --namespace kiamol-ch11-test \
                    # deploy to the test namespace
bulletin-board \
helm/bulletin-board
```

The deployment uses Helm with a chart that has values for the parts of the image name. They're set from the same variables used in the build stage, which are compiled from the Docker registry Secret and the build number in Jenkins. In my case the first build pushes an image to Docker Hub named `sixeyed/bulletin-board:1-kiamol`, and installs a Helm release using that image. To run the build in your cluster and push to your registry, you just need to log in to Jenkins and enable the build - the pipeline itself is already set up.

TRY IT NOW Jenkins is running and configured, but the pipeline job isn't enabled. Log in to enable the job, and you will see the pipeline execute and the app deployed to the cluster.

```
# get the URL for Jenkins:
kubectl get svc jenkins -o
    jsonpath='http://.status.loadBalancer.ingress[0].*:8080/job/kiamol'

# browse and login with username kiamol and password kiamol;
# if Jenkins is still setting itself up you'll see a wait screen

# click enable for the Kiamol job and wait...
# when the pipeline completes, check the deployment:
kubectl get pods -n kiamol-ch11-test -l app.kubernetes.io/name=bulletin-board -o=custom-
    columns=NAME:.metadata.name,IMAGE:.spec.containers[0].image

# find the URL of the test app:
kubectl get svc -n kiamol-ch11-test bulletin-board -o
    jsonpath='http://.status.loadBalancer.ingress[0].*:8012'

# and browse
```

The build should be fast because it's using the same BuildKit server that has already cached the images for the Buildpack build from section 11.2. When the build has completed you can browse to the application deployed by Helm in the test namespace and see the app running - mine is in figure 11.14.

Browse to Jenkins and log in with credentials `kiamol/kiamol` to enable the pipeline project.

```
PS>kubectl get svc jenkins -o jsonpath='http://.status.loadBalancer.ingress[0].*:8080/job/kiamol'
http://localhost:8080/job/kiamol
PS>
PS>
```

```
PS>kubectl get pods -n kiamol-ch11-test -l app.kubernetes.io/name=bulletin-board -o custom-columns=NAME:.metadata.name,IMAGE:.spec.containers[0].image
NAME                                IMAGE
bulletin-board-5c4475f6d4-cxkqc    docker.io/sixeyed/bulletin-board:1-kiamol
PS>
PS>kubectl get svc -n kiamol-ch11-test bulletin-board -o jsonpath='http://.status.loadBalancer.ingress[0].*:8012'
http://localhost:8012
```

When the pipeline completes you can read the logs for each stage in Jenkins, and you'll see the Pod is running a container with the image from your registry.

The app has been deployed with Helm to the test namespace.

Figure 11.14 The pipeline in action, built and deployed to Kubernetes without Docker or Dockerfiles

So far so good. We're playing the ops role so we understand all the moving parts in the delivery of this app - we would own the pipeline in the Jenkinsfile and the application specs in the Helm chart. There are lots of small fiddly details in there, like the templated image name and the image pull Secret in the Deployment YAML - but from the developer point of view, that's all hidden.

The dev view is that you can work on the app using your local environment, push changes and see them running at the test URL, without worrying what happens in between. We can see that workflow now. You made an application change earlier to add event descriptions to the site, and to deploy that all you need to do is push the changes to your local Git server and wait for the Jenkins build to complete.

TRY IT NOW Push your code change to your Gogs server; Jenkins will see the change within one minute and start a new build. That will push a new image version to your registry and update the Helm release to use that version.

```
# add your code change and push it to Git:  
git add bulletin-board/src/backend/events.js  
git commit -m 'Add event descriptions'  
git push gogs  
  
# browse back to Jenkins and wait for the new build to finish  
  
# check the application Pod is using the new image version:  
kubectl get pods -n kiamol-ch11-test -l app.kubernetes.io/name=bulletin-board -o=custom-  
columns=NAME:.metadata.name,IMAGE:.spec.containers[0].image  
  
# browse back to the app
```

This is the `git push` PaaS workflow applied to Kubernetes. We're dealing with a simple app here, but the approach is the same for a large system with many components - a shared namespace could be the deployment target for all the latest versions, pushed by many different teams. Figure 11.15 shows an application update in Kubernetes triggered from a push of code, with no requirement for developers to use Docker, Kubernetes or Helm.

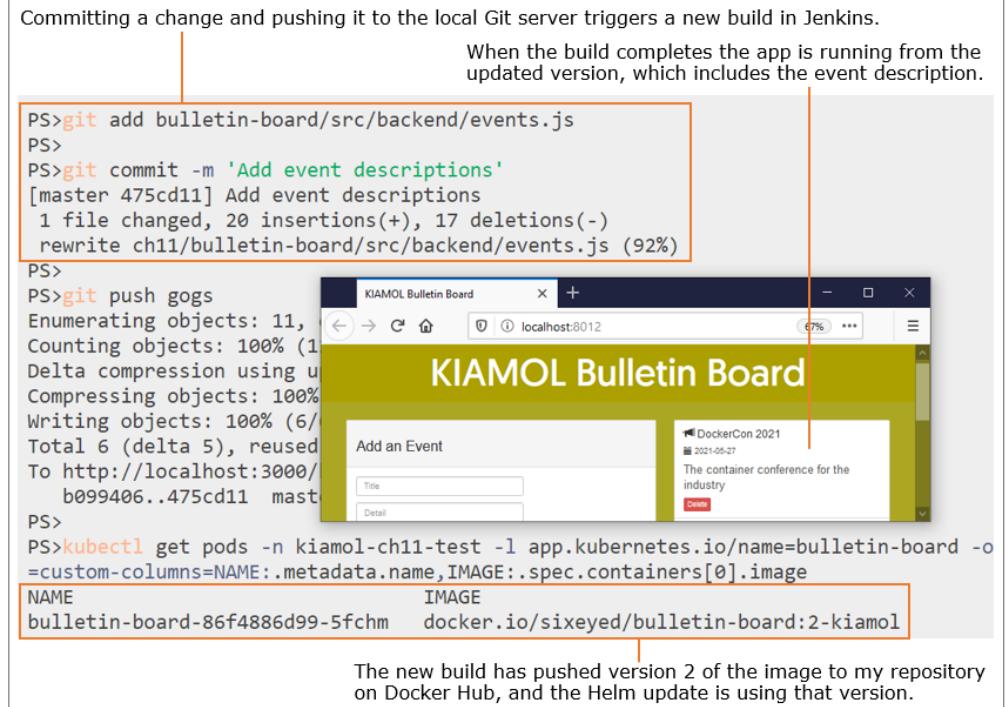


Figure 11.15 It's PaaS on your own Kubernetes cluster. A lot of complexity is hidden from the developer

Of course the PaaS approach and the Docker approach are not mutually exclusive. If your cluster is running on Docker then you can take advantage of a simpler build process for Docker-based apps, but still support a Docker-free PaaS approach for other apps, all in the same cluster. There are definite benefits and drawbacks to each approach, and we'll end by looking at how you should choose between them.

11.5 Evaluating developer workflows on Kubernetes

In this chapter we've looked at developer workflows at extreme ends of the spectrum, from teams who fully embrace containers and want to make them front and center in every environment, to teams who don't want to add any ceremony to their dev process, want to keep working natively and leave all the container bits to the CI/CD pipeline. There are plenty of places in between, and the likelihood is that you'll build an approach to suit your organization, your application architectures and your Kubernetes platform.

The decision is as much about culture as about technology. Do you want every team to level up on container knowledge, or do you want to centralize that knowledge in a service team and leave the developer teams to focus on delivering software? Although I'd love to see copies of *Learn Docker in a Month of Lunches* and *Learn Kubernetes in a Month of Lunches* on every

desk, skilling up on containers needs a pretty big commitment. But here are the major advantages I see in keeping Docker and Kubernetes visible in your projects:

- the PaaS approach is complicated and bespoke - you'll be plugging together lots of different technologies with different maturity levels and support structures
- the Docker approach is flexible - you can add any dependencies and setup you need in a Dockerfile, while PaaS approaches are more prescriptive, so they won't fit every app
- PaaS technologies don't have the optimizations you can get when you fine-tune your Docker images; the bulletin board image from the Docker workflow is 95MB compared to 1GB for the Buildpacks version - that's a much smaller surface area to secure
- the commitment to learning Docker and Kubernetes pays off because they're portable skills - developers can easily move between projects using a standard toolset
- teams don't have to use the full container stack, they can opt-out at different stages - some devs might just use Docker to run containers, others might use Docker Compose and others Kubernetes
- distributed knowledge makes for a better collaborative culture - centralized service teams can be resented for being the only ones who get to play with all the fun technology.

Ultimately it's a decision for your organization and teams, and the pain of migrating from the current workflow to the desired workflow needs to be considered. In my own consulting work I'm often balancing dev and ops roles, and I tend to be pragmatic. When I'm actively developing I use native tooling (I typically work on .NET projects using Visual Studio), but before I push any changes I run the CI process locally to build container images with Docker Compose and then spin everything up in my local Kubernetes cluster. That won't fit every scenario, but I find it a good balance between development speed and confidence that my changes will work the same way in the next environment.

That's all for the developer workflow, so we can tidy up the cluster before we move on - but leave your build components running (Gogs, BuildKit and Jenkins) - you'll need them for the lab.

TRY IT NOW Remove the bulletin board deployments.

```
# uninstall the Helm release from the pipeline:  
helm -n kiamol-ch11-test uninstall bulletin-board  
  
# delete the manual deployment:  
kubectl delete all -l app=bulletin-board
```

11.6 Lab

This lab is a bit nasty, so I'll apologize in advance - but I want you to see that going down the PaaS path with a custom set of tools has danger in store. The bulletin board app for this chapter

used a very old version of the Node runtime, version 10.5.0, and in the lab that needs updating to a more recent version. There's a new source code folder for the lab which uses Node 10.6.0, and your job is to set up a pipeline to build that version - and then find out why it fails and fix it. There are a few hints here because the goal isn't for you to learn Jenkins, but to see how to debug failing pipelines.

- Start by creating a new item from the Jenkins home page, choose the option to copy an existing job and copy the `kiamol` job; call the new job anything you like
- In the new job configuration in the Pipeline tab, change the path to the pipeline file to the new source code folder - `ch11/lab/bulletin-board/Jenkinsfile`
- Build the job and look through the logs to find out why it failed
- You'll need to make a change in the lab source folder and push it to Gogs to fix the build

My sample solution is up on GitHub with some screenshots for the Jenkins setup to help you: <https://github.com/sixeyed/kiamol/blob/master/ch11/lab/README.md>

12

Empowering self-healing apps

Kubernetes models your application with abstractions over the compute and networking layers. The abstractions allow Kubernetes to control network traffic and container lifetimes, so it can take corrective action if parts of your app fail. If you have enough detail in your specifications, the cluster can find and fix temporary problems and keep applications online. Those are self-healing applications, which ride out any transient issues without needing a human to guide them. In this chapter you'll learn about how to model that, using container probes to test for health and imposing resource limits so apps don't soak up too much compute.

There are limits to Kubernetes' healing powers, and you'll learn those in this chapter too. We're mainly going to look at how you keep your apps running without manual administration, but we'll also look again at applications updates. Updates are the most likely cause of downtime, and we'll look at some additional features of Helm which can keep your apps healthy during update cycles.

12.1 Routing traffic to healthy Pods using readiness probes

Kubernetes knows if your Pod container is running, but it doesn't know if the application inside the container is healthy. Every app will have its own idea of what "healthy" means - it might be a 200 OK response to an HTTP request - and Kubernetes provides a generic mechanism for testing health using *container probes*. Docker images can have healthchecks configured, but Kubernetes ignores them in favor of its own probes. Probes are defined in the Pod spec and they execute on a fixed schedule, testing some aspect of the application and returning an indicator to say if the app is still healthy.

If the probe response says the container is unhealthy, Kubernetes will take action - and the action it takes depends on the type of probe. *Readiness probes* take action at the network level, so they're for components which listen for network requests. If the Pod container is unhealthy, the Pod is taken out of the ready state, and it is removed from the list of active Pods for a Service. Figure 12.1 shows how that looks for a Deployment with multiple replicas, where one Pod is unhealthy.

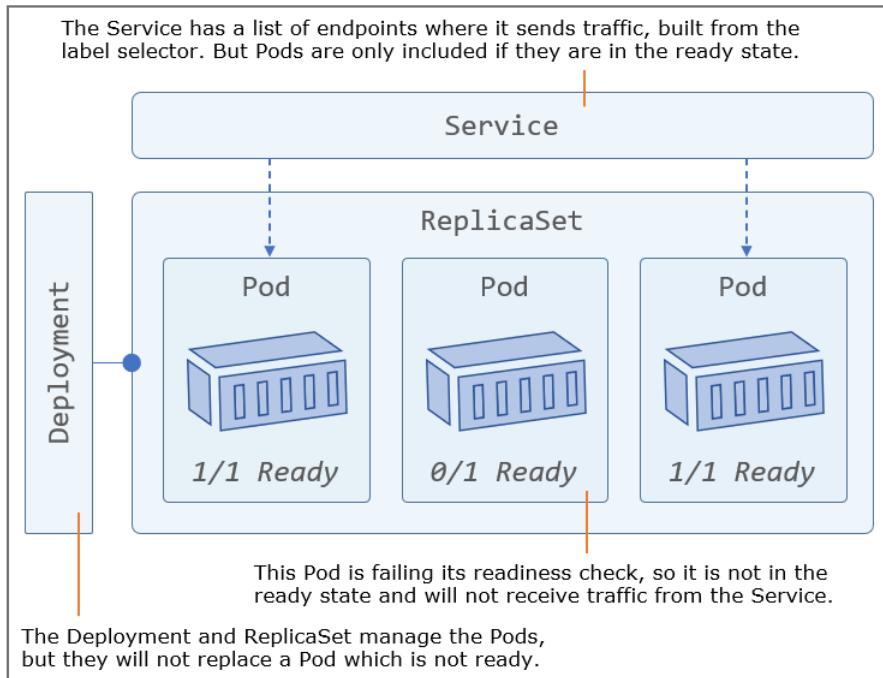


Figure 12.1 The list of endpoints for a Service excludes Pods which are not ready to receive traffic

Readiness probes are a great way to manage temporary load issues. Some Pods might be overloaded, returning a 503 status code to every request. If the readiness probe checks for a 200 response and those Pods return 503 they will be removed from the Service and they'll stop receiving requests. Kubernetes keeps running the probe after it has failed, so if the overloaded Pod has a chance to recover while it's resting, the probe will succeed again and the Pod will be enlisted back into the Service.

The random number generator we've used in this book has a couple of useful features we can use to see how this works. The API can run in a mode where it fails after a certain number of requests, and it has an HTTP endpoint which returns whether it is healthy or in a failed state. We'll start by running it without a readiness probe so we can understand the problem.

TRY IT NOW Run the API with multiple replicas and see what happens when the application fails without any container probes to test it.

```
# switch to the chapter's directory:  
cd ch12  
  
# deploy the random number API:  
kubectl apply -f numbers/
```

```
# wait for it to be ready:  
kubectl wait --for=condition=ContainersReady pod -l app=numbers-api  
  
# check the Pod's registered as Service endpoints:  
kubectl get endpoints numbers-api  
  
# save the URL for the API in a text file:  
kubectl get svc numbers-api -o jsonpath='http://{.status.loadBalancer.ingress[0].*}:8013' >  
api-url.txt  
  
# call the API - after returning, this app server is now unhealthy:  
curl "$(cat api-url.txt)/rng"  
  
# test the health endpoints to check:  
curl "$(cat api-url.txt)/healthz"; curl "$(cat api-url.txt)/healthz"  
  
# and confirm the Pods used by the service:  
kubectl get endpoints numbers-api
```

You'll see from that exercise that the Service keeps both Pods in its list of endpoints, even though one of them is unhealthy and will always return a 500 error response. My output in figure 12.2 shows two IP addresses in the endpoint list before and after the request which causes one instance to become unhealthy.

There are two Pods matching the Service selector, so there are two endpoints.

The API is configured to fail after a single request, so now one of the application containers is unhealthy.

```

PS>cd ch12
PS>
PS>kubectl apply -f numbers/
service/numbers-api created
deployment.apps/numbers-api created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=numbers-api
pod/numbers-api-7789f9c587-g859k condition met
pod/numbers-api-7789f9c587-rrg4p condition met
PS>
PS>kubectl get endpoints numbers-api
NAME      ENDPOINTS          AGE
numbers-api  10.1.1.194:80,10.1.1.195:80  13s
PS>
PS>kubectl get svc numbers-api -o jsonpath='http://{.status.loadBalancer.ingress[0]
.*}:8013' > api-url.txt
PS>
PS>curl "$(cat api-url.txt)/rng"
63
PS>
PS>curl "$(cat api-url.txt)/healthz"; curl "$(cat api-url.txt)/healthz"
{"message": "Unhealthy"}Ok
PS>
PS>kubectl get endpoints numbers-api
NAME      ENDPOINTS          AGE
numbers-api  10.1.1.194:80,10.1.1.195:80  65s

```

Checking the health endpoint shows one healthy and one unhealthy Pod - these requests are load-balanced, so you may need to repeat a few times to see this.

Both Pods are still in the Service list.

Figure 12.2 Application containers may be unhealthy but the Pod stays in the ready state

This happens because Kubernetes doesn't know one of the Pods is unhealthy. The application in the Pod container is still running and Kubernetes doesn't know there's a health endpoint it can use to see if the app really is working correctly. You can give it that information with a readiness probe in the container spec for the Pod - listing 12.1 shows an update to the API spec which includes the health check.

Listing 12.1 - api-with-readiness.yaml, a readiness probe for the API container

```

spec:           # this is the Pod spec in the Deployment
  containers:
    - image: kiamol/ch03-numbers-api
      readinessProbe:        # probes are set at the container level
        httpGet:
          path: /healthz    # this is an HTTP GET, using the health URL
          port: 80
        periodSeconds: 5    # the probe fires every five seconds

```

Kubernetes supports different types of container probe - this uses an HTTP GET action, which is perfect for web applications and APIs. The probe tells Kubernetes to test the `/healthz` endpoint every five seconds; if the response has an HTTP status code between 200 and 399 then the probe succeeds; any other status code and it will fail. The random number API returns a 500 code when it's unhealthy, so we can see the readiness probe in action.

TRY IT NOW Deploy the updated spec and verify that the Pod with the failed application is removed from the Service.

```
# deploy the updated spec from listing 12.1:  
kubectl apply -f numbers/update/api-with-readiness.yaml  
  
# wait for the replacement Pods:  
kubectl wait --for=condition=ContainersReady pod -l app=numbers-api,version=v2  
  
# check the endpoints:  
kubectl get endpoints numbers-api  
  
# trigger an application container to become unhealthy:  
curl "$(cat api-url.txt)/rng"  
  
# wait for the readiness probe to take effect:  
sleep 10  
  
# check the endpoints again:  
kubectl get endpoints numbers-api
```

You can see my output in figure 12.3 - the readiness probe detects one of the Pods is unhealthy because the response to the HTTP request returns 500. That Pod's IP address gets removed from the Service endpoint list, so it won't receive any more traffic.

The new Pod spec includes a readiness probe for the API container.

One application container is now unhealthy.

```

PS>kubectl apply -f numbers/update/api-with-readiness.yaml
deployment.apps/numbers-api configured
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=numbers-api,version=v2
pod/numbers-api-6c87c8fd69-rvw25 condition met
pod/numbers-api-6c87c8fd69-v7sq4 condition met
PS>
PS>kubectl get endpoints numbers-api
NAME      ENDPOINTS          AGE
numbers-api  10.1.1.196:80,10.1.1.197:80  14m
PS>
PS>curl "$(cat api-url.txt)/rng"
41
PS>
PS>sleep 10
PS>
PS>kubectl get endpoints numbers-api
NAME      ENDPOINTS          AGE
numbers-api  10.1.1.196:80  14m

```

The readiness probe will fire and one Pod will leave the ready state.
Now only the healthy Pod is registered as a Service endpoint.

Figure 12.3 Failing readiness probes move Pods out of the ready state so they're removed from Services

This app is also a good example of how readiness probes on their own can be dangerous. The logic in the random number API means once it has failed it will always fail, so the unhealthy Pod will stay excluded from the Service and the application will run below the expected capacity. Deployments do not replace Pods which leave the ready state when a probe fails, so we're left with two Pods running but only one receiving traffic. That situation gets much worse if the other Pod fails too.

TRY IT NOW Only one Pod is in the Service list. Make a request and that Pod goes unhealthy too, and both Pods are removed from the Service.

```

# check the Service endpoints:
kubectl get endpoints numbers-api

# call the API, triggering it to go unhealthy:
curl "$(cat api-url.txt)/rng"

# wait for the readiness probe to fire:
sleep 10

# check the endpoints again:
kubectl get endpoints numbers-api

# and the Pod status:

```

```
kubectl get pods -l app=numbers-api

# we could reset the API... but there are no Pods ready to
# receive traffic so this will fail:
curl "$(cat api-url.txt)/reset"
```

Now we're in a fix - both Pods have failed readiness probes and Kubernetes has removed them both from the Service endpoint list. That leaves the Service with no endpoints, so the app is offline - as you can see in figure 12.4. The situation now is that any clients trying to use the API will get a connection failure rather than an HTTP error status code, and that's true for administrators who try to reset the app using the special admin URL.

We have two Pods but only one is ready, so only one entry in the Service endpoints.

Now the second Pod fails its readiness probe and there are no endpoints in the Service.

```
PS>kubectl get endpoints numbers-api
NAME      ENDPOINTS      AGE
numbers-api  10.1.1.196:80  21m
PS>
PS>curl "$(cat api-url.txt)/rng"
98
PS>sleep 10
PS>
PS>kubectl get endpoints numbers-api
NAME      ENDPOINTS      AGE
numbers-api          22m
PS>
PS>kubectl get pods -l app=numbers-api
NAME                  READY   STATUS    RESTARTS   AGE
numbers-api-6c87c8fd69-rvw25  0/1     Running   0          8m31s
numbers-api-6c87c8fd69-v7sq4  0/1     Running   0          8m26s
PS>
PS>curl "$(cat api-url.txt)/reset"
curl: (52) Empty reply from server
```

The Pods are running, but the containers are not in the ready state because of the failing probes.

The API has a handy reset function to make it healthy again, but this request fails because there are no endpoints in the Service so no Pods to send it to.

Figure 12.4 Probes are supposed to help the app, but they can remove all Pods from a Service

If you're thinking "this isn't a self-healing app" - you're absolutely right, but remember that the application is in a failed state anyway. Without the readiness probe the app still doesn't work, but with the readiness probe it's protected from incoming requests until it recovers and is able to handle them. You need to understand the failure modes of your application to know what will happen when probes fail, and whether the app is likely to recover by itself.

The random number API never becomes healthy again but we can fix the failed state by restarting the Pod. Kubernetes will do that for you if you include another health check in the container spec: a *liveness probe*.

12.2 Restarting unhealthy Pods with liveness probes

Liveness probes use the same health check mechanism as readiness probes - the test configurations might be identical in your Pod spec - but the action for a failed probe is different. Liveness probes take action at the compute level, restarting Pods if they become unhealthy. A restart is when Kubernetes replaces the Pod container with a new one; the Pod itself isn't replaced; it continues to run on the same node but with a new container.

Listing 12.2 shows a liveness probe for the random number API. This probe uses the same HTTP GET action to run the probe, but it has some additional configuration. Restarting a Pod is more invasive than removing it from a Service, and the extra settings help to ensure that only happens when we really need it.

Listing 12.2 - api-with-readiness-and-liveness.yaml, adding a liveness probe

```
livenessProbe:
  httpGet:
    path: /healthz          # HTTP GET actions can be used in liveness and
    port: 80                # readiness probes - they use the same spec
    periodSeconds: 10
    initialDelaySeconds: 10 # wait 10 seconds before running the first probe
    failureThreshold: 2     # allow two probes to fail before taking action
```

This is a change to the Pod spec, so applying the update will create new replacement Pods that start off healthy. This time when a Pod becomes unhealthy after the application fails, it will be removed from the Service thanks to the readiness probe, then it will be restarted thanks to the liveness probe, and then the Pod will be added back into the Service.

TRY IT NOW Update the API and verify that liveness and readiness checks combined keep the application healthy.

```
# update the Pod spec from listing 12.2:
kubectl apply -f numbers/update/api-with-readiness-and-liveness.yaml

# wait for the new Pods:
kubectl wait --for=condition=ContainersReady pod -l app=numbers-api,version=v3

# check the Pod status:
kubectl get pods -l app=numbers-api -o wide

# and the Service endpoints:
kubectl get endpoints numbers-api # two

# cause one application to become unhealthy:
curl "$(cat api-url.txt)/rng"

# wait for the probes to fire and check the Pods again:
sleep 20
kubectl get pods -l app=numbers-api
```

In that exercise you'll see the liveness probe in action, restarting the Pod when the application fails. The restart is a new Pod container but the Pod environment is the same - so it has the same IP address, and if the container mounted an EmptyDir volume in the Pod it would have access to the files written by the previous container. You can see in figure 12.5 that both Pods are running and ready after the restart, so Kubernetes fixed the failure and healed the application.

The two new Pods are both running and ready; they have readiness and liveness checks configured.

The Service endpoints are the two Pod IP addresses.

```

PS>kubectl apply -f numbers/update/api-with-readiness-and-liveness.yaml
deployment.apps/numbers-api configured
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=numbers-api,version=v3
pod/numbers-api-75ff44ff4b-6gnnc condition met
pod/numbers-api-75ff44ff4b-lsgjf condition met
PS>
PS>kubectl get pods -l app=numbers-api -o wide
NAME           READY   STATUS    RESTARTS   AGE   IP          NODE
              NOMINATED NODE   READINESS GATES
numbers-api-75ff44ff4b-6gnnc  1/1     Running   0          15s  10.1.1.199  dock
er-desktop      <none>  <none>
numbers-api-75ff44ff4b-lsgjf  1/1     Running   0          22s  10.1.1.198  dock
er-desktop      <none>  <none>
PS>
PS>kubectl get endpoints numbers-api
NAME            ENDPOINTS          AGE
numbers-api     10.1.1.198:80,10.1.1.199:80  93m
PS>
PS>curl "$(cat api-url.txt)/rng"
65
PS>sleep 20
PS>
PS>kubectl get pods -l app=numbers-api -o wide
NAME           READY   STATUS    RESTARTS   AGE   IP          NODE
              NOMINATED NODE   READINESS GATES
numbers-api-75ff44ff4b-6gnnc  1/1     Running   0          77s  10.1.1.199  dock
er-desktop      <none>  <none>
numbers-api-75ff44ff4b-lsgjf  1/1     Running   1          84s  10.1.1.198  dock
er-desktop      <none>  <none>

```

Now one application container will become unhealthy - the Pod gets removed from the Service and then restarted.

Both Pods are running and ready again, but one has been restarted because of the failing liveness probe.

Figure 12.5 Readiness probes and liveness probes combined help keep applications online

Restarts aren't a permanent fix if the app keeps failing without a healthy streak, because Kubernetes won't indefinitely restart a failing Pod - but for transient issues it works well, provided the application can restart successfully in a replacement container. Probes are also

useful to keep applications healthy during upgrades, because rollouts only proceed as new Pods enter the ready state, so if a readiness probe fails that will pause the rollout.

We'll show that with the to-do list application, with specifications that include liveness and readiness checks for the web application Pod and the database. The web probes use the same HTTP GET action we've already seen, but the database doesn't have an HTTP endpoint we can use. Instead the spec uses the other types of probe action which Kubernetes supports - the TCP socket action, which checks that a port is open and listening for incoming traffic, and the exec action which runs a command inside the container. Listing 12.3 shows the probe setup.

listing 12.3 - todo-db.yaml, using TCP and command probes

```
spec:
  containers:
    - image: postgres:11.6-alpine
      # full spec includes environment config
      readinessProbe:
        tcpSocket:
          port: 5432
          # the readiness probe tests the
          # database is listening on the port
        periodSeconds: 5
      livenessProbe:
        # the liveness probe runs a Postgres tool
        exec:
          # which confirms the database is running
          command: ["pg_isready", "-h", "localhost"]
        periodSeconds: 10
      initialDelaySeconds: 10
```

When you deploy this you'll see the app works in the same way as always, but now it's protected against transient failures in both the web and database components.

TRY IT NOW Run the to-do list app with the new self-healing specification.

```
# deploy the web and database:
kubectl apply -f todo-list/db/ -f todo-list/web/

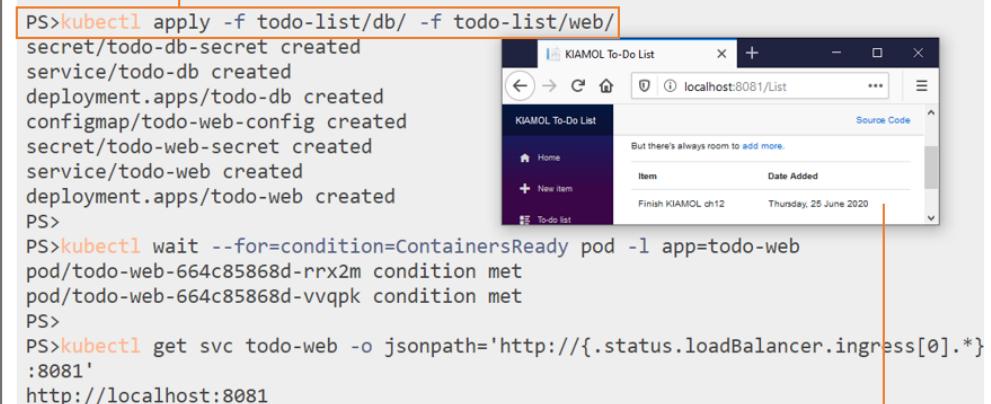
# wait for the app to be ready:
kubectl wait --for=condition=ContainersReady pod -l app=todo-web

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

# browse to the app and add a new item
```

Nothing new here - you can see my output in figure 12.6. But the database probes mean Postgres won't get any traffic until the database is ready, and if the Postgres server fails then the database Pod will be restarted, with the replacement using the same data files in the EmptyDir volume in the Pod.

The latest to-do app specification includes readiness and liveness probes. The database uses TCP and command probes as the container doesn't run an HTTP server.



```
PS>kubectl apply -f todo-list/db/ -f todo-list/web/
secret/todo-db-secret created
service/todo-db created
deployment.apps/todo-db created
configmap/todo-web-config created
secret/todo-web-secret created
service/todo-web created
deployment.apps/todo-web created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=todo-web
pod/todo-web-664c85868d-rrx2m condition met
pod/todo-web-664c85868d-vvqpk condition met
PS>
PS>kubectl get svc todo-web -o jsonpath='http://{.status.loadBalancer.ingress[0].*}:8081'
http://localhost:8081
```

The app works in the usual way which means all probes must be passing, and traffic is passing from the Services to the web Pods and the database Pod.

Figure 12.6 Probes are firing and returning healthy responses, so the app works in the usual way

Container probes can also keep an application running if an update goes wrong. There's a new database spec for the to-do app which upgrades the version of Postgres, but it also overrides the container command so it sleeps instead of starting Postgres. This is a classic left-over-from-debugging mistake, where someone wanted to start a Pod with the correct configuration but without running the app, so they could exec inside the container to check the environment, but they didn't revert their change. If the Pod didn't have any probes the update would succeed and take down the app - the sleep command keeps the Pod container running, but there's no database server for the website to use. The probes stop that happening and keep the app available.

TRY IT NOW Deploy the bad update and verify that the failing probes in the new Pod prevent the original Pod from being removed.

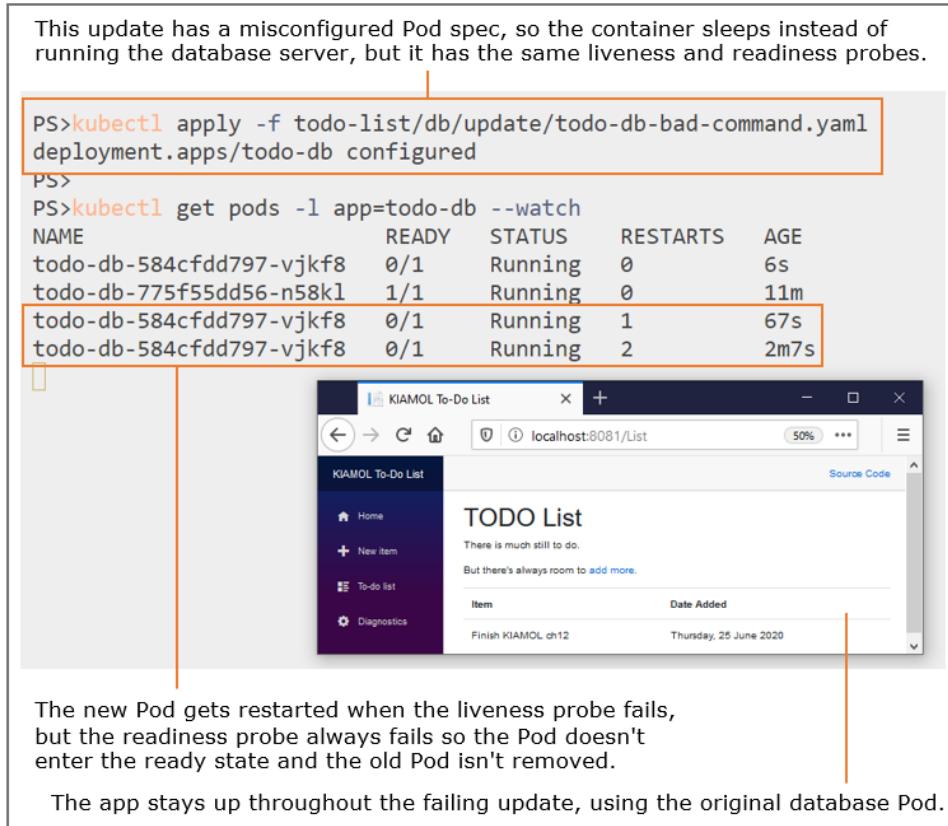
```
# apply the update:
kubectl apply -f todo-list/db/update/todo-db-bad-command.yaml

# watch the Pod status changing:
kubectl get pods -l app=todo-db --watch

# refresh the app to check it still works
# ctrl-c or cmd-c to exit the Kubectl watch
```

You can see my output in figure 12.7 - the replacement database Pod is created, but it never enters the ready state because the readiness probe checks port 5342 for a process

listening, and there isn't one. The Pod will keep restarting too because the liveness probe runs a command which checks Postgres is ready to receive client connections. While the new Pod keeps failing, the old one is left running and the app keeps working.



This update has a misconfigured Pod spec, so the container sleeps instead of running the database server, but it has the same liveness and readiness probes.

```
PS>kubectl apply -f todo-list/db/update/todo-db-bad-command.yaml
deployment.apps/todo-db configured
```

NAME	READY	STATUS	RESTARTS	AGE
todo-db-584cfdd797-vjkf8	0/1	Running	0	6s
todo-db-775f55dd56-n58k1	1/1	Running	0	11m
todo-db-584cfdd797-vjkf8	0/1	Running	1	67s
todo-db-584cfdd797-vjkf8	0/1	Running	2	2m7s

The new Pod gets restarted when the liveness probe fails, but the readiness probe always fails so the Pod doesn't enter the ready state and the old Pod isn't removed.

The app stays up throughout the failing update, using the original database Pod.

Figure 12.7 Rollouts wait for new Pods to become ready, so probes protect against failed updates

If you leave this app running for another five or so minutes and check the Pod status again, you'll see the new Pod goes into the CrashLoopBackOff status. This is how Kubernetes protects the cluster from wasting compute resources on applications which constantly fail - it adds a time delay between Pod restarts, and that delay increases with each restart. If you see a Pod in CrashLoopBackOff it usually means the app is beyond repair.

The to-do app is in the same situation now that we first saw in chapter 9 when rollouts fail. The Deployment is managing two ReplicaSets, and its goal is to scale the old one down to zero as soon as the new one is up to capacity. But the new ReplicaSet never reaches capacity - because the probes in the new Pod constantly fail. The Deployment stays like this, hoping it

can eventually finish the rollout. Kubernetes doesn't have an automatic rollback option, but Helm does and you can extend your Helm charts to support healthy upgrades.

12.3 Deploying upgrades safely with Helm

A little bit of Helm goes a long way. You learned the basics in chapter 10 and you don't need to dig too deeply into the templating functions and the dependency management to make good use of Helm for safe application upgrades. Helm supports atomic installs and upgrades, which automatically roll back if they fail, and it also has a deployment lifecycle you can hook into to run validation jobs before and after installation.

The source folder for this chapter has multiple Helm charts for the to-do app which represent different versions (normally that would be a single Helm chart which evolved with each release). The v1 chart deploys the app using the same liveness and readiness checks we used in section 12.2, the only difference is that the database uses a PersistentVolumeClaim, so data is preserved between upgrades. We'll start by clearing down the previous exercises and installing the Helm version.

TRY IT NOW Run the to-do app using the same Pod specs, but deployed using a Helm chart.

```
# remove all existing apps from the chapter:
kubectl delete all -l kiamol=ch12

# install the Helm release:
helm install --atomic todo-list todo-list/helm/v1/todo-list/

# browse to the app and add a new item
```

Version 1 of the app is now running through Helm, and there's nothing new here except that the chart contains a file called NOTES.txt in the templates folder which displays the helpful text you see after installation. My output is in figure 12.8 - I haven't included an application screenshot, so you'll just have to take my word that I browsed and added an item saying "finish chapter 12".

Installing a release using Helm's atomic option means it will roll back if the deployment fails - including failing container probes.

```

PS>kubectl delete all -l kiamol=ch12
service "numbers-api" deleted
service "todo-db" deleted
service "todo-web" deleted
deployment.apps "numbers-api" deleted
deployment.apps "todo-db" deleted
deployment.apps "todo-web" deleted
PS>
PS>helm install --atomic todo-list todo-list/helm/v1/todo-list/
NAME: todo-list
LAST DEPLOYED: Thu Jun 25 10:23:13 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Installed Kiamol to-do list 0.1.0. This is how to get the URL:
$ kubectl get svc todo-list-web -o jsonpath='http://{{.status.loadBalancer.ingress[0].*.}:8012}'
PS>
PS>kubectl get svc todo-list-web -o jsonpath='http://{{.status.loadBalancer.ingress[0].*.}:8012}'
http://localhost:8012

```

This chart writes out notes after successful deployment, providing the Kubectl command you need to run to browse to the app.

Figure 12.8 Installing apps with Helm waits for container probes to be healthy

Version 2 of the Helm chart attempts the same database image upgrade we saw in section 12.2, complete with the misconfiguration in the command for the Postgres container. When you deploy this with Helm the same thing happens under the hood - Kubernetes updates the Deployment which adds a new ReplicaSet, and that ReplicaSet never reaches capacity because the Pod readiness probe fails. But Helm checks the status of the rollout and if it doesn't succeed within a specific period, it automatically rolls back.

TRY IT NOW Upgrade the to-do app release using Helm - the upgrade fails because the Pod spec is misconfigured, and Helm rolls it back.

```

# list the current Pod status and container image:
kubectl get pods -l app=todo-list-db -o=custom-
  columns=NAME:.metadata.name,STATUS:.status.phase,IMAGE:.spec.containers[0].image

# upgrade the release with Helm - this will fail:
helm upgrade --atomic --timeout 30s todo-list todo-list/helm/v2/todo-list/

# list the Pods again:
kubectl get pods -l app=todo-list-db -o=custom-

```

```
columns=NAME:.metadata.name,STATUS:.status.phase,IMAGE:.spec.containers[0].image
# browse back to the app and refresh the list
```

If you check the Pod list a few times in that exercise you'll see the rollback happening - as you can see in figure 12.9. At first there's a single Pod running Postgres 11.6, then it's joined by a new Pod running 11.8 but that's the Pod with the failing container probes. The Pod isn't ready within the Helm timeout period so the upgrade is rolled back and the new Pod is removed; it doesn't keep restarting and hit CrashLoopBackOff as it did with the Kubectl update.

The v1 Helm release uses Postgres 11.6.

The Helm upgrade is rolled back because the new Pod fails its probes and doesn't become ready within the specified 30 second timeout period.

```
PS>kubectl get pods -l app=todo-list-db -o=custom-columns=NAME:.metadata.name,STATUS:.status.phase,IMAGE:.spec.containers[0].image
NAME STATUS IMAGE
todo-list-db-58596944f9-jz4t8 Running postgres:11.6-alpine
PS>
PS>helm upgrade --atomic --timeout 30s todo-list todo-list/helm/v2/todo-list/
Error: UPGRADE FAILED: release todo-list failed, and has been rolled back due to atomic being set: timed out waiting for the condition
PS>
PS>kubectl get pods -l app=todo-list-db -o=custom-columns=NAME:.metadata.name,STATUS:.status.phase,IMAGE:.spec.containers[0].image
NAME STATUS IMAGE
todo-list-db-58596944f9-jz4t8 Running postgres:11.6-alpine
todo-list-db-745c86758-6qwjb Running postgres:11.8-alpine
PS>
PS>kubectl get pods -l app=todo-list-db -o=custom-columns=NAME:.metadata.name,STATUS:.status.phase,IMAGE:.spec.containers[0].image
NAME STATUS IMAGE
todo-list-db-58596944f9-jz4t8 Running postgres:11.6-alpine
PS>
```

There's a new Pod running Postgres 11.8, but this is the failed upgrade, and Helm has scheduled it to be removed.

Only the original Pod is still running. The upgrade failed gracefully and the app stayed up.

Figure 12.9 The upgrade fails because the new Pod doesn't become ready, and Helm rolls back

The to-do app has been online without interruption or reduced capacity during the failed upgrade to version 2. The next version fixes the upgrade by removing the bad container command in the Pod spec, and it also adds an extra template for a Kubernetes Job which you can run as a deployment test with Helm. Tests run on-demand and not as part of the install, so they're perfect for smoke tests - automated test suites which you run to confirm that a successful release is working correctly. Listing 12.4 shows a test for the to-do database.

Listing 12.4 - todo-db-test-job.yaml, a Kubernetes Job to run as a Helm test

```
apiVersion: batch/v1
```

```

kind: Job                      # this is a standard Job spec
metadata:
  # metadata includes name and labels
  annotations:
    "helm.sh/hook": test      # this annotation tells Helm the Job can
spec:                           # be run in the test suite for the release
  completions: 1
  backoffLimit: 0              # the Job should run once and not retry
  template:
    spec:                     # container spec runs a SQL query
      containers:
        - image: postgres:11.8-alpine
          command: ["psql", "-c", "SELECT COUNT(*) FROM \"public\".\"Todos\""]

```

We met Jobs in chapter 8, and Helm makes good use of them. Job specs include an expectation of how many times they should run to successful completion, and Helm uses that to evaluate if the test succeeds. The v3 upgrade should succeed, and when it completes you can run the test job which runs a SQL statement to confirm the to-do database is accessible.

TRY IT NOW Upgrade to the v3 chart which fixes the Postgres update. Then run the test with Helm and check the logs for the Job Pod.

```

# run the upgrade:
helm upgrade --atomic --timeout 30s todo-list todo-list/helm/v3/todo-list/

# list the database Pods and images:
kubectl get pods -l app=todo-list-db -o=custom-
  columns=NAME:.metadata.name,STATUS:.status.phase,IMAGE:.spec.containers[0].image,IP:
    .status.podIPs[].ip

# check the database Service endpoints:
kubectl get endpoints todo-list-db

# now run the test Job with Helm:
helm test todo-list

# and check the output:
kubectl logs -l job-name=todo-list-db-test

```

I've snipped my output in figure 12.10, but the detail is all there - the upgrade is successful, but there are no tests as part of the upgrade command. The database is now using the upgraded version of Postgres, and when the test runs the Job connects to the database and confirms the data is still there.

The v3 chart has fixed the Postgres spec, so now the upgrade is successful.

```

PS>helm upgrade --atomic --timeout 30s todo-list todo-list/helm/v3/todo-list/
Release "todo-list" has been upgraded. Happy Helming!
PS>kubectl get pods -l app=todo-list-db -o=custom-columns=NAME:.metadata.name,STATUS:.status.phase,IMAGE:.spec.containers[0].image,IP:.status.podIPs[].ip
NAME STATUS IMAGE IP
todo-list-db-968ffff78f-592nt Running postgres:11.8-alpine 10.1.1.217
PS>
PS>kubectl get endpoints todo-list-db
NAME ENDPOINTS AGE
todo-list-db 10.1.1.217:5432 34m
PS>
PS>helm test todo-list
NAME: todo-list
LAST DEPLOYED: Thu Jun 25 10:55:01 2020
NAMESPACE: default
STATUS: deployed
REVISION: 4
TEST SUITE: todo-list-db-test
Last Started: Thu Jun 25 10:58:10 2020
Last Completed: Thu Jun 25 10:58:12 2020
Phase: Succeeded
PS>kubectl logs -l job-name=todo-list-db-test
count
-----
1

```

The new database Pod is running the upgraded Postgres version, and its probes are successful so it's listed as the Service endpoint.

The Helm test suite passes - the Job is not deleted so the logs are available.

Figure 12.10 Running test suites on-demand with Helm lets you smoke test your app at any time

Helm manages Jobs for you - it doesn't clean up completed Jobs, so you can check the Pod status and logs if you need to - but it replaces them when you repeat the test command, so you can re-run the test suite as often as you like. There's one other use for Jobs which helps to make sure upgrades are safe, by running them before upgrades so you can check the current release is in a valid state to be upgraded.

This is especially useful if you support multiple versions of your app but only with incremental upgrades, so v1.1 needs to upgrade to v1.2 before it can upgrade to v2. The logic for that might involve querying the API version for different services or the schema version of a database, and Helm can run it all in a Job which has access to all the other Kubernetes objects - sharing the same ConfigMaps and Secrets as the application Pods. Listing 12.5 shows a pre-upgrade test in v4 of the to-do Helm chart.

Listing 12.5 - todo-db-check-job.yaml, a Job which runs before a Helm upgrade

```

apiVersion: batch/v1
kind: Job # standard Job spec again

```

```

metadata:
  # metadata has name and labels
  annotations:
    "helm.sh/hook": pre-upgrade      # this runs before an upgrade and
    "helm.sh/hook-weight": "10"        # tells Helm the order to create
spec:
  template:
    spec:
      restartPolicy: Never
      containers:
        - image: postgres:11.8-alpine
          # env includes secrets
          command: ["/scripts/check-postgres-version.sh"]
      volumeMounts:
        - name: scripts           # mounts the configmap volume
          mountPath: "/scripts"

```

There are two templates for the pre-upgrade check: one is the Job spec and the other is a ConfigMap which contains the script to run in the Job. You use annotations to control where Jobs need to run in the Helm lifecycle, and this Job will only run for upgrades, not as part of a new install. The weighting annotations make sure the ConfigMap is created before the Job. Lifecycles and weights let you model complex validation steps in Helm, but this one is simple - it upgrades the database image, but only if the release is currently running 11.6.

TRY IT NOW The upgrade from v3 to v4 isn't valid because v3 has already upgraded the Postgres version.
Run the upgrade to verify that it doesn't get deployed.

```

# run the upgrade to v4 - this will fail:
helm upgrade --atomic --timeout 30s todo-list todo-list/helm/v4/todo-list/

# list the Jobs:
kubectl get jobs --show-labels

# print the output of the pre-upgrade Job:
kubectl logs -l job-name=todo-list-db-check

# confirm that the database Pod is unchanged:
kubectl get pods -l app=todo-list-db -o=custom-
  columns=NAME:.metadata.name,STATUS:.status.phase,IMAGE:.spec.containers[0].image

```

In this exercise you'll see that Helm effectively blocks the upgrade, because the pre-upgrade hook runs and the Job fails. That's all recorded in the history for the release, which will show that the latest upgrade failed and the release was rolled back to the last good revision. My output is in figure 12.11, and throughout this update the app was still available.

This upgrade fails because the pre-upgrade Job didn't complete successfully. Helm doesn't show the output of the Job logs, but the Jobs are retained.

```
PS> helm upgrade --atomic --timeout 30s todo-list todo-list/helm/v4/todo-list/
Error: UPGRADE FAILED: release todo-list failed, and has been rolled back due to atomic being set: pre-upgrade hooks failed: job failed: BackoffLimitExceeded
PS>
PS> kubectl get jobs --show-labels
NAME           COMPLETIONS   DURATION   AGE   LABELS
todo-list-db-check  0/1        54s        54s   kiamol=ch12
todo-list-db-test  1/1        2s         30m   kiamol=ch12
PS>
PS> kubectl logs -l job-name=todo-list-db-check
** ERROR - Postgres not at expected version - wanted: 11.6, got: PostgreSQL 11.8 - CANNOT UPGRADE **
PS>
PS> kubectl get pods -l app=todo-list-db -o=custom-columns=NAME:.metadata.name,STATUS:.status.phase,IMAGE:.spec.containers[0].image
NAME           STATUS   IMAGE
todo-list-db-968ffff78f-592nt  Running  postgres:11.8-alpine
```

The logs from the failed check Job show the validation logic - this upgrade path is not supported.

The v3 database Pod is still running, and the app is still working correctly.

Figure 12.11 Pre-upgrade Jobs in Helm charts let you validate the release is suitable to upgrade

It's good to understand what Helm brings in terms of keeping your applications healthy, because pre-upgrade validation and automatic rollbacks help to keep your application upgrades self-healing too. Helm isn't a pre-requisite for that, but if you're not using Helm you should consider implementing these features using Kubectl in your deployment pipeline.

There's one more part of application health which we'll cover in this chapter - managing the compute resources available to your Pod containers.

12.4 Protecting apps and nodes with resource limits

Containers are a virtualized environment for your application process - Kubernetes builds that environment, and you know that Kubernetes creates the container filesystem and sets up networking. The container environment also includes memory and CPU, and those can be managed by Kubernetes too, but by default they're not. That means Pod containers get access to all the memory and CPU on the node where they're running. That's bad for two reasons: apps can max out on memory and crash, or they can starve the node of resources so other apps can't run.

You can limit the resources available to a container in the Pod spec, and those limits are like container probes - you really shouldn't go to production without them. Apps with memory leaks can ruin your cluster very quickly, and causing a CPU spike is a nice easy attack vector for the bad people. In this section you'll learn how to spec your Pods to prevent that, and we'll start with a new app which has a large appetite for memory.

TRY IT NOW Clear down from the last exercise and run the new app - it doesn't do anything other than allocate memory and log how much it has allocated. This Pod runs without any container limits.

```
# remove the Helm release to free up resources:
helm uninstall todo-list

# print how much memory your nodes have:
kubectl get nodes -o jsonpath='{.items[].status.allocatable.memory}'

# deploy the memory allocating app:
kubectl apply -f memory-allocator/

# wait a few minutes then see how much memory it has allocated:
kubectl logs -l app=memory-allocator --tail 1
```

The memory allocator app grabs about 10MB of memory every 5 seconds, and it will keep going until it uses all the memory in your lab cluster. You can see from my output in figure 12.12 that my Docker Desktop node has access to about 25GB of memory, and the allocator app had grabbed almost 1.5GB when I took the screenshot.

Kubectl can query nodes for information in the same way as other objects - my cluster has a single node with about 25GB memory available for Kubernetes.

```
PS>helm uninstall todo-list
release "todo-list" uninstalled
PS>
PS>kubectl get nodes -o jsonpath='{.items[].status.allocatable.memory}'
26109304Ki
PS>
PS>kubectl apply -f memory-allocator/
deployment.apps/memory-allocator created
PS>
PS># wait a few minutes
PS>
PS>kubectl logs -l app=memory-allocator --tail 1
Allocated ~1450MiB
```

The app in this Pod keeps allocating memory and never releases it.

After a while the container has allocated almost 1.5GB of memory, and it will keep going.

Figure 12.12 Don't run this app in production - it just keeps allocating memory until it has it all

As long as the app is running it will keep allocating memory, so we need to get on quickly before my machine dies and I lose the edits to this chapter. Listing 12.6 shows an updated Pod spec which includes resource limits, restricting the app to 50MB of memory.

Listing 12.6 - memory-allocator-with-limit.yaml, adding memory limits to the container

```
spec: # the Pod spec in the Deployment
```

```

containers:
  - image: kiamol/ch12-memory-allocator
    resources:
      limits:           # resource limits constrain the compute power
      memory: 50Mi     # for the container; this limits RAM to 50MB

```

Resources are specified at the container level, but this is a new Pod spec so when you deploy the update you'll get a new Pod. The replacement will start off with zero memory allocated and it will start allocating 10MB every five seconds again - but now it will hit a limit at 50MB and Kubernetes will take action.

TRY IT NOW Deploy an update to the memory allocator app with the resource limits defined in listing 12.6

```

# apply the update:
kubectl apply -f memory-allocator/update/memory-allocator-with-limit.yaml

# wait for the app to allocate a chunk of memory:
sleep 20

# print the application logs:
kubectl logs -l app=memory-allocator --tail 1

# and watch the status of the Pod:
kubectl get pods -l app=memory-allocator --watch

```

In this exercise you'll see that Kubernetes enforces the memory limit - when the app tries to allocate more than 50MB the container is replaced, and you can see the Pod enters the OOMKilled status. Exceeding the limit causes a Pod restart, so this has the same drawback as a failing liveness probe - if the replacement containers keep failing, the Pod restarts will take longer and longer as Kubernetes applies the crash-loop backoff, as you see in figure 12.13.

This update adds a memory resource limit to the Pod container.

```

PS>kubectl apply -f memory-allocator/update/memory-allocator-with-limit.yaml
deployment.apps/memory-allocator configured
PS>
PS>sleep 20
PS>
PS>kubectl logs -l app=memory-allocator --tail 1
Allocated ~30MiB
PS>
PS>kubectl get pods -l app=memory-allocator --watch
NAME          READY   STATUS    RESTARTS   AGE
memory-allocator-7d85c77dc-hpqmg  1/1     Running   2          56s
memory-allocator-7d85c77dc-hpqmg  0/1     OOMKilled  2          66s
memory-allocator-7d85c77dc-hpqmg  0/1     CrashLoopBackOff  2          80s
memory-allocator-7d85c77dc-hpqmg  1/1     Running   3          93s

```

The container has allocated 30MB, so it's near to the 50MB limit.

Watching the Pod status change, you can see that Kubernetes kills the container when it exceeds the memory limit - the Pod restarts and flips between Running and CrashLoopBackoff status.

Figure 12.13 Memory limits are hard limits - if a container exceeds them it gets killed and the Pod restarts

(If you run this exercise in Kind you may not see the Pod being restarted - memory limits in Kind only get applied if your Linux host is running without swap memory enabled.

The hard part of applying resource constraints is working out what the limits should be. You'll need to factor in some performance testing to see what your app can manage with, and be aware that some application platforms will grab more than they need if they see lots of available memory. You should be generous with your initial releases, then look to bring the limits down as you get more feedback from your monitoring.

You can apply resource limits in another way too, by specifying maximum quotas for a namespace. This is especially useful for shared clusters where you use namespaces to divide the cluster for different teams or environments; you can enforce limits on the total amount of resources the namespace can use. Listing 12.7 shows the spec for a ResourceQuota object, which restricts the total amount of memory available to 150MB in the namespace called kiamol-ch12-memor.

Listing 12.7 - 02-memory-quota.yaml, setting memory quota for a namespace

```

apiVersion: v1
kind: ResourceQuota
metadata:                                     # the ResourceQuota is applied
  name: memory-quota                         # at the specified namespace
  namespace: kiamol-ch12-memory
spec:                                         # quotas can include CPU and memory
  hard:

```

```
limits.memory: 150Mi
```

Container limits are reactive, so Pods are restarted when the memory limit is exceeded. Resource quotas are proactive, so Pods won't be created if the limits they specify exceed what's available in the quota. If there's a quota in place then every Pod spec needs to include a resource section so Kubernetes can compare what the spec needs to what's currently available in the namespace. There's an updated version of the memory allocator spec to demonstrate that, where the Pod specifies a limit which is greater than the quota.

TRY IT NOW Deploy a new version of the memory allocator in its own namespace with a resource quota applied.

```
# delete the existing app:  
kubectl delete deploy memory-allocator  
  
# deploy namespace, quota and new Deployment:  
kubectl apply -f memory-allocator/namespace-with-quota/  
  
# print the status of the ReplicaSet:  
kubectl get replicaset -n kiamol-ch12-memory  
  
# and show the events in the ReplicaSet:  
kubectl describe replicaset -n kiamol-ch12-memory
```

You'll see from the output of the ReplicaSet that it has zero Pods out of a desired total of one. It can't create the Pod because it would exceed the quota for the namespace, as you can see in figure 12.4. The controller keeps trying to create the Pod, but it won't succeed unless enough quota becomes available - that could be from other Pods terminating, but in this case there aren't any so it would need to be an update to the quota.

Deploys a quota in the namespace with 150MB memory limit, and a Pod spec which sets a 200MB memory limit.

The ReplicaSet cannot create the Pod because there is not enough memory quota in the namespace.

```

PS>kubectl delete deploy memory-allocator
deployment.apps "memory-allocator" deleted
PS>
PS>kubectl apply -f memory-allocator/namespace-with-quota/
namespace/kiamol-ch12-memory created
resourcequota/memory-quota created
deployment.apps/memory-allocator created
PS>
PS>kubectl get replicaset -n kiamol-ch12-memory
NAME           DESIRED   CURRENT   READY   AGE
memory-allocator-579cb6f6d7   1         0         0       6s
PS>
PS>kubectl describe replicaset -n kiamol-ch12-memory
Name:           memory-allocator-579cb6f6d7
Namespace:      kiamol-ch12-memory
Conditions:
  Type     Status  Reason
  ----     -----  -----
  ReplicaFailure  True    FailedCreate
Events:
  Type     Reason     Age           From               Message
  ----     -----     --           --               -----
  Warning  FailedCreate  19s          replicaset-controller  Error creating: pods "memory-allocator-579cb6f6d7-782qs" is forbidden: exceeded quota: memory-quota, requested: limits.memory=200Mi, used: limits.memory=0, limited: limits.memory=150M

```

The events in the ReplicaSet show this is a quota issue, but the controller will keep trying to create the Pod.

Figure 12.14 Quotas with hard limits prevent Pods being created if they would exceed the quota

Kubernetes can also apply CPU limits to containers and quotas, but they work in a slightly different way. Containers with a CPU limit run with a fixed amount of processing power and they can use as much of that CPU as they like - they aren't replaced if they hit the limit. You can limit a container to one half of a CPU core, and it can run at 100% CPU while all the other cores on the node remain idle and available for other containers. Calculating Pi is a compute-intensive operation, and we can see the effect of applying a CPU limit on the Pi application we've used before in the book.

TRY IT NOW Run the Pi application with and without CPU limits and compare its performance.

```

# show the total CPU available to the nodes:
kubectl get nodes -o jsonpath='{.items[].status.allocatable.cpu}'

# deploy Pi without any CPU limits:
kubectl apply -f pi/

```

```
# get the URL for the app:  
kubectl get svc pi-web -o  
    jsonpath='http://{.status.loadBalancer.ingress[0].*}:8012/?dp=50000'  
  
# browse to the URL and see how long the calculation takes  
  
# now update the Pod spec with a CPU limit:  
kubectl apply -f pi/update/web-with-cpu-limit.yaml  
  
# refresh the Pi app and see how long the calculation takes
```

My output is in figure 12.15 - your timings will be different, depending on how much CPU is available on your node. Mine has 8 cores and with no limits the app calculates Pi to 50,000 decimal places consistently within 3.4 seconds. After the update the app container is limited to one quarter of one core, and the same calculation takes 14.4 seconds.

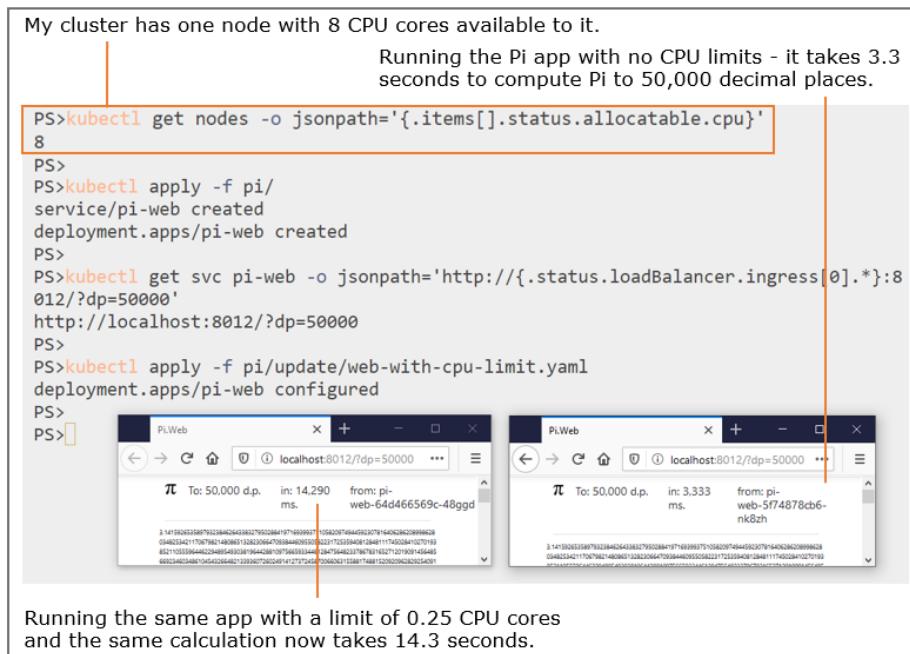


Figure 12.15 Squint and you'll see that limiting CPU has an impact on calculation speed

Kubernetes defines CPU limits using a fixed unit, where 1 represents a single core. You can use multiples to give your app container access to many cores, or divide a single core into "millicores", where 1 millicore is one-thousandth of a core. Listing 12.8 shows the CPU limit applied to the Pi container in the last exercise, where 250 millicores is one quarter of one core.

Listing 12.8 - web-with-cpu-limit.yaml

```

spec:
  containers:
    - image: kiamol/ch05-pi
      command: ["dotnet", "Pi.Web.dll", "-m", "web"]
      resources:
        limits:
          cpu: 250m    # 250 millicores limits the container to 0.25 cores

```

I'm focusing on one resource at a time so you can clearly see the impact, but typically you should include both CPU and memory limits so your apps don't surge and starve the cluster. Resource specs can also include a *requests* section which state how much CPU and memory the container is expected to use - that helps Kubernetes decide which node should run the Pod, and we'll cover it more when we get to scheduling in chapter 18.

We'll finish this chapter with one more exercise to show how CPU limits can be applied to a quota for a namespace, and what it means when the quota is exceeded. The new spec for the Pi application tries to run two replicas with 300 millicore CPU limits in a namespace which has a quota with a maximum of 500 millicores.

TRY IT NOW Run an updated Pi application in its own namespace, which has a CPU quota applied.

```

# remove the existing app:
kubectl delete deploy pi-web

# deploy namespace, quota and the new app spec:
kubectl apply -f pi/namespace-with-quota/

# print the ReplicaSet status:
kubectl get replicaset -n kiamol-ch12-cpu

# list the endpoints for the Service:
kubectl get endpoints pi-web -n kiamol-ch12-cpu

# and show the events for the ReplicaSet:
kubectl describe replicaset -n kiamol-ch12-cpu

```

In that exercise you can see that quotas apply across all Pods in the namespace. The ReplicaSet is running with one Pod instead of two, because the first Pod allocated 300m CPU, which only left 200m in the quota - not enough for the second Pod to run. Figure 12.16 shows the failure reason in the events for the ReplicaSet. The Pi app is still running, but under capacity because there isn't enough CPU available.

Deploys the Pi app with two replicas, each set with a 300m CPU limit, into a namespace set with a 500m CPU quota.

```

PS> kubectl delete deploy pi-web
deployment.apps "pi-web" deleted
PS>
PS> kubectl apply -f pi/namespace-with-quota/
namespace/kiamol-ch12-cpu created
resourcequota/cpu-quota created
service/pi-web created
deployment.apps/pi-web created
PS>
PS> kubectl get replicaset -n kiamol-ch12-cpu
NAME          DESIRED   CURRENT   READY   AGE
pi-web-5fc85c7d69   2         1         1     8s
PS>
PS> kubectl get endpoints pi-web -n kiamol-ch12-cpu
NAME      ENDPOINTS   AGE
pi-web    10.1.1.225:80   16s
PS>
PS> kubectl describe replicaset -n kiamol-ch12-cpu
Name:           pi-web-5fc85c7d69
Namespace:      kiamol-ch12-cpu

Conditions:
  Type        Status  Reason
  ----        ----   -----
  ReplicaFailure  True    FailedCreate
Events:
  Type      Reason     Age           From            Message
  ----      ----     --           --             -----
  Normal   SuccessfulCreate  39s          replicaset-controller  Created pod:
pi-web-5fc85c7d69-7kdss
  Warning  FailedCreate   39s          replicaset-controller  Error creating
pod: pods "pi-web-5fc85c7d69-gtcfm" is forbidden: exceeded quota: cpu-quota, requested
limits.cpu=300m, used: limits.cpu=300m, limited: limits.cpu=500m

```

The ReplicaSet only creates one Pod.

The Service is using the Pod.

The controller can't create the second Pod because there's not enough CPU quota remaining.

Figure 12.16 Hard CPU limits in quotas are enforced to block objects exceeding the total limit

Quotas are more for protecting your cluster than the apps themselves, but they're a good way of enforcing that all Pod specs have limits specified. If you're not dividing up your cluster with namespaces you can still apply a quota with large CPU and memory limits to the default namespace to make sure Pod specs include limits of their own.

Resource limits, container probes and atomic upgrades all help to keep your apps running in the face of normal failure conditions. These should be on your roadmap to production, but you also need to be aware that Kubernetes can't repair every kind of failure.

12.5 Understanding the limits of self-healing apps

Kubernetes allocates a Pod to a node, and that's the node where it will run. Pods aren't replaced unless the node goes offline, so all the repair mechanisms we've seen in this chapter work by restarting the Pod - replacing the application container. You need to make sure your app can tolerate that, especially in the multi-container scenarios we covered in chapter 7, because init containers are executed again and sidecars are replaced when a Pod restarts.

Pod restarts are fine for most scenarios with temporary failures, but repeated failures will end up in a crash-loop backoff state which can take your app offline. Kubernetes doesn't provide any configuration options for how many restarts are allowed or the backoff period, and it doesn't support replacing failed Pods with a new Pod on a different node. Those features are requested but until they land, your nicely configured self-healing app still has the potential for all its Pods to be in a backoff state with no endpoints in the Service.

That edge case usually appears as a result of misconfigured specs or fatal problems with the application, which take more intervention than Kubernetes can manage by itself. For the typical failure states the combination of container probes and resource limits go a long way to keeping your app running smoothly all by itself.

And that's all for self-healing apps, so we can tidy up the cluster in preparation for the lab.

TRY IT NOW Remove the objects from this chapter.

```
# delete namespaces:  
kubectl delete ns -l kiamol=ch12  
kubectl delete all -l kiamol=ch12  
  
# delete all the leftover objects:  
kubectl delete secret,configmap,pvc -l kiamol=ch12
```

12.6 Lab

I've got a nice little capacity planning exercise for you in this lab. The goal is to divide your cluster into three environments to run the Pi app - dev, test and UAT. UAT should be limited to 50% of your node's total CPU, and dev and test to 25% each. Your Pi Deployment should be set with limits so it can run at least four replicas in every environment, and then you need to verify how much you can scale up to in UAT.

- Start by deploying the namespaces and Services in the lab folder
- Then work out the CPU capacity of your node and deploy resource quotas to limit CPU in each namespace (you'll need to write the quota specs)
- Update the Deployment spec in web.yaml to include a CPU limit which allows four replicas to run in each namespace
- When everything's running, scale up the UAT Deployment to 8 replicas, and try to find out why they don't all run.

This is a good exercise to help you understand how CPU resources get shared and to practice working with multiple namespaces. My solution is up on GitHub for you to check:

<https://github.com/sixeyed/kiamol/blob/master/ch12/lab/README.md>

13

Centralizing logs with Fluentd and Elasticsearch

Applications generate lots of logs which often aren't very useful. As you scale up your apps across multiple Pods running in a cluster it's very difficult to manage those logs using standard Kubernetes tooling. Organizations usually deploy their own logging framework which uses a collect-and-forward model to read container logs and send them to a central store where they can be indexed, filtered and searched. You'll learn how to do that in this chapter using the most popular technologies in this space - Fluentd and Elasticsearch. Fluentd is the collector component and it has some very nice integrations with Kubernetes; Elasticsearch is the storage component and it can run as Pods in the cluster or as an external service.

There are a couple of points to be aware of before we start. The first is that this model assumes your application logs are written to the container's standard output streams so Kubernetes can find them. We covered that in chapter 7, with sample apps that wrote to standard out directly or used a logging sidecar to relay logs. The second is that the logging model in Kubernetes is very different from Docker - so if you've read *Learn Docker in a Month of Lunches*, this chapter takes a different approach.

13.1 How Kubernetes stores log entries

Kubernetes has a very simplistic approach to log management - it collects log entries from the container runtime and stores them as files on the node running the container. If you want to do anything more advanced then you need to deploy your own log management system, and fortunately you have a world-class container platform to run it on. The moving pieces of the logging system collect logs from the nodes, forward them to a centralized store and provide a UI to search and filter them. Figure 13.1 shows the technologies we'll use in this chapter.

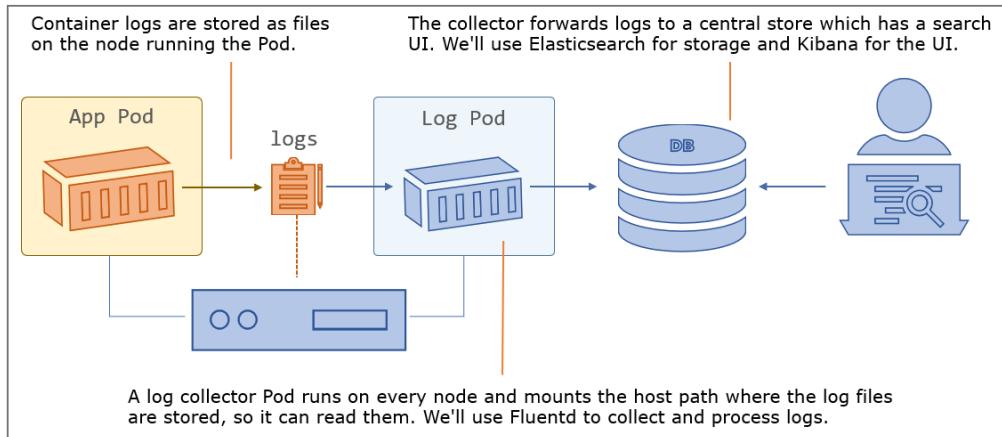


Figure 13.1 Logging in Kubernetes uses a collector like Fluentd to read the log files from the node

Nodes store log entries exactly as they come from the container, using filenames which include the namespace, pod and container names. The standard naming system makes it easy for the log collector to add metadata to the log entries to identify the source - and because the collector runs as a Pod itself, it can query the Kubernetes API server to get even more details. Fluentd adds Pod labels and the image tag as additional metadata, which you can use to filter or search the logs.

It's straightforward to deploy the log collector, but we'll start by exploring the raw log files on the node to see what we're working with. The pre-requisite for any of this is to get application logs out of the container, whether the app writes those logs directly or you use a sidecar container. Start by deploying the timecheck app from chapter 7 in a couple of different configurations to generate some logs.

TRY IT NOW Run the timecheck app using different setups in different namespaces, then check the logs to see how you work with them natively in Kubectl.

```
# switch to the chapter's folder:
cd ch13

# deploy the timecheck app in dev and test namespaces:
kubectl apply -f timecheck/

# wait for dev to spin up:
kubectl wait --for=condition=ContainersReady pod -l app=timecheck -n kiamol-ch13-dev

# and check the logs:
kubectl logs -l app=timecheck --all-containers -n kiamol-ch13-dev --tail 1

# wait for test to spin up:
kubectl wait --for=condition=ContainersReady pod -l app=timecheck -n kiamol-ch13-test

# check those logs:
```

```
kubectl logs -l app=timecheck --all-containers -n kiamol-ch13-test --tail 1
```

You'll see from that exercise that in a realistic cluster environment it's hard to work with container logs directly - my output is in figure 13.2. You have to use one namespace at a time, you can't identify the Pod which logged the message and you can only filter by a number of log entries or a time period.

This runs the timecheck app in two namespaces, using two Pods in the test namespace. That will produce a lot of log entries.

To read the dev logs you need to specify a label selector and a namespace.

