SEPT 2020 EDITION

```
$ 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

Nigel Poulton
& Pushkar Joglekar

The Kubernetes Book

Nigel Poulton

This book is for sale at http://leanpub.com/thekubernetesbook

This version was published on 2020-09-24

Tweet This Book!

Please help Nigel Poulton by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought The Kubernetes Book from @nigelpoulton and can't wait to get into this!

The suggested hashtag for this book is #kubernetes.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#kubernetes

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

Contents

0: About the book

This is an *up-to-date* book about Kubernetes that's short and straight-to-the-point.

Paperback editions

There are a few different versions of the paperback available:

- I self-publish paperback copies on Amazon in as many markets as possible
- A special-edition paperback is available for the Indian sub-continent via Shroff Publishers
- A simplified Chinese paperback is available via Posts & Telecom Press Co. LTD in China

Why is there a special paperback edition for the Indian sub-continent?

At the time of writing, the Amazon self-publishing service was not available in India. This meant I did'nt have a way to get paperback copies to readers in India. I considered several options and decided to partner with Shroff Publishers who have made a low-cost paperback available to readers in the Indian sub-continent. I'm grateful to Shroff for helping me make the book available to as many readers as possible.

Audio book

There's a highly entertaining audio version of the March 2019 edition available from Audible. This edition has a few minor tweaks to the examples and labs so that they're easier to follow in an audio book. But aside from that, you get the full experience.

eBook and Kindle editions

The easiest place to get an electronic copy is leanpub.com. It's a slick platform and updates are free and simple.

You can also get a Kindle edition from Amazon, which also entitles you to free updates. However, Kindle is notoriously bad at delivering updates. If you have problems getting updates to your Kindle edition, contact Kindle Support and they'll resolve the issue.

Feedback

If you like the book and it added value, please share the love by recommending it to a friend and leaving a review on Amazon (you can leave an Amazon review even if you bought it somewhere else).

Why should anyone read this book or care about Kubernetes?

Kubernetes is white-hot, and Kubernetes skills are in high demand. So, if you want to push ahead with your career and work with a technology that's shaping the future, you need to read this book. If you don't care about your career and are fine being left behind, don't read it. It's the truth.

Should I buy the book if I've already watched your video training courses?

Kubernetes is Kubernetes. So yes, there's obviously similarities between my books and video courses. But reading books and watching videos are totally different experiences and have very different impacts on learning. In my opinion, videos are more fun, but books are easier to make notes in and flick through when you're trying to find something.

If I was you, I'd watch the videos *and* get the book. They complement each other, and learning via multiple methods is a proven strategy.

Some of my Video courses:

- Getting Started with Kubernetes (pluralsight.com)
- Kubernetes Deep Dive (acloud.guru)
- Kubernetes 101 (nigelpoulton.com)
- Docker Deep Dive (pluralsight.com)

Updates to the book

I've done everything I can to make sure your investment in this book is maximized to the fullest extent.

All Kindle and Leanpub customers receive all updates at no extra cost. Updates work well on Leanpub, but it's a different story on Kindle. Many readers complain that their Kindle devices don't get access to updates. This is a common issue, and one that is easily resolved by **contacting Kindle Support**.

If you buy a paperback version from **Amazon.com**, you can get the Kindle version at the discounted price of $2.99. This is done via the *Kindle Matchbook* program. Unfortunately, Kindle Matchbook is only available in the US, and it's buggy — sometimes the Kindle Matchbook icon doesn't appear on the book's Amazon selling page. Contact Kindle Support if you have issues like this and they'll sort things out.

Things will be different if you buy the book through other channels, as I have no control over them. I'm a techie, not a book publisher ¯_(□)_/¯

The book's GitHub repo

The book has a GitHub repo with all of the YAML code and examples used throughout the book:

`https://github.com/nigelpoulton/TheK8sBook`

Versions of the book

Kubernetes is developing fast! As a result, the value of a book like this is inversely proportional to how old it is. Whoa, that's a mouthful. Put in other words, the older any Kubernetes book is, the less valuable it is. With this in mind, **I'm committed to updating the book at least once per year**. And when I say "update", I mean real updates — every word and concept is reviewed, and every example is tested and updated. **I'm 100% committed to making this book the best Kubernetes book in the world**.

If an update every year seems like a lot… welcome to the new normal.

We no longer live in a world where a 2-year-old technology book is valuable. In fact, I question the value of a 1-year-old book on a topic that's developing as fast as Kubernetes. Don't get me wrong, as an author I'd love to write a book that was useful for 5 years. But that's not the world we live in. Again… welcome to the new normal.

- **Version 7**: September 2020. Tested against Kubernetes1.18. Added new chapter on StatefulSets. Added glossary of terms.
- **Version 6: February 2020. All content tested with Kubernetes version 1.16.6. Added new chapter on service discovery. Removed Appendix as people thought it gave the book an unfinished feel.
- **Version 5** November 2019. All content updated and examples tested on Kubernetes 1.16.2. Added new chapter on ConfigMaps. Moved *Chapter 8* to the end as an appendix and added overview of service mesh technology to the appendix.
- **Version 4** March 2019. All content updated and all examples tested on the latest versions of Kubernetes. Added new Storage Chapter. Added new real-world security section with two new chapters.
- **Version 3** November 2018. Re-ordered some chapters for better flow. Removed the *ReplicaSets* chapter and shifted that content to an improved *Deployments* chapter. Added new chapter giving overview of other major concepts not covered in dedicated chapters.
- **Version 2.2** January 2018. Fixed a few typos, added several clarifications, and added a couple of new diagrams.
- **Version 2.1** December 2017. Fixed a few typos and updated Figures 6.11 and 6.12 to include missing labels.
- **Version 2**. October 2017. Added new chapter on ReplicaSets. Added significant changes to Pods chapter. Fixed typos and made a few other minor updates to existing chapters.
- **Version 1** July 2017. Initial version.

1: Kubernetes primer

This chapter is split into two main sections.

- Kubernetes background – where it came from etc.
- Kubernetes as the Operating System of the cloud

Kubernetes background

Kubernetes is an *application orchestrator*. For the most part, it orchestrates containerized cloud-native microservices apps. How about that for a sentence full of buzzwords!

You'll come across those terms a lot as you work with Kubernetes, so let's take a minute to explain what each one means.

What is an orchestrator

An *orchestrator* is a system that deploys and manages applications. It can deploy your applications and dynamically respond to changes. For example, Kubernetes can:

- deploy your application
- scale it up and down dynamically according to demand
- self-heal it when things break
- perform zero-downtime rolling updates and rollbacks
- and more

And the best part about Kubernetes… it can do all of that without you having to supervise or get involved in decisions. Obviously you have to set things up in the first place, but once you've done that, you can sit back and let Kubernetes work its magic.

What is a containerised app

A *containerized application* is an app that runs in a container.

Before we had containers, applications ran on physical servers or in virtual machines. Containers are the next iteration of how we package and run our apps, and they're faster, more lightweight, and more suited to modern business requirements than servers and virtual machines.

Think of it this way:

- Applications ran on physical servers in the age of open-system (roughly the 1980s and 1990s)
- Applications ran in virtual machines in the age of virtual machines (2000s and into the 2010s)
- Applications run in containers in the cloud-native era (now)

While Kubernetes can orchestrate other workload types, including virtual machines and serverless functions, it's most commonly used to orchestrate containerised apps.

What is a cloud-native app

A *cloud-native application* is an application that is designed to meet modern business demands (auto-scaling, self-healing, rolling updates etc.) and can run on Kubernetes.

I feel like it's important to be clear that cloud-native apps are not applications that will only run on a public cloud. Yes, they absolutely can run on a public cloud, but they can run anywhere that you have Kubernetes – even your on-premises data center.

What is a microservices app

A *microservices app* is a business application that is built from lots of small specialised parts that communicate and form a meaningful application. For example, you might have an e-commerce app that comprises all of the following small specialised components:

- web front-end
- catalog service
- shopping cart
- authentication service
- logging service
- persistent store
- more…

Each of these individual services is called a microservice. Typically, each can be coded and looked after by a different team, and each can have its own release cadence and can be scaled independently of all others. For example, you can patch and scale the logging microservice without affecting any of the other application components.

Building applications this way is an important aspect of a cloud-native application.

With all of this in mind, let's re-phrase that definition that was full of buzzwords…

Kubernetes deploys and manages (orchestrates) applications that are packaged and run as containers (containerized) and that are built in ways (cloud-native microservices) that allow them to scale, self-heal, and be updated inline with modern business requirements.

We'll talk about these concepts a lot throughout the book, but for now, this should help you understand some of the main industry buzzwords.

Where did Kubernetes come from

Let's start from the beginning…

Amazon Web Services (AWS) changed the world when it brought us modern-day cloud computing. Since then, everyone else has been trying to catch-up.

One of the companies trying to catch-up was Google. Google has its own very good cloud, and needs a way to abstract the value of AWS, and make it easier for potential customers to use the Google Cloud.

Google has boatloads of experience working with containers at scale. For example, huge Google applications, such as Search and Gmail, have been running at extreme scale on containers for a lot of years – since way before Docker brought us easy-to-use containers. To orchestrate and manage these containerised apps, Google had a couple of in-house proprietary systems. They took the lessons learned from these in-house systems, and created a new platform called Kubernetes, and donated it to the newly formed Cloud Native Computing Foundation (CNCF) in 2014 as an open-source project.

Figure 1.1

Since then, Kubernetes has become the most important cloud-native technology on the planet.

Like many of the modern cloud-native projects, it's written in Go (Golang), it's built in the open on GitHub (at `kubernetes/kubernetes`), it's actively discussed on the IRC channels, you can follow it on Twitter (@kubernetesio), and `slack.k8s.io` is a pretty good slack channel. There are also regular meetups and conferences all over the planet.

Kubernetes and Docker

Kubernetes and Docker are complementary technologies. For example, it's common to develop your applications with Docker and use Kubernetes to orchestrate them in production.

In this model, you write your code in your favourite languages, then use Docker to package it, test it, and ship it. But the final steps of deploying and running it is handled by Kubernetes.

At a high-level, you might have a Kubernetes cluster with 10 nodes to run your production applications. Behind the scenes, each node is running Docker as its *container runtime*. This means that Docker is the low-level technology that starts and stops the containerised applications. Kubernetes is the higher-level technology that looks after the bigger picture, such as; deciding which nodes to run containers on, deciding when to scale up or down, and executing updates.

Figure 1.2 shows a simple Kubernetes cluster with some nodes using Docker as the container runtime.

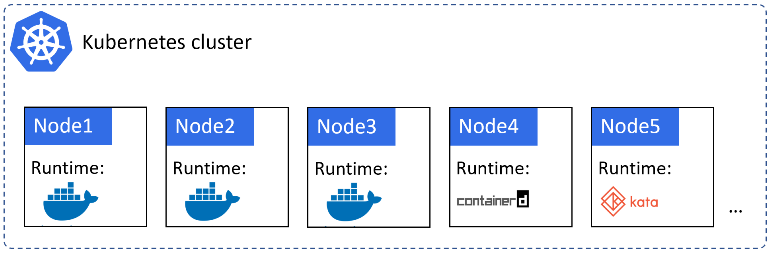


Figure 1.2

As can be seen in Figure 1.2, Docker isn't the only *container runtime* that Kubernetes supports. In fact, Kubernetes has a couple of features that abstract the container runtime (make it interchangeable):

1. The *Container Runtime Interface (CRI)* is an abstraction layer that standardizes the way 3rd-party container runtimes interface with Kubernetes. It allows the container runtime code to exist outside of Kubernetes, but interface with it in a supported and standardized way.
2. *Runtime Classes* is a new feature that was introduced in Kubernetes 1.12 and promoted to beta in 1.14. It allows for different *classes* of runtimes. For example, the *gVisor* or *Kata Containers* runtimes might provide better workload isolation than the *Docker* and *containerd* runtimes.

At the time of writing, `containerd` is catching up to Docker as the most commonly used container runtime in Kubernetes. It is a stripped-down version of Docker with just the stuff that Kubernetes needs. It's pronounced *container dee*.

While all of this is interesting, it's low-level stuff that shouldn't impact your Kubernetes learning experience. For example, whichever container runtime you use, the regular Kubernetes commands and patterns will continue to work as normal.

What about Kubernetes vs Docker Swarm

In 2016 and 2017 we had the *orchestrator wars* where Docker Swarm, Mesosphere DCOS, and Kubernetes competed to become the de-facto container orchestrator. To cut a long story short, Kubernetes won.

It's true that Docker Swarm and other container orchestrators still exist, but their development and market-share are small compared to Kubernetes.

Kubernetes and Borg: Resistance is futile!

There's a good chance you'll hear people talk about how Kubernetes relates to Google's *Borg* and *Omega* systems.

As previously mentioned, Google has been running containers at scale for a long time – apparently crunching through billions of containers a week. So yes, Google has been running things like *search*, *Gmail*, and *GFS* on **lots** of containers for a very long time.

Orchestrating these containerised apps was the job of a couple of in-house Google technologies called *Borg* and *Omega*. So, it's not a huge stretch to make the connection with Kubernetes – all three are in the game of orchestrating containers at scale, and they're all related to Google.

However, it's important to understand that Kubernetes **is not** an open-sourced version of *Borg* or *Omega*. It's more like Kubernetes shares its DNA and family history with Borg and Omega. A bit like this… In the beginning was Borg, and Borg begat Omega. Omega *knew* the open-source community and begat her Kubernetes ;-)

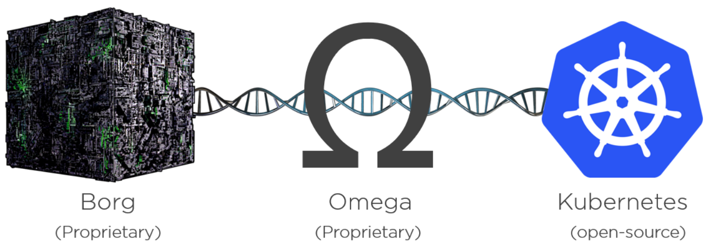


Figure 1.3 - Shared DNA

The point is, all three are separate, but all three are related. In fact, some of the people who built Borg and Omega are involved in building Kubernetes. So, although Kubernetes was built from scratch, it leverages much of what was learned at Google with Borg and Omega.

As things stand, Kubernetes is an open-source project donated to the CNCF in 2014, 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've already passed version 1.16.

Kubernetes – what's in the name

The name **Kubernetes** (koo-ber-net-eez) comes from the Greek word meaning *Helmsman* – the person who steers a seafaring ship. This theme is reflected in the logo.

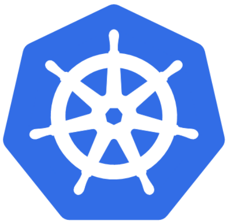


Figure 1.4 - The Kubernetes logo

Apparently, some of the people involved in the creation of Kubernetes wanted to call it *Seven of Nine*. If you know your Star Trek, you'll know that *Seven of Nine* is a female **Borg** rescued by the crew of the USS Voyager under the command of Captain Kathryn Janeway. Sadly, copyright laws prevented it from being called *Seven of Nine*. However, the seven spokes on the logo are a tip-of-the-hat to *Seven* of Nine.

One last thing about the name before moving on. You'll often see Kubernetes shortened to **K8s** (pronounced "Kates"). The number 8 replaces the 8 characters between the **K** and the **s** – great for tweets and lazy typists like me ;-)

The operating system of the cloud

Kubernetes has emerged as the de facto platform for deploying and managing cloud-native applications. In many ways, it's like an operating system (OS) for the cloud. Consider this:

- You install a traditional OS (Linux or Windows) on a server, and the OS *abstracts* the physical server's resources and *schedules* processes etc.
- You install Kubernetes on a cloud, and it *abstracts* the cloud's resources and *schedules* the various microservices of cloud-native applications

In the same way that Linux *abstracts* the hardware differences of different server platforms, Kubernetes *abstracts* the differences between different private and public clouds. Net result… as long as you're running Kubernetes, it doesn't matter if the underlying systems are on premises in your own data center, edge clusters, or in the public cloud.

With this in mind, Kubernetes enables a true *hybrid cloud*, allowing you to seamlessly move and balance workloads across multiple different public and private cloud infrastructures. You can also migrate *to* and *from* different clouds, meaning you can choose a cloud today and not have to stick with that decision for the rest of your life.

Cloud scale

Generally speaking, cloud-native microservices applications make our previous scalability challenges look easy – we've just said that Google goes through billions of containers per week!

That's great, but most of us aren't the size of Google. What about the rest of us?

Well… as a general rule, if your legacy apps have hundreds of VMs, there's a good chance your containerized cloud-native apps will have thousands of containers. With this in mind, we desperately need help managing them.

Say hello to Kubernetes.

Also, we live in a business and technology world that is increasingly fragmented and in a constant state of disruption. With this in mind, we desperately need a framework and platform that is widely accepted and hides the complexity.

Again, say hello to Kubernetes.

Application scheduling

A typical computer is a collection of CPU, memory, storage, and networking. But modern operating systems have done a great job abstracting most of that. For example, how many developers care which CPU core or exact memory address their application uses? Not many, we let the OS take care of things like that. And it's a good thing, it's made the world of application development a far friendlier place.

Kubernetes does a similar thing with cloud and data center resources. At a high-level, a cloud or data center is a pool of compute, network and storage. Kubernetes abstracts it. This means we don't have hard code which node or storage volume our applications run on, we don't even have to care which cloud they run on – we let Kubernetes take care of that. Gone are the days of naming your servers, mapping storage volumes in a spreadsheet, and otherwise treating your infrastructure assets like *pets*. Systems like Kubernetes don't care. Gone are the days of taking your app and saying *"Run this part of the app on this exact node, with this IP, on this specific volume…"*. In the cloud-native Kubernetes world, we just say *"Hey Kubernetes, here's an app. Please deploy it and make sure it keeps running…"*.

A quick analogy…

Consider the process of sending goods via a courier service.

You package the goods in the courier's standard packaging, put a label on it, and hand it over to the courier. The courier is responsible for everything. This includes; all the complex logistics of which planes and trucks it goes on, which highways to use, and who the drivers should be etc. They also provide services that let you do things like track your package and make delivery changes. The point is, the only thing that you have to do is package and label the goods, and the courier abstracts everything else and takes care of scheduling and other logistics.

It's the same for apps on Kubernetes. You package the app as a container, give it a declarative manifest, and let Kubernetes take care of deploying it and keeping it running. You also get a rich set of tools and APIs that let you introspect (observe and examine) your app. It's a beautiful thing.

Chapter summary

Kubernetes was created by Google based on lessons learned running containers at scale for many years. It was donated to the community as an open-source project and is now the industry standard API 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 easily between cloud platforms. It's open-sourced under the Apache 2.0 license and lives within the Cloud Native Computing Foundation (CNCF).

Tip!

Kubernetes is a fast-moving project under active development. But don't let this put you off – embrace it. **Change is the new normal**.

To help you keep up-to-date, I suggest you subscribe to a couple of my YouTube channels:

- #KubernetesMoment: a short weekly video discussing or explaining something about Kubernetes
- Kubernetes this Month: A monthly roundup of all the important things going on in the Kubernetes world

You should also check out:

- My website at nigelpoulton.com
- My video training courses at pluralsight.com and acloud.guru
- My hands-on learning at MSB (msb.com)
- KubeCon and your local Kubernetes and cloud-native meetups

2: Kubernetes principles of operation

In this chapter, you'll learn about the major components required to build a Kubernetes cluster and deploy an app. The aim is to give you an overview of the major concepts. So don't worry if you don't understand everything straight away, we'll cover most things again as we progress through the book.

We'll divide the chapter as follows:

- Kubernetes from 40K feet
- Masters and nodes
- Packaging apps
- Declarative configuration and desired state
- Pods
- Deployments
- Services

Kubernetes from 40K feet

At the highest level, Kubernetes is two things:

- A cluster for running applications
- An orchestrator of cloud-native microservices apps

Kubernetes as a cluster

Kubernetes is like any other cluster – a bunch of nodes and a control plane. The control plane exposes an API, has a scheduler for assigning work to nodes, and state is recorded in a persistent store. Nodes are where application services run.

It can be useful to think of the *control plane* as the brains of the cluster, and the *nodes* as the muscle. In this analogy, the control plane is the brains because it implements all of the important features such as auto-scaling and zero-downtime rolling updates. The nodes are the muscle because they do the every-day hard work of executing application code.

Kubernetes as an orchestrator

Orchestrator is just a fancy word for a system that takes care of deploying and managing applications.

Let's look at a quick analogy.

In the real world, a football (soccer) team is made of individuals. No two individuals are the same, and each has a different role to play in the team – some defend, some attack, some are great at passing, some tackle, some shoot… Along comes the coach, and he or she gives everyone a position and organizes them into a team with a purpose. We go from Figure 2.1 to Figure 2.2.

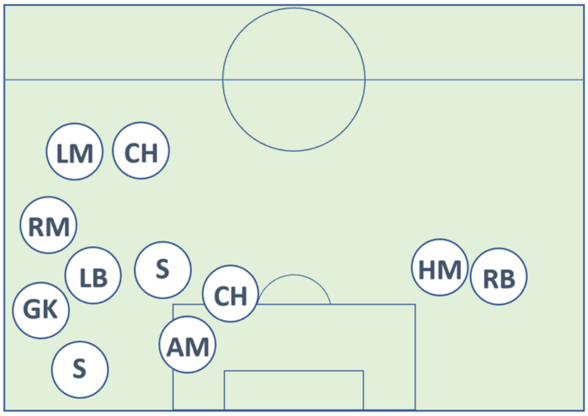


Figure 2.1

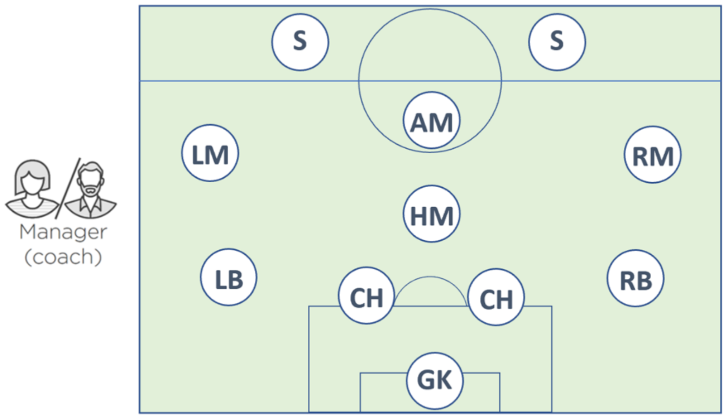


Figure 2.2

The coach also makes sure the team maintains its formation, sticks to the game-plan, and deals with any injuries and other changes in circumstance.

Well guess what… microservices apps on Kubernetes are the same.

Stick with me on this…

We start out with lots of individual specialised services. Some serve web pages, some do authentication, some perform searches, others persist data. Kubernetes comes along – a bit like the coach in the football analogy –- organizes everything into a useful app and keeps things running smoothly. It even responds to events and other changes.

In the sports world we call this *coaching*. In the application world we call it *orchestration*. Kubernetes *orchestrates* cloud-native microservices applications.

How it works

To make this happen, you start out with an app, package it up and give it to the cluster (Kubernetes). The cluster is made up of one or more *masters* and a bunch of *nodes*.

The masters, sometimes called *heads* or *head nodes*, are in-charge of the cluster. This means they make the scheduling decisions, perform monitoring, implement changes, respond to events, and more. For these reasons, we often refer to the masters as the *control plane*.

The nodes are where application services run, and we sometimes call them the *data plane*. Each node has a reporting line back to the masters, and constantly watches for new work assignments.

To run applications on a Kubernetes cluster we follow this simple pattern:

1. Write the application as small independent microservices in our favourite languages.
2. Package each microservice in its own container.
3. Wrap each container in its own Pod.
4. Deploy Pods to the cluster via higher-level controllers such as; *Deployments, DaemonSets, StatefulSets, CronJobs etc.*

Now then… we're still near the beginning of the book and you're not expected to know what all of this means yet. However, at a high-level, *Deployments* offer scalability and rolling updates, *DaemonSets* run one instance of a service on every node in the cluster, *StatefulSets* are for stateful application components, and *CronJobs* are for short-lived tasks that need to run at set times. There are more than these, but these will do for now.

Kubernetes likes to manage applications *declaratively*. This is a pattern where you describe how you want your application to look and feel in a set of YAML files. You `POST` these files to Kubernetes, then sit back while Kubernetes makes it all happen.

But it doesn't stop there. Because the declarative pattern tells Kubernetes how an application should look, Kubernetes can watch it and make sure things don't stray from what you asked for. If something isn't as it should be, Kubernetes tries to fix it.

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

Masters and nodes

A Kubernetes cluster is made of *masters* and *nodes*. These are Linux hosts that can be virtual machines (VM), bare metal servers in your data center, or instances in a private or public cloud.

Masters (control plane)

A Kubernetes *master* is a collection of system services that make up the control plane of the cluster.

The simplest setups run all the master *services* on a single host. However, this is only suitable for labs and test environments. For production environments, multi-master high availability (HA) is a **must have**. This is why the major cloud providers implement HA masters as part of their *hosted Kubernetes* platforms such as Azure Kubernetes Service (AKS), AWS Elastic Kubernetes Service (EKS), and Google Kubernetes Engine (GKE).

Generally speaking, running 3 or 5 replicated masters in an HA configuration is recommended.

It's also considered a good practice not to run user applications on masters. This allows masters to concentrate entirely on managing the cluster.

Let's take a quick look at the different master services that make up the control plane.

The API server

The API server is the Grand Central Station of Kubernetes. All communication, between all components, must go through the API server. We'll get into the detail later in the book, but it's important to understand that internal system components, as well as external user components, all communicate via the same API.

It exposes a RESTful API that you `POST` YAML configuration files to over HTTPS. These YAML files, which we sometimes call *manifests*, contain the ***desired state*** of your application. This desired state includes things like; which container image to use, which ports to expose, and how many Pod replicas to run.

All requests to the API Server are subject to authentication and authorization checks, but once these are done, the config in the YAML file is validated, persisted to the cluster store, and deployed to the cluster.

The cluster store

The cluster store is the only *stateful* part of the control plane, and it persistently stores the entire configuration and state of the cluster. As such, it's a vital component of the cluster – no cluster store, no cluster.

The cluster store is currently based on **etcd**, a popular distributed database. As it's the *single source of truth* for the cluster, you should run between 3-5 etcd replicas for high-availability, and you should provide adequate ways to recover when things go wrong.

On the topic of *availability*, etcd prefers consistency over availability. This means that it will not tolerate a split-brain situation and will halt updates to the cluster in order to maintain consistency. However, if etcd becomes unavailable, applications running on the cluster should continue to work, you just won't be able to update anything.

As with all distributed databases, consistency of writes to the database is vital. For example, multiple writes to the same value originating from different nodes needs to be handled. etcd uses the popular RAFT consensus algorithm to accomplish this.

The controller manager

The controller manager implements all of the background control loops that monitor the cluster and respond to events.

It's a *controller of controllers*, meaning it spawns all of the independent control loops and monitors them.

Some of the control loops include; the node controller, the endpoints controller, and the replicaset controller. Each one runs as a background watch-loop that is constantly watching the API Server for changes -– the aim of the game is to ensure the *current state* of the cluster matches the *desired state* (more on this shortly).

The logic implemented by each control loop is effectively this:

1. Obtain desired state
2. Observe current state
3. Determine differences

4. Reconcile differences

This logic is at the heart of Kubernetes and declarative design patterns.

Each control loop is also extremely specialized and only interested in its own little corner of the Kubernetes cluster. No attempt is made to over-complicate things by implementing awareness of other parts of the system – each control loop takes care of its own business and leaves everything else alone. This is key to the distributed design of Kubernetes and adheres to the Unix philosophy of building complex systems from small specialized parts.

> **Note**: Throughout the book we'll use terms like *control loop, watch loop,* and *reconciliation loop* to mean the same thing.

The scheduler

At a high level, the scheduler watches the API server for new work tasks and assigns them to appropriate healthy nodes. Behind the scenes, it implements complex logic that filters out nodes incapable of running the task, and then ranks the nodes that are capable. The ranking system is complex, but the node with the highest-ranking score is selected to run the task.

When identifying nodes that are capable of running a task, the scheduler performs various predicate checks. These include; is the node tainted, are there any affinity or anti-affinity rules, is the required network port available on the node, does the node have sufficient free resources etc. Any node incapable of running the task is ignored, and the remaining nodes are ranked according to things such as; does the node already have the required image, how much free resource does the node have, how many tasks is the node already running. Each criterion is worth points, and the node with the most points is selected to run the task.

If the scheduler cannot find a suitable node, the task cannot be scheduled and is marked as pending.

The scheduler isn't responsible for running tasks, just picking the nodes a task will run on.

The cloud controller manager

If you're running your cluster on a supported public cloud platform, such as AWS, Azure, GCP, DO, IBM Cloud etc. your control plane will be running a *cloud controller manager*. Its job is to manage integrations with underlying cloud technologies and services such as, instances, load-balancers, and storage. For example, if your application asks for an internet facing load-balancer, the cloud controller manager is involved in provisioning an appropriate load-balancer on your cloud platform.

Control Plane summary

Kubernetes masters run all of the cluster's control plane services. Think of it as the brains of the cluster where all the control and scheduling decisions are made. Behind the scenes, a master is made up of lots of small specialized control loops and services. These include the API server, the cluster store, the controller manager, and the scheduler.

The API Server is the front-end into the control plane and all instructions and communication must go through it. By default, it exposes a RESTful endpoint on port 443.

Figure 2.3 shows a high-level view of a Kubernetes master (control plane).

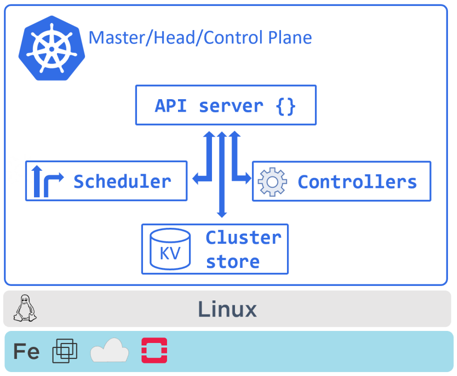


Figure 2.3 - Kubernetes Master

Nodes

Nodes are the workers of a Kubernetes cluster. At a high-level they do three things:

1. Watch the API Server for new work assignments
2. Execute new work assignments
3. Report back to the control plane (via the API server)

As we can see from Figure 2.4, they're a bit simpler than *masters*.

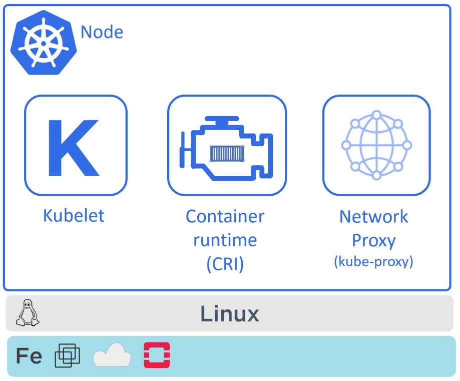


Figure 2.4 - Kubernetes Node (formerly Minion)

Let's look at the three major components of a node.

Kubelet

The Kubelet is the star of the show on every node. It's the main Kubernetes agent, and it runs on every node in the cluster. In fact, it's common to use the terms *node* and *kubelet* interchangeably.

When you join a new node to a cluster, the process installs kubelet onto the node. The kubelet is then responsible for registering the node with the cluster. Registration effectively pools the node's CPU, memory, and storage into the wider cluster pool.

One of the main jobs of the kubelet is to watch the API server for new work assignments. Any time it sees one, it executes the task and maintains a reporting channel back to the control plane.

If a kubelet can't run a particular task, it reports back to the master and lets the control plane decide what actions to take. For example, if a Kubelet cannot execute a task, it is **not** responsible for finding another node to run it on. It simply reports back to the control plane and the control plane decides what to do.

Container runtime

The Kubelet needs a *container runtime* to perform container-related tasks -– things like pulling images and starting and stopping containers.

In the early days, Kubernetes had native support for a few container runtimes such as Docker. More recently, it has moved to a plugin model called the Container Runtime Interface (CRI). At a high-level, the CRI masks the internal machinery of Kubernetes and exposes a clean documented interface for 3rd-party container runtimes to plug into.

There are lots of container runtimes available for Kubernetes. One popular example is `cri-containerd`. This is a community-based open-source project porting the CNCF `containerd` runtime to the CRI interface. It has a lot of support and is replacing Docker as the most popular container runtime used in Kubernetes.

> **Note**: `containerd` (pronounced "container-dee") is the container supervisor and runtime logic stripped out from the Docker Engine. It was donated to the CNCF by Docker, Inc. and has a lot of community support. Other CRI-compliant container runtimes exist.

Kube-proxy

The last piece of the *node* puzzle is the kube-proxy. This runs on every node in the cluster and is responsible for local cluster networking. For example, it makes sure each node gets its own unique IP address, and implements local IPTABLES or IPVS rules to handle routing and load-balancing of traffic on the Pod network.

Kubernetes DNS

As well as the various control plane and node components, every Kubernetes cluster has an internal DNS service that is vital to operations.

The cluster's DNS service has a static IP address that is hard-coded into every Pod on the cluster, meaning all containers and Pods know how to find it. Every new service is automatically registered with the cluster's DNS so that all components in the cluster can find every Service by name. Some other components that are registered with the cluster DNS are StatefulSets and the individual Pods that a StatefulSet manages.

Cluster DNS is based on CoreDNS (https://coredns.io/).

Now that we understand the fundamentals of masters and nodes, let's switch gears and look at how we package applications to run on Kubernetes.

Packaging apps for Kubernetes

For an application to run on a Kubernetes cluster it needs to tick a few boxes. These include:

1. Packaged as a container
2. Wrapped in a Pod
3. Deployed via a declarative manifest file

It goes like this… You write an application service in a language of your choice. You build it into a container image and store it in a registry. At this point, the application service is *containerized*.

Next, you define a Kubernetes Pod to run the containerized application. At the kind of high level we're at, a Pod is just a wrapper that allows a container to run on a Kubernetes cluster. Once you've defined the Pod, you're ready to deploy it on the cluster.

It is possible to run a standalone Pod on a Kubernetes cluster. But the preferred model is to deploy all Pods via higher-level controllers. The most common controller is the *Deployment*. It offers scalability, self-healing, and rolling updates. You define *Deployments* in YAML manifest files that specifies things like; which image to use and how many replicas to deploy.

Figure 2.5 shows application code packaged as a *container*, running inside a *Pod*, managed by a *Deployment* controller.

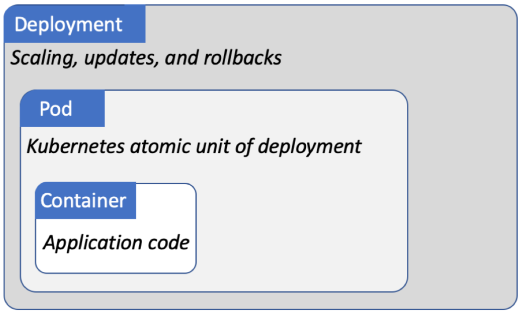

Deployment
Scaling, updates, and rollbacks
Pod
Kubernetes atomic unit of deployment
Container
Application code

Figure 2.5

Once everything is defined in the *Deployment* YAML file, you `POST` it to the API Server as the *desired state* of the application and let Kubernetes implement it.

Speaking of desired state…

The declarative model and desired state

The *declarative model* and the concept of *desired state* are at the very heart of Kubernetes.

In Kubernetes, the declarative model works like this:

1. Declare the desired state of an application (microservice) in a manifest file
2. POST it to the API server

3. Kubernetes stores it in the cluster store as the application's *desired state*
4. Kubernetes implements the desired state on the cluster
5. Kubernetes implements watch loops to make sure the *current state* of the application doesn't vary from the *desired state*

Let's look at each step in a bit more detail.

Manifest files are written in simple YAML, and they tell Kubernetes how you want an application to look. This is called the *desired state*. It includes things such as; which image to use, how many replicas to run, which network ports to listen on, and how to perform updates.

Once you've created the manifest, you `POST` it to the API server. The most common way of doing this is with the `kubectl` command-line utility. This sends the manifest to the control plane as an HTTP POST, usually on port 443.

Once the request is authenticated and authorized, Kubernetes inspects the manifest, identifies which controller to send it to (e.g. the *Deployments controller*), and records the config in the cluster store as part of the cluster's overall *desired state*. Once this is done, the work gets scheduled on the cluster. This includes the hard work of pulling images, starting containers, building networks, and starting the application's processes.

Finally, Kubernetes utilizes background reconciliation loops that constantly monitor the state of the cluster. If the *current state* of the cluster varies from the *desired state*, Kubernetes will perform whatever tasks are necessary to reconcile the issue.

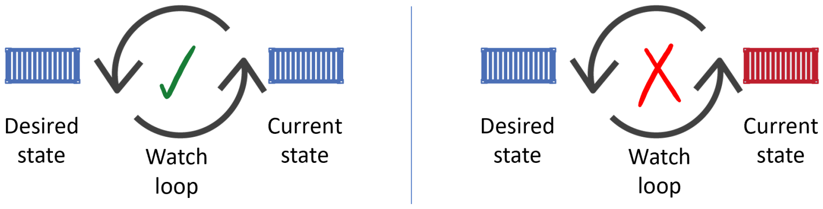


Figure 2.6

It's important to understand that what we've described is the opposite of the traditional *imperative model*. The imperative model is where you issue long lists of platform-specific commands to build things.

Not only is the declarative model a lot simpler than long scripts with lots of imperative commands, it also enables self-healing, scaling, and lends itself to version control and self-documentation. It does this by telling the cluster *how things should look*. If they stop looking like this, the cluster notices the discrepancy and does all of the hard work to reconcile the situation.

But the declarative story doesn't end there – things go wrong, and things change. When they do, the ***current state*** of the cluster no longer matches the ***desired state***. As soon as this happens, Kubernetes kicks into action and attempts to bring the two back into harmony.

Let's consider an example.

Declarative example

Assume you have an app with a desired state that includes 10 replicas of a web front-end Pod. If a node that was running two replicas fails, the *current state* will be reduced to 8 replicas, but the *desired state* will still be 10. This will be observed by a reconciliation loop and Kubernetes will schedule two new replicas to bring the total back up to 10.

The same thing will happen if you intentionally scale the desired number of replicas up or down. You could even change the image you want to use. For example, if the app is currently using `v2.00` of an image, and you update the desired state to use `v2.01`, Kubernetes will notice the difference and go through the process of updating all replicas so that they are using the new version specified in the new *desired state*.

To be clear. Instead of writing a long list of commands to go through the process of updating every replica to the new version, you simply tell Kubernetes you want the new version, and Kubernetes does the hard work for us.

Despite how simple this might seem, it's extremely powerful and at the very heart of how Kubernetes operates. You give Kubernetes a declarative manifest that describes how you want an application to look. This forms the basis of the application's desired state. The Kubernetes control plane records it, implements it, and runs background reconciliation loops that constantly check that what is running is what you've asked for. When current state matches desired state, the world is a happy place. When it doesn't, Kubernetes gets busy fixing it.

Pods

In the VMware world, the atomic unit of scheduling is the virtual machine (VM). In the Docker world, it's the container. Well… in the Kubernetes world, it's the ***Pod***.

It's true that Kubernetes runs containerized apps. However, you cannot run a container directly on a Kubernetes cluster – containers must **always** run inside of Pods.

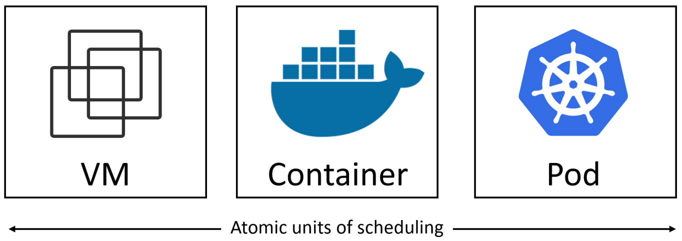


Figure 2.7

Pods and containers

The very first thing to understand is that the term *Pod* comes from a *pod of whales* – in the English language we call a group of whales a *pod of whales*. As the Docker logo is a whale, it makes sense that we call a group of containers a *Pod*.

The simplest model is to run a single container per Pod. However, there are advanced use-cases that run multiple containers inside a single Pod. These *multi-container Pods* are beyond the scope of what we're discussing here, but powerful examples include:

- Service meshes
- Web containers supported by a *helper* container that pulls the latest content
- Containers with a tightly coupled log scraper

The point is, a Kubernetes Pod is a construct for running one or more containers. Figure 2.8 shows a multi-container Pod.

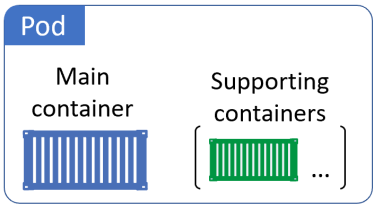


Figure 2.8

Pod anatomy

At the highest-level, a *Pod* is a ring-fenced environment to run containers. The Pod itself doesn't actually run anything, it's just a sandbox for hosting containers. Keeping it high level, you ring-fence an area of the host OS, build a network stack, create a bunch of kernel namespaces, and run one or more containers in it. That's a Pod.

If you're running multiple containers in a Pod, they all share the **same Pod environment**. This includes things like the IPC namespace, shared memory, volumes, network stack and more. As an example, this means that all containers in the same Pod will share the same IP address (the Pod's IP). This is shown in Figure 2.9.

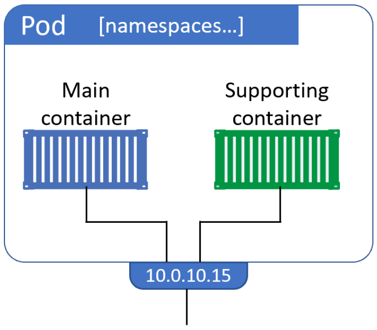


Figure 2.9

If two containers in the same Pod need to talk to each other (container-to-container within the Pod) they can use ports on the Pod's `localhost` interface as shown in Figure 2.10.

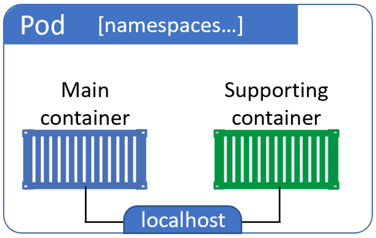


Figure 2.10

Multi-container Pods are ideal when you have requirements for tightly coupled containers that may need to share memory and storage. However, if you don't **need** to tightly couple your containers, you should put them in their own Pods and loosely couple them over the network. This keeps things clean by having each Pod dedicated to a single task. It also creates a lot of network traffic that is un-encrypted. You should seriously consider using a service mesh to secure traffic between Pods and application services.

Pods as the unit of scaling

Pods are also the minimum unit of scheduling in Kubernetes. If you need to scale your app, you add or remove Pods. You **do not** scale by adding more containers to an existing Pod. Multi-container Pods are only for situations where two different, but complimentary, containers need to share resources. Figure 2.11 shows how to scale the `nginx` front-end of an app using multiple Pods as the unit of scaling.

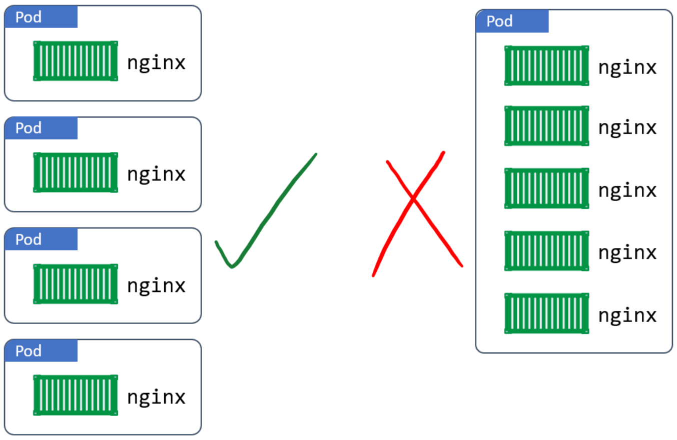


Figure 2.11 - Scaling with Pods

Pods - atomic operations

The deployment of a Pod is an atomic operation. This means that a Pod is only considered ready for service when all of its containers are up and running. There is never a situation where a partially deployed Pod will service requests. The entire Pod either comes up and is put into service, or it doesn't, and it fails.

A single Pod can only be scheduled to a single node. This is also true of multi-container Pods – all containers in the same Pod will run on the same node.

Pod lifecycle

Pods are mortal. They're created, they live, and they die. If they die unexpectedly, you don't bring them back to life. Instead, Kubernetes starts a new one in its place. However, even though the new Pod looks, smells, and feels like the old one, it isn't. It's a shiny new Pod with a shiny new ID and IP address.

This has implications on how you should design your applications. Don't design them so they are tightly coupled to a particular instance of a Pod. Instead, design them so that when Pods fail, a totally new one (with a new ID and IP address) can pop up somewhere else in the cluster and seamlessly take its place.

Deployments

Most of the time you'll deploy Pods indirectly via a higher-level controller. Examples of higher-level controllers include; *Deployments*, *DaemonSets*, and *StatefulSets*.

For example, a Deployment is a higher-level Kubernetes object that wraps around a particular Pod and adds features such as scaling, zero-downtime updates, and versioned rollbacks.

Behind the scenes, Deployments, DaemonSets and StatefulSets implement a controller and a watch loop that is constantly observing the cluster making sure that current state matches desired state.

Deployments have existed in Kubernetes since version 1.2 and were promoted to GA (stable) in 1.9. You'll see them a lot.

Services and network stable networking

We've just learned that Pods are mortal and can die. However, if they're managed via Deployments or DaemonSets, they get replaced when they fail. But replacements come with totally different IP addresses. This also happens when you perform scaling operations – scaling up adds new Pods with new IP addresses, whereas scaling down takes existing Pods away. Events like these cause a lot of *IP churn*.

The point I'm making is that **Pods are unreliable**, which poses a challenge… Assume you've got a microservices app with a bunch of Pods performing video rendering. How will this work if other parts of the app that need to use the rendering service cannot rely on the rendering Pods being there when they need them?

This is where *Services* come in to play. **Services provide reliable networking for a set of Pods**.

Figure 2.12 shows the uploader microservice talking to the renderer microservice via a Kubernetes Service object. The Kubernetes Service is providing a reliable name and IP, and is load-balancing requests to the two renderer Pods behind it.

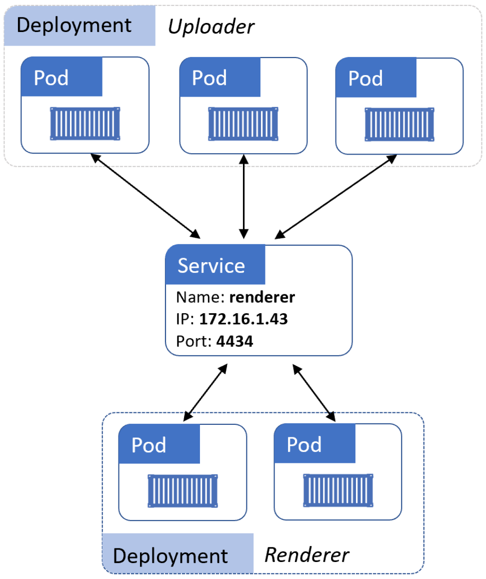


Figure 2.12

Digging into a bit more detail. Services are fully-fledged objects in the Kubernetes API – just like Pods and Deployments. They have a front-end that consists of a stable DNS name, IP address, and port. On the back-end, they load-balance across a dynamic set of Pods. As Pods come and go, the Service observes this, automatically updates itself, and continues to provide that stable networking endpoint.

The same applies if you scale the number of Pods up or down. New Pods are seamlessly added to the Service and will receive traffic. Terminated Pods are seamlessly removed from the Service and will not receive traffic.

That's the job of a Service – it's a stable network abstraction point that provides TCP and UDP load-balancing across a dynamic set of Pods.

As they operate at the TCP and UDP layer, Services do not possess application intelligence and cannot provide application-layer host and path routing. For that, you need an Ingress, which understands HTTP and provides host and path-based routing.

Connecting Pods to Services

Services use *labels* and a *label selector* to know which set of Pods to load-balance traffic to. The Service has a *label selector* that is a list of all the *labels* a Pod must possess in order for it to receive traffic from the Service.

Figure 2.13 shows a Service configured to send traffic to all Pods on the cluster tagged with the following three labels:

- zone=prod
- env=be
- ver=1.3

Both Pods in the diagram have all three labels, so the Service will load-balance traffic to them.

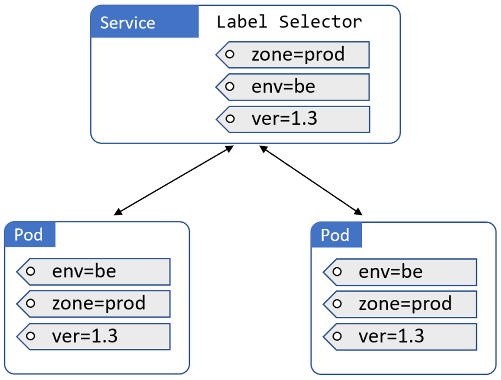


Figure 2.13

Figure 2.14 shows a similar setup. However, an additional Pod, on the right, does not match the set of labels configured in the Service's label selector. This means the Service will not load balance requests to it.

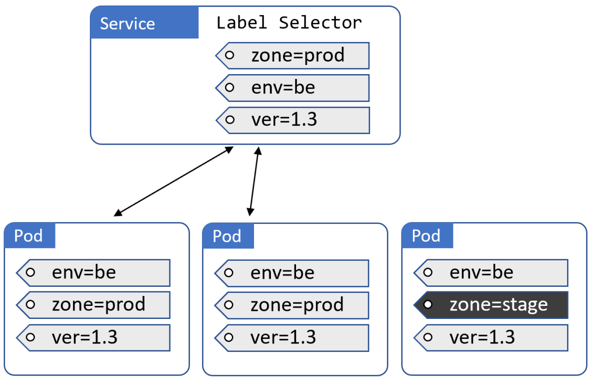


Figure 2.14

One final thing about Services. They only send traffic to **healthy Pods**. This means a Pod that is failing health-checks will not receive traffic from the Service.

That's the basics. Services bring stable IP addresses and DNS names to the unstable world of Pods.

Chapter summary

In this chapter, we introduced some of the major components of a Kubernetes cluster.

The masters are where the control plane components run. Under-the-hood, there are several system-services, including the API Server that exposes a public REST interface to the cluster. Masters make all of the deployment and scheduling decisions, and multi-master HA is important for production-grade environments.

Nodes are where user applications run. Each node runs a service called the `kubelet` that registers the node with the cluster and communicates with the API Server. It watches the API for new work tasks and maintains a reporting channel. Nodes also have a container runtime and the `kube-proxy` service. The container runtime, such as Docker or containerd, is responsible for all container-related operations. The `kube-proxy` is responsible for networking on the node.

We also talked about some of the major Kubernetes API objects such as Pods, Deployments, and Services. The Pod is the basic building-block. Deployments add self-healing, scaling and updates. Services add stable networking and load-balancing.

Now that we've covered the basics, let's get into the detail.

3: Installing Kubernetes

In this chapter, we'll look at a few quick ways to install Kubernetes.

There are three typical ways of getting a Kubernetes:

1. Test playground
2. Hosted Kubernetes
3. DIY installation

Kubernetes playgrounds

Test playgrounds are the simplest ways to get Kubernetes, but they're not intended for production. Common examples include *Magic Sandbox (msb.com)*, *Play with Kubernetes (https://labs.play-with-k8s.com/)*, and *Docker Desktop*.

With Magic Sandbox, you register for an account and login. That's it, you've instantly got a fully working multi-node private cluster that's ready to go. You also get curated lessons and hands-on labs.

Play with Kubernetes requires you to login with a GitHub or Docker Hub account and follow a few simple steps to build a cluster that lasts for 4 hours.

Docker Desktop is a free desktop application from Docker, Inc. You download and run the installer, and after a few clicks you've got a single-node development cluster on your laptop.

Hosted Kubernetes

Most of the major cloud platforms now offer their own *hosted Kubernetes* services. In this model, control plane (masters) components are managed by your cloud platform. For example, your cloud provider makes sure the control plane is highly available, performant, and handles all control plane upgrades. On the flipside, you have less control over versions and have limited options to customise.

Irrespective of pros and cons, *hosted Kubernetes* services are as close to zero-effort production-grade Kubernetes as you will get. In fact, the Google Kubernetes Engine (GKE) lets you deploy a production-grade Kubernetes cluster and the Istio service mesh with just a few simple clicks. Other clouds offer similar services:

- AWS: Elastic Kubernetes Service (EKS)
- Azure: Azure Kubernetes Service (AKS)
- Linode: Linode Kubernetes Engine (LKE)
- DigitalOcean: DigitalOcean Kubernetes
- IBM Cloud: IBM Cloud Kubernetes Service
- Google Cloud Platform: Google Kubernetes Engine (GKE)

With these offerings in mind, ask yourself the following question before building your own Kubernetes cluster: *Is building and managing your own Kubernetes cluster the best use of your time and other resources?* If the answer isn't "**Hell yes**", I strongly suggest you ***consider*** a hosted service.

DIY Kubernetes clusters

By far the hardest way to get a Kubernetes cluster is to build one yourself.

Yes, DIY installations are a lot easier than they used to be, but they're still hard. However, they provide the most flexibility and give you ultimate control over your configuration – which can be a good thing and a bad thing.

Installing Kubernetes

There are a ridiculous number of different ways to get a Kubernetes cluster and we're not trying to show them all (there are probably hundreds). The methods shown here are simple and I've chosen them because they're quick and easy ways to get a Kubernetes cluster that you can follow to most of the examples with.

All of the examples will work on Magic Sandbox and GKE, and *most* of them will work on other installations. Ingress examples and volumes may not work on platforms like Docker Desktop and Play with Kubernetes.

We'll look at the following:

- Play with Kubernetes (PWK)
- Docker Desktop: local development cluster on your laptop
- Google Kubernetes Engine (GKE): production-grade hosted cluster

Play with Kubernetes

Play with Kubernetes (PWK) is a quick and simple way to get your hands on a development Kubernetes cluster. All you need is a computer, an internet connection, and an account on Docker Hub or GitHub.

However, it has a few of limitations to be aware of.

- It's time-limited – you get a cluster that lasts for 4 hours
- It lacks some integrations with external services such as cloud-based load-balancers and volumes
- It often suffers from capacity issues (it's offered as a free service)

Let's see what it looks like (the commands may be slightly different).

1. Point your browser at https://https://labs.play-with-k8s.com/
2. Login with your GitHub or Docker Hub account and click `Start`
3. Click `+ ADD NEW INSTANCE` from the navigation pane on the left of your browser

   You will be presented with a terminal window in the right of your browser. This is a Kubernetes node (`node1`).
4. Run a few commands to see some of the components pre-installed on the node.

