Hands on

QUICK START KUBERNETES

2022 Edition

nigelpoulton

Quick Start Kubernetes

Nigel Poulton

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

This version was published on 2022-01-19

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

About the author

Nigel Poulton (@nigelpoulton)

Hi, I'm Nigel. I live in the UK and I'm a techoholic. In fact, working with technologies like the cloud and containers is *living the dream* for me!

My early career was massively influenced by a book called *Mastering Windows Server 2000* by Mark Minasi. Ever since reading it, I wanted to write my own books so I could influence people's lives and careers the way Mark's book influenced mine. Since then, I've been fortunate enough to author several best-selling books, including *Data Storage Networking*, *Docker Deep Dive*, and *The Kubernetes Book*. I feel so privileged to have reached so many people and genuinely appreciate the feedback I receive.

I'm also the author of best-selling video training courses on Docker and Kubernetes. My videos are always entertaining, and occasionally laugh-out-loud funny (not my words).

See a full list of my videos at `https://nigelpoulton.com/video-courses`

At my website, nigelpoulton.com, you'll also find all my books, my blog, weekly newsletter, and other things that I'm doing.

When I'm not working with containers and Kubernetes, I'm dreaming about them. When I'm not dreaming about them, I'm spending time with my family. I also like American muscle cars, playing golf, reading sci-fi, and I support Sunderland AFC (the greatest football/soccer team in the world).

I'm happy to connect and I'm always looking for ways to improve my books and videos. These are some of the easiest ways to get connected.

- twitter.com/nigelpoulton
- linkedin.com/in/nigelpoulton/
- nigelpoulton.com
- qskbook@nigelpoulton.com

About the book

As the title states, this is a **quick start** guide to Kubernetes. It doesn't cover everything, just the important bits. And it covers them as clearly as possible in an engaging way. It's also a great mix of theory and optional hands-on.

The goal is to demystify Kubernetes and get you some hands-on experience.

Who is the book for

I wrote the book for anyone and everyone that needs to get up-to-speed with the *fundamentals* of Kubernetes and likes to *learn by doing.*

The entire book, theory and labs, have been hand-crafted over many years so that technical and non-technical readers will benefit equally. So, whether you work in marketing, management, architecture, or one of the many hands-on roles, you're going to love the book.

What does the book cover

You'll learn ***why*** we have Kubernetes, ***what*** it is, and ***where*** it's going.

On the theory front, you'll learn about microservices, orchestration, why Kubernetes is the OS of the cloud, and the architecture of a Kubernetes cluster. On the hands-on front, you'll have the opportunity to build a cluster, containerize an app, deploy it, break it, see Kubernetes fix it, scale it, and perform an application update.

And as this is a quick start guide, you'll be up-to-speed in no time at all.

Will the book make you a Kubernetes expert

No, but it will start you on your journey to *becoming* an expert.

Will you know what you're talking about when you finish the book

Yes, you'll be able to hold your own in conversations about Kubernetes at the coffee machine and with your peers.

Editions

The following English language editions are available in as many Amazon territories as possible:

- Hardback
- Paperback
- Kindle

Ebook copies are also available from Leanpub.com

The following translations are available on Amazon and Leanpub (or will be very soon).

- French, German, Hindi, Italian, Portuguese, Russian, Simplified Chinese, Spanish.

Terminology and responsible language

Throughout the book I capitalize Kubernetes API objects. Wow, we haven't even started and I'm throwing jargon around :-D

Put more simply, Kubernetes *features* such as Pods and Services are spelled with a capital letter. This helps you know when I'm talking about a Kubernetes "Pod" and not a "pod" of whales.

The book also follows guidelines from the Inclusive Naming Initiative (inclusivenaming.org) which promotes responsible language. As an example, the Kubernetes project has replaced the potentially harmful term "master" with "control plane node". This book does the same and attempts to follow all guidance from the inclusive naming initiative.

Feedback

If you like the book and it's helped your career, share the love by recommending it to a friend and leaving a review on Amazon.

Modern books live and die by Amazon reviews and stars, so I'd be grateful if you could write one. You can sometimes leave an Amazon review even if you got the book from somewhere else.

For other feedback you can reach me at `qskbook@nigelpoulton.com`.

The sample app

As this is a hands-on book, it has a sample app.

It's a simple Node.js app available on GitHub at:

`https://github.com/nigelpoulton/qsk-book/`

Don't stress about the app and GitHub if you're not a developer. The focus of the book is Kubernetes, not the app. Plus, we explain everything in layman terms when talking about the app, and you don't have to know how to use GitHub.

If you're interested, the code for the app is in the `App` folder and comprises the following files.

- **app.js:** This is the main application file. It's a Node.js web app.
- **bootstrap.css:** This is a design template for the application's web page.
- **package.json:** This file lists any dependencies the app has.
- **views:** This is a folder for the contents of the application's web page.
- **Dockerfile:** This file tells Docker how to build the app into a container.

If want to download the app now, run the following command. You'll need `git` installed on your machine. It's OK if you don't have `git`, we show you how to get it and download the app later in the book.

```
$ git clone https://github.com/nigelpoulton/qsk-book.git
```

```
$ cd qsk-book
```

Finally, the app is checked at least once per year for package updates and known vulnerabilities.

1: What is Kubernetes

The goal of this chapter is simple… describe Kubernetes in the clearest way possible. Oh, and do it without putting you to sleep.

At its core, Kubernetes is an *orchestrator* of *cloud-native microservices* applications.

But that's a lot of buzzwords in a short sentence. So, let's explain the following:

- What are microservices
- What is cloud-native
- What is an orchestrator

What are microservices

In the past, we built and deployed *monolithic applications*. That's jargon for an application where *every feature is bundled as a single large package*. If you look at Figure 1.1, you'll see the web front end, the authentication, the logging, the data store, the reporting system… all tightly coupled and bundled as a single app. This means if you want to change one part of it, you have to change all of it.

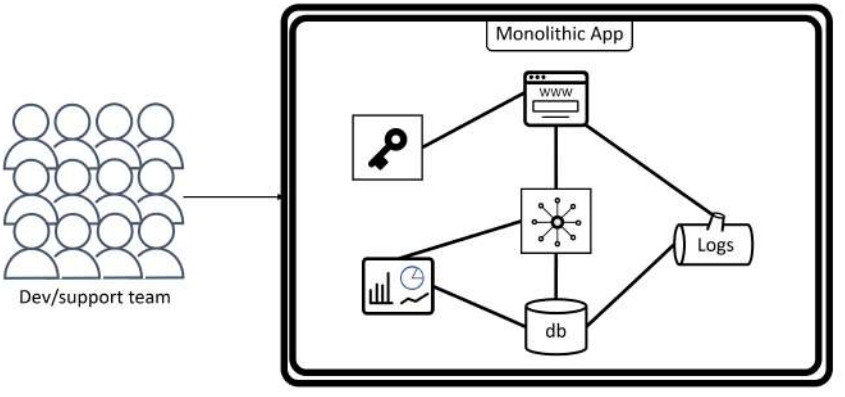


Figure 1.1

As a quick example, if you need to patch or update the reporting feature of the app in Figure 1.1, you have to take the entire app down and patch/update the whole thing. Working like this requires painful amounts of planning, carries huge risk and complexity, and normally has to be done over long boring weekends with everyone in the office consuming too much pizza and coffee.

However, the pain of monolithic applications doesn't stop there. If you want to scale a single feature, you have to scale the whole thing.

Basically, every application feature is bundled, deployed, upgraded, and scaled, as a large single (monolithic) unit. This is clunky and far from ideal.

On the flip side, a *microservices application* takes the same set of features – the web front end, the authentication, the logging, the data store, the reporting system – and treats each one as its own small application. This is where the term "microservice" comes from.

If you look closely at Figure 1.2, you'll see it's the exact same set of application features as Figure 1.1. The difference is that each feature is developed independently, each one is deployed independently, and each can be updated and scaled independently. However, they work together to create the exact same *application experience.*

The most common pattern is to develop and deploy each microservice as its own container. For example, a container for the web front-end microservice, a different container for the authentication microservice, a different one for the reporting microservice etc. Each is independent, but loosely coupled over the network to create the same app experience.

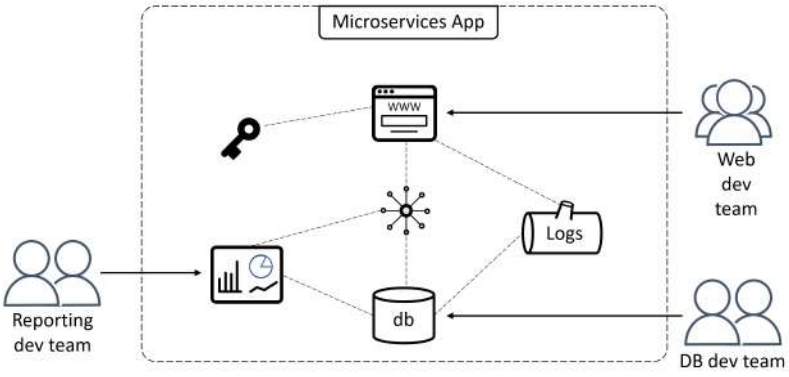


Figure 1.2

Microservices are loosely coupled by design, and this is fundamental to the ability to change one microservice without affecting others. Technically speaking, each microservice normally exposes an API over an IP network that the other microservices use to consume to it.

The following car analogy might help if you're new to the concept of APIs.

Cars come in all shapes and sizes with engine configurations ranging from inline-fours, flat-sixes, V8s and even electric motors. However, all this complexity is hidden from the driver via standardised controls that include a steering wheel, accelerator and brake pedals, and a speedometer. In this model, the controls are the car's API – how we consume its capabilities. A major benefit of this model is the ability to get into any car in the world and be able to drive it. For example, I learned to drive in a low powered front-wheel drive petrol engine car with a 4-cylinder engine. However, I can step into an all-wheel drive electric car and be able to drive it without having to learn any new skills – the complexity of the engine/motor and the drive system are abstracted via the standardised steering wheel and foot pedals (API). Continuing the example, it's possible to swap engines in a car, replace wheels and tyres, upgrade exhaust systems, and more… all without the driver having to learn any new driving skills.

Bringing this back to microservices applications. As long as you don't change the API for a microservice, you can patch or update it without the other microservices and application users having to change.

As well as the ability to update and scale microservices independently, the microservices *design pattern* lends itself to smaller, more agile, development teams that can iterate on features faster. This is based on the *two pizza team rule* coined by

Jeff Bezos that states if you can't feed a development team on two pizzas, the team is too big. Generally speaking, teams of 2-8 can communicate and work together with more agility and less politics than bigger teams.

There are other advantages to the microservices design pattern, but hopefully you get the picture – developing features as independent microservices allows them to be developed, deployed, updated, scaled, and more, without impacting other parts of the application.

However, microservices isn't all green hills and blue skies. They can be complex beasts with lots of moving parts managed by different teams. This needs careful management.

Finally, these two ways of designing applications – monolithic vs microservices – are called *design patterns.* The microservices design pattern is the most common pattern in the current cloud era.

What is cloud-native

This is an easy one as we've covered some of it already.

A *cloud-native* app must:

- Scale on demand
- Self-heal
- Support rolling updates
- Run anywhere that has Kubernetes

Let's just take a second to define what some of those buzzwords mean.

Scaling on demand is the ability for an application, and associated infrastructure, to automatically grow and shrink to meet current requirements. For example, an online retail app might need to scale up infrastructure and application resources during special holiday periods, and then scale them back down when the holiday ends. If configured correctly, Kubernetes can automatically scale applications and infrastructure up when demand increases. It can also scale them down when demand drops off.

Not only does this help businesses react more quickly to unexpected changes, it reduces infrastructure costs when scaling down.

Kubernetes can also *self-heal* applications and individual microservices. This requires a bit more knowledge of Kubernetes that we'll cover later. But for now, when you deploy an application to Kubernetes, you tell Kubernetes what it should look like – things like how many instances of each microservice and which networks to attach to. Kubernetes saves this as *desired state* and watches the app to make sure it always matches *desired state*. If something changes, may be a microservice crashes, Kubernetes notices this and spins up a replacement. This is called *self-healing* or *resiliency*.

Rolling updates is the ability to update parts of an application without taking it offline and potentially without clients even noticing. It's a game-changer in the modern always-on business world, and we'll see it in action later.

One final point. Being *cloud-native* has almost nothing to do with the public cloud. It's the set of features and capabilities we've discussed. As such, a cloud-native application can run anywhere you have Kubernetes – AWS, Azure, Linode, your on-premises datacenter, or your Raspberry Pi cluster at home.

In summary, cloud-native apps are resilient, automatically scale, and can be updated without downtime. They can also run anywhere you have Kubernetes, even on prem.

What is an orchestrator

I always find an analogy helps with this.

An orchestra is a group of individual musicians that play different musical instruments. Each musician and instrument can be different and have a different role to play when the music starts. There are violins, cellos, harps, oboes, flutes, clarinets, trumpets, trombones, drums, and even triangles. Each one is different and has a different role in the orchestra.

As shown in Figure 1.3, each one is an individual and hasn't been assigned a role – it's a mess, the drum kit is even upside-down.

Figure 1.3

A *conductor* comes along, with the sheet music and a baton, and enforces order. She groups the strings together at the front of the stage, woodwind in the middle, brass a little further back, and percussion high up at the back. She also directs everything – telling each group when to play, how loud or soft to play, and how fast to play.

In short, the conductor takes the chaos from Figure 1.3, imposes the order in Figure 1.4, and ensures the music is played as the composer intended.

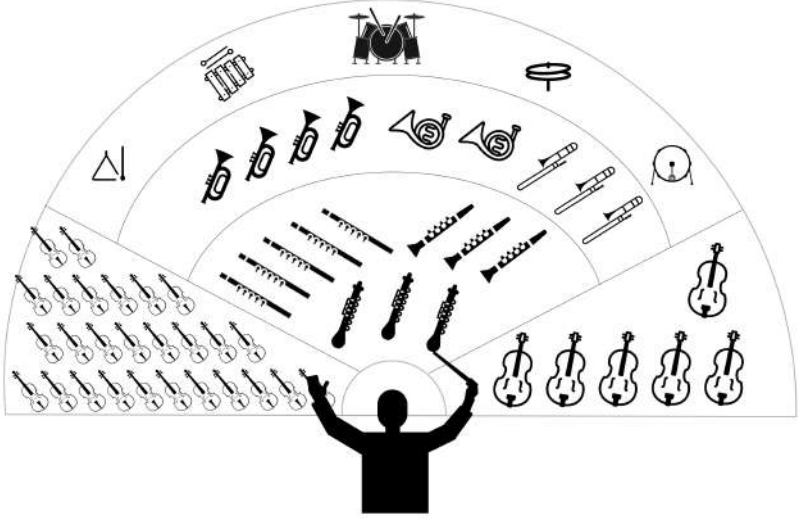


Figure 1.4

Well, cloud-native microservices applications are just like orchestras. Seriously, stick with me…

Each cloud-native app is made of lots of small microservices that do different jobs. Some serve web requests, some authenticate sessions, some do logging, some persist data, some generate reports. But just like an orchestra, they need someone, or something, to organise them into a useful app.

Enter Kubernetes.