```
PS>cd ch13
PS>
PS>kubectl apply -f timecheck/
namespace/kiamol-ch13-dev created
deployment.apps/timecheck created
namespace/kiamol-ch13-test created
configmap/timecheck-config created
deployment.apps/timecheck created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=timecheck -n kiamol-ch13-dev
pod/timecheck-555555bfd-vjzhz condition met
PS>
PS>kubectl logs -l app=timecheck --all-containers -n kiamol-ch13-dev --tail 1
2020-06-29 20:06:17.779 +00:00 [INF] Environment: DEV; version: 1.0; time check: 20:06.17
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=timecheck -n kiamol-ch13-test
pod/timecheck-cf4d4f685-fz5m7 condition met
pod/timecheck-cf4d4f685-rwq4c condition met
PS>
PS>kubectl logs -l app=timecheck --all-containers -n kiamol-ch13-test --tail 1
2020-06-29 20:06:42.821 +00:00 [INF] Environment: TEST; version: 1.1; time check: 20:06.42
2020-06-29 20:06:33.996 +00:00 [INF] Environment: TEST; version: 1.1; time check: 20:06.33
```

In test you can see the latest log from each Pod, but there's no way to identify which Pod wrote the log. Kubectl 1.18 address that with a prefix option, but without that it's hard to correlate the output with the app.

Figure 13.2 Kubectl is great for quickly checking logs, but it's harder with many Pods in many namespaces

Kubectl is the simplest option for reading logs, but ultimately the log entries come from the files on each node which means you have other options to work with logs. The source for this chapter includes a simple sleep Deployment which mounts the log path on the node as a HostPath volume, and you can use that to explore the log files even if you don't have direct access to the nodes.

TRY IT NOW Run a Pod with a volume mount for the host's log directory and explore the files using the mount.

```
# run the Deployment:  
kubectl apply -f sleep.yaml  
  
# connect to a session in the Pod container:  
kubectl exec -it deploy/sleep -- sh  
  
# browse to the host log mount:  
cd /var/log/containers/  
  
# list the timecheck log files:  
ls timecheck*kiamol-ch13*_logger*  
  
# and view the contents of the dev log file:  
cat $(ls timecheck*kiamol-ch13-dev_logger*) | tail -n 1  
  
# exit from the session:  
exit
```

Each Pod container has a file for its log output. The timecheck app uses a sidecar container called `logger` to relay the logs from the application container, and you can see in figure 13.3 the standard naming convention Kubernetes uses for log files: `pod-name_namespace_container-name-container-id.log`. The filename has enough data to identify the source of the logs, and the content of the file is the raw JSON log output from the container runtime.

This Pod spec has a volume mount to the log directory on the host node where all the container log files are stored.

The log file names use a standard format including the Pod name, namespace, container name and container ID.

```

PS>kubectl apply -f sleep.yaml
deployment.apps/sleep created
PS>
PS>kubectl exec -it deploy/sleep -- sh
/ #
/ # cd /var/log/containers/
/var/log/containers #
/var/log/containers # ls timecheck*kiamol-ch13* logger*
timecheck-555555bdf-vjzhz_kiamol-ch13-dev_logger-0758af957fff1a7b102cfde7956964ac30
d8e636ecb33a13e6b222e59eeacf74f.log
timecheck-cf4d4f685-f25m7_kiamol-ch13-test_logger-c701156604fbbfb0ae3695b0d5e4763d1
1cd8ee8612cdbc8308585960e1d39da.log
timecheck-cf4d4f685-rwq4c_kiamol-ch13-test_logger-d71b15d0d35bdc1ce0e0625d477634c32
e0acaea9cb3231dbf88978aa1b2ff86.log
/var/log/containers #
/var/log/containers # cat $(ls timecheck*kiamol-ch13-dev_logger*) | tail -n 1
{"log": "2020-06-29 20:17:22.779 +00:00 [INF] Environment: DEV; version: 1.0; time c
heck: 20:17.22\n", "stream": "stdout", "time": "2020-06-29T20:17:23.682712Z"}
/var/log/containers #
/var/log/containers # exit
PS>
```

Log entries are stored as JSON with fields for the message, timestamp and stream.

Figure 13.3 For a modern platform, Kubernetes has an old-school approach to log storage

Log files are retained after a Pod restart but most Kubernetes implementations include a log rotation system running on the nodes - outside of Kubernetes - to prevent logs swallowing up all your disk. Collecting and forwarding logs to a central store lets you keep them for longer and isolate log storage in one place, and that also applies to logs from core Kubernetes components. The Kubernetes DNS server, API server and network proxy all run as Pods, and you can view and collect logs from them in the same way as application logs.

TRY IT NOW Not every Kubernetes node runs the same core components, but you can use the sleep Pod to see which common components are running on your node.

```
# connect to a session in the Pod:
kubectl exec -it deploy/sleep -- sh

# browse to the host path volume:
cd /var/log/containers/

# the network proxy runs on every node:
cat $(ls kube-proxy*) | tail -n 1
# if your cluster uses Core DNS you'll see logs here:
cat $(ls coredns*) | tail -n 1
```

```
# and if your node is running the API server you'll see these logs:
cat $(ls kube-apiserver*) | tail -n 1

# leave the session:
exit
```

You might get a different output from that exercise, depending on how your lab cluster is set up. The network proxy Pod runs on every node so you should see those logs, but you'll only see DNS logs if your cluster is using CoreDNS (which is the default DNS plugin) and you'll only see API server logs if your node is running the API server. My output from Docker Desktop is in figure 13.4; if you see something different you can run `ls *.log` to see all the Pod log files on your node.

Core Kubernetes components run in Pods and the container log output is stored in the same way as application Pods, in log files on the node.

```
PS>kubectl exec -it deploy/sleep -- sh
/ #
/ # cd /var/log/containers/
/var/log/containers #
/var/log/containers # cat $(ls kube-proxy*) | tail -n 1
>{"log": "I0629 19:55:09.045880      1 shared_informer.go:230] Caches are synced for endpoints config \n", "stream": "stderr", "time": "2020-06-29T19:55:09.0493553Z"}
/var/log/containers #
/var/log/containers # cat $(ls coredns*) | tail -n 1
>{"log": "linux/amd64, go1.12.8, 795a3eb\n", "stream": "stdout", "time": "2020-06-29T19:55:09.2239534Z"}
/var/log/containers #
/var/log/containers # cat $(ls kube-apiserver*) | tail -n 1
>{"log": "I0629 20:05:45.466396      1 controller.go:606] quota admission added evaluator for: events.events.k8s.io\n", "stream": "stderr", "time": "2020-06-29T20:05:45.4665941Z"}
/var/log/containers #
/var/log/containers # exit
```

None of these entries are very entertaining, but you should collect core system logs as well as application logs for troubleshooting cluster issues.

Figure 13.4 Collecting and forwarding logs from the node will also include all the system Pod logs

Now that you know how container logs are processed and stored by Kubernetes, you can see how a centralized log system makes troubleshooting so much easier. A collector runs on every node, grabbing entries from the log files and forwarding them, and in the rest of the chapter you'll learn how to implement that with the EFK stack - Elasticsearch, Fluentd and Kibana.

13.2 Collecting logs from nodes with Fluentd

Fluentd is a CNCF project, so it has a sound foundation behind it and it's a mature and very popular product. There are alternative log collection components but Fluentd is a good choice

because it has a powerful processing pipeline to manipulate and filter log entries, and a pluggable architecture so it can forward logs to different storage systems. It also comes in two variants - the full Fluentd is fast and efficient and has over 1,000 plugins, but we'll be using the minimal alternative called Fluent Bit.

Fluent Bit was originally developed as a lightweight version of Fluentd for embedded applications like IoT devices, but it has all the functionality you need for log aggregation in a full Kubernetes cluster. Every node will run a log collector so it makes sense to keep the impact of that component small, and Fluent Bit happily runs in a few tens of megabytes of memory. The Fluent Bit architecture in Kubernetes is straightforward - a DaemonSet runs a collector Pod on every node, which uses a HostPath volume mount to access the log files, just like in the sleep example we've used. Fluent Bit supports different outputs, so we'll start simple and just log to the console in the Fluent Bit Pod.

TRY IT NOW Deploy Fluent Bit with a configuration set up to read the timecheck log files and write them to the standard output stream of the Fluent Bit container.

```
# deploy the DaemonSet and ConfigMap:
kubectl apply -f fluentbit/

# wait for Fluent Bit to start up:
kubectl wait --for=condition=ContainersReady pod -l app=fluent-bit -n kiamol-ch13-logging

# check the logs of the Fluent Bit Pod:
kubectl logs -l app=fluent-bit -n kiamol-ch13-logging --tail 2
```

My output is in figure 13.5 where you can see the logs from the timecheck containers being surfaced in the Fluent Bit container. The Pods creating the log entries are in different namespaces, but Fluent Bit reads them from the files on the node - the content is the raw JSON plus a more precise timestamp which Fluent Bit adds to each log entry.

Deploys the Fluent Bit log processor in its own logging namespace.

```

PS> kubectl apply -f fluentbit/
namespace/kiamol-ch13-logging created
configmap/fluent-bit-config created
daemonset.apps/fluent-bit created
PS>
PS> kubectl wait --for=condition=ContainersReady pod -l app=fluent-bit -n kiamol-ch13-logging
pod/fluent-bit-wwmzq condition met
PS>
PS> kubectl logs -l app=fluent-bit -n kiamol-ch13-logging --tail 2
{"date":1593502383.721161,"log":"2020-06-30 07:33:02.957 +00:00 [INF] Environment: DEV; version: 1.0; time check: 07:33.02\n","stream":"stdout","time":"2020-06-30T07:33:03.7211608Z"}
{"date":1593502386.98684,"log":"2020-06-30 07:33:06.659 +00:00 [INF] Environment: TEST; version: 1.1; time check: 07:33.06\n","stream":"stdout","time":"2020-06-30T07:33:06.9868398Z"}

```

Fluent Bit is configured to read the timecheck log files as input, and forward them to its own standard output stream. You can see log entries from both the dev and test namespaces.

Figure 13.5 A very basic Fluent Bit configuration can still aggregate log entries from multiple Pods

There's nothing in the DaemonSet spec for Fluent Bit that you haven't already seen. I'm using a separate namespace for logging because you typically want it to run as a shared service used by all the applications running on the cluster, and a namespace is a good way to isolate all the objects. It's simple to run the Fluent Bit Pods, but the complexity comes in configuring the log processing pipeline, and we'll need to dive into that to get the most out of the logging model. Figure 13.6 shows the stages of the pipeline and how you can use them.

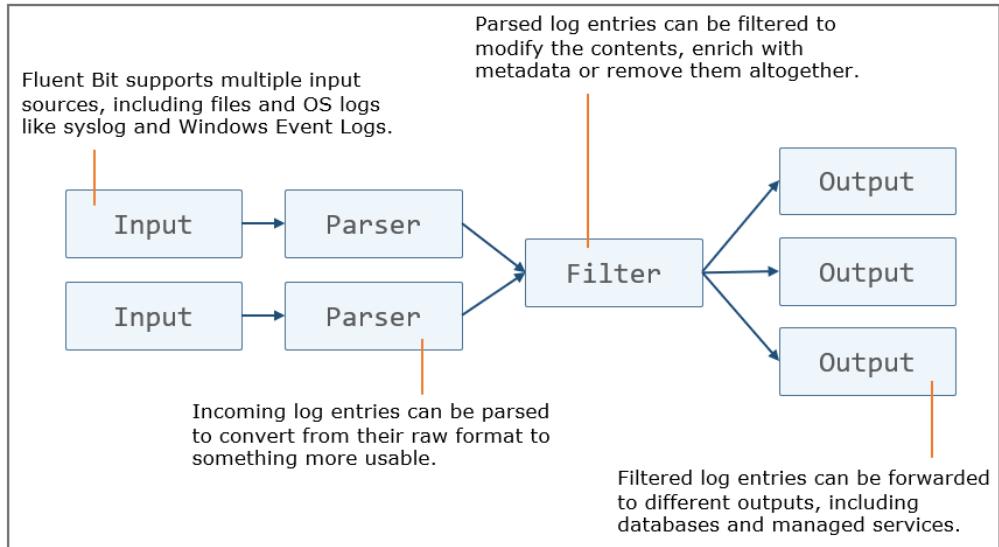


Figure 13.6 Fluent Bit's processing pipeline is super flexible, using plugin modules for every stage

We're currently running a very simple configuration with three stages: the *input* stage reads log files, the *parser* stage deconstructs the JSON log entries and the *output* stage writes each log as a separate line to the standard output stream in the Fluent Bit container. The JSON parser is standard for all container logs and isn't very interesting, so we'll focus on the input and output configuration in listing 13.1.

Listing 13.1 - fluentbit-config.yaml, a simple Fluent Bit pipeline

```
[INPUT]
  Name      tail      # reads from the end of a file
  Tag       kube.*    # use a prefix for the tag
  Path      /var/log/containers/timecheck*.log
  Parser    docker    # parses the JSON container logs
  Refresh_Interval 10   # frequency to check the file list

[OUTPUT]
  Name      stdout    # writes to standard out
  Format   json_lines # formatting each log as a line
  Match    kube.*     # writes logs with a kube tag prefix
```

Fluent Bit uses *tags* to identify the source of a log entry - the tag gets added at the input stage, and it can be used to route logs to other stages. In this configuration the log file name is used as the tag, prefixed with `kube`. The match rule routes all the `kube` tagged entries to the output stage so every log is printed out, but the input stage only reads the `timecheck` log files, so those are the only log entries you see.

You don't really want to filter the input files - that's just a quick way to get started without flooding you with log entries. It's better to read all the input and then route logs based on tags, so you only store the entries you're interested in. Fluent Bit has built-in support for Kubernetes with a *filter* that can enrich log entries with metadata to identify the Pod that created it. The filter can also be configured to build a custom tag for each log which includes the namespace and Pod name; using that you can alter the pipeline so only the logs from the test namespace are written to standard out.

TRY IT NOW Update the Fluent Bit ConfigMap to use the Kubernetes filter and restart the DaemonSet to apply the config change. Then print the latest log from the timecheck app to see what the filter does.

```
# update the data pipeline configuration files:  
kubectl apply -f fluentbit/update/fluentbit-config-match.yaml  
  
# restart the DaemonSet so a new Pod gets the changed config:  
kubectl rollout restart ds/fluent-bit -n kiamol-ch13-logging  
  
# wait for the new logging Pod:  
kubectl wait --for=condition=ContainersReady pod -l app=fluent-bit -n kiamol-ch13-logging  
  
# and print the last log entry:  
kubectl logs -l app=fluent-bit -n kiamol-ch13-logging --tail 1
```

You can see from my output in figure 13.7 that a lot more data is coming through Fluent Bit - the log entry is the same but it's been enriched with the details of the source of the log. The Kubernetes filter fetches all that data from the API server. That gives you the additional context you really need when you're analyzing logs to track down issues - seeing the image hash for the container will let you check the software version with complete certainty.

This updated configuration applies the Kubernetes filter and restricts the output to logs from the test namespace.

Fluent Bit doesn't pick up config changes, so the Pod needs to be restarted.

```

PS>kubectl apply -f fluentbit/update/fluentbit-config-match.yaml
configmap/fluent-bit-config configured
PS>
PS>kubectl rollout restart ds/fluent-bit -n kiamol-ch13-logging
daemonset.apps/fluent-bit restarted
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=fluent-bit -n kiamol-ch1
3-logging
pod/fluent-bit-dfv2w condition met
PS>
PS>kubectl logs -l app=fluent-bit -n kiamol-ch13-logging --tail 1
{"date":1593505469.649095,"log":"2020-06-30 08:24:28.869 +00:00 [INF] Environment:
TEST; version: 1.1; time check: 08:24.28\n","stream":"stdout","time":"2020-06-30T08
:24:29.6490953Z","kubernetes":{"pod_name":"timecheck-cf4d4f685-rwq4c","namespace_na
me":"kiamol-ch13-test","pod_id":"9e8420f6-4cf5-4fc2-980d-7ce1870bae4d","labels":{"a
pp":"timecheck","pod-template-hash":"cf4d4f685"},"host":"docker-desktop","container
_name":"logger","docker_id":"fad628e7a3d248c9c882816191a34976525b158b9a9ca48bf86352
11336149c9","container_hash":"kiamol/ch03-sleep@sha256:f5edb5cb6df332c1bd77b33e4af1
13a3418666638065d074f8ae1abb4b5af567","container_image":"kiamol/ch03-sleep:latest"}
}

```

The timecheck app is producing the same log entries, and the Kubernetes filter enriches them with metadata about the source - including the Pod name, the container image tag and the unique hash of the image.

Figure 13.7 Filters enrich log entries - the single log message now has 14 additional metadata fields

The Fluent Bit configuration for this is a little bit tricky. The Kubernetes filter works out of the box to fetch all the Pod metadata, but building a custom tag for routing needs some fiddly regular expressions. That's all in the configuration files in the ConfigMap you deployed in the last exercise, but I'm not going to focus on it because I can't stand regular expressions. And also because there's no need - the setup is completely generic, so you can plug the input, filter and parser config into your own cluster and it will work for your apps without any changes.

The output configuration will be different because that's how you configure the targets. We'll look at one more feature of Fluent Bit before we plug in the log storage and search components - routing log entries to different outputs. The regular expression in the input configuration sets a custom tag for entries in the format `kube.namespace.container_name.pod_name`, and that can be used in matches to route logs differently based on their namespace or pod name. Listing 13.2 shows an updated output configuration with multiple destinations.

Listing 13.2 - fluentbit-config-match-multiple.yaml, routing to multiple outputs

[OUTPUT]		
Name	stdout	# the standard out plugin will
Format	json_lines	# only print log entries where
Match	kube.kiamol-ch13-test.*	# the namespace is test

```
[OUTPUT]
Name      counter          # the counter prints a count of
Match    kube.kiamol-ch13-dev.*   # logs from the dev namespace
```

Fluent Bit supports many output plugins - from plain TCP to Postgres and cloud services like Azure Log Analytics. We've used the standard output stream so far, which just relays log entries to the console. The counter plugin is a very simple output which just prints how many log entries have been collected. When you deploy the new configuration you'll continue to see the log lines from the test namespace, and you'll also see a counter of log entries from the dev namespace.

TRY IT NOW Update the configuration to use multiple outputs and print the logs from the Fluent Bit Pod.

```
# update configuration and restart Fluent Bit:
kubectl apply -f fluentbit/update/fluentbit-config-match-multiple.yaml

kubectl rollout restart ds/fluent-bit -n kiamol-ch13-logging

kubectl wait --for=condition=ContainersReady pod -l app=fluent-bit -n kiamol-ch13-logging

# print the last two log lines:
kubectl logs -l app=fluent-bit -n kiamol-ch13-logging --tail 2
```

The counter in this exercise isn't especially useful, but it's there to show you that the complex bits in the early part of the pipeline make for very easy routing later in the pipeline. Figure 13.8 shows I have different output for logs in different namespaces, and I can configure that purely using match rules in the output stages.

This configuration update sets two outputs, standard out for logs from the test namespace and a simple counter for logs from the dev namespace.

```
PS>kubectl apply -f fluentbit/update/fluentbit-config-match-multiple.yaml
configmap/fluent-bit-config configured
PS>
PS>kubectl rollout restart ds/fluent-bit -n kiamol-ch13-logging
daemonset.apps/fluent-bit restarted
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=fluent-bit -n kiamol-ch1
3-logging
pod/fluent-bit-psg64 condition met
PS>
PS>kubectl logs -l app=fluent-bit -n kiamol-ch13-logging --tail 2
{"date":1593508266.990381,"log":"2020-06-30 09:11:06.659 +00:00 [INF] Environment:
TEST; version: 1.1; time check: 09:11:06\n","stream":"stdout","time":"2020-06-30T09
:11:06.9903809Z","kubernetes":{"pod_name":"timecheck-cf4d4f685-fz5m7","namespace_na
me":"kiamol-ch13-test","pod_id":"6306ef10-7337-4957-a5cb-588a694bbe51","labels":{"a
pp":"timecheck","pod-template-hash":"cf4d4f685"},"host":"docker-desktop","container
_name":"logger","docker_id":"5b995663270a1a568e5667e9ab00fdc6478693f47a87469e527e06
52614e106a","container_hash":"kiamol/ch03-sleep@sha256:f5edb5cb6df332c1bd77b33e4af1
13a3418666638065d074f8ae1abb4b5af567","container_image":"kiamol/ch03-sleep:latest"}
}
1593508267.953121,1 (total = 2266)
```

The dev logs pass through the same pipeline so they have all the same metadata applied, but the output only shows a count of entries - 2,266. The test log entries are shown in full.

Figure 13.8 Different outputs in Fluent Bit can reshape the data - the counter just shows a count

It should be clear how you can plug a sophisticated logging system on top of the simple log files that Kubernetes writes. The data pipeline in Fluent Bit lets you enrich log entries and route them to different outputs. If the output you want to use isn't supported by Fluent Bit then you can switch to the parent Fluentd project which has a larger set of plugins (including MongoDB and AWS S3) - the pipeline stages and configuration are very similar. We'll be using Elasticsearch for storage, which is perfect for high-performance search, and very simple to integrate with Fluent Bit.

13.3 Shipping logs to Elasticsearch

Elasticsearch is a production-grade open-source database - it stores items as *documents* in collections called *indexes*. It's a very different storage model from a relational database because there isn't a fixed schema for every document in an index - each data item can have its own set of fields. That works nicely for centralized logging where the log items from different systems will have different fields. Elasticsearch runs as a single component with a REST API to insert and query data. There's a companion product called Kibana which provides a very usable front-end to query Elasticsearch. You can run both components in Kubernetes in the same shared logging namespace as Fluent Bit.

TRY IT NOW Deploy Elasticsearch and Kibana - the storage and front-end components of the logging system.

```
# create the Elasticsearch deployment and wait for the Pod:  
kubectl apply -f elasticsearch/  
  
kubectl wait --for=condition=ContainersReady pod -l app=elasticsearch -n kiamol-ch13-  
logging  
  
# create the Kibana deployment and wait for it to start:  
kubectl apply -f kibana/  
  
kubectl wait --for=condition=ContainersReady pod -l app=kibana -n kiamol-ch13-logging  
  
# get the URL for Kibana:  
kubectl get svc kibana -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:5601' -n  
kiamol-ch13-logging
```

This is a basic deployment of Elasticsearch and Kibana using a single Pod for each, as you see in figure 13.9. Logs are important so you'll want to model for high-availability in production. Kibana is a stateless component so you can increase the replica count to increase reliability. Elasticsearch works nicely as a StatefulSet across multiple Pods using persistent storage, or you can use a managed Elasticsearch service in the cloud. When you have Kibana running you can browse to the URL - we'll be using it in the next exercise.

These Elasticsearch and Kibana Deployments use custom container images, which are much smaller than the official images. You should switch to the official images in production.

```
PS>kubectl apply -f elasticsearch/  
service/elasticsearch created  
deployment.apps/elasticsearch created  
PS>  
PS>kubectl wait --for=condition=ContainersReady pod -l app=elasticsearch -n kiamol-  
ch13-logging  
pod/elasticsearch-57b6f8d8fc-d7vw4 condition met  
PS>  
PS>kubectl apply -f kibana/  
service/kibana created  
deployment.apps/kibana created  
PS>  
PS>kubectl wait --for=condition=ContainersReady pod -l app=kibana -n kiamol-ch13-lo  
gging  
pod/kibana-68bbb448-nqxz2 condition met  
PS>  
PS>kubectl get svc kibana -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:5  
601' -n kiamol-ch13-logging  
http://localhost:5601
```

Kibana connects to the Elasticsearch REST API and uses port 5601 for the web UI.

Figure 13.9 Running Elasticsearch with a Service so Kibana and Fluent Bit can use the REST API

Fluent Bit has an Elasticsearch output plugin which creates a document for each log entry using the Elasticsearch REST API. The plugin needs to be configured with the domain name of the Elasticsearch server, and you can optionally specify the index where documents should be created. That lets you isolate log entries from different namespaces in different indexes, using multiple output stages - listing 13.3 separates log entries from Pods in the test namespace and Kubernetes system Pods.

Listing 13.3 - fluentbit-config-elasticsearch.yaml, storing logs in Elasticsearch indexes

```
[OUTPUT]
  Name      es          # logs from the test namespace
  Match    kube.kiamol-ch13-test.*
  Host     elasticsearch
  Index    test         # and created as documents in
                      # the "test" index

[OUTPUT]
  Name      es          # system logs are created in
  Match    kube.kube-system.*
  Host     elasticsearch
  Index    sys          # the "sys" index in the same
                      # Elasticsearch server
```

If there are log entries which don't match any output rules, they get discarded. When you deploy this updated configuration, the Kubernetes system logs and the test namespace logs get saved in Elasticsearch, but the logs from the dev namespaces aren't saved.

TRY IT NOW Update the Fluent Bit configuration to send logs to Elasticsearch, then connect to Kibana and set up a search over the test index.

```
# deploy the updated configuration from listing 13.3
kubectl apply -f fluentbit/update/fluentbit-config-elasticsearch.yaml

# update Fluent Bit and wait for it to restart
kubectl rollout restart ds/fluent-bit -n kiamol-ch13-logging
kubectl wait --for=condition=ContainersReady pod -l app=fluent-bit -n kiamol-ch13-logging

# now browse to Kibana and set up the search:
# - click Discover on the left navigation and add a new index pattern
# - enter "test" as the index pattern
# - in the next step select @timestamp as the time filter field
# - click Create Index Pattern
# - click Discover again on the left navigation to see the logs
```

There are a few manual steps there because Kibana isn't a great product to automate - my output in figure 13.10 shows the index pattern being created. When you finish that exercise you'll have a powerful, fast and easy-to-use search engine for all the container logs in the test namespace. The Discover tab in Kibana shows you the rate of documents stored over time - which is the rate that logs are processed - and you can drill down into each document to see the log details.

This configuration sends log entries to Elasticsearch so they can be queried using Kibana.

```
PS>kubectl apply -f fluentbit/update/fluentbit-config-elasticsearch.yaml
configmap/fluent-bit-config configured
```

```
PS>
PS>kubectl rollout restart ds/fluent-bit -n kiamol-ch13-logging
daemonset.apps/fluent-bit restarted
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=fluent-bit -n kiamol-ch1
3-logging
pod/fluent-bit-zxlf4 condition met
PS>
PS>[Kibana]
```

You need to manually complete the setup in Kibana, using test as the index pattern in this screen, and selecting the @timestamp field in the next screen.

Figure 13.10 Setting up Fluent Bit to send logs to Elasticsearch and Kibana to search the test index

Elasticsearch and Kibana are well-established technologies but if you're new to them now is a good time to look around the Kibana UI. You'll see a list of fields on the left of the Discover page which you can use to filter the logs - and those fields contain all the Kubernetes metadata, so you can filter by Pod name, host node, container image and more. You can build dashboards which show headline statistics for logs split by application, which would be useful to show a sudden spike in error logs, and you can search for specific values across all documents - which is a good way to find application logs when a user gives you the ID from an error message.

I won't spend too long on Kibana, but one more exercise will show how useful it is to have a centralized logging system. We'll deploy a new application into the test namespace, and its logs will automatically get picked up by Fluent Bit and flow through into Elasticsearch without any changes to configuration. When the app shows an error to the user, we can track that down easily in Kibana.

TRY IT NOW Deploy the random number API we've used before - the one that crashes after the first use - along with a proxy which caches the response and almost fixes the problem. Try the API and when you get an error, you can search for the failure ID in Kibana.

```
# deploy the API and proxy:  
kubectl apply -f numbers/  
  
# wait for the app to start up:  
kubectl wait --for=condition=ContainersReady pod -l app=numbers-api -n kiamol-ch13-test  
  
# get the URL to use the API via the proxy:  
kubectl get svc numbers-api-proxy -o  
  jsonpath='http://{{.status.loadBalancer.ingress[0].*:8080/rng}}' -n kiamol-ch13-test  
  
# browse to the API, wait 30 seconds and refresh until you get an error;  
# browse to Kibana and enter this query in the search bar:  
# kubernetes.labels.app:numbers-api AND log:<failure-ID-from-the-API>
```

My output in figure 13.11 is tiny but you can see what's happening - I got a failure ID from the API and I've pasted that into the search bar for Kibana, which returns a single match. The log entry contains all the information I need to go and investigate the Pod if I need to, and Kibana also has a useful option to display documents before and after a match, which I could use to show the log entries surrounding the failure log.

Deploys the random number API along with a caching proxy. The API fails after the first call, but the proxy uses the cached response for 30 seconds.

```
PS>kubectl apply -f numbers/
service/numbers-api created
deployment.apps/numbers-api created
service/numbers-api-proxy created
configmap/numbers-api-proxy-config created
deployment.apps/numbers-api-proxy created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=numbers-api -n kiamol-ch13-test
pod/numbers-api-554474cc8b-mfnmv condition met
PS>
PS>kubectl get svc numbers-api-proxy -o jsonpath='http://{{.status.loadBalancer.ingress[0].*:8080/rng}}' -n kiamol-ch13-test
http://localhost:8080/rng
```

When the cache expires, the proxy tries the API again and the user gets a failure message with a unique ID.

An operator can investigate the issue by searching for that failure ID in the logs, which identifies the Pod that caused the error.

Figure 13.11 The logging system in action - tracking down failures from user-facing error messages

Searchable, centralized logging removes a lot of the friction from troubleshooting - with the bonus that these are all open-source components so you can run the exact same logging stack in every environment. Using the same diagnostic tools in dev and test that you use in production should help product teams understand the level of logging that's useful, and improve the quality of the system logs. Good quality logging is important but it seldom ranks highly in a product backlog - so in some applications you're going to be stuck with logs that aren't very useful. Fluent Bit has a couple more features which can help there too.

13.4 Parsing and filtering log entries

The ideal application would produce structured log data with fields for the severity of the entry, and the name of the class writing the output, along with an ID for the type of event and the key data items of the event. You could use the values of those fields in your Fluent Bit pipeline

to filter messages, and the fields would be surfaced in Elasticsearch so you can build more precise queries. Most systems don't produce logs like that, they just emit text - but if the text uses a known format then Fluent Bit can parse it into fields as it passes through the pipeline.

The random number API is a simple example - log entries are lines of text which look like this: <6>Microsoft.Hosting.Lifetime[0] Now listening on: http://[::]:80. The first part in angle brackets is the priority of the message, followed by the name of the class and the event ID in square brackets, then the actual content of the log. The format is the same for every log entry, so a Fluent Bit parser can split the log into individual fields. You have to use a regular expression for that and listing 13.4 shows my best effort, which just extracts the priority field and leaves everything else in the message field.

Listing 13.4 - fluentbit-config-parser.yaml, a custom parser for application logs

```
[PARSER]
  Name      dotnet-syslog          # name of the parser
  Format    regex                 # parse with a regular expression
  Regex     ^\<(?<priority>[0-9]+)\>*(?<message>.*)>$      # yuck
```

When you deploy this configuration Fluent Bit will have a new custom parser called `dotnet-syslog` available to use, but it won't apply it to any logs. The pipeline needs to know which log entries should use custom parsers, and Fluent Bit lets you set that up with annotations in your Pods. These act like hints, telling the pipeline to apply a named parser to any logs which have originated from this Pod. Listing 13.5 shows the parser annotation for the random number API Pod - it's as simple as that.

Listing 13.5 - api-with-parser.yaml, Pod spec with a custom Fluent Bit parser

```
# this is the Pod template in the Deployment spec
template:
  metadata:                      # labels are used for selectors and
    labels:                       # operations, annotations are often
      app: numbers-api           # used for integration flags
    annotations:
      fluentbit.io/parser: dotnet-syslog    # use the parser for Pod logs
```

Parsers can be much more effective than my custom one, and the Fluent Bit team have some sample parsers in their documentation - including one for Nginx. I'm using Nginx as a proxy for the random number API, and in the next exercise we'll add parsers to each component with annotations and see how structured logging makes for more targeted searching and filtering in Kibana.

TRY IT NOW Update the Fluent Bit configuration to add parsers for the random number app and the Nginx proxy, then update those deployments to add annotations specifying the parser. Try the app and check the logs in Kibana.

```
# update the pipeline configuration:
kubectl apply -f fluentbit/update/fluentbit-config-parser.yaml

# restart Fluent Bit:
kubectl rollout restart ds/fluent-bit -n kiamol-ch13-logging
```

```
kubectl wait --for=condition=ContainersReady pod -l app=fluent-bit -n kiamol-ch13-logging  
# update the app Deployments, adding parser annotations:  
kubectl apply -f numbers/update/  
  
# wait for the API to be ready:  
kubectl wait --for=condition=ContainersReady pod -l app=numbers-api -n kiamol-ch13-test  
  
# use the API again and browse to Kibana to see the logs
```

You can see in figure 13.12 that the promoted fields from the parser are available for Kibana to filter on, without me having to build up my own query. In my screenshot I've filtered to show logs from one Pod which have a priority value of 4 (which is a warning level). When you run this yourself you'll see that you can also filter for the API proxy Pod, and then the log entries include fields for the HTTP request path and the response code, all parsed from Nginx text logs.

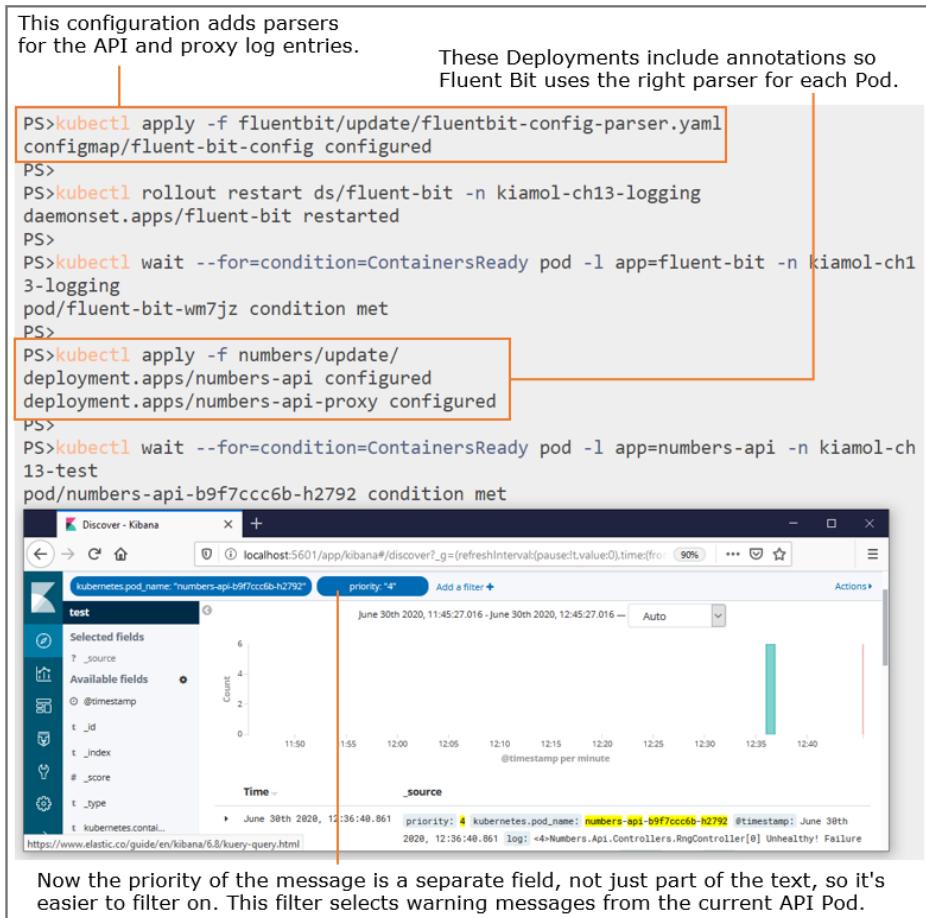


Figure 13.12 Parsed fields from the logs are indexed, so filters and searches are faster and simpler

There's one final benefit of the centralized logging system with Fluent Bit - the data processing pipeline is independent of the applications and it can be a better place to apply filtering. That mythical ideal app would be able to increase or decrease logging levels on the fly, without an application restart - but you know from chapter 4 that many apps need a Pod restart to pick up the latest config changes. That's not good when you're troubleshooting a live issue because it means restarting the affected app if you need to increase the logging level.

Fluent Bit doesn't support live config reloads itself, but restarting the log collector Pods is less invasive than restarting application Pods, and Fluent Bit will pick up where it left off so you won't miss any log entries. With this approach you can log at a more verbose level in your applications and filter in the Fluent Bit pipeline. Listing 13.6 shows a filter which only includes

logs from the random number API if the priority field has a value of 2, 3, or 4 - filtering out lower priority entries.

Listing 13.6 - fluentbit-config-grep.yaml, filtering logs based on field values

```
[FILTER]
  Name      grep          # grep is a search filter
  Match    kube.kiamol-ch13-test.api.numbers-api*
  Regex    priority [234]   # even I can manage this regex
```

More regular expression wrangling here, but you can see why it's important to have text log entries split into fields that the pipeline can access. The grep filter can include or exclude logs by evaluating a regular expression over a field. When you deploy this updated configuration, the API can happily write log entries at level 6 but they get dropped by Fluent Bit and only the more important entries will make it to Elasticsearch.

TRY IT NOW Deploy the updated configuration so only high priority logs from the random number API get saved. Delete the API Pod and in Kibana you won't see any of the startup log entries - but they're still there in the Pod logs.

```
# apply the grep filter from listing 13.6:
kubectl apply -f fluentbit/update/fluentbit-config-grep.yaml

kubectl rollout restart ds/fluent-bit -n kiamol-ch13-logging

# delete the old API pod so we get a fresh set of logs:
kubectl delete pods -n kiamol-ch13-test -l app=numbers-api

kubectl wait --for=condition=ContainersReady pod -l app=numbers-api -n kiamol-ch13-test

# use the API and refresh until you see a failure

# print the logs from the Pod:
kubectl logs -n kiamol-ch13-test -l app=numbers-api

# now browse to Kibana and filter to show the API Pod logs
```

That exercise shows you how Fluent Bit can filter out logs very effectively, only forwarding log entries you care about to the target output. It also shows that the lower level logging hasn't disappeared - the raw container logs are all available to see with Kubectl, it's only the subsequent log processing that stops them going to Elasticsearch. In a real troubleshooting scenario you may be able to use Kibana to identify the Pod causing the problem and then drill down with Kubectl, as you see in figure 13.13.

This configuration updates the pipeline so only high priority logs from the API get saved.

Deleting the API Pod will create a new one with a new set of logs.

```

PS> kubectl apply -f fluentbit/update/fluentbit-config-grep.yaml
configmap/fluent-bit-config configured
PS>
PS> kubectl rollout restart ds/fluent-bit -n kiamol-ch13-logging
daemonset.apps/fluent-bit restarted
PS>
PS> kubectl delete pods -n kiamol-ch13-test -l app=numbers-api
pod "numbers-api-b9f7ccc6b-flfj7" deleted
PS>
PS> kubectl wait --for=condition=ContainersReady pod -l app=numbers-api -n kiamol-ch
13-test
pod/numbers-api-b9f7ccc6b-6q89s condition met
PS>
PS> kubectl logs -n kiamol-ch13-test -l app=numbers-api
<6>Microsoft.Hosting.Lifetime[0] Now listening on: http://[::]:80
<6>Microsoft.Hosting.Lifetime[0] Application started. Press Ctrl+C to shut down.
<6>Microsoft.Hosting.Lifetime[0] Hosting environment: Production
<6>Microsoft.Hosting.Lifetime[0] Content root path: /app
<4>Numbers.Api.Controllers.RngController[0] Unhealthy! Failure ID: d1aff0c4-9beb-41
c0-8621-a7c7b1f76fce
PS>
PS>
```

I've hidden the graph here; the filter for the new API Pod shows that only the priority 4 logs have been saved, the priority 6 logs were removed in the pipeline.

But the log entries were still created by the app and they can be seen in the Pod logs.

Figure 13.13 Filtering log entries in Fluent Bit saves on storage and you can easily change the filter

There's plenty more to Fluent Bit than we've covered in these simple pipelines - you can modify log contents, throttle the rate of incoming logs, and even run custom scripts triggered by log entries. But we've covered all the main features you're likely to need, and we'll wrap up by looking at the collect-and-forward logging model compared to other options.

13.5 Understanding logging options in Kubernetes

Kubernetes has an expectation that your application logs will come from the container's standard output streams. It collects and stores all the content from those streams and that powers the logging model we've covered in this chapter. It's a generic and flexible approach,

and the technology stack we've used is reliable and performant, but there are inefficiencies along the way. Figure 13.14 shows some of the issues getting logs from containers into searchable storage.

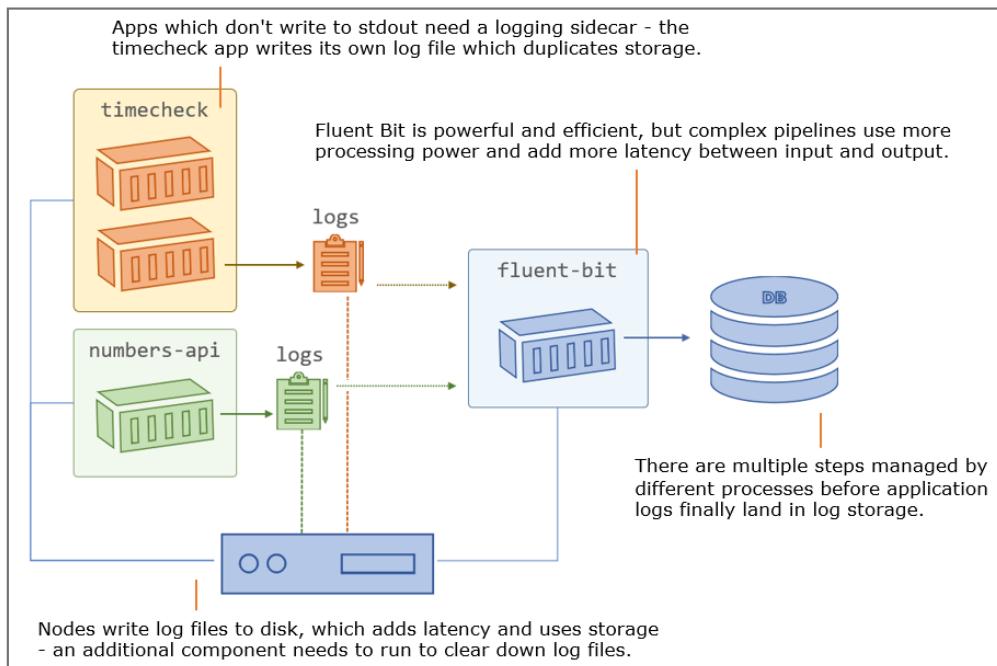


Figure 13.14 The goal is to get application logs into Elasticsearch, but it takes many steps to get there

There are alternative architectures which are simpler and have fewer moving pieces. You could write logs directly to Elasticsearch from your application code, or you could run a sidecar in every application Pod which reads from whatever log sink the app uses and pushes entries to Elasticsearch. That would give you a lot more control over the log data you store, without resorting to regular expressions to parse text strings. It ties you in to Elasticsearch (or whichever storage system you use), but that may not be a big concern if that system gives you all you need.

A custom logging framework might be appealing for the first app you run on Kubernetes, but as you move more workloads to the cluster it's going to restrict you. Requiring apps to log directly to Elasticsearch won't fit for existing apps that write to operating system logs, and you'll soon find your logging sidecar isn't flexible enough and needs tweaking for every new application. The advantage of the Fluentd/Fluent Bit model is that it's a standard approach with a community behind it; fiddling with regular expressions is much less hassle than writing and maintaining your own log collection and forwarding code.

That's all for application logs, so we can clear down the cluster to get ready for the lab.

TRY IT NOW Remove this chapter's namespaces and the remaining Deployment.

```
kubectl delete ns -l kiamol=ch13  
kubectl delete all -l kiamol=ch13
```

13.6 Lab

In this lab you play the part of an operator who needs to deploy a new app into a cluster which is using the logging model from this chapter. You'll need to check the Fluent Bit configuration to find the namespace you should use for your app, and then deploy the simple versioned website we've used before in the book. Here are the parts to this lab:

- start by deploying the logging components in the `lab/logging` folder
- deploy the app from the `vweb` folder to the correct namespace so logs are collected and verify you can see the logs in Kibana
- you'll see the logs are plain text, so the next step is to update your Deployment to use the correct parser - the app runs on Nginx and there's already an Nginx parser set up for you in the Fluent Bit configuration
- when you confirm the new logs in Kibana you'll see there are lots where the status code is 304, which tells the browser to use its cached version of the page. Those logs aren't interesting so the final task is to update the Fluent Bit configuration to filter them out.

This is a very real-world task where you'll need all the basic skills of navigating around Kubernetes to find and update all the pieces. My solution is in the usual place on GitHub for you to check: <https://github.com/sixeyed/kiamol/blob/master/ch13/lab/README.md>

14

Monitoring applications and Kubernetes with Prometheus

Monitoring is the companion to logging - your monitoring system tells you something is wrong and then you can dig into the logs to find out the detail. And like logging you want to have a centralized system to collect and visualize metrics about all your application components. There's an established approach for monitoring in Kubernetes using another CNCF project - Prometheus, which is a server application that collects and stores metrics. In this chapter you'll learn how to deploy a shared monitoring system in Kubernetes with dashboards that show the health of individual applications, and the cluster as a whole.

Prometheus runs on many platforms but it's particularly well suited to Kubernetes. You run Prometheus in a Pod which has access to the Kubernetes API server, and then Prometheus queries the API to find all the targets it needs to monitor. When you deploy new apps you don't need to make any setup changes — Prometheus discovers them automatically and starts collecting metrics. Kubernetes apps are particularly well suited to Prometheus too - you'll see in this chapter how to make good use of the sidecar pattern, so every app can provide some metrics to Prometheus, even if the application itself isn't Prometheus-ready.

14.1 How Prometheus monitors Kubernetes workloads

Metrics in Prometheus are completely generic - each component you want to monitor has an HTTP endpoint which returns all the values which are important to that component. A web server includes metrics for the number of requests it serves and a Kubernetes node includes metrics for how much memory is available. Prometheus doesn't care what's in the metrics, it just stores everything the component returns; what's important to Prometheus is list of targets it needs to collect from. Figure 14.1 shows how that works in Kubernetes, using Prometheus' built-in service discovery.

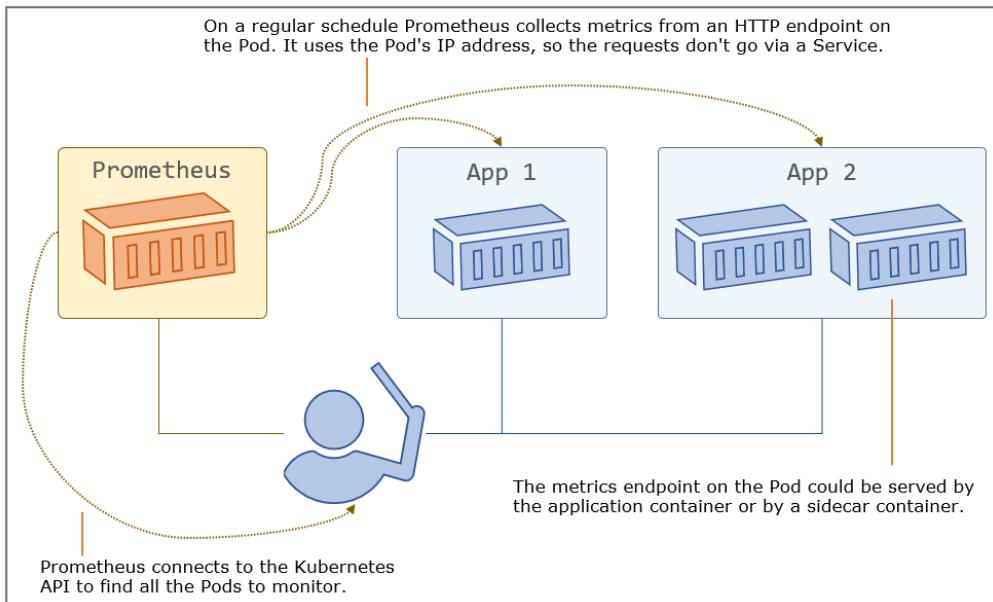


Figure 14.1 Prometheus uses a pull model to collect metrics, automatically finding targets

The focus on this chapter is getting Prometheus working nicely with Kubernetes, to give you a dynamic monitoring system that keeps working as your cluster expands with more nodes running more applications. I won't go into much detail on how you add monitoring to your applications or what metrics you should record - Appendix 2 of this book is the chapter *Adding observability with containerized monitoring* from *Learn Docker in a Month of Lunches*, which will give you that additional detail.

We'll start by getting Prometheus up and running. The Prometheus server is a single component which takes care of service discovery and metrics collection and storage, and it has a basic web UI which you can use to check the status of the system and run simple queries.

TRY IT NOW Deploy Prometheus in a dedicated monitoring namespace, configured to find apps in a test namespace (the test namespace doesn't exist yet).

```
# switch to this chapter's folder:
cd ch14

# create the Prometheus Deployment and ConfigMap:
kubectl apply -f prometheus/

# wait for Prometheus to start:
kubectl wait --for=condition=ContainersReady pod -l app=prometheus -n kiamol-ch14-monitoring

# get the URL for the web UI:
```

```
kubectl get svc prometheus -o jsonpath='http://{.status.loadBalancer.ingress[0].*}:9090' -n kiamol-ch14-monitoring

# browse to the UI and look at the /targets page
```

Prometheus calls metric collection *scraping*. When you browse to the Prometheus UI you'll see there are no scrape targets, although there is a category called `test-pods` which lists zero targets - figure 14.2 shows my output. The `test-pods` name comes from the Prometheus configuration you deployed - in a ConfigMap which the Pod reads from.

Prometheus runs in a custom monitoring namespace
- it's a service shared by all apps, like logging.

```
PS>cd ch14
PS>
PS>kubectl apply -f prometheus/
namespace/kiamol-ch14-monitoring created
configmap/prometheus-config created
service/prometheus created
deployment.apps/prometheus created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=prometheus -n kiamol-ch14-monitoring
pod/prometheus-6554c8c696-c4hlc condition met
PS>
PS>kubectl get svc prometheus -o jsonpath='http://{.status.loadBalancer.ingress[0].*}:9090' -n kiamol-ch14-monitoring
http://localhost:9090
PS>
PS>
```

Targets

All	Unhealthy				
test-pods (0/0 up)	show less				
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error

Targets are the list of components Prometheus will scrape metrics from. This configuration looks for Pods in a test namespace which doesn't exist, so there are no targets yet.

Figure 14.2 No targets yet, but Prometheus will keep checking the Kubernetes API for new Pods

Configuring Prometheus to find targets in Kubernetes is fairly straightforward, although the terminology is confusing at first. Prometheus uses *jobs* to define a related set of targets to scrape, which could be multiple components of an application. The scrape configuration can be as simple as a static list of domain names which Prometheus polls to grab the metrics, or it can use dynamic service discovery. Listing 14.1 shows the beginning of the `test-pods` job configuration, which uses the Kubernetes API for service discovery.

Listing 14.1 - prometheus-config.yaml, scrape configuration with Kubernetes

```

scrape_configs:          # this is YAML inside the ConfigMap
  - job_name: 'test-pods'      # one job which is used for test apps
    kubernetes_sd_configs:   # find targets from the Kubernetes API
      - role: pod             # search for Pods
    relabel_configs:         # and apply these filtering rules
      - source_labels:
          - __meta_kubernetes_namespace
        action: keep           # only include Pods where the namespace
        regex: kiamol-ch14-test # is the test namespace for this chapter

```

It's the `relabel_configs` section which needs explanation. Prometheus stores metrics with *labels*, which are key-value pairs that identify the source system and other relevant information. You'll use labels in queries to select or aggregate metrics, and you can also use them to filter or modify metrics before they get stored in Prometheus. This is *relabelling* and conceptually it's similar to the data pipeline in Fluent Bit - it's your chance to discard data you don't want and reshape the data you do want.

Regular expressions rear their unnecessarily complicated heads in Prometheus too, but it's rare that you need to make changes - the pipeline you set up in the relabelling phase should be generic enough to work for all your apps. The full pipeline in the configuration file applies these rules:

- only include Pods from the namespace `kiamol-ch14-test`
- use the Pod name as the value of the Prometheus `instance` label
- use the app label in the Pod metadata as the value of the Prometheus `job` label
- use optional annotations in the Pod metadata to configure the scrape target

This is a very convention-driven approach - as long as your apps are modelled to suit the rules, they'll automatically be picked up as monitoring targets. Prometheus uses the rules to find Pods which match, and for each target it collects metrics by making an HTTP GET request to the `/metrics` path. Prometheus needs to know which network port to use, so the Pod spec needs to explicitly include the container port - but that's a good practice anyway because it helps to document your application's setup. Let's deploy a simple app to the test namespace and see what Prometheus does with it.

TRY IT NOW Deploy the `timecheck` application into the test namespace. The spec matches all the Prometheus `scrape` rules, so the new Pod should be found and added as a `scrape` target.

create the test namespace and the timecheck Deployment:

```

kubectl apply -f timecheck/
# wait for the app to start:
kubectl wait --for=condition=ContainersReady pod -l app=timecheck -n kiamol-ch14-test

# refresh the target list in the Prometheus UI and confirm the
# timecheck Pod is listed, then browse to the /graph page, select
# timecheck_total from the dropdown list and click Execute

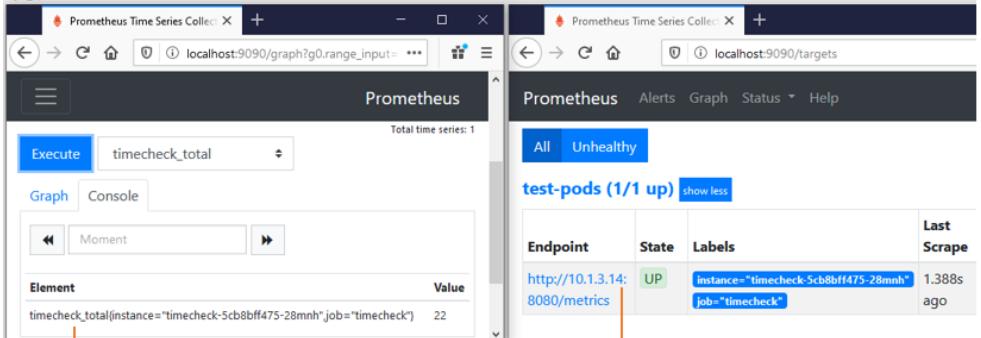
```

My output is in figure 14.3, where I've opened two browser windows so you can see what happened when the app was deployed. Prometheus saw the timecheck Pod being created, and it matched all the rules in the relabel stage, so it got added as a target. The Prometheus configuration is set to scrape targets every 30 seconds. The timecheck app has a /metrics endpoint which returns a count for how many timecheck logs it has written. When I queried that metric in Prometheus, the app had written 22 log entries.

The timecheck app gets deployed to the test namespace and it matches all the Prometheus rules, so it will be added as a scrape target.

```
PS>kubectl apply -f timecheck/
namespace/kiamol-ch14-test created
deployment.apps/timecheck created
```

PS>
 PS>`kubectl wait --for=condition=ContainersReady pod -l app=timecheck -n kiamol-ch14-test
pod/timecheck-5cb8bff475-28mnh condition met`
 PS>



Endpoint	State	Labels	Last Scrape
http://10.1.3.14:8080/metrics	UP	instance="timecheck-5cb8bff475-28mnh" job="timecheck"	1.388s ago

The Pod is listed as a target - this is the Pod's IP address, the app doesn't have a Service.

The graph page lists all the metrics Prometheus has collected. timecheck_total is a counter which the timecheck app records - this is the latest value.

Figure 14.3 Deploying an app to the test namespace - Prometheus finds it and starts collecting metrics

There are two very important things to realize here: the application itself needs to provide the metrics as Prometheus is just a collector; and those metrics represent the activity for one instance of the application. The timecheck app isn't a web application, it's just a background process so there's no Service directing traffic to it. Prometheus gets the Pod IP address when it queries the Kubernetes API and it makes the HTTP request directly to the Pod. You can configure Prometheus to query Services too, but then you'd get a target which is a load-balancer across multiple Pods and you want Prometheus to scrape each Pod independently.

You'll use the metrics in Prometheus to power dashboards showing the overall health of your apps, and you may aggregate across all the Pods to get the headline values - but you

need to be able to drill down too, to see if there are differences between the Pods. That will help you identify if some instances are performing badly, and that will feed back into your health checks. We can scale up the timecheck app to see the importance of collecting at the individual Pod level.

TRY IT NOW Add another replica to the timecheck app. It's a new Pod that matches the Prometheus rules, so it will be discovered and added as another scrape target.

```
# scale the Deployment to add another Pod:  
kubectl scale deploy/timecheck --replicas 2 -n kiamol-ch14-test  
  
# wait for the new Pod to spin up:  
kubectl wait --for=condition=ContainersReady pod -l app=timecheck -n kiamol-ch14-test  
  
# back in Prometheus check the target list and in the graph page  
# execute queries for timecheck_total and dotnet_total_memory_bytes
```

You'll see in this exercise that Prometheus finds the new Pod and starts scraping it. Both Pods record the same metrics and the Pod name is set as a label on each metric. The query for the `timecheck_total` metric now returns two results - one for each Pod, and you can see in figure 14.4 that one Pod has done a lot more work than the other.

Adds another replica to the Deployment - this is a new Pod which matches the Prometheus rules, so it gets picked up as a target.

```
PS>kubectl scale deploy/timecheck --replicas 2 -n kiamol-ch14-test
deployment.apps/timecheck scaled
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=timecheck -n kiamol-ch14
-test
pod/timecheck-5cb8bff475-28mnh condition met
pod/timecheck-5cb8bff475-rz6qn condition met
PS>
```

Endpoint	State	Labels	Last Scrape
http://10.1.3.14:8080/metrics	UP	instance="timecheck-5cb8bff475-28mnh", job="timecheck"	26.23s ago
http://10.1.3.15:8080/metrics	UP	instance="timecheck-5cb8bff475-rz6qn", job="timecheck"	2.983s ago

The new Pod is recording the same set of metrics, but it's a separate instance of the app so it will have its own values.

The query shows two results - one for each Pod. The Pod name is included in the metric's instance label.

Figure 14.4 Every instance records its own metrics so you need to collect from every Pod

The timecheck counter is a metric which is explicitly captured in the application code. Most languages have a Prometheus client library which you can plug into your build. The libraries let you capture application-specific details like this, and they also collect generic information about the application runtime. This is a .NET application and the Prometheus client library records runtime details like the amount of memory and CPU in use, and the number of threads running. In the next section we'll run a distributed application where every component exposes Prometheus metrics, and we'll see how useful an application dashboard is when it includes runtime performance as well as application details.

14.2 Monitoring apps built with Prometheus client libraries

Appendix 2 walks through adding metrics to an app which shows a picture from NASA's Astronomy Photo of the Day (APOD) service. The components of that app are in Java, Go and Node.js and they all use a Prometheus client library to expose runtime and application metrics. This chapter includes Kubernetes manifests for the app which deploy to the test namespace, so all the application Pods will be discovered by Prometheus.

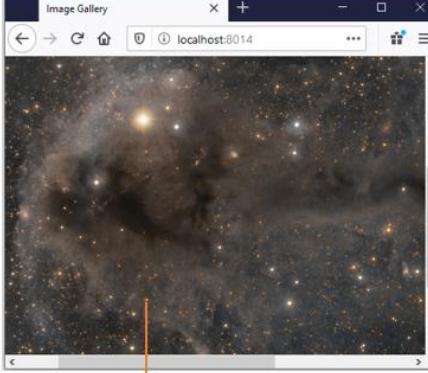
TRY IT NOW Deploy the APOD app to the test namespace and confirm that the three components of the app get added as Prometheus targets.

```
# deploy the app:  
kubectl apply -f apod/  
  
# wait for the main component to start:  
kubectl wait --for=condition=ContainersReady pod -l app=apod-api -n kiamol-ch14-test  
  
# get the app URL:  
kubectl get svc apod-web -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8014' -n  
kiamol-ch14-test  
  
# browse to the app and then refresh the Prometheus targets
```

You can see my output in figure 14.5 with a very pleasant image of something called Lynds Dark Nebula 1251. The application is working as expected, and Prometheus has discovered all the new Pods. Within 30 seconds of deploying the app you should see that the state of all the new targets is up, which means Prometheus has successfully scraped them.

There are multiple components in the APOD app written in different languages. They all use a Prometheus client library to record metrics.

```
PS>kubectl apply -f apod/
service/apod-api created
deployment.apps/apod-api created
service/apod-log created
deployment.apps/apod-log created
service/apod-web created
deployment.apps/apod-web created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=apod-api -n kiamol-ch14-test
pod/apod-api-75fcdb9d6-7c9vz condition met
PS>
PS>kubectl get svc apod-web -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8014' -n kiamol-ch14-test
http://localhost:8014
```



Endpoint	State	Labels
http://10.1.3.16:80/actuator/prometheus	UP	instance="apod-api-75fcdb9d6-7c9vz"; job="apod-api"
http://10.1.3.17:80/metrics	UP	instance="apod-log-9b5cdcb9-6hdq"; job="apod-log"
http://10.1.3.18:80/metrics	UP	instance="apod-web-7cf794886-pt5j"; job="apod-web"
http://10.1.3.14:8080/metrics	UP	instance="timecheck-5e8bf1475-28mesh"; job="timecheck"
http://10.1.3.15:8080/metrics	UP	instance="timecheck-5e8bf1475-28mesh"; job="timecheck"

Lynds Dark Nebula 1251. Very nice.

The new Pods have been added as targets - the job label copies the value of the Pod's app label.

Figure 14.5 The APOD components all have Services but they are still scraped at the Pod level

I've got another two important things to point out in this exercise. First - the Pod specs all include a container port which states the application container is listening on port 80, and that's how Prometheus finds the target to scrape. The Service for the web UI actually listens on port 8014, but Prometheus goes directly to the Pod port. Second - the API target isn't using the standard `/metrics` path, because the Java client library uses a different path, and I've used an annotation in the Pod spec to state the correct path.

Convention-based discovery is great because it removes a lot of repetitive configuration and the potential for mistakes, but not every app will fit with the conventions. The relabelling pipeline we're using in Prometheus gives us a nice balance. The default values will work for any apps that meet the convention, but any which don't can override the defaults with annotations. Listing 14.2 shows how the override is configured to set the path to the metrics.

Listing 14.2 prometheus-config.yaml, using annotations to override default values

```
- source_labels:    # this is a relabel config in the test-pods job
  - __meta_kubernetes_pod_annotationpresent_prometheus_io_path
  - __meta_kubernetes_pod_annotation_prometheus_io_path

  regex: true;(.*)  # if the Pod has an annotation named prometheus.io/path

  target_label: __metrics_path__  # set the target path from the annotation
```

This is way less complicated than it looks. The rule says: if the Pod has an annotation called `prometheus.io/path`, then take the value of that annotation and use it as the metrics path. Prometheus does it all with labels, so every Pod annotation becomes a label with the name `__meta_kubernetes_pod_annotation_<annotation-name>`, and there's an accompanying label called `__meta_kubernetes_pod_annotationpresent_<annotation-name>` which you can use to check if the annotation exists. Any apps which use a custom metrics path need to add the annotation - listing 14.3 shows that for the APOD API.

Listing 14.3 - api.yaml, the path annotation in the API spec

```
template:          # this is the pod spec in deployment
  metadata:
    labels:
      app: apod-api      # used as the job label in Prometheus
    annotations:
      prometheus.io/path: "/actuator/prometheus"  # sets the metrics path
```

So the complexity is centralized in the Prometheus configuration and it's really easy for application manifests to specify overrides. The relabelling rules aren't so complex when you work with them a little more, and you're usually following exactly the same pattern. The full Prometheus configuration includes similar rules for apps to override the metrics port, and to opt out of scraping altogether.

While you've been reading this Prometheus has been busily scraping the timecheck and APOD apps. If you have a look at the metrics in the Graph page of the Prometheus UI then you'll see around 200 metrics being collected. The UI is great for running queries and quickly seeing the results, but you can't use it to build a dashboard showing all the key metrics for your app in a single screen. For that you can use Grafana, another open-source project in the container ecosystem, which comes recommended by the Prometheus team.

TRY IT NOW Deploy Grafana with ConfigMaps that set up the connection to Prometheus, and include a dashboard for the APOD app.

```
# deploy Grafana in the monitoring namespace:
kubectl apply -f grafana/

# wait for it to start up:
kubectl wait --for=condition=ContainersReady pod -l app=grafana -n kiamol-ch14-monitoring

# get the URL for the dashboard:
kubectl get svc grafana -o
```

```
jsonpath='http://{.status.loadBalancer.ingress[0].*}:3000/d/kb5nhJAZk' -n kiamol-ch14-monitoring

# browse to the URL; login with username kiamol and password kiamol
```

You can see my output in figure 14.6 - the dashboard is tiny, but it gives you an idea of how you can transform raw metrics into an informative view of system activity. Each visualization in the dashboard is powered by a Prometheus query, which Grafana runs in the background. There's a row for each component and that includes a mixture of runtime metrics - processor and memory usage, and application metrics - HTTP requests and cache usage.

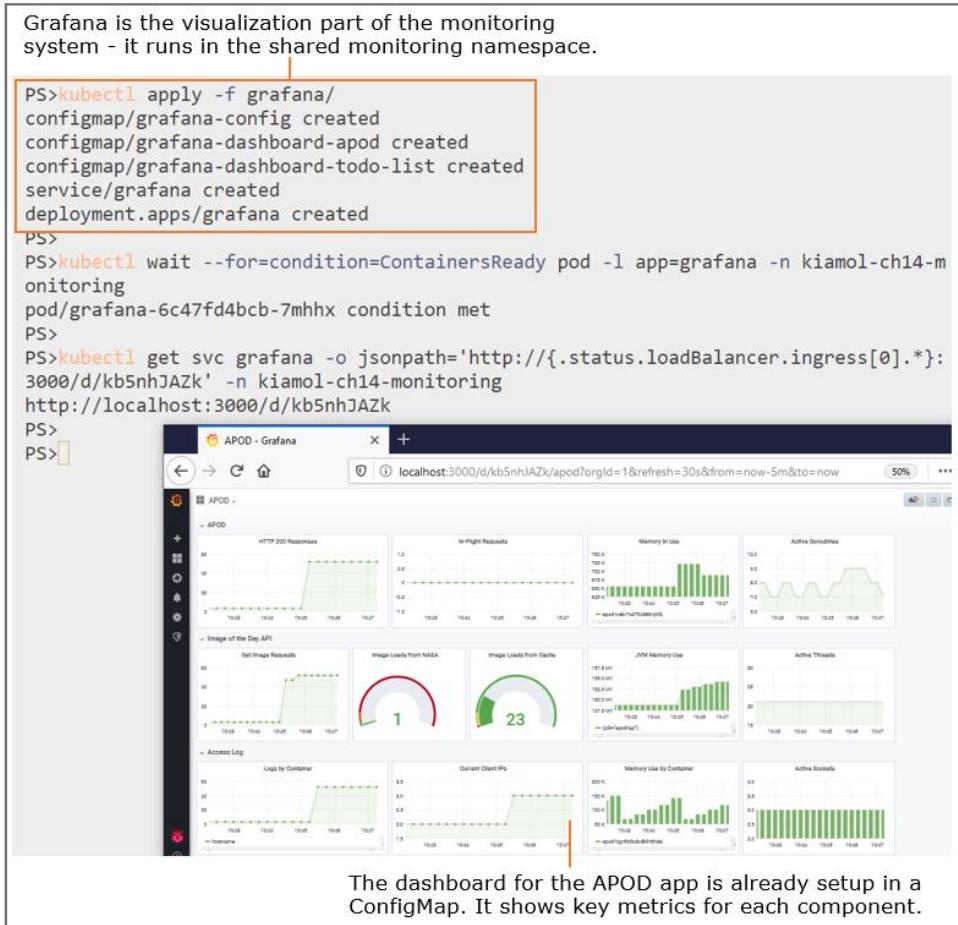


Figure 14.6 Application dashboards give a quick insight into performance - the graphs are all powered from Prometheus metrics

Dashboards like this will be a joint effort that cuts across the organization - the support team will set the requirements of what they need to see, and the application dev and ops teams will make sure the app captures the data and the dashboard shows it. Just like the logging system we looked at in chapter 13, this is a solution built from lightweight open-source components, so devs can run the same monitoring system on their laptops that runs in production. That helps with performance testing and debugging in development and test.

Moving to centralized logging with Prometheus will require development effort, but it can be an incremental process where you start with basic metrics and add to them as teams start to come up with more requirements. I've added Prometheus support to the to-do list app for this chapter, and it took about a dozen lines of code. There's a simple dashboard for the app ready to use in Grafana, so when you deploy the app you'll be able to see the starting point for a dashboard that will improve with future releases.

TRY IT NOW Run the to-do list app with metrics enabled and use the app to produce some metrics. There's already a dashboard in Grafana to visualize the metrics.

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

# wait for it to start:
kubectl wait --for=condition=ContainersReady pod -l app=todo-web -n kiamol-ch14-test

# browse to app & insert an item
# then run some load in with a script - on Windows:
.\loadgen.ps1
# OR on macOS/Linux:
chmod +x ./loadgen.sh && ./loadgen.sh

# get the URL for the new dashboard:
kubectl get svc grafana -o
  jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:3000/d/Eh0VF3iGz' -n kiamol-
ch14-monitoring

# browse to the dashboard
```

There's not much in that dashboard, but it's a lot more information than no dashboard at all. It tells you how much CPU and memory the app is using inside the container, the rate at which tasks are being created and the average response time for HTTP requests. You can see my output in figure 14.7 where I've added some tasks and sent some traffic in with the load generation script.

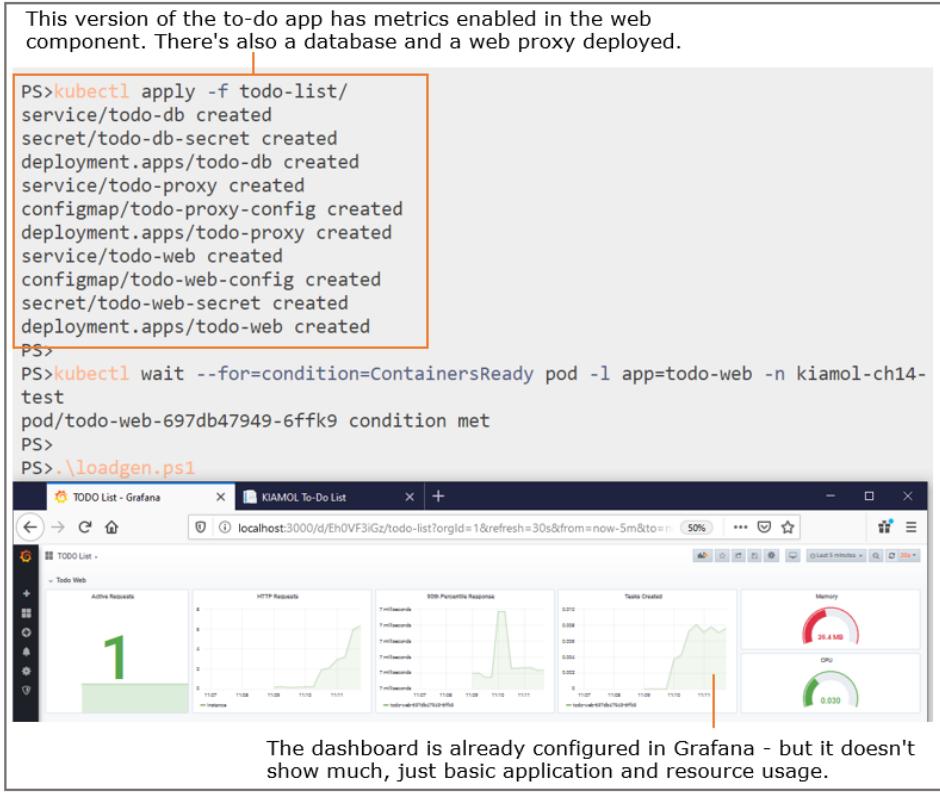


Figure 14.7 A simple dashboard powered by the Prometheus client library and a few lines of code

All those metrics are coming from the to-do application Pod. There are two other components to the app in this release: a Postgres database for storage, and an Nginx proxy. Neither of those components have native support for Prometheus, so they're excluded from the target list - otherwise Prometheus would keep trying to scrape metrics and failing. It's the job of whoever models the application to know that a component doesn't expose metrics, and to specify that it should be excluded. Listing 14.4 shows that's done with a simple annotation.

Listing 14.4 - proxy.yaml, a Pod spec which excludes itself from monitoring

```
template: # this is the Pod spec in the deployment
  metadata:
    labels:
      app: todo-proxy
  annotations: # this annotation is used to
    prometheus.io/scrape: "false" # exclude the target in Prometheus
```

Components don't need to have native support for Prometheus and provide their own metrics endpoint to be included in your monitoring system. Prometheus has its own ecosystem

- as well as client libraries which you can use to add metrics to your own applications, there are a whole set of exporters which can extract and publish metrics for third-party applications. We can use exporters to add the missing metrics for the proxy and database components.

14.3 Monitoring third-party apps with metrics exporters

Most applications record metrics in some way, but older apps won't collect and expose them in Prometheus format. Exporters are separate applications which understand how the target app does its monitoring, and can convert those metrics to Prometheus format. Kubernetes provides the perfect way to run an exporter alongside every instance of an application using a sidecar container - this is the adapter pattern we covered in chapter 7.

Nginx and Postgres both have exporters available which we can run as sidecars to improve the monitoring dashboard for the to-do app. The Nginx exporter reads from a status page on the Nginx server and converts the data to Prometheus format. Remember that all the containers in a Pod share the network namespace, so the exporter container can access the Nginx container at the `localhost` address. The exporter provides its own HTTP endpoint for metrics on a custom port, so the full Pod spec includes the sidecar container and an annotation to specify the metrics port - listing 14.5 shows the key parts.

