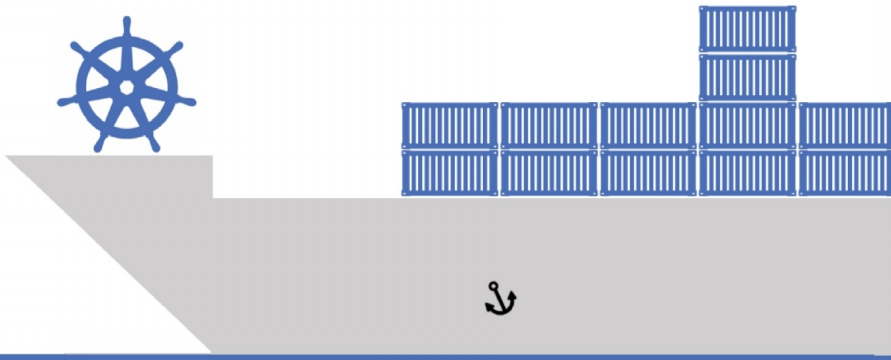
The Kubernetes Book



Version 2 (Oct 2017)

Nigel Poulton

The Kubernetes Book

Nigel Poulton

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

This version was published on 2017-10-27

* * * * *

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.

* * * * *

Huge thanks to my wife and kids for putting up with a geek in the house who genuinely thinks he's a bunch of software running inside of a container on top of midrange biological hardware. It can't be easy living with me!

Massive thanks as well to everyone who watches my Pluralsight videos. I love connecting with you and really appreciate all the feedback I've gotten over the years. This was one of the major reasons I decided to write this book! I hope it'll be an amazing tool helping to you drive your careers even further forward.

One final word to all you 'old dogs' out there… get ready to learn some new tricks!

Table of Contents

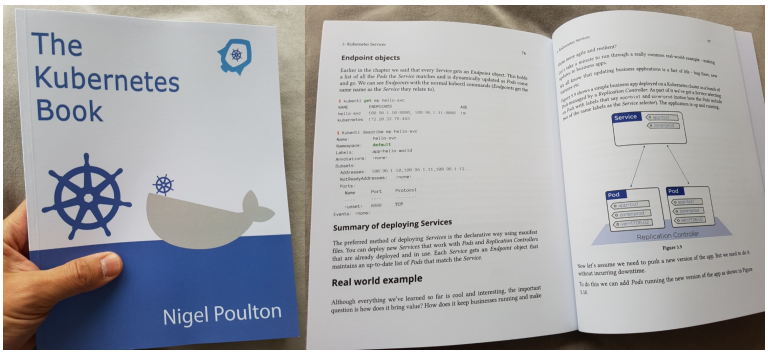
0: About the book

This is an **up-to-date* book about Kubernetes. No prior knowledge required!

If you're interested in Kubernetes, *want to know how it works and how to do things properly*, this book is dedicated to you!

What about a paperback edition

I'm a fan of ink and paper, and I'm also a fan of **quality products**. So I've made a high-quality, full-color, paperback edition available via Amazon. If you like paperbacks, you'll love this! No cheap paper, and definitely no black-and-white diagrams from the 1990's!



While I'm talking about Amazon and paperbacks… I'd appreciate it if you'd give the book a review on Amazon. It'll take two-minutes, and you know you should! Especially considering the long dark months I devoted to writing it for you :-P

Why should I read this book or care about Kubernetes?

Alongside Docker, Kubernetes is the hottest infrastructure technology on the planet! It's literally changing the way we do everything in IT.

So, if you want to push ahead with your career and work with one of the coolest technologies to every hit the scene, you need to read this book. If you'd rather get left behind… don't read the book. It's the truth.

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

You're asking the wrong person :-D

Kubernetes is Kubernetes. So there's gonna be some duplicate content - there's not a lot I can do about that!

But… I'm a huge believer in learning via multiple methods. It's my honest opinion that a combination of video training and books is way forward. Each brings its own strengths, and each reinforces the other. So yes, I think you should consume both! But I suppose I *would* say that ;-)

Final word: The book has enough 4 and 5-star ratings to reassure you it's a good investment of your time and money.

Versions of the book

Kubernetes is developing fast! As a result, the value of a book like this is inversely proportional to how old it is! In other words, the older this book is, the less valuable it is. So, I'm committed to **at least two updates per year**. If my *Docker Deep Dive* book is anything to go by, it'll be more like an updated version every 2-3 months!

Does that seem like a lot? **Welcome to the new normal!**

We no-longer live in a world where a 5-year-old book is valuable. **On a topic like Kubernetes, I even doubt the value of a 1-year-old book!** As an author, I really wish that wasn't true. But it is! Again… welcome to the new normal!

Don't worry though, your investment in this book is safe!

If you buy the paperback copy from **Amazon**, you get the Kindle version for dirt-cheap! And the Kindle and Leanpub versions get access to all updates at no extra cost! That's the best I can currently do!

I you buy the book through other channels, things might be different - I don't control other channels - I'm a techie, not a book distributor.

Below is a list of versions:

- **Version 1.** Initial version.
- **Version 2.** Updated content for Kubernetes 1.8.0. Added new chapter on ReplicaSets. Added significant changes to Pods chapter. Fixed typos and made a few other minor updates to existing chapters.

Having trouble getting the latest updates on your Kindle?

Unfortunately, Kindle isn't great at delivering updates. But the work-around is easy:

Go to http://amzn.to/2l53jdg

Under `Quick Solutions` (on the left) select `Digital Purchases`. Select `Content and Devices` for **The Kubernetes Book** order. Your book should show up in the list with a button that says "Update Available". Click that button. Delete your old version in Kindle and download the new one.

1: Kubernetes Primer

This chapter is split into two main sections.

- Kubernetes background - where it came from etc.
- The idea of Kubernetes as a data center OS

Kubernetes background

Kubernetes is an orchestrator. More specifically, it's an orchestrator of containerized apps. This means it helps us deploy and maintain applications that are distributed and deployed as containers. It does *scaling*, *self-healing*, *load-balancing* and lots more.

Starting from the beginning… Kubernetes came out of Google! In the summer of 2014 it was open-sourced and handed over to the Cloud Native Computing Foundation (CNCF).

Figure 1.1

Since then, it's gone on to become one of the most important container-related technologies in the world - on a par with Docker.

Like many of the other container-related projects, it's written in Go (Golang). It lives on Github at `kubernetes/kubernetes`. It's actively discussed on the IRC channels, you can follow it on Twitter (@kubernetesio), and this is pretty good slack channel - slack.k8s.io. There are also regular meetups going on all over the planet!

Kubernetes and Docker

The first thing to say about Kubernetes and Docker is that they're complimentary technologies.

For example, it's very popular to deploy Kubernetes with Docker as the container runtime. This means Kubernetes orchestrates one or more hosts that run containers, and Docker is the technology that starts, stops, and otherwise manages the containers. In this model, Docker is a lower-level technology

manages the containers. In this model, Docker is a lower-level technology that is orchestrated and managed by Kubernetes.

At the time of writing, Docker is in the process of breaking-out individual components of its stack. One example is `containerd` - the low-level container supervisor and runtime components. Kubernetes has also released the Container Runtime Interface (CRI) - a runtime abstraction layer that 3rd-arty container runtimes can plug in to and seamlessly work with Kubernetes. On the back of these two important projects is a project to implement `containerd` with the CRI and potentially make it the default Kubernetes container runtime (author's personal opinion). The project is currently a *Kubernetes Incubator* project with an exciting future.

Although `containerd` will not be the only container runtime supported by Kubernetes, it will almost certainly replace Docker as the most common, and possibly default. Time will tell.

The important thing, is that none of this will impact your experience as a Kubernetes user. All the regular Kubernetes commands and patterns will continue to work as normal.

What about Kubernetes vs Docker Swarm

At *DockerCon EU* in Copenhagen in October 2017, Docker, Inc. formally announced native support for Kubernetes in Docker Enterprise Edition (Docker EE).

> **Note:** All of the following is my personal opinion (everything in the book is my personal opinion). None of the following should be considered as the official position of Docker, Inc.

This was a significant announcement. It essentially "blessed" Kubernetes to become the industry-standard container orchestrator.

Now then, I am aware that Kubernetes did not *need* Docker's "blessing". I'm also aware that the community had already chosen Kubernetes. And… that Docker was bowing to the inevitable. However, it was still a significant move. Docker, Inc. has always been heavily involved in community projects, and already, the number of Docker, Inc. employees working openly on Kubernetes and Kubernetes-related projects has increased. Clearly this was a good announcement for Kubernetes.

On the topic of Docker Swarm, the announcement means that the orchestration components of Docker Swarm (a rival orchestrator to Kubernetes) will probably become less of a focus for Docker, Inc. It will continue to be developed, but the long-term strategic orchestrator for containerized applications is Kubernetes!

Kubernetes and Borg: Resistance is futile!

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

It's no secret that Google has been running many of its systems on containers for years. Legendary stores of them crunching through *billions of containers a week* are retold at meetups all over the world. So yes, for a very long time – even before Docker came along - Google has been running things like *search*, *Gmail*, and *GFS* on containers. And **lots** of them!

Pulling the strings and keeping those billions of containers in check are a couple of in-house technologies and frameworks called *Borg* and *Omega*. So, it's not a huge stretch to make the connection between *Borg* and *Omega*, and Kubernetes - they're all in the game of managing containers at scale, and they're all related to Google.

This has occasionally led to people thinking Kubernetes is an open-source version of either *Borg* or *Omega*. But it's not! It's more like Kubernetes shares its DNA and family history with them. A bit like this… *In the beginning was Borg…… and Borg begat Omega. And Omega *knew* the open-source community and begat her Kubernetes.**

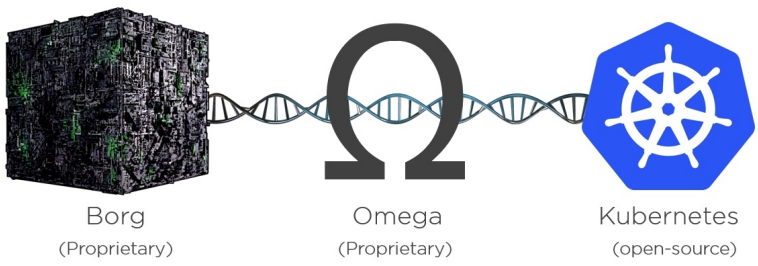


Figure 1.2 - Shared DNA

The point is, all three are separate, but all three are related. In fact, a lot of the people involved with building Borg and Omega were also involved in building Kubernetes.

So, although Kubernetes was built from scratch, it leverages much or what was learned at Google with Borg and Omega.

As things stand, Kubernetes is open-source project under the CNCF, licensed under the Apache 2.0 license, and version 1 shipped way back in July 2015.

Kubernetes - what's in the name

The name **Kubernetes** comes from the Greek word meaning *Helmsman* - that's the person who steers a ship. This theme is reflected in the logo.

Figure 1.3 - The Kubernetes logo

> **Rumor:** There's a good rumor that Kubernetes was originally going to be called *Seven of Nine*. If you know your Star Trek, you'll know that *Seven of Nine* is a female **Borg** rescued by the crew of the USS Voyager under the command of Captain Catherine Janeway. It's also rumored that the logo has 7 spokes because of *Seven of Nine*. These could be nothing more than rumors, but I like them!

One last thing about the name before moving on… You'll often see the name shortened to **k8s**. The idea is that the number 8 replaces the 8 characters in between the K and the S – great for tweets and lazy typists like me ;-)

A data center OS

As we said in the intro, I'm assuming you've got a basic knowledge of what containers are and how they work. If you don't, go watch my 5-star video course here https://app.pluralsight.com/library/courses/docker-containers-big-picture/table-of-contents

Generally speaking, containers make our old scalability challenges seem laughable - we've already talked about Google going through billions of containers per week!!

But not everybody is the size of Google. What about the rest of us?

As a general rule, if your legacy apps had hundreds of VMs, there's a good chance your containerized apps will have thousands of containers! If that's

true, we desperately need a way to manage them.

Say hello to Kubernetes!

When getting your head around something like Kubernetes it's important to get your head around modern data center architectures. For example, we're abandoning the traditional view of the data center as collection of computers, in favor of the more powerful view that the data center **is a single large computer**.

So what do we mean by that?

A typical computer is a collection of CPU, RAM, storage, and networking. But we've done a great job of building operating systems (OS) that abstract away a lot of that detail. For example, it's rare for a developer to care which CPU core or memory DIM their application uses – we let the OS decide all of that. And it's a good thing, the world of application development is a far friendlier place because of it.

So, it's quite natural to take this to the next level and apply those same abstractions to data center resources - to view the data center as just a pool of compute, network and storage and have an over-arching system that abstracts it. This means we no longer need to care about which server or LUN our containers are running on - just leave this up to the data center OS.

Kubernetes is one of an emerging breed of data center operating systems aiming to do this. Others do exist, Mesosphere DCOS is one. These systems are all in the *cattle business*. Forget about naming your servers and treating them like *pets*. These systems don't care. Gone are the days of taking your app and saying *"OK run this part of the app on this node, and run that part of it on that node…"*. In the Kubernetes world, we're all about saying *"hey Kubernetes, I've got this app and it consists of these parts… just run it for me please"*. Kubernetes then goes off and does all the hard scheduling work.

It's a bit like sending goods with a courier service. Package the goods in the courier's standard packaging, label it and give it to the courier. The courier takes care of everything else – all the complex logistics of which planes and trucks it goes on, which drivers to use etc. The only thing that the courier requires is that it's packaged and labelled according to their requirements.

The same goes for app in Kubernetes. Package it as a container, give it a declarative manifest, and let Kubernetes take care of running it and keeping it running. It's a beautiful thing!

While all of this sounds great, don't take this *data center OS* thing too far. It's not a DVD install, you don't end up with a shell prompt to control your entire data center, and you definitely don't get a solitaire card game included! We're still at the very early stages in the trend.

Some quick answers to quick questions

After all of that, you're probably pretty skeptical and have a boat-load of questions. So here goes trying to pre-empt a couple of them…

Yes, this is forward thinking. In fact, it's almost bleeding edge. But it's here, and it's real! Ignore it at your own peril.

Also, I know that most data centers are complex and divided into zones such as DMZs, dev zones, prod zones, 3rd party equipment zones, line of business zones etc. However, within each of these zones we've still got compute, networking and storage, and Kubernetes is happy to dive right in and start using them. And no, I don't expect Kubernetes to take over your data center. But it will become a part of it.

Kubernetes is also very platform agnostic. It runs on bare metal, VMs, cloud instances, OpenStack, pretty much anything with Linux.

Chapter summary

Kubernetes is *the* leading container orchestrator that lets us manage containerized apps at scale. We give it an app, tell it what we want the app to look like, and let Kubernetes make all the hard decisions about where to run it and how to keep it running.

It came out of Google, is open-sourced under the Apache 2.0 license, and lives within the Cloud Native Computing Foundation (CNCF).

Disclaimer!

Kubernetes is a fast-moving project under active development. So things are changing fast! But don't let that put you off - embrace it! Rapid change like this is the new normal!

As well as reading the book, I suggest you follow @kubernetesio on Twitter, hit the various k8s slack channels, and attend your local meetups. These will all help to keep you up-to-date with the latest and greatest in the Kubernetes world. I'll also be updating the book regularly and producing more video training courses!

2: Kubernetes principles of operation

In this chapter, we'll learn about the major components required to build a Kubernetes cluster and deploy a simple app. The aim of the game is to give you a big-picture.

You're not supposed to understand it all at this stage - we will dive into more detail in later chapters!

We'll divide the chapter up like this:

- Kubernetes from 40K feet
- Masters and nodes
- Declarative model and desired state
- Pods
- Services
- Deployments

Kubernetes from 40K feet

At the highest level, Kubernetes is an orchestrator of containerized apps. Ideally microservice apps. *Microservice app* is just a fancy name for an application that's made up of lots of small and independent parts - we sometimes call these small parts *services*. These small independent services work together to create a meaningful/useful app.

Let's look at a quick analogy.

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

Figure 2.1

Figure 2.2

The coach also makes sure that the team maintains its formation and sticks to the plan. Well guess what! Microservice apps in the Kubernetes world are just the same!

We start out with an app made up of multiple services. Each service is packaged as a *Pod* and no two services are the same. Some might be load-balancers, some might be web servers, some might be for logging… Kubernetes comes along - a bit like the coach in the football analogy – and organizes everything into a useful app.

In the application world, we call this "**orchestration**".

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

The masters are in-charge of the cluster and make all the decisions about which nodes to schedule application services on. They also monitor the cluster, implement changes, and respond to events. For this reason, we often refer to the master as the *control plane*.

Then the nodes are where our application services run. They also report back to the masters and watch for changes to the work they've been scheduled.

At the time of writing, the best way to package and deploy a Kubernetes application is via something called a *Deployment*. With Deployments, we start out with our application code and we containerize it. Then we define it as a Deployment via a YAML or JSON manifest file. This manifest file tells Kubernetes two important things:

- What our app should look like – what images to use, ports to expose, networks to join, how to perform update etc.
- How many replicas of each part of the app to run (scale)

Then we give the file to the Kubernetes master which takes care of deploying it on the cluster.

But it doesn't stop there. Kubernetes is constantly monitoring the Deployment to make sure it is running exactly as requested. If something isn't as it should be, Kubernetes tries to it.

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

Masters and nodes

A Kubernetes cluster is made up of masters and nodes. These are Linux hosts running on anything from VMs, bare metal servers, all the way up to private and public cloud instances.

Masters (control plane)

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

The simplest (and most common) setups run all the master *services* on a single host. However, multi-master HA is becoming more and more popular, and is a *must have* for production environments. Looking further into the future, we might see the individual services comprise the control plane split-out and distributed across the cluster - a distributed control plane.