Kubernetes takes a mess of independent microservices and organises them into a meaningful app as shown in Figure 1.5. As previously mentioned, it can scale the app, self-heal it, update it, and more.

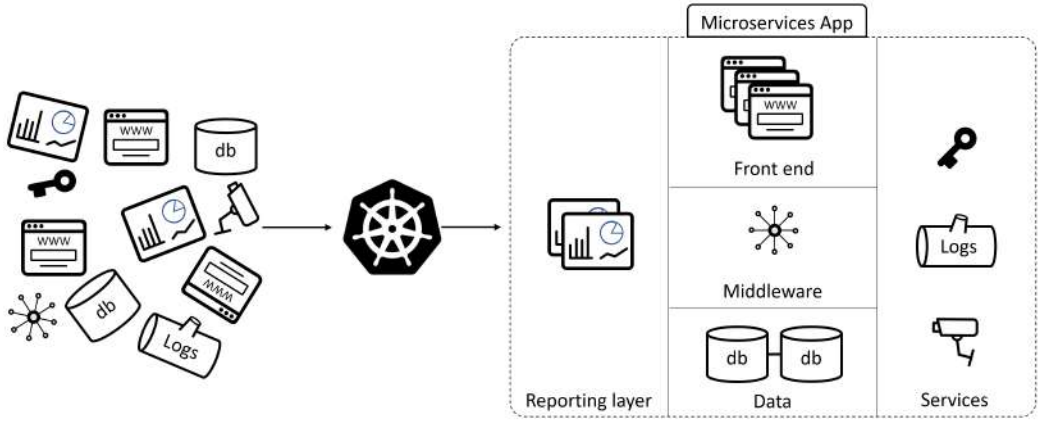


Figure 1.5

In summary, orchestrators like Kubernetes bring together different microservices and organise them into an application that brings value. It also provides and manages cloud-native features such as scaling, self-healing, and updates.

Other useful Kubernetes stuff to know

The name "Kubernetes" comes from the Greek word meaning "helmsman". This is a nautical/sailing term for the person that steers a ship. See Figure 1.6.

Figure 1.6

The wheel of a ship is called the "helm" and is where the Kubernetes logo comes from.

Figure 1.7

However, if you look closely, you'll see the Kubernetes logo has 7 spokes instead of the usual 6 or 8. This is because Kubernetes was originally based on an internal Google tool called "Borg", and the founders wanted to name Kubernetes "Seven of Nine" after the famous Borg drone from Star Trek.

If you know much about Star Trek, you'll know that Seven of Nine is a Borg drone rescued by the crew of the USS Voyager under the command of Captain Kathryn Janeway on stardate 25479. Unfortunately, copyright laws prevented Kubernetes from being called "Seven of Nine". However, the founders wanted some form of reference back to Borg and Star Trek, so they gave the logo *seven* spokes in a subtle

reference to "Seven of Nine".

You'll also see Kubernetes shortened to "K8s" where the "8" represents the 8 characters in Kubernetes between the "K" and the "s". It's generally pronounced "kates" and started the joke that Kubernetes has a girlfriend called Kate.

None of this will make you better at deploying and managing cloud-native microservices apps, but it's useful background knowledge ;-)

Chapter summary

At the top of the chapter, we said *Kubernetes is an orchestrator of cloud-native microservices applications.*

Now that we've busted some jargon, you know this means *"Kubernetes runs and manages applications comprised of small, specialised parts that can self-heal, and scale and be updated independently without requiring downtime."* Those specialised parts are called *microservices* and each one is usually deployed in its own container.

But that's still a lot to learn, and you're not expected to fully understand things yet. That's fine, and we'll continue explaining things throughout the rest of the book, and we'll get hands-on with a bunch of examples that will really drive things home.

2: Why we need Kubernetes

No prizes for guessing the objective of this chapter ;-)

Anyway, we'll split the conversation into two parts:

- Why tech companies need Kubernetes
- Why the user community needs Kubernetes

Both are important, and both play a major role in why Kubernetes is here for the long run. Some of the points will also help you avoid potential pitfalls when starting out with Kubernetes.

Why tech companies need Kubernetes

It all starts with AWS.

In the mid-to-late 2000's, Amazon fired a rocket up the proverbial backside of the tech industry, and the world has never been the same.

Prior to 2006 there was a status quo in the tech industry. The big tech companies were making easy money selling servers, network switches, storage arrays, licenses for monolithic apps and many other things. Then, from way out in the left field, Amazon launched AWS and turned the world upside down. It was the birth of modern cloud computing.

At first, none of the big tech companies paid much attention – they were too busy raking the cash in selling the same old stuff they'd been selling for decades. In fact, some of the major tech companies thought they could end the threat of AWS via crude campaigns of misinformation. A lot of them started out saying the cloud wasn't a real thing. When that didn't work, they performed a 180, admitted it was real, and immediately rebranded their existing legacy products as "cloud". When that didn't work, they started building their own clouds and cloud services, and they've been playing catch-up ever since.

A couple of things to note.

First up, that's *the condensed version of cloud history according to Nigel.*

Second up, the misinformation initially spread by the tech industry is known as *fear uncertainty and doubt (FUD).*

Anyway, let's get into a bit more detail.

Once AWS started stealing customers and future business, the industry needed a counterattack. Their first major retaliation was OpenStack. To keep a long story short, OpenStack was a community project that tried to create an open-source alternative to AWS. It was a noble project and a lot of good people contributed. But ultimately it never threatened AWS – Amazon had too much of a head start and were innovating at warp speed. OpenStack tried hard, but AWS brushed it aside without skipping a beat.

So, it was back to the drawing board for the industry.

While all of this was going on, and even before, Google was using Linux containers to run most of its services at massive scale. It's no secret that Google have been deploying billions of containers per week for as long anyone can remember. Scheduling and managing these billions of containers was a proprietary internal tool called *Borg.* Google being Google, they learned a bunch of lessons using Borg and built newer system called *Omega.*

Anyway, some of the folks inside of Google wanted to take the lessons learned from Borg and Omega and build something better and make it open-source and available to the community. And that's how Kubernetes came into existence in the summer of 2014.

Now then, Kubernetes is not an open-source version of Borg or Omega. It is a new project, built from scratch, to be an open-source orchestrator of containerised applications. It's connection to Borg and Omega is that its initial developers worked at Google on Borg and Omega, and that it was built with lessons learned from those proprietary internal Google technologies.

Back to our story about AWS eating everyone's lunch…

When Google open-sourced Kubernetes in 2014 Docker was taking the world by storm. As a result, Kubernetes was seen primarily as a tool to help users manage the explosive growth of containers. And while that's true, it's only half the story. Kubernetes also does an amazing job of *abstracting* underlying cloud and server infrastructure – basically *commoditizing infrastructure.*

"Abstracting and commoditizing infrastructure" is a fancy way of saying *Kubernetes makes it so you don't have to care whose cloud or servers your apps are running on.* In fact, this is at heart of the notion that *Kubernetes is the operating system (OS) of the cloud.* In the same way Linux and Windows mean you don't have to care if your apps are running on Dell, Cisco, HPE, or Nigel Poulton servers… Kubernetes means you don't have to care if your apps are running on AWS, Azure, or the Nigel Poulton cloud.

Abstracting clouds meant Kubernetes presented an opportunity for the tech industry to wipe out the value of AWS – just write your applications to run on Kubernetes and it makes no difference whose cloud is underneath. Thanks to Kubernetes, the playing field is levelled.

This ability to remove the value of AWS is a major reason vendors place Kubernetes front-and-center in their offerings. This creates a strong, bright, and long future for Kubernetes, which in turn, gives the user community a safe and vendor-neutral platform to bet their cloud future on.

Speaking of end users…

Why the user community needs Kubernetes

We've just made the case for a long and bright future for Kubernetes with all the major tech companies backing it. In fact, it grew so rapidly and became so important, even Amazon reluctantly embraced it. That's right, even the mighty Amazon and AWS couldn't ignore Kubernetes.

Anyway, the user community needs platforms it can build on, safe in the knowledge those platforms will be good long-term technology investments. As things stand, Kubernetes looks like it'll be with us for a very long time.

Another reason the user community needs and loves Kubernetes is back to the notion of *Kubernetes as the OS of the cloud*.

We've already said Kubernetes can abstract lower-level on-prem and cloud infrastructure, allowing you to write your apps to run on Kubernetes without even knowing which cloud is behind it. Well, this has a few side benefits, including:

- You can deploy to one cloud today and switch to another tomorrow
- You can run multi-cloud

- You can more easily ramp onto a cloud and then ramp off back to on-prem

Basically, applications written for Kubernetes will run anywhere that you have Kubernetes. It's a lot like writing apps for Linux – if you write your apps to work on Linux, it doesn't matter if Linux is running on Supermicro servers in your garage or AWS cloud instances on the other side of the planet.

All of this is great for end users. I mean who doesn't want a platform that brings flexibility and has a solid future!

Chapter Summary

In this chapter, you learned that the major tech companies need Kubernetes to be a success. This creates a strong future for Kubernetes and makes it a safe platform for users and companies to invest in. Kubernetes also abstracts underlying infrastructure the same way operating systems like Linux and Windows do. This is why you'll hear it referred to as *the OS of the cloud.*

3: What does Kubernetes look like

We've already said Kubernetes is the *OS of the cloud*. As such, it sits between applications and infrastructure. Kubernetes runs on infrastructure, and applications run on Kubernetes. This is shown in in Figure 3.1

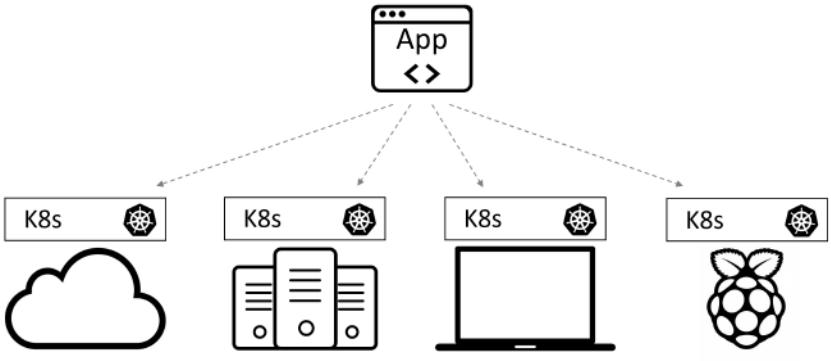


Figure 3.1

The diagram shows 4 Kubernetes installations running on 4 different infrastructure platforms. Because Kubernetes abstracts the underlying infrastructure, the application at the top of the diagram can run on any of the Kubernetes installations. You can also migrate it from one Kubernetes installation to another.

We call a Kubernetes installation a *Kubernetes cluster*.

There are a couple of things worth clarifying about Figure 3.1.

Firstly, it's unusual for a single Kubernetes cluster to span multiple infrastructures. For example, you aren't likely to see a single Kubernetes cluster spanning multiple clouds. Likewise, you're unlikely to see clusters that span on-prem and the public cloud. This is mainly due to network speed and reliability. Kubernetes needs reliable low-latency networks connecting the nodes in a cluster.

Secondly, although Kubernetes can run on many platforms, applications that run on Kubernetes have stricter requirements. You'll see this later in the chapter, but Windows applications will only run on Kubernetes clusters with Windows nodes,

Linux apps will only run on clusters with Linux nodes, and apps written for ARM/Raspberry Pis require clusters with ARM nodes.

Control plane nodes and worker nodes

A *Kubernetes cluster* is one or more machines with Kubernetes installed. The *machines* can be physical servers, virtual machines (VM), cloud instances, your laptop, Raspberry Pis, and more. Installing Kubernetes on these machines and connecting them together creates a *Kubernetes cluster*. You can then deploy applications to the cluster.

We normally refer to machines in a Kubernetes cluster as *nodes.*

Speaking of nodes, a Kubernetes cluster has two types:

- Control plane nodes
- Worker nodes

Some older documentation may refer to control plane nodes as "masters". This terminology has been phased out as part of the Inclusive Naming Initiative (inclusivenaming.org) which seeks to remove potentially harmful language from technology projects.

Control plane nodes host the internal Kubernetes services, whereas worker nodes are where user applications run.

Figure 3.2 shows a 6-node Kubernetes cluster with 3 control plane nodes and 3 worker nodes. It's good practice to prevent user applications from running on control plane nodes and only run them on worker nodes.

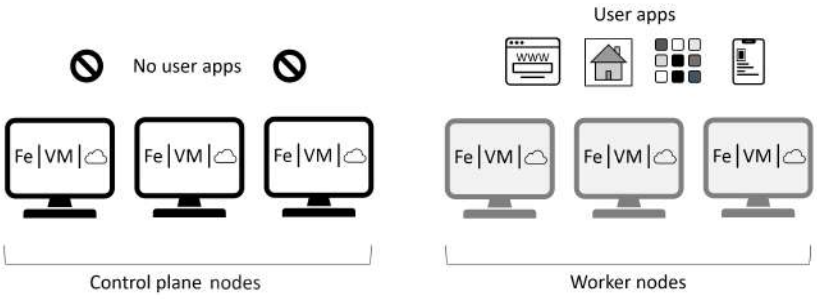


Figure 3.2

Control plane nodes

Control plane nodes host the internal Kubernetes systems services required for Kubernetes to operate. Collectively we call them the *control plane.* If that sounds like jargon, it's just a fancy way of saying the brains of Kubernetes.

With this in mind, it's good practice to have multiple control plane nodes for high availability (HA). This way, if one of them fails, the cluster can remain operational. In the real world. it's common to have 3 or 5 control plane nodes in a production cluster and to spread them across failure domains – don't stick them all on the same floor tile under the same leaky aircon unit on the same glitchy power supply.

Figure 3.3 shows a highly-available control plane with 3 nodes. Each one is in a separate failure domain with separate network and power infrastructures etc.

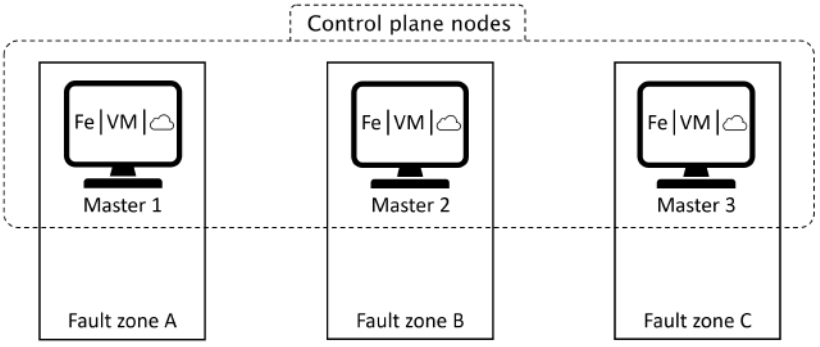


Figure 3.3

Control plane nodes run the following services that form the control plane (brains of the cluster):

- API Server
- Scheduler
- Store
- Cloud controller
- More…

The *API Server* is the **only** part of a Kubernetes cluster you directly interact with. For example, when you send commands to the cluster, they go to the API Server. When you receive responses, they come from the API server.

The *Scheduler* chooses which worker nodes to run user applications on.

The *Store* is where the state of the cluster and all applications is stored.

