```
$ kubectl get pod jump-pod -o yaml
apiVersion: v1
kind: Pod
metadata:
  name: jump-pod
  namespace: default
spec:
  containers:
  - image: nigelpoulton/curl:1.0
    imagePullPolicy: IfNotPresent
    name: jump-ctr
    stdin: true
    tty: true
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-2g29h
      readOnly: true
  dnsPolicy: ClusterFirst
```

THE KUBERNETES BOOK

2025 Edition

Nigel Poulton
& Pushkar Joglekar

The
Kubernetes
Book

2025 Edition

Weapons-grade Kubernetes learning!

Nigel Poulton @nigelpoulton

About this edition

This edition was published in February 2025.

In writing this edition, I've gone over every word in every chapter, ensuring everything is up-to-date with the latest trends and patterns in the industry. I've added new Kubernetes features, such as native sidecar containers, and I've updated the installation chapter to show you how to get the latest free Docker Desktop multi-node Kubernetes cluster. Finally, I've tested and updated all examples and hands-on tasks to work with Kubernetes v1.32. Enjoy the book, and get ready to master Kubernetes!

All typos are mine. Or should that be typo's… ;-)

Education is about inspiring and creating opportunities. I hope this book, and my video training courses, inspire you and create lots of opportunities!

A huge thanks to my family for putting up with me. I'm a geek who thinks he's software running on midrange biological hardware. I know it's not easy living with me.

Thanks to everyone who watches my Pluralsight and A Cloud Guru training videos. I love connecting with you and appreciate all the feedback I've had over the years. This feedback is what inspired me to write this book. I think you'll love it, and I hope it helps drive your career forward.

@nigelpoulton

About the authors

Nigel Poulton (@nigelpoulton)

Nigel is a technology geek who is passionate about learning new technologies and making them easier for others to learn. He's the author of best-selling books on Docker and Kubernetes, and is the author of **AI Explained: Facts, Fiction, and Future**, an exciting read into the impacts of AI on society and the future of humanity. .

Nigel is a Docker Captain and has held senior technology roles at large and small enterprises.

In his free time, he listens to audiobooks and coaches youth football (soccer). He wishes he lived in the future and could understand the mysteries of life and the universe. He's passionate about learning, cars, and football. He lives in England with his fabulous wife and three children.

Contributing author: Pushkar Joglekar

Pushkar contributed the technical content for Chapters 16 and 17.

Pushkar is a Sr. Cloud Security Engineer II, leading organization-wide security programs for a multi-billion dollar Fintech Company. He is also a Kubernetes Project Maintainer and was a Technical Security Lead for VMware Tanzu prior to his current role. Before that, he built multiple "secure by design" production container deployments for a Fortune 500 company and has been a frequent speaker at KubeCon.

When not securing containers, he explores neighborhood bike trails and captures beautiful sunsets through his camera while sipping homemade masala ginger chai. He lives with his wonderful wife, who is the real engineer among them.

Table of Contents

0: Preface

Kubernetes is developing fast, so I update the book every year. And when I say *update*, I mean real updates — I review every word and every concept, and test every example against the latest versions of Kubernetes. I'm 100% committed to ensuring this remains the best Kubernetes book in the world.

As an author, I'd love to write a book and never touch it again for five years. Unfortunately, a two-year-old book on Kubernetes could be dangerously out of date.

Paperbacks, hardbacks, eBooks, audio, and translations

This 2025 edition is published in all of the following formats:

- Paperback
- Hardback
- eBook

eBook copies are available on Kindle, Leanpub, and various other sources. I recommend Leanpub, as I publish there first, and you get free lifetime updates that work.

Various translations are also available but sometimes lag behind the English language editions.

I've also created the following *collector's editions.* Each has a themed front cover, but the content is exactly the same as the regular English-language edition.

- Starfleet paperback
- Borg hardback

- Klingon paperback

Kindle updates

Unfortunately, Kindle readers cannot get updates — even if you delete the book and buy it again, you'll still get the older version you originally purchased. I have no control over this and was devastated when Amazon introduced this change. Feel free to contact me at **tkb@nigelpoulton.com** and I'll do my best to help.

The sample app and GitHub repo

There's a GitHub repo with all the app files and YAML code used throughout the book.

To follow the examples, you'll need to install **git**, clone the repo, and switch to the 2025 branch. And don't worry if that sounds confusing. You don't need to be a GitHub expert, as I'll always show you the commands you need to run.

Install **got** onto your computer. The easiest way is to search the web for **git** installation instructions and follow them for your platform (Linux/Mac/Windows…).

Once you've installed **git**, you can clone the repo and switch branches.

Run the following command to clone the repo. It creates a new directory called `TKB` and copies all the files and folders.

```
$ git clone https://github.com/nigelpoulton/TKB.git
```

Change into the new `TKB` directory and switch to the 2025 branch.

```
$ cd TKB

$ git fetch origin
```

```
$ git checkout -b 2025 origin/2025
```

Congratulations. You've cloned the book'sGitHub repo and switched to the `2025` branch.

Windows users

Most of the commands in the hands-on sections work on Linux, Mac, and Windows. However, a few of them require minor changes to work on Windows. Whenever this happens, I explain what you need to do.

However, to prevent myself from repeating the same thing too often, I don't always tell Windows users to replace backslashes with backticks for line breaks. With this in mind, Windows users should do one of the following every time the book splits a command over multiple lines using backslashes:

- Remove the backslash and run the command on a single line
- Replace the backslashes with backticks

All other differences are explained in full every time.

Terminology and responsible language

Throughout the book, I capitalize the first letter of Kubernetes objects such as Pods and Services. This helps you know when I'm talking about a Kubernetes "LoadBalancer" and not a cloud "load balancer".

The book also follows guidelines from the [Inclusive Naming Initiative](), which promotes responsible language.

Feedback

If you like the book and it helps your career, share the love by recommending it to a friend and leaving a review on Amazon, Goodreads, or wherever you buy your books.

For other feedback, you can email me at **tkb@nigelpoulton.com** or reach me on any of the following.

1: Kubernetes primer

This chapter gets you up-to-speed with the basics and background of Kubernetes, and I've divided it as follows:

- Important Kubernetes background
- Kubernetes: the Operating System of the cloud

Important Kubernetes background

Kubernetes is an orchestrator of containerized cloud-native microservices applications.

That's a lot of jargon, so let's explain it.

Orchestration

An *orchestrator* is a system that deploys applications and dynamically responds to changes. For example, Kubernetes can:

- Deploy applications
- Scale them up and down based on demand
- Self-heal them when things break
- Perform rollouts and rollbacks
- Lots more

The best part is that it does all of this without **you** having to get involved. You need to configure a few things up front, but once you've done that, you sit back and let Kubernetes work its magic.

Containerization

Containerization is the process of packaging applications and dependencies as images and then running them as containers.

It can be useful to think of containers as the next generation of virtual machines (VM). Both are ways of packaging and running applications, but containers are smaller, faster, and more portable.

Despite these advantages, containers haven't entirely replaced VMs, and you'll see them running side-by-side in most environments. But containers are the first-choice solution for most new applications.

Cloud native

Cloud-native applications possess cloud-like features such as *auto-scaling, self-healing, automated updates, rollbacks,* and more.

Simply running a regular application in the public cloud **does not** make it *cloud-native.*

Microservices

Microservices applications are built from many small, specialized, independent parts that work together to form a useful application.

Consider an e-commerce app with the following six features:

- Web front-end
- Catalog
- Shopping cart
- Authentication
- Logging
- Store

To make this a *microservices app*, you design, develop, deploy, and manage each feature as its own small application that we call a *microservice.* As such, this application has six microservices.

Designing like this brings huge flexibility by allowing all six microservices to have their own small development teams and their own release cycles. It also lets you scale and update each one independently.

The most common pattern is to deploy each microservice as its own container. This means one or more web front-end containers, one or more catalog containers, one or more shopping cart containers, etc. Scaling any part of the app is as simple as adding or removing containers.

Now that we've explained a few things let's revisit and rewrite that jargon-filled sentence from the start of the chapter.

The original sentence read; *"Kubernetes is an orchestrator of containerized cloud-native microservices applications."* We now know this means: *Kubernetes deploys, scales, self-heals, and updates applications where individual application features are packaged and deployed as containers.*

Hopefully, that's clarified some of the jargon. But don't worry if some of it still feels fuzzy, we'll cover everything again in more detail throughout the book.

Where did Kubernetes come from

Kubernetes was developed by a group of Google engineers partly in response to Amazon Web Services (AWS) and Docker.

AWS changed the world when it invented modern cloud computing, and the rest of the industry needed to catch up.

One of the companies catching up was Google. They'd built their own cloud but needed a way to abstract the value of AWS **and** make it as easy as possible for customers to get off AWS and onto their cloud. They also ran their own production apps, such as *Search* and *Gmail*, on billions of containers per week.

At the same time, Docker was taking the world by storm, and users needed help managing explosive container growth.

This led a group of Google engineers to take the lessons they'd learned using their internal container management tools and create a new tool called ***Kubernetes***. In 2014, they open-sourced Kubernetes and donated it to the newly formed *Cloud Native Computing Foundation (CNCF)*.

At the time of writing, Kubernetes is over 10 years old and has experienced incredible growth and adoption. However, at its core, it still does the two things Google and the rest of the industry need:

1. It abstracts infrastructure (such as AWS)
2. It simplifies applications portability

These are two of the biggest reasons Kubernetes is important to the industry.

Kubernetes and Docker

All the early versions of Kubernetes shipped with Docker as its container runtime. This means Kubernetes used Docker for low-

level tasks such as creating, starting, and stopping containers. However, two things happened:

1. Docker got bloated
2. People created lots of Docker alternatives

As a result, the Kubernetes project created the *container runtime interface (CRI)* to make the runtime layer *pluggable*. This means you can now pick and choose the best runtimes for your needs. For example, some runtimes provide better isolation, some provide better performance, some work with Wasm containers, and more.

Kubernetes 1.24 finally removed support for Docker as a runtime as it was bloated and overkill for what Kubernetes needed. Since then, most new Kubernetes clusters ship with **containerd** (pronounced *"container dee"*) as the default runtime. Fortunately, containerd is a stripped-down version of Docker optimized for Kubernetes, that fully supports applications containerized by Docker. In fact, Docker, containerd, and Kubernetes all work with images and containers that implement the *Open Container Initiative (OCI)* standards.

Figure 1.2 shows a four-node Kubernetes cluster running multiple container runtimes.
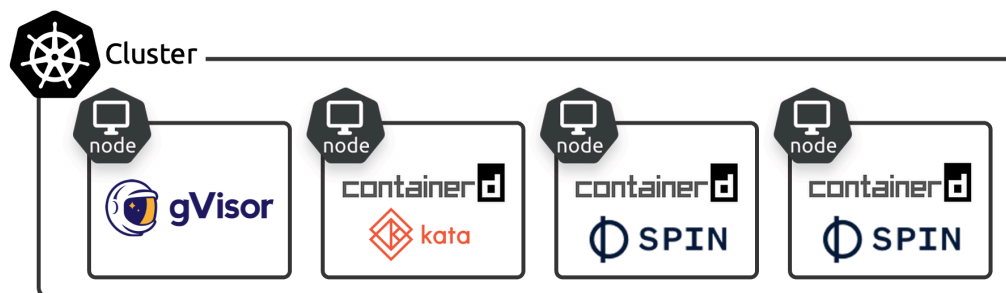
Figure 1.2 - Single cluster with multiple runtimes

Notice how some of the nodes have multiple runtimes. Configurations like this are fully supported and increasingly

common. You'll work with a configuration like this in Chapter 9 when you deploy a Wasm (WebAssembly) app to Kubernetes.

What about Docker Swarm

In 2016 and 2017, Docker Swarm, Mesosphere DCOS, and Kubernetes competed to become the industry standard container orchestrator. Kubernetes won.

However, Docker Swarm is still being developed and is still used by small companies needing a simple alternative to Kubernetes.

Kubernetes and Borg: Resistance is futile!

We already said that Google has been running containers at massive scale for a very long time. Well, they had two in-house tools called *Borg* and *Omega* orchestrating these billions of containers. So, it's easy to make the connection with Kubernetes — all three orchestrate containers at scale, and all three are related to Google.

However, Kubernetes is **not** an open-source version of Borg or Omega. It's more like Kubernetes shares its DNA and family history with them.
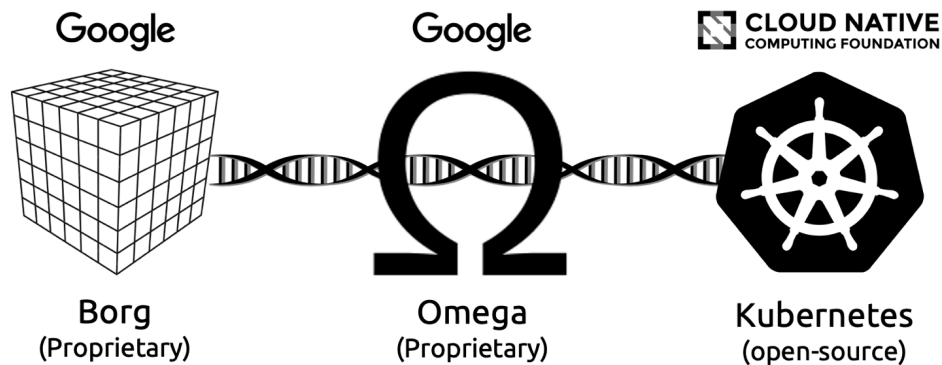
Figure 1.3 - Shared DNA

As things stand, Kubernetes is an open-source project owned by the CNCF. It's licensed under the Apache 2.0 license, version 1.0 shipped way back in July 2015, and at the time of writing, we're already at version 1.32 and averaging three new releases per year.

Kubernetes — what's in the name

Most people pronounce Kubernetes as *"koo-ber-net-eez"*, but the community is very friendly, and people won't mind if you pronounce it differently.

The word Kubernetes originates from the Greek word for *helmsman* which is the person who steers a ship. You can see this in the logo, which is a ship's wheel.

Figure 1.4 - The Kubernetes logo

Some of the original engineers wanted to call Kubernetes *Seven of Nine.* This is because Google's *Borg* project inspired Kubernetes, and a famous *Borg* drone from the TV series Star Trek Voyager is called *Seven of Nine.* Copyright laws didn't allow this, so they gave the logo *seven* spokes as a subtle reference to <u>Seven</u> of Nine.

One last thing about the name. You'll often see it shortened to *K8s* and pronounced as "kates". The number 8 replaces the eight characters between the "K" and the "s".

Kubernetes: the operating system of the cloud

Kubernetes is the de facto platform for cloud-native applications, and we sometimes call it *the operating system (OS) of the cloud.* This is because it abstracts the differences between cloud platforms the same way that operating systems like Linux and Windows abstract the differences between servers:

- Linux and Windows *abstract* server resources and *schedule* application processes
- Kubernetes *abstracts* cloud resources and *schedules* application microservices

As a quick example, you can schedule applications on Kubernetes without caring if they're running on AWS, Azure, Civo Cloud, GCP, or your on-premises data center. This makes Kubernetes a key enabler for:

- Hybrid cloud
- Multi-cloud
- Cloud migrations

In summary, Kubernetes makes it easier to deploy to one cloud today and migrate to another cloud tomorrow.

Application scheduling

One of the main things an OS does is simplify the scheduling of work tasks.

For example, computers are complex collections of hardware resources such as CPU, memory, storage, and networking. Thankfully, modern operating systems hide most of this, making the world of application development a far friendlier place. For example, how many developers need to care which CPU core, memory DIMM, or flash chip their code uses? Most of the time, we let the OS decide.

Kubernetes does a similar thing with clouds and data centers.

At a high level, a cloud or data center is a complex collection of resources and services. Kubernetes abstracts most of this, making the resources easier to consume. Again, how often do you care which compute node, which failure zone, or which storage volume your app uses? Most of the time, you'll be happy to let Kubernetes decide.

Chapter summary

Kubernetes was created by Google engineers based on lessons learned running containers at hyper-scale for many years. They donated it to the community as an open-source project and it is now the industry standard platform for deploying and managing cloud-native applications. It runs on any cloud or on-premises data center and abstracts the underlying infrastructure. This allows you to build hybrid clouds, as well as migrate on, off, and between different clouds. It's open-sourced under the Apache 2.0 license and is owned and managed by the Cloud Native Computing Foundation (CNCF).

Don't be afraid of all the new terminology. I'm here to help, and you can reach me at any of the following:

- LinkedIn: linkedin.com/in/nigelpoulton/
- BlueSky: @nigelpoulton.com
- X: @nigelpoulton
- Web: nigelpoulton.com
- Email: tkb@nigelpoulton.com

2: Kubernetes principles of operation

This chapter introduces you to the major Kubernetes components and prepares you for upcoming chapters. This chapter won't make you an expert, but it will lay important foundations you'll build on throughout the book.

We'll cover all of the following:

- Kubernetes from 40K feet
- Control plane nodes and worker nodes
- Packaging apps for Kubernetes
- The declarative model and desired state
- Pods
- Deployments
- Services

Kubernetes from 40K feet

Kubernetes is both of the following:

- A cluster
- An orchestrator

Kubernetes: Cluster

A *Kubernetes cluster* is one or more *nodes* providing CPU, memory, and other resources for application use.

Kubernetes supports two node types:

- Control plane nodes

- Worker nodes

Both types can be physical servers, virtual machines, or cloud instances, and both can run on ARM and AMD64/x86-64. Control plane nodes must be Linux, but worker nodes can be Linux or Windows.

Control plane nodes implement the Kubernetes intelligence, and every cluster needs at least one. However, you should have three or five for high availability (HA).

Every control plane node runs every control plane service. These include the API server, the scheduler, and the controllers that implement cloud-native features such as self-healing, autoscaling, and rollouts.

Worker nodes are where you run your business applications.

Figure 2.1 shows a cluster with three control plane nodes and three workers.
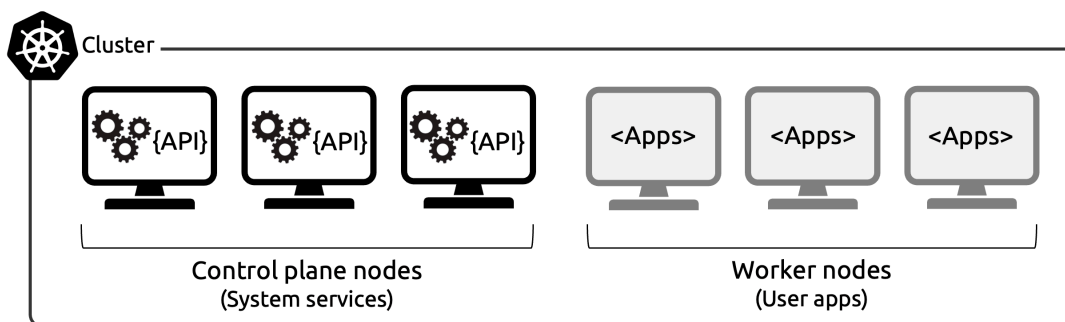
Figure 2.1

It's common to run user applications on control plane nodes in development and test environments. However, many production environments restrict user applications to worker nodes so that control plane nodes can focus their resources on cluster operations.

Doing this allows control plane nodes to focus on managing the cluster.

Kubernetes: Orchestrator

Orchestrator is jargon for a system that deploys and manages applications.

Kubernetes is the industry-standard orchestrator and can intelligently deploy applications across nodes and failure zones for optimal performance and availability. It can also fix things when they break, scale things when demand changes, and manage rollouts and rollbacks.

That's the big picture. Let's dig a bit deeper.

Control plane and worker nodes

We already said a Kubernetes cluster is one or more *control plane nodes* and *worker nodes*.

Control plane nodes must be Linux, but workers can be Linux or Windows.

Almost all cloud-native apps are Linux apps and require Linux worker nodes. However, you'll need one or more Windows worker nodes if you have cloud-native Windows apps. Fortunately, a single Kubernetes cluster can have a mix of Linux and Windows worker nodes, and Kubernetes is intelligent enough to schedule apps to the correct nodes.

The control plane

The *control plane* is a collection of system services that implement the brains of Kubernetes. It exposes the API, schedules apps,

implements self-healing, manages scaling operations, and more.

The simplest clusters run a single control plane node and are best suited for labs and testing. For production clusters, you should run three or five control plane nodes and spread them across availability zones for high availability, as shown in Figure 2.2
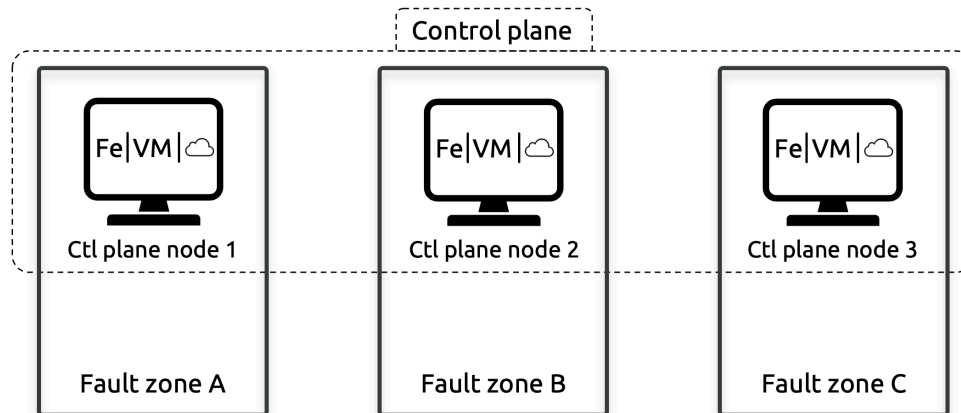
Figure 2.2 Control plane high availability

As previously mentioned, it's usually a production best practice to run all user apps on worker nodes, allowing control plane nodes to allocate their resources to cluster-related operations.

Most clusters run every control plane service on every control plane node for HA.

Let's take a closer look at the major control plane services.

The API server

The *API server* is the front end of Kubernetes, and all commands and requests go through it. Even internal control plane services communicate with each other via the API server.

It exposes a RESTful API over HTTPS, and all requests are subject to authentication and authorization. For example, deploying or

updating an app follows this process:

1. Describe the application in a YAML configuration file
2. **Post the configuration file to the API server**
3. The request will be authenticated and authorized
4. The application definition will be persisted in the cluster store
5. The application's containers will be scheduled to nodes in the cluster

The cluster store

The cluster store holds the desired state of all applications and cluster components, and it's the only *stateful* part of the control plane.

It's based on the *etcd* distributed database, and most Kubernetes clusters run an etcd replica on every control plane node for HA. However, large clusters that experience a high rate of change may run a separate etcd cluster for better performance.

Be aware that a highly available cluster store is not a substitute for backup and recovery. You still need adequate ways to recover the cluster store when things go wrong.

Regarding *availability*, etcd prefers an odd number of replicas to help avoid *split brain* conditions. This is where replicas experience communication issues and cannot be sure if they have a quorum (majority).

Figure 2.3 shows two etcd configurations experiencing a *network partition* error. Cluster A on the left has four nodes and is experiencing a split brain with two nodes on either side and neither having a majority. Cluster B on the right only has three nodes but is not experiencing a split-brain as **Node A** knows it does not have a majority, whereas **Node B** and **Node C** know they do.
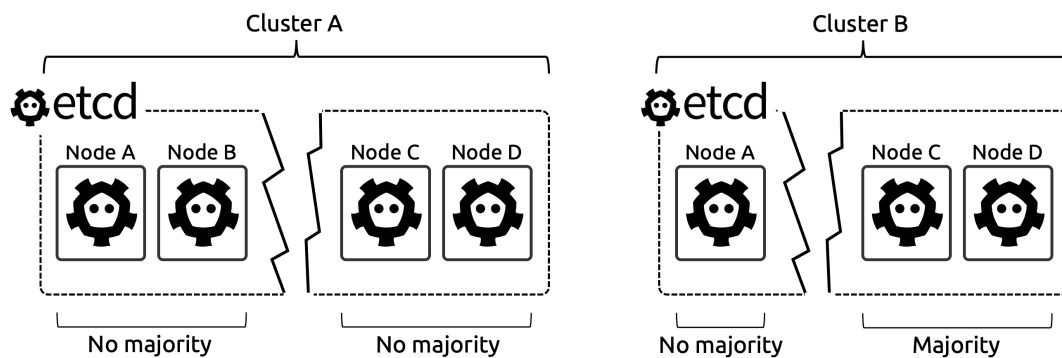
Figure 2.3. HA and split-brain conditions

If a split-brain occurs, etcd goes into read-only mode preventing updates to the cluster. User applications will continue working, but Kubernetes won't be able to scale or update them.

As with all distributed databases, consistency of writes is vital. For example, multiple writes from different sources to the same can cause corruption. etcd uses the *RAFT* consensus algorithm to prevent this from happening.

Controllers and the controller manager

Kubernetes uses *controllers* to implement a lot of the cluster intelligence. Each controller runs as a process on the control plane, and some of the more common ones include:

- The Deployment controller
- The StatefulSet controller
- The ReplicaSet controller

Lots of others exist, and we'll cover some of them later in the book. However, they all run as background watch loops, reconciling observed state with desired state.

That's a lot of jargon, and we'll cover it in detail later in the chapter. But for now, it means controllers ensure the cluster runs what you

asked it to run. For example, if you ask for three replicas of an app, a controller will ensure you have three healthy replicas and take appropriate actions if you don't.

Kubernetes also runs a *controller manager* that is responsible for spawning and managing the individual controllers.

Figure 2.4 gives a high-level overview of the controller manager and controllers.
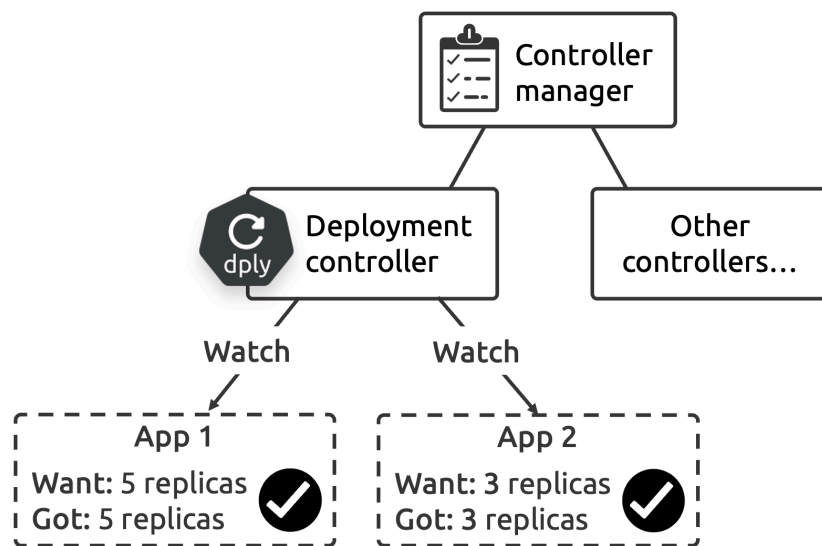
Figure 2.4. Controller manager and controllers

The scheduler

The *scheduler* watches the API server for new work tasks and assigns them to healthy worker nodes.

It implements the following process:

1. Watch the API server for new tasks
2. Identify capable nodes
3. Assign tasks to nodes

Identifying capable nodes involves predicate checks, filtering, and a ranking algorithm. It checks for taints, affinity and anti-affinity rules, network port availability, and available CPU and memory. It ignores nodes incapable of running the tasks and ranks the remaining ones according to factors such as whether it already has the required image, the amount of available CPU and memory, and number of tasks it's currently running. Each is worth points, and the nodes with the most points are selected to run the tasks.

The scheduler marks tasks as pending if it can't find a suitable node.

If the cluster is configured for *node autoscaling*, the pending task kicks off a cluster autoscaling event that adds a new node to the cluster and the scheduler assigns the task to the new node.

The cloud controller manager

If your cluster is on a public cloud, such as AWS, Azure, GCP, or Civo Cloud, it will run a *cloud controller manager* that integrates the cluster with cloud services, such as instances, load balancers, and storage. For example, if you're on a cloud and an application requests a load balancer, the cloud controller manager provisions one of the cloud's load balancers and connects it to your app.

Control Plane summary

The control plane implements the brains of Kubernetes, including the API Server, the scheduler, and the cluster store. It also implements controllers that ensure the cluster runs what you asked it to run.

Figure 2.5 shows a high-level view of a Kubernetes control plane node.
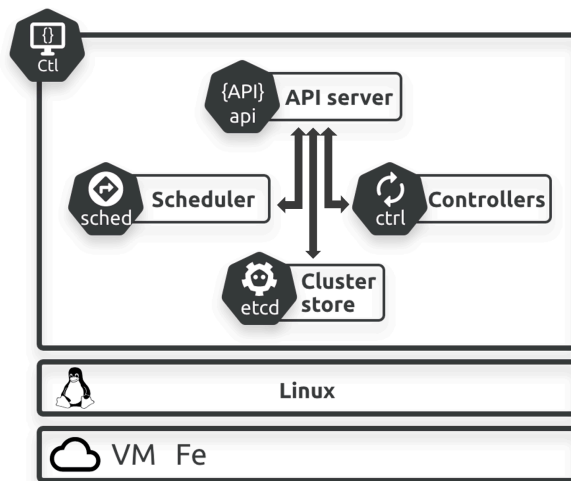
Figure 2.5 - Control plane node

You should run three or five control plane nodes for high availability, and large busy clusters might run a separate etcd cluster for better cluster store performance.

The API server is the Kubernetes frontend, and **all** communication passes through it.

Worker nodes

Worker nodes run your business applications and look like Figure 2.6.
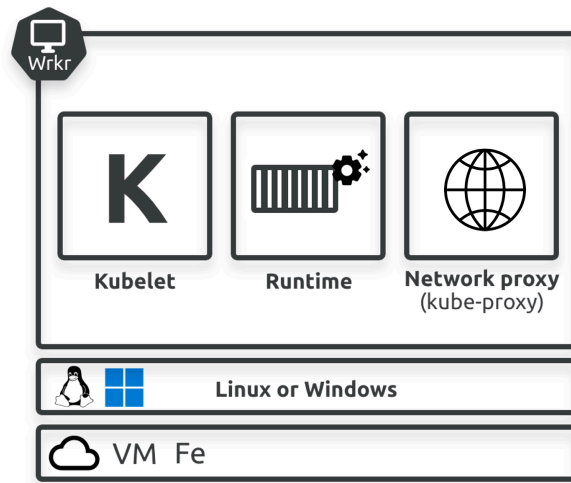
Figure 2.6 - Worker node

Let's look at the major worker node components.

Kubelet

The *kubelet* is the main Kubernetes agent and handles all communication with the cluster.

It performs the following key tasks:

- Watches the API server for new tasks
- Instructs the appropriate runtime to execute tasks
- Reports task status to the API server

If a task won't run, the kubelet reports the problem to the API server and lets the control plane decide what actions to take.

Runtime

Every worker node has one or more *runtimes* for executing tasks.

Most new Kubernetes clusters pre-install the **containerd** runtime and use it to execute tasks. These tasks include:

- Pulling container images
- Managing lifecycle operations such as starting and stopping containers

Older clusters shipped with the Docker runtime, but this is no longer supported. RedHat OpenShift clusters use the CRI-O runtime. Lots of others exist, and each has its pros and cons.

We'll use some different runtimes in the Wasm chapter.

Kube-proxy

Every worker node runs a *kube-proxy* service that implements cluster networking and load balances traffic to tasks running on the node.

Now that you understand the fundamentals of the control plane and worker nodes, let's switch gears and see how to package applications so they'll run on Kubernetes.

Packaging apps for Kubernetes

Kubernetes runs containers, VMs, Wasm apps, and more. However, all of them need wrapping in Pods before they'll run on Kubernetes.

We'll cover Pods shortly, but for now, think of them as a thin wrapper that abstracts different types of tasks so they can run on Kubernetes. The following courier analogy might help.

Couriers allow you to ship books, clothes, food, electrical items, and more, so long as you use their approved packaging and labels. Once you've packaged and labeled your goods, you hand them to the courier for delivery. The courier then handles the complex logistics of which planes and trucks to use, secure hand-offs to local delivery hubs, and eventual delivery to customers. They also provide services for tracking packages, changing delivery details, and

attesting successful delivery. All **you** have to do is package and label the goods.

Running applications on Kubernetes is similar. Kubernetes can run containers, VMs, Wasm apps, and more, as long as you wrap them in Pods. Once wrapped in a Pod, you give the Pod to Kubernetes, and Kubernetes runs it. This includes the complex logistics of choosing appropriate nodes, joining networks, attaching volumes, and more. Kubernetes even lets you query apps and make changes.

Consider a quick example.

You write an app in your favorite language, containerize it, push it to a registry, and wrap it in a Pod. At this point, you can give the Pod to Kubernetes, and Kubernetes will run it. However, you'll almost always deploy and manage Pods via higher-level controllers. For example, you can wrap Pods inside of *Deployments* for scaling, self-healing, and rollouts.

Don't worry about the details yet, we'll cover everything in more depth and with lots of examples later in the book. Right now, you only need to know two things:

1. Apps need to be wrapped in Pods to run on Kubernetes
2. Pods get wrapped in higher-level controllers for advanced features

Let's quickly go back to the courier analogy to help explain the role of controllers.

Most couriers offer additional services such as insurance for the goods you're shipping, refrigerated delivery, signature and photographic proof of delivery, express delivery services, and more.

Again, Kubernetes is similar. It implements controllers that add value, such as ensuring the health of apps, automatically scaling when demand increases, and more.

Figure 2.7 shows a container wrapped in a Pod, which, in turn, is wrapped in a Deployment. Don't worry about the YAML configuration yet, it's just there to seed the idea.


```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  replicas: 10
  ...
  strategy:
    type: RollingUpdate
    rollingUpdate:
  ...
  template:
    spec:
      containers:
      - name: hello-world
        image: …/k8sbook:1.0
        ports:
        - containerPort: 8080
```
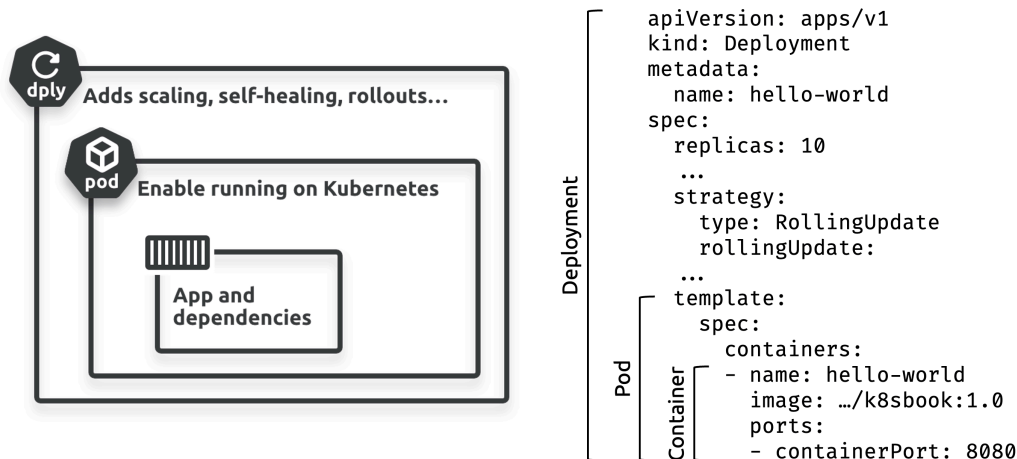
Figure 2.7 - Object nesting

The important thing to understand is that each layer of wrapping adds something:

- The container wraps the app and provides dependencies
- The Pod wraps the container so it can run on Kubernetes
- The Deployment wraps the Pod and adds self-healing, scaling, and more

You post the Deployment (YAML file) to the API server as the *desired state* of the application, and Kubernetes implements it.

Speaking of desired state…

The declarative model and desired state

The *declarative model* and *desired state* are at the core of how Kubernetes operates. They work on three basic principles:

- Desired state
- Observed state
- Reconciliation

Desired state is what you want, *observed state* is what you have, and *reconciliation* is the process of keeping observed state in sync with desired state.

> **Terminology:** We use the terms *actual state, current state*, and *observed state* to mean the same thing — the most up-to-date view of the cluster.

In Kubernetes, the declarative model works like this:

1. You describe the desired state of an application in a YAML manifest file
2. You post the YAML file to the API server
3. Kubernetes records this in the cluster store as a record of intent
4. A controller notices the observed state of the cluster doesn't match the new desired state
5. The controller makes the necessary changes to reconcile the differences
6. The controller keeps running in the background, ensuring observed state always matches desired state

Let's have a closer look.

You write manifest files in YAML that tell Kubernetes what you want an application to look like. We call this desired state, which includes which images to use, how many replicas, which network ports, and more.

Once you've created the manifest, you post it to the API server where it's authenticated and authorized. The most common way of posting YAML files to Kubernetes is with the `kubectl` command-line utility.

Once authenticated and authorized, Kubernetes persists the configuration to the cluster store as a record of intent.

At this point, the observed state of the cluster doesn't match your new desired state. A controller will notice this and begin the process of reconciliation. This will involve making all the changes described in the YAML file and is likely to include scheduling new Pods, pulling images, starting containers, attaching them to networks, and starting application processes.

Once the reconciliation completes, observed state will match desired state, and everything will be OK. However, the controllers keep running in the background, ready to reconcile any future differences.

It's important to understand that what we've described is very different from the traditional *imperative model*:

- The *imperative model* requires complex scripts of platform-specific commands to achieve an end-state
- The *declarative model* is a simple platform-agnostic way of *describing* an end state

Kubernetes supports both but prefers the *declarative model*. This is because the declarative model integrates with version control systems and enables self-healing, autoscaling, and rolling updates.

Consider a couple of simple declarative examples.

Assume you've deployed an app from a YAML file requesting ten replicas. If a node running two of the replicas fails, the observed state will drop to 8 replicas and no longer match the desired state of 10. That's OK, a controller will see the difference and schedule 2 new replicas to bring the total back up to 10.

The same will happen for an app update. For example, if you update the YAML, telling the app to use a newer version of the

image and post the change to Kubernetes, the relevant controller will notice the difference and replace the replicas running the old version with new replicas running the new version.

If you try to perform an update like this imperatively, you'll need to write complex scripts to manage, monitor, and heath-check the entire update process. To do it declaratively, you only need to change a single line of YAML, and Kubernetes does everything else.

Despite its simplicity, this is extremely powerful, and it's fundamental to the way Kubernetes works.

Pods

The atomic unit of scheduling in VMware is the virtual machine (VM). In Kubernetes, it's the *Pod*.

Yes, Kubernetes runs containers, VMs, Wasm apps, and more. But they all need wrapping in Pods.

Pods and containers

The simplest configurations run a single container per Pod, which is why we sometimes use the terms *Pod* and *container* interchangeably. However, there are powerful use cases for multi-container Pods, including:

- Service meshes
- Helper services that initialize environments
- Apps with tightly coupled helper functions such as log scrapers

Figure 2.8 shows a multi-container Pod with a main application container and a service mesh *sidecar*. Sidecar is jargon for a helper container that runs in the same Pod as the main app container and provides additional services. In Figure 2.8, the service mesh sidecar encrypts network traffic and provides telemetry.
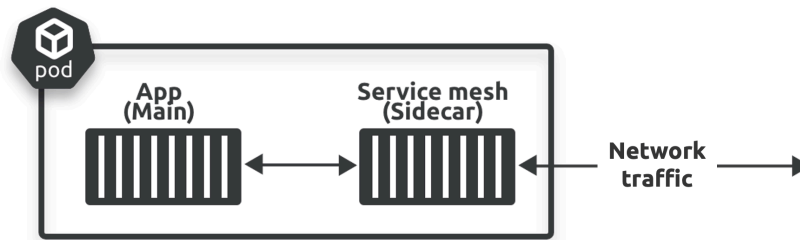
Figure 2.8 - Multi-container service mesh Pod

Multi-container Pods also help us implement the *single responsibility principle* where every container performs a single task. In Figure 2.8, the main app container might be serving a message queue or some other core application feature. Instead of adding the encryption and telemetry logic into the main app, we keep the app simple and implement the additional services in the service mesh container in the same Pod.

Pod anatomy

Each Pod is a shared execution environment for one or more containers. The *execution environment* includes a network stack, volumes, shared memory, and more.

Containers in a *single-container Pod* have the execution environment to themselves, whereas containers in a *multi-container Pod* share it.

As an example, Figure 2.9 shows a multi-container Pod with both containers sharing the Pods IP address. The main application container is accessible outside the Pod on `10.0.10.15:8080`, and the sidecar on `10.0.10.15:5005`. If they need to communicate with each other, container-to-container within the Pod, they can use the Pod's `localhost` interface.
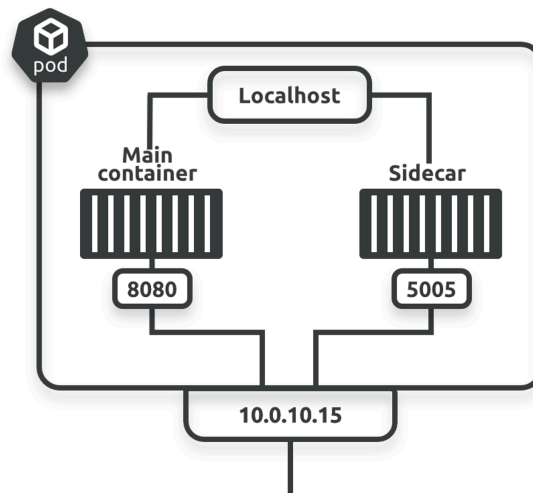
Figure 2.9 - Multi-container Pod sharing Pod IP

You should choose a multi-container Pod when your application has tightly coupled components needing to share resources such as memory or storage. In most other cases, you should use single-container Pods and loosely couple them over the network.

Pod scheduling

Kubernetes always schedules containers in the same Pod to a single node. This is because Kubernetes schedules Pods, not individual containers. But it's also because Pods are a shared execution environment, and you can't easily share memory, networking, and volumes across different nodes.

Starting a Pod is also an *atomic operation.* This means Kubernetes only marks a Pod as ready when all its containers are running. For example, if a Pod has two containers and only one is started, the Pod is not ready.

Pods as the unit of scaling

Pods are the minimum unit of scheduling in Kubernetes. As such, scaling an application **up** adds more Pods and scaling it **down** deletes Pods. You **do not** scale by adding more containers to existing Pods. Figure 2.10 shows how to scale the **web-fe** microservice using Pods as the unit of scaling.
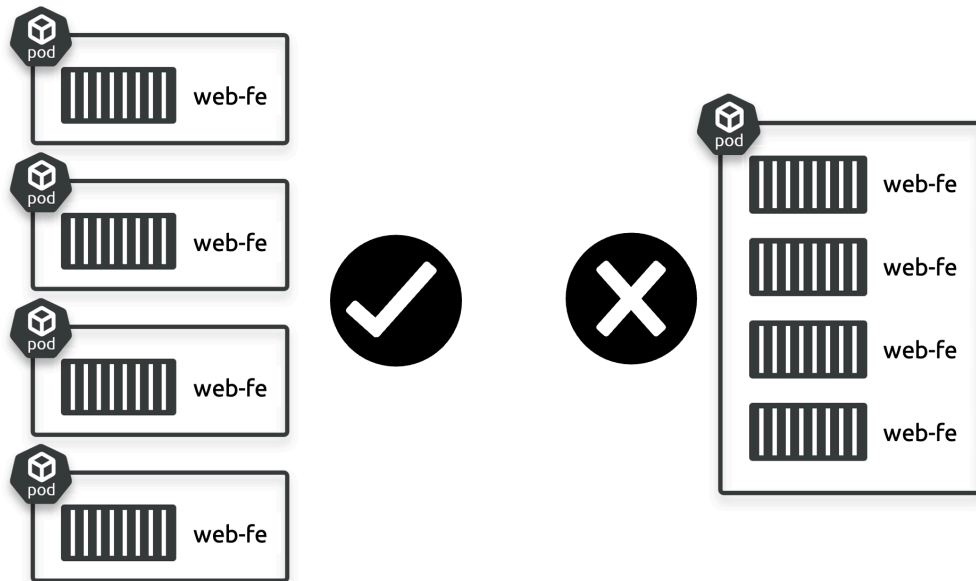
Figure 2.10 - Scaling with Pods

Pod lifecycle

Pods are mortal — they're created, they live, and they die. Anytime one dies, Kubernetes replaces it with a new one. Even though the new one looks, smells, and feels the same as the old one, it's always a shiny new one with a new ID and new IP.

This forces you to design applications to be loosely coupled so they're immune to individual Pod failures.

Pod immutability

Pods are immutable. This means you never change them once they're running.

For example, if you need to change or update a Pod, you always replace it with a new one running the updates. You should never log on to a Pod and change it. This means any time we talk about *"updating Pods"*, we always mean deleting the old one and replacing it with a new one. This can be a huge mindset change for some of us, but it fits nicely with modern tools and GitOps-style workflows.

Deployments

Even though Kubernetes works with Pods, you'll almost always deploy them via higher-level controllers such as *Deployments, StatefulSets,* and *DaemonSets*. These are all control plane services that operate as background watch loops, reconciling observed state with desired state.

Deployments add self-healing, scaling, rolling updates, and versioned rollbacks to stateless apps.

Refer back to Figure 2.7 to see how Deployments wrap Pods.

Service objects and stable networking

Earlier in the chapter, we said that Pods are mortal and can die. However, if a failed Pod is managed by a controller, it gets replaced by a new Pod with a new ID and a new IP address. The same thing happens with rollouts and scaling operations:

- Rollouts replace old Pods with new ones with new IPs
- Scaling up adds new Pods with new IPs

- Scaling down deletes existing Pods.

Events like these generate *IP churn* and make Pods unreliable. For example, clients cannot make reliable connections to individual Pods as Kubernetes doesn't guarantee they'll exist.

This is where *Services* come into play by providing reliable networking for groups of Pods.

Figure 2.11 shows internal and external clients connecting to a group of Pods via a Kubernetes Service. The *Service* (capital "S" because it's a Kubernetes API resource) provides a reliable name and IP, and load balances requests to the Pods behind it.

Name: back-end
IP: 172.16.1.43
Port: 4434
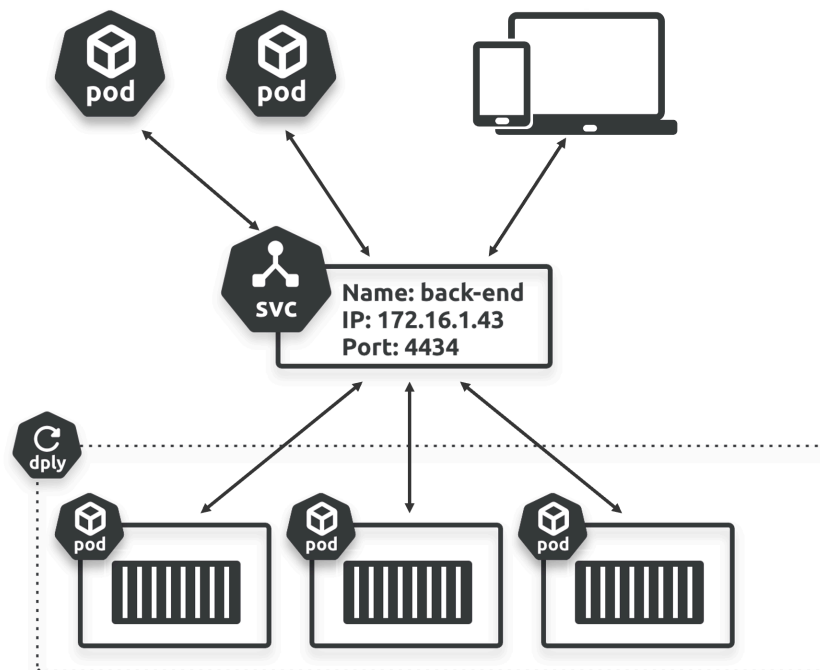
Figure 2.11

You should think of Services as having a front end and a back end. The front end has a stable DNS name, IP address, and network port. The back end uses labels to load balance traffic across a dynamic set of Pods.

Services keep a list of healthy Pods as scaling events, rollouts, and failures cause Pods to come and go. This means they'll always direct traffic to active healthy Pods. The Service also guarantees the name, IP, and port on the front end will never change.

Chapter summary

This chapter introduced you to some of the major Kubernetes features.

Control plane nodes host the control plane services that implement the intelligence of Kubernetes. They can be physical servers, VMs, cloud instances, and more. Production clusters usually run three or five control plane nodes for high availability.

Control plane services include the API server, the scheduler, the cluster store, and various controllers.

Worker nodes are where you run your business applications and can also be physical servers, VMs, cloud instances, and more.

Every worker node runs the kubelet service that watches the API server for new work tasks and reports back on task status.

Worker nodes also run one or more runtimes and the kube-proxy service. Runtimes perform low-level operations such as starting and stopping containers and Wasm apps. The kube-proxy handles all networking tasks on the node.

You learned that Kubernetes supports declarative and imperative methods of deploying and managing applications but prefers the declarative method. This is where you describe the desired state of something in a YAML configuration file that you give to Kubernetes and leave Kubernetes to deploy and manage it. Controllers run on the control plane and make sure observed state matches desired state. This is called reconciliation.

You also learned about Pods, Deployments, and Services. Pods allow containers and other workloads to run on Kubernetes. Deployments add self-healing, scaling, and rollouts. Services add reliable networking and basic load-balancing.

3: Getting Kubernetes

This chapter shows you how to install and configure the tools you'll need to follow every example in the book. These include:

1. Docker
2. A Kubernetes cluster
3. The `kubectl` command line utility

The easiest way to get all three is Docker Desktop. It installs Docker, includes a multi-node Kubernetes cluster, and automatically installs and configures `kubectl`. I use it every day, and I highly recommend it. However, you can't use the Docker Desktop Kubernetes cluster for the examples in Chapter 8 and Chapter 11. This is because they integrate with cloud load balancers and cloud storage services. For those chapters, you'll need a Kubernetes cluster in the cloud.

I recommend most readers install Docker Desktop because you get Docker and `kubectl`. You can then choose whether to build a cluster in the cloud or use the one that ships with Docker Desktop. You can even use Docker Desktop's built-in cluster for some examples and only build a cluster in the cloud for Chapters 8 and 11.

I've divided the chapter into the following sections:

- Install everything with Docker Desktop
- Build a Kubernetes cluster in the Linode Cloud

I'll provide a link that gets you $100 of free Linode credit that lasts for 60 days and is more than enough to complete all the examples in the book.

Install everything with Docker Desktop

You'll complete all of the following in this section:

- Create a Docker account (they're free)
- Install Docker Desktop
- Deploy Docker Desktop's built-in multi-node Kubernetes cluster

  Note: Docker Desktop is free for personal and educational use. If you **use it for work**, and your company has more than 250 employees or does more than $10M USD in annual revenue, you have to pay for a license.

Create a Docker account

Go to `app.docker.com/signup` and fill in your details to create a free **Personal** account.

Install Docker Desktop

Once you've created your account, complete the following steps to install Docker Desktop. You need Docker Desktop version 4.38 or newer:

1. Search the web for *Docker Desktop*
2. Download the installer for your system (Linux, Mac, or Windows)
3. Fire up the installer and follow the next, next, next instructions

Windows users should install the WSL 2 subsystem when prompted.

After the installation completes, you may need to start the app manually. Once it's running, Mac users get a whale icon in the menu bar at the top, whereas Windows users get the whale in the system

tray at the bottom. Clicking the whale exposes some basic controls and shows whether Docker Desktop is running.

Open a terminal and run the following commands to ensure Docker and `kubectl` are installed and working.

```
$ docker --version
Docker version 27.5.1, build 9f9e405

$ kubectl version --client=true -o yaml
clientVersion:
  major: "1"
  minor: "31"
  platform: darwin/arm64
```

Congratulations, you've installed Docker and `kubectl`.

Deploy Docker Desktop's built-in multi-node Kubernetes cluster

Docker Desktop v4.38 and later ship with a built-in multi-node Kubernetes cluster that's simple to deploy and use.

Click the Docker whale icon in your menu bar or system tray and sign in. You'll see your username instead of the **Sign in/Sign up** option if you're already signed in.

Once signed in, click the Docker whale again and choose the **Settings** option to open the GUI.

Ensure the **Use containerd for pulling and storing images** feature is enabled on the **General** tab. You may need to click **Apply & restart** to make the changes.

Select **Kubernetes** from the left navigation bar, check the **Enable Kubernetes** option, and choose the **kind (sign-in required)** option. It's important you choose this option, as the other option (**kubeadm**) only creates a single-node cluster.

If you're running Docker Desktop v4.38 or later and don't see the **kind (sign-in required)** option you can try using the *konami code* to enable it. To do this, enter the following key sequence from the Kubernetes settings page — up, up, down, down, left, right, left, right, b, a. This opens the **Experimental features** page where you can enable the **MultiNodeKubernetes** feature. Once you've done this, you can go back and try enabling the **kind (sign-in required)** cluster again.

Assuming you've chosen the **kind (sign-in required)** option, move the **Node(s)** slider to **3**, and check the box next to **Show system containers (advanced)** option. You can also enable the Kubernetes Dashboard, but none of the book's examples use it. Your options should look like Figure 3.1.
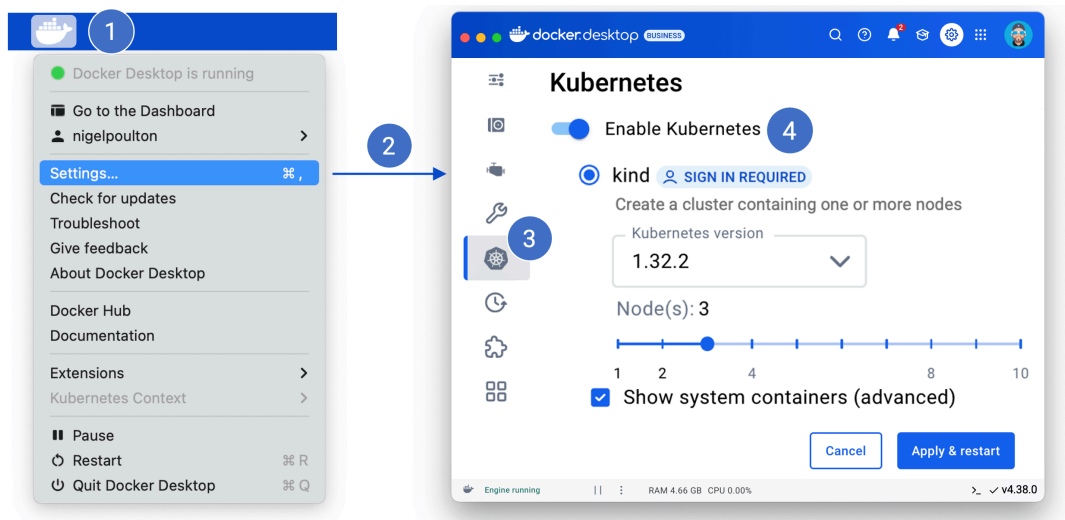
Figure 3.1 - Docker Desktop Kubernetes configuration

This will create a three-node cluster with one control plane node and two workers.

Click **Apply & restart**. It'll take a minute or two for your cluster to start, and you can monitor the build progress from the same screen. Once it's running, you'll see **Kubernetes running** in green text at the bottom of the Docker Desktop UI.

Run the following command to see your cluster.

```
$ kubectl get nodes
NAME                    STATUS    ROLES           AGE    VERSION
desktop-control-plane   Ready     control-plane   10m    v1.32.2
desktop-worker          Ready     <none>          10m    v1.32.2
desktop-worker2         Ready     <none>          10m    v1.32.2
```

You should see three nodes with the names shown in the example.

Go back to Docker Desktop and navigate to the **Containers** tab, where you'll see three containers with the same names as your three cluster nodes. This is because Docker Desktop runs your Kubernetes cluster as containers. Don't let this confuse you, the user experience is exactly the same.
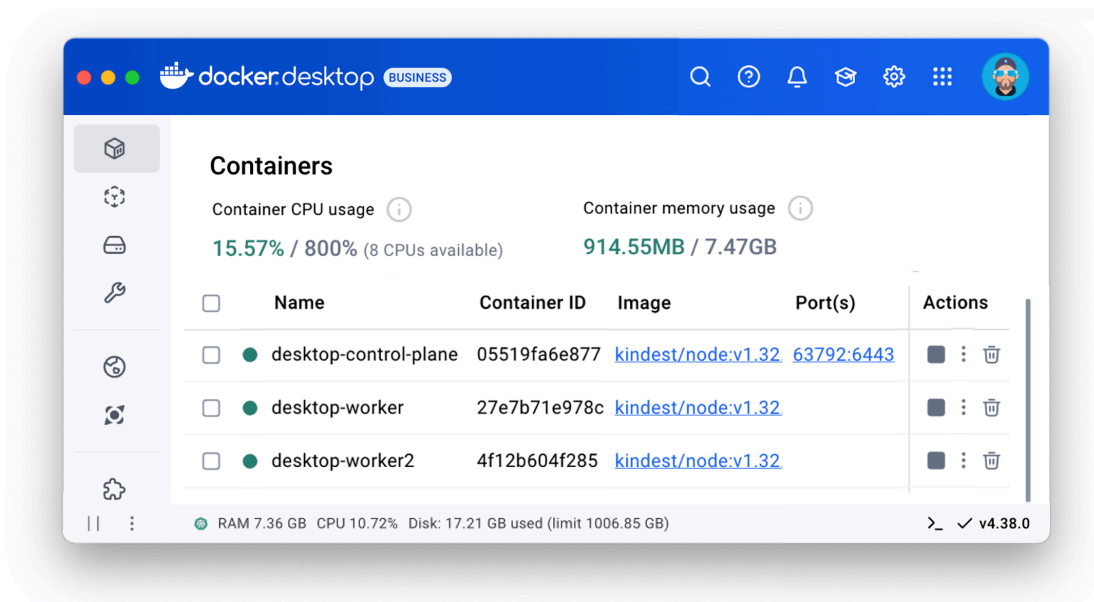
Figure 3.2 - Cluster nodes as containers

Congratulations, you've installed Docker and `kubectl` and deployed a multi-node Kubernetes cluster to your laptop. You'll be able to use this cluster for most of the examples in the book. However, you'll need a Kubernetes cluster in the cloud if you want to follow the Ingress and Storage examples in Chapter 8 and Chapter 11. The next section shows you how to build one.

Build a Linode Kubernetes Engine (LKE) cluster in the Linode Cloud

This section shows you how to get a multi-node Kubernetes cluster in the cloud using *Linode Kubernetes Engine (LKE).* Most other clouds have their own Kubernetes service, and you can use any of them. However, I've designed the examples in Chapter 11 to work with Linode cloud storage, meaning you'll have to change the storage configuration files if you use a different cloud.

I also provide a link that should get new users $100 of free Linode credit, which is more than enough to complete all the examples in the book.

Build a Kubernetes cluster in the Linode Cloud

LKE is a *hosted Kubernetes* service where Linode builds the cluster and manages availability, performance, and updates. Hosted Kubernetes services like this are close as you'll get to a zero-effort *production-grade* Kubernetes cluster.

You'll complete the following steps to build your LKE cluster:

- Sign up for a Linode account
- Create your LKE cluster
- Configure `kubectl`
- Test your LKE cluster

Sign up for a Linode account

Go to the following link and sign up for your free $100 credit that lasts for sixty days. You'll need to enter valid billing info in case you spend more or use it for longer than 60 days.

```
https://bit.ly/4b7YZix
```

If the link doesn't work, you can try the full URL.

```
https://www.linode.com/lp/refer/?
r=6107b344722dbd6017ea12da672510a85f8b5e84
```

I'm unable to help if the link doesn't work. In that case, you'll have to sign up from the main `linode.com` home page, and you may not get the free credit. Either way, you'll need to create an account and provide payment details.

Create your LKE cluster

Once you've created your account, go to `cloud.linode.com`, select **Kubernetes** from the left navigation pane, and click the **Create Cluster** button. Give your cluster the following details:

- **Cluster label:** tkb
- **Region:** Choose a region close to you
- **Kubernetes Version:** Choose a recent version
- **HA Control Plane:** No
- **Control Plane ACL:** Disabled
- **Add Node Pools:** Select the **Shared CPU** tab and add 3 x **Linode 2GB** nodes

Your options will look like Figure 3.2, and you'll have an estimated cluster cost.
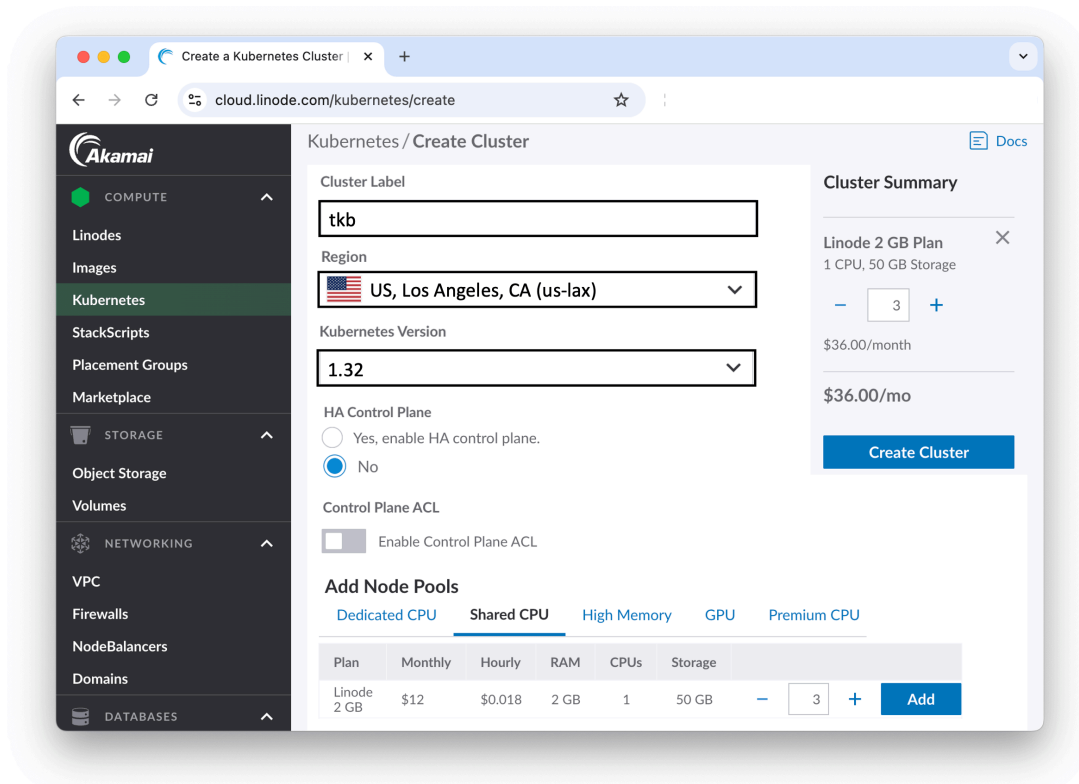
Figure 3.3 - LKE cluster settings

Click the **Create Cluster** button when you're happy with your cluster configuration and associated costs. It can take a minute or two for Linode to build your cluster.

Your LKE cluster is ready when all three nodes show green in the console, and the only thing left to do is configure `kubectl` to connect to it.

Configure `kubectl`

`kubectl` is the Kubernetes command-line tool and you'll use it in all the hands-on examples. You'll already have it if you installed Docker Desktop. If you don't have it, search the web for *install kubectl* and follow the instructions for your system. It's important that you install a version that is within one minor version of your Kubernetes cluster. For example, if your cluster is running Kubernetes v1.32.x,

your `kubectl` should be no lower than v1.31.x and no higher than v1.33.x.

Behind the scenes, `kubectl` reads your *kubeconfig* file to know which cluster to send commands to and which credentials to authenticate with. Your kubeconfig file is called `config` and lives in the following hidden directories depending on your system (you'll need to configure your system to show hidden folders):

- macOS: `/Users/<username>/.kube/`
- Windows: `C:\Users\<username>\.kube\`

Complete one of the following sections depending on whether you already have a kubeconfig file.

If you don't have a kubeconfig file

Only complete these steps if you're sure you don't have a hidden folder called `.kube` in your home directory with a file called `config`:

1. Create a hidden folder in your home directory called `.kube` (be sure to include the leading period)
2. Download your LKE kubeconfig file into this new `~/.kube` directory
3. Rename the file to `config`

It's vital that the filename is `config` with **no** filename extensions such as `.yaml`. You may have to configure your system to show filename extensions.

You're ready to test your LKE cluster.

If you already have a kubeconfig file

Complete these steps if you already have a kubeconfig file. You'll rename your existing kubeconfig file, merge its contents with your

LKE kubeconfig file and, use the new merged version.

Rename your existing kubeconfig file to `config-bkp`.

Download your LKE kubeconfig file from your Linode Cloud dashboard and copy it into your `~/.kube` directory. If you built your LKE cluster according to the earlier instructions, it will be called `tkb-kubeconfig.yaml`. You'll need to adjust some of the following commands if your downloaded file has a different name.

Run the following command to merge the configurations from both files and check it worked.

```
$ export KUBECONFIG=~/.kube/config-bkp:~/.kube/tkb-kubeconfig.yaml

$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://127.0.0.1:54225
<Snip>
```

If you look closely, you should see your LKE cluster's details listed in the `cluster, user`, and `context` sections.

Run the following commands to export the merged configuration into a new kubeconfig file called `config` and then set the `KUBECONFIG` environment variable to use the new file.

```
$ kubectl config view --flatten >  ~/.kube/config

$ export KUBECONFIG=~/.kube/config
```

The last thing to do is set your *current context* so that `kubectl` commands go to your LKE cluster.

If you installed Docker Desktop you can easily switch between contexts by clicking the Docker whale and choosing your LKE context from the **Kubernetes Context** option. If you didn't install

Docker Desktop, you can run the following commands to list your available contexts and switch to your LKE context. Remember, your LKE context name will be different from mine.

```
$ kubectl config get-contexts
CURRENT   NAME            CLUSTER         AUTHINFO          NAMESPACE
*         docker-desktop  docker-desktop  docker-desktop
          lke349416-ctx   lke349416       lke349416-admin   default

$ kubectl config use-context lke349416-ctx
Switched to context "lke349416-ctx".
```

Once you've completed these steps, you can move to the next section to test your LKE cluster.