It's also considered a good practice **not** to run application workloads on the master. This allows the master to concentrate entirely on looking after the state of the cluster.

Let's take a quick look at the major pieces that make up the Kubernetes master.

The API server

The API Server (apiserver) is the frontend into the Kubernetes control plane. It exposes a RESTful API that preferentially consumes JSON. We POST manifest files to it, these get validated, and the work they define gets deployed to the cluster.

You can think of the API server as the brains of the cluster.

The cluster store

If the API Server is the brains of the cluster, the *cluster store* is its memory. The config and state of the cluster gets persistently stored in the cluster store, which is the only stateful component of the cluster and is vital to its operation - no cluster store, no cluster!

The cluster store is based on **etcd**, the popular distributed, consistent and watchable key-value store. As it is the *single source of truth* for the cluster, you should take care to protect it and provide adequate ways to recover it if things go wrong.

The controller manager

The controller manager (kube-controller-manager) is currently a bit of a monolith - it implements a few features and functions that'll probably get split out and made pluggable in the future. Things like the node controller, endpoints controller, namespace controller etc. They tend to sit in loops and watch for changes – the aim of the game is to make sure the *current state* of the cluster matches the *desired state* (more on this shortly).

The scheduler

At a high level, the scheduler (kube-scheduler) watches for new workloads and assigns them to nodes. Behind the scenes, it does a lot of related tasks such as evaluating affinity and anti-affinity, constraints, and resource management.

Control Plane summary

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

The API Server is the front-end into the master and the only component in the control plane that we interact with directly. By default, it exposes a RESTful

endpoint on port 443.

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

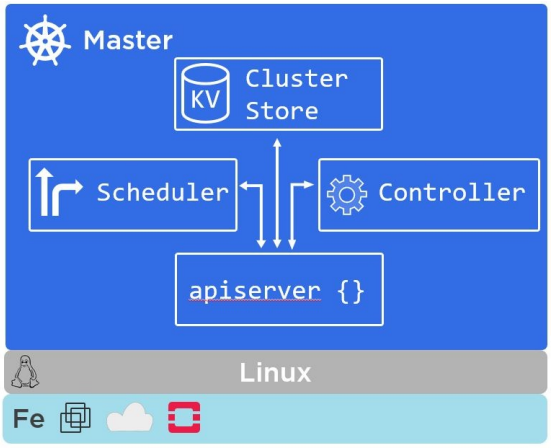


Figure 2.3

Nodes

First up, *nodes* used to be called *minions*. So, when some of the older docs and blogs talk about minions, they're talking about *nodes*.

As we can see from Figure 2.4, nodes are a bit simpler than *masters*. The only things that we care about are the *kubelet*, the *container runtime*, and the *kube-proxy*.

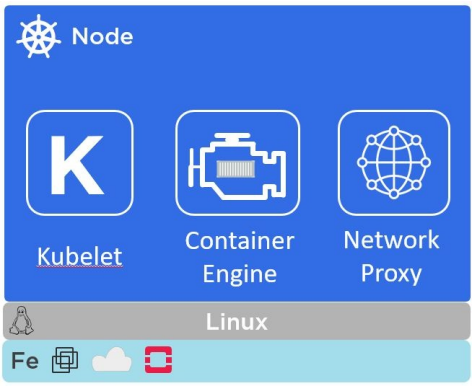


Figure 2.4

Kubelet

First and foremost is the kubelet. This is the main Kubernetes agent that runs on all cluster nodes. In fact, it's fair to say that the kubelet *is* the node. You install the kubelet on a Linux host and it registers the host with the cluster as a node. It then watches the API server for new work assignments. Any time it sees one, it carries out the task and maintains a reporting channel back to the master.

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

On the topic of reporting back, the kubelet exposes an endpoint on port `10255` where you can inspect it. We're not going to spend time on this in the book, but it is worth knowing that port `10255` on your nodes lets you inspect aspects of the kubelet.

Container runtime

The Kubelet needs to work with a container runtime to do all the container management stuff – things like pulling images and starting and stopping containers. More often than not, the container runtime that Kubernetes uses is Docker. In the case of Docker, Kubernetes talks natively to the Docker Remote API.

More recently, Kubernetes has released the Container Runtime Interface (CRI). This is an abstraction layer for external (3rd-party) container runtimes to plug in to. Basically, the CRI masks the internal machinery of Kubernetes and exposes a clean documented container runtime interface.

The CRI is now the default method for container runtimes to plug-in to Kubernetes. The `containerd` CRI project is a community-based open-source project porting the CNCF `containerd` runtime to the CRI interface. It has a lot of support and will probably replace Docker as the default, and most popular, container runtime used by Kubernetes.

> **Note:** `containerd` is the container supervisor and runtime logic stripped out of the Docker Engine. It was donated to the CNCF by Docker, Inc. and has a lot of community support.

At the time of writing, Docker is still the most common container runtime used by Kubernetes.

Kube-proxy

The last piece of the puzzle is the kube-proxy. This is like the network brains of the node. For one thing, it makes sure that every Pod gets its own unique IP address. It also does lightweight load-balancing on the node.

The declarative model and desired state

The *declarative model* and the concept of *desired state* are two things at the very heart of the way Kubernetes works. Take them away and Kubernetes crumbles!

In Kubernetes, the two concepts work like this:

1. We declare the desired state of our application (microservice) in a manifest file
2. We POST it the API server
3. Kubernetes stores this in the cluster store as the application's desired state
4. Kubernetes deploys the application on the cluster
5. Kubernetes implements watch loops to make sure the cluster doesn't vary from desired state

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

Manifest files are either YAML or JSON, and they tell Kubernetes how we want our application to look. We call this is the desired state. It includes things like which image to use, how many replicas to have, which network to operate on, and how to perform updates.

Once we've created the manifest, we POST it to the API server. The most common way of doing this is with the `kubectl` command. This sends the manifest to port 443 on the master.

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

Finally, Kubernetes sets up background reconciliation loops that constantly monitor the state of the cluster. If the current state of the cluster varies from the desired state Kubernetes will try and rectify it.

It's important to understand that what we've described is the opposite of the imperative model. The imperative model is where we issue lots of platform specific commands.

Not only is the declarative model a lot simpler than long lists of imperative commands, it also enables self-healing, scaling, and lends itself to version control and self-documentation!

But the declarative story doesn't end there. Things go wrong, and things change, and when they do, the ***current state*** of the cluster no longer matches the ***desired state***. As soon as this happens Kubernetes kicks into action and does everything it can to bring the two back into harmony. Let's look at an example.

Assume we have an app with a desired state that includes 10 replicas of a web front-end Pod. If a node that was running two replicas dies, the current state will be reduced to 8 replicas, but the desired state will still be 10. This will be picked up by a reconciliation loop and Kubernetes will schedule two new replicas on other nodes in the cluster.

The same thing will happen if we intentionally scale the desired number of replicas up or down. We could even change the image we want the web front-end to use. For example, if the app is currently using the `v2.00` image, and we update the desired state to use the `v2.01` image, Kubernetes will go through the process of updating all replicas so that they are using the new image.

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

Pods

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

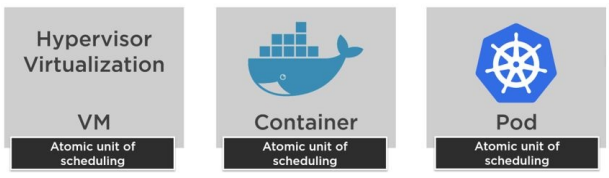


Figure 2.5

Pods and containers

It's true that Kubernetes runs containerized apps. But those containers **always** run inside of Pods! You cannot run a container directly on a Kubernetes cluster.

However, it's a bit more complicated than that. The simplest model is to run a single container inside of a Pod, but there are advanced use-cases where you can run multiple containers inside of a single Pod. These *multi-container Pods* are beyond the scope of this book, but common examples include the following:

- web containers supported a *helper* container that ensures the latest content is available to the web server.
- web containers with a tightly coupled log scraper tailing the logs off to a logging service somewhere else.

These are just a couple of examples.

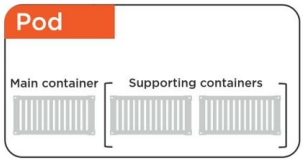


Figure 2.6

Pod anatomy

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

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

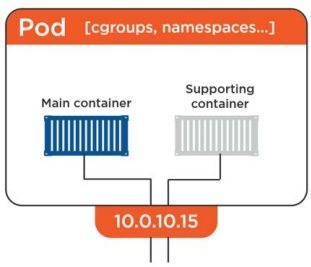


Figure 2.7

If those containers need to talk to each other (container-to-container within the Pod) they can use the Pods `localhost` interface.

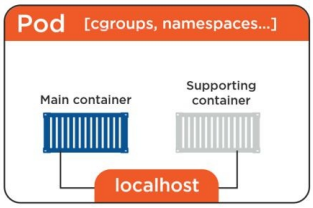


Figure 2.8

This means that multi-container Pods are ideal when you have requirements for tightly coupled containers – maybe they need to share memory and storage etc. However, if you don't **need** to tightly couple your containers, you should put them in their own Pods and loosely couple them over the network. Figure 2.9 shows two tightly coupled containers sharing memory and storage inside a single Pod. Figure 2.10 shows two loosely coupled containers in separate Pods on the same network.

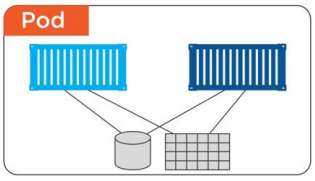


Figure 2.9 - Tightly coupled Pods

Figure 2.10 - Loosely coupled Pods

Pods as the atomic unit

Pods are also the minimum unit of scaling in Kubernetes. If you need to scale your app, you do so by adding or removing Pods. You **do not** scale by adding more of the same containers to an existing Pod! Multi-container Pods are for two complimentary containers that need to be intimate - they are not for scaling. Figure 2.11 shows how to scale the `nginx` front-end of an app using multiple Pods as the unit of scaling.

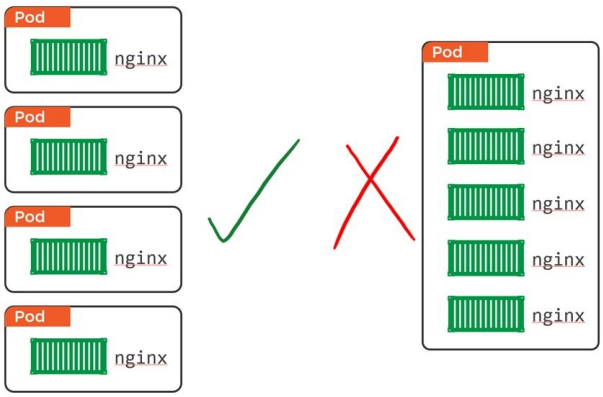


Figure 2.11 - Scaling with Pods

The deployment of a Pod is an all-or-nothing job. You never get to a situation where you have a partially deployed Pod servicing requests. The entire Pod either comes up and it's put into service, or it doesn't, and it fails. A Pod is never declared as up and available until every part of it is up and running.

A Pod can only exist on a single node. This is true even of multi-container Pods, making them ideal when complimentary containers need to be scheduled side-by-side on the same node.

Pod lifecycle

Pods are mortal. They're born, they live, and they die. If they die unexpectedly, we don't bother trying to bring them back to life! Instead, Kubernetes starts another one in its place – but it's not the same Pod, it's a shiny new one that just happens to look, smell, and feel exactly like the one that just died.

Pods should be treated as cattle - don't build your Kubernetes apps to be emotionally attached to their Pods so that when one dies you get sad and try to nurse it back to life. Build your apps so that when their Pods fail, a totally new one (with a new ID and IP address) can pop up somewhere else in the cluster and take its place.

Deploying Pods

We normally deploy Pods indirectly as part of something bigger, such as a *ReplicaSet* or *Deployment* (more on these later).

Deploying Pods via ReplicaSets

Before moving on to talk about *Services*, we need to give a quick mention to *ReplicaSets (rs)*.

A ReplicaSet is a higher-level Kubernetes object that wraps around a Pod and adds features. As the names suggests, they take a Pod template and deploy a desired number of *replicas* of it. They also instantiate a background reconciliation loop that checks to make sure the right number of replicas are always running – desired state vs actual state.

ReplicaSets can be deployed directly. But more often than not, they are deployed indirectly via even higher-level objects such as Deployments.

Services

We've just learned that Pods are mortal and can die. If they are deployed via ReplicaSets or Deployments, when they fail, they get replaced with new Pods somewhere else in the cluster - these Pods have totally different IPs! This also happens when we scale an app - the new Pods all arrive with their own new IPs. It also happens when performing rolling updates - the process of replacing old Pods with new Pods results in a lot of IP churn.

The moral of this story is that **we can't rely on Pod IPs**. But this is a problem. Assume we've got a microservice app with a persistent storage backend that other parts of the app use to store and retrieve data. How will this work if we can't rely on the IP addresses of the backend Pods?

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

Take a look at Figure 2.12. This shows a simplified version of a two-tier app with a web front-end that talks to a persistent backend. But it's all Pod-based, so we know the IPs of the backend Pods can change.

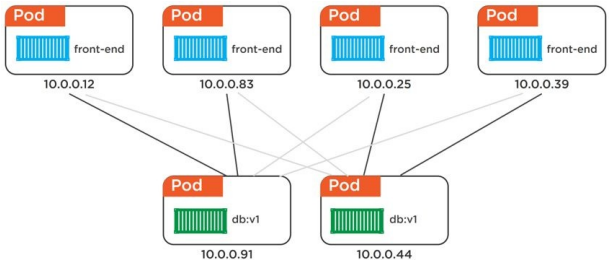


Figure 2.12

If we throw a Service object into the mix, as shown in Figure 2.13, we can see how the front-end can now talk to the reliable IP of the Service, which in-turn load-balances all requests over the backend Pods behind it. Obviously, the Service keeps track of which Pods are behind it.

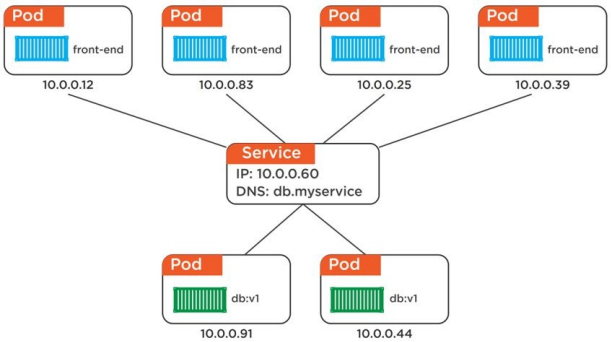


Figure 2.13

Digging in to a bit more detail. A Service is a fully-fledged object in the Kubernetes API just like Pods, ReplicaSets, and Deployments. They provide stable DNS, IP addresses, and support TCP and UDP (TCP by default). They also perform simple randomized load-balancing across Pods, though more advanced load balancing algorithms may be supported in the future. This adds up to a situation where Pods can come and go, and the Service automatically updates and continues to provide that stable networking endpoint.

The same applies if we scale the number of Pods - all the new Pods, with the new IPs, get seamlessly added to the Service and load-balancing keeps working.

So that's the job of a Service – it's a stable network abstraction point for multiple Pods that provides basic load balancing.

Connecting Pods to Services

The way that a Service knows which Pods to load-balance across is via labels.

Figure 2.14 shows a set of Pods labelled as `prod`, `BE` (short for backend) and `1.3`. These Pods are loosely associated with the service because they share the same labels.

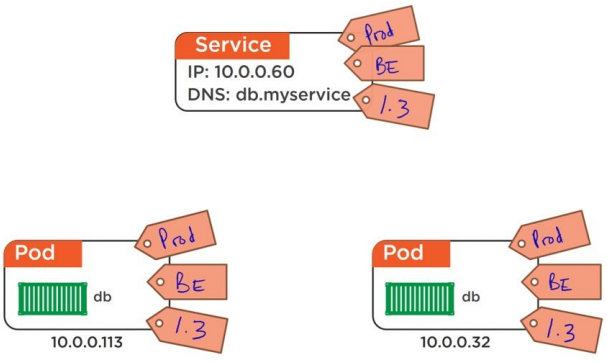


Figure 2.14

Figure 2.15 shows a similar setup, but with an additional Pod that does not share the same labels as the Service. Because of this, the Service will not load balance requests to it.

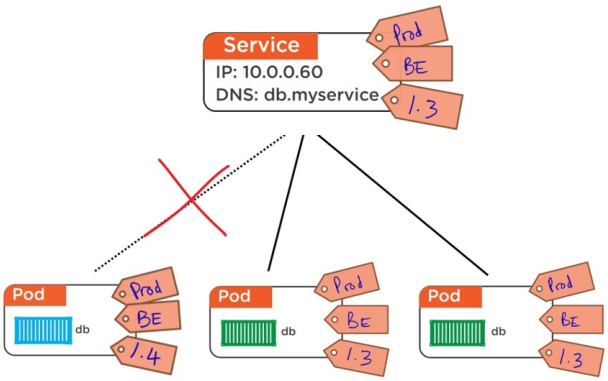


Figure 2.15

One final thing about Services. They only send traffic to healthy Pods. This means if a Pod is failing health-checks, it will not receive traffic form the Service.

So yeah, Services bring stable IP addresses and DNS names to the unstable world of Pods!

Deployments

Deployments build on top of ReplicaSets, add a powerful update model, and make versioned rollbacks simple. As a result, they are considered the future of Kubernetes application management.

In order to do this, they leverage the declarative model that is infused throughout Kubernetes.

They've been first-class REST objects in the Kubernetes API since Kubernetes 1.2. This means we define them in YAML or JSON manifest files that we POST to the API server in the normal manner.

Deployments and updates

Rolling updates are a core feature of Deployments. For example, we can run multiple concurrent versions of a Deployment in true blue/green or canary fashion.