```
$ docker version
Docker version 19.03.11-ce...

$ kubectl version --output=yaml
clientVersion:
...
  major: "1"
  minor: "18"
```

As the output shows, the node already has Docker and `kubectl` (the Kubernetes client) pre-installed. Other tools, including `kubeadm`, are also pre-installed. More on these tools later.

It's also worth noting that although the command prompt is a `$`, you're actually running as `root`. You can confirm this by running `whoami` or `id`.

5. Run the provided `kubeadm init` command to initialize a new cluster

   When you added a new instance in step 3, PWK gave you a short list of commands to initialize a new Kubernetes cluster. One of these was `kubeadm init...`. Running this command will initialize a new cluster and configure the API server to listen on the correct IP interface.

   You may be able to specify the version of Kubernetes to install by adding the `--kubernetes-version` flag to the command. The latest versions can be seen at https://github.com/kubernetes/kubernetes/releases. Not all versions work with PWK.

```
$ kubeadm init --apiserver-advertise-address $(hostname -i) --pod-network-cidr...
[kubeadm] WARNING: kubeadm is in beta, do not use it for prod...
[init] Using Kubernetes version: v1.18.8
[init] Using Authorization modes: [Node RBAC]
<Snip>
Your Kubernetes master has initialized successfully!
<Snip>
```

   Congratulations! You have a brand new single-node Kubernetes cluster. The node that you executed the command from (`node1`) is initialized as the *master*.

   The output of the `kubeadm init` gives you a short list of commands it wants you to run. These will copy the Kubernetes config file and set permissions. You can ignore these, as PWK has already configured them for you. Feel free to poke around inside of `$HOME/.kube`.

6. Verify the cluster with the following `kubectl` command.

```
$ kubectl get nodes
NAME      STATUS     ROLES    AGE       VERSION
node1     NotReady   master   1m        v1.18.4
```

   The output shows a single-node Kubernetes cluster. However, the status of the node is `NotReady`. This is because you haven't configured the *Pod network* yet. When you first logged on to the PWK node, you were given three commands to configure the cluster. So far, you've only executed the first one (`kubeadm init...`).

7. Initialize the Pod network (cluster networking).

   Copy the second command from the list of three commands that were printed on the screen when you first created `node1` (it will be a `kubectl apply` command). Paste it onto a new line in the terminal. In the book, the command may wrap over multiple lines and insert backslashes (\).

```
$ kubectl apply -f https://raw.githubusercontent.com...
configmap/kube-router-cfg created
daemonset.apps/kube-router created
serviceaccount/kube-router created
clusterrole.rbac.authorization.k8s.io/kube-router created
clusterrolebinding.rbac.authorization.k8s.io/kube-router created
```

8. Verify the cluster again to see if `node1` has changed to `Ready` (it may take a few seconds to transition to ready).

   ```
   $ kubectl get nodes
   NAME      STATUS    ROLES    AGE       VERSION
   node1     Ready     master   2m        v1.18.4
   ```

   Now that the *Pod network* has been initialized and the control plane is `Ready`, you're ready to add some worker nodes.

9. Copy the long `kubeadm join` that was displayed as part of the output from the `kubeadm init` command.

   When you initialized the new cluster with `kubeadm init`, the final output of the command listed a `kubeadm join` command to use when adding nodes. This command includes the cluster join-token, the IP socket that the API server is listening on, and other bits required to join a new node to the cluster. Copy this command and be ready to paste it into the terminal of a new node (`node2`).

10. Click the `+ ADD NEW INSTANCE` button in the left pane of the PWK window.

You'll be given a new node called `node2`.

1. Paste the `kubeadm join` command into the terminal of `node2`.

The join-token and IP address will be different in your environment.

```
$ kubeadm join --token 948f32.79bd6c8e951cf122 10.0.29.3:6443...
Initializing machine ID from random generator.
[preflight] Skipping pre-flight checks
<Snip>
Node join complete:
* Certificate signing request sent to master and response received.
* Kubelet informed of new secure connection details.
```

1. Switch back to `node1` and run another `kubectl get nodes`

```
$ kubectl get nodes
NAME      STATUS    ROLES     AGE       VERSION
node1     Ready     master    5m        v1.18.4
node2     Ready     <none>    1m        v1.18.4
```

Your Kubernetes cluster now has two nodes – one master and one worker node.

Feel free to add more nodes.

Congratulations! You have a fully working Kubernetes cluster that you can use as a test lab.

It's worth pointing out that `node1` was initialized as the Kubernetes *master* and additional nodes will join the cluster as * worker nodes. *PWK usually puts a blue icon next to *masters* and a transparent one next to *nodes*. This helps you identify which is which.

Finally, PWK sessions only last for 4 hours and are obviously not intended for production use.

Have fun!

Docker Desktop

Docker Desktop is a great way to get a local development cluster on your Mac or Windows laptop. With a few easy steps, you get a single-node Kubernetes cluster that you can develop and test with.

It works by creating a virtual machine (VM) on your laptop and starting a single-node Kubernetes cluster inside that VM. It also configures a `kubectl` client with a context that allows it to talk to the cluster. Finally, you get a simple GUI that lets you perform basic operations such as switching between all of your `kubectl` contexts.

> **Note**: A kubectl context is a bunch of settings that `kubectl` which cluster to send commands to and which credentials to authenticate with.

1. Point your web browser to `www.docker.com` and choose `Products > Desktop`.
2. Follow the links to download for either Mac or Windows.

   You may need to login to the Docker Store. Accounts are free, and so is the product.
3. Open the installer and follow the simple installation instructions.

   Once the installer is complete, you'll get a whale icon on the Windows task bar, or the menu bar on a Mac.
4. Click the whale icon (you may need to right-click it), go to `Settings` and enable Kubernetes from the `Kubernetes` tab.

You can open a terminal window and see your cluster:

```
$ kubectl get nodes
NAME                 STATUS   ROLES    AGE   VERSION
docker-for-desktop   Ready    master   28d   v1.16.6
```

Congratulations, you now have a local development cluster.

Google Kubernetes Engine (GKE)

Google Kubernetes Engine is a *hosted Kubernetes* service that runs on the Google Cloud Platform (GCP). Like most *hosted Kubernetes* services, it provides:

- A fast and easy way to get a production-grade Kubernetes cluster
- A managed control plane (you do not manage the *masters*)
- Itemized billing

> **Warning**: GKE and other hosted Kubernetes services are not free. Some services might provide a *free tier* or an amount of *initial free credit*. However, generally speaking, you have to pay to use them.

Configuring GKE

To work with GKE you'll need an account on the Google Cloud with billing configured and a blank project. These are all simple to setup, so we won't spend time explaining them here – for the remainder of this section we'll be assuming you have these.

The following steps will walk you through configuring GKE via a web browser. Some of the details might change in the future, but the overall flow will be the same.

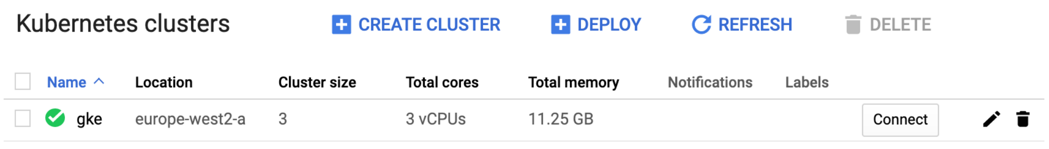
1. From within the Console of your Google Cloud Platform (GCP) project, open the navigation pane on the left-hand side and select `Kubernetes Engine > Clusters`. You may have to click the three horizontals bars (hamburger) at the top left of the Console to make the navigation pane visible.
2. Click the `Create cluster` button.

   This will start the wizard to create a new Kubernetes cluster.
3. Give the cluster a meaningful name and description.
4. Choose whether you want a `Regional` or `Zonal` cluster. Regional is newer and potentially more resilient – your masters and nodes will be distributed across multiple zones but still accessible via a single highly-available endpoint.
5. Choose the Region or Zone for your cluster.
6. Select the `Master version`. This is the version of Kubernetes that will run on your master and nodes. You are limited to the versions available in the drop-down lists. Choose an up-to-date version.

   An alternative to setting the `Master version` is to choose a *release channel* that influences the way your cluster will be upgraded to new releases.
7. At this point you can specify more advanced options available in the left pane. These include things such as whether Nodes will run Docker or containrd, and whether or not to enable the Istio service mesh. It's worth exploring these options, but you can leve all them all as defaults.
8. Once you're happy with your options, click `Create`.

Your cluster will now be created.

The clusters page shows a high-level overview of the Kubernetes clusters you have in your project. Figure 3.1 shows a single 3-node cluster called `gke`.



Name	Location	Cluster size	Total cores	Total memory	Notifications	Labels
gke	europe-west2-a	3	3 vCPUs	11.25 GB		

Figure 3.1

You can click the cluster name to drill into more detail.

Clicking the `> CONNECT` icon towards the top of the web UI gives you a command you can run on your laptop to configure `kubectl` to talk to your cluster. Copy this command to your clipboard.

The following step will only work if you have the `gcloud` and `kubectl` downloaded and installed. They can both be installed from here `https://cloud.google.com/sdk/`.

Once you have `gcloud` installed and configured, open a terminal and paste the long `gcloud` command into it. This will configure your `kubectl` client to talk to your new GKE cluster.

Run a `kubectl get nodes` command to list the nodes in the cluster.

```
$ kubectl get nodes
NAME             STATUS     AGE    VERSION
gke-cluster...   Ready      5m     v1.17.9-gke.1503
gke-cluster...   Ready      6m     v1.17.9-gke.1503
gke-cluster...   Ready      6m     v1.17.9-gke.1503
```

Congratulations! You know how to create a production-grade Kubernetes cluster using Google Kubernetes Engine (GKE). You also know how to inspect it and connect to it.

You can use this cluster to follow along with the examples in the book. However, be sure to delete your GKE cluster as soon as you are finished using it. GKE, and other hosted K8s platforms, may incur costs even when they are not in use.

Other installation methods

As previously stated, there are lots of ways to install Kubernetes. These include:

- kops
- kubeadm

kops is an opinionated tool for installing Kubernetes on AWS. The term *opinionated* means that it wants to configure Kubernetes in a particular way and doesn't let you customise very much. If you need more freedom with your installation you might want to consider **kubeadm**.

Previous versions of the book dedicated multiple tens of pages to each method. However, the material was extremely dry and difficult to follow. In this version I recommend readers follow online guides to build Kubernetes with either **kops** or **kubeadm**.

kubectl

`kubectl` is the main Kubernetes command-line tool and is what you will use for your day-to-day Kubernetes management activities. It might be useful to think of `kubectl` as *SSH for Kubernetes*. It's available for Linux, Mac and Windows.

As it's the main command-line tool, it's important that you use a version that is no more than one minor version higher or lower than your cluster. For example, if your cluster is running Kubernetes 1.18.x, your `kubectl` should be between 1.17.x and 1.19.x.

At a high-level, `kubectl` converts user-friendly commands into the JSON payload required by the API server. It uses a configuration file to know which cluster and API server endpoint to `POST` commands to.

The `kubectl` configuration file is called `config` and lives in `$HOME/.kube`. It contains definitions for:

- Clusters
- Users (credentials)
- Contexts

Clusters is a list of clusters that `kubectl` knows about and is ideal if you plan on using a single workstation to manage multiple clusters. Each cluster definition has a name, certificate info, and API server endpoint.

Users let you define different users that might have different levels of permissions on each cluster. For example, you might have a *dev* user and an *ops* user, each with different permissions. Each *user* definition has a friendly name, a username, and a set of credentials.

Contexts bring together clusters and users under a friendly name. For example, you might have a context called `deploy-prod` that combines the `deploy` user credentials with the `prod` cluster definition. If you use `kubectl` with this context you will be POSTing commands to the API server of the `prod` cluster as the `deploy` user.

The following is a simple `kubectl` config file with a single cluster called `minikube`, a single user called `minikube`, and a single context called `minikube`. The `minikube` context combines the `minikube` user and cluster, and is also set as the default context.

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: C:\Users\nigel\.minikube\ca.crt
    server: https://192.168.1.77:8443
  name: minikube
contexts:
- context:
    cluster: minikube
    user: minikube
  name: minikube
current-context: minikube
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate: C:\Users\nigel\.minikube\client.crt
    client-key: C:\Users\nigel\.minikube\client.key
```

You can view your `kubectl` config using the `kubectl config view` command. Sensitive data will be redacted from the output.

You can use `kubectl config current-context` to see your current context. The following example shows a system where `kubectl` is configured to issue commands to a cluster that is called `eks-k8sbook`.

```
$ kubectl config current-context
eks_k8sbook
```

You can change the current/active context with `kubectl config use-context`. The following command will set the current context to `docker-desktop` so that future commands will be sent to the cluster defined in the `docker-desktop` context. It obviously requires that a context called `docker-desktop` exists in the `kubectl` config file.

```
$ kubectl config use-context docker-desktop
Switched to context "docker-desktop".

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

Chapter summary

In this chapter, you saw a few ways to get a Kubernetes cluster.

You saw how fast and simple it is to setup a Kubernetes cluster on Play with Kubernetes (PWK) where you get a 4-hour playground without having to install anything on your laptop or in your cloud.

You saw how to setup Docker Desktop for a great single-node developer experience on our laptops.

You learned how to spin up a managed/hosted Kubernetes cluster in the Google Cloud using Google Kubernetes Engine (GKE).

The chapter finished up with an overview of `kubectl`, then Kubernetes command-line tool.

4: Working with Pods

We'll split this chapter into two main parts:

- Theory
- Hands-on

Let's crack on with the theory.

Pod theory

The atomic unit of scheduling in the virtualization world is the Virtual Machine (VM). This means **deploying applications** in the virtualization world is done by scheduling them on VMs.

In the Docker world, the atomic unit is the container. This means **deploying applications** on Docker is done by scheduling them inside of containers.

In the Kubernetes world, the atomic unit is the *Pod*. Ergo, **deploying applications** on Kubernetes means stamping them out in Pods.

This is fundamental to understanding Kubernetes, so be sure to tag it in your brain as important >> Virtualization does VMs, Docker does containers, and **Kubernetes does Pods**.

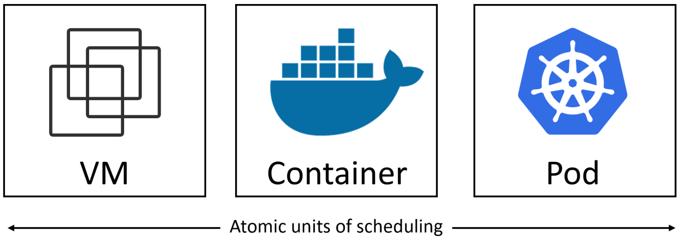


Figure 4.1

As Pods are the fundamental unit of deployment in Kubernetes, it's vital you understand how they work.

> **Note**: We're going to talk a lot about Pods in this chapter. However, don't lose sight of the fact that Pods are just a vehicle for **deploying applications**.

Pods vs containers

In a previous chapter we said that a Pod hosts one or more containers. From a footprint perspective, this puts Pods somewhere in between containers and VMs – they're a tiny bit bigger than a container, but a lot smaller than a VM.

Digging a bit deeper, a Pod is a *shared execution environment* for one or more containers.

The simplest model is the one-container-per-Pod model. However, multi-container Pods are gaining in popularity and are important for advanced configurations.

An application-centric use-case for multi-container Pods is co-scheduling tightly-coupled workloads. For example, two containers that share memory won't work if they are scheduled on different nodes in the cluster. By putting both containers inside the same Pod, you ensure that they are scheduled to the same node and share the same execution environment.

An infrastructure-centric use-case for multi-container Pods is a service mesh. In the service mesh model, a proxy container is inserted into every application Pod. This proxy container handles all network traffic entering and leaving the Pod, meaning it is ideally placed to implement features such as traffic encryption, network telemetry, intelligent routing, and more.

Multi-container Pods: the typical example

A common example for comparing single-container and multi-container Pods is a web server that utilizes a file synchronizer.

In this example there are two clear *concerns*:

1. Serving the web page
2. Making sure the content is up to date

The question is whether to address the two concerns in a single container or two separate containers.

In this context, a *concern* is a requirement or a task. Generally speaking, microservices design patterns dictate that we *separate concerns*. This means we only every deal with one concern per container.

Assuming the previous example, this will require two containers: one for the web service, another for the file-sync service.

This model of separating concerns has a lot of advantages, including:

- Different teams can be responsible for each of the two containers
- Each container can be scaled independently
- Each container can be developed and iterated independently
- Each container can have its own release cadence
- If one fails, the other keeps running

Despite the benefits of separating concerns, it's often a requirement to co-schedule those separate containers in a single Pod. This makes sure both containers are scheduled to the same node and share the same execution environment (the Pod's environment).

Common use-cases for multi-container Pods include; two containers that need to share memory or share a volume (see Figure 4.2).

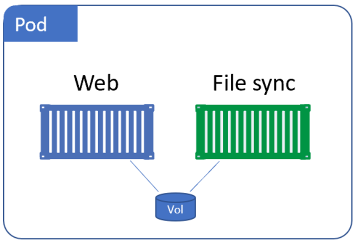


Figure 4.2

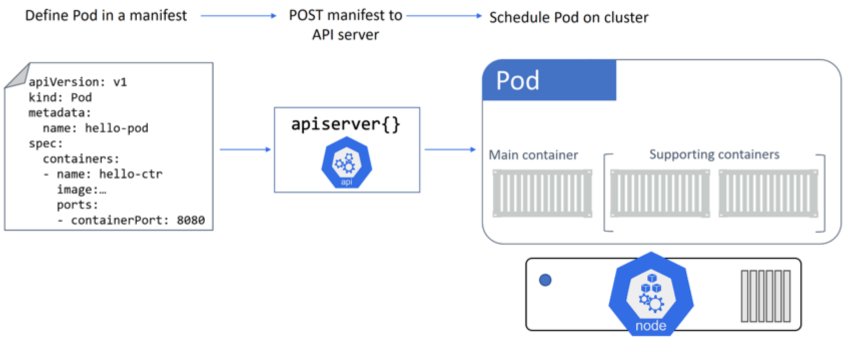
The simplest way to share a volume with two containers is to configure the two containers as part of the same Pod. This will ensure they run on the same node and can access the same shared execution environment (this includes any volumes).

In summary, the general rule is to separate concerns by designing containers do a single job. The simplest model schedules a single container per Pod, but more advanced use-cases place multiple container per Pod.

How do we deploy Pods

Remember that Pods are just a vehicle for executing an application. Therefore, any time we talk about running or deploying Pods, we're talking about running and deploying applications.

To deploy a Pod to a Kubernetes cluster you define it in a *manifest file* and `POST` that manifest file to the API Server. The control plane verifies the configuration of the YAML file, writes it to the cluster store as a record of intent, and the scheduler deploys it to a healthy node with enough available resources. This process is identical for single-container Pods and multi-container Pods.



```
apiVersion: v1
kind: Pod
metadata:
  name: hello-pod
spec:
  containers:
  - name: hello-ctr
    image:…
    ports:
    - containerPort: 8080
```

Figure 4.3

Let's dig a bit deeper…

The anatomy of a Pod

At the highest level, a Pod is a shared execution environment for one or more containers. *Shared execution environment* means that the Pod has a set of resources that are shared by every container that is part of the Pod. These resources include; IP addresses, ports, hostname, sockets, memory, volumes, and more…

If you're using Docker as the container runtime, a Pod is actually a special type of container called a **pause container**. That's right, a Pod is just a fancy name for a special container. This means containers running inside of Pods are really containers running inside of containers. For more information, watch "Inception" by Christopher Nolan, starring Leonardo DiCaprio ;-)

Seriously though, the Pod (pause container) is just a collection of system resources that containers running inside of it will inherit and share. These system resources are kernel namespaces and include:

- **Network namespace**: IP address, port range, routing table…
- **UTS namespace**: Hostname
- **IPC namespace**: Unix domain sockets…

As we just mentioned, this means that all containers in a Pod share a hostname, IP address, memory address space, and volumes.

Let's look at how this affects networking.

Pods and shared networking

Each Pod creates its own network namespace. This includes; a single IP address, a single range of TCP and UDP ports, and a single routing table. If a Pod has a single container, that container has full access to the IP, port range and routing table. If it's a multi-container Pod, all containers in the Pod will share the IP, port range and routing table.

Figure 4.4 shows two Pods, each with its own IP. Even though one of them is a multi-container Pod, it still only gets a single IP.

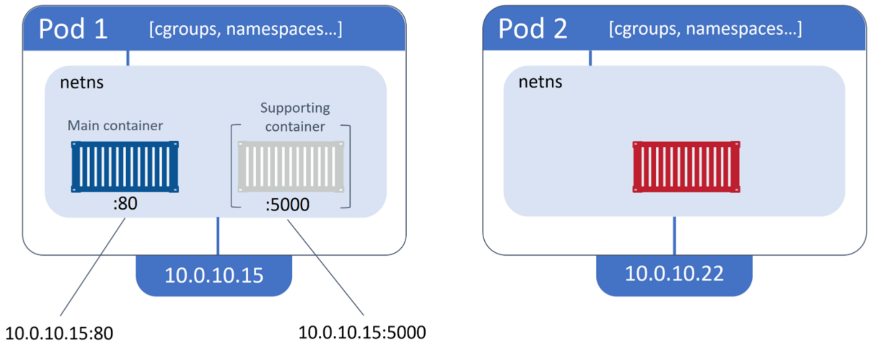


Figure 4.4

In Figure 4.4, external access to the containers in Pod 1 is achieved via the IP address of the Pod coupled with the port of the container you wish to reach. For example, `10.0.10.15:80` will get you to the main container. Container-to-container communication works via the Pod's localhost adapter and port number. For example, the main container can reach the supporting container via `localhost:5000`.

One last time (apologies if it feels like I'm over-repeating myself)… Each container in a Pod shares the **Pod's** entire network namespace – IP, `localhost` adapter, port range, routing table, and more.

However, as we've already said, it's more than just networking. All containers in a Pod have access to the same volumes, the same memory, the same IPC sockets, and more. Technically speaking, the Pod holds all the namespaces, any containers that are part of the Pod inherit them and share them.

This networking model makes *inter-Pod* communication really simple. Every Pod in the cluster has its own IP addresses that's fully routable on the *Pod network*. If you read the chapter on installing Kubernetes, you'll have seen how we created a Pod network at the end of the *Play with Kubernetes*, and *kubeadm* sections. Because every Pod gets its own routable IP, every Pod on the Pod network can talk directly to every other Pod without the need for nasty port mappings.

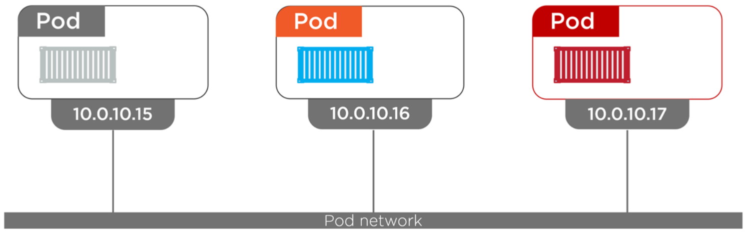


Figure 4.5 Inter-Pod communication

As previously mentioned, *intra-Pod* communication – where two containers in the same Pod need to communicate – can happen via the Pod's `localhost` interface.

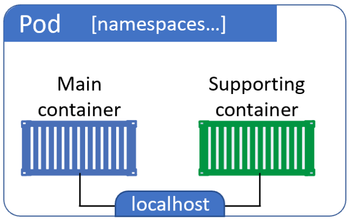


Figure 4.6 Intra-Pod communication

If you need to make multiple containers in the same Pod available to the outside world, you can expose them on individual ports. Each container needs its own port, and two containers in the same Pod cannot use the same port.

In summary. It's all about the **Pod!** The **Pod** gets deployed, the **Pod** gets the IP, the **Pod** owns all of the namespaces… The **Pod** is at the center of the Kuberverse.

Pods and cgroups

At a high level, Control Groups (cgroups) are a Linux kernel technology that prevents individual containers from consuming all of the available CPU, RAM and IOPS on a node. You could say that cgroups actively *police* resource usage.

Individual containers have their own cgroup limits.

This means it's possible for two containers in the same Pod to have their own set of cgroup limits. This is a powerful and flexible model. If we assume the typical multi-container Pod example from earlier in the chapter, you could set a cgroup limit on the file sync container so that it has access to less resources than the web service container. This might reduce the risk of it starving the web service container of CPU and memory.

Atomic deployment of Pods

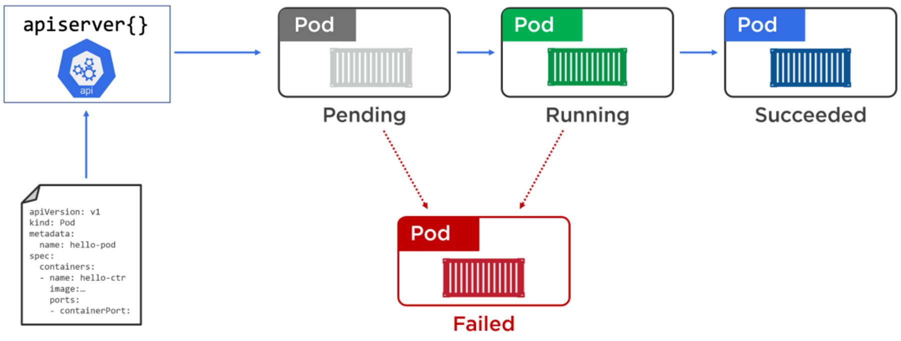
Deploying a Pod is an *atomic operation*. This means it's an all-or-nothing operation – there's no such thing as a partially deployed Pod that can service requests. It also means that all containers in a Pod will be scheduled on the same node.

Once all Pod resources are ready, the Pod can start servicing requests.

Pod lifecycle

The lifecycle of a typical Pod goes something like this. You define it in a YAML manifest file and `POST` the manifest to the API server. Once there, the contents of the manifest are persisted to the cluster store as a record of intent (desired state), and the Pod is scheduled to a healthy node with enough resources. Once it's scheduled to a node, it enters the *pending* state while the container runtime on the node downloads images and starts any containers. The Pod remains in the *pending* state until **all of its resources** are up and ready. Once everything's up and ready, the Pod enters the *running* state. Once it has completed all of its tasks, it gets terminated and enters the *succeeded* state.

When a Pod can't start, it can remain in the *pending* state or go to the *failed* state. This is all shown in Figure 4.7.