Listing 14.5 - proxy-with-exporter.yaml, adding a metrics exporter container

```
template:                      # Pod spec in the deployment
  metadata:
    labels:
      app: todo-proxy
    annotations:           # the exclusion annotation is gone
      prometheus.io/port: "9113"   # this specifies the metrics port
  spec:
    containers:
      - name: nginx
        # ... nginx spec is unchanged

      - name: exporter          # the exporter is a sidecar
        image: nginx/nginx-prometheus-exporter:0.8.0
        ports:
          - name: metrics
            containerPort: 9113    # it specifies the metrics port
        args:
          # and loads metrics from Nginx
          - -nginx.scrape-uri=http://localhost/stub_status
```

The scrape exclusion has been removed, so when you deploy this update Prometheus will scrape the Nginx Pod on port 9113, where the exporter is listening. All the Nginx metrics will be stored by Prometheus, and the Grafana dashboard can be updated to add a row for the proxy. We're not going to get into the Prometheus query language (PromQL) or building Grafana dashboards in this chapter - dashboards can be imported from JSON files, and there's an updated dashboard ready to be deployed.

TRY IT NOW Update the proxy Deployment to add the exporter sidecar, and load an updated dashboard into the Grafana ConfigMap.

```
# add the proxy sidecar:  
kubectl apply -f todo-list/update/proxy-with-exporter.yaml  
  
# wait for it to spin up:  
kubectl wait --for=condition=ContainersReady pod -l app=todo-proxy -n kiamol-ch14-test  
  
# print the logs of the exporter:  
kubectl logs -l app=todo-proxy -n kiamol-ch14-test -c exporter  
  
# update the app dashboard:  
kubectl apply -f grafana/update/grafana-dashboard-todo-list-v2.yaml  
  
# restart Grafana to load the new dashboard:  
kubectl rollout restart deploy grafana -n kiamol-ch14-monitoring  
  
# refresh the dashboard - log in with kiamol/kiamol again
```

The Nginx exporter doesn't provide a huge amount of information - but the basic details are there. You can see in figure 14.8 we get the number of HTTP requests and a lower level breakdown of how Nginx handles connection requests. Even with this simple dashboard you can see a correlation between the traffic Nginx is handling and the traffic the web app is handling - which suggests the proxy isn't caching responses and is calling the web app for every request.

This update adds an exporter which makes HTTP metrics available for the Nginx proxy.

The exporter is a sidecar which reads metrics from the Nginx container and exposes them in Prometheus format.

```
PS> kubectl apply -f todo-list/update/proxy-with-exporter.yaml
deployment.apps/todo-proxy configured
```

```
PS>
PS> kubectl wait --for=condition=ContainersReady pod -l app=todo-proxy -n kiamol-ch14-test
pod/todo-proxy-78b48978b8-vmjzd condition met
PS>
```

```
PS> kubectl logs -l app=todo-proxy -n kiamol-ch14-test -c exporter
2020/07/03 10:48:58 Starting NGINX Prometheus Exporter Version= GitCommit=
2020/07/03 10:48:58 Listening on :9113
2020/07/03 10:48:58 NGINX Prometheus Exporter has successfully started
PS>
```

```
PS> kubectl apply -f grafana/update/grafana-dashboard-todo-list-v2.yaml
configmap/grafana-dashboard-todo-list configured
PS>
PS> kubectl rollout restart deploy grafana -n kiamol-ch14-monitoring
deployment.apps/grafana restarted
```

The new dashboard adds a row for the proxy which shows incoming web requests.

Figure 14.8 Collecting proxy metrics with an exporter adds another level of detail to the dashboard

It would be nice to get a bit more information from Nginx - like the breakdown of HTTP status codes in the response - but exporters can only relay the information available from the source system, which isn't much for Nginx. Other exporters provide far more detail but you need to focus your dashboard so it shows key indicators. More than a dozen or so visualizations and the dashboard becomes overwhelming, and if it doesn't convey useful information at a glance then it's not doing a very good job.

There's one more component to add to the to-do list dashboard: the Postgres database. Postgres stores all sorts of useful information in tables and functions inside the database, and the exporter runs queries to power its metrics endpoint. The setup for the Postgres exporter follows the same pattern we've seen in Nginx - in this case the sidecar is configured to access Postgres on `localhost`, using the same Kubernetes Secret which the Postgres container uses

for the admin password. There's a final update to the application dashboard to show the key database metrics from the exporter.

TRY IT NOW Update the database Deployment spec, adding the Postgres exporter as a sidecar container. Then update the to-do list dashboard with a new row to show database performance.

```
# add the exporter sidecar to Postgres:  
kubectl apply -f todo-list/update/db-with-exporter.yaml  
  
# wait for the new Pod to start:  
kubectl wait --for=condition=ContainersReady pod -l app=todo-db -n kiamol-ch14-test  
  
# print the logs from the exporter:  
kubectl logs -l app=todo-db -n kiamol-ch14-test -c exporter  
  
kubectl apply -f grafana/update/grafana-dashboard-todo-list-v3.yaml  
kubectl rollout restart deploy grafana -n kiamol-ch14-monitoring
```

I've zoomed out and scrolled down in figure 14.9 so you can see the new visualizations, but the whole dashboard is a joy to behold in full-screen mode. A single page shows you how much traffic is coming to the proxy, how hard the app is working and what users are actually doing, and what's happening inside the database. You can get the same level of detail in your own apps with client libraries and exporters, and you're looking at just a few days' effort.

Adds an exporter sidecar for database metrics to the Postgres Pod.

The exporter shares credentials with the Postgres container so it can run queries to populate the metrics endpoint (I've abbreviated the logs).

```
PS> kubectl apply -f todo-list/update/db-with-exporter.yaml
deployment.apps/todo-db configured
```

```
PS>
PS> kubectl wait --for=condition=ContainersReady pod -l app=todo-db -n kiamol-ch14-test
pod/todo-db-97d896688-zb4fx condition met
```

```
PS>
PS> kubectl logs -l app=todo-db -n kiamol-ch14-test -c exporter
time="2020-07-03T11:06:08Z" level=info msg="Starting Server: :9187" source="postgres_exporter.go:1672"
```

```
PS>
PS> kubectl apply -f grafana/update/grafana-dashboard-todo-list-v3.yaml
configmap/grafana-dashboard-todo-list configured
```

```
PS>
PS> kubectl rollout restart deploy grafana -n kiamol-ch14-monitoring
deployment.apps/grafana restarted
```

The new dashboard has a database row which shows key metrics, like the amount of data read, and the number of open connections and transactions.

Figure 14.9 The database exporter records metrics about data activity which add detail to the dashboard

Exporters are there to add metrics to apps which don't have Prometheus support. If your goal is to move a set of existing applications onto Kubernetes then you may not have the luxury of a development team to add custom metrics. For those apps you can take to the extreme the approach that some monitoring is better than nothing, using the Prometheus Blackbox exporter.

The Blackbox exporter can run in a sidecar and make TCP or HTTP requests to your application container, and provide a basic metrics endpoint to say whether or not the application is up. It's a similar approach to adding container probes to your Pod spec, except that the Blackbox exporter is for information only. You can run a dashboard to show the status of an app if it isn't a good fit for Kubernetes' self-healing mechanisms, like the random number API we've used in this book.

TRY IT NOW Deploy the random number API with a Blackbox exporter, and the simplest possible Grafana dashboard. You can break the API by using it repeatedly and then reset it so it works again, and the dashboard tracks the status.

```
# deploy the API to the test namespace:  
kubectl apply -f numbers/  
  
# add the new dashboard to Grafana:  
kubectl apply -f grafana/update/numbers-api/  
  
# get the URL for the API:  
kubectl get svc numbers-api -o jsonpath='{.status.loadBalancer.ingress[0].*:8016/rng}' -n kiamol-ch14-test  
  
# use the API by visiting the /rng URL - it will break  
# after three calls; then visit /reset to fix it  
  
# get the dashboard URL and load it in Grafana:  
kubectl get svc grafana -o jsonpath='{.status.loadBalancer.ingress[0].*:3000/d/Tb6isdMMk}' -n kiamol-ch14-  
monitoring
```

The random number API doesn't have Prometheus support but running the Blackbox exporter as a sidecar container gives a very basic insight into the application status. Figure 14.10 shows a dashboard which is mostly empty, but the two visualizations show whether the app is healthy or not, and the historical trend of the status as the app flips between unhealthy and being reset.

The random number API has no Prometheus support, but it's deployed here with a simple Blackbox exporter which reports on the HTTP status of the API container.

```
PS> kubectl apply -f numbers/
service/numbers-api created
deployment.apps/numbers-api created
PS>
PS> kubectl apply -f grafana/update/numbers-api/
configmap/grafana-dashboard-numbers-api created
deployment.apps/grafana configured
PS>
PS> kubectl get svc numbers-api -o jsonpath='{.status.loadBalancer.ingress[0].*:8016/rng}' -n kiamol-ch14-test
# app - http://localhost:8016/rng
PS>
PS> kubectl get svc grafana -o jsonpath='{.status.loadBalancer.ingress[0].*:3000/d/Tb6isdMMk}' -n kiamol-ch14-monitoring
# dashboard - http://localhost:3000/d/Tb6isdMMk
```

The API breaks if you use it too much, and then you can reset it. The status in Grafana shows the app is currently broken, but has had periods of working successfully.

Figure 14.10 Even a simple dashboard is useful - this shows the current and historical status of the API

The Pod spec for the random number API follows a very similar pattern to Nginx and Postgres in the to-do app - the Blackbox exporter is configured as an additional container and specifies the port where metrics are exposed. The Pod annotations customize the path to the metrics URL, so when Prometheus scrapes metrics from the sidcar it calls the Blackbox exporter which checks the API is responding to HTTP requests.

Now we have dashboards for three different apps which all have different levels of detail, because the application components aren't consistent with the data they collect. But all the components have something in common - they're all running in containers on Kubernetes. In the next section you'll learn how to get the next level of detail, configuring Prometheus to collect platform metrics from the cluster.

14.4 Monitoring containers and Kubernetes objects

Prometheus integrates with Kubernetes for service discovery, but it doesn't collect any metrics from the API. You can get metrics about Kubernetes objects and container activity from two additional components: *cAdvisor*, a Google open-source project, and *kube-state-metrics* which

is part of the wider Kubernetes organization on GitHub. Both run as containers in the cluster but they collect data from different sources. cAdvisor collects metrics from the container runtime, so it runs as a DaemonSet with a Pod on each node to report on that node's containers. kube-state-metrics queries the Kubernetes API so it can run as a Deployment with a single replica on any node.

TRY IT NOW Deploy the metric collectors for cAdvisor and kube-state-metrics and update the Prometheus configuration to include them as scrape targets.

```
# deploy cAdvisor and kube-state-metrics:
kubectl apply -f kube/

# wait for cAdvisor to start:
kubectl wait --for=condition=ContainersReady pod -l app=cadvisor -n kube-system

# update the Prometheus config:
kubectl apply -f prometheus/update/prometheus-config-kube.yaml

# wait for the ConfigMap to update in the Pod:
sleep 30

# use an HTTP POST to reload the Prometheus configuration:
curl -X POST $(kubectl get svc prometheus -o
    jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:9090/-/reload' -n kiamol-ch14-
    monitoring)

# browse to the Prometheus UI - in the Graph page you'll see
# metrics listed covering containers and Kubernetes objects
```

In that exercise you'll see thousands of new metrics being collected by Prometheus. The raw data includes the compute resources used by every container and the status of every Pod. My output is in figure 14.11. When you run this exercise you can check the Targets page in the Prometheus UI to confirm that the new targets are being scraped. Prometheus doesn't automatically reload configuration, so in the exercise there's a delay to give Kubernetes time to propagate the ConfigMap update, and the Curl command forces a config reload in Prometheus.

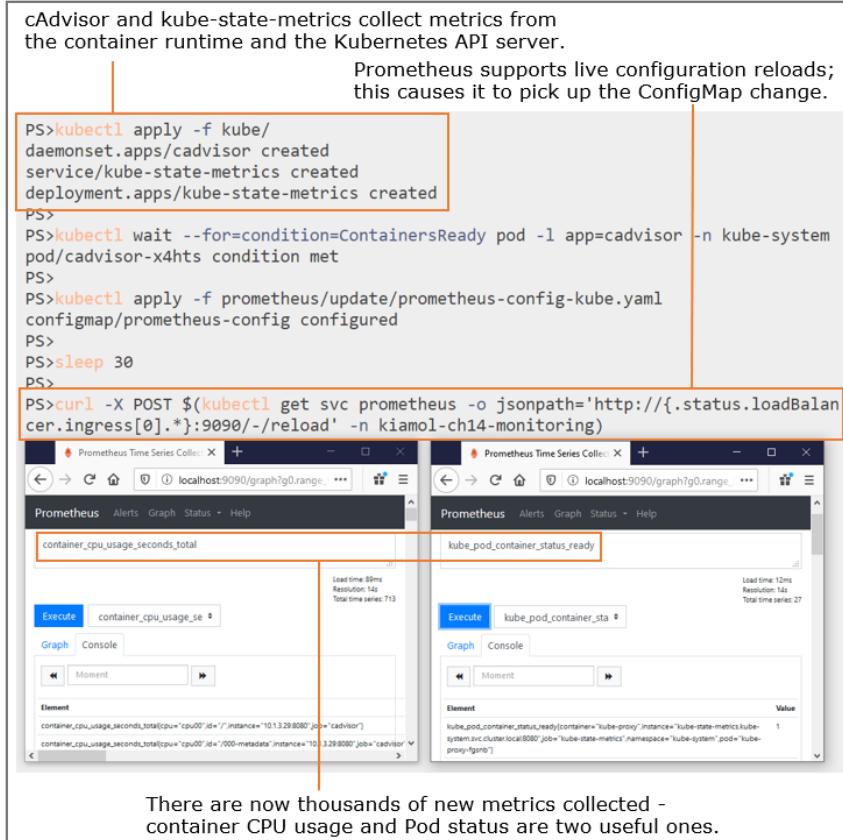


Figure 14.11 New metrics show activity at the cluster and container levels

The updated Prometheus configuration you just deployed includes two new job definitions, shown in listing 14.6. Kube-state-metrics is specified as a static target using the full DNS name of the Service - a single Pod collects all the metrics so there's no load-balancing issue here. cAdvisor uses Kubernetes service discovery to find every Pod in the DaemonSet - which would present one target for each node in a multi-node cluster.

Listing 14.6 - prometheus-config-kube.yaml, new scrape targets in Prometheus

```

- job_name: 'kube-state-metrics'          # Kubernetes metrics use a
  static_configs:                         # static configuration with DNS
    - targets:
        - kube-state-metrics.kube-system.svc.cluster.local:8080
        - kube-state-metrics.kube-system.svc.cluster.local:8081

- job_name: 'cadvisor'                   # container metrics use
  kubernetes_sd_configs:                 # Kubernetes service discovery
    - role: pod                          # to find all the DaemonSet

```

```

relabel_configs:                                # Pods, by namespace and label
  - source_labels:
    - __meta_kubernetes_namespace
    - __meta_kubernetes_pod_labelpresent_app
    - __meta_kubernetes_pod_label_app
    action: keep
    regex: kube-system;true;cadvisor

```

Now we have the opposite problem from the random number dashboard - there's far too much information in the new metrics, so the platform dashboard will need to be highly selective if it's going to be useful. I have a sample dashboard prepared which is a good starter. It includes current resource usage and all available resource quantities for the cluster, together with some high-level breakdowns by namespace and warning indicators for the health of the nodes.

TRY IT NOW Deploy a dashboard for key cluster metrics and with an update to Grafana so it loads the new dashboard.

```

# create the dashboard ConfigMap and update Grafana:
kubectl apply -f grafana/update/kube/

# wait for Grafana to load:
kubectl wait --for=condition=ContainersReady pod -l app=grafana -n kiamol-ch14-monitoring

# get the URL for the new dashboard:
kubectl get svc grafana -o
  jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:3000/d/oWe9aYxmk' -n kiamol-
  ch14-monitoring

# browse to the dashboard

```

This is another dashboard which is meant for the big screen, so the screenshot in figure 14.12 doesn't do it justice. When you run the exercise you can examine it more closely - the top row shows memory usage, the middle row is CPU usage and the bottom row shows the status of Pod containers.

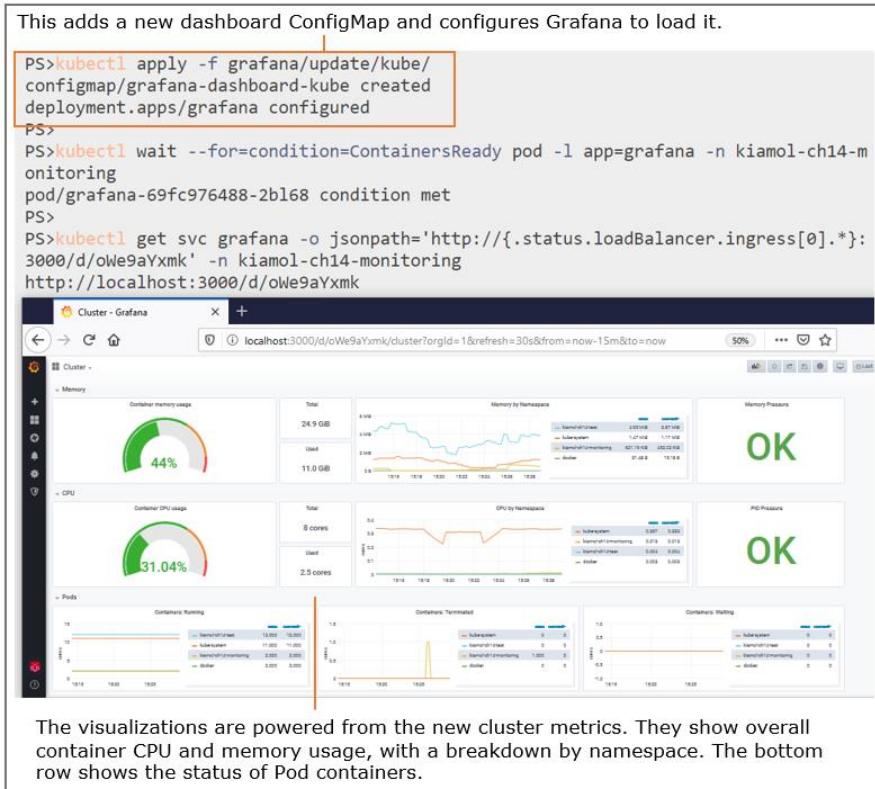


Figure 14.12 Another tiny screenshot - run the exercise in your own cluster to see it full-size

A platform dashboard like this is pretty low-level - it's really just showing you if your cluster is near saturation point. The queries which power this dashboard will be more useful as alerts, warning you if resource usage is getting out of hand. Kubernetes has pressure indicators which are useful there - the memory pressure and process pressure values are shown in the dashboard, and there's also a disk pressure indicator. Those values are significant because if a node comes under compute pressure it can terminate Pod containers. Those would be good metrics to alert on because if you reach that stage you probably need to page someone to come and nurse the cluster back to health.

Platform metrics have another use - adding detail to application dashboards where the app itself doesn't provide detailed enough metrics. The platform dashboard shows compute resource usage aggregated across the whole cluster, but cAdvisor collects it at the container level. It's the same with kube-state-metrics, where you can filter metrics for a specific workload to add platform information to the application dashboard. We'll make a final dashboard update in this chapter, adding details from the platform to the random number app.

TRY IT NOW Update the dashboard for the random number API to add metrics from the platform. This is just a Grafana update; there are no changes to the app itself or to Prometheus.

```
# update the dashboard:
```

```
kubectl apply -f grafana/update/grafana-dashboard-numbers-api-v2.yaml

# restart Grafana so it reloads the dashboard:
kubectl rollout restart deploy grafana -n kiamol-ch14-monitoring

# wait for the new Pod to start:
kubectl wait --for=condition=ContainersReady pod -l app=grafana -n kiamol-ch14-monitoring

# browse back to the random number API dashboard
```

You can see my output in figure 14.13 - it's still a pretty basic dashboard, but at least we now have some detail which could help correlate any issues. If the HTTP status code shows as 503 we can quickly see if the CPU is spiking too - and if the Pod labels contained an application version (which they should), we could identify which release of the app was experiencing the problem.

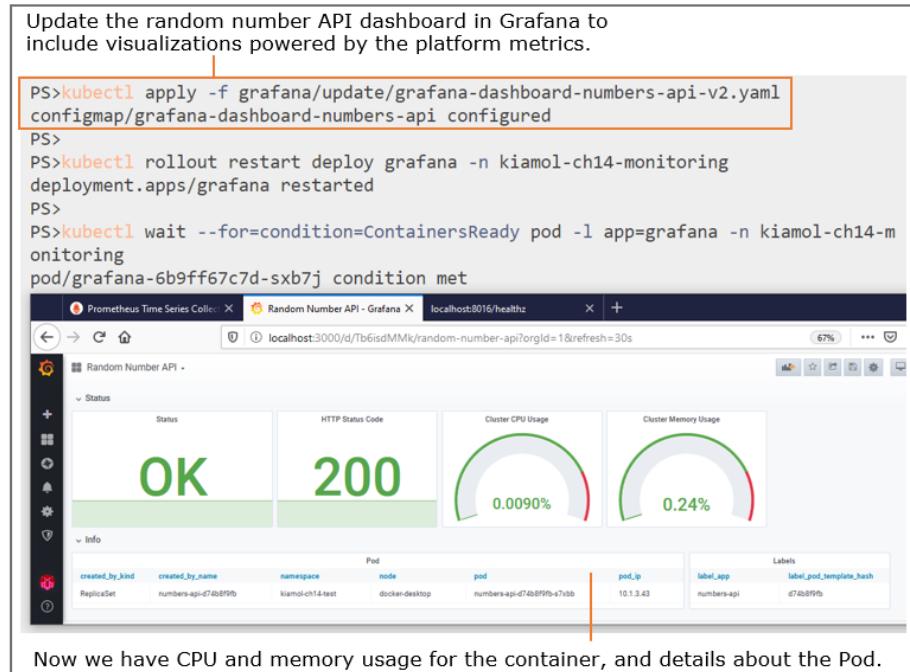


Figure 14.13 Augmenting basic health stats with container and Pod metrics adds correlation

There's a lot more to monitoring than I won't cover here, but now you have a solid grounding in how Kubernetes and Prometheus work together. The main pieces you're missing are collecting metrics at the server level and configuring alerts. Server metrics get you data like disk and network usage - you collect them by running exporters directly on the nodes (using the Node Exporter for Linux servers and the Windows Exporter for Windows servers), and you use service discovery to add the nodes as scrape targets. And Prometheus has a sophisticated alerting system which uses PromQL queries to define alerting rules. You configure alerts so that when rules get triggered Prometheus will send emails, create Slack messages or send a notification through PagerDuty.

We'll wrap up the chapter looking at the full architecture of Prometheus in Kubernetes, and digging in to which pieces need custom work and where the effort needs to go.

14.5 Understanding the investment you make in monitoring

When you step outside of core Kubernetes and into the ecosystem, you need to understand whether the project you take a dependency on will still exist in five years, or one year - or by the time the chapter you're writing makes it to the printing press. I've been careful in this book to only include ecosystem components which are open-source and heavily-used and have an established history and governance model. The monitoring architecture in figure 14.14 uses components which all meet those criteria.

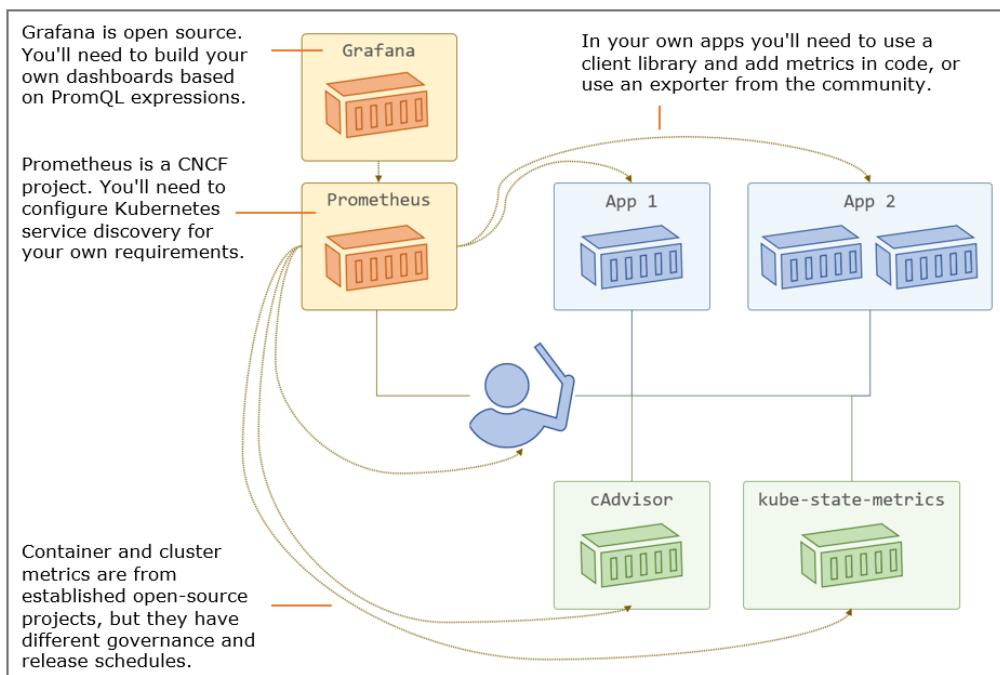


Figure 14.14 Monitoring doesn't come for free - it needs development and dependencies on OSS projects

I make that point because the move to Prometheus will involve development work - you need to record interesting metrics for your applications to make your dashboards truly useful. You should feel confident about making that investment because Prometheus is the most popular tool for monitoring containerized applications, and the project was the second to graduate in the CNCF - after Kubernetes itself. There's also work underway to take the Prometheus metric format into an open standard (called OpenMetrics), so other tools will be able to read application metrics exposed in the Prometheus format.

What you include in those metrics will depend on the nature of your applications, but a good general approach is to follow the guidelines from Google's Site Reliability Engineering practice. It's usually pretty simple to add the four *golden signals* to your app metrics - latency, traffic, errors and saturation (Appendix 2 walks through how those look in Prometheus). But the real value comes when you think about application performance from the user experience perspective. A graph that shows heavy disk usage in your database doesn't tell you much - but if you can see that a high percentage of users don't complete a purchase because your website's checkout page takes too long to load, that's worth knowing.

That's all for monitoring now, so we can clear down the cluster ready for the lab.

TRY IT NOW Delete the namespaces for this chapter, and the objects created in the system namespace.

```
kubectl delete ns -l kiamol=ch14
kubectl delete all -n kube-system -l kiamol=ch14
```

14.6 Lab

Another investigative lab for this chapter. In the lab folder there's a set of manifests for a slightly simpler deployment of Prometheus, and a basic deployment of Elasticsearch. The goal is to run Elasticsearch with metrics flowing into Prometheus. Here's the detail:

- Elasticsearch doesn't provide its own metrics, so you'll need to find a component which does that for you
- The Prometheus configuration will tell you which namespace you need to use for Elasticsearch, and the annotation you need for the metrics path
- You should include a version label in your Elasticsearch Pod spec, so Prometheus will pick that up and add it to the metric labels

You'll need to hunt around the documentation for Prometheus to get started, and that should show you the way. My solution is up on GitHub for you to check in the usual place:

<https://github.com/sixeyed/kiamol/blob/master/ch14/lab/README.md>

15

Managing incoming traffic with Ingress

Services bring network traffic into Kubernetes and you can have multiple LoadBalancer Services with different public IP addresses to make your web apps available to the world. But that's a management headache because it means allocating a new IP address for every application and mapping addresses to apps with your DNS provider. Getting traffic to the right app is a routing problem and you can manage it inside Kubernetes instead, using *ingress*. Ingress uses a set of rules to map domain names and request paths to applications, so you can have a single IP address for your whole cluster and route all traffic internally.

Routing by domain name is an old problem which has usually been solved with a *reverse proxy*, and Kubernetes uses a pluggable architecture for ingress - you define the routing rules as standard resources and you deploy your choice of reverse proxy to receive traffic and act on the rules. All the major reverse proxies have Kubernetes support, along with a new species of container-aware reverse proxy. They all have different capabilities and working models, and in this chapter you'll learn how you can use ingress to host multiple apps in your cluster with two of the most popular: Nginx and Traefik.

15.1 How Kubernetes routes traffic with ingress

We've used Nginx as a reverse proxy several times in this book already (17, by my count) but we've always used it for one application at a time - we had a reverse proxy to cache responses from the Pi app in chapter 6, and another for the random number API in chapter 13. Ingress moves the reverse proxy into a central role, running it as a component called the *ingress controller*, but the approach is the same: the proxy receives external traffic from a LoadBalancer Service, and it fetches content from apps using ClusterIP Services. Figure 15.1 shows the architecture.

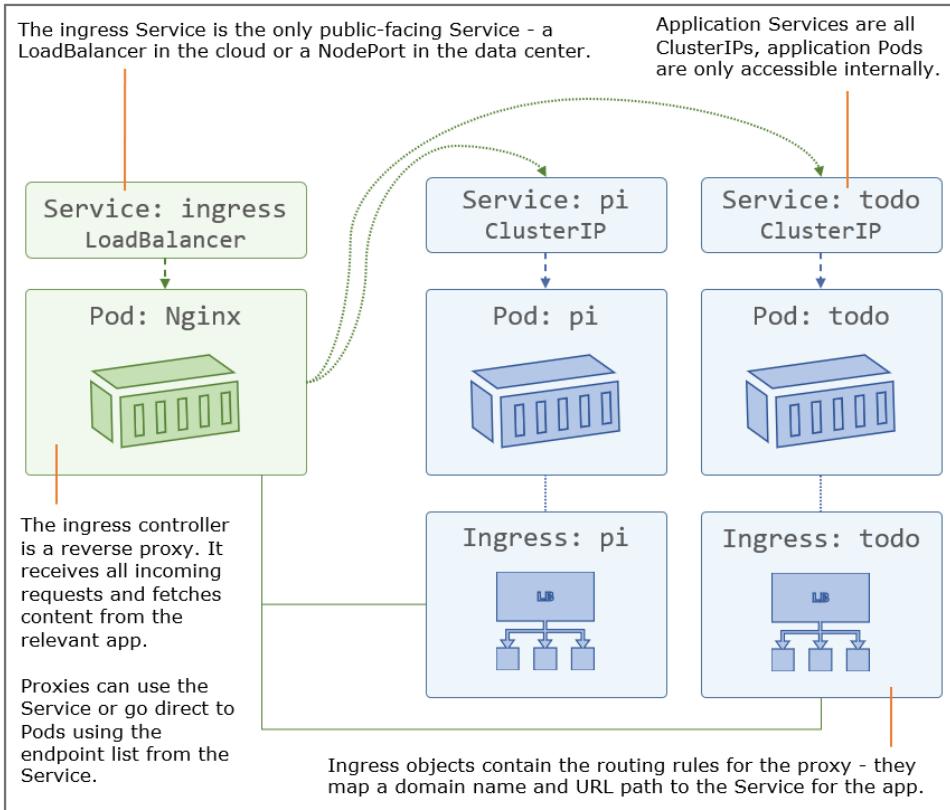


Figure 15.1 Ingress controllers are the entrypoint to the cluster, routing traffic based on Ingress rules

The important thing in that diagram is the ingress controller, which is the pluggable reverse proxy - it could be one of a dozen options including Nginx, HAProxy, Contour and Traefik. The Ingress object stores routing rules in a generic way, and the controller feeds those rules into the proxy. Proxies have different feature sets and the Ingress spec doesn't attempt to model every possible option, so controllers add support for those features using annotations. You'll learn in this chapter that the core functionality of routing and HTTPS support is simple to work with, but the complexity is in the ingress controller deployment and its additional features.

We'll start by running the basic Hello, World web app from way back in chapter 2, keeping it as an internal component with a ClusterIP Service and using the Nginx ingress controller to route traffic.

TRY IT NOW Run the Hello, World app and confirm that it's only accessible inside the cluster, or externally using a port-forward in Kubectl.

```
# switch to this chapter's folder:
cd ch15

# deploy the web app:
kubectl apply -f hello-kiamol/

# confirm the Service is internal to the cluster:
kubectl get svc hello-kiamol

# start a port-forward to the app:
kubectl port-forward svc/hello-kiamol 8015:80

# browse to http://localhost:8015
# then Ctrl-C/Cmd-C to exit the port-forward
```

There's nothing new in the Deployment or Service specs for that application - no special labels or annotations, no new fields you haven't already worked with. You can see in figure 15.2 that the Service has no external IP address, and I can only access the app while I have a port-forward running.

This application spec deploys a ClusterIP Service which makes the app accessible to Pods but not the outside world.

```
PS>cd ch15
PS>
PS>kubectl apply -f hello-kiamol/
deployment.apps/hello-kiamol created
service/hello-kiamol created
PS>
PS>kubectl get svc hello-kiamol
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
hello-kiamol   ClusterIP   10.105.138.207   <none>        80/TCP      6s
PS>
PS>kubectl port-forward svc/hello-kiamol 8015:80
Forwarding from 127.0.0.1:8015 -> 80
Forwarding from [::1]:8015 -> 80
Handling connection for 8015
[...]

```

You can only browse to the app while you have a port-forward running in Kubectl.

Figure 15.2 ClusterIP Services make an app available internally - it can go public with Ingress

To make the app available using Ingress rules, we need an ingress controller. Controllers manage other objects - you know that Deployments manage ReplicaSets and ReplicaSets manage Pods. Ingress controllers are slightly different; they run in standard Pods and monitor Ingress objects, when they see any changes they update the rules in the proxy. We'll start with the Nginx ingress controller, which is part of the wider Kubernetes project. There's a production-

ready Helm chart for the controller, but I'm using a much simpler deployment - even so, there are a few security components in the manifest which we haven't covered yet, so I won't go through it (there are comments in the YAML if you want to investigate).

TRY IT NOW Deploy the Nginx ingress controller - this uses the standard HTTP and HTTPS ports in the Service, so you need to have ports 80 and 443 available on your machine.

```
# create the Deployment and Service for the Nginx ingress controller:  
kubectl apply -f ingress-nginx/  
  
# confirm the service is publicly available:  
kubectl get svc -n kiamol-ingress-nginx  
  
# get the URL for the proxy:  
kubectl get svc ingress-nginx-controller -o  
  jsonpath='http://{{.status.loadBalancer.ingress[0].*}}' -n kiamol-ingress-nginx  
  
# browse to the URL - you'll get an error
```

When you run that exercise you'll see a 404 error page when you browse. That proves the Service is receiving traffic and directing it to the ingress controller, but there aren't any routing rules yet so Nginx has no content to show and it returns the default not-found page. My output is in figure 15.3, where you can see the Service is using the standard HTTP port.

The ingress controller runs in a Deployment with a LoadBalancer service
- all the other objects are for securing access to the Kubernetes API.

```
PS># make sure ports 80 and 443 aren't in use
PS>
PS>kubectl apply -f ingress-nginx/
namespace/kiamol-ingress-nginx created
configmap/ingress-nginx-controller created
service/ingress-nginx-controller created
deployment.apps/ingress-nginx-controller created
serviceaccount/ingress-nginx created
clusterrole.rbac.authorization.k8s.io/ingress-nginx created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx created
role.rbac.authorization.k8s.io/ingress-nginx created
rolebinding.rbac.authorization.k8s.io/ingress-nginx created
PS>
PS>kubectl get svc -n kiamol-ingress-nginx
NAME                      TYPE      CLUSTER-IP      EXTERNAL-IP   PORT(S)
(Selector)                AGE
ingress-nginx-controller   LoadBalancer   10.109.72.23   localhost     80:3
1504/TCP,443:30864/TCP    8s
PS>
PS>kubectl get svc ingress-nginx-controller -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}' -n kiamol-ingress-nginx
http://localhost
PS>
PS>
```

The ingress Service listens on the standard HTTP and HTTPS ports. The ingress controller runs Nginx which responds to requests - there are no Ingress rules yet, so the proxy returns a 404.

Figure 15.3 Ingress controllers receive incoming traffic but they need routing rules to know what to do with it

Now that we have an application running and an ingress controller, we just need to deploy an Ingress object with routing rules. That will tell the controller which application Service to use for each incoming request. Listing 15.1 shows the simplest rule for an Ingress object - this will route every request coming into the cluster to the Hello, World application.

Listing 15.1 - localhost.yaml, a routing rule for the Hello, World app

```
apiVersion: networking.k8s.io/v1beta1 # beta API versions mean the spec
kind: Ingress                         # isn't final and could change
metadata:
  name: hello-kiamol
spec:
  rules:
    - http:                                # Ingress is only for HTTP traffic
      paths:
        - path: /                           # maps every incoming request
```

```
backend: # to the hello-kiamol Service
  serviceName: hello-kiamol
  servicePort: 80
```

The ingress controller is watching for new and changed Ingress objects, so when you deploy this it will add the rule to the Nginx configuration. In Nginx terms it will set up a proxy server where the hello-kiamol Service is the upstream - the source of the content, and it will serve that content for incoming requests to the root path.

TRY IT NOW Create the Ingress rule which publishes the Hello, World app through the ingress controller.

```
# deploy the rule:
kubectl apply -f hello-kiamol/ingress/localhost.yaml

# confirm the Ingress object is created:
kubectl get ingress

# refresh your browser from the last exercise
```

Well - that was simple. Map a path to the backend Service for the application in an Ingress object and the controller takes care of everything else. My output in figure 15.4 shows the localhost address which previously returned a 404 error now returns the Hello, World app in all its glory.

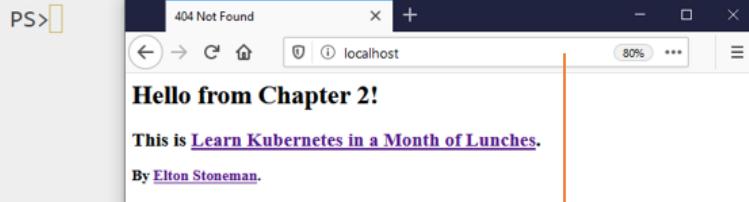
This ingress object matches any host domain name and every path, so all incoming requests will use the same app.

```
PS>kubectl apply -f hello-kiamol/ingress/localhost.yaml
ingress.networking.k8s.io/hello-kiamol created
```

```
PS>
```

NAME	CLASS	HOSTS	ADDRESS	PORTS	AGE
hello-kiamol	<none>	*		80	5s

```
PS>
```



My ingress Service listens on localhost - browsing to any URL will show the Hello, World app.

Figure 15.4 Ingress object rules link the ingress controller to the app Service

Ingress is usually a centralized service in the cluster, like logging and monitoring. An admin team might deploy and manage the ingress controller, while each product team owns the Ingress objects that route traffic to their apps. That creates the potential for collisions - Ingress rules do not have to be unique, and one team's update could end up redirecting all of some other team's traffic to some other app. In practice that doesn't happen because those apps will be hosted at different domains, and the Ingress rules will include a domain name to restrict their scope.

15.2 Routing HTTP traffic with ingress rules

Ingress only works for web traffic - HTTP and HTTPS requests - because it needs to use the route specified in the request to match it to a backend service. There are two parts to the route in an HTTP request: the host and the path. The host is the domain name - like `www.manning.com`, and the path is the location of the resource, like `/dotd` for the Deal of the Day page. Listing 15.2 shows an update to the Hello, World Ingress object which uses a specific host name. Now the routing rules will only apply if the incoming request is for the host `hello.kiamol.local`.

Listing 15.2 - hello.kiamol.local.yaml, specifying a host domain for Ingress rules

```
spec:
  rules:
    - host: hello.kiamol.local          # this restricts the scope of the
      http:                            # rules to a specific domain
        paths:
          - path: /                   # all paths in that domain will
            backend:                 # be fetched from the same service
              serviceName: hello-kiamol
              servicePort: 80
```

When you deploy this you won't be able to access the app because the domain name `hello.kiamol.local` doesn't exist. Web requests normally look up the IP address for a domain name from a public DNS server, but all computers also have their own local list in a *hosts file*. In the next exercise you'll deploy the updated Ingress object and register the domain name in your local hosts file - you'll need admin access in your terminal session for that.

TRY IT NOW Editing the hosts file is restricted - you'll need to use the "Run as Administrator" option for your terminal session in Windows and have scripts enabled with the `Set-ExecutionPolicy` command. Be ready to enter your admin (`sudo`) password in Linux or macOS.

```
# add domain to hosts - on Windows:
./add-to-hosts.ps1 hello.kiamol.local ingress-nginx

# OR on Linux/macOS:
chmod +x add-to-hosts.sh && ./add-to-hosts.sh hello.kiamol.local ingress-nginx

# update the Ingress object, adding the host name:
kubectl apply -f hello-kiamol/ingress/hello.kiamol.local.yaml
```

```
# confirm the update:  
kubectl get ingress  
  
# browse to http://hello.kiamol.local
```

In that exercise the existing Ingress object is updated, so there's still only one routing rule for the ingress controller to map. Now that rule is restricted to an explicit domain name - you can see in figure 15.5 that the request to `hello.kiamol.local` returns the app, and I've also browsed to the ingress controller at `localhost` which returns a 404 because there are no rules for the `localhost` domain.

Adds a line to the local hosts file, so the `hello.kiamol.local` domain name resolved to the IP address for the ingress controller.

The scope of the Ingress rule is restricted to this domain.

```
PS>./add-to-hosts.ps1 hello.kiamol.local ingress-nginx
PS>
PS>kubectl apply -f hello-kiamol/ingress/hello.kiamol.local.yaml
ingress.networking.k8s.io/hello-kiamol configured
PS>
PS>kubectl get ingress
NAME      CLASS    HOSTS          ADDRESS        PORTS   AGE
hello-kiamol <none>  hello.kiamol.local  localhost     80      31m
```

NAME	CLASS	HOSTS	ADDRESS	PORTS	AGE
hello-kiamol	<none>	hello.kiamol.local	localhost	80	31m

hello.kiamol.local/ 404 Not Found

Hello from Chapter 2!

This is [Learn Kubernetes in a Month of Lunches](#).

By [Elton Stoneman](#).

404 Not Found

nginx/1.19.0

Browse to `hello.kiamol.local` and you get the Hello, World app.

But there are no Ingress rules for `localhost` or default rules which apply to all domains, so the ingress Service URL returns a 404.

Figure 15.5 You can publish apps by domain name with Ingress rules and use them locally by editing your hosts file

Routing is an infrastructure-level concern, but like the other shared services we've seen in this section of the book, it runs in lightweight containers so you can use exactly the same setup in development, test and production environments. That lets you run multiple apps in your non-production cluster with friendly domain names, without having to use different ports - the Service for the ingress controller uses the standard HTTP port for every app.

You need to fiddle with your hosts file if you want to run multiple apps with different domains in your lab environment. Typically all the domains will resolve to 127.0.0.1, which is your machine's local address. Organizations might run their own DNS server in a test environment

so anyone can access `hello.kiamol.test` from the company network and it will resolve to the IP address of the test cluster, running in the data center. Then in production the DNS resolution is from a public DNS service, so `hello.kiamol.net` resolves to a Kubernetes cluster running in the cloud.

You can combine host names and paths in Ingress rules to present a consistent set of addresses for your application, although you could be using different components in the backend. You might have a REST API and a website running in separate Pods, and you could use Ingress rules to make the API available on a subdomain (`api.rng.com`) or as a path on the main domain (`rng.com/api`). Listing 15.3 shows Ingress rules for the simple versioned web app from chapter 9 - where both versions of the app are available from one domain.

Listing 15.3 - vweb/ingress.yaml, Ingress rules with host name and paths

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: vweb
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
    - host: vweb.kiamol.local
      http:
        paths:
          - path: /
            backend:
              serviceName: vweb-v2
              servicePort: 80
          - path: /v1
            backend:
              serviceName: vweb-v1
              servicePort: 80
```

Modelling paths adds complexity because you're presenting a fake URL which needs to be modified to match the real URL in the service. In this case the ingress controller will respond to requests for `http://vweb.kiamol.local/v1` and fetch the content from the `vweb-v1` service. But the application doesn't have any content at `/v1`, so the proxy needs to rewrite the incoming URL - that's what the annotation does in listing 15.3, it's a very basic example which ignores the path in the request and always uses the root path in the backend. You can't express URL rewrites with the Ingress spec, so it needs custom support from the ingress controller - a more realistic rewrite rule would use regular expressions to map the requested path to the target path.

We'll deploy this simple version to avoid any regular expressions, and see how the ingress controller uses routing rules to identify the backend service and to modify the request path.

TRY IT NOW Deploy a new app with a new Ingress rule and add a new domain to your hosts file to see the ingress controller serving multiple applications from the same domain.

```
# add the new domain name - on Windows:  
./add-to-hosts.ps1 vweb.kiamol.local ingress-nginx  
  
# OR on Linux/macOS:  
./add-to-hosts.sh vweb.kiamol.local ingress-nginx  
  
# deploy the app, Service and Ingress:  
kubectl apply -f vweb/  
  
# confirm the Ingress domain:  
kubectl get ingress  
  
# browse to http://vweb.kiamol.local  
# and http://vweb.kiamol.local/v1
```

In figure 15.6 you can see that two separate apps are made available at the same domain name, using the request path to route between different components - which are different versions of the application in this exercise.

Adds another domain name to the hosts file, which resolves to the ingress controller IP address.

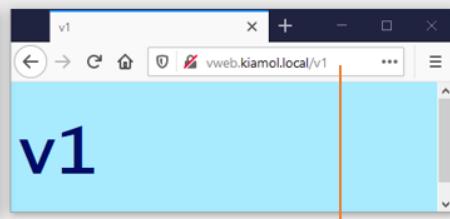
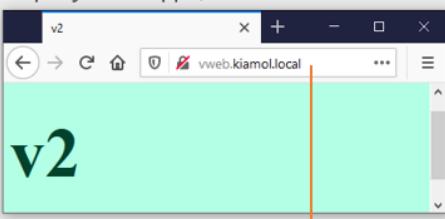
The new Ingress has rules for host domain and URL path.

```
PS> ./add-to-hosts.ps1 vweb.kiamol.local ingress-nginx
```

```
PS>
```

```
PS> kubectl apply -f vweb/
```

```
ingress.networking.k8s.io/vweb created  
service/vweb-v1 created  
deployment.apps/vweb-v1 created  
service/vweb-v2 created  
deployment.apps/vweb-v2 created
```



The root path for the domain is mapped to the v2 application.

The /v1 path for the domain is mapped to the v1 application - the proxy strips the request path and fetches content from the root path.

Figure 15.6 Ingress routing on host name and path presents multiple apps on the same domain name

Mapping the routing rules is the most complicated part of publishing a new app to your ingress controller, but it does give you a lot of control. The Ingress rules are the public face of your app and you can use them to compose several components, or to restrict access to features. We've

seen in this section of the book that apps work better in Kubernetes if they have health endpoints for container probes and metrics endpoints for Prometheus to scrape - but those shouldn't be publicly available. You can use Ingress to control that, using exact path mappings so only paths which are explicitly listed are available outside of the cluster.

Listing 15.4 shows an example of that for the to-do list app - it's abridged because the downside with this approach is that you need to specify every path you want to publish, so any paths not specified are blocked.

Listing 15.4 - ingress-exact.yaml, using exact path matching to restrict access

```
rules:
  - host: todo.kiamol.local
    http:
      paths:
        - pathType: Exact          # exact matching means only the /new
          path: /new               # path is matched - there are other
          backend:                  # rules for the /list and root paths
            serviceName: todo-web
            servicePort: 80
        - pathType: Prefix         # prefix matching means any path that
          path: /static            # starts with /static will be mapped
          backend:                  # include subpaths like /static/app.css
            serviceName: todo-web
            servicePort: 80
```

The to-do list app has several paths which shouldn't be available outside of the cluster - as well as `/metrics` there's a `/config` endpoint which lists all the application config and a diagnostics page. None of those paths are included in the new Ingress spec, and we can see that they're effectively blocked when the rules are applied. The `PathType` field is a later addition to the Ingress spec, so your Kubernetes cluster needs to be running at least version 1.18 - otherwise you'll get an error in this exercise.

TRY IT NOW Deploy the to-do list app with an Ingress spec that allows all access and then update it with exact path matching and confirm that the sensitive paths are no longer available.

```
# add a new domain for the app - on Windows:
./add-to-hosts.ps1 todo.kiamol.local ingress-nginx

# OR on Linux/macOS:
./add-to-hosts.sh todo.kiamol.local ingress-nginx

# deploy the app with an Ingress object that allows all paths:
kubectl apply -f todo-list/

# browse http://todo.kiamol.local/metrics

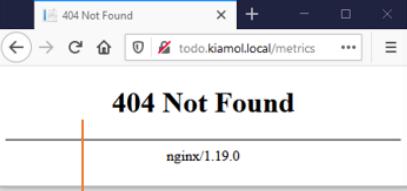
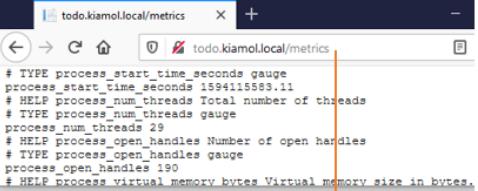
# update the Ingress with exact paths:
kubectl apply -f todo-list/update/ingress-exact.yaml

# browse again - app works but metrics and diagnostics blocked
```

You'll see when you run that exercise that all the sensitive paths are blocked when you deploy the updated Ingress rules - my output is in figure 15.7. It's not a perfect solution but you can extend your ingress controller to show a friendly 404 error page instead of the Nginx default (Docker has a nice example, try <https://www.docker.com/not-real-url>). The app still shows a menu for the diagnostics page because it's not an app setting which removes the page; it's happening earlier in the process.

```
Add another domain to the hosts file.
The initial deployment uses an Ingress that allows all paths.

PS>./add-to-hosts.ps1 todo.kiamol.local ingress-nginx
PS>
PS>kubectl apply -f todo-list/
service/todo-db created
secret/todo-db-secret created
deployment.apps/todo-db created
ingress.networking.k8s.io/todo created
service/todo-web created
configmap/todo-web-config created
secret/todo-web-secret created
deployment.apps/todo-web created
PS>
PS>kubectl apply -f todo-list/update/ingress-exact.yaml
ingress.networking.k8s.io/todo configured



Before the update, the metrics path is publicly accessible - not good as it contains sensitive information about the internals of the app.

After the Ingress update there is no route for the metrics path, so the ingress controller returns the standard 404 page.
```

Figure 15.7 Exact path matching in Ingress rules can be used to block access to features

The separation between Ingress rules and the ingress controller makes it very easy to compare different proxy implementations and see which gives you the combination of features and usability you're happy with. But it comes with a warning because there isn't a strict ingress controller specification, and not every controller implements Ingress rules in the same way. Some controllers ignore the PathType field, so if you're relying on that to build up an access list with exact paths, you may find your site becomes access-all-areas if you switch to a different ingress controller.

Kubernetes does let you run multiple ingress controllers, and in a complex environment you may do that to provide different sets of capabilities for different applications.

15.3 Comparing ingress controllers

Ingress controllers come in two categories - reverse proxies which have been around for a long time and work at the network level, fetching content using host names, and modern proxies which are platform-aware and can integrate with other services (cloud controllers can provision external load balancers). Choosing between them comes down to the feature set and your own technology preferences. If you have an established relationship with Nginx or HAProxy, you can continue that in Kubernetes. Or if you have an established relationship with Nginx or HAProxy you might be glad to try a more lightweight modern option.

Your ingress controller becomes the single public entrypoint for all the apps in your cluster, so it's a good place to centralize common concerns. All controllers support SSL termination, so the proxy provides the security layer and you get HTTPS for all your applications. Most controllers support web application firewalls, so you can provide protection from SQL injection and other common attacks in the proxy layer. Some controllers have special powers - we've already used Nginx as a caching proxy, and you can use it for caching at the ingress level too.

TRY IT NOW Deploy the Pi application using Ingress, then update the Ingress object so the Pi app makes use of the Nginx cache in the ingress controller.

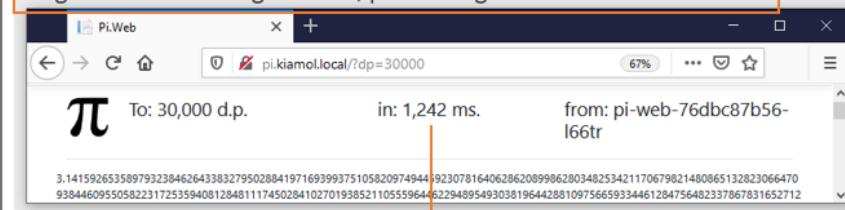
```
# add the Pi app domain to the hosts file - Windows:  
./add-to-hosts.ps1 pi.kiamol.local ingress-nginx  
  
# OR Linux/macOS:  
./add-to-hosts.sh pi.kiamol.local ingress-nginx  
  
# deploy the app and a simple Ingress:  
kubectl apply -f pi/  
  
# browse to http://pi.kiamol.local?dp=30000  
# refresh and confirm the page load takes the same time  
  
# deploy an update to the Ingress to use caching:  
kubectl apply -f pi/update/ingress-with-cache.yaml  
  
# browse to the 30K Pi calculation again - the first  
# load takes a few seconds, but now a refresh will be fast
```

You'll see in this exercise that the ingress controller is a powerful component in the cluster. You can add caching to your app just by specifying new ingress rules - no updates to the application itself, and no new components to manage. The only requirement is that the HTTP responses from your app include the right caching headers, which they should anyway. Figure 15.8 shows my output where the Pi calculation took 1.2 seconds but the response came from the ingress controller's cache, so the page loaded pretty much instantly.

Add the Pi domain to the hosts file.

The updated Ingress uses caching in the ingress controller for content from the Pi app.

```
PS> ./add-to-hosts.ps1 pi.kiamol.local ingress-nginx
PS>
PS>kubectl apply -f pi/
ingress.networking.k8s.io/pi created
service/pi-web created
deployment.apps/pi-web created
PS>
PS>kubectl apply -f pi/update/ingress-with-cache.yaml
ingress.networking.k8s.io/pi configured
```



This response was loaded after the caching update - the original calculation took 1.2 seconds but the response was cached, so repeat requests to the same page load instantly.

Figure 15.8 If your ingress controller supports response caching, that's an easy performance boost

Not every ingress controller provides a response cache, so that's not a specific part of the Ingress spec. Any custom configuration is applied with annotations which the controller picks up. Listing 15.5 shows the metadata for the updated cache setting you applied in the last exercise - if you're familiar with Nginx you'll recognize these as the proxy cache settings you would normally set in the configuration file.

Listing 15.5 - ingress-with-cache.yaml, using the Nginx cache in the ingress controller

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  annotations:           # the ingress controller looks in annotations for
    name: pi              # custom configuration - this adds proxy caching
  annotations:
    nginx.ingress.kubernetes.io/proxy-buffering: "on"
    nginx.ingress.kubernetes.io/configuration-snippet: |
      proxy_cache static-cache;
      proxy_cache_valid 10m;
```

The configuration in an Ingress object applies to all of its rules, but if you need different features for different parts of your app, you can have multiple ingress rules. That's true for the to-do list

app which needs some more help from the ingress controller to work properly at scale. Ingress controllers use load-balancing if there are many Pods for a Service, but the to-do app has some cross-site forgery protection which breaks if the request to create a new item is sent to a different app container from the one which originally rendered the new item page. Lots of apps have a restriction like this, which proxies solve using *sticky sessions*.

Sticky sessions are a mechanism for the ingress controller to send requests from the same end user to the same container, which is often a requirement for older apps where components aren't stateless. It's something to avoid where possible because it limits the cluster's potential for load balancing, so in the to-do list app we want to restrict it to just one page. Figure 15.9 shows the Ingress rules we'll apply to get different features for different parts of the app.

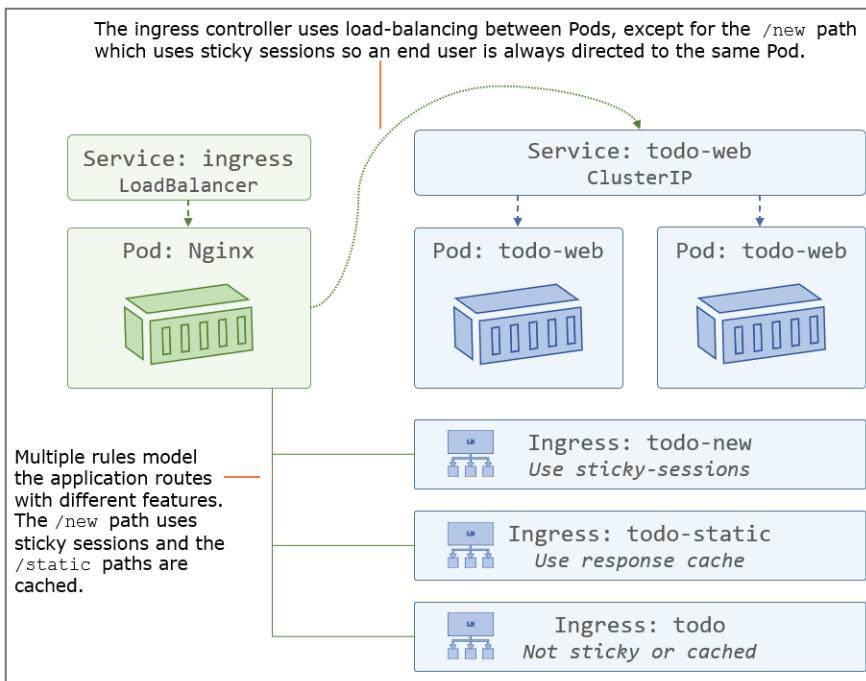


Figure 15.9 A domain can be mapped with multiple Ingress rules, using different proxy features

We can scale up the to-do app now to understand the problem, and then apply the updated Ingress rules to fix it.

TRY IT NOW Scale up the to-do application to confirm that it breaks without sticky sessions - then deploy the updated Ingress rules from figure 15.9 and confirm it all works again.

```
# scale up and the ingress controller load-balances between the Pods:
```

```
kubectl scale deploy/todo-web --replicas 3

# wait for the new Pods to start:
kubectl wait --for=condition=ContainersReady pod -l app=todo-web

# browse to http://todo.kiamol.local/new and add an item
# this will fail and show a 400 error page

# print the application logs to see the issue:
kubectl logs -l app=todo-web --tail 1 --since 60s

# update Ingress to add sticky sessions:
kubectl apply -f todo-list/update/ingress-sticky.yaml

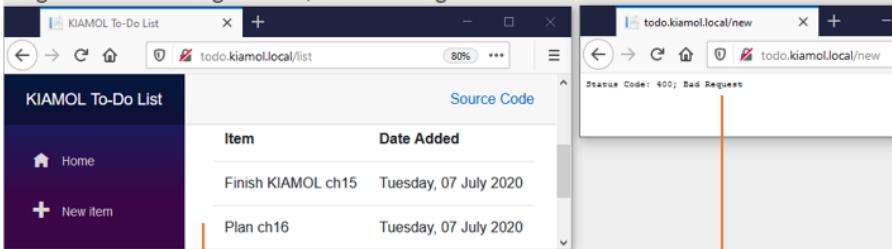
# browse again and add a new item - this time it works
```

You can see my output in figure 15.10, but unless you run the exercise yourself you'll have to take my word for which is the "before" and which is the "after" screenshot. Scaling up the application replicas means requests from the ingress controller get load-balanced, which triggers the anti-forgery error. Applying sticky sessions stops load balancing on the new item path, so a user's requests are always routed to the same Pod and the forgery check passes.

Scaling up causes a problem when you create a new item because of anti-forgery checks in the application.

Users see an unhelpful error and the logs record an anti-forgery message.

```
PS>kubectl scale deploy/todo-web --replicas 3
deployment.apps/todo-web scaled
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=todo-web
pod/todo-web-6d5c557f-2d99k condition met
pod/todo-web-6d5c557f-d8bm5 condition met
pod/todo-web-6d5c557f-tjlvd condition met
PS>
PS>kubectl logs -l app=todo-web --tail 1 --since 60s
at Microsoft.AspNetCore.Antiforgery.DefaultAntiforgery.GetCookieTokenDoe
sNotThrow(HttpContext httpContext)
PS>
PS>kubectl apply -f todo-list/update/ingress-sticky.yaml
ingress.networking.k8s.io/todo-new created
ingress.networking.k8s.io/todo-static created
ingress.networking.k8s.io/todo configured
```



This is the error page caused by the failing forgery check.

After the Ingress update the new-item page uses sticky sessions and the app works again.

Figure 15.10 Proxy features can fix problems as well as improving performance

The Ingress resources for the to-do app use a combination of host, paths and annotations to set all the rules and the features to apply. Behind the scenes the job of the controller is to convert those rules into proxy configuration - which in the case of Nginx means writing a config file. There are lots of optimizations in the controller to minimize the number of file writes and configuration reloads, but as a result the Nginx configuration file is horribly complex. If you opt for the Nginx ingress controller because you have Nginx experience and you'd be comfortable debugging the config file, you're in for an unpleasant surprise.

TRY IT NOW The Nginx configuration is in a file in the ingress controller Pod. Run a command in the Pod to check the size of the file.

```
# run the wc command to see how many lines are in the file:
kubectl exec -n kiamol-ingress-nginx deploy/ingress-nginx-controller -- sh -c 'wc -l
/etc/nginx/nginx.conf'
```

Figure 15.11 shows there are over 1,700 lines in my Nginx configuration file. If you run `cat` instead of `wc` then you'll find the contents are strange, even if you're familiar with Nginx (the controller uses Lua scripts so it can update endpoints without a config reload).

The controller writes the Nginx configuration file to translate Ingress rules for the proxy.

```
PS>kubectl exec -n kiamol-ingress-nginx deploy/ingress-nginx-controller --
sh -c 'wc -l /etc/nginx/nginx.conf'
1766 /etc/nginx/nginx.conf
```

Over 1700 lines in this config file - not going to enjoy debugging that.

Figure 15.11 The generated Nginx configuration file is not made to be human-friendly

The ingress controller owns that complexity, but it's a critical part of your solution and you need to be happy with how you'll troubleshoot and debug the proxy. This is when you might want to consider an alternative ingress controller which is platform-aware and doesn't run from a complex configuration file. We'll look at Traefik in this chapter - it's an open-source proxy which has been gaining popularity since it launched in 2015. Traefik understands containers and it builds its routing list from the platform API, natively supporting Docker and Kubernetes, so there is no config file to maintain.

Kubernetes supports multiple ingress controllers running in a single cluster. They'll be exposed as LoadBalancer services so in production you might have different IP addresses for different ingress controllers, and you'll need to map domains to ingress in your DNS configuration. In our lab environment we'll be back to using different ports - we'll start by deploying Traefik with a custom port for the ingress controller Service.

TRY IT NOW Deploy Traefik as an additional ingress controller in the cluster.

```
# create the Traefik Deployment, Service and security resources:
kubectl apply -f ingress-traefik/

# get the URL for the Traefik UI running in the ingress controller:
kubectl get svc ingress-traefik-controller -o
    jsonpath='http://{{.status.loadBalancer.ingress[0].*.}}:8080' -n kiamol-ingress-
traefik

# browse to the admin UI to see the routes Traefik has mapped
```

You'll see in that exercise that Traefik has an admin UI - it shows you the routing rules the proxy is using, and as traffic passes through it can collect and show performance metrics. It's much easier to work with than the Nginx config file. Figure 15.12 shows there are two *routers* which

are the incoming routes Traefik manages. If you explore the dashboard you'll see those aren't Ingress routes, they're internal routes for Traefik's own dashboard - Traefik has not picked up any of the existing Ingress rules in the cluster.

Running Traefik as an ingress controller - with a Service and Deployment, and security settings for accessing the Kubernetes API.

Traefik runs an admin UI, which you can access through the Service.

```
PS>kubectl apply -f ingress-traefik/
namespace/kiamol-ingress-traefik created
service/ingress-traefik-controller created
deployment.apps/ingress-traefik-controller created
serviceaccount/ingress-traefik created
clusterrole.rbac.authorization.k8s.io/ingress-traefik-controller created
clusterrolebinding.rbac.authorization.k8s.io/ingress-traefik-controller created
PS>
PS>kubectl get svc ingress-traefik-controller -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8080' -n kiamol-ingress-traefik
http://localhost:8080
```

The screenshot shows the Traefik Admin UI with three main sections: Routers, Services, and Middlewares. Each section contains a donut chart and a table of metrics. The metrics are as follows:

Section	Metric	Value
Routers	Success	100%
	Warnings	0%
	Errors	0%
Services	Success	100%
	Warnings	0%
	Errors	0%
Middlewares	Success	100%
	Warnings	0%
	Errors	0%

Traefik uses its own terminology - routers are the frontend paths, services are the backend content providers, and middlewares are features for manipulating behavior. None of the existing Ingress rules have been applied by Traefik.

Figure 15.12 Traefik is a container-native proxy that builds routing rules from the platform and has a UI to display them

Why hasn't Traefik built up a set of routing rules for the to-do list or the Pi application? It would do if we had configured it differently, and all the existing routes would be available via the Traefik Service, but that's not how you use multiple ingress controllers - they would end up fighting over incoming requests. You run more than one controller to provide different proxy capabilities, and you need the application to choose which one to use. You do that with *ingress*

classes, which are a similar concept to storage classes. Traefik has been deployed with a named ingress class and only Ingress objects which request that class will be routed through Traefik.

The ingress class isn't the only difference between ingress controllers, and you may need to model your routes quite differently for different proxies. Figure 15.13 shows how the to-do app needs to be configured in Traefik - there's no response cache in Traefik so we don't get caching for static resources, and sticky sessions are configured at the Service level, so we need an additional Service for the new item route.

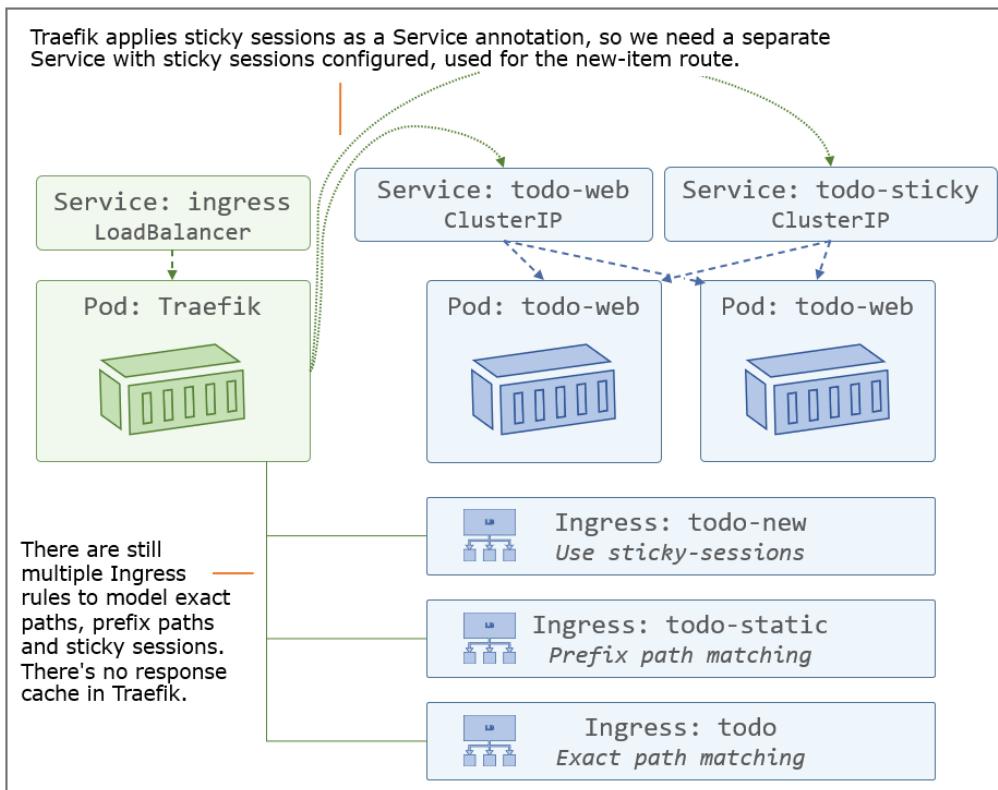


Figure 15.13 Ingress controllers work differently and your route model will need to change accordingly

That's a significantly different model from the Nginx routing in figure 15.9, so if you do plan to run multiple ingress controllers you need to appreciate that there's a high risk of misconfiguration, with teams confusing the different capabilities and approaches. Traefik uses annotations on Ingress resources to configure the routing rules - listing 15.6 shows a spec for the new-item path which selects Traefik as the ingress class and uses an annotation for exact path matching, as Traefik doesn't support the PathType field.

Listing 15.6 - ingress-traefik.yaml, selecting the ingress class with Traefik annotations

```

apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: todo2-new
  annotations:
    kubernetes.io/ingress.class: traefik
    traefik.ingress.kubernetes.io/router.pathmatcher: Path
spec:
  rules:
    - host: todo2.kiamol.local
      http:
        paths:
          - path: /new
            backend:
              serviceName: todo-web-sticky # use the Service which has sticky
              servicePort: 80 # sessions configured for Traefik

```

We'll deploy a new set of Ingress rules using a different host name, so we can route traffic to the same set of to-do list Pods via Nginx or Traefik.

TRY IT NOW Publish the to-do app through the Traefik ingress controller, using the Ingress routes modelled in figure 15.13.

```

# add a new domain for the app - on Windows:
./add-to-hosts.ps1 todo2.kiamol.local ingress-traefik

# OR on Linux/macOS:
./add-to-hosts.sh todo2.kiamol.local ingress-traefik

# apply the new Ingress rules and sticky Service:
kubectl apply -f todo-list/update/ingress-traefik.yaml

# refresh the Traefik admin UI to confirm the new routes
# browse to http://todo2.kiamol.local:8015

```

Traefik watches for events from the Kubernetes API server and it refreshes its routing list automatically. When you deploy the new Ingress objects you'll see the paths shown as routers in the Traefik dashboard, linked to the backend Services. Figure 15.14 shows part of the routing list, together with the to-do app available through the new URL.

Apply new Ingress rules to make the app available through the Traefik ingress controller, which is published through the LoadBalancer Service on port 8015.

```
PS>./add-to-hosts.ps1 todo2.kiamol.local ingress-traefik
PS>
PS>kubectl apply -f todo-list/update/ingress-traefik.yaml
service/todo-web-sticky created
ingress.networking.k8s.io/todo2-new created
ingress.networking.k8s.io/todo2-static created
ingress.networking.k8s.io/todo2 created
```

The terminal window shows the execution of PowerShell commands to update an Ingress rule, resulting in the creation of several Kubernetes resources. The Traefik UI screenshot shows the router list with new paths added, and the browser window shows the application's homepage at todo2.kiamol.local:8015/list.

The router list in Traefik shows the new to-do list paths.

And the same app is available - using the same Pods - via the Traefik ingress Service.

Figure 15.14 Ingress controllers achieve the same goals from different configuration models

If you're evaluating ingress controllers, you should look at the ease of modelling your application paths, together with the approach to troubleshooting and the performance of the proxy. Dual-running controllers in a dedicated environment helps with that as you can isolate other factors and run comparisons using the same application components. A more realistic app will have more complex Ingress rules, and you'll want to be comfortable with how the controller implements features like rate limiting, URL rewrites and client IP access lists.

The other major feature of ingress is publishing apps over HTTPS without configuring certificates and security settings in your applications. This is one area which is consistent between ingress controllers, and in the next section we'll see it with Traefik and Nginx.

15.4 Using ingress to secure your apps with HTTPS

Your web apps should be published over HTTPS, but encryption needs server certificates and certificates are sensitive data items. It's a good practice to make HTTPS an ingress concern, because it centralizes certificate management. Ingress resources can specify a TLS certificate in a Kubernetes Secret (TLS is Transport Layer Security, the encryption mechanism for HTTPS).

Moving TLS away from application teams means you can have a standard approach to provisioning, securing and renewing certificates - and you won't have to spend time explaining why packaging certs inside a container image is a bad idea.

All ingress controllers support loading a TLS certificate from a Secret, but Traefik makes it easier still. If you want to use HTTPS in dev and test environments without provisioning any secrets, Traefik can generate its own self-signed certificate when it runs. You configure that with annotations in the Ingress rules to enable TLS and the default certificate resolver.

TRY IT NOW Using Traefik's generated certificate is a quick way to test your app over HTTPS. It's enabled with more annotations in the Ingress objects.

```
# update the Ingress to use Traefik's own certificate:
kubectl apply -f todo-list/update/ingress-traefik-certResolver.yaml

# browse to dfg
# you'll see a warning in your browser
```

Browsers don't like self-signed certificates because anyone can create them - there's no verifiable chain of authority. You'll see a big warning when you first browse to the site, telling you it's not safe, but you can proceed and the to-do list app will load. You can see in figure 15.15 that the site is encrypted with HTTPS, but with a warning so you know it's not really secure.