Kubernetes can also detect and stop rollouts if the new version is not working.

Finally, rollbacks are super simple!

In summary, Deployments are the future of Kubernetes application management. They build on top of Pods and ReplicaSets by adding a ton of cool stuff like versioning, rolling updates, concurrent releases, and simple rollbacks.

Chapter summary

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

The master is where the Kubernetes control plane services live. It's a combination of several system-services, including the API server that exposes a REST interface to the control plane. Masters make all of the deployment and scheduling decisions, and multi-master HA is important for production-grade environments.

Nodes are where application workloads run. Each node runs a service called the `kubelet` that registers the node with the cluster, and communicates with the master. This includes receiving new work tasks and reporting back about them. Nodes also run a container runtime and a `kube-proxy` service. The container runtime, such as Docker or containerd, is responsible for all container related operations. The `kube-proxy` service is responsible for networking on the node.

We also talked about some of the main Kubernetes API objects, such as Pods, ReplicaSets, Services, and Deployments. The Pod is the basic building-block, ReplicaSets add self-healing and scaling, Services add stable networking and load-balancing, and Deployments add a powerful update model and simple rollbacks.

Now that we know a bit about the basics, we're going to start getting into detail.

3: Installing Kubernetes

In this chapter, we'll take a look at some of the ways to install and get started with Kubernetes.

We'll look at:

- Play with Kubernetes (PWK)
- Using Minikube to install Kubernetes on a laptop
- Installing Kubernetes in the Google Cloud with the Google Container Engine (GKE)
- Installing Kubernetes on AWS using the `kops` tool
- Installing Kubernetes manually using `kubeadm`

Two things to point out before diving in…

Firstly, there are a lot more ways to install Kubernetes. The options I've chosen for this chapter are the ones I think will be most useful.

Secondly, Kubernetes is a fast-moving project. This means some of what we'll discuss here will change. But don't worry, I'm keeping the boo up-to-date, so nothing will be irrelevant.

Play with Kubernetes

Play with Kubernetes (PWK) is a web-based Kubernetes playground that you can use for free. All you need is a web browser and an internet connection. It is the fastest, and easiest, way to get your hands on Kubernetes.

Let's see what it looks like.

1. Point your browser at http://play-with-k8s.com
2. Confirm that you're a human and lick `+ ADD NEW INSTANCE`

   You will be presented with a terminal window in the right of your browser. This is a Kubernetes node (`node1`).

3. Run a few commands to see some of the components pre-installed on the node.

```
$ docker version
Docker version 17.06.0-ce, build 02c1d87

$ kubectl version --output=yaml
clientVersion:
buildDate: 2017-06-29T23:15:59Z
compiler: gc
gitCommit: d3ada0119e776222f11ec7945e6d860061339aad
gitTreeState: clean
gitVersion: v1.7.0
goVersion: go1.8.3
major: "1"
minor: "7"
platform: linux/amd64
```

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

It is also worth noting that although the command prompt is a `$`, we're actually running as `root` on the node. You can confirm this with the `whoami` or `id` commands.

4. Use the `kubeadm` command to initialise a new cluster

   When you added a new instance in step 2, PWK gave you a short list of commands that will initialize a new Kubernetes cluster. One of these was `kubeadm init`. The following command will initialize a new cluster and configure the API server to listen on the correct IP interface.

   ```
   $ kubeadm init --apiserver-advertise-address $(hostname -i)
   [kubeadm] WARNING: kubeadm is in beta, do not use it for prod...
   [init] Using Kubernetes version: v1.7.9
   [init] Using Authorization modes: [Node RBAC]
   <Snip>
   Your Kubernetes master has initialized successfully!
   <Snip>
   ```

   Congratulations! You now have a brand new single-node Kubernetes cluster! The node that we executed the command from (`node 1`) is initialized as the *master*.

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

5. Use the `kubectl` command to verify the cluster

```
$ kubectl get nodes
NAME      STATUS     AGE       VERSION
node1     NotReady   1m        v1.7.0
```

The output shows a single-node Kubernetes cluster. However, the status of the node is `NotReady`. This is because there is no Pod network configured. When you first logged on to the PWK node, you were given a list of three commands to configure the cluster. So far, we've only executed the first one (`kubeadm init`).

6. Initialize cluster networking (a Pod network)

   Copy the second command from the list of three commands were printed on the screen when you first created `node1` (this will be a `kubectl apply` command). Paste it onto a new line in the terminal.

```
$ kubectl apply -n kube-system -f \
 "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | ba
 tr -d '\n')"

 serviceaccount "weave-net" created
 clusterrole "weave-net" created
 clusterrolebinding "weave-net" created
 daemonset "weave-net" created
```

7. Verify the cluster again to see if `node1` has changed to `Ready`

```
$ kubectl get nodes
NAME      STATUS    AGE       VERSION
node1     Ready     2m        v1.7.0
```

   Now that Pod networking has been configured and the cluster has transitioned into the `Ready` status, you're now ready to add more nodes.

8. Copy the `kubeadm join` command from the output of the `kubeadm init`

   When you initialized the new cluster with `kubeadm init`, the final output of the command listed a `kubeadm join` command that could be used to add more nodes to the cluster. This command included the cluster join-token and the IP socket that that the API server is listening on. Copy this command and be ready to paste it into the terminal of a new node (`node2`).

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

You will be given a new node called <uniqueID>_node2. We'll call this `node2` for the remainder of the steps.

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

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

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

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

```
$ kubectl get nodes
NAME      STATUS    AGE       VERSION
node1     Ready     5m        v1.7.0
node2     Ready     1m        v1.7.0
```

Your Kubernetes cluster now has two nodes.

Feel free to add more nodes with the `kubeadm join` command.

Congratulations! You have a fully working Kubernetes cluster.

It's worth pointing out that `node1` was initialized as the Kubernetes master, and additional nodes will join the cluster as nodes. PWK gives Masters a blue icon next to their names, and Nodes a transparent icon. This helps you easily identify Masters and Nodes.

PWK sessions only last for 4 hours and are obviously not intended for production use. You are also limited to the version of Kubernetes that the platform provides. Despite all of this, it is excellent for learning and testing. You can easily add and remove instances, as well as tear everything down and start again from scratch.

Play with Kubernetes is a great project ran by a couple of excellent Docker Captains that I know personally. I highly recommend it as the best place to build your first Kubernetes cluster!

Minikube

Minikube is great if you're a developer and need a local Kubernetes development environment on your laptop. What it's **not** great for is **production**. You've been warned!

Basic Minikube architecture

If you know Docker, Minikube is similar to *Docker for Mac* and *Docker for Windows* - a super-simple way to spin up something on your laptop.

Figure 3.1 shows how the implementation of Minikube is similar to Docker for Mac and Docker for Windows. On the left is the high-level Minikube architecture, on the right is the *Docker for Mac* or *Docker for Windows* architecture.

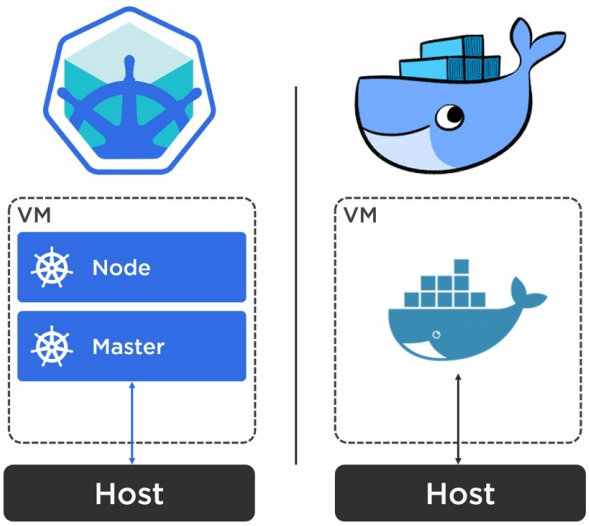


Figure 3.1

At a high level, you download the Minikube installer and type `minikube start`. This creates a local virtual machine (VM) and spins up Kubernetes cluster inside of the VM. It also sets up your local shell environment so that you can access the cluster directly from your current shell (no requirement to SSH into the VM).

Inside of the Minikube VM there are two high level things we're interested in:

- First up, there's the *localkube* construct. This runs a Kubernetes node and a Kubernetes master. The master includes an API server and all of the other control plane stuff.
- Second up, the container runtime is pre-installed. At the time of writing this defaults to Docker, though you can specify *rkt* instead if you require.

This architecture is shown in Figure 3.2.

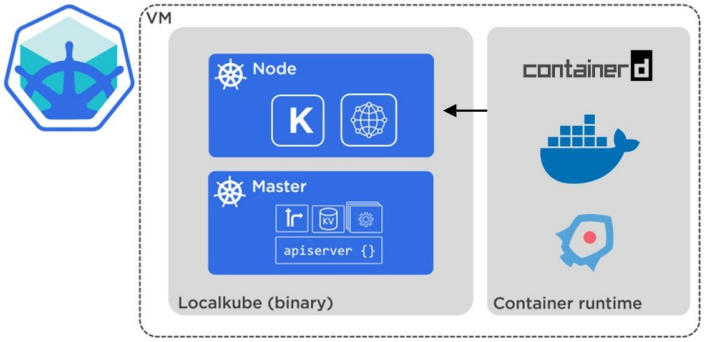


Figure 3.2

Finally, outside of the VM we have `kubectl`, the Kubernetes client. When you run `minikube start`, your environment will be configured so that `kubectl` will issue commands to the Kubernetes cluster running inside the Minikube.

As interesting as the architecture is, the most important aspect of Minikube is the slick and smooth experience that accurately replicates a Kubernetes cluster.

Installing Minikube

You can get Minikube for Mac, Windows, and Linux. We'll take a quick look at Mac and Windows, as this is what most people run on their laptops.

> **Note:** Minikube requires virtualization extensions enabled in your system's BIOS.

Installing Minikube on Mac

Before jumping in and installing Minikube it's probably a good idea to install `kubectl` (the Kubernetes client) on your Mac. You will use this later to issue commands to the Minikube cluster.

1. Use Brew to install `kubectl`

   ```
   $ brew install kubectl
   Updating Homebrew...
   ```

   This puts the kubectl binary in `/usr/local/bin` and makes it executable.

2. Verify that the install worked.

   ```
   $ kubectl version --client
   Client Version: version.Info{Major:"1", Minor:"8"...
   ```

Now that we've installed the `kubectl` client, let's install Minikube.

1. Use Brew to install Minikube.

   ```
   $ brew cask install minikube
   ==> Downloading https://storage.googlapis.com/minikube...
   ```

   Provide your password if prompted.

2. Use Brew to install the **xhyve** lightweight hypervisor for Mac.

   Other Hypervisor options are available - VirtualBox and VMware Fusion - but we're only showing xhive.

   ```
   $ brew install docker-machine-driver-xhyve
   ==> Downloading https://homebrew.bintray.combottles...
   ```

3. Set the user owner of **xhyve** to be `root` (the following command should be issued on a single line and there should be no backslashes in the command `\`).

   ```
   $ sudo chown root:wheel $(brew --prefix)/opt/docker-machine-driver-xhy
   /docker-machine-driver-xhyve
   ```

4. Grant it the ability to `setuid` (the following command should be issued on a single line and there should be no backslashes in the command `\`).

   ```
   $ sudo chmod u+s $(brew --prefix)/opt/docker-machine-driver-xhyve/bin/
   -machine-driver-xhyve
   ```

5. Start Minikube with the following command.

   ```
   $ minikube start --vm-driver=xhyve
   Starting local Kubernetes cluster...
   Starting VM...
   ```

   `minikube start` is the simplest way to start Minikube. Specifying the `--vm-driver=xhyve` flag will force it to use the **xhyve** hypervisor instead of VirtualBox.

You now have a Minikube instance up and running on your Mac!

Use `kubectl` to verify the Minikube install

The `minikube start` operation configures your shell so that you can use `kubectl` against your new Minikube. Test this by running the following `kubectl` command from same shell that you ran `minikube start` from.

```
$ kubectl config current-context
minikube
```

Great, your kubectl context is set to Minikube (this means `kubectl` commands will be sent to the Minikube cluster).

It's worth pointing out that `kubectl` can be configured to talk to any Kubernetes cluster by setting different contexts - you just need to switch between contexts to send commands to different clusters.

Use the `kubectl get nodes` command to list the nodes in the cluster.

```
$ kubectl get nodes
NAME       STATUS   AGE   VERSION
minikube   Ready    1m    v1.8.0
```

That's our single-node Minikube cluster ready to use!

Deleting a Minikube cluster

We spun up the Minikube cluster with a single `minikube start` command. We can stop it with a `minikube stop` command.

```
$ minikube stop
Stopping local Kubernetes cluster...
Machine stopped
```

Stopping a Minikube keeps all the config on disk. This makes it easy to start it up again and pick things up from where you left off.

To blow it away completely - leaving no trace - use the `minikube delete` command.

```
$ minikube delete
Deleting local Kubernetes cluster...
Machine deleted
```

How simple was that!

Running a particular version of Kubernetes inside of Minikube

Minikube lets you specify the version of Kubernetes you want to run. This can be particularly useful if you need to match the version of Kubernetes used in your production environment.

Use `minikube get-k8s-versions` to list the versions available.

```
$ minikube get-k8s-versions
The following Kubernetes versions are available:
        - v1.8.0
        - v1.7.5
        - v1.7.4
        - v1.7.3
        - v1.7.2
        - v1.7.0
        <Snip>
```

Use the following command to start a Minikube cluster running Kubernetes version 1.7.0.

```
$ minikube start \
  --vm-driver=xhyve \
  --kubernetes-version="v1.7.0"

  Starting local Kubernetes cluster...
  Starting VM...
```

Run another `kubectl get nodes` command to verify the version.

```
$ kubectl get nodes
NAME       STATUS   AGE   VERSION
minikube   Ready    1m    v1.7.0
```

Bingo!

So that's Minikube on Mac. Now let's look at it on Windows 10.

Installing Minikube on Windows 10

In this section we'll show you how to use Minikube with Hyper-V as the virtual machine manager. Other options exist, but we will not be showing them here. We will also be using a PowerShell terminal opened with Administrative privileges.

Before installing Minikube, let's install the `kubectl` client. There are a couple of ways to do this:

1. Using the Chocolaty package manager
2. Downloading via your web browser

If you are using Chocolaty, you can install it with the following command.

```
> choco install kubernetes-cli
```

If you are not using Chocolaty, you can download it via your web browser.

Point your web browser to https://kubernetes.io/docs/tasks/tools/install-kubectl/ and click the `Windows` tab under the **Install kubectl binary via curl** heading. You will be given a link to download the latest version using your browser. You do not need to use `curl`.

Once the download is complete, copy the `kubectl.exe` file to a folder in your system's %PATH%.

Verify the installation with a `kubectl version` command.

```
> kubectl version --client=true --output=yaml
clientVersion:
  buildDate: 2017-09-28T22:57:57Z
  compiler: gc
  gitCommit: 6e937839ac04a38cac63e6a7a306c5d035fe7b0a
  gitTreeState: clean
  gitVersion: v1.8.0
  goVersion: go1.8.3
  major: "1"
  minor: "8"
  platform: windows/amd64
```

Now that you have `kubectl`, you can proceed to install Minikube for Windows.

1. Open a web browser to the Minikube Releases page on GitHub
    - https://github.com/kubernetes/minikube/releases
2. Download the 64-bit Windows installer (currently called `minikube-installer.exe`).
3. Start the installer and click through the wizard accepting the default options.
4. Make sure Hyper-V has an external vSwitch .

    Open Hyper-V Manager and go to `Virtual Switch Manager...`. If there is no Virtual Switch configured with the following two options, create a new one:

    - `Connection type = External network`
    - `Allow management operating system to share this network adapter`

    For the remainder of this section we will assume that you have Hyper-V configured with an external vSwitch called `external`. If you're has a different name, you will have to substitute your name in the following commands.

5. Verify the Minikube version with the following command.

```
> minikube version
minikube version: v0.22.3
```

6. Use the following command to start a local Minikube instance running Kubernetes version 1.8.0.

   You can use the `minikube get-k8s-versions` command to see a list of available Kubernetes versions.

   The following command assumes a Hyper-V vSwitch called `external`. You may have named yours differently.

```
> minikube start \
  --vm-driver=hyperv \
  --hyperv-virtual-switch="external" \
  --kubernetes-version="v1.8.0"
  --memory=4096

  Starting local Kubernetes cluster...
  Starting VM...
  139.09 MB / 139.09 MB [================] 100.00% 0s
  <Snip>
  Starting cluster components...
  Kubectl is now configured to use the cluster.
```

7. Verify the installation by checking the version of the Kubernetes master.

```
> kubectl version --output=yaml
clientVersion:
<Snip>
serverVersion:
  buildDate: 2017-10-17T15:09:55Z
  compiler: gc
  gitCommit: 0b9efaeb34a2fc51ff8e4d34ad9bc6375459c4a4
  gitTreeState: dirty
  gitVersion: v1.8.0
  goVersion: go1.8.3
  major: "1"
  minor: "8"
  platform: linux/amd64
```

   If the target machine actively refuses the network connection with a `Unable to connect to the server: dial tcp...` error. This is most likely a network related error. Make sure that your *external* vSiwtch is configured correctly, and that you specified it correctly with the `--hyperv-virtual-switch` flag. `kubectl` talks to Kubernetes inside the `minikube` Hyper-V VM over port 8443.

Congratulations! You've got a fully working Minikube cluster up and running on your Windows 10 PC.

You can now type `minikube` on the command line to see a full list of minikube sub-commands. A good one to try out might be `minikube dashboard` which will open the Minikube dashboard GUI in a new browser tab.