```
apiVersion: v1
kind: Pod
metadata:
  name: hello-pod
spec:
  containers:
  - name: hello-ctr
    image:…
    ports:
    - containerPort:
```

Figure 4.7 Pod lifecycle

Pods that are deployed via Pod manifest files are *singletons* – they are not managed by a controller that might add features such as auto-scaling and self-healing capabilities. For this reason, we almost always deploy Pods via higher-level controllers such as *Deployments* and *DaemonSets*, as these can reschedule Pods when they fail.

On that topic, it's important to think of Pods as *mortal*. When they die, they're gone. There's no bringing them back from the dead. This follows the *pets vs cattle* analogy, and Pods should be treated as *cattle*. When they die, you replace them with another. There's no tears and no funeral. The old one is gone, and a shiny new one – with the same config, but a different ID and IP – magically appears and takes its place.

This is one of the main reasons you should design your applications so that they don't store *state* in Pods. It's also why we shouldn't rely on individual Pod IPs. Singleton Pods are not reliable!

Pod theory summary

1. Pods are the atomic unit of scheduling in Kubernetes
2. You can have more than one container in a Pod. Single-container Pods are the simplest, but multi-container Pods are ideal for containers that need to be tightly coupled. They're also great for logging and service meshes
3. Pods get scheduled on nodes – you can't schedule a single Pod instance to span multiple nodes
4. Pods are defined declaratively in a manifest file that is POSTed to the API Server and assigned to nodes by the scheduler
5. You almost always deploy Pods via higher-level controllers

Hands-on with Pods

It's time to see Pods in action.

For the examples in the rest of this chapter we'll use the 3-node cluster shown in Figure 4.8.

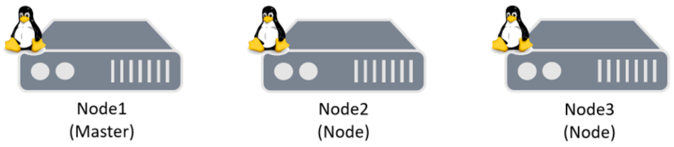


Figure 4.8

It doesn't matter where this cluster is or how it was deployed. All that matters is that you have three Linux hosts configured into a Kubernetes cluster with at least one master and two nodes. You'll also need `kubectl` installed and configured to talk to the cluster.

Three super-quick ways to get a Kubernetes cluster include:

- Sign-up to msb.com and get access to your own private online cluster (plus a lot of learning content)
- Download and install Docker Desktop to your computer
- Search "play with kubernetes" and get a temporary online playground

Following the Kubernetes mantra of *composable infrastructure*, you define Pods in manifest files, POST these to the API server, and let the scheduler instantiate them on the cluster.

Pod manifest files

For the examples in this chapter we're going to use the following Pod manifest. It's available in the book's GitHub repo under the `pods` folder called pod.yml:

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-pod
  labels:
    zone: prod
    version: v1
spec:
  containers:
  - name: hello-ctr
    image: nigelpoulton/k8sbook:latest
    ports:
    - containerPort: 8080
```

Let's step through what the YAML file is describing.

Straight away we can see four top-level resources:

- `apiVersion`
- `kind`
- `metadata`
- `spec`

The `.apiVersion` field tells you two things – the *API group* and the *API version*. The usual format for `apiVersion` `<api-group>/<version>`. However, Pods are defined in a special API group called the *core* group which omits the *api-group* part. For example, StorageClass objects are defined in `v1` of the `storage.k8s.io` API group and are described in YAML files as `storage.k8s.io/v1`. However, Pods are in the *core* API group which is special, as it omits the API group name, so we describe them in YAML files as just `v1`.

It's possible for a resource to be defined in multiple versions of an API group. For example, `some-api-group/v1` and `some-api-group/v2`. In this case, the definition in the newer group will probably include additional features and fields that extend the capabilities of the resource. Think of the *apiVersion* field as defining the schema – newer is usually better. Interestingly, there may be occasions where you deploy an object via one version in the YAML file, but when you introspect it, the return values show it as another version. For example, you may deploy an object by specifying `v1` in the YAML file, but when you run commands against it the returns might show it as `v1beta1`. This is normal behaviour.

Anyway, Pods are defined at the `v1` path.

The `.kind` field tells Kubernetes the type of object is being deployed.

So far, you know you're deploying a Pod object as defined in `v1` of the *core API group*.

The `.metadata` section is where you attach a name and labels. These help you identify the object in the cluster, as well as create loose couplings between different objects. You can also define the Namespace that an object should be deployed to. Keeping things brief, Namespaces are a way to logically divide a cluster into multiple virtual clusters for management purposes. In the real world, it's highly recommended to use namespaces, however, you should not think of them as strong security boundaries.

The `.metadata` section of this Pod manifest is naming the Pod "hello-pod" and assigning it two labels. Labels are simple key-value pairs, but they're insanely powerful. We'll talk more about labels later as you build your knowledge.

As the `.metadata` section does not specify a Namespace, the Pod will be deployed to the `default` Namespace. It's not good practice to use the default namespace in the real world, but it's fine for these examples.

The `.spec` section is where you define the containers that will run in the Pod. This example is deploying a Pod with a single container based on the `nigelpoulton/k8sbook:latest` image. It's calling the container `hello-ctr` and exposing it on port `8080`.

If this was a multi-container Pod, you'd define additional containers in the `.spec` section.

Manifest files: Empathy as Code

Quick side-step.

Configuration files, like Kubernetes manifest files, are excellent sources of documentation. As such, they have some secondary benefits. Two of these include:

- Speeding-up the on-boarding process for new team members
- Bridging the gap between developers and operations

For example, if you need a new team member to understand the basic functions and requirements of an application, get them to read the application's Kubernetes manifest files.

Also, if your operations teams complain that developers don't give accurate application requirements and documentation, make your developers use Kubernetes. Kubernetes forces developers to describe their applications through Kubernetes manifests, which can then be used by operations staff to understand how the application works and what it requires from the environment.

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

I understand that describing YAML files as *"empathy as code"* sounds a bit extreme. However, there is merit to the concept – they definitely help.

Back on track…

Deploying Pods from a manifest file

If you're following along with the examples, save the manifest file as `pod.yml` in your current directory and then use the following `kubectl` command to POST the manifest to the API server.

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

Although the Pod is showing as created, it might not be fully deployed and available yet. This is because it takes time to pull the image.

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

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

You can see that the container is still being created – probably waiting for the image to be pulled from Docker Hub.

You can add the `--watch` flag to the `kubectl get pods` command so that you can monitor it and see when the status changes to `Running`.

Congratulations! Your Pod has been scheduled to a healthy node in the cluster and is being monitored by the local `kubelet` process. The `kubelet` process is the Kubernetes agent running on the node.

In future chapters, you'll see how to connect to the web server running in the Pod.

Introspecting running Pods

As good as the `kubectl get pods` command is, it's a bit light on detail. Not to worry though, there's plenty of options for deeper introspection.

First up, the `kubectl get` command offers a couple of really simple flags that give you more information:

The `-o wide` flag gives a couple more columns but is still a single line of output.

The `-o yaml` flag takes things to the next level. This returns a full copy of the Pod manifest from the cluster store. The output is broadly divided into two parts:

- desired state (`.spec`)
- current observed state (`.status`)

The following command shows a snipped version of a `kubectl get pods -o yaml` command.

```
$ kubectl get pods hello-pod -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      ...
  name: hello-pod
  namespace: default
spec:  #Desired state
  containers:
  - image: nigelpoulton/k8sbook:latest
    imagePullPolicy: Always
    name: hello-ctr
    ports:
status:  #Observed state
  conditions:
  - lastProbeTime: null
    lastTransitionTime: 2019-11-19T15:24:24Z
    state:
      running:
        startedAt: 2019-11-19T15:26:04Z
...
```

Notice how the output contains more values than you initially set in the 13-line YAML file. Where does this extra information come from?

Two main sources:

- The Kubernetes Pod object has far more settings than we defined in the manifest. Those that are not set explicitly are automatically expanded with default values by Kubernetes.
- When you run a `kubectl get pods` with `-o yaml` you get the Pods *current observed state* as well as its *desired state*. This observed state is listed in the `.status` section.

kubectl describe

Another great Kubernetes introspection command is `kubectl describe`. This provides a nicely formatted multi-line overview of an object. It even includes some important object lifecycle events. The following command describes the state of the hello-pod Pod.

```
$ kubectl describe pods hello-pod
Name:         hello-pod
Namespace:    default
Node:         k8s-slave-lgpkjg/10.48.0.35
Start Time:   Tue, 19 Nov 2019 16:24:24 +0100
Labels:       version=v1
              zone=prod
Status:       Running
IP:           10.1.0.21
Containers:
  hello-ctr:
    Image:          nigelpoulton/k8sbook:latest
    Port:           8080/TCP
    Host Port:      0/TCP
    State:          Running
Conditions:
  Type           Status
  Initialized    True
  Ready          True
  PodScheduled   True
...
Events:
  Type    Reason     Age   Message
  ----    ------     ----  -------
  Normal  Scheduled  2m   Successfully assigned...
  Normal  Pulling    2m   pulling image "nigelpoulton/k8sbook:latest"
  Normal  Pulled     2m   Successfully pulled image
  Normal  Created    2m   Created container
  Normal  Started    2m   Started container
```

The output has been snipped to help it fit the book.

kubectl exec: running commands in Pods

Another way to introspect a running Pod is to log into it or execute commands in it. You can do both of these with the `kubectl exec` command. The following example shows how to execute a `ps aux` command in the first container in the `hello-pod` Pod.

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

You can also log-in to containers running in Pods using `kubectl exec`. When you do this, your terminal prompt will change to indicate your session is now running inside of a container in the Pod, and you'll be able to execute commands from there (as long as the command binaries are installed in the container).

The following `kubectl exec` command will log-in to the first container in the `hello-container` Pod. Once inside the container, install the `curl` utility and run a `curl` command to transfer data from the process listening on port `8080`.

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

# apk add curl
<Snip>

# curl localhost:8080
<html><head><title>Pluralsight Rocks</title><link rel="stylesheet" href="http://netdna.bootstrapcdn.co\
m/bootstrap/3.1.1/css/bootstrap.min.css"/></head><body><div class="container"><div class="jumbotron"><\
h1>Yo Pluralsighters!!!</h1><p>Click the button below to head over to my podcast...</p><p> <a href="ht\
tp://intechwetrustpodcast.com" class="btn btn-primary">Podcast</a></p><p></p></div></div></body></html\
>
```

The `-it` flags make the `exec` session interactive and connects STDIN and STDOUT on your terminal to STDIN and STDOUT inside the first container in the Pod. When the command completes, your shell prompt will change to indicate your shell is now connected to the container.

If you are running multi-container Pods, you will need to pass the `kubectl exec` command the `--container` flag and give it the name of the container that you want to create the exec session with. If you do not specify this flag, the command will execute against the first container in the Pod. You can see the ordering and names of containers in a Pod with the `kubectl describe pods <pod>` command.

kubectl logs

One other useful command for introspecting Pods is the `kubectl logs` command. Like other Pod-related commands, if you don't use `--container` to specify a container by name, it will execute against the first container in the Pod. The format of the command is `kubectl logs <pod>`.

There's obviously a lot more to Pods than what we've covered. However, you've learned enough to get started.

Clean-up up the lab by typing `exit` to quit your exec session inside the container, then run `kubectl delete` to delete the Pod.

```
# exit
$ kubectl delete -f pod.yml
pod "hello-pod" deleted
```

Chapter Summary

In this chapter, you learned that the atomic unit of deployment in the Kubernetes world is the *Pod*. Each Pod consists of one or more containers and gets deployed to a single node in the cluster. The deployment operation is an all-or-nothing *atomic operation*.

Pods are defined and deployed declaratively using a YAML manifest file, and it's normal to deploy them via higher-level controllers such as Deployments. You use the `kubectl` command to `POST` the manifest to the API Server, it gets stored in the cluster store and converted into a PodSpec that is scheduled to a healthy cluster node with enough available resources.

The process on the worker node that accepts the PodSpec is the `kubelet`. This is the main Kubernetes agent running on every node in the cluster. It takes the PodSpec and is responsible for pulling all images and starting all containers in the Pod.

If you deploy a singleton Pod (a Pod that is not deployed via a controller) to your cluster and the node it is running on fails, the singleton Pod is not rescheduled on another node. Because of this, you almost always deploy Pods via higher-level controllers such as Deployments and DaemonSets. These add capabilities such as self-healing and rollbacks which are at the heart of what makes Kubernetes so powerful.

5: Kubernetes Deployments

In this chapter, you'll see how *Deployments* bring self-healing, scalability, rolling updates, and versioned rollbacks to Kubernetes.

We'll divide the chapter as follows:

- Deployment theory
- How to create a Deployment
- How to perform a rolling update
- How to perform a rollback

Deployment theory

At a high level, you start with application code. That gets packaged as a container and wrapped in a Pod so it can run on Kubernetes. However, Pods don't self-heal, they don't scale, and they don't allow for easy updates or rollbacks. Deployments do all of these. As a result, you'll almost always deploy Pods via a Deployment controller.

Figure 5.1 shows some Pods being managed by a Deployment controller.

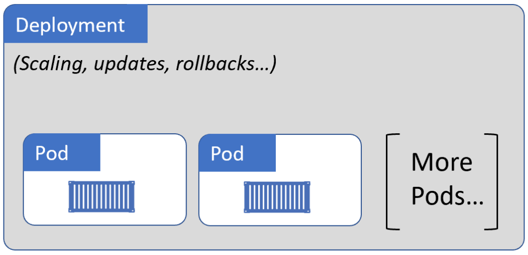


Figure 5.1

It's important to know that a single Deployment object can only manage a single Pod template. For example, if you have an application with a Pod template for the web front-end and another Pod template for the catalog service, you'll need two Deployments. However, as you saw in Figure 5.1, a Deployment can manage multiple replicas of the same Pod. For example, Figure 5.1 could be a Deployment that currently manages two replicated web server Pods.

The next thing to know is that Deployments are fully-fledged objects in the Kubernetes API. This means you define them in manifest files that you `POST` to the API Server.

The last thing to note, is that behind-the-scenes, Deployments leverage another object called a ReplicaSet. While it's best practice that you don't interact directly with ReplicaSets, it's important to understand the role they play.

Keeping it high-level, Deployments use ReplicaSets to provide self-healing and scaling.

Figure 5.2. shows the same Pods managed by the same Deployment. However, this time we've added a ReplicaSet object into the relationship and shown which object is responsible for which feature.

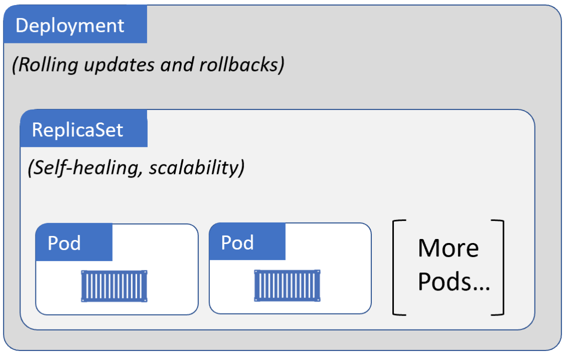


Figure 5.2

In summary, think of *Deployments* as managing *ReplicaSets*, and *ReplicaSets* as managing *Pods*. Put them all together, and you've got a great way to deploy and manage applications on Kubernetes.

Self-healing and scalability

Pods are great. They augment containers by allowing co-location of containers, sharing of volumes, sharing of memory, simplified networking, and a lot more. But they offer nothing in the way of self-healing and scalability – if the node a Pod is running on fails, the Pod will not be restarted.

Enter Deployments…

Deployments augment Pods by adding things like self-healing and scalability. This means:

- If a Pod managed by a Deployment fails, it will be replaced – *self-healing*.
- If a Pod managed by a Deployment sees increased load, you can easily add more of the same Pod to deal with the load – *scaling*.

Remember though, behind-the-scenes, Deployments use an object called a ReplicaSet to accomplish self-healing and scalability. However, ReplicaSets operate in the background and you should always carry out operations against the Deployment. For this reason, we'll focus on Deployments.

It's all about the *state*

Before going any further, it's critical to understand three concepts that are fundamental to everything about Kubernetes:

- Desired state
- Current state (sometimes called *actual state* or *observed state*)
- Declarative model

Desired state is what you **want**. *Current state* is what you **have**. If the two match, everybody's happy.

The *declarative model* is a way of telling Kubernetes what your *desired state* is, without having to get into the detail of *how* to implement it. You leave the *how* up to Kubernetes.

The declarative model

There are two competing models. The *declarative model* and the *imperative model.*

The declarative model is all about describing the end-goal – telling Kubernetes what you want. The imperative model is all about long lists of commands to reach the end-goal – telling Kubernetes **how** to do something.

The following is an extremely simple analogy that might help:

- **Declarative**: I need a chocolate cake that will feed 10 people.
- **Imperative**: Drive to the store. Buy; eggs, milk, flour, cocoa powder… Drive home. Turn on oven. Mix ingredients. Place in baking tray. Place tray in oven for 30 minutes. Remove from oven and turn oven off. Add icing. Leave to stand.

The declarative model is stating what you want (chocolate cake for 10). The imperative model is a long list of steps required to make a chocolate cake for 10.

Let's look at a more concrete example.

Assume you've got an application with two services – front-end and back-end. You've built container images so that you can have a Pod for the front-end service, and a separate Pod for the back-end service. To meet expected demand, you always need 5 instances of the front-end Pod, and 2 instances of the back-end Pod.

Taking the declarative approach, you write a configuration file that tells Kubernetes what you want your application to look like. For example, *I want 5 replicas of the front-end Pod all listening externally on port 80 please. And I also want 2 back-end Pods listening internally on port 27017.* That's the desired state. Obviously, the YAML format of the config file will be different, but you get the picture.

Once you've described the desired state, you give the config file to Kubernetes and sit back while Kubernetes does the hard work of implementing it.

But things don't stop there… Kubernetes implements watch loops that are constantly checking that you've got what you asked for – does current state match desired state.

Believe me when I tell you, it's a beautiful thing.

The opposite of the declarative model is the imperative model. In the imperative model, there's no concept of what you actually want. At least there's no *record* of what you want, all you get is a list of instructions.

To make things worse, imperative instructions might have multiple variations. For example, the commands to start `containerd` containers are different from the commands to start `gVisor` containers. This ends up being more work, prone to more errors, and because it's not declaring a desired state, there's no self-healing.

Believe me when I tell you, this isn't so beautiful.

Kubernetes supports both models, but strongly prefers the declarative model.

Reconciliation loops

Fundamental to desired state is the concept of background reconciliation loops (a.k.a. control loops).

For example, ReplicaSets implement a background reconciliation loop that is constantly checking whether the right number of Pod replicas are present on the cluster. If there aren't enough, it adds more. If there are too many, it terminates some.

To be crystal clear, **Kubernetes is constantly making sure that *current state* matches *desired state*.**

If they don't match – maybe desired state is 10 replicas, but only 8 are running – Kubernetes declares a red-alert condition, orders the control plane to battle-stations and brings up two more replicas. And the best part… it does all of this without calling you at 04:20 am!

But it's not just failure scenarios. These very-same reconciliation loops enable scaling.

For example, if you `POST` an updated config that changes replica count from 3 to 5, the new value of 5 will be registered as the application's new *desired state*. The next time the ReplicaSet reconciliation loop runs, it will notice the discrepancy and follow the same process – sounding the claxon horn for red alert and spinning up two more replicas.

It really is a beautiful thing.

Rolling updates with Deployments

As well as self-healing and scaling, Deployments give us zero-downtime rolling-updates.

As previously mentioned, Deployments use ReplicaSets for some of the background legwork. In fact, every time you create a Deployment, you automatically get a ReplicaSet that manages the Deployment's Pods.

> **Note**: Best practice states that you should not manage ReplicaSets directly. You should perform all actions against the Deployment object and leave the Deployment to manage ReplicaSets.

It works like this. You design applications with each discrete service as a Pod. For convenience – self-healing, scaling, rolling updates and more – you wrap Pods in Deployments. This means creating a YAML configuration file describing all of the following:

- How many Pod replicas
- What image to use for the Pod's container(s)
- What network ports to use
- Details about how to perform rolling updates

You `POST` the YAML file to the API server and Kubernetes does the rest.

Once everything is up and running, Kubernetes sets up watch loops to make sure observed state matches desired state.

All good so far.

Now, assume you've experienced a bug, and you need to deploy an updated image that implements a fix. To do this, you update the **same Deployment YAML file** with the new image version and re-POST it to the API server. This registers a new desired state on the cluster, requesting the same number of Pods, but all running the new version of the image. To make this happen, Kubernetes creates a new ReplicaSet for the Pods with the new image. You now have two ReplicaSets – the original one for the Pods with the old version of the image, and a new one for the Pods with the updated version. Each time Kubernetes increases the number of Pods in the new ReplicaSet (with the new version of the image) it decreases the number of Pods in the old ReplicaSet (with the old version of the image). Net result, you get a smooth rolling update with zero downtime. And you can rinse and repeat the process for future updates – just keep updating that manifest file (which should be stored in a version control system).

Brilliant.

Figure 5.3 shows a Deployment that has been updated once. The initial deployment created the ReplicaSet on the left, and the update created the ReplicaSet on the right. You can see that the ReplicaSet for the initial deployment has been wound down and no longer has any Pod replicas. The ReplicaSet associated with the update is active and owns all of the Pods.

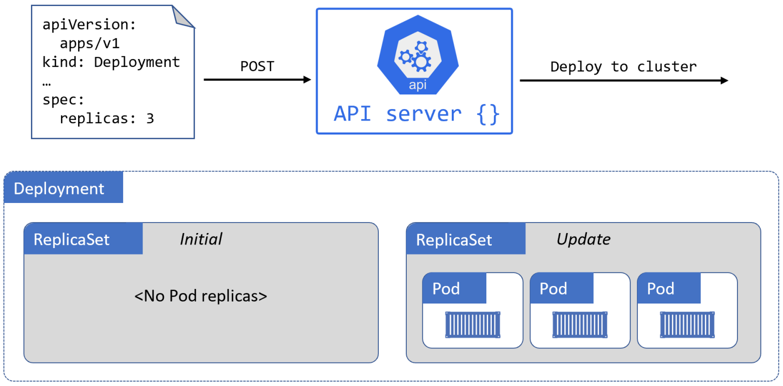


Figure 5.3

It's important to understand that the old ReplicaSet still has its entire configuration, including the older version of the image it used. This will be important in the next section.

Rollbacks

As we've seen in Figure 5.3, older ReplicaSets are wound down and no longer manage any Pods. However, they still exist with their full configuration. This makes them a great option for reverting to previous versions.

The process of rolling back is essentially the opposite of a rolling update – wind one of the old ReplicaSets up, and wind the current one down. Simple.

Figure 5.4 shows the same app rolled back to the initial revision.

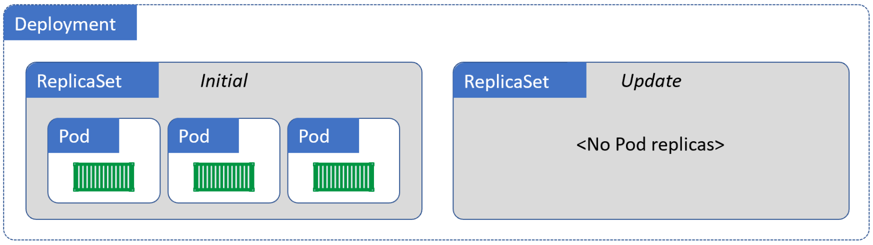


Figure 5.4

That's not the end though. There's built-in intelligence that lets us say things like *"wait X number of seconds after each Pod comes up before proceeding to the next Pod"*. There's also startup probes, readiness probes, and liveness probes that can check the health and status of Pods. All-in-all, Deployments are excellent for performing rolling updates and versioned rollbacks.

With all of that in mind, let's get your hands dirty and create a Deployment.

How to create a Deployment

In this section, you'll create a brand-new Kubernetes Deployment from a YAML file. You can do the same thing imperatively using the `kubectl run` command, but you shouldn't. The right way is the declarative way.

The following YAML snippet is the Deployment manifest file that you'll use. It's available in the book's GitHub repo in the "deployments" folder and is called `deploy.yml`.

The examples assume you've got a copy in your system's PATH, and is called `deploy.yml`.

```
apiVersion: apps/v1  #Older versions of k8s use apps/v1beta1
kind: Deployment
metadata:
  name: hello-deploy
spec:
  replicas: 10
  selector:
    matchLabels:
      app: hello-world
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-pod
        image: nigelpoulton/k8sbook:latest
        ports:
        - containerPort: 8080
```

> **Warning**: The images used in this book are not maintained and may contain vulnerabilities and other security issues. Use with caution.

Let's step through the config and explain some of the important parts.

Right at the very top you specify the API version to use. Assuming that you're using an up-to-date version of Kubernetes, Deployment objects are in the `apps/v1` API group.

Next, the `.kind` field tells Kubernetes you're defining a Deployment object.

The `.metadata` section is where we give the Deployment a name and labels.

The `.spec` section is where most of the action happens. Anything directly below `.spec` relates to the Pod. Anything nested below `.spec.template` relates to the Pod template that the Deployment will manage. In this example, the Pod template defines a single container.

`.spec.replicas` tells Kubernetes how may Pod replicas to deploy. `spec.selector` is a list of labels that Pods must have in order for the Deployment to manage them. And `.spec.strategy` tells Kubernetes how to perform updates to the Pods managed by the Deployment.

Use `kubectl apply` to implement it on the cluster.

> **Note:** `kubectl apply` POSTs the YAML file to the Kubernetes API server.

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

The Deployment is now instantiated on the cluster.

Inspecting Deployments

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

```
$ kubectl get deploy hello-deploy
NAME          DESIRED   CURRENT  UP-TO-DATE  AVAILABLE   AGE
hello-deploy  10        10       10          10          24s

$ kubectl describe deploy hello-deploy
Name:          hello-deploy
Namespace:     default
Selector:      app=hello-world
Replicas:                10 desired | 10 updated | 10 total ...
StrategyType:            RollingUpdate
MinReadySeconds:         10
RollingUpdateStrategy:   1 max unavailable, 1 max surge
Pod Template:
  Labels:                app=hello-world
  Containers:
        hello-pod:
          Image:         nigelpoulton/k8sbook:latest
          Port:          8080/TCP
<SNIP>
```

The command outputs have been trimmed for readability. Yours will show more information.

As we mentioned earlier, Deployments automatically create associated ReplicaSets. Use the following `kubectl` command to confirm this.

```
$ kubectl get rs
NAME                  DESIRED   CURRENT  READY   AGE
hello-deploy-7bbd...  10        10       10      1m
```

Right now you only have one ReplicaSet. This is because you've only performed the initial rollout of the Deployment. You can also see that the name of the ReplicaSet matches the name of the Deployment with a hash on the end. The hash is a hash of the Pod template section (anything below `.spec.template`) of the YAML manifest file.

You can get more detailed information about the ReplicaSet with the usual `kubectl describe` command.

Accessing the app

In order to access the application from a stable name or IP address, or even from outside the cluster, you need a Kubernetes Service object. We'll discuss Service objects in detail in the next chapter, but for now it's enough to know they provide a stable DNS name and IP address for a set of Pods.

The following YAML defines a Service that will work with the Pod replicas previously deployed. The YAML is included in the "deployments" folder of the book's GitHub repo called `svc.yml`.

```
apiVersion: v1
kind: Service
metadata:
  name: hello-svc
  labels:
    app: hello-world
spec:
  type: NodePort
  ports:
  - port: 8080
    nodePort: 30001
    protocol: TCP
  selector:
    app: hello-world
```

Deploy it with the following command (the command assumes the manifest file is called `svc.yml` and is in your system's PATH).

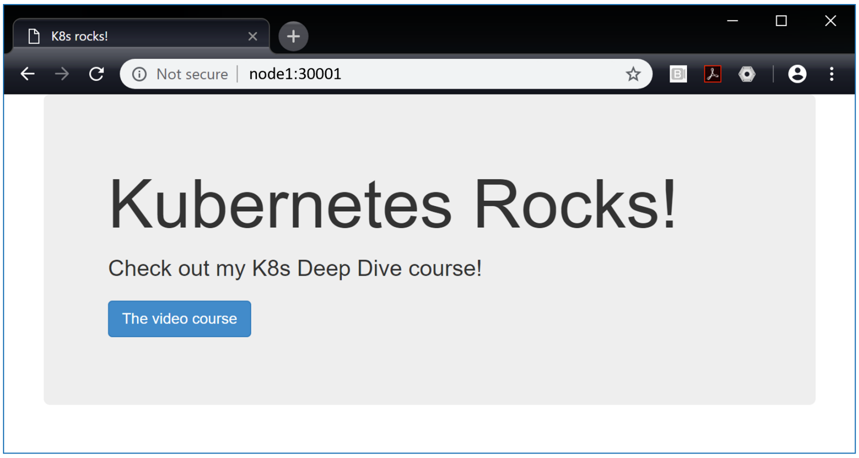
```
$ kubectl apply -f svc.yml
service/hello-svc created
```

Now that the Service is deployed, you can access the app from either of the following:

1. From inside the cluster using the DNS name `hello-svc` on port `8080`
2. From outside the cluster by hitting any of the cluster nodes on port `30001`

Figure 5.5 shows the Service being accessed from outside of the cluster via a node called `node1` on port `30001`. It assumes that `node1` is resolvable, and that port `30001` is allowed by any intervening firewalls.

If you are using Minikube, you should append port `30001` to the end of the Minikube IP address. Use the `minikube ip` command to get the IP address of your Minikube.



Kubernetes Rocks!

Check out my K8s Deep Dive course!

The video course

Figure 5.5

Performing a rolling update

In this section, you'll see how to perform a rolling update on the app you've just deployed. We'll assume the new version of the app has already been created and containerized as a Docker image with the `edge` tag. All that is left to do is use Kubernetes to push the update to production. For this example, we're ignoring real-world CI/CD workflows and version control tools.

The first thing you need to do is update the image tag used in the Deployment's manifest file. The initial version of the app used an image tagged as `nigelpoulton/k8sbook:latest`. You'll update the `.spec.template.spec.containers` section of the Deployment manifest to reference the new `nigelpoulton/k8sbook:edge` image. This will ensure that next time the manifest is POSTed to the API server, all Pods in the Deployment will be replaced with new ones running the new `edge` image.

The following is the updated `deploy.yml` manifest file – the only change is to `.spec.template.spec.containers.image` indicated by the commented line.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deploy
spec:
  replicas: 10
  selector:
    matchLabels:
      app: hello-world
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-pod
        image: nigelpoulton/k8sbook:edge   # This line changed
        ports:
        - containerPort: 8080
```

Before POSTing the updated configuration to Kubernetes, let's look at the settings that govern how the update will proceed.

The `.spec` section of the manifest contains all of the settings relating to how updates will be performed. The first value of interest is `.spec.minReadySeconds`. This is set to `10`, telling Kubernetes to wait for 10 seconds between each Pod being updated. This is useful for throttling the rate at which updates occur – longer waits give you a chance to spot problems and avoid situations where you update all Pods to a faulty configuration.

There is also a nested `.spec.strategy` map that tells Kubernetes you want this Deployment 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`)

As the desired state of the app demands 10 replicas, `maxSurge: 1` means you will never have more than 11 Pods during the update process, and `maxUnavailable: 1` means you'll never have less than 9. The net result will be a rolling update that updates two Pods at a time (the delta between 9 and 11 is 2).

With the updated manifest ready, you can initiate the update by re-POSTing the updated YAML file to the API server.

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

The update may take some time to complete. This is because it will iterate two Pods at a time, pulling down the new image on each node, starting the new Pods, and then waiting 10 seconds before moving on to the next two.

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

```
$ kubectl rollout status deployment hello-deploy
Waiting for rollout to finish: 4 out of 10 new replicas...
Waiting for rollout to finish: 4 out of 10 new replicas...
Waiting for rollout to finish: 5 out of 10 new replicas...
^C
```

If you press `Ctrl+C` to stop watching the progress of the update, you can run `kubectl get deploy` commands while the update is in process. This lets you see the effect of some of the update-related settings in the manifest. For example, the following command shows that 5 of the replicas have been updated and you currently have 11. 11 is 1 more than the desired state of 10. This is a result of the `maxSurge=1` value in the manifest.

```
$ kubectl get deploy
NAME          DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
hello-deploy  10         11         5             9            28m
```

Once the update is complete, we can verify with `kubectl get deploy`.

```
$ kubectl get deploy hello-deploy
NAME          DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
hello-deploy  10         10         10            10           39m
```

The output shows the update as complete – 10 Pods are up to date.

You can get more detailed information about the state of the Deployment with the `kubectl describe deploy` command. This will include the new version of the image in the `Pod Template` section of the output.

If you've been following along with the examples, you'll be able to hit `refresh` in your browser and see the updated app (Figure 5.6).The old version of the app displayed "Kubernetes Rocks!", the new version displays "The Kubernetes Book!!!".

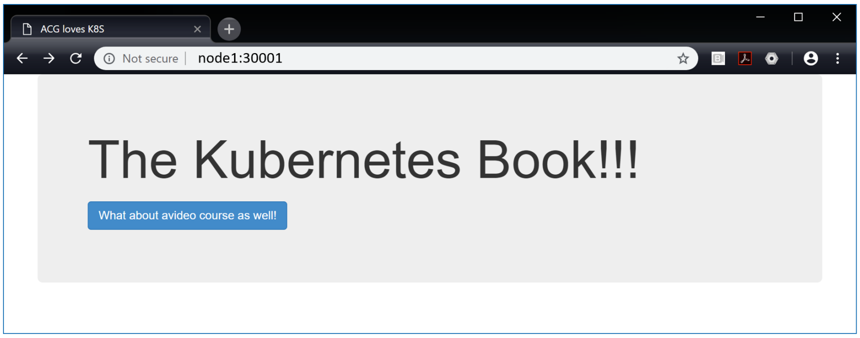


Figure 5.6

How to perform a rollback

A moment ago, you used `kubectl apply` to perform a rolling update on a Deployment. You used the `--record` flag so that Kubernetes would maintain a documented revision history of the Deployment. The following `kubectl rollout history` command shows the Deployment with two revisions.

```
$ kubectl rollout history deployment hello-deploy
deployment.apps/hello-deploy
REVISION  CHANGE-CAUSE
1         <none>
2         kubectl apply --filename-deploy.yml --record=true
```

Revision `1` was the initial deployment that used the `latest` image tag. Revision `2` is the rolling update you just performed. You can see that the command used to invoke the update has been recorded in the object's history.

This is only there because you used the `--record` flag as part of the command to invoke the update. This might be a good reason for you to use the `--record` flag.

Earlier in the chapter we said that updating a Deployment creates a new ReplicaSet, and that any previous ReplicaSets are not deleted. You can verify this with a `kubectl get rs`.

```
$ kubectl get rs
NAME                  DESIRED  CURRENT  READY  AGE
hello-deploy-6bc8...  10       10       10     10m
hello-deploy-7bbd...  0        0        0      52m
```

The output shows that the ReplicaSet for the initial revision still exists (`hello-deploy-7bbd...`) but that it has been wound down and is not managing any replicas. The `hello-deploy-6bc8...` ReplicaSet is the one from the latest revision and is active with 10 replicas under management. However, the fact that the previous version still exists makes rollbacks extremely simple.

If you're following along, it's worth running a `kubectl describe rs` against the old ReplicaSet to prove that its configuration still exists.

The following example uses the `kubectl rollout` command to roll the application back to revision 1. This is an imperative operation and not recommended. However, it can be convenient for quick rollbacks, just remember to update your source YAML files to reflect the imperative changes you make to the cluster.