Test your LKE cluster

Open a terminal and run the following command.

```
$ kubectl get nodes
NAME                              STATUS   ROLES     AGE    VERSION
lke349416-551020-184a46360000     Ready    <none>    19m    v1.32.1
lke349416-551020-1c6f99c20000     Ready    <none>    19m    v1.32.1
lke349416-551020-47ad6c5c0000     Ready    <none>    19m    v1.32.1
```

You should see three nodes, and their names should begin with `lke`. They should also have `<none>` in the `ROLES` column indicating they're all worker nodes. This is because Linode manages your control plane and hides it from you.

If you get an error or your node names don't start with `lke`, the most likely cause is a problem with your kubeconfig file. Review the previous processes and make sure you completed the steps exactly.

You're ready to go if you see three nodes with names starting with `lke`.

Remember to delete your LKE cluster when you're finished with it to avoid unwanted costs.

More about `kubectl` and your kubeconfig file

Every time you execute a `kubectl` command it does the following three things:

1. Converts the command into an HTTP REST request
2. Sends the request to the Kubernetes cluster defined in the `current-context` of your kubeconfig file
3. Uses the credentials specified in the `current-context` of your kubeconfig file

Your kubeconfig file is called `config` and lives in your home directory's hidden `.kube` folder. It defines:

- Clusters
- Users (credentials)
- Contexts
- Current context

The `clusters` section is a list of known Kubernetes clusters, the `users` section is a list of user credentials, and the `contexts` section is where you match clusters and credentials. For example, you might have a context called **ops-prod** that combines the **ops** credentials with the **prod** cluster. If this is also defined as your *current context,* `kubectl` will send all commands to your **prod** cluster and authenticate with the **ops** credentials.

Here is a simple kubeconfig file with a single cluster called **shield**, a single user called **coulson**, and a single context called **director**. The **director** context combines the **coulson** credentials and the **shield** cluster. It's also set as the default context.

```
apiVersion: v1
kind: Config
clusters:                          <<---- All known clusters
are listed in this block
- name: shield                     <<---- Friendly name for a
cluster
```

```
  cluster:
    server: https://192.168.1.77:8443         <<---- Cluster's AIP
endpoint (API server)
    certificate-authority-data: LS0tLS...     <<---- Cluster's
certificate
users:                                        <<---- Users are in this
block
- name: coulson                               <<---- Friendly name (not
used by Kubernetes)
  user:
    client-certificate-data: LS0tLS1CRU...    <<---- User
certificate/credentials
    client-key-data: LS0tLS1CRUdJTiBFQyB      <<---- User private key
contexts:                                     <<---- List of contexts
(cluster:user pairs)
- context:
  name: director                              <<---- Context called
"director"
    cluster: shield                           <<---- Send commands to
this cluster
    user: coulson                             <<---- Authenticate with
these credentials
current-context: director                     <<---- kubectl will use
this context
```

You can run a `kubectl config view` command to view your kubeconfig. The command will redact sensitive data. You can also run a `kubectl config current-context` to see your current context.

The following example shows a system configured to send `kubectl` commands to the cluster and credentials defined in the **docker-desktop** context.

```
$ kubectl config current-context
docker-desktop
```

If you installed Docker Desktop, you can easily switch between contexts by clicking the Docker whale and choosing the **Kubernetes Context** option.

Chapter summary

This chapter showed you a couple of ways to get a Kubernetes cluster. However, lots of other options exist.

I use Docker Desktop's multi-node Kubernetes cluster most days. However, I also use k3d, KinD, and minikube to get Kubernetes clusters on my laptop. The advantage of Docker Desktop is that it ships with the full suite of Docker development tools and automatically installs `kubectl`.

You also created a hosted Kubernetes cluster on Linode Kubernetes Engine (LKE) and configured `kubectl` to use it. However, this cluster may cost money and you should delete it when you're finished with it.

The chapter finished with an overview of `kubectl` and your kubeconfig file.

4: Working with Pods

Every app on Kubernetes runs inside a Pod.

- When you deploy an app, you deploy it in a Pod
- When you terminate an app, you terminate its Pod
- When you scale an app up, you add more Pods
- When you scale an app down, you remove Pods
- When you update an app, you deploy new Pods

This makes Pods important and is why this chapter goes into detail.

I've given the chapter two main parts:

- Pod Theory
- Hands-on with Pods

If some of the content we're about to cover feels familiar, it's because we're building on some of the concepts introduced in Chapter 2.

We're also about to discover that Kubernetes uses Pods to run many different workload types. However, most of the time, Pods run containers, so most of the examples will reference containers.

Pod theory

Kubernetes uses Pods for a lot of reasons. They're an abstraction layer, they enable resource sharing, they add features, they enhance scheduling, and more.

Let's take a closer look at some of those.

Pods are an abstraction layer

Pods abstract the workload details. This means you can run containers, VMs, serverless functions, and Wasm apps inside Pods and Kubernetes doesn't know the difference.

Using Pods as an abstraction layer benefits Kubernetes **and** workloads:

- *Kubernetes* can focus on deploying and managing Pods without having to care what's inside them
- Heterogenous *workloads* can run side-by-side on the same cluster, leverage the full power of the declarative Kubernetes API, and get all the other benefits of Pods

Containers and Wasm apps work with standard Pods, standard workload controllers, and standard runtimes. However, serverless functions and VMs need a bit of extra help.

Serverless functions run in standard Pods but require apps like Knative to extend the Kubernetes API with custom resources and controllers. VMs are similar and need apps like KubeVirt to extend the API.

Figure 4.1 shows four different workloads running on the same cluster. Each workload is wrapped in a Pod, managed by a controller, and uses a standard runtime. VM workloads run in a *VirtualMachineInstance (VMI)* instead of a Pod, but these are very similar to Pods and utilize a lot of Pod features.
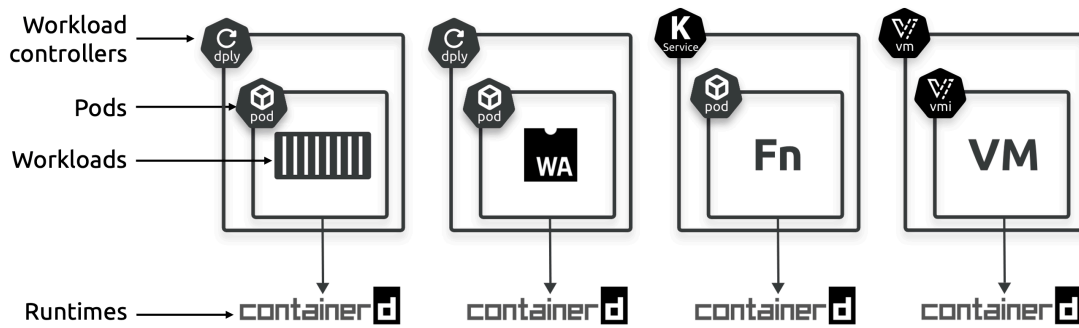
Figure 4.1 - Different workloads wrapped in Pods

Pods augment workloads

Pods augment workloads in many ways, including all of the following:

- Resource sharing
- Advanced scheduling
- Application health probes
- Restart policies
- Security policies
- Termination control
- Volumes

The following command shows a complete list of Pod attributes and returns over 1,000 lines. Press the spacebar to page through the output and press q to return to your prompt.

```
$ kubectl explain pods --recursive | more
KIND:      Pod
VERSION:  v1
DESCRIPTION:
     Pod is a collection of containers that can run on a host. This
resource is
     created by clients and scheduled onto hosts.
FIELDS:
   apiVersion          <string>
   kind                <string>
   metadata              <Object>
```

```
        annotations            <map[string]string>
        labels             <map[string]string>
        name                    <string>
        namespace        <string>
<Snip>
```

You can even drill into specific Pod attributes and see their supported values. The following example drills into the Pod *restartPolicy* attribute.

```
$ kubectl explain pod.spec.restartPolicy
KIND:     Pod
VERSION:  v1
FIELD:    restartPolicy <string>
DESCRIPTION:
    Restart policy for all containers within the pod. One of Always,
OnFailure, Never.
    Default to Always.
    More info: https://kubernetes.io/docs/concepts/workloads/pods/pod-
lifecycle/...
    Possible enum values:
    - `"Always"`
    - `"Never"`
    - `"OnFailure"`
```

Despite adding so much, Pods are lightweight and add very little overhead.

Pods enable resource sharing

Pods run one or more containers, and all containers in the same Pod share the Pod's *execution environment*. This includes:

- Shared filesystem and volumes (`mnt` namespace)
- Shared network stack (`net` namespace)
- Shared memory (`IPC` namespace)
- Shared process tree (`pid` namespace)
- Shared hostname (`uts` namespace)

Figure 4.2 shows a multi-container Pod with both containers sharing the Pod's volume and network resources.
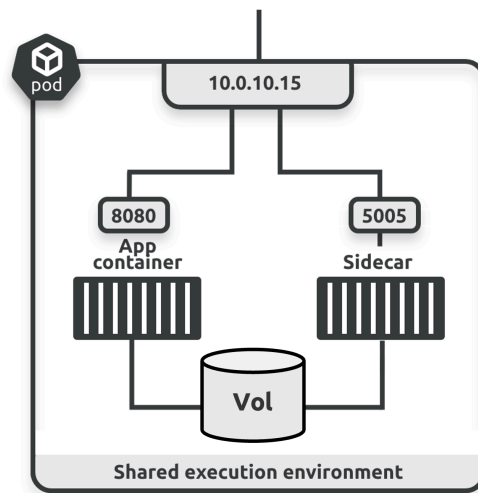
Figure 4.2 - Multi-container Pod sharing IP and volume

Other apps and clients can access the containers via the Pod's `10.0.10.15` IP address — the main app container is available on port `8080` and the sidecar on port `5005`. The two containers can use the Pod's localhost adapter if they need to communicate with each other inside the Pod. Both containers also mount the Pod's volume and can use it to share data. For example, the sidecar container might sync static content from a remote Git repo and store it in the volume where the main app container reads it and serves it as a web page.

Pods and scheduling

Before going any further, remember that *nodes* are host servers that can be physical servers, virtual machines, or cloud instances. *Pods* wrap containers and execute on *nodes.*

Kubernetes guarantees that all containers in a Pod will be scheduled to the same cluster node. Despite this, you should only put containers in the same Pod if they **need** to share resources such as memory, volumes, and networking. If your only requirement is to schedule two workloads to the same node, you should put

them in separate Pods and use one of the following options to ensure they're scheduled to the same node.

- nodeSelectors
- Affinity and anti-affinity
- Topology spread constraints
- Resource requests and resource limits

nodeSelectors are the simplest way of running Pods on specific nodes. You give it a list of labels, and the scheduler will only assign the Pod to a node with all the labels.

Affinity and *anti-affinity* rules are like a more powerful nodeSelector.

As the names suggest, they support scheduling alongside resources (affinity) and away from resources (anti-affinity). But they also support hard and soft rules, and they can select on Pods as well as nodes:

- Affinity rules *attract*
- Anti-affinity rules *repel*
- Hard rules must be *obeyed*
- Soft rules are only *suggestions* and *best effort*

Selecting on **nodes** is common and works like a nodeSelector where you supply a list of labels, and the scheduler assigns the Pod to **nodes** with those labels.

It works the same for selecting on Pods. You provide a list of labels, and Kubernetes ensures the Pod will run on the same nodes as other **Pods** with those labels.

Consider a couple of examples.

A *hard node affinity rule* specifying the `project=tkb` label tells the scheduler it can only run the Pod on nodes with that label. It won't schedule the Pod if it can't find a node with that label. If it was a

soft rule, the scheduler would *try* to find a node with the label, but if it can't find one, it'll still schedule the Pod. If it was an anti-affinity rule, the scheduler would look for nodes that **don't** have the label. The logic works the same for Pod-based rules.

Topology spread constraints are a flexible way of intelligently spreading Pods across your infrastructure for availability, performance, locality, or any other requirements. A typical example is spreading Pods across your cloud or data center's underlying availability zones for high availability (HA). However, you can create custom domains for almost anything, such as scheduling Pods closer to data sources, closer to clients for improved network latency, and many more reasons.

Resource requests and *resource limits* are very important, and every Pod should use them. They tell the scheduler how much CPU and memory a Pod needs, and the scheduler uses them to ensure they run on nodes with enough resources. If you don't specify them, the scheduler cannot know what resources a Pod requires and may schedule it to a node with insufficient resources.

Deploying Pods

Deploying a Pod includes the following steps:

1. Define the Pod in a YAML *manifest file*
2. Post the *manifest* to the API server
3. The request is authenticated and authorized
4. The Pod spec is validated
5. The scheduler filters nodes based on nodeSelectors, affinity and anti-affinity rules, topology spread constraints, resource requirements and limits, and more
6. The Pod is assigned to a healthy node meeting all requirements
7. The kubelet on the node watches the API server and notices the Pod assignment

8. The kubelet downloads the Pod spec and asks the local runtime to start it
9. The kubelet monitors the Pod status and reports status changes to the API server

If the scheduler can't find a suitable node, it marks it as pending.

Deploying a Pod is an *atomic operation*. This means a Pod only starts servicing requests when all its containers are running.

Pod lifecycle

Pods are designed to be *mortal* and *immutable.*

Mortal means you cannot restart a failed or deleted Pod. Yes, Kubernetes will *replace* failed Pods if a higher-level controller manages them. But this isn't the same as restarting and fixing a failed Pod.

Immutable means you cannot modify them after you've deployed them. This can be a huge mindset change if you're from a traditional background where you regularly patched live servers and logged on to them to make fixes and configuration changes. If you need to change a Pod, you create a **new one** with the changes, delete the old one, and replace it with the new one. If you need to write data to a Pod, you should attach a volume to it and store the data in the volume. This way, you can still access the data and the volume after the Pod is gone.

Let's look at a typical Pod lifecycle.

You define a Pod in a declarative YAML object that you post to the API server. It goes into the *pending* phase while the scheduler finds a node to run it on. Assuming it finds a node, the Pod gets scheduled, and the local kubelet instructs the runtime to start its containers. Once all of its containers are running, the Pod enters the

running phase. It remains in the running phase indefinitely if it's a long-lived Pod, such as a web server. If it's a short-lived Pod, such as a batch job, it enters the *succeeded* state as soon as all containers complete their tasks. You can see this in Figure 4.3.
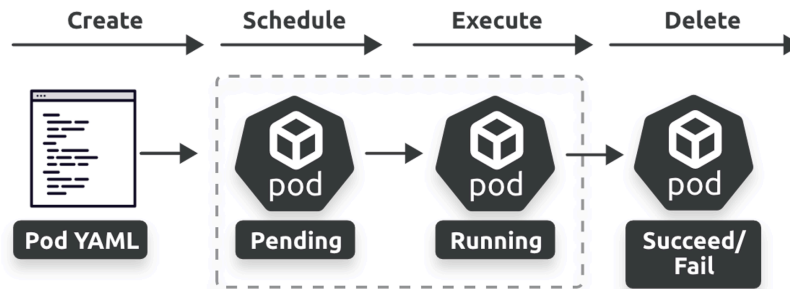
Figure 4.3 - Pod lifecycle

A quick side note about running VMs on Kubernetes. VMs are the opposite of containers, in that they are designed to be *mutable* and *immortal.* For example, you can restart them, change their configurations, and even migrate them. This is very different from the design goals of Pods and is why KubeVirt wraps VMs in a *modified Pod called a VirtualMachineInstance (VMI)* and manages them using custom workload controllers.

Restart Policies

Earlier in the chapter, we said Pods augment apps with restart policies. However, these apply to individual containers and not the Pod.

Let's consider some scenarios.

You use a Deployment controller to schedule a Pod to a node, and the node fails. When this happens, the Deployment controller notices the failed node, **deletes** the Pod, and replaces it with a **new one** on a surviving node. Even though the new Pod is based on the same Pod spec, it has a new UID, a new IP address, and no

state. It's the same when nodes evict Pods during node maintenance or due to resource juggling — the evicted Pod is deleted and replaced with a new one on another node.

The same thing even happens during scaling operations, updates, and rollbacks. For example, scaling down deletes Pods, and scaling up always adds new Pods.

The take-home point is that anytime we say we're *updating* or *restarting* Pods, we really mean replacing them with new ones.

Although Kubernetes can't restart Pods, it **can** restart containers. This is always done by the local kubelet and governed by the value of the Pod's `spec.restartPolicy`, which can be any of the following:

- Always
- Never
- OnFailure

The values are self-explanatory: `Always` will always attempt to restart a container, `Never` will never attempt a restart, and `OnFailure` will only attempt a restart if the container fails with an error code. The policy is Pod-wide, meaning it applies to all containers in the Pod except for *init containers.* More on init containers later.

The restart policy you choose depends on the nature of the app — whether it's a *long-living* container or a *short-living* container.

Long-living containers host apps such as web servers, data stores, and message queues that run indefinitely. If they fail, you normally want to restart them, so you'll typically give them the `Always` restart policy.

Short-living containers are different and typically run batch-style workloads that run a task through to completion. Most of the time, you're happy when they complete, and you only want to restart

them if they fail. As such, you'll probably give them the `OnFailure` restart policy. If you don't care if they fail, give them the `Never` policy.

In summary, Kubernetes never restarts Pods — when they fail, get scaled up and down, and get updated, Kubernetes always deletes old Pods and creates new ones. However, it can restart individual containers on the same node.

Static Pods vs controllers

There are two ways to deploy Pods:

1. Directly via a Pod manifest (rare)
2. Indirectly via a workload resource and controller (most common)

Deploying directly from a Pod manifest creates a *static Pod* that cannot self-heal, scale, or perform rolling updates. This is because they're only managed by the kubelet on the node they're running on, and kubelets are limited to restarting containers on the same node. Also, if the node fails, the kubelet fails with it and cannot do anything to help the Pod.

On the flip side, Pods deployed via *workload resources* get all the benefits of being managed by a highly available *controller* that can restart them on other nodes, scale them when demand changes, and perform advanced operations such as rolling updates and versioned rollbacks. The local kubelet can still attempt to restart failed containers, but if the node fails or gets evicted, the controller can restart it on a different node. More on workload resources and controllers in Chapter 6.

Remember, when we say *restart the Pod*, we mean replace it with a new one.

The pod network

Every Kubernetes cluster runs a *pod network* and automatically connects all Pods to it. It's usually a flat Layer-2 overlay network that spans every cluster node and allows every Pod to talk directly to every other Pod, even if the remote Pod is on a different cluster node.

Your *pod network* is implemented by a third-party plugin that interfaces with Kubernetes via the *Container Network Interface (CNI)*.

You choose a network plugin at cluster build time, and it configures the Pod network for the entire cluster. Lots of plugins exist, and each one has its pros and cons. However, at the time of writing, Cilium is the most popular and implements a lot of advanced features such as security and observability.

Figure 4.4 shows three nodes running five Pods. The pod network spans all three nodes, and all five Pods connect to it. This means all of the Pods can communicate despite being on different nodes. Notice how the nodes connect to external networks and do not connect directly to the pod network.
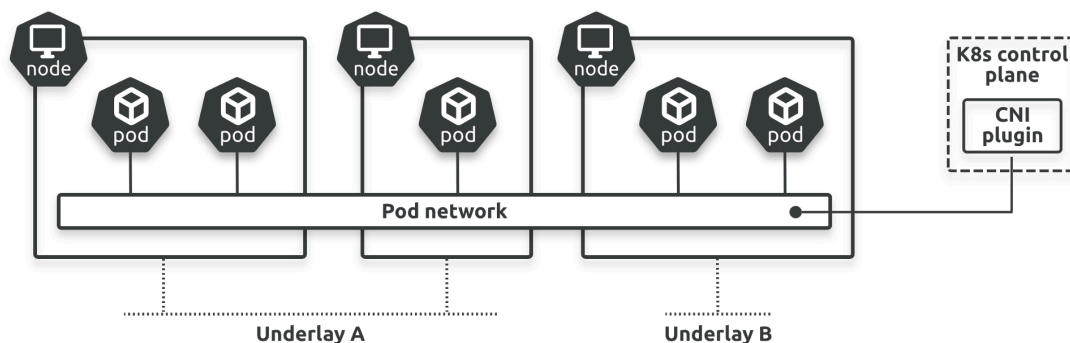
Figure 4.4 The pod network

It's common for newly created clusters to implement a very open pod network with little or no security. This makes Kubernetes easy to use and avoids frustrations commonly associated with network security. However, you should use Kubernetes Network Policies and other measures to secure it.

Multi-container Pods

Multi-container Pods are a powerful pattern and are very popular in the real world.

According to microservices design patterns, every container should have a single clearly defined responsibility. For example, an application syncing content from a repository and serving it as a web page has two distinct responsibilities:

1. Sync the content
2. Serve the web page

You should design this app with two microservices and give each one its own container — one container responsible for *syncing* the content and the other responsible for *serving* the content. We call this *separation of concerns*, or the *single responsibility principle,* and it keeps containers small and simple, encourages reuse, and makes troubleshooting easier.

Most of the time, you'll put application containers in their own Pods and they'll communicate over the pod network. However, sometimes it's better to put them in the same Pod. For example, sticking with the *sync and serve* app, putting both containers in the same Pod allows the **sync** container to pull content from the remote system and store it in a shared volume where the **web** container can read it and serve it. See Figure 4.5.
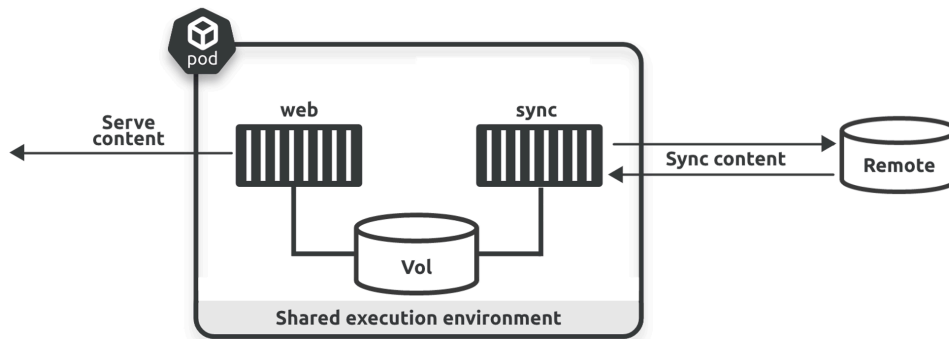
Figure 4.5 - Multi-container Pod

Kubernetes has two main patterns for multi-container Pods: *init containers* and *sidecar containers*. Let's look at both.

Multi-container Pods: Init containers

Init containers are a special type of container defined in the Kubernetes API. You run them in the same Pod as application containers, but Kubernetes guarantees they'll start and complete <u>before</u> the main app container starts. It also guarantees they'll only run once.

The purpose of init containers is to prepare and initialize the environment so it's ready for application containers.

Consider a couple of quick examples.

You have an application that should only start when a remote API is ready to accept connections. Instead of complicating the main application with the logic to check the remote API, you run that logic in an init container in the same Pod. When you deploy the Pod, Kubernetes starts the init container first, which sends periodic requests to the API server until it receives a response. While it's doing this, Kubernetes prevents the application container from starting. However, as soon as the init container receives a response

from the API server, it completes, and Kubernetes starts the application.

Assume you have another application that needs a one-time clone of a remote repository before starting. Again, instead of bloating and complicating the main application with the code to clone and prepare the content (knowledge of the remote server address, certificates, auth, file sync protocol, checksum verifications, etc.), you implement that in an init container that is guaranteed to complete the task before the main application container starts.

You can list multiple init containers per Pod and Kubernetes runs them in the order they appear in the Pod manifest. They all have to complete before Kubernetes moves on to start regular application containers, and if any init container fails, Kubernetes attempts to restart it. However, if you've set the Pod's `restartPolicy` to `Never`, Kubernetes will fail the Pod.

A drawback of init containers is that they're limited to running tasks **before** the main app container starts. For something that runs alongside the main app container, you need a *sidecar container*.

Multi-container Pods: Sidecars

The job of a *sidecar container* is to add functionality to an application without having to add it directly to the application container. Examples include sidecars that scrape logs, monitor & sync remote content, broker connections, munge data, and encrypt network traffic.

Figure 4.6 shows a multi-container Pod with an application container and a service mesh sidecar intercepting and encrypting all network traffic. In this example, it's vital that the sidecar starts before the main application container and keeps running for the entire life of the Pod — if the sidecar isn't running, the application container cannot use the network.
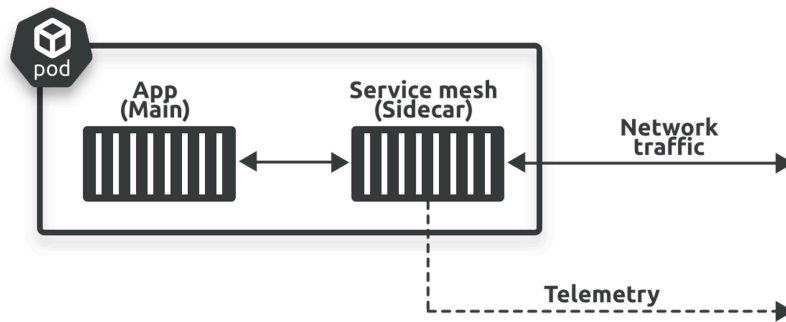
Figure 4.6 - Service mesh sidecar

Older versions of Kubernetes had no concept of sidecar containers, and we had to implement them as regular containers. However, this was problematic as there was no reliable way to start sidecars before app containers, keep them running alongside app containers, or stop them after app containers. Fortunately, Kubernetes v1.28 introduced native sidecars as an alpha feature and progressed them to beta status in v1.29. As of v1.32, sidecar containers are still a beta feature, and you should use them with caution. However, they're enabled by default and used by many notable projects, including Argo CD and Istio. We should expect them to reach the GA (stable) milestone very soon.

You'll see this in more detail in the hands-on section, but you define sidecars as init containers (`spec.initContainers`) with the `restartPolicy` set to `Always`. If you do this, Kubernetes guarantees they will:

- Start before the main application container
- Keep running alongside the main application container
- Terminate after the main application container

Aside from the above, they follow the other rules of init containers, such as startup order, and you can attach probes to manage and monitor their lifecycles.

Pod theory summary

Pods are the atomic unit of scheduling on Kubernetes and abstract the details of the workloads inside them. They also enable advanced scheduling and many other features.

Many Pods run a single container, but multi-container Pods are more powerful. You can use multi-container Pods to tightly-couple workloads that need to share resources such as memory and volumes. You can also use multi-container Pods to augment apps (sidecar pattern) and initialize environments (init pattern).

You define Pods in declarative YAML objects, but you'll usually deploy them via higher-level workload controllers that augment them with superpowers such as self-healing, autoscaling, and more.

Time to see some examples.

Hands-on with Pods

If you're following along, clone the book's GitHub repo and switch to the **2025** branch.

```
$ git clone https://github.com/nigelpoulton/TKB.git
Cloning into 'TKB'...

$ cd TKB

$ git fetch origin

$ git checkout -b 2025 origin/2025
```

Be sure to run all commands from the `pods` folder.

Pod manifest files

Let's see our first Pod manifest. This is the `pod.yml` file from the `pods` folder.

```
kind: Pod
apiVersion: v1
metadata:
  name: hello-pod
  labels:
    zone: prod
    version: v1
spec:
  containers:
  - name: hello-ctr
    image: nigelpoulton/k8sbook:1.0
    ports:
    - containerPort: 8080
    resources:
      limits:
        memory: 128Mi
        cpu: 0.5
```

It's a simple example, but straight away you can see four top-level fields:

- kind
- apiVersion
- metadata
- spec

The `kind` field tells Kubernetes what type of object you're defining. This one's defining a Pod, but if you were defining a Deployment, the `kind` field would say **Deployment**.

`apiVersion` tells Kubernetes what version of the API to use when creating the object.

So far, this manifest describes a Pod and tells Kubernetes to build it using the `v1` Pod schema.

The `metadata` section names the Pod **hello-pod** and gives it two labels. You'll use the labels in a future chapter to connect it to a Service for networking.

Most of the action happens in the `spec` section. This example defines a single-container Pod with an application container called **hello-ctr**. The container is based on the `nigelpoulton/k8sbook:1.0` image, listens on port `8080`, and tells the scheduler it needs a maximum of 128MB of memory and half a CPU.

Manifest files: Empathy as Code

Quick side-step.

Kubernetes YAML files are excellent sources of documentation, and you can use them to get new team members up to speed quickly and help bridge the gap between developers and operations.

For example, new team members can read your YAML files and quickly learn your application's basic functions and requirements. Operations teams can also use them to understand application requirements such as network ports, CPU and memory requirements, and much more.

You can also store them in source control repositories for easy versioning and running diffs against other versions.

Nirmal Mehta described these side benefits as a form of *empathy as code* in his 2017 DockerCon talk entitled *A Strong Belief, Loosely Held: Bringing Empathy to IT*.

Deploying Pods from a manifest file

Run the following `kubectl apply` command to deploy the Pod. The command sends the `pod.yml` file to the API server defined in the current context of your *kubeconfig* file. It also authenticates the request using credentials from your kubeconfig file.

```
$ kubectl apply -f pod.yml
pod/hello-pod created
```

Although the output says the Pod is created, it might still be pulling the image and starting the container.

Run a `kubectl get pods` to check the status.

```
$ kubectl get pods
NAME        READY    STATUS             RESTARTS    AGE
hello-pod   0/1      ContainerCreating  0           9s
```

The Pod in the example isn't fully created yet — the `READY` column shows zero containers ready, and the `STATUS` column shows why.

This is a good time to mention that Kubernetes automatically pulls (downloads) images from Docker Hub. To use another registry, just add the registry's URL before the image name in the YAML file.

Once the `READY` column shows `1/1` and the `STATUS` column shows `Running`, your Pod is running on a healthy cluster node and actively monitored by the node's kubelet.

You'll see how to connect to the app and test it in future chapters.

Introspecting Pods

Let's look at some of the main ways you'll use `kubectl` to monitor and inspect Pods.

kubectl get

You've already run a `kubectl get pods` command and seen that it returns a single line of basic info. However, the following flags get you a lot more info:

- `-o wide` gives a few more columns but is still a single line of output
- `-o yaml` gets you everything Kubernetes knows about the object

The following example shows the output of a `kubectl get pods` with the `-o yaml` flag. I've snipped the output, but you can see it's divided into two main parts:

- spec
- status

The `spec` section shows the *desired state* of the object, and the `status` section shows the *observed state*.

```
$ kubectl get pods hello-pod -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      <Snip>
  name: hello-pod
  namespace: default
spec:                           <<---- Desired state is in this block
  containers:
  - image: nigelpoulton/k8sbook:1.0
    imagePullPolicy: IfNotPresent
    name: hello-ctr
    ports:
    <Snip>
status:                         <<---- Observed state is in this block
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2024-01-03T18:21:51Z"
    status: "True"
    type: Initialized
  <Snip>
```

The full output contains much more than the 17-line YAML file you used to create the Pod. So, where does Kubernetes get all this extra detail?

Two main sources:

- Pods have a lot of properties, and anything you don't explicitly define in a YAML file gets populated with defaults

- The `status` section shows you the current state of the Pod and isn't part of your YAML file

kubectl describe

Another great command is `kubectl describe`. This gives you a nicely formatted overview of an object, including lifecycle events.

```
$ kubectl describe pod hello-pod
Name:          hello-pod
Namespace:     default
Labels:        version=v1
               zone=prod
Status:        Running
IP:            10.1.0.103
Containers:
  hello-ctr:
    Container ID:   containerd://ec0c3e...
    Image:          nigelpoulton/k8sbook:1.0
    Port:           8080/TCP
    <Snip>
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  <Snip>
Events:
  Type    Reason     Age      Message
  ----    ------     ----     -------
  Normal  Scheduled  5m30s    Successfully assigned ...
  Normal  Pulling    5m30s    Pulling image "nigelpoulton/k8sbook:1.0"
  Normal  Pulled     5m8s     Successfully pulled image ...
  Normal  Created    5m8s     Created container hello-ctr
  Normal  Started    5m8s     Started container hello-ctr
```

I've snipped the output for the book, but you'll learn a lot if you study the full output on your own system.

kubectl logs

You can use the `kubectl logs` command to pull the logs from any container in a Pod. The basic format of the command is `kubectl`

`logs <pod>.`

If you run the command against a multi-container Pod, you automatically get the logs from the first container in the Pod. However, you can override this by using the `--container` flag and specifying the name of a different container. If you're unsure of container names or the order in which they appear in a multi-container Pod, just run a `kubectl describe pod <pod>` command. You can get the same info from the Pod's YAML file.

The following YAML shows a multi-container Pod with two containers. The first container is called **app**, and the second is called **syncer**. Running a `kubectl logs` against this Pod without specifying the `--container` flag will get you the logs from the **app** container.

```
kind: Pod
apiVersion: v1
metadata:
  name: logtest
spec:
  containers:
  - name: app                        <<---- First container (default)
    image: nginx
      ports:
        - containerPort: 8080
  - name: syncer                     <<---- Second container
    image: k8s.gcr.io/git-sync:v3.1.6
    volumeMounts:
    - name: html
<Snip>
```

You'd run the following command if you wanted the logs from the **syncer** container. Don't run this command, as you haven't deployed this Pod.

```
$ kubectl logs logtest --container syncer
```

kubectl exec

The `kubectl exec` command is a great way to execute commands inside running containers.

You can use `kubectl exec` in two ways:

1. Remote command execution
2. Exec session

Remote command execution lets you send commands to a container from your local shell. The container executes the command and returns the output to your shell.

An *exec session* connects your local shell to the container's shell and is the same as being logged on to the container.

Let's look at both, starting with remote command execution.

Run the following command from your local shell. It's asking the first container in the **hello-pod** Pod to run a `ps` command.

```
$ kubectl exec hello-pod -- ps
PID   USER      TIME  COMMAND
  1   root      0:00 node ./app.js
 17   root      0:00 ps aux
```

The container executed the `ps` command and displayed the result in your local terminal.

The format of the command is `kubectl exec <pod> -- <command>`, and you can execute any command installed in the container. Remember to use the `--container` flag if you want to run the command in a specific container.

Try running the following command.

```
$ kubectl exec hello-pod -- curl localhost:8080
OCI runtime exec failed:...... "curl": executable file not found in
```

```
$PATH
```

This one failed because this container doesn't have the `curl` command.

Let's use `kubectl exec` to get an interactive exec session to the same container. This works by connecting your terminal to the container's terminal, and it feels like an SSH session.

Run the following command to create an exec session to the first container in the **hello-pod** Pod. Your shell prompt will change to indicate you're connected to the container's shell.

```
$ kubectl exec -it hello-pod -- sh
#
```

The `-it` flag tells `kubectl exec` to make the session interactive by connecting your shell's STDIN and STDOUT streams to the STDIN and STDOUT of the first container in the Pod. The `sh` command starts a new shell process in the session, and your prompt will change to indicate you're now inside the container.

Run the following commands from within the exec session to install the `curl` binary and then execute a `curl` command.

```
# apk add curl
<Snip>

# curl localhost:8080
<html><head><title>K8s rocks!</title><link rel="stylesheet"
href="http://netdna....
```

Making changes like this to live Pods is an *anti-pattern* as Pods are designed as immutable objects. However, it's OK for demonstration purposes like this.

Pod hostnames

Pods get their names from their YAML file's `metadata.name` field and Kubernetes uses this as the hostname for every container in the Pod.

If you're following along, you'll have a single Pod called **hello-pod**. You deployed it from the following YAML file that sets the Pod name as **hello-pod**.

```
kind: Pod
apiVersion: v1
metadata:
  name: hello-pod        <<---- Pod hostname. Inherited by all
containers.
  labels:
  <Snip>
```

Run the following command from inside your existing exec session to check the container's hostname. The command is case-sensitive.

```
$ env | grep HOSTNAME
HOSTNAME=hello-pod
```

As you can see, the container's hostname matches the name of the Pod. If this was a multi-container Pod, all of its containers would have the same hostname.

Because of this, you should ensure that Pod names are valid DNS names (a-z, 0-9, the minus and period signs).

Type `exit` to quit your exec session and return to your local terminal.

Check Pod immutability

Pods are designed as immutable objects, meaning you shouldn't change them after you've deployed them.

Immutability applies at two levels:

- Object immutability (the Pod)
- App immutability (containers)

Kubernetes handles *object immutability* by preventing changes to a running Pod's configuration. However, Kubernetes can't always prevent you from changing the app and filesystem inside of containers. You're responsible for ensuring containers and their apps are stateless and immutable.

The following example uses `kubectl edit` to edit a live Pod object. Try and change any of these attributes:

- Container name
- Container port
- Resource requests and limits

You need to run this command from your local terminal, and it will open the Pod's configuration in your default editor. For Mac and Linux users, it will typically open the session in `vi`, whereas for Windows, it's usually `notepad.exe.`

```
$ kubectl edit pod hello-pod

# Please edit the object below. Lines beginning with a '#' will be
ignored...
apiVersion: v1
kind: Pod
metadata:
  <Snip>
  labels:
    version: v1
    zone: prod
  name: hello-pod
  namespace: default
  resourceVersion: "432621"
  uid: a131fb37-ceb4-4484-9e23-26c0b9e7b4f4
spec:
  containers:
  - image: nigelpoulton/k8sbook:1.0
    imagePullPolicy: IfNotPresent
```

```
  name: hello-ctr                    <<---- Try to change this
ports:
- containerPort: 8080                <<---- Try to change this
  protocol: TCP
resources:
  limits:
    cpu: 500m                        <<---- Try to change this
    memory: 256Mi                    <<---- Try to change this
  requests:
    cpu: 500m                        <<---- Try to change this
    memory: 256Mi                    <<---- Try to change this
```

Edit the file, save your changes, and close your editor. You'll get a message telling you the changes are forbidden because the attributes are immutable.

If you get stuck inside the `vi` session, you can probably exit by typing the following key combination — `:q!` and then pressing **RETURN.**

Resource requests and resource limits

Kubernetes lets you specify *resource requests* and *resource limits* for every container in a Pod.

- *Requests* are minimum values
- *Limits* are maximum values

Consider the following snippet from a Pod YAML:

```
resources:
  requests:                <<---- Minimums for scheduling
    cpu: 0.5
    memory: 256Mi
  limits:                  <<---- Maximums for kubelet to cap
    cpu: 1.0
    memory: 512Mi
```

This container needs a minimum of 256Mi of memory and half a CPU. The scheduler reads this and assigns it to a node with enough resources. If it can't find a suitable node, it marks the Pod as

pending, and the Cluster Autoscaler will attempt to provision a new cluster node.

Assuming the scheduler finds a suitable node, it assigns the Pod to the node, and the kubelet downloads the Pod spec and asks the local runtime to start it. As part of the process, the kubelet reserves the *requested* CPU and memory, guaranteeing the resources will be there when needed. It also tells the runtime to set a resource cap based on each container's *resource limits*. In this example, it asks the runtime to set a cap of one CPU and 512Mi of memory. Most runtimes will enforce these limits, but how each runtime implements this can vary.

While a container executes, it is guaranteed access to its minimum requirements (*requests*). It can also use more if the node has additional resources available, but it's never allowed to use more than what you specify in its *limits*.

For multi-container Pods, the scheduler combines the requests for all containers and finds a node with enough resources to satisfy the full Pod.

If you've been following the examples closely, you'll have noticed that the `pod.yml` you used to deploy the **hello-pod** only specified resource limits — it didn't specify resource requests. However, some command outputs have displayed limits **and** requests. This is because Kubernetes automatically sets requests to match limits if you don't specify requests.

Multi-container Pod example – init container

The following YAML defines a multi-container Pod with an init container and an app container. It's from the `initpod.yml` file in the `pods` folder of the book's GitHub repo.

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: initpod
  labels:
    app: initializer
spec:
  initContainers:
  - name: init-ctr
    image: busybox:1.28.4
    command: ['sh', '-c', 'until nslookup k8sbook; do echo waiting for
k8sbook service;\
             sleep 1; done; echo Service found!']
  containers:
    - name: web-ctr
      image: nigelpoulton/web-app:1.0
      ports:
        - containerPort: 8080
```

Defining containers under the `spec.initContainers` block makes them init containers that Kubernetes guarantees will run and complete before it starts regular containers.

Regular containers are defined under the `spec.containers` block and will not start until **all** init containers successfully complete.

This example has a single init container called **init-ctr** and a single app container called **web-ctr**. The init container runs a loop looking for a Kubernetes Service called **k8sBook**. It will remain running in this loop until you create the Service. Once you create the Service, the init container will see it and exit, allowing the app container to start. You'll learn about *Services* in a future chapter.

Deploy the multi-container Pod with the following command and then run a `kubectl get pods` with the `--watch` flag to see if it comes up.

```
$ kubectl apply -f initpod.yml
pod/initpod created

$ kubectl get pods --watch
NAME        READY    STATUS       RESTARTS    AGE
initpod     0/1      Init:0/1     0           6s
```

The `Init:0/1` status tells you that the init container is still running, meaning the main container hasn't started yet. If you run a `kubectl describe` command, you'll see the overall Pod status is `Pending.`

```
$ kubectl describe pod initpod
Name:             initpod
Namespace:        default
Priority:         0
Service Account:  default
Node:             docker-desktop/192.168.65.3
Labels:           app=initializer
Annotations:      <none>
Status:           Pending                <<---- Pod status
<Snip>
```

The Pod will remain in this phase until you create a Service called **k8sbook**.

Run the following commands to create the Service and re-check the Pod status.

```
$ kubectl apply -f initsvc.yml
service/k8sbook created

$ kubectl get pods --watch
NAME      READY   STATUS            RESTARTS   AGE
initpod   0/1     Init:0/1          0          15s
initpod   0/1     PodInitializing   0          3m39s
initpod   1/1     Running           0          3m57s
```

The init container completes when it sees the Service, and the main application container starts. Give it a few seconds to fully start.

If you run another `kubectl describe` against the **initpod** Pod, you'll see the init container is in the *terminated* state because it completed successfully (exit code 0).

Multi-container Pod example – sidecar container

The job of a sidecar container is to augment an application container by providing a secondary service such as log scraping or

synchronizing with a remote repository.

Kubernetes runs Sidecar containers in the same Pod as application containers so they can share resources such as volumes. Kubernetes also guarantees that sidecars will start before app containers, run as long as app containers run, and terminate after app containers.

You define sidecar containers in YAML manifest files below `spec.initContainers` and set the container's `restartPolicy` to `Always.` This restart policy is what sets sidecars apart from regular init containers and ensures they'll run alongside the app container. You define application containers in the same Pod manifest below `spec.containers.`

The following YAML file defines the multi-container Pod you're about to deploy. It has a single sidecar container called **ctr-sync** and a single app container called **ctr-web**. The **ctr-sync** sidecar has the `restartPolicy: Always` setting, which only applies to the container and overrides any restart policy you might set at the Pod level.

```
apiVersion: v1
kind: Pod
metadata:
  name: git-sync
  labels:
    app: sidecar
spec:
  initContainers:
  - name: ctr-sync                       ---┐
    restartPolicy: Always                   |        <<---- Setting to
"Always" makes this a sidecar
    image: k8s.gcr.io/git-sync:v3.1.6    |
    volumeMounts:                           |
    - name: html                         | S
      mountPath: /tmp/git                | i
    env:                                    | d
    - name: GIT_SYNC_REPO                | e
      value: https://github.com...       | c
    - name: GIT_SYNC_BRANCH              | a
      value: master                      | r
    - name: GIT_SYNC_DEPTH               |
      value: "1"                         |
```

```
    - name: GIT_SYNC_DEST         |
      value: "html"               ---┘
  containers:
  - name: ctr-web                 ---┐
    image: nginx                     | A
    volumeMounts:                    | p
    - name: html                     | p
      mountPath: /usr/share/nginx/  ---┘
  volumes:
  - name: html
    emptyDir: {}
```

The **ctr-sync** sidecar container watches a GitHub repo and synchronizes any changes into a shared volume called **html**. The **ctr-web** app container watches this shared volume and serves a web page from its contents. In the example you're about to follow, you'll start the app, update the remote GitHub repo, and prove that the sidecar container synchronizes the updates.

> **Note:** You'll need Kubernetes v1.29 or later and a GitHub account to follow this example. GitHub accounts are free.

You'll complete the following steps:

1. Fork the GitHub repo
2. Update the YAML file with the URL of **your forked repo**
3. Deploy the app
4. Connect to the app and see it display *This is version 1.0*
5. Make a change to **your fork** of the GitHub repo
6. Verify your changes appear on the app's web page

1. Fork the GitHub repo

You'll need a GitHub account to complete this step. They're free.

Point your browser to this URL:

`https://github.com/nigelpoulton/ps-sidecar`.

Click the **Fork** dropdown button, choose the **+ Create a new fork** option, fill in the required details, and click the green **Create fork** button.

This will take you to your newly forked repo where you can copy its URL. Be sure to copy the URL of your forked repo.

2. Update the YAML file

Return to your local machine, edit the `initsidecar.yml` file in the `pods` directory, paste the copied URL into the `GIT_SYNC_REPO` field, and save your changes.

3. Deploy the sidecar app

Run the following command to deploy the application from the `initsidecar.yml` file. It will deploy the Pod with the app and sidecar containers, as well as a Service that you'll use to connect to the app.

```
$ kubectl apply -f initsidecar.yml
pod/git-sync created
service/svc-sidecar created
```

Check the Pod status.

```
$ kubectl get pods
NAME            READY      STATUS       RESTARTS      AGE
git-sync        2/2        Running      0             12s
```

The Pod is running, and both containers are ready.

The following command is more complicated, but shows that Kubernetes deployed the **ctr-sync** container as an init container and the **ctr-web** container as a regular container.

```
$ kubectl get pod -o "custom-columns="\
  "NAME:.metadata.name,"\
  "INIT:.spec.initContainers[*].name,"\
  "CONTAINERS:.spec.containers[*].name"
```

```
NAME          INIT        CONTAINERS
git-sync      ctr-sync    ctr-web
```

Describe the Pod and view the `Events` section to confirm Kubernetes started the **ctr-sync** sidecar Pod before the **ctr-web** app Pod. I've trimmed the output to show the relevant parts.

```
$ kubectl describe pod git-sync

Name:             git-sync
Status:           Running
<Snip>
Events:
  Type      Reason      Age     From        Message
  ----      ------      ----    ----        -------
  Normal    Created     19s     kubelet     Created container ctr-sync
  Normal    Started     19s     kubelet     Started container ctr-sync
  Normal    Pulling     18s     kubelet     Pulling image "nginx"
  Normal    Created     17s     kubelet     Created container ctr-web
  Normal    Started     17s     kubelet     Started container ctr-web
```

The timestamps show that Kubernetes started the **ctr-sync** sidecar container before it started the **ctr-web** app container.

4. Connect to the app

Once the Pod is running, you can run a `kubectl get svc svc-sidecar` command and copy the value from the `EXTERNAL-IP` column. This will be a public IP or DNS name if you're running in the cloud. You'll need to use `localhost` if you're running a local Docker Desktop cluster.

Paste the IP or DNS name into a new browser tab to see the web page. It will display **This is version 1.0**.

5. Make a change to your fork of the GitHub repo

You **must** complete this step against **your forked repo.**

Go to your forked repo and edit the `index.html` file. Change the `<h1>` line to something different, then save and commit your changes.

6. Verify your changes appear on the web page

Refresh the app's web page to see your updates.

Congratulations. The sidecar started before the app container and is still running. You proved this using `kubectl` commands, but you also proved it by changing the contents of your forked GitHub repo and witnessing your changes appear in the application.

Clean up

If you've been following along, you'll have the following deployed to your cluster.

Pods	Services
hello-pod	
initpod	k8sbook
git-sync	svc-sidecar

Delete them with the following commands.

```
$ kubectl delete pod hello-pod initpod git-sync
pod "hello-pod" deleted
pod "initpod" deleted
pod "git-sync" deleted

$ kubectl delete svc k8sbook svc-sidecar
service "k8sbook" deleted
service "svc-sidecar" deleted
```

You can also delete objects via their YAML files.

```
$ kubectl delete -f initsidecar.yml -f initpod.yml -f pod.yml -f
initsvc.yml
pod "git-sync" deleted
```

```
service "svc-sidecar" deleted
pod "initpod" deleted
pod "hello-pod" deleted
service "k8sbook" deleted
```

You may also want to delete your GitHub fork.

Chapter Summary

In this chapter, you learned that Kubernetes deploys all applications inside Pods. The apps can be containers, serverless functions, Wasm apps, and VMs. However, they're usually containers, so we usually refer to Pods in terms of executing *containers.*

As well as abstracting different types of applications, Pods provide a shared execution environment, advanced scheduling, application health probes, and lots more.

Pods can be single-container or multi-container, and all containers in a multi-container Pod share the Pod's networking, volumes, and memory.

You'll usually deploy Pods via higher-level workload controllers such as Deployments, Jobs, and DaemonSets. Third-party tools, such as Knative and KubeVirt, extend the Kubernetes API with custom resources and custom workload controllers that allow Kubernetes to run serverless and VM workloads.

You define Pods in declarative YAML files that you post to the API server, and the control plane schedules them to the cluster. You'll usually use `kubectl apply` to post the YAML manifests to the API server, and the scheduler will deploy them.

5: Virtual clusters with Namespaces

Namespaces are a way of dividing a Kubernetes cluster into multiple *virtual clusters.*

This chapter sets the foundation for Namespaces, gets you up to speed with creating and managing them, and introduces some use cases. You'll see them in action in future chapters.

I've divided the chapter as follows:

- Intro to Namespaces
- Namespace use cases
- Default Namespaces
- Creating and managing Namespaces
- Deploying to Namespaces

Intro to Namespaces

The first thing to know is that *Kubernetes Namespaces* are not the same as *Linux kernel namespaces*.

- *Kernel namespaces* partition operating systems into virtual operating systems called *containers*
- *Kubernetes Namespaces* partition Kubernetes clusters into virtual clusters called *Namespaces*

   Note: We'll capitalize *Namespace* when referring to Kubernetes Namespaces. This follows the pattern of capitalizing Kubernetes API resources and clarifies that we're referring to Kubernetes Namespaces, not kernel namespaces.

It's also important to know that Namespaces are a form of *soft isolation* and enable *soft multi-tenancy*. For example, you can create Namespaces for your **dev, test,** and **qa** environments and apply different quotas and policies to each. However, a compromised workload in one Namespace can easily impact workloads in other Namespaces.

The following command shows whether objects are *namespaced* or not. As you can see, most objects are namespaced, meaning you can deploy them to specific namespaces with custom policies and quotas. Objects that aren't namespaced, such as Nodes and PersistentVolumes, are *cluster-scoped* and you cannot deploy them to specific Namespaces.

```
$ kubectl api-resources
NAME                      SHORTNAMES    ...     NAMESPACED   KIND
nodes                     no                    false        Node
persistentvolumeclaims    pvc                   true
PersistentVolumeClaim
persistentvolumes         pv                    false
PersistentVolume
pods                      po                    true         Pod
podtemplates                                    true         PodTemplate
replicationcontrollers    rc                    true
ReplicationController
resourcequotas            quota                 true         ResourceQuota
secrets                                         true         Secret
serviceaccounts           sa                    true
ServiceAccount
services                  svc                   true         Service
<Snip>
```

Unless you specify otherwise, Kubernetes deploys objects to the `default` Namespace.

Namespace use cases

Namespaces are a way for multiple tenants to share the same cluster.

Tenant is a loose term that can refer to individual applications, different teams or departments, and even external customers. How you implement Namespaces and what you consider as *tenants* is up to you, but it's most common to use Namespaces to divide clusters for use by tenants within the same organization. For example, you might divide a production cluster into the following three Namespace to match your organizational structure:

- finance
- hr
- corporate-ops

You'd deploy Finance apps to the **finance** Namespace, HR apps to the **hr** Namespace, and Corporate apps to the **corporate-ops** Namespace. Each Namespace can have its own users, permissions, resource quotas, and policies.

Using Namespaces to divide a cluster among external tenants isn't as common. This is because they only provide soft isolation and cannot prevent compromised workloads from escaping the Namespace and impacting workloads in other Namespaces. At the time of writing, the most common way of strongly isolating tenants is to run them on their own clusters and their own hardware.

Figure 5.1 shows a cluster on the left using Namespaces for soft multi-tenancy. All apps on this cluster share the same nodes and control plane, and compromised workloads can impact both Namespaces. The two clusters on the right provide strong isolation by implementing two separate clusters, each on dedicated hardware.
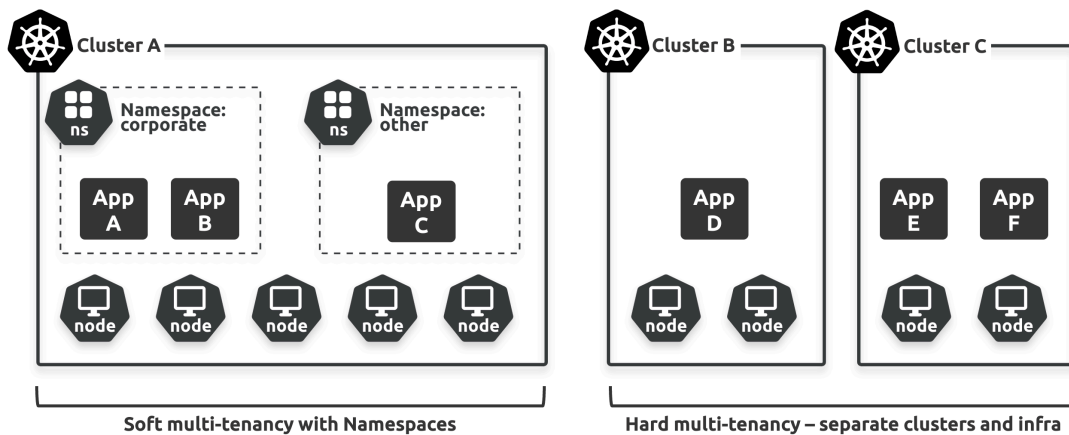
Figure 5.1 - Soft and hard isolation

In summary, Namespaces are lightweight and easy to manage but only provide soft isolation. Running multiple clusters costs more and introduces more management overhead, but it offers strong isolation.

Default Namespaces

Every Kubernetes cluster has a set of pre-created Namespaces.

Run the following command to list yours.

```
$ kubectl get namespaces
NAME              STATUS     AGE
default           Active     2d
kube-system       Active     2d
kube-public       Active     2d
kube-node-lease   Active     2d
```

The `default` Namespace is where new objects go if you don't specify a Namespace when creating them. `kube-system` is where control plane components such as the internal DNS service and the metrics server run. **kube-public** is for objects that need to be readable by anyone. And last but not least, **kube-node-lease** is used for node heartbeats and managing node leases.

Run a `kubectl describe` to inspect one of the Namespaces on your cluster. You can substitute `namespace` with `ns` when working with `kubectl`.

```
$ kubectl describe ns default
Name:           default
Labels:         kubernetes.io/metadata.name=default
Annotations:    <none>
Status:         Active
No resource quota.
No LimitRange resource.
```

You can also add `-n` or `--namespace` to `kubectl` commands to filter results against a specific Namespace.

Run the following command to list all Service objects in the `kube-system` Namespace. Your output might be different.

```
$ kubectl get svc --namespace kube-system
NAME                  TYPE          CLUSTER-IP      EXTERNAL-IP
PORT(S)
kube-dns              ClusterIP     10.43.0.10      <none>
53/UDP,53/TCP,9153...
metrics-server        ClusterIP     10.43.4.203     <none>
443/TCP
traefik-prometheus    ClusterIP     10.43.49.213    <none>
9100/TCP
traefik               LoadBalancer  10.43.222.75    <pending>
80:31716/TCP,443:31...
```

You can also use the `--all-namespaces` flag to return objects from all Namespaces.

Creating and managing Namespaces

In this section, you'll see how to create, inspect, and delete Namespaces.

You'll need a clone of the book's GitHub repo if you want to follow along. And you'll need to be on the **2025** branch.

```
$ git clone https://github.com/nigelpoulton/TKB.git
<Snip>

$ cd TKB

$ git fetch origin

$ git checkout -b 2025 origin/2025
```

You'll also need to run all commands from the `namespaces` directory.

Namespaces are first-class resources in the `core v1` API group. This means they're stable, well-understood, and have been around for a long time. It also means you can work with them imperatively (via the CLI) and declaratively (via config files). We'll do both.

Run the following imperative command to create a new Namespace called **hydra**.

```
$ kubectl create ns hydra
namespace/hydra created
```

Now, create one declaratively from the `shield-ns.yml` YAML file. It's a simple file defining a single Namespace called `shield`.

```
kind: Namespace
apiVersion: v1
metadata:
  name: shield
  labels:
    env: marvel
```

Create it with the following command.

```
$ kubectl apply -f shield-ns.yml
namespace/shield created
```

List all Namespaces to see the two new ones you created.

```
$ kubectl get ns
NAME            STATUS    AGE
<Snip>
```

```
hydra          Active    49s
shield         Active    3s
```

If you know anything about the Marvel Cinematic Universe, you'll know Shield and Hydra are bitter enemies and should never share the same cluster with only Namespaces separating them.

Delete the **hydra** Namespace.

```
$ kubectl delete ns hydra
namespace "hydra" deleted
```

Configure kubectl for a specific Namespace

When working with Namespaces, you'll quickly realize it's painful having to add the `-n` or `--namespace` flag on all your `kubectl` commands. A better way is to set your *kubeconfig* to automatically run commands against a specific Namespace.

Run the following command to configure your kubeconfig to run all future `kubectl` commands against the `shield` Namespace.

```
$ kubectl config set-context --current --namespace shield
Context "tkb" modified.
```

Run a few simple `kubectl get` commands to test it works. The `shield` Namespace is empty, so your commands won't return any objects.

Deploying objects to Namespaces

As previously mentioned, most objects are Namespaced, and Kubernetes deploys new objects to the `default` Namespace unless you specify otherwise.

There are two ways to deploy objects to specific Namespaces:

- Imperatively
- Declaratively

To do it imperatively, add the `-n` or `--namespace` flag to commands. To do it declaratively, you specify the Namespace in the object's YAML manifest.

Let's use the declarative method to deploy an app to the `shield` Namespace.

The application is defined in the `app.yml` file in the `namespaces` folder of the book's GitHub repo. It defines three objects: a ServiceAccount, a Service, and a Pod. The following YAML extract shows all three objects targeted at the `shield` Namespace.

Don't worry if you don't understand everything in the YAML. You only need to know it defines three objects and targets each one at the `shield` Namespace.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  namespace: shield      <<---- Namespace
  name: default
---
apiVersion: v1
kind: Service
metadata:
  namespace: shield      <<---- Namespace
  name: the-bus
spec:
  type: LoadBalancer
  ports:
  - port: 8080
    targetPort: 8080
  selector:
    env: marvel
---
apiVersion: v1
kind: Pod
metadata:
  namespace: shield      <<---- Namespace
  name: triskelion
<Snip>
```

Deploy it with the following command. Don't worry if you get a warning about a missing annotation for the ServiceAccount.

```
$ kubectl apply -f app.yml
serviceaccount/default configured
service/the-bus configured
pod/triskelion created
```

Run a few commands to verify all three objects are in the `shield` Namespace. You don't need to add the `-n shield` flag if you configured `kubectl` to automatically target the `shield` Namespace.

```
$ kubectl get pods -n shield
NAME          READY   STATUS    RESTARTS   AGE
triskelion    1/1     Running   0          48s

$ kubectl get svc -n shield
NAME      TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
the-bus   LoadBalancer   10.43.30.174   localhost     8080:31112/TCP
52s
```

Now that you've deployed the app, point your browser or a `curl` command to the value in the `EXTERNAL-IP` column on port `8080`. Some Docker Desktop clusters incorrectly display a `172` IP address in the `EXTERNAL-IP`. You'll need to substitute this with `localhost` and connect to `localhost:8080`.

```
$ curl localhost:8080

<!DOCTYPE html>
<html>
<head>
    <title>AOS</title>
    <Snip>
```

Congratulations. You've created a Namespace and deployed an app to it. Connecting to the app is exactly the same as connecting to an app in the `default` Namespace.

Clean up

The following commands will clean up your cluster and revert your kubeconfig to use the `default` Namespace.

Delete the `shield` Namespace. This will automatically delete the Pod, Service, and ServiceAccount you deployed to it. It may take a few seconds for the command to complete.

```
$ kubectl delete ns shield
namespace "shield" deleted
```

Reset your kubeconfig so it uses the `default` Namespace. If you don't do this, future commands will run against the deleted `shield` Namespace and return no results.

```
$ kubectl config set-context --current --namespace default
Context "tkb" modified.
```

Chapter Summary

In this chapter, you learned that Kubernetes uses Namespaces to divide clusters for resource and accounting purposes. Each Namespace can have its own users, RBAC rules, and resource quotas, and you can selectively apply policies to Namespaces. However, they're not a strong workload isolation boundary, so you cannot use them for hard multi-tenancy.

If you don't specify one at deploy time, Kubernetes deploys objects to the `default` Namespace.

6: Kubernetes Deployments

This chapter shows you how to use *Deployments* to add cloud-native features such as self-healing, scaling, rolling updates, and versioned rollbacks to stateless apps on Kubernetes.

I've organized the chapter as follows:

- Deployment theory
- Create a Deployment
- Manually scale an app
- Perform a rollout
- Perform a rollback

Deployment theory

Deployments are the most popular way of running *stateless* apps on Kubernetes. They add self-healing, scaling, rollouts, and rollbacks.

Consider a quick example.

Assume you have a requirement for a web app that needs to be resilient, scale on demand, and be frequently updated. You write the app, containerize it, and define it as a Pod. However, before sending the Pod to Kubernetes, you wrap it in a Deployment so it gets the resiliency, scaling, and update capabilities. You then post the Deployment to Kubernetes, where the Deployment controller deploys the Pod.

At this point, your cluster is running a single Deployment managing a single Pod.

If the Pod fails, the Deployment controller replaces it with a new one. If demand increases, the Deployment controller can scale the

app by deploying more identical Pods. When you update the app, the Deployment controller deletes the old Pods and replaces them with new ones.

Assume you add a shopping cart service that also needs to be resilient, scalable, and regularly updated. You'd containerize this, define it in its own Pod, wrap the Pod in its own Deployment, and deploy it to the cluster.

At this point, you'd have two Deployments managing two different microservices.

Figure 6.1 shows a similar setup with the Deployment controller watching and managing two Deployments. The **web** Deployment manages four identical web server Pods, and the **cart** Deployment manages two identical shopping cart Pods.
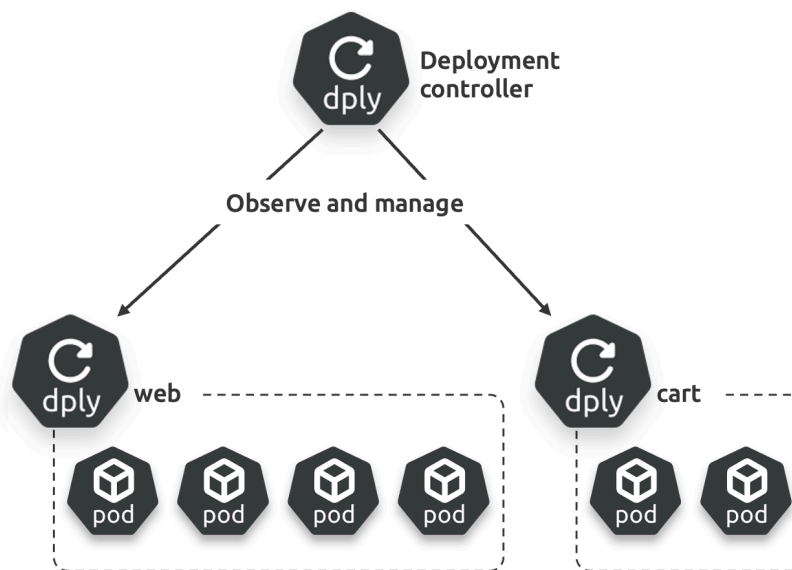
Figure 6.1 - Deployments

Under the hood, Deployments follow standard Kubernetes architecture comprising:

1. A *resource*
2. A *controller*

At the highest level, *resources* **define** objects and *controllers* **manage** them.

The Deployment *resource* exists in the `apps/v1` [API](#) and defines all supported attributes and capabilities.

The Deployment *controller* is a control plane service that watches Deployments and reconciles observed state with desired state.

Deployments and Pods

Every Deployment manages one or more identical Pods.

For example, an application comprising a web service and a shopping cart service will need two Deployments — one for managing the web Pods and the other for managing the shopping cart Pods. Figure 6.1 showed the **web** Deployment managing four identical web Pods and the **cart** Deployment managing two identical shopping cart Pods.