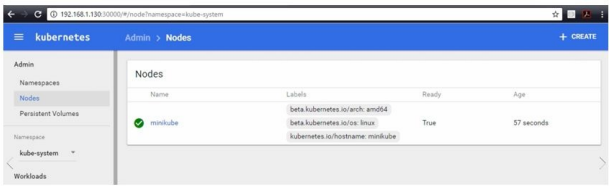


Figure 3.3

So that's Minikube! Probably the best way to spin up a simple Kubernetes cluster on your Mac or PC. But it's not for production!

Google Container Engine (GKE)

First up, that's not a typo in the title above. We shorten Google Container Engine to GKE not GCE. The first reason is that GKE is packaged **K**ubernetes running on the Google Cloud, so the K is for Kubernetes. The second reason is that GCE is already taken for Google Compute Engine.

Anyway, a quick bit of background to set the scene. GKE is layered on top of Google Compute Engine (GCE). GCE provides the low-level virtual machine instances and GKE lashes the Kubernetes and container magic on top.

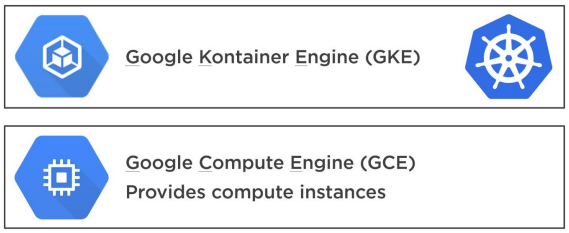


Figure 3.4

The whole raison d'être behind GKE is to make Kubernetes and container orchestration accessible and simple! Think of GKE as *Kubernetes as a Service* – all the goodness of a fully-featured, production-grade Kubernetes cluster pre-packaged ready for us consume. McDonalds anyone!?

Configuring GKE

To work with GKE you'll need an account on the Google Cloud with billing configured and a blank project setting up. These are all really easy to setup, so we won't spend time explaining them here - for the remainder of this section I'm assuming you already have an account with billing configured and a new project created.

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

1. From within the Console of your Google Cloud Platform (GCP) project, open the navigation pane on the left-hand side and select `Container Engine` from under the `COMPUTE` section. You may have to click the three horizontals bars at the top-left of the Console to make the navigation pane visible.
2. Make sure that `Container clusters` is selected in the left-hand navigation pane and click the `+ CREATE CLUSTER` button.

   This will start the wizard to create a new Kubernetes cluster.

3. Enter a `Name` and `Description` for the cluster.
4. Select the GCP `Zone` that you want to deploy the cluster to. At the time of writing, a cluster can only exist in a single zone.
5. Select the `Cluster Version`. This is the version of Kubernetes that will run on your master and nodes. You are limited to the versions available in the drop-down list.
6. Select the `Machine type` that you want your cluster nodes to be based on.
7. Under `Node image` chose `COS`. This is the Container Optimized OS image based on Chromium OS. It supersedes the older `container-vm` image.
8. Use the `Size` field to choose how many nodes you want in the cluster. This is the number of nodes in the cluster and does not include the master/control plane which is built and maintained for you in the background.
9. Leave all other options as their defaults and click `Create`.

   You can also click the `More` link to see a long list of other options you can customize. It's worth taking a look at these but we won't be discussing them in this book.

Your cluster will now be created!

Exploring GKE

Now that you have a cluster, it's time to have a quick look at it.

Make sure you're logged on to the GCP Console and are viewing `Container clusters` under `Container Engine`.

The `Container clusters` page shows a high-level overview of the container clusters you have in your project. Figure 3.5 shows a single cluster with 3 nodes running version 1.7.6 of Kubernetes.

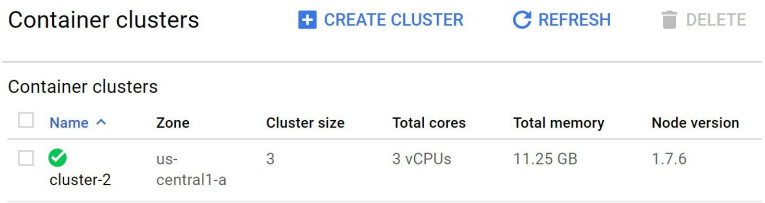


Figure 3.5

Click the cluster name to drill in to more detail. Figure 3.6 shows a screenshot of some of the detail you can view.

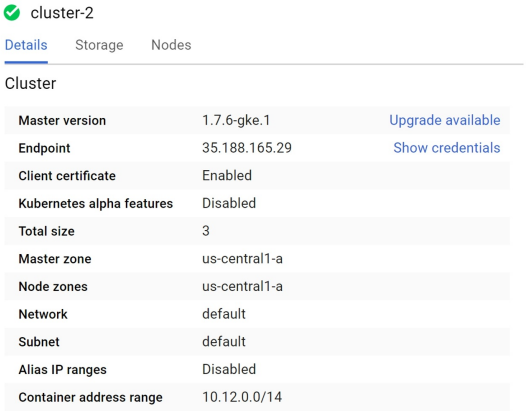


Figure 3.6

Clicking the `> CONNECT` console icon towards the top right of the web UI (not shown in the Figure above) gives you the option to open a Cloud Shell session that you can inspect and manage your cluster from.

Open a Cloud Shell session and run the `gcloud container clusters list` command. This will show you the same basic cluster information that you saw on the `Container clusters` web view.

```
$ gcloud container clusters list
NAME   ZONE       MASTER  MASTER_IP    NODE_VER  NODES  STATUS
clus2  us-centr   1.7.6   35.188...    1.7.6     3      RUNNING
```

The output above has been clipped to make it more readable.

If you click `> CONNECT` and choose the `Show gcloud command` option, you will be presented with the commands needed to configure kubectl to talk to the cluster. Copy and paste those commands into the Cloud Shell session so that you can manage your Kubernetes cluster with `kubectl`.

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

```
$ kubectl get nodes
NAME             STATUS     AGE    VERSION
gke-cluster...   Ready      5m     v1.7.6
gke-cluster...   Ready      6m     v1.7.6
gke-cluster...   Ready      6m     v1.7.6
```

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

Installing Kubernetes in AWS

There's more than one way to install Kubernetes in AWS. We're going to show you how to do it using the **kops** tool that is under active development. So, expect some of the specifics to change over time.

> **Note:** `kops` is short for Kubernetes Operations. At the time of writing, the only provider it supports is AWS. However, support for more platforms is in development. At the time of writing there is also no `kops` binary for Windows.

To follow along with the rest of this section you'll need a decent understanding of AWS as well as all of the following:

- `kubectl`
- the `kops` binary for your OS
- the `awscli` tool
- the credentials of an AWS account with the following permissions:
  - AmazonEC2FullAccess
  - AmazonRoute53FullAccess
  - AmazonS3FullAccess
  - IAMFullAccess
  - AmazonVPCFullAccess

The examples below are from a Linux machine, but it works the same from a Mac (and probably Windows in the future). The example also uses a publicly

routable DNS domain called `tf1.com` that is hosted with a 3rd party provider such as GoDaddy. Within that domain I have a subdomain called `k8s` delegated Amazon Route53. You will need to use a different domain in your own lab.

Download and install `kubectl`

For Mac, the download and installation is a simple `brew install kubectl`.

The following procedure is for a Linux machine.

1. Use the following command to download the latest `kubectl` binary.

   ```
   $ curl -LO https://storage.googleapis.com/kubernetes-release/release/$
   -s https://storage.googleapis.com/kubernetes-release/release/stable.tx
   /linux/amd64/kubectl
   ```

   The command above is a single command, but is quite long and will wrap over multiple lines in the book and insert backslashes at the end of each printed line `\`. These backslashes are not part of the command. It will download the `kubectl` binary to your home directory.

2. Make the downloaded binary executable.

   ```
   $ chmod +x ./kubectl
   ```

3. Add it to a directory in your PATH

   ```
   $ mv ./kubectl /usr/local/bin/kubectl
   ```

4. Run a `kubectl` command to make sure it's installed and working.

Download and install `kops`

1. Download the `kops` binary with the following `curl` command (the command should be issued on one line and have no backslashes `\` in it).

   ```
   $ curl -LO https://github.com/kubernetes/kops/releases/download/1.5.3/
   inux-amd64
   ```

2. Make the downloaded binary executable.

   ```
   $ chmod +x kops-linux-amd64
   ```

3. Move it to a directory in your path.

```
$ mv kops-linux-amd64 /usr/local/bin/kops
```

4. Run a `kops version` command to verify the installation.

```
$ kops version
Version 1.5.3 (git-46364f6)
```

Install and configure the AWS CLI

The example below shows how to install the AWS CLI from the default app repos used by Ubuntu 16.04.

1. Run the following command to install the AWS CLI

```
$ sudo apt-get install awscli -y
```

2. Run the `aws configure` command to configure your instance of the AWS CLI

   You will need the credentials of an AWS IAM account with *AmazonEC2FullAccess*, *AmazonRoute53FullAccess*, *AmazonS3FullAccess*, *IAMFullAccess*, and *AmazonVPCFullAccess* to complete this step.

```
$ aws configure
AWS Access Key ID [None]: **************
AWS Secret Access Key [None]: **************
Default region name [None]: enter-your-region-here
Default output format [None]:
$
```

3. Create a new S3 bucket for kops to store config and state information.

   The domain name you use in the example below will be different if you are following along. The `cluster1` is the name of the cluster we will create, `k8s` is the subdomain delegated to AWS Route53, and `tf1.com` is the public domain I have hosted with a 3rd party. `tf1.com` is fictional and only being used in these examples to keep the command line arguments short.

```
$ aws s3 mb s3://cluster1.k8s.tf1.com
make_bucket: cluster1.k8s.tf1.com
```

4. List your S3 buckets and `grep` for the name of the bucket you created. This will prove that the bucket created successfully.

```
$ aws s3 ls | grep k8s
2017-06-02 13:09:11 cluster1.k8s.tf1.com
```

5. Tell **kops** where to find its config and state.

```
$ export KOPS_STATE_STORE=s3://cluster1.k8s.tf1.com
```

6. Make your AWS SSH key available to **kops**.

   The command below will copy the keys in your `authorized_keys` file to a new file in `~/.ssh/id_rsa.pub`. This is because **kops** expects to find your AWS SSH key in a file called id_rsa.pub in your profile's hidden `ssh` directory.

```
$ cp ~/.ssh/authorized_keys ~/.ssh/id_rsa.pub
```

7. Create a new cluster with the following `kops create cluster` command.

```
$ kops create cluster \
  --cloud=aws --zones=eu-west-1b \
  --dns-zone=k8s.tf1.com \
  --name cluster1.k8s.tf1.com  --yes
```

   The command is broken down as follows. `kops create cluster` tells **kops** to create a new cluster. `--cloud=aws` tells kops to create this cluster in AWS using the AWS provider. `--zones=eu-west-1b` tells **kops** to create cluster in the eu-west-1b zone. We tell it to use the delegated zone with the `--dns-zone=k8s.tf1.com` flag. We name the cluster with the `--name` flag, and the `--yes` flag tells **kops** to go ahead and deploy the cluster. If you omit the `--yes` flag a cluster config will be created but it will not be deployed.

   It may take a few minutes for the cluster to deploy. This is because **kops** is doing all the hard work creating the AWS resources required to build the cluster. This includes things like a VPC, EC2 instances, launch configs, auto scaling groups, security groups etc. After it has built the AWS infrastructure it also has to build the Kubernetes cluster.

8. Once the cluster is deployed you can validate it with the `kops validate cluster` command.

```
$ kops validate cluster
Using cluster from kubectl context: cluster1.k8s.tf1.com

INSTANCE GROUPS
```

```
NAME      ROLE     MACHINETYPE  MIN  MAX  SUBNETS
master..  Master   m3.medium    1    1    eu-west-1b
nodes     Node     t2.medium    2    2    eu-west-1b

NODE STATUS
NAME              ROLE      READY
ip-172-20-38..    node      True
ip-172-20-58..    master    True
ip-172-20-59..    node      True

Your cluster cluster1.k8s.tf1.com is ready
```

Congratulations! You now know how to create a Kubernetes cluster in AWS using the `kops` tool.

Now that your cluster is up and running you can issue `kubectl` commands against it, and it might be worth having a poke around in the AWS console to see some of the resources that `kops` created.

Deleting a Kubernetes cluster in AWS with `kops`

To delete the cluster, you just created you can use the `kops delete cluster` command.

The command below will delete the cluster we created earlier. It will also delete all of the AWS resources created for the cluster.

```
$ kops delete cluster --name=cluster1.k8s.tf1.com --yes
```

Manually installing Kubernetes

In this section, we'll see how to use the `kubeadm` command to perform a manual install of Kubernetes. `kubeadm` is a core Kubernetes project tool that's pretty new at the time I'm writing this book. However, it's got a promising future and the maintainers of the project are keen not to mess about with command line features etc. So, the commands we shown here shouldn't change too much in the future (hopefully).

The examples in this section are based on Ubuntu 16.04. If you are using a different Linux distro some of the commands in the pre-reqs section will be different. However, the procedure we're showing can be used to install Kubernetes on your laptop, in your data center, or even in the cloud.

We'll be walking through a simple example using three Ubuntu 16.04 machines configured as one master and two nodes as shown in Figure 3.7. It

doesn't matter if these machines are VMs on your laptop, bare metal servers in your data center, or instances in the public cloud - `kubeadm` doesn't care!

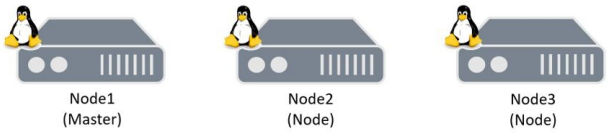


Figure 3.7

The high-level plan will be to initialize a new cluster with **node1** as the master. We'll create an overlay network, then add **node2** and **node3** as nodes. All three nodes will get:

- Docker
- `kubeadm`
- kubelet
- `kubectl`
- The CNI

Docker is the container runtime, `kubeadm` is the tool we'll use the build the cluster, **kubelet** is the Kubernetes node agent, `kubectl` is the standard Kubernetes client, and CNI (Container Network Interface) installs support for CNI networking.

Pre-requisites

The following commands are specific to Ubuntu 16.04 and need to be ran on **all three nodes**.

```
apt-get update && apt-get install -y apt-transport-https

curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg |
apt-key add -

cat <<EOF >/etc/apt/sources.list.d/kubernetes.list deb
http://apt.kubernetes.io/ kubernetes-xenial main EOF

apt-get update
```

These commands set things up (on Ubuntu 16.04) so that we can install the right packages from the right repos.

1. Install Docker, kubeadm, kubectl, kubelet, and the CNI.

   ```
   $ apt-get install docker.io kubeadm kubectl kubelet kubernetes-cni
   Reading package lists... Done
   ```

```
Building dependency tree
<SNIP>
```

2. Run the same command again to see version info

```
$ apt-get install docker.io kubeadm kubectl kubelet kubernetes-cni
<SNIP>
docker.io is already at the latest version (1.12.6-0ubuntu1~16.04.1).
kubeadm is already at the latest version (1.6.1-00).
kubectl is already at the latest version (1.6.1-00).
kubelet is already at the latest version (1.6.1-00).
kubernetes-cni is already at the latest version (0.5.1-00).
```

That's the pre-reqs done.

Initialize a new cluster

Initializing a new Kubernetes cluster with `kubeadm` is as simple as typing `kubeadm init`.

```
$ kubeadm init
<SNIP>
Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run (as a regular user):

sudo cp /etc/kubernetes/admin.conf $HOME/
sudo chown $(id -u):$(id -g) $HOME/admin.conf
export KUBECONFIG=$HOME/admin.conf

<SNIP>
You can join any number of machines by running the following on each nod
as root:

kubeadm join --token b90685.bd53aca93b758efc 172.31.32.74:6443
```

The command pulls all required images and creates all the required system Pods etc. The output of the command gives you a few more commands that you need to run to set your local environment. It also gives you the `kubeadm join` command and token required to add additional nodes.

Congratulations! That's a brand-new Kubernetes cluster created comprising a single master.

Complete the process by running the commands listed in the output of the `kubeadm init` command shown above.

```
$ sudo cp /etc/kubernetes/admin.conf $HOME/
$ sudo chown $(id -u):$(id -g) $HOME/admin.conf
```

```
$ export KUBECONFIG=$HOME/admin.conf
```

These commands may be different, or even no longer required in the future. However, they copy the Kubernetes config file form `/etc/kubernetes` into your home directory, change the ownership to you, and export an environment variable that tells Kubernetes where to find its config. In the real world, you may want to make the environment variable a permanent part of your shell profile.

Verify that the cluster created successfully with a `kubectl` command.

```
$ kubectl get nodes
NAME    STATUS      AGE   VERSION
node1   NotReady    1m    v1.6.1
```

Run another `kubectl` command to find the reason why the cluster `STATUS` is showing as `NotReady`.

```
$ kubectl get pods --all-namespaces
NAMESPACE    NAME                   READY   STATUS     RESTARTS  AGE
kube-system  kube-apiserver-node1   1/1     Running    0         1m
kube-system  kube-dns-39134729...   0/3     Pending    0         1m
kube-system  kube-proxy-bp4hc       1/1     Running    0         1m
kube-system  kube-scheduler-node1   1/1     Running    0         1m
```

This command shows all pods in all namespaces - this includes system pods in the system (kube-system) namespace.

As we can see, none of the `kube-dns` pods are running. This is because we haven't created a pod network yet.

Create a pod network. The example below creates a multi-host overlay network provided by Weaveworks. However, other options exist, and you do not have to go with the example shown here.

```
$ kubectl apply --filename https://git.io/weave-kube-1.6
clusterrole "weave-net" created
serviceaccount "weave-net" created
clusterrolebinding "weave-net" created
daemonset "weave-net" created
```

Be sure to use the right version of the Weaveworks config file. Kubernetes v1.6.0 introduced some significant changes in this area meaning that older config files will not work with Kubernetes version 1.6 and higher.