The *Cloud controller* allows Kubernetes to integrate with cloud services such as storage and load-balancers. The hands-on examples in later chapters integrate a cloud load-balancer with an app you'll deploy to a Kubernetes cluster.

There are more services in a Kubernetes control plane, but those are the important ones for this book.

Worker nodes

Worker nodes are where user applications run and can be Linux or Windows. A single cluster can have a mix of Linux and Windows worker nodes, and Linux apps will run on Linux worker nodes, whereas Windows apps will run on Windows worker nodes.

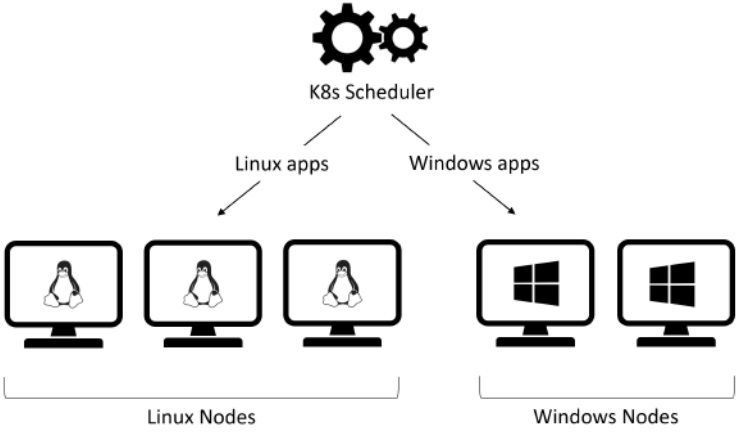


Figure 3.4

All worker nodes run a couple of services worth knowing about:

- Kubelet
- Container runtime

The `kubelet` is the main Kubernetes agent. It joins the worker nodes to the cluster and communicates with the control plane – things like receiving tasks and reporting on the status of tasks.

The *container runtime* starts and stops containers.

While container runtimes are low-level tools that are beyond the scope of this book, it's important to understand the following.

Docker was the original *container runtime* used by Kubernetes. However, the container runtime interface (CRI) was introduced to Kubernetes in 2016 and made this layer pluggable. As a result, there are a lot of different container

runtimes to choose from. Containerd (pronounced "container-dee") is a stripped-down version of Docker and the most popular container runtime on modern Kubernetes clusters. It fully supports container images created by Docker.

Hosted Kubernetes

Hosted Kubernetes is a consumption model where your cloud provider rents you a Kubernetes cluster. Sometimes we call it *Kubernetes as a service.*

As you'll see in later chapters, hosted Kubernetes is one of the simplest ways to get Kubernetes.

In the hosted model, the cloud provider builds the Kubernetes cluster, owns the control plane, and is responsible for all of the following:

- Control plane performance
- Control plane availability
- Control plane updates

You're usually responsible for:

- Worker nodes
- User applications
- Paying the bill

Figure 3.5 shows the basic architecture of hosted Kubernetes.

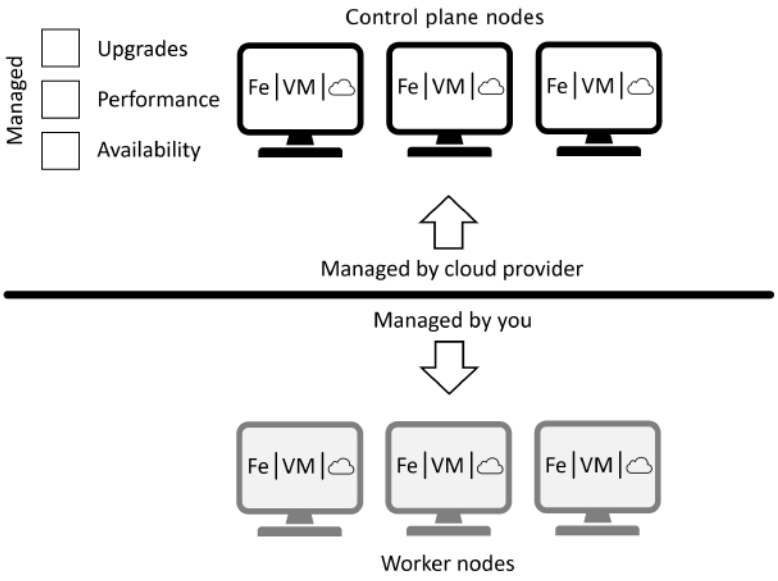


Figure 3.5

Most of the cloud providers have hosted Kubernetes services. Some of the more popular ones include:

- AWS: Elastic Kubernetes Service (EKS)
- Azure: Azure Kubernetes Service (AKS)
- Civo Cloud Kubernetes
- DO: Digital Ocean Kubernetes Service (DOKS)
- GCP: Google Kubernetes Engine (GKE)
- Linode: Linode Kubernetes Engine (LKE)

Others exist, and not all hosted Kubernetes services are equal. As a quick example, Linode Kubernetes Engine (LKE) is one of the simplest to configure and use. However, it lacks some of the features and configuration options offered by others. You should try a few before deciding which is best for you.

Managing Kubernetes with the `kubectl` command line tool

Most of the day-to-day management of a Kubernetes cluster can be done using the Kubernetes command line tool called `kubectl`. There are lots of ways to pronounce the name of the tool, but I pronounce it "kube see tee ell".

Management tasks include deploying and managing applications, checking the health of the cluster and applications, and performing updates to the cluster and applications.

You can get `kubectl` for Linux, Mac OS, Windows, and various ARM/Raspberry Pi related operating systems. You'll see how to install it in the next chapter.

The following `kubectl` command lists all nodes in a cluster. You'll run plenty of commands in the hands-on sections in later chapters.

```
$ kubectl get nodes
NAME                STATUS   ROLES                  AGE   VERSION
qsk-book-server-0   Ready    control-plane,etcd,    15s   v1.23.1
qsk-book-agent-2    Ready    <none>                 15s   v1.23.1
qsk-book-agent-0    Ready    <none>                 13s   v1.23.1
qsk-book-agent-1    Ready    <none>                 10s   v1.23.1
```

Chapter summary

In this chapter, you learned that a Kubernetes cluster comprises control plane nodes and worker nodes. These can run almost anywhere, including bare metal servers, virtual machines, and in the cloud. Control plane nodes run the back-end services that keep the cluster running, whereas worker nodes are where business applications run.

Most cloud platforms offer a hosted Kubernetes service that makes it easy to get a *"production-grade"* cluster where the cloud provider manages performance, availability, and updates. You manage the worker nodes and pay the bill.

You also learned that `kubectl` is the Kubernetes command line tool.

4: Getting Kubernetes

Kubernetes runs on everything from laptops and home Raspberry Pi clusters, all the way to high performance highly available clusters in the cloud.

In this chapter, you'll see the following easy ways to get Kubernetes on your laptop and in the cloud:

- Kubernetes on your laptop with Docker Desktop
- Kubernetes in the cloud with Linode Kubernetes Engine (LKE)

If you already have a working Kubernetes cluster, you may be able to use that.

Kubernetes on your laptop with Docker Desktop

Docker Desktop gets you a single-node Kubernetes cluster on your laptop that's great for development and learning. It includes the Kubernetes command line utility (`kubectl`) and a full set of Docker tools.

This set of tools means you can use Docker to build applications into container images that you can deploy to a Kubernetes cluster. Not bad for a free tool that's easy to download and use.

Install Docker Desktop

You can install Docker Desktop on most Windows 10, Windows 11, and Mac OS machines. You may also need a Docker account to use certain features of Docker Desktop and Docker Hub. Personal accounts are free and give access to enough features to follow all examples in the book. Go to docker.com and click `Sign in` to create a free account.

Type "download docker desktop" into your favorite search engine and follow the links to download the installer for your machine. Once downloaded, it's a

`next next next` installer that requires admin privileges. If the Windows installer prompts you to install WSL 2 components, say yes.

After the installation completes, you may need to manually start Docker Desktop.

Once Docker Desktop is running, you may have to manually start Kubernetes. Do this by clicking the whale icon (in the top menu bar on Mac OS or the system tray on Windows), choosing `Preferences` > `Kubernetes` and then selecting the `Enable Kubernetes` checkbox. See Figure 4.1.

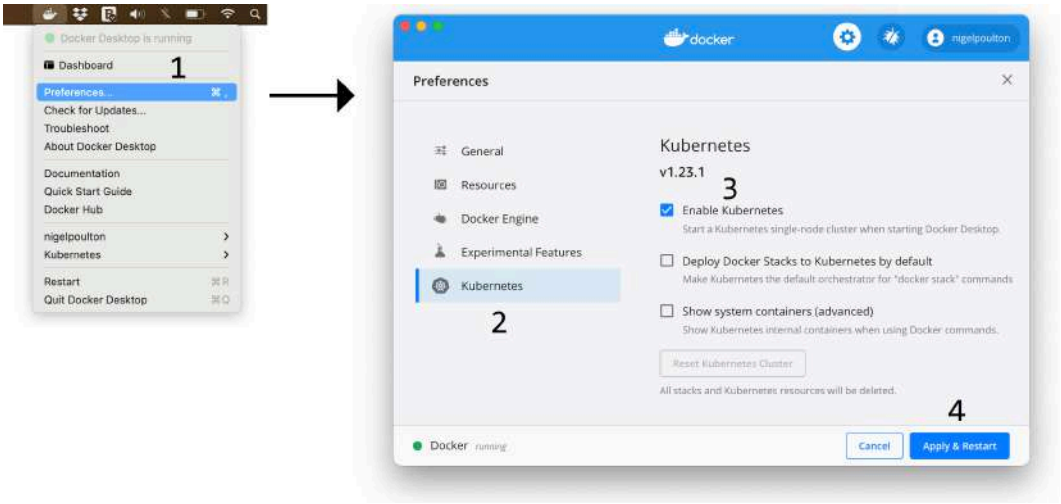


Figure 4.1

Windows users should switch Docker Desktop into ***Linux containers*** mode to follow along with the examples in the rest of the book. To do this, right-click the Docker whale in the system tray and choose `Switch to Linux containers`. This will enable your Windows machine to run Linux containers.

Run the following commands from a terminal to verify the installation worked:

```
$ docker --version
Docker version 20.10.11, build dea9396

$ kubectl version -o yaml
clientVersion:
  <Snip>
  gitVersion: v1.23.1
  major: "1"
  minor: "23"
  platform: darwin/amd64
serverVersion:
  <Snip>
  gitVersion: v1.23.1
  major: "1"
  minor: "23"
  platform: linux/amd64
```

The command output has been snipped to make it easier to read.

At this point, you have Docker and a single-node Kubernetes cluster running on your laptop and will be able to follow along with most of the examples in the book.

Kubernetes in the cloud with Linode Kubernetes Engine (LKE)

As you'd expect, you can run Kubernetes on every cloud, and most clouds have a Kubernetes-as-a-service offering. I've chosen to use Linode Kubernetes Engine (LKE) for the examples because it's outrageously simple, and it builds Kubernetes clusters fast. Feel free to use another cloud-based Kubernetes cluster, they should all work for the examples.

> **Note:** Cloud platforms often have offers for new customers. At the time of writing, Linode is offering new customers $100 of free credit that can be used in the first 60 days after signing up. This is more than enough to complete all the examples in the book.

What you get with Linode Kubernetes Engine (LKE)

Linode Kubernetes Engine (LKE) is a hosted Kubernetes offering from Linode. As such:

- It costs money (costs are clearly explained)
- It's easy to setup
- The control plane is managed by Linode and hidden from you
- It offers advanced integrations with other cloud services (storage, load-balancers etc.)

We'll show you how to build a Kubernetes cluster with two worker nodes. We'll also show you how to get and configure the Kubernetes command line utility (`kubectl`). Later in the book you'll see how to use Kubernetes to provision and utilise a Linode load balancer and integrate it with the sample app.

What you don't get with Linode Kubernetes Engine (LKE)

You don't get any Docker tools with LKE. If you want to follow all examples, you'll need Docker. The easiest way to get Docker is by following the Docker Desktop instructions from the previous sections.

Get an LKE cluster

Point your browser to `linode.com` and sign-up for an account. It's a simple process, but you will have to provide billing details. The costs are worth it if you're serious about learning Kubernetes. Costs are pretty low as long as you remember to delete your clusters when you're finished with them.

Once you're set-up and logged in to the Linode Cloud Console, click `Kubernetes` from the left navigation bar and choose `Create a Cluster`.

Choose a `cluster label` (name for your cluster), `Region`, and `Kubernetes Version`. Then add two `Linode 2GB Shared CPU` instances for your `Node Pool`. The configuration is shown in Figure 4.2.

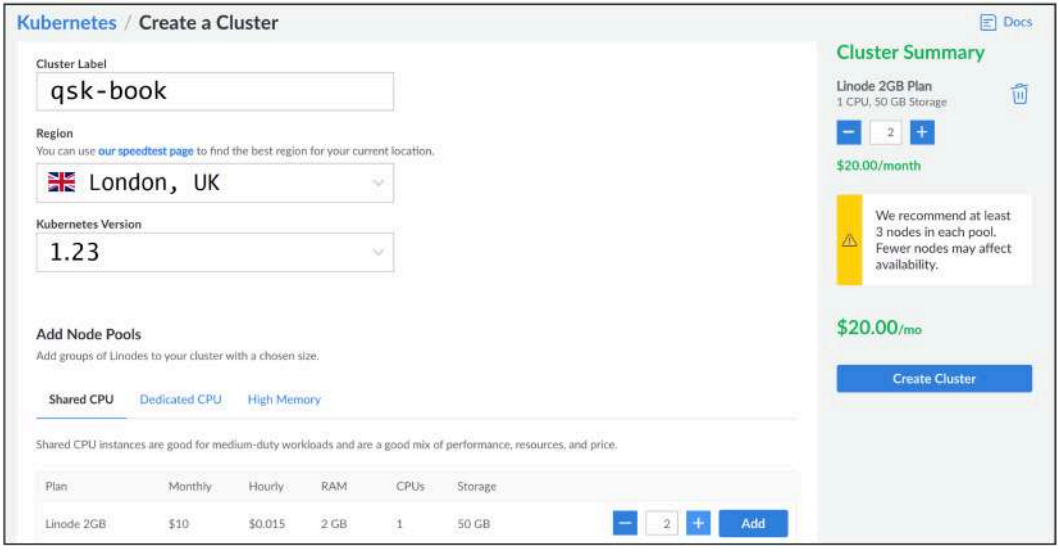


Figure 4.2

Take careful note of the potential costs displayed on the right. Your costs may differ from those shown in the book.

Click `Create Cluster` when you're sure of your configuration.

It may take a minute or two for your cluster to build.

When it's ready, the console will show your two nodes as `Running` and will display their IP addresses. It will also show your `Kubernetes API Endpoint` in URL format.

At this point, your LKE cluster is running with a high-performance highly-available control plane that is managed by Linode and hidden from you. It also has two running worker nodes. It's similar to the config shown in Figure 4.3.