Figure 6.2 shows a Deployment YAML file requesting four replicas of a single Pod. If you increase the replica count to six, it will deploy and manage two more identical Pods.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  replicas: 4
  ...
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
template:
  spec:
    containers:
    - name: web-app
      image: my-web-app
      ports:
      - containerPort: 8080
```

Give me 4 replicas of this Pod
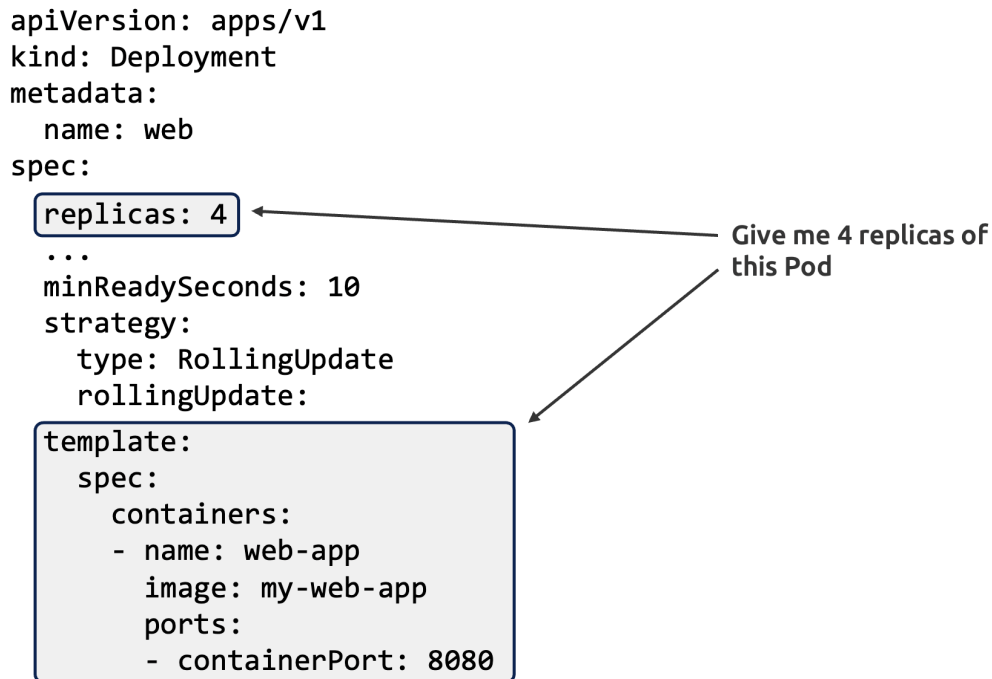
Figure 6.2

Notice how the Pod is defined in a `template` embedded in the Deployment YAML. You'll see this referred to as the *Pod template.*

Deployments and ReplicaSets

We've repeatedly said that Deployments add self-healing, scaling, rollouts, and rollbacks. However, behind the scenes, there's a different resource called a *ReplicaSet* that provides self-healing and scaling.

Figure 6.3 shows the overall architecture of containers, Pods, ReplicaSets, and Deployments. It also shows how they map into a Deployment YAML. In the diagram **rs** is short for ReplicaSet.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
replicas: 2
selector:
  matchLabels:
    app: hello-world
template:
  metadata:
    labels:
      app: hello-world
spec:
  containers:
  - name: hello-pod
    image: nigelpoulton/k8sbook:1.0
    ports:
    - containerPort: 8080
```
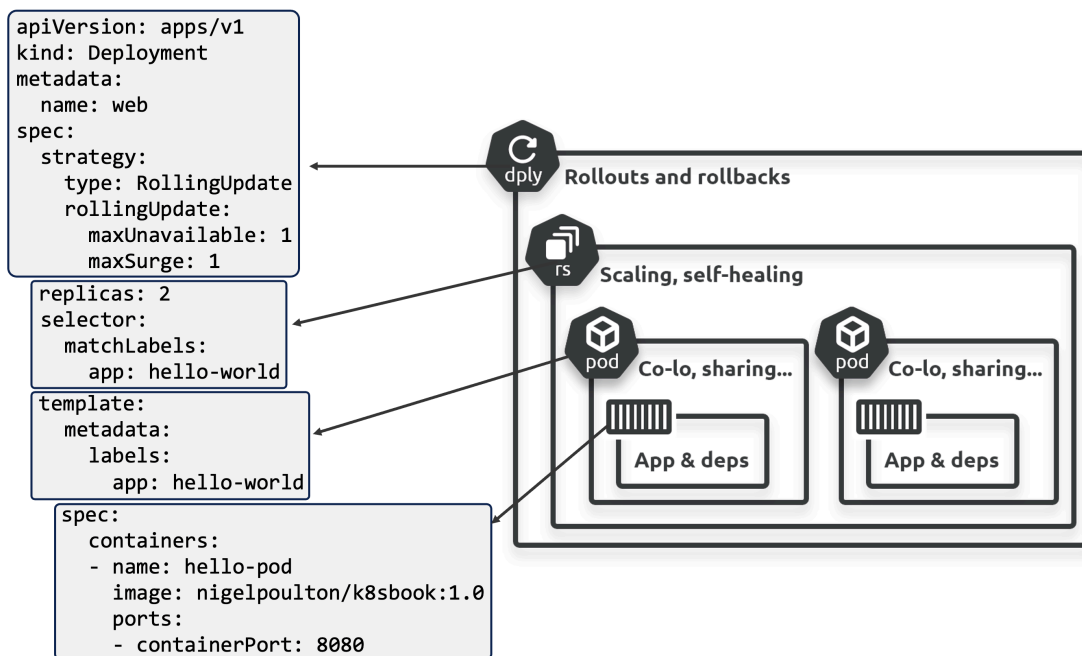
Figure 6.3

Posting this Deployment YAML to the cluster will create a Deployment, a ReplicaSet, and two identical Pods running identical containers. The Pods are managed by the ReplicaSet, which, in turn, is managed by the Deployment. However, you perform **all** management via the Deployment and never directly manage the ReplicaSet or Pods.

A quick word on scaling

It's possible to scale your apps manually, and we'll see how to do that shortly. However, Kubernetes has several *autoscalers* that automatically scale your apps and infrastructure. Some of them include:

- The Horizontal Pod Autoscaler
- The Vertical Pod Autoscaler
- The Cluster Autoscaler

The Horizontal Pod Autoscaler (HPA) adds and removes **Pods** to meet current demand. It's automatically installed on most clusters and widely used.

The Cluster Autoscaler (CA) adds and removes cluster **nodes** so you always have enough to run all scheduled Pods. This is also installed by default and widely used.

The Vertical Pod Autoscaler (VPA) increases and decreases the CPU and memory allocated to running Pods to meet current demand. It isn't installed by default, has several known limitations, and is less widely used. Current implementations work by deleting the existing Pod and replacing it with a new one every time it scales the Pods resources. This is disruptive and can even result in Kubernetes scheduling the new Pod to a different node. Work is underway to enable *in-place updates* to live Pods, but it's currently an early alpha feature.

Community projects like karmada take things further by allowing you to scale apps across multiple clusters.

Let's consider a quick example using the HPA and CA.

You deploy an application to your cluster and configure an HPA to autoscale the number of application Pods between two and ten. Demand increases, and the HPA asks the scheduler to increase the number of Pods from two to four. This works, but demand continues rising, and the HPA asks the scheduler for another two Pods. However, the scheduler can't find a node with sufficient resources and marks the two new Pods as *pending*. The CA notices the pending Pods and dynamically adds a new cluster node. Once the node joins the cluster, the scheduler assigns the pending Pods to it.

The process works the same for scaling down. For example, the HPA reduces the number of Pods when demand decreases. This may trigger the CA to reduce the number of cluster nodes. When

removing a cluster node, Kubernetes evicts all Pods on the node and replaces them with new Pods on surviving nodes.

You'll sometimes hear people refer to *multi-dimensional autoscaling.* This is jargon for combining multiple scaling methods — scaling Pods and nodes, or scaling apps horizontally (adding more Pods) and vertically (adding more resources to existing Pods).

It's all about the *state*

Before going any further, it's vital that you understand the following concepts. If you already know them, you can skip to the *Rolling updates with Deployments* section.

- Desired state
- Observed state (sometimes called *actual state* or *current state*)
- Reconciliation

Desired state is what you *want, observed state* is what you *have*, and the goal is for them to always match. When they don't match, a controller starts a *reconciliation* process to bring observed state into sync with desired state.

The *declarative model* is how we declare a desired state to Kubernetes without telling Kubernetes *how* to implement it. You leave the *how* up to Kubernetes.

Declarative vs Imperative

The declarative model *describes* an end goal — you tell Kubernetes what you want. The imperative model requires long lists of commands that tell Kubernetes *how* to reach the end goal.

The following analogy will help:

- *Declarative:* Give me a chocolate cake to feed ten people.

- *Imperative:* Drive to store. Buy eggs, milk, flour, cocoa powder… Drive home. Preheat the oven. Mix the ingredients. Place in a cake tin. If a fan-assisted oven, place the cake in the oven for 30 minutes. If not a fan-assisted oven, place the cake in the oven for 40 minutes. Set a timer. Remove from the oven when the timer expires and turn the oven off. Leave to stand until cool. Add frosting.

The declarative model is simpler and leaves the *how* up to Kubernetes. The imperative model is much more complex as you need to provide all the steps and commands that will *hopefully* achieve an end goal — in this case, making a chocolate cake for ten people.

Let's look at a more concrete example.

Assume you have an application with two microservices — a front-end and a back-end. You anticipate needing five front-end replicas and two back-end replicas.

Taking the declarative approach, you write a simple YAML file requesting five front-end Pods listening externally on port `80`, and two back-end Pods listening internally on port `27017`. You then give the file to Kubernetes and sit back while Kubernetes makes it happen. It's a beautiful thing.

The opposite is the imperative model. This is usually a long list of complex instructions with no concept of desired state. And, making things worse, imperative instructions can have endless potential variations. For example, the commands to pull and start *containerd* containers are different from the commands to pull and start *CRI-O* containers. This results in more work and is prone to more errors, and because it's not declaring a desired state, there's no self-healing. It's devastatingly ugly.

Kubernetes supports both models but strongly prefers the declarative model.

Note: containerd and CRI-O are container runtimes that run on Kubernetes worker nodes and perform low-level tasks such as starting and stopping containers.

Controllers and reconciliation

Reconciliation is fundamental to *desired state*.

For example, Kubernetes implements ReplicaSets as a background controller running in a reconciliation loop, ensuring the correct number of Pod replicas are always present. If there aren't enough Pods, the ReplicaSet adds more. If there are too many, it terminates some.

Assume a scenario where your desired state is ten replicas, but only eight are present. It makes no difference if this is due to failures or if an autoscaler has requested an increase. Either way, the ReplicaSet controller creates two new replicas to sync observed state with desired state. And the best part is that it does it without needing help from you!

The exact same reconciliation process enables self-healing, scaling, rollouts, and rollbacks.

Let's take a closer look at rolling updates and rollbacks.

Rolling updates with Deployments

Deployments are amazing at zero-downtime rolling updates (rollouts). But they work best if you design your apps to be:

1. Loosely coupled via APIs
2. Backward and forward compatible

Both are hallmarks of modern cloud-native microservices apps and work as follows.

Your microservices should always be loosely coupled and only communicate via well-defined APIs. Doing this means you can update and patch any microservice without having to worry about impacting others — all connections are via formalized APIs that expose documented interfaces and hide specifics.

Ensuring releases are backward and forward-compatible means you can perform independent updates without caring which versions of clients are consuming your service. A simple non-tech analogy is a car. Cars expose a standard driving *"API"* that includes a steering wheel and foot pedals. As long as you don't change this "API", you can re-map the engine, change the exhaust, and get bigger brakes, all without the driver having to learn any new skills.

With these points in mind, zero-downtime rollouts work like this.

Assume you're running five replicas of a stateless microservice. Clients can connect to any of the five replicas as long as all clients connect via backward and forward-compatible APIs. To perform a rollout, Kubernetes creates a new replica running the new version and terminates one running the old version. At this point, you've got four replicas on the old version and one on the new. This process repeats until all five replicas are on the new version. As the app is stateless and multiple replicas are up and running, clients experience no downtime or interruption of service.

There's a lot more going on behind the scenes, so let's take a closer look.

Each microservice is built as a container and wrapped in a Pod. You then wrap the Pod for each microservice in its own Deployment for self-healing, scaling, and rolling updates. Each Deployment describes all the following:

- Number of Pod replicas
- Container images to use
- Network ports

- How to perform rolling updates

You post Deployment YAML files to the API Server, and the ReplicaSet controller ensures the correct number of Pods get scheduled. It also watches the cluster, ensuring observed state matches desired state. A Deployment sits above the ReplicaSet, governing its configuration and adding mechanisms for rollouts and rollbacks.

All good so far.

Now, assume you're exposed to a known vulnerability and need to release an update with the fix. To do this, you update the **same Deployment YAML file** with the new Pod spec and re-post it to the API server. This updates the existing Deployment object with a new desired state requesting the same number of Pods, but all running the newer version containing the fix.

At this point, observed state no longer matches desired state — you've got five old Pods, but you want five new ones.

To reconcile, the Deployment controller creates a new ReplicaSet defining the same number of Pods but running the newer version. You now have two ReplicaSets — the original one for the Pods with the old version and the new one for the Pods with the new version. The Deployment controller systematically increments the number of Pods in the new ReplicaSet as it decrements the number in the old ReplicaSet. The net result is a smooth, incremental rollout with zero downtime.

The same process happens for future updates — you keep updating the same Deployment manifest, which you should store in a version control system.

Figure 6.4 shows a Deployment that you've updated once. The initial release created the ReplicaSet on the left, and the update created the one on the right. You know the update is complete

because the ReplicaSet on the left isn't managing any Pods, whereas the one on the right is managing three live Pods.
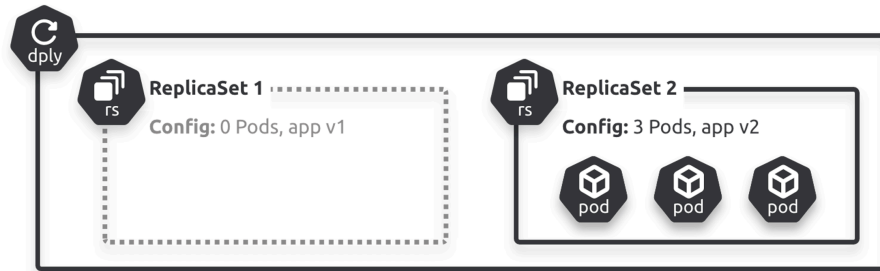
Figure 6.4

In the next section, you'll see why it's important that the old ReplicaSet still exists with its configuration intact.

Rollbacks

As you saw in Figure 6.4, older ReplicaSets are wound down and no longer manage any Pods. However, their configurations still exist and can be used to easily roll back to earlier versions.

The rollback process is the opposite of a rollout — wind an old ReplicaSet up while the current one winds down.

Figure 6.5 shows the same app rolled back to the previous config with the earlier ReplicaSet on the left managing all the Pods.
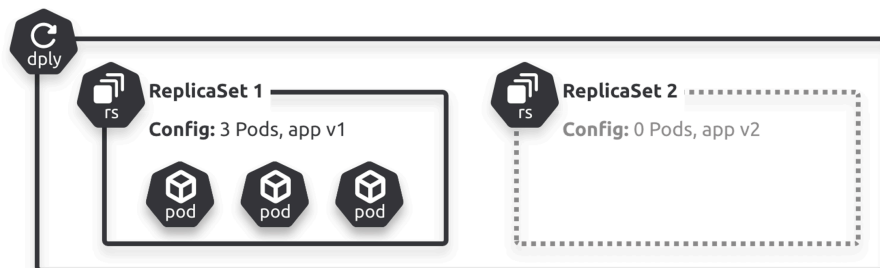
Figure 6.5

But that's not the end. Kubernetes gives you fine-grained control over rollouts and rollbacks. For example, you can insert delays, control the pace and cadence of releases, and even probe the health and status of updated replicas.

But talk is cheap. Let's see Deployments in action.

Create a Deployment

You'll need the lab files from the book's GitHub repo if you want to follow along. Run the following commands if you haven't already got them.

```
$ git clone https://github.com/nigelpoulton/TKB.git
Cloning into 'TKB'...

$ cd TKB

$ git fetch origin

$ git checkout -b 2025 origin/2025
```

It's important that you work from the `2025` branch and run all commands from the `deployments` folder.

```
$ cd deployments
```

We'll use the `deploy.yml` file, as shown in the following snippet. It defines a single-container Pod wrapped in a Deployment. I've annotated it and snipped it to draw your attention to the parts we'll focus on.

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: hello-deploy        <<---- Deployment name (must be valid DNS
name)
spec:
  replicas: 10              <<---- Number of Pod replicas to deploy &
manage
  selector:
```

```
   matchLabels:
     app: hello-world
 revisionHistoryLimit: 5
 progressDeadlineSeconds: 300
 minReadySeconds: 10
 strategy:                    <<---- This block defines rolling update
settings
   type: RollingUpdate
   rollingUpdate:
     maxUnavailable: 1
     maxSurge: 1
 template:                    <<---- Below here is the Pod template
   metadata:
     labels:
       app: hello-world
   spec:
     containers:
     - name: hello-pod
       image: nigelpoulton/k8sbook:1.0
       ports:
       - containerPort: 8080
```

There's a lot going on in the file, so let's explain the most important parts.

The first two lines tell Kubernetes to create a Deployment object based on the version of the Deployment resource defined in the `apps/v1` API.

The `metadata` section names the Deployment **hello-deploy**. You should always give objects valid DNS names. This means you should only use alphanumerics, the dot, and the dash in object names.

Most of the action happens in the `spec` section.

`spec.replicas` asks for ten Pod replicas. In this case, the ReplicaSet controller will create ten replicas of the Pod defined in the `spec.template` section.

`spec.selector` is a list of labels the Deployment and ReplicaSet controllers look for when deciding which Pods they manage. This

label selector has to match the Pod labels in the Pod template block (`spec.template.metadata.labels`). In this example, both specify the `app=hello-world` label.

`spec.revisionHistoryLimit` tells Kubernetes to keep the previous five ReplicaSets so you can roll back to the last five versions. Keeping more gives you more rollback options, but keeping too many can bloat the object and cause problems on large clusters with lots of releases.

`spec.progressDeadlineSeconds` tells Kubernetes to give each new replica a five-minute start window before reporting the replica as stalled. All replicas get their own window, meaning each replica has its own five-minute window to come up properly (progress).

`spec.strategy` tells the Deployment controller how to update the Pods when performing a rollout. We'll explain these settings later in the chapter when you perform a rollout.

Finally, everything below `spec.template` defines the Pod this Deployment will manage. This example defines a single-container Pod using the `nigelpoulton/k8sbook:1.0` image.

Run the following command to create the Deployment on your cluster.

> **Note:** All `kubectl` commands include the necessary authentication tokens from your kubeconfig file.

```
$ kubectl apply -f deploy.yml
deployment.apps/hello-deploy created
```

At this point, the Deployment configuration is persisted to the cluster store as a record of intent, and Kubernetes has scheduled ten replicas to healthy worker nodes. The Deployment and

ReplicaSet controllers are also running in the background, watching the state of play and eager to perform their reconciliation magic.

Feel free to run a `kubectl get pods` command to see the ten Pods.

Inspecting Deployments

You can use the normal `kubectl get` and `kubectl describe` commands to see Deployment and ReplicaSet details.

```
$ kubectl get deploy hello-deploy
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
hello-deploy   10/10   10           10          105s

$ kubectl describe deploy hello-deploy
Name:                   hello-deploy
Namespace:              default
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=hello-world
Replicas:               10 desired | 10 updated | 10 total | 10
available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        10
RollingUpdateStrategy:  1 max unavailable, 1 max surge
Pod Template:
  Labels:  app=hello-world
  Containers:
   hello-pod:
    Image:         nigelpoulton/k8sbook:1.0
    Port:          8080/TCP
<SNIP>
OldReplicaSets:  <none>
NewReplicaSet:   hello-deploy-54f5d46964 (10/10 replicas created)
<Snip>
```

I've trimmed the outputs for readability, but take a minute to examine them, as they contain a lot of information that will reinforce what you've learned.

As mentioned earlier, Deployments automatically create associated ReplicaSets. Verify this with the following command.

```
$ kubectl get rs
NAME                       DESIRED   CURRENT   READY   AGE
```

```
hello-deploy-54f5d46964     10          10          10          3m45s
```

You only have one ReplicaSet as you've only performed an initial rollout. However, its name matches the Deployment's name with a hash added to the end. This is a crypto-hash of the Pod template section of the Deployment manifest (everything below `spec.template`). You'll see this shortly, but making changes to the Pod template section initiates a rollout and creates a new ReplicaSet with a hash of the updated Pod template.

You can get more detailed information about the ReplicaSet with a `kubectl describe` command. Your ReplicaSet will have a different name.

```
$ kubectl describe rs hello-deploy-54f5d46964
Name:           hello-deploy-54f5d46964
Namespace:      default
Selector:       app=hello-world,pod-template-hash=54f5d46964
Labels:         app=hello-world
                pod-template-hash=54f5d46964
Annotations:    deployment.kubernetes.io/desired-replicas: 10
                deployment.kubernetes.io/max-replicas: 11
                deployment.kubernetes.io/revision: 1
Controlled By:  Deployment/hello-deploy        <<---- Deployment that
own this ReplicaSet
Replicas:       10 current / 10 desired
Pods Status:    10 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=hello-world
           pod-template-hash=54f5d46964
  Containers:
   hello-pod:
    Image:          nigelpoulton/k8sbook:1.0
    Port:           8080/TCP
<Snip>
```

Notice how the output is similar to the Deployment output. This is because the Deployment controls the ReplicaSet's configuration, meaning ReplicaSet info gets displayed in the output of Deployment commands. The ReplicaSet's status (observed state) also gets rolled up into the Deployment status.

Accessing the app

The Deployment is running, and you've got ten replicas. However, you need a Kubernetes Service object to be able to connect to the app. We'll cover Services in the next chapter, but for now, it's enough to know that they provide network access to Pods.

The following YAML is from the `lb.yml` file in the `deployments` folder. It defines a Service that works with the Pods you just deployed.

```
apiVersion: v1
kind: Service
metadata:
  name: lb-svc
  labels:
    app: hello-world
spec:
  type: LoadBalancer
  ports:
  - port: 8080
    protocol: TCP
  selector:
    app: hello-world          <<---- Send traffic to Pods with this
label
```

Deploy it with the following command.

```
$ kubectl apply -f lb.yml
service/lb-svc created
```

Verify the Service configuration and copy the value in the `EXTERNAL-IP` column. Some Docker Desktop clusters incorrectly return a `172` IP address in the `EXTERNAL-IP` column, substitute this for `localhost`.

```
$ kubectl get svc lb-svc
NAME      TYPE           CLUSTER-IP       EXTERNAL-IP    PORT(S)
lb-svc    LoadBalancer   10.100.247.251   localhost      8080:31086/TCP
```

Open a new browser tab and connect to the value in the `EXTERNAL-IP` field on port `8080`. This will be `localhost:8080` if you're on a local

Docker Desktop cluster. It'll be a public IP or DNS name on port `8080` if your cluster is in the cloud.

Figure 6.6 shows a browser accessing the app on `localhost:8080`.
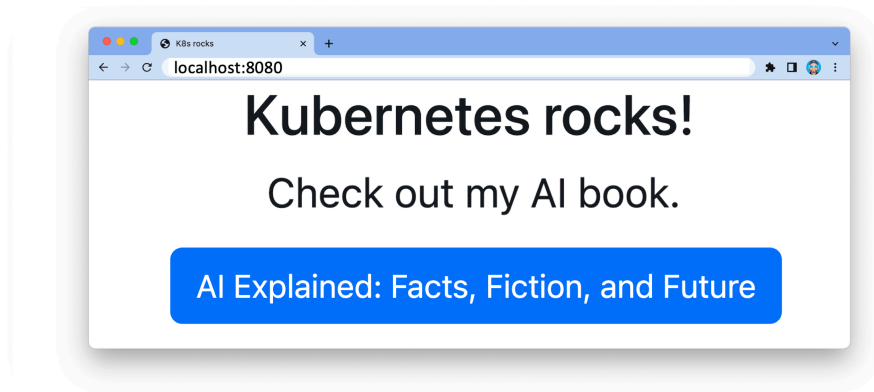
Figure 6.6

Manually scale the app

You can manually scale Deployments in two ways:

- Imperatively
- Declaratively

The imperative method uses the **`kubectl scale`** command, whereas the declarative method requires you to update the Deployment YAML file and re-post it to the cluster. We'll show you both, but Kubernetes prefers the declarative method.

Verify that you currently have ten replicas.

```
$ kubectl get deploy hello-deploy
NAME            READY    UP-TO-DATE    AVAILABLE    AGE
hello-deploy    10/10    10            10           28m
```

Run the following commands to imperatively scale down to five replicas and verify the operation worked.

```
$ kubectl scale deploy hello-deploy --replicas 5
deployment.apps/hello-deploy scaled

$ kubectl get deploy hello-deploy
NAME            READY    UP-TO-DATE    AVAILABLE    AGE
hello-deploy    5/5      5             5            29m
```

Congratulations, you've successfully scaled the Deployment down to five replicas. However, there's a potential problem…

The current state of your environment no longer matches your declarative manifest — there are five replicas on your cluster, but your Deployment YAML still defines 10. This can cause issues when using the YAML file to perform future updates. For example, updating the image version in the YAML file and re-posting it to the cluster will also change the number of replicas back to 10, which you might not want. For this reason, you should always keep your YAML manifests in sync with your live environment, and the easiest way to do this is by making all changes declaratively via your YAML manifests.

Let's re-post the YAML file to return the replica count to 10.

```
$ kubectl apply -f deploy.yml
deployment.apps/hello-deploy configured

$ kubectl get deploy hello-deploy
NAME            READY    UP-TO-DATE    AVAILABLE    AGE
hello-deploy    10/10    10            10           38m
```

You may have noticed that scaling operations are almost instantaneous. This is not the case with rolling updates which you're about to see next.

Kubernetes also has autoscalers that automatically scale Pods and infrastructure based on current demand.

Perform a rolling update

The terms *rollout, release, zero-downtime update,* and *rolling update* mean the same thing, and we'll use them interchangeably.

I've already created the new version of the app, tested it, and uploaded it to Docker Hub with the `nigelpoulton/k8sbook:2.0` tag. All that's left is for you to perform the rollout. We're ignoring real-world CI/CD workflows and version control tools to simplify the process and keep the focus on Kubernetes.

Before continuing, it's vital you understand that all *update* operations are actually **replacement** operations. When you update a Pod, you're actually deleting it and replacing it with a new one. This is because Pods are *immutable objects,* so you never change or update them after you've deployed them.

The first step is to update the image version in the `deploy.yml` file. Use your favorite editor to update the image version to `nigelpoulton/k8sbook:2.0` and save your changes.

The following trimmed output shows which line in the file to update.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deploy
spec:
  replicas: 10
  <Snip>
  template:
    <Snip>
    spec:
      containers:
      - name: hello-pod
        image: nigelpoulton/k8sbook:2.0          <<---- Update this
line to 2.0
        ports:
        - containerPort: 8080
```

The next time you post the file to Kubernetes, the Deployment controller will delete every Pod running the **1.0** version and replace them with new Pods running the **2.0** version. However, before doing that, let's look at the settings governing how Kubernetes executes the rollout.

The `spec` section of the YAML file contains all the settings that tell Kubernetes how to perform the update.

```
<Snip>
revisionHistoryLimit: 5            <<---- Keep the config from the five
previous versions
progressDeadlineSeconds: 300       <<---- Give each new replica five
minutes to start
minReadySeconds: 10                <<---- Wait 10 seconds after the
previous replica has started
strategy:
  type: RollingUpdate              <<---- Incrementally replace replicas
  rollingUpdate:
    maxUnavailable: 1              <<---- Can take one replica away
during update operation
    maxSurge: 1                    <<---- Can add one extra replica
during update operation
<Snip>
```

`revisionHistoryLimit` tells Kubernetes to keep the configs from the previous five releases for easy rollbacks.

`progressDeadlineSeconds` tells Kubernetes to give each new Pod replica a five-minute window to start properly before assuming it's failed. It's OK if they start faster.

`spec.minReadySeconds` throttles the rate at which Kubernetes replaces replicas. This configuration tells Kubernetes to wait 10 seconds between each replica. Longer waits give you a better chance of catching problems and preventing scenarios where you replace all replicas with broken ones. In the real world, you'll need to make this value large enough to trap common failures.

There is also a nested `spec.strategy` map telling Kubernetes to:

- Update using the `RollingUpdate` strategy
- Never have more than one Pod below desired state (`maxUnavailable: 1`)
- Never have more than one Pod above desired state (`maxSurge: 1`)

The desired state of this app is ten replicas. Therefore, `maxSurge: 1` means Kubernetes can go up to 11 replicas during the rollout, and `maxUnavailable: 1` allows it to go down to 9. The net result is a rollout that updates two Pods at a time (the delta between 9 and 11 is 2).

This is all great, but how does Kubernetes know which Pods to delete and replace?

Labels!

If you look closely at the `deploy.yml` file, you'll see the Deployment spec has a selector block. This is a list of labels the Deployment controller looks for when finding Pods to update during rollouts. In this example, the controller will look for Pods with the `app=hello-world` label. If you look at the Pod template towards the bottom of the file, you'll notice it creates Pods with this same label. Net result: This deployment creates Pods with the `app=hello-world` label and selects Pods with the same label when performing updates, etc.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deploy
spec:
  selector:                    <<---- The Deployment will manage all
    matchLabels:               <<---- replicas on the cluster with
      app: hello-world         <<---- this label
      <Snip>
  template:
    metadata:
      labels:
        app: hello-world       <<---- Matches the label selector above
<Snip>
```

Pods and Deployments are both immutable, meaning you cannot change the selector or labels after you create the Deployment.

Run the following command to post the updated manifest to the cluster and start the rollout.

```
$ kubectl apply -f deploy.yml
deployment.apps/hello-deploy configured
```

The rollout replaces two Pods at a time with a ten-second wait after each. This means it will take a minute or two to complete

You can monitor the progress with `kubectl rollout status.`

```
$ kubectl rollout status deployment hello-deploy
Waiting for deployment "hello-deploy" rollout... 4 out of 10 new
replicas...
Waiting for deployment "hello-deploy" rollout... 4 out of 10 new
replicas...
Waiting for deployment "hello-deploy" rollout... 6 out of 10 new
replicas...
^C
```

If you quit monitoring the progress while the rollout is still happening, you can run `kubectl get deploy` commands and see the effect of the update-related settings. For example, the following command shows that six replicas have already been updated, and you currently have nine. Nine is one less than the desired state of ten and is the result of the `maxUnavailable=1` value in the manifest.

```
$ kubectl get deploy hello-deploy
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
hello-deploy   9/10    6            9           63m
```

Pausing and resuming rollouts

You can use `kubectl` to pause and resume rollouts.

If your rollout is still in progress, pause it with the following command.

```
$ kubectl rollout pause deploy hello-deploy
deployment.apps/hello-deploy paused
```

Running a `kubectl describe` command during a paused rollout provides some interesting info.

```
$ kubectl describe deploy hello-deploy
Name:                   hello-deploy
Namespace:              default
Annotations:            deployment.kubernetes.io/revision: 2
Selector:               app=hello-world
Replicas:               10 desired | 6 updated | 11 total | 9 available
| 2 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        10
RollingUpdateStrategy:  1 max unavailable, 1 max surge
<Snip>
Conditions:
  Type           Status   Reason
  ----           ------   ------
  Available      True     MinimumReplicasAvailable
  Progressing    Unknown  DeploymentPaused
OldReplicaSets:  hello-deploy-54f5d46964 (3/3 replicas created)
NewReplicaSet:   hello-deploy-5f84c5b7b7 (6/6 replicas created)
```

The `Annotations` line shows the object is on revision 2 (revision 1 was the initial rollout and the current update is revision 2). `Replicas` shows the rollout is incomplete. The third line from the bottom shows the Deployment condition as progressing but paused. Finally, on the last two lines, you can see the ReplicaSet for the initial release is managing three replicas, and the one for the new release is managing 6.

If a scale-up event occurs during a rollout, Kubernetes will balance the additional replicas across both ReplicaSets. In this example, if the Deployment scales to 20 by adding 10 new replicas, Kubernetes will assign ~3 of the new replicas to the old ReplicaSet and ~6 to the new one.

Run the following command to resume the rollout.

```
$ kubectl rollout resume deploy hello-deploy
deployment.apps/hello-deploy resumed
```

Once complete, you can check the status with `kubectl get deploy`.

```
$ kubectl get deploy hello-deploy
NAME           READY   UP-TO-DATE    AVAILABLE    AGE
hello-deploy   10/10   10            10           71m
```

The output shows the rollout as complete — 10 Pods are up-to-date and available.

If you've followed along, refresh your browser and see the updated app. The new version includes more text and uses the book's short name on the button.

said **Kubernetes rocks!**, this one says **WebAssembly is coming!** I may change what this says in the future. The important point is that it's changed.
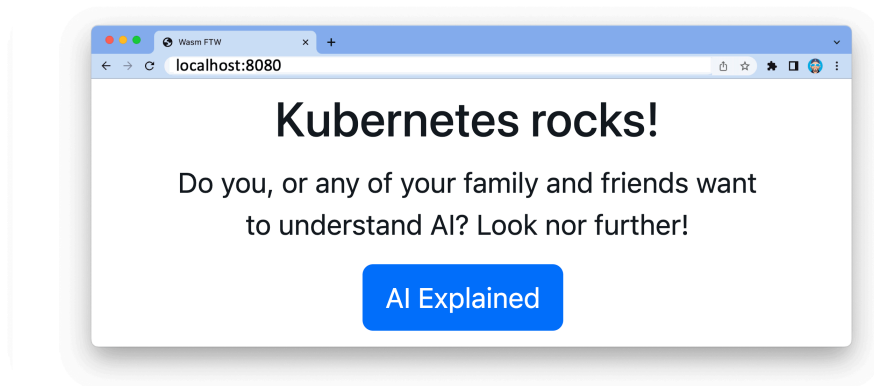
Figure 6.7

Perform a rollback

As previously mentioned, Kubernetes keeps old ReplicaSets as a documented revision history and an easy way to roll back. The following command shows the history of the Deployment with two revisions.

```
$ kubectl rollout history deployment hello-deploy
deployment.apps/hello-deploy
REVISION   CHANGE-CAUSE
1          <none>
2          <none>
```

Revision 1 was the initial release based on the **1.0** image. Revision 2 is the rollout that updated the Pods to run version **2.0** of the image.

The following command shows the two ReplicaSets associated with each revision.

```
$ kubectl get rs
NAME                        DESIRED   CURRENT   READY   AGE
hello-deploy-5f84c5b7b7     10        10        10      27m
hello-deploy-54f5d46964     0         0         0       93m
```

The next `kubectl describe` command runs against the old ReplicaSet and proves its configuration still references the old image version. Your ReplicaSets will have different names.

```
$ kubectl describe rs hello-deploy-54f5d46964
Name:          hello-deploy-54f5d46964
Namespace:     default
Selector:      app=hello-world,pod-template-hash=54f5d46964
Labels:        app=hello-world
               pod-template-hash=54f5d46964
Annotations:   deployment.kubernetes.io/desired-replicas: 10
               deployment.kubernetes.io/max-replicas: 11
               deployment.kubernetes.io/revision: 1
Controlled By: Deployment/hello-deploy
Replicas:      0 current / 0 desired
Pods Status:   0 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Containers:
   hello-pod:
    Image:       nigelpoulton/k8sbook:1.0        <<---- Still
configured with old version
    Port:        8080/TCP
    <Snip>
```

The line you're interested in is the one shown second-from-last in the book and lists the old image version. This means flipping the

Deployment back to this ReplicaSet will automatically replace all Pods with new ones running the **1.0** image.

> **Note:** Don't get confused if you hear rollbacks referred to as *updates*. That's exactly what they are. They follow the same logic and rules as an update/rollout — terminate Pods with the current image and replace them with Pods running the new image. In the case of a rollback, the new image is actually an older one.

The following example uses `kubectl rollout` to revert the application to revision 1. This is an imperative command and not recommended. However, it's convenient for quick rollbacks, just remember to update your source YAML files to reflect the changes.

```
$ kubectl rollout undo deployment hello-deploy --to-revision=1
deployment.apps "hello-deploy" rolled back
```

Although it might look like the operation is instantaneous, it isn't. Like we just said, rollbacks follow the same rules as rollouts. You can verify this and track the progress with the following `kubectl get deploy` and `kubectl rollout` commands.

```
$ kubectl get deploy hello-deploy
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
hello-deploy    9/10    6            9           96m

$ kubectl rollout status deployment hello-deploy
Waiting for deployment "hello-deploy"... 6 out of 10 new replicas have
been updated...
Waiting for deployment "hello-deploy"... 7 out of 10 new replicas have
been updated...
Waiting for deployment "hello-deploy"... 8 out of 10 new replicas have
been updated...
Waiting for deployment "hello-deploy"... 1 old replicas are pending
termination...
Waiting for deployment "hello-deploy"... 9 of 10 updated replicas are
available...
^C
```

As with the rollout, the rollback replaces two Pods at a time and waits ten seconds after each.

Congratulations. You've performed a rolling update and a successful rollback.

Rollouts and labels

You've already seen that Deployments and ReplicaSets use labels and selectors to determine which Pods they own and manage.

In earlier versions of Kubernetes, Deployments would seize ownership of static Pods if their labels matched the Deployment's label selector. However, recent versions of Kubernetes prevent this by adding a system-generated `pod-template-hash` label to Pods created by controllers.

Consider a quick example. Your cluster has five static Pods with the `app=front-end` label. You add a new Deployment requesting ten Pods with the same label. Older versions of Kubernetes would see the existing five static Pods with the same label, seize ownership of them, and only create five new ones. The net result would be ten Pods with the `app=front-end` label, all owned by the Deployment. However, the original five static Pods might be running a different app, and you might not want the Deployment managing them.

Fortunately, modern versions of Kubernetes tag all Pods created by a Deployment (ReplicaSet) with the `pod-template-hash` label. This stops higher-level controllers from seizing ownership of existing static Pods.

Look closely at the following snipped output to see how the `pod-template-hash` label connects Deployments to ReplicaSets, and ReplicaSets to Pods.

```
$ kubectl describe deploy hello-deploy
Name:       hello-deploy
```

```
<Snip>
NewReplicaSet:   hello-deploy-54f5d46964

$ kubectl describe rs hello-deploy-54f5d46964
Name:            hello-deploy-54f5d46964
<Snip>>
Selector:        app=hello-world,pod-template-hash=54f5d46964

$ kubectl get pods --show-labels
NAME                        READY    STATUS     LABELS
hello-deploy-54f5d46964..   1/1      Running    app=hello-world,pod-
template-hash=54f5d46964
hello-deploy-54f5d46964..   1/1      Running    app=hello-world,pod-
template-hash=54f5d46964
hello-deploy-54f5d46964..   1/1      Running    app=hello-world,pod-
template-hash=54f5d46964
hello-deploy-54f5d46964..   1/1      Running    app=hello-world,pod-
template-hash=54f5d46964
<Snip>
```

ReplicaSets include the `pod-template-hash` label in their label selectors, but Deployments don't. This is fine because it's actually ReplicaSets that manage Pods.

You shouldn't attempt to modify the `pod-template-hash` label.

Clean up

Use `kubectl delete -f deploy.yml` and `kubectl delete -f lb.yml` to delete the Deployment and Service created in the examples.

Chapter summary

In this chapter, you learned that Deployments are a great way to manage stateless apps on Kubernetes. They augment Pods with self-healing, scalability, rolling updates, and rollbacks.

Like Pods, Deployments are objects in the Kubernetes API, and you should work with them declaratively. They're defined in the `apps/v1` API and implement a controller running as a reconciliation loop on the control plane.

Behind-the-scenes Deployments use ReplicaSets to create, terminate, and manage the number of Pod replicas. However, you shouldn't directly create or edit ReplicaSets, you should always configure them via a Deployment.

You can manually scale Deployments by editing the Deployment YAML and re-posting it to the cluster. However, Kubernetes has autoscalers that automatically scale Deployments based on demand.

Rolling updates happen by deleting old Pods and replacing them with new ones in a controlled, organized manner.

7: Kubernetes Services

Pods are unreliable, and you should never connect directly to them. You should **always** connect via a Service.

I've organized this chapter as follows:

- Service theory
- Hands-on with Services

Service Theory

Kubernetes treats Pods as ephemeral objects and deletes them when any of the following events occur:

- Scale-down operations
- Rolling updates
- Rollbacks
- Node maintenance/evictions
- Failures

This means they're unreliable, and apps can't rely on them being there to respond to requests. Fortunately, Kubernetes has a solution — *Service objects* sit in front of one or more identical Pods and expose them via a **reliable** DNS name, IP address, and port.

Figure 7.1 shows a client connecting to an application via a Service called **app1**. The client connects to the name or IP of the Service, and the Service forwards requests to the application Pods behind it.
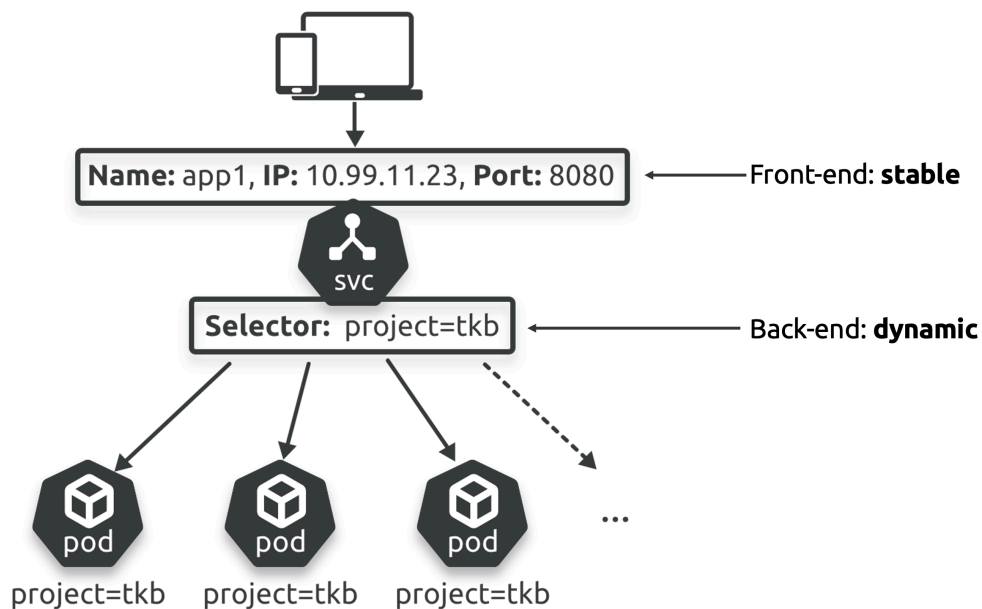
Figure 7.1 - Clients accessing Pods via a Service

Note: Services are resources in the Kubernetes API, so we capitalize the "S" to avoid confusion with other uses of the word.

Every Service has a front end and a back end. The front end includes a DNS name, IP address, and network port that Kubernetes guarantees will never change. The back end is a label selector that sends traffic to healthy Pods with matching labels. Looking back to Figure 7.1, the client sends traffic to the Service on either `app1:8080` or `10.99.11.23:8080`, and Kubernetes guarantees it will reach a Pod with the `project=tkb` label.

Services are also intelligent enough to maintain a list of healthy Pods with matching labels. This means you can scale up and down, perform rollouts and rollbacks, and Pods can even fail, but the Service will always have an up-to-date list of active healthy Pods.

Labels and loose coupling

Services use *labels* and *selectors* to know which Pods to send traffic to. This is the same technology that tells Deployments which Pods they manage.

Figure 7.2 shows a Service selecting Pods with the `project=tkb` and `zone=prod` labels.
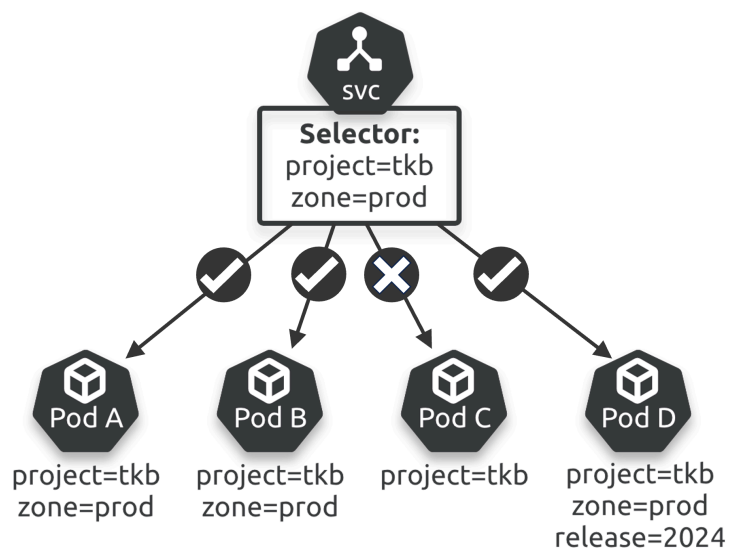
Figure 7.2 - Services and labels

In this example, the Service sends traffic to **Pod A, Pod B**, and **Pod D** because they have all the labels it's looking for. It doesn't matter that Pod D has additional labels. However, it won't send traffic to Pod C because it doesn't have both labels. The following YAML defines a Deployment and a Service. The Deployment will create Pods with the `project=tkb` and `zone=prod` labels, and the Service will send traffic to them.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tkb-2024
spec:
  replicas: 10
```

```
  <Snip>
  template:
    metadata:
      labels:
        project: tkb        ----┐  Create Pods
        zone: prod          ----┘  with these labels
    spec:
      containers:
  <Snip>
---
apiVersion: v1
kind: Service
metadata:
  name: tkb
spec:
  ports:
  - port: 8080
  selector:
    project: tkb            ----┐  Send to Pods
    zone: prod              ----┘  with these labels
```

Behind the scenes with EndpointSlices

Whenever you create a Service, Kubernetes automatically creates
an associated EndpointSlice to track healthy Pods with matching
labels.

It works like this.

Every time you create a Service, the *EndpointSlice controller*
automatically creates an associated *EndpointSlice object*.
Kubernetes then watches the cluster, looking for Pods matching the
Service's label selector. Any new Pods matching the selector get
added to the EndpointSlice, whereas any deleted Pods get removed.
Applications send traffic to the Service name, and the application's
container uses the cluster DNS to resolve the name to the Service's
IP address. The container then sends the traffic to the Service's IP,
and the Service forwards it to one of the Pods listed in the
EndpointSlice.

Older versions of Kubernetes used an *Endpoints* object instead of
EndpointSlices. They're functionally identical, but EndpointSlices

perform better on large busy clusters.

Service types

Kubernetes has several types of Services for different use cases and requirements. The main ones are:

- ClusterIP
- NodePort
- LoadBalancer

ClusterIP is the most basic and provides a reliable endpoint (name, IP, and port) on the internal Pod network. *NodePort* Services build on top of ClusterIP and allow external clients to connect via a port on every cluster node. *LoadBalancers* build on top of both and integrate with cloud load balancers for extremely simple access from the internet.

All three are important, so let's look at each in turn.

ClusterIP Services - Accessing apps from inside the cluster

ClusterIP is the default Service type. These get a DNS name and IP address that are programmed into the internal network fabric and are **only accessible from inside the cluster.** This means:

- The IP is only routable on the internal Pod network
- The name is automatically registered with the cluster's internal DNS
- All containers are pre-programmed to use the cluster's DNS to resolve names

Let's consider an example.

You're deploying an application called **skippy**, and you want other applications on the cluster to access it via that name. To satisfy

these requirements, you create a new ClusterIP Service called **skippy**. Kubernetes creates the Service, assigns it an internal IP, and creates the DNS records in the cluster's internal DNS. Kubernetes also configures all containers on the cluster to use the cluster DNS for name resolution. This means every app on the cluster can connect to the new app using the **skippy** name.

However, ClusterIP Services aren't routable and require access to the cluster's internal DNS service, meaning they don't work outside the cluster.

We'll dive into this in the service discovery chapter.

NodePort Services - Accessing apps from outside the cluster

NodePort Services build on top of ClusterIP Services by adding a dedicated port on every cluster node that external clients can use. We call this dedicated port the *"NodePort"*.

The following YAML shows a NodePort Service called **skippy**.

```
apiVersion: v1
kind: Service
metadata:
  name: skippy              <<---- Registered with the internal cluster
DNS (ClusterIP)
spec:
  type: NodePort            <<---- Service type
  ports:
  - port: 8080              <<---- Internal ClusterIP port
    targetPort: 9000        <<---- Application port in container
    nodePort: 30050         <<---- External port on every cluster node
(NodePort)
  selector:
    app: hello-world
```

Posting this to Kubernetes will create a ClusterIP Service with the usual internally routable IP and DNS name. It will also publish port `30050` on every cluster node and map it back to the ClusterIP. This

means external clients can send traffic to any cluster node on port `30050` and reach the Service and its Pods.

Figure 7.3 shows a NodePort Service exposing three Pods on every cluster node on port `30050`. Step 1 shows an external client hitting a node on the NodePort. Step 2 shows the node forwarding the request to the ClusterIP of the Service inside the cluster. The Service picks a Pod from the EndpointSlice's always-up-to-date list in step 3 and forwards it to the chosen Pod in step 4.
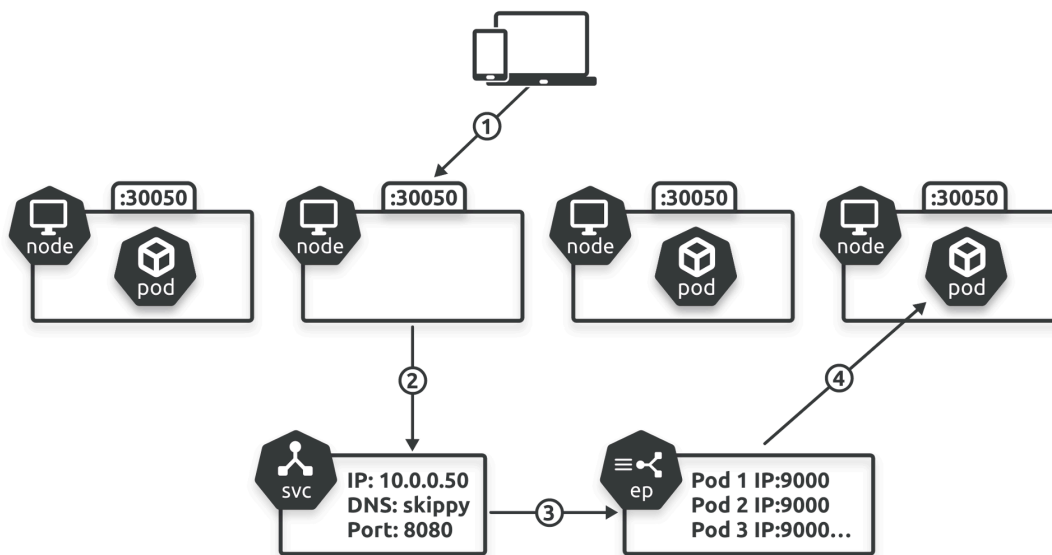
Figure 7.3 - NodePort Service

The external client can send the request to any cluster node, and the Service can send the request to any of the three healthy Pods. In fact, future requests will probably go to other Pods as the Service performs simple round-robin load balancing.

However, NodePort Services have two significant limitations:

- They use high-numbered ports between `30000-32767`
- Clients need to know the names or IPs of nodes, as well as whether nodes are healthy

This is why most people use *LoadBalancer Services* instead.

LoadBalancer Services - Accessing apps via load balancers

LoadBalancer Services are the easiest way to expose services to external clients. They simplify NodePort Services by putting a cloud load balancer in front of them.

Figure 7.4 shows a LoadBalancer Service. As you can see, it's basically a NodePort Service fronted by a highly-available load balancer with a publicly resolvable DNS name and low port number.
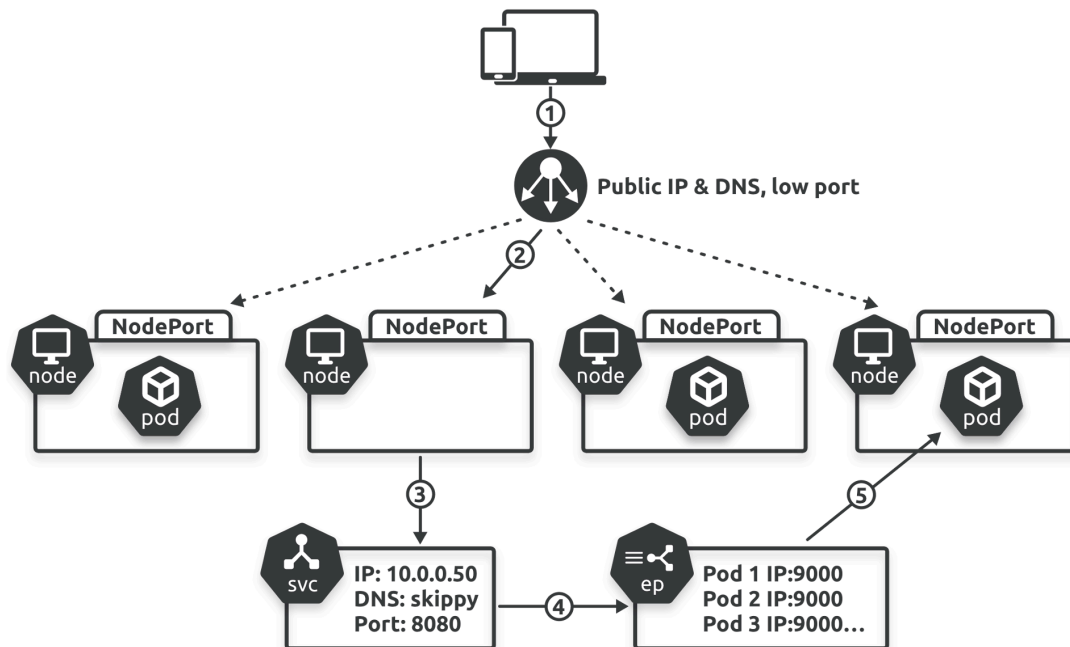
Figure 7.4 - LoadBalancer Service

The client connects to the load balancer via a reliable, friendly DNS name on a low-numbered port, and the load balancer forwards the request to a NodePort on a healthy cluster node. From there, it's the same as a NodePort Service — send to the internal ClusterIP Service, select a Pod from the EndpointSlice, and send the request to the Pod.

The following YAML creates a LoadBalancer Service listening on port `8080` and maps it all the way through to port `9000` on Pods with the `project=tkb` label. It automatically creates the required NodePort and ClusterIP constructs in the background.

```
apiVersion: v1
kind: Service
metadata:
  name: lb                   <<---- Registered with cluster DNS
spec:
  type: LoadBalancer
  ports:
  - port: 8080               <<---- Load balancer port
    targetPort: 9000         <<---- Application port inside container
  selector:
    project: tkb
```

You'll create and use a LoadBalancer Service in the hands-on section later.

Summary of Service theory

Services sit in front of Pods and make them accessible via a reliable network endpoint.

The front end of a Service provides an IP address, DNS name, and a port that is guaranteed to be stable for the entire life of the Service. The back-end load balances traffic over a dynamic set of Pods that match a label selector.

ClusterIP Services are the default and provide reliable endpoints on the internal cluster network. NodePorts and LoadBalancers provide external endpoints.

LoadBalancer Services create a load balancer on the underlying cloud platform, as well as all the constructs and mappings to forward traffic from the load balancer to the Pods.

Hands-on with Services

This section shows you how to work with Services imperatively and declaratively. As always, Kubernetes prefers the declarative method of deploying and managing everything with YAML files. However, it can be helpful to know the imperative commands.

You'll need all of the following if you're following along:

- Clone of the book's GitHub repo
- Kubernetes cluster

You'll be creating and working with LoadBalancer Services, and you can use any of the clusters we showed you how to create in Chapter 3. If your cluster is in the cloud, Kubernetes will provision one of your cloud's internet-facing load balancers and provide you with public IPs or public DNS names. If you're using a local cluster, such as Docker Desktop, the experience will be the same, but you'll use local constructs such as `localhost`.

If you don't already have a copy of the book's GitHub repo, clone it with the following command and then switch to the `2025` branch.

```
$ git clone https://github.com/nigelpoulton/TKB.git
Cloning into 'TKB'...

$ cd TKB

$ git fetch origin

$ git checkout -b 2025 origin/2025
```

Switch to the `services` directory.

```
$ cd TKB/services
```

Run the following command to deploy a sample app called **svc-test**. It's a Deployment that creates ten Pods running a web app listening on port `8080` and with the `chapter=services` label.

```
$ kubectl apply -f deploy.yml
deployment.apps/svc-test created
```

Ensure the Pods were successfully deployed and then continue to the next section.

```
$ kubectl get deploy svc-test
NAME          READY     UP-TO-DATE     AVAILABLE     AGE
svc-test      10/10     10             10            24s
```

Working with Services imperatively

The `kubectl expose` command creates a Service for an existing Deployment. It's intelligent enough to inspect the running Deployment and create all the required constructs, such as IP address, label selector, DNS records, and correct port mappings.

Run the following command to create a new LoadBalancer Service for the Pods in the **svc-test** Deployment.

```
$ kubectl expose deployment svc-test --type=LoadBalancer
service/svc-test exposed
```

List services to see its basic config. It may take a minute for the `EXTERNAL-IP` column to populate if you're running on a cloud.

```
$ kubectl get svc -o wide
NAME          TYPE            CLUSTER-IP      EXTERNAL-IP     PORT(S)
SELECTOR
kubernetes    ClusterIP       10.96.0.1       <none>          443/TCP
<none>
svc-test      LoadBalancer    10.10.19.33     212.2.245.220   8080:31755/TCP
chapter=services
```

The first line is a system Service called **Kubernetes** that exposes the Kubernetes API to all Pods and containers on the cluster.

Your Service is on the second line, and there's a lot of info, so let's step through it.

First up, it's been allocated the same name as the Deployment it's sitting in front of — **svc-test**.

The `TYPE` column shows this one's a LoadBalancer Service, and the one in the example is assigned an `EXTERNAL-IP` of `212.2.245.220`. If you're on a local cluster such as Docker Desktop, the `EXTERNAL-IP` will show `localhost`. Some Docker Desktop clusters incorrectly return a `172` IP address in the `EXTERNAL-IP` column, it should be `localhost`.

The `CLUSTER-IP` column lists the Service's internal IP that's only routable on the internal cluster network.

The `PORT(S)` column shows the load balancer port (`8080`) and the NodePort (`31755`). By default, the load balancer port matches the port the app listens on, but you can override this. The NodePort value is randomly assigned from between `30000-32767`.

The `SELECTOR` column matches the Pod labels.

A couple of things are worth noting.

First up, the command inspected the running Deployment and created the correct port mappings and label selector — the app is listening on port `8080`, and all 10 Pods have the `chapter=services` label.

Second up, even though it's a LoadBalancer Service, it also created all the ClusterIP and NodePort constructs. This is because LoadBalancer Services build on top of NodePort Services, which, in turn, build on top of ClusterIP Services, as shown in Figure 7.5.
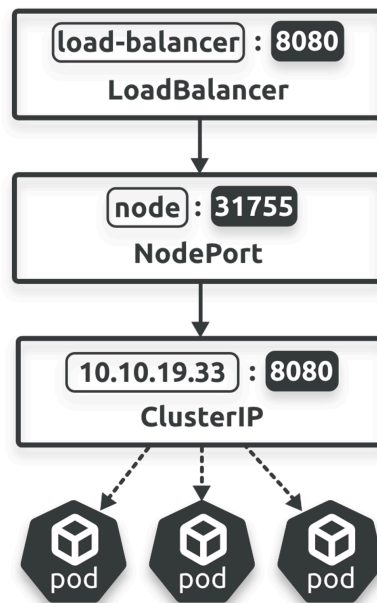
Figure 7.5 - Service stacking

The `kubectl describe` command gives you even more detail.

```
$ kubectl describe svc svc-test
Name:                   svc-test
Namespace:              default
Labels:                 <none>
Annotations:            <none>
Selector:               chapter=services
Type:                   LoadBalancer
IP Family Policy:       SingleStack
IP Families:            IPv4
IP:                     10.10.19.33
IPs:                    10.10.19.33
LoadBalancer Ingress:   212.2.245.220
Port:                   <unset>  8080/TCP       <<---- Load balancer
port
TargetPort:             8080/TCP               <<---- Application
port in container
NodePort:               <unset>  31755/TCP     <<---- NodePort on
each cluster node
Endpoints:
10.1.0.200:8080,10.1.0.201:8080,10.1.0.202:8080 + 7 more...
Session Affinity:       None
External Traffic Policy:  Cluster
Events:                 <none>
```

The output repeats much of what you've already seen, and I've added comments to a few lines to clarify the different port-related values.

There are also a few additional lines of interest.

`Endpoints` is the list of healthy matching Pods from the Service's EndpointSlice object.

`Session Affinity` allows you to control session *stickiness* — whether or not client connections always go to the same Pod. The default is *None* and forwards multiple connections from the same client to different Pods. You should try the *ClientIP* option if your app stores state in Pods and requires session stickiness. However, this is an *anti-pattern* as microservices apps should be designed for process disposability where clients can connect to any Pod.

`External Traffic Policy` dictates whether traffic hitting the Service will be load balanced across Pods on all cluster nodes or just Pods on the node the traffic arrives on. The default is `Cluster`, and it sends traffic to Pods on all cluster nodes but obscures source IP addresses. The other option is `Local`, which only sends traffic to Pods on the node the traffic arrives on but preserves source IPs.

If your cluster runs dual-stack networking, your output may also list IPv6 addresses.

Test if the Service works by pointing your browser to the value in the `EXTERNAL-IP` column on port `8080`.
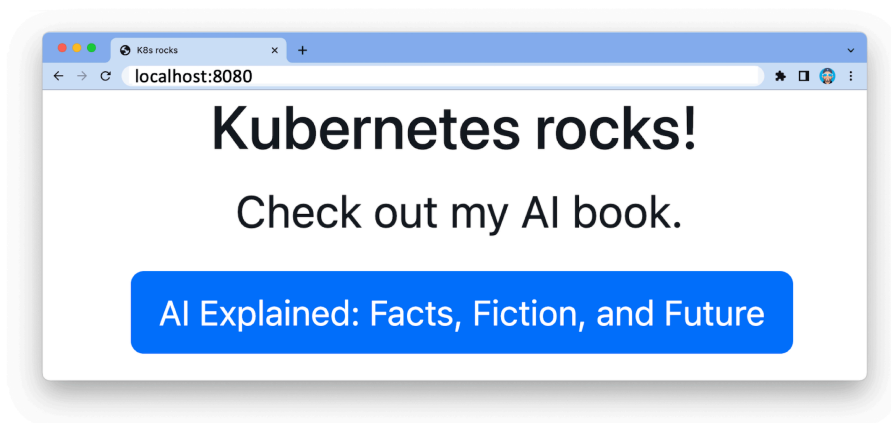
Figure 7.6

It works. Your app is running inside a container and listening on port `8080`. You created a LoadBalancer Service that listens on port `8080` and forwards traffic to a NodePort Service on each cluster node, which, in turn, forwards it to a ClusterIP Service on port `8080`. From there, it's sent to a Pod hosting an app replica on port `8080`.

Coming up next, you'll do it all again but declaratively. But you'll need to clean up first.

```
$ kubectl delete svc svc-test
service "svc-test" deleted
```

The declarative way

It's time to do things the proper way — the Kubernetes way.

A Service manifest file

The following YAML is from the `lb.yml` file, and you'll use it to deploy a LoadBalancer Service declaratively.

```
kind: Service
apiVersion: v1
metadata:
  name: svc-lb
```

```
spec:
  type: LoadBalancer
  ports:
  - port: 9000           <<---- Load balancer port
    targetPort: 8080     <<---- Application port inside container
  selector:
    chapter: services
```

Let's step through it.

The first two lines tell Kubernetes to deploy a Service object based on the `v1` schema.

The `metadata` block tells Kubernetes to name this Service **svc-lb** and register the name with the internal cluster DNS. You can also define custom labels and annotations here.

The `spec` section defines all the front-end and back-end details. This example tells Kubernetes to deploy a `LoadBalancer` Service that listens on port `9000` on the front end and sends traffic to Pods with the `chapter=services` label on port `8080`.

Deploy it with the following command.

```
$ kubectl apply -f lb.yml
service/svc-lb created
```

Inspecting Services

Services are regular API resources, meaning you can inspect them with the usual `kubectl get` and `kubectl describe` commands.

```
$ kubectl get svc svc-lb
NAME        TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)
svc-lb      LoadBalancer  10.43.191.202   212.2.247.202
9000:30202/TCP
```

If you cluster is in the cloud, your output will show `<pending>` in the `EXTERNAL-IP` column while your cloud platform provisions a load

balancer and allocates it an IP address. Keep refreshing the command until an address appears.

The Service in the example is exposed to the internet via a cloud load balancer on `212.2.247.202`. If you're running a local Docker Desktop cluster, you'll access it via your laptop's `localhost` interface. If your Docker Desktop cluster shows a `172` IP address in the **EXTERNAL-IP** column, ignore it and use `localhost` on port `9000`.

Once Kubernetes has created your Service, point your browser to the value in the **EXTERNAL-IP** column on port `9000` to make sure you can see the app. Remember to use `localhost` for a Docker Desktop cluster.

Let's look at your Service's EndpointSlices before cleaning up.

EndpointSlice objects

Earlier in the chapter, you learned that every Service gets one or more of its own EndpointSlice objects. These are where Kubernetes keeps its up-to-date list of healthy Pods matching the label selector, and you can inspect them with the usual **kubectl** commands.

The examples are from a cluster running dual-stack networking. Notice how two EndpointSlices exist — one for the IPv4 mappings and the other for IPv6. Your cluster may only have IPv4 mappings.

```
$ kubectl get endpointslices
NAME             ADDRESSTYPE   PORTS    ENDPOINTS
AGE
svc-lb-n7jg4     IPv4          8080     10.42.1.16,10.42.1.17,10.42.0.19
+ 7 more...    2m1s
svc-lb-9s6sq     IPv6          8080     fd00:10:244:1::c,fd00:10:244:1::9
+ 7 more...    2m1s

$ kubectl describe endpointslice svc-lb-n7jg4
Name:        svc-lb-n7jg4
Namespace:   default
Labels:      chapter=services
             endpointslice.kubernetes.io/managed-by=endpointslice-
```

```
controller.k8s.io
               kubernetes.io/service-name=svc-lb
Annotations:  endpoints.kubernetes.io/last-change-trigger-time: 2024-
01-01T18:13:40Z
AddressType:  IPv4
Ports:
  Name      Port  Protocol
  ----      ----  --------
  <unset>  8080  TCP
Endpoints:
  - Addresses:  10.42.1.16
    Conditions:
      Ready:      true
    Hostname:    <unset>
    TargetRef:   Pod/svc-lb-9d7b4cf9d-hnvbf
    NodeName:    k3d-tkb-agent-2
    Zone:        <unset>
  - Addresses:  10.42.1.17
<Snip>
Events:          <none>
```

The full command output has a block for each healthy Pod containing useful info. If a Service maps to more than 100 Pods, it will have more than one EndpointSlice.

Clean up

Run the following command to delete the Deployment and Service created in the examples. Kubernetes will automatically delete Endpoints and EndpointSlices when you delete their associated Service.

```
$ kubectl delete -f deploy.yml -f lb.yml
deployment.apps "svc-lb" deleted
service "svc-lb" deleted
```

Chapter Summary

In this chapter, you learned that Services provide reliable networking for Pods. They have a front end with a DNS name, IP address, and port that Kubernetes guarantees will never change.

They also have a back-end that sends traffic to healthy Pods matching a label selector.

ClusterIP Services provide reliable networking on the internal Kubernetes network, NodePort Services expose a port on every cluster node, and LoadBalancer Services integrate with cloud platforms to create highly available internet-facing load balancers.

Finally, Services are first-class objects in the Kubernetes API and should be managed declaratively through version-controlled YAML files.