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

Although it might look like the rollback operation is instantaneous, it's not. Rollbacks follow the same rules set out in the rolling update sections of the Deployment manifest – `minReadySeconds: 10`, `maxUnavailable: 1`, and `maxSurge: 1`. You can verify this and track the progress with the following `kubectl get deploy` and `kubectl rollout` commands.

```
$ kubectl get deploy hello-deploy
NAME          DESIRED  CURRNET  UP-TO-DATE  AVAILABE  AGE
hello-deploy  10       11       4           9         45m

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

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

Use `kubectl delete -f deploy.yml` and `kubectl delete -f svc.yml` to delete the Deployment and Service used in the examples.

Just a quick reminder. The rollback operation you just initiated was an imperative operation. This means that the current state of the cluster will not match your source YAML files – the latest version of the YAML file lists the `edge` image, but you've rolled the cluster back to the `latest` image. This is a problem with the imperative approach. In the real world, following a rollback operation like this, you should manually update your source YAML files to reflect the changes incurred by the rollback.

Chapter summary

In this chapter, you learned that *Deployments* are a great way to manage Kubernetes apps. They build on top of Pods by adding self-healing, scalability, rolling updates, and rollbacks. Behind-the-scenes, they leverage ReplicaSets for the self-healing and scalability parts.

Like Pods, Deployments are objects in the Kubernetes API, and you should work with them declaratively.

When you perform updates with the `kubectl apply` command, older versions of ReplicaSets get wound down, but they stick around making it easy to perform rollbacks.

6: Kubernetes Services

In the previous chapters, we've looked at some Kubernetes objects that are used to deploy and run applications. We looked at Pods as the most fundamental unit for deploying microservices applications, then we looked at Deployment controllers that add things like scaling, self-healing, and rolling updates. However, despite all of benefits of Deployments, **we still cannot rely on individual Pod IPs!** This is where Kubernetes *Service* objects come into play – they provide stable and reliable networking for a set of dynamic Pods.

We'll divide the chapter as follows:

- Setting the scene
- Theory
- Hands-on
- Real world example

Setting the scene

Before diving in, we need to remind ourselves that Pod IPs are unreliable. When Pods fail, they get replaced with new Pods that have new IPs. Scaling-up a Deployment introduces new Pods with new IP addresses. Scaling-down a Deployment removes Pods. This creates a large amount of *IP churn*, and creates the situation where Pod IPs cannot be relied on.

You also need to know 3 fundamental things about Kubernetes Services.

First, let's clear up some terminology. When talking about *Service* with a capital "S", we're talking about the Service object in Kubernetes that provides stable networking for Pods. Just like a *Pod*, *ReplicaSet*, or *Deployment*, a Kubernetes ***Service*** is a REST object in the API that you define in a manifest and POST to the API Server.

Second, you need to know that every Service gets its own **stable IP address**, its own **stable DNS name**, and its own **stable port**.

Third, you need to know that Services leverage *labels* to dynamically select the Pods in the cluster they will send traffic to.

Theory

Figure 6.1 shows a simple Pod-based application deployed via a Kubernetes Deployment. It shows a client (which could be another component of the app) that does not have a reliable network endpoint for accessing the Pods. Remember, it's a bad idea to talk directly to an individual Pod because that Pod could disappear at any point via scaling operations, updates and rollbacks, and failures.

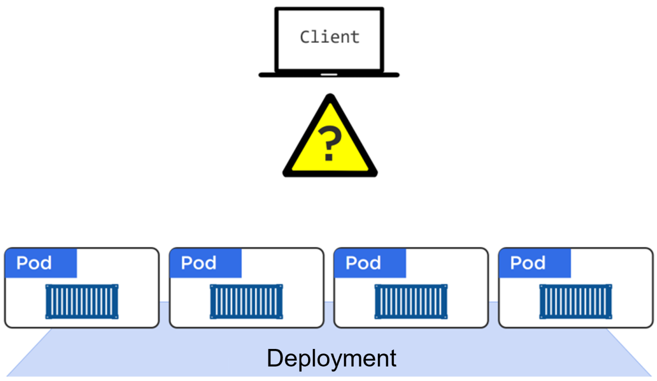


Figure 6.1

Figure 6.2 shows the same application with a Service added into the mix. The Service is associated with the Pods and fronts them with a stable IP, DNS, and port. It also load-balances requests across the Pods.

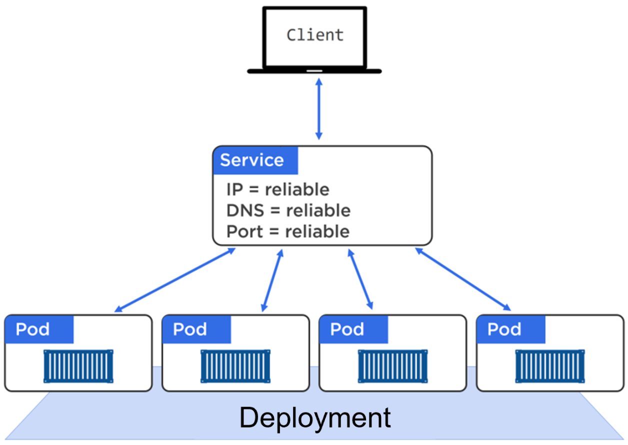


Figure 6.2

With a Service in front of a set of Pods, the Pods can scale up and down, they can fail, and they can be updated, rolled back… and while events like these occur, the Service in front of them observes the changes and updates its list of healthy Pods. But it never changes the stable IP, DNS, and port that it exposes.

Think of Services as having a static front-end and a dynamic back-end. The front-end, consisting of the IP, DNS name, and port, and never changes. The back-end, consisting of the Pods, can be constantly changing.

Labels and loose coupling

Services are loosely coupled with Pods via *labels* and *label selectors*. This is the same technology that loosely couples Deployments to Pods and is key to the flexibility provided by Kubernetes. Figure 6.3 shows an example where 3 Pods are labelled as `zone=prod` and `version=1`, and the Service has a *label selector* that matches.

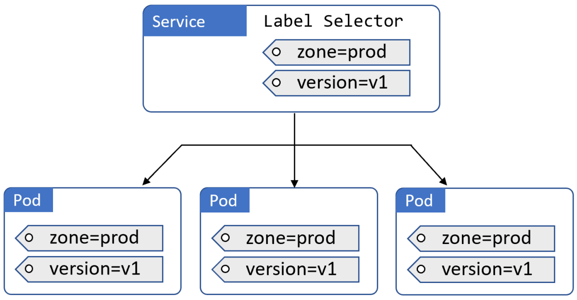


Figure 6.3

In Figure 6.3, the Service is providing stable networking to all three Pods – you can send requests to the Service and it will forward them on to the Pods. It also provides simple load-balancing.

For a Service to match a set of Pods, and therefore send traffic to them, the Pods must possess every label in the Services label selector. However, the Pod can have additional labels that are not listed in the Service's label selector. If that's confusing, the examples in Figures 6.4 and 6.5 should help.

Figure 6.4 shows an example where the Service does not match any of the Pods. This is because the Service is looking for Pods that have two labels, but the Pods only possess one of them. The logic behind this is a Boolean `AND`.

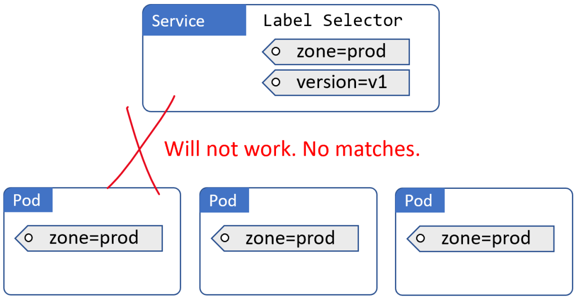


Figure 6.4

Figure 6.5 shows an example that does work. It works because the Service is looking for two labels and the Pods in the diagram possess both. It doesn't matter that the Pods possess additional labels that the Service isn't looking for. The Service is looking for Pods with two labels, it finds them, and ignores the fact that the Pods have additional labels – all that is important is that the Pods possess the labels the Service is looking for.

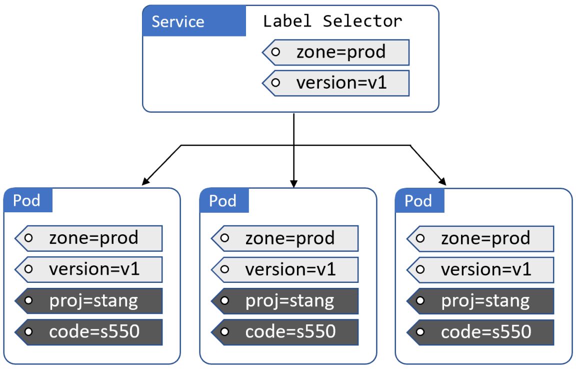


Figure 6.5

The following excerpts, from a Service YAML and Deployment YAML, show how *selectors* and *labels* are implemented. I've added comments to the lines of interest.

svc.yml

```
apiVersion: v1
kind: Service
metadata:
  name: hello-svc
spec:
  ports:
  - port: 8080
  selector:
    app: hello-world   # Label selector
    # Service is looking for Pods with the label `app=hello-world`
```

deploy.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deploy
spec:
  replicas: 10
  selector:
    matchLabels:
      app: hello-world
  template:
    metadata:
      labels:
        app: hello-world   # Pod labels
        # The label matches the Service's label selector
    spec:
      containers:
      - name: hello-ctr
```

```
        image: nigelpoulton/k8sbook:latest
        ports:
        - containerPort: 8080
```

In the example files, the Service has a label selector (`.spec.selector`) with a single value `app=hello-world`. This is the label that the Service is looking for when it queries the cluster for matching Pods. The Deployment specifies a Pod template with the same `app=hello-world` label (`.spec.template.metadata.labels`). This means that any Pods it deploys will have the `app=hello-world` label. It is these two attributes that loosely couple the Service to the Deployment's Pods.

When the Deployment and the Service are deployed, the Service will select all 10 Pod replicas and provide them with a stable networking endpoint and load-balance traffic to them.

Services and Endpoint objects

As Pods come-and-go (scaling up and down, failures, rolling updates etc.), the Service dynamically updates its list of healthy matching Pods. It does this through a combination of the label selector and a construct called an *Endpoints* object.

Each Service that is created, automatically gets an associated *Endpoints* object. All this Endpoints object is, is a dynamic list of all of the healthy Pods on the cluster that match the Service's label selector.

It works like this…

Kubernetes is constantly evaluating the Service's label selector against the current list of healthy Pods on the cluster. Any new Pods that match the selector get added to the Endpoints object, and any Pods that disappear get removed. This means the Endpoints object is always up to date. Then, when a Service is sending traffic to Pods, it queries its Endpoints object for the latest list of healthy matching Pods.

When sending traffic to Pods, via a Service, an application will normally query the cluster's internal DNS for the IP address of a Service. It then sends the traffic to this stable IP address and the Service sends it on to a Pod. However, a *Kubernetes-native* application (that's a fancy way of saying an application that understands Kubernetes and can query the Kubernetes API) can query the Endpoints API directly, bypassing the DNS lookup and use of the Service's IP.

Now that you know the fundamentals of how Services work, let's look at some use-cases.

Accessing Services from inside the cluster

Kubernetes supports several *types* of Service. The default type is **ClusterIP**.

A ClusterIP Service has a stable IP address and port that is only accessible from inside the cluster. It's programmed into the network fabric and guaranteed to be stable for the life of the Service. *Programmed into the network fabric* is fancy way of saying the network *just knows about it* and you don't need to bother with the details (stuff like low-level IPTABLES and IPVS rules etc).

Anyway, the ClusterIP gets registered against the name of the Service on the cluster's internal DNS service. All Pods in the cluster are pre-programmed to know about the cluster's DNS service, meaning all Pods are able to resolve Service names.

Let's look at a simple example.

Creating a new Service called "magic-sandbox" will trigger the following. Kubernetes will register the name "magic-sandbox", along with the ClusterIP and port, with the cluster's DNS service. The name, ClusterIP, and port are guaranteed to be long-lived and stable, and all Pods in the cluster send service discovery requests to the internal DNS and will therefore be able to resolve "magic-sandbox" to the ClusterIP. IPTABLES or IPVS rules are distributed across the cluster that ensure traffic sent to the ClusterIP gets routed to Pods on the backend.

Net result… as long as a Pod (application microservice) knows the name of a Service, it can resolve that to its ClusterIP address and connect to the desired Pods.

This only works for Pods and other objects on the cluster, as it requires access to the cluster's DNS service. It does not work outside of the cluster.

Accessing Services from outside the cluster

Kubernetes has another type of Service called a **NodePort Service**. This builds on top of ClusterIP and enables access from outside of the cluster.

You already know that the default Service type is ClusterIP, and it registers a DNS name, virtual IP, and port with the cluster's DNS. A different type of Service, called a NodePort Service builds on this by adding another port that can be used to reach the Service from outside the cluster. This additional port is called the *NodePort*.

The following example represents a NodePort Service:

- **name**: magic-sandbox
- **clusterIP**: 172.12.5.17
- **port**: 8080
- **nodePort**: 30050

This `magic-sandbox` Service can be accessed from inside the cluster via `magic-sandbox` on port `8080`, or `172.12.5.17` on port `8080`. It can also be accessed from outside of the cluster by sending a request to the IP address of any cluster node on port `30050`.

At the bottom of the stack are cluster nodes that host Pods. You add a Service and use labels to associate it with Pods. The Service object has a reliable NodePort mapped to every node in the cluster –- the NodePort value is the same on every node. This means that traffic from outside of the cluster can hit any node in the cluster on the NodePort and get through to the application (Pods).

Figure 6.6 shows a NodePort Service where 3 Pods are exposed externally on port `30050` on every node in the cluster. In step 1, an external client hits **Node2** on port `30050`. In step 2 it is redirected to the Service object (this happens even though **Node2** isn't running a Pod from the Service). Step 3 shows that the Service has an associated Endpoint object with an always-up-to-date list of Pods matching the label selector. Step 4 shows the client being directed to **pod1** on **Node1**.

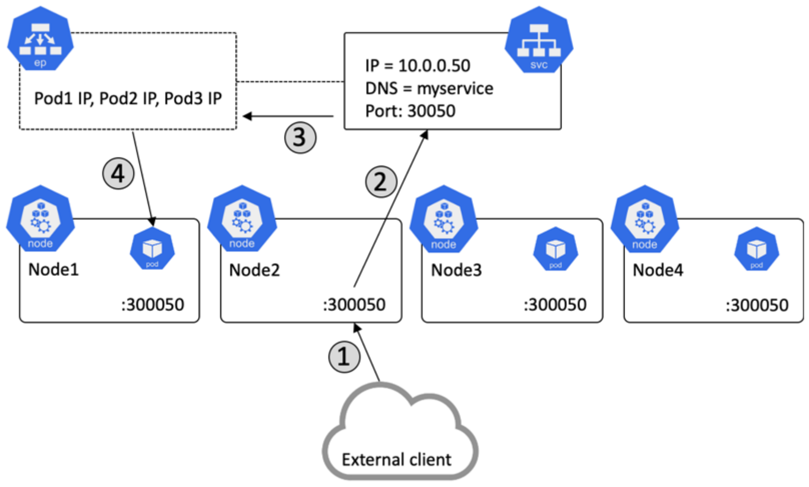


Figure 6.6

The Service could just as easily have directed the client to pod2 or pod3. In fact, future requests may go to other Pods as the Service performs basic load-balancing.

There are other types of Services, such as LoadBalancer, and ExternalName.

LoadBalancer Services integrate with load-balancers from your cloud provider such as AWS, Azure, DO, IBM Cloud, and GCP. They build on top of NodePort Services (which in turn build on top of ClusterIP Services) and allow clients on the internet to reach your Pods via one of your cloud's load-balancers. They're extremely easy to setup. However, they only work if you're running your Kubernetes cluster on a supported cloud platform. E.g. you cannot leverage an ELB load-balancer on AWS if your Kubernetes cluster is running on Microsoft Azure.

ExternalName Services route traffic to systems outside of your Kubernetes cluster (all other Service types route traffic to Pods in your cluster).

Service discovery

Kubernetes implements Service discovery in a couple of ways:

- DNS (preferred)
- Environment variables (definitely not preferred)

DNS-based Service discovery requires the DNS *cluster-add-on* – this is just a fancy name for the native Kubernetes DNS service. I can't remember ever seeing a cluster without it, and if you followed the installation methods from the "Installing Kubernetes" chapter, you'll already have this. Behind the scenes it implements:

- Control plane Pods running a DNS service
- A Service object called `kube-dns` that sits in front of the Pods
- Kubelets program every container with the knowledge of the DNS (via `/etc/resolv.conf`)

The DNS add-on constantly watches the API server for new Services and automatically registers them in DNS. This means every Service gets a DNS name that is resolvable across the entire cluster.

The alternative form of service discovery is through environment variables. Every Pod gets a set of environment variables that resolve every Service currently on the cluster. However, this is an extremely limited fall-back in case you're not using DNS in your cluster.

A major problem with environment variables is that they're only inserted into Pods when the Pod is initially created. This means that Pods have no way of learning about new Services added to the cluster after the Pod itself is created. This is far from ideal, and a major reason DNS is the preferred method. Another limitation can be in clusters with a lot of Services.

Summary of Service theory

Services are all about providing stable networking for Pods. They also provide load-balancing and ways to be accessed from outside of the cluster.

The front-end of a Service provides a stable IP, DNS name and port that is guaranteed not to change for the entire life of the Service. The back-end of a Service uses labels to load-balance traffic across a potentially dynamic set of application Pods.

Hands-on with Services

We're about to get hands-on and put the theory to the test.

You'll augment a simple single-Pod app with a Kubernetes Service. And You'll learn how to do it in two ways:

- The imperative way (not recommended)
- The declarative way

The imperative way

Warning! The imperative way is **not** the Kubernetes way. It introduces the risk that you make imperative changes and never update your declarative manifests, rendering the manifests incorrect and out-of-date. This introduces the risk that stale manifests are subsequently used to update the cluster at a later date, unintentionally overwriting important changes that were made imperatively.

Use `kubectl` to declaratively deploy the following Deployment (later steps will be done imperatively).

The YAML file is called `deploy.yml` and can be found in the `services` folder in the book's GitHub repo.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-deploy
spec:
  replicas: 10
  selector:
    matchLabels:
      app: hello-world
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-ctr
        image: nigelpoulton/k8sbook:latest
        ports:
        - containerPort: 8080
```

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

Now that the Deployment is running, it's time to imperatively deploy a Service for it.

The command to imperatively create a Kubernetes Service is `kubectl expose`. Run the following command to create a new Service that will provide networking and load-balancing for the Pods deployed in the previous step.

```
$ kubectl expose deployment web-deploy \
  --name=hello-svc \
  --target-port=8080 \
  --type=NodePort

service/hello-svc exposed
```

Let's explain what the command is doing. `kubectl expose` is the imperative way to create a new *Service* object. `deployment web-deploy` is telling Kubernetes to expose the `web-deploy` Deployment that you created in the previous step. `--name=hello-svc` tells Kubernetes to name this Service "hello-svc", and `--target-port=8080` tells it which port the app is listening on (this is **not** the cluster-wide NodePort that you'll access the Service on). Finally, `--type=NodePort` tells Kubernetes you want a cluster-wide port for the Service.

Once the Service is created, you can inspect it with the `kubectl describe svc hello-svc` command.

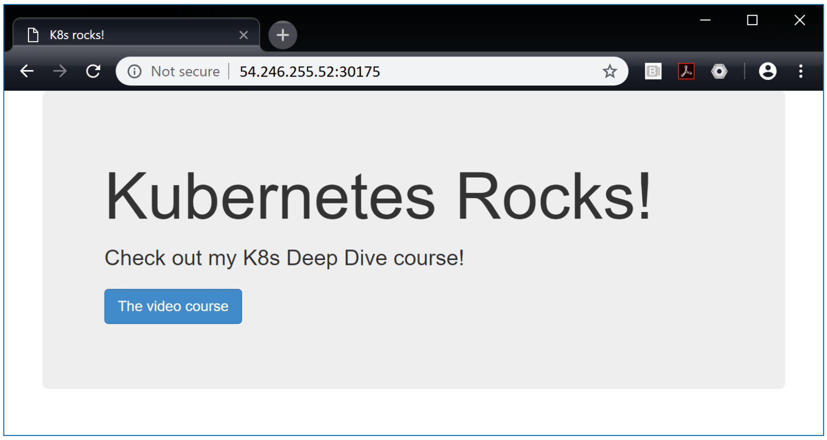
```
$ kubectl describe svc hello-svc
Name:                     hello-svc
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=hello-world
Type:                     NodePort
IP:                       192.168.201.116
Port:                     <unset>  8080/TCP
TargetPort:               8080/TCP
NodePort:                 <unset> 30175/TCP
Endpoints:                192.168.128.13:8080,192.168.128.249:8080, + more...
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

Some interesting values in the output include:

- `Selector` is the list of labels that Pods must have in order for the Service to send traffic to them
- `IP` is the permanent internal ClusterIP (VIP) of the Service
- `Port` is the port that the Service listens on inside the cluster
- `TargetPort` is the port that the application is listening on
- `NodePort` is the cluster-wide port that can be used to access it from outside the cluster
- `Endpoints` is the dynamic list of healthy Pod IPs currently match the Service's label selector.

Now that you know the cluster-wide port that the Service is accessible on (`30175`), you can open a web browser and access the app. In order to do this, you will need to know the IP address of at least one of the nodes in your cluster, and you will need to be able to reach it from your browser – e.g. a publicly routable IP if you're accessing via the internet.

Figure 6.7 shows a web browser accessing a cluster node with an IP address of `54.246.255.52` on the cluster-wide NodePort `30175`.



Kubernetes Rocks!

Check out my K8s Deep Dive course!

Figure 6.7

The app you've deployed is a simple web app. It's built to listen on port `8080`, and you've configured a Kubernetes *Service* to map port `30175` on every cluster node back to port `8080` on the app. By default, cluster-wide ports (NodePort values) are between 30,000 - 32,767. In this example it was dynamically assigned, but you can also specify a port.

Coming up next you're going to see how to do the same thing the proper way – the declarative way. To do that, you need to clean up by deleting the Service you just created. You can do this with the following `kubectl delete svc` command

```
$ kubectl delete svc hello-svc
service "hello-svc" deleted
```

The declarative way

Time to do things the proper way… the Kubernetes way.

A Service manifest file

You'll use the following Service manifest file to deploy the same *Service* that you deployed in the previous section. However, this time you'll specify a value for the cluster-wide port.

```
apiVersion: v1
kind: Service
metadata:
  name: hello-svc
spec:
  type: NodePort
  ports:
  - port: 8080
    nodePort: 30001
    targetPort: 8080
    protocol: TCP
  selector:
    app: hello-world
```

Let's step through some of the lines.

Services are mature objects and are fully defined in the `v1` core API group (`.apiVersion`).

The `.kind` field tells Kubernetes you're defining a Service object.

The `.metadata` section defines a name for the Service. You can also apply labels here. Any labels you add here are used to identify the Service and are not related to labels for selecting Pods.

The `.spec` section is where you actually define the Service. In this example, you're telling Kubernetes to deploy a `NodePort` Service. The `port` value configures the Service to listen on port `8080` for internal requests, and the `NodePort` value tells it to listen on `30001` for external requests. The `targetPort` value is part of the Service's back-end configuration and tells Kubernetes to send traffic to the application Pods on port `8080`. Then you're explicitly telling it to use TCP (default).

Finally, `.spec.selector` tells the Service to send traffic to all Pods in the cluster that have the `app=hello-world` label. This means it will provide stable networking and load-balancing across all Pods with that label.

Before deploying and testing the Service, let's remind ourselves of the major Service types.

Common Service types

The three common *ServiceTypes* are:

- `ClusterIP`. This is the default option and gives the *Service* a stable IP address internally within the cluster. It will not make the Service available outside of the cluster.
- `NodePort`. This builds on top of `ClusterIP` and adds a cluster-wide TCP or UDP port. It makes the Service available outside of the cluster on a stable port.
- `LoadBalancer`. This builds on top of `NodePort` and integrates with cloud-based load-balancers.

There's another Service type called `ExternalName`. This is used to direct traffic to services that exist outside of the Kubernetes cluster.

The manifest needs POSTing to the API server. The simplest way to do this is with `kubectl apply`.

The YAML file is called `svc.yml` and can be found in the `services` folder of book's GitHub repo.

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

This command tells Kubernetes to deploy a new object from a file called `svc.yml`. The `.kind` field in the YAML file tells Kubernetes that you're deploying a new Service object.

Introspecting Services

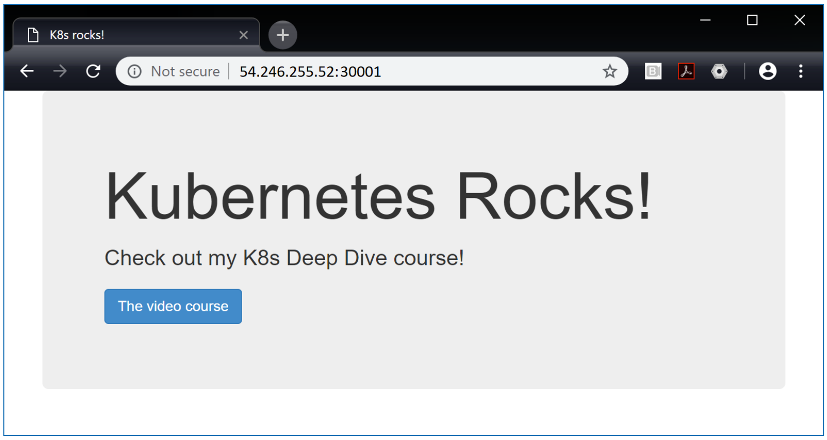
Now that the Service is deployed, you can inspect it with the usual `kubectl get` and `kubectl describe` commands.

```
$ kubectl get svc hello-svc
NAME        TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
hello-svc   NodePort   100.70.40.2     <none>        8080:30001/TCP   8s

$ kubectl describe svc hello-svc
Name:                     hello-svc
Namespace:                default
Labels:                   <none>
Annotations:              kubectl.kubernetes.io/last-applied-configuration...
Selector:                 app=hello-world
Type:                     NodePort
IP:                       100.70.40.2
Port:                     <unset>  8080/TCP
TargetPort:               8080/TCP
NodePort:                 <unset>  30001/TCP
Endpoints:                100.96.1.10:8080, 100.96.1.11:8080, + more...
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

In the previous example, you exposed the Service as a `NodePort` on port `30001` across the entire cluster. This means you can point a web browser to that port on any node and reach the Service and the Pods it's proxying. You will need to use the IP address of a node you can reach, and you will need to make sure that any firewall and security rules allow the traffic to flow.

Figure 6.8 shows a web browser accessing the app via a cluster node with an IP address of `54.246.255.52` on the cluster-wide port `30001`.



Kubernetes Rocks!

Check out my K8s Deep Dive course!

Figure 6.8

Endpoints objects

Earlier in the chapter, we said that every Service gets its own Endpoints object with the same name as the Service. This object holds a list of all the Pods the Service matches and is dynamically updated as matching Pods come and go. You can see Endpoints with the normal `kubectl` commands.

In the following command, you use the Endpoint controller's `ep` shortname.

```
$ kubectl get ep hello-svc
NAME         ENDPOINTS                                        AGE
hello-svc    100.96.1.10:8080, 100.96.1.11:8080 + 8 more...   1m

$ Kubectl describe ep hello-svc
Name:         hello-svc
Namespace:    default
Labels:       <none>
Annotations:  endpoints.kubernetes.io/last-change...
Subsets:
  Addresses:  100.96.1.10,100.96.1.11,100.96.1.12...
  NotReadyAddresses:   <none>
  Ports:
    Name      Port     Protocol
    ----      ----     --------
    <unset>   8080     TCP
Events: <none>
```

Summary of deploying Services

As with all Kubernetes objects, the preferred way of deploying and managing Services is the declarative way. Labels allow them to send traffic to a dynamic set of Pods. This means you can deploy new Services that will work with Pods and Deployments that are already running on the cluster and already in-use. Each Service gets its own Endpoints object that maintains an up-to-date list of matching Pods.

Real world example

Although everything you've learned so far is cool and interesting, the important questions are: *How does it bring value?* and *How does it keep businesses running and make them more agile and resilient?*

Let's take a minute to run through a common real-world example – making updates to applications.

We all know that updating applications is a fact of life – bug fixes, new features, performance improvements etc.

Figure 6.9 shows a simple application deployed on a Kubernetes cluster as a bunch of Pods managed by a Deployment. As part of it, there's a Service selecting on Pods with labels that match `app=biz1` and `zone=prod` (notice how the Pods have both of the labels listed in the label selector). The application is up and running.

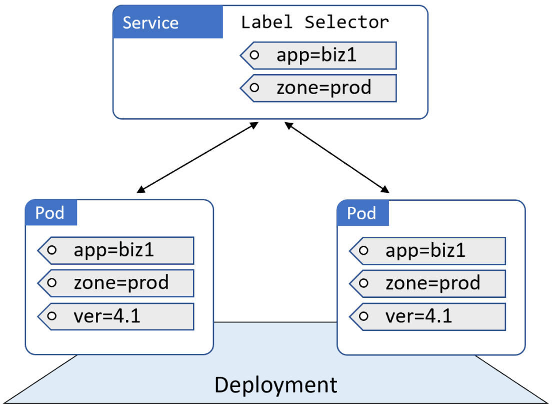


Figure 6.9

Now assume you need to push a new version, but you need to do it without causing downtime.

To do this, you can add Pods running the new version of the app as shown in Figure 6.10.

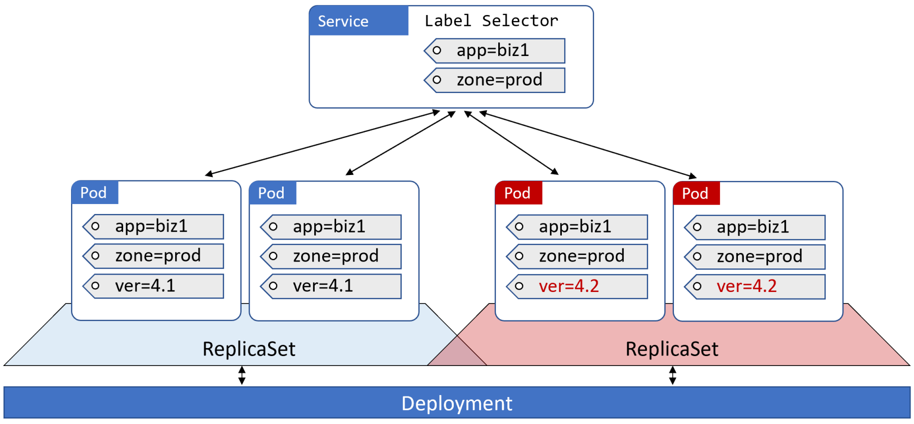


Figure 6.10

Behind the scenes, the updated *Pods* are labelled so that they match the existing label selector. The Service is now load-balancing requests across **both versions of the app** (`version=4.1` and `version=4.2`). This happens because the Service's label selector is being constantly evaluated, and its Endpoint object is constantly being updated with new matching Pods.

Once you're happy with the updated version, forcing **all traffic** to use it is as simple as updating the Service's label selector to include the label `version=4.2`. Suddenly the older Pods no longer match, and the Service will only forward traffic to the new version (Figure 6.11).

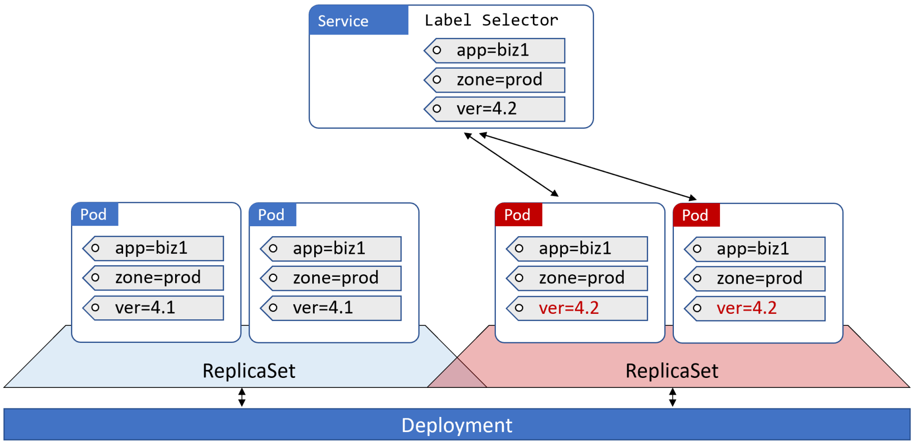


Figure 6.11

However, the old version still exists, you're just not sending traffic to it anymore. This means that if you experience an issue with the new version, you can switch back to the previous version by simply changing the label selector on the Service to select on `version=4.1` instead of `version=4.2`. See Figure 6.12.

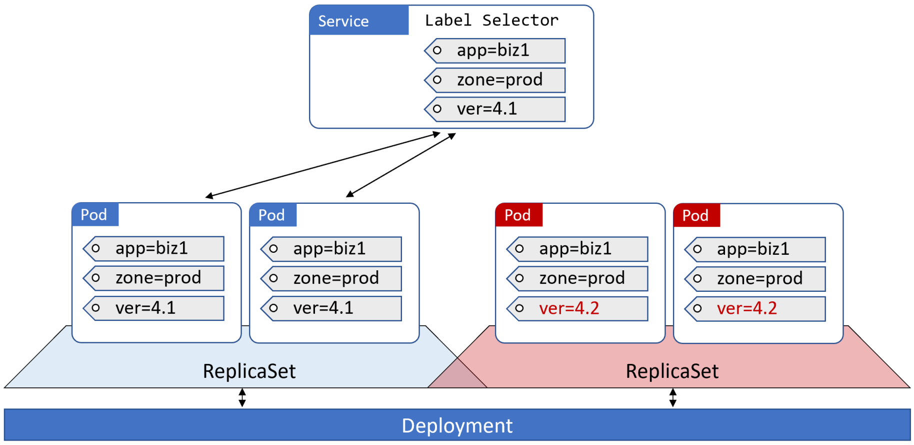


Figure 6.12

Now everybody's getting the old version.

This functionality can be used for all kinds of things – blue-greens, canaries, you name it. So simple, yet so powerful.

Clean-up the lab with the following commands. These will delete the Deployment and Service used in the examples.

```
$ kubectl delete -f deploy.yml
$ kubectl delete -f svc.yml
```

Chapter Summary

In this chapter, you learned that *Services* bring stable and reliable networking to apps deployed on Kubernetes. They also perform load-balancing and allow you to expose elements of your application to the outside world (outside of the Kubernetes cluster).

The front-end of a Service is fixed, providing stable networking for the Pods behind it. The back-end of a Service is dynamic, allowing Pods to come and go without impacting the ability of the Service to provide load-balancing.

Services are first-class objects in the Kubernetes API and can be defined in the standard YAML manifest files. They use label selectors to dynamically match Pods, and the best way to work with them is declaratively.

7: Service discovery

In this chapter, you'll learn what service discovery is, why it's important, and how it's implemented in Kubernetes. You'll also learn some troubleshooting tips.

To get the most from this chapter, you should know what a Kubernetes Service object is and how they work. This was covered in the previous chapter.

This chapter is split into the following sections:

- Quick background
- Service registration
- Service discovery
- Service discovery and Namespaces
- Troubleshooting service discovery

Quick background

Applications run inside of containers and containers run inside of Pods. Every Kubernetes Pod gets its own unique IP address, and all Pods connect to the same flat network called the *Pod network*. However, Pods are ephemeral. This means they come and go and should not be considered reliable. For example, scaling operations, rolling updates, rollbacks and failures all cause Pods to be added or removed from the network.

To address the unreliable nature of Pods, Kubernetes provides a *Service* object that sits in front of a set of Pods and provides a reliable name, IP address, and port. Clients connect to the Service object, which in turn load-balances requests to the target Pods.

> **Note**: The word "service" has lots of meanings. When we use it with a capital "S" we are referring to the Kubernetes Service object that provides stable networking to a set of Pods.

Modern cloud-native applications are comprised of lots of small independent microservices that work together to create a useful application. For these microservices to work together, they need to be able to discover and connect to each other. This is where *service discovery* comes into play.

There are two major components to service discovery:

- Service registration
- Service discovery

Service registration

Service registration is the process of a microservice registering its connection details in a *service registry* so that other microservices can discover it and connect to it.

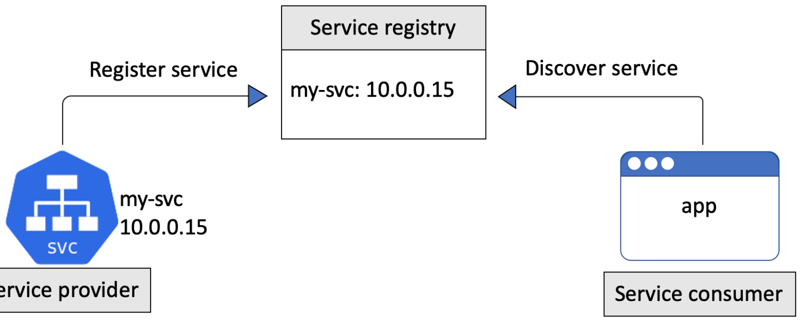


Figure 7.1

A few important things to note about this in Kubernetes:

1. Kubernetes uses an internal DNS service as its service registry
2. Services register with DNS (not individual Pods)
3. The *name*, *IP address*, and *network port* of every Service is registered

For this to work, Kubernetes provides a *well-known* internal DNS service that we usually call the "cluster DNS". The term *well known* means that it operates at an address known to every Pod and container in the cluster. It's implemented in the `kube-system` Namespace as a set of Pods managed by a Deployment called `coredns`. These Pods are fronted by a Service called `kube-dns`. Behind the scenes, it's based on a DNS technology called CoreDNS and runs as a *Kubernetes-native application.*

The previous sentence contains a lot of detail, so the following commands show how its implemented. You can run these commands on your own Kubernetes clusters.

```
$ kubectl get svc -n kube-system -l k8s-app=kube-dns
NAME       TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                  AGE
kube-dns   ClusterIP   192.168.200.10   <none>        53/UDP,53/TCP,9153/TCP   3h44m

$ kubectl get deploy -n kube-system -l k8s-app=kube-dns
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
coredns   2/2     2            2           3h45m

kubectl get pods -n kube-system -l k8s-app=kube-dns
NAME                       READY   STATUS    RESTARTS   AGE
coredns-5644d7b6d9-fk4c9   1/1     Running   0          3h45m
coredns-5644d7b6d9-s5zlr   1/1     Running   0          3h45m
```

Every Kubernetes Service is automatically registered with the cluster DNS when it's created. The registration process looks like this (exact flow might slightly differ):

1. You POST a new Service manifest to the API Server
2. The request is authenticated, authorized, and subjected to admission policies
3. The Service is allocated a virtual IP address called a ClusterIP
4. An Endpoints object (or Endpoint slices) is created to hold a list of Pods the Service will load-balance traffic to
5. The *Pod network* is configured to handle traffic sent to the ClusterIP (more on this later)
6. The Service's name and IP are registered with the cluster DNS

Step 6 is the secret sauce in the service registration process.

We mentioned earlier that the cluster DNS is a *Kubernetes-native application*. This means it knows it's running on Kubernetes and implements a controller that watches the API Server for new Service objects. Any time it observes a new Service object, it creates the DNS records that allow the Service name to be resolved to its ClusterIP. This means that applications and Services do not need to perform service registration – the cluster DNS is constantly looking for new Services and automatically registers their details.

It's important to understand that the name registered for the Service is the value stored in its `metadata.name` property. The ClusterIP is dynamically assigned by Kubernetes.

```
apiVersion: v1
kind: Service
metadata:
  name: ent <<---- Name registered with cluster DNS
spec:
  selector:
    app: web
  ports:
    ...
```

At this point, the front-end configuration of the Service is registered (name, IP, port) and the Service can be discovered by applications running in other Pods.

The service back-end

Now that the front-end of the Service is registered, the back-end needs building. This involves creating and maintaining a list of Pod IPs that the Service will load-balance traffic to.

As explained in the previous chapter, every Service has a `label selector` that determines which Pods the Service will load-balance traffic to. See Figure 7.2.

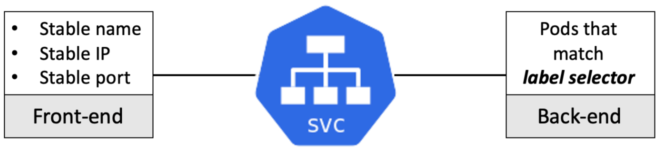


Figure 7.2

Kubernetes automatically creates an Endpoints object (or Endpoint slices) for every Service. These hold the list of Pods that match the label selector and will receive traffic from the Service. They're also critical to how traffic is routed from the Service's ClusterIP to Pod IPs (more on this soon).

The following command shows an Endpoints object for a Service called `ent`. It has the IP address and port of two Pods that match the label selector.