Check if the cluster status has changed from `NotReady` to `Ready`.

```
$ kubectl get nodes
NAME    STATUS      AGE   VERSION
node1   Ready       4m    v1.6.1
```

Great, the cluster is ready and the DNS pods will now be running.

Now that the cluster is up and running it's time to add some nodes.

To do this we need the cluster's join token. You might remember that this was provided as part of the output when the cluster was first initialized. Scroll back up to that output, copy the `kubeadm join` command to the clipboard and then run it on **node2** and **node3**.

> **Note:** The following must be performed on **node2** and **node3** and you must have already installed the pre-reqs (Docker, kubeadm, kubectl, kubelet, CNI) on these nodes.

```
node2$ kubeadm join --token b90685.bd53aca93b758efc 172.31.32.74:6443
<SNIP>
Node join complete:
* Certificate signing request sent to master and response received
* Kubelet informed of new secure connection details.
```

Repeat the command on **node3**.

Make sure that the nodes successfully registered by running another `kubectl get nodes` on the master.

```
$ kubectl get nodes
NAME    STATUS      AGE   VERSION
node1   Ready       7m    v1.6.1
node2   Ready       1m    v1.6.1
node3   Ready       1m    v1.6.1
```

Congratulations! You've manually built a 3-node cluster using `kubeadm`. But remember that it's running a single master without H/A.

Chapter summary

In this chapter we learned how to install Kubernetes in various ways and on various platforms. We learned how to use Minikube to quickly spin up a development environment on your Mac or Windows laptop. We learned how to spin up a managed Kubernetes cluster in the Google Cloud using Google Container Engine (GKE). Then we looked at how to use the `kops` tool to spin

up a cluster in AWS using the AWS provider. We finished the chapter seeing how to perform a manual install using the `kubeadm` tool.

There are obviously other ways and places we can install Kubernetes. But the chapter is already long enough and I've pulled out way too much of my hair already :-D

4: Working with Pods

We'll split this chapter in to two main parts:

- Theory
- Hands-on

So let's crack on with the theory.

Pod theory

In the VMware and Hyper-V worlds, the atomic unit of scheduling is the Virtual Machine (VM). Deploying workloads in these environments means stamping them out in VMs.

In the Docker world, the atomic unit is the container. Even though Docker now supports services and stacks, the smallest and most atomic unit of scheduling is still the container.

In the Kubernetes world, the atomic unit is the *Pod*!

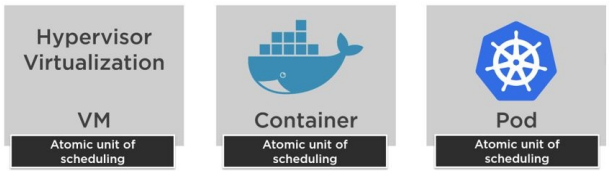


Figure 4.1

Be sure to take that and mark it in your brain as important - Virtualization does VM's, Docker does containers, and **Kubernetes does Pods!**

Pods vs containers

At a high-level, Pods sit somewhere in between a container and a VM. They're bigger, and arguably more high level than a container, but they're a lot smaller than a VM.

Digging a bit deeper, a Pod is just a shared execution environment for one or more containers. More often than not, a Pod only has one container. But multi-container Pods are definitely a thing. For example, multi-container Pods are excellent for co-scheduling tightly-coupled containers - containers that

share resources and wouldn't work well if they were scheduled on different nodes in the cluster.

Pods: the canonical example

The example we all use when describing the advantages of multi-container Pods is a web server that has a file synchronizer.

In this example we have two clear concerns:

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

Micro-service design patterns dictate that we should keep these concerns separate. One way to do that is deploy them in separate containers - one container for the web service, another container for the file-sync service.

This approach has a lot of advantages.

Instead of building a mini-monolith where a single container runs the web service *and* file-sync service, we are building two micro-services, each with its own separate concern. This means we can have different teams responsible for each of the two services. We can scale each service independently. We can also update them independently. And if the container running the file-sync service fails, the web service can stay up (though it may end up serving stale content).

However, despite designing them as separate autonomous services, it makes sense to run them side-by-side on the same node. Figure 4.2 shows the basic architecture of this example.

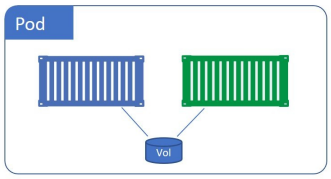


Figure 4.2

The simplest way to implement the shared volume is to schedule the two containers on the same node. Pods let us do this. By running the web service container and the file-sync container in the same Pod, we ensure they are deployed to the same node. We also give them a shared operating environment where both contains can access the same shared memory and shared volumes etc. More on all of this later.

How do we deploy Pods

To deploy a Pod to a Kubernetes cluster we define it in a *manifest file* and then POST that manifest file to the API server. The master examines it, records it in the cluster store, and the scheduler deploys the Pod to a healthy node with enough available resources. Whether or not the Pod has one or more containers is defined in the manifest file.

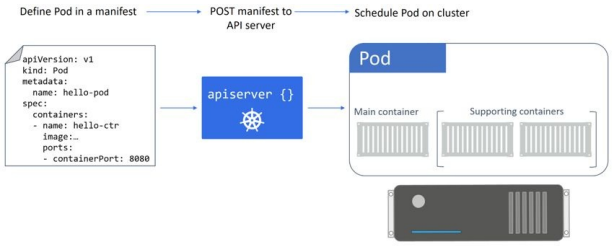


Figure 4.3

Let's dig a bit deeper…

The anatomy of a Pod

At the highest level, a Pod is a shared execution environment for one or more containers. "shared execution environment" means that all container in a Pod share the IP address, hostname, sockets, memory, volumes, and more, of the Pod they run *inside* of.

Under the hood, a Pod is something called a **pause container**. That's right, a Pod just a fancy name for a special container!

This means the containers that run inside of Pods are really containers running inside of containers! For more information, go and watch "Inception" by Christopher Nolan, starring Leonardo DiCaprio :-D

Seriously though, the Pod (pause container) is used to create a common set of namespaces that all other containers in the Pod will inherit and share. These include:

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

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

Let's look a bit closer at how this affects Pod and container networking.

Each Pod creates its own network namespace - a single IP address, a single range of ports, and a single routing table. This is true even if the Pod is a multi-container Pod - each container in a Pod shares the Pods IP, range of ports, and routing table.

Figure 4.4 shows two Pods, each with its own IP. Even though one of the Pods is hosting two containers, it still only gets a single IP.

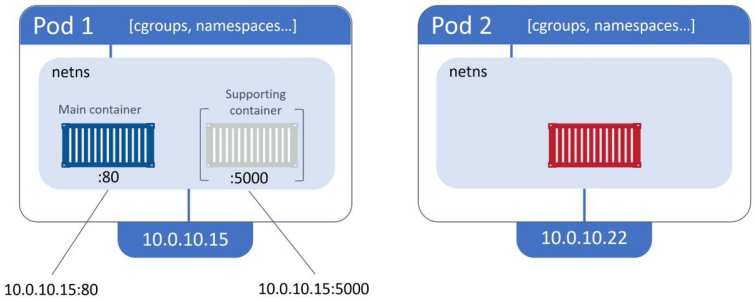


Figure 4.4

In this example, we can access the individual containers in Pod 1 using a combination of the Pod IP, coupled with the containers individual port number (80 and 5000).

One last time (apologies if it feels like over-repeating myself), each container in a Pod shares the **Pod's** entire network namespace. This includes; `localhost` adapter, port range, routing table, and more.

But as we've said, it's more than just networking. All containers in a Pod have access to the same volumes, the same memory, the same IPC sockets and more. Technically speaking, the Pod (pause container) holds all the namespaces, any containers the in Pod inherit and share these Pod namespaces.

This *Pod networking* model makes *inter-Pod* communication really simple. Every Pod in the cluster has its own IP addresses that's fully routable on the Pod overlay network. If you read the chapter on installing Kubernetes, you'll have seen how we created a Pod network at the end of the *Play with Kubernetes*, and *Manual install* sections. Because every Pod gets its own routable IP, every Pod can talk directly to every other Pod. No need to mess around with things like nasty port mappings!

Figure 4.5 Inter-pod communication

Intra-pod communication - where two containers in the same Pod need to communicate - can happen via the Pods `localhost` interface.

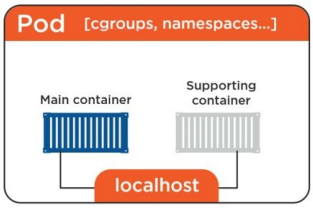


Figure 4.6 Intra-pod communication

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

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

Pods and cgroups

At a high level, Control Groups (cgroups) are what stop individual containers from consuming all of the available CPU, RAM and IOPS on a node. I suppose we could say that they "police" resource usage.

Individual containers have their own cgroup limits.

This means it's possible for two containers in a single Pod to have their own set of cgroup limits. This is a powerful and flexible model. If we assume the canonical multi-container Pod example from earlier in the chapter, we could set a cgroup limit on the file sync container so that it had access to less resources than the web service container, and was unable to starve the web service container of CPU and memory.

Atomic deployment of Pods

When we deploy a Pod, it's an all or nothing job. There's no such thing as a partially-deployed Pod.

For example, either: everything inside of a Pod coms up and the Pod becomes available, **OR**, everything doesn't come up and the Pod fails. For example, you can never have a situation where you have a multi-container Pod with one of its containers up and accessible but the other container in a failed state! That's not how it works. Nothing in the Pod is made available until the entire Pod is up. Then, once all Pod resources are ready, the Pod is made available.

It's also important to know that any given Pod can only be running on a single node. This is the same as containers or VMs - you can't have part of a Pod on one node and another part of it on another node. One Pod gets scheduled to one node!

This principle is at the heart of the Pods co-scheduling ability. If you *need* two containers/services to run on the same node, deploy them in the same Pod.

Pod lifecycle

The lifecycle of typical Pod goes something like this. You define it in a YAML or JSON manifest file. Then you throw that manifest at the API server and the Pod it defines gets scheduled to a healthy node. Once it's scheduled to a node, it goes into the *pending* state while the node downloads images and fires up any containers. The Pod remains in this *pending* state until all of its resources are up and ready. Once everything's up and ready, the Pod enters the *running* state. Once it's completed its task in life, it gets terminated and enters the *succeeded* state.

If a Pod fails, it is not rescheduled! For this reason, we rarely deploy them directly. It is far more common to deploy them via higher-level constructs such as ReplicaSets, DaemonSets, and Deployments.

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

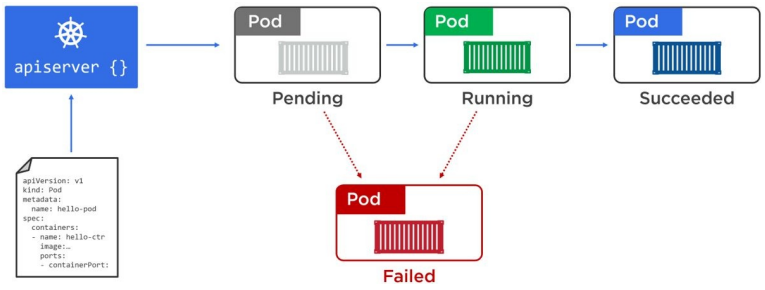


Figure 4.7 Pod lifecycle

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

This is one of the main reasons you should code your applications so that they don't store *state* in the Pod, and also why they shouldn't rely on individual Pod IPs etc.

Pod theory summary

1. Pods are the smallest unit of scheduling on Kubernetes
2. You can have more than one container in a Pod. Single-container Pods are the most common type, but multi-container Pods are ideal for containers that need to be tightly coupled - maybe they need to share memory or volumes.
3. Pods get scheduled on nodes – you can't schedule a single Pod to span multiple nodes.
4. They get defined declaratively in a manifest file that we POST to the API server and let the scheduler assign them to nodes.

Hands-on with Pods

It's time to see Pods in action!

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

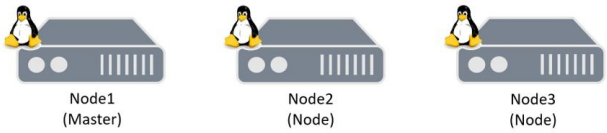


Figure 4.8

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

Following the Kubernetes mantra of *composable infrastructure*, we define Pods in manifest files, POST them to the API server, and let the scheduler take care of instantiating them on the cluster.

Pod manifest files

For the examples in this chapter we're going to use the following Pod manifest (called pod.yml):

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

Although this file is in YAML format, you can also use JSON.

Straight away we can see 4 top-level resources that we're going to need to understand: `.apiVersion`, `.kind`, `.metadata`, and `.spec`.

> **Note:** In this chapter we're only going to step lightly through the sections and fields in the manifest file. This is so that you don't get overwhelmed. In the chapter on *ReplicaSets* we're going to give a more comprehensive dive into manifest files.

As the name suggests, the `.apiVersion` field specifies the version of the API that we'll be using. `v1` has been around since 2015, includes an extensive Pod schema, and it's stable. If you're working with some of the newer constructs such as *ReplicaSets* and *Deployments*, you'll have to specify a newer version of the API.

Next up, the `.kind` field tells Kubernetes what kind of object to deploy - in this example we're telling it to Deploy a Pod. As we progress through the book we'll populate this with various different values – *Services*, *Deployments*, *DaemonSets*, etc.

Then we define some `.metadata`. In this example, we're naming the Pod "hello-pod" and giving it a couple of labels. Labels are simple key-value pairs, bit they're insanely powerful! We'll talk more about labels later as we build our Kubernetes knowledge step-by-step.

Last but not least, we've got the `.spec` section. This is where we define what's *in* the Pod. In this example, we're defining a single-container Pod, calling it "hello-ctr", basing it on the `nigelpoulton/k8sbook:latest` image, and we're telling it to expose port 8080.

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

Manifest files: Empathy as Code

Quick side-step.

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

- Helping speed-up the on-boarding process for new team members
- Helping bridge the gap between developers and operations

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

If you have a problem with developers not clearly-stating their application's requirements to operations, get them to use Kubernetes and Docker. As they describe their applications through Dockerfiles and Kubernetes manifests, these can then be used by operations staff to understand how the application works and what it requires of the environment.

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

While describing config files like these as *empathy as code* might be a bit of a stretch, there is some merit to the concept. They definitely help in the ways previously mentioned.

Deploying Pods form a manifest file

If you're following along with the examples, save the following manifest file as `pod.yml` in your current directory.

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

Use the following `kubectl` command to POST the manifest to the API server and deploy a Pod from it.

```
$ kubectl create -f pod.yml
pod "hello-pod" created
```

Although the Pod is showing as created, it might not be fully deployed on the cluster yet. This is because it can take time to pull the required image to start the container inside the Pod.

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

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

We can see that the container is still being created - no doubt waiting for the image to be pulled from Docker Hub.

You can use the `kubectl describe pods hello-pod` command to drill into more detail.

If you run the `kubectl get pods` command again, after a short wait, you should see the `STATUS` as "Running"

```
$ kubectl get pods
NAME        READY    STATUS     RESTARTS   AGE
hello-pod   1/1      Running    0          2m
```

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

Introspecting running Pods

As good as the `kubectl get pods` command is, it's a bit light on the amount of information it divulges. Not to worry though, there's plenty of options for deeper introspection. Let's take a look at some.

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

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

The `-o yaml` and `-o json` flags take things to the next level! Both return full copies of the Pod manifest from the cluster store.

The command below shows a snipped version of a `kubectl get pods -o json` command.

```
$ kubectl get pods -o json hello-pod
{
    "apiVersion": "v1",
    "items": [
        {
            "apiVersion": "v1",
            "kind": "Pod",
            "metadata": {
                "creationTimestamp": "2017-10-21T15:28:46Z",
                "labels": {
                    "version": "v1",
                    "zone": "prod"
                },
<Snip>
    "containerStatuses": [
        {
            "containerID": "docker://5ad6319...f3907160",
            "image": "nigelpoulton/k8sbook:latest",
            "imageID": "docker-pullable://nigelpoulton/k8sbook...",
            "lastState": {},
            "name": "hello-ctr",
            "ready": true,
            "restartCount": 0,
            "state": {
                "running": {
                    "startedAt": "2017-10-21T15:29:02Z"
                }
            }
```

```
        }
<Snip>
```

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

Two main sources:

- The Kubernetes Pod object contains a lot of properties - far more than we defined in the manifest. Those that you don't specify are automatically expanded with default values by Kubernetes.
- When you run a `kubectl get pods` with `-o yaml` or `-o json`, you get the Pods *current state* as well as its *desired state*. However, when you define a Pod in a manifest file, you only declare the desired state.

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

```
$ kubectl describe pods hello-pod
Name:           hello-pod
Namespace:      default
Node:           node2/10.0.48.4
Start Time:     Sat, 21 Oct 2017 15:28:46 +0000
Labels:         version=v1
                zone=prod
Annotations:    <none>
Status:         Running
IP:             10.44.0.1
Containers:
  hello-ctr:
    Container ID:       docker://5ad63194...bcf3907160
    Port:               8080/TCP
    State:              Running
      Started:          Sat, 21 Oct 2017 15:29:02 +0000
    Ready:              True
    Restart Count:      0
    Environment:        <none>
Conditions:
  Type          Status
  Initialized   True
  Ready         True
  PodScheduled  True
Volumes:
  default-token-xsd49:
    Type:       Secret (a volume populated by a Secret)
    Events:
      1stSeen  From                Type    Reason   Message
      -------  -----               ------  ------   -------
      1m       default-scheduler   Normal  Sched    Success
```

```
  1m        kubelet, node2      Normal  MountVol succeeded
  1m        kubelet, node2      Normal  Pulling  pulling
```

The output above has been snipped to better fit the page.