8: Ingress

Ingress is all about accessing multiple web applications through a single LoadBalancer Service.

You'll need a working knowledge of Kubernetes Services before reading this chapter. If you don't already have this, consider going back and reading the previous chapter first.

I've divided this chapter into the following three sections:

- Setting the scene for Ingress
- Ingress architecture
- Hands-on with Ingress

We'll capitalize *Ingress* as it's a resource in the Kubernetes API. We'll also use the terms *LoadBalancer* and *load balancer* as follows:

- *LoadBalancer* refers to a Kubernetes Service object of `type=LoadBalancer`
- *load balancer* refers to one of your cloud's internet-facing load balancers

As an example, when you create a Kubernetes *LoadBalancer* Service, Kubernetes talks to your cloud platform and provisions a cloud *load balancer.*

Ingress was promoted to *generally available (GA)* in Kubernetes version 1.19 after being in beta for over 15 releases. During the 3+ years it was in alpha and beta, service meshes increased in popularity, and there's now some overlap in functionality. As a result, if you're planning to deploy a service mesh, you may not need Ingress.

Setting the Scene for Ingress

The previous chapter showed you how to use NodePort and LoadBalancer Services to expose applications to external clients. However, both have limitations.

NodePort Services only work on high port numbers, and clients need to keep track of node IP addresses. LoadBalancer Services fix this but only provide a one-to-one mapping between internal Services and cloud load balancers. This means a cluster with 25 internet-facing apps will need 25 cloud load balancers, and cloud load balancers cost money! Your cloud may also limit the number of load balancers you can create.

Ingress fixes this by letting you expose multiple Services through a single cloud load balancer.

It does this by creating a single cloud load balancer on port `80` or `443` and using *host-based* and *path-based* routing to map connections to different Services on the cluster. We'll explain this jargon soon.

Ingress architecture

Ingress is defined in the `networking.k8s.io/v1` API sub-group, and it requires the usual two constructs:

1. A resource
2. A controller

The resource *defines* the routing rules, and the controller *implements* them.

However, Kubernetes doesn't have a built-in Ingress controller, meaning you need to install one. This differs from Deployments, ReplicaSets, Services, and most other resources that have built-in

pre-configured controllers. However, some cloud platforms simplify this by allowing you to install one when you build the cluster. We'll show you how to install the popular NGINX Ingress controller in the hands-on section.

Once you have an *Ingress controller*, you deploy *Ingress resources* with rules telling the controller how to route requests.

On the topic of *routing,* Ingress operates at *layer 7* of the OSI model, also known as the *application layer*. This means it can inspect HTTP headers and forward traffic based on hostnames and paths.

> **Note:** The *OSI model* is the industry-standard reference model for TCP/IP networking and has seven layers numbered 1-7. The lowest layers are concerned with signaling and electronics, the middle layers deal with reliability through acknowledgements and retries, and the higher layers add services for things like HTTP. Ingress operates at layer 7, also known as the *application layer*, and implements HTTP intelligence.

The following table shows how hostnames and paths can route to backend ClusterIP Services.

Host-based example	Path-based example	Backend K8s Service
shield.mcu.com	mcu.com/shield	shield
hydra.mcu.com	mcu.com/hydra	hydra

Figure 8.1 shows two requests hitting the same cloud load balancer. Behind the scenes, DNS name resolution maps both hostnames to the same load balancer IP. An Ingress controller watches the load balancer and routes the requests based on the hostnames in the HTTP headers. In this example, it routes `shield.mcu.com` to the **shield** ClusterIP Service, and `hydra.mcu.com` to the **hydra** Service.

The logic is the same for path-based routing, and we'll see both in the hands-on section.
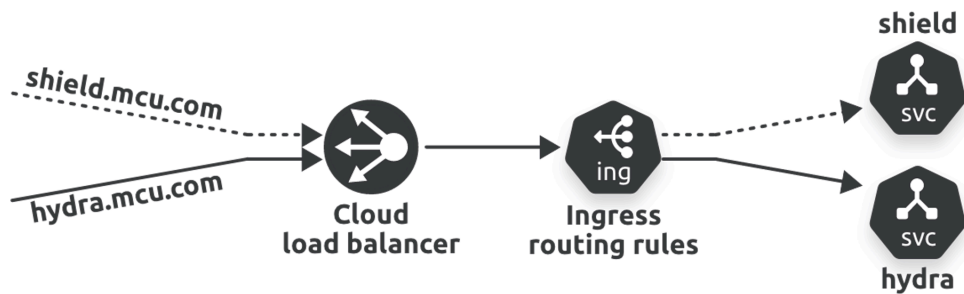
Figure 8.1 Host-based routing

In summary, a single Ingress can expose multiple Kubernetes Services through a single cloud load balancer. You create and deploy Ingress resources that tell your Ingress controller how to route requests based on hostnames and paths in request headers. You might have to install an Ingress controller manually.

Let's see it in action.

Hands-on with Ingress

This section doesn't work with the multi-node Kubernetes cluster that ships with Docker Desktop v4.38 or earlier. It *may* work with future releases. I recommend you work with a cloud-based cluster such as the LKE cluster we show you how to build in Chapter 3.

You'll need both of these if you're following along:

- A Kubernetes cluster
- A clone of the book's GitHub repo

If you don't already have it, clone the book's GitHub repo and switch to the 2025 branch.

```
$ git clone https://github.com/nigelpoulton/TKB.git
Cloning from...

$ cd TKB

$ git fetch origin

$ git checkout -b 2025 origin/2025
```

Change into the `ingress` directory and run all commands from there.

You'll complete all of the following steps:

1. Install the NGINX Ingress controller
2. Configure an Ingress class
3. Deploy a sample app
4. Configure an Ingress object
5. Inspect the Ingress object
6. Configure DNS name resolution
7. Test the Ingress

Install the NGINX Ingress controller

You'll install the NGINX controller from a YAML file hosted in the Kubernetes GitHub repo. It installs a bunch of Kubernetes constructs, including a Namespace, ServiceAccounts, ConfigMaps, Roles, RoleBindings, and more.

Install it with the following command. I've split the command over two lines because the URL is so long. You'll have to run it on a single line.

```
$ kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/
controller-v1.12.0/deploy/static/provider/cloud/deploy.yaml

namespace/ingress-nginx created
serviceaccount/ingress-nginx created
<Snip>
```

Run the following command to check the `ingress-nginx` Namespace and ensure the *controller* Pod is running. It may take a few seconds for it to enter the running phase, and Windows users will need to replace the backslash (\ ) at the end of the first line with a backtick (`).

```
$ kubectl get pods -n ingress-nginx \
  -l app.kubernetes.io/name=ingress-nginx

NAME                                       READY   STATUS
RESTARTS   AGE
ingress-nginx-admission-create-789md       0/1     Completed   0
25s
ingress-nginx-admission-patch-tc4cl        0/1     Completed   0
25s
ingress-nginx-controller-7445ddc6c4-csf98  0/1     Running     0
26s
```

Don't worry about the *Completed* Pods. These were short-lived Pods that initialized the environment.

Once the *controller* Pod is running, you have an NGINX Ingress controller and are ready to create some Ingress objects. However, before doing that, let's look at *Ingress classes*.

Ingress classes

Ingress classes allow you to run multiple Ingress controllers on a single cluster. The process is simple:

1. You map each Ingress controller to its own Ingress class
2. When you create Ingress objects, you assign them to an Ingress class

If you're following along, you'll have at least one Ingress class called **nginx**. This was created when you installed the NGINX controller.

```
$ kubectl get ingressclass
NAME     CONTROLLER                PARAMETERS     AGE
```

```
nginx    k8s.io/ingress-nginx    <none>        2m25s
```

You'll have multiple classes if your cluster already had an Ingress controller.

Take a closer look at the **nginx** Ingress class with the following command. There is no shortname for Ingress class objects.

```
$ kubectl describe ingressclass nginx
Name:          nginx
Labels:        app.kubernetes.io/component=controller
               app.kubernetes.io/instance=ingress-nginx
               app.kubernetes.io/name=ingress-nginx
               app.kubernetes.io/part-of=ingress-nginx
               app.kubernetes.io/version=1.9.4
Annotations:   <none>
Controller:    k8s.io/ingress-nginx
Events:        <none>
```

With an Ingress controller and Ingress class in place, you're ready to deploy and configure an Ingress object.

Configure host-based and path-based routing

This section deploys two apps and a single Ingress object. The Ingress will route traffic to both apps via a single load balancer. This can be a cloud-based load balancer or `localhost` on some local clusters.

You'll complete all the following steps:

1. Deploy an app called **shield** and front it with a ClusterIP Service (backend) called **svc-shield**
2. Deploy an app called **hydra** and front it with a ClusterIP Service (backend) called **svc-hydra**
3. Deploy an Ingress object that creates a single load balancer and routing rules for the following hostnames and paths
   - Host-based: `shield.mcu.com` >> **svc-shield**
   - Host-based: `hydra.mcu.com` >> **svc-hydra**

- Path-based: `mcu.com/shield` >> **svc-shield**
- Path-based: `mcu.com/hydra` >> **svc-hydra**

4. Configure DNS name resolution to that `shield.mcu.com`, `hydra.mcu.com`, and `mcu.com` point to your load balancer

Figure 8.2 shows the overall architecture using host-based and path-based routing.
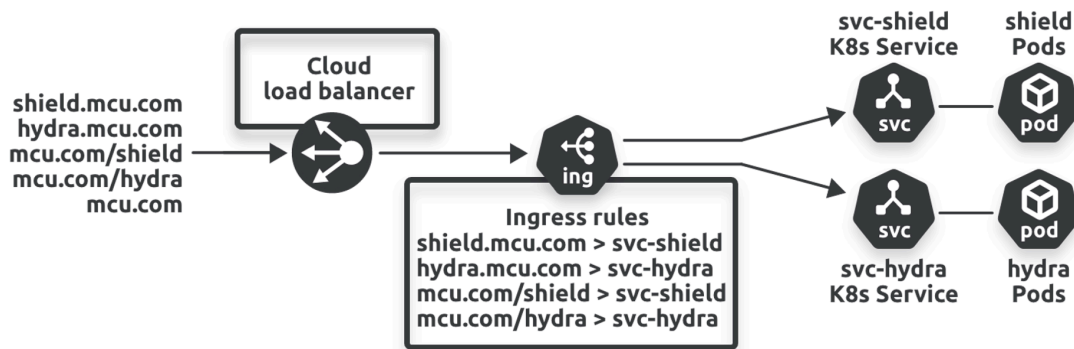
Figure 8.2 Host-based routing

Traffic flow to the **shield** Pods will be as follows:

1. Client sends traffic to `shield.mcu.com` or `mcu.com/shield`
2. DNS name resolution ensures the traffic goes to the cloud load balancer
3. Ingress controller reads the HTTP headers and finds the hostname (`shield.mcu.com`) or path (`mcu.com/shield`)
4. Ingress rule triggers and routes the traffic to the **svc-shield** ClusterIP backend Service
5. The ClusterIP Service ensures the traffic reaches a shield Pod

Deploy the sample environment

This section deploys the two apps and ClusterIP Services that the Ingress will route traffic to.

The lab is defined in the `app.yml` file in the `ingress` folder and comprises the following.

- An app called **shield**, listening on port `8080`, and fronted by a ClusterIP Service called **svc-shield**
- Another app called **hydra**, also listening on port `8080`, and fronted by a ClusterIP Service called **svc-hydra**

Deploy it with the following command.

```
$ kubectl apply -f app.yml
service/svc-shield created
service/svc-hydra created
pod/shield created
pod/hydra created
```

Once the Pods and Services are up and running, proceed to the next section to create the Ingress.

Create the Ingress object

You'll deploy the ingress object defined in the `ig-all.yml` file. It describes an Ingress object called **mcu-all** with four rules.

```
1   apiVersion: networking.k8s.io/v1
2   kind: Ingress
3   metadata:
4     name: mcu-all
5     annotations:
6       nginx.ingress.kubernetes.io/rewrite-target: /
7   spec:
8     ingressClassName: nginx
9     rules:
10   - host: shield.mcu.com      ----┐
11     http:                         |
12       paths:                      |
13       - path: /                   |
14         pathType: Prefix          | Host rule block for shield app
15         backend:                  |
16           service:                |
17             name: svc-shield      |
18             port:                 |
19               number: 8080    ----┘
20   - host: hydra.mcu.com       ----┐
```

```
21      http:                      |
22        paths:                   |
23        - path: /                |
24          pathType: Prefix       | Host rule block for hydra app
25          backend:               |
26            service:             |
27              name: svc-hydra    |
28              port:              |
29                number: 8080  ----┘
30    - host: mcu.com
31      http:
32        paths:
33        - path: /shield    ----┐
34          pathType: Prefix     |
35          backend:             |
36            service:           |  Path rule block for shield app
37              name: svc-shield |
38              port:            |
39                number: 8080  ----┘
40        - path: /hydra     ----┐
41          pathType: Prefix     |
42          backend:             |
43            service:           | Path rule block for shield app
44              name: svc-hydra  |
45              port:            |
46                number: 8080  ----┘
```

Let's step through it.

The first two lines tell Kubernetes to deploy an Ingress object based on the schema in the `networking.k8s.io/v1` API.

Line four calls the Ingress `mcu-all.`

The annotation on line six tells the controller to make a best-effort attempt to rewrite paths to the path your app expects. This example rewrites incoming paths to "/". For example, traffic hitting the load balancer on the `mcu.com/shield` path will have the path rewritten to `mcu.com/`. You'll see an example shortly. This annotation is specific to the NGINX Ingress controller, and you'll have to comment it out if you're using a different controller.

The `spec.ingressClassName` field on line eight tells Kubernetes this Ingress object needs to be managed by the NGINX Ingress

controller you installed earlier. You'll have to change this line, or comment it out, if you're using a different Ingress controller.

The file contains four rules:

- Lines 10-19 define a host-based rule for traffic arriving on `shield.mcu.com`
- Lines 20-29 define a host-based rule for traffic arriving on `hydra.mcu.com`
- Lines 30-39 define a path-based rule for traffic arriving on `mcu.com/shield`
- Lines 40-49 define a path-based rule for traffic arriving on `mcu.com/hydra`

Let's look at one of the host-based rules and then a path-based rule.

The following *host-based* rule triggers on traffic arriving via `shield.mcu.com` at the root "/" path and forwards it to the ClusterIP back-end Service called **svc-shield** on port `8080`.

```
- host: shield.mcu.com          <<---- Traffic arriving via this
hostname
  http:
    paths:
    - path: /                   <<---- Arriving at root (no subpath
specified)
      pathType: Prefix
      backend:                  <<---- The next five lines send
traffic to an
        service:                <<---- existing "backend" ClusterIP
Service
          name: svc-shield      <<---- called "svc-shield"
          port:                 <<---- that's listening on
            number: 8080        <<---- port 8080
```

The following *path-based* rule triggers when traffic arrives on `mcu.com/shield`. It gets routed to the same **svc-shield** back-end Service on the same port.

```
   - host: mcu.com                       <<---- Traffic arriving via this
hostname
    http:
     paths:
     - path: /shield                     <<---- Arriving on this subpath
       pathType: Prefix
       backend:                          <<---- The next five lines send
traffic to an
        service:                         <<---- existing "backend" ClusterIP
Service
           name: svc-shield   <<---- called "svc-shield"
           port:              <<---- that's listening on
             number: 8080     <<---- port 8080
```

Deploy the Ingress object with the following command.

```
$ kubectl apply -f ig-all.yml
ingress.networking.k8s.io/mcu-all created
```

Inspecting Ingress objects

List all Ingress objects in the default Namespace. If your cluster is on a cloud, it can take a minute or so to get an ADDRESS while the cloud provisions the load balancer.

```
$ kubectl get ing
NAME        CLASS    HOSTS                                ADDRESS
PORTS
mcu-all     nginx    shield.mcu.com,hydra.mcu.com,mcu.com   212.2.246.150
80
```

The CLASS field shows which Ingress class is handling this set of rules. It may show as <None> if you only have one Ingress controller and didn't configure classes. The HOSTS field is a list of hostnames the Ingress will handle traffic for. The ADDRESS field is the load balancer endpoint. If you're on a cloud, it will be a public IP or public DNS name. If you're on a local cluster, it'll probably be localhost. The PORTS field can be 80 or 443.

On the topic of ports, Ingress only supports HTTP and HTTPS.

Describe the Ingress. I've trimmed the output to fit the page.

```
$ kubectl describe ing mcu-all
Name:            mcu-all
Namespace:       default
Address:         212.2.246.150
Ingress Class:   nginx
Default backend: <default>
Rules:
  Host              Path      Backends
  ----              ----      --------
  shield.mcu.com    /         svc-shield:8080 (10.36.1.5:8080)
  hydra.mcu.com     /         svc-hydra:8080 (10.36.0.7:8080)
  mcu.com           /shield   svc-shield:8080 (10.36.1.5:8080)
                    /hydra    svc-hydra:8080 (10.36.0.7:8080)
Annotations:        nginx.ingress.kubernetes.io/rewrite-target: /
Events:             <none>
  Type      Reason    Age               From                      Message
  ----      ------    ----              ----                      -------
  Normal    Sync      27s (x2 over 28s) nginx-ingress-controller
Scheduled for sync
```

Let's step through the output.

The `Address` line is the IP or DNS name of the load balancer created by the Ingress. It might be `localhost` on local clusters.

`Default backend` is where the controller sends traffic arriving on a hostname or path it doesn't have a route for. Not all Ingress controllers implement a default backend.

The `Rules` section defines the mappings between *hosts*, *paths*, and *backends*. Remember that **backends** are usually ClusterIP Services that send traffic to Pods.

You can use annotations to define controller-specific features and integrations with your cloud back end. This example tells the controller to rewrite all paths so they look like they arrived on root "/". This is a *best-effort* approach, and as you'll see later, it doesn't work with all apps.

At this point, your load balancer is created. You can probably view it through your cloud console if you're on a cloud platform. Figure 8.3

shows how it looks on the Google Cloud back end if your cluster is on Google Kubernetes Engine (GKE).

Figure 8.3 Cloud back-end load balancer configuration

If you've been following along, you'll have all of the following:

- Two apps and associated ClusterIP Services
- Load balancer (cloud-based or `localhost`)
- Ingress (controller and resource) configured to route traffic

The only thing left to configure is DNS name resolution so that `shield.mcu.com`, `hydra.mcu.com` and `mcu.com` all send traffic to the load balancer.

Configure DNS name resolution

In the real world, you'll configure your internal DNS or internet DNS to point hostnames to the Ingress load balancer. How you do this varies depending on your environment and who provides your internet DNS.

If you're following along, the easiest thing to do is edit the `hosts` file on your local computer and map the hostnames to the Ingress load balancer.

On Mac and Linux, this file is `/etc/hosts`, and you'll need root permissions to edit it. On Windows, it's `C:\Windows\System32\drivers\etc\hosts`, and you'll need to open it as an administrator.

Windows users will need to open `notepad.exe` as an administrator and then open the `hosts` file in `C:\Windows\System32\drivers\etc`. Make sure the open dialog window is set to open **All files (.)**.

Create three new lines mapping `shield.mcu.com`, `hydra.mcu.com`, and `mcu.com` to the IP of the load balancer. Use the IP from the output of a `kubectl get ing mcu-all` command. If you're using a local cluster and yours says `localhost`, use the `127.0.0.1` IP address.

```
$ sudo vi /etc/hosts

# Host Database
<Snip>
212.2.246.150 shield.mcu.com
212.2.246.150 hydra.mcu.com
212.2.246.150 mcu.com
```

Remember to save your changes.

With this done, any traffic you send to `shield.mcu.com`, `hydra.mcu.com`, or `mcu.com` will be sent to the Ingress load balancer.

Test the Ingress

Open a web browser and try the following URLs:

- shield.mcu.com
- hydra.mcu.com
- mcu.com

Figure 8.4 shows the overall architecture and traffic flow. Traffic hits the load balancer that Kubernetes automatically created when you deployed the Ingress. The traffic arrives on port `80` and the Ingress sends it to an internal ClusterIP Service based on the hostname in the headers. Traffic for `shield.mcu.com` goes to the **svc-shield** Service, and traffic for `hydra.mcu.com` goes to the **svc-hydra** Service.
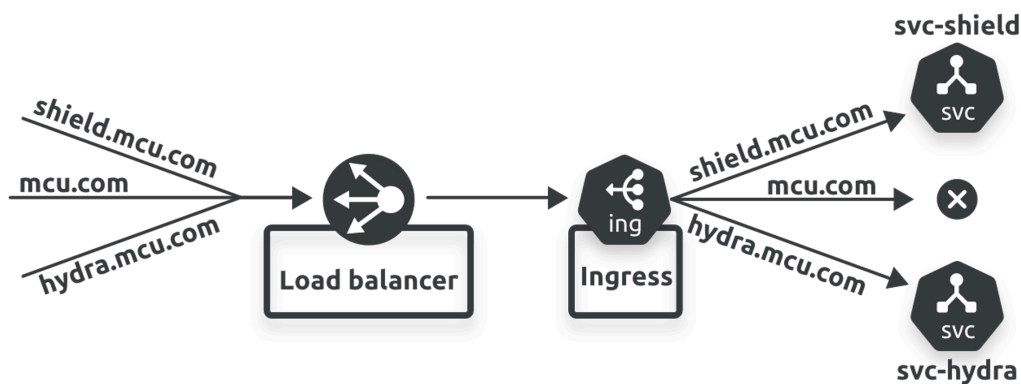
Figure 8.4 - host-based routing

Notice that requests to `mcu.com` are routed to the *default backend*. This is because you didn't create an Ingress rule for `mcu.com`. Depending on your Ingress controller, the message returned will be different, and your Ingress may not even implement a default backend. The default backend configured by the GKE built-in Ingress returns a helpful message saying, *response 404 (backend NotFound), service rules for [/] non-existent*.

Now try connecting to either of the following:

- mcu.com/shield
- mcu.com/hydra

For path-based routing like this, the Ingress uses the *rewrite targets* feature as specified in the object annotation. However, the image doesn't display because path rewrites like this don't work for all apps.

Congratulations, you've successfully configured Ingress for host-based and path-based routing — you've got two applications fronted by two ClusterIP Services, but both are published through a single load balancer created and managed by Kubernetes Ingress!

Clean up

If you're following along, you'll have all of the following on your cluster:

Pods	Services	Ingress controllers	Ingress resources
shield	svc-shield	ingress-nginx	mcu-all
hydra	svc-hydra		

Delete the Ingress resource.

```
$ kubectl delete -f ig-all.yml
ingress.networking.k8s.io "mcu-all" deleted
```

Delete the Pods and ClusterIP Services. It may take a few seconds for the Pods to terminate gracefully.

```
$ kubectl delete -f app.yml
service "svc-shield" deleted
service "svc-hydra" deleted
pod "shield" deleted
pod "hydra" deleted
```

Delete the NGINX Ingress controller. I've split the command over two lines so its fits the page better. You'll have to run it on one line,

and it can take about a minute for the command to complete and release your terminal.

```
$ kubectl delete -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/
controller-v1.12.0/deploy/static/provider/cloud/deploy.yaml

namespace "ingress-nginx" deleted
serviceaccount "ingress-nginx" deleted
<Snip>
```

Finally, **don't forget to revert your `/etc/hosts` file if you added manual entries earlier**.

```
$ sudo vi /etc/hosts

# Host Database
<Snip>
212.2.246.150 shield.mcu.com        <<---- Delete this entry
212.2.246.150 hydra.mcu.com         <<---- Delete this entry
212.2.246.150 mcu.com               <<---- Delete this entry
```

Chapter summary

In this chapter, you learned that Ingress is a way to expose multiple applications (ClusterIP Services) via a single cloud load balancer. They're stable objects in the API but have features that overlap with a lot of service meshes. If you're running a service mesh, you may not need Ingress.

Lots of Kubernetes clusters require you to install an Ingress controller, and lots of options exist. However, some hosted Kubernetes services make things easy by shipping with a built-in Ingress controller.

Once you've installed an Ingress controller, you create and deploy Ingress objects, which are lists of rules governing how incoming traffic is routed to applications on your cluster. It supports host-based and path-based HTTP routing.

9: Wasm on Kubernetes

Wasm (WebAssembly) is driving a new wave of cloud computing, and platforms like Kubernetes and Docker are evolving to take advantage.

Virtual Machines were the first wave, containers were the second, and Wasm is the third. Each subsequent wave enables smaller, faster, and more portable applications that can go places and do things the previous waves couldn't.
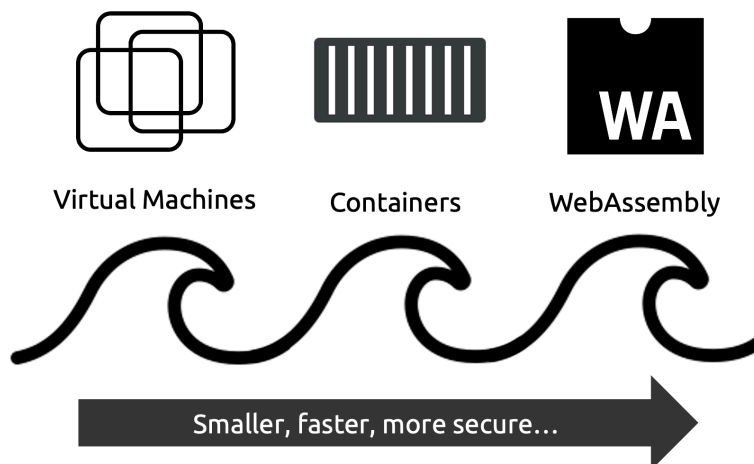
Figure 9.1

I've split the chapter as follows:

- Wasm Primer
- Understanding Wasm on Kubernetes
- Hands-on with Wasm on Kubernetes

The **Wasm Primer** section gets you up-to-speed on what Wasm is and its pros and cons. The **Understanding Wasm on Kubernetes** section overviews the requirements for running Wasm apps on Kubernetes. Finally, the **Hands-on with Wasm on**

Kubernetes section walks you through the end-to-end process of building and configuring a Kubernetes cluster capable of running Wasm apps, as well as writing, compiling, containerizing, and running a Wasm app on Kubernetes.

There are other simpler ways of creating Wasm apps and configuring Kubernetes to run them. For example, the [SpinKube](#) project from Fermyon automates many of the things you're about to learn, and you'll probably use something like SpinKube in the real world. But doing everything manually, as you will in this chapter, will help you gain a deeper understanding.

A quick word on terminology.

The terms *Wasm* and *WebAssembly* mean the same thing, and we'll use them interchangeably. In fact, Wasm is short for WebAssembly and isn't an acronym. This means the correct way to write it is **Wasm**, not ~~WASM~~. However, be kind to people and don't be critical of unimportant mistakes like this.

Also, Wasm on Kubernetes is only one of many use cases covered by terms such as *"WebAssembly outside the browser"*, *"WebAssembly on the server"*, *"WebAssembly in the cloud"*, and *"WebAssembly at the edge"*.

Wasm Primer

WebAssembly first appeared on the scene in 2017 and immediately made a name for itself by speeding up web apps. Fast-forward eight years, and it's an official W3C standard, it's in all the major browsers, and it's the go-to solution for web games and web apps that require high performance without sacrificing security and portability.

It should, therefore, come as no surprise that cloud entrepreneurs observed the rise of WebAssembly and realized it would be a great

technology for cloud apps.

In fact, Wasm is such a great fit for the cloud that Docker Founder Solomon Hykes famously tweeted, *"If Wasm+WASI existed in 2008, we wouldn't have needed to create Docker. That's how important it is. WebAssembly on the server is the future of computing. A standardized system interface was the missing link. Let's hope WASI is up to the task!"*.

He quickly followed up with another tweet saying he expected a future where Linux containers and Wasm containers work side-by-side, and Docker works with them all.

Well, Solomon's predicted future is already here. Docker has excellent support for Wasm, and you can run Linux containers side-by-side with Wasm containers in the same Kubernetes Pod. However, the Wasm standards and ecosystem are still relatively new, meaning traditional Linux containers remain the best solution for many cloud apps and use cases.

On the technical side, Wasm is a binary instruction set architecture (ISA) like ARM, x86, MIPS, and RISC-V. This means programming languages can compile source code into *Wasm binaries* that will run on any system with a *Wasm runtime*. Wasm apps execute inside a deny-by-default secure sandbox that distrusts the application, meaning access to everything is denied and must be explicitly allowed. This is the opposite of containers that start with everything wide open.

WASI is the WebAssembly System Interface and allows sandboxed Wasm apps to securely access external services such as key-value stores, networks, the host environment, and more. WASI is absolutely vital to the success of Wasm outside the browser, and at the time of writing, WASI Preview 2 is released and is a huge step forward.

You'll sometimes see WASI Preview 2 written as *WASI 0.2* and *wasip2.*

Let's quickly cover the security, portability, and performance aspects of Wasm.

Wasm security

Wasm starts with everything locked down. Containers start with everything wide open.

Despite containers defaulting to an open security policy, it's important to acknowledge the incredible work done by the community securing containers and container orchestration platforms. It's easier than ever to run secure containerized apps, especially on hosted Kubernetes platforms. However, the allow-by-default model with broad access to a shared kernel will always present security challenges for containers.

Wasm is very different. Wasm apps execute in a deny-by-default sandbox where the runtime has to explicitly allow access to resources outside the sandbox. You should also know that this sandbox has been battle-hardened through many years of use in one of the most hostile environments in the world… the web!

Wasm portability

It's a common misconception that containers are portable. They're not!

We only *think* containers are portable because they're smaller than VMs and easier to copy between hosts and registries. However, this isn't *true portability.* In fact, containers are *architecture-dependent*, meaning they're **not** portable. For example, you cannot run an ARM container on an AMD processor or a Windows container on a Linux system.

Yes, it's true that build tools have made it a lot easier to build container images for different platforms. However, each container still only works on one platform or architecture, and many organizations end up with *image sprawl*. As an oversimplified example, I maintain two images for most of the apps in this book — one for Linux on ARM and another for Linux on AMD64. Sometimes, I'll update an app and forget to build the Linux/amd64 image, causing examples to fail for readers running Kubernetes on Linux/amd64.

WebAssembly solves this and delivers on the ***build once, run everywhere*** promise!

It does this by implementing its own bytecode format that requires a runtime to execute. You build an app once as a Wasm binary, which you can then run on any system with a Wasm runtime.

As a quick example, I built the sample app for this chapter on an ARM-based Mac. However, I compiled it to Wasm, meaning it'll run on any host with an appropriate Wasm runtime. Later in the chapter, we'll execute it on a Kubernetes cluster that could be on your laptop, in a data center, or in the cloud. It can also run on any architecture supported by Kubernetes. Wasm runtimes even exist for exotic architectures that you find on IoT and edge devices.

Speaking of IoT devices, Wasm apps are typically a lot smaller than Linux containers, meaning you can run them in resource-constrained environments, such as the edge and IoT, where you can't run containers.

In summary, Wasm delivers on the promise of *build once, run anywhere.*

Wasm performance

As a general rule, VMs take minutes to start, and containers take seconds, but Wasm gets us into the exciting world of sub-second start times. So much so that Wasm *cold starts* are so fast they don't feel like cold starts. For example, Wasm apps commonly start in around ten milliseconds or less. And with the right optimizations, some can start in microseconds!

This is game-changing and is driving a lot of the early use cases. For example, Wasm is great for event-driven architectures like serverless functions. It also makes things like true scale-to-zero a possibility.

Quick recap

Wasm apps are smaller, faster, more portable, and more secure than traditional Linux containers. However, it's still the early days, and Wasm isn't the right choice for everything. Currently, Wasm is a great choice for event handlers and anything needing super-fast startup times. It can also be great for IoT, edge computing, and building extensions and plugins. However, at the time of writing, containers may still be the better choice for traditional cloud apps where networking, heavy I/O, and connecting to other services are requirements.

Despite all of this, Wasm is developing fast, and WASI Preview 2 is a significant step forward.

Now that we know a bit about Wasm, let's see how it can work with Kubernetes.

Understanding Wasm on Kubernetes

This section introduces the major requirements for running Wasm apps on Kubernetes clusters that use *containerd*. Other ways to run Wasm apps on Kubernetes exist.

This is also just an overview section. We'll cover everything in more detail in the hands-on section.

Kubernetes is a high-level *orchestrator* that uses other tools to perform low-level tasks such as creating, starting, and stopping containers. The most common configuration is Kubernetes using containerd to manage these lower-level tasks.

Figure 9.2 shows Kubernetes scheduling tasks to a worker node that's running containerd. In the example, containerd receives the work tasks and instructs *runc* to build the containers and start the apps. Once the containers are running, runc exits, leaving the *shim* processes to maintain the connections between the running containers and containerd.
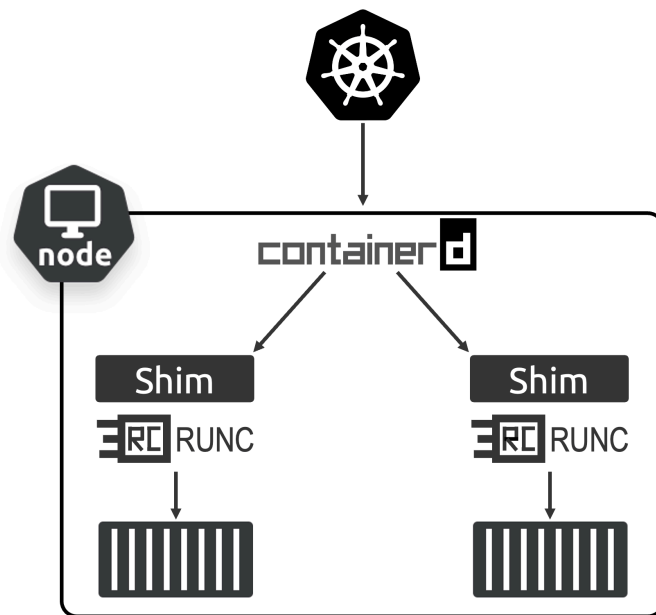
Figure 9.2

In this architecture, everything below containerd is hidden from Kubernetes. This means you can replace runc and the standard shim with a Wasm runtime and a Wasm shim. Figure 9.3 shows the same node running two additional Wasm workloads.
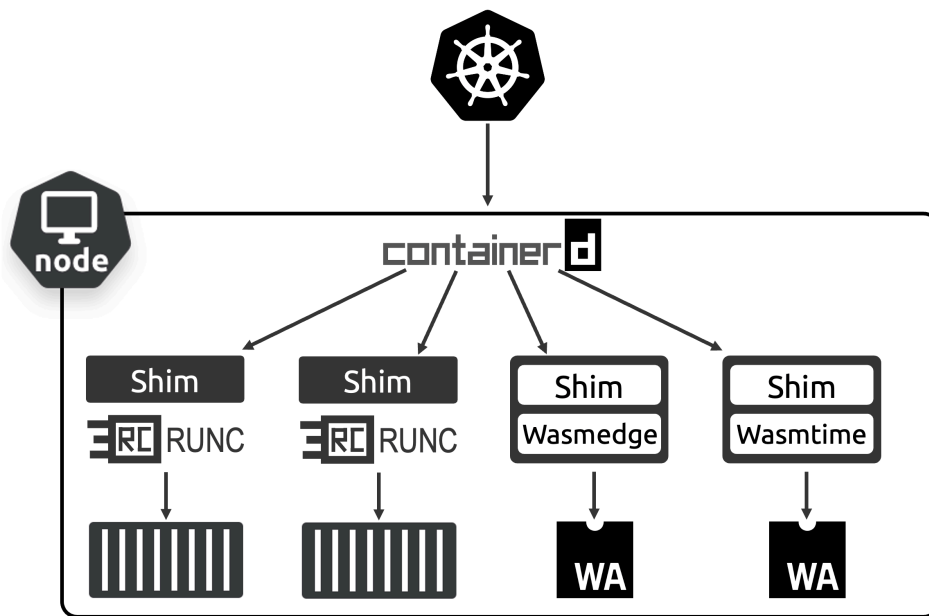
Figure 9.3

Remember, there's still a single containerd instance on the node, and Kubernetes doesn't see anything below containerd. This is a fully supported configuration, and we'll deploy something very similar in the hands-on section.

It's also worth noting that the Wasm shim architecture differs from the runc shim architecture. As shown in Figure 9.4, a Wasm shim is a single binary that includes the shim code **and** the Wasm runtime code.
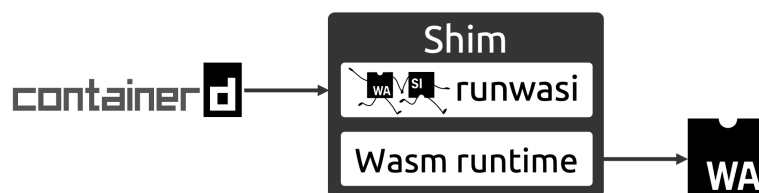
Figure 9.4

The Wasm shim code that interfaces with containerd is *runwasi*, but each shim can embed a specific Wasm runtime. For example, the

Spin shim embeds the runwasi Rust library and the Spin runtime code. Likewise, the Slight shim embeds runwasi and the Slight runtime. In each shim, the embedded Wasm runtime creates the *Wasm host* and executes the Wasm app, while runwasi does all the translating and interfacing with containerd.

One last thing on shims. containerd mandates that we name all shim binaries as follows:

- Use the `containerd-shim-` prefix
- Specify the name of the runtime
- Specify the version

As an example, the Spin shim is called `containerd-shim-spin-v2`.

Figure 9.5 shows a Kubernetes cluster with two nodes running different shims. One is running the WasmEdge shim and the other is running the Spin shim. In configurations like this, Kubernetes needs help scheduling workloads to nodes with the correct shims. We provide this help through *node labels* and *RuntimeClass* objects. Node 2 in the diagram has the `spin=yes` label, and Kubernetes has a RuntimeClass object that selects on this label and specifies the target runtime in the `handler` property. This ensures any Pod referencing this RuntimeClass will be scheduled to Node 2 and use the Spin runtime (*handler*).

Don't worry if this is confusing, everything will all fall into place when we do the hands-on exercises.
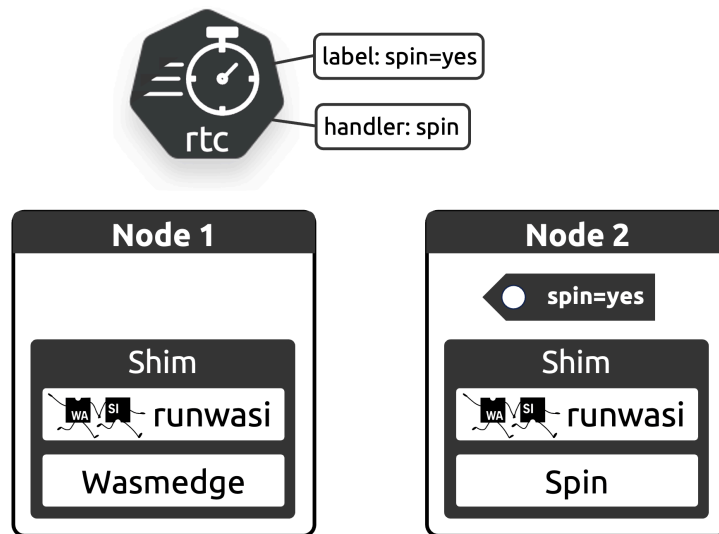
Figure 9.5

The workflow to deploy a Wasm app to a Kubernetes cluster using containerd is as follows:

1. Write the app and compile it as a Wasm binary
2. Package the Wasm binary as an OCI image and store it in an OCI registry
3. Install the appropriate Wasm shim on at least one cluster node and label the node
4. Create a RuntimeClass that specifies the node labels and Wasm shim
5. Create a Pod for the Wasm app (use the Wasm image from step 2)
6. Reference the RuntimeClass in the Pod
7. Deploy the Pod to Kubernetes

All of the following will happen when you deploy the Pod:

1. Kubernetes will schedule the Pod a node matching the node selector in the RuntimeClass
2. The kubelet on the node will pass the work to containerd with the shim info from the RuntimeClass

3. containerd will start the app using the shim requested in the RuntimeClass

Talk is cheap. Let's do it.

Hands-on with Wasm on Kubernetes

Before starting this section, you'll need a clone of the book's GitHub repo and you'll need to be on the `2025` branch.

```
$ git clone https://github.com/nigelpoulton/TKB.git
<Snip>

$ cd TKB

$ git fetch origin

$ git checkout -b 2025 origin/2025
```

Now change into the `wasm` folder.

```
$ cd TKB/wasm
```

In this section, you'll complete all the following steps to write a Wasm app and run it on a multi-node Kubernetes cluster:

1. Install and test the pre-requisites
2. Write and compile the Wasm app
3. Build it into an OCI image and push it to an OCI registry
4. Build and configure a **new** multi-node Kubernetes cluster for Wasm
5. Deploy the app to Kubernetes

In the real world, cloud platforms and tools such as SpinKube will simplify and automate much of the process. However, you'll perform the steps manually, giving you a deeper understanding of everything involved so you're ready to deploy and manage Wasm apps on Kubernetes in the real world.

Install and test the pre-requisites

You'll need all of the following if you plan on following along:

- An up-to-date version of Docker Desktop with Wasm support enabled
- Rust 1.82 or later with the **wasm32-wasip1** target installed
- Spin 3.1.2 or later
- k3d 5.8.1 or later

See Chapter 3 if you haven't installed Docker.

Go to `https://www.rust-lang.org/tools/install` to install Rust.

Once you've installed it, run the following command to install the **wasm32-wasip1** target so Rust can compile code to Wasm binaries.

```
$ rustup target add wasm32-wasip1
info: downloading component 'rust-std' for 'wasm32-wasip1'
info: installing component 'rust-std' for 'wasm32-wasip1'
```

Spin is a popular Wasm framework that includes a Wasm runtime and tools to build and work with Wasm apps. Search the web for *install Fermyon Spin* and follow the installation instructions for your platform.

Run the following command to confirm the installation worked.

```
$ spin --version
spin 3.1.2 (3d37bd8 2025-01-13)
```

With Rust and Spin configured, it's time to write the app.

Write and compile the Wasm app

In this section, you'll use Spin to build and compile a Wasm app.

Run the following **spin new** command from within the **wasm** folder and complete the prompts as shown. This will scaffold a simple Spin app that responds to web requests on port 80 on the /tkb path. TKB is short for *The Kubernetes Book*.

```
$ spin new tkb-wasm -t http-rust
Description []: My first Wasm app
HTTP path [/...]: /tkb
```

You'll have a new directory called **tkb-wasm** containing everything needed to build and run the app.

Change into the **tkb-wasm** directory and list its contents. If your system doesn't have the **tree** command, you can try running an **ls -R** or equivalent Windows command.

```
$ cd tkb-wasm

$ tree
├── Cargo.toml
├── spin.toml
└── src
    └── lib.rs

2 directories, 3 files
```

We're only interested in two files:

- **spin.toml** tells Spin how to build and run the app
- **src/lib.rs** is the app source code

Edit the **src/lib.rs** file so that it returns the text **The Kubernetes Book loves Wasm!**. Only change the text on the line indicated by the annotation in the snippet.

```
use spin_sdk::http::{IntoResponse, Request, Response};
<Snip>
fn handle_tkb_wasm(req: Request) -> anyhow::Result<impl IntoResponse> {
    println!("Handling request to {:?}", req.header("spin-full-url"));
    Ok(Response::builder()
        .status(200)
        .header("content-type", "text/plain")
```

```
        .body("The Kubernetes Book loves Wasm!")            <<----
Only change this line
        .build())
}
```

Save your changes and run a `spin build` to compile the app as a
Wasm binary. Behind the scenes, `spin build` runs a more
complicated `cargo build` command from the Rust toolchain.

```
$ spin build
Building component tkb-wasm with `cargo build --target wasm32-wasip1 --
release`
    Updating crates.io index
    <Snip>
Finished building all Spin components
```

Congratulations, you just built and compiled a Wasm app!

The application binary is called `tkb_wasm.wasm` in the
`target/wasm32-wasip1/release/` folder. This will run on any
machine with the Spin Wasm runtime. Later in the chapter, you'll
run it on a Kubernetes node with the **Spin** Wasm runtime.

Build an OCI image and push it to an OCI registry

Now that you've compiled the app, the next step is to package it as
a container so you can share it on an OCI registry and run it on
Kubernetes.

The first thing you need is a Dockerfile telling Docker how to
package it as an OCI image.

Create a new `Dockerfile` in the `tkb-wasm` folder with the following
content. Be sure to include the periods at the end of the last two
lines.

```
FROM scratch
COPY /target/wasm32-wasip1/release/tkb_wasm.wasm .
COPY spin.toml .
```

The `FROM scratch` line tells Docker to package your Wasm app inside an empty *scratch* image instead of a typical Linux base image. This keeps the image small and helps build a minimal container at runtime. You can do this because Wasm apps don't *need* a container with a Linux filesystem and other constructs. However, platforms such as Docker and Kubernetes use tools that expect to work with basic container constructs and packaging your Wasm app in a scratch image accomplishes this. At runtime, the Wasm app and Wasm runtime will execute inside a minimal container that is basically just namespaces and cgroups (no filesystem etc.).

The first `COPY` instruction copies the compiled Wasm binary into the container's root folder. The second one copies the `spin.toml` file into the same root folder.

The `spin.toml` file tells the spin runtime where the Wasm app is and how to execute it. Right now, it expects the Wasm app to be in the `target/wasm32-wasip1/release` folder, but the Dockerfile copies it to the root folder in the container. This means you need to update the `spin.toml` file to expect it in the root (`/`) folder.

Edit the `spin.toml` file and strip the leading path from the `[component.tkb-wasm] source` field to look like this. The annotation in the snippet is only there to show you which line to change, do not include it in your file.

```
$ vim spin.toml
<Snip>
[component.tkb-wasm]
source = "tkb_wasm.wasm"       <<---- Remove the leading path from this
line
<Snip>
```

At this point, you have all the following:

- A Wasm app (Wasm binary)

- A `spin.toml` file telling the Spin Wasm runtime how to execute the Wasm app
- A `Dockerfile` telling Docker how to build the Wasm app into an OCI image

Run the following command to build the Wasm app into an OCI image. You'll need to use your own Docker Hub username on the last line if you plan on pushing it to a registry in a later step.

```
$ docker build \
  --platform wasi/wasm \
  --provenance=false \
  -t nigelpoulton/k8sbook:wasm-0.2 .
```

The `--platform wasi/wasm` flag sets the OS and Architecture of the image. Tools like `docker run` and containerd can read these attributes at runtime to help them create the container and run the app.

Check the image exists on your local machine. Feel free to run a `docker inspect` and verify the OS and Architecture attributes.

```
$ docker images
REPOSITORY              TAG         IMAGE ID        CREATED
SIZE
nigelpoulton/k8sbook    wasm-0.2    a003c43b1308    7 seconds ago
104kB
```

Notice how small the image is. Similar *hello world* Linux containers are usually several megabytes in size.

Congratulations! You've created a Wasm app and packaged it as an OCI image that you can push to a registry so that Kubernetes can pull it later. You don't have to push the image to a registry, as I have a pre-created image you can use. However, if you do push it to a registry, you'll need to replace the image tag with the one you created in the earlier step. You'll also need an account on the registry you're pushing to.

```
$ docker push nigelpoulton/k8sbook:wasm-0.2
The push refers to repository [docker.io/nigelpoulton/k8sbook]
4073bf46d785: Pushed
7893057c9bbc: Pushed
wasm-0.2: digest:
sha256:a003c43b1308dce78c8654b7561d9a...779c5c9f9b51979c9925f6f size:
695
```

So far, you've written an app, compiled it to Wasm, packaged it as an OCI image, and pushed it to a registry. Next, you'll build and configure a Kubernetes cluster capable of running Wasm apps.

Build and configure a new multi-node Kubernetes cluster for Wasm

This section shows you how to build a new k3d Kubernetes cluster on your local machine and configure it for Wasm. It's based on a custom k3d image that includes pre-installed Wasm shims that other clusters might not include. This means you'll need to build this exact cluster if you want to follow along.

You'll complete all of the following steps:

- Install k3d
- Build a 3-node Kubernetes cluster (one control plane node and two workers)
- Inspect the Wasm configuration on one of the worker nodes
- Label one of the worker nodes so that Kubernetes knows it can run Wasm apps
- Create a `RuntimeClass` so Kubernetes knows where to schedule Wasm apps

Go to the `k3d.io` homepage and scroll down until you find the installation instructions for your platform. Follow the instructions and then run a `k3d --version` command to ensure it is installed correctly.

Once you've installed k3d, run the following command to create a new k3d cluster called **wasm**. Doing this will also change your *kubectl context* to the new cluster.

```
$ k3d cluster create wasm \
     --image ghcr.io/deislabs/containerd-wasm-
shims/examples/k3d:v0.11.1 \
     -p "5005:80@loadbalancer" --agents 2
```

The first line creates a new cluster called **wasm**.

The `--image` flag tells k3d which image to use to build the control plane node and worker nodes. This is a special image that includes containerd Wasm shims.

The `-p` flag creates a load balancer that connects to an Ingress on the cluster and maps port `5005` on your host machine to an Ingress on port `80` inside the cluster.

The `--agents 2` flag creates two worker nodes.

Once the cluster is up, you can test connectivity with the following command. You should see three nodes — one control plane node and two workers.

```
$ kubectl get nodes
NAME                 STATUS    ROLES           AGE    VERSION
k3d-wasm-server-0    Ready     control-plane   17s    v1.27.8+k3s2
k3d-wasm-agent-1     Ready     <none>          15s    v1.27.8+k3s2
k3d-wasm-agent-0     Ready     <none>          15s    v1.27.8+k3s2
```

You need at least one cluster node with both of the following if you want to run Wasm workloads:

1. containerd installed and running
2. A containerd Wasm shim installed and registered

Exec onto the **k3d-wasm-agent-1** worker node and check if containerd is running.

```
$ docker exec -it k3d-wasm-agent-1 ash

$ ps | grep containerd
PID    USER     COMMAND
98     0          containerd
<Snip>
```

Now check the node for Wasm shims. The shim files should be in
the `/bin` directory and named according to the *containerd shim
naming convention*, which prefixes the shim name with `containerd-
shim-` and requires a version number at the end. The following
output shows five shims — `containerd-shim-runc-v2` is the default
shim for executing Linux containers, and the other four are Wasm
shims. The important one for us is the Spin shim called `containerd-
shim-spin-v2.`

```
$ ls /bin | grep shim
containerd-shim-lunatic-v1
containerd-shim-runc-v2
containerd-shim-slight-v1
containerd-shim-spin-v2
containerd-shim-wws-v1
```

The presence of a Wasm shim in the filesystem isn't enough, they
also need to be registered with containerd and loaded as part of the
containerd config.

Check the `config.toml` containerd configuration file for Wasm shim
entries. You can usually find the file in `/etc/containerd`, but k3d
currently stores it in a different location. I've trimmed the output so
it only shows the Wasm runtimes.

```
$ cat /var/lib/rancher/k3s/agent/etc/containerd/config.toml

<Snip>
[plugins.cri.containerd.runtimes.spin]
  runtime_type = "io.containerd.spin.v2"

[plugins.cri.containerd.runtimes.slight]
  runtime_type = "io.containerd.slight.v1"

[plugins.cri.containerd.runtimes.wws]
  runtime_type = "io.containerd.wws.v1"
```

```
[plugins.cri.containerd.runtimes.lunatic]
  runtime_type = "io.containerd.lunatic.v1"
```

You can also run the following command to verify the active containerd config. It parses the output for references to the Spin Wasm shim.

```
$ containerd --config \
  /var/lib/rancher/k3s/agent/etc/containerd/config.toml \
  config dump | grep spin

<Snip>
  [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.spin]
    runtime_type = "io.containerd.spin.v2"

[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.spin.options]
```

You've confirmed that containerd is running and the Spin Wasm shim is present and registered. This means the node can run Spin apps in Wasm containers.

All nodes in your k3d cluster are running the same shims, meaning every node can run Wasm apps and no further work is needed. However, most real-world environments have heterogeneous node configurations where different nodes have different shims and runtimes. In these scenarios, you need to label nodes and create RuntimeClasses to help Kubernetes schedule work to the correct nodes.

We'll label the **agent-1** node with the `wasm=yes` label and create a RuntimeClass that targets nodes with that label.

Run the following command to add the `wasm=yes` label to the **agent-1** worker. You'll need to type `exit` to quit your exec session and return to your host's terminal first.

```
# exit

$ kubectl label nodes k3d-wasm-agent-1 wasm=yes
node/k3d-wasm-agent-1 labeled
```

Verify the operation worked. Your output may include a lot more labels.

```
$ kubectl get nodes --show-labels | grep wasm=yes
NAME                STATUS    ROLES     VERSION          LABELS
k3d-wasm-agent-0    Ready     <none>    v1.27.8+k3s2
beta.kubernetes...,wasm=yes
```

Run the following command to create the **rc-spin** RuntimeClass.

```
$ kubectl apply -f rc-spin.yml
runtimeclass.node.k8s.io/rc-spin created
```

The `scheduling.nodeSelector` field ensures that Pods referencing this RuntimeClass will only be scheduled to nodes with the `wasm=yes` label. The `handler` field tells containerd on the node to use the `spin` shim to execute Wasm apps.

Check you created it correctly.

```
$ kubectl get runtimeclass
NAME        HANDLER    AGE
rc-spin     spin       14s
```

At this point, your Kubernetes cluster has everything it needs to run Wasm workloads — the **agent-1** worker node is labeled and has four Wasm shims installed, and you've created a RuntimeClass to schedule Wasm tasks to the node.

Deploy and test the app

The app is defined in the `app.yml` file in the `wasm` folder of the book's GitHub repo and comprises a Deployment, a Service, and an Ingress.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wasm-spin
spec:
```

```
    replicas: 3
    <Snip>
    template:
      metadata:
        labels:
          app: wasm
      <Snip>
      spec:
        runtimeClassName: rc-spin                  <<---- Referencing
the RuntimeClass
        containers:
          - name: testwasm
            image: nigelpoulton/k8sbook:wasm-0.1   <<---- Pre-created
image
            command: ["/"]
```

The important part of the Deployment YAML is the reference to the RuntimeClass in the Pod spec. This ensures Kubernetes will schedule all three replicas to a node that meets the `nodeSelector` requirements in the RuntimeClass — nodes with the `wasm=yes` label. Kubernetes will schedule all three replicas to the **agent-1** node in our example.

The YAML file also has an Ingress and a Service that I'm not showing. The Ingress directs traffic arriving on the `"/"` path to a ClusterIP Service called **wasm-spin** which forwards the traffic to all Pods with the `app=wasm` label on port `80`. The replicas defined in the Deployment all have the `app=wasm` label.

You can see the traffic flow in Figure 9.6.
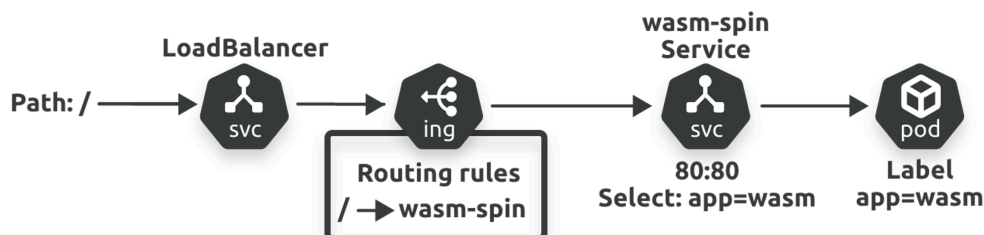
Figure 9.6

In this next step, you'll deploy the app defined in the `app.yml` file. It uses a pre-created Wasm image from the book's Docker Hub repo. If you want to use the image you created in the earlier steps, edit your `app.yml` file and change the `image` field.

```
$ kubectl apply -f app.yml
deployment.apps/wasm-spin created
service/svc-wasm created
ingress.networking.k8s.io/ing-wasm created
```

Check the status of the Deployment with a `kubectl get deploy wasm-spin` command.

Wait for all three replicas to be ready, and then run the following command to ensure they're all scheduled to the **agent-1** worker node.

```
$ kubectl get pods -o wide
NAME                          READY    STATUS      ...    NODE
wasm-spin-5f6fccc557-5jzx6    1/1      Running     ...    k3d-wasm-
agent-1
wasm-spin-5f6fccc557-c2tq7    1/1      Running     ...    k3d-wasm-
agent-1
wasm-spin-5f6fccc557-ft6nz    1/1      Running     ...    k3d-wasm-
agent-1
```

Kubernetes has scheduled all three to the **agent-1** node. This means the label and RuntimeClass worked as expected.

Test the app with the following `curl` command. You can also point your browser to `http://localhost:5005/tkb`.

```
$ curl http://localhost:5005/tkb
The Kubernetes Book loves Wasm!
```

Congratulations, the Wasm app is running on your Kubernetes cluster!

Clean up

If you followed along, you'll have all the following artifacts that you may wish to clean up:

- k3d Kubernetes cluster called **wasm**
- Wasm OCI image stored in an OCI registry
- Wasm OCI image on your local host
- Spin app on your local machine

The easiest way to clean up your Kubernetes cluster is to delete it with this command.

```
$ k3d cluster delete wasm
```

If you want to keep the cluster and only delete the resources, run the following two commands.

```
$ kubectl delete -f app.yml
deployment.apps "wasm-spin" deleted
service "svc-wasm" deleted
ingress.networking.k8s.io "ing-wasm" deleted

$ kubectl delete runtimeclass rc-spin
runtimeclass.node.k8s.io "rc-spin" deleted
```

You can delete the Wasm image on your local machine with the following command. Be sure to substitute the name of your image.

```
$ docker rmi nigelpoulton/k8sbook:wasm-0.1
```

When you created the app with `spin new` and `spin build`, you got a new directory called `tkb-wasm` containing all the application artifacts. Use your favorite tool to delete the directory and all files in it. **Be sure to delete the correct directory!**

Set your Kubernetes context back to the cluster you've been using for the other examples in the book. If you've got Docker Desktop, click the Docker whale and choose the context from the

Kubernetes context option. If you don't have Docker Desktop you can run the following commands. The first one lists your contexts and the second one sets your current context to **docker-desktop**. You'll need to set yours back to the correct context in your environment.

```
$ kubectl config get-contexts
CURRENT    NAME             CLUSTER          AUTHINFO          NAMESPACE
           docker-desktop   docker-desktop   docker-desktop
           lke349416-ctx    lke349416        lke349416-admin   default
*          k3d-wasm         k3d-wasm         admin@k3d-wasm

$ kubectl config current-context docker-desktop
docker-desktop
```

Chapter Summary

Wasm is powering the third wave of cloud computing, and platforms like Docker and Kubernetes are evolving to work with it. Docker can already build Wasm apps into container images, run them with `docker run`, and host them on Docker Hub. Projects like containerd and [runwasi](#) make it possible to run Wasm containers on Kubernetes and projects like SpinKube make it easy.

Wasm is a binary instruction set that programming languages use as a compilation target — instead of compiling to something like *Linux on ARM*, you compile to *Wasm*.

Compiled Wasm apps are tiny binaries that can run anywhere with a Wasm runtime. Wasm apps are smaller, faster, more portable, and more secure than traditional Linux containers. However, at the time of writing, Wasm apps cannot do everything that Linux containers can.

The high-level process is to write apps in existing languages, compile them as Wasm binaries, and then use tools such as `docker build` and `docker push` to build them into OCI images and push them to OCI registries. From there, you can wrap them in

Kubernetes Pods and run them on Kubernetes just like regular containers.

Kubernetes clusters running containerd have a growing choice of Wasm runtimes that are implemented as *containerd shims*. To run a Wasm app on a Kubernetes cluster with containerd, you need to install and register a Wasm shim on at least one worker node. You then need to label the node and reference the label in a RuntimeClass so the scheduler can assign Wasm apps to it. In the real world, projects like SpinKube and cloud services automate many of these tasks.

For further information, I recommend you look into the [SpinKube](#) project and the containerd shim lifecycle [community proposal](#).

10: Service discovery deep dive

In this chapter, you'll learn about service discovery, why it's important, and how it's implemented in Kubernetes. You'll also learn some troubleshooting tips.

You'll get the most from this chapter if you understand Kubernetes Services. If you don't already know this, you should read Chapter 7 first.

I've split the chapter into the following sections:

- Setting the scene
- The Service registry
- Service registration
- Service discovery
- Service discovery and Namespaces
- Troubleshooting service discovery

> **Note:** The word *service* has a lot of different meanings, so I've capitalized the first letter when referring to the Service resource in the Kubernetes API.

Setting the scene

Finding things on busy platforms like Kubernetes is hard, *service discovery* makes it easy.

Most Kubernetes clusters run hundreds or thousands of microservices apps. Most of them behind their own Service and get their own reliable names and IPs. When one app talks to another, it actually talks to the other app's Service. For the remainder of this chapter, any time we say an app needs to find or talk to another

app, we mean it needs to find or talk to the **_Service_** in front of the other app.

Figure 10.1 shows **app-a** talking to **app-b** via its Service.
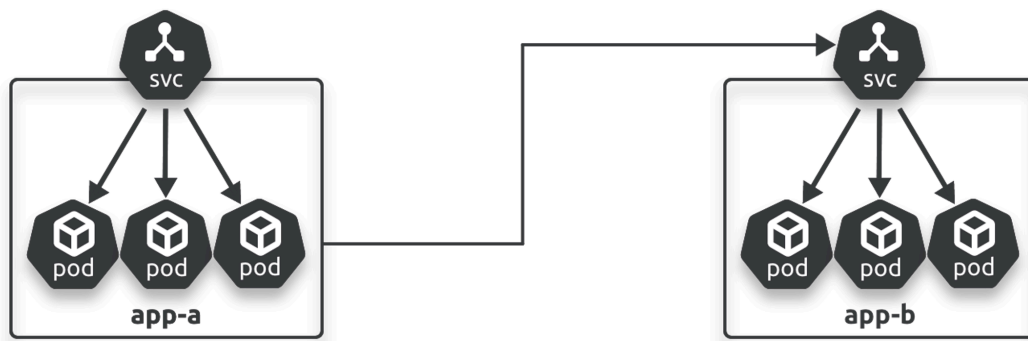
Figure 10.1 - Apps connect via Services

Apps need two things to be able to send requests to other apps:

1. A way to know the name of the other app (the name of its Service)
2. A way to convert the name into an IP address

Developers are responsible for step 1 — ensuring apps know the _names_ of the other apps and microservices they consume. Kubernetes is responsible for step 2 — converting names to IP addresses.

Figure 10.2 is a high-level view of the overall process with four main steps:

- **Step 1:** The developer configures **app-a** to talk to **app-b**
- **Step 2: app-a** asks Kubernetes for the IP address of **app-b**
- **Step 3:** Kubernetes returns the IP address
- **Step 4: app-a** sends requests to **app-b's** IP address
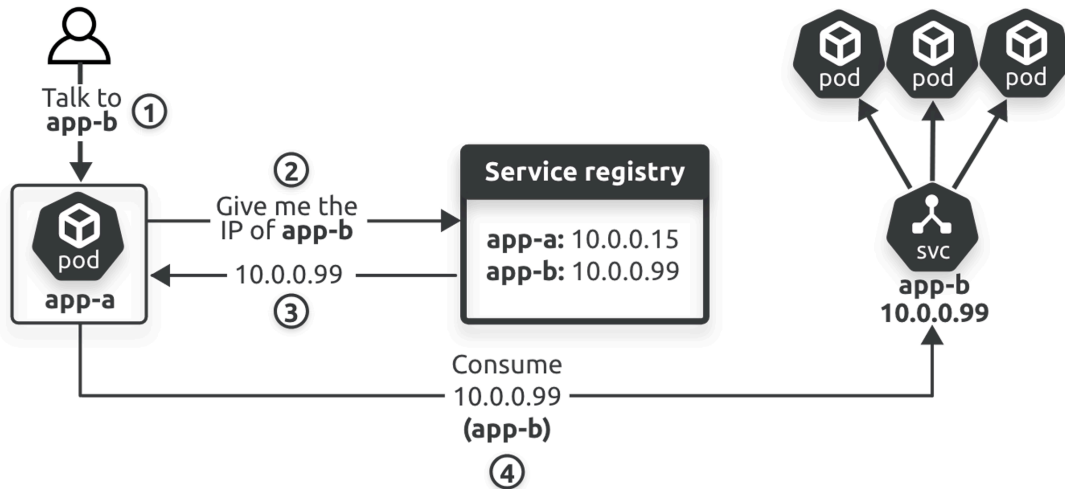
Figure 10.2

Step 1 is the only manual step. Kubernetes handles steps 2, 3, and 4 automatically.

Let's take a closer look.

The service registry

The job of a service registry is to maintain a list of Service names and their associated IP addresses.

Every Kubernetes cluster has a built-in *cluster DNS* that it uses as its service registry. It's a Kubernetes-native application running on the control plane of every Kubernetes cluster as two or more Pods managed by a Deployment and fronted by its own Service. The Deployment is usually called **coredns** or **kube-dns**, and the Service is always called **kube-dns**.

Figure 10.3 shows the Kubernetes service registry architecture. It also shows a Service registering its name and IP and two containers using it for service discovery. As you'll find out later, Kubernetes makes *service registration* and *service discovery* automatic.
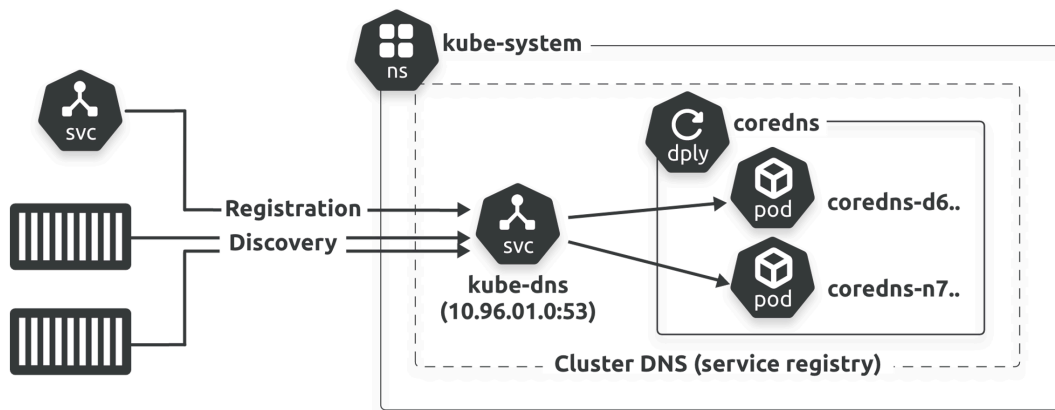
Figure 10.3 - Cluster DNS architecture

The following commands show the Pods, Deployment, and Service that comprise the cluster DNS (service registry). They match what is in Figure 10.3, and you can run the commands on your own cluster.

This command lists the Pods running the cluster DNS. They normally use the `registry.k8s.io/coredns/coredns` image, but some other clusters use a different image and may call the Pods and Deployment **kube-dns** instead of **coredns**.