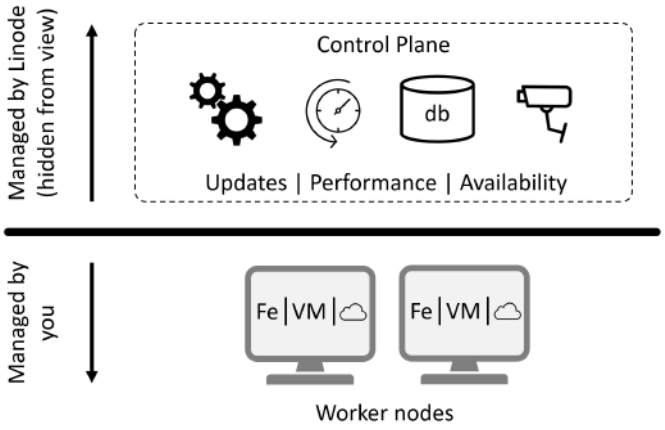


Figure 4.3

Now that you have a Kubernetes cluster, you need to install and configure `kubectl` so you can manage your cluster. If you have Docker Desktop, you'll already have `kubectl` and can skip to the section on configuring it.

If you haven't got Docker Desktop, you can install `kubectl` with any of the following methods (other ways to install it exist).

Install kubectl on Mac OS

Before following these steps, check if you already have it by typing `kubectl` at the command line.

The easiest way to install `kubectl` on Mac OS is using Homebrew.

```
$ brew install kubectl
<Snip>

$ kubectl version --client -o yaml
clientVersion:
  <snip>
  major: "1"
  minor: "23"
  platform: darwin/amd64
```

Install kubectl on Windows 10 and Windows 11

Before proceeding, type `kubectl` on a command line to make sure you don't already have it installed.

The easiest way to install `kubectl` on Windows 10 and Windows 11 is via Chocolatey. We show you how to install it using the PowerShell gallery in a later step in case you don't use Chocolatey.

```
> choco install kubernetes-cli

> kubectl version --client -o yaml
clientVersion:
  <Snip>
  major: "1"
  minor: "23"
  platform: windows/amd64
```

If you don't use Chocolatey, the following steps will install `kubectl` using standard PowerShell tools. Be sure to substitute the `-DownloadLocation` in the second command for a valid download location on your machine. The `-DownloadLocation` is where `kubectl` will be downloaded to and it should be in your system's `%PATH%`, or you should copy it to a folder in your system's `%PATH%`.

```
> Install-Script -Name 'install-kubectl' -Scope CurrentUser -Force

> install-kubectl.ps1 -DownloadLocation C:\Users\nigel\bin

> kubectl version --client -o yaml
clientVersion:
  <Snip>
  major: "1"
  minor: "23"
  platform: windows/amd64
```

If you receive a "command not found" error, make sure `kubectl` is present in a folder in your system's `%PATH%`.

`kubectl` is now installed and ready to be configured to talk to your Kubernetes cluster.

Configure kubectl to talk to your LKE cluster

`kubectl` has a config file that holds cluster information and credentials. On both Mac OS and Windows, it's called `config` and located in the following directories:

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

Even though the file is called `config` we call it the "kubeconfig" file.

The easiest way to configure `kubectl` to connect to your LKE cluster is to:

1. Make a backup copy of any existing kubeconfig file on your computer
2. Download and use the LKE kubeconfig file

For the following to work, you may have to configure your computer to show hidden folders. On Mac OS type `Command + Shift + period`. On Windows 10, type "folder" into the Windows search bar (next to the Windows flag home button) and select the `File Explorer Options` result. Select the `View` tab and click the `Show hidden files, folders, and drives` button.

Click the "Kubernetes" link in the left navigation bar of the Linode Cloud Console to display a list of your LKE clusters and click the `Download kubeconfig` link for your cluster. Locate the downloaded file, copy it to the hidden `./kube` folder in your home directory, and rename it to `config`. You'll have to rename any existing kubeconfig files before doing this.

Once you've downloaded your LKE kubeconfig and copied to the correct location and name, `kubectl` commands should work. You can test this with the following command.

```
$ kubectl get nodes
NAME                          STATUS   ROLES    AGE    VERSION
lke47224-75467-61c34614cbae   Ready    <none>   2m49s  v1.23.1
lke47224-75467-61c34615230e   Ready    <none>   100s   v1.23.1
```

The output shows an LKE cluster with two worker nodes. You know the cluster is on LKE because the node names start with `lke`. Control plane nodes are not displayed as they are managed by LKE and hidden from view.

At this point, your LKE cluster is up and running and you can use it to follow the examples in the book.

Remember that LKE is a cloud service and costs money. Be sure to delete it when you no longer need it. Forgetting to do this will incur unwanted costs.

Chapter summary

Docker Desktop is a great way to get the Docker tools and a Kubernetes cluster on your Windows or Mac OS computer. It's free to download and use, and automatically installs and configures `kubectl`. It's not intended for production use, but you can use it to follow along with the examples later in the book.

Linode Kubernetes Engine (LKE) is a simple-to-use hosted Kubernetes service. Linode manages the control plane and lets you size and spec as many worker nodes as you need. It requires you to manually update your local `kubeconfig` file. It also costs money to run an LKE cluster, so you should size it appropriately and remember to delete it when you're finished.

There are lots of other ways and places to get Kubernetes, but the ways we've shown here are enough to get you started and ready for the upcoming examples.

5: Creating a containerized app

In this chapter, you'll complete a typical workflow to build an application into a container image. This process is called *containerization* and the resulting app is called a *containerized app.*

You'll use Docker to containerize the app, and the steps are not specific to Kubernetes. In fact, you won't use Kubernetes in this chapter. However, you'll deploy the containerized app to Kubernetes in the following chapters.

> **Docker and Kubernetes:** There's been a lot of talk in the industry about Kubernetes dropping support for Docker, so let's clear things up. Kubernetes is in the process of dropping support for Docker as a container runtime. This means future versions of Kubernetes won't use Docker to start and stop containers. However, container images created by Docker are still 100% supported in Kubernetes. This is because Kubernetes and Docker both work with container images based on the Open Container Initiative (OCI) standards. In summary, even when Kubernetes no longer supports Docker as a container runtime, images created by Docker will still work.

You can skip this chapter if you're already familiar with Docker and creating containerized apps – a pre-created containerized app is available on Docker Hub that you can use in future chapters.

Still here?

Great. The workflow you'll follow is shown in Figure 5.1. We'll touch on step 1, but the main focus will be on steps 2 and 3. Future chapters will cover step 4.

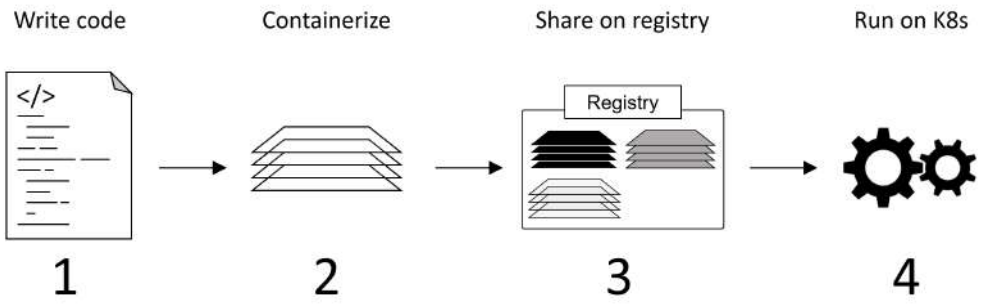


Figure 5.1

The chapter is divided as follows:

- Pre-requisites
- Get the application code
- Containerize the app
- Host the container image on a registry

Pre-requisites

To create the containerized app described in this chapter, you'll need the following:

- The `git` command line tool
- Docker
- A Docker account

The next section will show you how to install `git`.

For Docker, I recommend you follow the steps in Chapter 3 to install *Docker Desktop*.

Docker accounts are free, and you only need one if you want to save the containerized app to Docker Hub in later steps. If you choose not to do this step, there is a pre-created image you can use instead. However, if you're serious about learning Docker or Kubernetes, a Docker account is very useful. You can sign-up at `docker.com`.

Install git

Use any of the following methods to install the `git` command line tool.

Mac OS using Homebrew

If you have Homebrew on your Mac, you can use the following commands to install `git` and verify the installation.

```
$ brew install git

$ git --version
git version 2.34.1
```

Windows using Chocolatey

If you have Chocolatey on your Windows machine you can install the `git` command line tool with the following command.

```
> choco install git.install

> git --version
git version 2.34.1
```

Mac OS or Windows using GitHub Desktop installer

GitHub Desktop is a desktop UI for working with GitHub and has Mac OS and Windows installers at `desktop.github.com`. Once downloaded and installed, you can use it to install the `git` command line tool.

Verify your installation with the `git --version` command.

With the pre-requisites taken care of, you're ready to complete the following steps to build a sample application into a container image (containerize the app):

1. Get the application code
2. Use Docker to build the container image
3. Use Docker to push the image to Docker Hub (optional)

Get the application code

The book's GitHub repo contains the code for a simple web application. You need to download the app files to your local machine so you can build the app into a container image.

Run the following command to create a new folder in your current directory and copy the contents of the repo to it.

```
$ git clone https://github.com/nigelpoulton/qsk-book.git
Cloning into 'qsk-book'...
```

> **Note:** GitHub is an online platform for hosting and collaborating on software. The software hosted on GitHub is organized in "repos" (repositories), and the act of "cloning a repo" is technical jargon for making a copy of the software on your local machine.

You now have a copy of the repo in a new folder called `qsk-book`. Change into this directory and run an `ls` command to list its contents.

```
$ cd qsk-book

$ ls
App
deploy.yml
pod.yml
readme.md
rolling-update.yml
svc-cloud.yml
svc-local.yml
```

The `App` folder is where the application source code and config files are. Change directory into it and list the files it contains.

```
$ cd App

$ ls -l
Dockerfile
app.js
bootstrap.css
package.json
views
```

These files make up the application and it's good to know a bit about each file.

- `Dockerfile`. This file isn't part of the application. It contains a list of instructions Docker executes to build the app into a container image (containerize the app)
- `app.js` is the main application file. It's a Node.js app
- `bootstrap.css` is a stylesheet template that determines how the application's web page looks
- `package.json` lists the application's dependencies
- `views` is a folder containing the HTML to populate the app's web page

The file of most interest to us in containerizing the app is the `Dockerfile`. It contains instructions Docker uses to build the app into a container image. Ours is simple and looks like this.

```
FROM node:current-slim
LABEL MAINTAINER=nigelpoulton@hotmail.com
COPY . /src
RUN cd /src; npm install
EXPOSE 8080
CMD cd /src && node ./app.js
```

Let's step through and see what each line does.

The `FROM` instruction tells Docker which version of Linux we want the app to run on. In this example, we're telling Docker to run the app on the version of Linux

contained in the `node:current-slim` image. Applications need an OS to run on, and this base image provides that OS.

The `COPY` instruction tells Docker to copy the application and dependencies from the current directory (signified by the period ".") into the `/src` directory in the `node:current-slim` image pulled in the previous step. This will copy all files in the same directory as the `Dockerfile` into the container image.

The `RUN` instruction tells Docker to run an `npm install` command from within the `/src` directory. This will install the dependencies listed in `package.json`.

The `EXPOSE` instruction lists the network port the app will listen on. This is also specified in the main `app.json` file.

The `CMD` instruction is the main application process that will run when Kubernetes starts the container.

In summary, the Dockerfile tells Docker to base the app on the `node:current-slim` image, copy in our app code, install dependencies, document the network port, and set the app to run._

Once you've cloned the repo it's time to build the app into a container image.

Build the container image

The process of building an application into a container image is called *containerization*. When the process is complete, the app is said to be *containerized*. As a result, we'll be using the terms *container image* and *containerized app* interchangeably.

Use the following `docker image build` command to containerize the app. A few quick things to note.

- The command reads the `Dockerfile` for instructions how to build the image
- The command has to run from the directory with the `Dockerfile`
- Substitute `nigelpoulton` with your own Docker account ID
- Include the period (".") at the end of the command

```
$ docker image build -t nigelpoulton/qsk-book:1.0 .

[+] Building 66.9s (7/7) FINISHED                        0.1s
<Snip>
=> naming to docker.io/nigelpoulton/qsk-book:1.0         0.0s
```

You now have a new container image on your local machine containing the app and its dependencies. This is the containerized app.

Us the following command to list the image. The name of your image might be different, and the output may display more than one image.

```
$ docker image ls
REPOSITORY                TAG     IMAGE ID       CREATED          SIZE
nigelpoulton/qsk-book     1.0     bab4706d517c   19 seconds ago   255MB
```

If you're running Docker Desktop you may see multiple images labelled "`k8s.gcr...`". These are needed to run the local Kubernetes cluster.

Now that you've successfully *containerized* the application and dependencies into an image on your local machine, the next step is to host it in a centralized registry.

Host the image on a registry

This section is optional, and you'll need a Docker account if you want to follow along. If you don't complete this section, you can use the publicly available `nigelpoulton/qsk-book:1.0` image in later steps.

Container registries are centralised places where you store container images. Some, like the one we'll use, are hosted on the public internet, whereas as others can be hosted on your own private network. However, no matter where they're hosted, they're places to store container images so they're secured and easily accessible.

There are many container registries available. However, we'll use Docker Hub as it's the most popular and easiest to use. Feel free to visit `hub.docker.com` and have a look around.

Use the following command to push your new image to Docker Hub. Remember to substitute `nigelpoulton` with your own Docker account ID. The operation will fail if you use `nigelpoulton` as you don't have permission to push images to my repositories.

```
$ docker image push nigelpoulton/qsk-book:1.0

The push refers to repository [docker.io/nigelpoulton/qsk-book]
8efe8541d82c: Pushed
fcc4707bc9ec: Pushed
480780c1297d: Mounted from library/node
8aee3140ddb6: Mounted from library/node
e30d94d2a229: Mounted from library/node
197720a4c240: Mounted from library/node
2edcec3590a4: Mounted from library/node
1.0: digest: sha256:899e55614414a91a...abdbe54c61e size: 1787
```

Go to `hub.docker.com` and make sure the image is present. Remember to browse your own repos.

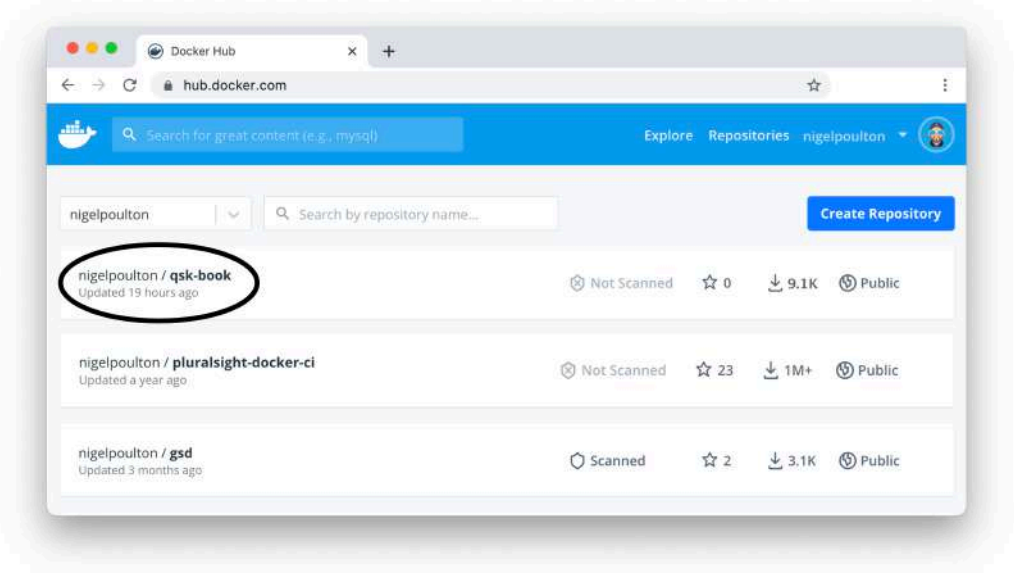


Figure 5.2

At this point, you've containerized the application into a container image and pushed it to the Docker Hub registry. You're now ready to deploy it to Kubernetes.