Traefik can provision TLS certificates from external issuers, or it can generate its own. This Ingress update publishes the to-do app over HTTPS using Traefik's certs.

```
PS>kubectl apply -f ./todo-list/update/ingress-traefik-certResolver.yaml
ingress.networking.k8s.io/todo2-new configured
ingress.networking.k8s.io/todo2-static configured
ingress.networking.k8s.io/todo2 configured
```

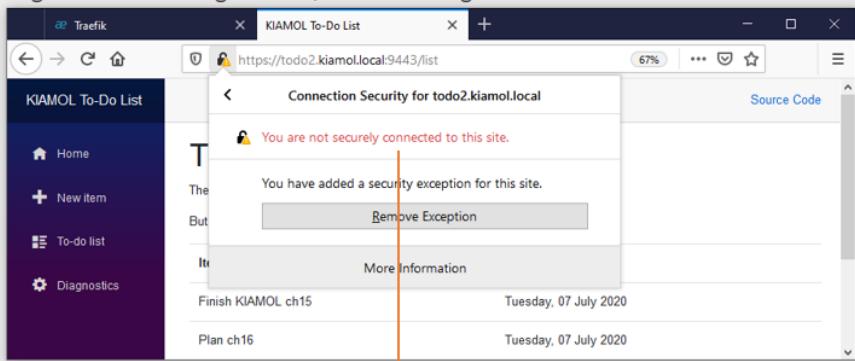


Figure 15.15 Not all HTTPS is secure - self-signed certificates are fine for dev and test environments

Your organization will probably have its own idea about certificates. If you're able to own the provisioning process then you can have a fully-automated system where your cluster fetches short-lived certs from a Certificate Authority (CA), installs them and renews them when required. Let's Encrypt is a great choice - it issues certs free of charge with an easily automated process. Traefik has native integration with Let's Encrypt and for other ingress controllers you can use the open-source cert-manager tool (<https://cert-manager.io>).

Not everyone's ready for an automated provisioning process - some issuers require a human to download certificate files, or your organization may create certificate files from its own Certificate Authority for non-production domains. Then you'll need to deploy the TLS certificate and key files as a Secret in the cluster. That's a common scenario so we'll walk through it in the next exercise, generating a certificate of our own.

TRY IT NOW Run a Pod which generates a custom TLS certificate, and connect to the Pod to deploy the cert files as a Secret - the Pod spec is configured to connect to the Kubernetes API server it's running on.

```
# run the Pod - this generates a certificate when it starts:
kubectl apply -f ./cert-generator.yaml

# connect to the Pod:
kubectl exec -it deploy/cert-generator -- sh

# inside the Pod confirm the cert files have been created:
ls

# rename the cert files - Kubernetes requires specific names:
mv server-cert.pem tls.crt
mv server-key.pem tls.key

# create and label a Secret from the cert files:
kubectl create secret tls kiamol-cert --key=tls.key --cert=tls.crt
kubectl label secret/kiamol-cert kiamol=ch15

# exit the Pod:
exit

# back on the host - confirm the Secret is there:
kubectl get secret kiamol-cert --show-labels
```

That exercise simulates the situation where someone gives you a TLS certificate as a pair of PEM files, which you need to rename and use as the input to create a TLS Secret in Kubernetes. The certificate generation is all done using a tool called OpenSSL, and the only reason for running it inside a Pod is to package up the tool and the scripts to make it easy to use. Figure 15.16 shows my output, with a Secret created in the cluster which can be used by an Ingress object.

This Pod spec runs from a container image which uses the OpenSSL tool to generate a Certificate Authority and a TLS certificate.

The server PEM files are the certificate and key - your files will have the same name but it will be your own certificate.

```
PS>kubectl apply -f ./cert-generator.yaml
deployment.apps/cert-generator created
PS>
PS>kubectl exec -it deploy/cert-generator -- sh
/certs #
/certs # ls
ca-key.pem      ca.pem        server-key.pem
ca.password     server-cert.pem
/certs #
/certs # mv server-cert.pem tls.crt
/certs # mv server-key.pem tls.key
/certs #
/certs # kubectl create secret tls kiamol-cert --key=tls.key --cert=tls.crt
secret/kiamol-cert created
/certs #
/certs # kubectl label secret/kiamol-cert kiamol=ch15
secret/kiamol-cert labeled
/certs #
/certs # exit
PS>
PS>kubectl get secret kiamol-cert --show-labels
NAME          TYPE           DATA   AGE    LABELS
kiamol-cert   kubernetes.io/tls 2      13s   kiamol=ch15
```

TLS certs are so common Kubernetes has a specific type of Secret to store them.

The Secret was created inside the cert-generator Pod using the local API server, so now the Secret is available to use in Ingress objects.

Figure 15.16 If you're given PEM files from a certificate issuer, you can create them as a TLS Secret

HTTPS support is very simple with an ingress controller. You add a TLS section to your Ingress spec and state the name of the Secret to use - and that's it. Listing 15.7 shows an update to the Traefik ingress which applies the new certificate to the `todo2.kiamol.local` host.

Listing 15.7 - ingress-traefik-https.yaml, using the standard Ingress HTTPS feature

```
spec:
  rules:
  - host: todo2.kiamol.local
    http:
      paths:
      - path: /new
        backend:
          serviceName: todo-web-sticky
          servicePort: 80
    tls:
      - secretName: kiamol-cert          # the TLS section switches on HTTPS
                                                # using the cert in this Secret
```

The TLS field with the Secret name is all you need, and it's portable across all ingress controllers. When you deploy the updated Ingress rules the site will be served over HTTPS with your custom certificate. You'll still get a security warning from the browser because the Certificate Authority is untrusted - but if your organization has its own CA then it will be trusted by your machine and the organization's certificates will be valid.

TRY IT NOW Update the to-do list Ingress objects to publish HTTPS using the Traefik ingress controller and your own TLS cert.

```
# apply the Ingress update:
kubectl apply -f todo-list/update/ingress-traefik-https.yaml

# browse to https://todo2.kiamol.local:9443
# there's still a warning but this time it's because
# the KIAMOL CA isn't trusted
```

You can see my output in figure 15.17 - I've opened the certificate details in one screen to confirm this is my own "kiamol" certificate. I accepted the warning in the second screen and the to-do list traffic is now encrypted with the custom cert. The script which generates the certificate sets it for all the `kiamol.local` domains we've used in this chapter, so the cert is valid for the address, but it's not from a trusted issuer.

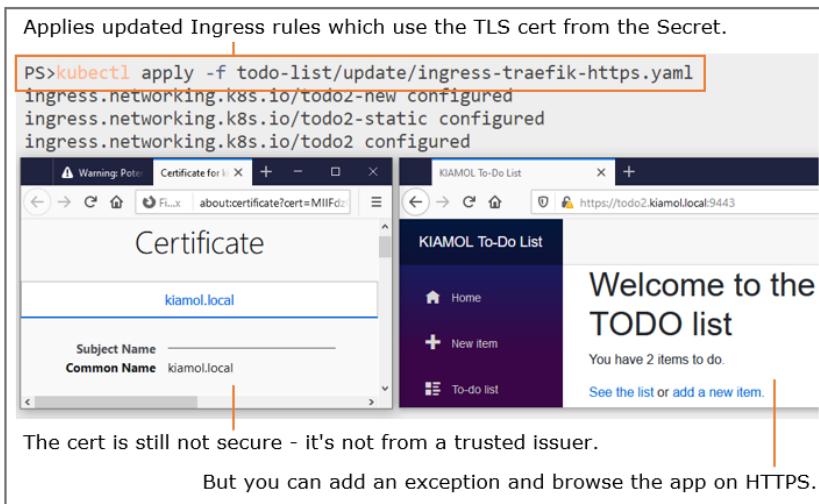


Figure 15.17 Ingress controllers can apply TLS certs from Kubernetes Secrets - if the cert came from a trusted issuer the site would be secure

We'll switch back to Nginx for the final exercise - using the same certificate with the Nginx ingress controller, just to show the process is identical. The updated Ingress specs use the same

rules as the last Nginx deployment, but now they add the TLS field with the same Secret name as listing 15.7.

TRY IT NOW Update the to-do Ingress rules for Nginx, so the app is available using HTTPS over the standard port 443 which the Nginx ingress controller is using.

```
# update the Ingress resources:  
kubectl apply -f todo-list/update/ingress-https.yaml  
  
# browse to https://todo.kiamol.local  
# accept the warnings to view the site  
  
# confirm that HTTP requests are redirected to HTTPS:  
curl http://todo.kiamol.local
```

I cheated when I ran that exercise and added the Kiamol CA to my trusted issuer list in the browser - you can see in figure 15.18 that the site is shown as secure, without any warnings, which is what you'd see for an organization's own certificates. You can also see that the ingress controller takes care of redirecting HTTP requests to HTTPS - the 308 response in the Curl command is all taken care of by Nginx.

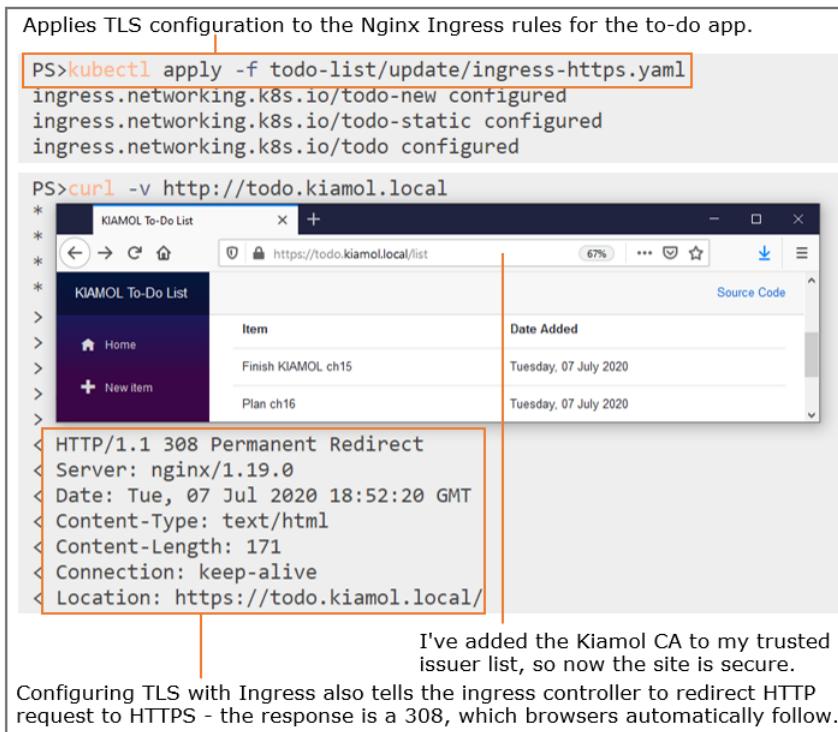


Figure 15.18 The TLS Ingress configuration works in the same way with the Nginx ingress controller

The HTTPS part of ingress is solid and easy to use, and it's good to head to the end of the chapter on a high note. But there's a lot of complexity in using an ingress controller, and in some cases you'll spend more time crafting your Ingress rules than you will on modelling the deployment of the app.

15.5 Understanding ingress and ingress controllers

You'll almost certainly run an ingress controller in your cluster, because it centralizes routing for domain names and moves TLS certificate management away from the applications. The Kubernetes model uses a common Ingress spec and a pluggable implementation which is very flexible, but the user experience is not straightforward. The Ingress spec only records the most basic routing details, and to use more advanced features from your proxy you'll need to add chunks of configuration as annotations.

Those annotations are not portable, and there is no interface specification for the features an ingress controller must support. There will be a migration project if you want to move from Nginx to Traefik or HAProxy or Contour (an open-source project accepted into the CNCF on the very day I wrote this chapter). And you may find the features you need aren't all available. The Kubernetes community is aware of the limitations of ingress, and is working on a long-term replacement called the *Service API*, but as of 2020 that's still in the early stages.

That's not to say that ingress should be avoided - it's the best option right now and it's likely to be the production choice for many years. It's worth evaluating different ingress controllers but then settling on a single option. Kubernetes supports multiple ingress controllers, but the trouble will really start if you use different implementations and have to manage sets of ingress rules with incompatible feature sets invoked through incomprehensible annotations. In this chapter we looked at Nginx and Traefik which are both good options - but there are plenty of others, including commercial options backed with support contracts.

We're done with ingress now, so we can tidy up the cluster to get ready for the lab.

TRY IT NOW Clear down the ingress namespaces and the application resources.

```
kubectl delete ns,all,secret,ingress -l kiamol=ch15
```

15.6 Lab

A nice lab for you here, following on the pattern from chapters 13 and 14. Your job is to build the Ingress rules for the Astronomy Picture of the Day app. Simple...

- start by deploying the ingress controller in the `lab/ingress-nginx` folder
- the ingress controller is restricted to look for Ingress objects in one namespace, so you'll need to figure out which one and deploy the `lab/apod/` folder to that namespace
- the web app should be published at `www.apod.local` and the API at `api.apod.local`
- we want to prevent DDoS attacks, so you should use the rate limiting feature in the ingress controller to prevent too many requests from the same IP address
- and the ingress controller uses a custom class name, so you'll need to find that too.

This is partly about digging into the ingress controller config, and partly about the documentation for the controller - be aware that there are two Nginx ingress controllers. We've used the one from the Kubernetes project in this chapter but there's an alternative published by the Nginx project... My solution is ready for you to check against:

<https://github.com/sixeyed/kiamol/blob/master/ch15/lab/README.md>

16

Securing applications with Policies, Contexts and Admission Control

Containers are a lightweight wrapper around application processes. They start quickly and add very little overhead to your app because they use the operating system kernel of the machine they're running on. That makes them super efficient at the cost of strong isolation - containers can be compromised, and a compromised container could provide unrestricted access to the server and to all the other containers running on it. Kubernetes has lots of features to secure your applications, but none of them are enabled by default. In this chapter you'll learn how to use the security controls in Kubernetes, and how to set up your cluster so those controls are required for all your workloads.

Securing applications in Kubernetes is about limiting what containers can do, so if an attacker exploits an app vulnerability to run commands in the container, they can't get beyond that container. That means restricting network access to other containers and the Kubernetes API, restricting mounts of the host's filesystem, and limiting the operating system features the container can use. We'll cover the essential approaches but the security space is large and evolving. This is a longer chapter and you're about to learn a lot, but it will only be the start of your journey to a secure Kubernetes environment.

16.1 Securing communication with network policies

Restricting network access is one of the simplest ways to secure your applications. Kubernetes has a flat networking model where every Pod can reach every other Pod by its IP address, and Services are accessible throughout the cluster. There's no reason why the Pi web application

should access the to-do list database, or why the Hello, World web app should use the Kubernetes API - but by default they can. You learned in chapter 15 how you can use Ingress resources to control access to HTTP routes, but that only applies to external traffic coming into the cluster. You also need to control access within the cluster, and for that Kubernetes has *network policies*.

Network policies work like firewall rules, blocking traffic to or from Pods at the port level. The rules are very flexible, using label selectors to identify objects. You can deploy a blanket deny-all policy to stop outgoing traffic from all Pods, or you can deploy a policy which restricts incoming traffic to a Pod's metrics port, so it can only be accessed from Pods in the monitoring namespace. Figure 16.1 shows how that looks in the cluster.

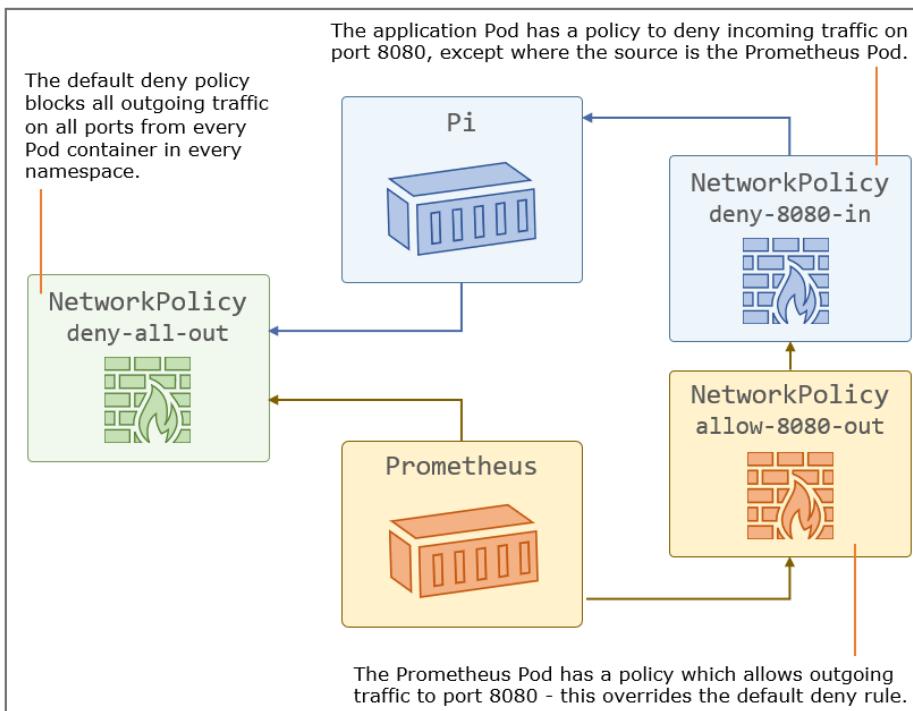


Figure 16.1 Network policy rules are granular - you can apply cluster-wide defaults with Pod overrides

NetworkPolicy objects are separate resources which means they could be modelled outside of the application by a security team, or they could be built by the product team. Or of course each team might think the other team has it covered and apps make it to production without any policies - which is a problem. We'll deploy an app that's slipped through with no policies and look at the problems it has.

TRY IT NOW Deploy the Astronomy Picture of the Day app and confirm that the app components can be accessed by any Pod.

```
# switch to the chapter folder:  
cd ch16  
  
# deploy the APOD app:  
kubectl apply -f apod/  
  
# wait for it to be ready:  
kubectl wait --for=condition=ContainersReady pod -l app=apod-web  
  
# browse to the Service on port 8016 if you want to see today's picture  
  
# now run a sleep Pod:  
kubectl apply -f sleep.yaml  
  
# confirm the sleep Pod can use the API:  
kubectl exec deploy/sleep -- curl -s http://apod-api/image  
  
# and read the metrics from the access log:  
kubectl exec deploy/sleep -- sh -c 'curl -s http://apod-log/metrics | head -n 2'
```

You can clearly see the issue in that exercise - the whole of the cluster is wide open, so from the sleep Pod you can access the APOD API and the metrics from the access log component. Figure 16.2 shows my output, and let's be clear that there's nothing special about the sleep Pod, it's just a simple way to demonstrate the problem - any container in the cluster can do the same.

The APOD app is distributed across many components -
the web Pod needs to access the API and access log.

This sleep Pod doesn't need to use APOD components,
but without network policies it can access them.

```

PS>cd ch16
PS>
PS>kubectl apply -f apod/service/apod-api.yaml
service/apod-api created
deployment.apps/apod-api created
service/apod-log created
deployment.apps/apod-log created
service/apod-web created
deployment.apps/apod-web created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=apod-web
pod/apod-web-7cd794886-qdxks condition met
PS>
PS>kubectl apply -f sleep.yaml
deployment.apps/sleep created
PS>
PS>kubectl exec deploy/sleep -- curl -s http://apod-api/image
{"url": "https://apod.nasa.gov/apod/image/2007/ISS063-E-39888_APOD1050.jpg",
"caption": "Comet NEOWISE from the ISS", "copyright": null}
PS>
PS>kubectl exec deploy/sleep -- sh -c 'curl -s http://apod-log/metrics | head -n 2'
# HELP access_log_total Access Log - total log requests
# TYPE access_log_total counter

```

All the APOD components are available. If they were in a different namespace
the URL would need to change but they would still be accessible.

Figure 16.2 The downside of Kubernetes' flat networking model is that every Pod is accessible

Pods should be isolated so they only receive traffic from the components which need to access them, and they only send traffic to components they need to access. Network policies model that with *ingress rules* (don't confuse them with Ingress resources), which restrict incoming traffic and *egress rules* which restrict outgoing traffic. In the APOD app the only component which should have access to the API is the web app. Listing 16.1 shows that as an ingress rule in a NetworkPolicy object.

Listing 16.1 - networkpolicy-api.yaml, restricting access to Pods by their labels

```

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: apod-api
spec:
  podSelector:          # this is the Pod where the rule applies
    matchLabels:
      app: apod-api
  ingress:              # rules default to deny, so this rule

```

```

- from:          # denies all ingress except where the
  - podSelector: # source of the traffic is a Pod with
    matchLabels: # the apod-web label
      app: apod-web
  ports:          # the restriction is by port
  - port: api     # this port is named in the API Pod spec

```

The NetworkPolicy spec is fairly straightforward and rules can be deployed in advance of the application so it's secure as soon as the Pods start. Ingress and egress rules follow the same pattern, and both can use namespace selectors as well as Pod selectors. You can create global rules and then override them with more fine-grained rules at the application level.

There is one big problem with network policy - when you deploy the rules, they probably won't do anything. Just like Ingress objects need an ingress controller to act on them, NetworkPolicy objects rely on the network implementation in your cluster to enforce them. When you deploy this policy in the next exercise, you'll probably be disappointed to find the APOD API is still not restricted to the web app.

TRY IT NOW Apply the network policy and see if your cluster actually enforces it.

```

# create the policy:
kubectl apply -f apod/update/networkpolicy-api.yaml

# confirm it is there:
kubectl get networkpolicy

# try to access the API from the sleep Pod
# - this is not permitted by the policy:
kubectl exec deploy/sleep -- curl http://apod-api/image

```

You can see in figure 16.3 that the sleep Pod can access the API - the NetworkPolicy which limits ingress to the web Pods is completely ignored. I'm running this on Docker Desktop, but you'll get the same results with a default setup in K3s, AKS or EKS.

Applies an ingress rule to the API Pod so it can only receive traffic from the APOD web app.

The NetworkPolicy object is created.

```
PS>kubectl apply -f apod/update/networkpolicy-api.yaml
networkpolicy.networking.k8s.io/apod-api created
```

NAME	POD-SELECTOR	AGE
apod-api	app=apod-api	5s

```
PS>kubectl get networkpolicy
```

```
PS>kubectl exec deploy/sleep -- curl -s http://apod-api/image
{"url":"https://apod.nasa.gov/apod/image/2007/ISS063-E-39888_APOD1050.jpg",
"caption":"Comet NEOWISE from the ISS","copyright":null}
```

But it's not enforced. In my cluster the network plugin doesn't support network policy.

Figure 16.3 The network setup in your Kubernetes cluster may not enforce network policies

The networking layer in Kubernetes is pluggable and not every network plugin supports NetworkPolicy enforcement. The simple networks in standard cluster deployments don't have support, so you get into this tricky situation where you can deploy all your NetworkPolicy objects, but you don't know whether they're being enforced unless you test them. The cloud platforms have different levels of support here - you can specify a network policy option when you create an AKS cluster; with EKS you need to manually install a different network plugin after you create the cluster.

Which is all very frustrating for you following along with these exercises (and me writing them), but it causes a much more dangerous disconnect for organizations using Kubernetes in production. You should look to adopt security controls early in the build cycle, so NetworkPolicy rules are applied in your dev and test environments to run the app close to production config. A misconfigured network policy can easily break your app, but you won't know that if your non-production environments don't enforce policy.

If you want to see NetworkPolicy in action the next exercise creates a custom cluster using Kind with Calico - an open-source network plugin which enforces policy. You'll need Docker and the Kind command line installed for this. Be warned: **this exercise alters the Linux configuration for Docker, and will make your original cluster unusable**. Docker Desktop users can fix everything with the *Reset Kubernetes* button and Kind users can replace their old cluster with a new one but other setups might not be so lucky - it's fine to skip these exercises; we'll switch back to your normal cluster in the next section.

TRY IT NOW Create a new cluster with Kind and deploy a custom network plugin.

```
# install the Kind command line using instructions at
# https://kind.sigs.k8s.io/docs/user/quick-start/
```

```
# create a new cluster with a custom Kind configuration:
kind create cluster --image kindest/node:v1.18.4 --name kiamol-ch16 --config
    kind/kind-calico.yaml

# install the Calico network plugin:
kubectl apply -f kind/calico.yaml

# wait for Calico to spin up:
kubectl wait --for=condition=ContainersReady pod -l k8s-app=calico-node -n kube-
    system

# confirm your new cluster is ready:
kubectl get nodes
```

My output in figure 16.4 is abbreviated; you'll see lots more objects being created in the Calico deployment. At the end I have a new cluster which enforces network policy. Unfortunately the only way to know if your cluster uses a network plugin which does enforce policy is to set up your cluster with a network plugin that you know enforces policy...

Create a new Kind cluster without installing the default network plugin.

```
PS>kind create cluster --image kindest/node:v1.18.4 --name kiamol-ch16 --co
nfig kind/kind-calico.yaml
Creating cluster "kiamol-ch16" ...
• Ensuring node image (kindest/node:v1.18.4) ...
✓ Ensuring node image (kindest/node:v1.18.4) ...

Set kubectl context to "kind-kiamol-ch16"
You can now use your cluster with:

kubectl cluster-info --context kind-kiamol-ch16

Thanks for using kind! 😊
PS>
PS>kubectl apply -f kind/calico.yaml
configmap/calico-config created
```

```
PS>kubectl wait --for=condition=ContainersReady pod -l k8s-app=calico-node
-n kube-system
pod/calico-node-mm5r7 condition met
PS>
PS>kubectl get nodes
NAME                  STATUS   ROLES      AGE     VERSION
kiamol-ch16-control-plane  Ready   master    115s   v1.18.4
```

Deploy Calico - the plugin modifies the underlying network.
You will lose access to your old cluster if you do this.

When the node is ready, the network components are all online.

Figure 16.4 Installing Calico gives you a cluster with network policy support - at the cost of your other cluster

Now we can try again. This is a completely new cluster with nothing running, but of course Kubernetes manifests are portable so we can quickly deploy the APOD app again and try it out. (Kind supports multiple clusters running different Kubernetes versions with different configurations, so it's a great option for test environments, but it's not as developer-friendly as Docker Desktop or K3s).

TRY IT NOW Repeat the APOD and sleep deployments and confirm that the network policy blocks unauthorized traffic.

```
# deploy the APOD app to the new cluster:
kubectl apply -f apod/

# wait for it to spin up:
kubectl wait --for=condition=ContainersReady pod -l app=apod-web

# deploy the sleep Pod:
kubectl apply -f sleep.yaml

# confirm the sleep Pod has access to the APOD API:
kubectl exec deploy/sleep -- curl -s http://apod-api/image

# apply the network policy:
kubectl apply -f apod/update/networkpolicy-api.yaml

# confirm the sleep Pod can't access the API:
kubectl exec deploy/sleep -- curl -s http://apod-api/image

# but the APOD web app still can:
kubectl exec deploy/apod-web -- wget -O- -q http://apod-api/image
```

Figure 16.5 shows what we expected the first time around - only the APOD web app can access the API; the sleep app times out when it tries to connect because the network plugin blocks the traffic.

Without network policy the new sleep Pod can access the API Pod.

The network policy blocks the traffic.

```
PS>kubectl apply -f apod/
service/apod-api created
deployment.apps/apod-api created
service/apod-log created
deployment.apps/apod-log created
service/apod-web created
deployment.apps/apod-web created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=apod-web
pod/apod-web-7cd794886-f9lwn condition met
PS>
PS>kubectl apply -f sleep.yaml
deployment.apps/sleep created
PS>
PS>kubectl exec deploy/sleep -- curl -s http://apod-api/image
{"url":"https://apod.nasa.gov/apod/image/2007/ISS063-E-39888_AP0D1050.jpg",
 "caption":"Comet NEOWISE from the ISS","copyright":null}
PS>
PS>kubectl apply -f apod/update/networkpolicy-api.yaml
networkpolicy.networking.k8s.io/apod-api created
PS>
PS>kubectl exec deploy/sleep -- curl -s http://apod-api/image
command terminated with exit code 28
PS>
PS>kubectl exec deploy/apod-web -- wget -O- -q http://apod-api/image
{"url":"https://apod.nasa.gov/apod/image/2007/ISS063-E-39888_AP0D1050.jpg",
 "caption":"Comet NEOWISE from the ISS","copyright":null}
```

The APOD web app can still access the API.

Figure 16.5 Calico enforces policy so traffic to the API Pod is only allowed from the web Pod

Network policies are an important security control in Kubernetes, and they're very attractive to infrastructure teams who are used to firewalls and segregated networks. But you need to understand where policies fit in your developer workflow if you do choose to adopt them. If engineers run their own clusters without enforcement and you only apply policy later in the pipeline, you're running in very different environments and something will get broken.

I've only covered the basic details of the NetworkPolicy API here, because the complexity is more in the cluster configuration than in the policy resources. If you want to explore further there's a great GitHub repo full of network policy recipes published by Ahmet Alp Balkan, an engineer at Google: [ahmetb/kubernetes-network-policy-recipes](https://github.com/ahmetb/kubernetes-network-policy-recipes).

Now let's clear up the new cluster and see if your old cluster still works.

TRY IT NOW Remove the Calico cluster and see if the old cluster is still accessible.

```
# delete the new cluster:  
kind delete cluster --name kiamol-ch16  
  
# list your Kubernetes contexts:  
kubectl config get-contexts  
  
# switch back to your previous cluster:  
<your_old_cluster_name>  
  
# see if you can connect:  
kubectl get nodes
```

Your previous cluster is probably no longer accessible because of the network changes Calico made - even though Calico isn't running now. Figure 16.6 shows me about to hit the *Reset Kubernetes* button in Docker Desktop; if you're using Kind you'll need to delete and recreate your original cluster, and if you're using something else and it doesn't work - I did warn you.

The new cluster changed the network in Docker Desktop, so my old cluster is still running, but it's unreachable.

```

PS>kind delete cluster --name kiamol-ch16
Deleting cluster "kiamol-ch16" ...
PS>
PS>kubectl config get-contexts
CURRENT      NAME           CLUSTER          AUTHINFO
              docker-desktop   docker-desktop   docker-desktop
PS>
PS>kubectl config set-context docker-desktop
Context "docker-desktop" modified.
PS>
PS>kubectl get nodes
Unable to connect to the server: dial tcp [::1]:8080: connectex: No connection could be made because the target machine actively refused it.

```

Settings

General	Kubernetes
<input checked="" type="radio"/> Resources	v1.18.3
<input type="radio"/> Docker Engine	<input checked="" type="checkbox"/> Enable Kubernetes Start a Kubernetes single-node cluster when starting Docker Desktop.
<input type="radio"/> Command Line	<input type="checkbox"/> Deploy Docker Stacks to Kubernetes by default Make Kubernetes the default orchestrator for "docker stack" commands (changes ~/.docker/config.json")
<input type="radio"/> Kubernetes	<input type="checkbox"/> Show system containers (advanced) Show Kubernetes internal containers when using Docker commands.

Reset Kubernetes Cluster

All stacks and Kubernetes resources will be deleted.

Resetting the Kubernetes cluster in Docker Desktop - or deleting and creating a new cluster in Kind - will fix it all up.

Figure 16.6 Calico running in a container was able to reconfigure my network and break things

Now that we're all back to normal (hopefully), we can move on to securing containers themselves, so applications don't have privileges like reconfiguring the network stack.

16.2 Restricting container capabilities with security contexts

Container security is really about Linux security and the access model for the container user (Windows Server containers have a different user model which doesn't have the same issues). Linux containers usually run as the *root* super-admin account, and unless you explicitly configure the user, *root* inside the container is *root* on the host too. If an attacker can break out of a container running as *root*, they're in charge of your server now. That's a problem for all container runtimes, but Kubernetes adds a few more problems of its own.

In the next exercise you'll run the Pi web application with a basic Deployment configuration. That container image is packaged on top of the official .NET Core application runtime image from Microsoft, and the Pod spec isn't deliberately insecure, but you'll see that the defaults aren't encouraging.

TRY IT NOW Run a simple application and check the default security situation.

```
# deploy the app:  
kubectl apply -f pi/  
  
# wait for the container to start:  
kubectl wait --for=condition=ContainersReady pod -l app=pi-web  
  
# print the name of the user in the Pod container:  
kubectl exec deploy/pi-web -- whoami  
  
# try to access the Kubernetes API server:  
kubectl exec deploy/pi-web -- sh -c 'curl -k -s https://kubernetes.default | grep message'  
  
# and print the API access token:  
kubectl exec deploy/pi-web -- cat /run/secrets/kubernetes.io/serviceaccount/token
```

The behavior is scary: the app runs as root, it has access to the Kubernetes API server and it even has a token set up so it can authenticate with Kubernetes - figure 16.7 shows it all in action. Running as root magnifies any exploit an attacker can find in the application code or the runtime. Having access to the Kubernetes API means an attacker doesn't even need to break out of the container - they can use the token to query the API and do interesting things, like fetch the contents of Secrets (depending on the access permissions for the Pod, which you'll learn about in chapter 17).

Deploys the Pi application from a simple spec which doesn't apply any security controls.
The container runs processes as root, which is a big risk if the app gets compromised.

```

PS>kubectl apply -f pi/
deployment.apps/pi-web created
service/pi-web created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=pi-web
pod/pi-web-76dbc87b56-nmxr4 condition met
PS>
PS>kubectl exec deploy/pi-web -- whoami
root
PS>
PS>kubectl exec deploy/pi-web -- sh -c 'curl -k -s https://kubernetes.default | grep message'
"message": "forbidden: User \"system:anonymous\" cannot get path \"/\"
PS>
PS>kubectl exec deploy/pi-web -- cat /run/secrets/kubernetes.io/serviceaccount/token
eyJhbGciOiJSUzI1NiIsImtpZCI6IkZudld40TRGNl9KSl0qYTgwZndoZjExcjcyRVROZ0FZWdh
GajhXV3RJU0UiFQ.eyJpc3MiOiJrdWJlcml5ldGVzL3NlcnpY2VhY2NvdW50Iiwia3ViZXJuZXr

```

The container can make network calls to the Kubernetes API server - this returns a 403 because the request didn't provide any credentials.

Luckily Kubernetes provides the container with an access token which attackers can use to authenticate with the API.

Figure 16.7 If you've heard the phrase "secure by default", it wasn't said about Kubernetes

Kubernetes provides multiple security controls at the Pod and container level, but they're not enabled by default because they could break your app. You can run containers as a different user, but some apps only work if they're running as root; you can drop Linux capabilities to restrict what the container can do, but then some app features could fail. This is where automated testing comes in, because you can increasingly tighten the security around your apps, running tests at each stage to confirm everything still works.

The main control you'll use is the `SecurityContext` field, which applies security at the Pod and container level. Listing 16.2 shows a Pod `SecurityContext` which explicitly sets the user and the Linux group (a collection of users), so all the containers in the Pod run as the *unknown* user rather than root.

Listing 16.2 - deployment-podsecuritycontext.yaml, running as a specific user

```

spec:
  securityContext:
    runAsUser: 65534
    runAsGroup: 3000

```

That's simple enough, but moving away from root has repercussions and the Pi spec needs a few more changes. The app listens on port 80 inside the container, and Linux requires elevated permissions to listen on that port. Root has the permission but the new user doesn't so the app will fail to start - it needs some additional config in an environment variable to set the app to listen on port 5001 instead, which is valid for the new user. This is the sort of detail you need to drive out for each app or each class of application, and you'll only find the requirements when the app stops working.

TRY IT NOW Deploy the secured Pod spec -this uses a non-root user and an unrestricted port, but the port mapping in the Service hides that detail from consumers.

```
# add the non-root SecurityContext:
kubectl apply -f pi/update/deployment-podsecuritycontext.yaml

# wait for the new Pod:
kubectl wait --for=condition=ContainersReady pod -l app=pi-web

# confirm the user:
kubectl exec deploy/pi-web -- whoami

# list the API token files:
kubectl exec deploy/pi-web -- ls -l /run/secrets/kubernetes.io/serviceaccount/token

# print out the access token:
kubectl exec deploy/pi-web -- cat /run/secrets/kubernetes.io/serviceaccount/token
```

Running as a non-root user addresses the risk of an application exploit escalating into a full server takeover, but as you see in figure 16.8 it doesn't solve all the problems. The Kubernetes API token is mounted with permissions for any account to read it, so an attacker could still use the API in this setup. What they can do with the API depends on how your cluster is configured - in early versions of Kubernetes they'd be able to do everything. The identity for the API server is different from the Linux user and it might have admin rights in the cluster even though the container process is running as a least-privilege user.

The Pod SecurityContext sets the container user to a least-privilege account.

But the token file grants read access to all users.

```
PS>kubectl apply -f pi/update/deployment-podsecuritycontext.yaml
deployment.apps/pi-web configured
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=pi-web
pod/pi-web-74c97dbd7b-9tqkf condition met
PS>
PS>kubectl exec deploy/pi-web -- whoami
nobody
PS>
PS>kubectl exec deploy/pi-web -- ls -l /run/secrets/kubernetes.io/serviceaccount/token
lrwxrwxrwx 1 root root 12 Jul 10 10:41 /run/secrets/kubernetes.io/serviceaccount/token -> ..data/token
PS>
PS>kubectl exec deploy/pi-web -- cat /run/secrets/kubernetes.io/serviceaccount/token
eyJhbGciOiJSUzI1NiIsImtpZCI6IkZudld40TRGNl9KSldqYTgwZndOZjExcjcyRVROZ0FZWdhGajhXV3RJU0UifQ.eyJpc3MiOiJrdWJlcmt5ldGVzL3NlcnPzY2VhY2NvdW50Iiwia3ViZXJuZXr
```

So the new user could still access the Kubernetes API server.

Figure 16.8 You need a security in-depth approach to Kubernetes; one setting is not enough

An option in the Pod spec stops Kubernetes mounting the access token, which you should include for every app that doesn't actually need to use the Kubernetes API. That will be pretty much everything, except workloads like ingress controllers which need to find Service endpoints. It's a safe option to set, but the next level of runtime control will need more testing and evaluation. The SecurityContext field in the container spec allows for more fine-grained control than at the Pod level. Listing 16.3 shows a set of options which work for the Pi app.

Listing 16.3 - deployment-no-serviceaccount-token.yaml, tighter security policies

```
spec:
  automountServiceAccountToken: false      # removes the API token
  securityContext:
    runAsUser: 65534                      # applies to all containers
    runAsGroup: 3000
  containers:
    - image: kiamol/ch05-pi
      # ...
      securityContext:
        allowPrivilegeEscalation: false    # applies for this container
        capabilities:
          drop:
            - all                         # process from escalating to
                                            # higher privileges and drops
                                            # all additional capabilities
```

The capabilities field lets you explicitly add and remove Linux kernel capabilities - this app works happily with all capabilities dropped, but other apps will need some added back in. One feature this app doesn't support is the `readOnlyRootFilesystem` option. That's a powerful one to include if your app works with a read-only filesystem, because it means attackers can't write files - so they can't download malicious scripts or binaries. How far you take this depends on the security profile of your organization. You can mandate that all apps need to run as non-root, with all capabilities dropped and with a read-only filesystem, but that might mean you need to rewrite most of your apps.

A pragmatic approach is to secure your existing apps as tightly as you can at the container level, and make sure you have security in depth around the rest of your policies and processes. The final spec for the Pi app is not perfectly secure, but it's a big improvement on the defaults - and the application still works.

TRY IT NOW Update the Pi app with the final security configuration.

```
# update to the Pod spec in listing 16.3:
kubectl apply -f pi/update/deployment-no-serviceaccount-token.yaml

# confirm the API token doesn't exist:
kubectl exec deploy/pi-web -- cat /run/secrets/kubernetes.io/serviceaccount/token

# but that the API server is still accessible:
kubectl exec deploy/pi-web -- sh -c 'curl -k -s https://kubernetes.default | grep message'

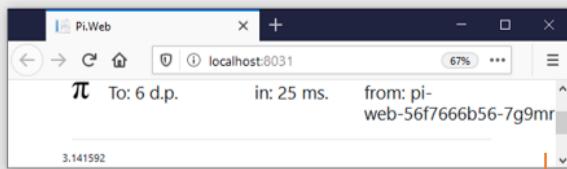
# get the URL and check the app still works:
kubectl get svc pi-web -o jsonpath='http://.status.loadBalancer.ingress[0].*:8031'
```

You can see my output in figure 16.9 - the app can still reach the Kubernetes API server, but it has no access token so an attacker would need to do more work to send valid API requests. And applying a NetworkPolicy to deny ingress to the API server would remove that option altogether.

This spec tightens security for the app - it could do more but the app doesn't work with all of Kubernetes' security controls.

The API token doesn't exist this time.

```
PS>kubectl apply -f pi/update/deployment-no-serviceaccount-token.yaml
deployment.apps/pi-web configured
PS>
PS>kubectl exec deploy/pi-web -- cat /run/secrets/kubernetes.io/serviceaccount/token
cat: /run/secrets/kubernetes.io/serviceaccount/token: No such file or directory
command terminated with exit code 1
PS>
PS>kubectl exec deploy/pi-web -- sh -c 'curl -k -s https://kubernetes.default.svc.cluster.local/api/v1/pods/pi-web-56f7666b56-7g9mr/serviceaccount/default | grep message'
"message": "forbidden: User \"system:anonymous\" cannot get path \"/\"",
PS>
PS>kubectl get svc pi-web -o yaml | jq '.status.loadBalancer.ingress[0].ip'
http://localhost:8031
PS>
PS>
```



The API server can still be reached, but without the token it can't be used.

The app still works and Pi is still the same number.

Figure 16.9 The secured app is the same for users, but much less fun for attackers

You need to invest in adding security to your apps, but if you have a reasonably small range of application platforms then you can build up generic profiles: you might find that all your .NET apps can run as non-root but they need a writeable filesystem and all your Go apps can run with a read-only filesystem but need some Linux capabilities added. The challenge then is to making sure your profiles actually get applied, and Kubernetes has a nice feature for that: *admission control*.

16.3 Blocking and modifying workloads with webhooks

Every object you create in Kubernetes goes through a process to check if it's OK for the cluster to run that object. That's admission control and we saw an admission controller at work in chapter 12, trying to deploy a Pod spec which requested more resources than the namespace had available. The ResourceQuota admission controller is a built-in controller which stops workloads running if they exceed quotas, and Kubernetes has a plug-in system so you can add your own admission control rules.

Two other controllers add that extensibility: the ValidatingAdmissionWebhook which works like ResourceQuota to allow or block object creation, and the MutatingAdmissionWebhook which can actually edit object specs, so the object which gets created is different from the request. Both controllers work in the same way - you create a configuration object specifying the object lifecycles you want to control, and a URL for a web server which applies the rules. Figure 16.10 shows how the pieces fit together.

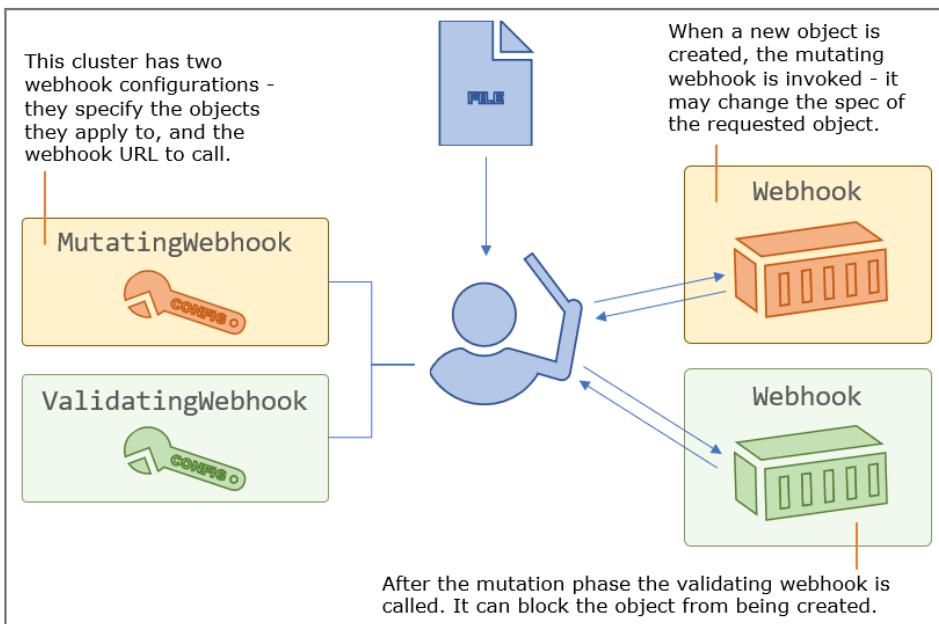


Figure 16.10 Admission webhooks let you apply your own rules when objects are created

Admission webhooks are hugely powerful because Kubernetes calls into your own code, which can be running in any language you like and can apply whatever rules you need. In this section we'll apply some webhooks I've written in Node.js - you won't need to edit any code, but you can see in listing 16.4 that the code isn't particularly complicated.

Listing 16.4 - validate.js, custom logic for a validating webhook

```
# the incoming request has the object spec - this checks to see
# if the service token mount property is set to false -
# if not the response stops the object being created:

if (object.spec.hasOwnProperty("automountServiceAccountToken")) {
    admissionResponse.allowed =
        (object.spec.automountServiceAccountToken == false);
}
```

Webhook servers can run anywhere - inside or outside of the cluster - but they must be served on HTTPS. The only complication comes if you want to run webhooks inside your cluster signed by your own Certificate Authority (CA), because the webhook configuration needs a way to trust the CA. This is a common scenario, so we'll walk through that complexity in the next exercises.

TRY IT NOW Start by creating a certificate and deploying the webhook server to use that certificate.

```
# run the Pod to generate a certificate:  
kubectl apply -f ./cert-generator.yaml  
  
# when the container is ready, the certificate is done:  
kubectl wait --for=condition=ContainersReady pod -l app=cert-generator  
  
# and the Pod has created it as a TLS Secret:  
kubectl get secret -l kiamol=ch16  
  
# deploy the webhook server, using the TLS Secret:  
kubectl apply -f admission-webhook/  
  
# print the CA certificate:  
kubectl exec -it deploy/cert-generator -- cat ca.base64
```

The last command in that exercise will fill your screen with base-64 encoded text, which you'll need in the next exercise (don't worry about writing it down though; we'll automate all the steps). You've now got the webhook server running, secured by a TLS certificate issued by a custom CA - my output is in figure 16.11.

This tool creates a CA and a TLS certificate, and deploys the cert as a Secret in the cluster.

The webhook server is secured with HTTPS, using the TLS Secret.

```
PS>kubectl apply -f ./cert-generator.yaml
deployment.apps/cert-generator created
```

```
PS>
PS>kubectl wait --for=condition=ContainersReady pod pod/cert-generator-6db9bfbcc5-brdgw condition met
PS>
PS>kubectl get secret -l kiamol=ch16
NAME          TYPE        DATA   AGE
admission-webhook-secret  kubernetes.io/tls  2       3m7s
PS>
PS>kubectl apply -f admission-webhook/
service/admission-webhook created
deployment.apps/admission-webhook created
PS>
PS>kubectl exec -it deploy/cert-generator -- cat ca.base64
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUZnVENDQT0tZ0F3SUJBZ01VTGppb3pIbj:Gdmgb1VLdXhPNUkzb3NFRndjd0RRWUpLb1pJaHZjTkFRRUwKQ1FBd1VERUxNQWthQTFVRUJoTl
```

Webhook configurations need to trust the CA if it's not a recognised issuer.
This file contains the CA cert in the format Kubernetes expects.

Figure 16.11 Webhooks are potentially dangerous, so they need to be secured with HTTPS

The Node.js app is running and it has two endpoints - a validating webhook which checks that all Pod specs have the `automountServiceAccountToken` field set to false, and a mutating webhook which applies a container `SecurityContext` set with the `runAsNonRoot` flag. Those two policies are intended to work together to ensure a base level of security for all applications. Listing 16.5 shows the spec for the `ValidatingWebhookConfiguration` object.

Listing 16.5 - validatingWebhookConfiguration.yaml, applying a webhook

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: ValidatingWebhookConfiguration
metadata:
  name: servicetokenpolicy
webhooks:
  - name: servicetokenpolicy.kiamol.net
    rules:
      - operations: [ "CREATE", "UPDATE" ]           # these are the object
        apiGroups: [ "" ]                          # types and operations
        apiVersions: [ "v1" ]                      # which invoke the
        resources: [ "pods" ]                     # webhook - all Pods
    clientConfig:
      service:
        name: admission-webhook                 # the webhook Service to call
        namespace: default
        path: "/validate"                         # URL for the webhook
```

```
caBundle: {{ .Values.caBundle }} # the CA certificate
```

Webhook configurations are very flexible; you can set the types of operation and the types of object the webhook operates on. You can have multiple webhooks configured for the same object - validating webhooks are all called in parallel and any one of them can block the operation. This YAML file is part of a Helm chart which I'm using just for this config object, as an easy way to inject the CA certificate. A more advanced Helm chart would include a job to generate the cert, and deploy the webhook server along with the configurations - but then you wouldn't see how it all fits together.

TRY IT NOW Deploy the webhook configuration, passing the CA cert from the generator Pod as a value to the local Helm chart. Then try to deploy an app which fails the policy.

```
# install the configuration object:
helm install validating-webhook admission-webhook/helm/validating-webhook/ --set
  caBundle=$(kubectl exec -it deploy/cert-generator -- cat ca.base64)

# confirm it's been created:
kubectl get validatingwebhookconfiguration

# try to deploy an app:
kubectl apply -f vweb/v1.yaml

# check the webhook logs:
kubectl logs -l app=admission-webhook --tail 3

# show the ReplicaSet status for the app:
kubectl get rs -l app=vweb-v1

# and show the details:
kubectl describe rs -l app=vweb-v1
```

In that exercise you can see the strength and the limitation of validating webhooks. The webhook operates at the Pod level and it stops Pods being created if they don't match the service token rule. But it's the ReplicaSet and the Deployment which try to create the Pod and they don't get blocked by the admission controller, so you have to dig a bit deeper to find why the app isn't running. My output is in figure 16.12, where the describe command is abridged to show just the error line.

Creates a validating webhook, using the local webhook server and the CA certificate from the generator.

The webhook gets called and the spec fails to meet the rules.

```
PS>helm install validating-webhook admission-webhook/helm/validating-webhook
/ --set caBundle=$(kubectl exec -it deploy/cert-generator -- cat ca.base64)
NAME: validating-webhook
LAST DEPLOYED: Fri Jul 10 13:19:24 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
PS>
PS>kubectl get validatingwebhookconfiguration
NAME          WEBHOOKS   AGE
servicetokenpolicy  1        12s
PS>
PS>kubectl apply -f vweb/v1.yaml
service/vweb-v1 created
deployment.apps/vweb-v1 created
PS>
PS>kubectl logs -l app=admission-webhook --tail 3
info: Validating: vweb-v1-647d5657b-mxmlf; request UID: 3d736c6d-7f19-4554-9
dbc-49b0eaa6e5a0
info: - no automountServiceAccountToken
info: validated request UID: 3d736c6d-7f19-4554-9dbc-49b0eaa6e5a0
PS>
PS>kubectl get rs -l app=vweb-v1
NAME      DESIRED   CURRENT   READY   AGE
vweb-v1-647d5657b  1         0         0       14s
PS>
PS>kubectl describe rs -l app=vweb-v1
Name:           vweb-v1-647d5657b
Warning FailedCreate 17s (x14 over 58s) replicaset-controller Error cr
eating: admission webhook "servicetokenpolicy.kiamol.net" denied the request
: automountServiceAccountToken must be set to false
```

The deployment doesn't fail, but the ReplicaSet is not up to scale.

Digging into the detail we can see that the webhook blocked the object creation - this is the response message from the Node.js webhook server.

Figure 16.12 Validating webhooks can block object creation whether it was instigated by a user or a controller

You need to think carefully about the objects and operations you want your webhook to act on. This validation could happen at the Deployment level instead, which would give a better user experience - but it would miss Pods created directly, or by other types of controller. It's also important to return a very clear message in the webhook response, so users know how to fix the issue. The ReplicaSet will just keep trying to create the Pod and failing (it's tried 18 times

on my cluster while I've been writing this), but the failure message tells me what to do and this one is easy to fix.

One of the problems with admission webhooks is that they score very low on discoverability. You can use Kubectl to check if there are any validating webhooks configured, but that doesn't tell you anything about the actual rules, so you need to have that documented outside of the cluster. The situation gets even more confusing with mutating webhooks, because if they work as expected they give users a different object from the one they tried to create. In the next exercise you'll see that a well-intentioned mutating webhook can break applications.

TRY IT NOW Configure a mutating webhook using the same webhook server, but a different URL path. This webhook adds security settings to Pod specs. Deploy another app and you'll see the changes from the webhook stop the app from running.

```
# deploy the webhook configuration:
helm install mutating-webhook admission-webhook/helm/mutating-webhook/ --set
  caBundle=$(kubectl exec -it deploy/cert-generator -- cat ca.base64)

# confirm it's been created:
kubectl get mutatingwebhookconfiguration

# deploy a new web app:
kubectl apply -f vweb/v2.yaml

# print the webhooks server logs:
kubectl logs -l app=admission-webhook --tail 5

# show the status of the ReplicaSet:
kubectl get rs -l app=vweb-v2

# and the details:
kubectl describe pod -l app=vweb-v2
```

Oh dear. The mutating webhook adds a SecurityContext to the Pod spec with the runAsNonRoot field set to true. That flag tells Kubernetes not to run any containers in the Pod if they're configured to run as root - which this app is, because it's based on the official Nginx image which does use root. As you can see in figure 16.13, describing the Pod tells you what the problem is but it doesn't state that the spec has been mutated - users will be highly confused when they check their YAML again and there is no runAsNonRoot field.

This mutating webhook adds security so containers don't run as the root user.

When the new app is deployed, the webhook adds a new field as a patch to the spec.

```

PS>helm install mutating-webhook admission-webhook/helm/mutating-webhook/
set caBundle=$(kubectl exec -it deploy/cert-generator -- cat ca.base64)
NAME: mutating-webhook
LAST DEPLOYED: Fri Jul 10 14:12:57 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
PS>
PS>kubectl get mutatingwebhookconfiguration
NAME          WEBHOOKS   AGE
nonrootpolicy  1          7s
PS>
PS>kubectl apply -f vweb/v2.yaml
service/vweb-v2 created
deployment.apps/vweb-v2 created
PS>
PS>kubectl logs -l app=admission-webhook --tail 5
info: Mutating object: undefined; request UID: c2974b99-81b7-4bc2-ad01-c72b00c7faaa7
info: - added runAsNonRoot patch
info: Mutated request UID: c2974b99-81b7-4bc2-ad01-c72b0c7faaa7
info: Validating: vweb-v2-74877cc694-kfq4t; request UID: a22f19de-c51f-4574-9df7-89adac4657bd
info: Validated request UID: a22f19de-c51f-4574-9df7-89adac4657bd
PS>
PS>kubectl get rs -l app=vweb-v2
NAME          DESIRED   CURRENT   READY   AGE
vweb-v2-74877cc694  1         1         0        28s
PS>
PS>kubectl describe pod -l app=vweb-v2
Name:           vweb-v2-74877cc694-kfq4t
Warning Failed  8s (x6 over 58s) kubelet, docker-desktop  Error: container has runAsNonRoot and image will run as root

```

But this image does use root, so the container doesn't start and the app never runs.

Figure 16.13 Mutating webhooks can cause application failures which are hard to debug

The logic inside a mutating webhook is entirely down to you - you can accidentally change objects with an invalid spec they will never deploy. It's a good idea to have a more restrictive object selector for your webhook configurations - listing 16.5 applies to every Pod, but you can add namespace and label selectors to narrow down the scope. This webhook has been built with sensible rules, and if the Pod spec already contains a `runAsNonRoot` value the webhook leaves it alone, so apps can be modelled to explicitly require the root user.

Admission controller webhooks are a useful tool to know about and they let you do some very cool things. You can add sidecar containers to Pods with mutating web hooks - so you could use a label to identify all the apps which write log files and have a webhook automatically add a logging sidecar to those Pods. Webhooks can be dangerous, which is something you can mitigate with good testing and selective application in your config objects, but they will always be invisible because the logic is hidden inside the webhook server.

In the next section we'll look at an alternative approach which uses validating webhooks under the hood, but wraps them in a management layer. *Open Policy Agent* (OPA) lets you define your rules in Kubernetes objects which are discoverable in the cluster and don't require custom code.

16.4 Controlling admission with Open Policy Agent

OPA is a unified approach to writing and implementing policies. The goal is to provide a standard language for describing all kinds of policy, and integrations to apply policies in different platforms. You can describe data access policies and deploy them in SQL databases, and you can describe admission control policies for Kubernetes objects. OPA is another CNCF project and it provides a much cleaner alternative to custom validating webhooks with OPA Gatekeeper.

There three parts to using OPA Gatekeeper - you deploy the Gatekeeper components in your cluster, which include a webhook server and a generic `ValidatingWebhookConfiguration`; then you create a *constraint template* which describes the admission control policy; then you create a specific *constraint* based on the template. It's a very flexible approach where you can build a template for the policy "all Pods must have the expected labels", and then deploy a constraint to say which labels are needed in which namespace.

We'll start by removing the custom webhooks we added and deploying OPA Gatekeeper, ready to apply some admission policies.

TRY IT NOW Uninstall the webhook components and deploy Gatekeeper.

```
# remove the webhook configurations created with Helm:
helm uninstall mutating-webhook
helm uninstall validating-webhook

# and the Node.js webhook server:
kubectl delete -f admission-webhook/

# deploy Gatekeeper:
kubectl apply -f opa/
```

I've abbreviated my output in figure 16.14 - when you run the exercise you'll see the OPA Gatekeeper deployment installs a lot more objects, including things we haven't come across yet called CustomResourceDefinitions (CRDs). We'll cover those in more detail in chapter 20 when we look at extending Kubernetes - for now it's enough to know that CRDs let you define new types of object which Kubernetes stores and manages for you.

Remove all the webhooks so we can switch to OPA and apply rules with policies.

```
PS>helm uninstall mutating-webhook
release "mutating-webhook" uninstalled
PS>
PS>helm uninstall validating-webhook
release "validating-webhook" uninstalled
PS>
PS>kubectl delete -f admission-webhook/
service "admission-webhook" deleted
deployment.apps "admission-webhook" deleted
PS>
PS>kubectl apply -f opa/
namespace/gatekeeper-system created

secret/gatekeeper-webhook-server-cert created
service/gatekeeper-webhook-service created
deployment.apps/gatekeeper-audit created
deployment.apps/gatekeeper-controller-manager created
validatingwebhookconfiguration.admissionregistration.k8s.io/gatekeeper-valid
ating-webhook-configuration created
```

OPA Gatekeeper installs a lot of objects, including its own webhook server, TLS cert and validating webhook. There are no default policies, so this just sets up the framework.

Figure 16.14 OPA Gatekeeper takes care of all the tricky parts of running a webhook server

Gatekeeper uses CRDs so you can create templates and constraints as ordinary Kubernetes objects, defined in YAML and deployed with Kubectl. The template contains the generic policy definition in a language called Rego (pronounced "ray-go"). It's an expressive language which lets you evaluate the properties of some input object to check if they meet your requirements. It's another thing to learn, but Rego has some big advantages - policies are fairly easy to read and they live in your YAML files, so they're not hidden in the code of a custom webhook; and there are lots of sample Rego policies to enforce the kind of rules we've looked at in this chapter. Listing 16.6 shows a Rego policy which requires objects to have labels.

Listing 16.6 - requiredLabels-template.yaml, a basic Rego policy

```
# this fetches all the labels on the object and all the
# required labels from the constraint; if required labels
# are missing, that's a violation which blocks object creation

violation[{"msg": msg, "details": {"missing_labels": missing}}] {
    provided := {label | input.review.object.metadata.labels[label]}
    required := {label | label := input.parameters.labels[_]}
    missing := required - provided
    count(missing) > 0
    msg := sprintf("you must provide labels: %v", [missing])
}
```

You deploy that policy with Gatekeeper as a constraint template, and then you deploy a constraint object which enforces the template. In this case the template is called RequiredLabels and it uses parameters to define the labels which are required. Listing 16.7 shows a specific constraint for all Pods to have `app` and `version` labels.

Listing 16.7 - requiredLabels.yaml, a constraint from a Gatekeeper template

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: RequiredLabels
  # the API and kind identify this as
metadata:
  # a Gatekeeper constraint from the
  name: requiredlabels-app
  # RequiredLabels template
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]           # the constraint applies to all Pods
  parameters:
    labels: ["app", "version"] # and requires two labels to be set
```

This is much easier to read and you can deploy many constraints from the same template. The OPA approach lets you build a standard policy library, which users can apply in their application specs without needing to dig into the Rego. In the next exercise you'll deploy the constraint from listing 16.7 with another constraint that requires all Deployments, Services and ConfigMaps to have a `kiamol` label. Then you'll try to deploy a version of the to-do app which fails all those policies.

TRY IT NOW Deploy required label policies with Gatekeeper and see how they get applied.

```
# create the constraint template first:
kubectl apply -f opa/templates/requiredLabels-template.yaml

# then the constraint:
kubectl apply -f opa/constraints/requiredLabels.yaml

# the to-do list spec doesn't meet the policies:
kubectl apply -f todo-list/

# confirm the app isn't deployed:
kubectl get all -l app=todo-web
```

You can see in figure 16.15 that this is a very clean user experience - the objects we're trying to create don't have the required labels so they get blocked and we see the message from the Rego policy in the output from Kubectl.

The constraint template defines the rule with a set of parameters.

Constraints use the template and supply parameter values. These require different labels for different object types.

```

PS>kubectl apply -f opa/templates/requiredLabels-template.yaml
constrainttemplate.templates.gatekeeper.sh/requiredlabels created
```

```

PS>kubectl apply -f opa/constraints/requiredLabels.yaml
requiredlabels.constraints.gatekeeper.sh/requiredlabels-kiamol created
requiredlabels.constraints.gatekeeper.sh/requiredlabels-app created
```

```

PS>
PS>kubectl apply -f todo-list/
```

Error from server ([denied by requiredlabels-kiamol] you must provide labels : {"kiamol"}): error when creating "todo-list\\web.yaml": admission webhook "validation.gatekeeper.sh" denied the request: [denied by requiredlabels-kiamol] you must provide labels: {"kiamol"}
Error from server ([denied by requiredlabels-kiamol] you must provide labels : {"kiamol"}): error when creating "todo-list\\web.yaml": admission webhook "validation.gatekeeper.sh" denied the request: [denied by requiredlabels-kiamol] you must provide labels: {"kiamol"}
Error from server ([denied by requiredlabels-kiamol] you must provide labels : {"kiamol"}): error when creating "todo-list\\web.yaml": admission webhook "validation.gatekeeper.sh" denied the request: [denied by requiredlabels-kiamol] you must provide labels: {"kiamol"}
PS>
PS>`kubectl get all -l app=todo-web`
No resources found in default namespace.

This to-do list spec fails all the policies. The failures are for objects which Kubectl creates - the Deployment, Service and ConfigMap - so the errors are shown in the output.

Figure 16.15 Deployment failures show a clear error message returned from the Rego policy

Gatekeeper evaluates constraints using a validating webhook, and it's very obvious when there are failures in the object you're creating. It's a bit less clear when objects created by controllers fail validation, because the controller itself can be fine. We saw that in section 16.3 and because Gatekeeper uses the same validation mechanism, it has the same issue. You'll see that if you update the to-do app so the Deployment meets the label requirements, but the Pod spec doesn't.

TRY IT NOW Deploy an updated to-do list spec, which has the correct labels for all objects except the Pod.

```

# deploy the updated manifest:
kubectl apply -f todo-list/update/web-with-kiamol-labels.yaml

# show the status of the ReplicaSet:
kubectl get rs -l app=todo-web

# and print the detail:
kubectl describe rs -l app=todo-web
```

```
# remove the to-do app in preparation for the next exercise:  
kubectl delete -f todo-list/update/web-with-kiamol-labels.yaml
```

You'll find in that exercise that the admission policy worked but you only see the problem when you dig into the description for the failing ReplicaSet, as in figure 16.16. That's not such a great user experience. You could fix this with a more sophisticated policy which applies at the Deployment level and checks labels in the Pod template - that would just be a change to the Rego in the constraint template.

The objects Kubectl creates all have the correct labels now, so there are no errors.
 But the Pod spec doesn't have all the required labels, so the ReplicaSet can't create the Pod.

```
PS>kubectl apply -f todo-list/update/web-with-kiamol-labels.yaml  
service/todo-web created  
configmap/todo-web-config created  
deployment.apps/todo-web created  
PS>  
PS>kubectl get rs -l app=todo-web  
NAME DESIRED CURRENT READY AGE  
todo-web-84d79cf6b9 1 0 0 6s  
PS>  
PS>kubectl describe rs -l app=todo-web  
Name: todo-web-84d79cf6b9  
  
Warning FailedCreate 16s (x13 over 36s) replicaset-controller Error creating: admission webhook "validation.gatekeeper.sh" denied the request: [denied by requiredlabels-app] you must provide labels: {"version"}  
PS>  
PS>kubectl delete -f todo-list/update/web-with-kiamol-labels.yaml  
service "todo-web" deleted  
configmap "todo-web-config" deleted  
deployment.apps "todo-web" deleted
```

The error message from Gatekeeper is clear, but it's hidden in the ReplicaSet details.

Figure 16.16 OPA Gatekeeper makes for a better process but it's still a wrapper around validating webhooks

We'll finish this section with a set of admission policies which cover some more production best practices, all of which help to make your apps more secure:

- all Pods should have container probes defined. This is for keeping your apps healthy, but a failed healthcheck could also indicate unexpected activity from an attack;
- Pods can only run containers from approved image repositories. This ensures malicious payloads can't be deployed, if you restrict containers to a set of "golden" repositories with secured production images;
- all containers must have memory and CPU limits set. This prevents a compromised container maxing out the compute resources of the node and starving all the other Pods.

These are generic policies which apply to pretty much every organization. You can add to them with constraints that require network policies for every app and security contexts for every Pod. As you've learned in this chapter not all rules are universal so you might need to be selective on how you apply those constraints. In the next exercise you'll apply the production constraint set to a single namespace.

TRY IT NOW Deploy a new set of constraints and a version of the to-do app where the Pod spec fails most of the policies.

```
# create templates for the production constraints:  
kubectl apply -f opa/templates/production/  
  
# and create the constraints:  
kubectl apply -f opa/constraints/production/  
  
# deploy the new to-do spec:  
kubectl apply -f todo-list/production/  
  
# confirm the Pods aren't created:  
kubectl get rs -n kiamol-ch16 -l app=todo-web  
  
# and show the error details:  
kubectl describe rs -n kiamol-ch16 -l app=todo-web
```

You can see in figure 16.17 that the Pod spec fails all the rules except one - my image repository policy allows any images from Docker Hub in the `kiamol` organization, so the to-do app image is valid. But there's no version label, no health probes and no resource limits, and this spec is not fit for production.

This is a useful set of constraints which require generic security controls in Pod specs.

```
PS>kubectl apply -f opa/templates/production/
constrainttemplate.templates.gatekeeper.sh/policycontainerprobes created
constrainttemplate.templates.gatekeeper.sh/policyimagerepository created
constrainttemplate.templates.gatekeeper.sh/policyresourcelimits created
PS>
PS>kubectl apply -f opa/constraints/production/
policycontainerprobes.constraints.gatekeeper.sh/container-probes created
policyimagerepository.constraints.gatekeeper.sh/image-repository created
policyresourcelimits.constraints.gatekeeper.sh/resource-limits created
PS>
PS>kubectl apply -f todo-list/production/
namespace/kiamol-ch16 created
service/todo-web created
configmap/todo-web-config created
deployment.apps/todo-web created
PS>
PS>kubectl get rs -n kiamol-ch16 -l app=todo-web
NAME           DESIRED   CURRENT   READY   AGE
todo-web-84d79cf6b9   1         0         0       6s
PS>
PS>kubectl describe rs -n kiamol-ch16 -l app=todo-web
Name:           todo-web-84d79cf6b9
from similar events): Error creating: admission webhook "validation.gatekeeper.sh" denied the request: [denied by requiredlabels-app] you must provide labels: {"version"} [denied by container-probes] Container <web> in your <Pod> <todo-web-84d79cf6b9-lrh5c> has no <readinessProbe> [denied by container-probes] Container <web> in your <Pod> <todo-web-84d79cf6b9-lrh5c> has no <livenessProbe> [denied by resource-limits] container <web> has no resource limits
```

The ReplicaSet gets created but no Pods - the spec is missing a version label, as well as readiness and liveness probes, and resource limits.

Figure 16.17 All constraints are evaluated and you see the full list of errors in the Rego output

Just to prove those policies are achievable and OPA Gatekeeper will actually let the to-do app run, you can apply an updated spec which meets all the rules for production. If you compare the YAML files in the production folder and the update folder, you'll see the new spec just adds the required fields to the Pod template; there are no significant changes in the app.

TRY IT NOW Apply a production-ready version of the to-do spec and confirm the app really runs.

```
# this spec meets all production policies:
kubectl apply -f todo-list/production/update

# wait for the Pod to start:
kubectl wait --for=condition=ContainersReady pod -l app=todo-web -n kiamol-ch16
```

```
# confirm it's running:  
kubectl get pods -n kiamol-ch16 --show-labels  
  
# get the URL for the app and browse:  
kubectl get svc todo-web -n kiamol-ch16 -o  
  jsonpath='http://{.status.loadBalancer.ingress[0].*}:8019'
```

Figure 16.18 shows the app running, after the updated deployment has been permitted by OPA Gatekeeper.

This updated Pod spec has all the required fields to comply with production policy.
The Pod starts and it has the expected app and version labels.

```
PS>kubectl apply -f todo-list/production/update  
namespace/kiamol-ch16 unchanged  
service/todo-web unchanged  
configmap/todo-web-config unchanged  
deployment.apps/todo-web configured
```

```
PS>  
PS>kubectl wait --for=condition=ContainersReady pod -l app=todo-web -n kiamol-ch16  
pod/todo-web-f778599b4-qcwsf condition met  
PS>
```

```
PS>kubectl get pods -n kiamol-ch16 --show-labels  
NAME           READY   STATUS    RESTARTS   AGE    LABELS  
todo-web-f778599b4-qcwsf  1/1     Running   0          23s   app=todo-web,pod-template-hash=f778599b4,version=0.16.0
```

```
PS>  
PS>kubectl get svc todo-web -n kiamol-ch16 -o jsonpath='http://{.status.loadBalancer.ingress[0].*}:8019'  
http://localhost:8019  
PS>  
PS>
```

Finally we have a working app - although the to-do list is empty, which is probably not accurate.

Figure 16.18 Constraints are powerful but you need to make sure apps can actually comply

Open Policy Agent is a much cleaner way to apply admission controls than custom validating webhooks, and the sample policies we've looked at are only some simple ideas to get you started. Gatekeeper doesn't have mutation functionality, but you can combine it with your own webhooks if you have a clear case to modify specs. You could use constraints to ensure every pod spec includes an application-profile label, and then mutate specs based on your profiles -

setting your .NET Core apps to run as a non-root user and switching to a read-only filesystem for all your Go apps.

Securing your apps is about closing down exploit paths, and a thorough approach includes all the tools we've covered in this chapter and more besides. We'll finish up with a look at a secure Kubernetes landscape.

16.5 Understanding security-in-depth in Kubernetes

Build pipelines can be compromised, container images can be modified, containers can run vulnerable software as privileged users, and attackers with access to the Kubernetes API could even take control of your cluster. You won't know your app is 100% secure until it has been replaced and you can confirm there were no security breaches during its operation. Getting to that happy place means applying security in depth across your whole software supply chain. This chapter has focused on securing apps at runtime, but you should start before that by scanning container images for known vulnerabilities.

Security scanners look inside an image, identify the binaries and check them on CVE (Common Vulnerabilities and Exposures) databases. Scans tell you if there are known exploits in the application stack, dependencies or operating system tools in your image. Commercial scanners have integrations with managed registries (you can use Aqua Security with Azure Container Registry), or you can run your own (Harbor is the CNCF registry project, and it supports the open-source scanners Clair and Trivy).

You can set up a pipeline where images are only pushed to a production repository if the scan is clear - combine that with a repository admission policy and you can effectively ensure that containers only run if the image is safe. A secure image running in a securely configured container is still a target though, and you should look at runtime security with a tool that monitors containers for unusual activity, and can generate alerts or shut down suspicious behavior. Falco is the CNCF project for runtime security, and there are supported commercial options from Aqua and Sysdig (among others).

Overwhelmed? You should think about securing Kubernetes as a roadmap which starts with the techniques I've covered in this chapter. You can adopt security contexts first, then network policies and then move onto admission control when you're clear about the rules that matter to you. Role-based access control is the next stage, which we cover in chapter 17. Security scanning and runtime monitoring are further steps you can take if your organization has enhanced security requirements. But I won't throw anything more at you now - let's tidy up and get you ready for the lab.

TRY IT NOW Delete all the objects we created.

```
kubectl delete -f opa/constraints/ -f opa/templates/ -f opa/gatekeeper.yaml
kubectl delete all,ns,secret,networkpolicy -l kiamol=ch16
```

16.6 Lab

At the start of the chapter I said that volume mounts for host paths are a potential attack vector, but we didn't address that in the exercises - so we'll do it in the lab. This is a perfect scenario for admission control, where Pods should be blocked if they use volumes which mount sensitive paths on the host. We'll use OPA Gatekeeper and I've written the Rego for you, so you just need to write a constraint.

- start by deploying `gatekeeper.yaml` in the lab folder
- then deploy the constraint template in `restrictedPaths-template.yaml` - you'll need to look at the spec to see how to build your constraint
- write and deploy a constraint which uses the template and restricts these host paths: `/`, `/bin` and `/etc`. The constraint should only apply to Pods with the label `kiamol=ch16-lab`.
- deploy `sleep.yaml` in the lab folder and your constraint should prevent the Pod from being created because it uses restricted volume mounts.