Another way to introspect a running Pod is to log into it or execute commands in it. We do this with the `kubectl exec` command. The example below shows how to execute a `ps aux` command in the first container in the `hello-pod` Pod.

```
$ kubectl exec hello-pod ps aux
USER  PID %CPU %MEM    VSZ   START  TIME COMMAND
root  1  0.0  0.0     1024   15:28  0:00 /pause
root  7  0.0  0.0    11356   15:29  0:00 /bin/sh -c cd /src && node ./app.j
root  13  0.0  0.0  717820   15:29  0:00 node ./app.js
root  22  0.0  0.0   13376   19:13  0:00 ps aux
```

You can log-in to the first container in the Pod with the following command. Once inside the container you can execute normal commands (as long as the command binaries are installed in the container).

```
$ kubectl exec -it hello-pod sh
sh-4.1 # curl localhost:8080
<html><head><title>Pluralsight Rocks</title><link rel="stylesheet" href="ht
p://netdna.bootstrapcdn.com/bootstrap/3.1.1/css/bootstrap.min.css"/></head>
body><div class="container"><div class="jumbotron"><h1>Yo Pluralsighters!!!
/h1><p>Click the button below to head over to my podcast...</p><p> <a href=
http://intechwetrustpodcast.com" class="btn btn-primary">Podcast</a></p><p>
/p></div></div></body></html>
```

The `-it' flags make the` exec` session interactive and connect STDIN and STDOUT on your terminal windows with STDIN and STDOUT inside the container. When the command completes executing your shell prompt will change, indicating that you are now in a shell running inside the container.

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

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

Chapter Summary

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

The best way to deploy a Pod is declaratively using a YAML or JSON manifest file. We use the `kubectl` command to POST the manifest to the API server, it gets stored in the cluster store and converted into a PodSpec that gets scheduled onto a healthy cluster node with enough available resources.

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

If a Pod fails, it is not automatically rescheduled. Because of this, we usually deploy them via higher-level constructs such as ReplicaSets, Deployments, and DaemonSets. These higher-level constructs add things like self-healing and roll-backs, and are at the heart of what makes Kubernetes so powerful.

5: ReplicaSets

Pods are the basic building-block in Kubernetes. But if we deploy them directly, all we get is a single *vulnerable* Pod. We don't get any self-healing or scalability. So, if that Pod fails, it's game over!

Because of this, we almost always deploy Pods via higher-level objects like *ReplicaSets* and *Deployments*. These bring all the goodness of self-healing, scalability, and more.

> You may hear people use the terms *ReplicationController* and *ReplicaSet* interchangeably. This is because they do the same thing. However, *ReplicaSets* are the future, and *ReplicationControllers* are being phased out.

One final thing before moving on. More often than not, you're going to deploy your applications via *Deployments* rather than ReplicaSets. However, Deployments build on top of ReplicaSets, so it's vital that you understand what we're going to cover in this chapter. Don't skip it.

We'll split the remainder of the chapter in to two main parts:

- Theory
- Hands-on

ReplicaSet theory

Let's start out with the high-level stuff.

Pods wrap around containers and bring a bunch of benefits: co-location, shared volumes, simple networking, secrets, and more. In a similar way, ReplicaSets wrap around Pods and bring a host of benefits. See Figure 5.1.

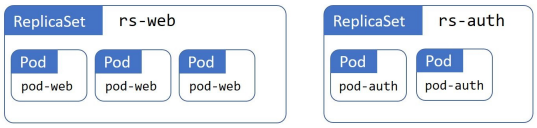


Figure 5.1

Before getting into the benefits, we need to make one thing about ReplicaSets crystal-clear - it is a one-to-one relationship between a ReplicaSet and a particular Pod!

As an example, assume you have a micro-service app comprising three services. You will more than likely build this so each service is contained within its own Pod. This will give you three Pods - one for each service. To manage these three Pods, you'll need three ReplicaSets. That's because a single ReplicaSet cannot manage more than one type of Pod. Look closer at Figure 5.1 and notice how the `rs-web` ReplicaSet is managing the `pod-web` Pods, and the `rs-auth` ReplicaSet is managing the `pod-auth` Pods. One ReplicaSet can only manage one Pod type.

ReplicaSet basics

A ReplicaSet defines two important things:

- The Pod template
- The desired number of replicas

The *Pod template* tells the ReplicaSet what type of Pods to deploy. Things like; container image, network ports, and even labels. The *desired number of replicas* tell the ReplicaSet how many of these Pods to deploy. Figure 5.2 shows a simple ReplicaSet manifest with the Pod template and number of replicas highlighted.

```
apiVersion: apps/v1beta2
kind: ReplicaSet
metadata:
  name: web-rs
spec:
  replicas: 8
  selector:
    matchLabels:
      app: hello-world
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-ctr
        image: nigelpoulton/k8sbook:latest
        ports:
        - containerPort: 8080
```

Figure 5.2

As an example, you might need four instances of a particular front-end web Pod constantly running. You'd use a ReplicaSet to accomplish this.

That, right there, is the basics of ReplicaSets! They make sure we always have the right number of the right Pod.

Let’s get into a bit of detail.

It’s all about the state

It’s critical that you understand three concepts that are fundamental to everything about Kubernetes (and therefore fundamental to ReplicaSets):

- Desired state
- Current state (sometimes called actual state)
- Declarative model

Desired state is what you **want**. *Current state* is what you’ve **got**. The aim-of-the-game is for the two to match - current state should always match desired state.

A *declarative model* lets you describe your desired state without having to get into the detail of *how* to get there.

The declarative model

We’ve got two competing models. The declarative model and the imperative model.

Declarative models are all about describing the end-goal. Imperative models are all about lists of commands that will get you to an end goal.

Let’s look at an example.

Assume you’ve got an application with two services: front-end and back-end. You’ve configured things so that you have a Pod template for the front-end service, and a separate Pod template for the back-end service. To meet expected demand, you always need 5 instances of the front-end Pod running, and 2 instances of the back-end Pod running.

Taking the declarative approach, you create a configuration file that describes your front-end and back-end Pods (what images and ports they use etc.). You also declare that you always need 5 front-end Pods and 2 back-end Pods. Simple! You give that file to Kubernetes and sit back while Kubernetes does all the heavy-lifting. But it doesn’t stop there… because you’ve told Kubernetes what your desired state is (always have 5 front-end Pods and 2 back-end Pods), Kubernetes watches the cluster and automatically recovers from failures!

The opposite of the declarative model is the imperative model. In the imperative model, there's no concept of desired state. Instead, you write a script with the all the steps and commands required to pull the front-end and back-end images, and then to create 5 front-end containers and 2 back-end containers. And you have to care about details such as which container runtime you're using (the commands to start `containerd` containers are different from the commands to start `rkt` containers etc.). This ends up being a lot more work, a lot more error-prone, and because it's not declaring a desired state, there's no self-healing. Boo!

Kubernetes supports both models, but it strongly prefers the declarative model. You should only use the imperative model in times of absolute emergency!

Reconciliation loops

Fundamental to all of this, are background reconciliation loops.

ReplicaSets implement a background reconciliation loop that is constantly monitoring the cluster. It's checking to see if current state matches desired state. If it doesn't, it wakes up the control-plane and Kubernetes takes steps to fix the situation.

To be crystal clear - **Kubernetes is constantly striving for harmony between current state and desired state!**

If the two diverge - may be desired state is 10 replicas, but only 8 are running - Kubernetes switches to red-alert, orders the control plane to battle-stations, and brings up two more replicas. And the best part… it does all of this without paging you at silly o'clock in the morning!

But it's not just failure scenarios. This very-same reconciliation loop allows us to scale up and down.

For example, imagine scaling an application from 3 Pod replicas to 5. To do this, we POST an updated version of the ReplicaSet manifest file to the API server where this gets registered as the application's new desired state. Then, the next time the reconciliation loop runs, it notices that desired state is 5 and current state is only 3, and the same routine is followed - Kubernetes calls red alert, and the control plane spins up two more replicas.

I think it's fair to say that *desired state* is the raison d'Ãªtre of Kubernetes, and the reconciliation loop is the beating heart of Kubernetes!

If you understand these concepts, you'll go far with Kubernetes.

ReplicaSet manifest files

As with Pods and other Kubernetes objects, ReplicaSets are deployed declaratively using YAML or JSON manifest files.

If you're new to Kubernetes, these manifest files can be scary. Don't worry though, we're about to lay them bare!

The YAML file below is a fully-working ReplicaSet manifest. It works with Kubernetes 1.8 and later, requiring the `apps/v1beta2` API. At a high-level, it deploys 10 replicas of a single-container Pod. The Pod is based on the `nigelpoulton/k8sbook:latest` image and listens on port 8080.

> For versions of Kubernetes prior to 1.8 you should replace the `apps/v1beta2` API with `extensions/v1beta1`. You should also remove the three consecutive lines starting with `selector`, `matchLabels`, and `app: hello-world`.

```
apiVersion: apps/v1beta2
kind: ReplicaSet
metadata:
  name: web-rs
spec:
  replicas: 10
  selector:
    matchLabels:
      app: hello-world
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-ctr
        image: nigelpoulton/k8sbook:latest
        ports:
        - containerPort: 8080
```

As with Pod manifests, ReplicaSet manifests have 4 top-level resources:

- apiVersion (`.apiVersion`)
- kind (`.kind`)
- metadata (`.metadata`)
- spec (`.spec`)

It’s common to use a simple *dot-notation* when referring to components of Kubernetes manifest. The four top-level resources shown above are `ReplicaSet.apiVersion`, `ReplicaSet.kind`, `ReplicaSet.metadata`, and `ReplicaSet.spec`. However, for brevity, we normally omit the `ReplicaSet` prefix and simply name them `.apiVersion`, `.kind`, `.metadata`, and `.spec`.

Figure 5.3 is a visual breakdown of the manifest into these 4 top-level resources.

Figure 5.3

Let’s step through the file in a bit more detail.

All Kubernetes manifest files start out by declaring the version of the API to use (`.apiVersion`). Different versions of the API use different object schemas. This means the schema for a ReplicaSet in the `apps/v1beta2` API will have different properties and fields than the schema for the ReplicaSet object in the `extensions/v1beta1` API. Starting with Kubernetes 1.8, you should use `apps/v1beta2` for deploying ReplicaSets. For older versions of Kubernetes, you can use the older `extensions/v1beta1`.

The `.kind` field tells Kubernetes which type of object is being defined. In this example it’s a ReplicaSet. Behind the scenes, this makes sure the manifest gets delivered to the ReplicaSet controller on the Kubernetes control plane. The `.kind` field is always a string and always written in CamelCase.

The only *required* field in the `.metadata` section is the name of the ReplicaSet. In this example, we’re naming it “web-rs”. The name of a ReplicaSet is a string, and it has to be unique in the namespace the ReplicaSet exists in (the `default` namespace in this instance). You can add labels and other metadata in this section.

The final top-level resource is the `.spec` section. This is where the main

configuration of the ReplicaSet lives.

In our example, `.spec.replicas` is declaring a desired state where 10 copies of the ReplicaSet's Pod will always be running. This property is at the heart of ReplicaSets. It is used when the ReplicaSet is initially deployed, as well as during scaling and healing operations. For example, when you scale a ReplicaSet, you update this field.

`.spec.selector` is new in the `apps/v1beta2` API in Kubernetes 1.8. It lists the label(s) that a Pod must have for it to be managed by the ReplicaSet. In this example, we're telling the ReplicaSet to manage all Pods in the cluster that have the label `app: hello-world`.

`.spec.template` is the Pod template. It describes the Pod that this ReplicaSet will deploy. This includes initial deployment, as well as scaling and self-healing operations. For example, when scaling or healing, the ReplicaSet deploys additional copies of the Pod defined in this section.

`.spec.template.metadata.labels` is the list of labels that the ReplicaSet will apply to Pods it creates. In this example, all Pods deployed by the ReplicaSet will be tagged with the `app: hello-world` label. If you're using the `apps/v1beta2` API, it is vital that this matches the ReplicaSet's label selector specified in `.spec.selector`. If it does not, the API will reject the ReplicaSet.

Finally, `.spec.template.spec` defines the containers that will run in the Pod. This example is a single-container Pod, but multi-container Pods have multiple `.spec.template.spec.containers` entries.

Kubernetes manifest files can be daunting. So don't worry if it's not crystal clear yet. Mark this section and come back to it later after we've written a and deployed a few manifests. It won't be long before it all makes sense!

Before moving on, it's worth noting that the Pod template section (`.spec.template`) in a ReplicaSet manifest is exactly the same as a standalone Pod manifest. This is because ReplicaSets create standard Pods using the standard Pod-related API calls and endpoints. The only difference is that ReplicaSets wrap Pods in a blanket of self-healing and scalability.

Pod custody

It's important to understand that **ReplicaSets and Pods are loosely coupled**. The only thing a Pod needs, for it to be managed by a ReplicaSet, is to match

the label selector of the ReplicaSet. This is powerful!

For example, a Pod can be managed by a ReplicaSet even if it wasn't created by the ReplicaSet - it just has to match the ReplicaSet's label selector. When a new ReplicaSet starts managing an existing Pod, we say the ReplicaSet has *adopted* the Pods.

Another example of the power of loose coupling is the ability to delete a ReplicaSet without deleting the Pods it's managing. For this to work, you have to pass a special flag to the delete command (we'll see this later).

That's enough theory, let's get hands-on!

Hands-on

ReplicaSets are not defined in the `v1` API. This means you need to use a beta API. If you're using Kubernetes 1.8 or higher, you should use the `apps/v1beta2` API. If you are using Kubernetes 1.7 or earlier, you should use the `extensions/v1beta1` API.

The remaining examples assume you are using Kubernetes 1.8 or higher, with the `apps/v1beta2` API or newer.

Use the `kubectl version` command to determine the version of Kubernetes you are running.

```
$ kubectl version --output=yaml
<Snip>
serverVersion:
  buildDate: 2017-10-17T15:09:55Z
  compiler: gc
  gitCommit: 0b9efaeb34a2fc51ff8e4d34ad9bc6375459c4a4
  gitTreeState: dirty
  gitVersion: v1.8.0
  goVersion: go1.8.3
  major: "1"
  minor: "8"
  platform: linux/amd64
```

The `major` and `minor` entries tell us that we are using version 1.8.

You can also use `kubectl api-versions` to see a list of supported API versions.

Create a ReplicaSet

Create a ReplicaSet manifest file called `rs.yaml` with the following YAML content.

> **Note:** Remember to use the `extensions/v1beta1` API if you are using a version of Kubernetes prior to 1.8. You should also remove the three lines that represent `.spec.selector.matchLabels`.

```
apiVersion: apps/v1beta2
kind: ReplicaSet
metadata:
  name: web-rs
spec:
  replicas: 10
  selector:
    matchLabels:
      app: hello-world
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-ctr
        image: nigelpoulton/k8sbook:latest
        ports:
        - containerPort: 8080
```

The first thing to note is that `.spec.replicas` is requesting 10 replicas. This gets recorded in the cluster store as part of the apps *desired state*.

Next, the ReplicaSet will select and manage all Pods in the cluster with the `app: hello-world` label (`.spec.selector`). This includes any existing Pods with that label.

Finally, the Pod template (`.spec.template`) describes a single-container Pod based on the `nigelpoulton/k8sbook:latest` image that listens on port 8080.

Use the `kubectl` command to POST the ReplicaSet manifest to the Kubernetes API server.

```
$ kubectl create -f rs.yaml
replicaset "web-rs" created
```

Notice that you do not have to tell `kubectl` the type of object you are creating. It can derive this from the `kind` field in the manifest file.

Note: If you are using the `apps/v1beta2` API, ReplicaSet manifests are POSTed to `/apis/apps/v1beta2/namespaces/{namespace}/replicasets`.

Introspect the ReplicaSet

You can use the usual `kubectl get` and `kubectl describe` commands to inspect the `web-rs` *ReplicaSet*.

```
$ kubectl get rs/web-rs
NAME    DESIRED  CURRENT  READY  CONTAIENRS  IMAGES  SELECTOR
web-rs  10       10       10     hello-ctr   nigel.. app=hello-world
```

See how we can shorten ReplicaSet to `rs` on the `kubectl` command line.

The output above shows that all 10 replicas are up and ready. It also shows that the ReplicaSet is selecting Pods that have the `app=hello=world` label.

Check that the Pods deployed by the ReplicaSet are tagged with the `app=hello-world` label.

```
$ kubectl get pods --show-labels
NAME           READY   STATUS    RESTARTS   AGE    LABELS
web-rs-4qdkj   1/1     Running   0          1m     app=hello-world
web-rs-jmzvx   1/1     Running   0          1m     app=hello-world
web-rs-pffbk   1/1     Running   0          1m     app=hello-world
web-rs-pvr4g   1/1     Running   0          1m     app=hello-world
web-rs-qhwsb   1/1     Running   0          1m     app=hello-world
web-rs-vfrtf   1/1     Running   0          1m     app=hello-world
web-rs-ww277   1/1     Running   0          1m     app=hello-world
web-rs-xhffd   1/1     Running   0          1m     app=hello-world
web-rs-zfk69   1/1     Running   0          1m     app=hello-world
web-rs-zpwcm   1/1     Running   0          1m     app=hello-world
```

Label selectors are simple and powerful, but we need to understand a few things about them:

1. If you are using the `apps/v1beta2` API, the ReplicaSet label selector (`.spec.selector`) **must** match the Pod template labels (`.spec.template.metadata.labels`). If it doesn't, Kubernetes will reject the ReplicaSet. If you do not specify a label selector for the ReplicaSet, it will default to the same value as the label specified in the Pod template.
2. `.spec.selector` is immutable as of `apps/v1beta2`. So no changing it after you deploy the ReplicaSet.