Chapter summary

In this chapter, you learned that a *containerized app* is just a regular app that's built and packaged as a container image.

You used `git` to clone the book's GitHub repo into a local folder on your computer and then used Docker to containerize the app and push it to Docker Hub. Along the way, you learned that a Dockerfile is a list of instructions that tell Docker how to containerize an app.

6: Running an app on Kubernetes

In this chapter, you'll deploy a simple containerized application to a Kubernetes cluster. If you've been following along, it'll be the app you created and containerized in the previous chapter. If you skipped the previous chapter, no sweat, you can use the publicly available copy of the app on Docker Hub.

You'll need a Kubernetes cluster to follow along. See Chapter 3 if you need help. If you're using Docker Desktop on Windows, you should be running in ***Linux containers*** mode (right-click the Docker whale in the system tray and choose `Switch to Linux containers`).

This is how we'll go about things:

- Verify your Kubernetes cluster
- Deploy the app to your Kubernetes cluster
- Connect to the app

Verify your Kubernetes cluster

You need the `kubectl` command line utility and a working Kubernetes cluster to follow these steps.

Run the following command to verify you're connected to your Kubernetes cluster and your cluster is operational.

Docker Desktop example.

```
$ kubectl get nodes
NAME             STATUS   ROLES           AGE   VERSION
docker-desktop   Ready    control-plane   33m   v1.23.1
```

Notice how the Docker Desktop cluster only returns a single node on one line of output. This is because it's a single-node cluster. In this configuration, the single

node acts as a *control plane node* and a *worker node*. Aside from that, the important points are that `kubectl` can communicate with your cluster and all nodes show as `Ready`.

Linode Kubernetes Engine (LKE) example.

```
$ kubectl get nodes
NAME                          STATUS   ROLES    AGE   VERSION
lke47224-75467-61c34614cbae   Ready    <none>   94m   v1.23.1
lke47224-75467-61c34615230e   Ready    <none>   93m   v1.23.1
```

The number of nodes returned by the command will depend on how many you added to your cluster. Hosted Kubernetes platforms, such as LKE, do not return *control plane nodes* as they're managed by the cloud platform and hidden from view. You can be sure you're communicating with an LKE cluster because the names begin with `lke`. All nodes should be in the `Ready` state.

If `kubectl` connects to the wrong cluster/nodes and you're running Docker Desktop, you can click the Docker whale icon and select the correct cluster as shown in Figure 6.1.

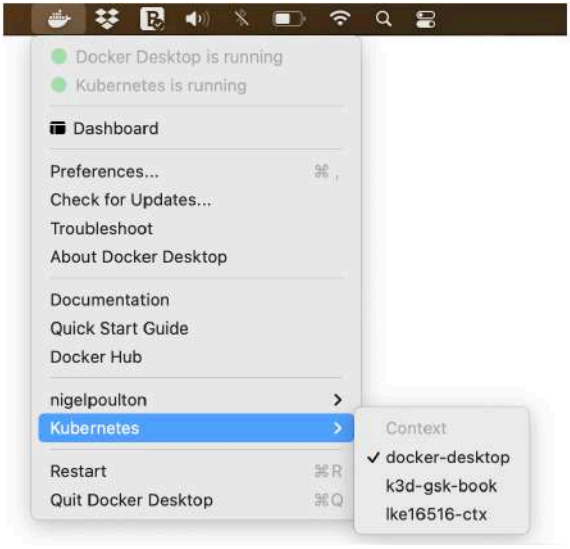


Figure 6.1

If you're not using Docker Desktop and `kubectl` is connecting to the wrong cluster, you can change it with the following procedure.

List all contexts defined in your kubeconfig file.

```
$ kubectl config get-contexts
CURRENT   NAME             CLUSTER          AUTHINFO
          docker-desktop   docker-desktop   docker-desktop
          k3d-qsk-book     k3d-qsk-book     admin@k3d-qsk-book
*         lke16516-ctx     lke16516         lke16516-admin
```

The output show lists three contexts, and the current context set to `lke16516-ctx`. Your output will be different.

The following command switches to the `docker-desktop` context. You may need to change to a different context.

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

As long as the `kubectl get nodes` command returns the correct nodes and lists them as `Ready`, you're ready to proceed with the next section.

Deploy the app to Kubernetes

In the previous chapter, you containerized a Node.js web application into a container image and stored it on Docker Hub. You're about to deploy that application to your cluster inside a Kubernetes Pod.

Although Kubernetes orchestrates and runs *containers*, these containers have to be wrapped in a Kubernetes construct called a *Pod.*

Just think of a Pod as a lightweight wrapper around a container. In fact, we sometimes use the terms *container* and *Pod* interchangeably. For now, you just need to know that Kubernetes runs containers inside of Pods.

A Kubernetes Pod definition

The Pod you'll deploy is defined in a YAML file called `pod.yml` located in the root of the book's GitHub repo. You can give the file any name you like, but the contents of the file follow strict YAML rules. If you don't already know, YAML is a language commonly used for configuration files. Oh, and it's painfully strict about proper use of indentation.

```
apiVersion: v1
kind: Pod
metadata:
  name: first-pod
  labels:
    project: qsk-book
spec:
  containers:
    - name: web
      image: nigelpoulton/qsk-book:1.0
      ports:
        - containerPort: 8080
```

The role of the Pod is to encapsulate a container so it can run on Kubernetes. If you look closely at the YAML file, you'll see the last 5 lines define the container image you created in the previous chapter.

Let's step through the file and understand what it's defining.

The `apiVersion` and `kind` lines tell Kubernetes the type and version of object being deployed. In this case, we're deploying a Pod object as defined in the `v1` API. That's a lot of jargon that is basically telling Kubernetes to deploy a Pod based on version 1 (`v1`) of the Pod specification.

The `metadata` block lists the Pod name and a single label. The name helps us identify and manage the Pod when it's running. The label (`project = qsk-book`) is useful for organising Pods and associating them with other objects such as load balancers. We'll see labels in action later.

The `spec` section lists details of the container this Pod will execute. Remember to change to your own image if you followed the examples in the previous chapter

and pushed your own image to your own repository. If you didn't push an image to your own repo, just leave the file as it is and use the `nigelpoulton/qsk-book:1.0` image.

Figure 6.2 shows how the Pod is wrapping the container. Remember, this Pod wrapper is mandatory for containers to run on Kubernetes and is very lightweight by only adding metadata.

Figure 6.2

Deploy the app (Pod)

The app you'll deploy is in a Pod called `first-pod` and defined in a YAML file called `pod.yml`. The simplest way to deploy it is using `kubectl` to post the YAML file to Kubernetes.

Run the following command to list any Pods that might already be running on your cluster. If you're working with a new cluster, as explained in Chapter 3, you won't have any running Pods.

```
$ kubectl get pods
No resources found in default namespace.
```

Deploy the Pod with the following command and verify the operation. The first command must be executed from the same directory as the `pod.yml` file. This is the root directory of the GitHub repo. If you're currently in the `App` directory (check with `pwd`) you'll need to back up one directory level with the "`cd ..`" command.

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

$ kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
first-pod   1/1     Running   0          10s
```

Congratulations, the containerized app is running inside a Pod on a Kubernetes cluster!

The `kubectl apply` command lets you specify a file (`-f`) to send to the Kubernetes API Server. Kubernetes will read the file and store the configuration in the cluster store. The scheduler will then find a worker node to run the Pod.

If you run the second command too soon after the first, the Pod might not have reached the `Running` state.

`kubectl` provides the `get` and `describe` commands to query the configuration and state of objects. You've already seen that `kubectl get` provides summary info. The following example shows that `kubectl describe` returns a lot more detail. In fact, I've trimmed the output as some of you complain if I take up too much space with long command outputs ;-)

```
$ kubectl describe pod first-pod

Name:         first-pod
Namespace:    default
Node:         docker-desktop/192.168.65.3
Labels:       project=qsk-book
Status:       Running
IPs:
  IP:  10.1.0.11
Containers:
  web-ctr:
    Container ID:   containerd://a1ec2a8b7180e1...
    Image:          nigelpoulton/qsk-book:1.0
    Port:           8080/TCP
    State:          Running
```

```
    <Snip>
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  <Snip>
  Normal  Created    110s  kubelet            Created container web-ctr
  Normal  Started    110s  kubelet            Started container web-ctr
```

Although the Pod is up and the application is running, you need another Kubernetes object to provide network connectivity to it.

Connect to the app

Connecting to the app in the Pod requires a separate object called a Service. The whole reason for Service objects is to provide stable network connectivity to apps running in Pods.

> **Note:** "Object" is a technical term used to describe something running on Kubernetes. You've already deployed a Pod *object.* You're about to deploy a Service *object* to provide connectivity to the app running in the Pod.

A Kubernetes Service definition

The `svc-local.yml` file defines a Service object to provide connectivity if you're running on a Docker Desktop or other non-cloud local cluster. The `svc-cloud.yml` file defines a Service object to provide connectivity if your cluster is in the cloud (use this if you're running on an LKE cluster as described in Chapter 3).

The following listing shows the contents of the `svc-cloud.yml` file.

```
apiVersion: v1
kind: Service
metadata:
  name: cloud-lb
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 8080
  selector:
    project: qsk-book
```

Let's step through it.

The first two lines are similar to the `pod.yml` file. They tell Kubernetes to deploy a Service object using the `v1` specification.

The `metadata` section names the Service "cloud-lb".

The `spec` section is where the magic happens. The `spec.type: LoadBalancer` field tells Kubernetes to provision an internet-facing load-balancer on the underlying cloud platform. For example, if your cluster is running on AWS, this Service will automatically provision an AWS Network Load Balancer (NLB) or Classic Load Balancer (CLB). This `spec` section will configure an internet-facing load-balancer on the underlying cloud that will accept traffic on port `80` and forward on port `8080` to any Pods with the `project: qsk-book` label.

Give that a second to sink in and read through it again if it's confusing.

The `svc-local.yml` file defines a NodePort Service instead of a LoadBalancer Service. This is because Docker Desktop and other laptop-based clusters do not have access to internet-facing load-balancers.

A quick word about labels

You might remember a while back we said Kubernetes uses *labels* to associate objects. Well, look closely at the `pod.yml` and `svc-cloud.yml` files and notice how they both reference the `project: qsk-book` label.

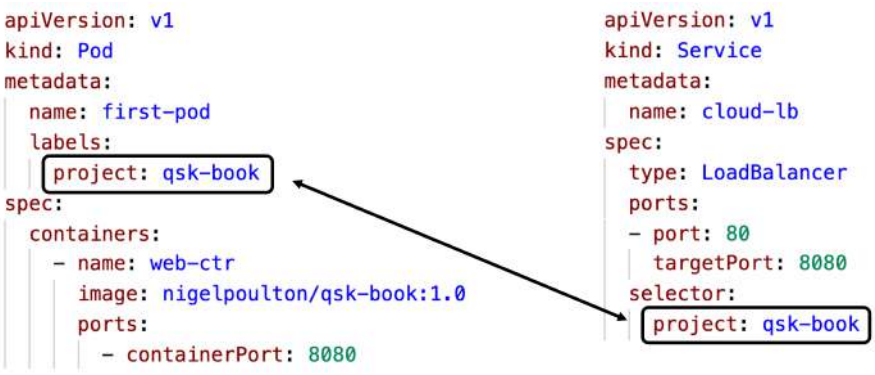


Figure 6.3

The Pod carries the label, whereas the Service object uses it to make selections. This combination allows the Service to forward traffic to all Pods on the cluster carrying the label. Kubernetes is also clever enough to keep an up-to-date list of all Pods carrying the label, updated in real time.

Currently, you only have one Pod carrying the label. However, if you add more Pods with the same label, Kubernetes will observe this and start forwarding traffic to them all. You'll see this in action in the next chapter.

Deploy the Service

As with Pods, you can deploy Service objects with `kubectl apply`.

As mentioned previously, the GitHub repo has two Services:

- `svc-cloud.yml` is for use on cloud-based clusters. We'll call this one the "cloud Service"
- `svc-local.yml` is for use on local clusters, such as Docker Desktop, that do not have access to load-balancers. We'll call this one the "local Service"

The *cloud Service* tells Kubernetes to provision one of your cloud's internet-facing load-balancers. It works with all the major clouds and is a simple way to expose your application to the internet.

The *local Service* exposes the application via every node in the cluster on a common network port. The example we're using will expose the application on port `31111` on every cluster node. If you're using Docker Desktop, this will expose

the application via the `localhost` adapter on the host machine you have Docker Desktop installed on. Don't worry if this sounds confusing, we'll walk through an example and explain it.

We'll look at the Docker Desktop (non-cloud) example first. Skip to the next section if you're using a cluster in in the cloud.

Connecting to the app if your cluster is not in the cloud, such as Docker Desktop

The following command deploys a Service called `svc-local` as defined in the `svc-local.yml` file in the root of the GitHub repo. The name of the Service and the file do not have to match, but you must run the command from the directory where the `svc-local.yml` file is located.

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

Use the following command to verify the Service is up and running.