```
$ kubectl get pods -n kube-system -l k8s-app=kube-dns
NAME                        READY   STATUS    RESTARTS   AGE
coredns-76f75df574-d6nn5    1/1     Running   0          13d
coredns-76f75df574-n7qzk    1/1     Running   0          13d
```

The next command shows the Deployment that manages the Pods. It ensures there is always the correct number of cluster DNS Pods.

```
$ kubectl get deploy -n kube-system -l k8s-app=kube-dns
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
coredns    2/2     2            2           13d
```

This final command shows the Service in front of the cluster DNS Pods. It's always called **kube-dns**, but it gets a different IP on each cluster. As you'll find out later, Kubernetes automatically configures every container to use this IP for service discovery.

```
$ kubectl get svc -n kube-system -l k8s-app=kube-dns
NAME        TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
kube-dns    ClusterIP   10.96.0.10     <none>
53/UDP,53/TCP,9153/TCP   13d
```

In summary, every Kubernetes cluster runs an internal cluster DNS service that it uses as its service registry. It maps every Service's name and IP, and runs on the control plane as a set of Pods managed by a Deployment and fronted by a Service.

Let's switch our focus to service registration.

Service registration

The most important thing to know about service registration on Kubernetes is that it's automatic!

At a high level, you develop applications and put them behind Services for reliable names and IPs. Kubernetes automatically registers these Service names and IPs with the service registry.

From now on, we'll call the service registry the *cluster DNS.*

There are three steps in service registration:

1. Give the Service a name
2. Assign the Service an IP
3. Register the name and IP with the cluster DNS

Developers are responsible for point one. Kubernetes handles points two and three.

Consider a quick example.

You're developing a new web app that other apps will connect to using the **valkyrie-web** name. To accomplish this, you deploy the app behind a Kubernetes Service called **valkyrie-web**. Kubernetes

ensures the Service name is unique and automatically assigns it an IP address (ClusterIP). It also registers the name and IP in the cluster DNS.

The registration process is automatic because the cluster DNS is a *Kubernetes-native application* that watches the API server for new Services. Whenever it sees a new one, it automatically registers its name and IP. This means your applications don't need any service registration logic — you put them behind a Service, and the cluster DNS does everything else.

Figure 10.4 summarises the service registration process and adds some of the details from Chapter 7.

```
┌──────────────┐   ┌──────────────┐   ┌──────────────┐   ┌──────────────┐
│ POST Service │   │   Service    │   │    Config    │   │ Cluster DNS  │
│ YAML to API  │──▶│  created &   │──▶│ persisted to │──▶│   sees new   │
│   server     │   │  assigned    │   │ cluster store│   │   Service    │
│              │   │  ClusterIP   │   │              │   │              │
└──────────────┘   └──────────────┘   └──────────────┘   └──────────────┘

┌──────────────┐   ┌──────────────┐   ┌──────────────┐   ┌──────────────┐
│ Cluster DNS  │   │EndpointSlices│   │ Kube-proxies │   │  IPVS rules  │
│ creates DNS  │──▶│ created with │──▶│ pull Service │──▶│   created    │
│   records    │   │   Pod IPs    │   │    config    │   │              │
└──────────────┘   └──────────────┘   └──────────────┘   └──────────────┘
```
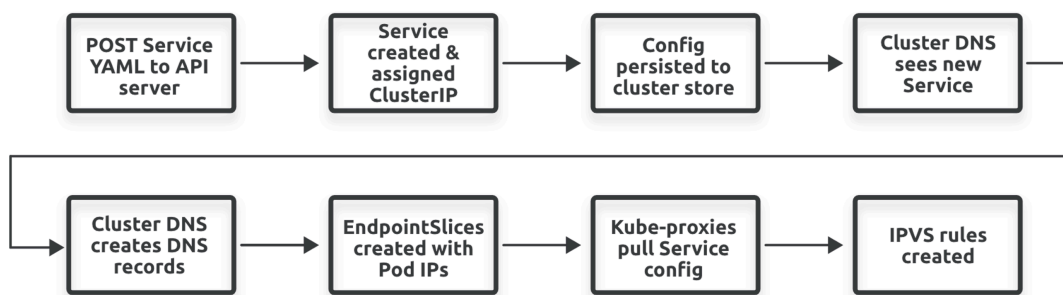
Figure 10.4 - Service registration flow

Let's step through the diagram.

You deploy a new app defined in a YAML file describing a Deployment and a Service. You post it to Kubernetes, where it's authenticated and authorized. Kubernetes allocates a ClusterIP to the Service and persists its configuration to the cluster store. The cluster DNS observes the new Service and registers the appropriate DNS A and SRV records. Associated EndpointSlice objects are created to hold the list of healthy Pod IPs that match the Service's label selector. The kube-proxy process on every cluster node observes the new objects and creates local routing rules (IPVS) so that requests to the Service's ClusterIP get routed to Pods.

In summary, every app sits behind a Service for a reliable name and IP. The cluster DNS watches the API server for new Service objects and automatically registers their names and IPs.

Let's look at service discovery.

Service discovery

Applications talk to other applications via names. However, they need to convert these names into IP addresses, which is where service discovery comes into play.

Assume you have a cluster with two apps called **enterprise** and **cerritos**. The **enterprise** app sits behind a ClusterIP Service called **ent**, and the **cerritos** app sits behind one called **cer**. Kubernetes automatically assigned both Services a ClusterIP, and the cluster DNS automatically registered them. Right now, things are as follows.

App	Service name	ClusterIP
Enterprise	ent	192.168.201.240
Cerritos	cer	192.168.200.217
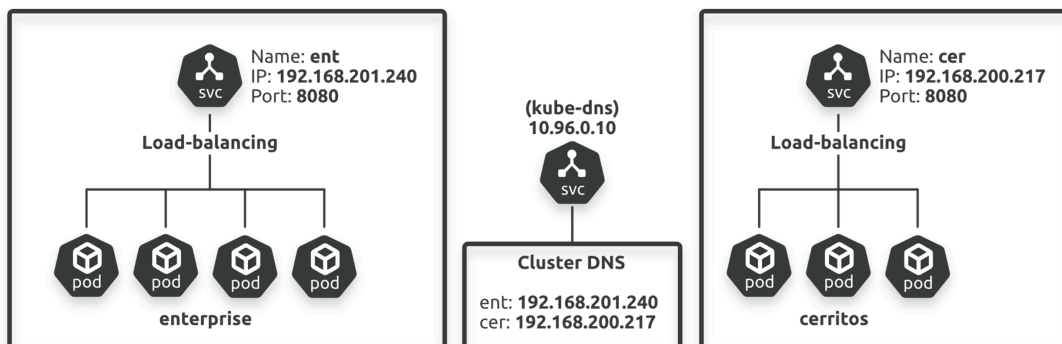
Figure 10.5

If either of the apps wants to connect to the other, it needs to know its name and how to convert it to an IP.

Developers are responsible for coding applications with the names of the applications they consume, but Kubernetes provides the mechanisms to convert the names to IPs.

Consider a quick example where the **enterprise** app from Figure 10.5 needs to send requests to the **cerritos** app. For this to work, the enterprise app developers need to configure it with the name of the Service in front of the cerritos app. Assuming they did this, the enterprise app will send requests to **cer**. However, it needs a way to convert **cer** into an IP address. Fortunately, Kubernetes configures every container to ask the cluster DNS to convert names to IPs. This means the **enterprise** app containers will send the **cer** name to the cluster DNS, and the cluster DNS will return the ClusterIP. The app then sends requests to the IP.

As previously mentioned, Kubernetes configures every container to use the cluster DNS for service discovery. It does this by automatically configuring every container's `/etc/resolv.conf` file with the IP address of the cluster DNS Service. It also adds search domains to append to unqualified names.

An *unqualified name* is a short name such as **ent**. Appending a search domain converts it to a fully qualified domain name (FQDN) such as `ent.default.svc.cluster.local`.

The following extract is from a container's `/etc/resolv.conf` file configured to send service discovery requests (DNS queries) to the cluster DNS at `10.96.0.10`. It also lists three search domains to append to unqualified names.

```
$ cat /etc/resolv.conf
search default.svc.cluster.local svc.cluster.local cluster.local
nameserver 10.96.0.10          <<---- ClusterIP of internal cluster
DNS
options ndots:5
```

The following command proves the **nameserver** IP in the previous `/etc/resolv.conf` file matches the IP address of the cluster DNS (the **kube-dns** Service).

```
$ kubectl get svc -n kube-system -l k8s-app=kube-dns
NAME       TYPE        CLUSTER-IP     PORT(S)                 AGE
kube-dns   ClusterIP   10.96.0.10     53/UDP,53/TCP,9153/TCP  13d
```

Now that you know the basics, let's see how the **enterprise** app from Figure 10.5 sends requests to the **cerritos** app.

First, the **enterprise** app needs to know the name of the **cer** Service fronting the cerritos app. That's the job of the enterprise app developers. Assuming it knows the name, it sends requests to **cer**. The network stack of the app's container automatically sends the name to the cluster DNS, asking for the associated IP. The cluster DNS responds with the ClusterIP of the **cer** Service, and the request gets sent to the IP. However, ClusterIPs are virtual IPs that require additional magic to ensure requests eventually reach the **cerritos** <u>Pods</u>.

ClusterIP routing

ClusterIPs are on a special network called the *service network* and there are no routes to it! This means every container sends ClusterIP traffic to its *default gateway*.

> **Terminology:** A default gateway is where a system sends network traffic when it doesn't know where else to send it. Default gateways then forward traffic to another device, hoping the next device will know where to send it.

The container's default gateway sends the traffic to the *node* it's running on. The node doesn't have a route to the service network either, so it sends it to its own default gateway. This causes the

node's kernel to process the traffic, which is where the magic happens…

Every Kubernetes node runs a system service called **kube-proxy** that implements a controller watching the API server for new Services and EndpointSlice objects. Whenever it sees them, it creates rules in the kernel to intercept ClusterIP traffic and forward it to individual Pod IPs.

This means that every time a node's kernel processes traffic for a ClusterIP, it redirects it to the IP of a healthy Pod matching the Service's label selector.

Summarising service discovery

Let's quickly summarise the service discovery process with the help of the flow diagram in Figure 10.6.
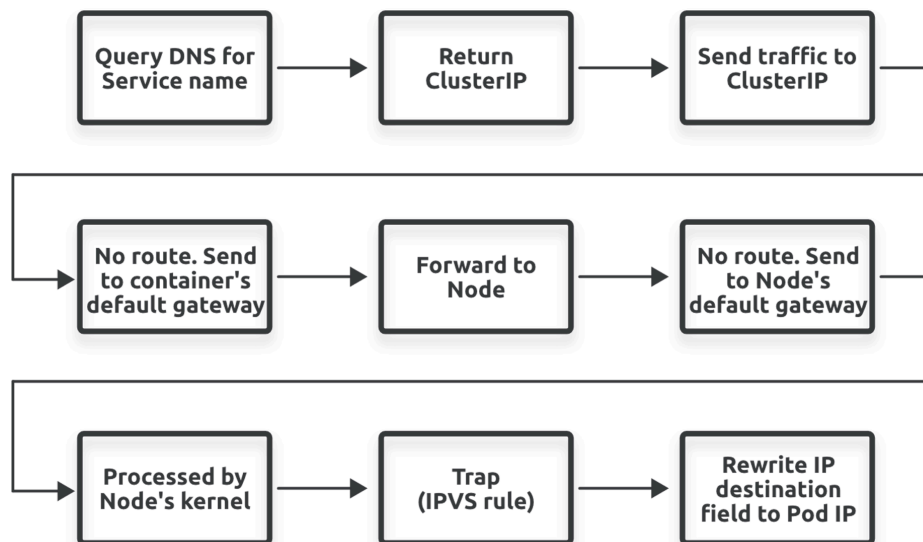
Figure 10.6

The enterprise app sends the request to the **cer** Service. The container converts this name to an IP address by sending it to the IP address of the cluster DNS configured in its `/etc/resolv.conf`

file. The cluster DNS returns the Service's ClusterIP, and the container sends the traffic to that IP address. However, ClusterIPs are on the *service network* and the container doesn't have a route to it. So, it sends it to its default gateway, which forwards it to the node it's running on. The node doesn't have a route either, so it sends it to its own default gateway. This causes the node's kernel to process the request and redirect it to the IP address of a Pod that matches the Service's label selector.

Service discovery and Namespaces

Every Kubernetes object gets a name in the *cluster address space*, and you can partition the address space with Namespaces.

The cluster address space is a DNS domain that we usually call the *cluster domain*. On most clusters, it's `cluster.local`, and object names have to be unique within it. For example, you can only have one Service called **cer** in the default Namespace, and its name will be `cer.default.svc.cluster.local`.

Long names like this are called *fully qualified domain names (FQDN)*, and the format is `<object-name>.<namespace>.svc.cluster.local.`

You can use Namespaces to partition the address space below the cluster domain. For example, if your cluster has two Namespaces called **dev** and **prod**, the address space will be partitioned as follows:

- **dev:** `<service-name>.dev.svc.cluster.local`
- **prod:** `<service-name>.prod.svc.cluster.local`

Object names must be unique *within* a Namespace but not *across* Namespaces. As a quick example, Figure 10.7 shows a single cluster divided into two Namespaces called **dev** and **prod**. Both Namespaces have identical instances of the **cer** Service. This makes

Namespaces an option for running parallel dev and prod configurations on the same cluster.
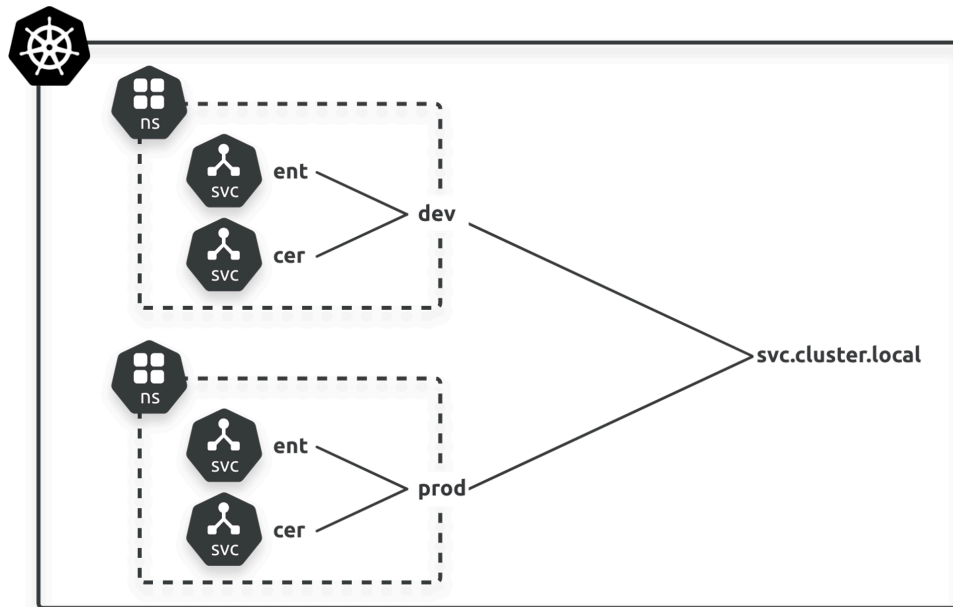
Figure 10.7 - Identical configurations in different Namespaces

Apps can use short names such as **ent** and **cer** to connect to Services in the local Namespace, but they must use fully qualified domain names to connect to Services in remote Namespaces.

Let's walk through a quick example.

Service discovery example

The following YAML is from the `sd-example.yml` file in the `service-discovery` folder of the book's GitHub repo. It deploys the configuration from Figure 10.8.

The file defines two Namespaces, two Deployments, two Services, and a standalone jump Pod. The Deployments and Services have identical names as they're in different Namespaces. The jump Pod is

only deployed to the **dev** Namespace. I've snipped the YAML in the book.
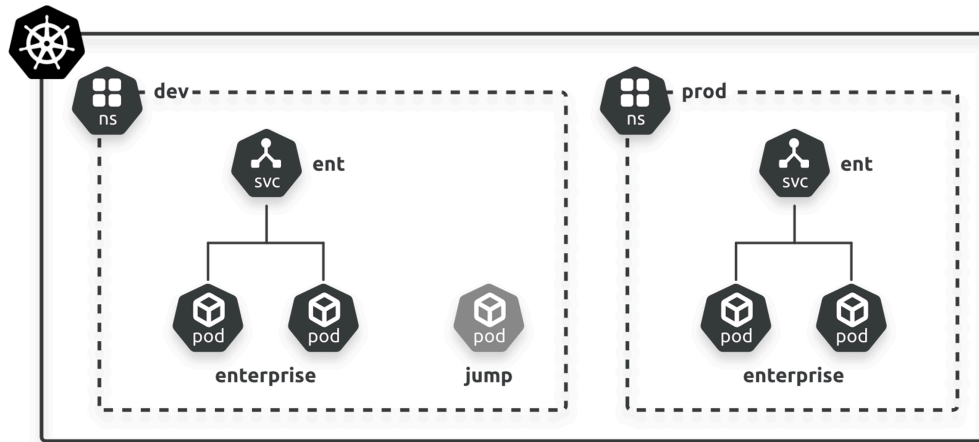
Figure 10.8

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: dev
---
apiVersion: v1
kind: Namespace
metadata:
  name: prod
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: enterprise
  namespace: dev
spec:
  replicas: 2
  template:
    spec:
      containers:
      - image: nigelpoulton/k8sbook:text-dev
        name: enterprise-ctr
        ports:
        - containerPort: 8080
---
apiVersion: apps/v1
kind: Deployment
metadata:
```

```yaml
  name: enterprise
  namespace: prod
spec:
  replicas: 2
  template:
    spec:
      containers:
      - image: nigelpoulton/k8sbook:text-prod
        name: enterprise-ctr
        ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: ent
  namespace: dev
spec:
  selector:
    app: enterprise
  ports:
    - port: 8080
  type: ClusterIP
---
apiVersion: v1
kind: Service
metadata:
  name: ent
  namespace: prod
spec:
  selector:
    app: enterprise
  ports:
    - port: 8080
  type: ClusterIP
---
apiVersion: v1
kind: Pod
metadata:
  name: jump
  namespace: dev
spec:
  terminationGracePeriodSeconds: 5
  containers:
  - name: jump
    image: ubuntu
    tty: true
    stdin: true
```

Run the following command to deploy everything. You need to run the command from within the `service-discovery` directory.

```
$ kubectl apply -f sd-example.yml
namespace/dev created
namespace/prod created
deployment.apps/enterprise created
deployment.apps/enterprise created
service/ent created
service/ent created
pod/jump-pod created
```

Check that Kubernetes deployed everything correctly. I've trimmed the outputs to fit the page, and I'm only showing some of the objects.

```
$ kubectl get all --namespace dev
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/enterprise    2/2     2            2           51s

NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
service/ent    ClusterIP   10.96.138.186   <none>        8080/TCP   51s
<Snip>

$ kubectl get all --namespace prod
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/enterprise    2/2     2            2           1m24s

NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
service/ent    ClusterIP   10.96.147.32    <none>        8080/TCP   1m25s
<snip>
```

You have two Namespaces called **dev** and **prod**, and each has an instance of the **enterprise** app and an instance of the **ent** Service. The **dev** Namespace also has a standalone Pod called **jump**.

Let's see how service discovery works *within* a Namespace and *across* Namespaces.

You'll do all of the following:

1. Log on to the jump Pod in the **dev** Namespace
2. Check its `/etc/resolv.conf` file

3. Connect to the **ent** Service in the *local* **dev** Namespace
4. Connect to the **ent** Service in the *remote* **prod** Namespace

The version of the app in each Namespace returns a different message so you can be sure you've connected to the right one.

Open an interactive exec session to the **jump** Pod's container. Your terminal prompt will change to indicate you're attached to the container.

```
$ kubectl exec -it jump --namespace dev -- bash
root@jump:/#
```

Inspect the contents of the container's `/etc/resolv.conf` file. It should have the IP address of your cluster's **kube-dns** Service as well as the search domain for the **dev** Namespace (`dev.svc.cluster.local`)

```
# cat /etc/resolv.conf
search dev.svc.cluster.local svc.cluster.local cluster.local
nameserver 10.96.0.10
options ndots:5
```

Install the `curl` utility.

```
# apt-get update && apt-get install curl -y
<snip>
```

Run the following `curl` command to connect to the **ent** Service on port `8080`. This will connect you to the instance in the local **dev** Namespace.

```
# curl ent:8080
Hello from the DEV Namespace!
Hostname: enterprise-76fc64bd9-lvzsn
```

The *Hello from the DEV Namespace* response proves the connection reached the instance in the **dev** Namespace.

The container automatically appended `dev.svc.cluster.local` to the name and sent the query to the cluster DNS specified in its `/etc/resolv.conf` file. The cluster DNS returned the ClusterIP for the **ent** Service in the local **dev** Namespace and the app sent the traffic to that IP address. En route to the node's default gateway, the traffic caused a trap in the node's kernel, resulting in the kernel redirecting it to a Pod hosting the app.

Run another `curl` command, but this time append the domain name of the **prod** Namespace. This will cause the cluster DNS to return the ClusterIP of the Service in the **prod** Namespace.

```
# curl ent.prod.svc.cluster.local:8080
Hello from the PROD Namespace!
Hostname: enterprise-5cfcd578d7-nvzlp
```

This time, the response comes from a Pod in the **prod** Namespace.

The tests prove that Kubernetes automatically resolves short names to the local Namespace, and that you need to specify FQDNs to connect across Namespaces.

Type `exit` to detach your terminal from the jump Pod.

Troubleshooting service discovery

Kubernetes makes service registration and service discovery automatic. However, a lot is happening behind the scenes, and knowing how to inspect and restart things is helpful.

As mentioned, Kubernetes uses the cluster DNS as its built-in service registry. This runs as one or more managed Pods with a Service object providing a stable endpoint. The important components are:

- **Pods:** Managed by the **coredns** Deployment

- **Service:** A ClusterIP Service called **kube-dns** listening on port `53` TCP/UDP
- **EndpointSlice objects:** Names pre-fixed with **kube-dns**

All of these objects are in the **kube-system** Namespace and tagged with the **k8s-app=kube-dns** label to help you find them.

Check that the **coredns** Deployment and its Pods are running.

```
$ kubectl get deploy -n kube-system -l k8s-app=kube-dns
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
coredns   2/2     2            2           14d

$ kubectl get pods -n kube-system -l k8s-app=kube-dns
NAME                       READY   STATUS    RESTARTS   AGE
coredns-76f75df574-6q7k7   1/1     Running   0          14d
coredns-76f75df574-krnr7   1/1     Running   0          14d
```

Check the logs from each of the **coredns** Pods. The following output is typical of a working DNS Pod. You'll need to use the name of a Pod from your environment.

```
$ kubectl logs coredns-76f75df574-n7qzk -n kube-system
.:53
[INFO] plugin/reload: Running configuration SHA512 = 591cf328cccc12b...
CoreDNS-1.11.1
linux/arm64, go1.20.7, ae2bbc2
```

Now check the Service and EndpointSlice objects. The output should show the service is up, has an IP address in the ClusterIP field, and is listening on port `53` TCP/UDP.

The ClusterIP address for the **kube-dns** Service must match the IP address in the `/etc/resolv.conf` files of all containers on the cluster. If it doesn't, containers will send DNS requests to the wrong place.

```
$ kubectl get svc kube-dns -n kube-system
NAME       TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)
AGE
kube-dns   ClusterIP   10.96.0.10   <none>
53/UDP,53/TCP,9153/TCP   14d
```

The associated **kube-dns** EndpointSlice object should also be up and have the IP addresses of the **coredns** <u>Pods</u> listening on port `53`.

```
$ kubectl get endpointslice -n kube-system -l k8s-app=kube-dns
NAME             ADDRESSTYPE   PORTS        ENDPOINTS
AGE
kube-dns-jb72g   IPv4          9153,53,53   10.244.1.9,10.244.1.14
14d
```

Once you've verified the fundamental DNS components are up and working, you can perform more detailed and in-depth troubleshooting. Here are some simple tips.

Start a troubleshooting Pod with your favorite networking tools installed (ping, traceroute, curl, dig, nslookup, etc.). The `registry.k8s.io/e2e-test-images/jessie-dnsutils` image is a popular choice if you don't have your own custom image. You can go to `explore.ggcr.dev` to browse the `registry.k8s.io/e2e-test-images` repo for newer versions.

The following command starts a new standalone Pod called **dnsutils** and will connect your terminal. It's based on the image just mentioned and may take a few seconds to start.

```
$ kubectl run -it dnsutils \
  --image registry.k8s.io/e2e-test-images/jessie-dnsutils:1.7
```

A common way to test if the cluster DNS is working is to use `nslookup` to resolve the **kubernetes** Service. This runs on every cluster and exposes the API server to all Pods. The query should return the name `kubernetes.default.svc.cluster.local` and its IP address.

```
# nslookup kubernetes
Server:    10.96.0.10
Address:   10.96.0.10#53
Name:      kubernetes.default.svc.cluster.local
Address:   10.96.0.1
```

The first two lines should show the IP address of your cluster DNS. The last two should show the FQDN of the **kubernetes** Service and its ClusterIP. You can verify the ClusterIP of the **kubernetes** Service by running a `kubectl get svc kubernetes` command.

Errors such as *nslookup: can't resolve kubernetes* are indicators that DNS isn't working. A possible solution is to delete the **coredns** Pods. This will cause the **coredns** Deployment to recreate them.

The following command deletes the DNS Pods. If you're still logged on to the **dnsutils** Pod, you'll need to type `exit` to disconnect before running the command.

```
$ kubectl delete pod -n kube-system -l k8s-app=kube-dns
pod "coredns-76f75df574-d6nn5" deleted
pod "coredns-76f75df574-n7qzk" deleted
```

Run a `kubectl get pods -n kube-system -l k8s-app=kube-dns` to verify they've restarted and then test DNS again.

Clean up

Run the following commands to clean up.

```
$ kubectl delete pod dnsutils
```

```
$ kubectl delete -f sd-example.yml
```

Chapter summary

In this chapter, you learned that Kubernetes uses the internal cluster DNS for service registration and service discovery. It's a Kubernetes-native application that watches the API server for newly created Service objects and automatically registers their names and IPs. The kubelet on each node also configures all containers to use the cluster DNS for service discovery.

The cluster DNS resolves Service names to ClusterIPs. These are stable virtual IPs on a special network called the service network. There are no routes to this network, but the kube-proxy configures all cluster nodes to redirect ClusterIP traffic to Pod IPs on the Pod network.

11: Kubernetes storage

Storing and retrieving data is critical to most real-world business applications. Fortunately, Kubernetes has a *persistent volume subsystem* that makes it easy to connect external storage systems that provide advanced data management services such as backup and recovery, replication, snapshots, encryption, and more.

I've divided the chapter into the following sections:

- The big picture
- Storage providers
- The Container Storage Interface (CSI)
- The Kubernetes persistent volume subsystem
- Dynamic provisioning with Storage Classes
- Hands-on

Kubernetes supports a variety of external storage systems. These include enterprise-class storage systems from providers such as EMC, NetApp, and all the major cloud providers.

I've based this chapter's hands-on examples on a hosted Kubernetes service called Linode Kubernetes Engine (LKE). It's simple to build and relatively cheap, but the examples won't work on other clouds. This is because every cloud has its own storage plugins with their own options and features. I've included files that work on Google Kubernetes Engine (GKE), but I can't include examples for every cloud. Fortunately, they all work on similar principles, and the things you'll learn in this chapter will serve as a solid foundation for working with other clouds.

The big picture

Kubernetes supports many types of storage from many different providers. These include *block, file,* and *object* storage from various external systems that can be in the cloud or your on-premises data centers.

Figure 11.1 shows the high-level architecture.

Storage providers
(Portworx, NetApp, EMC,
AWS EBS, Azure File...)

**Plugin layer
(CSI)**

pv   pvc   sc   ...

Persistent volume subsystem
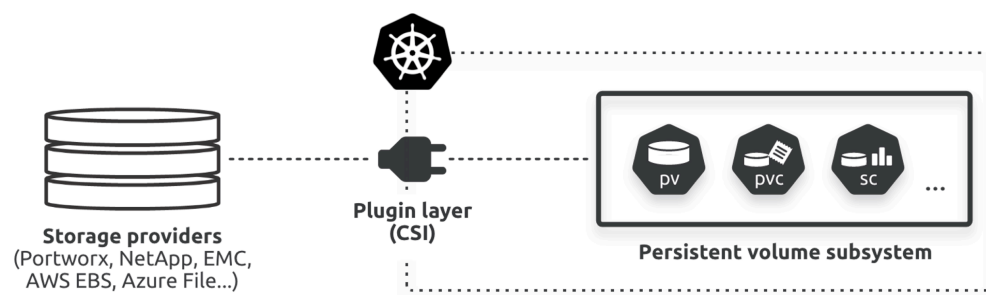
Figure 11.1

The *storage providers* are on the left. As mentioned, these are the external systems providing advanced storage services and can be on-premises systems such as EMC and NetApp, or storage services provided by your cloud.

In the middle of the diagram is the plugin layer. This is the interface between the external storage systems on the left and Kubernetes on the right. Modern plugins use the *Container Storage Interface (CSI)*, which is an industry-standard storage interface for container orchestrators such as Kubernetes. If you're a developer writing storage plugins, the CSI abstracts the internal Kubernetes machinery and allows you to develop *out-of-tree*.

> **Note:** Before the CSI, we developed all storage plugins as part of the main Kubernetes code tree (*in-tree*). This forced them to be open source and tied plugin updates and bug fixes to the Kubernetes release cycle. This was problematic for plugin

developers as well as the Kubernetes maintainers. Fortunately, CSI plugins don't need to be open source, and we can release them whenever required.

On the right of Figure 11.1 is the Kubernetes persistent volume subsystem. This is a standardized set of API objects that make it easy for applications to consume storage. There are a growing number of storage-related API objects, but the core ones are:

- PersistentVolumes (PV)
- PersistentVolumeClaims (PVC)
- StorageClasses (SC)

Throughout the chapter, we'll refer to these in a few different ways. Sometimes, we'll use their PascalCase truncated names — *PersistentVolume, PersistentVolumeClaim,* and *StorageClass.* Sometimes we'll use their acronyms — PV, PVC, and SC. And sometimes we'll just call them *persistent volumes,* and *storage classes*, etc.

PVs map to external volumes, PVCs grant access to PVs, and SCs make it all automatic and dynamic.

Consider the quick AWS example and workflow shown in Figure 11.2.
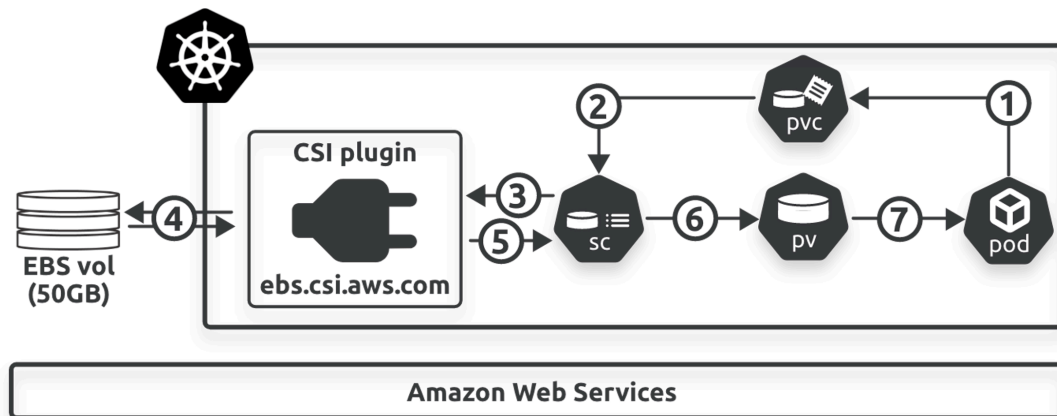
Figure 11.2 - Volume provisioning workflow

1. The Pod on the far right needs a 50GB volume and requests it via a PersistentVolumeClaim (PVC)
2. The PVC asks the StorageClass to create a new PV and associated volume on the AWS backend
3. The SC makes the call to the AWS backend via the AWS CSI plugin
4. The CSI plugin creates the 50GB EBS volume on AWS
5. The CSI plugin reports the creation of the external volume back to the SC
6. The SC creates the PV and maps it to the EBS volume on the AWS back end
7. The Pod mounts the PV and uses it

Before digging deeper, it's worth noting that Kubernetes has mechanisms to prevent multiple Pods from writing to the same PV. It also forces a 1:1 mapping between external volumes and PVs — you cannot map a single 50GB external volume to 2 x 25GB PVs.

Let's dig a bit deeper.

Storage Providers

As previously mentioned, Kubernetes lets you use storage from a wide range of external systems. We usually call these *providers* or *provisioners*.

Each *provider* supplies its own *CSI plugin* that exposes the backend's features and configuration options.

The provider usually distributes the plugin via a Helm chart or YAML installer. Once installed, the plugin runs as a set of Pods in the `kube-system` Namespace.

Some obvious restrictions apply. For example, you can't provision and mount AWS EBS volumes if your cluster is on Microsoft Azure. Locality restrictions may also apply. For example, Pods may have to be in the same region or zone as the storage they're accessing.

Other options, such as volume size, protection level, snapshot schedule, replication settings, encryption configuration, and more, are all configured via the backend's CSI plugin. Not all backends support the same features, and it's your responsibility to read the plugin's documentation and configure it properly.

The Container Storage Interface (CSI)

The CSI is an open-source project that defines an industry-standard interface so container orchestrators can leverage external storage resources in a uniform way. For example, it gives storage providers a documented interface to work with. It also means that CSI plugins should work on any orchestration platform that supports the CSI.

You can find a relatively up-to-date list of CSI plugins in the following repository. The repository refers to plugins as *drivers*.

- `https://kubernetes-csi.github.io/docs/drivers.html`

Most cloud platforms pre-install CSI plugins for the cloud's native storage services. You'll have to install plugins for third-party storage systems manually, but as previously stated, they're usually available as Helm charts or YAML files from the provider. Once installed, CSI plugins typically run as a set of Pods in the `kube-system` Namespace.

The Kubernetes persistent volume subsystem

The Persistent Volume Subsystem is a set of API objects that allow applications to request and access storage. It has the following resources that we'll look at and work with:

- PersistentVolumes (PV)
- PersistentVolumeClaims (PVC)
- StorageClasses (SC)

As previously mentioned, **PVs** make external volumes available on Kubernetes. For example, if you want to make a 50GB AWS volume available on your cluster, you'll need to map it to a PV. If a Pod wants to use it, it needs a **PVC** granting it access to the PV. **SCs** allow applications to create PVs and backend volumes dynamically.

Let's walk through another example.

Assume you have an external storage system with the following tiers of storage:

- Fast block (flash)
- Fast encrypted block (flash)
- Slow block (mechanical)
- File (NFS)

You create a StorageClass for each, so that all four tiers are available to Kubernetes.

External tier	Kubernetes StorageClass name	CSI plugin
Flash	sc-fast-block	csi.xyz.block
Flash encrypted	sc-fast-encrpted	csi.xyz.block
Mechanical disk	sc-slow	csi.xyz.block
NFS Filestore	sc-file	csi.xyz.file

Imagine you're asked to deploy a new application requiring 100GB of fast encrypted block storage. To accomplish this, you create a YAML file defining a Pod that references a PVC requesting a 100GB volume from the **sc-fast-encrypted** storage class.

You deploy the app by sending the YAML file to the API server. The SC controller observes the new PVC and instructs the CSI plugin to provision 100GB of encrypted flash storage on the external storage system. The external system creates the volume and reports back to the CSI plugin, which then informs the SC controller that maps it to a new PV. The Pod uses the PVC to mount and use the PV.

It's OK if some of this is still confusing. The hands-on examples will clarify everything.

Dynamic provisioning with Storage Classes

Storage classes are resources in the `storage.k8s.io/v1` API group. The resource type is `StorageClass`, and you define them in regular YAML files. You can use the `sc` shortname when using `kubectl`.

> **Note:** You can run a `kubectl api-resources` command to see a full list of API resources and their shortnames. It also shows each resource's API group and what its equivalent `kind` is.

As the name suggests, *StorageClasses* let you define different *classes* of storage that apps can request. How you define your classes is up to you and will depend on the types of storage you

have available. For example, your cloud may offer the following four
types of storage:

- Fast block (SSD)
- Fast encrypted block (SSD)
- Slow block (mechanical)
- File (NFS)

Let's look at an example.

A StorageClass YAML

The following YAML object defines a StorageClass called **fast-local**
that provisions encrypted SSD volumes capable of 10 IOPs per
gigabyte from the Ireland AWS region.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast-local
provisioner: ebs.csi.aws.com          <<---- AWS Elastic Block Store CSI
plugin
parameters:
  encrypted: true                     <<---- Create encrypted volumes
  type: io1                           <<---- AWS SSD drives
  iopsPerGB: "10"                     <<---- Performance requirement
allowedTopologies:                    <<---- Where to provision volumes
and replicas
- matchLabelExpressions:
  - key: topology.ebs.csi.aws.com/zone
    values:
    - eu-west-1a                      <<---- Ireland AWS region
```

As with all Kubernetes YAML files, `kind` and `apiVersion` tell
Kubernetes the type and version of the object you're defining.
`metadata.name` is an arbitrary string that gives the object a friendly
name, and the `provisioner` field tells Kubernetes which CSI plugin
to use — you'll need the plugin installed. The `parameters` block
defines the type of storage to provision, and the `allowedTopologies`
property lets you specify where replicas should go.

A few important things to note:

1. Storage classes are immutable — once you deploy them, you can't modify them
2. `metadata.name` should be meaningful, as it's how **you** and other objects refer to the class
3. We sometimes use the terms *provisioner*, *plugin*, and *driver* interchangeably
4. The `parameters` block is for plugin-specific values and is different for every plugin

Most storage systems have their own features, and it's your responsibility to read the documentation for your plugin and configure it.

Working with StorageClasses

The basic workflow for deploying and using a StorageClass is as follows:

1. Install and configure the CSI plugin
2. Create one or more StorageClasses
3. Deploy Pods with PVCs that request volumes via the StorageClasses

The list assumes you have an external storage system connected to your Kubernetes cluster. Most hosted Kubernetes services pre-install CSI drivers for the cloud's native storage backends, making it easier to consume them.

The following YAML snippet defines a Pod, a PVC, and an SC. You can define all three objects in the same YAML file by separating them with three dashes (`---`).

```
apiVersion: v1
kind: Pod                         <<---- 1. Pod
metadata:
```

```
    name: mypod
spec:
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: mypvc           <<---- 2. Request volume via the
"mypvc" PVC
  containers: ...
  <SNIP>
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mypvc                      <<---- 3. This is the "mypvc" PVC
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 50Gi                <<---- 4. Provision a 50Gi
volume...
  storageClassName: fast           <<---- 5. ...based on the "fast"
StorageClass
---
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast                       <<---- 6. This is the "fast"
StorageClass
provisioner: pd.csi.storage.gke.io    <<---- 7. Use this CSI plugin
parameters:
  type: pd-ssd                     <<---- 8. Provision this type of
storage
```

The YAML is truncated and doesn't include a full PodSpec. However, we can see the main workflow if we step through the numbered annotations:

1. A normal Pod object
2. The Pod requests a volume via the **mypvc** PVC
3. The file defines a PVC called **mypvc**
4. The PVC provisions a 50Gi volume
5. The volume will be provisioned via the **fast** StorageClass
6. The file defines the **fast** StorageClass
7. The StorageClass provisions volumes via the **pd.csi.storage.gke.io** CSI plugin

8. The CSI plugin will provision fast (**pd-ssd**) storage from the Google Cloud's storage backend

Let's look at a couple of additional settings before moving on to the demos.

Additional volume settings

StorageClasses give you lots of ways to control how volumes are provisioned and managed. We'll cover the following:

- Access mode
- Reclaim policy

Access mode

Kubernetes supports three volume access modes:

- `ReadWriteOnce` (RWO)
- `ReadWriteMany` (RWM)
- `ReadOnlyMany` (ROM)

`ReadWriteOnce` lets a single PVC bind to a volume in read-write (R/W) mode. Attempts to bind it from multiple PVCs will fail.

`ReadWriteMany` lets multiple PVCs bind to a volume in read-write (R/W) mode. *File* and *object* storage usually support this mode, whereas *block* storage usually doesn't.

`ReadOnlyMany` allows multiple PVCs to bind to a volume in read-only (R/O) mode.

It's also important to know that a PV can only be opened in one mode. For example, you cannot bind a single PV to one PVC in ROM mode and another PVC in RWM mode.

Reclaim policy

`ReclaimPolicies` tell Kubernetes what to do with a PV and associated external storage when its PVC is released. Two policies currently exist:

- Delete
- Retain

`Delete` is the most dangerous and is the default for PVs created dynamically via StorageClasses. It deletes the PV **and associated external storage** when the PVC is released. This means deleting the PVC will delete the PV and the external storage. Use with caution.

`Retain` will keep the PV and external storage after you delete the PVC. It's the safer option, but you have to reclaim resources manually.

Before doing the demos, let's summarize what you've learned about StorageClasses.

StorageClasses (SC) represent tiers of storage that applications call on to create volumes dynamically. You define them in regular YAML files that reference a plugin and tie them to a particular type of storage on a particular external storage system. For example, one SC might provision *high-performance AWS SSD storage in the AWS Mumbai Region*, while another might provision *slow AWS storage from a different AWS region.* Once deployed, the SC controller watches the API server for new PVCs referencing the SC. Each time you create a PVC that matches the SC, the SC dynamically creates the required volume on the external storage system and maps it to a PV that apps can mount and use.

There's always more detail, but you've learned enough to get you started.

Hands-on

This section walks you through dynamically provisioning volumes via StorageClasses. I've split the demos as follows:

- Use an existing StorageClass
- Create and use a new StorageClass

The demos are all based on **Linode Kubernetes Engine (LKE)** and won't work on other clouds. This is because every cloud and every storage provider has its own CSI plugins with their own configuration options and I don't have space in the book for them all. However, it's easy to build an LKE cluster to follow along, and even if you only read along, you'll still learn a lot.

The rest of this section assumes you have a clone of the book's GitHub repo and are working on the 2025 branch. It also assumes you're connected to your LKE cluster. See Chapter 3 if you need to build one.

```
$ git clone https://github.com/nigelpoulton/TKB.git
<Snip>

$ cd TKB

$ git fetch origin

$ git checkout -b 2025 origin/2025

$ cd storage
```

Use an existing StorageClass

Run the following command to see the pre-installed storage classes on your cluster. Most Kubernetes environments pre-create at least one storage class, and I've trimmed the output to fit the page.

```
$ kubectl get sc
                                                        RECLAIM
VOLUME          ALLOWVOLUME
```

```
NAME                         PROVISIONER            POLICY
BINDINGMODE    EXPANSION
linode-blck-stg              linodebs.csi.linode.com   Delete
Immediate      true
linode-blck-stg-retain (def) linodebs.csi.linode.com   Retain
Immediate      true
```

Let's examine the output.

My LKE cluster has two pre-created storage classes that both provision volumes via the `linodebs.csi.linode.com` CSI plugin (`PROVISIONER`) and both use the `Immediate` volume binding mode.

One uses the `Delete` reclaim policy and the other uses the `Retain`.

The `linode-block-storage-retain (default)` class on the second line is the default class, meaning your PVCs will use this class unless you specify a different one. You should probably specify a storage class for your important production apps, as the default class can differ between clusters, meaning you don't always get the same thing.

Some clusters pre-create lots of storage classes, and the following output is from an Autopilot regional cluster on Google Kubernetes Engine. It shows eight classes, five of which use the `filestore.csi.storage.gke.io` plugin to access Google Cloud's NFS-based Filestore storage, two use the `pd.csi.storage.gke.io` plugin to access the Google Cloud's block storage, and one uses the legacy in-tree `kubernetes.io/gce-pd` plugin (non-CSI).

```
                                              RECLAIM
NAME                 PROVISIONER              POLICY
VOLUMEBINDINGMODE
enterprise-multi..   filestore.csi.storage.gke.io   Delete
WaitForFirstConsumer
enterprise-rwx       filestore.csi.storage.gke.io   Delete
WaitForFirstConsumer
premium-rwo          pd.csi.storage.gke.io          Delete
WaitForFirstConsumer
premium-rwx          filestore.csi.storage.gke.io   Delete
WaitForFirstConsumer
standard             kubernetes.io/gce-pd           Delete    Immediate
```

```
standard-rwo (def)    pd.csi.storage.gke.io            Delete
WaitForFirstConsumer
standard-rwx          filestore.csi.storage.gke.io    Delete
WaitForFirstConsumer
zonal-rwx             filestore.csi.storage.gke.io    Delete
WaitForFirstConsumer
```

Run the following command to see detailed information about the
linode-block-storage class.

```
$ kubectl describe sc linode-block-storage
Name:                 linode-block-storage
IsDefaultClass:       No
Annotations:          lke.linode.com/caplke-version=v1.31.5-2025-02b
Provisioner:          linodebs.csi.linode.com
Parameters:           <none>
AllowVolumeExpansion: True
MountOptions:         <none>
ReclaimPolicy:        Delete
VolumeBindingMode:    Immediate
Events:               <none>
```

The important thing for this workflow is that it uses the `Delete`
reclaim policy. This means Kubernetes will automatically delete the
PV and associated back-end volume when you stop using the PVC.
You'll see this in action shortly.

List any existing PVs and PVCs so that you can easily identify the
ones you're about to create.

```
$ kubectl get pv
No resources found
$ kubectl get pvc
No resources found in default namespace.
```

The following YAML is from the `lke-pvc-test.yml` file in the `storage`
folder. It describes a PVC called **pvc-test** that provisions a 10GB
volume via the **linode-block-storage** storage class.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-test
spec:
```

```
  accessModes:
  - ReadWriteOnce
  storageClassName: linode-block-storage
  resources:
    requests:
      storage: 10Gi
```

Run the following command to create the PVC. Be sure to run it from the **storage** folder, and it will only work if you have an LKE cluster with a **linode-block-storage** storage class.

```
$ kubectl apply -f lke-pvc-test.yml
persistentvolumeclaim/pvc-test created
```

Run the following command to see the PVC.

```
$ kubectl get pvc
NAME        STATUS    VOLUME                CAPACITY    ACCESS MODES
STORAGECLASS
pvc-test    Bound     pvc-cc3dd8716c7d46c9  10Gi        RWO
linode-block-storage
```

Kubernetes has created the PVC and already bound it to a volume. This is because the **linode-block-storage** SC uses the `Immediate` volume binding mode, to automatically create the PV and backend storage without waiting for a Pod to claim it.

Run the following command to see the PV.

```
$ kubectl get pv
                              ACCESS    RECLAIM
NAME                 CAPACITY MODES     POLICY    STATUS   CLAIM
STORAGECLASS
pvc-cc3dd8716c7d46c9 10Gi     RWO       Delete    Bound    pvc-test
linode-block-storage
```

The PV also exists and is bound to the **pvc-test** claim. This means the volume should also exist on the Linode cloud.

Open your LKE dashboard (`cloud.linode.com`) and navigate to the **Volumes** tab to confirm you have a 10GB volume with the same name as your PVC in the same region as your LKE cluster.

Congratulations. You've successfully provisioned an external volume via one of your cluster's built-in SCs. Figure 11.3 shows how everything maps. The only thing missing is a Pod on the right that uses the PVC to gain access to the PV. You'll see this in the upcoming section.
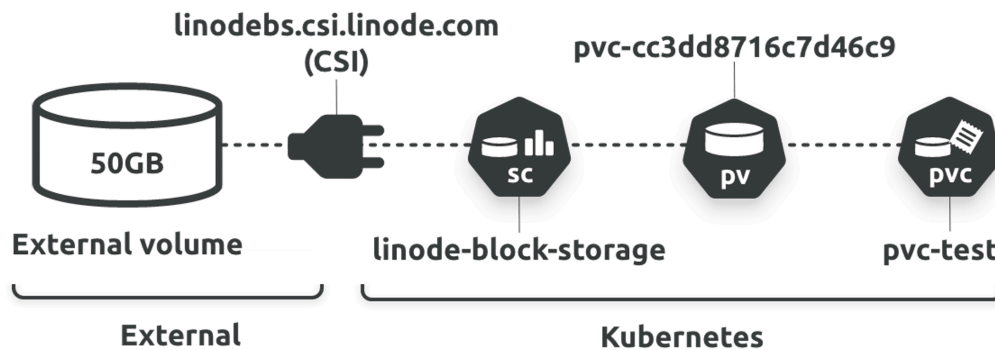
Figure 11.3 - How everything fits together

If you had configured the storage class with the `VolumeBindingMode` set to `WaitForFirstConsumer`, the storage class wouldn't create the PV or back-end volume until you deployed a Pod that used them.

Run the following command to delete the PVC.

```
$ kubectl delete pvc pvc-test
persistentvolumeclaim "pvc-test" deleted
```

Deleting the PVC will also delete the PV and associated volume on your LKE back end. This is because the storage class created them with the `ReclaimPolicy` set to `Delete`. Complete the following steps to verify this.

```
$ kubectl get pv
No resources found
```

Go to the **Volumes** tab of your LKE console and verify the back-end volume is gone.

Create and use a new StorageClass

In this section, you'll create a new StorageClass that implements *topology-aware provisioning.* This is where the storage class delays volume creation until a Pod requests it. This ensures the storage class will create the volume in the same region and zone as the Pod.

You'll create the storage class defined in the `lke-sc-wait-keep.yml` file in the `storage` folder of the book's GitHub repo. It defines a storage class called **block-wait-keep** with the following properties:

- Block storage
- Topology aware provisioning (`volumeBindingMode: WaitForFirstConsumer`)
- Keep volume and data when the PVC is deleted (`reclaimPolicy: Retain`)

You'll sometimes see *topology-aware provisioning* referred to as *provision on demand.*

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: block-wait-keep
provisioner: linodebs.csi.linode.com          <<---- CSI Plugin
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer       <<---- Provision on demand
reclaimPolicy: Retain                         <<---- Keep volume when PVC
released
```

Deploy the SC and verify it exists.

```
$ kubectl apply -f lke-sc-wait-keep.yml
storageclass.storage.k8s.io/block-wait-keep created

$ kubectl get sc

ALLOW
                                                      RECLAIM   VOLUME
```

```
VOLUME
NAME                       PROVISIONER             POLICY
BINDINGMODE          EXPANSION
block-wait-keep            linodebs.csi.linode.com   Retain
WaitForFirstConsumer  true
linode-blck-stg            linodebs.csi.linode.com   Delete
Immediate             true
linode-blck-stg-retain (def)  linodebs.csi.linode.com   Retain
Immediate             true
```

Once you've created the storage class, you can deploy the PVC defined in the `lke-pvc-wait-keep.yml` file. As you can see, it defines a PVC called **pvc-wait-keep** requesting a 20GB volume from the **block-wait-keep** SC.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-wait-keep
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
  storageClassName: block-wait-keep
```

Deploy it with the following command.

```
$ kubectl apply -f lke-pvc-wait-keep.yml
persistentvolumeclaim/pvc-wait-keep created
```

Confirm that Kubernetes created it, and check if it created the PV yet.

```
$ kubectl get pvc
NAME            STATUS    VOLUME    CAPACITY   ACCESS MODES
STORAGECLASS
pvc-wait-keep   Pending                                    block-
wait-keep

$ kubectl get pv
No resources found
```

The PVC is in the `Pending` state and the storage class hasn't created a PV yet. This is because the PVC uses a storage class with the `WaitForFirstConsumer` volume binding mode, which implements *topology-aware provisioning* by waiting for a Pod to reference the PVC before creating the PV and back-end volume on LKE.

The following YAML defines a Pod called **volpod** that uses the **pvc-wait-keep** PVC you just created. It's from the `lke-app.yml` file. I've annotated the sections referencing the PVC and mounting the volume.

```
apiVersion: v1
kind: Pod
metadata:
  name: volpod
spec:
  volumes:                        ----┐ Create a volume
  - name: data                        | called "data"
    persistentVolumeClaim:            | from the PVC
      claimName: pvc-wait-keep   ----┘ called "pvc-wait-keep
  containers:
  - name: ubuntu-ctr
    image: ubuntu:latest
    command:
    - /bin/bash
    - "-c"
    - "sleep 60m"
    volumeMounts:                   ----┐ Mount the
    - name: data                        | "data" volume
      mountPath: /tkb             ----┘ to /tkb
```

Deploy the Pod and then confirm the storage class created the PV.

```
$ kubectl apply -f lke-app.yml
pod/volpod created

$ kubectl get pvc
NAME            STATUS   VOLUME                   CAPACITY   ACCESS MODES
STORAGECLASS
pvc-wait-keep   Bound    pvc-279f09e083254fa9   20Gi        RWO
block-wait-keep

$ kubectl get pv
                                 ACCESS   RECLAIM
NAME                   CAPACITY  MODES    POLICY    STATUS   CLAIM
STORAGECLASS
```

```
pvc-279f09e083254fa9   20Gi        RWO      Retain   Bound   pvc-wait-
keep   block-wait-keep
```

The PVC is **bound** to a volume, a PV exists with the same name, and all the settings are as expected — capacity, reclaim policy, and storage class.

Run the following command to confirm the Pod is claiming and mounting the volume.

```
$ kubectl describe pod volpod
Name:           volpod
Namespace:      default
Node:           lke340882-541526-0198b5d80000/192.168.145.142
Status:         Running
IP:             10.2.1.3
<Snip>
Containers:
  ubuntu-ctr:
<Snip>
    Mounts:                                              ----┐
Mount the
      /tkb from data (rw)                                ----┘
"data" volume
<Snip>
Volumes:                                                 ----┐
  data:                                                      |
Create the "data" volume
    Type:       PersistentVolumeClaim (a reference to a PVC...)   |
from the PVC
    ClaimName:  pvc-wait-keep                                 |
called "pvc-wait-keep"
    ReadOnly:   false                                    ----┘
  <Snip>
```

Let's summarize what just happened:

1. You created a new StorageClass called **block-wait-keep** that provisions block storage from the Linode cloud
2. The StorageClass controller started watching the API server for new PVCs referencing your new **block-wait-keep** class
3. You created a PVC referencing the class, but it didn't create a volume because the volume binding mode is set to **WaitForFirstConsumer**

4. You deployed a Pod that used the PVC to request a new 20GB volume
5. The SC controller observed this and dynamically created PV and the external volume on the Linode cloud

Congratulations. You've created your own StorageClass that implements *topology-aware provisioning* by delaying PV creation until a Pod references the PVC. This ensures Kubernetes creates volumes in the same region and zone as the Pods.

Clean up

You need to do five things to clean up your environment:

1. Delete the Pod using the PVC
2. Delete the PVC
3. Delete the PV
4. Manually delete the volume on the Linode cloud back end
5. Delete the SC

Delete the Pod. It may take a few seconds for the operation to complete.

```
$ kubectl delete pod volpod
pod "volpod" deleted
```

Delete the PVC.

```
$ kubectl delete pvc pvc-wait-keep
persistentvolumeclaim "pvc-wait-keep" deleted
```

Even though you've deleted the Pod and the PVC, Kubernetes hasn't deleted the PV or external volume because the storage class created them with the `Retain` reclaim policy. This means you'll have to delete them manually.

Run the following command to delete the PV. Yours will have a
different name.

```
$ kubectl delete pv pvc-279f09e083254fa9
persistentvolume "pvc-279f09e083254fa9" deleted
```

Open your Linode cloud console and delete the volume from the
Volumes tab. Be sure to delete the volume with the same name as
the PV you just deleted! It will show as *unattached* in the **Attached
to** column. Not deleting the volume will incur unwanted Linode
costs.

Finally, delete the storage class.

```
$ kubectl delete sc block-wait-keep
storageclass.storage.k8s.io "block-wait-keep" deleted
```

Chapter Summary

In this chapter, you learned that Kubernetes has a powerful storage
subsystem that enables applications to dynamically provision and
use storage from various external providers.

Each external provider has its own CSI plugin that creates the
volumes and surfaces them inside Kubernetes. Most hosted
Kubernetes clusters pre-install CSI plugins that run as Pods in the
`kube-system` Namespace.

Once you've installed the CSI plugin, you create StorageClasses that
map to a type of storage on the external system. The StorageClass
controller operates as a background reconciliation loop on the
control plane, watching the API server for new PVCs. Whenever it
sees one, it creates the requested volume on the external system
and maps it to a new PV on Kubernetes. Pods can then use the PVC
to claim and mount the volume.

12: ConfigMaps and Secrets

Most business applications have two components:

- The application
- The configuration

Simple examples include web servers such as NGINX and httpd (Apache). Neither is very useful until you add a configuration.

In the past, we packaged applications and their configurations as a single easy-to-deploy unit, and we brought this pattern with us in the early days of cloud-native microservices. However, it's an *anti-pattern*, and you should decouple modern applications from their configurations, as it brings the following benefits:

- Reuse
- Simpler development and testing
- Simpler and less-disruptive changes

We'll explain all these and more as we go through the chapter.

> **Note:** An *anti-pattern* is something that seems like a good idea but turns out to be a bad idea.

I've divided the chapter as follows:

- The big picture
- ConfigMap theory
- Hands-on with ConfigMaps
- Hands-on with Secrets

Everything you'll learn about ConfigMaps applies to Secrets later in the chapter.

The big picture

As previously mentioned, most applications comprise an application binary and a configuration. Kubernetes lets you build and store them as separate objects and bring them together at run time.

Consider a quick example.

Imagine you work for a company with three environments:

- Dev
- Test
- Prod

You perform initial testing in the **dev** environment, more extensive testing in the **test** environment, and apps finally graduate to the **prod** environment. However, each environment has its own network and security policies, as well as its own unique credentials and certificates.

You currently package applications and their configurations together in the same image, forcing you to perform <u>all</u> of the following for <u>every</u> application:

- **Build** three images (one with the dev config, one with the test config, and one with prod)
- **Store** the images in three distinct repositories (one for the dev image, one for test, and one for prod)
- **Run** different configurations of each app in each of the three environments (the dev app in the dev environment, test in test, prod in prod)

Every time you change the app, even a small change like fixing a typo, you have to build, test, store, and re-deploy three times — once for dev, once for test, and once for prod.

It's also harder to troubleshoot and isolate issues when every update includes the app code and the config.

What it looks like in a decoupled world

Imagine you work for the same company, and they ask you to build a new web app. However, the company now decouples code and configurations.

You decide to base the new app on NGINX and create a hardened NGINX base image that other teams and applications can reuse with their own configurations. This means:

- You only **build** a single image that you'll use across all three environments
- You only **store** and protect that single image in a single repository
- You **run** the same version of this image in all your environments

To make this work, you build a single base image containing nothing more than the hardened NGINX with no embedded configuration.

You then create three *configurations* for **dev**, **test**, and **prod** that you'll apply at run time. Each one configures the hardened NGINX container with the app configuration, policy settings, and credentials for the correct environment. Other teams and applications can *reuse* the same hardened NGINX image for their own web apps by applying their own configurations.

In this model, you create and test a single version of NGINX, build it into a single image, and store it in a single repository. You can grant all developers access to pull the repository as it contains no sensitive data, and you can push changes to either the application or its configuration independently of each other. For example, if

there's a typo on the homepage, you can fix it in the configuration and push that to existing containers in all three environments. You no longer have to stop and replace every container in all three environments.

Let's see how Kubernetes makes this possible.

ConfigMap theory

Kubernetes has an API resource called a ConfigMap (CM) that lets you store configuration data outside of Pods and inject it at run time.

ConfigMaps are first-class objects in the *core* API group. They're also `v1`. This tells us a few things:

1. They're stable (`v1`)
2. They've been around for a while (new stuff never goes in the core API group)
3. You can define and deploy them in YAML files
4. You can manage them with `kubectl`

You'll typically use ConfigMaps to store non-sensitive configuration data such as:

- Environment variables
- Configuration files like web server configs and database configs
- Hostnames
- Service name and Service ports
- Account names

You should **not** use ConfigMaps to store sensitive data such as certificates and passwords, as Kubernetes makes no effort to protect their contents. For sensitive data, you should use Kubernetes Secrets as part of a comprehensive secrets management solution.

You'll work with Secrets later in the chapter.

How ConfigMaps work

At a high level, a ConfigMap is an object for storing configuration data that you can easily inject into containers at run time. You can configure them to work with environment variables and volumes so that they work seamlessly with applications.

Let's look a bit closer.

Behind the scenes, ConfigMaps are Kubernetes objects that hold a map of key-value pairs:

- **Keys** are an arbitrary name that can include alphanumerics, dashes, dots, and underscores
- **Values** can store anything, including full configuration files with multiple lines and carriage returns
- They're limited to 1MiB (1,048,576 bytes) in size

Here's an example of a ConfigMap with three entries.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: epl
data:
  Competition: Premier League    -----¬
  Season: 2024-2025                     | 3 x map entries (key:value
pairs)
  Champions: Liverpool           -----˩
```

Here's an example where the *value* is a complete configuration file.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: cm2
data:
  test.conf: |                   <<---- Key
    env = plex-test              -----¬
```

```
endpoint = 0.0.0.0:31001        |
char = utf8                     | Value
vault = PLEX/test               |
log-size = 512M           -----┘
```

Once you've created a ConfigMap, you can use any of the following methods to inject it into containers at run time:

1. Environment variables
2. Arguments to the container's startup command
3. Files in a volume

Figure 12.1 shows how the pieces connect. The ConfigMap exists outside the app, and its values are mapped to standard constructs such as environment variables and volumes that apps can easily access.
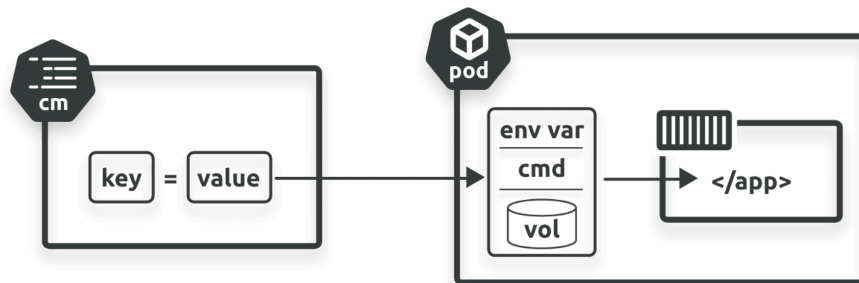
Figure 12.1

All three methods work without having to add Kubernetes knowledge to existing applications. However, the volume option is the most flexible as it enables updates. You'll try them all, but first, let's quickly mention *Kubernetes-native* applications.

ConfigMaps and Kubernetes-native apps

Kubernetes-native applications know they're running on Kubernetes and can talk to the Kubernetes API server. This has a lot of benefits, including the ability to read ConfigMaps directly through the API without having to mount them as volumes or via environment

variables. It also means live containers can see updates to the ConfigMap. However, apps like these can only run on Kubernetes and create *Kubernetes lock-in* where your apps only work on Kubernetes.

All of our examples use environment variables and volumes so that existing apps can access ConfigMap data without needing rewrites or being locked to Kubernetes.

Hands-on with ConfigMaps

You'll need a Kubernetes cluster and the lab files from the book's GitHub repo if you want to follow along.

```
$ git clone https://github.com/nigelpoulton/TKB.git
Cloning into 'TKB'...

$ cd TKB

$ git fetch origin

$ git checkout -b 2025 origin/2025

$ cd configmaps
```

Be sure you're on the `2025` branch, and run all commands from the `configmaps` folder.

As with most Kubernetes resources, you can create ConfigMaps imperatively and declaratively. We'll look at the imperative method first.

Creating ConfigMaps imperatively

You create ConfigMaps imperatively with the `kubectl create configmap` command. However, you can shorten `configmap` to `cm,` and the command accepts two sources of data:

- Literal values on the command line (`--from-literal`)
- From a file (`--from-file`)

Run the following command to create a ConfigMap called **testmap1** and populate it with two entries from command-line literals. Windows users should replace the backslashes with backticks at the end of the first two lines.

```
$ kubectl create configmap testmap1 \
  --from-literal shortname=SAFC \
  --from-literal longname="Sunderland Association Football Club"
```

Run the following command to see how Kubernetes stores map entries.

```
$ kubectl describe cm testmap1
Name:          testmap1
Namespace:     default
Labels:        <none>
Annotations:   <none>
Data
====
shortname:
----
SAFC
longname:
----
Sunderland Association Football Club
BinaryData
====
Events:   <none>
```

You can see it's just a map of key-value pairs dressed up as a Kubernetes object.

The following command uses the `--from-file` flag to create a ConfigMap called **testmap2** from a file called `cmfile.txt`. The file contains a single line of text, and you'll need to run the command from the `configmaps` folder.

```
$ kubectl create cm testmap2 --from-file cmfile.txt
configmap/testmap2 created
```

You'll inspect this one in the next section.

Inspecting ConfigMaps

ConfigMaps are first-class API objects, meaning you can inspect and query them like any other API object.

List all ConfigMaps in your current Namespace.

```
$ kubectl get cm
AME        DATA    AGE
testmap1   2       11m
testmap2   1       2m23s
```

The following `kubectl describe` command shows some interesting info about the **testmap2** map that you created from the local file:

- The operation created a single map entry
- The name of the *key* matches the name of the input file (`cmfile.txt`)
- The *value* stores the contents of the file

```
$ kubectl describe cm testmap2
Name:         testmap2
Namespace:    default
Labels:       <none>
Annotations:  <none>
Data
====
cmfile.txt:                    <<---- key
----
Kubernetes FTW!                <<---- value
BinaryData
====
Events:  <none>
```

You can also run a `kubectl get` command with the `-o yaml` flag to see the entire object.

```
$ kubectl get cm testmap2 -o yaml
apiVersion: v1
data:
```

```
  cmfile.txt: |
    Kubernetes FTW!
kind: ConfigMap
metadata:
  creationTimestamp: "2025-02-04T14:19:21Z"
  name: testmap2
  namespace: default
  resourceVersion: "18128"
  uid: 146da79c-aa09-4b10-8992-4dffa087dbfb
```

If you look closely, you'll notice the `spec` and `status` subresources are missing. This is because ConfigMaps don't have the concept of *desired state* and *observed state.* They have a `data` subresource instead.