This one is fairly straightforward, although you'll need to read about *match expressions* which is how Gatekeeper implements label selectors. My solution is up on GitHub:

<https://github.com/sixeyed/kiamol/blob/master/ch16/lab/README.md>

17

Securing resources with role-based access control

You have complete control over your lab cluster - you can deploy workloads, read secrets, even delete control plane components if you want to see how quickly they return. You don't want anyone to have that much power in a production cluster, because if they have full admin control then it's really their cluster. Their account could get compromised and then some rogue party deletes all your apps and turns your cluster into their personal Bitcoin miner. Kubernetes supports least-privilege access with role-based access control (RBAC). In this chapter you'll learn how RBAC works and some of the challenges that come with restricting access.

RBAC applies to end-users working with Kubectl and to internal components using the Kubernetes API with service account tokens. You need a different RBAC approach for each of those which we'll cover in this chapter, together with the best practices. You'll also learn how Kubernetes gets the credentials for external users and how you can manage end-users inside the cluster if you don't have an external authentication system. RBAC is a straightforward model but there are lots of moving pieces and it can be hard to keep track of who can do what, so we'll finish the chapter looking at management tools.

17.1 How Kubernetes secures access to resources

RBAC works by granting permissions to perform actions on resources - every resource type can be secured, so you can set up permissions to get Pod details, list Services, and delete Secrets. You apply permissions to a subject which could be a user, a system account or a group - but you don't apply them directly because that would create an unmanageable sprawl of permissions. You set permissions in a role and you apply a role to one or more subjects with a role binding - figure 17.1 shows the structure.

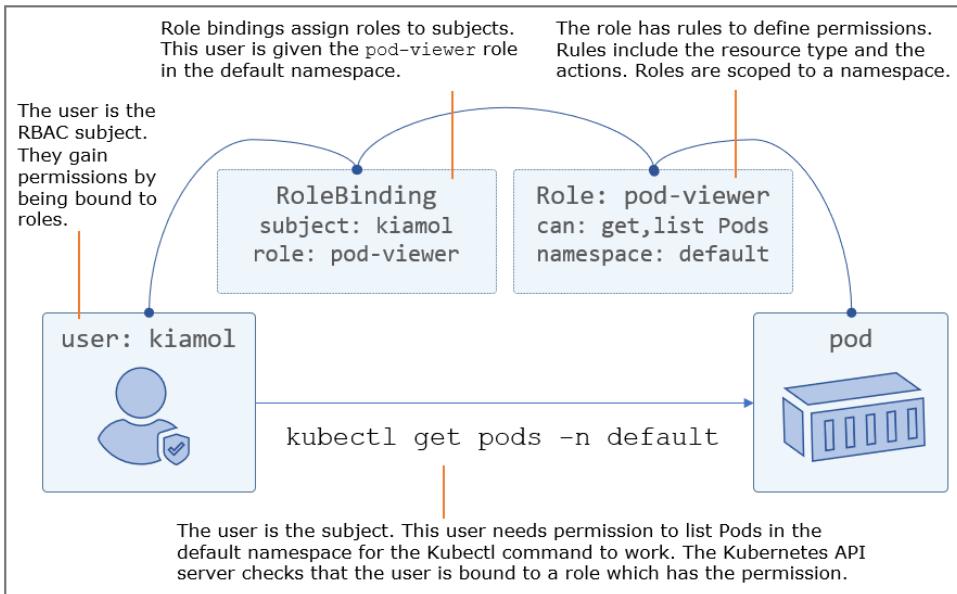


Figure 17.1 RBAC is a security abstraction, object permissions are granted through roles and bindings

Some Kubernetes resources are specific to a namespace and some are cluster-wide, so the RBAC structure actually has two sets of objects to describe and allocate permissions: Role and RoleBinding objects work on namespaced objects and ClusterRole and ClusterRoleBinding objects work on the whole cluster. RBAC is technically an optional component in Kubernetes, but it is enabled in almost all platforms now - you can check that it's enabled and see some of the default roles using standard Kubectl commands.

TRY IT NOW You can check the features in your cluster by printing the API versions it supports. Docker Desktop, K3s and all the cloud platforms support RBAC by default.

```
# switch to this chapter's folder:
cd ch17

# PowerShell doesn't have a grep command, run this on Windows to add it:
..\\grep.ps1

# check the API versions include RBAC:
kubectl api-versions | grep rbac

# show the admin cluster roles:
kubectl get clusterroles | grep admin

# and show the details of the cluster admin:
kubectl describe clusterrole cluster-admin
```

You see in figure 17.2 that there are a whole lot of built-in roles which are cluster-wide. One is cluster-admin and that's the role you have in your lab cluster - it has permissions for all actions (Kubernetes calls them *verbs*) on all resources, and that's why you can do anything you like. The next most powerful role is admin which has pretty much all permissions on all objects, but limited to a single namespace.

This adds a grep command to PowerShell - you only need it on Windows.

My cluster supports the RBAC APIs which means RBAC is enabled.

```
PS>cd ch19
PS>
PS> .\grep.ps1
PS>
PS>kubectl api-versions | grep rbac
rbac.authorization.k8s.io/v1
rbac.authorization.k8s.io/v1beta1
```

PS>kubectl get clusterroles | grep admin

admin		2020-07-20T21:28:24Z
cluster-admin		2020-07-20T21:28:24Z
system:aggregate-to-admin		2020-07-20T21:28:24Z
system:kubelet-api-admin		2020-07-20T21:28:24Z

PS>kubectl describe clusterrole cluster-admin

Name:	cluster-admin		
Labels:	kubernetes.io/bootstrapping=rbac-defaults		
Annotations:	rbac.authorization.kubernetes.io/autoupdate: true		
PolicyRule:			
Resources	Non-Resource URLs	Resource Names	Verbs
-----	-----	-----	-----
.	[]	[]	[*]
	[*]	[*]	[*]

These are some of the default cluster roles - admin and cluster-admin are meant for end-users, the other are used by internal components.

These asterisks mean the cluster-admin role can do everything.

Figure 17.2 RBAC has a default set of roles and bindings for users and service accounts

That's all very well, but *who* has the cluster-admin role? You don't log in to Kubectl with a username and password for your own lab and there is no user object in Kubernetes, so how does Kubernetes know who the user is? Kubernetes does not authenticate end-users; it relies on external identity providers and it trusts them to authenticate. In a production system your cluster would be configured to use your organization's existing authentication system - Active Directory, LDAP and OpenID Connect (OIDC) are all options.

Cloud platforms integrate Kubernetes with their own authentication, so AKS users authenticate with Azure AD accounts. You can configure your own OIDC provider but the setup

is quite heavy so in our lab cluster we'll stick with certificates. Kubernetes is able to issue client certificates for end users, which you request with a username. When the Kubernetes API server sees that certificate in incoming requests, it trusts the issuer (itself) and accepts that the user is who they say they are. We'll start by generating a new certificate for a user who will have limited access in the cluster.

TRY IT NOW Creating certificate requests for Kubernetes to sign needs a few steps, which are scripted in a container image. Run a Pod from that image to generate certs and then copy the cert and key onto your local machine.

```
# run the cert generator:
kubectl apply -f user-cert-generator.yaml

# wait for the container to start:
kubectl wait --for=condition=ContainersReady pod user-cert-generator

# print the logs:
kubectl logs user-cert-generator --tail 3

# copy the files onto your local disk:
kubectl cp user-cert-generator:/certs/user.key user.key
kubectl cp user-cert-generator:/certs/user.crt user.crt
```

You'll see in the output from the first command that the exercise creates some roles and bindings of its own. The Pod container runs a script which uses Kubectl to issue the client certificate, and that's a privileged action so the manifest makes sure the Pod has the permissions it needs. The steps are a bit convoluted, which is why I've wrapped them in a container image - the script `user-cert-generator/start.sh` does the work if you want to dig into the detail. My output in figure 17.3 shows that the certificate and key file are on my local machine, which is all I need to access my cluster as an authenticated user.

This runs a script to generate a client certificate signed by Kubernetes, which can be used to authenticate to the cluster. The script needs permissions to create the cert.

```
PS>kubectl apply -f user-cert-generator.yaml
serviceaccount/user-cert-generator created
pod/user-cert-generator created
clusterrole.rbac.authorization.k8s.io/create-approve-csr created
clusterrolebinding.rbac.authorization.k8s.io/user-cert-generator created
PS>
PS>kubectl logs user-cert-generator --tail 3
-----
Cert generated: /certs/user.key and /certs/user.crt
-----
PS>
PS>kubectl cp user-cert-generator:/certs/user.key user.key
tar: removing leading '/' from member names
PS>
PS>kubectl cp user-cert-generator:/certs/user.crt user.crt
tar: removing leading '/' from member names
PS>
PS>cat user.crt
-----BEGIN CERTIFICATE-----
MIIDHjCCAgagAwIBAgIQODxSHRSqlfM11fLHqJHG2DANBgkqhkiG9w0BAQsFADAV
MRMwEQYDVQQDEwprdWJlcM5ldGVzMB4XDTIwMDcyMTEwMzM10FoXDTIxMDcyMTEw
MzM10EwIBHElALV11ERbMCWIVvD7ANDmVRAATPvDPTlERTMAGC11ERbMC
-----
```

Users need the certificate and key file to authenticate.

The cert is a standard X.509 certificate, issued by Kubernetes.

Figure 17.3 Kubernetes can issue its own client certs for new authenticated users

If you're into OpenSSL and certificates you can decode that cert file and you'll see the common name is reader@kiamol.net - Kubernetes treats this as the name of the user, and that will be the subject we can use to apply RBAC. Permissions are all additive with RBAC, so subjects start with no permissions and as role bindings are applied they end up with the sum total of all role permissions. The RBAC model is grant-only, you can't deny permissions - the absence of a permission is the same as deny. We can set up a Kubectl context with the new user certs and confirm they start with zero access.

TRY IT NOW Create a new context in Kubectl using the generated cert as the credentials and confirm you can use it to access the cluster.

```
# set the credentials for a new context from the client cert:
kubectl config set-credentials reader --client-key=~/user.key --client-
certificate=~/user.crt --embed-certs=true

# set the cluster for the context:
kubectl config set-context reader --user=reader --cluster=$(kubectl config view -o
    jsonpath='{.clusters[0].name}')

# try to deploy a Pod using the new context -
```

```
# if your cluster is configured with authentication
# this won't work as the provider tries to authenticate:
kubectl apply -f sleep/ --context reader

# impersonate the user to confirm their permissions:
kubectl get pods --as reader@kiamol.net
```

There are two parts to a context in Kubectl - the user credentials and the cluster to connect to. In figure 17.4 you can see the user is configured with the client certificates which are embedded into the Kubectl config file. If you use client certificates like this then your config file is sensitive, if someone obtains a copy then they have all they need to connect to your cluster with one of your contexts.

Store the client certs as credentials in Kubectl and create a new context using the credentials. This authenticates using the certificate issued by Kubernetes.

```
PS>kubectl config set-credentials reader --client-key=./user.key
--client-certificate=./user.crt --embed-certs=true
User "reader" set.
PS>
PS>kubectl config set-context reader --user=reader --cluster $(kubectl config view -o jsonpath='{.clusters[0].name}')
Context "reader" created.
PS>
PS>kubectl apply -f sleep/ --context reader
Error from server (Forbidden): error when retrieving current configuration of:
Resource: "apps/v1, Resource=deployments", GroupVersionKind: "apps/v1, Kind=Deployment"
Name: "sleep", Namespace: "default"
from server for: "sleep\\sleep.yaml": deployments.apps "sleep" is forbidden: User "reader@kiamol.net" cannot get resource "deployments" in API group "apps" in the namespace "default"
PS>
PS>kubectl get pods --as reader@kiamol.net
Error from server (Forbidden): pods is forbidden: User "reader@kiamol.net" cannot list resource "pods" in API group "" in the name space "default"
```

The user is authenticated but not authorized - RBAC error messages state the user name and the permissions they lack.

Figure 17.4 Authenticated but unauthorized - users start with no RBAC roles

The last command in the exercise uses impersonation to confirm that the new user doesn't have any permissions, but Kubernetes doesn't store users. You can use any random string for the username in the `as` parameter and output will tell you it doesn't have permissions. What Kubernetes actually looks for is any role binding where the user name matches the name in the request. If there are no bindings then there are no permissions so the action is blocked, whether

the username exists in the authentication system or not. Listing 17.1 shows a role binding we can apply to give the new user read-only access to resources in the default namespace.

Listing 17.1 - reader-view-default.yaml, applying permissions with a role binding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: reader-view
  namespace: default          # this is the scope of the binding
subjects:
- kind: User
  name: reader@kiamol.net    # the subject is the new user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: view                 # gives them the view role for
  apiGroup: rbac.authorization.k8s.io # objects in the default namespace
```

This is a good way to get started with RBAC - use the pre-defined cluster roles and bind them to subjects for a specified namespace. As we go through the chapter you'll see how to build custom roles which are great when you want explicit access permissions, but they get hard to manage at scale. Role bindings abstract the subject from the roles they have, so you can make changes to access without changing roles or objects. When you deploy the role binding from listing 17.1 the new user will be able to view resources in the default namespace.

TRY IT NOW Apply a role binding and impersonate the new user to confirm that they have read-only access to resources.

```
# deploy a sleep Pod as your normal user:
kubectl apply -f sleep/

# deploy the role binding so the reader can view the Pod:
kubectl apply -f role-bindings/reader-view-default.yaml

# confirm the user sees Pods in the default namespace:
kubectl get pods --as reader@kiamol.net

# but is blocked from the system namespace:
kubectl get pods -n kube-sytsem --as reader@kiamol.net

# and can't delete Pods - this will fail:
kubectl delete -f sleep/ --as reader@kiamol.net
```

You can see the view role in action in figure 17.5 - the new user can list Pods, but only in the default namespace. The role has no permissions to delete objects so the reader user can see Pods but not delete them.

Apply a role binding giving the reader user access to view resources in the default namespace.

The user can list Pods, a permission which is in the view role.

```
PS>kubectl apply -f sleep/
deployment.apps/sleep created
PS>
PS>kubectl apply -f role-bindings/reader-view-default.yaml
rolebinding.rbac.authorization.k8s.io/reader-view created
PS>
PS>kubectl get pods --as reader@kiamol.net
NAME           READY   STATUS    RESTARTS   AGE
sleep-85fdd4cf75-xngp7  1/1     Running   0          15s
user-cert-generator  1/1     Running   0          126m
PS>
PS>kubectl get pods -n kube-system --as reader@kiamol.net
Error from server (Forbidden): pods is forbidden: User "reader@kiamol.net" cannot list resource "pods" in API group "" in the namespace "kube-system"
PS>
PS>kubectl delete -f sleep/ --as reader@kiamol.net
Error from server (Forbidden): error when deleting "sleep\sleep.yaml": deployments.apps "sleep" is forbidden: User "reader@kiamol.net" cannot delete resource "deployments" in API group "apps" in the namespace "default"
```

But the role is only scoped to the default namespace so the user has no access to kube-system.

And the view role does not have permission to delete resources.

Figure 17.5 Role bindings are scoped so the permissions only apply to one namespace

The disconnect between users and roles feels slightly odd and it can lead to problems. Kubernetes has no real integration with the authentication system so it doesn't validate that usernames are correct or that groups exist. It's strictly authorization you configure in RBAC as far as end users are concerned. But you know from chapter 16 that you also need to secure the cluster from inside for apps which use the Kubernetes API, and the cluster manages both authentication and authorization for service accounts.

17.2 Securing resource access within the cluster

Every namespace has a default service account created automatically, and any Pods which don't specify a service account will use the default. The default service account is just like any other RBAC subject - it starts off with no permissions until you add some, and you add them using the same role bindings and cluster role bindings that you use with end-user subjects. Service accounts are different because apps generally need a much more limited set of permissions, so the best practice is to create a dedicated service account for each component.

Creating a service account and setting up roles and bindings adds a lot of overhead, but remember this is not about restricting access to resources in your application model. You can include ConfigMaps and Secrets and whatever else you need in your manifests and that's not affected by the service account permissions at runtime. RBAC for service accounts is purely about securing apps which use the Kubernetes API server - apps like Prometheus which query the API to get lists of Pods. That should be rare with your standard business apps, so this is just about securing the special cases, where you'll run into problems if you use the default service account in the same namespace for every app. We'll start this section creating a new namespace to look at its default service account.

TRY IT NOW Create a new namespace and check the permissions of its service account - fixing a bug with Docker Desktop if you use it.

```
# on Docker Desktop for Mac run this to fix the RBAC setup:
kubectl patch clusterrolebinding docker-for-desktop-binding --type=json --patch
  $'[{"op":"replace", "path":"/subjects/0/name",
    "value":"system:serviceaccounts:kube-system"}]'

# OR on Docker Desktop for Windows:
kubectl patch clusterrolebinding docker-for-desktop-binding --type=json --patch
  '[{"op":"replace", "path":"/subjects/0/name\",
    \"value\":\"system:serviceaccounts:kube-system\"}]'

# create the new namespace:
kubectl apply -f namespace.yaml

# list service accounts:
kubectl get serviceaccounts -n kiamol-ch17

#check permissions for your own account:
kubectl auth can-i "*" "*"

# and the new service account:
kubectl auth can-i "*" "*" --as system:serviceaccount:kiamol-ch17:default
kubectl auth can-i get pods -n kiamol-ch17 --as system:serviceaccount:kiamol-
  ch17:default
```

The `can-i` command is a very useful way to check permissions without actually affecting any objects. You can see in figure 17.6 that the command can be combined with impersonation and a namespace scope to show the permissions for another subject - which could be a user or a service account.

Docker Desktop has an RBAC bug which gives service accounts more permissions than they should have - this fixes it.

Every namespace has a default service account which the cluster creates automatically.

```

PS>kubectl patch clusterrolebinding docker-for-desktop-binding --type=json --patch '[{"op":"replace", "path":"/subjects/0/name", "value":"system:serviceaccounts:kube-system"}]'
clusterrolebinding.rbac.authorization.k8s.io/docker-for-desktop-binding patched
PS>
PS>kubectl apply -f namespace.yaml
namespace/kiamol-ch19 created
PS>
PS>kubectl get serviceaccounts -n kiamol-ch19
NAME      SECRETS   AGE
default   1          9s
PS>
PS>kubectl auth can-i "*" "*"
yes
PS>
PS>kubectl auth can-i "*" "*" --as system:serviceaccount:kiamol-ch19:default
no
PS>
PS>kubectl auth can-i get pods -n kiamol-ch19 --as system:serviceaccount:kiamol-ch19:default
no

```

Using `can-i` with impersonation confirms the new service account does not have cluster admin permissions.

And can't even list Pods in its own namespace.

Figure 17.6 A system controller ensures every namespace has a default service account

You saw in the exercise that new service accounts start with zero permissions, which is where the trouble starts if you use the default account for all your apps. Each app will need its own permissions so they all get added to the default account which soon ends up with far more power than it needs. If any of the apps using that service account gets compromised, the attacker finds themselves with a bonus set of roles. The certificate generator from section 17.1 is a good example - it uses the Kubernetes API to issue client certificates, which is a privileged action. Listing 17.2 shows the cluster role the app's service account uses to get the permissions it needs - the full manifest also contains a cluster role binding.

Listing 17.2 - user-cert-generator.yaml, a custom cluster role for certificate generation

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: create-approve-csr

```

```

rules:
- apiGroups: ["certificates.k8s.io"]          # generating certs neds
  resources: ["certificatesigningrequests"]    # permission to create
  verbs: ["create", "get", "list", "watch"]     # a signing request
- apiGroups: ["certificates.k8s.io"]
  resources: ["certificatesigningrequests/approval"]
  verbs: ["update"]                           # and approve the request
- apiGroups: ["certificates.k8s.io"]
  resources: ["signers"]                      # using the cluster to sign
  resourceNames: ["kubernetes.io/kube-apiserver-client"]
  verbs: ["approve"]

```

Generating certificates needs permission to create signing requests and approve them using the API Server issuer. If I was being lazy I could bind this role to the default service account on the justification that this app isn't publicly accessible so the attack surface is small, and no other apps are using that account. But of course it's the default service account and Kubernetes mounts the account token by default, so if someone deploys a vulnerable web app in this same namespace, it will have permission to generate user certificates so attackers could generate their own credentials. But I didn't do that, I created a dedicated service account just for the cert generator.

TRY IT NOW Confirm that the custom service account for the cert generator app has permission to create certificate signing requests but standard service accounts don't have that permission.

```

# run a can-i check inside the cert generator Pod:
kubectl exec user-cert-generator -- kubectl auth can-i create csr --all-namespaces

# and use impersonation for the same check:
kubectl auth can-i create csr -A --as system:serviceaccount:default:user-cert-
generator

# confirm the new service account doesn't have the permission:
kubectl auth can-i create csr -A --as system:serviceaccount:kiamol-ch17:default

```

You'll see in that exercise that only the cert generator service account has the certificate permissions. In fact those are the only permissions that account has, so you can't use it to list namespaces or delete persistent volume claims or anything else. You also see in figure 17.7 that the syntax for referring to a service account is different from other resource references in Kubernetes - `system:serviceaccount` is the prefix, followed by the namespace and account name separated by colons.

Inside the Pod the Kubectl command uses the context of a custom service account.

Outside the Pod you can confirm the service account has permission to create a CertificateSigningRequest by using impersonation.

```
PS>kubectl exec user-cert-generator -- kubectl auth can-i create csr --all-namespaces
yes
PS>
PS>kubectl auth can-i create csr -A --as system:serviceaccount:default:user-cert-generator
yes
PS>
PS>kubectl auth can-i create csr -A --as system:serviceaccount:kiamol-ch19:default
no
```

And to confirm that the default service account in the new namespace cannot.

Figure 17.7 Using separate service accounts for apps ensures the least-privilege approach

Issuing client certs is not a typical app requirement, but creating a separate service account for every app that uses the Kubernetes API is still a best practice. We've got two more examples in this section to show you how apps might need to use the API and how to secure them. The key thing to understand in RBAC is the scope of your bindings, to be sure you're applying permissions at the right level - typically for a single namespace. The first example of that is for a simple web app which lists Pods and lets you delete them.

TRY IT NOW Run the kube explorer app which lists Pods in a web UI. This deployment uses a custom service account and role for permissions.

```
# deploy the app and the RBAC rules:
kubectl apply -f kube-explorer/

# wait for the Pod:
kubectl wait --for=condition=ContainersReady pod -l app=kube-explorer

# get the URL for the app:
kubectl get svc kube-explorer -o
    jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8019'

# browse to the app and confirm you can view and delete
# Pods in the default namespace; then add ?ns=kube-system
# to the URL and you'll see an error
```

My output is in figure 17.8 where you can see the app is happily listing and deleting Pods for the default namespace. When you switch to a different namespace you'll see an authorization error which comes from the Kubernetes API (it's an HTTP API so you actually get a 403 forbidden response). The app uses the service account token mounted in the Pod to authenticate, and the

account only has list and delete Pod permissions for the default namespace, no other namespaces.

The explorer app lists Pods - it has its own service account, role and role binding to get list and delete permissions for the default namespace.

```
PS>kubectl apply -f kube-explorer/
serviceaccount/kube-explorer created
deployment.apps/kube-explorer created
role.rbac.authorization.k8s.io/default-pod-reader created
rolebinding.rbac.authorization.k8s.io/kube-explorer-default created
service/kube-explorer created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=kube-explorer
pod/kube-explorer-5f778d9f6c-185hz condition met
PS>
PS>kubectl get svc kube-explorer -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8019'
http://localhost:8019
```

KIAMOL - KubeExplorer localhost:8019

Pod List
Namespace: default

Name	Image	Node	Phase	Action
kube-explorer-5f778d9f6c-185hz	kiamol/ch19-kube-explorer	docker-desktop	Running	Delete
sleep-85ffd4cf75-xngp7	kiamol/ch03-sleep	docker-desktop	Running	Delete

KIAMOL - KubeExplorer localhost:8019/Delete?name=sleep-85ffd4cf75-xngp7

Pod Deleted!
Namespace: default
Pod: sleep-85ffd4cf75-xngp7

Browse to the app and you'll see a list of Pods - click the Delete link and the Pod gets deleted. Try to use a different namespace by adding ?ns=kube-system to the URL and you'll see a permissions error.

Figure 17.8 An app with its own service account and just enough permissions to do its job

Little apps like this take no time to build as most languages have a Kubernetes library which takes care of authentication by using the token in the default path. They can be very useful for sharing a limited view of the cluster for teams who don't need Kubectl access, and who don't need to learn how Kubernetes works just to see what's running. The app needs RBAC permissions to list Pods and show their details and delete them. Right now those permissions are in a role which is bound to the default namespace - to make other namespaces available we need to add more roles and bindings.

Listing 17.3 shows the new rules which grant permission to get and list Pods in the kube-system namespace. The most important thing to take away here is that the namespace in the

role metadata is not just where the role gets created, it's also the scope where the role applies. This role grants access to Pods in the kube-system namespace.

Listing 17.3 - rbac-with-kube-system.yaml, applying a role to the system namespace

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: system-pod-reader
  namespace: kube-system          # scoped to the system namespace
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]           # the API group of the object spec
                                  # Pods are in the core group which
                                  # is identified with an empty string
```

The role binding which adds that role to the app's service account is in the same manifest file, but I've split it into two listings to examine them separately and make sense of all the namespaces. The namespace for the role is the scope of the permissions; the role binding refers to the role and it needs to be in the same namespace, but it also refers to the subject and that can be in a different namespace. In listing 17.4 the role binding is created in the `kube-system` namespace along with the role - between them they provide access to the Pods in that namespace. The subject is the app's service account which is in the `default` namespace.

Listing 17.4 - rbac-with-kube-system.yaml, role binding to a subject

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: kube-explorer-system
  namespace: kube-system          # needs to match the role
subjects:
- kind: ServiceAccount
  name: kube-explorer            # the subject can be in a
  namespace: default             # different namespace
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: system-pod-reader
```

You can see how things spiral when you have permission requirements which aren't a good fit for the standard cluster roles. In this case permissions are different for different namespaces, so that needs a set of roles and role bindings for each namespace. When you deploy these new rules, the app will be able to show system Pods - access rules are evaluated when the API call is made, so there are no changes to the app or the service account; the new permissions take effect right away.

TRY IT NOW Add access rules for system Pods to the explorer app.

```
# apply the new role and binding
kubectl apply -f kube-explorer/update/rbac-with-kube-system.yaml
```

```
# refresh the explorer app with the path /?ns=kube-system
# you can see the Pods now but you can't delete them
```

You can see in figure 17.9 that the payback for managing lots of roles and bindings is that you can build very fine-grained access policies. This app can list and delete Pods in the default namespace, but it can only list Pods in the system namespace. No other apps can get those permissions by accident; they would need to specify the kube-explorer service account and be deployed by someone who has access to use that account.

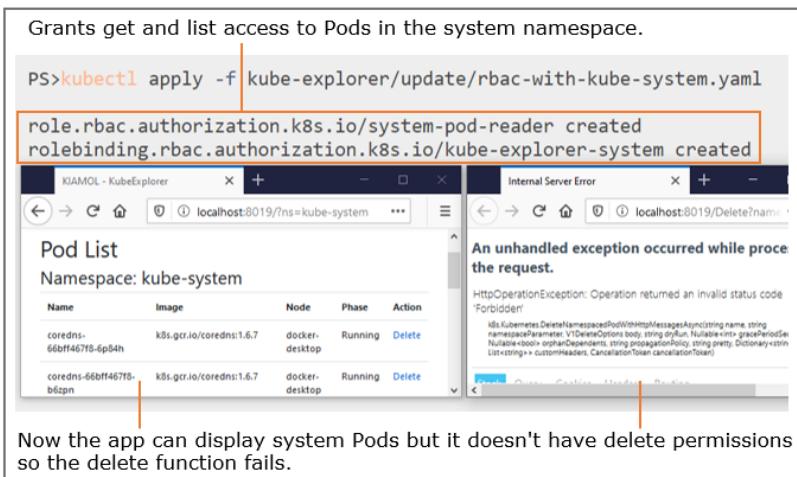


Figure 17.9 RBAC rules take immediate effect when you apply or remove a binding

The last example in this section shows how apps might need to get some configuration data from the cluster which isn't in a ConfigMap or Secret - we'll use the to-do app again. This version displays a banner message on the homepage, and it gets the content for the banner from the `kiamol` label on the namespace where the Pod is running. An init container sets up Kubectl with the service account token, grabs the label value for the namespace and writes it to a config file which the app container picks up. It's not a very realistic scenario, but it shows you how to inject data from the cluster into app configuration.

TRY IT NOW Deploy the new to-do list app and confirm the banner message is populated from the namespace label.

```
# print the label value on the namespace:
kubectl get ns kiamol-ch17 --show-labels

# deploy the to-do list app:
kubectl apply -f todo-list/

# wait for the Pod to start:
```

```
kubectl wait --for=condition=ContainersReady pod -l app=todo-web -n kiamol-ch17
# print the logs of the init container:
kubectl logs -l app=todo-web -c configurator --tail 1 -n kiamol-ch17
# get the URL and browse to the app:
kubectl get svc todo-web -n kiamol-ch17 -o
  jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8020'
```

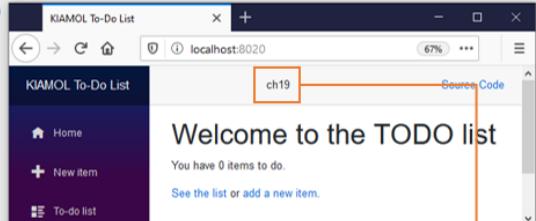
My output is in figure 17.10 where you can see the init container has fetched the metadata from the namespace. The application container doesn't use the Kubernetes API, but the service account token gets mounted to all containers in the Pod, so potentially the app could be compromised and an attacker could use the Kubernetes API in the context of the service account.

The label value should show in the app's banner.

The value is read by an init container and written to a config file. The role provides read access to the namespace.

```
PS>kubectl get ns kiamol-ch19 --show-labels
NAME      STATUS   AGE    LABELS
kiamol-ch19  Active  140m  kiamol=ch19
PS>
PS>kubectl apply -f todo-list/
serviceaccount/todo-web created
deployment.apps/todo-web created
clusterrole.rbac.authorization.k8s.io/ch19-reader created
clusterrolebinding.rbac.authorization.k8s.io/todo-web-reader created
service/todo-web created
PS>
PS>kubectl logs -l app=todo-web -c configurator --tail 1 -n kiamol-ch19
** Namespace label: ch19. Written to config file **
PS>
```

```
PS>kubectl get svc todo-web -n kiamol-ch19 -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8020'
```



The init container logs show that it has access to the namespace.

The app shows the value in the banner.

Figure 17.10 Applications might need to access configuration data from other cluster resources

There's not much an attacker could do if they did get access to the API from the to-do list app. The app uses a dedicated service account and that account has one single permission - it can get the details of the namespace `kiamol-ch17`. Listing 17.5 shows how rules inside roles and cluster roles can restrict permissions to named resources.

Listing 17.5 - 02-rbac.yaml, rules for named resources

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: ch17-reader
rules:
- apiGroups: [""]
  resources: ["namespaces"]
  resourceNames: ["kiamol-ch17"]
  verbs: ["get"]
```

One drawback with RBAC is that resources need to exist before the rules can be applied, so in this case the namespace and service account must exist before the role and role binding can be created. When you apply a folder of manifests with Kubectl it doesn't look for dependencies and create resources in the correct order, it just applies the files in filename order. That's why the RBAC manifest in listing 17.6 is called `02-rbac.yaml`, to ensure it gets created after the service account and namespace exist.

We've looked at drilling down into specific permissions for individual apps, but the other major function of RBAC is to apply the same set of rules to groups of subjects - we'll move onto that next.

17.3 Binding roles to groups of users and service accounts

Role bindings and cluster role bindings can be applied to groups, and both users and service accounts can belong to groups - although they work differently. End-users are authenticated outside of Kubernetes and the API trusts the username and group information which is presented. Users can be members of multiple groups, and the group name and membership are managed by the authentication system. Service accounts are more restricted; they always belong to two groups - a group for all service accounts in the cluster, and a group for all the service accounts in a namespace.

Groups are another kind of subject for a binding, so the spec is the same except that you bind a role - or cluster role - to a group instead of a user or service account. Kubernetes doesn't validate the group name, so it's down to you to make sure the group in the binding matches the group set by the authentication system. Certificates can include group information, so we can create use certs for users who belong to different groups.

TRY IT NOW Use the cert generator to create some more authenticated users, this time with group membership set in the certificate, for a Site Reliability Engineer (SRE) group and a group for testers.

```
# create two users:
```

```
kubectl apply -f user-groups/
# confirm that the SRE can't delete Pods:
kubectl exec sre-user -- kubectl auth can-i delete pods

# print the username and group in the cert:
kubectl exec sre-user -- sh -c 'openssl x509 -text -noout -in /certs/user.crt | grep Subject:'

# confirm the test user can't read Pod logs:
kubectl exec test-user -- kubectl auth can-i get pod/logs

# and print this certificate's details:
kubectl exec test-user -- sh -c 'openssl x509 -text -noout -in /certs/user.crt | grep Subject:'
```

Now you have two user certificates where the user belongs to a group representing their team - SREs and testers. There are no bindings in the cluster for either of the usernames or the groups, so the users can't do anything yet. You can see in figure 17.11 that Kubernetes uses standard fields in the certificate's subject - the common name is the username and the organization is the group (there can be multiple organizations mapping to multiple groups).

The SRE user should be able to delete Pods but they're a new user and they have no permissions yet.

The O in the cert is the organisation which is the `sre` group; the CN is the common name which is the `sre1` username.

```
PS>kubectl apply -f user-groups/
pod/sre-user created
pod/test-user created
PS>
PS>kubectl exec sre-user -- kubectl auth can-i delete pods
no
command terminated with exit code 1
PS>
PS>kubectl exec sre-user -- sh -c 'openssl x509 -text -noout -in /certs/u
ser.crt | grep Subject:'  
Subject: C = UK, ST = LONDON, L = London, O = sre, CN = sre1
PS>
PS>kubectl exec test-user -- kubectl auth can-i get pod/logs
no
command terminated with exit code 1
PS>
PS>kubectl exec test-user -- sh -c 'openssl x509 -text -noout -in /certs/
user.crt | grep Subject:'  
Subject: C = UK, ST = LONDON, L = London, O = test, CN = tester1
```

The test user doesn't have permission to read logs.

In this cert the group is `test` and the user is `tester1`.

Figure 17.11 Certificates can contain zero or more groups as well as the username

We do want SREs to be able to delete Pods and we do want testers to be able to read Pod logs. We can apply bindings at the group level to give all the users in the team the same permissions, and then we move responsibility for managing group membership to the authentication system. For SREs the simplest thing is for them to have the `view` role throughout the cluster so they can help diagnose issues, and the `edit` role for the namespaces their team manages. Those are built-in cluster roles so we just need bindings where the subject is the `sre` group. For testers we want the very restricted set of permissions in listing 17.6.

Listing 17.6 - test-group.yaml, a role to get Pod details and read logs

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: logs-reader
rules:
- apiGroups: [""]
  resources: ["pods", "pods/log"] # logs are a subresource of Pods
  verbs: ["get"] # which need explicit permissions
```

The full manifest from listing 17.6 includes a cluster role binding giving this role to the `test` group, which effectively gives testers permission to look at any Pods and view their logs - but not to list Pods or do anything to any other resources. Some resources have subresources which are permissioned separately, so if you have permission to get Pods you can't read the logs unless you also have permission to get the logs subresource. When you apply the bindings for the two groups, the users will have the permissions they need.

TRY IT NOW Empower the new users by binding the roles they need to the groups they belong to.

```
# apply the roles and bindings:
kubectl apply -f user-groups/bindings/

# confirm the SRE cannot delete in the default namespace:
kubectl exec sre-user -- kubectl auth can-i delete pods

# but can in the ch17 namespace:
kubectl exec sre-user -- kubectl auth can-i delete pods -n kiamol-ch17

# confirm the tester can't list Pods:
kubectl exec test-user -- kubectl get pods

# but can read the logs for a known Pod:
kubectl exec test-user -- kubectl logs test-user --tail 1
```

You can see in figure 17.12 that this works as expected; the users have the permissions they need but we apply them at a higher level, binding roles to groups. We can change permissions without modifying users or groups. New users can arrive and people can move between teams without any RBAC changes, as long as the authentication system is kept up to date so Kubernetes sees the current group membership in the API call.

These roles and bindings give the groups the permissions they need, and the users in those groups will inherit the permissions.

The SRE user can only delete Pods in the namespace for the apps they work on.

```
PS>kubectl apply -f user-groups/bindings/
clusterrolebinding.rbac.authorization.k8s.io/sre-view-cluster created
rolebinding.rbac.authorization.k8s.io/sre-edit-ch19 created
clusterrole.rbac.authorization.k8s.io/logs-reader created
clusterrolebinding.rbac.authorization.k8s.io/test-logs-cluster created
```

PS>

```
PS>kubectl exec sre-user -- kubectl auth can-i delete pods
no
command terminated with exit code 1
PS>
PS>kubectl exec sre-user -- kubectl auth can-i delete pods -n kiamol-ch19
yes
PS>
```

```
PS>kubectl exec test-user -- kubectl get pods
Error from server (Forbidden): pods is forbidden: User "tester1" cannot list resource "pods" in API group "" in the namespace "default"
command terminated with exit code 1
PS>
PS>kubectl exec test-user -- kubectl logs test-user --tail 1
** Using context for user: tester1; group: test
```

The tester cannot list Pods, but given a Pod name they can read the logs.

Figure 17.12 Permissions at group level give users what they need with an easy management experience

Which is all fine in a corporate environment with Active Directory configured, or a cloud environment with integrated authentication. In smaller-scale clusters you may want to manage authentication within Kubernetes, and generating certificates like we've done in this section isn't really feasible. Once you generate a certificate it stays valid for a year so it's hard to make changes to group membership, and if you want to revoke access you need to rotate the Kubernetes issuer certificate and to distribute new certificates to every user.

A common alternative is to misuse service accounts, creating a service account for every end-user and distributing a token for authentication with Kubectl. This approach doesn't scale to hundreds of users but it is a viable option if you don't have an external authentication system and you want secure access to Kubernetes for a small number of users. You have to take a slightly quirky approach to groups because you can't create a group and add a service account to it. Instead you need to treat namespaces as groups, create a namespace for each group you want and put all the service accounts in that group. Figure 17.13 shows the setup.

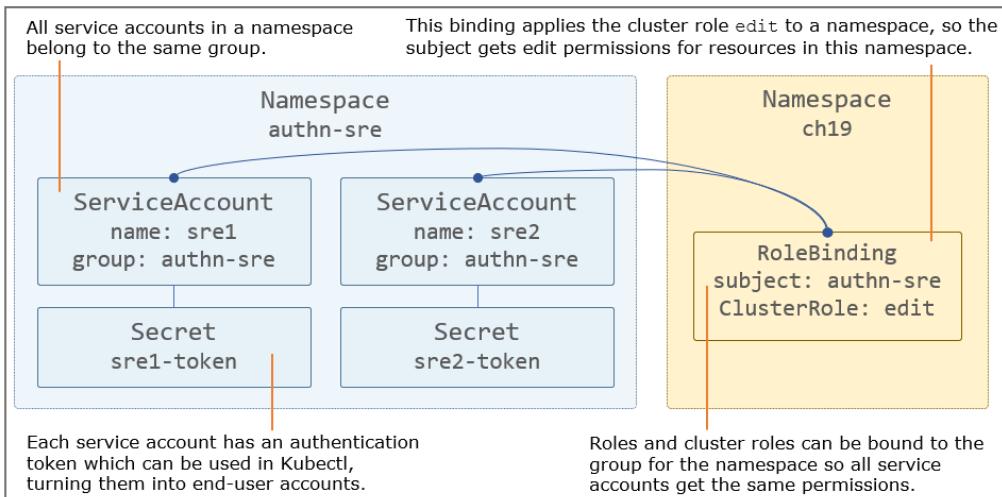


Figure 17.13 Service accounts have tokens and groups so they can be misused as user accounts

Service accounts work as user accounts because you can manage them within the cluster - creating or removing them easily and revoking access by deleting the token. We can recreate our SRE and tester users with service accounts, using separate namespaces to represent the two groups and applying bindings to the groups.

TRY IT NOW Create namespaces for the SRE and tester groups and a service account and token for a user in each namespace.

```
# create namespaces, service accounts and tokens:
kubectl apply -f user-groups/service-accounts/

# apply bindings to the groups:
kubectl apply -f user-groups/service-accounts/role-bindings/

# confirm the SRE group has view access across the cluster:
kubectl get clusterrolebinding sre-sa-view-cluster -o custom-
columns='ROLE:.roleRef.name,SUBJECT KIND:.subjects[0].kind,SUBJECT
NAME:.subjects[0].name'
```

The new cluster role bindings apply to the service account groups - which is the namespace where the service accounts have been created. The difference between these cluster roles and the user certificate cluster roles is just the group name - `system:serviceaccounts:kiamol-authn-sre` for SREs, which is the group for all the service accounts in the `kiamol-authn-sre` namespace, as you see in figure 17.14.

Create service accounts to use as client accounts - with a namespace for the group and a token for each service account.

Apply role bindings for the service account groups.

```
PS>kubectl apply -f user-groups/service-accounts/
namespace/kiamol-authn-sre created
serviceaccount/sre2 created
secret/sre2-sa-token created
namespace/kiamol-authn-test created
serviceaccount/tester2 created
secret/tester2-sa-token created
PS>
PS>kubectl apply -f user-groups/service-accounts/role-bindings/
clusterrolebinding.rbac.authorization.k8s.io/sre-sa-view-cluster created
rolebinding.rbac.authorization.k8s.io/sre-sa-edit-ch19 created
clusterrolebinding.rbac.authorization.k8s.io/test-sa-logs-cluster created
PS>
PS>kubectl get clusterrolebinding sre-sa-view-cluster -o custom-columns='
ROLE:.roleRef.name,SUBJECT KIND:.subjects[0].kind,SUBJECT NAME:.subjects[
0].name'
ROLE    SUBJECT KIND    SUBJECT NAME
view    Group        system:serviceaccounts:kiamol-authn-sre
```

The cluster role binding for SRE accounts - providing view access to all resources via the group for the namespace.

Figure 17.14 Faking an authentication system with service accounts, tokens and namespaces

Service accounts use a JSON Web Token (JWT) to authenticate - this is the token which is pre-populated in Pod volumes. Tokens are created as a Secret of type `kubernetes.io/service-account-token` and Kubernetes ensures every service account has at least one token. You can also create your own which makes it easy to distribute, rotate and revoke the token. The process is simple because Kubernetes actually generates the token; you just need to create an empty secret of the correct type and the cluster adds the data - you did that in the last exercise so these accounts are ready to use.

Kubectl supports lots of different authentication options - certificates we've already seen, and also username and password, third-party authentication and JWT. You can authenticate to the cluster by setting up a new context with credentials which use the token from the service account. Any Kubectl commands using that context will operate as the service account, so the permissions for the service account group will apply.

TRY IT NOW Create a new context in Kubectl for the SRE service account and confirm you can access the cluster. This will work with your cluster, whatever authentication system it uses.

```
# add a base64 command if you're using Windows:
.\base64.ps1

# decode the secret for the token and save to a file:
```

```
kubectl get secret sre2-sa-token -n kiamol-authn-sre -o jsonpath='{.data.token}' | base64 -d > sa-token

# load new credentials into Kubectl from the token file:
kubectl config set-credentials ch17-sre --token=$(cat sa-token)

# create a new context using the SRE credentials:
kubectl config set-context ch17-sre --user=ch17-sre --cluster $(kubectl config view -o jsonpath='{.clusters[0].name}')

# confirm you can delete Pods as the SRE account:
kubectl delete pods -n kiamol-ch17 -l app=todo-web --context ch17-sre
```

This exercise works on all clusters, even if they have third-party authentication configured because it uses Kubernetes' own authentication for service accounts. You can see in figure 17.15 that I can issue commands as the SRE service account and that user has access to delete Pods in the kiamol-ch17 namespace.

The data in the Secret is the JWT to authenticate the service account. Decode the base-64 to get the token.

Kubectl can use the JWT as credentials to authenticate.

```
PS> .\base64.ps1
PS>
PS>PS>kubectl get secret sre2-sa-token -n kiamol-authn-sre -o jsonpath='{.data.token}' | base64 -d > sa-token
PS>PS>kubectl config set-credentials ch19-sre --token=$(cat sa-token)
User "ch19-sre" set.
PS>
PS>PS>kubectl config set-context ch19-sre --user=ch19-sre --cluster $(kubectl config view -o jsonpath='{.clusters[0].name}')
Context "ch19-sre" created.
PS>
PS>PS>kubectl delete pods -n kiamol-ch19 -l app=todo-web --context ch19-sre
pod "todo-web-5579b9588c-2c2r7" deleted
```

Using the SRE context, Kubectl authenticates as the SRE service account, which does have access to delete Pods in the ch19 namespace.

Figure 17.15 Distributing the service account token lets users authenticate as the account

If you're into JWT you can decode the contents of that sa-token file (the online tool at <https://jwt.io> will do it for you), and you'll see that Kubernetes is the issuer and the subject is the service account name system:serviceaccount:kiamol-authn-sre:sre2. RBAC permissions are the combined roles for that service account, the group for the service account namespace and the group for all service accounts. A small note of caution here - be very careful with the subject when you grant roles to service account groups, it's easy to accidentally make every account a cluster admin by leaving the namespace off the subject (which is the exact

problem with some versions of Docker Desktop; see the history at <https://github.com/docker/for-mac/issues/4774>.

Now that we manage authentication within the cluster, it's easy to revoke access to a specific user by deleting the token for their service account. The token they present from `kubectl` no longer authenticates the service account, so they'll get unauthorized errors for every action

TRY IT NOW Stop users authenticating as the SRE service account by deleting the token.

```
# delete the access token:  
kubectl delete secret sre2-sa-token -n kiamol-authn-sre  
  
# wait for the token removal to reach controllers:  
sleep 30  
  
# now try to get Pods as the SRE account:  
kubectl get pods --context ch17-sre
```

Revoking access is easy, as you see in figure 17.16. Rotating tokens needs a bit more of a process; you can create a new token and send it to the user (securely!) and then delete the old token once the user has updated their context. Changing groups is more involved again because you need to create a new service account in the namespace for the group, create and send the new token and delete the old service account.

Deleting the token in the cluster means the service account can't be authenticated with that token.

```
PS>kubectl delete secret sre2-sa-token -n kiamol-authn-sre  
secret "sre2-sa-token" deleted  
PS>  
PS>sleep 30  
PS>  
PS>kubectl get pods --context ch19-sre  
error: You must be logged in to the server (Unauthorized)
```

Kubectl tries to use the old token and gets an unauthorized response.

Figure 17.16 You can remove access without removing service accounts by deleting tokens

Service accounts for users is a simple approach to securing your cluster, if you can live with the limitations. It might be a reasonable way to get started but you should understand your roadmap, which will most likely be to use OpenID Connect (OIDC) for authentication and bind your RBAC rules to OIDC username and group claims.

When you have your authentication system wired up and your RBAC rules set, you still have a challenge. Kubernetes doesn't have a great set of tools for auditing permissions, so in the next section we'll look at third-party options for verifying permissions.

17.4 Discovering and auditing permissions with plugins

The Kubectl `can-i` command is useful for checking if a user can perform a function, but that's the only tool you get for validating permissions and it doesn't really go far enough. You often want to approach it the other way around and ask who can perform a function, or print out a matrix of access permissions or search for an RBAC subject and see their roles. Kubectl has a plugin system to support additional commands, and there are plugins to satisfy all these different ways to approach RBAC.

The best way to add Kubectl plugins is with Krew, the plugin manager. You can install Krew directly on your machine but the installation isn't super-smooth and you might not want plugins installed on that machine (if you do, the install documentation is here <https://krew.sigs.k8s.io/docs/user-guide/setup/install>). I've prepped a container image with Krew ready installed which you can use that image to try out plugins in a safe way. The first one we'll look at is `who-can`, which is like the reverse of `can-i`.

TRY IT NOW Start the Krew Pod and connect to install and use the `who-can` plugin.

```
# run a Pod with Kubectl and Krew already installed:
kubectl apply -f rbac-tools/

# wait for the Pod:
kubectl wait --for=condition=ContainersReady pod -l app=rbac-tools

# connect to the container:
kubectl exec -it deploy/rbac-tools -- sh

# install the who-can plugin:
kubectl krew install who-can

# list who has access to the todo-list ConfigMap:
kubectl who-can get configmap todo-web-config
```

Any binary can be a Kubectl plugin, but Krew simplifies the setup and adds some process by curating useful plugins (Krew is a Kubernetes Special Interest Group project). Plugins operate in the context of your authenticated user so you need to be sure they don't do anything they shouldn't. The `who-can` plugin from that exercise is published by Aqua Security and as you see in figure 17.17 it works by traversing RBAC roles to find matching permissions, and then prints the subjects who have bindings for those roles.

Runs a Pod with Krew installed so you can try the plugins without altering your system
 Krew has a plugin catalog you can use to search for and install plugins - I've snipped the installation output.

```

PS>kubectl apply -f rbac-tools/
serviceaccount/rbac-tools created
deployment.apps/rbac-tools created
clusterrolebinding.rbac.authorization.k8s.io/rbac-tools-admin created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=rbac-tools
pod/rbac-tools-5cc47c5545-xwnhg condition met
PS>
PS>kubectl exec -it deploy/rbac-tools -- sh
/ #
/ # kubectl krew install who-can
Updated the local copy of plugin index.
Installing plugin: who-can
Installed plugin: who-can
  
```

/ # kubectl who-can get configmap todo-web-config
 No subjects found with permissions to get configmap assigned through Role Bindings

CLUSTERROLEBINDING	TYPE	SA-NAMESPACE	SUBJECT
cluster-admin	Group		system:masters
docker-for-desktop-binding	Group		system:serviceaccounts:kube-
system	Group		rbac-tools
rbac-tools-admin	ServiceAccount default		system:serviceaccounts:kiamo
sre-sa-view-cluster			
l-authn-sre	Group		

Plugins appear as Kubectl commands. who-can plugin starts from a resource and prints the subjects who have access, and the binding which gives them access. There are some familiar names here.

Figure 17.17 Plugins add new features to Kubectl - who-can is a useful tool for RBAC queries

We'll look at two more plugins which fill gaps in RBAC auditing. The next is `access-matrix` which operates on a resource type or a specific resource and prints all the access rules for that object, showing who can do what. This is the most useful tool for ongoing auditing, where you could run a scheduled job that collects a set of access matrices and confirms no unexpected permissions have been set.

TRY IT NOW Install the `access-matrix` plugin and print the matrices for different resources.

```
# install the plugin:
kubectl krew install access-matrix
```

```
# print the matrix for Pods:  
kubectl access-matrix for pods -n default  
  
# and for the to-do list ConfigMap:  
kubectl access-matrix for configmap todo-web-config -n default
```

The default output from `access-matrix` prints nice icons, as you see in figure 17.18 but you can configure plain ASCII output in the command line. My output is heavily snipped because the full permissions list includes all the controllers and other system components. The access matrix for a ConfigMap is more manageable, and you can see the SRE groups created in section 17.3 have list permission so they can see this object, but the `rbac-tools` service account has full update and delete permissions. That's because I was lazy for this exercise and gave the account the `cluster-admin` role instead of building a custom role with just the permissions the plugins need.

The `access-matrix` plugin lists who can perform actions on a resource.

I've snipped the output for Pods but you can see lots of system components. You know how Deployments work so it makes sense that Deployments can list Pods and ReplicaSets can create and delete them.

/ # kubectl krew install access-matrix	Updated the local copy of plugin index.					
Installing plugin: access-matrix	Installed plugin: access-matrix					
/ # kubectl access-matrix for pods -n default						
NAME	KIND	SA-NAMESPACE	LIST	CREATE	UPDATE	DELETE
deployment-controller	ServiceAccount	kube-system	✓	✗	✓	✗
replicaset-controller	ServiceAccount	kube-system	✓	✓	✗	✓
sre	Group		✓	✗	✗	✗
statefulset-controller	ServiceAccount	kube-system	✓	✓	✓	✓
system:kube-controller-manager	User		✓	✗	✗	✗
system:kube-scheduler	User		✓	✗	✗	✓
system:masters	Group		✓	✓	✓	✓
system:serviceaccounts:kiamol-authn-sre	Group		✓	✗	✗	✗
system:serviceaccounts:kube-system	Group		✓	✓	✓	✓
/ # kubectl access-matrix for configmap todo-web-config -n default						
NAME	KIND	SA-NAMESPACE	LIST	CREATE	UPDATE	DELETE
generic-garbage-collector	ServiceAccount	kube-system	✓	✗	✓	✓
horizontal-pod-autoscaler	ServiceAccount	kube-system	✓	✗	✗	✗
namespace-controller	ServiceAccount	kube-system	✓	✗	✗	✓
rbac-tools	ServiceAccount	default	✓	✓	✓	✓
resourcequota-controller	ServiceAccount	kube-system	✓	✗	✗	✗
sre	Group		✓	✗	✗	✗
system:kube-controller-manager	User		✓	✗	✗	✗
system:masters	Group		✓	✓	✓	✓
system:serviceaccounts:kiamol-authn-sre	Group		✓	✗	✗	✗

A ConfigMap also has lots of system components in the list, together with the SRE groups I created and the service account for this `rbac-tools` Pod. That account doesn't need all these permissions.

Figure 17.18 The `access-matrix` plugin shows who can do what with different resources

And the last plugin we'll look at is `rbac-lookup`, which is useful for searching RBAC subjects. It finds matches in users, service accounts and groups and shows you the roles that are bound to

the subject. It's a good one for checking RBAC from the user perspective, when you want to confirm that a subject has the correct roles assigned.

TRY IT NOW Install the rbac-lookup plugin and search for SRE and test subjects.

```
# install the plugin:  
kubectl krew install rbac-lookup  
  
# search for SRE:  
kubectl rbac-lookup sre  
  
# search for test:  
kubectl rbac-lookup test
```

One thing no plugin can do is give you the consolidated permissions for a user and all the groups they belong to, because Kubernetes has no idea about group membership. RBAC works by getting the consolidated permissions when a user is presented with the list of their groups, but outside of an incoming request there is no link between users and groups. That's different with service accounts as they always belong to known groups - you can see in figure 17.19 the SRE groups. There's no way to find which users belong to the `sre` user group, but you can list all the service accounts in the `kiamol-authn-sre` namespace to see who belongs to the service account group.

The rbac-lookup plugin searches for subjects and prints their roles.

There are two SRE group subjects - one for users and one for service accounts. Both have the same cluster and namespace roles.

```
/ # kubectl krew install rbac-lookup  
Updated the local copy of plugin index.  
Installing plugin: rbac-lookup  
Installed plugin: rbac-lookup
```

SUBJECT	SCOPE	ROLE
sre	cluster-wide	ClusterRole/view
sre	kiamol-ch19	ClusterRole/edit
system:serviceaccounts:kiamol-authn-sre	kiamol-ch19	ClusterRole/edit
system:serviceaccounts:kiamol-authn-sre	cluster-wide	ClusterRole/view

```
/ #  
/ # kubectl rbac-lookup test  
SUBJECT
```

SUBJECT	SCOPE	ROLE
system:serviceaccounts:kiamol-authn-test	cluster-wide	ClusterRole/logs-reader
test	cluster-wide	ClusterRole/logs-reader

There are also two test group subjects, both have the custom logs-reader role applied for the whole cluster.

Figure 17.19 Searching for RBAC subjects and printing the roles they're bound to

It's worth exploring the Krew catalog to find more plugins which will help your daily workflow. This section focused on the established plugins which help with RBAC; they're popular and have

been around for a long time, but there are plenty more gems to find. That's it for the practical work in this chapter, and we'll finish up looking at how RBAC affects your workloads and guidance for implementing access controls.

17.5 Planning your RBAC strategy

RBAC gives you a consistent way to secure Kubernetes for external users and apps running in the cluster. The mechanics of roles and bindings are the same for both types of subject, but they need very different approaches. Users are authenticated by a separate, trusted system which asserts the username and groups the user belongs to. Guidance here is to start with the predefined cluster roles - view, edit and admin and apply them to groups. The cluster admin role should be strictly guarded and only used if you really need it - preferably restricting it to service accounts for automated processes.

Think about using namespaces as a security boundary so you can limit scope further. If your authentication system has group information that identifies someone's team and their function on that team, that might be enough to map bindings to a role and a product namespace. Be wary of cluster-wide roles as they can be used to gain elevated access - if a user can read secrets in any namespace, they can potentially grab the JWT token for a privileged service account and get admin access to that namespace.

Service accounts should be used sparingly and only by apps which need access to the Kubernetes API server. You can disable auto-mounting the token in Pods as a property of the default service account, so Pods never see the token unless they actively request it. Remember that RBAC is not about restricting access at deployment time. Pods don't need a service account configured to use a ConfigMap which is mounted as a volume. You only need service accounts for apps which use the Kubernetes API, and in those cases each app should have a dedicated service account with just enough permissions for the app to do what it needs, preferably with named resources.

Applying RBAC isn't the end of the story. As your security profile matures you'll want to add auditing to ensure policies don't get subverted, and admission controllers to ensure new apps have the required security controls. But we won't cover that here; just be aware that securing your cluster is about using multiple approaches to gain security in depth.

Okay, time to clear down the cluster before you move onto the lab.

TRY IT NOW Remove all of the chapter's resources.

```
kubectl delete all,ns,rolebinding,clusterrolebinding,role,clusterrole,serviceaccount
-1 kiamol=ch17
```

17.6 Lab

The Kube Explorer app you ran in section 17.2 can also display Service Accounts, but it needs some more permissions to do that. There's a new set of manifests for the app in the lab folder

which give it permissions to access Pods in the default namespace. Your task for this lab is to add roles and bindings so the app can also:

- display and delete Pods in the kiamol-ch17-lab namespace
- display service accounts in the default namespace
- display service accounts in the kiamol-ch17-lab namespace

Look closely at how the existing RBAC rules are set up, and that should make this one straightforward; just remember how namespaces in the bindings affect scope. My solution is available for you to check:

<https://github.com/sixeyed/kiamol/blob/master/ch17/lab/README.md>

18

Deploying Kubernetes: multi-node and multi-architecture clusters

You can do an awful lot with Kubernetes without understanding the architecture of the cluster and how all the pieces fit together - you already have in the previous seventeen chapters. But that additional knowledge will help you understand what high availability looks like in Kubernetes, and what you need to think about if you want to run your own cluster. The best way to learn about all the Kubernetes components is to install a cluster from scratch, and that's what you'll do in this chapter. The exercises will start with plain virtual machines and walk you through a basic Kubernetes setup for a multi-node cluster, which you can use to run some of the sample apps you're familiar with from the book.

Every app we've run so far has used Linux containers built for Intel 64-bit processors, but Kubernetes is a multi-architecture platform. A single cluster can have nodes with different operating systems and different types of CPU, so you can run a variety of workloads. In this chapter you'll also add a Windows Server node to your cluster and run some Windows applications. That part is optional, but if you're not a Windows user it's worth following through those exercises to see how Kubernetes uses exactly the same modelling language for different architectures, with just a few tweaks to the manifests.

18.1 What's inside a Kubernetes cluster

You'll need some different tools to follow along in this chapter and some virtual machine images which will take a while to download. You can start the installation now and by the time we've finished looking at the architecture of Kubernetes, you should be ready to go.

TRY IT NOW You'll use Vagrant to run the virtual machines, which is a free open-source tool for VM management. You also need to have a Virtual Machine runtime - you can use VirtualBox, Hyper-V (on Windows)

or Parallels (on macOS) for these machines. Install Vagrant and then download the base VM images you'll use for your cluster.

```
# install Vagrant -  
# browse to https://www.vagrantup.com to download  
# OR on Windows you can use: choco install vagrant  
# OR on macOS you can use: brew cask install vagrant  
  
# Vagrant packages VM images into "boxes"  
# download a Linux box - this will ask you to choose a provider  
# and you should select your VM runtime:  
vagrant box add bento/ubuntu-20.04  
  
# and if you want to add Windows, download a Windows box:  
vagrant box add kiamol/windows-2019
```

Okay, while that's happening it's time to learn about the architecture of Kubernetes. You know that a cluster is composed of one or more servers called nodes, but those nodes can play different roles. There's the Kubernetes *control plane* which is the management side of the cluster (previously called master nodes), and then *nodes* which run your workloads (these were once called minions). At a high level the control plane is the thing that receives your `kubectl` deployment requests, and it actions them by scheduling Pods on the nodes. Figure 18.1 shows the cluster at the user-experience level.

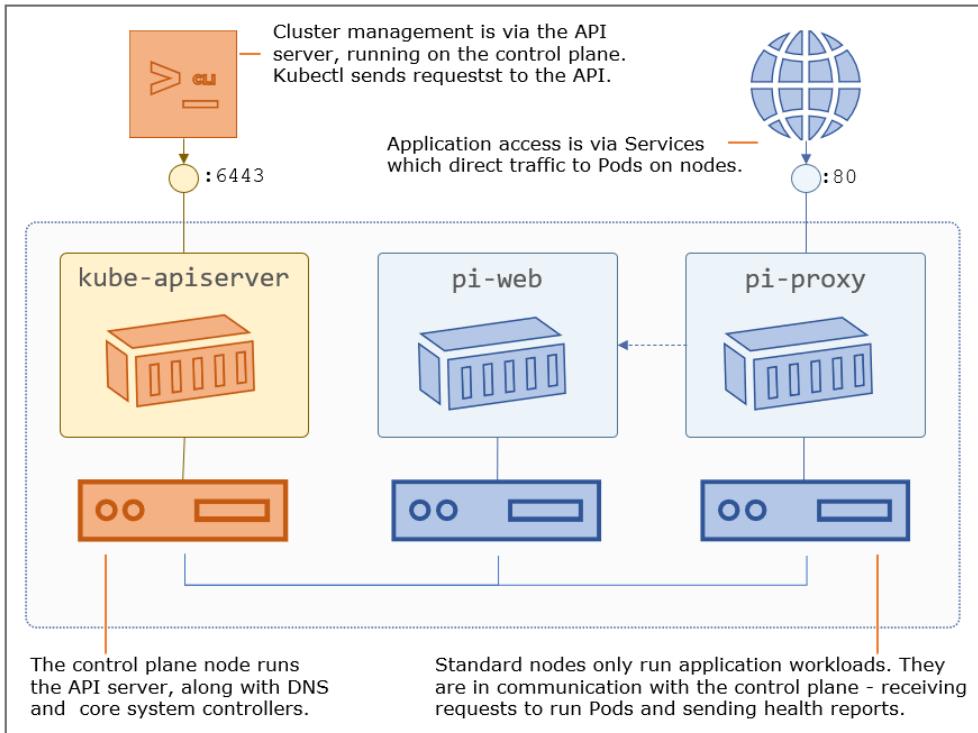


Figure 18.1 From the user perspective this is one cluster with management and application endpoints

In a managed Kubernetes platform in the cloud, the control plane is taken care of for you, so you only need to worry about your own nodes (in AKS the control plane is completely abstracted so you only see - and pay for - worker nodes). That's one reason why a managed platform is so attractive, because there are multiple components in the control plane, and you need to manage those if you run your own environment. These components are all critical for the cluster to function:

- the *API server* is the management interface - it's a REST API on an HTTPS endpoint which you connect to with Kubectl, and which Pods can use internally. It runs in the kube-apiserver Pod and it can be scaled up for high availability;
- the *scheduler* watches for new Pod requests and selects a node to run them. It runs in the kube-scheduler Pod, but it's a pluggable component - you can deploy your own custom scheduler;
- the *controller manager* runs core controllers - which are internal components, not visible controllers like Deployments. The kube-controller-manager Pod runs controllers which

- observe node availability and manage Service endpoints;
- etcd* is the Kubernetes data store where all cluster data gets stored. It's a distributed key-value database which replicates data across many instances.

You need to run multiple control plane nodes for high availability - an odd number to support failures, so if the manager node goes down the remaining nodes can vote for a new manager. Each control plane node runs an instance of all of these components, shown in figure 18.2. The API server is load-balanced, and the back-end data is replicated, so any control plane node can work on a request and it will be processed in the same way.

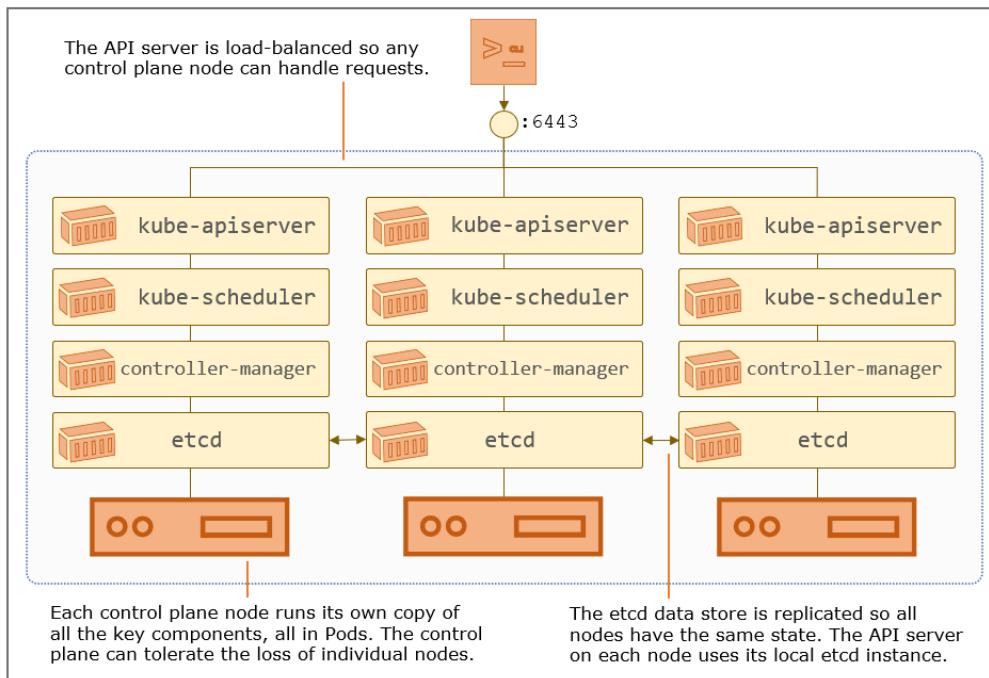


Figure 18.2 Production clusters need multiple control plane nodes for high availability

Your downloads should be nearly done now, so we'll just look at one more level of detail - the components running on each node. Nodes are responsible for creating Pods and ensuring their containers keep running, and for connecting Pods to the Kubernetes network:

- the *Kubelet* is a background agent that runs on the server - not in a Pod or in a container. It receives requests to create Pods and manages the Pod lifecycle, and also sends heartbeats to the API server confirming the node is healthy;
- the *kube-proxy* is the network component which routes traffic between Pods or from Pods to the outside world. It runs as a DaemonSet, with a Pod on each node managing that

- node's traffic;
- the *container runtime* - used by the Kubelet to manage Pod containers. Typically Docker, containerd or CRI-O, this is pluggable with any runtime that supports CRI (the Container Runtime Interface).

Figure 18.3 shows the internal components on each node - and these all run on the control plane nodes too.

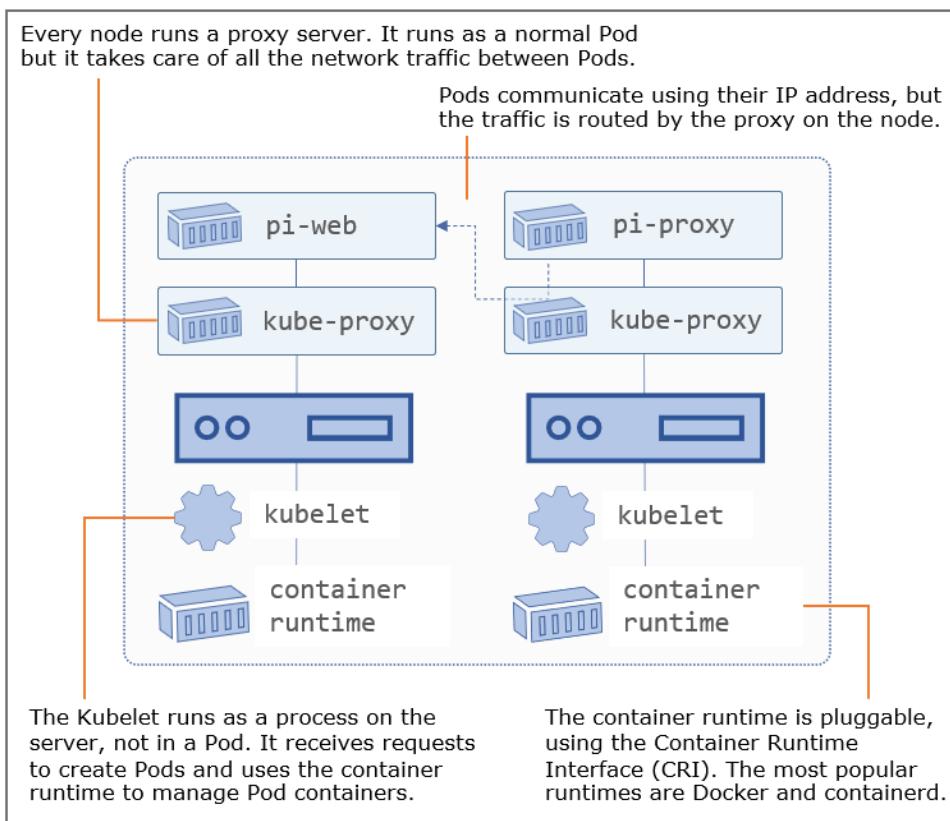


Figure 18.3 The next level of detail for the nodes - the Kubelet, proxy and container runtime

You can see that Kubernetes is a complex platform with lots of moving pieces. These are just the core components - there's also Pod networking, DNS and in the cloud a separate cloud controller manager which integrates with cloud services. You can deploy and manage your own Kubernetes cluster with 100% open-source components, but you need to be aware of the complexity you're taking on. And that's more than enough theory — let's go build a cluster.

18.2 Initializing the control plane

Most of the work in deploying Kubernetes is taken care of by a tool called Kubeadm - it's an administration command line which can initialize a new control plane, join nodes to the cluster and upgrade the Kubernetes version. Before you get to Kubeadm you need to install a bunch of dependencies. In a production environment you'd have these already installed in your VM image, but to show you how it works we'll start from scratch.

TRY IT NOW Run a Linux VM which will be the control plane node and install all the Kubernetes dependencies.

If you're using Hyper-V on Windows you'll need to run your shell as Administrator.

```
# switch to this chapter's source:
cd ch18

# use Vagrant to start a new VM - depending on your VM runtime
# you'll get prompts asking you to choose a network and to
# provide your credentials to mount folders from your machine:
vagrant up kiamol-control

# connect to the VM:
vagrant ssh kiamol-control

# this folder maps the ch18 source folder:
cd /vagrant/setup

# make the install script executable and run it:
sudo chmod +x linux-setup.sh && sudo ./linux-setup.sh

# confirm Docker has been installed:
which docker

# and all the Kubernetes tools:
ls /usr/bin/kube*
```

Figure 18.4 just gives you the highlights - creating the VM, running the setup script and verifying all the tools are there. If you inspect the `linux-setup.sh` script in this chapter's source you'll see that it installs Docker and the Kubernetes tools, and sets a couple of memory and network configurations for the server.

Vagrant is a tool for managing VMs. The chapter source has a Vagrantfile which defines the VMs for the cluster, and this starts the VM which will be the control plane node.

Vagrant shares your local folder with the VM, so you may be asked for credentials.

```
PS>cd ch17
PS>
PS>vagrant up kiamol-control
Bringing machine 'kiamol-control' up with 'hyperv' provider...
==> kiamol-control: Verifying Hyper-V is enabled...
==> kiamol-control: Verifying Hyper-V is accessible...
==> kiamol-control: Importing a Hyper-V instance

kiamol-control: Username (user[@domain]): elton
kiamol-control: Password (will be hidden):
```

Vagrant requires administrator access to create SMB shares and may request access to complete setup of configured shares.

```
==> kiamol-control: Setting hostname...
==> kiamol-control: Mounting SMB shared folders...
    kiamol-control: D:/scm/github/sixeyed/kiamol/ch17 => /vagrant
```

```
vagrant@kiamol-control:~$ cd /vagrant/setup
vagrant@kiamol-control:/vagrant/setup$ sudo chmod +x linux-setup.sh && sudo
./linux-setup.sh
Get:1 http://security.ubuntu.com/ubuntu focal-security InRelease [107 kB]

vagrant@kiamol-control:/vagrant/setup$ which docker
/usr/bin/docker
vagrant@kiamol-control:/vagrant/setup$ ls /usr/bin/kube*
/usr/bin/kubeadm  /usr/bin/kubectl  /usr/bin/kubelet
vagrant@kiamol-control:/vagrant/setup$
```

The setup script installs all the Kubernetes dependencies - Docker, the Kubelet, Kubectl and Kubeadm.

This confirms the pre-requisites are all installed.

Figure 18.4 Kubeadm is the tool which sets up the cluster, but it needs the container runtime and Kubelet

Now this machine is ready to become a Kubernetes control plane node. After all the build-up to get here the next exercise will be an anti-climax: you only need to run a single command to initialize the cluster and get all the control plane components running. Keep an eye on the output and you'll see all the pieces you learned about in section 18.1 starting up.