```
$ kubectl get svc
NAME         TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
svc-local    NodePort   10.108.72.184   <none>        80:31111/TCP   11s
```

The output shows the following.

The Service is called "svc-local" and has been running for 11 seconds.

The `CLUSTER-IP` value is an IP address on the internal Kubernetes Pod network and is used by other Pods and applications running on the cluster. We won't be connecting to that address as we're not connected to the Kubernetes Pod network.

As this is a `NodePort` Service, it can be accessed by connecting to any cluster node on port `31111` as specified in the `PORT(S)` column.

Your output will list another Service called `Kubernetes`. This is an internal Service used for service discovery.

Now that the Service is running, you can use it to connect to the app.

Open a web browser on the same machine as your Kubernetes cluster and type `localhost:31111` into the navigation bar. If you're using Docker Desktop, you should open a browser on the machine running Docker Desktop.

> **Warning!** Some older versions of Docker Desktop on Mac OS had a bug that prevented the NodePort being mapped to the `localhost` adapter. If your browser doesn't connect to the app, this may be why.

The web page will look like Figure 6.4

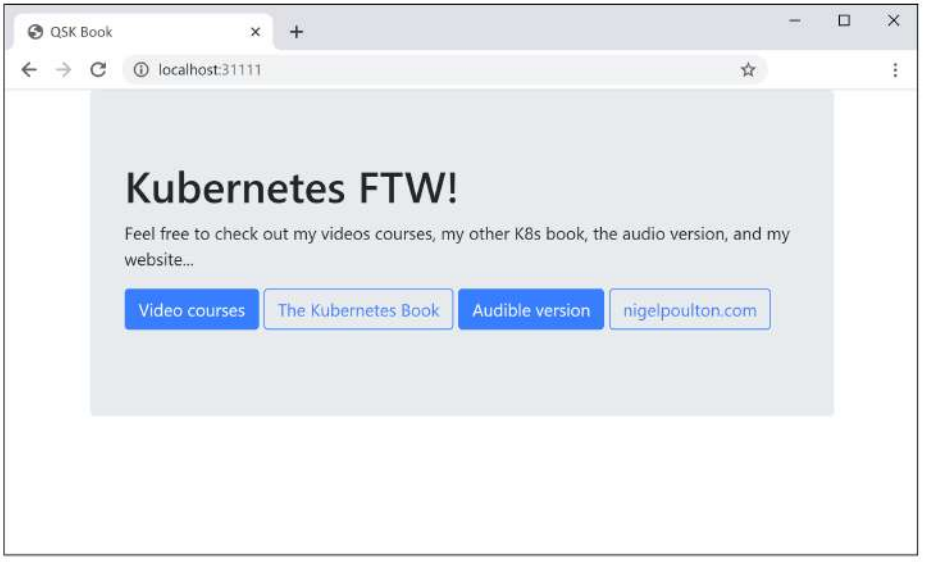


Figure 6.4

Congratulations, you've containerized an app, deployed it to Kubernetes, and connected to it via a Kubernetes Service.

Connecting to the app if your Kubernetes cluster is in the cloud

The following command deploys a load-balancer Service called `cloud-lb` as defined in the `svc-cloud.yml` file in the root of the GitHub repo. You must execute the command from within the same directory as the file.

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

Verify the Service with the following command. You can also run a `kubectl describe svc <service-name>` command to get more detailed info.

```
$ kubectl get svc
NAME       TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)
cloud-lb   LoadBalancer   10.128.29.224   212.71.236.112   80:30956/TCP
```

Your output may show `<pending>` in the `EXTERNAL-IP` column while your cloud provisions an internet-facing load-balancer. This can take a few minutes on some cloud platforms.

The output shows a lot, so let's explain the bits we're interested in.

The Service was created and the `TYPE` was correctly set to `LoadBalancer`. One of the underlying cloud's internet-facing load-balancers has been provisioned and assigned the IP address `212.71.236.112` as shown in the `EXTERNAL-IP` column (yours will be different). The load-balancer is listening on port `80` (the "80" part of the `80:30956/TCP` string).

To cut a long story short, you can point any browser to `212.71.236.112` on port `80` to connect to the app as shown in Figure 6.5. Remember to substitute the external IP address from your environment.

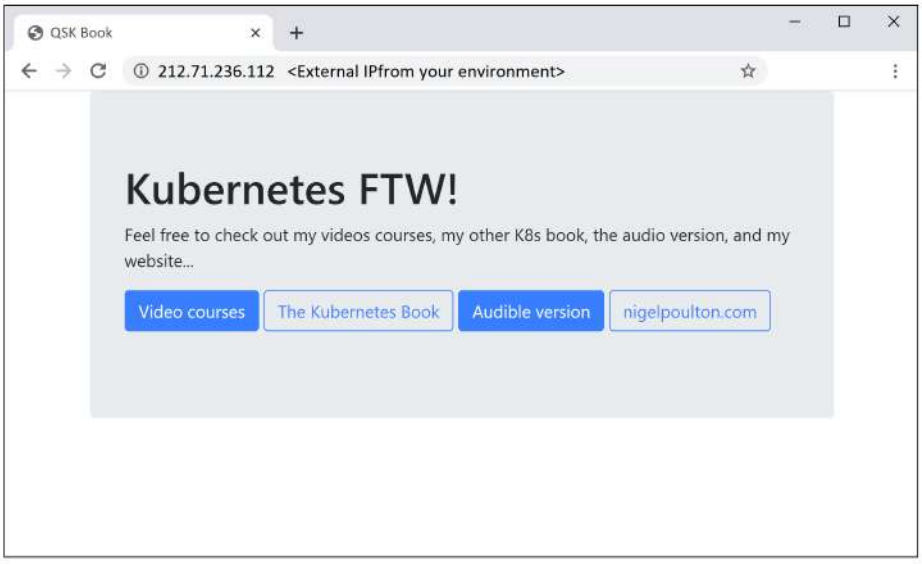


Figure 6.5

As with the local Docker Desktop example, the internal `CLUSTER-IP` is for use by other applications running inside the Kubernetes cluster, and the value to the right of the colon in the `PORT(S)` column is the port the app is exposed on via each cluster node. For example, if you know the IP addresses of your cluster nodes, you can connect to the app by connecting to any node's IP on the port listed to the right of the colon.

Congratulations, you've containerized an app, deployed it to Kubernetes, provisioned an internet-facing load-balancer, and connected to the app.

Clean-up

Let's delete the Pod and Service so you've got a clean cluster at the start of the next chapter.

List all Services on your cluster to get the name of the Service you deployed.

```
$ kubectl get svc
NAME       TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)
cloud-lb   LoadBalancer   10.128.29.224   212.71.236.112   80:30956/TCP
...
```

Run the following commands to delete the Service and Pod. It may take a few seconds for the Pod to terminate while it waits for the app to gracefully shutdown. Be sure to use the name of the Service from your environment.

```
$ kubectl delete svc cloud-lb
service "cloud-lb" deleted

$ kubectl delete pod first-pod
pod "first-pod" deleted
```

Chapter summary

In this chapter, you learned that containerized apps have to run inside of Pods if they want to run on Kubernetes. Fortunately, Pods are lightweight constructs and add no overhead to applications.

You saw a simple Pod defined in a YAML file and learned how to deploy it to Kubernetes with `kubectl apply`. You also learned how to inspect Pods, and other Kubernetes objects, with `kubectl get` and `kubectl describe`.

Finally, you learned that you need a Kubernetes Service if you want to connect to apps running in Pods.

So far, you've built, deployed, and connected to a containerized app. However, you've not seen self-healing, scaling, or any other cloud-native features that Kubernetes provides. You'll do all of that in the upcoming chapters.

7: Adding self-healing

In this chapter, you'll learn about the Kubernetes Deployment object and use it to make your app resilient and demonstrate self-healing.

The chapter is organised as follows:

- Intro to Kubernetes Deployments
- Self-heal from a Pod failure
- Self-heal from a worker node failure

Intro to Kubernetes Deployments

In Chapter 6, you learned that Kubernetes uses a dedicated *Service* object to provide network connectivity to apps running in Pods. It has another dedicated object, called a *Deployment*, to provide *self-healing.* In fact, Deployments also enable scaling and rolling updates.

As with Pods and Service objects, Deployments are defined in YAML manifest files.

Figure 7.1 shows a Deployment manifest. It's marked up to show how a container is nested in a Pod, and a Pod is nested in a Deployment.

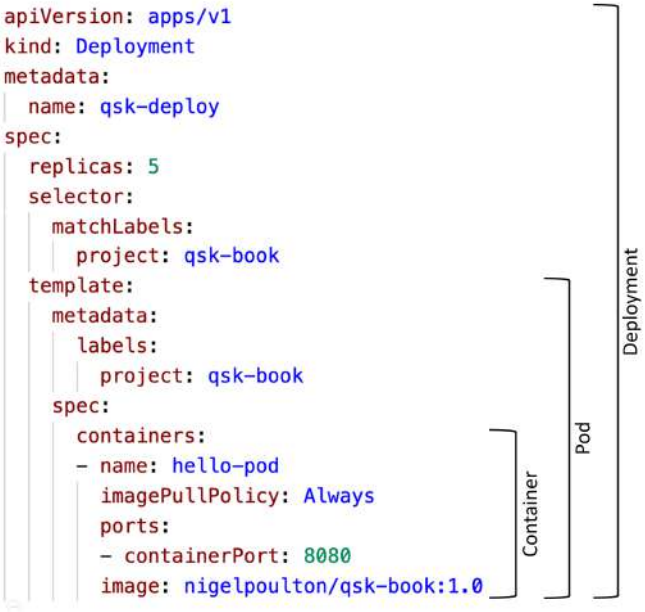


Figure 7.1

This nesting, or wrapping, is important in understanding how everything works.

- The container provides the OS and other app dependencies
- The Pod provides metadata and other constructs for the container to run on Kubernetes
- The Deployment provides cloud-native features, including self-healing

How Deployments work

There are two important elements to the working of Deployments.

1. The Deployment object
2. The Deployment controller

The *Deployment object* is the YAML configuration that defines the Pod and container. It also defines things such as how many Pod replicas to deploy.

The *Deployment controller* is a control plane process that is constantly monitoring the cluster making sure all Deployment objects are running as they are supposed to.

Consider a very quick example.

You define an application in a Kubernetes Deployment manifest. It defines 5 replicas of a Pod called `zephyr-one`. You use `kubectl` to send this to Kubernetes and Kubernetes schedules the 5 Pods on the cluster.

At this point, *observed state* matches *desired state*. That's jargon meaning the cluster is running what you asked it to run. But let's say a node fails and the number of `zephyr-one` Pods drops to 4. *Observed state* no longer matches *desired state* and you have a problem.

But don't stress. The Deployment controller is watching the cluster and will see the change. It knows you desire 5 Pods, but it can only observe 4. So, it'll start a 5th Pod to bring *observed state* back into sync with *desired state*. The technical term for this process is *reconciliation*, but we often call it self-healing.

Let's see it in action.

Self-heal from a Pod failure

In this section, you'll use a Kubernetes Deployment to deploy 5 replicas of a Pod. After that, you'll manually delete a Pod and see Kubernetes self-heal.

You'll use the `deploy.yml` manifest in the root of the GitHub repo. As seen in the following snippet, it defines 5 Pod replicas running the app you containerized in previous chapters. The YAML shown is annotated to help you understand it.

```
kind: Deployment                    <<== Type of object being defined
apiVersion: apps/v1                 <<== Version of object to deploy
metadata:
  name: qsk-deploy
spec:
  replicas: 5                       <<== How many Pod replicas to deploy
  selector:
    matchLabels:                    <<== Tells the Deployment controller
      project: qsk-book             <<== which Pods to manage
  template:
    metadata:
      labels:
        project: qsk-book           <<== Pod label
    spec:
      containers:
      - name: qsk-pod
        imagePullPolicy: Always           <<== Never use local images
        ports:
        - containerPort: 8080             <<== Network port
        image: nigelpoulton/qsk-book:1.0  <<== Container image to use
```

> **Terminology:** The terms *Pod*, *instance*, and *replica* are used to mean the same thing – an instance of a Pod running a containerized app. I normally use "replica".

Check for any Pods and Deployments already running on your cluster.

```
$ kubectl get pods
No resources found in default namespace.

$ kubectl get deployments
No resources found in default namespace.
```

Now use `kubectl` to deploy the Deployment to your cluster. Run the command from the same folder as the `deploy.yml` file.

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

Check the status of the Deployment and Pods it's managing.

```
$ kubectl get deployments
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
qsk-deploy   5/5     5            5           4m

$ kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
qsk-deploy-5c...pxhd     1/1     Running   0          48s
qsk-deploy-5c...4269     1/1     Running   0          48s
qsk-deploy-5c...dj7n     1/1     Running   0          48s
qsk-deploy-5c...s446     1/1     Running   0          48s
qsk-deploy-5c...rq8h     1/1     Running   0          48s
```

You can see that 5 out of 5 replicas are running and ready. The Deployment controller is also running on the control plane observing the state of things.

Pod failure

It's possible for Pods and the apps they're running to crash or fail. Kubernetes can *attempt* to self-heal a situation like this by starting a new Pod to replace the failed one.

Use `kubectl delete pod` to manually delete one of the Pods. Remember to use a Pod name from your environment.

```
$ kubectl delete pod qsk-deploy-5c4cd6db76-4pxhd
pod "qsk-deploy-5c4cd6db76-4pxhd" deleted
```

As soon as the Pod is deleted, the number of Pods on the cluster will drop to 4 and no longer match the *desired state* of 5. The Deployment controller will notice this and automatically start a new Pod to take the observed number of Pods back to 5.

List the Pods again to see if a new Pod has been started.

```
$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
qsk-deploy-5c4cd6db76-b4269   1/1     Running   0          15m
qsk-deploy-5c4cd6db76-cdj7n   1/1     Running   0          15m
qsk-deploy-5c4cd6db76-gs446   1/1     Running   0          15m
qsk-deploy-5c4cd6db76-krq8h   1/1     Running   0          15m
qsk-deploy-5c4cd6db76-qr46b   1/1     Running   0          33s
```

Notice how the last Pod in the list has only been running for 33 seconds. This is the replacement Pod Kubernetes started to reconcile desired state.

Congratulations. There are 5 Pods running and Kubernetes performed the self-healing without needing help from you.

Let's see how Kubernetes deals with the failure of a worker node.

Self-heal from a worker node failure

When a worker node fails, any Pods running on it are lost. If those Pods are managed by a controller such as a Deployment, replacement Pods will be started on surviving worker nodes in the cluster.

If your cluster is on a cloud that implements *node pools*, the failed worker node may also be replaced. This is not a feature of Kubernetes Deployments – Deployments only watch and manage Pods.

You can only follow the steps in this section if you have a multi-node cluster and the ability to delete worker nodes. If you built a multi-node cluster on Linode Kubernetes Engine, as explained in Chapter 3, you can follow along. If you're using a single-node Docker Desktop cluster, you'll have to be satisfied with reading along.

The following command lists all Pods on your cluster and the worker node each Pod is running on. The command output has been trimmed to fit the book.

```
$ kubectl get pods -o wide
NAME          READY   STATUS    <Snip>   NODE
qsk...b4269   1/1     Running   ...      lke...98
qsk...cdj7n   1/1     Running   ...      lke...98
qsk...gs446   1/1     Running   ...      lke...1a
qsk...krq8h   1/1     Running   ...      lke...1a
qsk...qr46b   1/1     Running   ...      lke...1a
```

See how both worker nodes are running multiple Pods. The next step will delete a worker node and take any Pods with it. The example will delete the `lke...98` worker node.

The following process shows how to delete a worker node on Linode Kubernetes Engine (LKE). Deleting it this way simulates sudden node failure. The process will be different if you're running on a different cloud.

1. View your LKE cluster in the Linode Cloud Console
2. Scroll down to `Node Pools`
3. Click one of your nodes to drill into it
4. Click the three dots and delete the node as shown in Figure 7.2

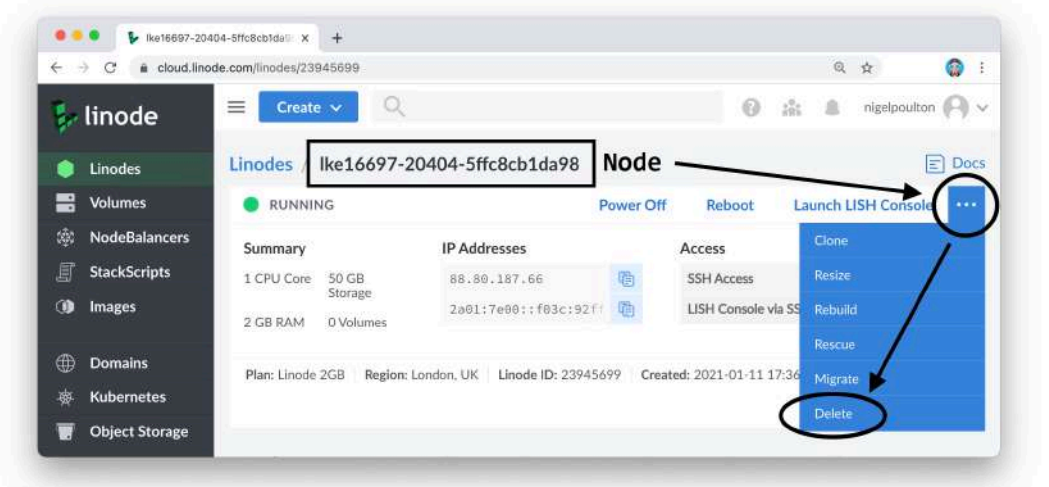


Figure 7.2

Verify the node has been deleted. If you wait too long to run this command, LKE will replace the deleted node. It can take a minute or two for the missing node to show in the command output.