```
$ kubectl get endpoint ent
NAME     ENDPOINTS                                  AGE
ent      192.168.129.46:8080,192.168.130.127:8080     14m
```

Figure 7.3 shows a Service called `ent` that will load-balance to two Pods. It also shows the Endpoints object with the IPs of the two Pods that match the Service's label selector.

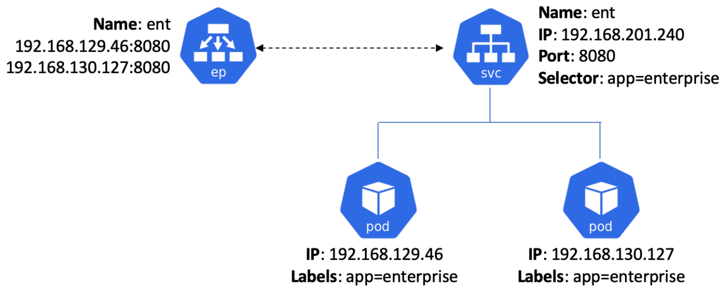


Figure 7.3

The kubelet process on every node is watching the API Server for new Endpoints objects. When it sees them, it creates local networking rules that redirect ClusterIP traffic to Pod IPs. In modern Linux-based Kubernetes cluster the technology used to create these rules is the Linux IP Virtual Server (IPVS). Older versions of Kubernetes used iptables.

At this point the Service is fully registered and ready to be discovered:

- Its front-end configuration is registered with DNS
- Its back-end configuration is stored in an Endpoints object (or Endpoint slices) and the network is ready to handle traffic

Let's summarise the service registration process with the help of a simple flow diagram.

Summarising service registration

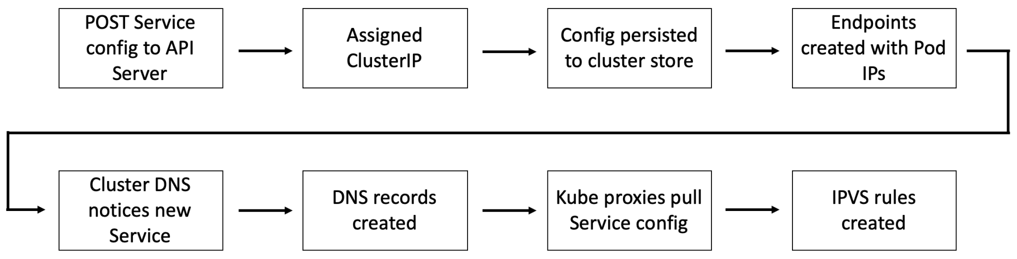

POST Service config to API Server
Assigned ClusterIP
Config persisted to cluster store
Endpoints created with Pod IPs
Cluster DNS notices new Service
DNS records created
Kube proxies pull Service config
IPVS rules created

Figure 7.4

You POST a new Service configuration to the API Server and the request is authenticated and authorized. The Service is allocated a ClusterIP and its configuration is persisted to the cluster store. An associated Endpoints object is created to hold the list of Pod IPs that match the label selector. The cluster DNS is running as a Kubernetes-native application and watching the API Server for new Service objects. It sees the new Service and registers the appropriate DNS A and SRV records. Every node is running a kube-proxy that sees the new Service and Endpoints objects and creates IPVS rules on every node so that traffic to the Service's ClusterIP is redirected to one of the Pods that match its label selector.

Service discovery

Let's assume there are two microservices applications on a single Kubernetes cluster – `enterprise` and `voyager`. The Pods for the `enterprise` app sit behind a Kubernetes Service called `ent` and the Pods for the `voyager` app sit behind another Kubernetes Service called `voy`.

Both are registered with DNS as follows:

- `ent`: 192.168.201.240
- `voy`: 192.168.200.217

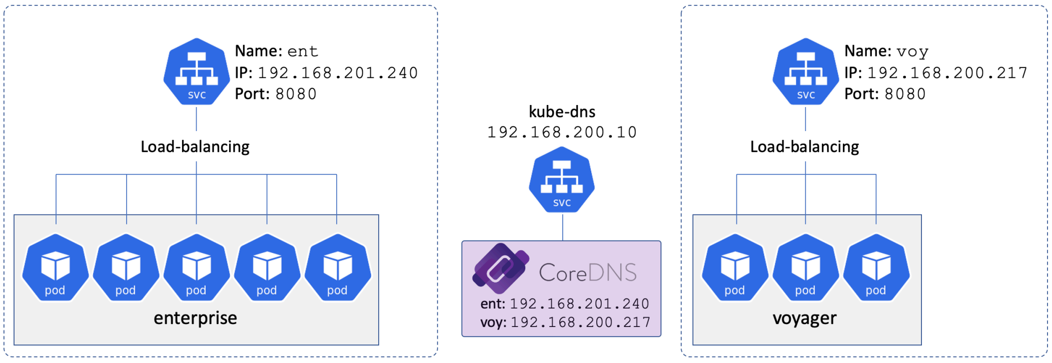


Figure 7.5

For service discovery to work, every microservice needs to know two things:

1. The **name** of the remote microservice they want to connect to
2. How to convert the **name** to an IP address

The application developer is responsible for point 1 – coding the microservice with the names of microservices they connect to. Kubernetes takes care of point 2.

Converting names to IP addresses using the cluster DNS

Kubernetes automatically configures every container so that it can find and use the cluster DNS to convert Service names to IPs. It does this by populating every container's `/etc/resolv.conf` file with the IP address of cluster DNS Service as well as any search domains that should be appended to unqualified names.

> **Note**: An "unqualified name" is a short name such as `ent`. Appending a search domain converts an unqualified name into a fully qualified domain name (FQDN) such as `ent.default.svc.cluster.local`.

The following snippet shows a container that is configured to send DNS queries to the cluster DNS at 192.168.200.10. It also lists the search domains to append to unqualified names.

```
$ cat /etc/resolv.conf
search svc.cluster.local cluster.local default.svc.cluster.local
nameserver 192.168.200.10
options ndots:5
```

The following snippet shows that `nameserver` in `/etc/resolv.conf` 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   192.168.200.10    53/UDP,53/TCP,9153/TCP   3h53m
```

If Pods in the `enterprise` app need to connect to Pods in the `voyager` app, they send a request to the cluster DNS asking it to resolve the name `voy` to an IP address. The cluster DNS will return the value of the ClusterIP (192.168.200.217).

At this point, the `enterprise` Pods have an IP address to send traffic to. However, this ClusterIP is a virtual IP (VIP) that requires some more network magic in order for requests to reach `voyager` Pods.

Some network magic

Once a Pod has the ClusterIP of a Service, it sends traffic to that IP address. However, the address is on a special network called the *service network* and there are no routes to it! This means the apps container doesn't know where to send the traffic, so it sends it to its *default gateway*.

> **Note**: A *default gateway* is where a device sends traffic that it doesn't have a specific route for. The *default gateway* will normally forward traffic to another device with a larger routing table that might have a route for the traffic. A simple analogy might be driving from City A to City B. The local roads in City A probably don't have signposts to City B, so you follow signs to the major highway/motorway. Once on the highway/motorway there is more chance that you will find directions to City B. If the first signpost doesn't have directions to City B you keep driving until you see a signpost for City B. Routing is similar, if a device doesn't have a route for the destination network, the traffic is routed from one default gateway to the next until hopefully a device has a route to the required network.

The containers default gateway sends the traffic to the Node it is running on.

The Node doesn't have a route to the *service network* either, so it sends the traffic to its own default gateway. Doing this causes the traffic to be processed by the Nodes kernel, which is where the magic happens!

Every Kubernetes Node runs a system service called `kube-proxy`. At a high-level, `kube-proxy` is responsible for capturing traffic destined for ClusterIPs and redirecting it to the IP addresses of Pods that match the Service's label selector. Let's look a bit closer…

`kube-proxy` is a Pod-based Kubernetes-native app that implements a controller that watches the API Server for new Service and Endpoints objects. When it sees them, it creates local IPVS rules that tell the Node to intercept traffic destined for the Service's ClusterIP and forward it to individual Pod IPs.

This means that every time a Nodes kernel processes traffic headed for an address on the *service network*, a trap occurs and the traffic is redirected to the IP of a healthy Pod matching the Service's label selector.

> Kubernetes originally used iptables to do this trapping and load-balancing. However, it was replaced by IPVS in Kubernetes 1.11. The is because IPVS is a high-performance kernel-based L4 load-balancer that scales better than iptables and implements better load-balancing algorithms.

Summarising service discovery

Let's quickly summarise the service discovery process with the help of the flow diagram in Figure 7.6.

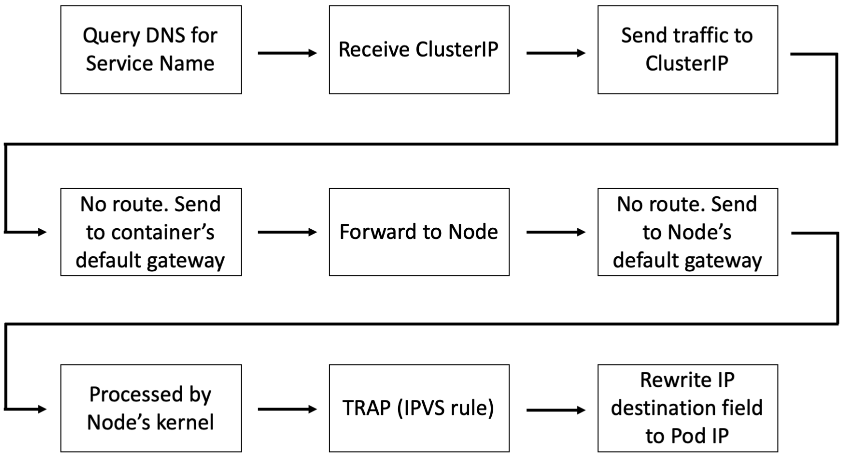

Query DNS for Service Name
Receive ClusterIP
Send traffic to ClusterIP
No route. Send to container's default gateway
Forward to Node
No route. Send to Node's default gateway
Processed by Node's kernel
TRAP (IPVS rule)
Rewrite IP destination field to Pod IP

Figure 7.6

Assume a microservice called "enterprise" needs to send traffic to a microservice called "voyager". To start this flow, the "enterprise" microservice needs to know the name of the Kubernetes Service object sitting in front of the "voyager" microservice B. We'll assume it's called "voy", but it is the responsibility of the application developer to ensure this is known.

An instance of the "enterprise" microservice sends a query to the cluster DNS (defined in the `/etc/resolv.conf` file of every container) asking it to resolve the name of the "voy" Service to an IP address. The cluster DNS replies with the ClusterIP (virtual IP) and the instance of the "enterprise" microservice sends requests to this ClusterIP. However, there are no routes to the *service network* that the ClusterIP is on. This means the requests are sent to the container's default gateway and eventually sent to the Node the container is running on. The Node has no route to the *service network* so it sends the traffic to its own default gateway. En-route, the request is processed by the Node's kernel. A trap is triggered and the request is redirected to the IP address of a Pod that matches the Services label selector.

The Node has routes to Pod IPs and the requests reach a Pod and are processed.

Service discovery and Namespaces

Two things are important if you want to understand how service discovery works *within* and *across* Namespaces:

1. Every cluster has an *address space*
2. Namespaces partition the cluster address space

Every cluster has an address space based on a DNS domain that we usually call the *cluster domain*. By default, it's called `cluster.local`, and Service objects are placed within that address space. For example, a Service called `ent` will have a fully qualified domain name (FQDN) of `ent.default.svc.cluster.local`

The format of the FQDN is `<object-name>.<namespace>.svc.cluster.local`

Namespaces allow you to partition the address space below the cluster domain. For example, creating a couple of Namespaces called `prod` and `dev` will give you two address spaces that you can place Services and other objects in:

- dev: <object-name>.dev.svc.cluster.local
- prod: <object-name>.prod.svc.cluster.local

Object names must be unique *within* Namespaces but not *across* Namespaces. This means that you cannot have two Service objects called "ent" in the same Namespace, but you can if they are in different Namespaces. This is useful for parallel development and production configurations. For example, Figure 7.7 shows a single cluster address divided into `dev` and `prod` with identical instances of the `ent` and `voy` Service are deployed to each.

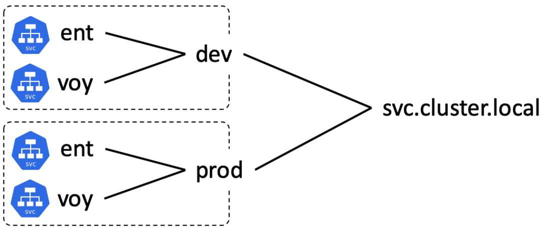


Figure 7.7

Pods in the `prod` Namespace can connect to Services in the local Namespace using short names such as `ent` and `voy`. To connect to objects in a remote Namespace requires FQDNs such as `ent.dev.svc.cluster.local` and `voy.dev.svc.cluster.local`.

As we've seen, Namespaces partition the cluster address space. They are also good for implementing access control and resource quotas. However, they are not workload isolation boundaries and should not be used to isolate hostile workloads.

Service discovery example

Let's walk through a quick example.

The following YAML is called `sd-example.yml` in the `service-discovery` folder of the books GitHub repo. It defines two Namespaces, two Deployments, two Services, and a standalone jump Pod. The two Deployments have identical names, as do the Services. However, they're deployed to different Namespace, so this is allowed. The jump Pod is deployed to the `dev` Namespace.

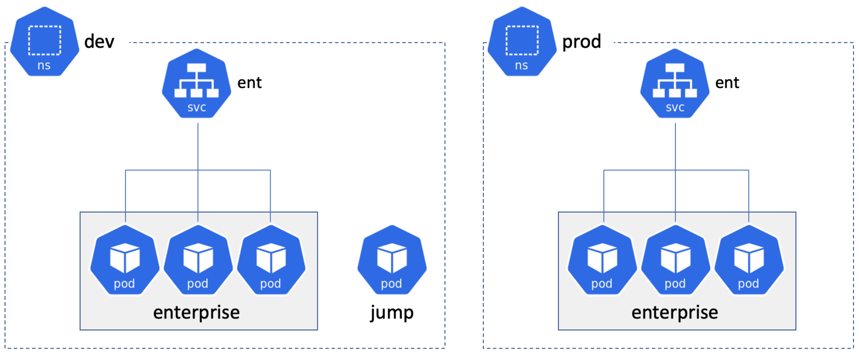


Figure 7.8

```
apiVersion: v1
kind: Namespace
metadata:
  name: dev
---
apiVersion: v1
kind: Namespace
metadata:
  name: prod
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: enterprise
  labels:
    app: enterprise
  namespace: dev
spec:
  selector:
    matchLabels:
      app: enterprise
  replicas: 2
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: enterprise
    spec:
      terminationGracePeriodSeconds: 1
      containers:
      - image: nigelpoulton/k8sbook:text-dev
        name: enterprise-ctr
        ports:
        - containerPort: 8080
```

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: enterprise
  labels:
    app: enterprise
  namespace: prod
spec:
  selector:
    matchLabels:
      app: enterprise
  replicas: 2
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: enterprise
    spec:
      terminationGracePeriodSeconds: 1
      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
```

Deploy the configuration to your cluster.

```
$ kubectl apply -f dns-namespaces.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 the configuration was correctly applied. The following outputs are trimmed and do not show all objects.

```
$ kubectl get all -n dev
NAME          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
service/ent   ClusterIP   192.168.202.57   <none>        8080/TCP   43s

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/enterprise   2/2     2            2           43s
<snip>

$ kubectl get all -n prod
NAME          TYPE        CLUSTER-IP        EXTERNAL-IP   PORT(S)    AGE
service/ent   ClusterIP   192.168.203.158   <none>        8080/TCP   82s

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/enterprise   2/2     2            2           52s
<snip>
```

The next steps will:

1. Log on to the main container of jump Pod in the `dev` Namespace
2. Check the container's `/etc/resolv.conf` file
3. Connect to the ent app in the `dev` Namespace using the Service's shortname
4. Connect to the ent app in the `prod` Namespace using the Service's FQDN

To help with the demo, the versions of the ent app used in each Namespace are different.

Log on to the jump Pod.

```
$ kubectl exec -it jump -n dev -- bash
root@jump:/#
```

Your terminal prompt will change to indicate you are attached to the jump Pod.

Inspect the contents of the `/etc/resolv.conf` file and check that the search domains listed include the `dev` Namespace (`search dev.svc.cluster.local`) and not the `prod` Namespace.

```
$ cat /etc/resolv.conf
search dev.svc.cluster.local svc.cluster.local cluster.local default.svc.cluster.local
nameserver 192.168.200.10
options ndots:5
```

The `nameserver` value will match the ClusterIP of the `kube-dns` Service on your cluster. This is the well-known IP address that handles DNS/service discovery traffic.

Install the `curl` utility.

```
$ apt-get update && apt-get install curl -y
<snip>
```

Use `curl` to connect to the version of the app running in `dev` by using the `ent` short name.

```
$ curl ent:8080
Hello from the DEV Namespace!
Hostname: enterprise-7d49557d8d-k4jjz
```

The "Hello from the DEV Namespace" response proves that `curl` connected to the dev instance of the app.

When the `curl` command was issued, the container automatically appended `dev.svc.cluster.local` to the `ent` name and sent the query to the IP address of the cluster DNS specified in `/etc/resolv.conf`. DNS returned the ClusterIP for the `ent` Service running in the local `dev` Namespace and the app sent the traffic to that address. En-route to the Nodes default gateway the traffic triggered a trap in the Node's kernel and was redirected to one of the Pods hosting the simple web application.

Run the `curl` command again, but this time append the domain name of the `prod` Namespace. This will cause the cluster DNS to return the ClusterIP for the instance in the `prod` Namespace and traffic will eventually reach a Pod running in `prod`

```
$ curl ent.prod.svc.cluster.local:8080
Hello from the PROD Namespace!
Hostname: enterprise-5464d8c4f9-v7xsk
```

This time the response comes from a Pod in the `prod` Namespace.

The test proves that short names are resolved to the local Namespace (the same Namespace the app is running in) and connecting across Namespaces requires FQDNs.

Remember to exit your terminal from the container by typing `exit`.

Troubleshooting service discovery

Service registration and discovery involves a lot of moving parts. If a single one of them stops working, the whole process can potentially break. Let's quickly run through what needs to be working and how to check them.

Kubernetes uses the cluster DNS as its service registry. It runs as a set of Pods in the `kube-system` Namespace with a Service object providing a stable network endpoint. The important components are:

- Pods: Managed by the `coredns` Deployment
- Service: A ClusterIP Service called `kube-dns` listening on port 53 TCP/UDP
- Endpoint: Also called `kube-dns`

All objects relating to the cluster DNS are tagged with the `k8s-app=kube-dns` label. This is helpful when filtering `kubectl` output.

Make sure that the `coredns` Deployment and its managed Pods are up and running.

```
$ kubectl get deploy -n kube-system -l k8s-app=kube-dns
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
coredns   2/2     2            2           2d21h

$ kubectl get pods -n kube-system -l k8s-app=kube-dns
NAME                       READY   STATUS    RESTARTS   AGE
coredns-5644d7b6d9-74pv7   1/1     Running   0          2d21h
coredns-5644d7b6d9-s759f   1/1     Running   0          2d21h
```

Check the logs from the each of the `coredns` Pods. You'll need to substitute the names of the Pods from your own environment. The following output is typical of a working DNS Pod.

```
$ kubectl logs coredns-5644d7b6d9-74pv7 -n kube-system
2020-02-19T21:31:01.456Z [INFO] plugin/reload: Running configuration...
2020-02-19T21:31:01.457Z [INFO] CoreDNS-1.6.2
2020-02-19T21:31:01.457Z [INFO] linux/amd64, go1.12.8, 795a3eb
CoreDNS-1.6.2
linux/amd64, go1.12.8, 795a3eb
```

Assuming the Pods and Deployment are working, you should also check the Service and associated Endpoints object. The output should show that 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 should match the IP address in the `/etc/resolv.conf` files of all containers running on the cluster. If the IP addresses are different, containers will send DNS requests to the wrong IP address.

```
$ kubectl get svc kube-dns -n kube-system
NAME       TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                  AGE
kube-dns   ClusterIP   192.168.200.10   <none>        53/UDP,53/TCP,9153/TCP   2d21h
```

The associated `kube-dns` Endpoints object should also be up and have the IP addresses of the `coredns` Pods listening on port 53 TCP and UDP.

```
$ kubectl get ep -n kube-system -l k8s-app=kube-dns
NAME       ENDPOINTS                                                        AGE
kube-dns   192.168.128.24:53,192.168.128.3:53,192.168.128.24:53 + 3 more...   2d21h
```

Once you've verified the fundamental DNS components are up and working, you can proceed to perform more detailed and in-depth troubleshooting. Here are a few basic tips.

Start a troubleshooting Pod that has your favourite networking tools installed (ping, traceroute, curl, dig, nslookup etc.). The standard `gcr.io/kubernetes-e2e-test-images/dnsutils:1.3` image is a popular choice if you don't have your own custom image with your tools installed. Unfortunately, there is no `latest` image in the repo. This means you have to specify a version. At the time of writing, 1.3 was the latest version.

The following command will start a new standalone Pod called netutils, based on the dnsutils image just mentioned. It will also log your terminal on to it.

```
$ kubectl run -it dnsutils \
  --image gcr.io/kubernetes-e2e-test-images/dnsutils:1.3
```

A common way to test DNS resolution is to use `nslookup` to resolve the `kubernetes.default` Service that sits in front of the API Server. The query should return an IP address and the name `kubernetes.default.svc.cluster.local`.

```
# nslookup kubernetes
Server:         192.168.200.10
Address:        192.168.200.10#53

Name:   kubernetes.default.svc.cluster.local
Address: 192.168.200.1
```

The first two lines should return the IP address of your cluster DNS. The last two lines 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 possible indicators that DNS is not working. A possible solution is to restart the `coredns` Pods. These are managed by a Deployment object and will be automatically recreated.

The following command deletes the DNS Pods and must be ran from a terminal with `kubectl` installed. If you're still logged on to the `netutils` Pod you'll need to type `exit` to log off.

```
$ kubectl delete pod -n kube-system  -l k8s-app=kube-dns
pod "coredns-5644d7b6d9-2pdmd" deleted
pod "coredns-5644d7b6d9-wsjzp" deleted
```

Verify that the Pods have restarted and test DNS again.

Summary

In this chapter, you learned that Kubernetes uses the internal cluster DNS for service registration and service discovery.

All new Service objects are automatically registered with the cluster DNS and all containers are configured to know where to find the cluster DNS. This means that all containers will talk to the cluster DNS when they need to resolve a name to an IP address.

The cluster DNS resolves Service names to ClusterIPs. These IP addresses are on a special network called the *service network* and there are no routes to this network. Fortunately, every cluster Node is configured to trap on packets destined for the *service network* and redirect them to Pod IPs on the Pod network.

8: Kubernetes storage

Storage is critical to most real-world production applications. Fortunately, Kubernetes has a mature and feature-rich storage subsystem called the *persistent volume subsystem*.

We'll divide this chapter as follows:

- The big picture
- Storage provisioners
- The Container Storage Interface (CSI)
- The Kubernetes persistent volume subsystem
- Storage Classes and Dynamic Provisioning
- Demo

The big picture

First things first, Kubernetes supports lots of types of storage from lots of different places. For example, iSCSI, SMB, NFS, and object storage blobs, all from a variety of external storage systems that can be in the cloud or in your on-premises data center. However, no matter what type of storage you have, or where it comes from, when it's exposed on your Kubernetes cluster it's called a ***volume***. For example, Azure File resources surfaced in Kubernetes are called *volumes*, as are block devices from AWS Elastic Block Store. All storage on a Kubernetes cluster is called a *volume*.

Figure 8.1 shows the high-level architecture.

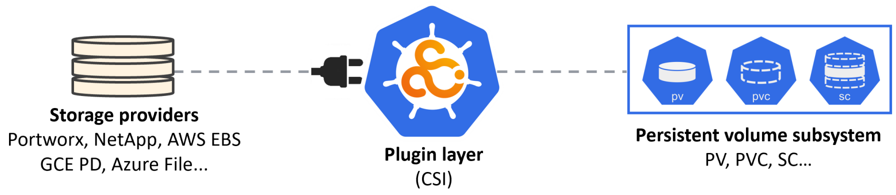


Figure 8.1

On the left, you've got storage providers. They can be your traditional enterprise storage arrays from vendors like EMC and NetApp, or they can be cloud storage services such as AWS Elastic Block Store (EBS) and GCE Persistent Disks (PD). All you need, is a plugin that allows their storage resources to be surfaced as volumes in Kubernetes.

In the middle of the diagram is the plugin layer. In the simplest terms, this is the glue that connects external storage with Kubernetes. Going forward, plugins will be based on the Container Storage Interface (CSI) which is an open standard aimed at providing a clean interface for plugins. If you're a developer writing storage plugins, the CSI abstracts the internal Kubernetes storage detail and lets you develop *out-of-tree*.

> **Note**: Prior to the CSI, all storage plugins were implemented as part of the main Kubernetes code tree (*in-tree*). This meant they all had to be open-source, and all updates and bug-fixes were tied to the main Kubernetes release-cycle. This was a nightmare for plugin developers as well as the Kubernetes maintainers. However, now that we have the CSI, storage vendors no longer need to open-source their code, and they can release updates and bug-fixes against their own timeframes.

On the right of Figure 8.1 is the Kubernetes persistent volume subsystem. This is a set of API objects that allow applications to consume storage. At a high-level, Persistent Volumes (PV) are how you map external storage onto the cluster, and Persistent Volume Claims (PVC) are like tickets that authorize applications (Pods) to use a PV.

Let's assume the quick example shown in Figure 8.2.

A Kubernetes cluster is running on AWS and the AWS administrator has created a 25GB EBS volume called "ebs-vol". The Kubernetes administrator creates a PV called "k8s-vol" that links back to the "ebs-vol" via the `kubernetes.io/aws-ebs` plugin. While that might sound complicated, it's not. The PV is simply a way of representing the external storage on the Kubernetes cluster. Finally, the Pod uses a PVC to claim access to the PV and start using it.

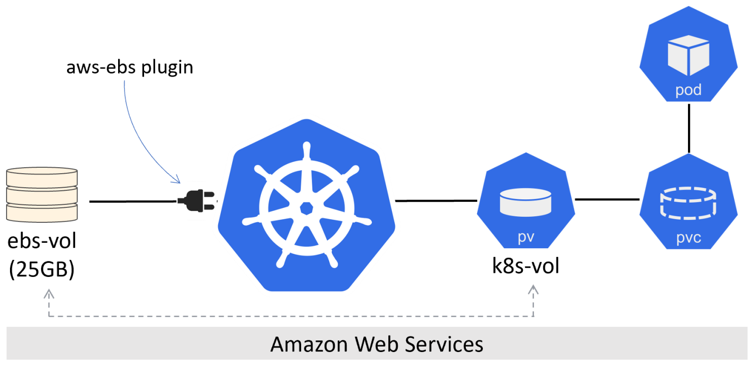


Figure 8.2

A couple of points worth noting.

1. There are rules safeguarding access to a single volume from multiple Pods (more on this later).
2. A single external storage volume can only be used by a single PV. For example, you cannot have a 50GB external volume that has two 25GB Kubernetes PVs each using half of it.

Now that you have an idea of the fundamentals, let's dig a bit deeper.

Storage Providers

Kubernetes can use storage from a wide range of external systems. These will often be native cloud services such as `AWSElasticBlockStore` or `AzureDisk`, but they can also be traditional on-premises storage arrays providing `iSCSI` or `NFS` volumes. Other options exist, but the take-home point is that Kubernetes gets its storage from a wide range of external systems.

Some obvious restrictions apply. For example, you cannot use the `AWSElasticBlockStore` provisioner if your Kubernetes cluster is running in Microsoft Azure.

The Container Storage Interface (CSI)

The CSI is an important piece of the Kubernetes storage jigsaw. However, unless you're a developer writing storage plugins, you're unlikely to interact with it very often.

It's an open-source project that defines a standards-based interface so that storage can be leveraged in a uniform way across multiple container orchestrators. In other words, a storage vendor *should* be able to write a single CSI plugin that works across multiple orchestrators like Kubernetes and Docker Swarm. In reality, Kubernetes is the focus.

In the Kubernetes world, the CSI is the preferred way to write drivers (plugins) and means that plugin code no longer needs to exist in the main Kubernetes code tree. It also provides a clean and simple interface that abstracts all the complex internal Kubernetes storage machinery. Basically, the CSI exposes a clean interface and hides all the ugly volume machinery inside of the Kubernetes code (no offense intended).

From a day-to-day management perspective, your only real interaction with the CSI will be referencing the appropriate plugin in your YAML manifest files. Also, it may take a while for existing in-tree plugins to be replaced by CSI plugins.

Sometimes we call plugins *"provisioners"*, especially when we talk about Storage Classes later in the chapter.

The Kubernetes persistent volume subsystem

From a day-to-day perspective, this is where you'll spend most of your time configuring and interacting with Kubernetes storage.

You start out with raw storage on the left of Figure 8.3. This *plugs in* to Kubernetes via a CSI plugin. You then use the resources provided by the persistent volume subsystem to leverage and use the storage in your apps.

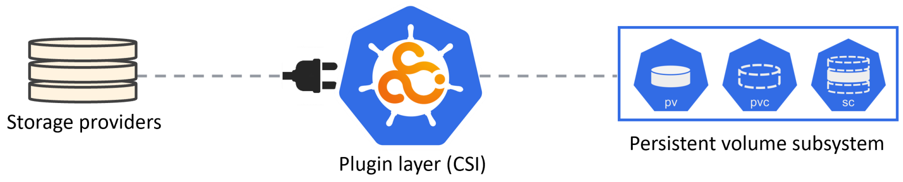


Figure 8.3

The three main resources in the persistent volume subsystem are:

- Persistent Volumes (PV)
- Persistent Volume Claims (PVC)
- Storage Classes (SC)

At a high level, **PVs** are how you represent storage in Kubernetes. **PVCs** are like tickets that grant a Pod access to a PV. **SCs** make it all dynamic.

Let's walk through a quick example.

Assume you have a Kubernetes cluster and an external storage system. The storage vendor provides a CSI plugin so that you can leverage its storage assets inside of your Kubernetes cluster. You provision 3 x 10GB volumes on the storage system and create 3 Kubernetes PV objects to make them available on your cluster. Each PV references one of the volumes on the storage array via the CSI plugin. At this point, the three volumes are visible and available for use on the Kubernetes cluster.

Now assume you're about to deploy an application that requires 10GB of storage. That's great, you already have three 10GB PVs. In order for the app to use one of them, it needs a PVC. As previously mentioned, a PVC is like a ticket that lets a Pod (application) use a PV. Once the app has the PVC, it can mount the respective PV into its Pod as a volume. Refer back to Figure 8.2 if you need a visual representation.

That was a high-level example. Let's do it.

This example is for a Kubernetes cluster running on the Google Cloud. I'm using a cloud option as they're the easiest to follow along with and you *may* be able to use the cloud's free tier/initial free credit. It's also possible to follow along on other clouds by changing a few values.

The example assumes 10GB SSD volume called "uber-disk" has been pre-created in the same Google Cloud Region or Zone as the cluster. The Kubernetes steps will be:

1. Create the PV
2. Create the PVC
3. Define the volume into a PodSpec
4. Mount it into a container

The following YAML file creates a PV object that maps back to the pre-created Google Persistent Disk called "uber-disk". The YAML file is available in the `storage` folder of the book's GitHub repo called `gke-pv.yml`.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv1
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: test
  capacity:
    storage: 10Gi
  persistentVolumeReclaimPolicy: Retain
  gcePersistentDisk:
    pdName: uber-disk
```

Let's step through the file.

PersistentVolume (PV) resources are defined in `v1` of the *core* API group. You're naming this PV "pv1", setting its access mode to `ReadWriteOnce`, and making it part of a class of storage called "test". You're defining it as a 10GB volume, setting a reclaim policy, and mapping it back to a **pre-created** GCE persistent disk called "uber-disk".

The following command will create the PV. It assumes the YAML file is in your `PATH` and is called `gke-pv.yml`. The operation will fail if you have not pre-created "uber-disk" on the back-end storage system (in this example the back-end storage is provided by Google Compute Engine).

```
$ kubectl apply -f gke-pv.yml
persistentvolume/pv1 created
```

Check the PV exists.

```
$ kubectl get pv pv1
NAME   CAPACITY   MODES   RECLAIM POLICY   STATUS      STORAGECLASS ...
pv1    10Gi       RWO     Retain           Available   test
```

If you want, you can see more detailed information with `kubectl describe pv pv1`, but at the moment you have what is shown in Figure 8.4.

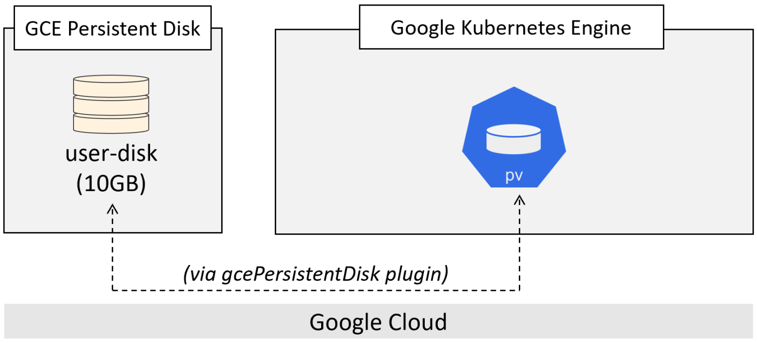


Figure 8.4

Let's quickly explain some of the PV properties set out in the YAML file.

`.spec.accessModes` defines how the PV can be mounted. Three options exist:

- `ReadWriteOnce` (RWO)
- `ReadWriteMany` (RWM)
- `ReadOnlyMany` (ROM)

`ReadWriteOnce` defines a PV that can only be mounted/bound as R/W by a single PVC. Attempts from multiple PVCs to bind (claim) it will fail.

`ReadWriteMany` defines a PV that can be bound as R/W by multiple PVCs. This mode is usually only supported by file and object storage such as NFS. Block storage usually only supports `RWO`.

`ReadOnlyMany` defines a PV that can be bound by multiple PVCs as R/O.

A couple of things are worth noting. First up, a PV can only be opened in one mode – it is not possible for a single PV to have a PVC bound to it in ROM mode and another PVC bound to it in RWM mode. Second up, Pods do not act directly on PVs, they always act on the PVC object that is bound to the PV.

`.spec.storageClassName` tells Kubernetes to group this PV in a storage class called "test". You'll learn more about storage classes later in the chapter, but you need this here to make sure the PV will correctly bind with a PVC in a later step.

Another property is `spec.persistentVolumeReclaimPolicy`. This tells Kubernetes what to do with a PV when its PVC has been released. Two policies currently exist:

- `Delete`
- `Retain`

`Delete` is the most dangerous, and is the default for PVs that are created dynamically via *storage classes* (more on these later). This policy deletes the PV **and associated storage resource on the external storage system**, so will result in data loss! You should obviously use this policy with caution.

`Retain` will keep the associated `PV` object on the cluster as well as any data stored on the associated external asset. However, it will prevent another `PVC` from using the PV in future.

If you want to re-use a *retained* PV, you need to perform the following three steps:

1. Manually delete the PV on Kubernetes
2. Re-format the associated storage asset on the external storage system to wipe any data
3. Recreate the PV

> **Tip**: If you are experimenting in a lab and re-using PVs, it's easy to forget that you will have to perform the previous three steps when trying to re-use an old deleted PV that has the *retain* policy.

`.spec.capacity` tells Kubernetes how big the PV should be. This value can be less than the actual physical storage asset but cannot be more. For example, you cannot create a 100GB PV that maps back to a 50GB device on the external storage system. But you can create a 50GB PV that maps back to a 100GB external volume (but that would be wasteful).

Finally, the last line of the YAML file links the PV to the name of the pre-created device on the back-end.

You can also specify vendor-specific attributes using the `.parameters` section of a PV YAML. You'll see more of this later when you look at *storage classes*, but for now, if your storage system supports pink fluffy NVMe devices, this is where you'd specify them.

Now that you've got a PV, let's create a PVC so that a Pod can claim access to the storage.

The following YAML defines a PVC that can be used by a Pod to gain access to the `pv1` PV you created earlier. The file is available in the `storage` folder in the book's GitHub repo called `gke-pvc.yml`.

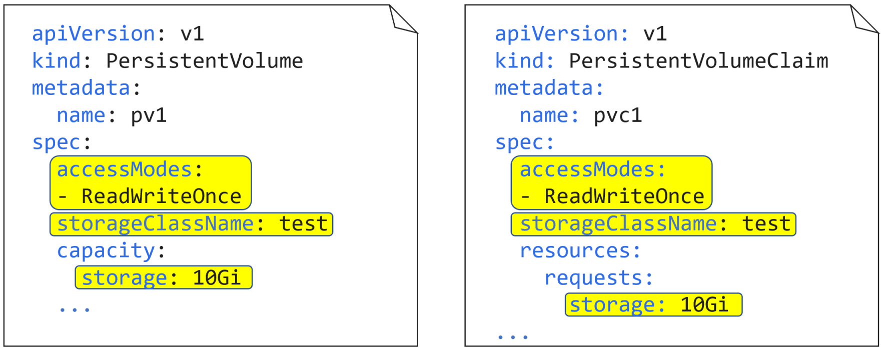
```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc1
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: test
  resources:
    requests:
      storage: 10Gi
```

As with the PV, PVCs are a stable `v1` resource in the *core* API group.

The most important thing to note about a PVC object is that the values in the `.spec` section must match with the PV you are binding it with. In this example, *access modes*, *storage class*, and *capacity* must match with the PV.

> **Note**: It's possible for a PV to have more capacity than a PVC. For example, a 10GB PVC can be bound to a 15GB PV (obviously this will waste 5GB of the PV). However, a 15GB PVC cannot be bound to a 10GB PV.

Figure 8.5 shows a side-by-side comparison of the example PV and PVC YAML files and highlights the properties that need to match.



```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv1
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: test
  capacity:
    storage: 10Gi
  ...
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc1
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: test
  resources:
    requests:
      storage: 10Gi
...
```

Figure 8.5

Deploy the PVC with the following command. It assumes the YAML file is called "gke-pvc.yml" and exists in your `PATH`.

```
$ kubectl apply -f gke-pvc.yml
persistentvolumeclaim/pvc1 created
```

Check that the PVC is created and bound to the PV.

```
$ kubectl get pvc pvc1
NAME   STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS
pvc1   Bound    pv1      10Gi       RWO            test
```

OK, you've got a PV called `pv1` representing 10GB of external storage on our Kubernetes cluster and you've bound a PVC called `pvc1` to it. Let's find out how a Pod can leverage that PVC and use the actual storage.

More often than not, you'll deploy your applications via higher-level controllers like *Deployments* and *StatefulSets*, but to keep the example simple, you'll deploy a single *Pod*. Pods deployed like this are often referred to as *"singletons"* and are not recommended for production as they do not provide high availability and cannot self-heal.

The following YAML defines a single-container Pod with a volume called "data" that leverages the PVC and PV objects you already created. The file is available in the `storage` folder of the book's GitHub repo called `volpod.yml`.