TRY IT NOW Use Kubeadm to initialize a new cluster with a fixed set of network addresses for Pods and Services.

```
# initialize a new cluster:
sudo kubeadm init --pod-network-cidr="10.244.0.0/16" --service-cidr="10.96.0.0/12" --
apiserver-advertise-address=$(cat /tmp/ip.txt)
```

```
# create a folder for the Kubectl config file:  
mkdir ~/.kube  
  
# copy the admin configuration to your folder:  
sudo cp /etc/kubernetes/admin.conf ~/.kube/config  
  
# make the file readable so Kubectl can access it:  
sudo chmod +r ~/.kube/config  
  
# confirm you have a Kubernetes cluster:  
kubectl get nodes
```

The output from initializing the cluster tells you what to do next - including the command you'll run on other nodes to join the cluster (you'll need that for later exercises, so be sure to copy it into a text file somewhere). The command also generates a config file which you can use to manage the cluster with Kubectl - you can see in figure 18.5 that the cluster exists with a single control-plane node, but the node isn't in the Ready status.

Kubeadm runs a series of checks to confirm your machine is suitable - the install script from the previous exercise set everything up so the checks all pass.

```
vagrant@kiamol-control:/vagrant/setup$ sudo kubeadm init --pod-network-cidr="10.244.0.0/16" --service-cidr="10.96.0.0/12"
W0713 10:59:57.286462    19160 configset.go:202] WARNING: kubeadm cannot validate component configs for API groups [kubelet.config.k8s.io kubeProxy.config.k8s.io]
[init] Using Kubernetes version: v1.18.5
[preflight] Running pre-flight checks
```

Then you can join any number of worker nodes by running the following on each as root:

```
kubeadm join 172.21.120.227:6443 --token zc9m76.8l4573wyjf9vokg3 \
--discovery-token-ca-cert-hash sha256:2c520ea15a99bd68b74d04f40056996dff
5b6ed1e76dfaeb0211c6db18ba0393
vagrant@kiamol-control:/vagrant/setup$ mkdir ~/.kube
vagrant@kiamol-control:/vagrant/setup$ sudo cp /etc/kubernetes/admin.conf ~/.kube/config
vagrant@kiamol-control:/vagrant/setup$ sudo chmod +r ~/.kube/config
vagrant@kiamol-control:/vagrant/setup$ kubectl get nodes
NAME           STATUS      ROLES   AGE     VERSION
kiamol-control  NotReady   master   4m10s   v1.18.5
```

The output of the init command is a join command - your own command will have a different IP address, token and CA hash. You'll need to copy it for later.

The cluster is created! It has a control plane node, but it's not ready to use yet.

Figure 18.5 Initializing the cluster is simple - Kubeadm starts all the control plane components

The cluster isn't ready because it doesn't have a Pod network installed. You know from chapter 16 that Kubernetes has a network plugin model and different plugins have different capabilities. We used Calico in that chapter to demonstrate network policy enforcement, and in this chapter we'll use Flannel (another open-source option), because it has the most mature support for clusters with mixed architectures. You install Flannel in the same way as Calico - with a Kubernetes manifest you apply on the control plane node.

TRY IT NOW Add a network plugin to your new cluster, using a ready-made Flannel manifest.

```
# deploy Flannel:
kubectl apply -f flannel.yaml

# wait for the DNS Pods to start:
kubectl -n kube-system wait --for=condition=ContainersReady pod -l k8s-app=kube-dns
```

```
# print the node status:  
kubectl get nodes  
  
# leave the control plane VM:  
exit
```

Kubeadm deploys the DNS server for the cluster as Pods in the `kube-system` namespace, but those Pods can't start until the network plugin is running. As soon as Flannel is deployed, the DNS Pods start and the node is ready. My output in figure 18.6 shows the multi-architecture support in Flannel - if you want to add an IBM mainframe to your Kubernetes cluster, you can do that.

Flannel is a network plugin with great cross-platform support. It runs a Pod on each node using DaemonSets for different architectures - supporting everything from the Raspberry Pi to IBM mainframes.

```
vagrant@kiamol-control:/vagrant/setup$ kubectl apply -f flannel.yaml  
podsecuritypolicy.policy/psp.flannel.unprivileged created  
clusterrole.rbac.authorization.k8s.io/flannel created  
clusterrolebinding.rbac.authorization.k8s.io/flannel created  
serviceaccount/flannel created  
configmap/kube-flannel-cfg created  
daemonset.apps/kube-flannel-ds-amd64 created  
daemonset.apps/kube-flannel-ds-arm64 created  
daemonset.apps/kube-flannel-ds-arm created  
daemonset.apps/kube-flannel-ds-ppc64le created  
daemonset.apps/kube-flannel-ds-s390x created  
vagrant@kiamol-control:/vagrant/setup$  
vagrant@kiamol-control:/vagrant/setup$ kubectl -n kube-system wait --for=condition=ContainersReady pod -l k8s-app=kube-dns  
pod/coredns-66bff467f8-kcbw5 condition met  
pod/coredns-66bff467f8-19f85 condition met  
vagrant@kiamol-control:/vagrant/setup$  
vagrant@kiamol-control:/vagrant/setup$ kubectl get nodes  
NAME           STATUS  ROLES   AGE    VERSION  
kiamol-control  Ready   master  16m    v1.18.5
```

DNS relies on the Pod network, so when DNS is running the network must be operational.

And now the control plane node is ready.

Figure 18.6 Kubeadm doesn't deploy a Pod network; Flannel is a good multi-architecture option

And that's all you need for the control plane. I've glossed over the network setup - the IP address ranges in the Kubeadm command use the configuration Flannel expects - but you'll need to plan that out so it works with your own network. And I skipped over the other 25 options for Kubeadm, but you'll want to research them if you're serious about managing your own cluster. Right now you have a simple single node cluster. You can't use it for application workloads yet

because the default setup restricts the control plane nodes, so they only run system workloads. Next we'll add another node and run some apps.

18.3 Adding nodes and running Linux workloads

The output of the Kubeadm initialization gives you the command you need to run on other servers to join them to the cluster. New Linux nodes need the same setup as the control plane node, with a container runtime and all the Kubernetes tools. In the next exercise you'll create another VM and install the pre-requisites using the same setup script you used for the control plane.

TRY IT NOW Create a second VM with Vagrant and install the pre-requisites for it to join the Kubernetes cluster.

```
# start the node VM:  
vagrant up kiamol-node  
  
# connect to the VM:  
vagrant ssh kiamol-node  
  
# run the setup script:  
sudo /vagrant/setup/linux-setup.sh
```

Figure 18.7 is pretty much the same as figure 18.5 except for the machine name, so you can see that the setup script should really be part of the VM provisioning. That way when you spin up a new machine with Vagrant (or Terraform or whatever tool works for your infrastructure), it will have all the pre-requisites and will be ready to join the cluster.

This starts a second VM which will join the cluster as a node.

Nodes need the same pre-requisites as the control plane, so we run the same setup script.

```

PS>vagrant up kiamol-node
Bringing machine 'kiamol-node' up with 'hyperv' provider...
==> Kiamol-node: Verifying Hyper-V is enabled...
==> Kiamol-node: Verifying Hyper-V is accessible...
==> Kiamol-node: Importing a Hyper-V instance
vagrant@kiamol-node:~$ sudo /vagrant/setup/linux-setup.sh
Hit:1 http://archive.ubuntu.com/ubuntu focal InRelease
Get:2 http://archive.ubuntu.com/ubuntu focal-updates InRelease [107 kB]
Setting up kubelet (1.18.5-00) ...
Created symlink /etc/systemd/system/multi-user.target.wants/kubelet.service
→ /lib/systemd/system/kubelet.service.
Setting up kubeadm (1.18.5-00) ...
Processing triggers for man-db (2.9.1-1) ...
net.bridge.bridge-nf-call-iptables = 1
vagrant@kiamol-node:~$
```

Kubeadm, the Kubelet and Docker are all installed so this VM is ready. The setup should be automated in the VM provisioning stage if you're doing this for real.

Figure 18.7 Installing required dependencies - nodes need the same initial setup as the control plane

Now you can double the size of your Kubernetes cluster by joining a new node. The output of the Kubeadm init command contains everything you need - a CA certificate hash so the new node can trust the control plane, and a join token so the control plane allows the new server to join. The join token is sensitive and you need to distribute it securely to stop rogue nodes joining your cluster - any machine with network access to the control plane and the token could potentially join. Your new VM is in the same virtual network as the control plane VM, so all you need to do is run the join command.

TRY IT NOW Join the cluster using Kubeadm and the join command from the control plane.

```

# you'll need to use your own join command;
# the control plane IP address, token and CA hash
# will be different, this just shows you how it looks:

sudo kubeadm join 172.21.125.229:6443
--token 3sqpc7.a19sx21toelnar5i
--discovery-token-ca-cert-hash
    sha256:ed01ef0e33f7ecd56f1d39b5db0fbaa56811ac055f43adb37688a2a2d9cc86b9

# if your token has expired, run this on the control plane node:
kubeadm token create --print-join-command
```

In that exercise you'll see logs from the Kubelet about TLS bootstrapping. The control plane generates TLS certificates for new nodes so the Kubelet can authenticate with the API server. You can customize the Kubeadm install to provide your own Certificate Authority, but that's another layer of detail (including certificate renewal and external CAs) that we won't cover here. Figure 18.8 shows my new node successfully joining the cluster with the simple default setup.

The join token and CA hash are security controls around nodes joining the cluster - the token has a short lifespan but you need to treat it as sensitive information.

```
vagrant@kiamol-node:~$ sudo kubeadm join 172.21.120.227:6443 --token 5wbq7j.
bew48gsfy0maa2bo --discovery-token-ca-cert-hash sha256:2c520ea15a99bd68b74d0
4f40056996dff5b6ed1e76dfaeb0211c6db18ba0393
W0713 11:47:53.750388 18304 join.go:346] [preflight] WARNING: JoinControlP
ane.controlPlane settings will be ignored when control-plane flag is not set
.
[preflight] Running pre-flight checks
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was recei
ved.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the clust
er.

Communication between nodes is secured - the control plane generates a
client certificate for the Kubelet to authenticate with the API server.
```

Figure 18.8 Joining a node to the cluster sets up secure communication to the control plane

The new node runs its own subset of the components already running on the control plane. The Kubelet runs as a background process outside of Kubernetes, communicating with Docker - which also runs as a background process. It runs two more components in Pods - the network proxy and the network plugin. All the other components - DNS, the controller manager and the API server - are specific to control plane nodes and won't run on standard nodes. Switch back to the control plane node and you can see the Pods.

TRY IT NOW `kubectl` is only set up on the control plane, although you can share the config file to connect from another machine. Switch back to that control plane node to see all the cluster resources.

```
# connect to the control plane node:
vagrant ssh kiamol-control

# print the node status:
kubectl get nodes
```

```
# list all the Pods on the new node:
kubectl get pods --all-namespaces --field-selector spec.nodeName=kiamol-node
```

My output is in figure 18.9 - you should see the same, unless the Flannel Pod is still starting up in which case the new node won't be ready yet.

You can administer the cluster from any machine by copying the Kubectl connection details. The node list shows both nodes are ready.

```
vagrant@kiamol-control:~$ kubectl get nodes
NAME           STATUS    ROLES      AGE     VERSION
kiamol-control  Ready     master    111m   v1.18.5
kiamol-node    Ready     <none>   64m    v1.18.5

vagrant@kiamol-control:~$ kubectl get pods --all-namespaces --field-selector
spec.nodeName=kiamol-node
NAMESPACE      NAME                               READY   STATUS    RESTARTS   AGE
kube-system   kube-flannel-ds-amd64-q97bd   1/1     Running   0          64m
kube-system   kube-proxy-w28fz      1/1     Running   0          64m
```

The new node is running two system Pods, for the network proxy and the network itself. These were created by the DaemonSets when the new node joined.

Figure 18.9 DaemonSets run a Pod on every node - the new Pod runs system components when it joins

You're good to deploy an app now, but you need to be aware of the limitations in this cluster. There is no default storage class set up and no volume provisioners, so you won't be able to deploy dynamic PersistentVolumeClaims (which we covered in chapter 6), and you're stuck using HostPath volumes. There's also no load balancer integration so you can't use LoadBalancer services. In the datacenter you could use Network File Shares (NFS) for distributed storage, and a project called MetalLB for load balancer support. That's all too much for this chapter, so we'll stick to simple apps with no storage requirements and use NodePort Services to get traffic into the cluster.

NodePorts are a much simpler type of Service - they work in the same way as other Services to distribute traffic to Pods, but they listen for incoming traffic on a specific port on the node. Every node listens on the same port, so any server can receive a request and route it to the correct Pod, even if that Pod is running on a different node. You can use NodePorts in on-premises clusters if you have an existing load balancer, but NodePorts are restricted to certain port ranges, so your load balancer will need to do some port mapping. Listing 18.1 shows a NodePort Service spec for the Astronomy Picture of the Day web application.

Listing 18.1 - web.yaml, a Service exposed as a NodePort

```
apiVersion: v1
kind: Service
```

```

metadata:
  name: apod-web
spec:
  type: NodePort      # every node will listen on the port
  ports:
    - port: 8016        # this is the internal ClusterIP port
      targetPort: web  # this is the container port to send to
      nodePort: 30000   # this is the port the node listens on -
    selector:           # it must be >= 30000, a security restriction
      app: apod-web

```

There are three components in the APOD application and the Service type is the only difference between this spec and the others we've deployed. When you run the app you might expect Kubernetes to distribute Pods all around the cluster, but remember that control plane nodes are isolated from user workloads by default.

TRY IT NOW Deploy the app to your new cluster and see where the Pods are scheduled to run.

```

# deploy the manifests in the usual way:
kubectl apply -f /vagrant/apod/

# print the Pod status:
kubectl get pods -o wide

```

You can see in figure 18.10 that every Pod is scheduled to run on the same node. That's a new VM with none of the container images for the book available, so it will download them from Docker Hub - the Pods will be in the ContainerCreating status while that happens. The largest image for this app is just over 200MB so it shouldn't take too long to start up.

This cluster is quite different from your usual lab cluster, but you use the same tools and the same specs to deploy applications.

```
vagrant@kiamol-control:~$ kubectl apply -f /vagrant/apod/
service/apod-api created
deployment.apps/apod-api created
service/apod-log created
deployment.apps/apod-log created
service/apod-web created
deployment.apps/apod-web created
vagrant@kiamol-control:~$ kubectl get pods -o wide
NAME           READY   STATUS             RESTARTS   AGE     IP
apod-api-7f8d797c48-qzz69  0/1    ContainerCreating  0          12s   <no>
ne>   kiamol-node  <none>           <none>
apod-log-9b5cdcd9-d58dm  0/1    ContainerCreating  0          12s   <no>
ne>   kiamol-node  <none>           <none>
apod-web-7cd794886-769hg 0/1    ContainerCreating  0          12s   <no>
ne>   kiamol-node  <none>           <none>
```

All the Pods are scheduled on the standard node. None are running because that node needs to pull the container images before it can run the Pods.

Figure 18.10 The user experience is the same for all Kubernetes clusters, mostly

If this is the first time you've used a different Kubernetes cluster from your normal lab environment - then it's now you see how powerful Kubernetes is. This is a completely different setup, probably using a different version of Kubernetes, maybe a different container runtime and a different host operating system. But with one change to the spec, you can deploy and manage the APOD app in exactly the same way you previously did. You could take any of the exercises from this book and deploy them here - but you'd have to make changes to Services and Volumes so they use valid types for the new cluster.

Kubernetes stops control plane nodes from running application workloads to make sure they don't get starved of compute resources. If your control plane node is maxing CPU calculating Pi to one million decimal places, there's nothing left for the API server and DNS Pods, and your cluster becomes unusable. You should leave that safety guard in place for real clusters, but in a lab setup you can relax it to get the most out of your servers.

TRY IT NOW Kubernetes uses taints to classify nodes, and the *master* taint prevents application workloads running. Remove that taint and scale up the app to see new Pods being scheduled on the control plane node.

```
# remove the master taint from all nodes:
kubectl taint nodes --all node-role.kubernetes.io/master-
# scale up, adding two more APOD API Pods:
kubectl scale deploy apod-api --replicas=3
# print the Pods to see where they're scheduled:
```

```
kubectl get pods -l app=apod-api -o wide
```

You'll learn all about taints and scheduling in chapter 19; for now it's enough to know that taints are a way of marking particular nodes to prevent Pods running on them. Removing the taint makes the control plane node eligible to run application workloads, so when you scale up the API deployment, the new Pods get scheduled on the control plane node. You can see in figure 18.11 that those Pods are in the ContainerCreating status because each node has its own image store, and the control plane node needs to download the API image. The size of your container images directly affects the speed at which you can scale up in this scenario, which is why you need to invest in optimizing your Dockerfiles.

Removing the master taint makes the control plane node eligible to run application Pods. The second node produces the error line because it doesn't have the taint.

```
vagrant@kiamol-control:~$ kubectl taint nodes --all node-role.kubernetes.io/master-
node/kiamol-control untainted
error: taint "node-role.kubernetes.io/master" not found
```

```
vagrant@kiamol-control:~$ kubectl scale deploy apod-api --replicas=3
deployment.apps/apod-api scaled
```

```
vagrant@kiamol-control:~$ kubectl get pods -l app=apod-api -o wide
NAME          READY   STATUS        RESTARTS   AGE     IP
apod-api-7f8d797c48-gvj24  0/1    ContainerCreating  0          5s    <no
ne>          kiamol-control  <none>       <none>
apod-api-7f8d797c48-qzz69  1/1    Running      0          17m   10.244.1.2
244.1.2      kiamol-node   <none>       <none>
apod-api-7f8d797c48-rzxhp  0/1    ContainerCreating  0          5s    <no
ne>          kiamol-control  <none>       <none>
```

The two new Pods are scheduled on the control plane node. They won't start until that node has pulled the container image from Docker Hub.

Figure 18.11 You can run application Pods on the control plane nodes, but you shouldn't do so in production

The app is running and the NodePort Service means all nodes are listening on port 30000 – including the control plane node. If you browse to any node's IP address, you'll see the APOD app. Your request gets directed to the web Pod on the standard node, and it makes an API call which could be directed to a Pod on either node.

TRY IT NOW Your Virtual Machine runtime sets up your network so you can access the VMs by their IP address.
Get the address of either node and then browse to it on your host machine.

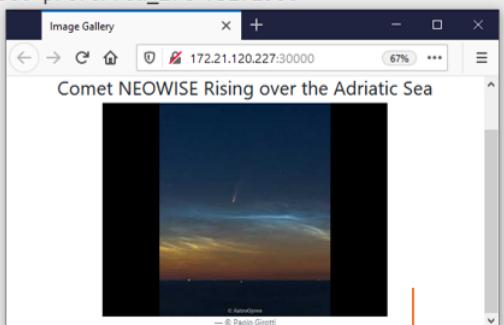
```
# print the IP address saved in the setup script:
cat /tmp/ip.txt
```

```
# browse to port 30000 on that address - if your VM
# provider uses a complicated network stack you may
# not be able to reach the VM externally :(
```

My output is in figure 18.12 - the picture I saw yesterday when I was working through the exercises was much more striking, but I didn't grab a screenshot so we've got this comet instead.

Both nodes are listening on port 30000 for the APOD web app Service. I can browse using the IP address of either node.

```
vagrant@kiamol-control:~$ ip -o -4 addr show eth0 | grep inet
2: eth0      inet 172.21.120.227/20 brd 172.21.127.255 scope global dynamic et
he\        valid_lft 75171sec preferred_lft 75171sec
vagrant@kiamol-control:~$
```



The control plane node forwards traffic to the Pod on the second node, and that Pod accesses the APOD API which is load-balanced across Pods on both nodes.

Figure 18.12 NodePort and ClusterIP Services span the Pod network so traffic can be routed to any node

Building your own Kubernetes cluster isn't all that complicated if you're happy to keep it simple - with NodePorts, HostPaths and maybe NFS volumes. If you want to you can extend this cluster and add more nodes; the Vagrant setup includes machine definitions for `kiamol-node2` and `kiamol-node3`, so you can repeat the first two exercises in this section with those VM names to build a four-node cluster. But that just gives you a boring all-Linux cluster. One of the great benefits of Kubernetes is that it can run all sorts of apps, and next we'll see how to add a different architecture to the cluster - a Windows server, so we can run all-Windows or hybrid Linux-Windows apps.

18.4 Adding Windows nodes and running hybrid workloads

The Kubernetes website itself says *Windows applications constitute a large portion of the services and applications that run in many organizations*, so bear that in mind if you're thinking of skipping this section. I won't go into a lot of detail about Windows containers and the

differences from Linux (for that you can read my book *Docker on Windows*) — just the basics so you can see how Windows apps fit in Kubernetes.

Container images are built for a specific architecture - a combination of operating system and CPU. Containers use the kernel of the machine they're running on, so that has to match the architecture of the image. You can build a Docker image on a Raspberry Pi but you can't run it on your laptop because the Pi uses an Arm CPU and your laptop uses Intel. It's the same with the operating system - you can build an image on a Windows machine to run a Windows app, but you can't run that in a container on a Linux server. Kubernetes supports different types of workload by having nodes with different architectures in the same cluster.

There are some restrictions to how diverse your cluster can be, but the diagram in figure 18.13 is something you can genuinely build. The control plane is Linux-only but the Kubelet and proxy are cross-platform. AWS has Arm-powered servers which are almost half the price of Intel equivalents and you can use them as nodes in EKS - so if you have a large application suite with some apps that work in Linux on Arm, some that need Linux on Intel and some that are Windows, you can run and manage them all in one cluster.

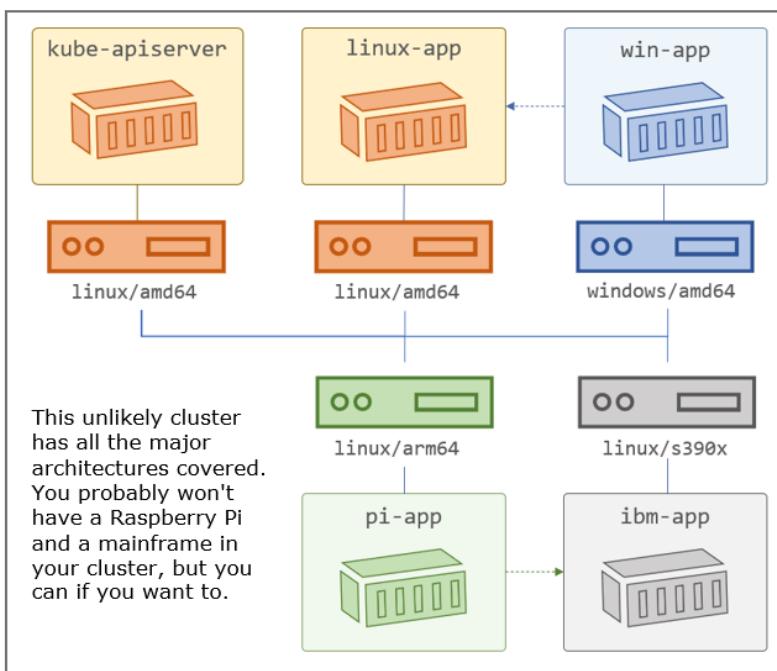


Figure 18.13 Who hasn't got a spare Raspberry Pi and IBM Z mainframe gathering dust?

Let's get to it and add a Windows node to your cluster. The approach is the same as adding a Linux node - spin up a new VM, add a container runtime and the Kubernetes tools and join it to

the cluster. Windows Server 2019 is the minimum version which is supported in Kubernetes, and Docker is the only container runtime available at the time of writing - containerd support for Windows is in progress.

TRY IT NOW Create a Windows VM and install the Kubernetes pre-requisites. Folder sharing in Vagrant doesn't always work for Windows so you'll download the setup script from the book's source on GitHub.

```
# spin up a Windows Server 2019 machine:  
vagrant up kiamol-node-win  
  
# connect - you need the password which is vagrant  
vagrant ssh kiamol-node-win  
  
# switch to PowerShell:  
powershell  
  
# download the setup script:  
curl.exe -s -o windows-setup.ps1  
    https://raw.githubusercontent.com/sixeyed/kiamol/master/ch18/setup/windows-  
    setup.ps1  
  
# run the script - this reboots the VM when it finishes:  
.windows-setup.ps1
```

The Windows setup needs to enable an operating system feature and install Docker; the script reboots the VM when that's done - you can see in figure 18.14 that my session is back to the normal command line.

This creates a Windows Server VM - the Vagrant process is the same as with Linux VMs.

```
PS>vagrant up kiamol-node-win
Bringing machine 'kiamol-node-win' up with 'hyperv' provider...
==> kiamol-node-win: Verifying Hyper-V is enabled...
==> kiamol-node-win: Verifying Hyper-V is accessible...
==> kiamol-node-win: Importing a Hyper-V instance
PS C:\Users\vagrant> curl.exe -s -O https://raw.githubusercontent.com/sixeyed/kiamol/master/ch17/setup/windows-setup.ps1
PS C:\Users\vagrant>
PS C:\Users\vagrant> ./windows-setup.ps1
```

Success	Restart	Needed	Exit Code	Feature	Result
True	Yes			SuccessRest...	{Containers}

WARNING: You must restart this server to finish the installation process.

```
PS C:\Users\vagrant>
PS>
```

The setup script enables container support and installs Docker.

When the script finishes it reboots the VM. Hit enter to return to your terminal if the VM session stops responding.

Figure 18.14 The first stage in adding a Windows node is installing the container runtime

This is only the first part of the setup, because the control plane needs to be configured for Windows support. The standard deployment of Flannel and the Kube proxy doesn't create a DaemonSet for Windows nodes, so we need to set that up as an additional step. The new DaemonSets use Windows container images in the spec and the Pods are configured to work with the Windows server to set up networking.

TRY IT NOW Deploy the new system components for the Windows node.

```
# connect to the control plane:
vagrant ssh kiamol-control

# create the Windows proxy:
kubectl apply -f /vagrant/setup/kube-proxy.yml

# create the Windows network:
kubectl apply -f /vagrant/setup/flannel-overlay.yml

# confirm the new DaemonSets are there:
kubectl get ds -n kube-system
```

Again, this is something you would add to the initial cluster setup if you're serious about running your own hybrid cluster. I've kept it as a separate step so you can see what needs to change to add Windows support - which you can do with an existing cluster, provided it's running

Kubernetes 1.14 or greater. My output in figure 18.15 shows the new Windows-specific DaemonSets, with a desired count of zero because the Windows node hasn't joined yet.

Creates configuration and DaemonSets to run the system components on Windows nodes.

```
vagrant@kiamol-control:~$ kubectl apply -f /vagrant/setup/kube-proxy.yml
configmap/kube-proxy-windows created
daemonset.apps/kube-proxy-windows created
vagrant@kiamol-control:~$
vagrant@kiamol-control:~$ kubectl apply -f /vagrant/setup/flannel-overlay.y
1
configmap/kube-flannel-windows-cfg created
daemonset.apps/kube-flannel-ds-windows-amd64 created
vagrant@kiamol-control:~$
vagrant@kiamol-control:~$ kubectl get ds -n kube-system
NAME          DESIRED   CURRENT   READY   UP-TO-DATE   AVA
LABEL        NODE SELECTOR   AGE
kube-flannel-ds-amd64   2         2         2         2           2
<none>           4h3m
kube-flannel-ds-windows-amd64  0         0         0         0           0
<none>           34s
kube-proxy       2         2         2         2           2
  kubernetes.io/os=linux  4h19m
kube-proxy-windows 0         0         0         0           0
  kubernetes.io/os=windows 41s
```

There are two desired nodes for the Linux proxy, but none for the Windows proxy or the Windows Flannel network (I've snipped the other architectures from the output). As soon as the Windows node joins, the new DaemonSets will schedule Pods.

Figure 18.15 Updating the control plane to schedule system components on Windows nodes

Join the Windows node and it will have Pods scheduled for the proxy and the network components, so it will download the images and start containers. One difference is that Windows containers are more restricted than Linux containers so the Flannel setup is slightly different - the Kubernetes Special Interest Group (SIG) for Windows publishes a helper script to set up Flannel and the Kubelet. I've got a snapshot of that script in the source for this chapter, which matches the Kubernetes version we're running; after this second setup script the node is ready to join.

TRY IT NOW Add the remaining dependencies to the Windows node and join it to the cluster.

```
# connect to the Windows node:
vagrant ssh kiamol-node-win

# run PowerShell:
powershell

# download the second setup script:
curl.exe -s -o PrepareNode.ps1
```

```

https://raw.githubusercontent.com/sixeyed/kiamol/master/ch18/setup/PrepareNode.ps1

# run the script:
.\PrepareNode.ps1

# run the join command - remember to use your command, this
# is just a reminder of how the command looks:
kubeadm join 172.21.120.227:6443
--token 5wbq7j.bew48gsfy0maa2bo
--discovery-token-ca-cert-hash
sha256:2c520ea15a99bd68b74d04f40056996dff5b6ed1e76dfaeb0211c6db18ba0393

```

The happy message "this node has joined the cluster" that you see in figure 18.16 is a bit ahead of itself. The new node needs to download the proxy and network images - and the Flannel image is 5GB so it will take a few minutes before the Windows node is ready.

This script downloads Kubeadm and sets up Flannel and the Kubelet on the Windows node.

There's lots of output, so I've trimmed it here. When the script finishes you can run your Kubeadm join command.

```

PS C:\Users\vagrant> curl.exe -s -o PrepareNode.ps1 https://raw.githubusercontent.com/sixeyed/kiamol/master/ch17/setup/PrepareNode.ps1
PS C:\Users\vagrant>
PS C:\Users\vagrant> .\PrepareNode.ps1
Using Kubernetes version: v1.18.5

Action          : Allow
Direction       : Inbound
DisplayGroup    :
DisplayName     : kubelet

PS C:\Users\vagrant> kubeadm join 172.21.120.227:6443 --token 5wbq7j.bew48gsfy0maa2bo --discovery-token-ca-cert-hash sha256:2c520ea15a99bd68b74d04f40056996dff5b6ed1e76dfaeb0211c6db18ba0393
W0713 15:38:40.229522    1760 join.go:346] [preflight] WARNING: JoinControlPlane.settings will be ignored when control-plane flag is not set
.
[preflight] Running pre-flight checks

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.

The joining process is the same as a Linux node, with the pre-flight checks and the TLS bootstrap. The Windows server is a Kubernetes node now but it needs to download the system images before it's ready to run workloads.

```

Figure 18.16 Joining the Windows node uses the same Kubeadm command as the Linux node

While that's downloading we'll look at modelling applications to run on different architectures. Kubernetes doesn't automatically work out which nodes are suitable for which Pods – there are a few intricate reasons why that isn't easy to do from the image name alone. Instead you add a selector in your Pod spec to specify the architecture the Pod needs. Listing 18.2 shows a selector which sets the Pod to run on a Windows node.

Listing 18.2 - api.yaml, using a node selector to request a specific operating system

```
spec:                                     # Pod spec in the Deployment
  containers:
    - name: api
      image: kiamol/ch03-numbers-api:windows  # a Windows-specific image
  nodeSelector:
    kubernetes.io/os: windows               # select nodes with Windows OS
```

That's it. Remember that Pods run on a single node, so if you have multiple containers in your Pod spec they all need to use the same architecture. It's a good practice to include a node selector for every Pod if you're running a multi-architecture cluster to ensure Pods always end up where they should – you can include an operating system, CPU architecture or both.

The listing in 18.2 is for a Windows version of the random number API and there's a Linux version of the website to go with it. The web spec includes a node selector for the Linux OS. You can deploy the app and the Pods will run on different nodes, but the website still accesses the API Pod in the usual way through a ClusterIP Service, even though it's running on Windows.

TRY IT NOW This is a hybrid application, with one Windows and one Linux component. Both use the same YAML format, and in the specs only the node selectors show they need to run on different architectures.

```
# connect to the control plane:
vagrant ssh kiamol-control

# wait for all the nodes to be ready:
kubectl -n kube-system wait --for=condition=Ready node --all

# deploy the hybrid app:
kubectl apply -f /vagrant/numbers

# wait for the Windows Pod to be ready:
kubectl wait --for=condition=ContainersReady pod -l app=numbers,component=api

# confirm where the Pods are running:
kubectl get pods -o wide -l app=numbers

# browse to port 30001 on any node to use the app
```

It's a shame the demo app in figure 18.17 is so basic, because this capability has taken many years and a ton of effort from the Kubernetes community and the engineering teams at Microsoft and Docker.

Ensures the Windows node has loaded system components and is ready to run workloads.

The web Deployment uses a Linux node selector and the API Deployment uses a Windows node selector.

```
vagrant@kiamol-control:~$ kubectl -n kube-system wait --for=condition=Ready
node --all
node/kiamol-control condition met
node/kiamol-node condition met
node/kiamol-node-win condition met
vagrant@kiamol-control:~$
vagrant@kiamol-control:~$ kubectl apply -f /vagrant/numbers
service/numbers-api created
deployment.apps/numbers-api created
service/numbers-web created
deployment.apps/numbers-web created
vagrant@kiamol-control:~$
vagrant@kiamol-control:~$ kubectl wait --for=condition=ContainersReady pod -l app=numbers,component=api
pod/numbers-api-8dff5b664-w7vfd condition met
vagrant@kiamol-control:~$
vagrant@kiamol-control:~$ kubectl get pods -o wide -l app=numbers
NAME                  READY   STATUS    RESTARTS   AGE     IP
NODE      NOMINATED-NODE  READINESS GATES
numbers-api-8dff5b664-w7vfd  1/1     Running   0          72s   10.244.2.4
kiamol-node-win <none>        <none>
numbers-web-8646d64c56-xzv1b  1/1     Running   0          72s   10.244.0.8
kiamol-control <none>        <none>
vagrant@kiamol-control:~$
```

The cluster has scheduled the API on the Windows node and the web app on the control plane. They communicate using the standard Pod network.

And the app works as expected.

Figure 18.17 A world-class container orchestrator running a hybrid app to generate one random number

By day I work as a consultant helping companies adopt container technologies and the pattern here is exactly what lots of organizations want to do with their Windows apps - move them to Kubernetes without any changes, and then gradually break up the monolithic architecture with new components running in lightweight Linux containers. It's a pragmatic and low-risk approach to modernizing applications that takes full advantage of all the features of Kubernetes, and it gives you an easy migration to the cloud.

Way back in chapter 1 I said that Kubernetes runs your applications, but it doesn't really care what those applications are. We'll prove that with one last deployment to the cluster - the

Windows Pet Shop app. This was a demo app Microsoft built in 2008 to showcase the latest features of .NET. It uses technologies and approaches which have long since been replaced, but the source code is still out there and I've packaged it up to run in Windows containers and published the images on Docker Hub. This exercise shows you that you really can run decade-old applications in Kubernetes without any changes to the code.

TRY IT NOW Deploy a legacy Windows app - this one downloads more large container images, so it will take a while to start.

```
# on the control plane, deploy the Petshop app:  
kubectl apply -f /vagrant/petshop/  
  
# wait for all the Pods to start - it might need more than five minutes:  
kubectl wait --for=condition=ContainersReady pod -l app=petshop --timeout=5m  
kubectl get pods -o wide -l app=petshop  
  
# browse to port 30002 on any node to see the app
```

And there you have it. Figure 18.18 could be faked - but it's not, and you can run this all yourself to prove it (I will admit it took me two attempts - my Windows VM lost network connectivity the first time around, which is most likely a Hyper-V issue). The Pet Shop is an app which last had a code change 12 years ago, now running in the latest version of Kubernetes.

The Pet Shop app is all-Windows. Windows apps can use all the usual Kubernetes features, like Services, Secrets and ConfigMaps.

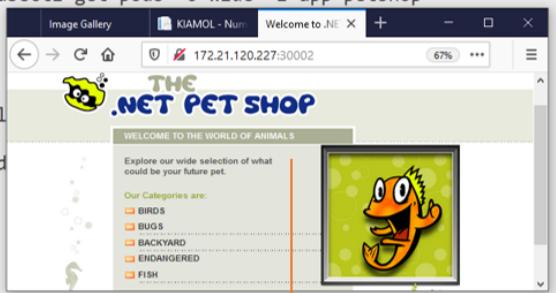
The node needs to download and extract three images, which is a few extra gigabytes so it might take a while for the app to start.

```
vagrant@kiamol-control:~$ kubectl apply -f /vagrant/petshop/
secret/petshop-connection-string-secret created
service/petshop-db created
deployment.apps/petshop-db created
service/petshop-web created
deployment.apps/petshop-web created
service/petshop-webservice created
deployment.apps/petshop-webservice created
vagrant@kiamol-control:~$
```

```
vagrant@kiamol-control:~$ kubectl wait --for=condition=ContainersReady pod -l app=petshop --timeout=5m
pod/petshop-db-b4b77968b-xhmnt condition met
pod/petshop-web-57fc985779-wfxll condition met
pod/petshop-webservice-5fd74d9fd-h8rtm condition met
vagrant@kiamol-control:~$
```

```
vagrant@kiamol-control:~$ kubectl get pods -o wide -l app=petshop
NAME
```

NODE
petshop-db-b4b77968b-xhmnt 244.2.8 kiamol-node-win
petshop-web-57fc985779-wfxll 244.2.9 kiamol-node-win
petshop-webservice-5fd74d9fd-h8rtm 244.2.7 kiamol-node-win



All Pods are running on the Windows node.

Welcome to the world of animals, 2008-style.

Figure 18.18 I bet there hasn't been a screenshot of the Pet Shop in a book for a long time

That's as far as we'll go with this cluster - although if you want to add more Windows nodes you can repeat the setup and join exercises from this section for two more machines defined in Vagrant: `kiamol-node-win2` and `kiamol-node-win3`. You'll just about squeeze the control plane node, three Linux nodes and three Windows nodes onto your machine if you have at least 16GB of memory. We'll finish up with a look at the considerations for multi-node Kubernetes clusters, and the future of multi-architecture.

18.5 Understanding Kubernetes at scale

Whether you diligently followed the exercises in this chapter or just skimmed through, you've got a good idea how complex it is to set up and manage a Kubernetes cluster, and you'll understand why I recommend Docker Desktop or kind for your lab environment. Deploying a

multi-node cluster is a great learning exercise to see how all the pieces fit together, but it's not something I'd recommend for production.

Kubernetes is all about high-availability and scale, and the more nodes you have the more complex it becomes to manage. You need multiple control plane nodes for high availability; if you lose the control plane then your apps keep running on the nodes but you can't manage them with Kubectl and they're no longer self-healing. The control plane stores all its data in etcd and for better redundancy you can run etcd outside of the cluster. For a performance improvement you can run an additional etcd database just to store the events Kubernetes records for objects. It's all starting to look complicated, and we're still only dealing with a single cluster, not high availability across multiple clusters.

You can build Kubernetes clusters that run at huge scale - the latest release supports up to 5,000 nodes and 150,000 Pods in a single cluster. In practice you're likely to hit performance issues with etcd or your network plugin when you get to around 500 nodes and you'll need to look at scaling parts of the control plane independently. The good news is that you can run a cluster with hundreds of worker nodes managed by a control plane with just three nodes, provided those nodes are fairly powerful. The bad news is you need to manage all that, and you'll need to decide if you're better off with one large cluster or multiple smaller clusters.

The other side of scale is being able to run as much of your application catalog as possible on a single platform. You saw in this chapter that you can run a multi-architecture cluster by adding Windows nodes - Arm and IBM nodes work in the same way - which means you can run pretty much anything in Kubernetes. Older applications bring their own challenges but one of the big advantages of Kubernetes is that you don't need to rewrite those apps. There are benefits to breaking down monolithic apps into a more cloud-native architecture, but that can be part of a longer term program, which starts with moving apps to Kubernetes as-is.

And we're out of room to continue the discussion. You should leave your cluster running for the lab, but when you're done come back to this final exercise to clear it down.

TRY IT NOW You have a few options for closing down your cluster - choose one after you've had a go at the lab.

```
# suspend the VMs - this preserves state so the VMs
# still consume disk or memory:
vagrant suspend

# OR stop the VMs - you can start them again, but they might
# get new IP addresses and then your cluster won't be accessible:
vagrant halt

# OR if you're really done with the cluster, delete everything:
vagrant destroy
```

18.6 Lab

Easy lab for this chapter, but it will need some research. Time passes after you deploy your cluster, and at some point the nodes will need maintenance work. Kubernetes lets you safely

take a node out of the cluster - moving its Pods to another node - and then bring it back online when you're done. Let's do that for the Linux node in the cluster. Just one hint: you can do it all with Kubectl.

This is a useful lab to work through because temporarily removing a node from service is something you'll want to do whichever platform you're using. My solution is up on GitHub for you in the usual place: <https://github.com/sixeyed/kiamol/blob/master/ch18/lab/README.md>

19

Controlling workload placement and automatic scaling

Kubernetes decides where to run your workloads, spreading them around the cluster to get the best use of your servers and the highest availability for your apps. Deciding which node is going to run a new Pod is the job of the scheduler, one of the control plane components. The scheduler uses all the information it can get to choose a node - it looks at the compute capacity of the server and the resources used by existing Pods. It also uses policies which you can hook into in your application specs to have more control over where Pods will run. In this chapter you'll learn how to direct Pods to specific nodes, and how to schedule the placement of Pods in relation to other Pods.

There are two other sides of workload placement which we'll also cover in this chapter: automatic scaling and Pod eviction. Scaling lets you specify the minimum and maximum number of replicas for your app, along with some metric for Kubernetes to measure how hard your app is working. If the Pods are overworked, the cluster scales up automatically, adding more replicas - and scales down again when the load reduces. Eviction is the extreme scenario where nodes are maxing out resources, and Kubernetes removes Pods to keep the server stable. There's some intricate detail coming up, but it's important to understand the principles to get the right balance of a healthy cluster and high performing apps.

19.1 How Kubernetes schedules workloads

Create a new Pod and it goes into the pending state until it's allocated to a node. The scheduler sees the new Pod and tries to find the best node to run it on. There are two parts to the scheduling process: first is *filtering* which excludes any unsuitable nodes, and then *scoring* to rank the remaining nodes and choose the best option. Figure 19.1 shows a simplified example.

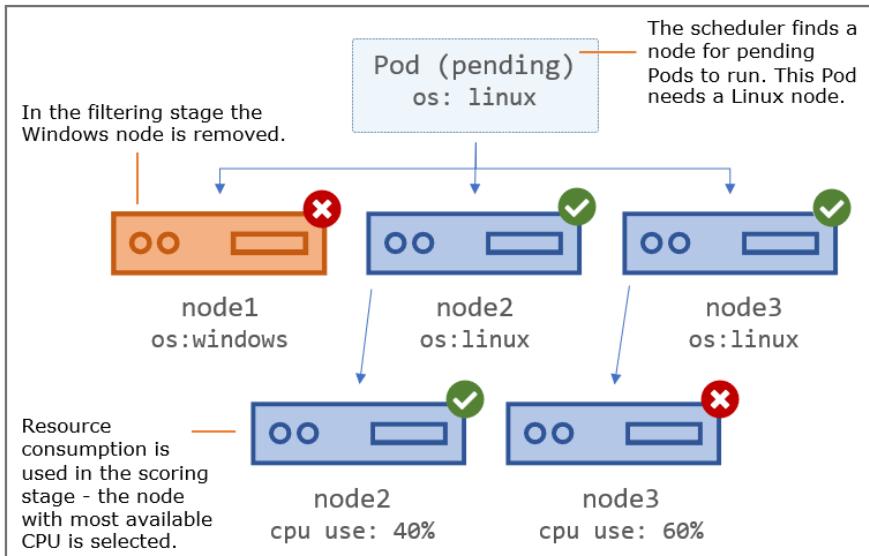


Figure 19.1 The scheduler selects nodes on their suitability and their current workload

You've already seen the filtering stage in action in chapter 17, when you learned that the control plane node is isolated from application workloads. That's done with a taint which is a way of flagging a node to say it isn't suitable for general work. The `master` taint is applied to control plane nodes by default, but taints are really like a specialized type of label and you can add your own taints to nodes. Taints have a key-value pair just like a label, and they also have an effect which tells the scheduler how to treat this node. You'll use taints to identify nodes which are different from the rest - in the next exercise we'll add a taint to record the type of disk a node has.

TRY IT NOW Run a simple sleep app and then add a taint to your node to see how it affects the workload.

```
# switch to the chapter's source:
cd ch19

# print the taints already on the nodes:
kubectl get nodes -o=jsonpath='{range
    .items[*]}{.metadata.name}{.spec.taints[*].key}{end}'

# deploy the sleep app:
kubectl apply -f sleep/sleep.yaml

# add a taint to all nodes:
kubectl taint nodes --all kiamol-disk=hdd:NoSchedule

# confirm the sleep Pod is still running:
kubectl get pods -l app=sleep
```

The key-value part of a taint is arbitrary, and you can use it to record whatever aspect of the node you care about - maybe some nodes have less memory or a slower network card. The effect of this taint is `NoSchedule` which means workloads won't be scheduled on this node unless they explicitly *tolerate* the taint. As you see in figure 19.2, applying a `NoSchedule` taint doesn't impact existing workloads - the sleep Pod is still running after the node has been tainted.

```
Lists nodes and their taints - I have a single node with no taints.

PS>cd ch18
PS>
PS>kubectl get nodes -o=jsonpath='{range .items[*]}{.metadata.name}{.spec.taints[*].key}{end}'
docker-desktop

PS>
PS>kubectl apply -f sleep/sleep.yaml
deployment.apps/sleep created
PS>
PS>kubectl taint nodes --all kiamol-disk=hdd:NoSchedule
node/docker-desktop tainted
PS>
PS>kubectl get pods -l app=sleep
NAME           READY   STATUS    RESTARTS   AGE
sleep-85fdd4cf75-xqg8j   1/1     Running   0          16s

Tainting a node is like adding a label, but
with an effect which impacts scheduling.

Adding a NoSchedule taint doesn't affect existing workloads - the sleep Pod is still running.
```

Figure 19.2 Pods need a toleration to run on a tainted node, unless they were running before the taint

Now the taint is in place the node will be filtered out by the scheduler for all new Pods, unless the Pod spec has a toleration for the taint. Tolerations say the workload acknowledges the taint and is happy to work with it. In this example we've flagged nodes with spinning disks which are probably lower performers than nodes with solid state disks - listing 19.1 includes a toleration to say this Pod is happy to run on these slower nodes.

Listing 19.1 - sleep2-with-tolerations.yaml, tolerating a tainted node

```
spec:                                     # the Pod spec in a Deployment
  containers:
    - name: sleep
      image: kiamol/ch03-sleep
  tolerations:
    - key: "kiamol-disk"        # lists taints this Pod is happy with
      operator: "Equal"         # the key, value and effect all need
      value: "hdd"              # to match the taint on the node
      effect: "NoSchedule"
```

We tainted every node in the last exercise so the scheduler will filter them all out for a new Pod request, unless the spec contains a toleration. Pods which can't be scheduled stay in the pending state until something changes - a new node joins without the taint, or the taint is removed from an existing node, or the Pod spec changes. As soon as a change happens and the scheduler can find a suitable placement, the Pod will get scheduled to run.

TRY IT NOW Try deploying a copy of the sleep app without a toleration - it will stay as pending. Update it to add the toleration and it will run.

```
# create a Pod without a toleration:
kubectl apply -f sleep/sleep2.yaml

# confirm the Pod is pending:
kubectl get po -l app=sleep2

# add the toleration from listing 19.1:
kubectl apply -f sleep/update/sleep2-with-tolerations.yaml
kubectl get po -l app=sleep2
```

That exercise used a Deployment, so the toleration actually got added to a new Pod and the new Pod got scheduled - you can see that in figure 19.3. But if you create a plain Pod without the toleration it will go into the pending state, and when you add the toleration, that same Pod will be scheduled; the scheduler keeps trying to find a node for any unscheduled Pods.

My node is tainted so any new Pods without a toleration can't be scheduled and they stay as pending.

```
PS>kubectl apply -f sleep/sleep2.yaml
deployment.apps/sleep2 created
PS>
PS>kubectl get po -l app=sleep2
NAME          READY   STATUS    RESTARTS   AGE
sleep2-585dd5c567-gltjx  0/1     Pending   0          6s
PS>
PS>kubectl apply -f sleep/update/sleep2-with-tolerations.yaml
deployment.apps/sleep2 configured
PS>
PS>kubectl get po -l app=sleep2
NAME          READY   STATUS    RESTARTS   AGE
sleep2-864f4964fc-tsbrl  1/1     Running   0          9s
```

Add a toleration and the app can run.
Tolerations are part of the Pod spec, so this is a change and the Deployment creates a new ReplicaSet. The Pod can start because it tolerates the tainted node.

Figure 19.3 If the Pod's toleration matches the node's taint then it can run on that node

The `NoSchedule` effect is a hard taint - it's comes into the scheduler's filtering stage so Pods won't run on tainted nodes unless they have a toleration. A softer alternative is `PreferNoSchedule` which moves the restriction to the scoring stage - nodes aren't filtered out, but they score lower than a node which doesn't have the taint. A `PreferNoSchedule` taint means Pods shouldn't run on that node unless they have a toleration for it, unless there are no other suitable nodes.

It's important to understand that taints and tolerations are for expressing something negative about the node which means it only suits certain Pods; it's not a positive association between a node and a Pod. A Pod with a toleration might run on a tainted node or it might not, so tolerations are not a good mechanism for ensuring Pods *only* run on certain nodes. You might need that for something like PCI compliance, where financial apps should only run on nodes which have been hardened. For that you need to use a `NodeSelector` which filters out nodes based on their labels - we used that in chapter 17 to make sure Pods ran on the correct CPU architecture. Listing 19.2 shows that different types of scheduler hints work together.

Listing 19.2 - sleep2-with-nodeSelector.yaml, a toleration and a node selector

```
spec:
  containers:
    - name: sleep
      image: kiamol/ch03-sleep
  tolerations: # the Pod tolerates nodes
    - key: "kiamol-disk" # with the hdd taint
      operator: "Equal"
      value: "hdd"
      effect: "NoSchedule"
  nodeSelector: # but will only run on nodes
    kubernetes.io/arch: zxSpectrum # which match this CPU type
```

This spec says the Pod will tolerate a node with the hard disk taint, but the architecture has to be a ZX Spectrum. You won't have a ZX Spectrum in your cluster so when you deploy this the new Pod won't be scheduled - I've chosen that CPU not just out of nostalgia, but to highlight that these labels are just key-value pairs, with no validation in them. The `os` and `arch` labels are set by Kubernetes on the nodes, but in your Pod spec you can use incorrect values and leave your Pods pending.

TRY IT NOW Deploy the sleep app update from listing 19.2 and see if there's a matching node on your cluster.

```
# show the node's labels:
kubectl get nodes --show-labels

# update the Deployment with an incorrect node selector:
kubectl apply -f sleep/update/sleep2-with-nodeSelector.yaml

# print the Pod status:
kubectl get pods -l app=sleep2
```

You can see in my output in figure 19.4 why we're using a Deployment for this app. The new Pod goes into the pending state and it will stay there until you add a ZX Spectrum to your cluster (which would mean building an 8-bit version of the Kubelet) - but the app is still up because the Deployment won't scale down the old ReplicaSet until the replacement is at the desired capacity.

Kubernetes applies lots of labels itself and you can add your own - they all behave in the same way, and you can modify or remove system labels.

```
PS>kubectl get nodes --show-labels
NAME      STATUS   ROLES   AGE     VERSION   LABELS
docker-desktop   Ready    master   16h    v1.18.3   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubernetes.io/host-name=docker-desktop,kubernetes.io/os=linux,node-role.kubernetes.io/master=
PS>
PS>kubectl apply -f sleep/update/sleep2-with-nodeSelector.yaml
deployment.apps/sleep2 configured
PS>
PS>kubectl get pods -l app=sleep2
NAME        READY   STATUS    RESTARTS   AGE
sleep2-55569bdbcd-22hxw  0/1     Pending   0          12s
sleep2-864f4964fc-tsbrl  1/1     Running   0          3h54m
```

This update adds a node selector which the scheduler can't match.
The app keeps running on the previous spec; the new spec Pod will stay pending.

Figure 19.4 Pods which can't be scheduled don't interrupt the app if it's running in a Deployment

Node selectors ensure that apps only run on nodes with specific label values, but you usually want some more flexibility than a straight equality match. A finer level of control comes with *affinity* and *anti-affinity*.

19.2 Directing Pod placement with affinity and anti-affinity

Kubernetes applies a standard set of labels to nodes, but standards change over time. The system-provided labels are prefixed with a namespace, and the namespace is versioned in the same way that the API for object specs is versioned. New clusters use `kubernetes.io` as the label prefix, but older versions use `beta.kubernetes.io`. The beta tag identifies that a feature isn't stable and the specification might change, but features can stay in beta through lots of Kubernetes versions. If you want Pods restricted to a certain architecture you need to allow for the beta namespace to make your spec portable across different versions of Kubernetes.

Affinity provides a rich way of expressing preferences or requirements to the scheduler. You can claim an affinity to certain nodes to ensure Pods land on those nodes. Affinity uses a node selector, but with a match expression rather than a simple equality check. Match expressions support multiple clauses, so you can build much more complex requirements. Listing 19.3 uses

affinity to say the Pod should run on a 64-bit Intel node, in a way which works for new and old clusters.

Listing 19.3 - sleep2-with-nodeAffinity-required.yaml, a Pod with Node affinity

```
affinity:                                # affinity expresses a requirement
  nodeAffinity:                          # or a preference for nodes
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:                # match expressions work on
          - key: kubernetes.io/arch       # labels, and you can supply
            operator: In                 # a list of values which
            values:                      # should match or not match
              - amd64
        - matchExpressions:                # multiple match expressions
          - key: beta.kubernetes.io/arch # work as a logical OR
            operator: In
            values:
              - amd64
```

The terrifying-sounding `requiredDuringSchedulingIgnoredDuringExecution` just means that this is a hard rule for the scheduler but it won't affect any existing Pods - they won't be removed once they're scheduled, even if the node labels change. The two match expressions cover the either/or case for new and old label namespaces, replacing the simpler node selector in listing 19.2. The full spec for listing 19.3 contains the hard-disk toleration, so when you deploy this the sleep app will stop waiting for a Spectrum to join the cluster and will run on your Intel node.

TRY IT NOW Update the sleep app - you should have one of the two architecture labels on your node, so the new Pod will run and replace the existing one.

```
# deploy the spec from listing 19.3:
kubectl apply -f sleep/update/sleep2-with-nodeAffinity-required.yaml

# confirm the new Pod runs:
kubectl get po -l app=sleep2
```

Figure 19.5 should hold no surprises for you, it just shows the new affinity rules being put in place. Now the scheduler can find a node which suits the requirements so the Pod runs.

The new affinity rules replace the node selector and now there is a match.

```
PS>kubectl apply -f sleep/update/sleep2-with-nodeAffinity-required.yaml
deployment.apps/sleep2 configured
```

```
PS>
PS>kubectl get pods -l app=sleep2
```

NAME	READY	STATUS	RESTARTS	AGE
sleep2-6c89f6f7b6-59cqr	1/1	Running	0	8s

The new Pod runs and replaces the previous Pod and the pending Pod.

Figure 19.5 Affinity allows for a more complex set of node selector rules

I've got one more example of node affinity because the syntax is a bit fiddly, but it's good to know what you can do with it. Node affinity is a clean way to express scheduling requirements which combine hard rules and soft preferences, and you can do more than you can with tolerations and plain node selectors. Listing 19.4 is an abbreviation of a spec which says to the scheduler: Pods must run on an Intel node and it must be Windows or Linux, but preferably Linux.

Listing 19.4 - sleep2-with-nodeAffinity-preferred.yaml, requirements and preferences

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/arch
              operator: In
              values:
                - amd64
            - key: kubernetes.io/os
              operator: In
              values:
                - linux
                - windows
    preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
            - key: kubernetes.io/os
              operator: In
              values:
                - linux
```

This is great if you have a multi-architecture cluster which is Linux-heavy, with just a few Windows nodes to run older applications. You can build Docker images which are multi-architecture, so the same image tag works on Linux and Windows (or Arm or any of the other OS and architecture combinations) so one container spec is good for multiple systems. Pods with this spec will prefer Linux nodes, but if the Linux nodes are saturated and there's capacity on the Windows nodes, then we'll use that capacity and run Windows Pods instead.

The affinity syntax is a little unwieldy because it's so generic. In the *required* rules multiple match expressions work as a logical AND, and multiple selectors work as an OR. In the *preferred* rules, multiple match expressions are an AND, and you use multiple preferences to describe an OR. The full spec of listing 19.4 includes OR logic to cover multiple namespaces; we won't run it because the output is the same as the last exercise, but it's a good one to refer back to if you're struggling to express your affinity. Figure 19.6 shows how the rules look.

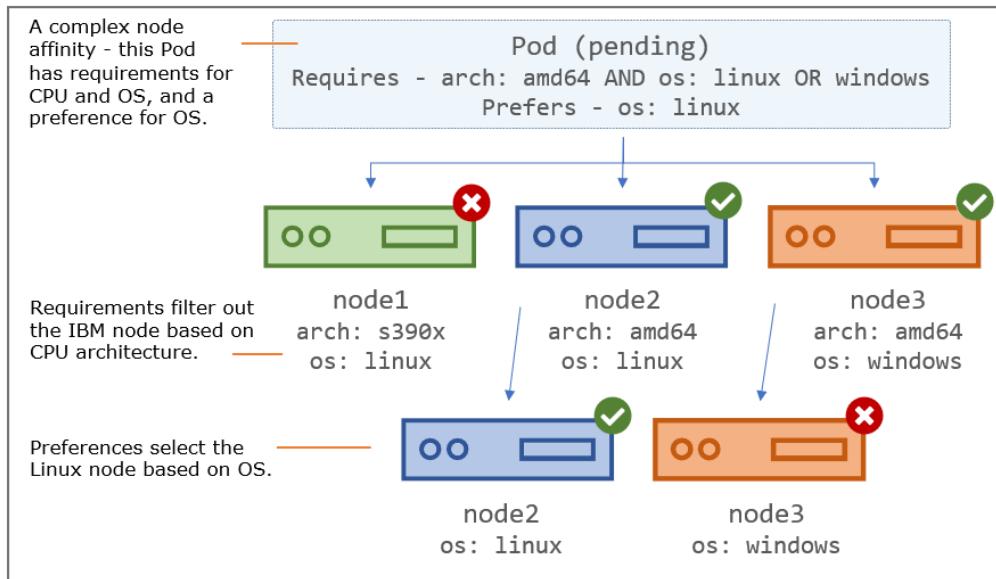


Figure 19.6 You can express affinity rules with multiple conditions using different node labels

You should get a good grasp of affinity because you can use it with more than just nodes - Pods can express an affinity to other Pods so they get scheduled on the same node, or an anti-affinity so they get scheduled on a different node. This supports two common use-cases. The first is where you want Pods for different components to be co-located to reduce network latency between them. The second is where you want replicas of the same component to be spread around the cluster to increase redundancy. Listing 19.5 shows that first scenario in the random number web app.

Listing 19.5 - web.yaml, Pod affinity to co-locate components

```

affinity:                                # affinity rules for Pods use
podAffinity:                             # the same spec as node affinity
  requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:          # this looks for the app and
          - key: app               # components labels to match
            operator: In
            values:
              - numbers
        - key: component
          operator: In
          values:
            - api
topologyKey: "kubernetes.io/hostname"

```

Pod affinity follows the same spec as node affinity, and you can include both if you really want to confuse your team. Listing 19.4 is a required rule so Pods will be left pending if the scheduler can't fulfill it. The match expressions work as a label selector, so this says the Pod must be scheduled on a node which is already running a Pod with the labels `app=numbers` and `component=api`. That's co-location, so a web Pod will always have a local API pod. Which just leaves the `topology` key to describe, and that will need its own paragraph.

Topology describes the physical layout of your cluster - where the nodes are located, which is set in node labels at different levels of detail. The hostname label is always present and is unique for the node, and clusters can add their own detail. Cloud providers usually add region and zone labels which state where the server is located; on-premises clusters might add datacenter and rack labels. A topology key sets the level where the affinity applies - hostname effectively means put the Pods on the same node, and zone would mean put the Pod on any node in the same zone as the other Pod. Hostname is a good enough topology key to see affinity in action, and you can do it on a single node.

TRY IT NOW Deploy the random number app with Pod affinity to ensure the web and API Pods run on the same node.

```
# remove the taint we applied to simplify things:
kubectl taint nodes --all kiamol-disk=hdd:NoSchedule-

# deploy the random number app:
kubectl apply -f numbers/

# confirm that both Pods are scheduled on the same node:
kubectl get pods -l app=numbers -o wide
```

You can see in figure 19.7 that this works as expected - I only have one node and of course both Pods are scheduled there. In a larger cluster the web Pod would stay pending until the API Pod got scheduled, and then it would follow the API Pod to the same node. If that node didn't have capacity to run the web Pod then it would stay as pending because this is a required rule (a preferred rule would allow the Pod to run on a node without an API Pod).

Using the taint command with a minus at the end removes a taint.

The web Pod spec expresses an affinity to the API Pod.

```
PS>kubectl taint nodes --all kiamol-disk=hdd:NoSchedule-
node/docker-desktop untainted
PS>
PS>kubectl apply -f numbers/
service/numbers-api created
deployment.apps/numbers-api created
service/numbers-web created
deployment.apps/numbers-web created
PS>
PS>kubectl get pods -l app=numbers -o wide
NAME                  READY   STATUS    RESTARTS   AGE     IP
  NODE           NOMINATED-NODE   READINESS GATES
numbers-api-7f759884bd-rscj1   1/1     Running   0          5s    10.1.0.76
  docker-desktop <none>        <none>
numbers-web-657ccf9b66-8jg89   1/1     Running   0          5s    10.1.0.77
  docker-desktop <none>        <none>
```

Both Pods are scheduled on the same node - you get the same result on a single-node cluster or a 100-node cluster.

Figure 19.7 Pod affinity controls workload placement in relation to existing workloads

Anti-affinity uses the same syntax and you can use it to keep Pods away from nodes or other Pods. Anti-affinity is useful in components which run at scale for high-availability - think back to the Postgres database in chapter 8. The app used a StatefulSet with multiple replicas, but the Pods themselves might all end up on the same node. That defeats the whole purpose because if the node goes down then it takes all the database replicas with it. Anti-affinity can be used to express the rule: keep me away from other Pods like me, which will spread Pods across different nodes. We won't go back to the StatefulSet; we'll keep it simple and deploy that rule for the random numbers API and see what happens when we scale up.

TRY IT NOW Update the API Deployment to use Pod anti-affinity so replicas all run on different nodes. Then scale up and confirm the status.

```
# add an anti-affinity rule to the API Pod:
kubectl apply -f numbers/update/api.yaml

# print the API Pod status:
kubectl get pods -l app=numbers

#scale up the API and the web components:
kubectl scale deploy/numbers-api --replicas 3
kubectl scale deploy/numbers-web --replicas 3

# print all the web and API Pod status:
```

```
kubectl get pods -l app=numbers
```

You'll want to go through figure 19.8 carefully, because the results are probably not what you expect. The updated API Deployment creates a new ReplicaSet which creates a new Pod but that Pod stays as pending, because the anti-affinity rules won't let it run on the same node as the existing API Pod - the one it's trying to replace. When the API Deployment gets scaled up another replica does run, so we have two API Pods on the same node - but this is the previous ReplicaSet, which doesn't include the anti-affinity rule. The Deployment is trying to honor the request for three replicas between the two ReplicaSets, and as the new Pods don't come online it scales another replica of the old Pod.

Adds an anti-affinity rule to prevent multiple API Pods on the same node.

The update fails - the new Pod can't run because I only have one node and it's already running the old API Pod.

```
PS>kubectl apply -f numbers/update/api.yaml
deployment.apps/numbers-api configured
```

PS>

```
PS>kubectl get pods -l app=numbers
NAME           READY   STATUS    RESTARTS   AGE
numbers-api-5f5bf479bb-2wb58  0/1     Pending   0          10s
numbers-api-7f759884bd-qbcqw  1/1     Running   0          23m
numbers-web-657ccf9b66-7bms9  1/1     Running   0          23m
PS>
PS>kubectl scale deploy/numbers-api --replicas 3
deployment.apps/numbers-api scaled
PS>
PS>kubectl scale deploy/numbers-web --replicas 3
deployment.apps/numbers-web scaled
PS>
PS>kubectl get pods -l app=numbers
NAME           READY   STATUS    RESTARTS   AGE
numbers-api-5f5bf479bb-2wb58  0/1     Pending   0          45s
numbers-api-5f5bf479bb-d2t21  0/1     Pending   0          24s
numbers-api-7f759884bd-prhmk  1/1     Running   0          23s
numbers-api-7f759884bd-qbcqw  1/1     Running   0          23m
numbers-web-657ccf9b66-7bms9  1/1     Running   0          23m
numbers-web-657ccf9b66-7k752  1/1     Running   0          15s
numbers-web-657ccf9b66-ngcj8  1/1     Running   0          15s
```

The API Deployment is managing two ReplicaSets, the latest Pods never run because of the anti-affinity rule, so it scales up the old ReplicaSet.

The web Deployment scales up to three - this node is running an API Pod so the affinity rule matches for all the new web Pods.

Figure 19.8 Node anti-affinity gives unexpected results on a single-node cluster

How about the web Pods - did you expect to see three of them running? Well, three are running whether you expected it or not. Affinity and anti-affinity rules only check for the existence of

Pods by their labels, not the count of Pods. The affinity rule for the web Pod says it needs to run where there's an API Pod, not where there's a single API Pod. If you wanted to have only one web Pod with only one API Pod you'd need an anti-affinity rule for other web Pods in the web Pod spec.

Scheduling preferences get complex because the scheduler considers so many factors in the decision. Your simple affinity rules might not work as you expect, and you'll need to investigate taints, node labels, Pod labels, resource limits and quotas - or even the scheduler log file on the control plane node. Remember that required rules will prevent your Pod running if the scheduler can't find a node, so think about having a backup preferred rule. This was one of the more intricate topics of the chapter; next we're going to look at how we can get Kubernetes to automatically schedule more replicas for us, and that's actually more straightforward than trying to control workload placement.

19.3 Controlling capacity with automatic scaling

Kubernetes can automatically scale your applications by adding or removing Pods. This is *horizontal* scaling because it makes use of your existing nodes; there is also cluster scaling which adds and removes nodes, but you mostly find that in cloud platforms. We'll stick with horizontal Pod autoscaling here, which has a slightly disjointed user experience like the NetworkPolicy objects we covered in chapter 16. You can deploy an autoscale spec that describes how you would like your Pods scaled, but Kubernetes won't do anything with it unless it can check the load of the existing Pods. The wider Kubernetes project provides the metrics-server component for basic load checks, but not all distributions include it.

TRY IT NOW Confirm if your cluster has the metrics-server component installed, and if not then deploy it. The metrics power autoscaling and the kubectl top command.

```
# top shows resource usage if you have metrics-server installed:
kubectl top nodes

# if you get an error about "heapster" then
# you need to install metrics-server:
kubectl apply -f metrics-server/

# wait for it to spin up:
kubectl wait --for=condition=ContainersReady pod -l k8s-app=metrics-server -n kube-system

# it takes a minute or so for collection to start:
sleep 60

# print the metric-server Pod logs:
kubectl logs -n kube-system -l k8s-app=metrics-server --tail 2

# and look at node usage again:
kubectl top nodes
```

Figure 19.9 shows that my lab environment doesn't have metrics-server deployed (it's not part of Docker Desktop or Kind but it is installed in K3s), but thankfully the remedy is much simpler than choosing a Pod network. The metrics-server deployment is a single implementation - if you get a response from the Kubectl top command, then your cluster is collecting all the metrics it needs for autoscaling; if not then just deploy metrics-server.

Metrics-server is an additional component. If you don't have it installed then you won't get results from the top command.

```

PS>kubectl top nodes
Error from server (NotFound): the server could not find the requested resource (get services http:heapster:)
PS>
PS>kubectl apply -f metrics-server/
clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader created
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-dele
gator created
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created
serviceaccount/metrics-server created
deployment.apps/metrics-server created
service/metrics-server created
clusterrole.rbac.authorization.k8s.io/system:metrics-server created
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l k8s-app=metrics-serv
e r -n kube-system
pod/metrics-server-9cf886767-jwnhh condition met
PS>
PS>sleep 60
PS>
PS>kubectl logs -n kube-system -l k8s-app=metrics-server --tail 2
I0716 15:00:30.202460      1 manager.go:120] Querying source: kubelet_summa
ry:docker-desktop
I0716 15:00:30.225390      1 manager.go:148] ScrapeMetrics: time: 29.3974ms
, nodes: 1, pods: 17
PS>
PS>kubectl top nodes
NAME          CPU(cores)   CPU%    MEMORY(bytes)  MEMORY%
docker-desktop  256M        3%     1650Mi        6%
```

When it's running, metrics-server collects basic CPU and memory stats from all nodes and Pods. And that powers the top command.

Figure 19.9 Metrics-server collects CPU and memory stats but it's an optional component

Don't confuse these with the metrics we set up with Prometheus in chapter 14. The stats collected by metrics-server just track the basic compute resources, CPU and memory, and it only returns the current value when it's queried. It's an easy option to use for autoscaling if your workloads are CPU or memory intensive because Kubernetes knows how to use it without

any extra configuration. Listing 19.6 shows a Pod autoscale spec which uses CPU as the metric to scale on.

Listing 19.6 - hpa-cpu.yaml, horizontal pod autoscaling by CPU load

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler          # I love this name
metadata:
  name: pi-cpu
spec:
  scaleTargetRef:                      # the target is the controller
    apiVersion: apps/v1                 # to scale - this targets the
    kind: Deployment                   # Pi app Deployment
    name: pi-web
  minReplicas: 1                       # range of the replica count
  maxReplicas: 5
  targetCPUUtilizationPercentage: 75   # target CPU usage
```

Autoscale parameters are defined in a separate object, the HorizontalPodAutoscaler (HPA) which operates on a scale target - a Pod controller like a Deployment or StatefulSet. It specifies the range of the replica count and the desired CPU utilization. The autoscaler works by monitoring the average CPU across all current Pods as a percentage of the requested CPU amount in the Pod spec. If average utilization is below the target then the number of replicas is reduced, down to the minimum. If utilization is above the target then new replicas are added, up to the maximum. The Pi web app we've used in this book is compute intensive, so it will show us how autoscaling works.

TRY IT NOW Deploy the Pi web app with an HPA and check the status of the Pods.

```
# deploy the manifests:
kubectl apply -f pi/

# wait for the Pod to start:
kubectl wait --for=condition=ContainersReady pod -l app=pi-web

# print the status of the autoscaler:
kubectl get hpa pi-cpu
```

Now the Pi application is running and the spec requests 125 millicores of CPU (one-eighth of one core). Initially there's a single replica, which is set in the Deployment spec, and now the HPA is watching to see if it needs to create more Pods. The HPA gets its data from metrics-server which takes a minute or two to catch up - you can see in figure 19.10 that the current CPU utilization is unknown, but that will soon change to 0% as the Pod isn't doing any work.

The HPA is configured to monitor average CPU across all the Pi pods - it will scale up if the average is greater than 75% of the requested CPU.

```
PS>kubectl apply -f pi/
deployment.apps/pi-web created
horizontalpodautoscaler.autoscaling/pi-cpu created
service/pi-web created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=pi-web
pod/pi-web-6dc54b7c66-t5s8k condition met
PS>
PS>kubectl get hpa pi-cpu
NAME      REFERENCE          TARGETS      MINPODS   MAXPODS   REPLICAS
AGE
pi-cpu   Deployment/pi-web <unknown>/75%    1          5          1
24s
```

It takes a minute or so before the new Pod gets scraped by the metrics-server and the stats become available to the HPA.

Figure 19.10 The HPA works with the metrics-server to collect stats and the Deployment to manage scale

This Pod spec has a 250m CPU limit, which is double the requested amount. Make a request to calculate Pi to a high level of decimal places and you'll soon max out 0.25 of a core and the average utilization will spike towards 200%. Then the HPA will kick in and scale up, adding new replicas to help with the load.

TRY IT NOW Run a script which makes some concurrent calls to the Pi web app, asking for 100,000 decimal places and causing high CPU load. Confirm the HPA scales up.

```
# run the load script - on Windows:
.\loadpi.ps1

# OR on Linux/macOS:
chmod +x ./loadpi.sh && ./loadpi.sh

# give the metrics-server and HPA time to work:
sleep 60

# confirm the Deployment has been scaled up:
kubectl get hpa pi-cpu #replicas increase

# print the Pod compute usage:
kubectl top pods -l app=pi-web
```

Whether the extra Pods actually do help with the load depends on your application. In this case whichever Pod receives the web request will process the calculation until it completes, so new Pods don't share the existing load. You can see in figure 19.11 that the original Pod is running at just about the maximum 250m, and the new Pods are doing nothing at 1m. But those

additional Pods increase the capacity of the app and they can work on any new requests that come in.

Generate some load to the Pi app and the CPU will go over the requested amount in the Pod spec, triggering the autoscaler.