```
$ kubectl get nodes
NAME                STATUS   ROLES        AGE    VERSION
lke...1a            Ready    <none>       3d1h   v1.23.1
lke...98            NotReady              3d1h   v1.23.1
```

Once Kubernetes sees the worker node is `NotReady` it will also notice the missing Pods and create replacements. Verify this. It may take a few seconds for the replacement Pods to reach the Running state.

```
$ kubectl get pods
NAME          READY   STATUS              <Snip>   NODE
qsk...b4269   1/1     Running             ...      lke...1a
qsk...cdj7n   1/1     Running             ...      lke...1a
qsk...gs446   1/1     Running             ...      lke...1a
qsk...6bqmk   0/1     ContainerCreating   ...      lke...1a
qsk...ps9nt   0/1     ContainerCreating   ...      lke...1a

<short time lapse>

$ kubectl get deployments
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
qsk-deploy   5/5     5            5           46m
```

The output shows that Kubernetes has created two new Pods to replace the two that were lost when the `lke...98` node was deleted. All new Pods were scheduled to `lke...1a` as it was the only surviving worker node in the cluster.

After a few more minutes LKE will replace the deleted node and return the cluster to 2 nodes. This is a feature of LKE and not the Kubernetes Deployment object. It works because LKE's implementation of *node pools* has the notion of *desired state*. When the cluster was created, you requested two worker nodes. When one was deleted, LKE noticed the change in state and added a new one to bring observed state back into sync with desired state.

Although your cluster is back to two worker nodes, Kubernetes will not re-balance existing Pods across them. As a result, you'll have a two-node cluster and all 5 Pods running on a single node.

Chapter summary

In this chapter, you learned that Kubernetes has an object called a Deployment that implements several cloud-native features. You learned there is a Deployment controller running on the control plane making sure the current observed state of the cluster matches what you asked for.

You also saw how Deployments wrap a Pod spec, which in turn wraps a container, which in turn hosts an app and dependencies.

You used `kubectl` to deploy an app via a Deployment object and tested self-healing. You manually deleted a Pod, as well as a worker node, and watched Kubernetes replace any lost Pods.

Linode Kubernetes Engine also replaced the deleted/broken worker node. This is not a feature of Kubernetes Deployments, and other cloud platforms also support self-healing of nodes that are part of a *node pool.*

8: Scaling the app

In this chapter, you'll use a couple of methods to scale the app up and down. The methods you'll use are *manual* and require a human to implement them. In the real world, Kubernetes has a separate object, called a *Horizontal Pod Autoscaler (HPA)*, for automatic scaling. However, that's beyond the scope of a quick start book.

The unit of scaling is the Pod. Therefore, scaling up will add more Pod replicas, whereas scaling down will delete Pod replicas.

The chapter is split as follows.

- Pre-requisites
- Manual scale up
- Manual scale down

Pre-requisites

If you've been following along, you'll have a Kubernetes cluster running a single Deployment that is currently managing 5 replicas of a simple containerized web app. If you already have this configuration, you can skip straight to the `Scale an application up` section.

If you haven't been following along, run the following command to deploy 5 replicas of the containerized app to your cluster. Be sure to run the command from the same directory as the `deploy.yml` file.

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

Run a `kubectl get deployments` command to make sure the application is running.

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

As soon as all 5 replicas are up and running you can move to the next section.

Scale an application up

In this section you'll manually edit the Deployment YAML file, increase the number of replicas to 10, and re-send it to Kubernetes.

Check the current number of replicas.

```
$ kubectl get deployment qsk-deploy
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
qsk-deploy   5/5     5            5           16h
```

Use your favorite editor to edit the `deploy.yml` file and set the `spec.replicas` field to 10 and **save your changes**.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: qsk-deploy
spec:
  replicas: 5            <<== Change this to 10
  selector:
    matchLabels:
      project: qsk-book
<Snip>
```

Use `kubectl` to re-send the updated file to Kubernetes. When Kubernetes receives the file, it'll change *desired state* from 5 replicas to 10. The Deployment controller will observe 5 replicas on the cluster and notice it doesn't match the new desired

state of 10. It will then schedule 5 new replicas to bring observed state in line with desired state.

Be sure you've saved your changes.

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

Run a couple of commands to check the status of the Deployment and the number of Pods being managed.

```
$ kubectl get deployment qsk-deploy
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
qsk-deploy   10/10   10           10          16h

$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
qsk-deploy-5c4cd6db76-4lsdq   1/1     Running   0          28s
qsk-deploy-5c4cd6db76-7z6bz   1/1     Running   0          28s
qsk-deploy-5c4cd6db76-g6xr4   1/1     Running   0          28s
qsk-deploy-5c4cd6db76-jwkch   1/1     Running   0          16h
qsk-deploy-5c4cd6db76-lzp44   1/1     Running   0          16h
qsk-deploy-5c4cd6db76-qv9bg   1/1     Running   0          28s
qsk-deploy-5c4cd6db76-v86c7   1/1     Running   0          28s
qsk-deploy-5c4cd6db76-v9j7f   1/1     Running   0          16h
qsk-deploy-5c4cd6db76-vbxqb   1/1     Running   0          16h
qsk-deploy-5c4cd6db76-zpvsg   1/1     Running   0          16h
```

It may take a few seconds for the 5 new Pods to start, but you can easily identify them based on their age.

If you've been following the examples from the previous chapter, the 5 new Pods will probably all be scheduled on the new node. This is because Kubernetes is intelligent enough to schedule the new Pods so that all 10 are balanced across the available worker nodes in the cluster.

Congratulations. You've manually scaled the application from 5 replicas to 10. The method you used is called the *declarative method* because you *declared* a new desired state in a YAML manifest file and used that file to update the cluster.

Scale an application down

In this section, you'll use the `kubectl scale` command to scale the number of Pods back down to 5. The method you'll use is called as the *imperative method* and is not as common as the *declarative method* where you make all config changes by updating YAML files and re-posting them to Kubernetes.

Run the following command.

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

Check the number of Pods. As always, it can take a few seconds for deleted Pods to terminate and the state of the cluster to settle.

```
$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
qsk-deploy-5c4cd6db76-4lsdq   1/1     Running   0          4m11s
qsk-deploy-5c4cd6db76-7z6bz   1/1     Running   0          4m11s
qsk-deploy-5c4cd6db76-g6xr4   1/1     Running   0          4m11s
qsk-deploy-5c4cd6db76-qv9bg   1/1     Running   0          4m11s
qsk-deploy-5c4cd6db76-v9j7f   1/1     Running   0          16h
```

Congratulations. You've manually scaled the application back down to 5 replicas.

Another word on labels

We've already said that Kubernetes Service objects use a combination of labels and label selectors to target traffic at particular Pods on the cluster. Well, when Kubernetes scales an application up, it adds more Pods with the same label. This ensures the newly added Pods will receive traffic from the Service. Also, when scaling down a Deployment by removing Pods, Kubernetes observes the deleted Pods and the Service stops sending traffic to them.

Important clean-up

Performing scaling operations imperatively with the `kubectl scale` command can be dangerous.

If you've been following along, you'll have 5 replicas running on your cluster. However, the `deploy.yml` configuration file still defines 10. If, at a later date, you edit the `deploy.yml` file to specify an updated version of the container image and re-send to Kubernetes, you'll also increase the number of replicas back up to 10. This might not be what you want. You should be very careful of this in the real world, as it can cause major issues.

With this in mind, it's considered a good practice to only perform updates *declaratively* by updating your YAML files and re-sending them to Kubernetes.

Edit the `deploy.yml` file and set the number of replicas back to 5 and save your changes. It now matches what is deployed to your cluster.

Chapter summary

In this chapter, you learned how to manually scale a Deployment by editing its YAML file. You also saw that it's possible to perform scaling operations with the `kubectl scale` command, but this is not the recommended method.

You saw Kubernetes attempt to balance new Pods over all cluster nodes. You also learned that Kubernetes has an object that can *automatically* scale the number of Pods based on demand.

9: Performing a rolling update

In this chapter, you'll perform a *zero-downtime rolling update* against the app deployed in previous chapters. If you're unsure what a rolling update is, great! You're about to find out.

We'll divide this chapter as follows.

- Pre-requisites
- Update the app

All the steps in this chapter can be completed on the Docker Desktop and Linode Kubernetes Engine (LKE) clusters that Chapter 3 showed you how to build. You can also use other Kubernetes clusters.

Pre-requisites

If you've been following along, you'll have everything in place to complete this chapter. If that's you, skip to the next section.

If you haven't been following along, follow these steps to setup your lab.

1. Get a Kubernetes cluster and configure `kubectl` (see Chapter 3)
2. Clone the book's GitHub repo (see Chapter 5)
3. Deploy the sample app and Service with the following command

The following commands need to be executed from the cloned folder containing the YAML files.

Docker Desktop/local cluster example

```
$ kubectl apply -f deploy.yml -f svc-local.yml
deployment.apps/qsk-deploy created
service/svc-local created
```

Linode Kubernetes Engine (LKE)/cloud cluster example

```
$ kubectl apply -f deploy.yml -f svc-cloud.yml
deployment.apps/qsk-deploy created
service/cloud-lb created
```

Run a `kubectl get deployments` and `kubectl get svc` command to make sure the application and Service is running.

```
$ kubectl get deployments
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
qsk-deploy   5/5     5            5           4m

NAME        TYPE       CLUSTER-IP     EXTERNAL-IP  PORT(S)          AGE
svc-local   NodePort   10.128.97.167  <none>       8080:31111/TCP   4m
```

It may take a minute for all 5 Pods to be ready, but as soon as they are you can proceed to the next section.

Update the app

The app is running with 5 replicas. You can verify this with `kubectl get deployments`.

You'll configure a rolling update that forces Kubernetes to update one replica at-a-time in a methodical manner until all 5 replicas are running the new version. Kubernetes offers a lot of options to control how updates happen, but we'll keep it simple and let you explore more advanced options in your own time.

There are various basic steps required to push an update to an app. Some have been done for you, whereas others you will complete.

The following have already been done for you.

1. Writing a new version of the app
2. Building a new container image for the new version
3. Pushing the new version to a container registry

You'll complete the following steps.

1. Edit the `deploy.yml` file to specify the new image version and configure update settings
2. Re-send the YAML file to Kubernetes
3. Observe the update process
4. Test the new version of the app

Edit the Deployment YAML file

Open the `deploy.yml` file and change the last line (26) to reference version 1.1 of the image. If you've been using your own image that you pushed to a registry, you'll have to change the whole line to include my repo (`nigelpoulton`).

Also, add the 6 new lines (10-15) as shown in the following listing.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: qsk-deploy
5 spec:
6   replicas: 5
7   selector:
8     matchLabels:
9       project: qsk-book
10  minReadySeconds: 20        <<== Add this line
11  strategy:                  <<== Add this line
12    type: RollingUpdate      <<== Add this line
13    rollingUpdate:           <<== Add this line
14      maxSurge: 1            <<== Add this line
15      maxUnavailable: 0      <<== Add this line
```

```
16  template:
17    metadata:
18      labels:
19        project: qsk-book
20    spec:
21      containers:
22      - name: hello-pod
23        imagePullPolicy: Always
24        ports:
25        - containerPort: 8080
26        image: nigelpoulton/qsk-book:1.1    <<== Set to 1.1
```

We'll explain what the new lines do in a moment. For now, a couple of points about making the updates.

YAML is obsessed about proper indentation. So, be sure you've indented each new line the correct number of ***spaces***. Also, the file uses spaces and **not tabs** for indentation. You cannot mix-and-match tabs and spaces in the same file, so you **must use spaces and not tabs**.

Kubernetes is also strict about use of *camelCase* and *PascalCase*. Be sure you use the correct casing for all text.

If you have issues editing the file, there's a pre-completed version in the GitHub repo called `rolling-update.yml`. You can use this instead.

Be sure to save your changes.

Understand the update settings

The next step is to send the updated file to Kubernetes. But let's explain what those lines you added will do.

```
10  minReadySeconds: 20
11  strategy:
12    type: RollingUpdate
13    rollingUpdate:
14      maxSurge: 1
15      maxUnavailable: 0
```

`minReadySeconds` on line 10 tells Kubernetes to wait for 20 seconds after updating each replica. So, Kubernetes will update the first replica, wait 20 seconds, update the second replica, wait 20 seconds, update the third, rinse and repeat…

Inserting waits like this gives you a chance to run tests and make sure the new replicas are working as expected. In the real world you'll probably wait longer than 20 seconds between replica updates.

Also, Kubernetes is not actually *updating* replicas. It's deleting the existing replicas and replacing them with a new ones running the new version.

Lines 11 and 12 force Kubernetes to perform all updates to this Deployment as *rolling updates*.

Lines 14 and 15 force Kubernetes to update one Pod at a time. It works like this…

`maxSurge=1` on line 14 allows Kubernetes to add one extra Pod during an update operation. Desired state requests 5 Pods, so Kubernetes is allowed to increase that to 6 during the update. `maxUnavailable=0` on line 15 prevents Kubernetes from reducing the number of Pods during an update. Again, desired state is 5 Pods, so Kubernetes isn't allowed to go lower than this. When combined, lines 14 and 15 force Kubernetes to add a 6th replica with the new version, then delete a replica running the old version. This process is repeated until all 5 Pods are running the desired version.

Perform the rolling update

Make sure you've saved the changes and send the updated configuration to Kubernetes.

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

Kubernetes will now start replacing the Pods, one at-a-time, with a 20 second wait between each.

Monitor and check the rolling update

You can monitor the progress of the job with the following command. The output has been trimmed to fit the page.

```
$ kubectl rollout status deployment qsk-deploy
Waiting for rollout to finish: 1 out of 5 have been updated...
Waiting for rollout to finish: 1 out of 5 have been updated...
Waiting for rollout to finish: 2 out of 5 have been updated...
Waiting for rollout to finish: 2 out of 5 have been updated...
Waiting for rollout to finish: 3 out of 5 have been updated...
Waiting for rollout to finish: 3 out of 5 have been updated...
Waiting for rollout to finish: 4 out of 5 have been updated...
Waiting for rollout to finish: 4 out of 5 have been updated...
Waiting for rollout to finish: 2 old replicas are pending termination...
Waiting for rollout to finish: 1 old replicas are pending termination...
deployment "qsk-deploy" successfully rolled out
```

You can also point your web browser at the app and keep refreshing the page. Some of your requests might return the original version of the app and others might return the new version. Once all 5 replicas are up to date, all requests will return the new version.

Figure 9.1

Congratulations. You've performed a successful rolling update of an application.

Clean-up

The following commands show how to delete the Deployment and Service from your cluster.

Be sure to use the correct Service name for your cluster.