```
apiVersion: v1
kind: Pod
metadata:
  name: volpod
spec:
  volumes:
  - name: data
    persistentVolumeClaim:
      claimName: pvc1
  containers:
  - name: ubuntu-ctr
    image: ubuntu:latest
    command:
    - /bin/bash
    - "-c"
    - "sleep 60m"
    volumeMounts:
    - mountPath: /data
      name: data
```

You can see that the first reference to storage is `.spec.volumes`. This defines a volume called "data" that leverages the previously created PVC called "pvc1".

You can run the `kubectl get pv` and `kubectl get pvc` commands to show that you've already created a PVC called "pvc1" that is bound to a PV called "pv1". `kubectl describe pv pv1` will also prove that `pv1` relates to a 10GB GCE persistent disk called "uber-disk".

Deploy the Pod with the following command.

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

You can run a `kubectl describe pod volpod` command to see that the Pod is successfully using the `data` volume and the `pvc1` claim.

Time for a quick summary before we look at how you can make all of this dynamic with *storage classes*.

You start out with storage assets on an external storage system. You use a CSI plugin to integrate the external storage system with Kubernetes, and you use Persistent Volume (PV) objects to make the external systems assets accessible and usable. Each PV is an object on the Kubernetes cluster that maps back to a specific storage asset (LUN, share, blob…) on the external storage system. Finally, for a Pod to use a PV, it needs a Persistent Volume Claim (PVC). This is like a ticket that grants the Pod to the PV. Once the PV and PVC objects are created and bound, the PVC can be referenced in a PodSpec and the associated PV mounted as a volume in a container.

Don't worry if this seems complicated, we'll pull it all together in a demo at the end of the chapter.

Storage Classes and Dynamic Provisioning

Everything you've seen so far is correct and fundamental to Kubernetes storage. But it doesn't scale – there's no way somebody managing a large Kubernetes environment can manually create and maintain large numbers of PVs and PVCs. You need something more dynamic.

Enter *storage classes…*

As the name suggests, storage classes allow you to define different *classes*, or tiers, of storage. How you define your classes is up to you, but will depend on the types of storage you have access to. For example, you might define a *fast* class, a *slow* class, and an *encrypted* class (your external storage system would need to support different speeds of storage and support encrypting volumes as Kubernetes does none of this).

As far as Kubernetes goes, storage classes are defined as resources in the `storage.k8s.io/v1` API group. The resource type is `StorageClass`, and you define them in regular YAML files that you POST to the API server for deployment. You can use the `sc` shortname to refer to StorageClass objects when using `kubectl`.

> **Note**: You can see a full list of API resources, and their shortnames, using the `kubectl api-resources` command. The output of the command shows; the API group that each resource belongs to (an empty string indicates the *core* API group), if the resource is namespaced, and what its equivalent `kind` is when writing YAML files.

A StorageClass YAML

The following is a simple example of a StorageClass YAML file. It defines a class of storage called "fast", that is based on AWS solid state drives (`io1`) in the Ireland Region (`eu-west-1a`). It also requests a performance level of 10 IOPs per gigabyte.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1
  zones: eu-west-1a
  iopsPerGB: "10"
```

As with all Kubernetes YAML, `kind` tells the API server what type of object is being defined, and `apiVersion` tells it which version of the schema to apply to the resource. `metadata.name` is an arbitrary string value that lets you give the object a friendly name – this example is defining a class called "fast". `provisioner` tells Kubernetes which plugin to use, and the `parameters` field lets you finely tune the type of storage to leverage from the back-end.

A few quick things worth noting:

1. StorageClass objects are immutable – this means you cannot modify them once deployed
2. `metadata.name` should be meaningful as it's how other objects will refer to the class
3. The terms *provisioner* and *plugin* are used interchangeably
4. The `parameters` section is for plugin-specific values, and each plugin is free to support its own set of values. Configuring this section requires knowledge of the storage plugin and associated storage back-end

Multiple StorageClasses

You can configure as many StorageClass objects as you need. However, each one relates to a single provisioner. For example, if you have a Kubernetes cluster with StorageOS and Portworx storage back-ends, you will need at least two StorageClass objects. That said, each back-end can offer multiple classes/tiers of storage, each of which can have its own StorageClass. For example, you could have the following two StorageClass objects for different classes of storage **from the same back-end**:

1. “fast-secure” for high performance encrypted volumes
2. “fast” for high-performance unencrypted volumes

An example of a StorageClass defining an encrypted volume on a Portworx back-end might look like this. It will only work if you have a Portworx.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: portworx-db-secure
provisioner: kubernetes.io/portworx-volume
parameters:
  fs: "xfs"
  block_size: "32"
  repl: "2"
  snap_interval: "30"
  io-priority: "medium"
  secure: "true"
```

As you can see, the `.parameters` section is long and lists some cryptic values. Configuring this section requires knowledge of the plugin and what is supported on the storage back-end. Consult your storage plugin documentation for details.

Implementing StorageClasses

The basic workflow for deploying *and using* a StorageClass on your cluster is as follows:

1. Create your Kubernetes cluster with a storage back-end
2. Ensure the plugin for the storage back-end is available
3. Create a StorageClass object
4. Create a PVC object that references the StorageClass by name
5. Deploy a Pod that uses volume based on the PVC

Notice that the workflow does **not** include creating a PV. This is because storage classes create PVs dynamically.

The following YAML snippet contains the definitions for a StorageClass, a PersistentVolumeClaim, and a Pod. All three objects can be defined in a single YAML file by separating each object with three dashes (`---`).

Notice how the PodSpec references the PVC by name, and in turn, the PVC references the SC by name.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast  # Referenced by the PVC
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mypvc  # Referenced by the PodSpec
  namespace: mynamespace
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 50Gi
  storageClassName: fast    # Matches name of the SC
---
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: mypvc  # Matches PVC name
  containers: ...
  <SNIP>
```

The previous YAML is truncated and does not include a full PodSpec.

So far, you've seen a few SC definitions. However, each one has been slightly different as each one has related to a different provisioner (storage plugin/back-end). You will need to refer to the documentation of your storage plugin to know which options your provisioner supports.

Let's quickly summarize what you've learned about storage classes before walking through a demo.

StorageClasses make it so that you don't have to create PVs manually. You create the StorageClass object and use a plugin to tie it to a particular type of storage on a particular storage back-end. For example, high-performance AWS SSD storage in the AWS Mumbai Region. The SC needs a name, and is defined in a YAML file that you deploy using `kubectl`. Once deployed, the StorageClass watches the API server for new PVC objects that reference its name. When matching PVCs appear, the StorageClass dynamically creates the required volume on the back-end storage system as well as the PV on Kubernetes.

There's always more detail, such as *mount options* and *volume binding modes*, but what you've learned so far is enough to get you more than started.

Let's bring everything together with a demo.

Demo

In this section, you'll walk through a demo that uses a StorageClass. The basic steps of the demo will be:

1. Create a StorageClass
2. Create a PVC
3. Create a Pod that leverages it all

The Pod will map a volume using the PVC, which in turn will trigger the SC to dynamically create a PV and associated external storage asset. The demo will be on the Google Cloud Platform and assumes you have a working cluster with `kubectl` correctly configured.

Clean-up

If you've been following along, you'll have a Pod, a PVC, and PV already created. Let's delete these before proceeding with the demo.

```
$ kubectl delete pods volpod
pod "volpod" deleted

$ kubectl delete pvc pvc1
persistentvolumeclaim "pvc1" deleted

$ kubectl delete pv pv1
persistentvolume "pv1" deleted
```

Create a StorageClass

We'll use the following YAML to create a StorageClass called "slow" based on Google GCE standard persistent disks. We won't get into the details of the storage back-end, but suffice to say it's a slow tier of disk. The YAML also sets the reclaim policy so that data will not be lost when PVC bindings are released. Finally, it uses an *annotation* to attempt to set this as the default storage class on the cluster.

Here's the YAML file, it's available in the storage folder of the book's GitHub repo called `google-sc.yml`.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
reclaimPolicy: Retain
```

Two things to note before you deploy the SC:

1. This lab was tested on Kubernetes 1.16.2
2. Setting default storage classes on the tested version of Kubernetes is done via an annotation. This is likely to change in future versions.

Deploy the SC with the following command:

```
$ kubectl apply -f google-sc.yml
storageclass.storage.k8s.io/slow created
```

You can check and inspect it with `kubectl get sc slow` and `kubectl describe sc slow`. For example:

```
$ kubectl get sc slow
NAME             PROVISIONER            AGE
slow (default)   kubernetes.io/gce-pd   32s
```

Create a PVC

Use the following YAML to create a PVC object that references the `slow` StorageClass created in the previous step. The YAML is available in the storage folder of the book's GH repo called `google-pvc.yml`.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pv-ticket
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: slow
  resources:
    requests:
      storage: 25Gi
```

The important things to note are that the PVC is called `pv-ticket`, it's linked to the `slow` class, and it's for a 25GB volume.

Let's deploy it.

```
$ kubectl apply -f google-pvc.yml
persistentvolumeclaim/pv-ticket created
```

Verify the operation with a `kubectl get pvc`.

```
$ kubectl get pvc pv-ticket
NAME        STATUS   VOLUME         CAPACITY   ACCESS MODES   STORAGECLASS
pv-ticket   Bound    pvc-881a23...  25Gi       RWO            slow
```

Notice that the PVC is already bound to the `pvc-881a23...` volume – you didn't have to manually create a PV. The mechanics behind the operation are as follows:

1. You created the `slow` StorageClass
2. A loop was created to watch the API Server for new PVCs referencing the `slow` StorageClass
3. You created the `pv-ticket` PVC that requested binding to a 25GB volume from the **slow StorageClass**
4. The StorageClass loop noticed this PVC and dynamically created the requested PV

Use the following command to verify the presence of the automatically created PV on the cluster.

```
$ kubectl get pv
NAME         CAPACITY  Mode  STATUS  CLAIM       STORAGECLASS
pvc-881...   25Gi      RWO   Bound   pv-ticket   slow
```

Some of the columns have been trimmed from the output to better fit the book.

The following YAML defines a single-container Pod. The Pod template defines a volume called `data` using the `pv-ticket` PVC. It also defines a container that mounts the `data` volume to `/data`. The YAML file is in the storage folder of the book's GitHub repo called `google-pod.yml`.

```
apiVersion: v1
kind: Pod
metadata:
  name: class-pod
spec:
  volumes:
  - name: data
    persistentVolumeClaim:
      claimName: pv-ticket
  containers:
  - name: ubuntu-ctr
    image: ubuntu:latest
    command:
    - /bin/bash
    - "-c"
    - "sleep 60m"
    volumeMounts:
    - mountPath: /data
      name: data
```

Deploy the Pod with `kubectl apply -f google-pod.yml`.

Congratulations. You've deployed a new default StorageClass and used a PVC to dynamically create a PV. You also have a Pod that has mounted the PVC as a volume in a container.

Clean-up

If you've followed along with the demo, you'll have a Pod called "class-pod" with a volume using the "pv-ticket" PVC that was dynamically created via the "slow" SC. The following commands will delete all of these objects.

```
$ kubectl delete pod class-pod
pod "class-pod" deleted

$ kubectl delete pvc pv-ticket
persistentvolumeclaim "pv-ticket" deleted

$ kubectl delete sc slow
storageclass.storage.k8s.io "slow" deleted
```

Using the default StorageClass

One last thing…

If your cluster has a *default storage class*, you can deploy a Pod using just a PodSpec and a PVC. You do not need to manually create a StorageClass. However, real-world production clusters will usually have multiple StorageClasses, so it's best practice to create and manage StorageClasses that suit your business and application needs. The default StorageClass is normally only useful in development environments and times when you do not have specific storage requirements.

Chapter Summary

In this chapter, you learned that Kubernetes has a powerful storage subsystem that allows it to leverage storage from a wide variety of external storage back-ends.

Each back-end requires a plugin so that its storage assets can be used on the cluster, and the preferred type of plugin is a CSI plugin. Once a plugin is enabled, Persistent Volumes (PV) are used to represent external storage resources on the Kubernetes cluster, and Persistent Volume Claims (PVC) are used to give Pods access to PV storage.

Storage Classes take things to the next level by allowing applications to dynamically request storage. You create a Storage Class object that references a class, or tier, of storage from a storage back-end. Once created, the Storage Class watches the API Server for new Persistent Volume Claims that reference the Storage Class. When a matching PVC arrives, the SC dynamically creates the storage and makes it available as a PV that can be mounted as a volume into a Pod (container).

9: ConfigMaps

Most business applications comprise two main parts:

- The application binary
- A configuration

A simple example is a web server such as NGINX or httpd (Apache). Neither are very useful without a configuration. However, when you combine the application with a configuration it becomes extremely useful.

In the past, we coupled the application and the configuration into a single easy to deploy unit. As we moved into the early days of cloud-native microservices applications we brought this model with us. However, it's an anti-pattern in the cloud-native world. Cloud-native microservices applications should de-couple the application and the configuration, bringing benefits such as:

- Re-usable application images
- Simpler testing
- Simpler and fewer disruptive changes

We'll explain all of these, and more, as we go through the chapter.

We'll split this chapter as follows:

- The big picture
- ConfigMap theory
- Hands-on with ConfigMaps

The big picture

As already mentioned, most applications comprise two distinct parts – the application binary and a configuration. This model doesn't change with cloud-native microservices applications running on Kubernetes. However, a core principle of these types of applications is decoupling the two components – you build and store them separately, but bring them together at runtime.

Let's consider an example to understand some of the benefits…

Quick example

Imagine the following.

You work for a company that deploys modern applications to Kubernetes, and you have three distinct environments:

- Dev

- Test
- Prod

Your developers write and update applications. Initial testing is performed in the *dev* environment, further testing is done in the *test* environment where more stringent rules and the likes are applied. Finally, stable components graduate to the *prod* environment.

Each environment has subtle differences, such as; number of nodes, configuration of nodes, network and security policies, different sets of credentials and certificates, and more.

You currently package each application microservice with its configuration baked into the container (the application and configuration are packaged as a single artefact). With this in mind, you have to perform all of the following for every business application:

- *build* three distinct images (one for dev, one for test, one for prod)
- *store* the images in three distinct repositories (dev, test, prod)
- *run* each version of the image in a specific environment (dev in dev, test in test, prod in prod)

Every time you make a change to an application configuration, you need to create an entire new image and perform some type of rolling update to the entire app – even if the change is something as simple as fixing a typo or changing the size or colour of a font ;-)

Analysing the example

There are several drawbacks to the approach of storing the application and its configuration as a single artefact (container image).

As your *dev*, *test*, and *prod* environments have different characteristics, each environment needs its own image. A *prod* image will not work in the *dev* or *test* environments because of the differences. This requires extra work to create and maintain 3x copies each application. This can complicate matters and increase the chances of misconfiguration.

You also have to store 3x images in 3 distinct repositories. Plus, you need to be very careful about permissions to repositories. This is because your *prod* images will contain sensitive configuration data, sensitive passwords, and sensitive encryption keys. You probably don't want dev and test engineers to have access to prod images – access to the images means access to the sensitive data stored in them.

Also, it's harder to troubleshoot an issue if you push an update that includes both an application binary update as well as a configuration update. If the two are tightly coupled, it's harder to isolate the fault. Also, if you need to make a minor configuration change (for example fix a prominent typo on a web page) you need to re-package, re-test, and re-deploy the entire application binary **and** configuration.

None of this is ideal.

What it looks like in a de-coupled world

Now consider you work for the same company but you do things differently. This time, your application and its configuration are de-coupled. This time;

- you build a single image that is shared across all three environments

- you store a single image in a single repository
- You run a single version of each image in all environments

To make this work, you build your application images as generically as possible with no embedded configuration. You then create and store configurations in separate objects and apply a configuration to the application at when you run it. For example, you have a single copy of a web server that you can deploy to all three environments. When you deploy it to *prod* you apply the *prod* configuration to it. When you run it in *dev*, you apply the *dev* configuration to it…

In this model, you create and test a single version of each application image that you store in a single repository. All staff can have access to the image repository as there is no sensitive data stored in the images. Finally, you can easily push changes to the application and its configuration independent of each other – updating a simple typo no longer requires the entire application binary and image to be rebuilt and re-deployed.

Let's see how Kubernetes makes this possible…

ConfigMap theory

Kubernetes provides an object called a ConfigMap (CM) that lets you store configuration data outside of a Pod. It also lets you dynamically inject the data into a Pod at run-time.

> **Note**: When we use the term *Pod* we mean the Pod and all of its containers. After all, it is ultimately the container that receives the configuration data.

ConfigMaps are first-class objects in the Kubernetes API under the *core* API group, and they're `v1`. This tells us a lot of things:

1. They're stable (`v1`)
2. They've been around for a while (the fact that they're in the core API group)
3. You can operate on them with the usual `kubectl` commands
4. They can be defined and deployed via the usual YAML manifests

ConfigMaps are typically used to store non-sensitive configuration data such as:

- Environment variable values
- Entire configuration files (things like web server configs and database configs)
- Hostnames
- Service ports
- Accounts names
- more…

You should **not** use ConfigMaps to store sensitive data such as certificates and passwords. Kubernetes provides a different object, called a *Secret*, for storing sensitive data. Secrets and ConfigMaps are very similar in design and implementation, the major difference is that Kubernetes takes steps obscure the values stored in Secrets. It makes no such efforts to obscure data stored in ConfigMaps.

How do ConfigMaps work

At a high-level, a ConfigMap is a place to store configuration data that can be seamlessly injected into containers at runtime, then leveraged in ways that are invisible to applications.

Let's look a bit closer…

Behind the scenes, ConfigMaps are a map of key/value pairs and we call each key/value pair an **entry**.

- **Keys** are an arbitrary name that can be created from alphanumerics, dashes, dots, and underscores
- **Values** can contain anything, including carriage returns
- We separate keys and values with a colon – key:value

Some simple examples might be:

- db-port:13306
- hostname:msb-prd-db1

More complex examples can store entire configuration files like this one:

`key:` conf `value:`

```
directive in;
main block;
http {
  server {
    listen        80 default_server;
    server_name   *.msb.com;
    root          /var/www/msb.com;
    index         index.html

    location / {
      root   /usr/share/nginx/html;
      index  index.html;
    }
  }
}
```

Once data is stored in a ConfigMap, it can be injected into containers at run-time via any of the following methods:

- environment variables
- arguments to the container's startup command
- files in a volume

All of the methods work seamlessly with existing applications. In fact, all an application sees is its configuration data in either; an environment variable, an argument to a startup command, or a file in a filesystem. The application is unaware that the data originally came from a ConfigMap.

Figure 9.1 shows how the pieces connect.

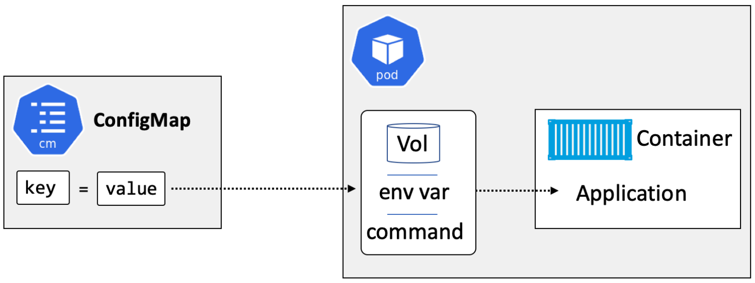


Figure 9.1

The most flexible of the three methods is the *volume* option, and the most limited is the *startup command*. We'll look at each in turn, but before we do that we'll quickly consider a *Kubernetes-native* application.

ConfigMaps and Kubernetes-native apps

A Kubernetes-native application is an application that knows it's running on Kubernetes and has the intelligence to query the Kubernetes API. As a result, a Kubernetes-native application can access ConfigMap data directly via the API without needing things like environment variables and volumes. This can simplify application configuration, but the application will only run on Kubernetes. At the time of writing, Kubernetes-native applications are rare.

Hands-on with ConfigMaps

As with most Kubernetes objects, you can create them imperatively and declaratively. We'll look at the imperative method first.

Creating ConfigMaps imperatively

The command to imperatively create a ConfigMap is `kubectl create configmap`, but you can shorten `configmap` to `cm`. The command accepts two sources of data:

- literal values on the command line (`--from-literal`)
- files referenced on the command line (`--from-file`)

Run the following command to create a ConfigMap called `testmap1`, populated with two map entries from literal values passed on the command line.

```
$ kubectl create configmap testmap1 \
  --from-literal shortname=msb.com \
  --from-literal longname=magicsandbox.com
```

The following describe command shows how the two entries are stored in the map.

```
$ kubectl describe cm testmap1
Name:         testmap1
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
shortname:
----
msb.com
longname:
----
magicsandbox.com
Events:  <none>
```

You can see that the object is essentially a map of key/value pairs dressed up as a Kubernetes object. The two map entries are exactly what you would expect from the inputs to the command: - Entry 1: shortname=msb.com - Entry 2: longname=magicsandbox.com

The next command will create a ConfigMap from a file called `cmfile.txt`. The command assumes you have a local file called `cmfile.txt` in your working directory. The file contains the following single line of text, and you can clone a copy from the book's GitHub repo under the configmaps directory.

```
Magic Sandbox, hands-on learning that blurs the lines between training and the real world.
```

Run this command to create the ConfigMap from the contents of the file. Notice that the command uses the `--from-file` argument instead of `--from-literal`.

```
$ kubectl create cm testmap2 --from-file cmfile.txt
configmap/testmap2 created
```

The following describe command is interesting as it shows the following:

- A single map entry was created
- The name of the entry's key is the name of the file (`cmfile.txt`)
- The entry's value is the contents of the file

```
$ kubectl describe cm testmap2
Name:         testmap2
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
cmfile.txt:
----
Magic Sandbox, hands-on learning that blurs the lines between training and the real world.
Events:  <none>
```

Inspecting ConfigMaps

ConfigMaps are first class API objects. This means you can inspect and query them in the same way as any other API object. You've already seen `kubectl describe` commands, but other `kubectl` commands also work. `kubectl get` can list all ConfigMaps, and the usual `-o yaml` and `-o json` flags pull the full configuration from the cluster store.

Run a `kubectl get` to list all ConfigMaps in your current Namespace.

```
$ kubectl get cm
AME       DATA   AGE
testmap1   2      11m
testmap2   1      2m23s
```

The following `kubectl get` with the `-o yaml` flag shows the entire configuration of the object and hints at something interesting.

```
$ kubectl get cm testmap1 -o yaml
apiVersion: v1
data:
  longname: magic-sandbox
  shortname: msb
kind: ConfigMap
metadata:
  creationTimestamp: "2019-10-27T11:42:23Z"
  name: testmap1
  namespace: default
  resourceVersion: "39223"
  selfLink: /api/v1/namespaces/default/configmaps/testmap1
  uid: 0b2f5daa-5905-419c-a1bc-0289e32fdead
```

The interesting thing to note is that ConfigMap objects don't have the concept of state (desired state and actual state). This is why they have a `data` block instead of `spec` and `status` blocks.

Let's find out how to create a ConfigMap declaratively before we look at how to inject data from one into a Pod.

Creating ConfigMaps declaratively

The following ConfigMap manifest defines two map entries; `firstname` and `lastname`. It is available in the book's GitHub repo under the configmap folder called `multimap.yml`. Alternatively, you can create an empty file and practice writing your own manifests from scratch.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: multimap
data:
  given: Nigel
  family: Poulton
```

You can see that a ConfigMap manifest has the normal `kind` and `apiVersion` fields, as well as the usual `metadata` section. However, as previously mentioned, they do not have a `spec` section. Instead, they have a `data` section that defines the map of key/values.

You can deploy it with the following command (the command assumes you have a copy of the file in your working directory called `multimap.yml`).

```
$ kubectl apply -f multimap.yml
configmap/multimap created
```

This next YAML looks slightly more complicated but it's actually not – it creates a ConfigMap with just a single map entry in the `data` block. It looks more complicated because the *value* portion of the map entry is a full configuration file.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: test-conf
data:
  test.conf: |
    env = plex-test
    endpoint = 0.0.0.0:31001
    char = utf8
    vault = PLEX/test
    log-size = 512M
```

The previous YAML file inserts a pipe character (|) after the name of the entry's *key* property. This tells Kubernetes that everything following the pipe is to be treated as a single literal value. Therefore, the ConfigMap object is called `test-config` and it contains a single map entry as follows:

- Key: `test.conf`
- Value: `env = plex-test endpoint = 0.0.0.0:31001 char = utf8 vault = PLEX/test log-size = 512M`

You can deploy the previous CM with the following `kubectl command`. The command assumes you have a local copy of the file called `singlemap.yml`.

```
$ kubectl apply -f singlemap.yml
configmap/test-conf created
```

List and describe the `multimap` and `test-conf` ConfigMaps you just created. The following shows the output of a `kubectl describe` against the `test-conf` map.

```
$ kubectl describe cm test-conf
Name:         test-conf
Namespace:    default
Labels:       <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration:
                {"apiVersion":"v1","data":{"test.config":"env =
                plex-test\nendpoint = 0.0.0.0:31001\nchar = utf8
                \nvault = PLEX/test\nlog-size = 512M\n"},"...
Data
====
test.config:
----
env = plex-test
endpoint = 0.0.0.0:31001
char = utf8
vault = PLEX/test
log-size = 512M

Events:  <none>
```

ConfigMaps are extremely flexible and can be used to insert complex configuration files such as JSON files and even scripts into containers at run-time.

Injecting ConfigMap data into Pods and containers

You've seen how to imperatively and declaratively create ConfigMap objects and populate them with data. Now let's see how to get that data into applications running in containers.

There are three main ways to inject ConfigMap data into a container:

- As environment variables
- As arguments to container startup commands
- As files in a volume

Let's look at each.

ConfigMaps and environment variables

A common way to get ConfigMap data into a container is via environment variables. You create the ConfigMap, then you map its entries into environment variables in the container section of a Pod template. When the container is started, the environment variables appear in the container as standard Linux or Windows environment variables.

Figure 9.2. shows this.

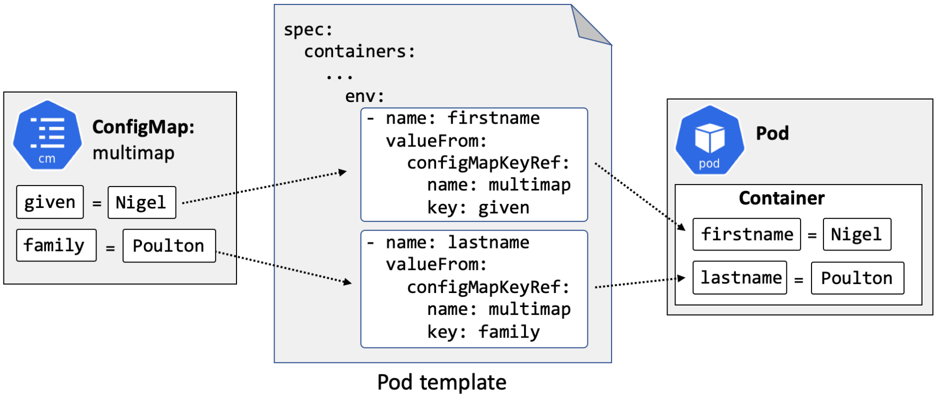


Figure 9.2

You already have a ConfigMap called `multimap` that has two values:

- given=Nigel
- family=Poulton

The following Pod manifest deploys a single container that creates two environment variables in the container.

- FIRSTNAME: Maps to the `given` entry in the `multimap` ConfigMap
- LASTNAME: Maps to the `family` entry in the `multimap` ConfigMap

When the Pod is scheduled and the container started, `FIRSTNAME` and `LASTNAME` will be created as standard Linux environment variables inside the container. These can then be used by applications running in the container.

There's a manifest called `envpod.yml` in the configmaps folder of the book's GitHub repo. The following commands will deploy the Pod from the `envpod.yml` file and then list environment variables that include the `name` string in their name – this will list the `firstname` and `lastname` variables. You'll see that they are populated with the values from the `multimap` ConfigMap.

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

$ kubectl exec envpod -- env | grep NAME
HOSTNAME=envpod
FIRSTNAME=Nigel
LASTNAME=Poulton
```

A drawback to using ConfigMaps with environment variables is that environment variables are static. This means that any updates you make to the values in the ConfigMap will not be reflected in running containers. For example, if you update the `given` and `family` values in the ConfigMap, environment variables in existing containers will not get the updates.

ConfigMaps and container startup commands

The concept of using ConfigMaps with container startup commands is simple. The high-level looks like this. It's possible to specify a startup command for a container, and you can customize that startup command with variables. Let's look at a simple example…

The following Pod template (the part of a YAML manifest that defines a Pod and its containers) defines a single container called `args1`. The container is based on the `busybox` image and runs the `/bin/sh` command outlined on line 5.

```
spec:
  containers:
    - name: args1
      image: busybox
      command: [ "/bin/sh", "-c", "echo First name $(FIRSTNAME) last name $(LASTNAME)" ]
      env:
        - name: FIRSTNAME
          valueFrom:
            configMapKeyRef:
              name: multimap
              key: given
        - name: LASTNAME
          valueFrom:
            configMapKeyRef:
              name: multimap
              key: family
```

If you look closely at the startup command you'll see that it references two variables; `FIRSTNAME` and `LASTNAME`. Each of these is defined in the `env:` section directly below the startup command.

- `FIRSTNAME` is based on the `given` entry in the `multimap` ConfigMap
- `LASTNAME` is based on the `family` entry in the same ConfigMap

The relationship is shown in in Figure 9.3.

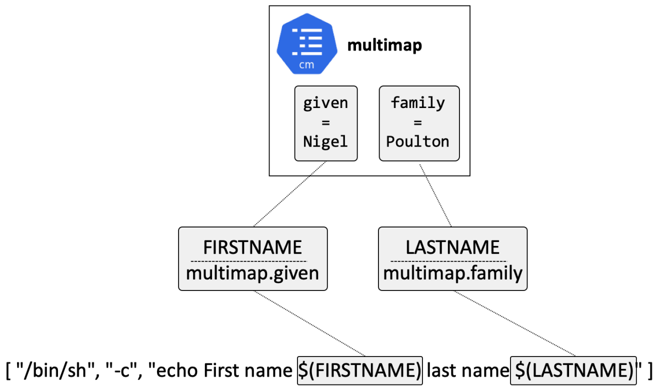


Figure 9.3

Running a Pod based on the previous YAML will print "First name Nigel last name Poulton" to the container's log file. You can see the logs of the container with command `$ kubectl logs <pod-name> -c args1`

Describing the Pod will yield the following lines describing the environment of the Pod.

```
Environment:
  FIRSTNAME:  <set to the key 'given' of config map 'multimap'>
  LASTNAME:  <set to the key 'family' of config map 'multimap'>
```

Using ConfigMaps with container startup commands suffers from the same limitations as using them with environment variables – updates to entries in the map will not be reflected in running containers.

ConfigMaps and volumes

Using ConfigMaps with volumes is the most flexible option. You can reference entire configuration files as well as make updates to the ConfigMap and have them reflected in running containers. This means you can make changes to entries in a ConfigMap after you've deployed a container, and those changes be seen in the container and available for running applications.

The high-level process for exposing ConfigMap data via a volume looks like this.

1. Create the ConfigMap
2. Create a *ConfigMap volume* in the Pod template
3. Mount the *ConfigMap volume* into the container
4. Entries in the ConfigMap will appear in the container as individual files

This process is shown in Figure 9.4

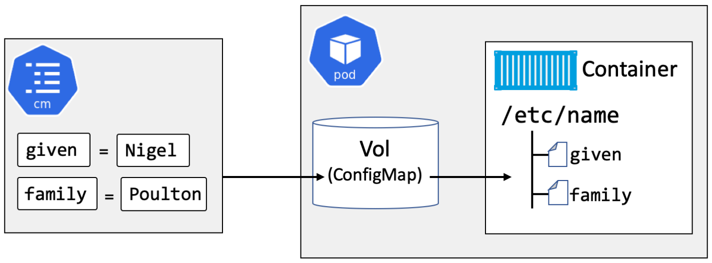


Figure 9.4

You still have the `multimap` ConfigMap with two values.

- given=Nigel
- family=Poulton

The following YAML creates 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`

```
apiVersion: v1
kind: Pod
metadata:
  name: cmvol
spec:
  volumes:
    - name: volmap
      configMap:
        name: multimap
  containers:
    - name: ctr
      image: nginx
      volumeMounts:
        - name: volmap
          mountPath: /etc/name
```

Let’s step through things in a little more detail…

The `spec.volumes` block creates a special type of volume called a *ConfigMap volume*. The volume is called **volmap** and based on the `multimap` ConfigMap. This means that the volume will be populated with the entries stored in the `data` block of the ConfigMap. In this example, the volume will have two files; `given` and `family`. The `given` file will have the contents `Nigel`, and the `family` file will have the contents `Poulton`.

The `spec.containers` block mounts the **volmap** volume into the container at `/etc/name`. This means that two files will appear in the container as:

- `/etc/name/given`
- `/etc/name/family`

The following commands deploy the container (from the `cmvol.yml` manifest) and then run a `kubectl exec` command to list the files in the ‘/etc/name/ directory.

```
$ kubectl apply -f cmpod.yml
pod/cmvol created

$ kubectl exec volpod -- ls /etc/name
family
given
```

Chapter Summary

ConfigMaps are the mechanism that Kubernetes provides for decoupling applications and their configuration.

ConfigMaps are first-class object in the Kubernetes API and can be created and manipulated with the usual `kubectl create`, `kubectl get`, and `kubectl describe` commands. They’re ideal for storing application configuration parameters as well as entire configuration files, but they shouldn’t be used to store sensitive data.

ConfigMap data gets injected into containers at run-time, and you can inject data via environment variables, container startup commands, and volumes. The volumes method is the most flexible as it allows you work with entire configuration files. It also allows updates to eventually be reflected already-running containers.

10: 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 an application that creates and saves valuable data. An example might be an app that saves data about client sessions and uses it for future client sessions. Other examples include databases and other data stores.

We'll divide the chapter as follows:

- The theory of StatefulSets
- Hands-on with StatefulSets

The theory section will introduce you to the way StatefulSets work and what they bring to the table. But don't worry if you don't understand everything at first, we'll cover most of it again when we walk through the hands-on section.

The theory of StatefulSets

It's often useful to compare StatefulSets with Deployments. Both are first-class objects in the Kubernetes API and follow the typical Kubernetes controller architecture. These controllers run as reconciliation loops that watch the state of the cluster, via the API Server, and are constantly moving the *observed state* of the cluster into sync with *desired state*. Deployments and StatefulSets also support self-healing, scaling, updates, and more.

However, there are some vital differences. StatefulSets guarantee:

- predictable and persistent Pod names
- predictable and persistent DNS hostnames
- predictable and persistent volume bindings

These three properties form the *state* of a Pod, sometimes referred to as the Pods *sticky ID*. This state/sticky ID is persisted across failures, scaling, and other scheduling operations, making StatefulSets ideal for applications where Pods are a little bit unique and not interchangeable.

As a quick example, failed Pods managed by a StatefulSet will be replaced by new Pods with the exact same Pod name, the exact same DNS hostname, and the exact same volumes. This is true even if the replacement Pod is started on a different cluster Node. The same is not true of Pods managed by a Deployment.

The following YAML snippet shows *some* of the fields in a typical StatefulSet manifest.

```
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
        ...
```

The name of the StatefulSet is `tkb-sts` and it defines three Pod replicas running the `mongo:latest` image. You post this to the API Server, it's persisted to the cluster store, the work is assigned to cluster Nodes, and the StatefulSet controller monitors the shared state of the cluster and makes sure *observed state* matches *desired state*.

That's the big picture. Let's take a look at some of the major characteristics of StatefulSets before walking through an example.

StatefulSet Pod naming

All Pods managed by a StatefulSet get *predictable* and *persistent* names. These names are vital, and are at the core of how Pods are started, self-healed, scaled, deleted, attached to volumes, and more.

The format of StatefulSet Pod names is `<StatefulSetName>-<Integer>`. The integer is a *zero-based index ordinal*, which is just a fancy way of saying "number starting from 0". The first Pod created by a StatefulSet always gets index ordinal "0", and each subsequent Pod gets the next highest ordinal. Assuming the previous YAML snippet, the first Pod created will be called `tkb-sts-0`, the second will be called `tkb-sts-1`, and the third will be called `tkb-sts-2`.

Be aware that StatefulSet names need to be a valid DNS names, so no exotic characters! You'll see why later.

Ordered creation and deletion

Another fundamental characteristic of StatefulSets is the controlled and ordered way they start and stop Pods.

StatefulSets create one Pod at a time, and always wait for previous Pods to be *running and ready* before creating the next. This is different from Deployments that use a ReplicaSet controller to start all Pods at the same time, causing potential race conditions.

As per the previous YAML snippet, `tkb-sts-0` will be started first and must be *running* and *ready* before the StatefulSet controller will start `tkb-sts-1`. The same applies to subsequent Pods – `tkb-sts-1` needs to be *running* and *ready* before `tkb-sts-2` starts etc. See Figure 10.1

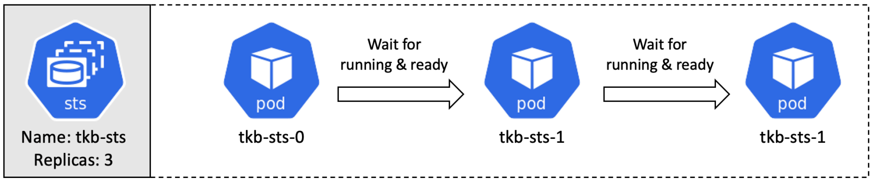


Figure 10.1

> **Note**: *Running* and *ready* are technical terms used to indicate all containers in a Pod are executing and the Pod is ready to service requests.

Scaling operations are also governed by the same ordered startup rules. For example, scaling from 3 to 5 replicas will start a new Pod called `tkb-sts-3` and wait for it the 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 (number) first, waits for it to fully terminate before terminating the Pod with the next highest ordinal.

Knowing the order in which Pods will be scaled down, as well as knowing that Pods will not be terminated in parallel, is a game-changer for many stateful apps. For example, clustered apps that store data are usually at high risk of losing data if multiple replicas go down at the same time. StatefulSets guarantee this will never happen, and you can insert other delays via things like `terminationGracePeriodSeconds` to further control the scaling down process. All in all, StatefulSets bring a lot to the table for clustered apps that store data.

Finally, it's worth noting that StatefulSet controllers do their own self-healing and scaling. This is architecturally different to Deployments which use a separate ReplicaSet controller for these operations.

Deleting StatefulSets

There are two major things to consider when deleting StatefulSets.

Firstly, deleting a StatefulSet does not terminate Pods in order. With this in mind, you may want to scale a StatefulSet to 0 replicas before deleting it.

You can also use `terminationGracePeriodSeconds` to further control the way Pods are terminated. It's common to set this to at least 10 seconds to give applications running in Pods a chance to flush local buffers and safely commit any writes that are still in-flight.

Volumes

Volumes are an important part of a StatefulSet Pods *sticky ID* (state).

When a StatefulSet Pod is created, any volumes it needs are created at the same time and named in a way to connect them to the right Pod . Figure 10.2 shows a StatefulSet called "ss" requesting 3 replicas. You can see how each Pod and volume (PVC) is created and how the names connect volumes to Pods.

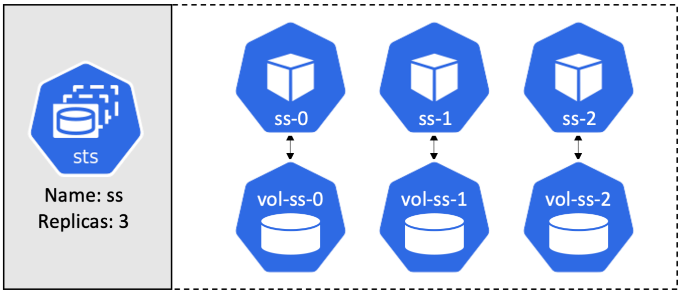


Figure 10.2

Volumes are appropriately decoupled from Pods via the normal Kubernetes persistent volume subsystem constructs (PersistentVolumes and PersistentVolumeClaims). This means volumes have separate lifecycles to Pods and allows volumes to survive Pod failures and termination operations. For example, any time a StatefulSet Pod fails or is terminated, the associated volumes are unaffected. This allows replacement Pods to attach to the same storage as the Pods they're replacing. This is true, even if the replacement Pod is scheduled to a different cluster Node.

The same is true for scaling operations. If a StatefulSet Pod is deleted as part of a scale-down operation, subsequent scale-up operations will attach new Pods to the existing volumes that match their names.

This behavior can be a life-saver 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 attempts to keep observed state in sync with desired state. The simplest example is a Pod failure. If you have a StatefulSet called `tkb-sts` with 5 replicas, and `tkb-sts-3` fails, the controller will start a replacement Pod with the same name and attach it to the same volumes.

However, if a failed Pod recovers after Kubernetes has replaced it, you'll have two identical Pods on the network writing to the same volumes. This can result in data corruption. With this in mind, the StatefulSet controller is extremely careful how it handles failures.

Potential Node failures are especially difficult to deal with. For example, if Kubernetes loses contact with a Node, how does it know if the Node is down and will never recover, or if it's a temporary glitch such as a network partition, a crashed Kubelet, or the Node is simply rebooting? To complicate matters further, the controller can't even force the Pod to terminate, as the Kubelet may never receive the instruction. With these things in mind, manual intervention is needed before Kubernetes will replace Pods on failed Nodes.

Network ID and headless Services

We've already said that StatefulSets are for applications that need Pods to be predictable and longer-lived. As a result, other parts of the application as well as other applications may need to connect directly to individual Pods. To make this possible, StatefulSets use a headless Service to create predictable DNS hostnames for every Pod replica. Other apps can then query DNS for the full list of Pod replicas and use these details to connect directly to Pods.

The following YAML snippet shows a headless Service called "mongo-prod" that is listed in the StatefulSet YAML as the *governing Service*.