3. You need to be **very careful** when creating label selectors. Kubernetes does not check for clashes with label selectors in use by other objects (ReplicaSets, Deployments etc.). This means it is possible to have more than one ReplicaSet with the same label selector. In this scenario, the two ReplicaSets will fight over managing the Pods. **You do not want to go there!**

Expansion and current state

When we deployed the ReplicaSet with the rs.yaml file, we only specified a few of the required fields and properties. The ones we didn't specify were automatically populated with defaults (expanded).

We can see this in action by running a `kubectl get rs web-rs --output=yaml` command. This returns a copy of the ReplicaSet manifest as stored in the cluster store - including the default values that Kubernetes added, plus some information about current state.

The following snipped output shows some of the default values as well as the current state.

```
$ kubectl get rs web-rs --output=yaml
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  creationTimestamp: 2017-10-23T21:07:57Z
  generation: 1
  labels:
    app: hello-world
  name: web-rs
  namespace: default
  <Snip...>
status:
  fullyLabeledReplicas: 10
  observedGeneration: 1
  replicas: 10
```

We can see from the output above that one of the default values that Kubernetes added is `namespace: default`. The most recently observed state (current state) is displayed in the nested `.status` property.

> **Note:** The complete definition of an object (e.g. a ReplicaSet) stored in the cluster store contains the desired state *and* the current state. The desired state is expressed in the object's `.spec` field, and the current state is added by Kubernetes as the read-only `.status` field. This means that when Kubernetes is reconciling the cluster by bringing current state in-

line with desired state, we could re-phrase this to say *it is bringing the `status` in-line with the `spec`.

Loose coupling of ReplicaSets and Pods

Let's demonstrate the loosely-coupled nature of ReplicaSets and Pods.

Verify that we already have 10 Pods running as part of the `web-rs` ReplicaSet.

```
$ kubectl get rs/web-rs
NAME      DESIRED   CURRENT   READY     AGE
web-rs    10        10        2         3m
$
$
$ kubectl get pods --show-labels
NAME           READY   STATUS    RESTARTS   AGE    LABELS
web-rs-2rr2z   1/1     Running   0          5m     app=hello-world
web-rs-42zq7   1/1     Running   0          5m     app=hello-world
web-rs-64sxx   1/1     Running   0          5m     app=hello-world
web-rs-hwbtd   1/1     Running   0          5m     app=hello-world
web-rs-q6h4v   1/1     Running   0          5m     app=hello-world
web-rs-t8rqw   1/1     Running   0          5m     app=hello-world
web-rs-v4jc2   1/1     Running   0          5m     app=hello-world
web-rs-wntz9   1/1     Running   0          5m     app=hello-world
web-rs-wskgq   1/1     Running   0          5m     app=hello-world
web-rs-x4skb   1/1     Running   0          5m     app=hello-world
```

In the next step, we'll use delete the ReplicaSet with the `--cascade=false` flag. This will delete the ReplicaSet without deleting the 10 Pods it currently owns. We'll also run some commands to verify the operation.

```
$ kubectl delete rs/web-rs --cascade=false
replicaset "web-rs" deleted
$
$ kubectl get rs
No resources found
$
$ kubectl get pods --show-labels
NAME           READY   STATUS    RESTARTS   AGE    LABELS
web-rs-2rr2z   1/1     Running   0          6m     app=hello-world
web-rs-42zq7   1/1     Running   0          6m     app=hello-world
web-rs-64sxx   1/1     Running   0          6m     app=hello-world
web-rs-hwbtd   1/1     Running   0          6m     app=hello-world
web-rs-q6h4v   1/1     Running   0          6m     app=hello-world
web-rs-t8rqw   1/1     Running   0          6m     app=hello-world
web-rs-v4jc2   1/1     Running   0          6m     app=hello-world
web-rs-wntz9   1/1     Running   0          6m     app=hello-world
web-rs-wskgq   1/1     Running   0          6m     app=hello-world
web-rs-x4skb   1/1     Running   0          6m     app=hello-world
```

The commands above show that the ReplicaSet is deleted but the Pods it created/owned are still running. This demonstrates the loosely-coupled nature of ReplicaSets and Pods.

The 10 running Pods are now orphaned and have no self-healing capabilities. Don't worry though, we can easily create a brand-new ReplicaSet and have it *adopt* the Pods.

Use the following command to create a new ReplicaSet (with the same config as the old one). The ReplicaSet will request 10 replicas of the same Pod that is already running. When the manifest is POSTed to the API server, Kubernetes will query the cluster to see if there are any Pods already running that match the Pod template and label selector. In this example, it will find 10.

```
$ kubectl create -f rs.yaml
replicaset "web-rs" created
```

Take a moment to understand what just happened there. We POSTed a ReplicaSet manifest file to the API server requesting a *desired state* of 10 replicas. Kubernetes checked the *current state* of the cluster and found that there were 10 replicas of the required Pod already running. So it didn't create any new ones. The only thing it did was instantiate the ReplicaSet object so that we get self-healing and simple scalability.

Verify that the ReplicaSet is running and that there are still only 10 Pods running.

```
$ kubectl get rs/web-rs
NAME      DESIRED   CURRENT   READY     AGE
web-rs    10        10        10        13s
$
$ kubectl get pods --show-labels
NAME           READY  STATUS    RESTARTS   AGE    LABELS
web-rs-2rr2z   1/1    Running   0          39m    app=hello-world
web-rs-42zq7   1/1    Running   0          39m    app=hello-world
web-rs-64sxx   1/1    Running   0          39m    app=hello-world
web-rs-hwbtd   1/1    Running   0          39m    app=hello-world
web-rs-q6h4v   1/1    Running   0          39m    app=hello-world
web-rs-t8rqw   1/1    Running   0          39m    app=hello-world
web-rs-v4jc2   1/1    Running   0          39m    app=hello-world
web-rs-wntz9   1/1    Running   0          39m    app=hello-world
web-rs-wskgq   1/1    Running   0          39m    app=hello-world
web-rs-x4skb   1/1    Running   0          39m    app=hello-world
```

If you look closely, the Pod names are the same as before. This is because the Pods remained running while we deleted and re-created the ReplicaSet.

Scaling with ReplicaSets

With the ReplicaSet already in place, it's really simple to scale the Pods it's managing.

Following the declarative model, the best way to scale a ReplicaSet is to edit its manifest file and re-POST it to the cluster. This makes sure that the manifest file is always up-to-date and in-sync with what is deployed.

> **Note:** It's a best practice to treat your manifest files like application code and check them into version control repositories.

Edit the `.spec.replicas` section of the manifest file to change the desired number of replicas to 5.

This shows the updated section of the manifest file.

```
spec:
  replicas: 5
```

Use the `kubectl apply` command to update the config of the ReplicaSet.

```
$ kubectl apply -f rs.yaml
Warning: kubectl apply should be used...
replicaset "web-rs" configured
```

Verify the operation.

```
$ kubectl get rs/web-rs
NAME      DESIRED   CURRENT   READY     AGE
web-rs    5         5         5         14m
$
$
NAME           READY   STATUS        RESTARTS   AGE    LABELS
web-rs-2rr2z   1/1     Running       0          50m    app=hello-world
web-rs-42zq7   0/1     Terminating   0          50m    app=hello-world
web-rs-64sxx   1/1     Running       0          50m    app=hello-world
web-rs-hwbtd   1/1     Running       0          50m    app=hello-world
web-rs-q6h4v   0/1     Terminating   0          50m    app=hello-world
web-rs-t8rqw   1/1     Running       0          50m    app=hello-world
web-rs-v4jc2   1/1     Running       0          50m    app=hello-world
web-rs-wntz9   0/1     Terminating   0          50m    app=hello-world
web-rs-wskgq   0/1     Terminating   0          50m    app=hello-world
web-rs-x4skb   0/1     Terminating   0          50m    app=hello-world
```

See how some of the Pods are in the `Terminating` phase. This is while they are gracefully terminated to bring the desired number down from 10 to 5.

Great. There are now only 5 Pods in the `Running` state.

Scaling the ReplicaSet up follows the same workflow - update the `.spec.replicas` field of the manifest file and use the `kubectl apply` command to re-POST it to the API server.

The output below shows the three new Pods being started following an update to the manifest file changing the desired number of replicas from 5 to 8.

```
NAME           READY   STATUS              AGE    LABELS
web-rs-2rr2z   1/1     Running             53m    app=hello-world
web-rs-64sxx   1/1     Running             53m    app=hello-world
web-rs-hwbtd   1/1     Running             53m    app=hello-world
web-rs-t8rqw   1/1     Running             53m    app=hello-world
web-rs-v4jc2   1/1     Running             53m    app=hello-world
web-rs-htccm   0/1     ContainerCreating   3s     app=hello-world
web-rs-sp7wm   0/1     ContainerCreating   3s     app=hello-world
web-rs-wn7mb   0/1     ContainerCreating   3s     app=hello-world
```

Only in emergencies!

It is possible to deploy and modify Kubernetes applications the imperative way (issuing commands to manipulate state rather than defining desired state in a manifest file). However, **you should only use the imperative form in cases of emergency!**

This is because of the risk that imperative changes made the cluster never find their way into the declarative manifest files. Subsequent POSTings of the out-dated manifest files to the API server will then overwrite the previous imperative changes. This can cause serious problems.

Imagine the following example. You have an application in production and the application is down! The problem lies with the version of the image in use, and you are forced to make an emergency change to the image to implement an updated image. It's an emergency, so you update the cluster via an imperative method. The change takes effect and the problem is resolved, but you forget to reflect the change in the application's manifest file stored in version control!! At this point, the manifest file still references the broken image. A month later, you scale the application by checking the manifest file out of version control, updating the `.spec.replicas field`, and re-applying it to the cluster. The change updates the desired number of replicas, but it also changes the image back to the old broken image!

Moral of the story… Declarative FTW! Imperative methods should only be used in absolute emergencies!!

Chapter summary

ReplicaSets bring the concepts of desired state to our applications. We specify a Pod template, a desired number of Pod replicas, and let Kubernetes perform the magic!

We define ReplicaSets with the usual YAML or JSON manifest file and feed it to the API server. This gets handed over to the ReplicaSet controller which makes sure the right number of the right Pod get instantiated. Fundamental to this is the all-powerful reconciliation loop that is constantly watching the cluster and making sure that current state and desired state match.

Even if we only need a single instance of a Pod, we should probably deploy it via a higher-level object like a ReplicaSet or Deployment. This will give the Pod self-healing capabilities and the ability to scale if we decide we need more in the future!

It’s common to deploy ReplicaSets indirectly via higher-level *Deployment* objects.

6: Kubernetes Services

In the previous chapters we've launched Pods and added self-healing and scalability via ReplicaSets. But we've explained that we can't rely on *Pod* IPs. In this chapter, we'll see how Kubernetes *Services* give us networking that we **can** rely on.

We'll split the chapter up like this:

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

We'll look at two typical access scenarios:

1. Accessing Pods from **inside** the cluster
2. Accessing Pods from **outside** the cluster

Setting the scene

Before diving in, we need to remind ourselves that **Pod IPs are unreliable.** When Pods fail, they are replaced with new Pods with new IPs. Scaling-up a ReplicaSet introduces new Pods with new, previously unknown, IP addresses. Scaling-down a ReplicaSet removes Pods. This removes network endpoints form the ReplicaSet/app. All of this means that Pod IPs unreliable.

We also need to know 3 things about Kubernetes Services.

First, we need to clear up some terminology. When talking about a *Service* in this chapter we're talking about the Service REST object in the Kubernetes API. Just like a *Pod*, *ReplicaSet*, or *Deployment*, a Kubernetes ***Service*** is an object in the API that we define in a manifest and POST to the API server.

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

Third, we need to know that a Service uses labels to dynamically associate with a set of Pods.

The last two points are what allow a Service to provide stable networking to a dynamic set of Pods.

Theory

Figure 6.1 shows a simple pod-based application deployed via a ReplicaSet. It shows a client (which could be another component of the app) that does not have a reliable network endpoint for accessing the Pods - remember that Pod IPs are unreliable.

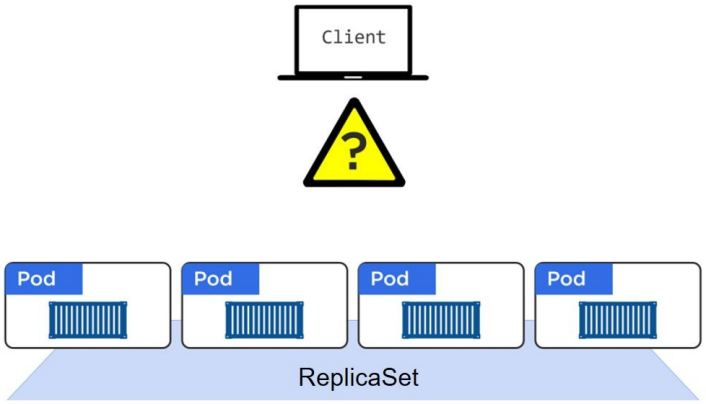


Figure 6.1

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

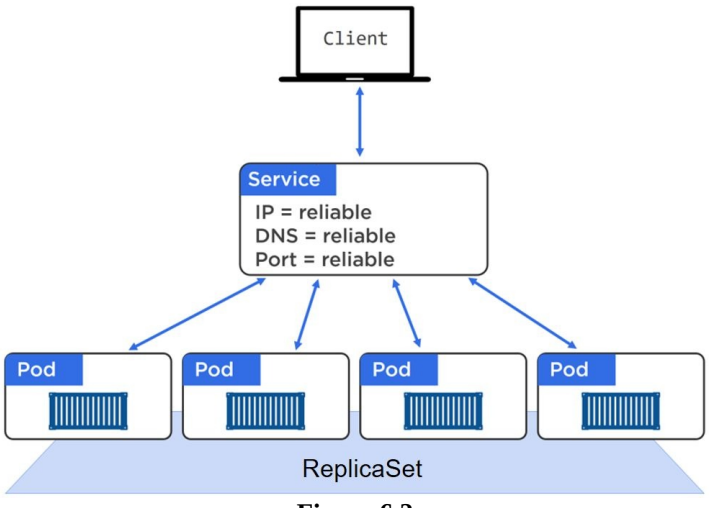


Figure 6.2

This means the Pods behind the Service can come-and-go as much as needed. They can scale up and down, they can fail, and they can be updated… and the Service in front of them notices the changes and updates its knowledge of the Pods. But it never changes the stable IP, DNS and port that it exposes!

Labels and loose coupling

Pods and Services are loosely coupled via labels and label selectors. This is the same technology and principle that links Pods with ReplicaSets. Figure 6.3 shows an example where 3 Pods are labelled as `zone=prod` and `version=1`, and the Service has a label selector that matches.

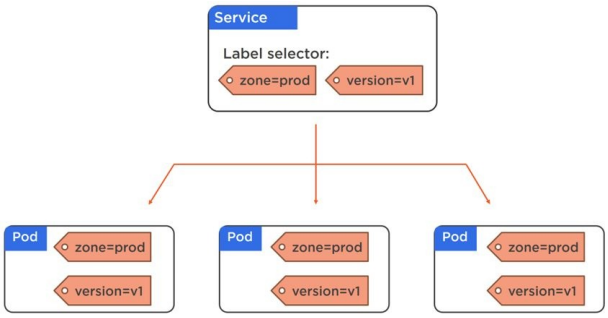


Figure 6.3

The Service shown in Figure 6.3 provides stable networking to all three Pods. It also provides simple load-balancing.

For a Service to match a set of Pods, and therefore provide stable networking and load-balance, it only needs to match some of the Pods labels. However, for a Pod to match a Service, the Pod must match all of the values in the Service's label selector. If that sounds confusing, the examples in Figure's 5.4 and 5.5 should help.

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

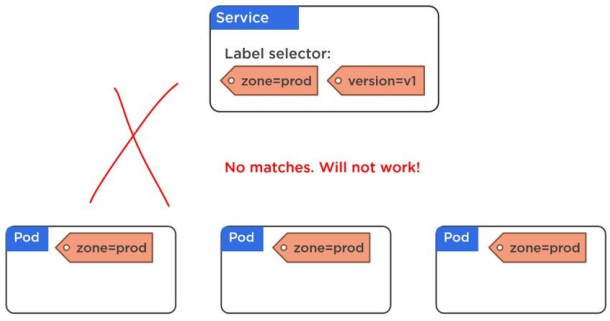


Figure 6.4

Figure 6.5 shows an example that does work. It works because the Service is selecting on two labels and the Pods have both. It doesn't matter that the Pods have additional labels that the Service is not selecting on. The Service selector is looking for Pods with two labels, it finds them, and ignores the fact that the

Pods have additional labels - all that is important is that the Pods have the labels the Service is looking for.

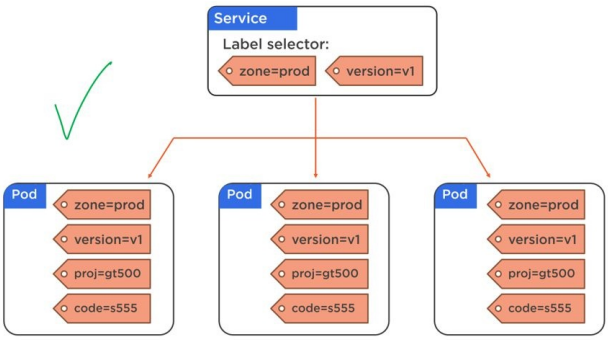


Figure 6.5

The following excerpts from a Service YAML, and ReplicaSet YAML, show how *selectors* and *labels* are implemented. We've added comments to the lines we're interested in.

svc.yml

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

rs.yml

```
apiVersion: apps/v1beta2
kind: ReplicaSet
metadata:
  name: hello-rs
spec:
  replicas: 10
  selector:
    matchLabels:
      app: hello-world
  template:
    metadata:
      labels:
        app: hello-world    # Pod labels
    spec:
```

```
containers:
- name: hello-ctr
  image: nigelpoulton/k8sbook:latest
  ports:
  - containerPort: 8080
```

In the example files, the Service has a label selector (`.metadata.labels`) with a single value `app=hello-world`. This is the label that the Service is looking for when it queries the cluster for Pods. The ReplicaSet specifies a Pod template with the same `app=hello-world` label (`.spec.template.metadata.labels`), meaning that any Pods it deploys will have the `app=hello-world` label. It is these two values that tie the Service to Pods deployed by the ReplicaSet.