```
$ kubectl delete deployment qsk-deploy
deployment.apps "qsk-deploy" deleted
```

If you're running a local cluster such as Docker Desktop, run the following command to delete the local Service.

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

If your cluster is in the cloud, run the following command to delete the cloud load-balancer Service.

```
$ kubectl delete svc cloud-lb
service "cloud-lb" deleted
```

If your cluster is in the cloud, **be sure to delete it when you no longer need it**. Failure to do this will incur unwanted costs.

Chapter summary

In this chapter, you learned how to perform a rolling update of an application deployed via a Kubernetes Deployment object.

You edited the Deployment YAML file and added instructions to control how the rolling update flowed. You also updated the version of the application image and sent the updated configuration to Kubernetes. Finally, you monitored and verified the operation.

10: What next

Congratulations on finishing the book, I hope you loved it!

If you read it all, and followed the examples, you've learned the basics and you're ready to take your next steps.

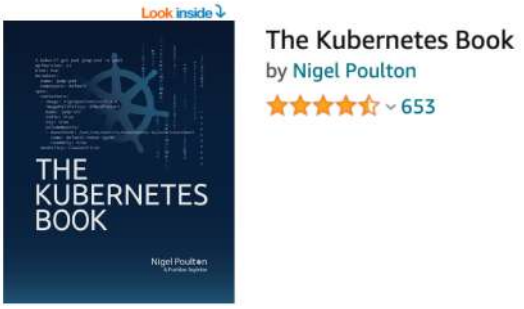
Here are some quick suggestions for next steps. And yes, I'm recommending a bunch of my own stuff. But here's the truth…

- If you liked this, you'll love my other stuff
- I'm super-busy and don't get a chance to read and test other people's stuff

Of course, if you didn't like this, I'm gutted. But that's life, and you probably won't like my other stuff either. If that's you, I'd love you to drop me an email at `qskbook@nigelpoulton.com` and tell me what you didn't like.

Other books

My other book, The Kubernetes Book, is regularly listed as a best-seller on Amazon and has the most Amazon star-ratings of any book on Kubernetes. It's written in the same style as this book but covers a lot more and goes into a lot more detail. For example, this book is ~15K words whereas The Kubernetes Book is over 75K words – a totally different beast. It's also updated annually, so if you buy it, you know you're getting the latest and greatest. It's available in e-book, paperback, hardback, a special large-print paperback edition, and an audio version is available (apparently it's a really good listen).



There's also a Klingon tribute edition and a Borg collector's edition!

The front cover of the Klingon edition is red, the name of the book is written in the Klingon language, and it has a special intro section. The rest of the book is written in English and is identical to the English language edition. The Borg collector's edition is a hardback with a green cover with the name of the book written in the Borg language. It also has a special intro section, but the contents of the book are written in English and identical to the English language edition. If you're into Star Trek, you'll love them both.

Video courses

I'm a huge fan of video courses. It's so much easier to explain things and have fun.

I recommend the following, and both platforms usually have offers where you can try before you buy.

1. Docker and Kubernetes: The Big Picture (pluralsight.com)

2. Containers on AWS Wavelength
3. Getting Started with Docker (pluralsight.com)
4. Getting Started with Kubernetes (pluralsight.com)
5. Kubernetes Deep Dive (acloud.guru)

You can see a list of all my video courses at `nigelpoulton.com/video-courses`. If you like learning new technologies, I highly recommend my "Containers on AWS Wavelength" course as it's super short and covers the converging worlds of cloud, containers, and 5G – three technologies that are changing the world.

Events

I'm a huge fan of community events. I prefer in-person events, but we've had some decent live-stream events in the last year or two.

My favourite in-person event is KubeCon, and I highly recommend you attend if you can. You'll meet great people and learn a lot form the sessions.

I also recommend local community meetups. Just google any of the following to find one local to you. You'll need to temporarily disable any VPN or other browser privacy tools for those searches to work ;-)

- "Kubernetes meetup near me"
- "Cloud native meetup near me"

Let's connect

I'm all about tech and I love connecting with you. While I can't be free tech support, I'm more than happy to help if you get stuck with the basics. Please don't be afraid to reach out and connect, I'm a nice guy :-D

- twitter.com/nigelpoulton
- https://www.linkedin.com/in/nigelpoulton/

Show some love

I'd love it if you'd leave the book a review and give it some stars on Amazon. You can sometimes leave an Amazon review if you got the book from somewhere else!

Have fun with Kubernetes!

Appendix A: Lab code

This appendix contains all the lab exercises from the book, in order. It assumes you've got a Kubernetes cluster, installed Docker, installed Git, and configured `kubectl` to talk to your cluster.

I've included this to make it easier to run through the labs for extra hands-on practice. It's also useful if you're trying to remember a particular command or example but can't remember which chapter it's from.

Chapter 5: Creating a containerized app

Clone the book's GitHub repo.

```
$ git clone https://github.com/nigelpoulton/qsk-book.git
Cloning into 'qsk-book'...
```

Change into the `qsk-book/App` directory and run an `ls` command to list its contents.

```
$ cd qsk-book/App
```

```
$ ls
Dockerfile   app.js   bootstrap.css
package.json   views
```

Run the following command to build the application into a container image. You must run the command from within the `App` directory. If you have a Docker Hub account, make sure you use your own Docker account ID.

```
$ docker image build -t nigelpoulton/qsk-book:1.0 .

[+] Building 66.9s (7/7) FINISHED                    0.1s
<Snip>
=> naming to docker.io/nigelpoulton/qsk-book:1.0     0.0s
```

Verify the newly created image is present on your local machine.

```
$ docker image ls
REPOSITORY               TAG    IMAGE ID       CREATED          SIZE
nigelpoulton/qsk-book    1.0    076860faac6c   21 hours ago     255MB
```

Push the image to Docker Hub. This step will only work if you have a Docker account. Remember to substitute your Docker account ID.

```
$ docker image push nigelpoulton/qsk-book:1.0

f4576e76ed1: Pushed
ca60f24a8154: Pushed
0dcc3a6346bc: Mounted from library/node
6f2e5c7a8f99: Mounted from library/node
6752c6e5a2a1: Mounted from library/node
79c320b5a45c: Mounted from library/node
e4b1e8d0745b: Mounted from library/node
1.0: digest: sha256:7c593...7198f1 size: 1787
```

Chapter 6: Running an app on Kubernetes

List the Nodes in your K8s cluster.

```
$ kubectl get nodes
NAME                          STATUS   ROLES    AGE   VERSION
lke16405-20053-5ff63e4400b7   Ready    <none>   55m   v1.23.1
lke16405-20053-5ff63e446413   Ready    <none>   55m   v1.23.1
```

The following command needs to run from the root of the GitHub repo. If you're currently in the `App` directory, you'll need to run the "cd .." command to back up one level.

Deploy the application defined in `pod.yml`.

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

Check the Pod is running.

```
$ kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
first-pod   1/1     Running   0          8s
```

Get detailed info about the running Pod. The output has been snipped.

```
$ kubectl describe pod first-pod

Name:         first-pod
Namespace:    default
Node:         docker-desktop/192.168.65.3
Labels:       project=qsk-book
Status:       Running
IPs:
  IP:  10.1.0.11
<Snip>
```

Deploy the Service. Use `svc-local.yml` if you're running a cluster on your laptop. Use `svc-cloud.yml` if your cluster's in the cloud.

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

Check the external IP (public IP) of the Service. Your Service will only have an external IP if it's running on a cloud.

```
$ kubectl get svc
NAME       TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)
cloud-lb   LoadBalancer   10.128.29.224   212.71.236.112   80:30956/TCP
```

Point your browser to the IP from the `EXTERAL-IP` column. If you're running your cluster on your laptop, point your browser to `localhost:31111`. For detailed information, see Chapter 6.

Run the following commands to delete the Pod and Service.

```
$ kubectl delete pod first-pod
pod "first-pod" deleted
```

If your cluster is in the cloud with the cloud load-balancer Service, run this command to delete it.

```
$ kubectl delete svc cloud-lb
service "cloud-lb" deleted
```

If your cluster is on your local laptop with the NodePort Service, run this command to delete it.

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

Chapter 7: Adding self-healing

Run the following command to deploy the application specified in `deploy.yml`. This will deploy the app with 5 Pod replicas.

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

Check the status of the Deployment and Pods it is managing.

```
$ kubectl get deployments
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
qsk-deploy   5/5     5            5           4m

$ kubectl get pods
NAME                     READY   STATUS      RESTARTS   AGE
qsk-deploy-6999...wv8    0/1     Running     0          4m
qsk-deploy-6999...9nl    0/1     Running     0          4m
qsk-deploy-6999...g8t    0/1     Running     0          4m
qsk-deploy-6999...xp7    0/1     Running     0          4m
qsk-deploy-6999...l7f    0/1     Running     0          4m
```

Delete one of the Pods. Your Pods will have different names.

```
$ kubectl delete pod qsk-deploy-69996c4549-r59nl
pod "qsk-deploy-69996c4549-r59nl" deleted
```

List the Pods to see the new Pod Kubernetes automatically started.

```
$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
qsk-deploy-69996c4549-mwl7f   1/1     Running   0          20m
qsk-deploy-69996c4549-9xwv8   1/1     Running   0          20m
qsk-deploy-69996c4549-ksg8t   1/1     Running   0          20m
qsk-deploy-69996c4549-qmxp7   1/1     Running   0          20m
qsk-deploy-69996c4549-hd5pn   1/1     Running   0          5s
```

The new Pod is the one that's been running for less time than the others.

Chapter 8: Scaling an app

Edit the `deploy.yml` file and change the number of replicas from 5 to 10. **Save your changes**.

Re-send the Deployment to Kubernetes.

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

Check the status of the Deployment.

```
$ kubectl get deployment qsk-deploy
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
qsk-deploy   10/10   10           10          4h43m
```

Scale the app down with `kubectl scale`.

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

Check the number of Pods.

```
$ kubectl get pods
qsk-deploy-bbc5cf95d-58r44   1/1     Running   0          4h55m
qsk-deploy-bbc5cf95d-6bqmk   1/1     Running   0          4h37m
qsk-deploy-bbc5cf95d-npk4c   1/1     Running   0          4h55m
qsk-deploy-bbc5cf95d-plcj2   1/1     Running   0          4h55m
qsk-deploy-bbc5cf95d-ps9nt   1/1     Running   0          4h37m
```

Edit the `deploy.yml` file and set the number of replicas back to 5 and **save your changes**.

Chapter 9: Performing a rolling update

Edit the `deploy.yml` file and change the image version from `1.0` to `1.1`.

Add the following lines in the `spec` section. See `rolling-update.yml` for reference.

```
minReadySeconds: 20
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1
    maxUnavailable: 0
```

Save your changes.

Send the updated YAML file to Kubernetes.

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

Check the status of the rolling update.

```
$ kubectl rollout status deployment qsk-deploy
Waiting to finish: 1 out of 5 new replicas have been updated...
Waiting to finish: 1 out of 5 new replicas have been updated...
Waiting to finish: 2 out of 5 new replicas have been updated...
<Snip>
```

The following commands will clean-up by deleting the Deployment and Service objects.

```
$ kubectl delete deployment qsk-deploy
deployment.apps "qsk-deploy" deleted
```

If your cluster is in the cloud with the cloud load-balancer Service, run this command to delete it.

```
$ kubectl delete svc cloud-lb
service "cloud-lb" deleted
```

If your cluster is on your local laptop with the NodePort Service, run this command to delete it.

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

If your Kubernetes cluster is running in the cloud, remember to delete it when you're done.

Terminology

This glossary defines some of the most common Kubernetes-related terms used in the book. I've only included terms used in the book. For a more comprehensive coverage of Kubernetes, see ***The Kubernetes Book***.

Ping me if you think I've missed anything important:

- qskbook@nigelpoulton.com
- https://nigelpoulton.com/contact-us
- https://twitter.com/nigelpoulton
- https://www.linkedin.com/in/nigelpoulton/

As always… I know that some folks are passionate about their own definitions of technical terms. I'm OK with that, and I'm not saying my definitions are better than anyone else's – they're just here to be helpful.

Term	Definition (according to Nigel)
API Server	Part of the K8s control plane and runs on control-plan nodes. All communication with Kubernetes goes through the API Server. `kubectl` commands and responses go through the API Server.
Container	An application and dependencies packaged to run on Docker or Kubernetes. As well as application stuff, every container is a virtual operating system with its own process tree, filesystem, shared memory, and more.
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 rollbacks. They're usually microservices apps and run on Kubernetes.
Container runtime	Low-level software running on every Kubernetes worker node responsible for pulling container images and starting and stopping containers. The most famous container runtime is Docker, however, **containerd** is becoming the most popular container runtime used by Kubernetes.

Term	Definition (according to Nigel)
Controller	Control plane process running as a reconciliation loop monitoring the cluster and making the necessary changes so the observed state of the cluster matches desired state.
Control plane node	Cluster node running control plane services. The brains of a Kubernetes cluster. You should deploy 3 or 5 for high availability.
Cluster store	Control plane feature that holds the state of the cluster and apps.
Deployment	Controller that deploys and manages a set of stateless Pods. Performs rolling updates and rollbacks and can self-heal.
Desired state	What the cluster and apps should be like. For example, the *desired state* of an application might be 5 replicas of xyz container listening on port 8080/tcp.
K8s	Shorthand way to write Kubernetes. 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 and updates 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.
Label	Metadata applied to objects for grouping. For example, Services send traffic to Pods based on matching labels.
Manifest file	YAML file that holds the configuration of one or more Kubernetes objects. For example, a Service manifest file is typically a YAML file that holds the configuration of a Service object. When you post a manifest file to the API Server, its configuration is deployed to the cluster.

Term	Definition (according to Nigel)
Microservices	A design pattern for modern applications. Application features are broken into their own small applications (microservices/containers) and communicate via APIs. They work together to form a useful application.
Node	Also known as worker node. The nodes in a cluster that run user applications. Must run the kubelet process and a container runtime.
Observed state	Also known as *current state* or *actual state*. The most up-to-date view of the cluster and running applications.
Orchestrator	Software that deploys and manages microservices apps. Kubernetes is the de facto orchestrator of microservices apps based on containers.
Pod	A thin wrapper that enables containers to run on Kubernetes. Defined in a YAML file. The smallest unit of deployment on a Kubernetes cluster.
Reconciliation loop	A controller process watching the state of the cluster, via the API Server, ensuring observed state matches desired state. The Deployment controller runs as a reconciliation loop.
Service	Capital "S". Kubernetes object for providing network access to apps running in Pods. Can integrate with cloud platforms and provision internet-facing load-balancers.
YAML	Yet Another Markup Language. The configuration language Kubernetes configuration files are written in.


The end beginning...
... of an exciting chapter of your career!