Autoscaling is reactive - it takes a while for the metrics to come through. 30 seconds would probably be enough.

```

PS>.\loadpi.ps1
PS>
PS>sleep 60
PS>
PS>kubectl get hpa pi-cpu
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
pi-cpu   Deployment/pi-web  164%/75%    1            5            3           14m
PS>
PS>kubectl top pods -l app=pi-web
NAME                           CPU(cores)      MEMORY(bytes)
pi-web-6dc54b7c66-t5s8k     206m          30Mi
PS>
PS>kubectl top pods -l app=pi-web
NAME                           CPU(cores)      MEMORY(bytes)
pi-web-6dc54b7c66-2lmxw     1m            21Mi
pi-web-6dc54b7c66-6hsnb     1m            21Mi
pi-web-6dc54b7c66-t5s8k     249m          35Mi

```

Average CPU is well above the target range, so the HPA adds two more Pods.

The new Pods aren't doing anything yet, but they're running and able to respond to new requests.

Figure 19.11 Autoscaling in action - the HPA triggers new Pods when CPU usage spikes

When you run that exercise you should see similar results - the HPA will scale up to three Pods, and before it goes any further the app returns the Pi response, CPU utilization falls and it doesn't need to scale up any more. The HPA adds more Pods every 15 seconds until the utilization is within target. With one Pod maxed out and two Pods doing nothing the average CPU falls 66% which is within the 75% target, so the HPA won't add any more Pods (you can repeat the load script a few more times to confirm it peaks to five Pods). When you stop making requests the load will fall to 0% again, and then the HPA waits to make sure the app stays within target for five minutes before it scales back down to one replica.

There are several parameters there: how long to wait before scaling up or down, how many Pods to scale up or down, how quickly to add or remove Pods. None of those values can be changed in the v1 HPA spec but they're all exposed in the v2 spec. The new spec is still a beta-2 feature in Kubernetes 1.18 and it's a pretty significant change. The single option to scale based on CPU has been replaced with a generic metrics section, and the scaling behavior can be controlled. Listing 19.7 shows the new spec in an update to the Pi HPA.

Listing 19.7 - hpa-cpu-v2.yaml, the extended HPA spec in version 2

```

metrics:                                # metrics spec inside a v2 HPA
- type: Resource
  resource:
    name: cpu                         # the resource to monitor is generic
    target:                            # this checks for CPU but you can use
      type: Utilization               # other metrics
      averageUtilization: 75
  behavior:
    scaleDown:                         # this sets the parameters
      stabilizationWindowSeconds: 30   # when the HPA is scaling
    policies:
      - type: Percent                 # down - it waits for 30s
        value: 50                     # and scales down by 50%
    periodSeconds: 15

```

I said autoscaling was more straightforward than affinity, but I really only meant v1. The v2 spec is complex because it supports other types of metrics, and you can make use of Prometheus metrics as the source for scaling decisions. You need a few more pieces for that so I won't go into the details, but remember it's an option - it means you can scale based on any metrics you collect, like the rate of incoming HTTP requests or the number of messages in a queue.

We're sticking with the 75% CPU target here which uses the same metrics-server stats, but we've tuned the scale-down behavior so we'll see the number of Pods come down much more quickly once the Pi requests are processed.

TRY IT NOW Update the HPA to use the v2 spec; this sets a low stabilization period for scale down events so you'll see the Pod count fall more quickly.

```

# update the HPA settings:
kubectl apply -f pi/update/hpa-cpu-v2.yaml

# run the load script again - on Windows:
.\loadpi.ps1

# OR on macOS/Linux:
./loadpi.sh

# wait for the HPA to scale up:
sleep 60

# confirm there are more replicas:
kubectl get hpa pi-cpu # go up

# there's no more load, wait for the HPA to scale down:
sleep 60

# confirm there's only one replica:
kubectl get hpa pi-cpu # go down

# and print the Deployment status:
kubectl get deploy pi-web

```

In that exercise you'll see the scale-up behavior works in the same way, because the v2 defaults are the same as the v1 defaults. The scale-down won't take so long this time, although you may see - as in figure 19.12 - that querying the HPA status doesn't reflect the change as quickly as the Deployment itself does.

Updating HPA settings - it's a separate object, so this is not a change to the Pod spec.
 Replicas scale up to three again under the same load.

```
PS>kubectl apply -f pi/update/hpa-cpu-v2.yaml
horizontalpodautoscaler.autoscaling/pi-cpu configured
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
pi-cpu	Deployment/pi-web	200%/75%	1	5	3	74m

```
PS>
PS>sleep 60
PS>
PS>
PS>kubectl get hpa pi-cpu
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
pi-cpu	Deployment/pi-web	0%/75%	1	5	3	76m

```
PS>
PS>kubectl get deploy pi-web
NAME      READY    UP-TO-DATE   AVAILABLE   AGE
pi-web   1/1      1           1           76m
```

Load has dropped and the HPA will scale down - it already has at this point, but the Kubectl output can lag by a few seconds.

The Deployment shows the correct scale, with no load the HPA has scaled down to the minimum level.

Figure 19.12 The new HPA spec favors a fast scale-down, suitable for bursty workloads

Changing the behavior lets you model how the HPA responds to scaling events. The defaults are a fairly conservative version of scale-up quickly and scale-down slowly, but you can switch it around so scaling up is more gradual and scaling down is immediate. "Immediate" isn't really true because there's a lag between the metrics being collected and made available to the HPA, but the delay will be just tens of seconds. The HPA is specific to one target, so you can have different scaling rules and rates for different parts of your app.

We've covered placing Pods and scaling up and down - and the HPA just instructs the controller to scale, so any scheduling requirements in the Pod spec apply to all Pods. The last topic on workload management is *preemption*, the process of deliberately causing Pods to fail.

19.4 Protecting resources with preemption and priorities

Sometimes Kubernetes realizes a node is working too hard and then it *preempts* that some Pods will fail and shuts them down in advance, giving the node time to recover. This is eviction and it only happens under extreme circumstances, where if the cluster didn't take action the node might become unresponsive. Evicted Pods stay on the node so you have the forensics to track down problems, but the Pod containers are stopped and removed - freeing up memory and disk. If the Pod is managed by a controller then a replacement Pod gets created, which will be scheduled to run on a different node.

Preemption is what happens if you get all your resource specs, quotas, scheduling and scaling wrong, so nodes end up running more Pods than they can manage and they get starved of memory or disk. If that happens then Kubernetes considers the node to be under pressure and it evicts Pods until the pressure situation ends. At the same time it adds a taint to the node so no new Pods get scheduled to run on it. As the pressure situation eases the taint gets removed and the node is able to accept new workloads.

There's no way to fake memory or disk pressure for demos or exercises, so to see how this works we're going to need to max out your lab. It's easier to do that with memory than with disk, but it still isn't easy: the default is to start eviction when the node has less than 100Mb of memory available, which means using up nearly all of your memory. If you want to follow along with the exercises in this section you really need to spin up a separate lab in a VM, so you can tweak the Kubernetes settings and max the memory on the VM rather than on your own machine.

TRY IT NOW Start a dedicated VM using Vagrant. This setup has Kind installed and 3GB RAM allocated. Then create a new Kind cluster with a custom Kubelet configuration that lowers the memory threshold for eviction.

```
# from the kiamol source root, create a VM with Vagrant:
cd ch01/vagrant/

# create the new VM:
vagrant up

# connect to the VM:
vagrant ssh

# switch to this chapter's source inside the VM:
cd /kiamol/ch19

# create a customized Kind cluster:
kind create cluster --name kiamol-ch19 --config ./kind/kiamol-ch19-config.yaml --image
kindest/node:v1.18.8

# wait for the node to be ready:
kubectl -n kube-system wait --for=condition=Ready node --all

# print the VM's memory stats:
./kind/print-memory.sh
```

The script in that exercise uses the same logic that the Kubelet uses to determine how much free memory it can access. You can see in figure 19.13 that my VM reports just under 3GB of total memory and just under 1.5GB free, and that's what Kubernetes sees.

This exercise uses a dedicated VM and creates a customized Kind cluster with a low memory threshold for eviction.

```
vagrant@kiamol:~$ cd /kiamol/ch18
vagrant@kiamol:/kiamol/ch18$ kind create cluster --name kiamol-ch18 --config
./kind/kiamol-ch18-config.yaml --image kindest/node:v1.15.7
Creating cluster "kiamol-ch18" ...
✓ Ensuring node image (kindest/node:v1.15.7) 
✓ Preparing nodes 
✓ Writing configuration 
✓ Starting control-plane 
✓ Installing CNI 
✓ Installing StorageClass 
Set kubectl context to "kind-kiamol-ch18"
vagrant@kiamol:/kiamol/ch18$ kubectl -n kube-system wait --for=condition=Rea
dy node --all
node/kiamol-ch18-control-plane condition met
vagrant@kiamol:/kiamol/ch18$ ./kind/print-memory.sh
-----
Memory capacity : 2892M
Memory available: 1435M
```

This script uses the same logic as the Kubelet to determine total and available memory. Running the OS, Docker and Kubernetes uses half the RAM so 1.5GB is free for apps.

Figure 19.13 If you want to test memory pressure, it's safer to do it in a dedicated environment

If you don't want to spin up a separate VM for these exercises that's fine - but remember the default memory threshold is 100MB. To force a memory pressure situation you're going to need to allocate almost all the memory on your machine, which will probably also cause a CPU spike and make the whole thing unresponsive. You can confirm the memory limit by checking the live configuration of the Kubelet - it has an HTTP endpoint you can use by proxying requests with Kubectl.

TRY IT NOW Query the configuration API on the node to see the active settings for the Kubelet and confirm that the eviction level has been set.

```
# run a proxy to the Kubernetes API server:
kubectl proxy
```

```
# in a new terminal connect to the VM again:
cd ch01/vagrant/
vagrant ssh

# make a GET request to see the Kubelet config:
curl -sSL "http://localhost:8001/api/v1/nodes/$(kubectl get node -o
jsonpath='{.items[0].metadata.name}')/proxy/configz"
```

In this environment the Kubelet is configured to trigger evictions when there is only 40% of memory available on the node, as you see in figure 19.14. That's a deliberately low threshold so we can easily trigger eviction; in a production environment you would have it much higher. If you use a different lab environment and there is no explicit setting in the Kubelet then you'll be using the default 100MB.

Kubectl proxy is like a port-forward to get access to the Kubernetes API server.

```
vagrant@kiamol:/kiamol/ch18$ kubectl proxy
Starting to serve on 127.0.0.1:8001

vagrant@kiamol:~$ curl -sSL "http://localhost:8001/api/v1/nodes/kiamol-ch18-control-plane/proxy/configz"
{
  "kubeletconfig": {
    "staticPodPath": "/etc/kubernetes/manifests",
    "syncFrequency": "1m0s",
    "fileCheckFrequency": "20s",
    "httpCheckFrequency": "1m0s"
  },
  "evictionHard": {
    "memory.available": "40%"
  },
  "evictionPressureTransitionPeriod": "30s",
  "enableControllerAttachDetach": true,
  "makeIPTablesUtlChains": true,
  "iptablesMasqueradeBit": 14,
  "iptablesDropBit": 15,
  "failSwapOn": false,
  "containerLogMaxSize": "10Mi",
  "containerLogMaxFiles": 5,
  "configMapAndSecretChangeDetectionStrategy": "Watch",
  "enforceNodeAllocatable": ["pods"]
}
```

You can use it to query the configz endpoint on a node to show the Kubelet configuration. There are a lot of config settings, the custom setting is to start eviction when the node has less than 40% memory available.

Figure 19.14 Low-level settings are specified in the Kubelet configuration, which you can view using the proxy

That's enough warnings (well - one more - I trashed my Docker Desktop environment trying to force memory pressure when I was planning these exercises. Kubectl wouldn't respond and I had to remove and reinstall everything). Let's go ahead and trigger preemption. I've built a container image which packages the Linux `stress` tool and I have a Deployment spec for four replicas which each allocate 300MB of memory. That should leave the node with less than 40% of total memory available, and push it into memory pressure.

TRY IT NOW Run an app which allocates a lot of memory and see how Kubernetes evicts Pods when the nodes are under memory pressure.

```
# the stress Pods will allocate 1.2GB of RAM:  
kubectl apply -f stress/stress.yaml  
  
# wait for them all to start:  
kubectl wait --for=condition=ContainersReady pod -l app=stress  
  
# print the node's memory stats:  
. ./kind/print-memory.sh  
  
# and list the Pods:  
kubectl get pods -l app=stress  
  
# remove the Deployment:  
kubectl apply -f stress/stress.yaml
```

Preemption happens fast and the transition between pressure states also happens fast. I've trimmed the output in figure 19.15 because I was too slow to run the Pod list command and by the time I did, the node had evicted 23 Pods.

These four Pods have allocated 1.2GB of memory between them.

The node has 26% of it's memory available, below the 40% custom eviction threshold.

```
vagrant@kiamol:/kiamol/ch18$ kubectl apply -f stress/stress.yaml
deployment.apps/stress created
vagrant@kiamol:/kiamol/ch18$
vagrant@kiamol:/kiamol/ch18$ kubectl wait --for=condition=ContainersReady pod -l app=stress
pod/stress-5c6fc498f-bj75f condition met
pod/stress-5c6fc498f-j8xxp condition met
pod/stress-5c6fc498f-lc7cl condition met
pod/stress-5c6fc498f-q5kgh condition met
vagrant@kiamol:/kiamol/ch18$ ./kind/print-memory.sh
Memory capacity : 2892M
Memory available: 756M
```

NAME	READY	STATUS	RESTARTS	AGE
stress-5c6fc498f-2sw4d	0/1	Evicted	0	4s
stress-5c6fc498f-6pvqc	0/1	Evicted	0	4s
stress-5c6fc498f-9djbz	0/1	Evicted	0	8s
stress-5c6fc498f-bj75f	1/1	Running	0	19s
stress-5c6fc498f-jcx8c	0/1	Pending	0	3s
stress-5c6fc498f-lc7cl	1/1	Running	0	19s
stress-5c6fc498f-q5kgh	1/1	Running	0	19s
stress-5c6fc498f-rh7g2	0/1	Evicted	0	11s
stress-5c6fc498f-wm7mv	0/1	Evicted	0	5s
stress-5c6fc498f-z2vtp	0/1	Evicted	0	9s

```
Vagrant@kiamol:/kiamol/ch18$ kubectl delete -f stress/stress.yaml
deployment.apps "stress" deleted
```

Three of the original Pods are still running, and one was evicted. Then the node had enough memory to run another Pod, which immediately tripped it into memory pressure and got evicted again. And repeat.

Figure 19.15 Lots of Pods. Resource-hungry Pods can cause an eviction/creation loop.

Why are there so many evictions? Why didn't Kubernetes just evict one Pod and leave the replacement in the pending state while the node was under memory pressure? It did. But as soon as the Pod got evicted, it freed up a bunch of memory and the node quickly tripped out of memory pressure. Meanwhile the Deployment created a new Pod to replace the evicted one, which ran on the no-longer-tainted node - but it immediately allocated a bunch more memory and tripped the pressure switch again for another eviction. That loop happens quickly with this app because it allocates lots of memory when it starts, but it's possible to get in the same situation with a real app if your cluster is overloaded.

Preemption events should be rare in a production environment, but if it does happen you want to make sure your least important workloads are evicted. The Kubelet decides which Pod

to evict in a memory pressure situation by ranking them. That ranking looks at the amount of memory the Pod is using relative to the amount requested in the Pod spec, and also the *priority class* of the Pod. Priority classes are the last new concept for this chapter, and they're a simple way to classify the importance of your workloads. Listing 19.8 shows a custom priority class with a low value.

Listing 19.8 - low.yaml, a class for low priority workloads

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: kiamol-low    # the Pod spec uses this name to state its priority
  value: 100          # priority is an integer up to 1 billion
  globalDefault: true
  description: "Low priority - OK to be evicted"
```

The numeric value is what decides the priority - bigger means more important. There aren't any default priority classes in Kubernetes, so you need to map your own if you want to safeguard the more important workloads. You attach a priority to a workload by adding the `PriorityClassName` field to the Pod spec. In the final exercise we'll deploy two versions of the stress app — one with high priority and one with low priority. When the memory pressure hits, we'll see the low priority Pods get evicted.

TRY IT NOW Run the stress exercise again, but this time with two Deployments each running two Pods. The memory allocation is the same but the Pods have different priorities.

```
# create the custom priority classes:
kubectl apply -f priority-classes/

# deploy apps with high and low priorities:
kubectl apply -f stress/with-priorities/

# wait for the Pods to run:
kubectl wait --for=condition=ContainersReady pod -l app=stress
wait

# print the memory stats:
./kind/print-memory.sh

# confirm that the node is under memory pressure:
kubectl describe node | grep MemoryPressure

# print the Pod list:
kubectl get pods -l app=stress
```

That all worked beautifully when I ran the exercise and you can see the output in figure 19.16. The node can only manage three Pods without tripping into memory pressure, and the evicted fourth Pod is always from the low priority spec; the high priority Pods keep running.

High and low priority classes can now be used in Pod specs.

Creates four Pods consuming 1.2GB of memory, but two are low priority and two are high priority.

```
vagrant@kiamol:/kiamol/ch18$ kubectl apply -f priority-classes/priorityclass.scheduling.k8s.io/kiamol-high created
priorityclass.scheduling.k8s.io/kiamol-low created
vagrant@kiamol:/kiamol/ch18$
vagrant@kiamol:/kiamol/ch18$ kubectl apply -f stress/with-priorities/deployment.apps/stress-high created
deployment.apps/stress-low created
vagrant@kiamol:/kiamol/ch18$
vagrant@kiamol:/kiamol/ch18$ kubectl wait --for=condition=ContainersReady pod -l app=stress
```

```
vagrant@kiamol:/kiamol/ch18$ kubectl describe node | grep MemoryPressure
  MemoryPressure  True    Thu, 16 Jul 2020 18:41:18 +0000  Thu, 16 Jul 2020 18:41:18 +0000  KubeletHasInsufficientMemory  kubelet has insufficient memory available
vagrant@kiamol:/kiamol/ch18$
```

NAME	READY	STATUS	RESTARTS	AGE
stress-high-78966f859d-6tp9j	1/1	Running	0	61s
stress-high-78966f859d-kjqtf	1/1	Running	0	61s
stress-low-56bf8d7498-5fkp7	0/1	Evicted	0	61s
stress-low-56bf8d7498-jgwxz	0/1	Evicted	0	61s
stress-low-56bf8d7498-rckhm	0/1	Evicted	0	61s
stress-low-56bf8d7498-v886t	1/1	Running	0	61s

You can see in the node details if it's under memory pressure - if it is then there's a taint added to stop it running more Pods.

There's still an eviction/creation loop, but always the low priority Pods get evicted.

Figure 19.16 Adding a priority class is a simple safeguard to protect key Pods from eviction

It's not just the priority class that keeps the high priority Pods running. All the Pods use more memory than they request, so they're all eligible to be evicted - and then the priority gets taken into account. If all eligible Pods have the same priority (or no priority) then the choice is made based on how much more memory is being used than requested, so the highest offenders get evicted. It's important to include resource requests in your Pod specs as well as limits, but a priority class is a useful protection for the more important workloads.

That was a wide-ranging tour around the major aspects of workload management. They're all features you'll use in production and we'll finish the chapter reviewing what they give you and how they work together.

19.5 Understanding the controls for managing workloads

Scheduling, autoscaling and eviction are all advanced topics and there are even more nuances than we've covered here. You'll certainly use them in your Kubernetes journey, and it's worth

bringing some controls in early on. They address different problems in managing your apps, but they all impact each other so you need to use them carefully.

Affinity is the feature you'll use most in large clusters - node affinity lets you segregate workloads with stricter isolation than you get with namespaces alone, and pod affinity lets you model the availability requirements of your apps. You can combine pod affinity with node topology to ensure replicas run across different failure domains, so the loss of one rack or zone doesn't bring your app down because other Pods are running in a different zone. Remember that required affinity is a hard rule for the scheduler - if you require Pods to be in different zones and you only have three zones, replica number four will forever stay pending.

Autoscaling is a great feature and easy to use if your app is CPU-bound. Then you can use the default metrics-server and the simple v1 HPA, making sure you have CPU requests in your Pod specs. Things get more complex if you want to scale based on higher-level metrics, but that's definitely worth investigating. Having your app scale automatically when key service levels are being missed is a major benefit of Kubernetes and something to work towards when you're established in production. Scaling just increases or decreases the number of replicas, so if you have affinity rules in your spec you need to make sure they can be met at the maximum scale level.

Preemption is Kubernetes' safety mechanism for dealing with nodes that run short on memory or disk - CPU is different because Pods can be throttled to reclaim CPU without stopping containers. Memory or disk pressure gets relieved by evicting Pods which is something you should rarely see if your cluster and apps are right-sized. You should include resource requests in your Pod specs so the worst offenders can be evicted, and consider priority classes if you really have some workloads which are more important than others. If you do get in a preemption situation you need to investigate quickly to make sure the node doesn't keep flipping in and out of pressure, constantly adding and then evicting Pods (we added node pressure indicators to the cluster dashboard in chapter 14 for this reason).

That's all for workload management, time to clear down the cluster(s) to make way for the lab.

TRY IT NOW Clean up your main lab environment; if you created a custom environment it can be removed.

```
# remove objects on your main cluster:  
kubectl delete all,priorityclass,hpa -l kiamol=ch19  
  
# if you deployed the metrics-server in section 19.3, remove it:  
kubectl delete -f metrics-server/  
  
# if you created a new VM in section 19.4, it can go too:  
cd ch01/vagrant/  
vagrant destroy
```

19.6 Lab

We're going into production with the Pi app! Your job for this lab is to add to the spec so we can control the workload. This is the setup we need:

- the app has to run in the EU region because of data sovereignty concerns
- it should autoscale based on target CPU utilization of 50%
- there must be between two and five replicas running
- the load should preferably be spread around multiple nodes in the EU region

You'll find examples in this chapter's exercises for all these, except one - and you might need to check the API docs to see how to build up the last rule. Remember that node topology is just done with labels and you can add any labels you want to your nodes. My solution is in the usual place: <https://github.com/sixeyed/kiamol/blob/master/ch19/lab/README.md>

20

Extending Kubernetes with Custom Resources and Operators

At the heart of Kubernetes is a highly-available database and a REST API with a consistent way of working with objects. When you create a Pod through the API, the definition is stored in the database, and a controller gets notified and knows that it needs to allocate the Pod to a node to get it running. It's a generic pattern where different controllers work on different types of object, and it's extensible so you can add your own resource definitions and your own custom controllers to act on those resources. That may sound like an obscure topic, but it's very common for products to extend Kubernetes to make the product itself easier to use. It's also a straightforward way to customize Kubernetes to make it work better in your organization.

Custom resources and controllers can hide a lot of the complexity in an application, and in this chapter you'll see how to define and work with them. The definition part is simple but controllers need custom code - we won't focus on the code, but we'll have some examples of customizations so you can see what they can do. We'll also cover the Operator pattern in this chapter, which is a way of using custom resources and controllers to automate the deployment and the ongoing operational tasks of an application.

20.1 How to extend Kubernetes with custom resources

Kubectl commands map closely to the Kubernetes REST API - when you run `kubectl get` it makes a request to the API to get a resource or a list of resources. A standard set of actions are available for all resources and you know from the RBAC rules in chapter 17 that they're defined as verbs: `create`, `get`, `list`, `watch` and `delete`. You can define a custom resource in Kubernetes and it gets automatic support for all those actions in the API; Kubernetes clients understand custom resources too, so you can work with them using Kubectl just like any other object. Figure 20.1 shows how the cluster supports custom resources.

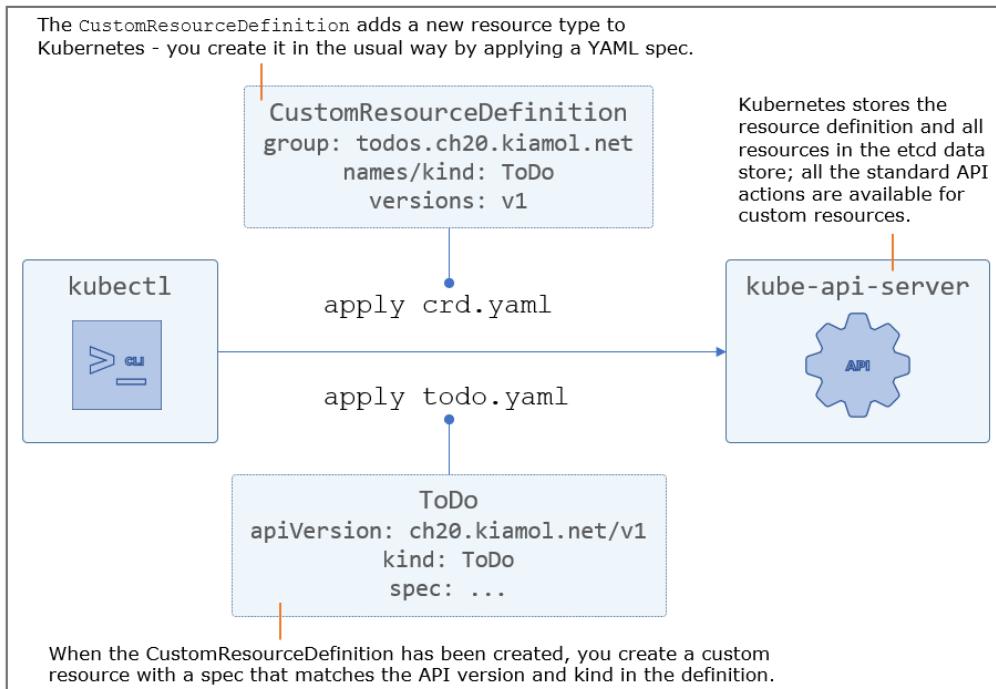


Figure 20.1 Kubernetes is extensible with custom resources, which work just like standard resources

You define standard Kubernetes objects in YAML specifying the kind of resource and all the fields in the spec - Pod specs have a container list, and container specs have an image name and a set of ports. Those fields are all stored in a schema in Kubernetes so it knows the structure of the resource and it can validate new objects. This is where the API version field comes in, so v1 of the `HorizontalPodAutoscaler` resource has a different structure from v2beta2. Custom resources have a known structure too and you create your own schema in a `CustomResourceDefinition` (CRD) object. Listing 20.1 shows a simple CRD for recording to-do items as Kubernetes objects.

Listing 20.1 - todo-crd.yaml, a CRD for storing to-do items in Kubernetes

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: todos.ch20.kiamol.net          # the name of the CRD needs to match
  # the names in the resource spec
spec:
  group: ch20.kiamol.net               # the group classifies a set of CRDs
  scope: Namespaced                     # can be cluster-wide or namespaced
  names:
    plural: todos                      # the names are how you refer to
    singular: todo                      # custom resources in YAML and Kubectl
```

```

kind: ToDo
versions:          # there can be multiple versions
- name: v1          # each version has a schema
  served: true      # make resources available in the API
  storage: true      # save resources in etcd
  schema:
    openAPIV3Schema:
      type: object
      properties:
        spec:           # the schema sets the structure of the
          type: object # custom resource - ToDo objects have a
          properties:   # spec field which has an item field
            item:
              type: string

```

The CRD structure itself is pretty verbose and the schema section is particularly awkward to read, but it uses the standard JSONSchema project to define the structure of the resource, and your definitions can be as complex or as simple as you need. The CRD in listing 20.1 is hard on the eyes but it makes for a very simple custom resource - listing 20.2 shows a ToDo item which uses this structure.

Listing 20.2 - todo-ch20.yaml, a ToDo custom resource

```

apiVersion: "ch20.kiamol.net/v1"    # group name and version of the CRD
kind: ToDo                         # the resource type
metadata:                          # standard metadata
  name: ch20
spec:
  item: "Finish KIAMOL Ch20"       # spec needs to match the CRD schema

```

Now that just looks like normal Kubernetes YAML, which is the whole point of CRDs - to store your own resources in Kubernetes and make them feel like standard objects. The API version identifies that this is a custom resource from this chapter of the book, and it's v1. The metadata is standard metadata so it can include labels and annotations, and the spec is the custom structure defined in the CRD. That's enough YAML now, so let's put this into practice and use Kubernetes as our to-do list app.

TRY IT NOW Deploy a custom resource definition and some resources to see how you work with them using Kubectl.

```

# switch to this chapter's source:
cd ch20

# deploy the CRD:
kubectl apply -f todo-custom/

# print CRD details:
kubectl get crd -l kiamol=ch20

# create some custom resources:
kubectl apply -f todo-custom/items/

```

```
# list all the resources:
kubectl get todos
```

You can see in figure 20.2 that you work with custom resources just like any other resource. Once the CRD is deployed the API now supports the ToDo object, and you can create items by applying YAML. Kubectl is the tool for managing custom resources, and now the get, describe and delete commands work in the same way for ToDo objects as for Pods.

Deploying the CRD creates the object structure, there are no custom resources yet.

```
PS>cd ch20
PS>
PS>kubectl apply -f todo-custom/
customresourcedefinition.apiextensions.k8s.io/todos.ch20.kiamol.net created
PS>
PS>kubectl get crd -l kiamol=ch20
NAME          CREATED AT
todos.ch20.kiamol.net  2020-07-25T14:10:08Z
PS>
PS>kubectl apply -f todo-custom/items/
todo.ch20.kiamol.net/ch20 created
todo.ch20.kiamol.net/ch21 created
PS>
PS>kubectl get todos
NAME   AGE
ch20   5s
ch21   5s
```

These are custom resources, defined in YAML specs with the API version and kind identifying the CRD.

You interact with custom resources in the same way as standard resources, using Kubectl commands to get, describe and delete them.

Figure 20.2 CRDs and custom resources are described in YAML and deployed with Kubectl

The CRD specification is very rich and lets you build lots of logic around your resources - including validation rules, subresources and multiple versions. We won't get into that level of detail, but you can be confident that custom resources are a mature feature in Kubernetes with all the functionality you need to manage an evolving set of object definitions. The purpose of CRDs is to provide a simplified user experience, and we can make a small change to the ToDo CRD to make it more usable.

TRY IT NOW An update to the CRD spec makes the Kubectl output more useful - this update doesn't affect the custom resources; it just adds to the columns which get printed.

```
# update the CRD, adding output columns:
kubectl apply -f todo-custom/update/

# list the to-do resources again:
kubectl get todos
```

```
# delete one of the resources:  
kubectl delete todo ch21  
  
# and show the detail of the other resource:  
kubectl describe todo ch20
```

That exercise updated the CRD with additional printer columns, so the API returns extra information in the get request. You can see in figure 20.3 that this is now a fully-featured to-do list application, which even lets you delete items so it's better than the to-do web app we've been running in this book.

CRDs can include custom output columns, so Kubectl shows more useful output when you list the resources - this update displays the key information for each item.

```
PS>kubectl apply -f todo-custom/update/  
customresourcedefinition.apiextensions.k8s.io/todos.ch20.kiamol.net configur  
ed  
PS>  
PS>kubectl get todos  
NAME ITEM DUE  
ch20 Finish KIAMOL Ch20 2020-07-26  
ch21 Plan KIAMOL Ch21 2020-07-27  
PS>  
PS>kubectl delete todo ch21  
todo.ch20.kiamol.net "ch21" deleted  
PS>  
PS>kubectl describe todo ch20  
Name: ch20  
Namespace: default  
Labels: kiamol=ch20  
Annotations: API Version: ch20.kiamol.net/v1  
Spec:  
  Due Date: 2020-07-26  
  Item: Finish KIAMOL Ch20
```

You can delete items with this approach to the to-do app.

Describing a custom resource shows the usual wealth of detail, including the API version of the CRD and the field values of the resource.

Figure 20.3 A fully-functional to-do app powered by Kubernetes with no custom code!

This is fine for a simple demo, and we could get distracted writing a custom controller that watches these resources, adds items to your Google calendar and sends email reminders when they're due - but we're not going to do that. It's certainly possible but it's not a very good use of custom resources because the objects we're storing and the actions they trigger have nothing to do with Kubernetes; we're not integrating with other objects or extending what the cluster can do, we're just using Kubernetes as a massively over-specced content management system. We can do better than that, and we'll start by clearing up the ToDo resources.

TRY IT NOW Remove the to-do CRD and confirm the custom resources are deleted too.

```
# list the CRDs registered in your cluster:  
kubectl get crds  
  
# delete the to-do item CRD:  
kubectl delete crd todos.ch20.kiamol.net  
  
# try and list the to-do items:  
kubectl get todos
```

You can see in that exercise and in figure 20.4 that custom resources can't exist without a CRD - Kubernetes won't store any unknown objects. Deleting the CRD deletes all its resources, so if you make use of custom resources you need to make sure your RBAC permissions around the CRD itself are pretty tight.

There's only one CRD in this cluster - some platforms might install their own custom resources by default.

Deleting the CRD also removes all the custom resources.

```
PS>kubectl get crds
NAME          CREATED AT
todos.ch20.kiamol.net  2020-07-25T14:10:08Z
PS>
PS>kubectl delete crd todos.ch20.kiamol.net
customresourcedefinition.apiextensions.k8s.io "todos.ch20.kiamol.net" deleted
PS>
PS>kubectl get todos
Error from server (NotFound): Unable to list "ch20.kiamol.net/v1, Resource=todos": the server could not find the requested resource (get todos.ch20.kiamol.net)
PS>
PS>kubectl get todos
error: the server doesn't have a resource type "todos"
```

It takes a few seconds for the CRD to be fully removed.
Now the response is correct, there is no to-do resource in the cluster.

Figure 20.4 Custom resources are removed when the CRD that defines them is removed

We'll go on to add a different CRD and pair it with a custom controller to add a user authentication system to Kubernetes.

20.2 Triggering workflows with custom controllers

You know from chapter 17 that production Kubernetes clusters are usually integrated with an external identity provider to authenticate users. Smaller organizations often use Service Accounts as end-user accounts which means you don't need an external system, but means you

need to manage namespaces for groups and deal with creating accounts and tokens. It's a situation where Kubernetes has all the pieces for you, but it needs a fair amount of work to put them together - and that's exactly where you should think about custom resources and custom controllers.

The custom resource here is a *user*, and the main workflows are adding and deleting users. A simple user CRD just needs to store a name and a group, maybe with contact details too. You can add and remove users with Kubectl and when that happens the workflows will get processed by a custom controller. The controller is just an application which runs in a Pod and connects to the Kubernetes API - it watches for changes to user objects, and then it creates or deletes the necessary resources: namespaces, service accounts and tokens. Figure 20.5 shows the add workflow, and the delete workflow is effectively the reverse.

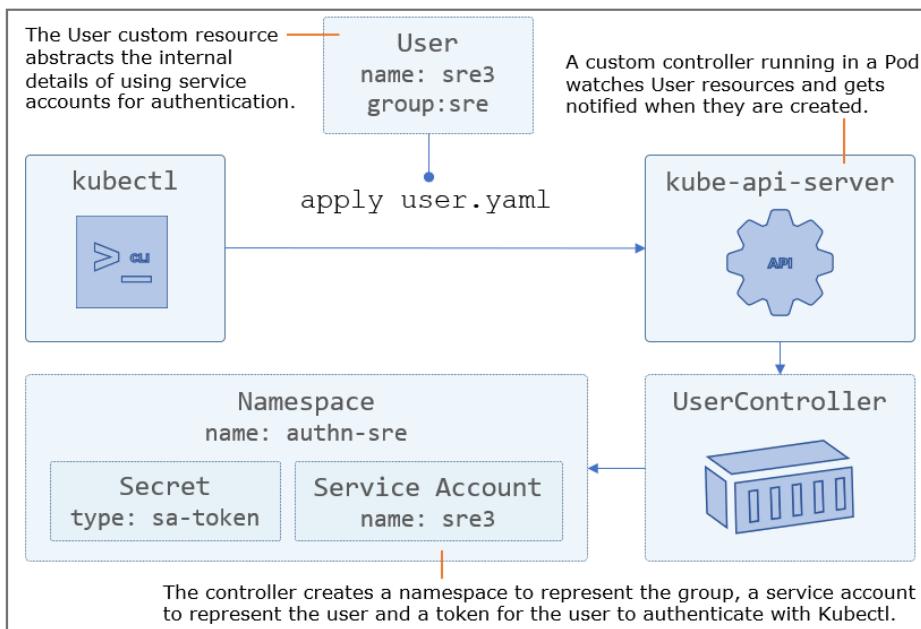


Figure 20.5 Adding a user in the custom authentication system creates all the Kubernetes resources

We'll start by deploying the user CRD and some users. The CRD is difficult to read because of the schema, but there's nothing new in it so we'll skip the listing (it's the file `user-crd.yaml` if you want to walk through it). The user resources themselves are simple. Listing 20.3 shows a user in the SRE team.

Listing 20.3 - user-crd.yaml, spec for a user resource

```
apiVersion: "ch20.kiamol.net/v1"      # this is the same group and version
kind: User                            # as the other CRDs in this chapter
```

```

metadata:
  name: sre3
spec:                      # the spec records the user details
  email: sre3@kiamol.net
  group: sre

```

One thing you need to be aware of with CRDs is that it takes a few seconds for them to register in the API server, so you can't usually deploy a folder of CRDs and custom resources in one go because the CRD often isn't ready in time. So you need to deploy the CRD first and then you can deploy the resources - we'll do that now with the SRE user and a test user.

TRY IT NOW Create the CRD for users and some user resources.

```

# deploy the CRD first:
kubectl apply -f users/crd/

# then the users:
kubectl apply -f users/

# print the user list:
kubectl get users

```

My output in figure 20.6 shows the user experience works well. The CRD only needs to be deployed once, and then you can add as many users as you like using simple YAML like in listing 20.3. Now there are custom user objects stored in Kubernetes, but there's no controller running so nothing will happen yet.

CRDs should be applied before any custom resources, the definition needs to exist before you can deploy any resources that use the definition.

```

PS>kubectl apply -f users/crd/
customresourcedefinition.apiextensions.k8s.io/users.ch20.kiamol.net created
PS>
PS>kubectl apply -f users/
user.ch20.kiamol.net/sre3 created
user.ch20.kiamol.net/tester3 created
PS>
PS>kubectl get users
NAME      EMAIL           GROUP
sre3     sre3@kiamol.net   sre
tester3  tester3@kiamol.net test

```

My authentication system is quite basic - it records contact details, and users must belong to a single group.

Figure 20.6 Creating the CRD and some users - this doesn't trigger anything without a controller

Custom controllers are usually written in Go and there are a few existing packages which take care of the boilerplate wiring up that you need to do. There are Kubernetes API clients in all the major languages though, and my user controller is written in .NET. I don't want to throw a pile

of source code at you, but there are a couple of things to realize about building custom controllers which could use an example. Listing 20.4 is some C# code which is part of the add-user workflow (the full file is in the source code for this chapter).

Listing 20.4 - UserAddedHandler.cs, using the Kubernetes API with a client library

```
// lists service accounts in a namespace, using a
// field selector to search for an account by name:
var accounts = _client.ListNamespacedServiceAccount(
    groupNamespaceName,
    fieldSelector: $"metadata.name={serviceAccountName}");  
  

// if there's no match then we need to create the account:
if (!serviceAccounts.Items.Any())
{  
  

    var serviceAccount = new V1ServiceAccount
    {
        // set up the spec of the service account
    };  
  

    // create the resource:
    _client.CreateNamespacedServiceAccount(
        serviceAccount,
        groupNamespaceName);
}
```

The first thing is that working with the Kubernetes API feels very natural because all the operations are effectively the same thing you do in Kubectl but with different syntax, so writing a controller is not very difficult. The second thing is that you typically build Kubernetes resources in code, so you need to translate the YAML in your head into a series of objects, so writing a controller is cumbersome. Luckily you've got me to do it for you, and when you deploy the user controller it will be notified about the new users straight away and will run the add-user workflow.

TRY IT NOW Deploy the custom controller and verify that the add-user process gets triggered and creates all the authentication resources.

```
# on Windows you'll need to run this so you can decode Secrets:
.\base64.ps1  
  

# deploy the custom controller:
kubectl apply -f user-controller/  
  

# wait for it to spin up:
kubectl wait --for=condition=ContainersReady pod -l app=user-controller  
  

# check the controller logs:
kubectl logs -l app=user-controller  
  

# print the secret which is the user's token:
kubectl get secret tester3-token -n kiamol-ch20-authn-test -o
```

```
jsonpath='{{.data.token}}' | base64 -d
```

You see in that exercise that the controller automatically performs all the actions we did manually in section 17.3 - creating a namespace for the group, creating a service account in the namespace and requesting a new token. Actually it does more than that because it checks if any of those resources exist first and it only creates them if it needs to. Figure 20.7 shows the outcome is a service account which can be secured by applying RBAC rules to the group (which is the namespace), and a token which can be distributed to the user for them to store in their Kubectl context.

The custom controller for user resources is just a normal Kubernetes app with a Deployment and some RBAC policies.

When the controller starts it watches for changes to user resources. It gets notified about the existing users and creates the other resources.

```
PS> .\base64.ps1
PS>
PS>  
PS>kubectl apply -f user-controller/
serviceaccount/user-controller created
clusterrole.rbac.authorization.k8s.io/user-controller created
clusterrolebinding.rbac.authorization.k8s.io/user-controller created
deployment.apps/user-controller created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=user-controller
pod/user-controller-7f5dddf648-g29zn condition met
PS>
PS>kubectl logs -l app=user-controller --tail 4
** Created group namespace: kiamol-ch20-authn-test
** Created service account: tester3, in group namespace: kiamol-ch20-authn-test
** Created token: tester3-token, in group namespace: kiamol-ch20-authn-test
* Handled event: Added, for user: tester3
PS>
PS>kubectl get secret tester3-token -n kiamol-ch20-authn-test -o jsonpath='{  
  .data.token}' | base64 -d  
eyJhbGciOiJSUzI1NiIsImtpZCI6In1BNWs2VDDveGRMd01Yzk5UMEhBMkViQkdJeDN6NFZKSUl  
TzM3MHVuMkkifQ.eyJpc3MiOiJrdWJlc5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRLc
```

The controller creates a service account Secret which Kubernetes populates. This is the (abbreviated) authentication token to send to the user.

Figure 20.7 The controller automates the on-boarding process, except for distributing credentials

Please don't use this controller as your production authentication system; it's just a quick sample of how CRDs work with custom controllers to extend your Kubernetes experience. The code doesn't deal with updates to objects, and the design only allows one group per user, but you can see that the basis for managing authentication within Kubernetes is all there. You could have a source code repo with all your user YAMLS and group RBAC rules and deploy that as part of provisioning any new cluster.

The basic role of any controller is to implement a control loop - constantly watching for changes to objects and performing whatever tasks are needed to get the actual state of the system into the desired state specified in the object. They do that by watching resources for changes - just like using the watch parameter in Kubectl. The watch is an endless loop, and it gets notified when objects are created, updated or deleted. The user controller added the users it found when it started up, and it's still running in the background waiting for you to add another user.

TRY IT NOW The controller is watching for new users; create another user resource and confirm that the controller does its stuff.

```
# deploy a new user in the same SRE group:
kubectl apply -f users/update/

# print the latest controller logs:
kubectl logs -l app=user-controller --tail 4

# confirm the new user has a token:
kubectl get secret sre4-token -n kiamol-ch20-authn-sre -o jsonpath='{.data.token}' |
base64 -d
```

My output in figure 20.8 shows that the controller is behaving as it should - the desired state is that the user should be created as a service account with a namespace for the group. The namespace already exists so the controller doesn't need to do anything there, it just creates the service account and token. Custom controllers need to work on the same principle as standard controllers, that they get to the desired state no matter what the initial state is.

Adding another user in an existing group. The controller ensures the namespace, service account and token exist -but it only creates them if they're not already there.

```
PS>kubectl apply -f users/update/
user.ch20.kiamol.net/sre4 created
PS>
PS>kubectl logs -l app=user-controller --tail 4
** Group namespace exists: kiamol-ch20-authn-sre
** Created service account: sre4, in group namespace: kiamol-ch20-authn-sre
** Created token: sre4-token, in group namespace: kiamol-ch20-authn-sre
* Handled event: Added, for user: sre4
PS>
PS>kubectl get secret sre4-token -n kiamol-ch20-authn-sre -o jsonpath='{.dat
a.token}' | base64 -d
eyJhbGciOiJSUzI1NiIsImtpZCI6In1BNWs2VDdveGRMd01YZk5UMEhBMkViQkdJeDN6NFZKSUlr
TzM3MHVuMkkifQeyJpc3MiOiJrdWJlc5ldGVzL3NlcnPjY2VhY2NvdW50Iiwia3ViZXJuZXRLc
```

The new user token. Looks the same as the other token, but only the preamble is the same. The full token is unique for each user.

Figure 20.8 The declarative desired-state approach should be used in custom controllers

Custom controllers also need to own the cleanup logic, because there's a disconnect between creating a custom resource and seeing a whole bunch of extra resources being created. This is one concern with extending Kubernetes, and it means your controller code needs to be solid to make sure any failures don't leave objects lying around which the cluster admin isn't expecting. That's particularly important for any sensitive data stored in Secrets, like the tokens created for users.

TRY IT NOW Delete the test user - now there are no users in the group so the namespace should be removed too.

```
# check the authentication namespaces:  
kubectl get ns -l kiamol=ch20  
  
# delete the test user:  
kubectl delete user tester3  
  
# print the controller logs:  
kubectl logs -l app=user-controller --tail 3  
  
# confirm the tester namespace has been removed:  
kubectl get ns -l kiamol=ch20
```

You can see in figure 20.9 that the controller doesn't explicitly delete the Secret, but this type of Secret gets deleted by Kubernetes anyway when the service account gets deleted. The controller does check to see if there are any more users in the group and if not it deletes the namespace. Woe betide you if you added any other resources in that namespace — they'll be gone now.

Deleting a user triggers the removal process in the controller.

The service account gets deleted and then there are no more accounts in the namespace for the group, so it gets deleted too.

```
PS>kubectl delete user tester3  
user.ch20.kiamol.net "tester3" deleted  
PS>  
PS>kubectl logs -l app=user-controller --tail 3  
** Deleted service account: tester3, in group namespace: kiamol-ch20-authn-test  
** No accounts left, deleted namespace: kiamol-ch20-authn-test  
* Handled event: Deleted, for user: tester3  
PS>  
PS>kubectl get ns -l kiamol=ch20  
NAME STATUS AGE  
kiamol-ch20-authn-sre Active 36m
```

The only authentication namespace left is for the SRE group.

Figure 20.9 Controllers get notified when objects are removed so they can clean up the resources they created

Custom resources are a powerful way to extend Kubernetes, especially for use-cases like this where you want to provide a higher-level abstraction over stock Kubernetes objects. But those objects are just normal resources in the cluster and your controller code needs to allow for admins coming along and deleting them, without realizing they're managed by a controller. The user controller should really also watch Secrets, service accounts and namespaces to re-create anything that gets deleted outside of the controller process.

More sophisticated controllers might deploy their own RBAC rules to limit interference and would support running across multiple Pods for high availability. If you want to explore a production-grade example of CRDs and custom controllers, the cert-manager project (<https://cert-manager.io>) is a great example - it adds TLS certificate management to Kubernetes and can request certs and apply them to your web apps. The next level of sophistication comes with the Operator pattern.

20.3 Using Operators to manage third-party components

Operators use custom resources and controllers to provide full-lifecycle management for an application. They're used for complex apps where there are lots of operational tasks which are beyond the standard Kubernetes feature set. Stateful apps are a good example - if you decide to run a database in Kubernetes then upgrading the database server might mean putting the database into read-only mode and taking a backup before the upgrade.

You can't express requirements like that with standard Kubernetes resources; you can achieve something like it with Helm install hooks, but often the logic is quite involved and you need more control. The goal of the Operator is to implement all those operational requirements with controllers and custom resources, abstracting the complexity with simple resources like a database object and a backup object.

Third-party components that your app relies on are much easier to work with if they can be deployed with an Operator, because it gives you an as-a-service experience where you can focus on your app and leave the dependencies to manage themselves. In this section we'll deploy a modified version of the to-do list web app, using Operators to manage the dependencies: a message queue for asynchronous communication between components, and a database.

TRY IT NOW This version of the to-do app uses a message queue server called NATS. The NATS team publish an Operator that runs highly-available clusters of queue servers.

```
# deploy the CRDs and RBAC rules for the Operator:
kubectl apply -f nats/operator/00-prereqs.yaml

# deploy the Operator itself:
kubectl apply -f nats/operator/10-deployment.yaml

# wait for it to spin up:
kubectl wait --for=condition=ContainersReady pod -l name=nats-operator

# list the CRDs to see the new NATS types:
kubectl get crd
```

NATS is a message queue which acts as a go-between for application components, so they communicate by passing messages instead of connecting to each other directly. It's a powerful and very capable technology (another CNCF project), but in a production environment it needs to be set up carefully for high availability to make sure messages don't get lost. No-one knows how to do that better than the NATS team, and they provide the Operator you've just deployed. As you see in figure 20.10 it adds a CRD for a NatsCluster object which you can use to deploy a distributed message queue.

The NATS operator is distributed on GitHub - I've cloned the manifests in the lab folder for this exercise. The Operator runs as a Deployment, using RBAC rules for itself and the queue servers it will create.

```
PS>kubectl apply -f nats/operator/00-prereqs.yaml
serviceaccount/nats-operator created
clusterrolebinding.rbac.authorization.k8s.io/nats-operator-binding created
clusterrole.rbac.authorization.k8s.io/nats-operator created
serviceaccount/nats-server created
clusterrole.rbac.authorization.k8s.io/nats-server created
clusterrolebinding.rbac.authorization.k8s.io/nats-server-binding created
PS>
PS>kubectl apply -f nats/operator/10-deployment.yaml
deployment.apps/nats-operator created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l name=nats-operator
pod/nats-operator-59f59c5f7f-xhdwn condition met
PS>
PS>kubectl get crd
NAME          CREATED AT
natsclusters.nats.io   2020-07-25T19:43:47Z
natsserviceroles.nats.io 2020-07-25T19:43:47Z
users.ch20.kiamol.net    2020-07-25T18:19:40Z
```

The Operator creates new CRDs when it runs - the NatsCluster resource type is the one you use to define new queue clusters.

Figure 20.10 Operators should be simple to deploy and create all the resources they need.

The updated to-do app uses a message queue to improve performance and scalability. When users save messages in the new version the web app sends a message to the queue and then returns to the user - another component listens for those messages and adds the item to the database. You can scale up to hundreds of web Pods without needing to scale up the database, because the queue acts like a buffer, smoothing out any peaks in traffic. The queue becomes a critical component in the app and listing 20.5 shows just how simple it is to deploy a production-grade queue using the NATS Operator.

Listing 20.5 - todo-list-queue.yaml, a custom resource for a message queue

```
apiVersion: nats.io/v1alpha2      # the CRD uses an alpha version
kind: NatsCluster                # but it has been stable for
```

```

metadata:                      # a few years
  name: todo-list-queue
spec:                          # the spec defines the size of
  size: 3                      # the queue cluster and NATS version
  version: "1.3.0"

```

There are just two fields in the NatsCluster resource - the number of Pods to run as queue servers you want in your highly-available queue cluster, and the version of NATS to use. When you deploy it the Operator will create a Service for apps to use the queue, a Service for the instances of NATS to co-ordinate with each other, and a set of Pods each running NATS and configured with a Secret to run as a highly-available distributed queue.

TRY IT NOW Create the NATS cluster resource and confirm that the Operator creates all the queue resources.

```

# deploy the queue spec from listing 20.5:
kubectl apply -f todo-list/msgq/

# list the queues:
kubectl get nats

# list the Pods created by the Operator:
kubectl get pods -l app=nats

# and the Services:
kubectl get svc -l app=nats

# and Secrets:
kubectl get secrets -l app=nats

```

Figure 20.11 shows my NATS cluster is already up and running. The container image is just a few megabytes in size so Pods will start quickly even on nodes which need to pull the image. If you describe one of the Pods you'll see the spec uses some of the best practices you've learned from this book, like container probes and Pod priority. But the Pods are not managed by a Deployment or StatefulSet, the NATS Operator is the Pod controller which means it can use its own approach for maintaining availability.

The NatsCluster defines a highly-available queue in a simple custom resource.

The NATS Operator creates Pods running a clustered queue server.

```

PS>kubectl apply -f todo-list/msgq/todo-list-queue.yaml
natscluster.nats.io/todo-list-queue created
PS>
PS>kubectl get nats
NAME          AGE
todo-list-queue 9s
PS>
PS>kubectl get pods -l app=nats
NAME           READY   STATUS    RESTARTS   AGE
todo-list-queue-1 1/1     Running   0          18s
todo-list-queue-2 1/1     Running   0          16s
todo-list-queue-3 1/1     Running   0          15s
PS>
PS>kubectl get svc -l app=nats
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
                AGE
todo-list-queue   ClusterIP   10.108.244.241 <none>       4222/TCP
22s
todo-list-queue-mgmt   ClusterIP   None           <none>       6222/TCP,8
222/TCP,7777/TCP 24s
PS>
PS>kubectl get secret -l app=nats
NAME          TYPE   DATA  AGE
todo-list-queue  Opaque 1      2m38s

```

Services are for the queue Pods to communicate and for applications to send and receive messages on the queue.

The cluster configuration is stored in a Secret - more advanced cluster specs can include authentication and authorization rules for using the queue.

Figure 20.11 Two lines of YAML in a custom resource gets you a distributed message queue

The Operator pattern is a loose definition - there's no Operator object in Kubernetes and it's down to the project team to decide how to design, build and distribute their Operator. The NATS Operator is deployed from YAML manifests which are released on GitHub; other projects might use Helm or a tool called the Operator Lifecycle Manager (OLM). OLM adds some consistency around Operators with a catalog to publish and distribute them, but it's one of those technologies at the fringe of the Kubernetes ecosystem which hasn't taken off so far.

You can visit the OperatorHub site (<https://operatorhub.io>) to see the kind of projects which are available through OLM. A few of them are maintained by product teams, others are published by third parties or individuals. At the time of writing there are three Operators for the Postgres database, none of them backed by the Postgres project, and they vary wildly in capabilities and ease of use. There are no Operators for MySQL, and although there is one for MariaDB (a fork of MySQL) it's maintained by one person on GitHub and that might not be the kind of support structure you're happy with for a core component.

Which is not to say that Operators are not a viable technology, just that the pattern isn't restricted to OLM. If you're looking for an Operator for a product you need to search more widely

than the OperatorHub site and investigate the maturity of the options. The to-do list app can use MySQL as a data store and there is a very good MySQL Operator from the team at Presslabs who operate MySQL at scale in Kubernetes for their WordPress platform. The Operator is easy to use, well documented and well maintained, and it's simple to install with Helm.

TRY IT NOW Deploy the MySQL Operator using Helm, which can deploy and manage replicated MySQL databases in the cluster.

```
# add the Helm repository for the Operator:  
helm repo add presslabs https://presslabs.github.io/charts  
  
# and deploy a known version:  
helm install mysql-operator presslabs/mysql-operator --version v0.4.0 --atomic  
  
# wait for the Operator Pod to spin up:  
kubectl wait --for=condition=ContainersReady pod -l app=mysql-operator  
  
# and list the CRDs it installs:  
kubectl get crd -l app=mysql-operator
```

The MySQL Operator gives you a database-as-a-service experience - the Helm release creates CRDs for database and database backup objects, and an Operator which runs controllers for those objects. My output in figure 20.12 is snipped, but the Helm release notes also show you how to create a database - you just need a Secret for the MySQL password and a MysqlCluster object.

This MySQL Operator is maintained by Presslabs who run MySQL in a multi-tenant Kubernetes model. It's easy to use and creates highly-available databases.

```
PS>helm repo add presslabs https://presslabs.github.io/charts  
"presslabs" has been added to your repositories  
PS>  
PS>helm install mysql-operator presslabs/mysql-operator --version v0.4.0 --atomic  
NAME: mysql-operator  
LAST DEPLOYED: Sat Jul 25 20:59:33 2020  
NAMESPACE: default  
STATUS: deployed  
  
PS>kubectl wait --for=condition=ContainersReady pod -l app=mysql-operator  
pod/mysql-operator-0 condition met  
PS>  
PS>kubectl get crd -l app=mysql-operator  
NAME                                     CREATED AT  
mysqlbackups.mysql.presslabs.org      2020-07-25T19:59:30Z  
mysqlclusters.mysql.presslabs.org     2020-07-25T19:59:30Z
```

The Operator installs CRDs to specify database servers and backup jobs.

Figure 20.12 Complex software is easy to deploy and manage if you find a good Operator

You can now deploy highly-available databases using a simple resource spec. Listing 20.6 shows the manifest for the to-do list database, and it also illustrates some of the limitations of custom resources. The CRD schema lets you set MySQL configuration and also customize the Pod definition the Operator generates for the database server, so you can set resources requests and limits, affinity rules and a priority class. Those are Kubernetes details which leak into the database object spec, so it's not purely a description of the database you need - but it's far simpler than the replicated Postgres database we set up from scratch in chapter 8.

Listing 20.6 - todo-list-db.yaml, a replicated MySQL database using the Operator

```
apiVersion: mysql.presslabs.org/v1alpha1
kind: MysqlCluster
metadata:
  name: todo-db
spec:
  mysqlVersion: "5.7.24"
  replicas: 2
  secretName: todo-db-secret
  podSpec:
    resources:
      limits:
        memory: 200Mi
        cpu: 200m
```

When you deploy the MysqlCluster object the Operator creates a StatefulSet to run a replicated MySQL database, and a set of Services for consumers to connect to the database. There are separate Services for the cluster as a whole and for the manager and replica nodes, so you can choose how you want your client applications to connect.

TRY IT NOW Deploy the database and confirm the expected resources are created by the Operator.

```
# create the MysqlCluster resource:
kubectl apply -f todo-list/db/

# confirm the status:
kubectl get mysql

# show the StatefulSet:
kubectl get statefulset todo-db-mysql -o wide

# and the database Services:
kubectl get svc -l app.kubernetes.io/component=database
```

You'll see when you look at the StatefulSet that the Pod runs a MySQL container and a set of sidecar containers, including the Prometheus exporter for MySQL - shown in figure 20.13. This is one of the big advantages of Operators: they model applications with best practices so you don't need to dig into the finer details yourself. If you look at the spec of one of the Pods you'll see it has the standard Prometheus annotations we used in chapter 14, so if you have Prometheus running in your cluster it will pick up the new database Pods without any extra configuration and you can add MySQL metrics to your dashboard.

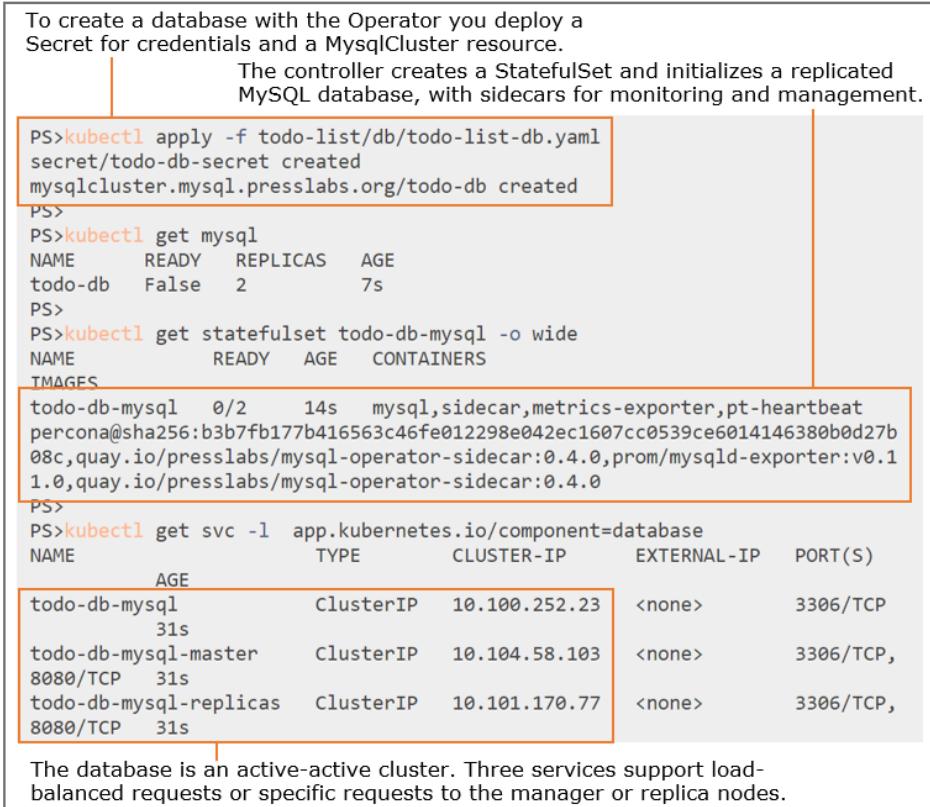


Figure 20.13 The Operator sets up an opinionated MySQL database with common best practices

Now we have a production-grade database and message queue running, defined in just 20 lines of YAML. We could standardize on NATS and MySQL for all our apps and the Operators would take care of multiple databases and queues - Operators are usually cluster-wide so you can still isolate application workloads in different namespaces. That's all the dependencies for the new to-do app, so we can deploy the rest of the components, the website and the message handler which saves data to the database.

TRY IT NOW Deploy the application components, a website and a message handler which both use the queue and the database.

```

# create shared app configuration:
kubectl apply -f todo-list/config/
# deploy the web app and handler:
kubectl apply -f todo-list/save-handler/ -f todo-list/web/
  
```

```
# wait for the Pods to start:  
kubectl wait --for=condition=ContainersReady pod -l app=todo-list  
  
# print the app logs:  
kubectl logs -l app=todo-list,component=save-handler  
  
# browse to the app Service on port 8020 and add a new item  
  
# print the latest logs from the message handler:  
kubectl logs -l app=todo-list,component=save-handler --tail 3
```

In that exercise you'll see the app works just the way it always has, and we've significantly reduced the amount of YAML we need to maintain compared to our custom Postgres version. My output in figure 20.14 actually hides the fact that the app doesn't quite work the way it did - now the to-do data is saved in a separate process you'll see there's a lag between adding an item and seeing it in the list, so you'll need to refresh. Welcome to *eventual consistency* which is a side-effect of the new messaging architecture; it has nothing to do with Operators so I'll leave you to research it if it's a new concept to you.

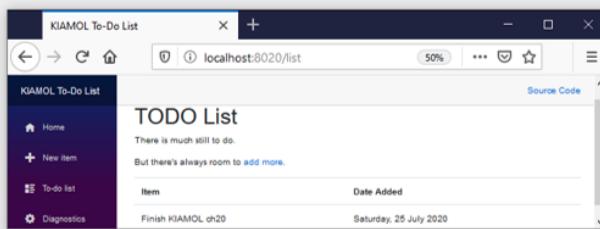
The application manifests are simple and they focus on the app components - dependencies are managed by Operators.

The handler connects to the queue and waits for messages.

```
PS>kubectl apply -f todo-list/config/
configmap/todo-list-config created
secret/todo-list-secret created
PS>
PS>kubectl apply -f todo-list/save-handler/ -f todo-list/web/
deployment.apps/todo-save-handler created
service/todo-web created
deployment.apps/todo-web created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=todo-list
pod/todo-save-handler-5df85b88b8-zh2fx condition met
pod/todo-web-8565fc8866-rb5pk condition met
PS>
PS>kubectl logs -l app=todo-list,component=save-handler
Connecting to message queue url: nats://todo-list-queue:4222
Listening on subject: events.todo.newitem, queue: save-handler
PS>
PS>kubectl logs -l app=todo-list,component=save-handler --tail 3
Received message, subject: events.todo.newitem
Saving item, added: 07/25/2020 20:04:55; event ID: da237566-6699-47ad-afdc-9
65395ce0e87
```

ID: 1; event ID: da237566-6699-47ad-afdc-965395ce0e87

PS>



The app looks the same to users but now the save workflow goes through the message queue, which is a much more scalable architecture.

Figure 20.14 Message queues and databases are critical components and the Operators run them with high availability

It's not just ease of deployment and high availability we get from the Operators: they'll also take care of safe upgrades to the core components and the MySQL database can be backed up to cloud storage by creating a MysqlBackup object. We won't go any further with that because we're not really running a production-grade to-do list application. In fact the setup we have running now is probably consuming quite a lot of resources on your lab machine, so we'll clear it down before we move on.

TRY IT NOW Remove the application, the custom resources and the Operators.

```
# delete the app components:
kubectl delete -f todo-list/web/ -f todo-list/save-handler/ -f todo-list/config/
# and the custom resources:
kubectl delete -f todo-list/db/ -f todo-list/msgq/
# uninstall the NATS Operator:
kubectl delete -f nats/operator/
# and the MySQL Operator:
helm uninstall mysql-operator
```

You can see my output in figure 20.15 where uninstalling everything is just the reverse of deployment. Operators don't necessarily remove every resource they create because they might contain data you don't want to lose. Expect to see ConfigMaps, Secrets, PersistentVolumeClaims and even CRDs hang around even after you remove an Operator - another good reason to use separate namespaces for your apps so you can remove everything cleanly.

Removing the custom resources triggers the delete workflow in the Operators, removing all the Pods, Services and StatefulSets.

```
PS>kubectl delete -f todo-list/web/ -f todo-list/save-handler/ -f todo-list/config/
service "todo-web" deleted
deployment.apps "todo-web" deleted
deployment.apps "todo-save-handler" deleted
configmap "todo-list-config" deleted
secret "todo-list-secret" deleted
PS>
PS>kubectl delete -f todo-list/db/ -f todo-list/msgq/
secret "todo-db-secret" deleted
mysqlcluster.mysql.presslabs.org "todo-db" deleted
natscluster.nats.io "todo-list-queue" deleted
PS>
PS>kubectl delete -f nats/operator/
serviceaccount "nats-operator" deleted
clusterrolebinding.rbac.authorization.k8s.io "nats-operator-binding" deleted
clusterrole.rbac.authorization.k8s.io "nats-operator" deleted
serviceaccount "nats-server" deleted
clusterrole.rbac.authorization.k8s.io "nats-server" deleted
clusterrolebinding.rbac.authorization.k8s.io "nats-server-binding" deleted
deployment.apps "nats-operator" deleted
PS>
PS>helm uninstall mysql-operator
release "mysql-operator" uninstalled
```

Removing the Operators stops the controllers running so custom resources won't be managed, even if the CRD is not removed by the Operator.

Figure 20.15 Operators don't necessarily clean up when they get deleted so you'll need to manually check for leftover resources

Operators are a very neat way to manage third-party dependencies. You need to put some effort into finding an Operator that works for you, and bear in mind many of these are open-source projects which may not have a lot of momentum. Compare the Prometheus Operator (one of the best examples on OperatorHub) which has 350 contributors and new updates pretty much daily, and the MySQL Operator from Oracle which has 18 contributors, and at the time of writing hasn't been worked on for two years. Lots of Operators are flagged as alpha or beta software and these are typically for critical components, so you need to be comfortable with the maturity level of anything you bring into your cluster.

But Operators are not just for third-party software; you can build your own Operator to simplify the deployment and ongoing maintenance of your own applications.

20.4 Building Operators for your own applications

There are two main reasons for building your own Operator. The first is for apps which have complex operational requirements and the second is for common components which are installed as services for many projects. An Operator for the to-do app might have custom upgrade logic - updating the Service to direct traffic to an "under maintenance" page, waiting for the message queue to be empty and then backing up the database. Any app with routine operations tasks that can be automated is a potential candidate for a custom Operator.

Building your own Operator is not a trivial task because it involves multiple custom resource types and multiple custom controllers. The complexity comes in mapping out all the scenarios, not just the workflows the Operator owns but any additional work it needs to do to put right any interference from human operators. There's a custom Operator in this chapter's resources but I won't focus on the code - it's an example of an as-a-service Operator for the web-ping application we used way back in chapter 10. That's an app which makes GET requests to a web address on a schedule and logs the response time. At a push I might be able to convince you that's a service many teams would use to monitor the uptime for their apps.

TRY IT NOW The web-ping Operator is deployed with YAML manifests like the NATS Operator. Install it to see how it runs and the CRDs it deploys.

```
# deploy RBAC rules and the Operator:
kubectl apply -f web-ping/operator/

# wait for it to be running:
kubectl wait --for=condition=ContainersReady pod -l app=web-ping-operator

# print the logs from the installer container:
kubectl logs -l app=web-ping-operator -c installer

# list new CRDs:
kubectl get crd -l operator --show-labels

# and the Pods for the Operator:
kubectl get pods -l app=web-ping-operator
```

You'll see in that exercise that the Operator Pod has several roles - it installs two CRDs and runs two containers, a custom controller for each custom resource. Figure 20.16 shows that the CRDs are for a WebPinger resource which defines the address and schedule to use, and a WebPingerArchive resource which is for archiving the results of a WebPinger resource.

The Operator manifest includes RBAC rules and a Deployment.

An init container in the Pod creates the CRDs.

```
PS>kubectl apply -f web-ping/operator/
serviceaccount/web-ping-operator created
clusterrole.rbac.authorization.k8s.io/web-ping-operator created
clusterrolebinding.rbac.authorization.k8s.io/web-ping-operator created
deployment.apps/web-ping-operator created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=web-ping-operator
pod/web-ping-operator-56dfc7d498-mpw4l condition met
PS>
PS>kubectl logs -l app=web-ping-operator -c installer
** Created CRD for Kind: WebPinger; ApiVersion: ch20.kiamol.net/v1
** Created CRD for Kind: WebPingerArchive; ApiVersion: ch20.kiamol.net/v1
* Done.
PS>
PS>kubectl get crd -l operator --show-labels
NAME                           CREATED AT          LABELS
webpingerarchives.ch20.kiamol.net 2020-07-26T18:46:08Z  kiamol=ch20,opera
tor=web-ping
webpingers.ch20.kiamol.net        2020-07-26T18:46:08Z  kiamol=ch20,opera
tor=web-ping
PS>
PS>kubectl get pods -l app=web-ping-operator
NAME                           READY   STATUS    RESTARTS   AGE
web-ping-operator-56dfc7d498-mpw4l 2/2     Running   0          53s
```

The CRDs are to deploy and manage web-ping instances.

The Operator Pod runs two containers, a custom controller for each resource type.

Figure 20.16 The web-ping Operator has a minimal manifest and deploys other resources when it runs

One of the goals of the Operator pattern is to keep things as simple as possible, so the installation is handled inside the Operator as much as possible. That keeps the deployment spec simple and removes any potential for errors - the Operator doesn't rely on a complex manifest (with the exception of RBAC rules which are needed in advance). The Operator spec you just deployed is in listing 20.7; there's an init container which creates the CRDs and two containers which run the controllers.

Listing 20.7 - 02-wpo-deployment.yaml, the Pod spec for the Operator

```
# this is the Pod spec in the Deployment resource
spec:
  serviceAccountName: web-ping-operator      # uses an account set up with
```

```

automountServiceAccountToken: true          # RBAC rules for access
initContainers:
  - name: installer                      # the installer creates CRDs
    image: kiamol/ch20-wpo-installer
containers:                                # app containers are controllers
  - name: pinger-controller               # app containers are controllers
    image: kiamol/ch20-wpo-pinger-controller
  - name: archive-controller
    image: kiamol/ch20-wpo-archive-controller

```

Not much can go wrong there. If the Operator needed ConfigMaps and Secrets, Services and PersistentVolumeClaims, it would own the creation of them all, keeping the complexity away from the administrator. The web-ping application has a few parameters to specify the address to test, the type of HTTP request and the interval between requests. The CRD lets users state those fields, and the custom controller running in the Operator creates a correctly configured Deployment for each instance of the app. Listing 20.8 shows a WebPinger resource configured to test my blog.

Listing 20.8 - webpinger-blog.yaml, a custom resource to test a web address

```

apiVersion: "ch20.kiamol.net/v1"
kind: WebPinger
metadata:
  name: blog-sixeyed-com
spec:                                     # parameters for the app are
  target: blog.sixeyed.com                 # much easier to specify in a
  method: HEAD                            # custom resource than using
  interval: "7s"                          # environment variables in a Pod

```

When you deploy this the Operator will create an instance of the web-ping app with a special configuration, logging responses to a file in JSON format for analysis. The Pod also includes a sidecar container which provides an HTTP API for clearing down the log file, and that powers the archive functionality.

TRY IT NOW Create a web ping resource and confirm the Operator creates an instance of the app which sends HTTP requests to my blog.

```

# create the custom resource:
kubectl apply -f web-ping/pingers/webpinger-blog.yaml

# print the latest logs from the Operator:
kubectl logs -l app=web-ping-operator -c pinger-controller --tail 4

# list web-ping Pods:
kubectl get po -l app=web-ping --show-labels

# print the logs from the app:
kubectl logs -l app=web-ping,target=blog.sixeyed.com -c web --tail 2

# and confirm logs are also written to the JSON file:
kubectl exec deploy/wp-blog-sixeyed-com -c web -- tail /logs/web-ping.log -n 2

```

This is a nice and easy way to deploy simple black-box observation of a site, and every team can include a WebPinger spec with their production deployments to keep an eye on the uptime of their app. If teams are familiar with the web-ping app, then it behaves in the same way as a manual deployment with human-readable logs printing to the standard output stream. As you see in figure 20.17 the logs are also written as JSON which is where the archive requirement comes in to protect disk space.

Creating a WebPinger resource triggers the custom controller in the Operator, which creates a ConfigMap, Deployment and Service.