Let's see how to create ConfigMaps declaratively.

Creating ConfigMaps declaratively

The following YAML is from the `fullname.yml` file in the book's GitHub repo and defines two map entries: **firstname** and **lastname**. It creates a ConfigMap called **multimap** with the usual `kind, apiVersion` and `metadata` fields. However, it has a `data` subresource instead of the usual `spec`.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: multimap
data:
  firstname: Nigel
  lastname: Poulton
```

Deploy it with the following command.

```
$ kubectl apply -f fullname.yml
configmap/multimap created
```

This next YAML object is from the `singlemap.yml` file and looks more complex than the previous one. However, it's actually simpler, as it only has a single entry in the `data` block. It looks more

complicated because the *value* entry contains an entire configuration file.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: test-config
data:
  test.conf: |               <<---- Key
    env = plex-test        -----┐
    endpoint = 0.0.0.0:31001   |
    char = utf8                | Value
    vault = PLEX/test          |
    log-size = 512M        -----┘
```

If you look closely, you'll see the pipe character (ı) after the name of the key property. This tells Kubernetes to treat everything after the pipe as a single value. If you deploy it, you'll get the ConfigMap shown in the following table called **test-config** with a single complex map entry:

Object name	Key	Value
test-config	test.conf	env = plex-test
		endpoint = 0.0.0.0:31001
		char = utf8
		vault = PLEX/test
		log-size = 512M

Deploy it with the following command.

```
$ kubectl apply -f singlemap.yml
configmap/test-config created
```

Run the following command to inspect it.

```
$ kubectl describe cm test-config
Name:         test-config
Namespace:    default
Labels:       <none>
Annotations:  <none>
Data
====
```

```
test.conf:
----
env = plex-test
endpoint = 0.0.0.0:31001
char = utf8
vault = PLEX/test
log-size = 512M
BinaryData
====
Events:  <none>
```

Injecting ConfigMap data into Pods and containers

There are three ways to inject ConfigMap data into containers:

- As environment variables
- As arguments to container startup commands
- As files in a volume

The first two inject data when you <u>create containers</u> and have no way of updating the values in a running container. The volumes option also injects data at creation time, but automatically pushes updates to live containers.

Let's look at both.

ConfigMaps and environment variables

Figure 12.2 shows the process of using environment variable to inject ConfigMap data into containers. You create the ConfigMap. You then map its entries into environment variables in the `containers` section of the Pod template. Finally, when Kubernetes starts the container, the environment variables appear as standard Linux or Windows environment variables, meaning apps can consume them without knowing about the ConfigMap.
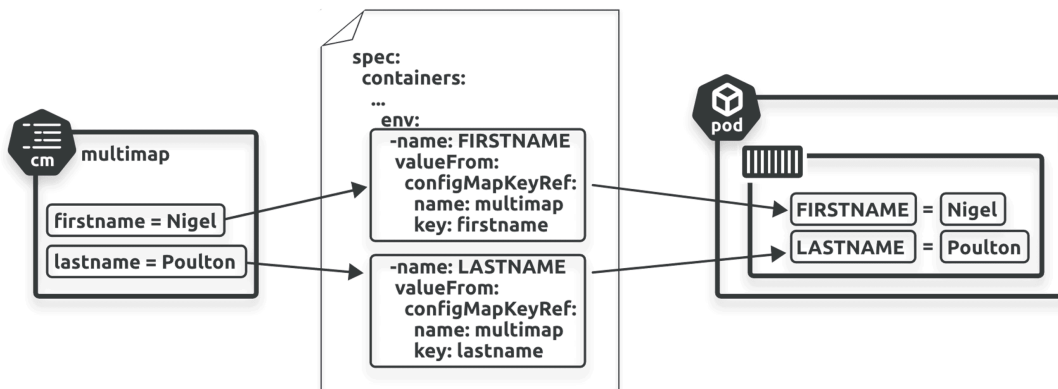
Figure 12.2

You've already deployed a ConfigMap called **multimap** with the following two entries:

- firstname=Nigel
- lastname=Poulton

The following Pod manifest deploys a single container with two environment variables mapped to the values in the ConfigMap. It's from the `podenv.yml` file you're about to deploy.

- **FIRSTNAME**: Maps to the **firstname** entry in the CM
- **LASTNAME**: Maps to the **lastname** entry in the CM

```
apiVersion: v1
kind: Pod
<Snip>
spec:
  containers:
    - name: ctr1
      env:
        - name: FIRSTNAME        <<---- Environment variable called
FIRSTNAME
          valueFrom:             <<---- based on
            configMapKeyRef:     <<---- a ConfigMap
              name: multimap     <<---- called "multimap"
              key: firstname     <<---- and populated by the value in
the "firstname" field
        - name: LASTNAME         <<---- Environment variable called
LASTNAME
          valueFrom:             <<---- based on
```

```
          configMapKeyRef:    <<---- a ConfigMap
            name: multimap    <<---- called "multimap"
            key: lastname     <<---- and populated by the value in
the "lastname" field
<Snip>
```

I've given the environment variable names capital letters so you can distinguish them from their ConfigMap counterparts. In reality, you can name them anything you like.

When Kubernetes schedules the Pod, and the container starts, it creates **FIRSTNAME** and **LASTNAME** as standard Linux environment variables so that applications can use them without knowing anything about ConfigMaps.

Run the following command to deploy a Pod from the `podenv.yml`.

```
$ kubectl apply -f podenv.yml
pod/envpod created
```

Run the following exec command to list environment variables in the container with the **"NAME"** string in their name. You'll see the `FIRSTNAME` and `LASTNAME` variables with their values from the ConfigMap.

Make sure the Pod is running before executing the command. Windows users need to replace the `grep NAME` argument with `Select-String -Pattern 'NAME'`.

```
$ kubectl exec envpod -- env | grep NAME
HOSTNAME=envpod
FIRSTNAME=Nigel
LASTNAME=Poulton
```

Remember though, environment variables are static. This means you can update the values in the map, but the **envpod** won't get the updates.

ConfigMaps and container startup commands

The concept of using ConfigMaps with container startup commands is simple. You specify the container's startup command in the Pod template and pass in environment variables as arguments.

The following example is from the `podstartup.yml` file. It describes a single container called **args1** based on the **busybox** image. It then defines and populates two environment variables from the **multimap** ConfigMap and **references them in the container's startup command.**

The main difference with the previous configuration is the `spec.containers.command` line that references the environment variables.

```
spec:
  containers:
    - name: args1
      image: busybox
      env:                           ----┐
        - name: FIRSTNAME               |
          valueFrom:                    |
            configMapKeyRef:            |
              name: multimap           | Same environment variable
mappings
              key: firstname           | as previous example. But this
time
        - name: LASTNAME               | used by the startup command
below
          valueFrom:                   |
            configMapKeyRef:            |
              name: multimap           |
              key: lastname     ----┘
      command: [ "/bin/sh", "-c", "echo First name $(FIRSTNAME) last
name $(LASTNAME)" ]
```

Figure 12.3 summarizes how the ConfigMap entries get populated to the environment variables and then referenced in the startup command.
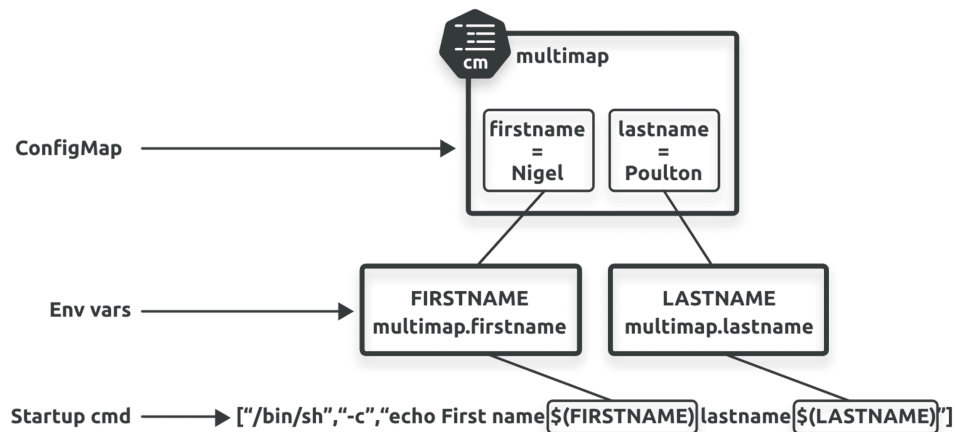
Figure 12.3 - Mapping ConfigMap entries to startup commands

Start a new Pod from the `podstartup.yml` file. The Pod will start, print **First name Nigel last name Poulton** to the container's logs, and then quit (succeed). It might take a few seconds for the Pod to start and execute.

```
$ kubectl apply -f podstartup.yml
pod/startup-pod created
```

Run the following command to inspect the container's logs and verify it printed **First name Nigel last name Poulton**.

```
$ kubectl logs startup-pod -c args1
First name Nigel last name Poulton
```

Describing the Pod will show the following data about the environment variables.

```
$ kubectl describe pod startup-pod
<Snip>
Environment:
  FIRSTNAME:  <set to the key 'firstname' of config map 'multimap'>
  LASTNAME:  <set to the key 'lastname' of config map 'multimap'>
<Snip>
```

As you've seen, using ConfigMaps with container startup commands still uses environment variables. As such, it suffers from the same

limitation — updates to the ConfigMap don't get pushed to existing variables.

If you ran the **startup-pod**, it should be in the completed state. This is because it completed its task and then terminated. Delete it.

```
$ kubectl delete pod startup-pod
pod "startup-pod" deleted
```

ConfigMaps and volumes

Using ConfigMaps with volumes is the most flexible option. They let you reference entire configuration files and they get updates. However, it can take a minute or two for the updates to appear in your containers.

The high-level process of injecting ConfigMap data into containers via volumes is as follows:

1. Create the ConfigMap
2. Define a *ConfigMap volume* in the Pod template
3. Mount the *ConfigMap volume* into the container
4. ConfigMap entries will appear as files inside the container

Figure 12.4 shows the process.
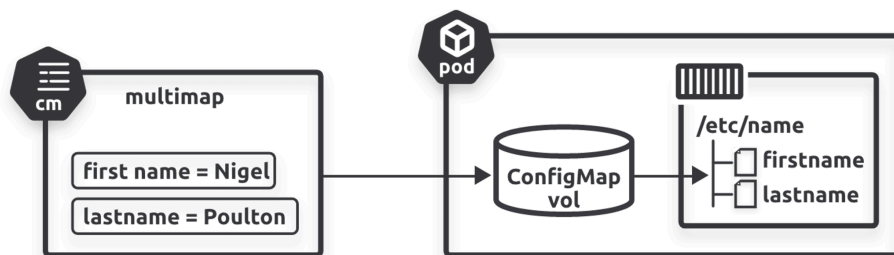
Figure 12.4 - Mapping ConfigMap entries through a volume

You've already deployed the **multimap** ConfigMap, and it has the following values:

- firstname=Nigel
- lastname=Poulton

The following YAML is from the `podvol.yml` file and defines a Pod called **cmvol** with the following configuration:

- `spec.volumes` creates a volume called **volmap** based on the **multimap** ConfigMap
- `spec.containers.volumeMounts` mounts the **volmap** volume to `/etc/name` in the container

```
apiVersion: v1
kind: Pod
metadata:
  name: cmvol
spec:
  volumes:
    - name: volmap                <<---- Create a volume called
"volmap"
      configMap:                  <<---- based on the ConfigMap
        name: multimap            <<---- called "multimap"
  containers:
    - name: ctr
      image: nginx
      volumeMounts:               <<---- These lines mount the
        - name: volmap            <<---- "volmap" volume into the
          mountPath: /etc/name    <<---- container at "/etc/name"
```

Run the following command to deploy the **cmvol** Pod from the previous YAML.

```
$ kubectl apply -f podvol.yml
pod/cmvol created
```

Wait for the Pod to enter the running phase and then run the following command to list the files in the container's `/etc/name/` directory.

```
$ kubectl exec cmvol -- ls /etc/name
firstname
lastname
```

You can see the container has two files matching the ConfigMap entries. Feel free to run additional `kubectl exec` commands to `cat` the contents of the files and ensure they match the values in the ConfigMap.

Now, let's prove that changes you make to the map appear in your live **cmvol** container.

Use `kubectl edit` to edit the ConfigMap and change any value in the data block. The command will open the YAML object in your default editor, which is usually `vi` on Linux and Mac, and usually `notepad.exe` on Windows. If you're uncomfortable using `vi`, you can manually edit the YAML file in a different editor and use `kubectl apply` to re-post it to the API server.

I've annotated the code block to show which lines to change.

```
$ kubectl edit cm multimap

# Please edit the object below. Lines beginning with a '#' will be
ignored,
# and an empty file will abort the edit. If an error occurs while
saving
# this file will be reopened with the relevant failures.
#
apiVersion: v1
data:
  City: Macclesfield        <<---- changed
  Country: UK               <<---- changed
kind: ConfigMap
metadata:
<Snip>
```

Save your changes and check if the updates appear in the container. **It may take a minute or so for your changes to appear.**

```
$ kubectl exec cmvol -- ls /etc/name
City
Country

$ kubectl exec cmvol -- cat /etc/name/Country
UK
```

Congratulations, you've surfaced the contents of the **multimap** ConfigMap into the container's filesystem via a *ConfigMap volume*, and you've proved the volume received your updates.

Hands-on with Secrets

Secrets work the same as ConfigMaps — they hold configuration data that Kubernetes injects into containers at run time. However, Secrets are designed to work as part of a more comprehensive *secrets management solution* for storing sensitive data such as passwords, certificates, and OAuth tokens.

Are Kubernetes Secrets secure?

The quick answer to this question is *no*. But here's the slightly longer answer…

A secure secrets management system involves a lot more than just Kubernetes Secrets. You have to consider all of the following and more:

- Encryption of secrets while at rest in the cluster store
- Encryption of secrets while in flight on the network
- Protection of secrets when surfaced in nodes/Pods/containers
- Controlling API access to secrets via least privilege RBAC
- Controlling access to etcd nodes (cluster store)
- Preventing privileged containers from accessing secrets
- Preventing exposure via source code repositories like GitHub
- Securely deleting secrets when no longer in use

Most new Kubernetes clusters do none of those things — they store them unencrypted in the cluster store, send them over the network in plain text, and mount them in containers as plain text! However, you can configure `EncryptionConfiguration` objects to encrypt Secrets at rest in the cluster store, deploy a service mesh to encrypt

all network traffic, and configure strong RBAC to secure API access to Secrets. You can even restrict access to control plane nodes and etcd nodes, securely delete secrets after use, and more.

In the real world, many production clusters implement service meshes that secure network traffic, and they store secrets outside of Kubernetes in 3rd party vaults such as HashiCorp's Vault or similar cloud services. Kubernetes even has a native *Secrets Store CSI Driver* for integrating with external vaults.

As you can imagine, there are lots of vaults and secrets management systems, and they all have their own complex configurations. With this in mind, we'll focus on what you get out-of-the-box with most Kubernetes installations and leave the intricacies of specific platforms to you.

A typical secrets workflow that only uses Kubernetes Secrets looks like this:

1. You create the Secret which Kubernetes persists to the cluster store as an **un-encrypted** object
2. You schedule a Pod with a container that uses the Secret
3. Kubernetes transfers the **un-encrypted** Secret over the network to the node running the Pod
4. The kubelet on the node starts the Pod and its containers
5. The container runtime mounts the Secret into the container via an in-memory *tmpfs* filesystem and decodes it from base64 to **plain text**
6. The application consumes it
7. When you delete the Pod, Kubernetes deletes the copy of the Secret on the node (it keeps the copy in the cluster store)

An obvious use case for Secrets is a TLS termination proxy. Figure 12.5 shows a single image configured with three different Secrets for three different environments. Each of the Secrets could contain

the TLS certificates for the specific environment, and Kubernetes loads the appropriate Secret into each container at run time.
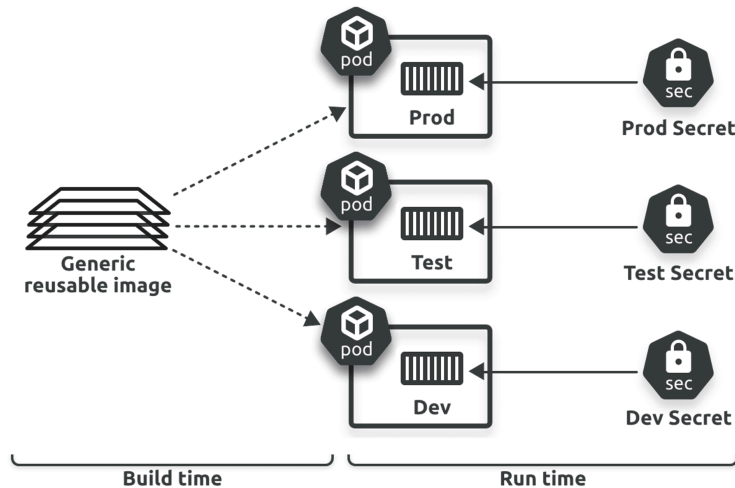
Figure 12.5 - Injecting Secrets at run time

Creating Secrets

Remember that most Kubernetes installations do nothing to encrypt Secrets in the cluster store or while in flight on the network. And even if you implement these, they're always surfaced as plain text in containers so that applications can easily consume them.

As with all API resources, you can create Secrets imperatively and declaratively.

Run the following command to create a new Secret called **creds**. Remember to replace the backslash with a backtick if you're on Windows.

```
$ kubectl create secret generic creds \
  --from-literal user=nigelpoulton \
  --from-literal pwd=Password123
```

You learned earlier that Kubernetes *obscures* Secrets by encoding them as base64 values. Check this with the following command.

```
$ kubectl get secret creds -o yaml
apiVersion: v1
kind: Secret
data:
  pwd: UGFzc3dvcmQxMjM=
  user: bmlnZWxwb3VsdG9u
<Snip>
```

The username and password values are both base64 encoded. Run the following command to decode them. You'll need the `base64` utility installed on your system for the command to work. If you don't have it, you can use an online decoder.

```
$ echo UGFzc3dvcmQxMjM= | base64 -d
Password123
```

The decoding completes successfully without a key, proving that base64 encoding is not secure. As such, you should never store Secrets on platforms like GitHub, as anyone with access can read them.

The following YAML object is from the `tkb-secret.yml` file in the `configmaps` folder. It describes a Secret called **tkb-secret** with two base64-encoded entries. You can add plain text entries by changing the `data` block to `stringData`, but you should never store either type on places like GitHub, as anyone with access can read them.

```
apiVersion: v1
kind: Secret
metadata:
  name: tkb-secret
  labels:
    chapter: configmaps
type: Opaque
data:                          <<---- Change to "stringData" for plain
text entries
  username: bmlnZWxwb3VsdG9u
  password: UGFzc3dvcmQxMjM=
```

Deploy it to your cluster. Be sure to run the command from the `configmaps` folder.

```
$ kubectl apply -f tkb-secret.yml
secret/tkb-secret created
```

Run `kubectl get` and `kubectl describe` commands to inspect it.

Using Secrets in Pods

In this section, you'll deploy a Pod that uses the **tkb-secret** you created in the previous section.

Secrets work like ConfigMaps, meaning you can inject them into containers as environment variables, command line arguments, or volumes. As with ConfigMaps, the most flexible option is a volume.

The following YAML is from the `secretpod.yml` file and describes a single-container Pod with a *Secret volume* called **secret-vol** based on the **tkb-secret** you created in the previous step. It mounts **secret-vol** into the container at `/etc/tkb`.

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-pod
  labels:
    topic: secrets
spec:
  volumes:
  - name: secret-vol            <<---- Volume name
    secret:                     <<---- Volume type
      secretName: tkb-secret    <<---- Populate volume with this
Secret
  containers:
  - name: secret-ctr
    image: nginx
    volumeMounts:
    - name: secret-vol          <<---- Mount the volume defined above
      mountPath: "/etc/tkb"     <<---- into this path
```

Secret volumes are resources in the Kubernetes API, and Kubernetes automatically mounts them as read-only to prevent containers and applications from accidentally mutating their contents.

Deploy the Pod with the following command. This causes Kubernetes to transfer the unencrypted Secret over the network to the kubelet on the node running the Pod. From there, the container runtime mounts it into the container using a *tmpfs* mount.

```
$ kubectl apply -f secretpod.yml
pod/secret-pod created
```

Run the following command to see the Secret mounted in the container as two files at `/etc/tkb` — one file for each entry in the Secret.

```
$ kubectl exec secret-pod -- ls /etc/tkb
password
username
```

If you inspect the contents of either file, you'll see they're mounted in plain text so that applications can easily consume them.

```
$ kubectl exec secret-pod -- cat /etc/tkb/password
Password123
```

Well done. You've created a Secret and mounted it into a Pod via a Secret volume. As you chose the volume option, you can update the Secret and the app will see it.

Remember, a complete secrets management solution involves a lot more than just storing sensitive data in Kubernetes Secrets. You need to encrypt them at rest, encrypt them in flight, control API access, restrict etcd node access, handle privileged containers, prevent Secret manifest files from being hosted on public source control repos, and more. Most real-world solutions store secret data outside of Kubernetes in a 3rd-party vault.

Clean up

Use `kubectl get` to list the Pods, ConfigMaps and Secrets you've deployed.

```
$ kubectl get pods
NAME            READY    STATUS       RESTARTS    AGE
cmvol           1/1      Running      0           27m
envpod          1/1      Running      0           22m
secret-pod      1/1      Running      0           16m

$ kubectl get cm
NAME               DATA    AGE
kube-root-ca.crt   1       71m    <<---- Don't delete this one
multimap           2       19m
test-config        1       17m
testmap1           2       24m
testmap2           1       22m

$ kubectl get secrets
NAME            TYPE        DATA    AGE
creds           Opaque      2       4m55s
tkb-secret      Opaque      2       3m35s
```

Now delete them. It can take a few seconds for the Pods to delete.

```
$ kubectl delete pods cmvol envpod secret-pod

$ kubectl delete cm multimap test-config testmap1 testmap2

$ kubectl delete secrets creds tkb-secret
```

Chapter Summary

ConfigMaps and Secrets are how you decouple applications from their configuration data.

Both are first-class objects in the Kubernetes API, meaning you can create them imperatively and declaratively and inspect them with **kubectl.**

ConfigMaps are designed for application configuration parameters and even entire configuration files, whereas Secrets are for sensitive data and intended for use as part of a wider secrets management solution.

You can inject both into containers at run time via environment variables, container start commands, and volumes. Volumes are the

preferred method as they propagate changes to live containers.

Kubernetes does not automatically encrypt Secrets in the cluster store or while transmitting them over the network.

13: StatefulSets

In this chapter, you'll learn how to use *StatefulSets* to deploy and manage stateful applications on Kubernetes.

For the purposes of this chapter, we're defining a *stateful application* as one that creates and saves valuable data. Examples include databases, key-value stores, and applications that save data about client sessions for use in future sessions.

I've arranged the chapter as follows:

- StatefulSet theory
- Hands-on with StatefulSets

The theory section introduces how StatefulSets work and why they're useful for stateful applications. But don't worry if you don't understand everything at first, you'll cover it all again in the hands-on section.

StatefulSet theory

It's helpful to compare StatefulSets with *Deployments.* Both are Kubernetes API resources and follow the standard Kubernetes controller architecture — control loops that reconcile *observed state* with *desired state.* Both manage Pods and add self-healing, scaling, rollouts, and more.

However, StatefulSets offer the following three features that Deployments do not:

1. Predictable and persistent Pod names and DNS names
2. Predictable and persistent volume bindings
3. Predictable startup and shutdown order

Points one and two form a Pod's *state*, and we sometimes refer to them as a Pod's *sticky ID*. StatefulSets even ensure Pod names and volume bindings persist across failures, scaling operations, and other scheduling events.

As a quick example, StatefulSet Pods that fail get replaced with new Pods with the same Pod name, the same DNS hostname, and connect to the same volumes. This is true even if Kubernetes starts the replacement Pod on a different cluster node. They'll even get the same name and volumes if they're terminated by a scale-down operation and then recreated by a scale-up operation. All of this makes them ideal for applications requiring unique reliable Pods.

The following YAML defines a simple StatefulSet called **tkb-sts** with three replicas running the `mongo:latest` image. You post this to the API server, it gets persisted to the cluster store, the scheduler assigns the replicas to worker nodes, and the StatefulSet controller ensures *observed state* matches *desired state*.

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: tkb-sts
spec:
  selector:
    matchLabels:
      app: mongo
  serviceName: "tkb-sts"
  replicas: 3
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
      - name: ctr-mongo
        image: mongo:latest
        ...
```

That's the big picture. Let's take a closer look before walking through an example.

StatefulSet Pod naming

Every Pod created by a StatefulSet gets a predictable name. In fact, Pod names are at the core of how StatefulSets start, self-heal, scale, delete Pods, and even how they connect Pods to volumes.

The format of StatefulSet Pod names is **<StatefulSetName>-<Integer>**. The integer is a *zero-based index ordinal*, which is a fancy way of saying *number starting from zero*. If we go with the previous YAML snippet, Kubernetes will name the first replica **tkb-sts-0**, the second **tkb-sts-1**, and the third **tkb-sts-2**.

Kubernetes also uses the StatefulSet name to create each replica's DNS name, so avoid using exotic characters that will create invalid DNS names.

Ordered creation and deletion

A critical difference between StatefulSets and Deployments is the way they create Pods.

- StatefulSets create one Pod at a time and wait for it to be running and ready before starting the next
- Deployments use a ReplicaSet controller to start all Pods at the same time, which can result in *race conditions*

Sticking with the previous YAML, the StatefulSet controller will start **tkb-sts-0** first and wait for it to be *running and ready* before starting **tkb-sts-1**. The same applies to subsequent Pods — the controller waits for **tkb-sts-1** to be *running and ready* before starting **tkb-sts-2** etc. See Figure 13.1
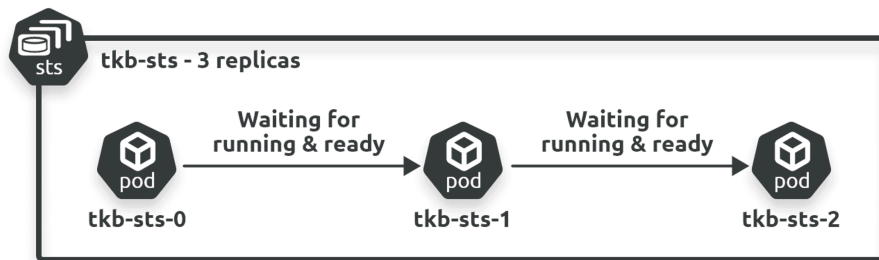
Figure 13.1

> **Note:** *Running and ready* is a term we use to indicate all containers in a Pod are *running* and the Pod is *ready* to service requests.

The same startup rules govern StatefulSet scaling operations. For example, scaling from 3 to 5 replicas will start a new Pod called **tkb-sts-3** and wait for it to be running and ready before creating **tkb-sts-4**. Scaling down follows the same rules in reverse — the controller terminates the Pod with the highest index ordinal and waits for it to fully terminate before terminating the Pod with the next highest number.

Guaranteeing the order in which Pods will be scaled down, and knowing that Kubernetes will never terminate them in parallel can be vital for stateful apps. For example, clustered apps can potentially lose data if multiple replicas terminate simultaneously.

Finally, it's worth noting that the StatefulSet controller does its own self-healing and scaling. This is architecturally different from Deployments, which use the ReplicaSet controller for these operations.

Deleting StatefulSets

You need to know two important things about deleting StatefulSets.

Firstly, deleting a StatefulSet object does **not** terminate its Pods in an orderly manner. This means you should scale a StatefulSet to zero replicas before deleting it!

Secondly, you can use `terminationGracePeriodSeconds` to further control Pod termination. For example, it's common to set this to at least 10 seconds to give applications time to flush any buffers and safely commit writes that are still *in flight*.

StatefulSets and Volumes

Volumes are an important part of a StatefulSet Pod's *sticky ID* (state).

When the StatefulSet controller creates Pods, it also creates any volumes they require. To help with this, Kubernetes gives the volumes special names linking them to the correct Pods. Figure 13.2 shows a StatefulSet called **tkb-sts** requesting three Pods, each with a single volume. You can see how Kubernetes uses names to connect volumes with Pods.
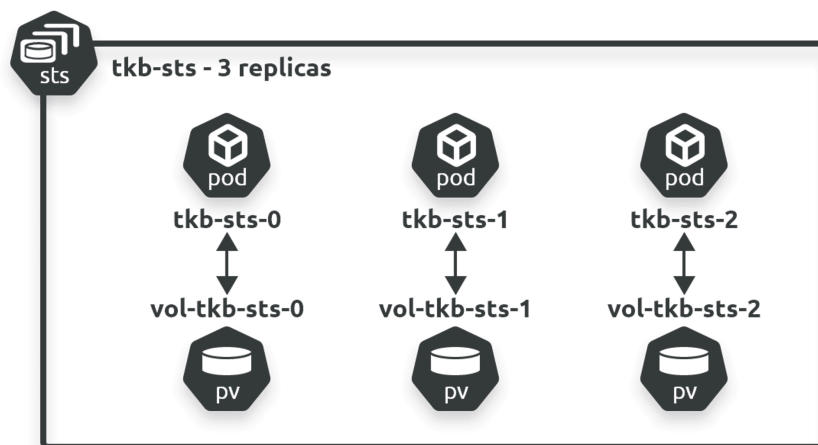
Figure 13.2

Despite being linked with specific Pod replicas, volumes are still decoupled from those Pods via the regular Persistent Volume Claim

system. This means volumes have separate lifecycles, allowing them to survive Pod failures and Pod termination operations. For example, when a StatefulSet Pod fails or is terminated, its associated volumes are unaffected. This allows replacement Pods to connect to the surviving volumes and data, even if Kubernetes schedules the replacement Pods to different cluster nodes.

The same thing happens during scaling operations. If a scale-down operation deletes a StatefulSet Pod, subsequent scale-up operations attach new Pods to the surviving volumes.

This behavior can be a lifesaver if you accidentally delete a StatefulSet Pod, especially if it's the last replica!

Handling failures

The StatefulSet controller observes the state of the cluster and reconciles observed state with desired state.

The simplest example is a Pod failure. If you have a StatefulSet called **tkb-sts** with five replicas and the **tkb-sts-3** replica fails, the controller starts a new Pod with the same name and attaches it to the surviving volumes.

Node failures can be more complex, and some older versions of Kubernetes require manual intervention to replace Pods that were running on failed nodes. This is because it can sometimes be hard for Kubernetes to know if a node has genuinely failed or is just rebooting from a transient event. For example, if a *"failed"* node recovers **after** Kubernetes has replaced its Pods, you'll end up with identical Pods trying to write to the same volume. This can result in data corruption.

Fortunately, newer Kubernetes versions are better at handling scenarios like this.

Network ID and headless Services

We've already said that StatefulSets are for applications that need Pods to be predictable and long-lived. One reason might be an external application that needs to connect and reconnect to the same Pod. Instead of using regular Kubernetes Services that load-balance requests across a set of Pods, StatefulSets use a special kind of Service called a *headless Service.* These create predictable DNS names for each StatefulSet Pod so that apps can query DNS (the service registry) for the full list of Pods and then connect directly to specific Pods.

The following YAML snippet shows a headless Service called **mongo-prod** and a StatefulSet called **sts-mongo**. This is a *headless Service* because it doesn't have a ClusterIP. It's also listed in the StatefulSet as the *governing Service.*

```
apiVersion: v1
kind: Service                    <<---- Service
metadata:
  name: mongo-prod
spec:
  clusterIP: None                <<---- Make it a headless Service
  selector:
    app: mongo
    env: prod
---
apiVersion: apps/v1
kind: StatefulSet                <<---- StatefulSet
metadata:
  name: sts-mongo
spec:
  serviceName: mongo-prod        <<---- Governing Service
```

Let's explain the terms *headless Service* and *governing Service*.

A *headless Service* is a regular Kubernetes Service object without a ClusterIP address (`spec.clusterIP: None`). You make it the StatefulSet's *governing Service* by listing it in the StatefulSet under `spec.serviceName.`

When you combine a headless Service with a StatefulSet like this, the Service creates DNS SRV and DNS A records for every Pod matching the Service's label selector. Other Pods and apps can then query DNS to get the names and IPs of all the StatefulSet's Pods. You'll see this later, but developers must code their applications to query DNS like this.

That covers most of the theory. Let's walk through an example to see how everything comes together.

Hands-on with StatefulSets

In this section, you'll deploy a working StatefulSet.

I've designed and tested the demos on Linode Kubernetes Engine (LKE) and a local Docker Desktop multi-node cluster. If your cluster is on a different cloud or local environment, you'll have to use a different StorageClass. I'll tell you when to do this.

If you haven't already done so, run the following command to clone the book's GitHub repo and switch to the `2025` branch.

```
$ git clone https://github.com/nigelpoulton/TKB.git

$ cd TKB

$ git fetch origin

$ git checkout -b 2025 origin/2025
```

Run all remaining commands from within the `statefulsets` folder.

You're about to deploy the following three objects:

1. A StorageClass
2. A headless Service
3. A StatefulSet

To make things easier to follow, you'll deploy and inspect each object individually. However, it's possible to group them into a single YAML file and deploy them with a single command (see the `app.yml` file in the `statefulsets` folder).

Deploy the StorageClass

StatefulSets need a way to dynamically create volumes. To do this, they need:

- A StorageClass (SC)
- A PersistentVolumeClaim (PVC)

The following YAML is from the `lke-sc.yml` file and defines a StorageClass called **block** that dynamically provisions block storage from the Linode Cloud using the LKE block storage CSI driver. If you're using a Docker Desktop multi-node cluster, you'll need to use the `dd-kind-sc.yml` file instead. If your cluster is on a different cloud, you can do either of the following:

- Create a new StorageClass called **block** for your own cloud — you'll need to create this yourself and configure the `provisioner` and `parameters` sections appropriately
- Use one of your cluster's existing StorageClasses and change the StorageClass name in the PVC in a later step

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: block                              <<---- The PVC will
reference this name
provisioner: linodebs.csi.linode.com       <<---- LKE block storage
CSI plugin
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
reclaimPolicy: Delete
```

Deploy the StorageClass. Remember to use the `dd-kind-sc.yml` file if you're using a local Docker Desktop multi-node cluster.

```
$ kubectl apply -f lke-sc.yml
storageclass.storage.k8s.io/block created
```

List your cluster's StorageClasses to make sure yours is in the list.

```
$ kubectl get sc

ALLOWVOLUME
NAME     PROVISIONER              RECLAIMPOLICY     VOLUMEBINDINGMODE
EXPANSION
block    linodebs.csi.linode.com   Delete
WaitForFirstConsumer    true
```

Your storage class is present, and your StatefulSet will use it to create new volumes dynamically.

Create a governing headless Service

It's helpful to visualize Service objects with a head and a tail. The *head* is the stable ClusterIP address, and the tail is the list of Pods it forwards traffic to. A headless Service is a regular Kubernetes Service object without the head/ClusterIP address.

The primary purpose of headless Services is to create DNS SRV records for StatefulSet Pods. Clients query DNS for individual Pods and then send queries directly to those Pods instead of via the Service's ClusterIP. This is why headless Services don't have a ClusterIP.

The following YAML is from the `headless-svc.yml` file and describes a headless Service called **dullahan** with no ClusterIP address (`spec.clusterIP: None`).

```
apiVersion: v1
kind: Service          <<---- Normal Kubernetes Service
metadata:
  name: dullahan       <<---- Only use valid DNS characters in name
  labels:
    app: web
spec:
  ports:
```

```
  - port: 80
    name: web
  clusterIP: None     <<---- Make this a headless Service
  selector:
    app: web
```

The only difference from a regular Service is that a headless Service
has its `clusterIP` set to `None`.

Run the following command to deploy the headless Service to your
cluster.

```
$ kubectl apply -f headless-svc.yml
service/dullahan created
```

Make sure it exists.

```
$ kubectl get svc
NAME            TYPE          CLUSTER-IP     EXTERNAL-IP     PORT(S)     AGE
dullahan        ClusterIP     None           <none>          80/TCP      31s
```

Deploy the StatefulSet

Now that you've created your storage class and a headless Service,
you can deploy the StatefulSet.

The following YAML is from the `sts.yml` file and defines the
StatefulSet.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: tkb-sts                         <<---- Call the StatefulSet
"tkb-sts"
spec:
  replicas: 3                           <<---- Deploy three replicas
  selector:
    matchLabels:
      app: web
  serviceName: "dullahan"               <<---- Make this the governing
Service
  template:
    metadata:
      labels:
```

```
        app: web
  spec:
    terminationGracePeriodSeconds: 10
    containers:
    - name: ctr-web
      image: nginx:latest
      ports:
      - containerPort: 80
        name: web
      volumeMounts:                              ----┐
      - name: webroot                                | Mount this volume
        mountPath: /usr/share/nginx/html    ----┘
  volumeClaimTemplates:                            ----┐
  - metadata:                                          |
      name: webroot                                    |
    spec:                                              |
      accessModes: [ "ReadWriteOnce" ]                 | Dynamically create
a 10GB volume
      storageClassName: "block"                        | via the "block"
storage class
      resources:                                       |
        requests:                                      |
          storage: 10Gi                    ----┘
```

There's a lot to take in, so let's step through the important parts.

Your StatefulSet is called **tkb-sts**, and Kubernetes uses this as the base for every replica and volume name.

Kubernetes will read the `spec.replicas` field and create 3 replicas called **tkb-sts-0, tkb-sts-1,** and **tkb-sts-2**. It will also create them in order and wait for each one to be *running and ready* before starting the next.

The `spec.serviceName` field designates the governing Service. This is the name of the headless Service you created in the previous step, and it creates the DNS SRV records for the StatefulSet replicas. We call it the *governing Service* because it's in charge of the StatefulSet's DNS subdomain. More on this later.

The remainder of the `spec.template` section defines the Pod template. This is where you define things like which container image to use and which ports to expose.

Last, but certainly not least, is the `spec.volumeClaimTemplates` section. Kubernetes uses this to dynamically create unique PVCs for each StatefulSet Pod. As it's requesting three replicas, Kubernetes will create three unique Pods based on the `spec.template` section and three unique PVCs based on the `spec.volumeClaimTemplates` section. It also ensures the Pods and PVCs get the appropriate names to be linked.

The following YAML snippet shows the volume claim template from the example. It defines a claim template called **webroot** requesting 10GB volumes from the **block** StorageClass.

```
volumeClaimTemplates:
- metadata:
    name: webroot
  spec:
    accessModes: [ "ReadWriteOnce" ]
    storageClassName: "block"
    resources:
      requests:
        storage: 10Gi
```

If you're not using an LKE cluster and you're using one of your cloud's built-in StorageClasses, you'll need to edit the `sts.yml` file and change the `storageClassName` field to a StorageClass on your cluster. You'll be OK if you created your own StorageClass and called it **block**.

Run the following command to deploy the StatefulSet.

```
$ kubectl apply -f sts.yml
statefulset.apps/tkb-sts created
```

Watch the StatefulSet as it ramps up to three replicas. It'll take a minute or two for the controller to create all three Pods and associated PVCs.

```
$ kubectl get sts --watch
NAME        READY    AGE
tkb-sts     0/3      14s
tkb-sts     1/3      30s
```

```
tkb-sts   2/3      60s
tkb-sts   3/3      90s
```

Notice how it took ~30 seconds to start the first replica. Once that was running and ready, it took another 30 seconds to start the second and another 30 for the third. This is the StatefulSet controller starting each replica in turn and waiting for them to be *running and ready* before starting the next.

Now, check the PVCs.

```
$ kubectl get pvc
NAME                 STATUS    VOLUME             CAPACITY   MODES
STORAGECLASS    AGE
webroot-tkb-sts-0    Bound     pvc-1146...f274    10Gi       RWO
block           100s
webroot-tkb-sts-1    Bound     pvc-3026...6bcb    10Gi       RWO
block           70s
webroot-tkb-sts-2    Bound     pvc-2ce7...e56d    10Gi       RWO
block           40s
```

You've got three new PVCs, and each one was created at the same time as one of the Pod replicas. If you look closely, you'll see that each PVC name includes the name of the volume claim template, the StatefulSet, and the associated Pod replica.

volumeClaimTemplate name	Pod Name	PVC Name
webroot	tkb-sts-0	webroot-tkb-sts-0
webroot	tkb-sts-1	webroot-tkb-sts-1
webroot	tkb-sts-2	webroot-tkb-sts-2

Congratulations, your StatefulSet is running and managing three Pods and three volumes.

Testing peer discovery

Let's explain how DNS hostnames and DNS subdomains work with StatefulSets.

All Kubernetes objects get a name within the cluster address space. You can specify a custom address space when you build a cluster, but most use the `cluster.local` DNS domain. Within this domain, Kubernetes constructs DNS subdomains as follows:

- `<object-name>.<service-name>.`
  `<namespace>.svc.cluster.local`

You've deployed three Pods called **tkb-sts-0, tkb-sts-1**, and **tkb-sts-2** in the **default** Namespace governed by the **dullahan** headless Service. This means your Pods will have the following fully qualified DNS names that are predictable and reliable:

- `tkb-sts-0.dullahan.default.svc.cluster.local`
- `tkb-sts-1.dullahan.default.svc.cluster.local`
- `tkb-sts-2.dullahan.default.svc.cluster.local`

It's the job of the headless Service to register these Pods and their IPs against the `dullahan.default.svc.cluster.local` name.

You'll test this by deploying a jump Pod with the `dig` utility pre-installed. You'll then `exec` onto the Pod and use `dig` to query DNS for the Service's SRV records.

Run the following command to deploy the jump Pod from the `jump-pod.yml` file.

```
$ kubectl apply -f jump-pod.yml
pod/jump-pod created
```

Exec onto the Pod.

```
$ kubectl exec -it jump-pod -- bash
root@jump-pod:/#
```

Your terminal prompt will change to indicate it's connected to the jump Pod. Run the following `dig` command from within the jump-pod.

```
# dig SRV dullahan.default.svc.cluster.local
<Snip>
;; QUESTION SECTION:
;dullahan.default.svc.cluster.local. IN SRV
;; ANSWER SECTION:
dullahan.default.svc.cluster.local. 30 IN SRV... tkb-sts-
1.dullahan.default.svc.cluster.local.
dullahan.default.svc.cluster.local. 30 IN SRV... tkb-sts-
0.dullahan.default.svc.cluster.local.
dullahan.default.svc.cluster.local. 30 IN SRV... tkb-sts-
2.dullahan.default.svc.cluster.local.
;; ADDITIONAL SECTION:
tkb-sts-0.dullahan.default.svc.cluster.local. 30 IN A 10.60.0.5
tkb-sts-2.dullahan.default.svc.cluster.local. 30 IN A 10.60.1.7
tkb-sts-1.dullahan.default.svc.cluster.local. 30 IN A 10.60.2.12
<Snip>
```

The output shows that clients asking about
`dullahan.default.svc.cluster.local` (**QUESTION SECTION**) will
get the DNS names (**ANSWER SECTION**) and IPs (**ADDITIONAL
SECTION**) of the three StatefulSet Pods. To be clear... the
ANSWER SECTION maps requests for
`dullahan.default.svc.cluster.local` to the three Pods, and the
ADDITIONAL SECTION maps the Pod names to IPs.

Type `exit` to return to your terminal.

Scaling StatefulSets

Each time Kubernetes scales up a StatefulSet, it creates new Pods
and PVCs. However, when scaling down, Kubernetes only
terminates Pods. This means future scale-up operations only need
to create new Pods and connect them back to the original PVCs.
Kubernetes and the StatefulSet controller handle all of this without
your help.

You currently have three StatefulSet Pods and three PVCs. Edit the
`sts.yml` file, change the replica count from 3 to 2, and save your
changes. When you've done that, run the following command to re-

post the updated configuration to the cluster. You'll have to type
`exit` if you're still connected to the jump Pod.

```
$ kubectl apply -f sts.yml
statefulset.apps/tkb-sts configured
```

Check the StatefulSet and verify the Pod count has reduced to 2.

```
$ kubectl get sts tkb-sts
NAME        READY    AGE
tkb-sts    2/2      12h

$ kubectl get pods
NAME          READY    STATUS    RESTARTS    AGE
tkb-sts-0    1/1      Running   0           12h
tkb-sts-1    1/1      Running   0           12h
```

You've successfully scaled the number of Pods down to 2. If you look
closely, you'll see that Kubernetes deleted the one with the highest
index ordinal and that you still have 3 PVCs. Remember, scaling a
StatefulSet down does **not** delete PVCs.

Verify this.

```
$ kubectl get pvc
NAME                  STATUS    VOLUME           CAPACITY    MODES
STORAGECLASS    AGE
webroot-tkb-sts-0    Bound     pvc-5955...d71c   10Gi        RWO
block           12h
webroot-tkb-sts-1    Bound     pvc-d62c...v701   10Gi        RWO
block           12h
webroot-tkb-sts-2    Bound     pvc-2e2f...5f95   10Gi        RWO
block           12h
```

The status for all three is still showing as `Bound` even though the
tkb-sts-2 Pod no longer exists. If you run a `kubectl describe`
against the **webroot-tkb-sts-2** PVC, you'll see the `Used by` field
shows as `<none>`.

The fact all three PVCs still exist means that scaling back up to 3
replicas will only require a new Pod. The StatefulSet controller will
create the new Pod and connect it to the existing PVC.

Edit the `sts.yml` file again, increment the number of replicas back to 3, and save your changes. After you've done this, run the following command to re-deploy the app.

```
$ kubectl apply -f sts.yml
statefulset.apps/tkb-sts configured
```

Give it a few seconds to deploy the new Pod and then verify with the following command.

```
$ kubectl get sts tkb-sts
NAME       READY   AGE
tkb-sts    3/3     12h
```

You're back to 3 Pods. Describe the new **tkb-sts-2** Pod and verify it mounted the **webroot-tkb-sts-2** volume. Replace the `grep ClaimName` argument with `Select-String -Pattern 'ClaimName'` if you're using Windows.

```
$ kubectl describe pod tkb-sts-2 | grep ClaimName
ClaimName:   webroot-tkb-sts-2
```

Congratulations, the new Pod automatically connected to the correct volume.

It's worth noting that Kubernetes puts scale-down operations on hold if any of the Pods are in a failed state. This protects the resiliency of the app and the integrity of any data.

You can also change how the StatefulSet controller starts and stops Pods by tweaking its `spec.podManagementPolicy` property. The default setting is `OrderedReady` and enforces the behavior of starting one Pod at a time and waiting for the previous Pod to be running and ready before starting the next. Changing the value to `Parallel` will cause the StatefulSet to act more like a *Deployment* where Pods are created and deleted in parallel. For example, scaling from 2 > 5 Pods will instantly create all three new Pods, whereas scaling down from 5 > 2 will delete three Pods in parallel. StatefulSet

naming rules are still enforced, as the setting only applies to scaling operations and does not impact rollouts and rollbacks.

Rollouts

StatefulSets support rolling updates (a.k.a. rollouts). You update the image version in the YAML file and re-post it to the API server, and the controller replaces the old Pods with new ones. However, it always starts with the highest numbered Pod and works down through the list, one at a time, until all Pods are on the new version. The controller also waits for each new Pod to be ready before replacing the one with the next lowest index ordinal.

For more information, run a `kubectl explain sts.spec.updateStrategy` command.

Test a Pod failure

The simplest way to test a failure is to manually delete a Pod. The StatefulSet controller will notice the failure and attempt to reconcile by starting a new Pod and connecting it to the same PVC and volume.

Let's test it.

Confirm you have three healthy Pods in your StatefulSet.

```
$ kubectl get pods
NAME         READY   STATUS    AGE
tkb-sts-0    1/1     Running   12h
tkb-sts-1    1/1     Running   12h
tkb-sts-2    1/1     Running   9m49s
```

Let's delete the **tkb-sts-0** Pod and see if the StatefulSet controller automatically recreates it.

```
$ kubectl delete pod tkb-sts-0
pod "tkb-sts-0" deleted
```

```
$ kubectl get pods --watch
NAME         READY   STATUS             RESTARTS   AGE
tkb-sts-0    1/1     Running            0          12h
tkb-sts-1    1/1     Running            0          12h
tkb-sts-2    1/1     Running            0          10m
tkb-sts-0    0/1     Terminating        0          12h
tkb-sts-0    0/1     Pending            0          0s
tkb-sts-0    0/1     ContainerCreating  0          0s
tkb-sts-0    1/1     Running            0          8s
```

Placing a `--watch` on the command lets you see the StatefulSet controller observe the terminated Pod and create the replacement. This was a clean failure, and the StatefulSet controller immediately created the replacement Pod.

The new Pod has the same name as the failed one. But does it have the same PVC?

Run the following command to confirm that Kubernetes connected the new Pod to the original **webroot-tkb-sts-0** PVC. Don't forget to replace the `grep ClaimName` argument with `Select-String -Pattern 'ClaimName'` if you're on Windows.

```
$ kubectl describe pod tkb-sts-0 | grep ClaimName
    ClaimName:   webroot-tkb-sts-0
```

It worked.

Test a node failure

Recovering from *potential* node failures is a lot more complex and may depend on your Kubernetes version. Modern Kubernetes clusters are far better at automatically replacing Pods from failed nodes, whereas older versions may require manual intervention. This was to prevent Kubernetes from misdiagnosing transient events as catastrophic node failures.

Let's test a simple node failure. I'll give instructions for simulating node failures on LKE clusters and Docker Desktop multi-node Kubernetes clusters, but the principles will be the same for other platforms.

Run the following command to list your StatefulSet Pods and the nodes they're running on.

```
$ kubectl get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE   IP           NODE
tkb-sts-0   1/1     Running   0          11m   10.2.0.132   lke343745-
544835-5547cbe00000
tkb-sts-1   1/1     Running   0          12h   10.2.0.3     lke343745-
544835-1fbc7b870000
tkb-sts-2   1/1     Running   0          21m   10.2.1.7     lke343745-
544835-2b6286320000
```

Look closely at the **NODE** column, and you'll see that Kubernetes has scheduled each replica on different nodes.

In the example, **tkb-sts-0** is running on the **lke343...cbe00000** node. Simulate a node failure by completing the following procedure. I'll show you how to do it on LKE and Docker Desktop multi-node clusters.

Delete a node on LKE

Go to your LKE Dashboard (`cloud.linode.com`), click the **Kubernetes** tab on the left, and click your cluster's name to open its summary page. Scroll down to your cluster's **Node Pool** and **Recycle** one of the nodes running a StatefulSet replica. This will delete and replace the node.
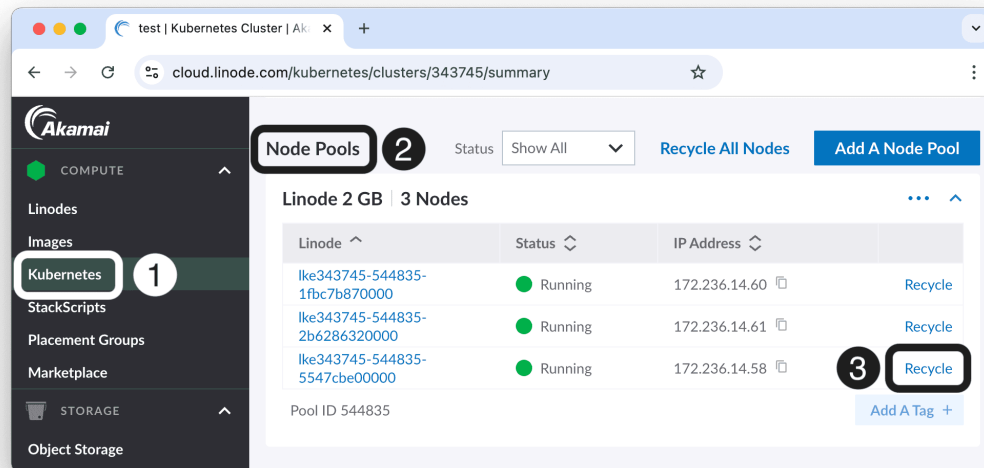
Figure 13.3 - Delete an LKE cluster node

Delete a node on a Docker Desktop multi-node cluster (kind)

This only works for *Docker Desktop multi-node (kind)* clusters, and you'll need to delete and recreate your cluster to restore it back to three nodes. You cannot follow along if you have a *Docker Desktop single node (kubeadm)* cluster.

Docker Desktop runs cluster nodes as containers, and a three-node cluster will have three containers called **desktop-control-plane, desktop-worker,** and **desktop-worker2.**

Open Docker Desktop and navigate to the **Containers** tab in the left navigation pane. Locate the container with the same name as the worker node you want to delete and click the trash icon to the right of the container to delete it. Be sure to delete a node running a StatefulSet replica. Figure 13.4 shows how to delete the **desktop-worker2** node.
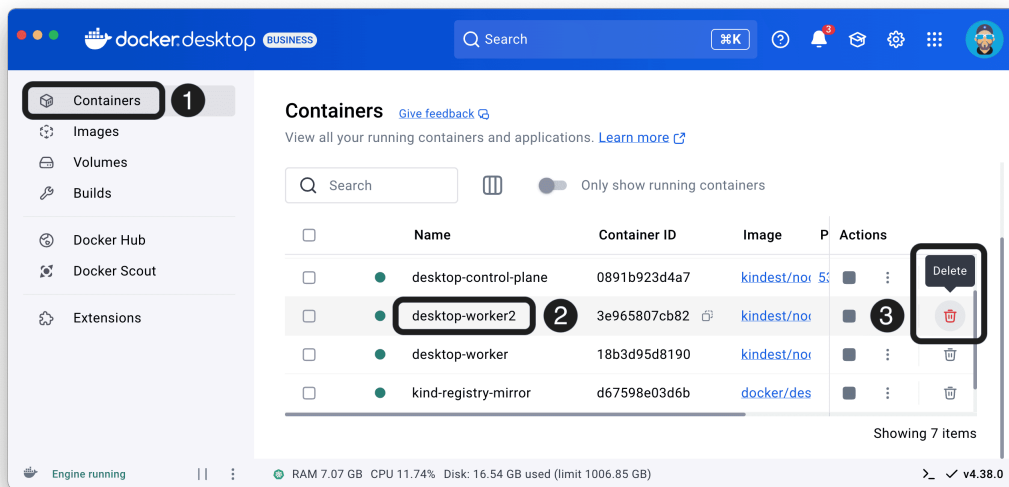
Figure 13.4 - Delete a Docker Desktop cluster node

Observe the StatefulSet recovery process

Once you've deleted your node, you can run the following command to witness the StatefulSet controller recover from the failure. It may take a minute or two for the process to complete while the StatefulSet controller observes the missing Pod and decides how to act.

```
$ kubectl get pods -o wide --watch
NAME          READY    STATUS              RESTARTS     AGE    IP
NODE
tkb-sts-0    1/1      Running              0            14m    10.2.0.132
lke343745...cbe00000
tkb-sts-1    1/1      Running              0            12h    10.2.0.3
lke343745...7b870000
tkb-sts-2    1/1      Running              0            30m    10.2.1.7
lke343745...86320000
tkb-sts-0    1/1      Terminating          0            14m    10.2.0.132
lke343745...cbe00000
<Snip>
tkb-sts-0    0/1      Completed            0            14m    10.2.0.132
lke343745...cbe00000
<Snip>
tkb-sts-0    0/1      Pending              0            0s     <none>
<none>
tkb-sts-0    0/1      Pending              0            0s     <none>
lke343745...7b870000
```

```
tkb-sts-0   0/1    ContainerCreating    0         0s     <none>
lke343745...7b870000
tkb-sts-0   0/1    ContainerCreating    0         110s   <none>
lke343745...7b870000
tkb-sts-0   1/1    Running              0         111s   10.2.0.4
lke343745...7b870000
```

Let's examine the output.

The sᴛᴀᴛᴜs column shows the **tkb-sts-0** Pod terminate, complete, enter the pending state, progress to the container creating state, and finally reach the running state. The Pod terminates when Kubernetes notices the missing node and the StatefulSet drops from three replicas to two. This results in the observed state of the cluster no longer matching your desired state, and the StatefulSet controller kicks into action and creates a new copy of the missing **tkb-sts-0** replica. The new replica enters the *pending* state while the scheduler allocates it to a surviving node. Once assigned to a node, it enters the *ContainerCreating* state while the node downloads the appropriate image and starts the container. It may appear to hang in this state while Kubernetes releases the previous PVC attachment. It finally binds the new replica to the PVC, enters the *running* state, and the StatefulSet returns to three replicas.

If you examine the ɴᴏᴅᴇ column, you'll see the original **tkb-sts-0** replica was running on the **lke343745...cbe00000** node, but Kubernetes has scheduled the replacement replica to the **lke343745...7b870000** node. This is because the previous node no longer exists.

If you're using an LKE cluster, you'll have a new node replacing the one you recycled. However, Kubernetes does not re-balance existing replicas to the new node.

Deleting StatefulSets

Earlier in the chapter, we said that Kubernetes doesn't terminate Pods in order when you delete a StatefulSet. Therefore, if your applications and data are sensitive to ordered shutdown, you should scale the StatefulSet to zero before deleting it.

Scale your StatefulSet to 0 replicas and confirm the operation. It may take a few seconds to scale all the way down to 0.

```
$ kubectl scale sts tkb-sts --replicas=0
statefulset.apps/tkb-sts scaled

$ kubectl get sts tkb-sts
NAME       READY   AGE
tkb-sts    0/0     13h
```

You can delete the StatefulSet as soon as it gets to zero replicas.

```
$ kubectl delete sts tkb-sts
statefulset.apps "tkb-sts" deleted
```

Feel free to exec onto the jump-pod and run another `dig` to prove that Kubernetes deleted the SRV records from the cluster DNS. You may have already terminated your jump pod when you deleted a cluster node.

Clean up

You've already deleted the StatefulSet and its Pods. However, the jump Pod, headless Service, volumes, and StorageClass still exist. You can delete them with the following commands if you've been following along. Failure to delete the volumes will incur unexpected cloud costs.

Delete the jump Pod. Don't worry if it's already gone.

```
$ kubectl delete pod jump-pod
```

Delete the headless Service.

```
$ kubectl delete svc dullahan
```

Delete the PVCs. This will delete the associated PVs and backend storage on the Linode Cloud. If you used your own StorageClass, you should check your storage backend to confirm that the external volumes also get deleted. Failure to delete the backend volumes may result in unwanted costs.

```
$ kubectl delete pvc webroot-tkb-sts-0 webroot-tkb-sts-1 webroot-tkb-
sts-2
```

Delete the StorageClass.

```
$ kubectl delete sc flash
```

If you deleted a node from a Docker Desktop multi-node (kind) cluster, you'll need to delete and rebuild your cluster back to the desired number of nodes.

Chapter Summary

In this chapter, you learned how to use StatefulSets to deploy and manage applications that work with persistent data and state.

StatefulSets can self-heal, scale up and down, and perform rollouts. Rollbacks require manual attention.

Each StatefulSet Pod gets a predictable and persistent name, DNS hostname, and its own unique volumes. These stay with the Pod for its entire lifecycle, including failures, restarts, scaling, and other scheduling operations. In fact, StatefulSet Pod names are integral to scaling operations and connecting to the right storage volumes.

Finally, StatefulSets are only a framework. We need to design applications to take advantage of the way they work.

14: API security and RBAC

Kubernetes is *API-centric* and the API is served through the *API server*. In this chapter, you'll follow a typical API request as it passes through various security-related checks.

I've divided the chapter into the following sections:

- API security big picture
- Authentication
- Authorization (RBAC)
- Admission control

See Chapter 15 for a more in-depth look at the design and structure of the API.

API security big picture

All of the following make CRUD-style requests to the API server (create, read, update, delete):

- Operators and developers using `kubectl`
- Pods
- Kubelets
- Control plane services
- Kubernetes-native apps

Figure 14.1 shows the flow of a typical API request passing through the standard checks. The flow is the same, no matter where the request originates. **authN** is short for authentication, whereas **authZ** is short for authorization.

Figure 14.1

Consider a quick example where a user called **grant-ward** is trying to create a Deployment called **hive** in the **terran** Namespace.

The **grant-ward** user uses the `kubectl apply` command to send the YAML file to Kubernetes to create the **Deployment** in the **terran** Namespace. The `kubectl` command-line tool generates a request to the API server with the user's credentials

embedded. The connection between `kubectl` and the API server is secured by TLS. As soon as the request reaches the API server, the *authentication* module determines whether the request originates from **grant-ward** or an imposter. Assuming it is **grant-ward**, the *authorization* module determines whether **grant-ward** has permission to create **Deployments** in the **terran** Namespace. If the request passes authentication **and** authorization, *admission controllers* ensure the Deployment object meets policy requirements. The request is executed only after passing authentication, authorization, and admission control checks.

The process is similar to flying on a commercial plane. You travel to the airport and *authenticate* yourself with a photo ID, usually your passport. Assuming you clear passport authentication, the system checks if you've bought a seat and are therefore *authorized* to board the plane. If you pass authentication and you're authorized to board, admission controls may check and apply airline policies such as restricting hand luggage and prohibiting alcohol and weapons in the cabin. After all that, you can finally take your seat and fly to your destination.

Let's take a closer look at authentication.

Authentication

Authentication is about proving your identity. You might see or hear it shortened to *authN*, pronounced *"auth en"*.

Credentials are at the heart of authentication, and **all requests to the API server include credentials**. It's the responsibility of the authentication layer to verify them. If verification fails, the API server returns an HTTP 401 and denies the request. If the request passes authentication, it moves on to authorization.

The Kubernetes authentication layer is pluggable, and popular modules include *client certificates, webhooks,* and integration with external identity management systems such as *Active Directory (AD)* and cloud-based *Identity Access Management (IAM)* systems. In fact, Kubernetes does **not** have its own built-in identity database. Instead, it forces you to use an external system. This avoids creating *yet another identity management silo*.

Out-of-the-box, most Kubernetes clusters support *client certificates*, but in the real world you'll want to integrate with your chosen cloud or corporate identity management system. Many hosted Kubernetes services automatically integrate with the underlying cloud's identity management system.

In these situations, Kubernetes offloads authentication to the external system.

Checking your current authentication setup

Your cluster details and user credentials are stored in a *kubeconfig* file. Tools like `kubectl` read this file to determine which cluster to send commands to and which credentials to use. The file is called `config` and lives in the following locations:

- **Windows:** `C:\Users\<user>\.kube\config`
- **Linux/Mac:** `/home/<user>/.kube/config`

Here's what a kubeconfig file looks like.

```
apiVersion: v1
kind: Config
clusters:                        <<---- Cluster block defining one or more clusters and
certs
- cluster:
  name: prod-shield             <<---- This block defines a cluster called "prod-shield"
    server: https://<url-or-ip-address-of-api-server>:443     <<---- This is the cluster's
URL
    certificate-authority-data: LS0tLS1C...LS0tCg==           <<---- Cluster's certificate
 users:                          <<---- Users block defining one or more users and
credentials
- name: njfury                  <<---- User called njfury
  user:
    as-user-extra: {}
    token: eyJhbGciOiJSUzI1NiIsImtpZCI6IlZwMzl...SZY3uUQ     <<---- User's credentials
contexts:                        <<---- Context block. A context is a cluster + user
- context:
  name: shield-admin            <<---- This block defines a context called "shield-admin"
    cluster: prod-shield        <<---- Cluster
    user: njfury                <<---- User
    namespace: default
current-context: shield-admin   <<---- Context used by kubectl
```

If you look closely, you'll see it has four top-level sections:

- Clusters
- Users
- Contexts
- Current-context

The `clusters` section defines one or more Kubernetes clusters. Each has a friendly name, an API server endpoint, and the public key of its certificate authority (CA).

The `users` section defines one or more users. Each user requires a name and token. The token is often an X.509 certificate signed by the cluster's CA (or a CA trusted by the cluster).

The `contexts` section is a list of user and cluster pairs, and the `current-context` is the cluster and user for `kubectl` commands.

The following YAML snippet is from the same kubeconfig file and will send all requests to the `prod-shield` cluster as the **njfury** user.

```
current-context: shield-admin    <<---- Current context
contexts:
- context:
  name: shield-admin             <<---- This block defines a context called "shield-admin"
    cluster: prod-shield         <<---- Send commands to this cluster
    user: njfury                 <<---- User to authenticate as
    namespace: default
```

It's the job of the cluster's authentication module to determine whether the user is genuinely **njfury**. If your cluster integrates with an external IAM system, it'll hand off authentication to that system.

Assuming authentication is successful, requests progress to the authorization phase.

Authorization (RBAC)

Authorization happens immediately after successful authentication, and you'll sometimes see it shortened to *authZ* (pronounced "auth zee").

Kubernetes authorization is also pluggable, and you can run multiple authZ modules on a single cluster. However, most clusters use RBAC. Also, if your cluster has multiple authorization modules, as soon as any module authorizes a request, it skips all other authZ modules and moves immediately to *admissions control*.

This section covers the following:

- RBAC big picture
- Users and permissions
- Cluster-level users and permissions
- Pre-configured users and permissions

RBAC big picture

The most common authorization module is RBAC (Role-Based Access Control). At the highest level, RBAC is about three things:

1. Users
2. Actions
3. Resources

Which *users* can perform which *actions* against which *resources*.

The following table shows a few examples.

User (subject)	Action	Resource	Effect
Bao	create	Pods	Bao can create Pods
Kalila	list	Deployments	Kalila can list Deployments
Josh	delete	ServiceAccounts	Josh can delete ServiceAccounts

RBAC is enabled on most Kubernetes clusters and is a *least-privilege deny-by-default system*. This means it locks everything down, and you need to create *allow rules* to open things up. In fact, Kubernetes doesn't support *deny rules* — it only supports *allow rules*. This might seem small, but it makes implementation and troubleshooting much simpler.

Users and Permissions

Two concepts are vital to understanding Kubernetes RBAC:

- Roles
- RoleBindings

Roles define a set of permissions, and *RoleBindings* bind them to users.