```
apiVersion: v1
kind: Service
metadata:
  name: mongo-prod
spec:
  clusterIP: None
  selector:
    app: mongo
    env: prod
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sts-mongo
spec:
  serviceName: mongo-prod
```

Let's explain the terms *headless Service* and *governing Service*.

A headless Service is just a regular Kubernetes Service object with `spec.clusterIP` set to `None`. It becomes a StatefulSets governing Service when you list it in the StatefulSet manifest under `spec.serviceName`.

When the two objects are combined like this, the Service will create DNS SRV records for each Pod replica matching the label selector of the headless Service. Other Pods can then find members of the StatefulSet by performing DNS lookups against the name of the headless Service. You'll see this in action later, and obviously applications will need to know to do this.

That covers most of the theory, let's walk through an example and see how it all comes together.

Hands-on with StatefulSets

In this section of the chapter, you'll deploy a working StatefulSet. The example is intended to demonstrate the way StatefulSets work and reinforce what you've already learned. It's not intended as a production-grade application configuration.

The examples cited will work on Kubernetes clusters running on GCP and GKE. The course's GitHub repo contains YAML files for other cloud platforms.

All of the YAML files we'll refer to are in the `StatefulSets` folder of the book's GitHub repo. You can clone the repo with the following command.

```
$ git clone https://github.com/nigelpoulton/TheK8sBook.git
```

If you're following along, you'll deploy the following three objects: 1. A StorageClass 2. A headless Service 3. A StatefulSet

To make things easier to follow, you'll inspect and deploy each object individually. However, all three objects can be grouped in a single YAML file separated by three dashes (see `app.yml` in the StatefulSets folder of the repo).

Deploying the StorageClass

StatefulSets that use volumes need to be able to create them dynamically. You need two objects to do this:

- StorageClass (SC)
- PersistentVolumeClaim (PVC)

The following YAML is from the `/StatefulSets/gcp-sc.yml` file and defines a StorageClass object called `flash` that will dynamically provision SSD volumes (`type=pd-ssd`) from GCP using the GCP persistent disk CSI driver (`pd.csi.storage.gke.io`). It will only work on Kubernetes clusters running on GCP or GKE with the CSI driver enabled. YAML files for creating StorageClasses on other cloud platforms, including AWS, Azure, and Linode are also provided in the repo.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: flash
provisioner: pd.csi.storage.gke.io
volumeBindingMode: WaitForFirstConsumer
allowVolumeExpansion: true
parameters:
  type: pd-ssd
```

Deploy the Storage class, being sure to use the appropriate file for your lab environment.

```
$ kubectl apply -f gcp-sc.yml
storageclass.storage.k8s.io/flash created
```

List your cluster's StorageClasses to make sure it was created correctly.

```
$ kubectl get sc
NAME   PROVISIONER            RECLAIMPOLICY  VOLUMEBINDINGMODE      ALLOWEXPANSION   AGE
flash  pd.csi.storage.gke.io  Delete         WaitForFirstConsumer   true             5s
```

With the StorageClass in place, PersistentVolumeClaims (PVC) can use it to dynamically create new volumes. We'll circle back to this later in the example.

Creating a governing headless Service

When learning about headless Services, it can be useful to visualize a Service object with a head and a tail. The head is the stable IP exposed on the network, and the tail is the list of Pods it will send traffic to. Therefore, a headless Service is a Service object without a ClusterIP.

The following YAML is from the `StatefulSets/headless-svc.yml` file and describes a headless Service called `dullahan` with no IP address (`spec.clusterIP: None`).

```
apiVersion: v1
kind: Service
metadata:
  name: dullahan
  labels:
    app: web
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: web
```

The only difference to a regular Service is that a headless Service must set the value `clusterIP` to `None`.

When combined with a StatefulSet, headless Services create predictable stable DNS entries for every Pod that matches the StatefulSets label selector. You'll see this in a later step.

Deploy the headless Service to your cluster.

```
$ kubectl apply -f headless-svc.yml
service/tkb-sts created
```

Verify the operation.

```
$ kubectl get svc
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.0.0.1     <none>        443/TCP   145m
dullahan     ClusterIP   None         <none>        80/TCP    10s
```

Deploy the StatefulSet

With the StorageClass and headless Service in place, it's time to deploy the StatefulSet.

The following YAML is from the `StatefulSets/sts.yml` file and defines the StatefulSet. Remember this is for learning purposes only, it's not intended as a production-grade deployment of an application.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: tkb-sts
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  serviceName: "dullahan"
  template:
    metadata:
      labels:
        app: web
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: ctr-web
        image: nginx:latest
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: webroot
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: webroot
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "flash"
      resources:
        requests:
          storage: 1Gi
```

There's a lot to take in, so let's step through the important parts.

The name of the StatefulSet is `tkb-sts`. This is important as it forms part of the name of every Pod the StatefulSet will create – every Pod created by this StatefulSet will have a name starting with `tkb-sts`.

The `spec.replicas` field defines 3 Pod replicas. These will be named `tsk-sts-0`, `tsk-sts-1`, and `tsk-sts-2`. They'll be created in numerical order, and the StatefulSet controller will wait for each replica 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 created in the previous step, and will create the DNS SRV records for each StatefulSet replica. It's called the *governing Service* because it's in charge of the DNS subdomain used by the StatefulSet.

The remainder of the `spec.template` section defines the Pod template that will be used to stamp out Pod replicas – things such as which container image to use and which ports to expose.

Last, but most certainly not least, is the `spec.volumeClaimTemplates` section.

Earlier in the chapter we said every StatefulSet that uses volumes needs to be able to create them dynamically. To do this you need a StorageClass and a PersistentVolumeClaim (PVC).

You've already created the StorageClass, so you're ready to go with that aspect. However, PVCs present an interesting challenge.

Each StatefulSet Pod needs its own unique storage. This means each one needs its own PVC. However, this isn't possible, as each Pod is created from the same template. Also, you'd have to pre-create a unique PVC for every potential StatefulSet Pod, which also isn't possible when you consider StatefulSets can be scaled up and down.

Clearly, a more intelligent StatefulSet-aware approach is needed. This is where *volume claim templates* come into play.

At a high-level, a volumeClaimTemplate dynamically creates a PVC each time a new Pod replica is dynamically created. It also contains the intelligence to name the PVC so it can be correctly attached to Pods. This way, the StatefulSet manifest contains a Pod template section for stamping out Pod replicas, and a volume claim template section for stamping out PVCs.

The following YAML snippet shows the volumeClaimTemplate from the example StatefulSet application. It defines a claim template called `webroot` requesting a 10GB volume from the `flash` StorageClass created earlier.

```
volumeClaimTemplates:
- metadata:
    name: webroot
  spec:
    accessModes: [ "ReadWriteOnce" ]
    storageClassName: "flash"
    resources:
      requests:
        storage: 10Gi
```

When the StatefulSet object is deployed, it will create three Pod replicas and three PVCs.

Deploy the StatefulSet and watch the Pods and PVCs get created.

```
$ kubectl apply -f sts.yml
statefulset.apps/tkb-sts created
```

Watch the StatefulSet ramp up to 3 running replicas. It'll take a minute or so for the 3 Pods and associated PVCs to be created – each Pod needs to be running and ready before the next one is started.

```
$ kubectl get sts --watch
NAME      READY   AGE
tkb-sts   0/3     10s
tkb-sts   1/3     23s
tkb-sts   2/3     46s
tkb-sts   3/3     69s
```

Notice how it took ∼23 seconds to start the first replica. Once that was running and ready, it took another 23 seconds to start the second, and then another 23 for the third.

Now check the PVCs.

```
$ kubectl get pvc
NAME               STATUS   VOLUME             CAPACITY   MODES   STORAGECLASS   AGE
webroot-tkb-sts-0  Bound    pvc-1146...f274    10Gi       RWO     flash          86s
webroot-tkb-sts-1  Bound    pvc-3026...6bcb    10Gi       RWO     flash          63s
webroot-tkb-sts-2  Bound    pvc-2ce7...e56d    10Gi       RWO     flash          40s
```

There are 3 new PVCs, each created at the same time as one of the StatefulSet Pod replicas. See how the name of each PVC is based on the name of the StatefulSet and the Pod it's associated with. The format of the PVC name is the volumeClaimTemplate name, followed by a dash, followed by the name of the Pod replica it's associated with.

```
Pod Name     |     PVC Name
tkb-sts-0   <->    webroot-tkb-sts-0
tkb-sts-0   <->    webroot-tkb-sts-1
tkb-sts-0   <->    webroot-tkb-sts-2
```

At this point, the StatefulSet is up and running and the application replicas are executing.

Testing peer discovery

You know that pairing a headless Service with a StatefulSet creates DNS SRV records for each StatefulSet Pod that matches the Service's label selector. You already have a headless Service and 3 StatefulSet Pods running, so you should have three DNS SRV records – one for each Pod.

However, before testing this, it's worth taking a moment to understand how DNS hostnames and DNS subdomains work with StatefulSets.

By default, Kubernetes places all objects within the `cluster.local` DNS subdomain. You can choose something different, but most lab environments use this domain, so we'll assume it in this example. Within that domain, Kubernetes constructs DNS subdomains as follows:

`<object-name>.<service-name>.<namespace>.svc.cluster.local`

- `svc` indicates the subdomain for objects behind a Service.

So far, you've got three Pods called `tkb-sts-0`, `tkb-sts-1`, and `tkb-sts-2` governed by the `dullahan` headless Service. This means the 3 Pods will have the following fully qualified DNS names:

- `tkb-sts-0.dullahan.default.svc.cluster.local`
- `tkb-sts-1.dullahan.default.svc.cluster.local`
- `tkb-sts-2.dullahan.default.svc.cluster.local`

To test this, you'll deploy a jump-pod that has the DNS `dig` utility pre-installed. You'll `exec` onto that Pod, and you'll use `dig` to query DNS for SRV records in the `dullahan.default.svc.cluster.local` subdomain.

Deploy the jump-pod from the `/StatefulSets/jump-pod.yml` file in the course's GitHub repo.

```
$ 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 is now connected to the jump-pod. Run the following dig command from the jump-pod.

```
$ dig SRV dullahan.default.svc.cluster.local
<Snip>
;; ADDITIONAL SECTION:
tkb-sts-0.dullahan.default.svc.cluster.local. 30 IN A 10.24.1.25
tkb-sts-2.dullahan.default.svc.cluster.local. 30 IN A 10.24.1.26
tkb-sts-1.dullahan.default.svc.cluster.local. 30 IN A 10.24.0.17
<Snip>
```

The query returns the fully qualified DNS names of each Pod, as well as its IP. Other applications, including the app itself, can use this method to discover the full list of Pods in the StatefulSet.

For this method of discovery to be useful, applications obviously need to know how to use it. They must also know the name of the StatefulSet's governing Service, and StatefulSet Pods must match the governing Service's label selector.

Scaling StatefulSets

Each time a StatefulSet is scaled up, a Pod **and** a PVC is created. However, when scaling a StatefulSet down, the Pod is terminated but the PVC is not. This means future scale-up operations only need to create a new Pod, which is then connected to the surviving PVC. The StatefulSet controller includes all of the intelligence to track and manage these mappings between Pods and PVCs.

You currently have 3 StatefulSet Pod replicas and 3 PVCs. Edit the `sts.yml` file and change the replica count from 3 down to 2 and save your change. When you've done that, run the following command to re-post the YAML file to the cluster.

```
$ kubectl apply -f sts.yml
statefulset.apps/tkb-sts configured
```

Check the state of the StatefulSet and verify the Pod count has reduced to 2.

```
$ kubectl get sts tkb-sts
NAME      READY   AGE
tkb-sts   2/2     14m

$ kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
tkb-sts-0   1/1     Running   0          15m
tkb-sts-1   1/1     Running   0          15m
```

The number of Pod replicas has been successfully scaled down to 2, and the Pod with the highest index ordinal was deleted. However, you will still have 3 PVCs – remember, scaling down and deleting Pod replicas does **not** delete the associated PVC. Verify this.

```
$ kubectl get pvc
NAME                STATUS   VOLUME             CAPACITY   MODES   STORAGECLASS   AGE
webroot-tkb-sts-0   Bound    pvc-1146...f274   10Gi       RWO     flash          15m
webroot-tkb-sts-1   Bound    pvc-3026...6bcb   10Gi       RWO     flash          15m
webroot-tkb-sts-2   Bound    pvc-2ce7...e56d   10Gi       RWO     flash          15m
```

The fact that all three PVCs still exist means that scaling back up to three replicas only requires a new Pod to be created. As the name of the surviving PVC is `webroot-tkb-sts-2`, the StatefulSet controller knows to automatically connect it to the new Pod.

Edit the `sts.yml` file and increment the number of replicas back to 3 and save your change. When you've done that, re-post the YAML file to the API Server with the following command.

```
$ 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     20m
```

You have 3 Pods again. Describe the new `webroot-tkb-sts-2` PVC to verify it's mounted by the correct Pod.

```
$ kubectl describe pvc webroot-tkb-sts-2 | grep Mounted
Mounted By:    tkb-sts-2
```

It's worth noting that scale down operations will be put on hold if any of the Pods are in a failed state. This is to protect the resiliency of the app and integrity of its data.

Finally, it's possible to tweak the controlled and ordered starting and stopping of Pods via the StatefulSet's `spec.podManagementPolicy` property.

The default setting is `OrderedReady` and implements the strict methodical ordering previously explained. Setting 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 create all three new Pods instantaneously, and scaling down from 5 > 2 will delete three Pods in parallel. StatefulSet naming rules are still implemented, and the setting only applies to scaling operations and does not impact rolling updates.

Performing rolling updates

StatefulSets support rolling updates. You update the image version in the YAML and re-post it to the API Server. Once authenticated and authorized, the controller will replace old Pods with new ones. However, the process always starts with the highest numbered Pod and works down through the set, one-at-a-time, until all Pods are on the new version. The controller also waits for each new Pod to be running and ready before replacing the one with the next lowest index ordinal.

For more information, run `$ kubectl explain sts.spec.updateStrategy`.

Test a Pod failure

A simple way to test a failure is to manually delete a Pod. This will delete the Pod but not the associated PVC. The StatefulSet controller will notice observed state vary from desired state, realise that a Pod has been deleted, and start a new identical Pod in its place. This new Pod will have the same name and will be connected to the same PVC.

Let's test it.

Confirm that you have three healthy Pods in your StatefulSet.

```
$ kubectl get pods
NAME        READY   STATUS   AGE
tkb-sts-0   1/1     Running  37m
tkb-sts-1   1/1     Running  37m
tkb-sts-2   1/1     Running  18m
```

You're about to delete the `tkb-sts-0` Pod. But before you do that, let's use `$ kubectl describe` to confirm the PVC it's currently using. You don't need to do this, as you can deduce the name of the PVC from the name of the volumeClaimTemplate and the StatefulSet. However, it's good to confirm.

```
$ kubectl describe pod tkb-sts-0
Name:         tkb-sts-0
Namespace:    default
<Snip>
Status:       Running
IP:           10.24.1.13
<Snip>
Volumes:
  webroot:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim...)
    ClaimName:  webroot-tkb-sts-0
<Snip>
```

Based on the output (your lab will be different) the values are as follows:

- Name: tkb-sts-0
- PVC: webroot-tkb-sts-0

Let's delete the `tkb-sts-0` Pod and see if the StatefulSet controller recreates it.

```
$ kubectl delete pod tkb-sts-0
pod "tkb-sts-0" deleted

$ kubectl get pods --watch
NAME        READY   STATUS              RESTARTS   AGE
tkb-sts-1   1/1     Running             0          43m
tkb-sts-2   1/1     Running             0          24m
tkb-sts-0   0/1     Terminating         0          43m
tkb-sts-0   0/1     Pending             0          0s
tkb-sts-0   0/1     ContainerCreating   0          0s
tkb-sts-0   1/1     Running             0          34s
```

Placing a `--watch` on the command shows the StatefulSet controller noticing the terminated Pod and creating a replacement – desired state is 3 replicas but observed state dropped to 2. As the failure is clean and easy to verify, the controller immediately kicked off the process to create a new Pod.

You can see the new Pod has the same name as the failed one, but does it have the same PVC?

The following command confirms it does.

```
$ kubectl describe pod tkb-sts-0 | grep ClaimName
    ClaimName:  webroot-tkb-sts-0
```

Recovering from potential Node failures is a lot more complex and requires manual intervention. This is because failed Nodes are notoriously hard to diagnose and confirm, and there's always a risk the failure could be transient. If the StatefulSet controller assumes a Node has failed and replaces any StatefulSet Pods, but the Node subsequently recovers, there's a chance of duplicate Pods on the network contending for the same storage. This can cause all kinds of bad things to happen, including corrupting data.

Deleting StatefulSets

Earlier in the chapter, you learned that deleting a StatefulSet doesn't terminate managed Pods in order. Therefore, if your application needs Pods shutting down in order, you should scale the StatefulSet to 0 replicas before initiating the delete operation.

Scale your StatefulSet to 0 replicas and confirm the operation. It may take a few seconds for the set 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     86m
```

Once the StatefulSet has no replicas you can delete it.

```
$ kubectl delete sts tkb-sts
statefulset.apps "tkb-sts" deleted
```

You can also delete the StatefulSet by referencing its YAML file with `$ kubectl delete -f sts.yml`.

Feel free to exec onto the jump-pod and run another `dig` to prove the SRV records have been removed from DNS.

At this point, the StatefulSet object is deleted, but the headless Service, StorageClass, and jump-pod still exist. You may want to delete them as well.

Chapter Summary

In this chapter, you learned how StatefulSets create and manage applications that need to persist state.

They can self-heal, scale up and down, and perform updates. Rollbacks require manual attention.

Each Pod replica spawned by a StatefulSet gets a predictable and persistent name, DNS hostname, and unique set of 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 storage volumes.

However, StatefulSets are only a framework. Applications need to be written in ways to take advantage of the way StatefulSets behave.

11: Threat modeling Kubernetes

Security is more important than ever before, and Kubernetes is no exception. Fortunately, there are a lot of things that can be done to secure Kubernetes, and we'll cover some of them in the next chapter. However, before we do this, it's worth taking a moment to model some of the common threats.

Threat model

Threat modeling is the process of identifying vulnerabilities so that we can put measures in place to prevent and mitigate them. In this chapter, we'll look at the popular **STRIDE** model and see how it can be applied to Kubernetes.

STRIDE defines six categories of potential threat:

- Spoofing
- Tampering
- Repudiation
- Information disclosure
- Denial of service
- Elevation of privilege

While the model is good, it's important to keep in mind that no threat model guarantees to cover all possible threats. However, models like this are useful at providing a structured way to look at an entire system.

For the rest of this chapter, we'll look at each of the six threat categories in turn. For each one, we'll give a quick description, and then look at some of the ways it applies to Kubernetes and how we can prevent and mitigate.

This chapter doesn't attempt to cover everything. It's intended to give you ideas and get you started.

Spoofing

Spoofing is pretending to be something, or somebody, you are not. In the context of information security, it's pretending to be a different user or entity, with the aim of gaining extra privileges on a system.

Let's look at how Kubernetes authenticates users to prevent spoofing.

Securing communications with the API server

Kubernetes is comprised of lots of small components that work together. These include control plane components such as the API server, controller manager, scheduler, cluster store, and others. It also includes node components such as the kubelet and container runtime. Each of these has its own set of privileges that allow it to interact with, and even modify the cluster. Even though Kubernetes implements a least-privilege model, spoofing the identity of any of these components can have unforeseen and potentially disastrous consequences.

Fortunately, Kubernetes implements a security model that requires components to authenticate via mutual TLS (mTLS). This requires both parties (the sender and the receiver) to authenticate each other via cryptographically signed certificates. This is good, and Kubernetes makes things easy by auto-rotating certificates etc. However, it's vital that you consider the following:

1. A typical Kubernetes installation will auto-generate a self-signed certificate authority (CA) during the bootstrap process. This is the CA that will issue certificates to all cluster components. It's better than nothing, but on its own it probably isn't enough for your production environment.
2. Mutual TLS is only as secure as the CA that issued the certificates. Compromising the CA can render the entire mTLS layer ineffective. So, 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 making sure the Kubernetes CA is not added as a trusted CA for any components outside of Kubernetes.

As mentioned in previous chapters, all interaction with Kubernetes is via the API server and subject to authentication and authorization checks. This is true for internal and external components. 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 components and the other for authenticating external components. To accomplish this, Kubernetes leverages an internal self-signed CA to issue keys to internal components, as well as one or more trusted 3rd-party CAs to issue keys to external components (Kubernetes obviously needs configuring to trust the 3rd-party CAs). This configuration ensures the API server trusts internal components possessing a certificate issued by the cluster's self-signed CA, as well as external components possessing a certificate signed by the 3rd-party CA.

Securing Pod communications

As well as spoofing access to the *cluster*, there is also the threat of spoofing an application for app-to-app communications. This is when one Pod spoofs another. Fortunately, we can leverage Kubernetes *Secrets* to mount certificates into Pods that can then be used to authenticate Pod identity.

While on the topic of Pods, every Pod has an associated `ServiceAccount` that is used to provide an identity for the Pod within the cluster. 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, it is recommended to set `automountServiceAccountToken` to false for Pods that do not 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
  <Snip>
```

If the service account token needs to be mounted, there are some non-default configurations that are worth exploring. These include:

- expirationSeconds
- audience

These let you force a time when the token will expire, as well as restrict it to only working with subsets of entities. The following example, inspired from official Kubernetes docs, sets an expiry period of one hour and restricts it to use with 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
          audience: vault
```

Tampering

Tampering is the act of changing something in a malicious way. In relation to information security, the goal of tampering is usually 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 counter-measure is to make it obvious when something has been tampered with. A common example, outside of information security, is drug packaging. Most over-the-counter drugs are packaged with tamper-proof seals. These make it obvious to the consumer if the product has been tampered with because the tamper-proof seal has been broken.

Let's first look at some of the cluster components that can be tampered with.

Tampering with Kubernetes components

All of the following Kubernetes components, if tampered with, can cause harm:

- 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 – You'll be warned if the data has been tampered with.

The following recommendations can also help prevent tampering with data when it is *at rest* in a Kubernetes cluster:

- 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 safely guard your SSH keys).
- Always perform SHA-2 checksums on downloaded binaries.
- Restrict access to your image registry and associated repositories.

This isn't an exhaustive list, but if you implement it, you will greatly reduce the chances of having your data 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 the change the binary's file attributes.

```
$ auditctl -w /var/lib/docker -p rwxa -k audit-docker
```

We'll refer to this example later in the chapter.

Tampering with applications running on Kubernetes

As well as infrastructure components, application components are also potential tampering targets.

A good way to prevent a live Pod from being tampered with, is setting its filesystems to read-only. This guarantees filesystem immutability and can be accomplished through a Pod Security Policy or the securityContext section of a Pod's manifest file.

> **Note:** PodSecurityPolicy objects are a relatively new feature that allows you to force security settings on all Pods in a cluster, or targeted sub-sets of Pods. They're a good way to enforce standards without developers and operations staff having to remember to do it for every individual Pod.

You can make a container's root filesystem read-only by setting the `readOnlyRootFilesystem` property to `true`. As previously mentioned, this can be set via a PodSecurityPolicy object, or in Pod manifest files. The same can be done for other filesystems that are mounted into containers via the `allowedHostPaths` property.

The following example shows how to use both settings in a Pod manifest. 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
    allowedHostPaths:
      - pathPrefix: "/test"
        readOnly: true
```

The same can be implemented in a PodSecurityPolicy object as follows:

```
apiVersion: policy/v1beta1  # Will change in future versions
kind: PodSecurityPolicy
metadata:
  name: tampering-example
spec:
  readOnlyRootFilesystem: true
  allowedHostPaths:
  - pathPrefix: "/test"
    readOnly: true
```

Repudiation

At a very high level, *repudiation* is casting doubt on something. *Non-repudiation* is providing proof about something. In the context of information security, non-repudiation is **proving** certain actions were carried out by certain individuals.

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 usually requires the correlation of several events over a period of time.

Fortunately, auditing of Kubernetes API server events can usually help answer these questions. The following is an example of an API server audit event (you may need to manually enable auditing on your API server).

```
{
  "kind":"Event",
  "apiVersion":"audit.k8s.io/v1",
  "metadata":{ "creationTimestamp":"2019-03-03T10:10:00Z" },
  "level":"Metadata",
  "timestamp":"2019-03-03T10:10:00Z",
  "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":"2010-03-03T10:10:00.123456Z",
  "stageTimestamp":"2019-03-03T10:10:00.123456Z"
}
```

Although the API server is central to most things in Kubernetes, it's not the only component that requires auditing for non-repudiation. At a minimum, you should also collect audit logs from container runtimes, kubelets, and the applications running on your cluster. This is without even mentioning network firewalls and the likes.

Once you start auditing multiple components, you quickly need a centralised 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…) and ships them to a secure central location.

If you do this, it's vital that the centralised log store is secure. If the security of the central log store is compromised, you can no longer 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 masters and nodes. For example, earlier in the chapter we showed an example that enabled 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=12345678abc\
 a1=0 a2=abc12345678 a3=a items=1 ppid=1234 pid=1234 auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 s\
gid=0 fsgid=0 tty=pts0 ses=1 comm="docker" exe="/var/lib/docker" subj=system_u:object_r:container_runt\
ime_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="/var/lib/docker" inode=123456 dev=fd:00 mode=0\
100600 ouid=0 ogid=0 rdev=00:00 obj=system_u:object_r:container_runtime_exec_t:s0
```

Audit logs like this, when combined and correlated with Kubernetes' audit features, create a comprehensive and trustworthy picture that cannot be repudiated.

Information Disclosure

Information disclosure is when sensitive data is leaked. There are lots of ways it can happen, from leaving an insecure USB drive on a plane, all the way to data stores being hacked and APIs that unintentionally expose sensitive data.

Protecting cluster data

In the Kubernetes world, the entire configuration of the cluster is stored in the cluster store (currently `etcd`). This includes network and storage configuration, as well as passwords and other sensitive data stored in Secrets. For obvious reasons, 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 will be seen in the next paragraph, gaining access to a cluster node can allow the logged-on user to bypass some of the security layers.

Kubernetes 1.7 introduced encryption of Secrets but doesn't enable it by default. Even when this becomes default, the data encryption key (DEK) is stored on the same node as the Secret! This means that gaining access to a node allows you to bypass encryption. This is especially worrying on nodes that host the cluster store (etcd nodes).

Fortunately, Kubernetes 1.11 enabled a beta feature that lets you store *key encryption keys (KEK)* outside of the Kubernetes cluster. These types of key are used to encrypt and decrypt data encryption keys and should be safely guarded. You should seriously consider Hardware Security Modules (HSM) or cloud-based Key Management Stores (KMS) for storing your key encryption keys.

Keep an eye on upcoming versions of Kubernetes for further improvements to encryption of Secrets.

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 a far better solution 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, keys for decrypting them should also be stored in Secrets.

With all of this, it's vital that you 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 all about making something unavailable. There are many types of DoS attack, 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 to overload the API server so that cluster operations grind to a halt (even essential system services have to communicate via the API server).

Let's take a look ot some potential Kubernetes systems that might be targets of DoS attacks, and some ways to protect and mitigate.

Protecting cluster resources against DoS attacks

It's a time-honored best practice to replicate essential control plane services on multiple nodes for high availability (HA). Kubernetes is no different, and you should run multiple master nodes in an HA configuration for your production environments. Doing this will prevent a single master from becoming a single point of failure. In relation to certain types of DoS attacks, an attacker would potentially need to attack more than one master to have a meaningful impact.

You should also consider replicating 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 allows the scheduler to spread your application over multiple nodes and availability zones. Not only might this allow the scheduler to run your application on a different node if the one it's currently running on is subject to a DoS attack. It also means that replicated parts of your application can be distributed over multiple nodes and zones, potentially rendering a DoS attack on any single node or zone ineffective (or less effective).

You should also configure appropriate limits for the following:

- Memory
- CPU
- Storage
- Kubernetes objects

Limiting Kubernetes objects includes things like; limiting the number of ReplicaSets, Pods, Services, Secrets, and ConfigMaps in a particular Namespace.

Placing limits on things can help prevent important system resources from being starved, therefore preventing potential DoS.

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
spec:
  hard:
    pods: "100"
```

Use the following command to apply it to the `skippy` namespace. The command assumes the manifest file is called `quota.yml`.

```
 $ kubectl apply -f quota.yml --namespace=skippy
```

One more feature – *podPidsLimit* – restricts the number of processes a Pod can create.

Assume a scenario where a Pod is the target of a fork bomb attack. This is a specialised attack where a rogue process creates as many new processes as possible in an attempt to consume all resources on a system and grind it to a halt. Placing a limit on the number of processes a Pod can create will prevent the Pod from exhausting the resources of the node and confine the impact of the attack to the Pod. Once the *podPidsLimit* is exhausted, a Pod will typically be restarted.

This also ensure a single Pod doesn't exhaust the PID range for all the other Pods on the node, including the Kubelet. One thing to note, is that setting the correct value requires a good estimate of how many Pods will run simultaneously on each node. Without a ballpark estimate, the admin may end up over or under allocating PIDs to each pod.

Protecting the API Server against DoS attacks

All communication in Kubernetes goes through the API server. The API server exposes a RESTful interface over a TCP socket, making it susceptible to botnet-based DoS attacks.

The following may be helpful in either preventing or mitigating such attacks.

- Highly available masters. Having multiple API server replicas running on multiple nodes across multiple availability zones.
- Monitoring and alerting of API server requests (based on sane thresholds)
- Not exposing the API server to the internet (firewall rules etc.)

As well as botnet DoS attacks, an attacker may also attempt to spoof a user or other control plane service in an attempt to cause an overload. Fortunately, Kubernetes has robust authentication and authorization controls in place to help prevent spoofing. However, even with a robust RBAC model, it is vital that you safeguard access to accounts with high privileges.

Protecting the cluster store against DoS attacks

Cluster configuration is stored in etcd, making it vital that etcd be available and secure. The following recommendations will 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 will install etcd on the same servers as the rest of the control plane. This is usually 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 probably 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 that 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 defence in depth, you should also implement application-layer authorization policies that implement least privilege.

Figure 10.1 shows how all of these can be combined to make it hard for an attacker to successfully DoS an application.

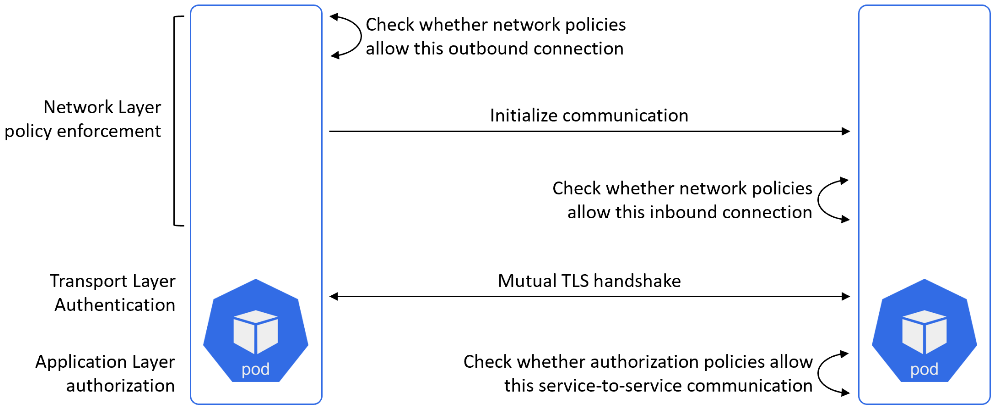

Check whether network policies allow this outbound connection
Network Layer policy enforcement
Initialize communication
Check whether network policies allow this inbound connection
Transport Layer Authentication
Mutual TLS handshake
Application Layer authorization
Check whether authorization policies allow this service-to-service communication
pod
pod

Figure 11.1

Elevation of privilege

Elevation of privilege, a.k.a privilege escalation, is gaining higher access than what is granted, usually in order 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, a common best practice is to always have *RBAC* and *node* enabled.

RBAC mode lets you restrict API operations to sub-sets of users. These *users* can be regular user accounts as well as system services. The idea is that all requests to the API server must be authenticated **and** authorized. Authentication ensures that requests are coming from a validated user – the user performing the request is who they claim to be. Authorization ensures the validated user is allowed to perform the requested operation on the targeted cluster resource. For example, can *Lily create Pods*? In this example, *Lily* is the user, *create* is the operation, and *Pods* is the resource. Authentication makes sure that it really is Lily 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 not be able 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 (cluster nodes). The types of requests made to the API server by nodes is obviously different to those generally made by regular users, and the node authorizer is designed to help with this.

RBAC and node are two recommended authorization modes. RBAC mode is extremely configurable, and you should use it to implement a least privilege model for users accessing the API server. When implemented, it is a deny-by-default system that requires you to specifically grant individual permissions. If implemented well, it does an excellent job of ensuring users and Service Accounts do not have more access than required.

Protecting Pods

The next few sections will look at a few of the 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 we proceed through the following sections, it's important to remember that a Pod is just an execution environment for one or more containers – application code runs in containers, which in turn, run inside of Pods. 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). Therefore, 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 that the `root` user of container often has unrestricted `root` access on the node as well. If that doesn't make you afraid, it should!

Fortunately, Kubernetes lets you 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 processes in all containers 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
```

`runAsUser` is one of many settings that can be configured as part of what we refer to as a `PodSecurityContext` (.spec.securityContext).

It is 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 or container. However, there is a high chance that this will cause problems if they are different containers. For example, two different containers with R/W access to the same host directory can cause data corruption (both writing to the same dataset without co-ordinating write operations). Shared security contexts also increase the possibility of a compromised container tampering with a dataset it should not 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 setting
```

This example sets the UID to `1000` at the Pod level but overrides it at the container level so that processes in one particular 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:

- Enabling *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 of the container. For example, UID `0` (the root user) in the container gets mapped to UID `1000` on the host. This can be a good solution for processes that *need* to run as root inside the container, but you should check whether it has full support from your version of Kubernetes and your container runtime.

Maintaining a map of UID usage is a clunky way to prevent multiple different Pods and containers using overlapping UIDs. It's a bit of a hack and requires strict adherence to a gated release process for releasing Pods into production.

> **Note**: A strict gated release process is a good thing for production environments. The *hacky* part of the previous section is the UID map itself, as well as the fact that you're introducing an external dependency and complicating releases and troubleshooting.

Drop capabilities

While *user namespaces* allow container processes to run as root inside the container but not on the host machine, it remains a fact that most processes do not need all of the privileges that the root has. However, it is equally true that many processes do require more privileges than a typical non-root user has. 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 that 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* like this allows you to be extremely granular when granting permissions. Instead of an all or nothing (root or non-root) approach, we can grant a process the exact set of privileges it requires to run.

There are currently over 30 capabilities and choosing the right ones can be daunting. With this in mind, an out-of-the-box Docker runtime drops over half of them by default. This is a *sensible-default* that is designed to allow most processes to run, without *leaving the keys in the front door*. While sensible defaults like these are better than nothing, they will usually not be enough for a lot of 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 will cause 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 your 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 fix and change to your application requires the exact 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 the default set of *capabilities* implemented by your chosen container runtime (E.g. containerd or Docker). However, you can override this in a Pod Security Policy, or 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 is similar in concept to *capabilities* but works on syscalls rather than capabilities.

The way that Linux processes ask the kernel to perform an operation is by issuing a syscall to the kernel. *seccomp* lets you configure which syscalls a particular container can make to the host kernel. As with capabilities, a least privilege model is preferred where the only syscalls a container is allowed to make are the ones it needs to in order to run.

Be careful though, Linux has over 300 syscalls, and at the time of writing *seccomp* is an alpha feature in Kubernetes. You should also check support from your container runtime.

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 on to the new process. We're obviously over-simplifying, but the original process is called the *parent* process, and the copy is called the *child*.

By default, Linux allows a *child* process to claim more privileges than its *parent*. This is usually a bad idea. In fact, you will often want a child process to have the same, or less 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 a Pod Security Policy or the securityContext property of an individual container.

The following Pod manifest shows how to prevent privilege escalation for an individual container.

```
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  containers:
  - name: demo
    image: example.io/simple:1.0
    securityContext:
      allowPrivilegeEscalation: false
```

Pod Security Policies

As we've seen throughout the chapter, we can enable security settings on a per-Pod basis by setting security context attributes in individual Pod YAML files. However, this approach doesn't scale, requires developers and operators to remember to do this for every Pod, and is prone to errors. *Pod Security Policies* offer a better way.

Pod Security Policies are a relatively new feature that allow you to define security settings at the cluster level. We can then apply these to targeted sets of Pods as part of the deployment process. As such, this solution scales better, requires less work from developers and admins, and is less prone to error. It also lends itself to situations where you have a team dedicated to securing apps in production.

Pod Security Policies are implemented as an *admission controller,* and in order to use them, a Pod's serviceAccount must be authorized to use it. Once this is done, their policies are applied to new requests to create Pods as they pass through the API admission chain.

Pod Security Policy example

Let's finish the chapter with a quick look at an example of a Pod Security Policy that covers many of the points discussed in this chapter, as well as some other known secure defaults.