The app runs and logs responses to the standard output stream.

```
PS>kubectl apply -f web-ping/pingers/webpinger-blog.yaml
webpinger.ch20.kiamol.net/blog-sixeyed-com created
PS>
PS>kubectl logs -l app=web-ping-operator -c pinger-controller --tail 4
** Created ConfigMap: wp-blog-sixeyed-com-config, in namespace: default
** Created Deployment: wp-blog-sixeyed-com, in namespace: default
** Created Service: wp-blog-sixeyed-com, in namespace: default
* Handled event: Added, for: blog-sixeyed-com
PS>
PS>kubectl get po -l app=web-ping --show-labels
NAME                               READY   STATUS    RESTARTS   AGE   LA
BELS
wp-blog-sixeyed-com-5ddb59c4b6-9mmxp   2/2     Running   0          40s   ap
p=web-ping,instance=wp-blog-sixeyed-com,pod-template-hash=5ddb59c4b6,target=
blog.sixeyed.com
PS>
PS>kubectl logs -l app=web-ping,target=blog.sixeyed.com -c web --tail 2
Making request number: 28; at 1595790397497
Got response status: 200 at 1595790397941; duration: 444ms
PS>
PS>kubectl exec deploy/wp-blog-sixeyed-com -c web -- tail /logs/web-ping.log
-n 2
{"level":"debug","message":"Making request number: 29; at 1595790404499","ti
imestamp":"2020-07-26T19:06:44.499Z"}
 {"level":"debug","message":"Got response status: 200 at 1595790405097; durat
ion: 598ms","timestamp":"2020-07-26T19:06:45.097Z"}
```

The ConfigMap adds a log file output so the app also writes logs in JSON.

Figure 20.17 The web-ping app is now easy to deploy and manage across multiple instances

Archiving is the only operational feature provided by the web-ping Operator, and it's simple to use - create a WebPingerArchive resource which specifies the target domain name. The custom controller for that resource looks for a web-ping Pod which matches the domain name, and uses the API in the sidecar container to grab a snapshot of the current log file and then clear the file down. This archive function is a good example of the work you need to do to automate

operational tasks - it's not just the CRD and the controller; the app itself needs a sidecar to provide additional admin features.

TRY IT NOW Test out the operational side of the web-ping app, creating an archive of the logs for the blog requests.

```
# print the number of lines in the log file:  
kubectl exec deploy/wp-blog-sixeyed-com -c web -- wc -l /logs/web-ping.log  
  
# create the archive resource:  
kubectl apply -f web-ping/pingers/archives/webpingarchive-blog.yaml  
  
# confirm that the Operator creates a Job for the archive task:  
kubectl get jobs -l kiamol=ch20  
  
# print the logs from the archive Pod:  
kubectl logs -l app=web-ping-archive,target=blog.sixeyed.com --tail 2  
  
# and confirm the application log file has been cleared down:  
kubectl exec deploy/wp-blog-sixeyed-com -c web -- wc -l /logs/web-ping.log
```

My output is in figure 20.18. This is a contrived example but it's a good way to see how Operators solve complex problems without getting lost in the actual problem. After the archive runs, the ping results are available in the Pod logs for the Job, while the web-ping Pod is still happily chewing up my bandwidth and has an empty log file to start filling again.

The web-ping Pod has been running for a while and there are 242 lines in the log file.

Creating an archive resource triggers the Operator controller to create a Job which archives the log file in the application Pod.

```
PS>kubectl exec deploy/wp-blog-sixeyed-com -c web -- wc -l /logs/web-ping.log
242 /logs/web-ping.log
```

PS>

```
PS>kubectl apply -f web-ping/pingers/archives/webpingerarchive.yaml
webpingerarchive.ch20.kiamol.net/blog-sixeyed-com created
```

PS>

```
PS>kubectl get jobs -l kiamol=ch20
NAME          COMPLETIONS   DURATION   AGE
wpa-blog-sixeyed-com-200726-1917  1/1        4s         7s
```

PS>

```
PS>kubectl logs -l app=web-ping-archive,target=blog.sixeyed.com --tail 2
{"level":"debug","message":"Making request number: 130; at 1595791111948","timestamp":"2020-07-26T19:18:31.948Z"}
 {"level":"debug","message":"Got response status: 200 at 1595791112284; duration: 336ms","timestamp":"2020-07-26T19:18:32.284Z"}
```

PS>

```
PS>kubectl exec deploy/wp-blog-sixeyed-com -c web -- wc -l /logs/web-ping.log
8 /logs/web-ping.log
```

The JSON log entries are now stored in the Pod for the archive Job.

The application log file has been cleared, there are 8 new lines from recent pings.

Figure 20.18 The archive workflow is managed by the Operator and triggered by creating another custom resource

Kubernetes Operators are usually written in Go and if you have the Go there are two tools which take care of a lot of the boilerplate code for you - KubeBuilder from Google and the Operator SDK which is part of the same toolset as Operator Lifecycle Manager. My Go isn't up to scratch yet so my Operator is written in .NET, and it took me about a day's worth of coding to build the Operator for this section. It's good fun digging into the Kubernetes API and writing code which creates and manages resources, and building up objects in code certainly makes you appreciate YAML a whole lot more.

But it's time to stop pinging my blog. This Operator doesn't have any admission controllers to stop you removing its CRDs, so you can delete them and that will trigger deletion of the custom resources - and then the controllers will clean up the resources they created.

TRY IT NOW Delete the Operator's CRDs and the custom resources will be deleted, triggering the removal workflows in the Operator.

```
# delete the CRDs by their labels:
kubectl delete crd -l operator=web-ping

# print the latest logs from the web-ping controller:
kubectl logs -l app=web-ping-operator -c pinger-controller --tail 4

# and from the archive controller:
```

```
kubectl logs -l app=web-ping-operator -c archive-controller --tail 2
```

You see in figure 20.19 that in this Operator it's controllers all the way down. The WebPing custom controller deletes the Deployment resource and then the system controllers delete the ReplicaSet and the Pod. The Operator doesn't try to replace or replicate Kubernetes functionality — it builds on it, using standard resources which have been used in production all around the world for many years, abstracting them to provide a simple user experience.

The Operator doesn't protect its CRDs. Deleting them deletes the custom resources.

```
PS>kubectl delete crd -l operator=web-ping
customresourcedefinition.apiextensions.k8s.io "webpingerarchives.ch20.kiamol.net" deleted
customresourcedefinition.apiextensions.k8s.io "webpingers.ch20.kiamol.net" deleted
PS>
PS>kubectl logs -l app=web-ping-operator -c pinger-controller --tail 4
** Deleted Service: wp-blog-sixeyed-com, in namespace: default
** Deleted Deployment: wp-blog-sixeyed-com, in namespace: default
** Deleted ConfigMap: wp-blog-sixeyed-com-config, in namespace: default
* Handled event: Deleted, for: blog-sixeyed-com
PS>
PS>kubectl logs -l app=web-ping-operator -c archive-controller --tail 2
** Deleted Job: wpa-blog-sixeyed-com-200726-1917, in namespace: default
* Handled event: Deleted, for: blog-sixeyed-com
```

Controllers are notified of the resource deletion and remove all the resources they created.

Figure 20.19 The custom controllers in this Operator manage standard Kubernetes resources

You need to understand how the Operator pattern works because you're sure to come across it in your Kubernetes journey - although you're more likely to use someone else's Operator than build your own. The key thing to understand is that it's a loose classification for a way of making apps simple to use and maintain, taking advantage of the extensibility of Kubernetes and making good use of the core system resources.

20.5 Understanding when to extend Kubernetes

We've covered a lot of ground in this chapter without digging too much into the detail. Extending Kubernetes is about getting the cluster to run your own code and what happens in that code depends on the problem you're trying to solve. The patterns are generic though, and figure 20.20 shows how all the pieces fit together with the Operator, custom resources and custom controllers for the web-ping application.

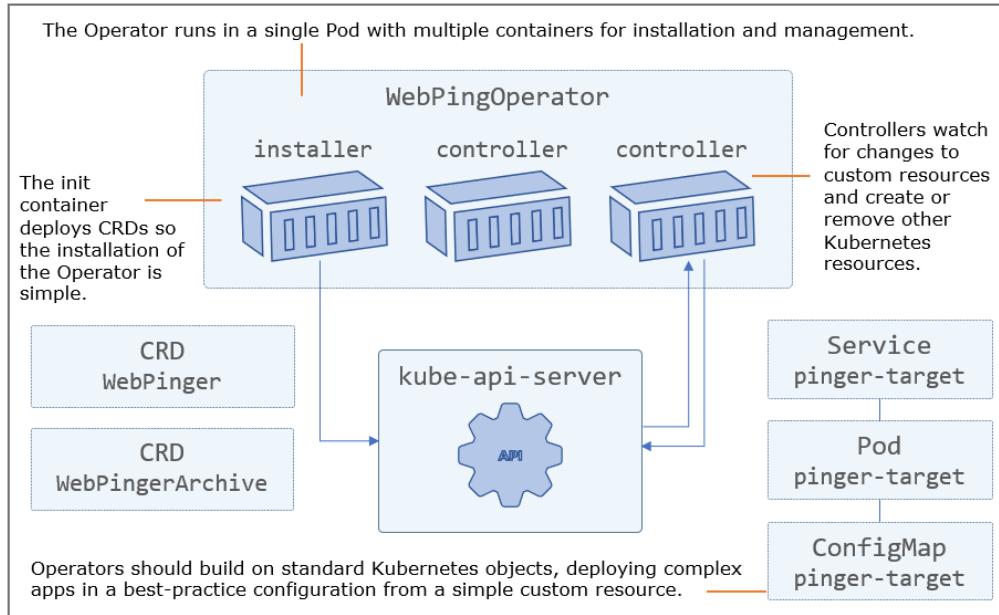


Figure 20.20 Operators and custom controllers make apps easy to manage - abstracting the complexity

There are a few guidelines for extending Kubernetes. The first is to make sure that you really need to - writing an Operator to save on the YAML for an app which runs a Deployment, a ConfigMap and a Service is overkill. If your goal is to ensure the app is deployed with the proper specification then it would be better to use admission controllers. Writing and maintaining custom controllers and Operators is a chunk of work, and if you don't map out all the workflows your app can get into an inconsistent state and then the Operator makes maintenance harder. Admins won't enjoy manually building the spec and deploying resources which custom controllers should own.

If you do have a clear need then start simple with CRDs and controllers and focus on the user experience; the whole point of custom resources is to simplify complex problems. Make use of the development toolkits if you're writing in Go and design your controllers to work with Kubernetes, building on the standard resources rather than reinventing them. It's always better to build a generic system when you have a couple of concrete examples to work from, so you know the generic approach will cover all the requirements. When you've done some complex upgrades and you know the workflows, or when you've deployed a common component multiple times and you know the variations, then it's time to design your Operator.

Third-party Operators are a great way to use somebody else's production experience to improve the reliability of your own applications. The key here is to find a good one, and that will take some investigation and experimentation with different options. Using an Operator to manage third-party components is a big dependency. You don't want to find that the project

stalls and you need to reverse engineer the Operator and take ownership yourself. The Operator Framework is the umbrella project which owns OLM and the Operator SDK and it was added as a new CNCF project a few weeks before I wrote this chapter, so that might bring some new energy to OperatorHub.

That's all for extending Kubernetes, so we can clean up before we get to the lab.

TRY IT NOW Clear down any remaining resources.

```
kubectl delete all,crd,secret,clusterrolebinding,clusterrole,serviceaccount,ns -l
    kiamol=ch20

kubectl delete pvc -l app=mysql-operator

kubectl delete configmap mysql-operator-leader-election
```

20.6 Lab

This lab will get you some experience writing a CRD and managing a custom controller. Don't worry: the controller is already written for you. In the lab folder there's a custom resource spec for a timecheck app, but there is no CRD so you can't deploy it. The task is to build the CRD, deploy the controller and the resource and verify it all works as expected. Just a couple of hints:

- the custom controller is all ready to go in the `timecheck-controller` folder
- your CRD names will need to match the ones in the resource
- you'll need to have a look at the logs when you deploy the controller; depending on the order in which you approach the lab it might not work as expected.

You can check my solution is up on GitHub as always:

<https://github.com/sixeyed/kiamol/blob/master/ch20/lab/README.md>

21

Running serverless functions in Kubernetes

Welcome to the last full chapter of the book - we're going to finish on a high note, learning how to turn your Kubernetes cluster into a serverless platform. There are lots of serverless platforms in the cloud but they're mostly bespoke systems and you can't easily move your AWS Lambda components to Azure Functions. The extensibility of Kubernetes makes it easy to deploy a serverless runtime in the cluster, which is just as portable as all your other apps. We'll cover some open-source projects in this chapter which give you a very Lambda-like experience, where you focus on the code and the platform packages and deploys it for you. Serverless functions run as containers in Pods so you manage them in the usual way, but the platform adds some higher-level abstractions.

There are several serverless platforms in the Kubernetes ecosystem which all take slightly different approaches. One of the most popular is the Knative project from Google, but it has an unusual workflow - you need to package your functions in Docker images yourself, and then Knative deploys them for you. I much prefer the code-first approach where you bring your code and the platform runs it in a container; that fits the goal of a very simple workflow for serverless functions. In this chapter we'll use Kubeless, another popular platform, and we'll also see how to abstract the serverless platform itself with the Serverless project.

21.1 How serverless platforms work in Kubernetes

What does serverless mean in the context of Kubernetes, where you obviously have servers involved because they're the nodes in your cluster. It's really about removing all the ceremony between writing code and having it running in a Pod - cutting out all the overhead in compiling the app, building a container image, designing the deployment and crafting YAML specs. AWS Lambda and Azure Functions have a CLI where you upload your code file and the function starts

running somewhere in the cloud. Serverless for Kubernetes gives you the same workflow but you know where your function is running - it will be in a Pod in your own cluster.

The Kubeless workflow is particularly neat: you just take your source code file and deploy it as a function using the Kubeless CLI. There are no additional artifacts to describe your function, and the CLI creates a custom resource with all the details and the source code. The Kubeless controller acts on the function resource and creates a Pod to run the function. You can trigger the function manually through the CLI or you can create a permanent trigger so the function listens for HTTP requests, subscribes to a message queue or runs on a schedule. Figure 21.1 shows the architecture of a Kubeless function.

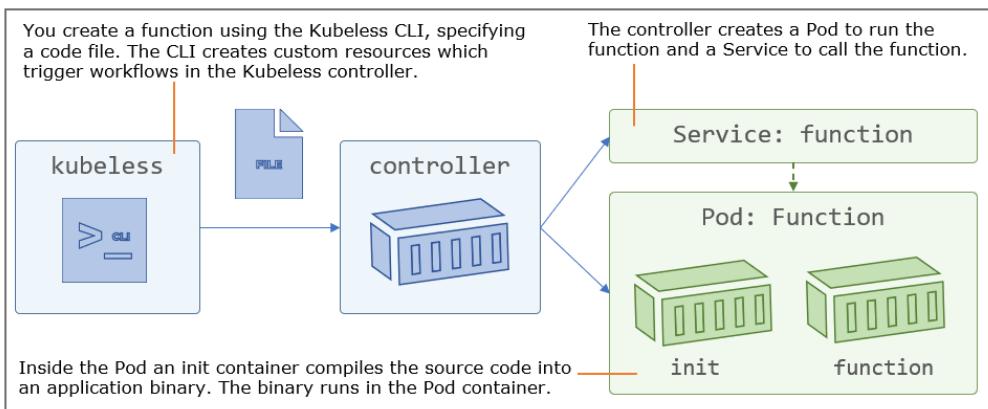


Figure 21.1 The serverless function with Kubeless turns your code into a running Pod

This workflow means you literally take a code file and run one command to get it running in a Pod, and another command if you want to expose it over HTTP. It's perfect for webhooks, integration components and simple APIs. Other serverless platforms support Kubernetes and work in similar ways - Nuclio, OpenWhisk and Fn Project all take your code, package it into a container to run, and support multiple triggers to invoke the function. They use standard resources like Pods and Services, and standard patterns like ingress controllers and message queues. In this chapter you'll use Kubeless to add new features to an existing app without changing the app itself - but we'll start simple with a Hello, Kiamol example.

TRY IT NOW Start by deploying Kubeless in your cluster - there's a snapshot of the most recent release in this chapter's folder.

```
# switch to this chapter's source:
cd ch21

# deploy the CRDs and controllers:
kubectl apply -f kubeless/
```

```
# wait for the controller to start:
kubectl wait --for=condition=ContainersReady pod -l kubeless=controller -n kubeless

# list the CRDs:
kubectl get crd
```

You can see in figure 21.2 that Kubeless uses the techniques you learned in chapter 20 - there are CustomResourceDefinitions for the HTTP and schedule triggers, and for serverless functions themselves. A controller runs which monitors all those resources and turns functions into Pods, HTTP triggers into ingress rules and scheduled triggers into CronJobs.

Kubeless extends Kubernetes with custom resources to define serverless functions.

```
PS>cd ch21
PS>
PS>kubectl apply -f kubeless/
namespace/kubeless created
clusterrole.rbac.authorization.k8s.io/kubeless-controller-deployer created
clusterrolebinding.rbac.authorization.k8s.io/kubeless-controller-deployer created
customresourcedefinition.apiextensions.k8s.io/functions.kubeless.io created
customresourcedefinition.apiextensions.k8s.io/httptriggers.kubeless.io created
customresourcedefinition.apiextensions.k8s.io/cronjobtriggers.kubeless.io created
configmap/kubeless-config created
deployment.apps/kubeless-controller-manager created
serviceaccount/controller-acct created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l kubeless=controller -n
kubeless
pod/kubeless-controller-manager-59d484f4d-pg8jg condition met
PS>
PS>kubectl get crd
NAME                               CREATED AT
cronjobtriggers.kubeless.io        2020-07-30T13:12:47Z
functions.kubeless.io               2020-07-30T13:12:47Z
httptriggers.kubeless.io           2020-07-30T13:12:47Z
```

The controller watches for changes to custom resources.

CRDs describe the function itself and the triggers to invoke it.

Figure 21.2 The Kubeless architecture adds new resources to provide the serverless abstraction

You can create Kubeless custom resources yourself, which fits neatly if your function code is all in source control and you already have a CI/CD process that uses Kubectl. The Kubeless CLI is an easier option - you run simple commands and it creates the resources for you. The CLI is a single binary you can install on macOS, Linux or Windows. You've installed enough software already though, so we'll run Kubeless in a Pod which has the CLI installed and Kubectl configured to work with your cluster.

TRY IT NOW Run the Kubeless CLI in a Pod and confirm it can connect to your cluster.

```
# create a Pod with the Kubeless CLI installed:
kubectl apply -f kubeless-cli.yaml

# wait for it to start:
kubectl wait --for=condition=ContainersReady pod kubeless-cli

# connect to a session in the Pod:
kubectl exec -it kubeless-cli -- sh

# print the Kubeless setup:
kubeless get-server-config

# stay connected to the Pod for the next exercise
```

Kubeless supports a lot of languages as you see in figure 21.3 - from common ones like Java, .NET and Python to interesting newcomers like Ballerina and Vert.x (which itself supports multiple JVM variants like Java, Kotlin and Groovy). If any of those fit with your tech stack then you can deploy functions with Kubeless, and it's a great way to evaluate new versions of your runtime or try out new languages.

You can install the Kubeless CLI yourself but this Docker image has it already installed.

```
PS>kubectl apply -f kubeless-cli.yaml
serviceaccount/kubeless-cli created
pod/kubeless-cli created
clusterrolebinding.rbac.authorization.k8s.io/kubeless-cli created
PS>
PS>kubectl wait --for=condition=ContainersReady pod kubeless-cli
pod/kubeless-cli condition met
PS>
PS>kubectl exec -it kubeless-cli -- sh
#
# kubeless get-server-config
INFO[0000] Current Server Config:
INFO[0000] Supported Runtimes are: ballerina0.981.0, dotnetcore2.0, dotnetcore
2.1, dotnetcore2.2, dotnetcore3.1, go1.13, go1.14, java1.8, java11, nodejs6, n
odejs8, nodejs10, nodejs12, php7.2, php7.3, python2.7, python3.4, python3.6, p
ython3.7, ruby2.3, ruby2.4, ruby2.5, ruby2.6, jvm1.8, nodejs_distroless8, node
jsCE8, vertx1.8
```

Kubeless supports lots of languages and lots of versions for your serverless functions.

Figure 21.3 Serverless functions can be written in all the major languages with Kubeless

Serverless functions are meant to do a single focused task and the source code should usually be a single file, but Kubeless does let you deploy larger projects. It understands the dependency management systems for all the runtimes, and it will fetch dependencies as part of the

deployment. You can upload a ZIP archive containing a whole project structure when you create a new function or you can upload a single file. Listing 21.1 shows a simple hello function in Java - don't worry about the source code too much — this is just to show you that there's a standard approach to writing a Kubeless function, no matter what language you use.

Listing 21.1 - hello-kiamol.java, a simple Java serverless function

```
# the code is in a normal Java class:
public class Kiamol {

    # this is the method which Kubeless invokes:
    public String hello(io.kubeless.Event event, io.kubeless.Context context) {

        # it just returns a string:
        return "Hello from chapter 21!";
    }
}
```

Every function receives two fields - one with details about the event including the type of trigger and any data sent by the caller, and one with the context of the function itself including the runtime and the timeout set for the function to complete. There's no service account token to authenticate with the Kubernetes API server, and your functions will normally be application features rather than Kubernetes extensions (although they do run in Pods so the token can be automounted in the filesystem if you do need it).

When functions are invoked they do whatever they need to do, and they can return a string which is sent as the response to the caller if the function was triggered by an HTTP request. The function code executes inside a Pod container so you can write log entries to the standard output stream and see them in Pod logs. You can deploy the simple function from listing 21.1 and check the Pod spec to see how Kubeless works.

TRY IT NOW Deploy the simple Hello Java function with the Kubeless CLI and look at the Kubernetes objects it creates.

```
# inside the Pod there is a copy of the book's code:
cd /kiamol/ch21

# deploy the Java function from listing 21.1:
kubeless function deploy hello-kiamol --runtime java11 --handler Kiamol.hello --from-
    file functions/hello-kiamol/hello-kiamol.java

# list all functions:
kubeless function ls

# and list Pods and ConfigMaps for the function:
kubectl get pods -l function=hello-kiamol
kubectl get cm -l function=hello-kiamol

# print the details, showing the build steps:
kubectl describe pod -l function=hello-kiamol | grep INFO | tail -n 5
```

Figure 21.4 ends with the logs from the init container in the Pod. Kubeless has a very nice way of packaging apps without needing to build and push a container image. Every supported runtime has an init container image which has all the build tools for the runtime - in this case it's the Java JDK and Maven for dependency management. The init container loads the function source code from a ConfigMap volume, builds the application and copies the output to an EmptyDir volume. The app container runs from an image with the language runtime and it launches the compiled application from the shared EmptyDir volume.

Deploying a serverless function in Kubeless is just a single command, identifying the language runtime to use, the path to the code file and the name of the method to call.

```
# cd /kiamol/ch21
#
# kubeless function deploy hello-kiamol --runtime java11 --handler Kiamol.hello
--from-file functions/hello-kiamol/hello-kiamol.java
INFO[0000] Deploying function...
INFO[0000] Function hello-kiamol submitted for deployment
INFO[0000] Check the deployment status executing 'kubeless function ls hello-ki
amol'
#
# kubeless function ls
NAME      NAMESPACE      HANDLER      RUNTIME      DEPENDENCIES      STATUS
hello-kiamol  default  Kiamol.hello  java11      0/1 NOT
READY
#
# kubectl get pods -l function=hello-kiamol
NAME          READY   STATUS    RESTARTS   AGE
hello-kiamol-65d65494c7-g5bcb   1/1     Running   0          56s
#
# kubectl get cm -l function=hello-kiamol
NAME      DATA      AGE
hello-kiamol  3        56s
#
# kubectl describe pod -l function=hello-kiamol | grep INFO | tail -n 5
[INFO] -----
[INFO] Total time: 4.951 s
[INFO] Finished at: 2020-07-30T14:30:28+00:00
[INFO] Final Memory: 20M/386M
[INFO] -----
```

Deploy the function and the Kubeless controller creates a Deployment with a single Pod and a ConfigMap.

The Pod container runs from a standard Java image, and there is an init container which compiles the code using the Java Development Kit. This output is from Maven, a Java build system.

Figure 21.4 Kubeless makes good use of init containers to compile functions without building images

This approach means a slower startup time for functions compared to platforms which build and push an image for every function, but it removes a lot of friction for developers. It also means your cluster doesn't need to be configured with Secrets that have write permission to registries,

and you don't even need to use a container runtime which can build and push images. Now you have a function running in a Pod and you didn't need a build server, and you didn't even need Java or Maven installed.

Function Pods have an HTTP server listening for requests and when you create a trigger it sends requests to the Service for the Pod. You can scale and autoscale functions in the same way as standard application Pods and requests get load-balanced by the Service in the usual way. Kubeless builds on the established Kubernetes resources and gives you an easy way to get your apps running. There are no triggers for this function yet, so you can't call it from outside the cluster but you can start a proxy with Kubectl and call the Service directly.

TRY IT NOW You can invoke the function with an HTTP request proxied by Kubectl, or you can use the Kubeless CLI.

```
# show the Service for the function:  
kubectl get svc -l function=hello-kiamol  
  
# start a proxy to route HTTP requests to the cluster:  
kubectl proxy -p 8080 &  
  
# call the function using the proxy:  
curl http://localhost:8080/api/v1/namespaces/default/services/hello-kiamol:http-  
function-port/proxy/  
  
# but it's simpler to call it with the CLI:  
kubeless function call hello-kiamol  
  
# we're done with the Pod session for now:  
exit
```

You can see in figure 21.5 that the Kubeless CLI gives you an easy way to interact with your functions, but every function is a Kubernetes app so you can also work with it using the usual Kubectl commands.

Kubeless functions have a ClusterIP Service so they can be invoked with an HTTP request.

```
# kubectl get svc -l function=hello-kiamol
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
hello-kiamol  ClusterIP  10.107.247.111  <none>        8080/TCP    62s
#
# kubectl proxy -p 8080 &
# Starting to serve on 127.0.0.1:8080

# curl http://localhost:8080/api/v1/namespaces/default/services/hello-kiamol:ht
tp-function-port/proxy/
Hello from chapter 21!#
#
# kubeless function call hello-kiamol
Hello from chapter 21!
```

You can call the function with curl through a Kubectl proxy and see the response from the Java code.

Or you can call the function with the Kubeless CLI.

Figure 21.5 Serverless is really a deployment abstraction - Kubeless creates standard Kubernetes resources

This function isn't too useful. One area where serverless functions really shine is adding new features to existing applications, without needing any changes or deployments to the main app. In the next sections we'll use serverless functions to add some much-needed features to the beloved (or maybe not by now) to-do app.

21.2 Triggering functions from HTTP requests

You learned about ingress in chapter 15 - it's the usual way to route incoming requests across multiple apps running in your cluster. Ingress rules also hide the detail of how a single application is put together, so different paths in the same domain name might be served by one component or by different components. You can leverage that with serverless functions to add new features that appear to be part of the main application.

We'll do that to add a new REST API for the to-do list application, building on the work we did in chapter 20. There we introduced a message queue for communication between the website and a message handler that saved new items to the database. Any component with access can post a message to the queue, so we can run a simple API in a serverless function to do that. Let's start by getting the to-do app running again.

TRY IT NOW Deploy the to-do app using simple Deployment specs for the NATS message queue and the database.

```
# deploy all the components of the app:
kubectl apply -f todo-list/config/ -f todo-list/db/ -f todo-list/msgq/ -f todo-
list/web/ -f todo-list/save-handler/
```

```
# wait for the application Pods to start:
kubectl wait --for=condition=ContainersReady pod -l app=todo-list

# fetch the URL for the app:
kubectl get svc todo-web -o
    jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8021'

# browse the app to confirm it's working
```

Nothing special in figure 21.6 - just the same old to-do app running. This isn't the full production deployment using Operators to manage the message queue and database, but it's the same architecture and functionality.

This version of the to-do app deploys its own NATS message queue and database, it doesn't use Operators for those components.

```
PS>kubectl apply -f todo-list/config/ -f todo-list/db/ -f todo-list/msgq/ -f todo-list/web/ -f todo-list/save-handler/
configmap/todo-list-config created
secret/todo-list-secret created
deployment.apps/todo-db created
persistentvolumeclaim/todo-db-pvc created
secret/todo-db-secret created
service/todo-db created
configmap/nats-config created
service/message-queue created
statefulset.apps/nats created
service/todo-web created
deployment.apps/todo-web created
deployment.apps/todo-save-handler created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l app=todo-list
pod/todo-save-handler-dc746d69c-8cfmz condition met
pod/todo-web-8565fc8866-5m9hn condition met
PS>
PS>kubectl get svc todo-web -o jsonpath='http://{{.status.loadBalancer.ingress[0].*}}:8021'
http://localhost:8021
```

It's the same old app but the messaging architecture means it will be easy to add features using serverless functions.

Figure 21.6 Serverless functions can integrate nicely with your app if it has the right architecture

I'm going to use lots of different languages for the serverless functions in this section so you get a feel for how they work and see the similarities between runtimes. The to-do API uses Node.js and it makes use of some additional libraries to send messages to NATS. Kubeless takes care of loading dependencies in the init container when a function Pod starts; you just need to specify the dependencies in a file using the standard format for the runtime. Listing 21.2 shows the main part of the API function using the NATS library to send messages.

Listing 21.2 - server.js, a serverless API in Node.js

```
# the function method receives the same event and context data:
```

```

function handler(event, context) {

    # inside the function the code builds a message:
    var newItemEvent = {
        Item: {
            Item: event.data,
            DateAdded: new Date().toISOString()
        }
    }

    # and publishes it to the NATS queue:
    nc.publish('events.todo.newitem', newItemEvent)
}

```

The Node.js function has the same structure as the Java function in listing 21.1 - it receives event and context objects with details about the call. The data is the new to-do item sent by the caller, and the code builds that into a message which it publishes to the queue. The format is the same that the website publishes, so the message handler will receive messages from both the API and the web app and save new items to the database. Alongside the code file is a package file which lists dependencies, so this is ready to deploy with Kubeless.

TRY IT NOW Functions with dependencies get deployed in the same way; you just need to specify the dependency file as well as the code file in the deployment command.

```

# connect to a session in the CLI Pod:
kubectl exec -it kubeless-cli -- sh

# switch to the chapter folder:
cd /kiamol/ch21

# deploy the API function with the dependencies:
kubeless function deploy todo-api --runtime nodejs12 --handler server.handler --from-
    file functions/todo-api/server.js --dependencies functions/todo-
        api/package.json

# show the function:
kubeless function ls todo-api

# wait for the Pod to be ready:
kubectl wait --for=condition=ContainersReady pod -l function=todo-api

# call the function:
kubeless function call todo-api --data 'Finish KIAMOL ch21'

# print the function logs:
kubeless function logs todo-api | grep event

# and the message handler logs:
kubectl logs -l component=save-handler --tail 1

# leave the session and refresh your browser
exit

```

Messaging architecture makes this sort of new functionality simple. The message handler listens for events when a new item is created and saves them to the database - it doesn't matter what the source of the event is. You can see in figure 21.7 that the API function publishes a message with a random event ID, and that's the message which the handler picks up. If you refresh your to-do list in the browser you'll see the new item is there.

Functions which make use of libraries are deployed with a dependencies file.
This is a Node.js app using a standard package.json dependency file.

```
PS>kubectl exec -it kubeless-cli -- sh
#
# cd /kiamol/ch21
#
# kubeless function deploy todo-api --runtime nodejs12 --handler server.handler
--from-file functions/todo-api/server.js --dependencies functions/todo-api/packa
ge.json
INFO[0000] Deploying function...
INFO[0000] Function todo-api submitted for deployment
INFO[0000] Check the deployment status executing 'kubeless function ls todo-api'

#
# kubeless function ls todo-api
NAME      NAMESPACE      HANDLER      RUNTIME      DEPENDENCIES
STATUS
todo-api    default    server.handler  nodejs12    nats: 1.4.9
1/1 READY                                     uuid: 8.2.0

# kubectl wait --for=condition=ContainersReady pod -l function=todo-api
pod/todo-api-74984f784d-m7449 condition met
#
# kubeless function call todo-api --data 'Finish KIAMOL ch21'

# kubeless function logs todo-api | grep event
** New item published, event ID: 5409a828-6128-4a72-91ab-2278b47482ff
#
# kubectl logs -l component=save-handler --tail 1
Item saved; ID: 1; event ID: 5409a828-6128-4a72-91ab-2278b47482ff
```

Calling the function publishes a message to NATS.

The original message handler in the to-do app sees the message and creates a new item in the database.

Figure 21.7 An API for the to-do app running as a serverless function in under 20 lines of code

Serverless functions fit very nicely with event-driven architectures because they can just plug into the message stream, generating or consuming different types of event. That doesn't mean messaging is a requirement because functions can always integrate with apps at different levels - without the message queue the new API function could have used database integration and written a new row into the table. It's better to have higher level integrations where you can let components own their own data, but your function code can do whatever fits with your current architecture.

Right now you have Services for the to-do web app and the API function. The next stage is to publish them both using ingress. You can choose your URL structure when you mingle serverless functions with an existing app. In this case I'm going to use a subdomain for the function so the app will be available at `todo.kiamol.local` and the function at `api.todo.kiamol.local`. To make that work you'll need to deploy an ingress controller and set up some domain names in your hosts file.

TRY IT NOW Deploy an ingress controller and add some domains to your hosts file - you'll need to run your terminal as Administrator on Windows and use `sudo` on Linux or macOS.

```
# run an Nginx ingress controller:
kubectl apply -f ingress-nginx/

# deploy ingress rules for the app and API:
kubectl apply -f todo-list/web/ingress/ -f functions/todo-api/ingress/

# print the ingress rules:
kubectl get ingress

# add domains to your hosts file - on Windows:
.\add-todo-to-hosts.ps1

# OR on Linux/macOS:
chmod +x ./add-todo-to-hosts.sh && ./add-todo-to-hosts.sh

# insert a new item with the API:
curl --data 'Plan KIAMOL ch22' http://api.todo.kiamol.local/todos

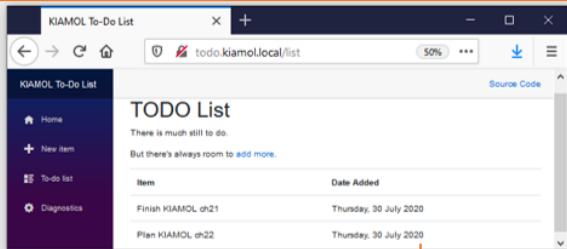
# browse to http://todo.kiamol.local/list
```

You'll see in that exercise that the ingress rules hide all the details about app Pods and function Pods and consumers just use URLs which appear to be different parts of one big application. There's a tiny screenshot in figure 21.8 which shows two items in the list; both were added by the new API function but they behave in just the same way as if they'd been added in the website.

Web applications in Pods and serverless functions can be made public using ingress rules and an ingress controller.

```

PS>kubectl apply -f ingress-nginx/
namespace/kiamol-ingress-nginx created
configmap/ingress-nginx-controller created
service/ingress-nginx-controller created
deployment.apps/ingress-nginx-controller created
serviceaccount/ingress-nginx created
clusterrole.rbac.authorization.k8s.io/ingress-nginx created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx created
role.rbac.authorization.k8s.io/ingress-nginx created
rolebinding.rbac.authorization.k8s.io/ingress-nginx created
PS>
PS>kubectl apply -f todo-list/web/ingress/ -f functions/todo-api/ingress/
ingress.networking.k8s.io/todo-web created
service/todo-web configured
ingress.networking.k8s.io/todo-api created
PS>
PS>kubectl get ingress
NAME      CLASS      HOSTS          ADDRESS    PORTS      AGE
todo-api  <none>    api.todo.kiamol.local  80        8s
todo-web   <none>    todo.kiamol.local    80        8s
PS>
PS>.\add-todo-to-hosts.ps1
PS>
PS>curl --data 'Plan KIAMOL ch22' http://api.todo.kiamol.local/todos
PS>
PS>
```



The API could be a path in the main app domain like /api but I'm using a separate subdomain.

Items added in the API work in the usual way.

Figure 21.8 Ingress rules hide the internal architecture, which could be one app or multiple functions

You can build an entire API from serverless functions using a different function for each path, but for large APIs that's going to mean a lot of Pods which each have their own compute requirements. Kubeless doesn't apply resource requests or limits by default, so running hundreds of function Pods is more likely to put your nodes under memory pressure than running a dozen replicas of a single API Pod. The eviction scenarios we covered in chapter 19 are more likely if you rely heavily on serverless functions, because every function will use some memory to load the language runtime.

That's not to say a serverless API isn't feasible; it just needs some additional planning - you can add a resource block to function specs if you create the custom resource yourself in YAML rather than using the Kubeless CLI. You'll also want to think carefully about the runtimes you use because image size affects your ability to scale quickly, and larger images provide a bigger surface area for attacks. As of Kubeless 1.0.7 the Go runtime image is under 60MB and the Node.js image is ten times the size.

Now that you see how serverless functions can extend an existing application, we'll round off the to-do app with a few more features using different languages and different triggers.

21.3 Triggering functions from events and schedules

All serverless platforms have a similar architecture where functions can be invoked by different triggers - HTTP requests, messages arriving on a queue and schedules are the common types of trigger. Separating the trigger from the function itself simplifies the code because the platform wires everything up for you and it means the same function can be called in different ways. We can use a message queue trigger to add an auditing feature to the to-do app, logging when new items are created.

The new feature will listen for the same new-item message that the existing message handler uses to save items to the database. Queues like NATS support a publish-subscribe pattern which means any number of subscribers can listen for the new-item messages and they will all get a copy. Kubeless will subscribe to the queue and call the function when there are incoming events, so there's no special messaging code inside the function. The audit handler just writes a log entry for every item it sees and the function code is just the two lines of Python in listing 21.3.

Listing 21.3 - audit.py, a Python auditing function

```
def handler(event, context):
    print(f"AUDIT @ {event['data'][ 'Item'][ 'DateAdded']}":
          {event[ 'data'][ 'Item'][ 'Item']})
```

There's no difference in the setup of the function; the standard event and context objects get provided by Kubeless no matter what type of trigger invokes the function. The `call` command in the Kubeless CLI works in the same way too, so you can deploy this function and verify it by sending fake data in the same format as the new-item message.

TRY IT NOW Deploy the Python audit function and invoke it directly with the Kubeless CLI to test it.

```
# deploy the function:
kubeless function deploy todo-audit --runtime python3.7 --handler audit.handler --
  from-file functions/todo-audit/audit.py

# wait for the function Pod to be ready:
kubectl wait --for=condition=ContainersReady pod -l function=todo-audit

# confirm the function status:
kubeless function ls todo-audit
```

```
# connect to a Kubeless CLI session:
kubectl exec -it kubeless-cli -- sh

# and call the new function:
kubeless function call todo-audit --data '{"Item":{"Item":"FAKE
ITEM!","DateAdded":"2020-07-31T08:37:41"}}'

# print function logs:
kubeless function logs todo-audit | grep AUDIT

# and leave the Pod session:
exit
```

You can see in figure 21.9 this is a really simple developer experience. When the function is deployed there are no default triggers so there's no way to invoke it except from the Kubeless CLI (or by proxying access to the function service). Developers can quickly deploy a function and test it, iterate over the code using the `kubeless update` command and only publish a trigger to wire up the function when they're happy with it.

Functions are deployed in the same way and create the same Kubernetes resources no matter what runtime they use.

```
PS>kubeless function deploy todo-audit --runtime python3.7 --handler audit.handler --from-file functions/todo-audit/audit.py
INFO[0000] Deploying function...
INFO[0000] Function todo-audit submitted for deployment
INFO[0000] Check the deployment status executing 'kubeless function ls todo-audit'

PS>
PS>kubectl wait --for=condition=ContainersReady pod -l function=todo-audit
pod/todo-audit-845df677bc-djxtb condition met
PS>
PS>kubeless function ls todo-audit
NAME      NAMESPACE      HANDLER      RUNTIME      DEPENDENCIES
STATUS
todo-audit    default    audit.handler    python3.7
1/1 READY
PS>
PS>kubectl exec -it kubeless-cli -- sh
#
# kubeless function call todo-audit --data '{"Item":{"Item":"FAKE ITEM!","DateAd
ded":"2020-07-31T08:37:41"}}'
#
# kubeless function logs todo-audit | grep AUDIT
```

You can use the `call` command to test a function without deploying any triggers - the data here is the same JSON payload the function will get from the NATS messages.

Functions are deployed in the same way and create the same Kubernetes resources no matter what runtime they use.

Figure 21.9 This is the value of the serverless workflow - deployment and test with single commands

Kubeless natively supports message triggers for the Kafka messaging system, and it has a pluggable architecture so you can add a trigger for NATS. The Kubeless project maintains that trigger (along with other plugins like a trigger for streams of data from AWS Kinesis) and you can deploy it to create a new CRD and controller for NATS trigger resources.

TRY IT NOW Deploy the NATS plugin for Kubeless and add a NATS trigger to invoke the audit function when messages are published to the new-item queue.

```
# deploy the NATS trigger:
kubectl apply -f kubeless/nats-trigger/

# wait for the controller to be ready:
kubectl wait --for=condition=ContainersReady pod -l kubeless=nats-trigger-controller
-n kubeless

# connect to a Kubeless session:
kubectl exec -it kubeless-cli -- sh
# create the trigger:
kubeless trigger nats create todo-audit --function-selector function=todo-audit --
trigger-topic events.todo.newitem

# leave the session:
exit

# call the API function:
curl --data 'Promote DIAMOL serialization on YouTube'
http://api.todo.kiamol.local/todos

# and print the audit logs:
kubectl logs -l function=todo-audit
```

The full logs from a function Pod are quite verbose because they include entries for the HTTP requests from the container's liveness probe. My output in figure 21.10 is snipped, but you can see the new workflow in action: an item is added using the API function through its HTTP trigger, that function drops a message onto the queue and that triggers the audit function which writes the log entry.

The NATS trigger is a separate install which adds a CRD and a new controller.

```
PS>kubectl apply -f kubeless/nats-trigger/
deployment.apps/nats-trigger-controller created
clusterrole.rbac.authorization.k8s.io/nats-controller-deployer created
clusterrolebinding.rbac.authorization.k8s.io/nats-controller-deployer created
customresourcedefinition.apiextensions.k8s.io/natstriggers.kubeless.io created
PS>
PS>kubectl wait --for=condition=ContainersReady pod -l kubeless=nats-trigger-con
troller -n kubeless
pod/nats-trigger-controller-df8995984-qt6sm condition met
PS>
PS>kubectl exec -it kubeless-cli -- sh
#
# kubeless trigger nats create todo-audit --function-selector function=todo-audi
t --trigger-topic events.todo.newitem
INFO[0000] NATS trigger todo-audit created in namespace default successfully!
#
# exit
command terminated with exit code 6
PS>
PS>curl --data 'Promote DIAMOL serialization on YouTube' http://api.todo.kiamol.
local/todos
PS>
PS>kubectl logs -l function=todo-audit
AUDIT @ 2020-07-31T08:37:41: FAKE ITEM!
AUDIT @ 2020-07-31T09:23:16.980Z: Promote DIAMOL serialization on YouTube
```

Creating a to-do item publishes a message to the queue.

The new function gets a copy of the message and writes a simple audit log. This one is about a series of videos for Manning's *Learn Docker in a Month of Lunches*, which you'll find on my YouTube channel.

Figure 21.10 Message queues decouple components; here the audit function is invoked when the API function publishes a message, without the functions communicating directly

This is another good example of how a flexible architecture helps you add features quickly and easily - and safely, because there's no change to the existing application to add this in. Heavily regulated industries like banking often have product backlogs which are almost entirely driven by new laws, and the ability to inject logic into existing workflows is a powerful argument for serverless. Behind the scenes the NATS trigger controller subscribes to the event messages, and when they arrive it invokes the function using its HTTP endpoint. That's all abstracted from the function code which can just focus on the task.

One more example for this section will round off the main features of Kubeless, using a scheduled trigger and creating functions through YAML instead of with the CLI. The Kubeless CLI is just a wrapper which creates custom resources for you. In the `todo-mutating-handler` folder are two YAML manifests for custom resources, one for a Function and one for a CronJobTrigger. I won't echo the specs here but if you look at the Function you'll see it uses

PHP and the source code is *inside* the custom resource spec. This approach works nicely with CI/CD pipelines because you can deploy using Kubectl without needing to craft Kubeless commands.

TRY IT NOW Deploy the new function as a custom resource -you don't need the Kubeless CLI for this workflow so you don't need to connect to a session in the CLI Pod.

```
# create the Kubeless resources:  
kubectl apply -f functions/todo-mutating-handler/  
  
# print all the schedule triggers:  
kubectl get cronjobtriggers  
  
# and the Kubernetes CronJobs:  
kubectl get cronjobs  
  
# wait for the Job to run:  
sleep 90  
  
# print the logs from the Job Pod:  
kubectl logs -l job-name --tail 2  
  
# refresh your to-do list in the browser
```

You'll see when you run this exercise that it adds some much-needed functionality to the to-do app, to sanitize the data that comes in. The CronJob makes a function call once every minute and the PHP script executes to clean up the data and make sure the to-do list items are useful tasks - you can see my output in figure 21.11.

Kubeless functions and triggers can be created without the CLI by applying manifests with custom resources.

A scheduled trigger creates a Kubernetes CronJob that invokes the function with an HTTP request.

```
PS>kubectl apply -f functions/todo-mutating-handler/
cronjobtrigger.kubeless.io/todo-mutating-handler created
function.kubeless.io/todo-mutating-handler configured
PS>
PS>kubectl get cronjobtriggers
NAME          AGE
todo-mutating-handler   8s
PS>
PS>kubectl get cronjobs
NAME           SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE
AGE
trigger-todo-mutating-handler  */1 * * * *  False     1        8s
13s
PS>
PS>sleep 90
PS>
PS>kubectl logs -l job-name --tail 2
* Connection #0 to host todo-mutating-handler.default.svc.cluster.local left intact
act
* Mutated...
PS>
PS>
```

This function updates every to-do item and replaces the text with a reminder for the reader...

Figure 21.11 This handler has some unusual behavior, but it shows what you can do with functions

Kubeless is a great way to get started with serverless and see if the functions-as-a-service model works for you. The focus on code makes Kubeless one of the better platforms for going serverless with Kubernetes, but the project hasn't been very active lately - partly because all the main features have been stable for a while. When you bring any open-source project into your organization you need to accept the risk that it might go stale and you'll need to spend your own engineering time helping to support it. You can mitigate that in this case by abstracting the serverless implementation using a generic project called Serverless.

21.4 Abstracting serverless functions with Serverless

You'll need to pay close attention to the capitalization in this section - *Serverless* is a project which standardizes function definitions and integrates with an underlying *serverless* platform to do the actual work. So you can deploy Serverless on top of Kubeless and use Serverless specs for your functions instead of using Kubeless directly. That means if you want to move away from

Kubeless at some point to Knative or OpenWhisk or Fn Project you can do it with a minimum amount of work, because Serverless supports those platforms too. Figure 21.12 shows the architecture of Serverless with Kubeless.

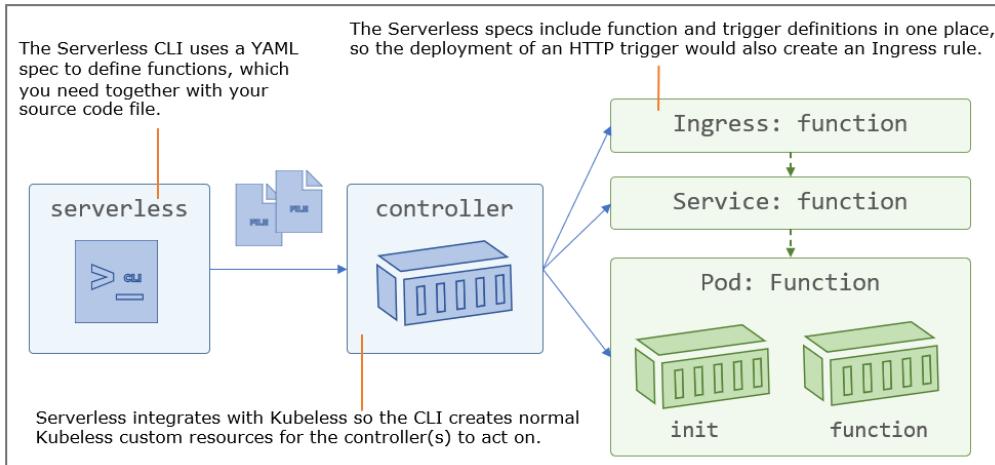


Figure 21.12 Serverless introduces its own specification language but uses an underlying serverless platform to run functions

Serverless isn't quite as clean as Kubeless because it adds an extra YAML specification for the function so you can't just bring your code file and set it running. On the plus side the specification is quite simple and it includes the function definition and the triggers in one place. Listing 21.4 shows the specification for the to-do API function. This file sits in the project folder along with the source code, and the code file itself is identical to the one you deployed with Kubeless in section 21.2.

Listing 21.4 - serverless.yml, a Serverless function spec

```
service: todo-api          # a service can group many functions
provider:
  name: kubeless          # the provider is the actual platform
  runtime: nodejs12         # you can use any runtime it supports

  hostname: api.todo.kiamol.local    # this is used for ingress rules

plugins:
  - serverless-kubeless
functions:                  # this is the function definition
  todo-api:
    description: 'ToDo list - create item API'
    handler: server.handler
    events:           # complete with the trigger events
      - http:
          path: /todos
```

The Serverless developer experience isn't quite as clean as Kubeless either. Serverless uses a command-line tool which is a Node.js package, and that means you need to install Node.js and then install Serverless which downloads a ton of dependencies. I've packaged the CLI in a container image so you don't need to do that, and in this section we'll replace the Kubeless functions with Serverless versions of the same code.

TRY IT NOW Remove the Kubeless functions and deploy them again using Serverless as the abstraction layer.

```
# delete the custom resources to remove functions and triggers:  
kubectl delete cronjobtriggers,natstriggers,httptriggers,functions --all  
  
# create a Pod with the Serverless CLI:  
kubectl apply -f serverless-cli.yaml  
  
# wait for it to be ready:  
kubectl wait --for=condition=ContainersReady pod serverless-cli  
  
# confirm the Serverless CLI is set up:  
kubectl exec serverless-cli -- serverless --version
```

The Serverless CLI uses *providers* to adapt the generic function spec into platform components. It effectively replaces the Kubeless CLI, using the Kubeless provider and a Kubernetes client library to create custom resources which the normal Kubeless controllers manage. Figure 21.13 shows the CLI is installed and running, but that's not quite all you need. The provider and the Kubernetes client library need to be installed in the project folder together with about 100 other dependencies.

Deleting the Kubeless custom resources removes all the Pods and Services.

```
PS>kubectl delete cronjobtriggers,natstriggers,httptriggers,functions --all
cronjobtrigger.kubeless.io "todo-mutating-handler" deleted
nattrigger.kubeless.io "todo-audit" deleted
function.kubeless.io "hello-kiamol" deleted
function.kubeless.io "todo-api" deleted
function.kubeless.io "todo-audit" deleted
function.kubeless.io "todo-mutating-handler" deleted
PS>
PS>kubectl apply -f serverless-cli.yaml
serviceaccount/serverless-cli created
pod/serverless-cli created
clusterrolebinding.rbac.authorization.k8s.io/serverless-cli created
PS>
PS>kubectl wait --for=condition=ContainersReady pod serverless-cli
pod/serverless-cli condition met
PS>
PS>kubectl exec -it serverless-cli -- serverless --version
Framework Core: 1.77.1
Plugin: 3.6.18
SDK: 2.3.1
Components: 2.33.0
```

The Serverless CLI needs Node.js and a lot of dependencies - running it in a Pod makes it easy to try it without all the installation steps.

Figure 21.13 Serverless provides an alternative deployment experience which will ultimately create the same function resources with Kubeless

So Serverless is not a simple project but it's extremely popular - it isn't only for serverless platforms running in Kubernetes either. You can use it as an abstraction layer for AWS Lambda and Azure Functions too. You can't just lift functions written for Kubeless and deploy them as Azure Functions because the platforms invoke methods in different ways with different parameters, but the core of the function code will be the same. Next we'll see how the deployment looks for the to-do API function using Serverless.

TRY IT NOW Create the to-do API function again with the same code file but using Serverless to define and deploy it.

```
# connect to a Serverless CLI session:
kubectl exec -it serverless-cli -- sh

# switch to the API code folder:
cd /kiamol/ch21/serverless/todo-api

# install all the deployment dependencies:
npm install

# and deploy the function:
serverless deploy

# list Kubeless functions:
kubectl get functions
```

```
# confirm the Pod has been created:  
kubectl get pods -l function=todo-api  
  
# and list HTTP triggers:  
kubectl get httptriggers
```

You can see in figure 21.14 that the net result of installing the Serverless function using Kubeless as the provider is the same as using the Kubeless CLI, or deploying custom resources directly. The setup stage you need to do for each project, but only the first time you deploy it or if you upgrade the provider, because Serverless is really just a client-side tool for deploying and managing functions.

Serverless is a client-side deployment tool - it runs on Node.js and uses libraries for the Kubeless provider and the Kubernetes API client. NPM is the Node.js package manager which installs the libraries.

```
PS>kubectl exec -it serverless-cli -- sh  
#  
# cd /kiamol/ch21/serverless/todo-api  
#  
# npm install  
added 113 packages from 165 contributors and audited 113 packages in 3.023s  
found 0 vulnerabilities  
  
# serverless deploy  
Serverless: Packaging service...  
Serverless: Excluding development dependencies...  
Serverless: Deploying function todo-api...  
Serverless: Function todo-api successfully deployed  
Serverless: Creating http trigger for: todo-api  
  
# kubectl get functions  
NAME      AGE  
todo-api  29s  
#  
# kubectl get pods -l function=todo-api  
NAME           READY   STATUS    RESTARTS   AGE  
todo-api-98f9ddc79-dsxrf  1/1     Running   0          36s  
#  
# kubectl get httptriggers  
NAME      AGE  
todo-api  3m14s
```

Deploying with Serverless creates a Kubeless function and HTTP trigger.

The custom resources are managed by the Kubeless controller in the usual way - the function controller creates a Pod to run the code.

Figure 21.14 Serverless is an abstraction over a serverless platform - this deployment creates a Kubeless function and trigger

We won't deploy the mutating function because I expect you got the message in the last section, but we'll go ahead and deploy the auditing function and then confirm everything still works as

expected. Serverless supports different event types to trigger functions and the audit function spec includes a queue trigger for the NATS new-item messages.

TRY IT NOW Still in your Serverless CLI session, switch to the folder for the auditing function and deploy it with the message queue trigger.

```
# switch to the function folder:  
cd /kiamol/ch21/serverless/todo-audit  
  
# deploy the Serverless Node.js dependencies:  
npm install  
  
# and the function itself:  
serverless deploy  
  
# confirm the function has been deployed:  
kubectl get functions  
  
# along with its trigger:  
kubectl get natstriggers  
  
# we're done with the CLI Pod now:  
exit
```

My output is in figure 21.15 where you see the end result is the same whether you use the Kubeless CLI, the Serverless CLI or Kubectl to apply custom resource specs. Those are all different abstractions around the serverless model, which itself is an abstraction over the standard Kubernetes application model.

You need to deploy the Serverless CLI dependencies in every function folder.

```
# cd /kiamol/ch21/serverless/todo-audit
#
# npm install
added 106 packages from 164 contributors and audited 106 packages in 2.751s
found 0 vulnerabilities

# serverless deploy
Serverless: Packaging service...
Serverless: Excluding development dependencies...
Serverless: Deploying function todo-audit...
Serverless: Function todo-audit successfully deployed
Serverless: Deploying NATS trigger: todo-audit-events-todo-newitem
#
# kubectl get functions
NAME      AGE
todo-api  32m
todo-audit 33s
#
# kubectl get natstriggers
NAME          AGE
todo-audit-events-todo-newitem  32s
```

The Serverless spec in this folder deploys the Python audit function and the NATS trigger. The output is the same as with the Kubeless CLI.

Figure 21.15 Functions with different types of trigger are defined in the Serverless YAML spec and deployed in the same way

One limitation with Serverless is that the CLI only operates within the context of a single function - you need to run commands in the function directory so the CLI can read the spec and find the function. You can group multiple functions in one folder and one Serverless spec, but they all have to use the same runtime so that's not an option for the polyglot functions in this section. In practice if you use Serverless you'll be mixing the Serverless CLI with Kubectl to get a full management experience. Now that the functions and the triggers are deployed we actually don't need to use Serverless at all to interact with them.

TRY IT NOW The HTTP trigger for the API function uses the same ingress rule and the NATS trigger for the audit function uses the same queue, so the end-to-end can be tested in the same way.

```
# if you're running in PowerShell, add a grep function to your session:
.\grep.ps1

# make a call to the API:
curl --data 'Sketch out volume III of the trilogy' http://api.todo.kiamol.local/todos

# print the latest logs from the message handler:
kubectl logs -l component=save-handler --tail 2
```

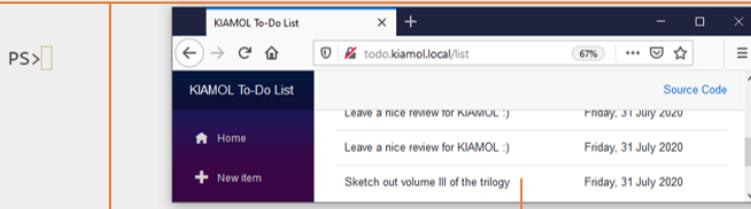
```
# and the audit function:  
kubectl logs -l function=todo-audit | grep AUDIT
```

It will be clear from that exercise and from figure 21.16 that Kubeless and Serverless layer on abstractions at the build and deploy stages, but they don't need to be used in the run stage. Functions can be delivered from single code files without a complex CI/CD pipeline, and even without a container image build. But the deployed components are just standard Kubernetes resources which you can manage in the usual way. If you have centralized logging and monitoring set up in your cluster then your serverless functions integrate with them in the same way as your other applications.

The Serverless spec for the API function uses the same host domain for the ingress rule:

```
PS>curl --data 'Sketch out volume III of the trilogy' http://api.todo.kiamol.local/todos  
PS>  
PS>kubectl logs -l component=save-handler --tail 2  
Saving item, added: 07/31/2020 15:39:55; event ID: 71f76433-5e52-4040-bc8e-2c367007b0b2  
Item saved; ID: 5; event ID: 71f76433-5e52-4040-bc8e-2c367007b0b2  
PS>  
PS>kubectl logs -l function=todo-audit | grep AUDIT
```

AUDIT @ 2020-07-31T15:39:55.539Z: Sketch out volume III of the trilogy



The new item message is recorded by the audit function.

And the item is shown in the web app in the usual list.

Figure 21.16 The functions deployed with Serverless behave in the same way with the Kubeless provider

That's as far as we'll go with serverless (and Serverless) - it's a useful architecture and it's good to have some experience using it with Kubernetes so you can understand how the transition from code to Pod works.

21.5 Understanding where serverless functions fit

The promise of a cloud-based serverless platform is that you bring your code and you forget about the operations side - deployment is simple and the platform keeps your code running and scales up and down with demand without any engagement from you. It's compelling for a lot of use cases but it has the major disadvantage that the serverless components of your app are

different from all the other components. The processes around serverless are minimal but they're still needed, and that leaves you with deployment pipelines, monitoring tools and troubleshooting workflows which are different from all your other apps.

That's where serverless on Kubernetes comes in. It's a compromise because you don't get the zero-ops promise of a cloud serverless platform and you don't get to scale at will because you need to balance compute resources with your other apps. But you've seen in this chapter that you do get the slick developer workflow and you do get to manage your serverless functions in the same way as your other Kubernetes deployments. It may not be something you bring into your cluster on day 2, but it's a powerful tool to have as an option.

Which just leaves the question - which tool, exactly? As of 2020 there are no serverless projects under the CNCF, and the options I've mentioned in this chapter are a mixture of open source and commercial projects with pretty varied adoption and activity rates. Kubeless and Knative are the major options right now and it's worth evaluating both - bearing in mind that the Serverless project is useful for isolating you from the underlying platform, at the cost of taking on more YAML specs. The CNCF ran a serverless working group which morphed into CloudEvents - a generic specification for events which trigger functions and the structure of their data, which is bringing standardization to serverless platforms.

And that brings us to the end, so all that's left is to tidy up and try the lab.

TRY IT NOW Remove the Kubeless components which will remove all the running functions, and then clean up the rest of the deployments.

```
kubectl delete -f kubeless/
kubectl delete ns,all,secret,configmap,pvc,ingress -l kiamol=ch21
```

21.6 Lab

I've mentioned Knative a few times in this chapter, and now it's your turn to try it out. Your task is to deploy a Knative version of the to-do list API which is available at the same URL you used in section 21.4. The Docker image is already built, but an experienced Kubernetes user like you won't need many hints - this is your chance to navigate the Knative documentation and see if you prefer the approach to Kubeless.

- the image for your API is called `kiamol/ch21-todo-api` - you can create it as a Knative function using the Knative CLI or with a custom resource definition
- the lab folder contains a Knative deployment and a new release of the to-do app
- this Knative setup uses a CNCF project called Contour as the ingress controller. The IP address to access your apps is in the `envoy` Service in the `contour-external` namespace.
- Knative uses the name of the Knative Service and the Kubernetes namespace to build the ingress domain name, so you'll need to be careful how you deploy your function.

Quite a few hints actually. This one may take some investigating but don't be disheartened - the solution is straightforward although there are a lot of stars to align to get the function running as you expect. My solution is on GitHub:

<https://github.com/sixeyed/kiamol/blob/master/ch21/lab/README.md>

22

Never the end

Books in the *Month of Lunches* series traditionally finish with a chapter called *Never the end*, to highlight that there's always more to learn. I think that's truer of Kubernetes than any other technology; this book turned out to be about 25% bigger than I originally intended and still there are topics I didn't even mention. I've tried to cover everything I consider important for you to have an authentic experience of using Kubernetes, and in this chapter I'll highlight some things I didn't have time for so you can go on to explore further. I've also got some guidance on choosing a Kubernetes platform and an introduction to the community.

22.1 Further reading by chapter

The main sources of further reading are the official Kubernetes website which contains documentation and guides, and the API Reference which has detailed descriptions of all the resources and their specifications. Those are good places to fill out some gaps from this book - and they're also the only resources you can use during the exams if you go for Kubernetes certification. As I planned each chapter there was a topic or two which I didn't have room for, and if you're looking to go further here are some of the things we skipped.

- Chapter 2 introduced the YAML spec we used in every other chapter, but I didn't cover *Kustomize*. That's a tool which simplifies YAML specs - you can have a base set of specs and a set of overlays which vary the configuration for different environments. It's powerful but simpler than the full templating experience in Helm.
- Chapter 3 introduced networking and the Service resource. A new extension called *Service Topology* gives you much finer control over load-balancing between Pods. You can specify that a network connection should use a Pod on the same node as the client, or a node in the same region to reduce network latency.
- Chapter 4 covered configuration in pretty exhaustive detail but I skipped *projected volumes*, which are a useful way of surfacing multiple ConfigMaps and Secrets into one

directory in the container filesystem. Good if you want to store configuration items independently for an app which needs to see them all in the same place.

- Chapter 5 missed out all the platform-specific volume types, which are the ones you really need to understand in production. The concepts are the same - you create a PVC and bind your Pod to it - but the underlying storage system makes a big difference. As an example the Azure Kubernetes Service supports volumes using Azure Files or Azure Disks. Files are easier to share between nodes, but Disks offer better performance.
- Chapter 6 showed how scaling works for Deployments and DaemonSets, running many replicas of the same Pod spec. Scaling up means running more Pods which go into the pending state to be allocated to a node by the scheduler. You can use the affinity and anti-affinity rules you learned in chapter 19 to control replica placement.
- Chapter 7 introduced multi-container Pods. There's a new type called *ephemeral containers* which is especially useful for debugging - it lets you run a temporary container inside an existing Pod. Ephemeral containers share the network space of the Pod and can share volumes and processes so they're great for investigating problems, particularly if your app container uses a minimal Docker image without tools like curl (which it should).
- Chapter 8 looked at creating a stable environment for apps with StatefulSets. In some cases you'll want consistent Pod naming with the startup order guarantee, so Pods can start in parallel. The *Pod management policy* in the spec allows for that.
- Chapter 9 walked through rollouts and rollbacks but you'll also want to understand *Pod disruption budgets* (PDBs). Those are separate objects which ensure a minimum number of Pods are kept available, which is an important safety control for cluster or deployment issues. PDBs take precedence over other actions, so you can't drain a node if removing its Pods would violate a PDB.
- Chapter 10 gave you an introduction to Helm with the main features covered, but an awful lot left out. The templating language in Helm supports control flow with if and else statements and loops. You can create named templates and segments and use them multiple times in your charts, which makes complex app designs or deployment options simpler to manage.
- Chapter 11 introduced some CI/CD patterns in Kubernetes but didn't mention *GitOps* which is a very attractive new approach where the cluster watches Git repositories to check for application releases. When a release is created the cluster deploys it automatically, so instead of an administrator having access to the cluster for app management, the cluster updates apps itself. Argo CD is the CNCF project for GitOps.
- Chapter 12 covered self-healing apps and resource limits but didn't get to *quality-of-service*, which is how Kubernetes provides service guarantees to Pods based on their resource specs. The classes are BestEffort, Burstable, and Guaranteed and you should understand how they work because they impact Pod eviction.
- Chapter 13 showed you how to collect, forward and store log entries from Pods. That includes Kubernetes components which run in Pods like the API server and DNS plugin, but you'll also want to collect logs from the Kubelet and the container runtime. How to

do that is platform-dependent but it's important get those logs into your centralized logging system too.

- Chapter 14 introduced Prometheus with a quick tour of exporters and client libraries. The Prometheus deployment was a simple one and you should take the time to understand the *Prometheus Operator*. It's a production-grade deployment which adds CRDs for Prometheus and Alertmanager instances and for scrape targets and alerting rules.
- Chapter 15 used Nginx and Traefik as examples of ingress controllers. You should add *Contour* to your list if you're evaluating options, it's a CNCF project which uses Envoy as the underlying proxy, a very fast and feature-rich technology.
- Chapter 16 looked at lots of security options but didn't include the *PodSecurityPolicy* resource, which lets you define rules Pods must adhere to before the cluster will run them. PodSecurityPolicies use an admission controller to block Pods which don't meet the policies you define and they're a powerful security control - but it's a new feature and isn't available in every platform.
- Chapter 17 on RBAC didn't cover *service account projection*, which is a way for Pod specs to request a custom token with a stated expiration time and audience for the JWT. That's useful if you only need a token for an init container, you can request one which will expire after initialization. You'll also want to understand how the Kubernetes API server audits requests, and there's a useful tool on GitHub called *audit2rbac* which can generate RBAC rules from an audit log.
- Chapter 18 gave you a flavor of installing and managing your own multi-node cluster, but there is much more to understand if you plan to do that for real. *Rancher Kubernetes Engine* (RKE) is an open-source product from Rancher which simplifies the installation and management of on-premises clusters. If you're looking at a hybrid environment you can have your on-prem clusters managed in the cloud using services like *Azure Arc*.
- Chapter 19 introduced the HorizontalPodAutoscaler but there are two other types of auto-scale. The *VerticalPodAutoscaler* is an add-on from the Kubernetes project which manages resources requests and limits based on actual usage, so Pods scale up or down and the cluster has an accurate resource view. *Cluster autoscaling* monitors the scheduler and if there are not enough compute resources to run pending Pods it adds a new node in the cluster. Cloud providers typically provide cluster autoscaling.
- Chapter 20 missed an alternative option for extending Kubernetes - *API aggregation*. CRDs add new resource types to the standard API, but the aggregation layer lets you plug whole new types of functionality into the API server. It's not commonly used, but it lets you add new capabilities to your cluster which are authenticated by Kubernetes.
- Chapter 21 introduced lots of serverless platform options, but there's another take on serverless which is about scaling down to zero when there's no work to be done. A CNCF project called *KEDA* (Kubernetes Event-driven Autoscaling) covers that. It monitors event sources like message queues or Prometheus metrics and scales existing applications up or down based on incoming load. It's a nice alternative to full serverless if you want the automatic management with your existing workflow.

And I didn't mention the Kubernetes *Dashboard*. That's a web UI you run in the cluster which shows you a graphical view of your workloads and their health. You can also deploy applications and edit existing resources with the Dashboard, so if you use it you need to be mindful of your RBAC rules and who gets access to the UI.

It's up to you how much more you feel you need to dig into those additional topics. They don't feature in the book because they're not core to your Kubernetes experience, or they use new features which don't have widespread adoption. However far you decide to go your next real task is to understand which Kubernetes platform you're going to use.

22.2 Choosing a Kubernetes platform

Kubernetes has a certification process for distributions and hosted platforms - if you're evaluating a platform you should start with the certified list, which you can find on the CNCF landscape website. Kubernetes is particularly well suited to cloud environments where you can spin up a cluster in minutes which is nicely integrated with other services from the provider. As a minimum you should expect this feature set, which you'll find in all the major clouds:

- integration with LoadBalancer Services so they're provisioned with a cloud load balancer which spans the cluster nodes and a public IP address;
- multiple storage classes with options like SMB and SSD so you can choose between IO performance and availability in your PersistentVolumeClaims;
- Secret storage so Secrets deployed to the cluster are actually created in a secure encrypted service, but are still accessible with the usual Secrets API;
- cluster autoscaling so you can set minimum and maximum cluster sizes and have nodes added or removed to meet demand as your Pods scale;
- multiple authentication options so end-users can access the cluster using their cloud credentials but you can also provision access for automation systems.

Your choice of cloud provider will probably be made for you if your organization has an existing relationship with one of the providers. Azure Kubernetes Service, Amazon's Elastic Kubernetes Service and Google Kubernetes Engine are all great options. If you do have a choice then you should also consider how quickly the platform rolls out new Kubernetes versions and just how managed the service is - are the nodes fully managed or do you need to roll out operating system patches?

When you settle on a hosted platform you're going to need to spend some time learning how the rest of the cloud features are integrated with Kubernetes. Annotations in object metadata are often used as a bridge to include configuration settings for the cloud service which are outside the standard Kubernetes model. It's a useful way of configuring your deployments so you get exactly what you want, but you need to be aware that any customizations reduce the portability of your apps.

The situation is slightly different for on-premises deployments. There is still the certification process and many products deliver a stock Kubernetes cluster with a bespoke management experience and a support team. Others wrap Kubernetes in their own tooling and modelling -

OpenShift does that in a big way, using its own resource definitions for runtime and network abstractions, and even using an alternative CLI. A wrapped distribution may offer features or processes you prefer to vanilla Kubernetes, which could make it a good choice as long as you're aware that it isn't just Kubernetes and it may be difficult to move your apps to another platform.

Running your own open-source Kubernetes cluster is definitely an option, one which is favored by startups or organizations with a large datacenter estate. If you're considering that route then you need to understand just how much you're taking on. Get everyone in your ops team a copy of this book and then ask them to go on to study and pass the Certified Kubernetes Administrator exam. If they're still keen on running a cluster then go for it, but be aware that their roles are likely to transition to full-time Kubernetes administrators. It's not just a case of administering the cluster, but also keeping up with the Kubernetes project itself.

22.3 Understanding how Kubernetes is built

Kubernetes has a quarterly release schedule so every three months new features are available, beta features graduate to general availability and old features get deprecated. Those are minor version updates - say from 1.17 to 1.18 - the major version is still 1 and there are no plans to jump to 2 anytime soon. The three most recent minor versions are supported by the project, which means you should plan to upgrade your cluster at least twice a year. There are patch releases whenever critical bugs or security issues require them; as an example the 1.18.6 release fixed an issue where containers could fill a node's disk without triggering eviction.

Releases are all on GitHub in the `kubernetes/kubernetes` repository and every release comes with an extensive changelog. Whether you're running your own cluster or using a hosted platform you need to know what's coming in a release before you upgrade because there are sometimes breaking changes. As features move through alpha and beta stages the API version in the spec changes, and eventually old versions are not supported. If you have Deployment specs that use the `apps/v1beta2` API versions you can't deploy them on Kubernetes from 1.16 - you need to change the spec to use the `apps/v1` version.

Special Interest Groups (SIGs) own different aspects of the Kubernetes project covering releases, technical topics and process - everything from authentication to contributor experience and Windows workloads. SIGs are public: they have regular online meetings which anyone can drop into, they have dedicated Slack channels and all their decisions are documented on GitHub. A technical SIG covers a wide area and is split into subprojects which own the design and delivery of one component of Kubernetes. Overall the Steering Committee owns the governance of the project and committee members are elected for a two-year term.

The open structure of Kubernetes drives innovation and quality. The governance model ensures individual organizations don't have an unfair representation so they can't steer roadmaps in directions that suit their products. Ultimately Kubernetes itself is stewarded by the CNCF which is chartered to promote the technology, protect the brand and serve the community.

22.4 Joining the community

That community includes you and me. If you want to see how the project evolves the best way to start is by joining Slack at kubernetes.slack.com (you'll find me there). Every SIG has its own channel so you can follow the areas you're interested in and there are general channels for release announcements, users and novices. There are meetup groups all over the world with regular events and the combined KubeCon and CloudNativeCon conferences are regional in Europe, America and Asia. All the Kubernetes code and documentation is open source and you can contribute on GitHub.

And that's all from me. Thank you for reading this book, and I hope it's helped you to become comfortable and confident with Kubernetes. It can be a career-changing technology and I hope everything you've learned here will help you on your journey.