The following resource manifest defines a Role object. It's called **read-deployments** and grants permission to `get, watch,` and `list` Deployment objects in the `shield` Namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: shield
  name: read-deployments
rules:
- verbs: ["get", "watch", "list"]   <<---- Allowed actions
  apiGroups: ["apps"]               ----┐ on resources
  resources: ["deployments"]        ----┘ of this type
```

However, Roles don't do anything until you bind them to users.

The following RoleBinding binds the previous **read-deployments** Role to a user called **sky**.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-deployments
  namespace: shield
subjects:
- kind: User
  name: sky                  <<---- Name of the authenticated user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: read-deployments     <<---- Bind this Role to the authenticated user above
  apiGroup: rbac.authorization.k8s.io
```

Deploying both objects to your cluster will allow an authenticated user called **sky** to run commands such as `kubectl get deployments -n shield`.

The username listed in the RoleBinding must be a string and has to match an authenticated user.

Looking closer at rules

Role objects have a `rules` section that defines the following three properties:

- verbs
- apiGroups
- resources

Together, they define which actions are allowed against which objects.

The `verbs` field lists permitted actions, whereas the `apiGroups` and `resources` fields identify which objects the actions are permitted on. The following snippet from the previous Role object allows read access (`get, watch` and `list`) against Deployment objects.

```
rules:
- verbs: ["get", "watch", "list"]
  apiGroups: ["apps"]
  resources: ["deployments"]
```

The following table shows some possible `apiGroup` and `resources` combinations.

apiGroup	resource	Kubernetes API path
""	pods	/api/v1/namespaces/{namespace}/pods
""	secrets	/api/v1/namespaces/{namespace}/secrets
"storage.k8s.io"	storageclass	/apis/storage.k8s.io/v1/storageclasses
"apps"	deployments	/apis/apps/v1/namespaces/{namespace}/deployments

An empty set of double quotes ("") in the `apiGroups` field indicates the `core` API group. You need to specify all other API groups as a string enclosed in double-quotes.

The following table lists the complete set of verbs Kubernetes supports for object access. It demonstrates the REST-based nature of the Kubernetes API by mapping the verbs to standard HTTP methods and HTTP response codes.

Kubernetes verb(s)	HTTP method	Common responses
create	POST	201 created, 403 Access Denied
get, list, watch	GET	200 OK, 403 Access Denied
update	PUT	200 OK, 403 Access Denied
patch	PATCH	200 OK, 403 Access Denied
delete	DELETE	200 OK, 403 Access Denied

Run the following command to show all API resources and supported verbs. The output is useful when you're building rule definitions.

```
$ kubectl api-resources --sort-by name -o wide
NAME          APIGROUP          KIND        VERBS
deployments   apps              Deployment  [create delete ... get list patch update
watch]
ingresses     networking.k8s.io Ingress     [create delete ... get list patch update
watch]
pods                            Pod         [create delete ... get list patch update
watch]
secrets                         Secret      [create delete ... get list patch update
watch]
services                        Service     [create delete get list patch update watch]
<Snip>
```

When building rules, you can use the asterisk (*) to refer to all API groups, all resources, and all verbs. For example, the following Role object grants all actions on all resources in every API group. It's just for demonstration purposes, and you probably shouldn't create rules like this.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: shield
  name: read-deployments
rules:
- verbs: ["*"]        <<---- All actions
  resources: ["*"]    <<---- on all resources
  apiGroups: ["*"]    <<---- in every API group
```

Cluster-level users and permissions

So far, you've seen Roles and RoleBindings. However, Kubernetes actually has four RBAC objects:

- Roles
- RoleBindings
- ClusterRoles
- ClusterRoleBindings

Roles and RoleBindings are *namespaced objects,* meaning you apply them to specific Namespaces. On the other hand, *ClusterRoles* and *ClusterRoleBindings* are cluster-

wide objects and apply to all Namespaces. They're all defined in the same API sub-group, and their YAML structures are almost identical.

A powerful pattern is to use ClusterRoles to define roles once at the cluster level and then use RoleBindings to bind them to multiple specific Namespaces. This lets you define common roles once and re-use them in as many Namespaces as required. Figure 14.2 shows a single ClusterRole applied to two Namespaces via two separate RoleBindings.
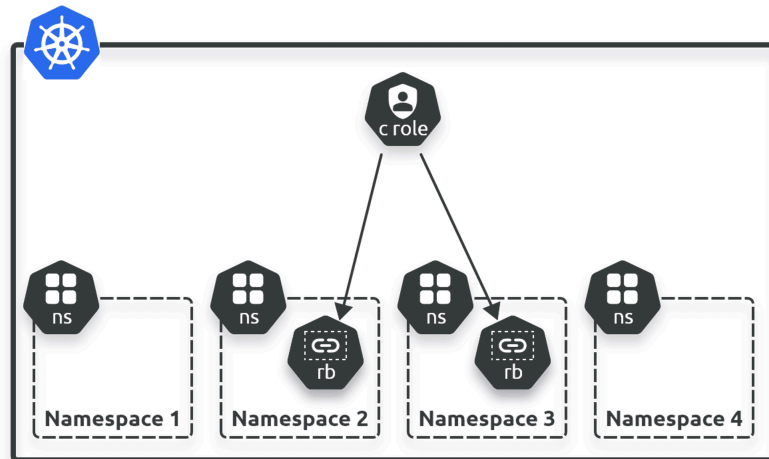
Figure 14.2 - Combining ClusterRoles and RoleBindings

The following YAML defines the **read-deployments** role from earlier. But this time it's a ClusterRole, meaning you can use RoleBindings to apply it to as many Namespaces as you need — one RoleBinding per Namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole                       <<---- Cluster-scoped role
metadata:
  name: read-deployments
rules:
- verbs: ["get", "watch", "list"]
  apiGroups: ["apps"]
  resources: ["deployments"]
```

If you look closely at the YAML, the only difference with the earlier one is that this one has its `kind` property set to `ClusterRole` and it doesn't have a `metadata.namespace` property.

Real-world example

Let's look at a real-world example.

Most Kubernetes clusters have a set of pre-created roles and bindings to help with initial configuration and getting started.

The following example walks you through how Docker Desktop's built-in multi-node Kubernetes cluster uses ClusterRoles and ClusterRoleBindings to grant cluster admin rights to the user configured in your kubeconfig file.

You can follow along if you're using Docker Desktop's built-in multi-node Kubernetes cluster that we showed you how to build in Chapter 3. Other clusters will do things slightly differently, but the principles will be similar.

Docker Desktop automatically configures your kubeconfig file with a client certificate defining an admin user. The certificate is signed by the cluster's built-in CA, meaning the cluster will trust its credentials.

Run the following command to see the Docker Desktop user entry in your kubeconfig file. I've trimmed the output to highlight the bits we're interested in.

```
$ kubectl config view
<Snip>
users:
- name: docker-desktop                 ----┐
  user:                                     | This is the Docker Desktop
    client-certificate-data: DATA+OMITTED   | kubeconfig entry for the built-in user
    client-key-data: DATA+OMITTED           | including client certificate
<Snip>                                  ----┘
```

The user entry is called **docker-desktop**, but this is just a friendly name. The username that `kubectl` authenticates with is embedded within the client certificate in the same section of the file.

Run the following long command to decode the username and group memberships embedded in your kubeconfig's client certificate. The command only works on Linux-style systems, and you'll need the `jq` utility installed. You'll also need to make sure your kubeconfig's *current context* is set to your Docker Desktop user and cluster.

```
$ kubectl config view --raw -o json \
    | jq ".users[] | select(.name==\"docker-desktop\")" \
    | jq -r '.user["client-certificate-data"]' \
    | base64 -d | openssl x509 -text | grep "Subject:"

    Subject: O=kubeadm:cluster-admins, CN=kubernetes-admin
```

The output shows that `kubectl` commands will authenticate as the **kubernetes-admin** user that is a member of the **kubeadm:cluster-admins** group.

> **Note:** Client certificates encode usernames in the `CN` property and group memberships in the `O` property.

Remember, the cluster's CA signs the certificate, so it will pass authentication.

Now that you know you're authenticating as the **kubernetes-admin** user, which is a member of the **kubeadm:cluster-admins** group, let's see how the cluster uses

ClusterRoles and ClusterRoleBindings to give you cluster-wide admin access.

Many Kubernetes clusters use a ClusterRole called **cluster-admin** to grant admin rights on the cluster. This means your **kubernetes-admin** user (a member of the **kubeadm:cluster-admins** group) needs binding to the **cluster-admin** ClusterRole. See Figure 14.3.
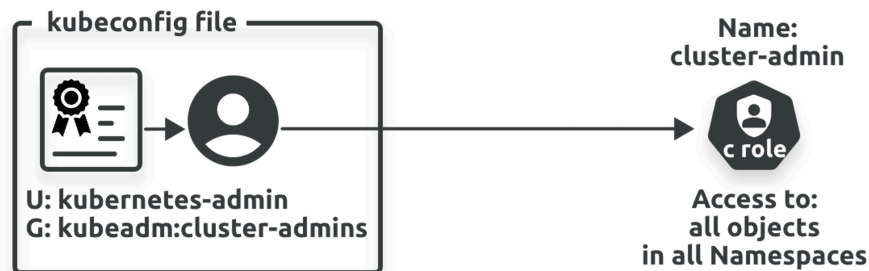
Figure 14.3

Run the following command to see what access the **cluster-admin** ClusterRole has.

```
$ 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
  ---------  -----------------  --------------  -----
  *.*        []                 []              [*]
             [*]                []              [*]
```

The `PolicyRule` section shows this Role has access to all *verbs* on all *resources* in all Namespaces. This means accounts bound to this role can *do everything to every object everywhere*. This is the equivalent of *root* and is a powerful and dangerous set of permissions.

Run the following command to see if your cluster has any ClusterRoleBindings referencing the **cluster-admin** role.

```
$ kubectl get clusterrolebindings | grep cluster-admin
NAME                      ROLE
cluster-admin             ClusterRole/cluster-admin
kubeadm:cluster-admins    ClusterRole/cluster-admin
```

It's referenced in two ClusterRoleBindings, and we're interested in the **kubeadm:cluster-admins** binding. Hopefully, it will list our **kubernetes-admin** user or the **kubeadm:cluster-admins** group our user is a member of.

Run the following command to inspect it.

```
$ kubectl describe clusterrolebindings kubeadm:cluster-admins
Name:          kubeadm:cluster-admins
Labels:         <none>
Annotations:   <none>
Subjects:                             ----┐
  Kind   Name                          | Bind subjects (users) that are
  ----   ----                          | members of the
  Group  kubeadm:cluster-admins        | "kubeadm:cluster-admins group"
Role:                                  | to the
  Kind:  ClusterRole                   | ClusterRole
  Name:  cluster-admin         ----┘ called "cluster-admin"
```

Great. It binds subjects (users) in the **kubeadm:cluster-admins** group to the **cluster-admin** Role. Our **kubernetes-admin** user is a member of that group, so it will get full admin rights on the cluster.

In summary, and as shown in Figure 14.4, Docker Desktop configures your kubeconfig file with a client certificate that identifies a user called **kubernetes-admin** that is a member of the **kubeadm:cluster-admins** group. The certificate is signed by the cluster's CA, meaning it will pass authentication. The Docker Desktop Kubernetes cluster has a ClusterRoleBinding called **kubeadm:cluster-admins** that binds authenticated members of the **kubeadm:cluster-admins** group to a ClusterRole called **cluster-admin**. This **cluster-admin** ClusterRole grants admin rights to all objects in all Namespaces.
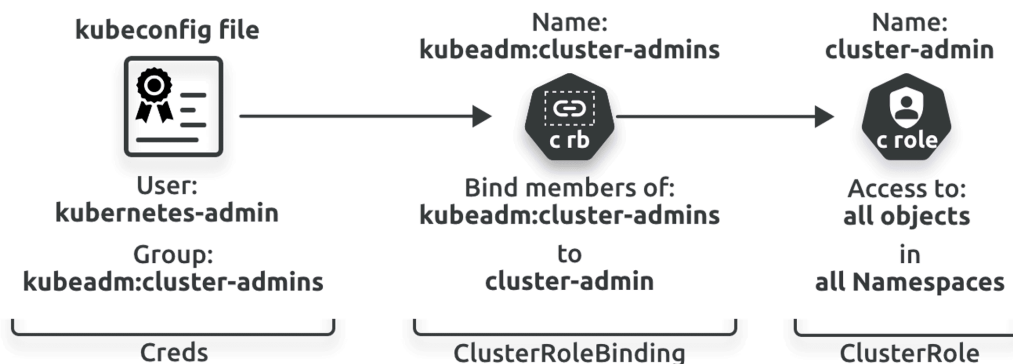
Figure 14.4 - Mapping kubectl users to cluster admin

Summarizing authorization

Authorization ensures authenticated users are allowed to execute actions. RBAC is a popular Kubernetes authorization module that implements least privilege access based on a deny-by-default model that denies all actions unless you create rules that allow them.

Kubernetes RBAC uses Roles and ClusterRoles to grant permissions, and it uses RoleBindings and ClusterRoleBindings to bind those permissions to users.

Once a request passes authentication and authorization, it moves to admission control.

Admission control

Admission control runs immediately after successful authentication and authorization and is all about *policies.*

Kubernetes supports two types of admission controllers:

- Mutating
- Validating

The names tell you a lot. *Mutating* controllers check for compliance and can modify requests, whereas *validating* controllers check for compliance but cannot modify requests.

Kubernetes always runs mutating controllers first, and both types only apply to requests attempting to modify the cluster. Read requests are not subjected to admission control.

As a quick example, you might have a production cluster with a policy that all new and updated objects must have the `env=prod` label. A mutating controller can check new and updated objects for the presence of the label and add it if it doesn't exist. However, a validating controller can only reject the request if the label doesn't exist.

Running the following command on a Docker Desktop cluster shows the API server is configured to use the `NodeRestriction` admission controller.

```
$ kubectl describe pod kube-apiserver-desktop-control-plane  \
  --namespace kube-system | grep admission

--enable-admission-plugins=NodeRestriction
```

Most real-world clusters will run a lot more admission controllers. For example, the `AlwaysPullImages` mutating admission controller sets the `spec.containers.imagePullPolicy` of all new Pods to `Always`. This forces the container runtime on all nodes to pull all images from the configured registry, preventing Pods from using locally cached images. It requires all nodes to have valid credentials to pull images.

If any admission controller rejects a request, the request is immediately denied without checking other admission controllers. This means all admission controllers must approve a request before it runs on the cluster.

As previously mentioned, there are lots of admission controllers, and they're very important in real-world production clusters.

Chapter summary

In this chapter, you learned that all requests to the API server include credentials and must pass authentication, authorization, and then admission control checks. The connection between the client and the API server is also secured with TLS.

The authentication layer validates the identity of requests, and most clusters support client certificates. However, production clusters should use enterprise-grade Identity and Access Management (IAM) solutions.

The authorization layer checks whether authenticated requests have permission to carry out specific actions. This layer is also pluggable, and the most common authorization module is RBAC. RBAC comprises four objects that let you define permissions and grant them to authenticated users.

Admission controllers kick in after authorization and are responsible for enforcing policies. Validating admission controllers reject requests if they don't meet policy requirements, whereas mutating controllers can modify requests to meet policy requirements.

15: The Kubernetes API

If you want to master Kubernetes, you need to understand the API and how it works. However, it's large and complex, and it can be confusing if you're new to APIs and uncomfortable with terms like *RESTful*. If that's you, this chapter blows the confusion away and gets you up to speed with the fundamentals of the Kubernetes API.

I've divided things up as follows, and the last two sections have lots of hands-on exercises:

- Kubernetes API big picture
- The API server
- The API

However, let's mention a few quick things before getting started.

I've included lots of jargon in this chapter to help you get used to it.

I highly recommend you complete the hands-on parts as they'll reinforce the theory.

Finally, Pods, Services, StatefulSets, StorageClasses, and more are all *resources* in the API. However, it's common to call them *objects* when deployed to a cluster. We'll use the terms *resource* and *object* interchangeably.

Kubernetes API big picture

Kubernetes is *API-centric.* This means all resources are defined in the *API,* and all configuration and querying goes through the *API server.*

Administrators and clients send requests to the API server to create, read, update, and delete objects. They'll often use `kubectl` to send these requests, but they can also craft them in code or generate them through API testing and development tools. The point is, no matter *how* you generate requests, they always go to the API server, where they're authenticated and authorized before being accepted. If it's a request to create an object, Kubernetes persists the object definition in the cluster store in its serialized state and schedules it to the cluster.

Figure 15.1 shows the high-level process and highlights the central nature of the API and API server.
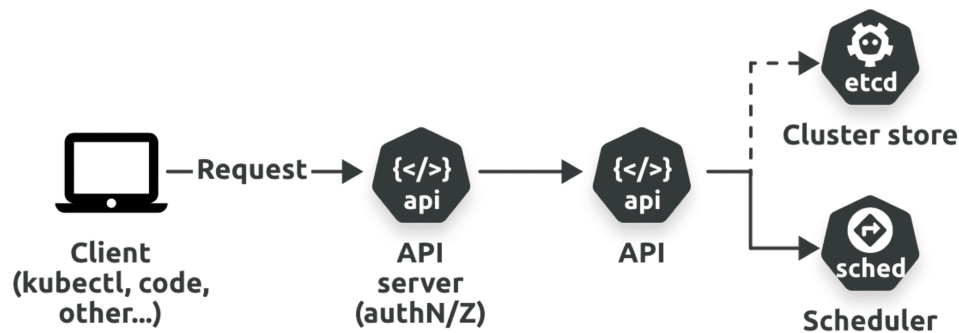
Figure 15.1

Let's demystify some jargon.

JSON serialization

What does it mean to persist an object to the cluster store in its *serialized state*?

Serialization is the process of converting an object into a string or stream of bytes so it can be sent over a network and persisted to a data store. The reverse process of converting a string or stream of bytes into an object is *deserialization*.

Kubernetes serializes objects, such as Pods and Services, as JSON strings and sends them over the network via HTTP. The process happens in both directions:

- Clients like `kubectl` serialize objects when posting them to the API server
- The API server serializes responses back to clients

As well as serializing objects for transit over the network, Kubernetes also serializes them for storage in the cluster store.

Kubernetes supports JSON and Protobuf serialization schemas. Protobuf is faster and more efficient, and it scales better than JSON. But it's harder to inspect and troubleshoot. At the time of writing, Kubernetes typically communicates with external clients via JSON but uses Protobuf when communicating with internal cluster components.

One final thing about serialization. When clients send requests to the API server, they use the `Content-Type` header to list the serialization schemas they support. For example, a client that only supports JSON will specify `Content-Type: application/json` in the HTTP header of all requests. Kubernetes will honor this with a serialized response in JSON. You'll see this in the hands-on sections later.

API analogy

Consider a quick analogy that might help you conceptualize the Kubernetes API. You can skip this section if you're already familiar with the concept of APIs.

Amazon sells lots of stuff:

1. That *stuff* is stored in warehouses and exposed online via the Amazon website
2. You use tools such as browsers and apps to search the website and buy stuff
3. Third parties sell their own stuff through Amazon, and you use the same browser and website
4. When you buy stuff through the website, it gets delivered to you, and you can start using it
5. The Amazon website lets you track your stuff while it's being prepared and delivered
6. Once it's delivered, you can use the Amazon website to order more or send stuff back

Kubernetes is very similar.

Kubernetes has lots of resources (stuff) such as Pods, Services, and Ingresses:

1. Kubernetes stuff is defined in the API and exposed through the API server
2. You use tools like `kubectl` to talk to the API server and request resources
3. Third parties define their own Kubernetes resources, and you use the same `kubectl` and API server to request them
4. When you request resources through the API server, they get created on your cluster, and you can start using them
5. The API server lets you watch your objects being created and deployed
6. Once they're created, you can use the API server to create more and even delete them

Figure 15.2 shows the comparison, and you can see a feature-for-feature comparison in the following table. However, this is just an analogy, so not everything matches perfectly.
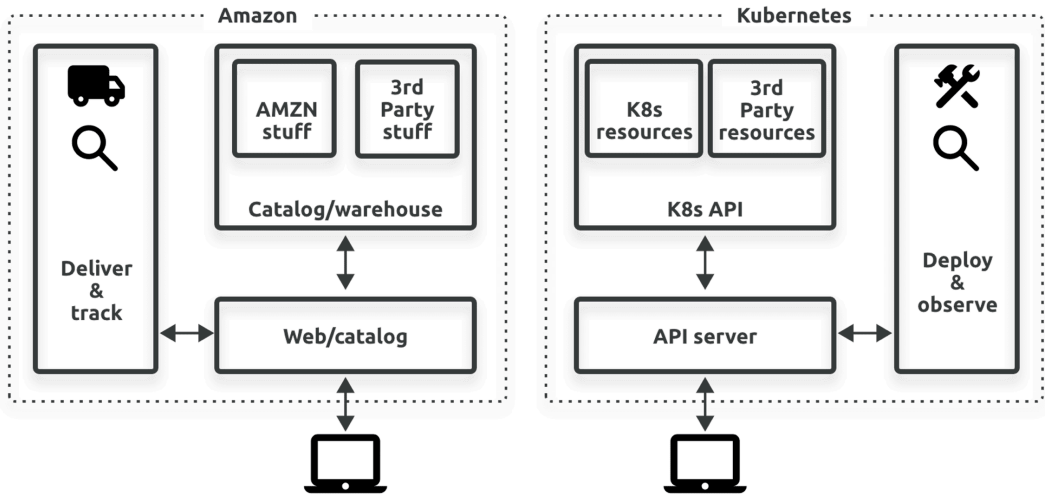
Figure 15.2

Amazon	Kubernetes
Stuff	Resources/objects
Warehouse	API
Browser	kubectl
Amazon website	API server

To recap. All deployable objects, such as Pods, Services, and Ingresses, are defined as resources in the API. If an object doesn't exist in the API, you can't deploy it. This is the same with Amazon — you can only buy stuff listed on the website.

API resources have properties you can inspect and configure. For example, you can configure all of the following Pod properties (we're only showing some):

- metadata (name, labels, Namespace, annotations…)
- restart policy
- service account name
- runtime class
- containers
- volumes

This is the same as buying stuff on Amazon. For example, when you buy a USB cable, you configure choices such as USB type, cable length, and cable color.

To deploy a Pod, you send a Pod YAML file to the API server. Assuming the YAML is valid and you're authorized to create Pods, Kubernetes deploys it to the cluster. After that, you can query the API server to get its status. When it's time to delete it, you send the delete request to the API server.

This is the same as buying stuff from Amazon. To buy the previously mentioned USB cable, you configure the color, cable length, and connector options and submit the request to the Amazon website. Assuming it's in stock and you provide the funds, it gets shipped to you. After that, you can use the website to track the shipment. If you need to return the item or make a complaint, you do all that through the Amazon website.

That's enough with analogies. Let's take a closer look at the API server.

The API server

The API server exposes the API over a RESTful HTTPS interface, and it's common for the API server to be exposed on port `443` or `6443`. However, you can configure it to operate on whatever port you require.

Run the following command to see the address and port your Kubernetes cluster is exposed on.

```
$ kubectl cluster-info
Kubernetes control plane is running at https://kubernetes.docker.internal:6443
```

The API server acts as the front-end to the API and is a bit like *Grand Central station* for Kubernetes — everything talks to everything else via REST API calls to the API server. For example:

- All `kubectl` commands go to the API server (creating, retrieving, updating, and deleting objects)
- All kubelets watch the API server for new tasks and report the status to the API server
- All control plane services share data and status info via the API server

Let's dig deeper and demystify more jargon.

The API server is a Kubernetes control plane service that some clusters run as a set of Pods in the `kube-system` Namespace. If you build and manage your own clusters, you need to ensure the control plane is highly available and has enough performance to ensure the API server responds quickly to requests. If you're using *hosted Kubernetes*, the API server implementation, including

performance and availability, is managed by your cloud provider and hidden from you.

The main job of the API server is to expose the API to clients inside and outside the cluster. It uses TLS to encrypt client connections, and it leverages authentication and authorization mechanisms to ensure only valid requests are accepted and executed. Requests from internal and external sources all have to pass through the same authentication and authorization.

The API is *RESTful*. This is jargon for a modern web API that accepts CRUD-style requests via standard HTTP methods. *CRUD-style operations* are simple **c**reate, **r**ead, **u**pdate, **d**elete operations, and they map to the standard POST, GET, PUT, PATCH, and DELETE *HTTP methods*.

The following table shows how CRUD operations, HTTP methods, and `kubectl` commands match up. If you've read the chapter on API security, you'll know we use the term *verb* to refer to CRUD operations.

K8s CRUD verb	HTTP method	kubectl example
create	POST	$ kubectl create -f <filename>
get list, watch	GET	$ kubectl get pods
update	PUT/PATCH	$ kubectl edit deployment <deployment-name>
delete	DELETE	$ kubectl delete ingress <ig-name>

As you can see, CRUD verb names, HTTP method names, and `kubectl` sub-command names don't always match. For example, a `kubectl edit` command uses the `update` CRUD verb and the HTTP `PATCH` method.

A word on REST and RESTful

You'll hear the terms REST and RESTful a lot. REST is short for **RE**presentational **S**tate **T**ransfer and is the industry standard for communicating with web-based APIs. Systems that use REST, such as Kubernetes, are often referred to as *RESTful.*

REST requests comprise a *verb* and a *path* to a resource. Verbs relate to actions and map to the standard HTTP methods you saw in the previous table. Paths are URI paths to the resource in the API.

> **Terminology:** We often use the term *verb* to refer to CRUD operations as well as HTTP methods. Basically, any time we say *verb*, we're referring to an action.

The following example shows a `kubectl` command and associated REST request to list all Pods in the `shield` Namespace. The `kubectl` tool converts the command to the REST request shown — notice how the REST request has the verb and path we just mentioned.

```
$ kubectl get pods --namespace shield

GET /api/v1/namespaces/shield/pods
```

Hands-on

You'll need a copy of the book's GitHub repo, and you'll need to work on the `2025` branch.

```
$ git clone https://github.com/nigelpoulton/TKB.git
<Snip>

$ cd TKB

$ git fetch origin

$ git checkout -b 2025 origin/2025
```

Run the following command to start a `kubectl proxy` session. This exposes the API on your `localhost` adapter and handles all authentication. Feel free to use a different port.

```
$ kubectl proxy --port 9000 &
[1] 27533
Starting to serve on 127.0.0.1:9000
```

With the proxy running, you can use tools like `curl` to form API requests.

Run the following command to list all Pods in the `shield` Namespace. The command issues an HTTP `GET`, and the URI is the path to Pods in the `shield` Namespace.

```
$ curl -X GET http://localhost:9000/api/v1/namespaces/shield/pods
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "9524"
  },
```

```
  "items": []
}
```

The request returned an empty list because there are no Pods in the `shield` Namespace.

Try this next request to get a list of all the Namespaces on your cluster.

```
$ curl -X GET http://localhost:9000/api/v1/namespaces
{
  "kind": "NamespaceList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "9541"
  },
  "items": [
    {
      "metadata": {
        "name": "kube-system",
        "uid": "f5d39dd2-ccfe-4523-b634-f48ba3135663",
        "resourceVersion": "10",
<Snip>
```

As you learned earlier in the chapter, Kubernetes uses JSON as its preferred serialization schema. This means a command such as `kubectl get pods --namespace shield` will generate a request with the content type set to `application/json`. Assuming it's authenticated and authorized, it will result in **HTTP 200 (OK)** response code, and Kubernetes will respond with a serialized JSON list of all Pods in the `shield` Namespace.

Run one of the previous `curl` commands again, but add the `-v` flag to see the send and receive headers. I've trimmed the response to fit the page and draw your attention to the most important parts.

```
$ curl -v -X GET http://localhost:9000/api/v1/namespaces/shield/pods

> GET /api/v1/namespaces/shield/pods HTTP/1.1     <<---- HTTP GET method to REST path
of Pods
> Accept: */*                                     <<---- Accept all serialization
schemas
>
< HTTP/1.1 200 OK                                 <<---- Accepted request and
starting response
< Content-Type: application/json                  <<---- Responding using JSON
serialization
< X-Kubernetes-Pf-Flowschema-Uid: d50...
< X-Kubernetes-Pf-Prioritylevel-Uid: 828...
<
{                                                 <<---- Start of response
(serialized object)
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
```

```
    "resourceVersion": "34217"
  },
  "items": []
}
```

Lines starting with > are header data *sent* by `curl`. Lines starting with < are header data *returned* by the API server.

The > lines show `curl` sending a GET request to the `/api/v1/namespaces/shield/pods` REST path and telling the API server it can accept responses using any valid serialization schema (`Accept: */*`). The lines starting with < show the API server returning an HTTP response code and using JSON serialization. The `X-Kubernetes` lines are priority and fairness settings specific to Kubernetes.

A word on CRUD

CRUD is an acronym for the four basic functions web APIs use to manipulate and persist objects — **C**reate, **R**ead, **U**pdate, **D**elete. As previously mentioned, the Kubernetes API exposes and implements CRUD-style operations via the common HTTP methods.

Let's consider an example.

The following JSON is from the `ns.json` file in the `api` folder of the book's GitHub repo. It defines a new Namespace object called `shield`.

```
{
  "kind": "Namespace",
  "apiVersion": "v1",
  "metadata": {
    "name": "shield",
    "labels": {
      "chapter": "api"
    }
  }
}
```

You could create it now with the `kubectl apply -f ns.json` command, but I don't want you to do that. You'll create it in a later step.

However, if you did run the command, `kubectl` would form a request to the API server using the HTTP POST method. This is why you'll occasionally hear people say they're *POSTing* a configuration to the API server. The POST method creates a new object of the specified resource type. In this example, it would create a new Namespace called `shield`.

The following is a simplified example of what the request header would look like. The body will be the contents of the JSON file.

Request header:

```
POST https://<api-server>/api/v1/namespaces
Content-Type: application/json
Accept: application/json
```

If the request is successful, the response will include a standard HTTP response code, content type, and payload like the following:

```
HTTP/1.1 200 (OK)
Content-Type: application/json
{
    <payload>
}
```

Run the following `curl` command to post the `ns.json` file to the API server. It relies on you still having the `kubectl proxy` process running from earlier (`kubectl proxy --port 9000 &`), and you'll need to run the command from the `api` directory where the `ns.json` file exists. If the `shield` Namespace already exists, you'll need to delete it before continuing.

Windows users will need to replace the backslash with a backtick and place a backtick immediately before the `@` symbol.

```
$ curl -X POST -H "Content-Type: application/json" \
  --data-binary @ns.json http://localhost:9000/api/v1/namespaces

{
  "kind": "Namespace",
  "apiVersion": "v1",
  "metadata": {
    "name": "shield",
<Snip>
```

The `-X POST` argument forces `curl` to use the HTTP POST method. The `-H "Content-Type..."` tells the API server the request contains serialized JSON. The `--data-binary @ns.json` specifies the manifest file, and the URI is the address the API server is exposed on by `kubectl proxy` and includes the REST path for the resource.

Verify the new `shield` Namespace exists.

```
$ kubectl get ns
NAME            STATUS   AGE
kube-system     Active   47h
kube-public     Active   47h
```

```
kube-node-lease    Active    47h
default            Active    47h
shield             Active    14s
```

Now delete the Namespace by running a curl command specifying the DELETE HTTP method.

```
$ curl -X DELETE \
  -H "Content-Type: application/json" http://localhost:9000/api/v1/namespaces/shield
{
  "kind": "Namespace",
  "apiVersion": "v1",
  "metadata": {
    "name": "shield",
    <Snip>
  },
  "spec": {
    "finalizers": [
      "kubernetes"
    ]
  },
  "status": {
    "phase": "Terminating"
  }
}
```

In summary, the **API server** exposes the API over a secure RESTful interface that lets you manipulate and query the state of objects on the cluster. It runs on the control plane, which needs to be highly available and have enough performance to service requests quickly.

The API

The API is where all Kubernetes resources are defined. It's large, modular, and RESTful.

When Kubernetes was new, the API was monolithic and all resources existed in a single global namespace. However, as Kubernetes grew, we split the API into smaller, more manageable groups that we call *named groups* or *sub-groups.*

Figure 15.3 shows a simplified view of the API with resources divided into groups.
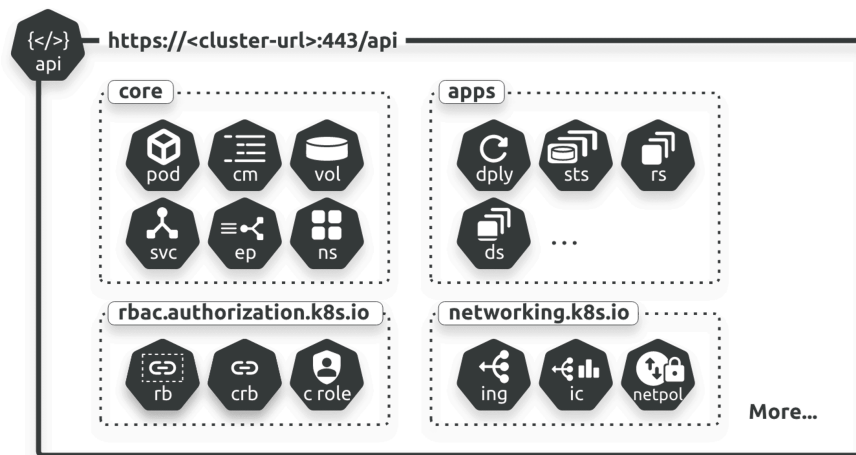
Figure 15.3 - Simplified view of Kubernetes API

The image shows the API with four groups. There are lots more than four, but I'm keeping the picture simple.

There are two types of API group:

- The **core** group
- The **named** groups

The core API group

The core group is where we define all the original objects from when Kubernetes was new (before it grew and we divided the API into groups). Some of the resources in this group include Pods, Nodes, Services, Secrets, and ServiceAccounts, and you can find them in the API below the `/api/v1` REST path. The following table lists some example paths for resources in the core group.

Resource	REST Path
Pods	/api/v1/namespaces/{namespace}/pods/
Services	/api/v1/namespaces/{namespace}/services/
Nodes	/api/v1/nodes/
Namespaces	/api/v1/namespaces/

Notice how some objects are namespaced and some aren't. Namespaced objects have longer REST paths as you have to include two additional segments — `../namespaces/{namespace}/...` For example, listing all Pods in the `shield` Namespace requires the following path.

```
GET /api/v1/namespaces/shield/pods/
```

Objects, such as nodes, that aren't namespaced have much shorter REST paths.

```
GET /api/v1/nodes/
```

Expected HTTP response codes for read requests are **200: OK** or **401: Unauthorized**.

On the topic of REST paths, **GVR** stands for **g**roup, **v**ersion, and **r**esource, and can be a good way to remember the structure of REST paths. Figure 15.4 shows the REST path to the `v1 StorageClasses` resource in the `storage.k8s.io` named group.
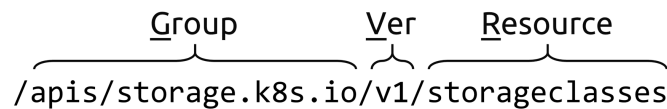
Figure 15.4

You shouldn't expect any new resources to be added to the core group.

Named API groups

The *named API groups* are where we add all new resources, and we sometimes call them *sub-groups*.

Each of the named groups is a collection of related resources. For example, the **apps** group defines resources such as Deployments, StatefulSets, and DaemonSets that manage **app**lication workloads. Likewise, we define Ingresses, Ingress Classes, and Network Policies in the `networking.k8s.io` group. Notable exceptions to this pattern are older resources in the core group that came along before the named groups existed. For example, Pods and Services are both in the core group. However, if we invented them today, we'd probably put Services in the `networking.k8s.io` group and Pods in the **apps** group.

Resources in the named groups live below the `/apis/{group-name}/{version}/` REST path. The following table lists some examples.

Resource	Path
Ingress	/apis/networking.k8s.io/v1/namespaces/{namespace}/ingresses/
ClusterRole	/apis/rbac.authorization.k8s.io/v1/clusterroles/
StorageClass	/apis/storage.k8s.io/v1/storageclasses/

Notice how the URI paths for named groups start with `/apis` (plural) and include the name of the group. This differs from the core group that starts with `/api` (singular) and doesn't include a group name. In fact, in some places, you'll see the core API group referred to by empty double quotes (""). This is because no thought was given to groups when we originally created the API — everything was *"just in the API".*

Dividing the API into smaller groups makes it more scalable and easier to navigate and extend.

Inspecting the API

The following commands are good ways for you to see API-related info.

The `kubectl api-resources` command lists all the API resources and groups your cluster supports. It also shows resource shortnames and whether they are namespaced or cluster-scoped. I've tweaked the output to fit the page and show a mix of resources from different groups.

```
$ kubectl api-resources
NAME                      SHORT   APIVERSION            NAMESPACED   KIND
namespaces                ns      v1                    false        Namespace
nodes                     no      v1                    false        Node
pods                      po      v1                    true         Pod
deployments               deploy  apps/v1               true         Deployment
replicasets               rs      apps/v1               true         ReplicaSet
statefulsets              sts     apps/v1               true         StatefulSet
cronjobs                  cj      batch/v1              true         CronJob
jobs                              batch/v1              true         Job
horizontalpodautoscalers  hpa     autoscaling/v2        true
HorizontalPodAutoscaler
ingresses                 ing     networking.k8s.io/v1  true         Ingress
networkpolicies           netpol  networking.k8s.io/v1  true         NetworkPolicy
storageclasses            sc      storage.k8s.io/v1     false        StorageClass
```

The next command shows which API versions your cluster supports. It doesn't list which resources belong to which APIs, but it's good for finding out whether your cluster has things like **alpha** APIs enabled. Notice how some API groups have multiple versions enabled, such as beta and stable, or v1 and v2.

```
$ kubectl api-versions
admissionregistration.k8s.io/v1
```

```
apiextensions.k8s.io/v1
apps/v1
<Snip>
autoscaling/v1
autoscaling/v2
v1
```

The next command is more complicated and only lists the `kind` and `version` fields for supported resources. It doesn't work on Windows.

```
$ for kind in `kubectl api-resources | tail +2 | awk '{ print $1 }'`; \
 do kubectl explain $kind; done | grep -e "KIND:" -e "VERSION:"

KIND:     Binding
VERSION:  v1
KIND:     ComponentStatus
VERSION:  v1
<Snip>
KIND:     HorizontalPodAutoscaler
VERSION:  autoscaling/v2
KIND:     CronJob
VERSION:  batch/v1
KIND:     Job
VERSION:  batch/v1
<Snip>
```

You can run the following commands if your `kubectl proxy` process is still running.

Run the following command to list all API versions available below the `core` API group. You should only see the `v1` version.

```
$ curl http://localhost:9000/api
{
  "kind": "APIVersions",
  "versions": [
    "v1"                    <<---- v1 version
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "172.21.0.4:6443"
    }
  ]
}
```

Run this command to list all named APIs and groups. I've trimmed the output to save space.

```
$ curl http://localhost:9000/apis
{
  "kind": "APIGroupList",
  "apiVersion": "v1",
  "groups": [
```

```
     <Snip>
     {
       "name": "apps",
       "versions": [
         {
           "groupVersion": "apps/v1",
           "version": "v1"
         }
       ],
       "preferredVersion": {
         "groupVersion": "apps/v1",
         "version": "v1"
       }
     },
     <Snip>
```

You can list specific object instances or lists of objects on your cluster. The
following command returns a list of all Namespaces.

```
$ curl http://localhost:9000/api/v1/namespaces
{
  "kind": "NamespaceList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "35234"
  },
  "items": [
    {
      "metadata": {
        "name": "kube-system",
        "uid": "05fefa13-cbec-458b-aece-d65eb1972dfb",
        "resourceVersion": "4",
        "creationTimestamp": "2025-02-12T09:59:42Z",
        "labels": {
          "kubernetes.io/metadata.name": "kube-system"
        },
        "managedFields": [
          {
            "manager": "Go-http-client",
            "operation": "Update",
            "apiVersion": "v1",
<Snip>
```

Feel free to poke around. You can put the same URI paths into a browser and
API tools like Postman.

Leave the `kubectl proxy` process running, as you'll use it again later in the
chapter.

Alpha beta and stable

Kubernetes has a well-documented process for accepting new API resources.
They come in as *alpha*, progress through *beta*, and eventually graduate as

Generally Available (GA). We sometimes refer to GA as *stable*.

API version	Track
Alpha	Experimental
Beta	Pre-release
GA (Generally Available))	Stable

Alpha resources are experimental and you should consider them *hairy and scary*. You should expect them to have bugs, drop features without warning, and change a lot when they move into beta. This is why most clusters disable them by default.

As a quick example, a new resource called **tkb** in the **apps** API group that goes through two alpha versions will have the following API names:

- /apis/apps/**v1alpha1**/tkb
- /apis/apps/**v1alpha2**/tkb

After alpha, it goes through beta testing.

Beta resources are considered *pre-release* and should be very close to what the developers expect the final GA release to look like. However, it's normal to expect minor changes when promoted to GA. Most clusters enable beta APIs by default, and you'll occasionally see beta resources in production environments. However, that's not a recommendation. You need to make those decisions yourself.

If you put the same **tkb** resource through two beta versions, Kubernetes will serve them via the following APIs:

- /apis/apps/**v1beta1**/tkb
- /apis/apps/**v1beta2**/tkb

The final phase after beta is *Generally Available (GA)*, sometimes referred to as *stable*.

GA resources are considered production-ready, and Kubernetes has a strong long-term commitment to them.

Most GA resources are `v1`. However, some have continued to evolve and progressed to `v2`. When you create a `v2` resource, you put it through the exact same incubation and graduation process. For example, the same **tkb** resource

in the **apps** API would go through the same alpha and beta process before reaching `v2`:

- /apis/apps/**v2alpha1**/tkb
- <Snip>
- /apis/apps/**v2beta1**/tkb
- <Snip>
- /apis/apps/**v2**/tkb

Real-world examples of paths to stable resources include the following:

- /apis/networking.k8s.io/**v1**/ingresses
- /apis/batch/**v1**/cronjobs
- /apis/autoscaling/**v2**/horizontalpodautoscalers

You can *deploy* an object via one API, and then read it back and manage it using a more recent API. For example, you can deploy an object via a `v1beta2` API and then update and manage it later through the stable `v1` API.

Resource deprecation

As previously mentioned, alpha and beta objects can experience a lot of changes before promotion to GA. However, GA objects don't change, and Kubernetes is strongly committed to maintaining long-term usability and support.

At the time of writing, Kubernetes has the following commitments to beta and GA resources:

- **Beta:** Resources in beta have a 9-month window to either release a newer beta version or graduate to GA. This is to prevent resources from stagnating in beta. For example, the Ingress resource remained in beta for over 15 Kubernetes releases!
- **GA:** GA resources are expected to be long-lived. When deprecated, Kubernetes continues to serve and support them for 12 months or three releases, whichever is longest. After this period they are removed. However, Kubernetes will only deprecate an existing stable resource after a newer stable version is available. For example, it will only deprecate a v1 resource if the v2 of the same resource is already released.

Recent versions of Kubernetes do three things when you deploy a deprecated resource:

1. Return a deprecation warning message on the CLI
2. Add a `k8s.io/deprecated:true` annotation to the audit record for the request
3. Set an `apiserver_requested_deprecated_apis` gauge metric

Deprecation warning messages on the CLI give you immediate feedback that you've used a deprecated API. The other two allow you to query audit logs and process logs to determine if you're using deprecated APIs. These last two can be useful when planning Kubernetes upgrades as they help you to know if you're using deprecated resources.

Extending the API

Kubernetes ships with a collection of built-in controllers that deploy and manage built-in resources. However, you can extend Kubernetes by adding your own resources and controllers.

This is a popular way for network and storage vendors to expose advanced features, such as snapshot schedules or IP address management, via the Kubernetes API. In the storage example, volumes are surfaced inside of Kubernetes via CSI drivers, Pods consume them via built-in Kubernetes resources such as StorageClasses and PersistentVolumeClaims, but advanced features such as snapshot scheduling can be managed via custom API resources and controllers. This allows developers and Kubernetes operators to deploy and manage everything via well-understood API interfaces and standard tools such as `kubectl` and YAML files.

The high-level pattern for extending the API involves two main things:

- Create your custom *resource*
- Write and deploy your custom *controller*

Kubernetes has a CustomResourceDefinition (CRD) object that lets you create new API resources that look, smell, and feel like native Kubernetes resources. You create your custom resource as a CRD and then use `kubectl` to create instances and inspect them just like you do with native resources. Your custom resources even get their own REST paths in the API.

The following YAML is from the `crd.yml` file in the `api` folder. It defines a new cluster-scoped custom resource called **books** in the `nigelpoulton.com` API group served via the `v1` path.

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: books.nigelpoulton.com
spec:
  group: nigelpoulton.com       <<---- API sub-group (a.k.a. "named API group")
  scope: Cluster                <<---- Can be "Namespaced" or "Cluster"
  names:
    plural: books               <<---- All resources need a plural and singular name
    singular: book              <<---- Singular names are used on CLI and command
outputs
    kind: Book                  <<---- kind property used in YAML files
    shortNames:
    - bk                        <<---- Short name that can be used by kubectl
  versions:                     <<---- Resources can be served by multiple API
versions
    - name: v1
      served: true              <<---- If set to false, "v1" will not be served
      storage: true             <<---- Store instances of the object as this version
      schema:                   <<---- This block defines the resource's properties
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                <Snip>
```

If you haven't already done so, run the following commands to clone the book's GitHub repo and switch to the `2025` branch.

```
$ git clone https://github.com/nigelpoulton/TKB.git
<Snip>

$ cd TKB

$ git fetch origin

$ git checkout -b 2025 origin/2025
```

Change into the `api` directory.

```
$ cd TKB/api
```

Run the following command to deploy the custom resource.

```
$ kubectl apply -f crd.yml
customresourcedefinition.apiextensions.k8s.io/books.nigelpoulton.com created
```

Congratulations, the new resource exists in the API and Kubernetes is serving it on the following REST path.

```
apis/nigelpoulton.com/v1/books/
```

Verify it exists in the API. Replace the `grep books` argument with `Select-String -Pattern 'books'` if you're using Windows.

```
$ kubectl api-resources | grep books
NAME        SHORTNAMES    APIGROUP              NAMESPACED    KIND
books       bk            nigelpoulton.com/v1   false         Book

$ kubectl explain book
GROUP:      nigelpoulton.com
KIND:       Book
VERSION:    v1
DESCRIPTION:
    <empty>
FIELDS:
    <Snip>
```

The following YAML is from the `book.yml` file and defines a new *Book* object called **ai**. Notice how the fields in the `spec` section match the names and types defined in the custom resource.

```
apiVersion: nigelpoulton.com/v1
kind: Book
metadata:
  name: ai
spec:
  bookTitle: "AI Explained"
  subTitle: "Facts, Fiction, and Future"
  topic: "Artificial Intelligence"
  edition: 1
  salesUrl: https://www.amazon.com/dp/1916585388
```

Deploy it with the following command.

```
$ kubectl apply -f book.yml
book.nigelpoulton.com/ai created
```

You can now list and describe it with the usual commands. The following command uses the resource's `bk` shortname.

```
$ kubectl get bk
NAME    TITLE           SUBTITLE                    EDITION    URL
ai      AI Explained    Facts, Fiction, and Future   1
www.amazon.com/dp/1916585388
```

You can also use tools like `curl` to query the new API group and resource.

The following commands start a `kubectl proxy` process and then list all resources under the new `nigelpoulton.com` named group. You don't need to start another proxy if you're still running the one from earlier in the chapter.

```
$ kubectl proxy --port 9000 &
[1] 14784
Starting to serve on 127.0.0.1:9000

$ curl http://localhost:9000/apis/nigelpoulton.com/v1/
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "nigelpoulton.com/v1",
  "resources": [
    {
      "name": "books",
      "singularName": "book",
      "namespaced": false,
      "kind": "Book",
      "verbs": [
        "delete",
        "deletecollection",
        "get",
        "list",
        "patch",
        "create",
        "update",
        "watch"
      ],
      "shortNames": [
        "bk"
      ],
      "storageVersionHash": "F2QdXaP5vh4="
    }
  ]
}
```

This is all good and interesting. However, custom resources don't do anything useful until you create a custom controller to do something with them. Writing your own controllers is beyond the scope of this chapter, but you've learned a lot about the Kubernetes API and how it works.

Clean up

If you've been following along, you'll have all the following resources that need cleaning up:

- A `kubectl proxy` process
- An **ai** book resource
- A `books.nigelpoulton.com` custom resource (CRD)

Run one of the following commands to get the process ID (PID) of your `kubectl proxy` process.

```
// Linux and Mac command
```

```
$ ps | grep kubectl
PID     TTY        TIME      CMD
27533   ttys001   0:03.13   kubectl proxy --port 9000

// Windows command

> tasklist | Select-String -Pattern 'kubectl'
Image Name       PID   Session Name   Session#
=============  =====  ============  ========
kubectl.exe    19776  Console              1
```

Run one of the following commands to kill it, and remember to use the PID from your system.

```
// Linux and Mac command

$ kill -9 27533
[1]  + 27533 killed     kubectl proxy --port 9000

// Windows command

> taskkill /F /PID 19776
SUCCESS: The process with PID 19776 has been terminated.
```

Run the following command to delete the **ai** book object.

```
$ kubectl delete book ai
book.nigelpoulton.com "ai" deleted
```

Now delete the `books.nigelpoulton.com` custom resource.

```
$ kubectl delete crd books.nigelpoulton.com
customresourcedefinition.apiextensions.k8s.io "books.nigelpoulton.com" deleted
```

Chapter summary

Now that you've read the chapter, all of the following should make sense. But don't worry if some of it is still confusing. APIs can be hard to understand, and the Kubernetes API is large and complex.

Anyway, here goes…

Kubernetes is an API-driven platform, and the API is exposed internally and externally via the API server.

The API server runs as a control plane service, and all internal and external clients interact with the API via the API server. This means your control plane needs to be highly available and high-performance. If it's not, you risk slow API responses or entirely losing access to the API.

The Kubernetes API is a modern resource-based RESTful API that accepts CRUD-style operations via uniform HTTP methods such as POST, GET, PUT, PATCH, and DELETE. It's divided into named groups for convenience and extensibility. Older resources created in the early days of Kubernetes exist in the original `core` group, which you access via the `/api/v1` REST path. All newer objects go into named groups. For example, we define newer network resources in the `networking.k8s.io` sub-group available at the `/apis/networking.k8s.io/v1/` REST path.

Most of the resources in the Kubernetes API are *objects*, so we sometimes use the terms *resources* and *objects* to mean the same thing. It's common to refer to their API definitions as resources or resource definitions, whereas running instances on a cluster are often referred to as objects. For example, "The Pod *resource* exists in the core API group, and there are five Pod *objects* running in the default Namespace."

All new resources enter the API as alpha, progress through beta, and hopefully graduate to GA. Alpha resources are considered *experimental,* and, therefore, subject to change and usually disabled on most clusters. Beta resources are considered *pre-release* and, therefore, more reliable and won't change much when they graduate to GA. Most clusters enable beta resources by default, but you should be cautious about using them in production. GA resources are considered *stable* and production-ready. Kubernetes has a strong commitment to GA resources and backs this up with a well-defined deprecation policy guaranteeing support for at least 12 months, or three versions, after the deprecation announcement.

Finally, the Kubernetes API is becoming the de facto cloud API, with many third-party technologies extending it so they can expose their own technologies through it. Kubernetes makes it easy to extend the API through CustomResourceDefinitions that make 3rd-party resources look and feel like native Kubernetes resources.

Hopefully, that made sense, but don't worry if you're still unsure about some of it. I recommend you play around with as many of the examples as possible. You should also consider reading the chapter again tomorrow — it's normal for new concepts to take a while to learn.

If you liked this or any other chapter, jump over to Amazon and show the book some love with a quick review. The cloud-native gods will smile on you ;-)

16: Threat modeling Kubernetes

Security is more important than ever, and Kubernetes is no exception. Fortunately, there's a lot you can do to secure Kubernetes, and you'll see some ways in the next chapter. However, before doing that, it's a good idea to model some of the common threats.

Threat modeling

Threat modeling is the process of identifying vulnerabilities so you can put measures in place to prevent and mitigate them. This chapter introduces the popular *STRIDE* model and shows how you can apply it to Kubernetes.

STRIDE defines six potential threat categories:

- Spoofing
- Tampering
- Repudiation
- Information disclosure
- Denial of service
- Elevation of privilege

While the model is good and provides a structured way to assess things, no model guarantees to cover all threats.

For the rest of this chapter, we'll look at each of the six threat categories. For each one, we'll give a quick description and then look at some of the ways it applies to Kubernetes.

The chapter doesn't try to cover everything. The goal is to give you ideas and get you started.

Spoofing

Spoofing is pretending to be somebody else with the aim of gaining extra privileges.

Let's look at some of the ways Kubernetes prevents different types of spoofing.

Securing communications with the API server

Kubernetes comprises lots of small components that work together. These include the API server, controller manager, scheduler, cluster store, and others. It also includes node components such as the kubelet and container runtime. Each has its own privileges that allow it to interact with and modify the cluster. Even though Kubernetes implements a least-privilege model, spoofing the identity of any of these can cause problems.

If you read the RBAC and API security chapter, you'll know that Kubernetes requires all components to authenticate via cryptographically signed certificates (mTLS). This is good, and Kubernetes makes it easy by automatically rotating certificates. However, you must consider the following:

1. A typical Kubernetes installation auto-generates a self-signed certificate authority (CA) that issues certificates to all cluster components. While this is better than nothing, it's not enough for production environments on its own.
2. Mutual TLS (mTLS) is only as secure as the CA issuing the certificates. Compromising the CA can render the entire mTLS layer ineffective. With this in mind, it's vital you keep the CA secure!

A good practice is to ensure that certificates issued by the internal Kubernetes CA are only used and trusted *within* the Kubernetes cluster. This requires careful approval of certificate signing requests,

as well as ensuring the Kubernetes CA doesn't get added as a trusted CA for any systems outside the cluster.

As mentioned in previous chapters, all internal and external requests to the API server are subject to authentication and authorization checks. As a result, the API server needs a way to authenticate (trust) internal and external sources. A good way to do this is to have **two** trusted key pairs:

- One for authenticating internal systems
- A second for authenticating external systems

In this model, you'd use the cluster's self-signed CA to issue keys to internal systems. You'd then configure Kubernetes to trust one or more trusted 3rd-party CAs for external systems.

Securing Pod communications

As well as spoofing access to the cluster, there's also the threat of spoofing app-to-app communications. In Kubernetes, this can be when one Pod spoofs another. Fortunately, Pods can have certificates to authenticate their identity.

Every Pod has an associated *ServiceAccount* that is used to provide an identity for the Pod. This is achieved by automatically mounting a service account token into every Pod as a *Secret*. Two points to note:

1. The service account token allows access to the API server
2. Most Pods probably don't need to access the API server

With these two points in mind, you should set `automountServiceAccountToken` to `false` for Pods that don't need to communicate with the API server. The following Pod manifest shows how to do this.

```
apiVersion: v1
kind: Pod
metadata:
  name: service-account-example-pod
spec:
  serviceAccountName: some-service-account
  automountServiceAccountToken: false        <<---- This line
  <Snip>
```

If the Pod does need to talk to the API server, the following non-default configurations are worth exploring:

- **expirationSeconds**
- **audience**

These let you force a time when the token will expire and restrict the entities it works with. The following example, inspired from the official Kubernetes docs, sets an expiry period of one hour and restricts it to the **vault** audience in a projected volume.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /var/run/secrets/tokens
      name: vault-token
  serviceAccountName: my-pod
  volumes:
  - name: vault-token
    projected:
      sources:
      - serviceAccountToken:
          path: vault-token
          expirationSeconds: 3600    <<---- This line
          audience: vault            <<---- And this one
```

Tampering

Tampering is the act of changing something in a malicious way to cause one of the following:

- **Denial of service**: Tampering with the resource to make it unusable
- **Elevation of privilege**: Tampering with a resource to gain additional privileges

Tampering can be hard to avoid, so a common countermeasure is to make it obvious when something has been tampered with. A common non-Kubernetes example is packaging medication — most over-the-counter drugs are packaged with tamper-proof seals that make it obvious if the product has been tampered with.

Tampering with Kubernetes components

Tampering with any of the following Kubernetes components can cause problems:

- etcd
- Configuration files for the API server, controller-manager, scheduler, etcd, and kubelet
- Container runtime binaries
- Container images
- Kubernetes binaries

Generally speaking, tampering happens either *in transit* or *at rest*. In transit refers to data while it is being transmitted over the network, whereas at rest refers to data stored in memory or on disk.

TLS is a great tool for protecting against *in-transit* tampering as it provides built-in integrity guarantees that warn you when data has been tampered with.

The following recommendations can also help prevent tampering with data when it is *at rest* in Kubernetes:

- Restrict access to the servers that are running Kubernetes components, especially control plane components
- Restrict access to repositories that store Kubernetes configuration files
- Only perform remote bootstrapping over SSH (remember to keep your SSH keys safe)
- Always run SHA-2 checksums against downloads
- Restrict access to your image registry and associated repositories

This isn't an exhaustive list. However, implementing it will significantly reduce the chances of your data being tampered with while at rest.

As well as the items listed, it's good production hygiene to configure auditing and alerting for important binaries and configuration files. If configured and monitored correctly, these can help detect potential tampering attacks.

The following example uses a common Linux audit daemon to audit access to the `docker` binary. It also audits attempts to change the binary's file attributes.

```
$ auditctl -w /usr/bin/docker -p wxa -k audit-docker
```

We'll refer to this example later in the chapter.

Tampering with applications running on Kubernetes

Malicious actors will also target application components, as well as infrastructure components.

A good way to prevent a live Pod from being tampered with is setting its filesystems to *read-only*. This guarantees filesystem

immutability and you can configure it via the `securityContext` section of a Pod manifest file.

You can make a container's root filesystem read-only by setting the `readOnlyRootFilesystem` property to `true`. You can do the same for other container filesystems via the `allowedHostPaths` property.

The following YAML shows how to configure both settings in a Pod manifest. In the example, the `allowedHostPaths` section makes sure anything mounted beneath `/test` will be read-only.

```
apiVersion: v1
kind: Pod
metadata:
  name: readonly-test
spec:
  securityContext:
    readOnlyRootFilesystem: true      <<---- R/O root filesystem
    allowedHostPaths:                 <<---- Make anything below
      - pathPrefix: "/test"           <<---- this mount point
        readOnly: true                <<---- read-only (R/O)
<Snip>
```

Repudiation

At a very high level, *repudiation* creates doubt about something. *Non-repudiation* provides proof about something. In the context of information security, non-repudiation is **proving** certain individuals carried out certain actions.

Digging a little deeper, non-repudiation includes the ability to prove:

- What happened
- When it happened
- Who made it happen
- Where it happened
- Why it happened
- How it happened

Answering the last two can be the hardest and usually requires the correlation of several events over a period of time.

Auditing Kubernetes API server events can help answer these questions. The following is an example of an API server audit event (you may need to enable auditing on your API server).

```
{
  "kind":"Event",
  "apiVersion":"audit.k8s.io/v1",
  "metadata":{ "creationTimestamp":"2022-11-11T10:10:00Z" },
  "level":"Metadata",
  "timestamp":"2022-11-11T10:10:00Z",
  "auditID":"7e0cbccf-8d8a-4f5f-aefb-60b8af2d2ad5",
  "stage":"RequestReceived",
  "requestURI":"/api/v1/namespaces/default/persistentvolumeclaims",
  "verb":"list",
  "user": {
    "username":"fname.lname@example.com",
    "groups":[ "system:authenticated" ]
  },
  "sourceIPs":[ "123.45.67.123" ],
  "objectRef": {
    "resource":"persistentvolumeclaims",
    "namespace":"default",
    "apiVersion":"v1"
  },
  "requestReceivedTimestamp":"2022-11-11T10:10:00.123456Z",
  "stageTimestamp":"2022-11-11T10:10:00.123456Z"
}
```

The API server isn't the only component you should audit for non-repudiation. At a minimum, you should collect audit logs from container runtimes, kubelets, and the applications running on your cluster. You should also audit non-Kubernetes infrastructure, such as network firewalls.

As soon as you start auditing multiple components, you'll need a centralized location to store and correlate events. A common way to do this is deploying an agent to all nodes via a DaemonSet. The agent collects logs (runtime, kubelet, application, etc) and ships them to a secure central location.

If you do this, the centralized log store must be secure. If it isn't, you won't be able to trust the logs, and their contents can be *repudiated*.

To provide non-repudiation relative to tampering with binaries and configuration files, it might be useful to use an audit daemon that watches for write actions on certain files and directories on your Kubernetes control plane nodes and worker nodes. For example, earlier in the chapter you saw a way to enable auditing of changes to the `docker` binary. With this enabled, starting a new container with the `docker run` command will generate an event like this:

```
type=SYSCALL msg=audit(1234567890.123:12345): arch=abc123 syscall=59
success=yes \
exit=0 a0=12345678abca1=0 a2=abc12345678 a3=a items=1 ppid=1234
pid=12345 auid=0 \
uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=pts0 ses=1
comm="docker" \
exe="/usr/bin/docker"
subj=system_u:object_r:container_runtime_exec_t:s0 \
key="audit-docker" type=CWD msg=audit(1234567890.123:12345):
cwd="/home/firstname"\
type=PATH msg=audit(1234567890.123:12345): item=0
name="/usr/bin/docker"\
 inode=123456 dev=fd:00 mode=0100600 ouid=0 ogid=0 rdev=00:00...
```

When combined and correlated with Kubernetes' audit features, audit logs like this create a comprehensive and trustworthy picture that cannot be repudiated.

Information Disclosure

Information disclosure is when sensitive data is leaked. Common examples include hacked data stores and APIs that unintentionally expose sensitive data.

Protecting cluster data

The entire configuration of a Kubernetes cluster is stored in the cluster store (usually etcd). This includes network and storage configuration, passwords, the cluster CA, and more. This makes the cluster store a prime target for information disclosure attacks.

As a minimum, you should limit **and** audit access to the nodes hosting the cluster store. As you'll see in the next paragraph, gaining access to a cluster node can allow the logged-on user to bypass some security layers.

Kubernetes 1.7 introduced encryption of Secrets but doesn't enable it by default. Even when this becomes the default, the *data encryption key (DEK)* is stored on the same node as the Secret! This means gaining access to a node lets you to bypass encryption. This is especially worrying on nodes that host the cluster store (etcd nodes).

Kubernetes 1.11 shipped with Key Management Service (KMS v1) as a beta feature allowing you to store *key encryption keys (KEK)* outside of your Kubernetes cluster. KEKs are used to encrypt and decrypt data encryption keys (DEK) and should be safely guarded. We call this approach "envelope encryption".

Since then, we've entirely redesigned KMS and replaced the old KMS v1 with KMS v2 which has been generally available (GA) since Kubernetes v1.29. It's a total redesign with many improvements, including:

- Generating a new DEK for each encryption — a new DEK is generated for every *write* operation and encrypted at rest with KEK
- The cache size for KMS v2 plugins is only limited by underlying control plane node resources. This allows previous DEKs to be

cached in memory for faster encryption and decryption operation
- Zero downtime re-encryption operations — Re-encryption of secrets after key rotations no longer requires API server downtime or Pod restarts
- KMS plugins run on control plane nodes and communicate with the API server via Unix sockets. KMS plugins then connect to an external KMS providers to handle the encryption and decryption of Data Encryption Keys (DEKs).

All of this means that attackers now need to compromise the Kubernetes control plane as well as the external KMS. A snapshot of a control plane node is not enough to read a Secret in plain text. For this reason, you should seriously consider storing your KEKs in Hardware Security Modules (HSM) or cloud-based Key Management Stores (KMS). Also, as with all things cryptography, you should follow the *trust but verify* model and ensure that the KMS v2 plugin based encryption is working as expected and plan for contingencies such as a lost KEK or loss of connectivity to external KMS provider.

Protecting data in Pods

As previously mentioned, Kubernetes has an API resource called a Secret that is the preferred way to store and share sensitive data such as passwords. For example, a front-end container accessing an encrypted back-end database can have the key to decrypt the database mounted as a Secret. This is far better than storing the decryption key in a plain-text file or environment variable.

It is also common to store data and configuration information outside of Pods and containers in Persistent Volumes and ConfigMaps. If the data on these is encrypted, you should store the keys for decrypting them in Secrets.

Despite all of this, you must consider the caveats outlined in the previous section relative to Secrets and how their encryption keys

are stored. You don't want to do the hard work of locking the house but leaving the keys in the door.

Denial of Service

Denial of Service (DoS) is about making something unavailable.

There are many types of DoS attacks, but a well-known variation is overloading a system to the point it can no longer service requests. In the Kubernetes world, a potential attack might be overloading the API server so that cluster operations grind to a halt (even internal systems use the API server to communicate).

Let's look at some potential Kubernetes systems that might be targets of DoS attacks, as well as some ways to protect and mitigate them.

Protecting cluster resources against DoS attacks

It's a time-honored best practice to replicate essential services on multiple nodes for high availability (HA). Kubernetes is no different, and you should run multiple control plane nodes in an HA configuration for your production environments. Doing this prevents any control plane node from becoming a single point of failure. In relation to certain types of DoS attacks, an attacker may need to attack more than one control plane node to have a meaningful impact.

You should also replicate control plane nodes across availability zones. This may prevent a DoS attack on the network of a particular availability zone from taking down your entire control plane.

The same principle applies to worker nodes. Having multiple worker nodes not only allows the scheduler to spread your applications over multiple availability zones, but it may also render DoS attacks on any single node or zone ineffective (or less effective).

You should also configure appropriate limits for the following:

- Memory
- CPU
- Storage

Limits like these can help prevent essential system resources from being starved, therefore preventing potential DoS.

Limiting *Kubernetes objects* can also be a good practice. This includes limiting things such as the number of ReplicaSets, Pods, Services, Secrets, and ConfigMaps in a particular Namespace.