The example is based on an example from the official Kubernetes docs[1]:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: 'docker/default'
    apparmor.security.beta.kubernetes.io/allowedProfileNames: 'runtime/default'
    seccomp.security.alpha.kubernetes.io/defaultProfileName:  'docker/default'
    apparmor.security.beta.kubernetes.io/defaultProfileName:  'runtime/default'
spec:
  privileged: false
  allowPrivilegeEscalation: false  # Prevent privilege escalation
  requiredDropCapabilities:
    - ALL # Drops all root capabilities (non-privileged user)
  # Allow core volume types.
  volumes:
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'downwardAPI'
    # Assume that PVs set up by the cluster admin are safe to use.
    - 'persistentVolumeClaim'
  hostNetwork: false # Prevent access to the host network namespace
  hostIPC: false # Prevent access to the host IPC namespce
  hostPID: false # Prevent access to the host PID namespace
  runAsUser:
    rule: 'MustRunAsNonRoot' # Prevent from running as root
  runAsGroup:
    rule: 'MustRunAs' # controls which primary Group ID containers are run with
    ranges:
      - min: 1
        max: 65535
  seLinux:
    rule: 'RunAsAny' # Any SELinux options can be used
  supplementalGroups:
    rule: 'MustRunAs' # Allow all except root (UID 0)
    ranges:
      - min: 1
        max: 65535
  fsGroup:
    rule: 'MustRunAs' # Sets range for groups that own Pod volumes
    ranges:
      - min: 1
        max: 65535
  readOnlyRootFilesystem: true # Force root filesystem to be R/O
  forbiddenSysctls:
  - '*' #Forbids any sysctls from being accessible from a pod
```

[1]https://kubernetes.io/docs/concepts/policy/pod-security-policy/#example-policies

There's no denying that configuring effective security policies is both important and challenging. A common practice is to start with a restrictive policy like the one just shown, then tweak it to fit your requirements. A lot of experimenting will be required.

It may also be a good idea to configure several Pod Security Policies that vary in how restrictive they are, then allow development teams to work with cluster administrators to choose the one that best fits the application.

Towards more secure Kubernetes

In 2019, the CNCF (Cloud Native Computing Foundation) commissioned a third-party security audit of Kubernetes. There were several findings, including threat modeling, manual code reviews, dynamic penetration testing, and cryptography review. All the findings were given a difficulty and severity level. This audit was thorough and conducted in a responsible manner ensuring all high severity findings were fixed prior to the release of the audit findings.

However, there are still many findings that need help from the community to resolve. If you feel you can helpreview the findings and get involved: http://tiny.cc/xwz3jz

The audit report is also an excellent way to learn more about Kubernetes and how the internals work. Studying reports like these is a great ay to level-up after reading this chapter.

Chapter summary

In this chapter, we used STRIDE to threat model Kubernetes. We stepped through the six categories of threat and looked at some ways to prevent and mitigate them.

We saw that one threat can often lead to another, and that there are multiple ways to mitigate a single threat. As always, defence in depth is a key tactic.

We finished the chapter by discussing how Pod Security Policies provide a flexible and scalable way to implement Pod security defaults.

In the next chapter, we'll see some best practices and lessons learned from running Kubernetes in production.

12: Real-world Kubernetes security

In the previous chapter, we threat modeled Kubernetes using STRIDE. In this chapter, we'll cover some common security-related challenges that you're likely to encounter when implementing Kubernetes in the real world.

While we accept that every Kubernetes deployment is different, there are many similarities. As a result, the examples that we cover will affect most Kubernetes deployments, large and small.

Now then, we won't be offering *cookbook style* solutions. Instead, we'll be looking at things from a high-level view, similar to what a *security architect* has.

We'll divide the chapter into the following four sections:

- CI/CD pipeline
- Infrastructure and networking
- Identity and access management
- Security monitoring and auditing

CI/CD pipeline

Containers are a revolutionary application *packaging* and *runtime* technology.

On the packaging front, we conveniently bundle application code and dependencies into an *image*. As well as code and dependencies, the image contains the commands required to run the application. This has allowed containers to hugely simplify the process of building, shipping, and running applications. It has also overcome the infamous "*it worked on my laptop*" issue.

However, containers also make running dangerous code easier than ever before.

With this in mind, let's look at some ways we can secure the flow of application code from a developer's laptop to production servers.

Image Repositories

We store images in registries, and registries are either public or private.

> **Note**: Each registry is divided into one or more repositories, and we actually store images in repositories.

Public registries are on the internet and are the easiest way to download images and run containers. However, it's important to understand that they host a mixture of *official images* and *community images*. Official images are usually provided by product vendors and have undergone a vetting process to ensure certain levels of quality. Typically, official images will; implement best practices, be scanned for known vulnerabilities, contain up-to-date code, and be supported by the product vendor. *Community images* are none of that. Yes, there are some excellent community images, but you should practice extreme caution when using them.

With all of this in mind, it's important that you implement a standard way for developers to obtain and consume images in your environments. It's also vital that any such process be as frictionless as possible for developers – if there's too much friction, your developers will look for ways to bypass them.

Let's discuss a few things that might help.

Use approved base images

Images are made up of multiple layers that build on top of each other to form a useful image. But all images start with a base layer.

Figure 12.1 shows a simple example of an image comprising three layers. The base layer contains the core OS and filesystem components that applications need in order to run. The middle layer contains the application library dependencies. The top layer contains the code that your developers have written. We call the combination of these layers an *image*, and it contains everything needed to run the application.

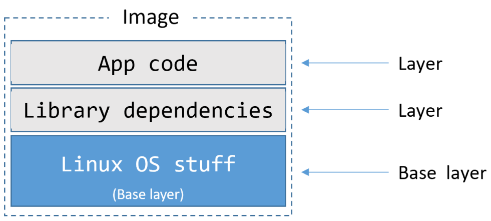


Figure 12.1

As all images have a base layer containing the required operating system (OS) and filesystem constructs for applications to build on, it's a common practice for organizations to have a small number of *approved base images*. It's also common, but not essential, for these base images to be derived from *official images*. For example, if you develop your applications on CentOS Linux, your base images *may* be based on the official CentOS image – you take the official CentOS base image and tweak it for your requirements.

In this model, all of your applications will build on top of a common approved base image like shown in Figure 12.2.

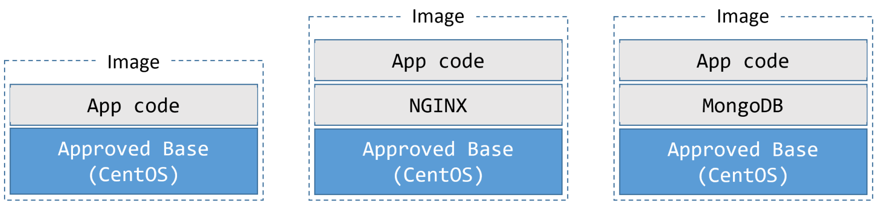


Figure 12.2

While there is some up-front effort required to create and implement base images, the long-term security benefits are worth it.

From a developer perspective, they can focus their entire efforts on the application and its dependencies without having to worry about maintaining OS components – don't worry about patching, drivers, audit settings, and more.

From an operations perspective, base images reduce software sprawl. This makes testing easier, as you will always be testing on a known base image. It makes pushing updates easier, you only need to update a small number of approved base images and have these easily rolled out to all developers. It also makes troubleshooting easier, as you have a small number of well-known base images providing your building blocks. It may also reduce the number of base image configurations that need tying into support contracts.

Non-standard base images

As good as it is to have a small number of approved base images, there may still be occasions when an application needs something different. This means you will need processes in place that:

- 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 is worth the effort)
- Determine the support implications of bringing an entirely new image into the environment

Generally speaking, updating an existing base image – such as adding a device driver for GPU computing – should be preferred over introducing an entirely new image.

Control access to images

Various options exist that allow you to protect your organization's container images. The most secure practical option is to host your own private registry within your own firewall. This allows your organization to manage how the registry is deployed, how it is replicated, and how it is patched. It also integrates permissions with existing identity management providers, such as Active Directory, and allows you to create repositories that fit your organizational structure.

If you do not have the means for a dedicated private registry, you can host your images in private repositories on public registries such as Docker Hub. However, this is not as secure as hosting your own private registry within your own firewalled network.

Whichever solution you choose, you should only host images that are approved to be used within your organization. Normally, these will be from a *trusted* source and vetted by your information security team. You should place access controls on the repositories that store these images, 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/nodes can `push` to repositories

Expanding on the list above…

If you are using a public registry, you will probably need to grant your worker nodes access to the internet so they can `pull` images. In this situation, a best practice is to limit internet access to the addresses and ports of any registries you use. You should also implement strong RBAC rules so that you can maintain control over who is pushing and pulling images from which repositories. For example, developers should probably be able to `push` and `pull` from non-production repositories, but not production. Whereas operations teams should probably be able to `pull` from non-production, as well as `push` and `pull` to production repos.

Finally, you may only want a sub-set of nodes (*build nodes*) to be able to `push` images. You may even wish to lock things down so that only your automated build systems can push to certain repositories.

Moving images from non-production to production

Many organizations have separate environments for development, testing, and production.

Generally speaking, development environments have less rules and are commonly used as places where developers can experiment. Such experimenting often involves using non-standard images that your developers eventually want to use in production.

The following sub-sections will outline some measures you can take to ensure only safe images get approved into production.

Vulnerability scanning

Top of the list for vetting images before allowing them into production should be *vulnerability scanning*. This is a process where your images are scanned at a binary level and their contents checked against a database of known security vulnerabilities.

If your organization has an automated CI/CD build pipeline, you should definitely integrate vulnerability scanning. As part of this, you should consider defining policies that automatically fail builds and quarantine images containing certain categories of vulnerabilities. For example, you might implement a build phase that scans images and automatically fails anything using images with known *critical* vulnerabilities.

Two things to keep in mind if you do this…

Firstly, scanning engines are only as good as the vulnerability databases they use.

Secondly, scanning engines might not implement intelligence. For example, a method in Python 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, the vulnerability *might* not be relevant and you *might* want to consider it a false positive. With this in mind, you may want to implement a solution that provides the ability to mark certain vulnerabilities as *not applicable*.

Configuration as code

Scanning application code for vulnerabilities is a widely adopted production best practice. However, reviewing application configurations, such as Dockerfiles and Kubernetes YAML files, is less widely adopted.

The *build once, run anywhere* mantra of containers means that a single container or Pod configuration can have hundreds or thousands of running instances. If a single one of these configurations pulls in vulnerable code, you can easily end up running hundreds or thousands of instances of vulnerable code. With this in mind, if you are not already reviewing your Dockerfiles and Kubernetes YAML files for security issues, you should start now!

A well-publicised example of not reviewing configurations was when an IBM data science experiment embedded private TLS keys in its container images. This made it possible for an attacker to pull the image and gain root access to the nodes that were hosting the containers. This would not have happened if a security review had been performed against the application's Dockerfiles.

There continue to be advancements in automating these types of checks 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 a *must have*. Fortunately, Kubernetes, and many container runtimes, support the ability to cryptographically sign and verify images.

In this model, developers cryptographically sign their images, and consumers cryptographically verify them when they `pull` and `run` them. This process gives the consumer confidence that the image they are working with is the image they asked for and has not been tampered with.

Figure 12.3 shows the high-level image signing and verification process.

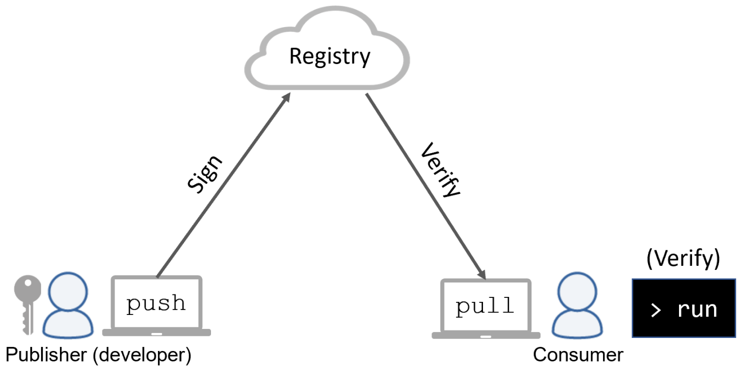


Figure 12.3

Image signing, and verification of signatures is usually implemented by the container runtime and Kubernetes does not get actively involved.

As well as signing images like this, higher-level tools, such as Docker Universal Control Plane, allow you to implement enterprise-wide policies that require certain teams to sign images before allowing them to be used.

Image promotion workflow

With everything that we've covered so far, a CI/CD pipeline for promoting an image to production should include as many of the following security-related steps as possible:

1. Configure environment to only `pull` and `run` signed images
2. Configure network rules to restrict which nodes can `push` and `pull` images
3. Configure repositories with RBAC rules
4. Developers build images using approved base images
5. Developers sign images and push to approved repos
6. Images are scanned for known vulnerabilities
    - Policies dictate whether images are promoted or quarantined based on scan results
7. Security team:

    - Reviews source code and scan results
    - Updates vulnerability rating as appropriate
    - Reviews container and Pod configuration files
8. Security team signs the image
9. All image pull and container run operations verify image signatures

These steps are examples and not intended to represent an exact workflow.

Let's switch our focus away from images and CI/CD pipelines.

Infrastructure and networking

In this section, we'll look at some of the ways we can isolate workloads.

We'll start at the cluster level, switch to the runtime level, and then look outside of the cluster at supporting infrastructure such as network firewalls.

Cluster-level workload isolation

Cutting straight to the chase, **Kubernetes does not support secure multi-tenant clusters**. **The only cluster-level security boundary in Kubernetes is the cluster itself**.

Let's look a bit closer…

The only way to divide a Kubernetes cluster is by creating *namespaces*. A Kubernetes namespace is not the same as a Linux kernel namespace, it is a logical partition of a single Kubernetes cluster. In fact, it's little more than a way of grouping resources and applying things like:

- Limits
- Quotas
- RBAC rules
- More…

The take-home point is that Kubernetes namespaces cannot guarantee a Pod in one namespace will not impact a Pod in another namespace. As a result, you should not run potentially hostile production workloads on the same physical cluster. The only way to run potentially hostile workloads, and guarantee true isolation, is to run them on separate clusters.

Despite this, Kubernetes namespaces are useful, and you *should* use them – just don't use them as security boundaries.

Let's look at how namespaces relate to *soft multi-tenancy* and *hard multi-tenancy*.

Namespaces and soft multi-tenancy

For our purposes, we'll define *soft multi-tenancy* as hosting multiple trusted workloads on shared infrastructure. By *trusted*, we mean workloads that do not require absolute guarantees that one Pod/container cannot impact another.

An example of two trusted workloads might be an e-commerce application with a web front-end service and a back-end recommendation service. Both services are part of the same e-commerce application, so are not hostile, but they might benefit from:

- Isolating the teams responsible for the different services
- Having different resource limits and quotas for each service

In this situation, a single cluster with one namespace for the front-end service and another for the back-end service might be a good solution. However, exploiting a vulnerability in one service might give the attacker access to Pods in the other service.

Namespaces and hard multi-tenancy

Let's define *hard multi-tenancy* as hosting untrusted and potentially hostile workloads on shared infrastructure. Only… as we said before, this isn't *currently* possible with Kubernetes.

This means that truly hostile workloads – workloads that require a strong security boundary – need to run on separate Kubernetes clusters! Examples include:

- Isolating production and non-production workloads on dedicated clusters
- Isolating different customers on dedicated clusters
- Isolating sensitive projects and business functions on separate clusters

Other examples exist, but you get the picture. If you have workloads that require strong separation, put them on their own clusters.

> **Note**: The Kubernetes project has a dedicated *Multitenancy Working Group* that is actively working on the multitenancy models that Kubernetes supports. This means that future releases of Kubernetes might support hard multitenancy.

Node isolation

There are times when individual applications require non-standard privileges such as running as root or executing non-standard syscalls. Isolating these on their own clusters might be overkill, but the increased risk of collateral damage would probably justify running them on a ring-fenced subset of worker nodes. In this case, if one Pod is compromised it can only impact other Pods on the same node.

You should also apply *defence in depth* principles by enabling stricter audit logging and tighter runtime defence 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 target workloads to sub-sets of nodes.

Runtime isolation

So far, we've looked at cluster-level isolation and node-level isolation. Now let's turn our attention to the various types of runtime isolation.

Containers versus virtual machines can be a polarizing topic. However, when it comes to workload isolation there is only one winner… the virtual machine!

The typical container model has multiple containers sharing a single kernel, and isolation is provided by kernel constructs that were never designed as *strong* security boundaries. We often call these *namespaced containers*.

In the hypervisor model, every virtual machine gets its own dedicated kernel and is strongly isolated from other virtual machines using hardware enforcement.

From a workload isolation perspective, virtual machines win.

However, it is becoming easier and more common to augment containers with additional kernel-level isolation technologies such as apparmor and SELinux, seccomp, capabilities, and user namespaces. Unfortunately, these can add significantly to the complexity of the setup and are still considered less secure than a virtual machine.

Another thing to consider is different classes of container runtime. Two prominent examples are **gVisor** and **Kata Containers**, both of which are re-writing the rules and providing stronger levels of workload isolation. Integrating runtimes like these with Kubernetes is made simple thanks to Kubernetes supporting the Container Runtime Interface (CRI) and Runtime Classes.

There are also projects that enable Kubernetes to orchestrate other workloads such as virtual machines and functions.

While all of this might feel overwhelming, everything discussed here needs to be considered when deciding what levels of isolation your workloads require.

To summarize, the following workload isolation options exist:

1. **Virtual Machines**: Every workload gets its own virtual machine and kernel. This provides excellent isolation but is relatively slow and heavy-weight.
2. **Traditional namespaced containers**: Every workload gets its own container but shares a common kernel. Not the best isolation, but fast and light-weight.
3. **Run every container in its own virtual machine**: This option attempts 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 this loses some of the appeal of containers and is not a popular solution.
4. **Use appropriate runtime classes**: This is extremely new but has a lot of potential. All workloads can run as containers, but workloads requiring stronger isolation are targeted to a class of container runtime that provides appropriate isolation (gVisor, Kata Containers etc.). Runtime classes is currently an alpha feature in Kubernetes.

A couple of other security-related things to consider…

Running lots of virtual machines can complicate things when it's time to patch operating systems. Also, running a mix of containers and virtual machines can increase network complexity.

Network isolation

On the topic of networking, firewalls are an integral part of any layered information security system. At a high level, they implement a set of rules that either *allow* or *deny* system-to-system communication.

As the names suggest, *allow rules* permit traffic to flow, whereas *deny rules* stop traffic flowing. The overall intent is to lock things down so that only authorized communications occur.

In Kubernetes, Pods communicate with each other over a special internal network called the *Pod network*. However, Kubernetes does not implement this *Pod network*, instead, it implements a plugin model called the Container Network Interface (CNI). Vendors and the community are responsible for writing the CNI plugins that actually provide the *Pod network*. Fortunately, there are lots of plugins available, and the networking options they support fall into the following two categories:

- Overlay
- BGP

Each of these is different, and each has a different impact on firewall implementation. Let's take a quick look at each.

Kubernetes and overlay networking

The most common way to build the *Pod network* is as an overlay network. In the Kubernetes world, overlay networking allows us to build a simple flat Pod network that hides any complexity that might exist between the nodes in the cluster. For example, you might have your cluster deployed across two different networks but have all Pods on a single flat Pod network. In this scenario, the Pods only know about the flat overlay Pod network and have no knowledge of the networks that the nodes are on. Figure 12.4 shows four nodes on two different networks, with Pods connected to a single overlay Pod network.

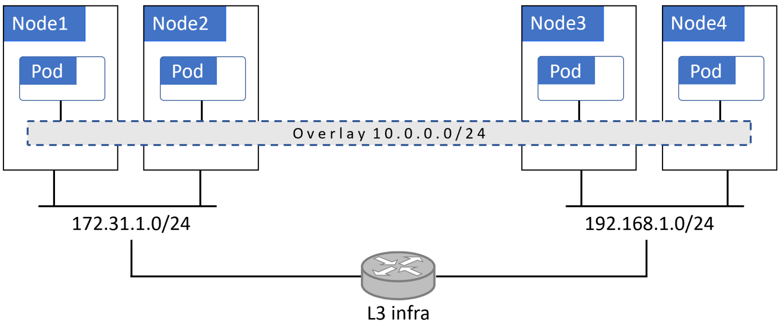


Figure 12.4

Generally speaking, overlay networks encapsulate packets for transmission over VXLAN tunnels. In this model, the overlay network is a virtual Layer 2 network operating on top of existing Layer 3 infrastructure. Traffic is encapsulated in order to pass between Pods on different nodes. This simplifies implementation, but encapsulation poses challenges for some firewalls. See Figure 12.5

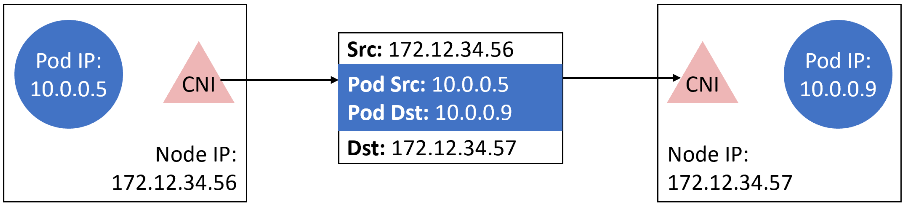


Figure 12.5

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 if you're new to BGP. 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.

This is similar to BGP. BGP Routing happens through a network of *peers* that help each other find a route for packets to go from one Pod to another.

BGP does not encapsulate packets, making life easier for firewalls. See Figure 12.6.

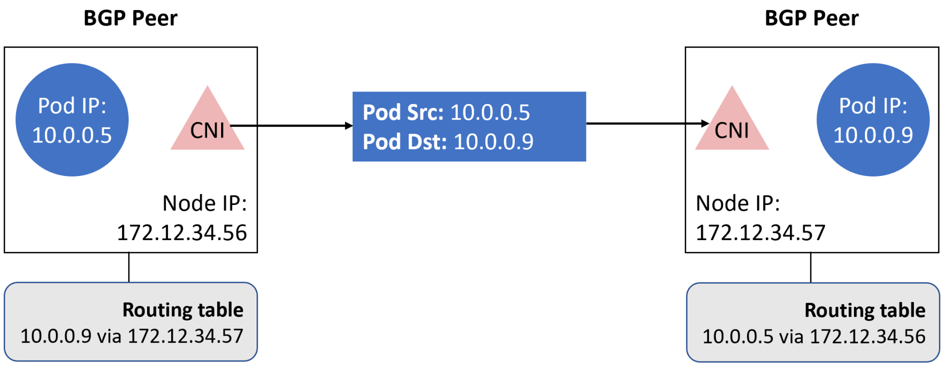


Figure 12.6

How this impacts firewalls

We've already defined a firewall as a network entity that allows or disallows 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

If your Pod network is an overlay network, source and destination Pod IP addresses are encapsulated so they can traverse the underlay network. This means firewalls that do not crack open packets and inspect their contents will not be able to filter based on Pod source and Pod destination IPs. You should consider this when choosing your Pod network and your firewall solutions.

With this in mind, if your Pod-to-Pod traffic has to traverse existing firewalls that do not perform deep packet inspection, it might be a better idea to choose a BGP-based Pod network. This is because BGP does not obscure Pod source and destination addresses.

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 systems (OS) features and are usually managed by the team that manages your OS. For example, the Linux sysadmins. Both solutions have their pros and cons, and a combination of the two is probably the most secure. However, you should consider things such as; whether your organization has a long and protracted procedure for implementing changes to physical firewalls. If it does, it might not suit the nature of your Kubernetes deployment and a different firewall solution might be preferable.

Packet capture

On the topic of networking and IP addresses, not only are Pod/container IP addresses sometimes obscured by encapsulation, they are also dynamic.

Pods and containers are designed to be ephemeral, meaning they are not long lived. Scaling part of an application up adds more Pods and more IP addresses, whereas scaling it down removes Pods and IP addresses. IP addresses can even be recycled and re-used by different Pods and containers. This causes a lot of *IP churn* and reduces how useful IP addresses are in 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 is 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 and other LDAP systems.

Most organizations already have a centralized IAM provider, such as Active Directory, that is integrated with company HR systems to simplify employee lifecycle management.

Fortunately, Kubernetes leverages existing IAM providers instead of implementing its own. For example, a new employee joining the company will automatically get an identity in Active Directory, which integrates with Kubernetes RBAC to automatically grant that user certain access to Kubernetes. Likewise, an employee leaving the company will automatically have his or her Active Directory identity removed or disabled, resulting in their access to Kubernetes being revoked.

RBAC went GA in Kubernetes 1.8 and it is highly recommended that you leverage its full capabilities.

Managing Remote SSH access to cluster nodes

Almost all Kubernetes administration is done via the API server, meaning it should be rare for a user to require remote SSH access to Kubernetes cluster nodes. In fact, remote SSH access to cluster nodes should only be for the following types of reason:

- Performing *node management* activities that cannot be performed via the Kubernetes API
- *Break the Glass* activities such as when the API server is down
- Deep troubleshooting

You should probably have tighter controls over who has remote access to the control plane nodes.

Multi-factor authentication (MFA)

With great power comes great responsibility…

Accounts with administrator 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, their use should be protected by multi-factor authentication (MFA) where possible. 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 device

Or…

- Stage 1: Tests *knowledge* of a username and password
- Stage 2: Tests something *about* the user, such as fingerprint or facial recognition

An easy, and important, place to implement multi-factor authentication is remote SSH access to cluster nodes. You should also consider it for access to workstations and user profiles that have kubectl installed.

Auditing and security monitoring

No system is 100% secure, and you should plan for the eventuality that your systems will be breached. When breaches happen, it is vital that 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 key to both of these requirements, 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* and *who did it…* In extreme circumstances, information like this can even be called upon in court.

Good auditing and monitoring solutions also help to identify vulnerabilities in your security systems.

With these points in mind, you should ensure that reliable auditing and monitoring is high on your list of priorities, and you should not go live in production without them.

Secure Configuration

There are various tools and checks that can be useful in ensuring your Kubernetes environment is provisioned according to best practices and in-line with company policies.

The Center for Information Security (CIS) has published an industry standard benchmark for Kubernetes security, and Aqua Security (aquasec.com) has written an easy-to-use tool called `kube-bench` to implement the CIS tests. In its most basic form, you run `kube-bench` against each node in your cluster and get a report outlining which tests passed and which failed.

Many organizations consider it a best practice to run `kube-bench` on all production nodes as part of the node provisioning process. Then, depending on your risk appetite, you can pass or fail provisioning tasks based on the results.

`kube-bench` reports can also serve as a valuable baseline in the aftermath of an incident. In situations like this, you run an additional `kube-bench` after a breach and compare the results with the initial baseline to determine if and where the configuration has changed.

Container and Pod lifecycle Events

As previously mentioned, Pods and containers are ephemeral in nature, meaning they don't live for long – certainly not as long as VMs and physical servers. This means you will see a lot of events announcing new Pods and containers, as well as a lot of events announcing terminated Pods and containers. It also means you may need a solution that stores container logs in an external system and keeps them around for a while after their Pods and containers have terminated. If you don't, you *may* find it frustrating that you do not have logs for old terminated containers available for inspection.

Logs entries relating to container lifecycle events may also be available from your container runtime (engine) logs.

Application logs

In some situation there is not a lot Kubernetes can do to protect the applications it is running. For example, Kubernetes cannot prevent an application from running vulnerable code. This means it is important to capture and analyse application logs as a way to identify potential security-related issues.

Fortunately, most containerized applications will log messages to standard out (stdout) and standard error (stderr), which are then directed to the container's logs. However, some applications send log messages to other locations such as proprietary log files, so be sure to check your application's documentation.

Actions performed by users

Most of your Kubernetes configuration will be done via the API server where all requests should be logged. However, it is also possible to gain remote SSH access to control plane nodes and directly manipulate Kubernetes objects. This may include local unauthenticated access to the API, as well as directly modifying control plane systems such as etcd.

We've already spoken about limiting who has remote SSH access to nodes and bolstering security via things like multi-factor authentication. However, logging all activities performed via SSH sessions and shipping those logs to a secure log aggregator is highly recommended. As is the practice of always having a second pair of eyes involved in remote access sessions.

Managing log data

A key advantage of containers is application density – we can run a lot more applications on our servers and in our data centers. While this is great, it has the side-effect of generating massive amounts of logging and audit-related data that can easily become too much to analyse using traditional tools. At the time of writing, there is a lot of work being done to resolve this, including areas such as machine learning, but there is currently no easy solution.

On the negative side, such vast amounts of log-related data makes proactive analysis difficult – too much data to analyse. However, on the positive side, we have a lot of valuable data that can be used by security first-responders as well as for post-event reactive analysis.

Migrating existing apps to Kubernetes

Every business has a mix of apps – some more business critical than others. With this in mind, it's important to adopt a careful and planned approach to migrating apps to Kubernetes.

One approach may be a crawl, walk and run strategy as follows:

1. *Crawl*: Threat modeling your existing apps will help you understand the current security posture of those applications. For example, which of your existing apps do and do not communicate over TLS.
2. *Walk*: When moving to Kubernetes, ensure the security posture of these apps remains unchanged; neither lower nor higher, just the same. For example, if an app does not communicate over TLS, do **not** change this as part of the migration.
3. *Run*: Start improving the security of applications after the migration is successful. Start with the simple non-critical apps, and carefully work your way up to the mission critical apps. You may also want to methodically deploy deeper levels of security. For example, initially configure apps to communicate over one-way TLS and then eventually over two-way TLS (potentially using a service mesh).

Real world example

A great example of a container-related vulnerability, that can be prevented by implementing some of the best practices we've discussed, occurred in February 2019. CVE-2019-5736 allows a container process running as `root` to escape its container and gain root access on the host **and** all containers running on that host.

As dangerous as the vulnerability is, the following things that we covered in this chapter would've prevented the issue:

- Vulnerability scanning
- Not running processes as root
- Enabling SELinux

As the vulnerability has a CVE number, security scanning tools would've found it and alerted on it. Also, organizations that did not allow container processes to run as root will have been protected, as the issue only affects processes running as root. Finally, common SELinux policies, such as those that ship with RHEL and CentOS, prevented the issue.

All in all, a great real-world example of the benefits of defence-in-depth and other security-related best practices.

Chapter summary

The purpose of this chapter was to give you an idea of some of the real-world security considerations effecting many Kubernetes clusters.

We started out by looking at ways to secure the software delivery pipeline by discussing some image-related best practices. These included; how to secure your image registries, scanning images for vulnerabilities, and cryptographically signing 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 talked about 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 be easily avoided by implementing some of the best practices already covered.

Hopefully you now have enough understanding to go away and start securing your own Kubernetes clusters.

Glossary

This glossary defines some of the most common Kubernetes-related terms used throughout the book. Ping me if you think I've missed anything important:

- https://nigelpoulton.com/contact-us
- https://twitter.com/nigelpoulton
- https://www.linkedin.com/in/nigelpoulton/

Now then… I'm well aware that some people are passionate about their own particular definitions of technical terms. I'm OK with that, and I'm not saying my definitions are the best – they're designed to be helpful for readers.

OK, here goes.

API Server: Exposes the features of Kubernetes over an HTTPS REST interface. All communication with Kubernetes goes through the API Server – even cluster components communicate via the API Server.

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.

Cloud native: This is a loaded term and means different things to different people. I personally consider an application to be *cloud-native* if it can self-heal, scale on-demand, perform rolling updates, and possibly rollbacks. They're usually microservices apps.

ConfigMap: Kubernetes object used to hold non-sensitive configuration data. A great way to add custom configuration data to a generic application template without editing the template.

Container Network Interface (CNI): Pluggable interface enabling different network topologies and architectures. 3rd-parties provide various CNI plugins that enable overlay networks, BGP networks, and various implementations of each.

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 Docker or containerd.

Container Runtime Interface (CRI): Interface 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.

Controller: Control plane process running as a reconciliation loop monitoring the cluster (via the API Server) and making the necessary changes so the observed state of the cluster matches desired state.

Cluster store: Holds cluster state, including desired state and observed state. Typically based on the etcd deistributed data store and runs on the Masters. Can be deployed separately to its own cluster for higher performance and higher availability.

Deployment: Controller that deploys and manages a set of stateless Pods. Performs rolling updates and versioned rollbacks. 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 5 replicas of xyz container listening on port 8080/tcp.

Endpoints object: Up-to-date list of healthy Pods that match a Service's label selector. Basically, it's the list of Pods that a Service will send traffic to. Might eventually be replaced by EndpointSlices.

K8s: Because writing Kubernetes is too hard ;-) The 8 replaces the eight characters in Kubernetes between the "K" and the "s". Pronounced "Kates". The reason why people say Kubernetes' girlfriend is called Kate ¯_(□)_/¯.

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 routing of traffic 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. For example, Services send traffic to Pods based on 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 the 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 the Service. When you post a manifest file to the API Server, its configuration is deployed to the cluster.

Master: The brains of a Kubernetes cluster. A node that hosts control plane features (API Server, cluster store, scheduler etc.). Usually deployed in highly available configurations of 3, 5, or 7.

Microservices: A design pattern for modern applications. Application features are broken out into their own small applications (microservices) and communicate via APIs. They work together to form a useful application experience.

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: The workers of a Kubernetes cluster. A cluster node designed to run user applications. Runs the kubelet process, a container runtime, and kube-proxy service.

Observed state: Also known as *current state* or *actual state.* This is the latest 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 lots of smaller apps that work together to form a useful application. Kubernetes orchestrates/manages these small apps and keeps them healthy, scales them up and down, and more…

PersistentVolume (PV): Kubernetes object used to map storage volumes on a cluster. Storage resources must be mapped to PVs before they can be used by applications.

PersistentVolumeClaim (PVC): Like a ticket/voucher that allows an app to use a 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.

Reconciliation loop: A controller process watching the state of the cluster, via the API Server, ensuring observed state matches desired state.

ReplicaSet: Runs as a controller and performs self-healing and scaling. Used by Deployments.

Secret: Like a ConfigMap for sensitive configuration data.

Service: Capital "S". Provides stable networking for a dynamic set of 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.

StatefulSet: Controller that deploys and manages stateful Pods. Similar to a Deployment, but for stateful applications.

StorageClass (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.

What next

There are lots of ways to take your Kubernetes journey to the next level, and fortunately, most of them are easy.

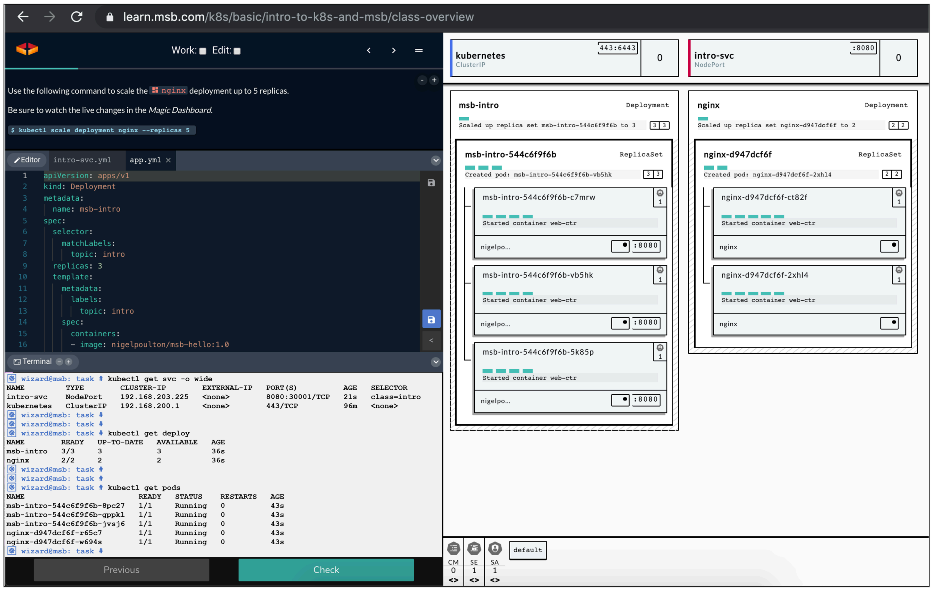
Practice makes perfect

I know I'm being *Captain Obvious* with this one, but there's no substitute for hands-on practice. Fortunately, it's never been easier to spin-up a Kubernetes playground where you can practice until you're a world authority!

I recommend the following:

- Magic Sandbox (msb.com)
- Play with Kubernetes
- Docker Desktop

Magic Sandbox is a company that I have extremely close connections with (I'm Head of Content). It's the ultimate place for getting hands-on with your own private fully-functioning multi-node cluster. You also get curated labs that you can follow along with, an amazing live dashboard that shows your cluster and applications in real-time, and much much more. It's a subscription-based service, but I highly recommend you check it out – it even has a free tier for you to sample.



Use the following command to scale the nginx deployment up to 5 replicas.

Be sure to watch the live changes in the *Magic Dashboard*.

```
$ kubectl scale deployment nginx --replicas 5
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: msb-intro
spec:
  selector:
    matchLabels:
      topic: intro
  replicas: 3
  template:
    metadata:
      labels:
        topic: intro
    spec:
      containers:
      - image: nigelpoulton/msb-hello:1.0
```

```
wizard@msb: task # kubectl get svc -o wide
NAME         TYPE        CLUSTER-IP        EXTERNAL-IP   PORT(S)          AGE   SELECTOR
intro-svc    NodePort    192.168.203.225   <none>        8080:30001/TCP   21s   class=intro
kubernetes   ClusterIP   192.168.200.1     <none>        443/TCP          96m   <none>
wizard@msb: task #
wizard@msb: task #
wizard@msb: task # kubectl get deploy
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
msb-intro   3/3     3            3           36s
nginx       2/2     2            2           36s
wizard@msb: task #
wizard@msb: task #
wizard@msb: task # kubectl get pods
NAME                           READY   STATUS    RESTARTS   AGE
msb-intro-544c6f9f6b-8pc27     1/1     Running   0          43s
msb-intro-544c6f9f6b-gppkl     1/1     Running   0          43s
msb-intro-544c6f9f6b-jvsj6     1/1     Running   0          43s
nginx-d947dcf6f-r65c7          1/1     Running   0          43s
nginx-d947dcf6f-w694s          1/1     Running   0          43s
wizard@msb: task #
```

```
Scaled up replica set msb-intro-544c6f9f6b to 3
Created pod: msb-intro-544c6f9f6b-vb5hk
Started container web-ctr
Started container web-ctr
Started container web-ctr
Scaled up replica set nginx-d947dcf6f to 2
Created pod: nginx-d947dcf6f-2xhl4
Started container web-ctr
Started container web-ctr
```

Play with Kubernetes gets you a time-limited internet-based sandbox – you get 4 hours on your very own cluster. It's provided as a free service, so sometimes performance and availability aren't good. But hey, it's free.

Docker Desktop is a free desktop app from Docker, Inc. available for your Mac and PC. It includes a single-node development cluster that's great if you need something to play around with on your laptop.

Other options exist, and all of them are a lot simpler than how things used to be. I remember studying for my MSCE in Windows NT and spending countless hours rebuilding NT domains from CD installs on dusty old Compaq PCs in my bedroom. Things are so much simpler these days, there is no excuse for not getting our hands dirty.

More books

I've got a book on Docker that's had stellar reviews, including being named as *Best Docker book of all time* by BookAuthority. Docker and containers are integral to Kubernetes, so if you need to know more about Docker and container, check it out – it's called Docker Deep Dive, and you can get it on Amazon and Leanpub.

Video training

If you liked this book, you'll **love** my video courses!

- Kubernetes 101 (nigelpoulton.com)
- Getting Started with Kubernetes (pluralsight.com)
- Kubernetes Deep Dive (acloud.guru/learn/kubernetes-deep-dive)

I've also got several Docker courses on Pluralsight.

If you're not a member of Pluralsight or A Cloud Guru, I recommend becoming one! Yes, they cost money, but they could be the best investments you ever make into your career! A monthly subscription on each platform gets you access to **every course in that platform's library** – everything from developer to IT ops. And if you're unsure about spending your money, there's usually a free trial where you can get free access for a limited time.

Events and meetups

You should hit events like KubeCon and ServiceMeshCon. There's so many great people at these events and so much to learn.

You should also get involved with your local Kubernetes, DevOps, cloud-native, and Docker meetups. Go to `meetup.com` and type in "Kubernetes" or "DevOps" and it'll find meetups in your local area. Alternatively you can just type something like "kubernetes meetup Manchester" into Google and it'll find your local meetups.

That's it for now. Keep learning!

Feedback

Massive thanks for reading my book. I'm humbled that you bought it and hope you loved it.

I'd really appreciate it if you'd:

- tell a friend or colleague about it
- leave a review on Amazon

It takes no time at all to write an Amazon review, and you can leave a review even if you bought the book form Leanpub or somewhere else.

Also, feel free to hit me on Twitter[2].

@nigelpoulton

And feel free to visit **nigelpoulton.com** to find all of my content, including; news & updates, latest YouTube videos, workshops, webinars, and more.

That's it. Live long and prosper…

[2]https://twitter.com/nigelpoulton