Here's an example manifest that limits the number of Pod objects in the `skippy` Namespace to 100.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pod-quota
  namespace: skippy
spec:
  hard:
    pods: "100"
```

One more feature — `podPidsLimit` — restricts the number of processes a Pod can create.

Assume a Pod is the target of a fork bomb attack where a rogue process attempts to bring the system down by creating enough processes to consume all system resources. If you've configured the Pod with `podPidsLimit` to restrict the number of processes the Pod can create, you'll prevent it from exhausting the node's resources and confine the attack's impact to the Pod. Kubernetes will normally restart a Pod if it exhausts its `podPidsLimit.`

This also ensures a single Pod doesn't exhaust the PID range for all the other Pods on the node, including the kubelet. However, setting the correct value requires a reasonable estimate of how many Pods

will run simultaneously on each node, and you can easily over or under-allocate PIDs to each pod without a ballpark estimate.

Protecting the API Server against DoS attacks

The API server exposes a RESTful interface over a TCP socket. This makes it a target for botnet-based DoS attacks.

The following may be helpful in either preventing or mitigating such attacks:

- Highly available control plane nodes — multiple replicas of the API server running on multiple nodes across multiple availability zones
- Monitoring and alerting on API server requests based on sane thresholds
- Using things like firewalls to limit API server exposure to the internet

As well as botnet DoS attacks, an attacker may also attempt to spoof a user or other control plane service to cause an overload. Fortunately, Kubernetes has robust authentication and authorization controls to prevent spoofing. However, even with a robust RBAC model, you must safeguard access to accounts with high privileges.

Protecting the cluster store against DoS attacks

Kubernetes stores cluster configuration in etcd. This makes it vital that etcd be available and secure. The following recommendations help accomplish this:

- Configure an HA etcd cluster with either 3 or 5 nodes
- Configure monitoring and alerting of requests to etcd
- Isolate etcd at the network level so that only members of the control plane can interact with it

A default installation of Kubernetes installs etcd on the same servers as the rest of the control plane. This is fine for development and testing. However, large production clusters should seriously consider a dedicated etcd cluster. This will provide better performance and greater resilience.

On the performance front, etcd is the most common choking point for large Kubernetes clusters. With this in mind, you should perform testing to ensure the infrastructure it runs on is capable of sustaining performance at scale — a poorly performing etcd can be as bad as an etcd cluster under a sustained DoS attack. Operating a dedicated etcd cluster also provides additional resilience by protecting it from other parts of the control plane that might be compromised.

Monitoring and alerting of etcd should be based on sane thresholds, and a good place to start is by monitoring etcd log entries.

Protecting application components against DoS attacks

Most Pods expose their main service on the network, and without additional controls in place, anyone with access to the network can perform a DoS attack on the Pod. Fortunately, Kubernetes provides Pod resource request limits to prevent such attacks from exhausting Pod and node resources. As well as these, the following will be helpful:

- Define Kubernetes Network Policies to restrict Pod-to-Pod and Pod-to-external communications
- Utilize mutual TLS and API token-based authentication for application-level authentication (reject any unauthenticated requests)

For defense in depth, you should also implement application-layer authorization policies that implement the least privilege.

Figure 16.1 shows how these can be combined to make it hard for an attacker to successfully DoS an application.
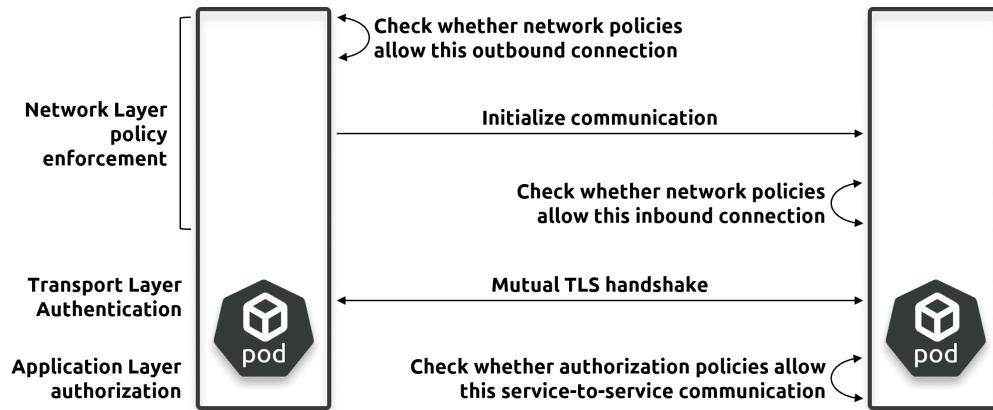

Figure 16.1

Elevation of privilege

Privilege escalation is gaining higher access than what is granted. The aim is to cause damage or gain unauthorized access.

Let's look at a few ways to prevent this in a Kubernetes environment.

Protecting the API server

Kubernetes offers several authorization modes that help safeguard access to the API server. These include:

- Role-based Access Control (RBAC)
- Webhook
- Node

You should run multiple authorizers at the same time. For example, it's common to use the *RBAC* and *node* authorizers.

RBAC mode lets you restrict API operations to sub-sets of users. These *users* can be regular user accounts or system services. The idea is that all requests to the API server must be authenticated **and** authorized. Authentication ensures that requests come from a validated user, whereas authorization ensures the validated user can perform the requested operation. For example, can *Mia create Pods*? In this example, *Mia* is the user, *create* is the operation, and *Pods* is the resource. Authentication makes sure that it really is Mia making the request, and authorization determines if she's allowed to create Pods.

Webhook mode lets you offload authorization to an external REST-based policy engine. However, it requires additional effort to build and maintain the external engine. It also makes the external engine a potential single point of failure for every request to the API server. For example, if the external webhook system becomes unavailable, you may be unable to make any requests to the API server. With this in mind, you should be rigorous in vetting and implementing any webhook authorization service.

Node authorization is all about authorizing API requests made by kubelets (Nodes). The types of requests made to the API server by kubelets are obviously different from those generally made by regular users, and the node authorizer is designed to help with this.

Protecting Pods

The following few sections will look at a few technologies that help reduce the risk of elevation of privilege attacks against Pods and containers. We'll look at the following:

- Preventing processes from running as root
- Dropping capabilities
- Filtering syscalls
- Preventing privilege escalation

As you proceed through these sections, it's important to remember that a Pod is just an execution environment for one or more containers. Some of the terminology used will refer to Pods and containers interchangeably, but usually we will mean container.

Do not run processes as root

The *root* user is the most powerful user on a Linux system and is always User ID 0 (UID 0). This means running application processes as root is almost always a bad idea as it grants the application process full access to the container. This is made even worse by the fact the root user of a container sometimes has unrestricted root access to the host system as well. If that doesn't make you afraid, nothing will!

Fortunately, Kubernetes allows you to force container processes to run as unprivileged non-root users.

The following Pod manifest configures all containers that are part of this Pod to run processes as UID 1000. If the Pod has multiple containers, all container processes will run as UID 1000.

```
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  securityContext:        <<---- Applies to all containers in this Pod
    runAsUser: 1000       <<---- Non-root user
  containers:
  - name: demo
    image: example.io/simple:1.0
```

The `runAsUser` property is one of many settings that fall under the category of *PodSecurityContext* (`spec.securityContext`).

It's possible for two or more Pods to be configured with the same `runAsUser` UID. When this happens, the containers from both Pods will run with the same security context and potentially have access

to the same resources. This *might* be fine if they are replicas of the same Pod. However, there's a high chance this will cause problems if they're not replicas. For example, two different containers with R/W access to the same volume can cause data corruption (both writing to the same dataset without coordinating write operations). Shared security contexts also increase the possibility of a compromised container tampering with a dataset it shouldn't have access to.

With this in mind, it is possible to use the `securityContext.runAsUser` property at the container level instead of at the Pod level:

```
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  securityContext:        <<---- Applies to all containers in this Pod
    runAsUser: 1000       <<---- Non-root user
  containers:
  - name: demo
    image: example.io/simple:1.0
    securityContext:
      runAsUser: 2000     <<---- Overrides the Pod-level setting
```

This example sets the UID to 1000 at the Pod level but overrides it at the container level so that processes in the **demo** container run as UID 2000. Unless otherwise specified, all other containers in the Pod will use UID 1000.

A couple of other things that might help get around the issue of multiple Pods and containers using the same UID include:

- User namespaces
- Maintaining a map of UID usage

User namespaces is a Linux kernel technology that allows a process to run as root within a container but run as a different user outside the container. For example, a process can run as UID 0 (the root

user) inside the container but get mapped to UID 1000 on the host. This can be a good solution for processes that need to run as root inside the container. However, you should check if it is fully-supported by your version of Kubernetes and your container runtime.

Capability dropping

While most applications don't need the complete set of root capabilities, they usually require more capabilities than a typical non-root user.

What we need, is a way to grant the exact set of privileges a process requires in order to run. Enter *capabilities*.

Time for a quick bit of background.

We've already said the root user is the most powerful user on a Linux system. However, its power is a combination of lots of small privileges that we call *capabilities*. For example, the `SYS_TIME` capability allows a user to set the system clock, whereas the `NET_ADMIN` capability allows a user to perform network-related operations such as modifying the local routing table and configuring local interfaces. The root user holds every *capability* and is, therefore, extremely powerful.

Having a modular set of capabilities allows you to be extremely granular when granting permissions. Instead of an all-or-nothing (root –vs– non-root) approach, you can grant a process the exact set of capabilities required.

There are currently over 30 capabilities, and choosing the right ones can be daunting. With this in mind, many container runtimes implement a set of *sensible defaults* that allow most processes to run without *leaving all the doors open*. While sensible defaults like

these are better than nothing, they're often not good enough for production environments.

A common way to find the absolute minimum set of capabilities an application requires, is to run it in a test environment with all capabilities dropped. This causes the application to fail and log messages about the missing permissions. You map those permissions to capabilities, add them to the application's Pod spec, and run the application again. You rinse and repeat this process until the application runs properly with the minimum set of capabilities.

As good as this is, there are a few things to consider.

Firstly, you **must** perform extensive testing of each application. The last thing you want is a production edge case that you hadn't accounted for in your test environment. Such occurrences can crash your application in production!

Secondly, every application revision requires the same extensive testing against the capability set.

With these considerations in mind, it is vital that you have testing procedures and production release processes that can handle all of this.

By default, Kubernetes implements your chosen container runtime's default set of capabilities (E.g., containerd). However, you can override this as part of a container's `securityContext` field.

The following Pod manifest shows how to add the `NET_ADMIN` and `CHOWN` capabilities to a container.

```
apiVersion: v1
kind: Pod
metadata:
  name: capability-test
spec:
  containers:
```

```
- name: demo
  image: example.io/simple:1.0
  securityContext:
    capabilities:
      add: ["NET_ADMIN", "CHOWN"]
```

Filter syscalls

Seccomp, short for secure computing, is similar in concept to capabilities but works by filtering syscalls rather than capabilities.

The way an application asks the Linux kernel to perform an operation is by issuing a *syscall*. seccomp lets you control which syscalls a particular container can make to the host kernel. As with capabilities, you should implement a least privilege model where the only syscalls a container can make are the ones it needs in order to run.

Seccomp went GA in Kubernetes 1.19, and you can use it in different ways based on the following seccomp profiles:

1. **Non-blocking**: Allows a Pod to run, but records every syscall to an audit log you can use to create a custom profile. The idea is to extensively test your application Pod in a dev/test environment. After that, you'll have a log file listing every syscall the Pod needs in order to run. You then use this to create a custom profile that only allows those syscalls (least privilege).
2. **Blocking**: Blocks all syscalls. It's extremely secure but prevents a Pod from doing anything useful.
3. **Runtime Default**: Forces a Pod to use the seccomp profile defined by its container runtime. This is a common place to start if you still need to create a custom profile. Profiles that ship with container runtimes are designed to be a balance of *usable* and *secure*. They're also thoroughly tested.
4. **Custom**: A profile that only allows the syscalls your application needs in order to run. Everything else is blocked. It's common

to extensively test your application in dev/test environment with a non-blocking profile that records all syscalls to an audit log. You then use this log to identify your app's syscalls and build the customized profile. The danger with this approach is that your app has some edge cases you miss during testing. If this happens, your application can fail in production when it hits an edge case and uses a syscall not captured during testing.

Custom profiles operate the *least privilege* model and are the preferred approach from a security perspective.

Mandatory Access Controls

Seccomp filters and Capabilities are great tools for helping us run processes with restricted privileges. However, we can take things even further with mandatory access control (MAC) systems such as AppArmor and SELinux. These are Linux-only technologies that you configure at the Kubernetes node level and then apply them to Pods via Pod Security Contexts.

Both technologies control how processes interact with other system resources and can be hard to configure. However, tools are available that simplify configuration by reading audit logs and generating profiles that you can test and tweak. However, once enabled, they are mandatory. This is different from seccomp and Capabilities, which are voluntary.

Some container runtimes apply a default SELinux profile to all containers. However, you can override this via the Pod's `securityContext` field.

A word of caution though. AppArmor and SELinux are powerful enforcement points that can stop your apps from working if misconfigured. As such, you should perform extensive testing before implementing them.

Prevent privilege escalation by containers

The only way to create a new process in Linux is for one process to clone itself and then load new instructions onto the new process. We're over-simplifying, but the original process is called the *parent* process, and the copy is called the *child* process.

By default, Linux allows a *child* process to claim more privileges than its *parent*. This is usually a bad idea. In fact, you'll often want a child process to have the same or fewer privileges than its parent. This is especially true for containers, as their security configurations are defined against their initial configuration and not against potentially escalated privileges.

Fortunately, it's possible to prevent privilege escalation through the `securityContext` property of individual containers, as shown.

```
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  containers:
  - name: demo
    image: example.io/simple:1.0
    securityContext:
      allowPrivilegeEscalation: false    <<---- This line
```

Standardizing Pod Security with PSS and PSA

Modern Kubernetes clusters implement two technologies to help enforce Pod security settings:

- **Pod Security Standards (PSS)** are policies that specify required Pod security settings
- **Pod Security Admission (PSA)** enforces one or more PSS policies when Pods are created

Both work together for effective centralized enforcement of Pod security — you choose which PSS policies to apply, and PSA enforces them.

Pod Security Standards (PSS)

Every Kubernetes cluster gets the following three PSS *policies* that are maintained and kept up-to-date by the community:

- Privileged
- Baseline
- Restricted

`Privileged` is a wide-open allow-all policy.

`Baseline` implements sensible defaults. It's more secure than the *privileged* policy but less secure than *restricted*.

`Restricted` is the gold standard that implements the current Pod security best practices. Be warned though, it's highly restricted, and lots of Pods will fail to meet its strict requirements.

At the time of writing, you cannot tweak or modify any of these policies, and you cannot import others or create your own.

Pod Security Admission (PSA)

Pod Security Admission (PSA) enforces your desired PSS policies. It works at the Namespace level and is implemented as a *validating admission controller*.

PSA offers three enforcement modes:

- `Warn`: Allows violating Pods to be created but issues a user-facing warning

- **`Audit`**: Allows violating Pods to be created but logs an audit event
- **`Enforce`**: Rejects Pods if they violate the policy

It's a good practice to configure every Namespace with at least the `baseline` policy configured to either `warn` or `audit`. This allows you to start gathering data on which Pods are failing the policy and why. The next step is to enforce the `baseline` policy and start warning and auditing on the `restricted` policy.

Any Namespaces without a Pod Security configuration are a gap in your security configuration, and you should attach a policy as soon as possible, even if it's only warning and auditing.

Applying the following label to a Namespace will apply the `baseline` policy to it. It will allow violating Pods to run but will generate a user-facing warning.

```
pod-security.kubernetes.io/warn: baseline
```

The format of the label is `<prefix>/<mode>: <policy>` with the following options:

- Prefix is always `pod-security.kubernetes.io`
- Mode is one of `warn, audit,` or `enforce`
- Policy is always one of `privileged, baseline` or `restricted`

PSAs operate as validating admission controllers, meaning they cannot modify Pods. They also cannot have any impact on running Pods.

PSA examples

Let's walk through some examples to show you Pod Security Admission in action. You'll complete the following steps:

1. Create a Namespace called `psa-test`
2. Apply a label to `enforce` the `baseline` PSS policy
3. Attempt to deploy a Pod that runs a privileged container (will fail)
4. Modify the Pod to conform to the PSS policy and re-deploy it (will work)
5. Test the potential impact of switching to the `restricted` policy
6. Switch to the `restricted` policy
7. Test any impact on existing Pods

You'll need `kubectl`, a Kubernetes cluster, and a local clone of the book's GitHub repo if you want to follow along. See Chapter 3 if you need these.

You can clone the book's GitHub repo and switch to the `2025` branch with the following commands.

```
$ git clone https://github.com/nigelpoulton/TKB.git
<Snip>

$ cd TKB

$ git fetch origin

$ git checkout -b 2025 origin/2025
```

Be sure to run the following commands from the `psa` directory.

Run the following command to create a new Namespace called `psa-test.`

```
$ kubectl create ns psa-test
```

Add the `pod-security.kubernetes.io/enforce=baseline` label to the new Namespace. This will prevent the creation of any new Pods violating the `baseline` PSS policy.

```
$ kubectl label --overwrite ns psa-test \
    pod-security.kubernetes.io/enforce=baseline
```

Verify the label was correctly applied.

```
$ kubectl describe ns psa-test

Name:          psa-test
Labels:        kubernetes.io/metadata.name=psa-test
               pod-security.kubernetes.io/enforce=baseline   <<----
label correctly applied
Annotations:  <none>
Status:        Active
```

The Namespace is created and the `baseline` policy enforced.

The following YAML is from the `psa-pod.yml` file and defines a privileged container that violates the `baseline` policy.

```
apiVersion: v1
kind: Pod
metadata:
  name: psa-pod
  namespace: psa-test     <<---- Deploy it to the new psa-test
Namespace
spec:
  containers:
  - name: psa-ctr
    image: nginx
    securityContext:
      privileged: true     <<---- Violates the baseline policy
```

Deploy it with the following command.

```
$ kubectl apply -f psa-pod.yml

Error from server (Forbidden): error when creating "psa-pod.yml": pods
"psa-pod" is
forbidden: violates PodSecurity "baseline:latest": privileged
(container "psa-ctr"
must not set securityContext.privileged=true)
```

The output shows the Pod creation was forbidden and lists the reason why.

Edit the `psa-pod.yml` and change the container's `securityContext.privileged` to `false` and save your changes.

```
apiVersion: v1
kind: Pod
<Snip>
spec:
  containers:
  - name: psa-ctr
    image: nginx
    securityContext:
      privileged: false       <<---- Change from true to false
```

Now try to deploy the Pod.

```
$ kubectl apply -f psa-pod.yml
pod/psa-pod created
```

It passed the requirements for the `baseline` policy and was successfully deployed.

You can use the `--dry-run=server` flag to test the impact of applying a PSS policy to a Namespace. Using this flag **will not** apply the policy.

```
$ kubectl label --dry-run=server --overwrite ns psa-test \
    pod-security.kubernetes.io/enforce=restricted

Warning: existing pods in namespace "psa-test" violate the new
PodSecurity enforce
level "restricted:latest"
Warning: psa-pod: allowPrivilegeEscalation != false, unrestricted
capabilities,
runAsNonRoot != true, seccompProfile
<Snip>
```

The output shows the **psa-pod** Pod fails to meet four policy requirements:

- The `allowPrivilegeEscalation` property is not set to false
- It's running unrestricted capabilities
- The `runAsNonRoot` field is not set to true
- It fails the `seccompProfile` test

Go ahead and apply the policy to the Namespace and see if it impacts the `psa-pod` that is already running.

```
$ kubectl label --overwrite ns psa-test \
    pod-security.kubernetes.io/enforce=restricted

Warning: existing pods in namespace "psa-test" violate the new
PodSecurity enforce level
"restricted:latest"
Warning: psa-pod: allowPrivilegeEscalation != false, unrestricted
capabilities,
runAsNonRoot != true, seccompProfile
namespace/psa-test labeled

$ kubectl get pods --namespace psa-test

NAME       READY    STATUS      RESTARTS    AGE
psa-pod    1/1      Running     0           3m9s
```

You get the same warning message, but it doesn't terminate existing Pods. This is because PSA runs as an admission controller and, therefore, only acts on the creation and modification of Pods.

Finally, it's possible to configure multiple policies and modes against a single Namespace. In fact, it's a common practice to do this.

The following example applies three labels to the `psa-test` Namespace. They *enforce* the `baseline` policy, and *warn* and *audit* against the `restricted` policy. This is a good way to implement the `baseline` policy and prepare for `restricted`.

```
$ kubectl label --overwrite ns psa-test \
    pod-security.kubernetes.io/enforce=baseline \
    pod-security.kubernetes.io/warn=restricted \
    pod-security.kubernetes.io/audit=restricted
```

You can run a `kubectl describe ns psa-test` command to ensure the labels were applied.

Alternatives to Pod Security Admission

As previously mentioned, PSS and PSA have limitations. These include being implemented as a validating admission controller and being unable to modify, import, or create your own policies. If you need more than PSS and PSA can offer, you may want to consider the following 3rd-party solutions:

- OPA Gatekeeper
- Kubewarden
- Kyverno

Others also exist.

Towards a more secure Kubernetes

As demonstrated by the following examples, Kubernetes is on a continual journey towards better security.

Starting with Kubernetes v1.26, all binary artifacts and container images used to build Kubernetes clusters are cryptographically signed.

The Kubernetes community maintains an official feed for all publicly announced Kubernetes vulnerabilities (CVEs). Since v1.27, a JSON and RSS feed that auto-refreshes when any new CVE is announced is available.

Starting from Kubernetes 1.27, all containers inherit a default seccomp profile from the container runtime that implements sensible security defaults. This requires the `--seccomp-default` on every kubelet.

Many cloud providers implement *confidential computing* services such as *confidential virtual machines* and *confidential containers* that Kubernetes can leverage to secure *data in use* by enabling

memory encryption for container workloads, etc. Some cloud providers even offer it as part of their hosted Kubernetes services.

An up-to-date third-party [security audit of Kubernetes](#) was published in April 2023 based on Kubernetes 1.24. It's the second report of its kind and follows on from the original in 2019. These are great tools for identifying potential threats to your Kubernetes environments, as well as potential ways to mitigate them.

Finally, the [Cloud Native Security Whitepaper](#) is worth reading as a way to level up and gain a more holistic perspective on securing cloud-native environments such as Kubernetes.

Chapter summary

This chapter taught you how the STRIDE model can be used to threat-model Kubernetes. You stepped through the six threat categories and looked at some ways to prevent and mitigate them.

You saw that one threat can often lead to another and that multiple ways exist to mitigate a single threat. As always, defense in depth is a key tactic.

The chapter finished by discussing how Pod Security Admission is the preferred way to implement Pod security defaults.

In the next chapter, you'll see some best practices and lessons learned from running Kubernetes in production.

17: Real-world Kubernetes security

The previous chapter showed you how to threat-model Kubernetes using the STRIDE model. In this chapter, you'll learn about security-related challenges you're likely to encounter when implementing Kubernetes in the real world.

The goal of the chapter is to show you things from the kind of high-level view a security architect has. It does not give *cookbook* style solutions.

The chapter is divided into the following four sections:

- Security in the software delivery pipeline
- Workload isolation
- Identity and access management
- Security monitoring and auditing

Security in the software delivery pipeline

Containers revolutionized the way we build, ship, and run applications. Unfortunately, this has also made it easier than ever to run dangerous code.

Let's look at some ways you can secure the supply chain that gets application code from a developer's laptop onto production servers.

Image Repositories

We store images in public and private registries that we divide into *repositories*.

Public registries are on the internet and are the easiest way to push and pull images. However, you should be very careful when using

them:

1. You need to adequately protect the images you store on public registries
2. You should not trust the images you pull from public registries

Some public registries have the concept of *official images* and *community images*. As a general rule, *official images* are safer than *community images*, but you should **always** do your due diligence.

Official images are usually provided by product vendors and undergo vigorous vetting processes to ensure quality. You should expect them to implement good practices, be regularly scanned for vulnerabilities, and contain up-to-date patches and fixes. Some of them may even be supported by the product vendor or the company hosting the registry.

Community images do not undergo rigorous vetting, and you should practice extreme caution when using them.

With these points in mind, you should implement a standardized way for developers to obtain and consume images. You should also make the process as frictionless as possible so that developers don't feel the need to bypass the process.

Let's discuss a few things that might help.

Use approved base images

Most images start with a *base layer* and then add other layers to form a useful image.

Figure 17.1 shows an oversimplified example of an image with three layers. The base layer has the core OS and filesystem components, the middle layer has the libraries and dependencies, and the top

layer has your app. The combination of the three is the *image* and contains everything needed to run the application.
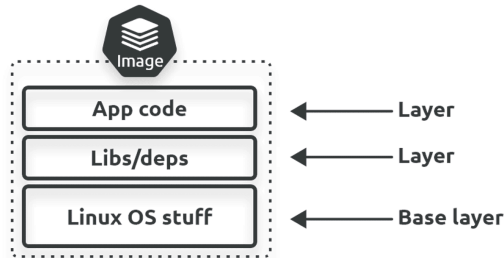

Figure 17.1 - Image layering

It's usually a good practice to maintain a small number of *approved base images*. These are usually derived from *official images* and hardened according to your corporate policies and requirements. For example, you might create a limited number of *approved base images* based on the official Alpine Linux image you've tweaked to meet your requirements (patches, drivers, audit settings, and more).

Figure 17.2 shows three applications built on top of two approved base images. The app on the left builds on top of your approved Alpine Linux base image, whereas the other two apps are web apps that build on top of your approved Alpin+NGINX base image.
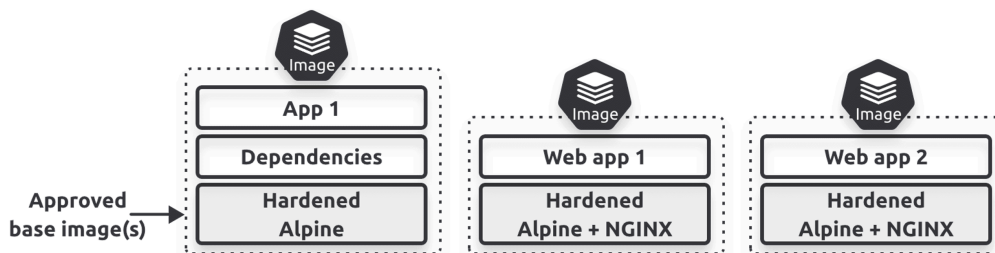

Figure 17.2 - Using approved base images

While you need to invest up-front effort to create your approved base images, they bring all the following benefits:

- Standard set of drivers
- Known patches
- Standardized audit settings
- Reduced software sprawl (less unofficial base images)
- Simplified testing (testing against a small set of known bases)
- Simplified updates (Fewer base images to patch)
- Simplified troubleshooting (a well-understood and limited set of base images)

Having an approved set of base images also allows developers to focus on applications without caring about OS-related stuff. It may also allow you to reduce the number of support contracts and suppliers you have to deal with.

Manage the need for non-standard base images

As good as having a small number of approved base images is, you may still have legitimate requirements for bespoke configurations. In these situations you'll need good processes to:

- Identify why an existing approved base image cannot be used
- Determine whether an existing approved base image can be updated to meet requirements (including if it's worth the effort)
- Determine the support implications of bringing an entirely new image into the environment

In most cases, you'll want to update an existing base image — such as adding a device driver for GPU computing — rather than introducing an entirely new image.

Control access to images

There are several ways to protect your organization's images.

A secure and practical option is to host your own *private registries* inside your own firewalls. This allows you to control how registries

are deployed, how they're replicated, and how they're patched. You can also create repositories and policies to fit your organizational needs, and integrate them with existing identity management providers such as Active Directory.

If you can't manage your own private registries, you can host your images in *private repositories* on public registries. However, not all public registries are equal, and you'll need to take great care in choosing the right one and configuring it correctly.

Whichever solution you choose, you should only host images that are approved for use within your organization. These will typically be from a *trusted* source and vetted by your information security team. You should place access controls on repositories so that only approved users can push and pull them.

Away from the registry itself, you should also:

- Restrict which cluster nodes have internet access, keeping in mind that your image registry may be on the internet
- Configure access controls that only allow authorized users and nodes to push to repositories

If you're using a public registry, you'll probably need to grant your cluster nodes access to the internet so they can pull images. In scenarios like this, it's a good practice to limit internet access to the addresses and ports your registries use. You should also implement strict RBAC rules on the registry to control who can push and pull images from which repositories. For example, you might restrict developers so they can only push and pull against *dev* and *test* repositories, whereas you may allow your operations teams to push and pull against *production* repos.

Finally, you may only want a subset of nodes (*build nodes*) to be able to *push* images. You may even want to lock things down so that only your automated build systems can push to specific repositories.

Moving images from non-production to production

Many organizations have separate environments for development, testing, and production.

As a general rule, development environments have fewer rules and are places where developers can experiment. This can involve non-standard images your developers eventually want to use in production.

The following sections outline some measures you can take to ensure that only safe images get approved for production.

Vulnerability scanning

Top of the list for vetting images before allowing them into production should be *vulnerability scanning*. These services scan your images at a binary level and check their contents against databases of known security vulnerabilities (CVEs).

You should integrate vulnerability scanning into your CI/CD pipelines and implement policies that automatically fail builds and quarantine images if they contain particular categories of vulnerabilities. For example, you might implement a build phase that scans images and automatically fails anything using an image with known *critical* vulnerabilities.

However, some scanning solutions are better than others and will allow you to create highly customizable policies.

For example, a Python *method* that performs TLS verification might be vulnerable to Denial of Service attacks when the `Common Name` contains a lot of wildcards. However, if you never use Python in this way, you might not consider the vulnerability relevant and want to mark it as a false positive. Not all scanning solutions allow you to do this.

Configuration as code

Scanning app code for vulnerabilities is widely accepted as good production hygiene. However, scanning your Dockerfiles, Kubernetes YAML files, Helm charts, and other configuration files is less widely adopted.

A well-publicized example of not reviewing configuration files was when an IBM data science experiment embedded private TLS keys in its container images. This meant attackers could pull the image and gain root access to the nodes hosting the containers. The whole thing would've been easily avoided if they'd performed a security review against their Dockerfiles.

There continue to be advancements in automating checks like these with tools that implement *policy as code* rules.

Sign container images

Trust is a big deal in today's world, and cryptographically signing content at every stage in the software delivery pipeline is becoming the norm. Fortunately, Kubernetes and most container runtimes support cryptographically signing and verifying images.

In this model, developers cryptographically sign their images, and consumers cryptographically verify them when they pull them and run them. This gives the consumer confidence they're working with the correct image and that it hasn't been tampered with.

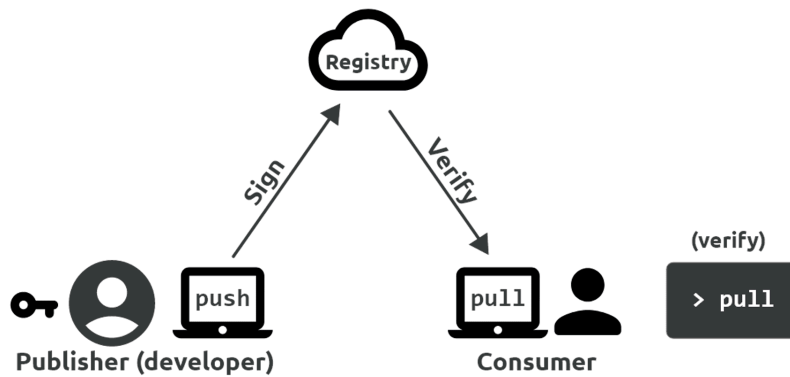Figure 17.3 shows the high-level process for signing and verifying images.

Figure 17.3

Image signing and verification is usually implemented by the container runtime.

You should look at tools that allow you to define and enforce enterprise-wide signing policies so it's not left up to individual users.

Image promotion workflow

With everything we've covered so far, your build pipelines should include as many of the following as possible:

1. Policies forcing the use of signed images
2. Network rules restricting which nodes can push and pull images
3. RBAC rules protecting image repositories
4. Use of approved base images
5. Image scanning for known vulnerabilities
6. Promotion and quarantining of images based on scan results
7. Review and scan infrastructure-as-code configuration files

There are more things you can do, and the list isn't supposed to represent an exact workflow.

Workload isolation

This section will show you some ways you can isolate workloads.

We'll start at the cluster level, switch to the runtime level, and then look outside the cluster at infrastructure such as network firewalls.

Cluster-level workload isolation

Cutting straight to the chase, **Kubernetes does not support secure multi-tenant clusters**. The only way to isolate two workloads is to run them on their own clusters with their own hardware.

Let's look a bit closer.

The only way to divide a Kubernetes cluster is by creating *Namespaces*. However, these are little more than a way of grouping resources and applying things such as:

- Limits
- Quotas
- RBAC rules

Namespaces do not prevent compromised workloads in one Namespace from impacting workloads in other Namespaces. This means you should never run hostile workloads on the same Kubernetes cluster.

Despite this, Kubernetes Namespaces are useful, and you *should* use them. Just don't use them as security boundaries.

Namespaces and soft multi-tenancy

For our purposes, soft multi-tenancy is hosting multiple *trusted workloads* on shared infrastructure. By *trusted*, we mean workloads

that don't require absolute guarantees that one workload cannot impact another.

An example of trusted workloads might be an e-commerce application with a web front-end service and a back-end recommendation service. As they're part of the same application, they're not hostile. However, you might want each one to have its own resource limits managed by different teams.

In situations like this, a single cluster with a Namespace for the front-end service and another for the back-end service might be a good solution.

Namespaces and hard multi-tenancy

We'll define *hard multi-tenancy* as hosting untrusted and potentially hostile workloads on shared infrastructure. However, as we said before, this isn't *currently* possible with Kubernetes.

This means workloads requiring a strong security boundary need to run on separate Kubernetes clusters! Examples include:

- Isolating production and non-production workloads
- Isolating different customers
- Isolating sensitive projects and business functions

Other examples exist, but the take-home point is that workloads requiring strong separation need their own clusters.

> **Note:** The Kubernetes project has a dedicated *Multitenancy Working Group* that's actively working on multitenancy models. This means that future Kubernetes releases might have better solutions for hard multitenancy.

Node isolation

There will be times when you have applications that require non-standard privileges, such as running as root or executing non-standard syscalls. Isolating these on their own clusters might be overkill, but you might justify running them on a ring-fenced subset of worker nodes. Doing this will restrict compromised workloads from only impacting other workloads on the same node.

You should also apply *defense in depth* principles by enabling stricter audit logging and tighter runtime defense options on nodes running workloads with non-standard privileges.

Kubernetes offers several technologies, such as labels, affinity and anti-affinity rules, and taints, to help you target workloads to specific nodes.

Runtime isolation

Containers versus virtual machines used to be a polarizing topic. However, when it came to *workload isolation* there is only ever one winner… virtual machines.

Most container platforms implement *namespaced containers*. This is a model where every container shares the host's kernel, and isolation is provided by kernel constructs, such as namespaces and cgroups, that were never designed as *strong* security boundaries. Docker, containerd, and CRI-O are popular examples of container runtimes and platforms that implement namespaced containers.

This is very different from the hypervisor model, where every virtual machine gets its own dedicated kernel and is strongly isolated from other virtual machines using hardware enforcement.

However, it's easier than ever to augment containers with security-related technologies that make them more secure and enable

stronger workload isolation. These technologies include AppArmor, SELinux, seccomp, capabilities, and user namespaces, and most container runtimes and hosted Kubernetes services do a good job of implementing sensible defaults for them all. However, they can still be complex, especially when troubleshooting.

You should also consider different classes of container runtimes. Two examples are **gVisor** and **Kata Containers**, both of which provide stronger levels of workload isolation and are easy to integrate with Kubernetes thanks to the *Container Runtime Interface (CRI)* and *Runtime Classes*.

There are also projects that enable Kubernetes to orchestrate other workload types, such as virtual machines, serverless functions, and WebAssembly.

While you might feel overwhelmed by some of this, you need to consider all of this when determining the isolation levels your workloads require.

To summarize, the following workload isolation options exist:

1. **Virtual Machines:** Every workload gets its own dedicated kernel. It provides excellent isolation but is comparatively slow and resource-intensive.
2. **Namespaced containers:** All containers share the host's kernel. These are fast and lightweight but require extra effort to improve workload isolation.
3. **Run every container in its own virtual machine:** Solutions like these attempt to combine the versatility of containers with the security of VMs by running every container in its own dedicated VM. Despite using specialized lightweight VMs, these solutions lose much of the appeal of containers, and they're not very popular.
4. **Use different runtime classes:** This allows you to run all workloads as containers, but you target the workloads requiring

stronger isolation to an appropriate container runtime.

5. **Wasm containers:** Wasm containers package Wasm (WebAssembly) apps in OCI containers that can execute on Kubernetes. These apps only use containers for packaging and scheduling, at run time they execute inside a secure deny-by-default Wasm host. See Chapter 9 for more detail.

Network isolation

Firewalls are an integral part of any layered information security system. The goal is only to allow authorized communications.

In Kubernetes, Pods communicate over an internal network called the *pod network*. However, Kubernetes doesn't implement the *pod network*. Instead, it implements a plugin model called the Container Network Interface (CNI) that allows 3rd-party vendors to implement the pod network. Lots of CNI plugins exist, but they fall into two broad categories:

- Overlay
- BGP

Each has a different impact on firewall implementation and network security.

Kubernetes and overlay networking

Most Kubernetes environments implement the pod network as a simple flat *overlay network* that hides any network complexity between cluster nodes. For example, you might deploy your cluster nodes across ten different networks connected by routers, but Pods connect to the flat pod network and communicate without needing to know any of the complexity of the host networking. Figure 17.4 shows four nodes on two separate networks and the Pods connected to a single overlay pod network.
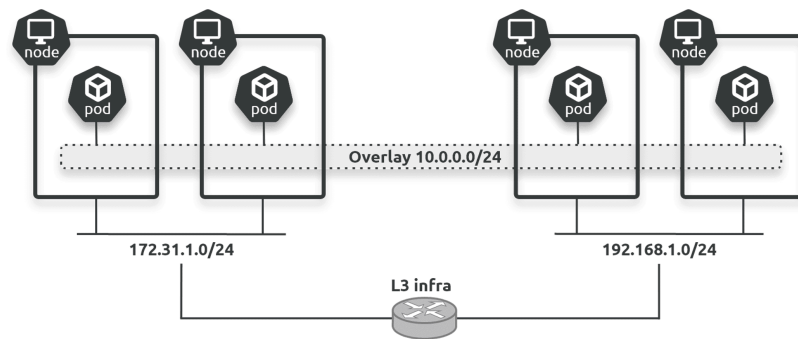
Figure 17.4

Overlay networks use VXLAN technologies to encapsulate traffic for transmission over a simple flat Layer-2 network operating on top of existing Layer-3 infrastructure. If that's too much network jargon, all you need to know is that overlay networks encapsulate packets sent by containers. This encapsulation hides the original source and target IP addresses, making it harder for firewalls to know what's going on. See Figure 17.5
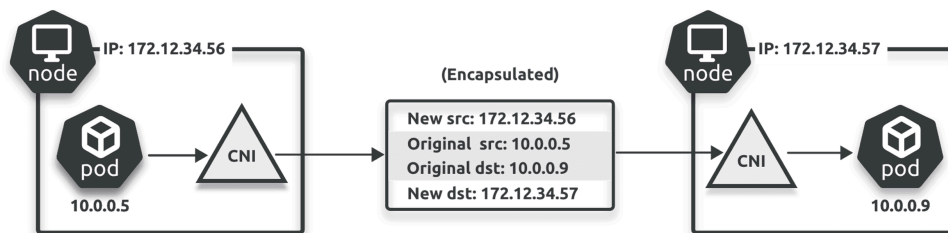

Figure 17.5 - Encapsulation on overlay network

Kubernetes and BGP

BGP is the protocol that powers the internet. However, at its core, it's a simple and scalable protocol that creates peer relationships that are used to share routes and perform routing.

The following analogy might help. Imagine you want to send a birthday card to a friend who you lost contact with and no longer

have their address. However, your child has a friend at school whose parents are still in touch with your old friend. In this situation, you give the card to your child and ask them to give it to their friend at school. This friend gives it to their parents, who deliver it to your friend.

BGP routing is similar and happens through a network of *peers* that help each other find routes.

From a security perspective, the important thing is that BGP doesn't encapsulate packets. This makes things much simpler for firewalls. Figure 17.6 shows the same setup using BGP. Notice how there's no encapsulation.

Figure 17.6 - No encapsulation on BGP network

How this impacts firewalls

We've already said that firewalls allow or disallow traffic flow based on source and destination addresses. For example:

- Allow traffic from the `10.0.0.0/24` network
- Disallow traffic from the `192.168.0.0/24` network

Suppose your pod network is an overlay network. In that case, all traffic will be encapsulated, and only firewalls that can open packets and inspect their contents will be able to make useful decisions on

whether to allow or deny traffic. You may want to consider a BGP pod network if your firewalls can't do this.

You should also consider whether to deploy *physical firewalls*, *host-based firewalls*, or a combination of both.

Physical firewalls are dedicated network hardware devices that are usually managed by a central team. Host-based firewalls are operating system (OS) features and are usually managed by the team that deploys and manages your OSes. Both solutions have pros and cons, and combining the two is often the most secure. However, you should consider whether your organization has a long and complex procedure for implementing changes to physical firewalls. If it does, it might not suit the nature of your Kubernetes environment.

Packet capture

On the topic of networking and IP addresses, not only are Pod IP addresses sometimes obscured by encapsulation, but they are also dynamic and can be recycled and re-used by different Pods. We call this *IP churn*, and it reduces how useful IP addresses are at identifying systems and workloads. With this in mind, the ability to associate IP addresses with Kubernetes-specific identifiers such as Pod IDs, Service aliases, and container IDs when performing things like packet capturing can be extremely useful.

Let's switch tack and look at some ways of controlling user access to Kubernetes.

Identity and access management (IAM)

Controlling user access to Kubernetes is important in any production environment. Fortunately, Kubernetes has a robust RBAC subsystem

that integrates with existing IAM providers such as Active Directory, other LDAP systems, and cloud-based IAM solutions.

Most organizations already have a centralized IAM provider that's integrated with company HR systems to simplify employee lifecycle management.

Fortunately, Kubernetes leverages existing IAM providers instead of implementing its own. This means new employees get an identity in the corporate IAM database, and assuming you make them members of the appropriate groups, they will automatically get permissions in Kubernetes. Likewise, when the employee leaves the organization, an HR process will automatically remove their identity from the IAM database, and their Kubernetes access will cease.

RBAC has been a stable Kubernetes feature since v1.8 and you should leverage its full capabilities.

Managing Remote SSH access to cluster nodes

You'll do almost all Kubernetes administration via REST calls to the API server. This means users should rarely need remote SSH access to Kubernetes cluster nodes. In fact, remote SSH access to cluster nodes should only be for the following types of activity:

- *Node management* activities that you cannot perform via the Kubernetes API
- *Break the Glass* activities, such as when the API server is down
- Deep troubleshooting

Multi-factor authentication (MFA)

With great power comes great responsibility.

Accounts with root access to the API server and root access to cluster nodes are extremely powerful and are prime targets for

attackers and disgruntled employees. As such, you should protect their use via multi-factor authentication (MFA). This is where a user has to input a username and password, followed by a second stage of authentication. For example:

- Stage 1: Tests *knowledge* of a username and password
- Stage 2: Tests *possession* of something like a one-time password

You should also secure access to workstations and user profiles that have kubectl installed.

Security monitoring and auditing

No system is 100% secure, and you should always plan for the eventuality that your systems will be breached. When breaches happen, it's vital you can do at least two things:

1. Recognize that a breach has occurred
2. Build a detailed timeline of events that cannot be repudiated

Auditing is critical to both of these, and the ability to build a reliable timeline helps answer the following post-event questions:

- What happened
- How did it happen
- When did it happen
- Who did it

In extreme circumstances, this information can be called upon in court.

Good auditing and monitoring solutions also help identify vulnerabilities in your security systems.

With these points in mind, you should ensure robust auditing and monitoring are high on your list of priorities, and you shouldn't go live in production without them.

Baseline best practices

There are various tools and checks that can help you ensure you provision your Kubernetes environment according to best practices and company policies.

The *Center for Information Security (CIS)* publishes an industry-standard benchmark for Kubernetes security, and Aqua Security (aquasec.com) has written an easy-to-use tool called [*kube-bench*](#) to run the CIS tests against your cluster and generate reports. Unfortunately, kube-bench can't inspect the control plane nodes of hosted Kubernetes services.

You should consider running kube-bench as part of the node provisioning process and pass or fail node provisioning based on the results.

You can also use kube-bench reports as a baseline for use in the aftermath of incidents. This allows you to compare the kube-bench reports from before and after the incident and determine *if* and *where* any configuration changes occurred.

Container and Pod lifecycle events

Pods and containers are ephemeral objects that come and go all the time. This means you'll see a lot of events announcing new ones and a lot of events announcing terminated ones.

With this in mind, consider configuring log retention to keep the logs from terminated Pods so they're available for inspection even after termination.

Your container runtime may also keep logs relating to container lifecycle events.

Forensic checkpointing

Forensics is the science of collecting and examining available evidence to construct a trail of events, especially when you suspect malicious behavior.

The ephemeral nature of containers has made this challenging in the past. However, recent technologies such as *Checkpoint/Restore in Userspace (CRIU)* are making it easier to silently capture the state of running containers and restore them in a sandbox environment for deeper analysis. At the time of writing, CRIU is an alpha feature in Kubernetes, and the only runtime currently supporting it is CRI-O.

Application logs

Application logs are also important when identifying potential security-related issues.

However, not all applications send their logs to the same place. Some send them to their container's *standard out (stdout)* or *standard error (stderr)* streams where your logging tools can pick them up alongside container logs. However, some send logs to proprietary log files in bespoke locations. Be sure to research this for each application and configure things so you don't miss logs.

Actions performed by users

Most of your Kubernetes configuration and administration will be done via the API server, where all requests should be logged. However, it's also possible for malicious actors to gain remote SSH access to control plane nodes and directly manipulate Kubernetes

objects. This may include access to the cluster store and etcd nodes.

We've already said you should limit SSH access to cluster nodes and bolster security with multi-factor authentication (MFA). However, you should also log all SSH activity and ship it to a secure log aggregator. You should also consider mandating that two competent people be present for all SSH access to control plane nodes.

Managing log data

A key advantage of containers is application density — you can run a lot more applications on your servers and in your datacenters. This results in massive amounts of log data and audit data that is overwhelming without specialized tools to sort and make sense of it. Fortunately, advanced tools exist that not only store the data, but can use it for proactive analysis as well as post-event reactive analysis.

Alerting for security-relevant events

As well as being useful for post-event analysis and repudiation, some events are significant enough to warrant immediate investigation. Examples include:

- *Privileged Pod creation by a human user:* Privileged Pods can often gain root-level access on the node, and you will typically have policies in place to prevent their creation. On the rare occasions they are needed, they will usually be created by automated processes with service accounts.
- *Exec sessions by human users:* Exec sessions grant *shell-like* access to containers and are typically only used to troubleshoot issues. You should investigate exec sessions that aren't for troubleshooting and consider deleting them to prevent tampering.

- *Attempts to access the cluster from the internet:* It's a common practice to prevent access to the control plane from the internet. As such, you should monitor for successful and unsuccessful attempts to connect to the control plane from the internet, and successful attempts will typically indicate a security misconfiguration you should fix.

Migrating existing apps to Kubernetes

It can be useful to use a crawl, walk, then run strategy when migrating applications to Kubernetes:

1. *Crawl*: Threat modeling your existing apps will help you understand their current security posture. For example, which of your existing apps do and don't communicate over TLS.
2. *Walk*: When moving to Kubernetes, ensure the security posture of these apps remains unchanged. For example, if an app doesn't communicate over TLS, do **not** change this as part of the migration.
3. *Run*: Start improving the security of applications after the migration. Start with simple non-critical apps, and carefully work your way up to mission-critical apps. You may also want to methodically deploy deeper levels of security, such as initially configuring apps to communicate over one-way TLS and then eventually over two-way TLS.

The key point is not to change the security posture of an app as part of migrating it to Kubernetes. This is because performing a migration **and** making changes can make it easier to misdiagnose issues — was it the security change or the migration?

Real-world example

An example of a container-related vulnerability that could've easily been prevented by implementing some of the best practices we've

discussed occurred in February 2019. CVE-2019-5736 allowed a container process running as root to gain root access on the worker node **and** all containers running on the host.

As dangerous as the vulnerability was, the following things covered in this chapter would've prevented the issue:

- Image vulnerability scanning
- Not running processes as root
- Enabling SELinux

As the vulnerability has a CVE number, scanning tools would've found it and alerted on it. Even if scanning platforms missed it, policies that prevent root containers and standard SELinux policies would have prevented exploitation of the vulnerability.

Chapter summary

The purpose of this chapter was to introduce some of the real-world security considerations affecting many Kubernetes.

We started by looking at ways to secure the software delivery pipeline and discussed some image-related best practices. These included securing your image registries, scanning images for vulnerabilities, and cryptographically signing and verifying images. Then, we looked at some of the workload isolation options that exist at different layers of the infrastructure stack. In particular, we looked at cluster-level isolation, node-level isolation, and some of the different runtime isolation options. We discussed identity and access management, including places where additional security measures might be useful. We then talked about auditing and finished up with a real-world issue that could have been avoided by implementing some of the best practices already covered.

Hopefully, you have enough to go away and start securing your own Kubernetes clusters.

Terminology

This glossary defines some of the most common Kubernetes-related terms used in the book. Ping me if you think I've missed anything important:

- tkb@nigelpoulton.com

Term	Definition (according to Nigel)
Admission controller	Code that validates or mutates resources to enforce policies. Runs as part of the API admission chain immediately after authentication and authorization.
Annotation	Object metadata that can be used to expose alpha or beta capabilities or integrate with third-party systems.
API	Application Programming Interface. In the case of Kubernetes, all resources are defined in the API, which is RESTful and exposed via the *API server*.
API group	A set of related API resources. For example, networking resources are usually located in the `networking.k8s.io` API group.
API resource	All Kubernetes objects, such as Pods, Deployments, and Services, are defined in the API as resources.
API Server	Exposes the API on a secure port over HTTPS. Runs on the control plane.
Cloud controller manager	Control plane service that integrates with the underlying cloud platform. For example, when creating a LoadBalancer Service, the cloud controller manager implements the logic to provision one of the underlying cloud's internet-facing load balancers.

Term	Definition (according to Nigel)
Cloud native	A loaded term that means different things to different people. Cloud native is a way of designing, building, and working with modern applications and infrastructure. I personally consider an application to be *cloud native* if it can self-heal, scale on-demand, perform rolling updates, and possibly rollbacks.
Cluster	A set of worker and control plane nodes that work together to run user applications.
Cluster store	Control plane feature that holds the state of the cluster and apps. Typically, it is based on the `etcd` distributed data store and runs on the control plane. It can be deployed to its own cluster for higher performance and higher availability.
ConfigMap	Kubernetes object used to hold non-sensitive configuration data. A great way to add custom configuration data to a generic container at runtime without editing the image.
Container	Lightweight environment for running modern apps. Each container is a virtual operating system with its own process tree, filesystem, shared memory, and more. One container runs one application process.
Container Network Interface (CNI)	Pluggable interface enabling different network topologies and architectures. 3rd-parties provide CNI plugins that enable overlay networks, BGP networks, and various implementations of each.

Term	Definition (according to Nigel)
Container runtime	Low-level software running on every cluster Node responsible for pulling container images, starting containers, stopping containers, and other low-level container operations. Typically containerd, Docker, or cri-o. Docker was deprecated in Kubernetes 1.20, and support was removed in 1.24.
Container Runtime Interface (CRI)	Low-level Kubernetes feature that allows container runtimes to be pluggable. With the CRI, you can choose the best container runtime for your requirements (Docker, containerd, cri-o, kata, etc.)
Container Storage Interface (CSI)	Interface enabling external 3rd-party storage systems to integrate with Kubernetes. Storage vendors write a CSI driver/plugin that runs as a set of Pods on a cluster and exposes the storage system's enhanced features to the cluster and applications.
containerd	Industry-standard container runtime used in most Kubernetes clusters. Donated to the CNCF by Docker, Inc. Pronounced "container dee".
Controller	Control plane process running as a reconciliation loop monitoring the cluster and making the necessary changes so the observed state of the cluster matches desired state.

Term	Definition (according to Nigel)
Control plane	The brains of every Kubernetes cluster. Comprises the API, API server, scheduler, all controllers, and more. These components run on all *control plane nodes* of every cluster.
control plane node	A cluster node hosting control plane services. Usually, it doesn't run user applications. You should deploy 3 or 5 for high availability.
cri-o	Container runtime. Commonly used in OpenShift-based Kubernetes clusters.
CRUD	The four basic Create, Read, Update, and Delete operations used by many storage systems.
Custom Resource Definition (CRD)	API resource used for adding your own resources to the Kubernetes API.
Data plane	The worker Nodes of a cluster that host user applications.
Deployment	Controller that deploys and manages a set of stateless Pods. Performs rollouts and rollbacks, and can self-heal. Uses a ReplicaSet controller to perform scaling and self-healing operations.
Desired state	What the cluster and apps should be like. For example, the *desired state* of an application microservice might be five replicas of xyz container listening on port 8080/tcp. Vital to reconciliation.

Term	Definition (according to Nigel)
Endpoints object	Up-to-date list of healthy Pods matching a Service's label selector. Basically, it's the list of Pods a Service will send traffic to. Might eventually be replaced by EndpointSlices.
etcd	The open-source distributed database used as the cluster store on most Kubernetes clusters.
Ingress	API resource that exposes multiple internal Services over a single external-facing LoadBalancer Service. Operates at layer 7 and implements path-based and host-based HTTP routing.
Ingress class	API resource that allows you to specify multiple different Ingress controllers on your cluster.
Init container	A specialized container that runs and completes before the main app container starts. Commonly used to check/initialize the environment for the main app container.
JSON	JavaScript Object Notation. The preferred format for sending and storing data used by Kubernetes.
K8s	Shorthand way to write Kubernetes. The "8" replaces the eight characters between the "K" and the "s" of Kubernetes. Pronounced "Kates". The reason why people say Kubernetes' girlfriend is called Kate.

Term	Definition (according to Nigel)
kubectl	Kubernetes command line tool. Sends commands to the API server and queries state via the API server.
Kubelet	The main Kubernetes agent running on every cluster Node. It watches the API Server for new work assignments and maintains a reporting channel back.
Kube-proxy	Runs on every cluster node and implements low-level rules that handle traffic routing from Services to Pods. You send traffic to stable Service names, and kube-proxy makes sure the traffic reaches Pods.
Label	Metadata applied to objects for grouping. Works with label selectors to match Pods with higher-level controllers. For example, Services send traffic to Pods based on sets of matching labels.
Label selector	Used to identify Pods to perform actions on. For example, when a Deployment performs a rolling update, it knows which Pods to update based on its label selector – only Pods with labels matching the Deployment's label selector will be replaced and updated.
Manifest file	YAML file that holds the configuration of one or more Kubernetes objects. For example, a Service manifest file is typically a YAML file that holds the configuration of a Service object. When you post a manifest file to the API Server, its configuration is deployed to the cluster.

Term	Definition (according to Nigel)
Microservices	A design pattern for modern applications. Application features are broken into their own small applications (microservices/containers) and communicate via APIs. They work together to form a useful application.
Namespace	A way to partition a single Kubernetes cluster into multiple virtual clusters. Good for applying different quotas and access control policies on a single cluster. Not suitable for strong workload isolation.
Node	Also known as worker node. The nodes in a cluster that run user applications. Runs the kubelet process, a container runtime, and kube-proxy.
Observed state	Also known as *current state* or *actual state*. The most up-to-date view of the cluster and running applications. Controllers are always working to make observed state match desired state.
Orchestrator	A piece of software that deploys and manages apps. Modern apps are made from many small microservices that work together to form a useful application. Kubernetes orchestrates/manages these, keeps them healthy, scales them up and down, and more… Kubernetes is the de facto orchestrator of microservices apps based on containers.

Term	Definition (according to Nigel)
Persistent Volume (PV)	Kubernetes object used to map storage volumes on a cluster. External storage resources must be mapped to PVs before they can be used by applications.
Persistent Volume Claim (PVC)	Like a ticket/voucher that allows an app to use a Persistent Volume (PV). Without a valid PVC, an app cannot use a PV. Combined with StorageClasses for dynamic volume creation.
Pod	Smallest unit of scheduling on Kubernetes. Every container running on Kubernetes must run inside a Pod. The Pod provides a shared execution environment – IP address, volumes, shared memory etc.
RBAC	Role-based access control. Authorization module that determines whether authenticated users can perform actions against cluster resources.
Reconciliation loop	A controller process watching the state of the cluster via the API Server, ensuring observed state matches desired state. Most controllers, such as the Deployment controller, run as a reconciliation loop.
ReplicaSet	Runs as a controller and performs self-healing and scaling. Used by Deployments.

Term	Definition (according to Nigel)
REST	REpresentational State Transfer. The most common architecture for creating web-based APIs. Uses the common HTTP methods (GET, POST, PUT, PATCH, DELETE) to manipulate and store objects.
Secret	Like a ConfigMap for sensitive configuration data. A way to store sensitive data outside of a container image and have it inserted into a container at runtime.
Service	Capital "S". Kubernetes object for providing network access to apps running in Pods. By placing a Service in front of a set of Pods, the Pods can fail, scale up and down, and be replaced without the network endpoint for accessing them changing. Can integrate with cloud platforms and provision internet-facing load balancers.
Service mesh	Infrastructure software that enables features such as encryption of Pod-to-Pod traffic, enhanced network telemetry, and advanced routing. Common service meshes used with Kubernetes include Consul, Istio, Linkerd, and Open Service Mesh. Others also exist.
Sidecar	A special container that runs alongside and augments a main app container. Service meshes are often implemented as sidecar containers that are injected into Pods and add network functionality.

Term	Definition (according to Nigel)
StatefulSet	Controller that deploys and manages stateful Pods. Similar to a Deployment, but for stateful applications.
Storage Class (SC)	Way to create different storage tiers/classes on a cluster. You may have an SC called "fast" that creates NVMe-based storage, and another SC called "medium-three-site" that creates slower storage replicated across three sites.
Volume	Generic term for persistent storage.
WebAssembly (Wasm)	Secure sandboxed virtual machine format for executing apps.
Worker node	A cluster node for running user applications. Sometimes called a "Node" or "worker".
YAML	Yet Another Markup Language. The configuration language you normally write Kubernetes configuration files in. It's a superset of JSON.

Outro

Thanks for reading my book. You're now ready to thrive in the cloud-native world.

About the front cover

I love the front cover of this book, and I'm grateful to the hundreds of people who voted on its design.

The YAML code on the left represents the technical nature of the book. The Kubernetes wheel represents the main topic. The vertical symbols on the right are cloud-native icons in the style of *digital rain* code from the Matrix movies. There's also a hidden message written in the Borg language from Star Trek.

A word on the book's diagrams

There's a great set of Kubernetes community icons available in the following GitHub repo.

```
https://github.com/kubernetes/community/tree/master/icons
```

I like them and use them extensively in blogs and video courses. However, they didn't look great in printed copies of the book. As a result, I created my own similar set for use in the book. It took a very long time to create them, so I hope you like them.

I am not trying to replace the community icons or say they aren't good. They just didn't look good in printed editions of the book.

Connect with me

I'd love to connect with you and talk about Kubernetes and other cool tech.

You can reach me at all of the following:

- Twitter: twitter.com/nigelpoulton
- LinkedIn: linkedin.com/in/nigelpoulton
- Bluesky: @nigelpoulton.com
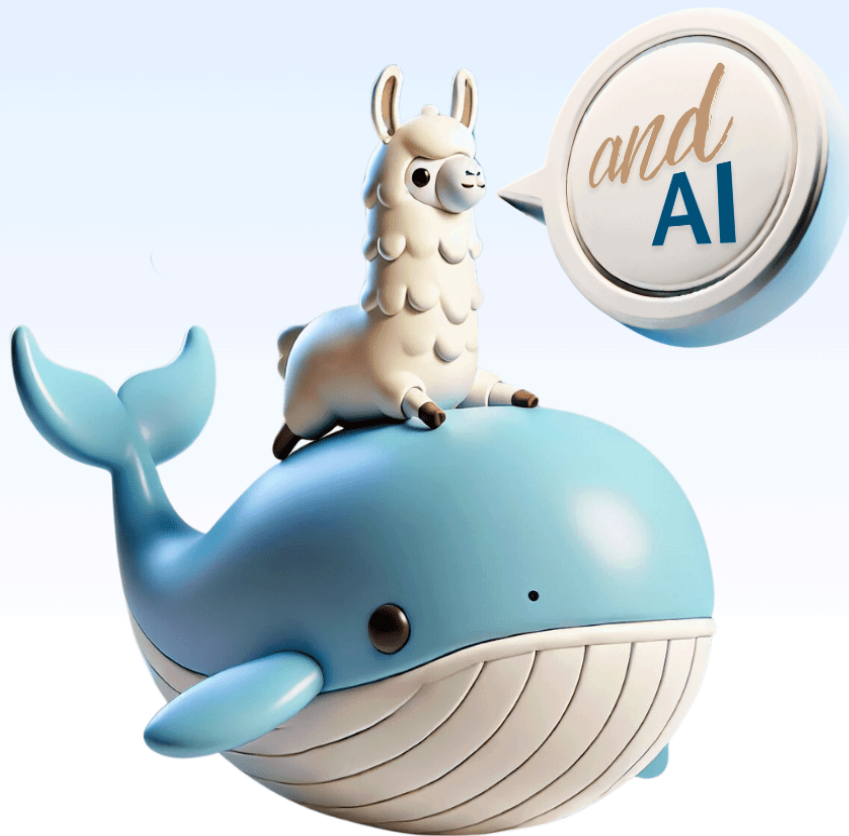- Web: nigelpoulton.com
- YouTube: youtube.com/nigelpoulton

Feedback and reviews

Books live and die by reviews and ratings.

I've spent over a year writing this book and keeping it up-to-date. So, I'd love it if you left a review on Amazon, Goodreads, or wherever you bought the book.
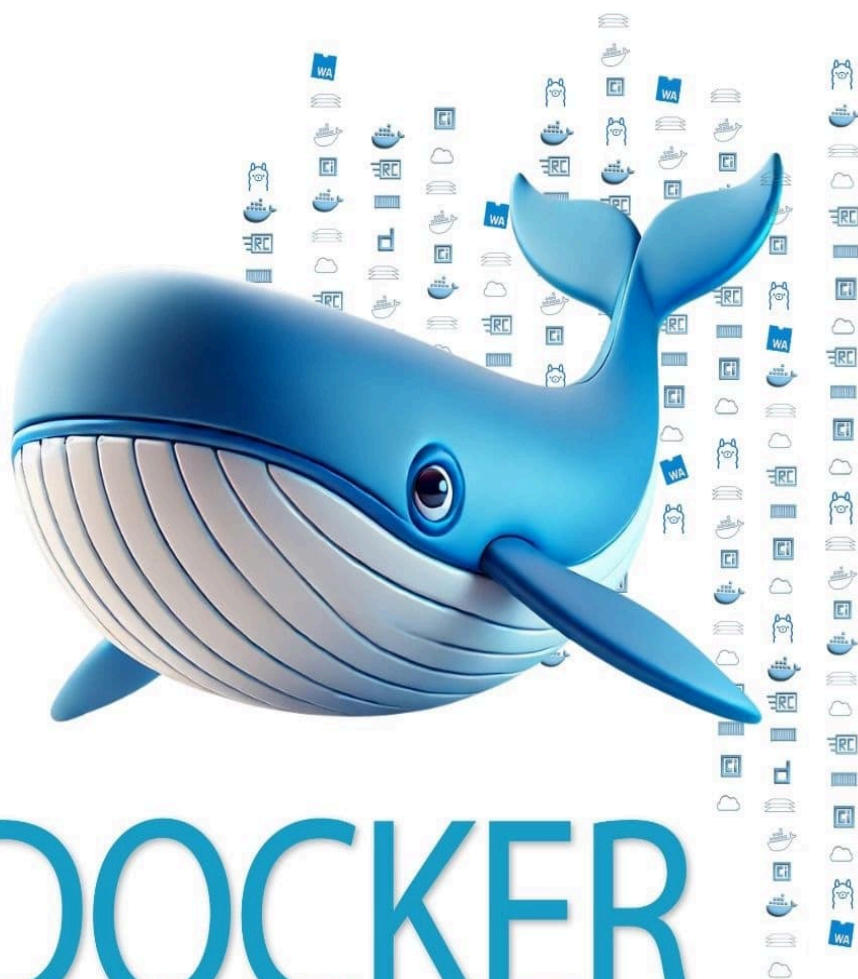
Feel free to email me at `tkb@nigelpoulton.com` if you want to suggest content or fixes.

More from the author

GETTING STARTED WITH
DOCKER
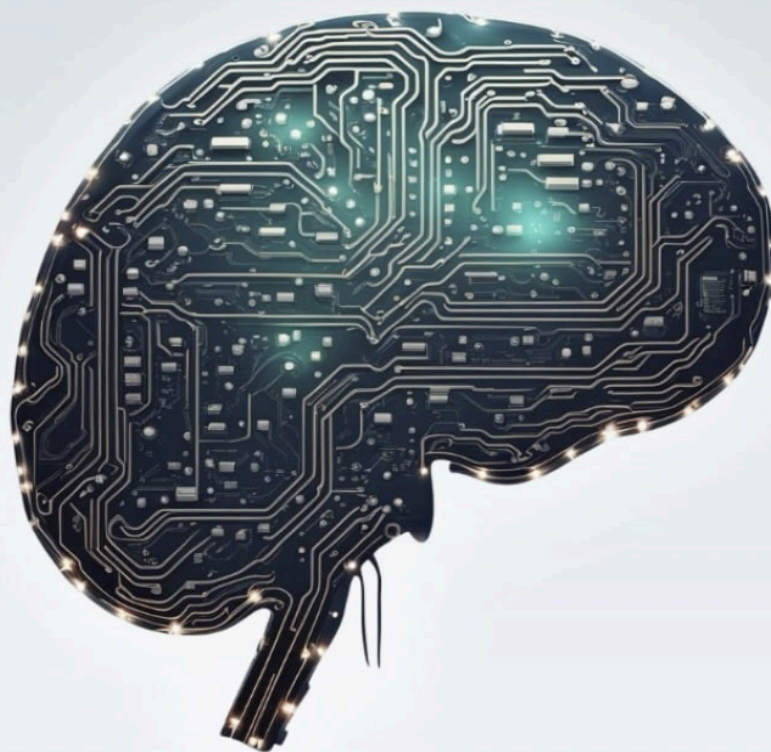### and AI

By Docker Captain
Nigel Poulton

DOCKER DEEP DIVE

ZERO TO DOCKER IN A SINGLE BOOK

By Docker Captain
Nigel Poulton

AI
EXPLAINED
Facts, Fiction & Future

Nigel Poulton