When the ReplicaSet and the Service are deployed, the Service will select all 10 Pod replicas and provide them with a stable networking endpoint and perform load-balancing.

Services and Endpoint objects

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

Each Service that is created automatically gets an associated *Endpoint* object. This Endpoint object is a dynamic list of all of the Pods that match the Service's label selector.

Kubernetes is constantly evaluating the Service's label selector against the current list of Pods in the cluster. Any new Pods that match the selector get added to the Endpoint object, and any Pods that disappear get removed. This ensures the Service is kept up-to-date as Pods come and go.

The Endpoint object has its own API endpoint that Kubernetes-native apps can query for the latest list of matching Pods. Non-native Kubernetes apps can use the Service's Virtual IP (VIP), which is also kept up-to-date.

Accessing from inside the cluster

When we create a Service object, Kubernetes automatically creates an internal DNS mapping for it. This maps the **name** of the Service to a VIP. For example, if you create a Service called "hellcat-svc", Pods in the cluster will be able to resolve "hellcat-svc" to the Service's VIP. Therefore, as long as you know the name of a Service, you will be able to connect to it by name.

Accessing from outside the cluster

We can also use Services to access Pods from outside of the cluster.

We already know that every Service gets a DNS name, Virtual IP, and port. A Service's port is *cluster-wide*, meaning it maps back to the Service form every node in the cluster! We can use this port to connect to the Service from outside of the cluster. Let's look at an example.

At the lowest level, we have *nodes* in the cluster hosting Pods. Then we create a Service and use labels to associate it with Pods. The Service object has a reliable port mapped to every node in the cluster – the port that the Service uses is the same on every node. This means that traffic from outside of the cluster can hit any node on that port and get through to the application (Pods).

Figure 6.6 shows an example where 3 Pods are exposed by a Service on port `30050` on every node in the cluster. In step 1 an external client hits **Node2** on port `30050`. In step 2 it is redirected to the Service object (this happens even though **Node2** isn't running a Pod from the Service). Step 3 shows that the Service has an associated Endpoint object with an always-up-to-date list of Pods matching the label selector. Step 4 shows the client being directed to **Pod1**.

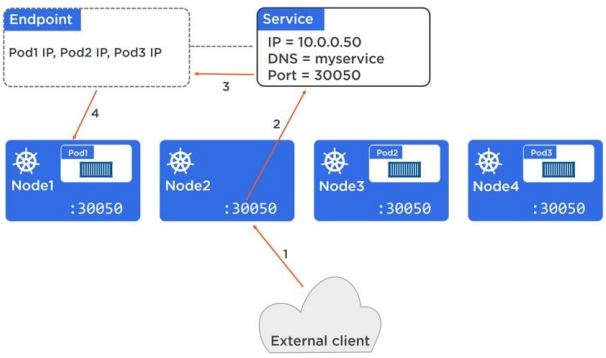


Figure 6.6

The Service could just as easily have directed the client to Pod 2 or Pod 3. In fact, future requests will go to other Pods as the Service load-balances traffic between them.

There are other options for accessing services from outside of a cluster, such as integrating with load-balancers from your cloud provider. But that starts getting platform-specific (implementation differences between Google and AWS etc.) and we're not going there.

Service discovery

Kubernetes implements Service discovery in a couple of ways:

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

DNS-based Service discovery requires the DNS *cluster-add-on*. If you followed the installation methods from the "Installing Kubernetes" chapter, you'll already have this. The DNS add-on implements a Pod-based DNS service in the cluster and configures all kubelets (nodes) to use it for DNS.

It constantly watches the API server for new Services and automatically registers them in DNS. This results in every Services getting a DNS name that is resolvable across the entire cluster.

The other way is through environment variables. Every Pod also gets a set of environment variables that assist with name resolution. This is a fall-back in case you're not using DNS in your cluster.

The problem with environment variables is that they are inserted into Pods at creation time. This means that Pods have no way of learning about new objects added to the cluster after the Pod itself is created. This is a major reason why DNS is the preferred method.

Summary of Service theory

Services are all about providing a stable networking endpoint for Pods. They also provide load-balancing and a cluster-wide port that the Service can be accessed on from outside of the cluster. We associate them with Pods using labels and label selectors.

Hands-on

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

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

- The imperative way (only use in emergencies)
- The declarative way

The imperative way

Warning! The imperative way is **not** the Kubernetes way! It introduces the risk that changes made imperatively never make it into declarative manifests,

rendering the manifests stale. This introduces the risk that these stale manifests are used to update the cluster at a later date, unintentionally overwriting the changes that were made imperatively.

Use the `kubectl` command to declaratively deploy the following ReplicaSet. The command assumes the ReplicaSet is defined in a file called `rs.yaml` and has the following content.

> You do not have to complete this step if you still have the ReplicaSet running from previous examples.

```
apiVersion: apps/v1beta2
kind: ReplicaSet
metadata:
  name: web-rs
spec:
  replicas: 10
  selector:
    matchLabels:
      app: hello-world
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-ctr
        image: nigelpoulton/k8sbook:latest
        ports:
        - containerPort: 8080
```

```
$ kubectl create -f rs.yaml
replicaset "web-rs" created
```

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

Use the following `kubectl` command to create a new Service that will provide networking and load-balancing for the Pods deployed in the previous step.

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

service "hello-svc" exposed
```

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

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

```
$ kubectl describe svc hello-svc
Name:                   hello-svc
Namespace:              default
Labels:                 app=hello-world
Annotations:            <none>
Selector:               app=hello-world
Type:                   NodePort
IP:                     100.70.80.47
Port:                   <unset> 8080/TCP
NodePort:               <unset> 30175/TCP
Endpoints:              100.96.1.10:8080, 100.96.1.11:8080, + more...
Session Affinity:       None
Events:                 <none>
```

Some interesting values in the output above include:

- `Selector` is the list of labels that Pods must have if they want to match the Service
- `IP` is the permanent virtual IP (VIP) of the Service on the Pod network
- `Port` is the port that the app listens on
- `NodePort` is the cluster-wide port
- `Endpoints` is the dynamic list of Pods that currently match the Service's label selector.

Now that we know the cluster-wide port that the Service is accessible on, we can open a web browser and access the app. In order to do this, you will need to know the IP address of at least one of the nodes in your cluster (it will need to be an IP address that you reach from your browser - e.g. a publicly routable IP if you will be accessing via the internet). Figure 6.7 shows a web browser accessing a cluster node with an IP address of `54.246.255.52` on the cluster-wide port `30175`.

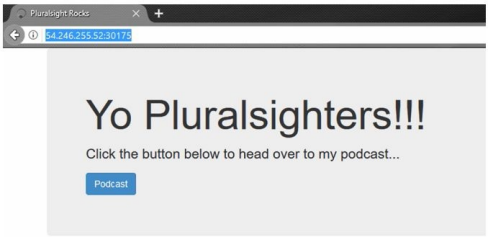


Figure 6.7

The app we've deployed is a simple web app. It's built to listen on port 8080, and we've configured a Kubernetes *Service* to map port `30175` on every cluster node back to port 8080 on the app. By default, cluster-wide ports (NodePort values) are between 30,000 - 32,767.

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

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

The declarative way

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

A Service manifest file

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

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

Let's step through some of the lines.

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

The `.kind` field tells Kubernetes to pass the manifest to the Service controller for deployment as a Service.

The `.metadata` section defines a name and a label for the Service. The label here is a label for the Service itself, and not the label that the Service uses as its selector.

The `.spec` section is where we actually define the Service. In this example, we're telling Kubernetes to deploy a `NodePort` Service* (other types such as `ClusterIP` and `LoadBalancer` exist) and to map port `8080` from the app to port `30001` on each node in the cluster. Then we're explicitly telling it to use TCP (default).

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

Common Service types

The three common *ServiceTypes* include:

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

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

```
$ kubectl create -f svc.yml
service "hello-svc" created
```

This command tells Kubernetes to deploy a new object from a file called `svc.yml`. The `-f` flag lets you tell Kubernetes which manifest file to use. Kubernetes knows to deploy a Service object based on the value of the `.kind` field in the manifest.

Introspecting Services

You can inspect the Service with the usual `kubectl get` and `kubectl describe` commands.

```
$ kubectl get svc hello-svc
NAME        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
hello-svc   100.70.40.2     <nodes>       8080:30001/TCP   8s
$
$
$ kubectl describe svc hello-svc
Name:                hello-svc
Namespace:           default
Labels:              app=hello-world
Annotations:         <none>
Selector:            app=hello-world
Type:                NodePort
IP:                  100.70.40.2
Port:                <unset> 8080/TCP
NodePort:            <unset> 30001/TCP
Endpoints:           100.96.1.10:8080, 100.96.1.11:8080, + more...
Session Affinity:    None
Events:              <none>
```

In the example above, we've exposed the Service as a `NodePort` on port `30001` across the entire cluster. This means we can point a web browser to that port on any node and get to the Service. You will need to use a node IP that you can reach, and you will need to make sure that any firewall and security rules allow the traffic.

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

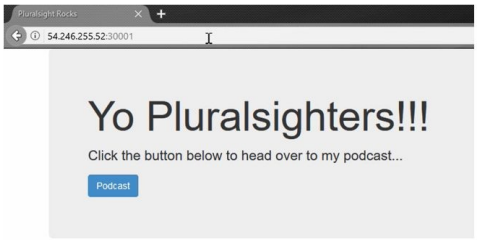


Figure 6.8

Endpoint objects

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

In the command below, we abbreviate the Endpoint to `ep`.

```
$ kubectl get ep hello-svc
NAME        ENDPOINTS                                AGE
hello-svc   100.96.1.10:8080, 100.96.1.11:8080       1m
kubernetes  172.20.32.78:443
```

```
$
$
$ Kubectl describe ep hello-svc
Name:         hello-svc
Namespace:    default
Labels:       app=hello-world
Annotations:  <none>
Subsets:
  Addresses:  100.96.1.10,100.96.1.11,100.96.1.12...
  NotReadyAddresses:   <none>
  Ports:
    Name      Port     Protocol
    ----      ----     --------
    <unset>   8080     TCP
Events: <none>
```

Summary of deploying Services

As with all Kubernetes objects, the preferred way of deploying and managing Services is the declarative way. Because they leverage labels they are dynamic. This means you can deploy new Services that will work with Pods and ReplicaSets that are already deployed and in use. Each Service gets an Endpoint object that maintains an up-to-date list of matching Pods.

Real world example

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

Let's take a minute to run through a really common real-world example - making updates to business apps.

We all know that updating business applications is a fact of life - bug fixes, new features etc.

Figure 6.9 shows a simple business app deployed on a Kubernetes cluster as a bunch of Pods managed by a ReplicaSet. As part of it, we've got a Service selecting on Pods with labels that match `app=biz1` and `zone=prod` (notice how the Pods have both of the labels the Service selector requires). The application is up and running.

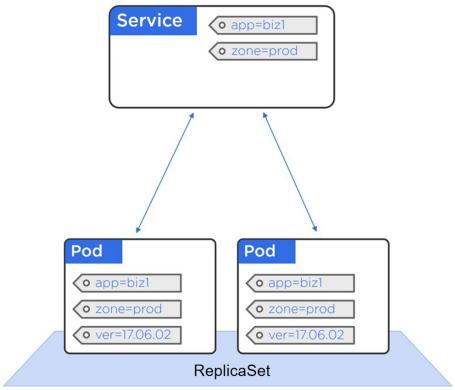


Figure 6.9

Let's assume we need to push a new version of the app. But we need to do it without incurring downtime.

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

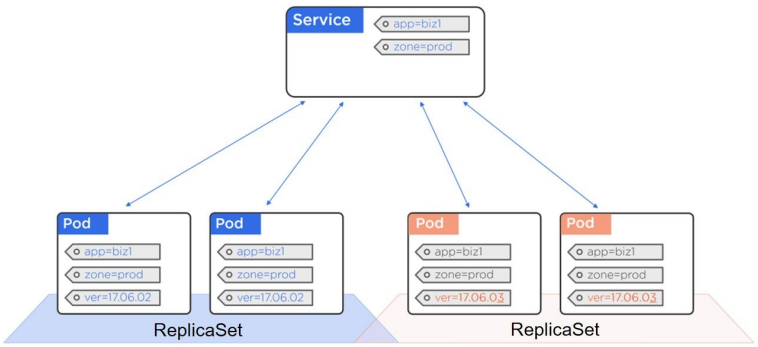


Figure 6.10

The updated *Pods* are deployed through their own ReplicaSet and are labelled so that they match the existing Service's label selector. The Service is now load-balancing requests across **both versions of the app** (`version=17.06.02` and `version=17.06.03`).

This happens because the Service's label selector is being constantly evaluated, and its Endpoint object and VIP load-balancing are constantly being updated with new matching Pods.

Forcing all traffic to the updated version is as simple as updating the Service's label selector to include the label `version=17.06.03`. Suddenly the older version no longer matches, and everyone's getting served the new version (Figure 6.11).

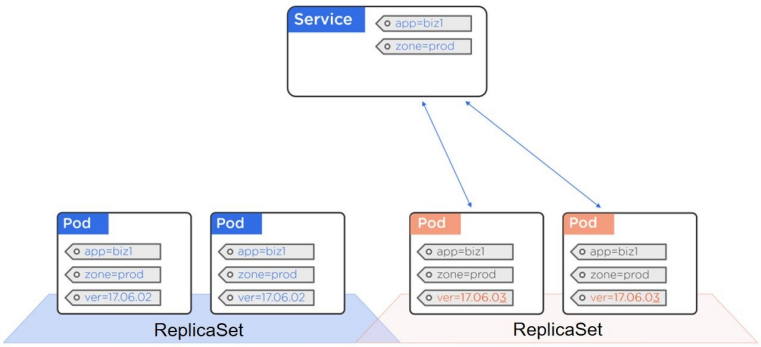


Figure 6.11

However, the old version still exists - we're just not using it any more. This means if we experience an issue with the new version, we can switch back to the previous version by simply changing the label selector to include `version=17.06.02` instead of `version=17.06.03`. See Figure 6.12.

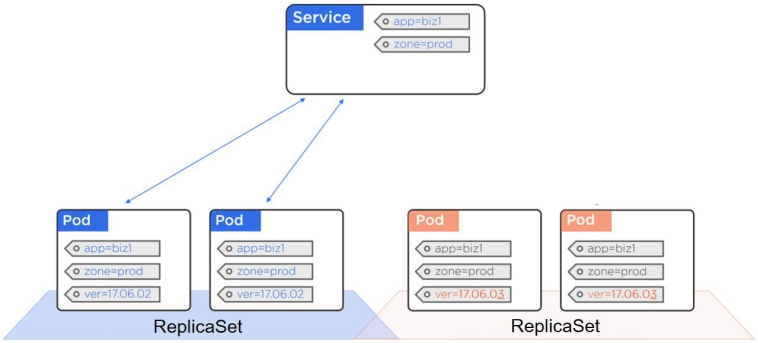


Figure 6.12

Now everybody's getting the old version.

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

Chapter Summary

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

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

7: Kubernetes Deployments

In this chapter, we'll see how Kubernetes *Deployments* build on everything we've learned so far, and add mature rolling-updates and rollbacks.

We'll split the chapter up as follows:

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

Deployment theory

The first thing to know about Deployments is that they are all about **rolling updates** and **seamless rollbacks**!

At a high level, we start with Pods as the basic Kubernetes object. We wrap them in a ReplicaSet for scalability, resiliency, and desired state. Then we add a Deployment for rolling updates and simple rollbacks.

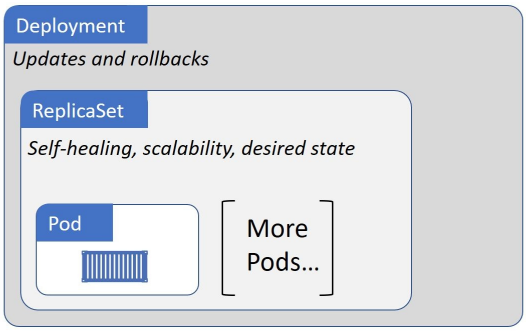


Figure 7.1

The next thing to know, is that they are fully-fledged objects in the Kubernetes API. This means we define them in a manifest file that we feed to the deployment controller via the API server.

To recap; *Deployments* manage *ReplicaSets*, and *ReplicaSets* manage *Pods*. Throw them into a pot, and we've got a pretty awesome way to deploy and manage Kubernetes apps!

Rolling updates the old way

Before we had Deployments we'd deploy our apps via *ReplicationControllers* (predecessor to ReplicaSets). To update the app, we'd create a new ReplicationController with different name and v different version label. Then we'd do a `kubectl rolling-update` and tell it to use the manifest file for the new ReplicationController. Kubernetes would then take care of the update.

This process worked. But it was a bit Frankenstein - a bit of a bolt-on. For one thing, it was all handled on the client instead of at the control plane. Also, rollbacks were a bit basic and there wasn't proper audit trailing.

Rolling updates with Deployments

Deployments are better!

With a Deployment, we create a manifest file and POST it to the API server. That gets given to the deployment controller and the app gets deployed on the cluster.

Behind the scenes, we get a ReplicaSet and a bunch of Pods – the Deployment takes care of creating all of that for us. That means we get all of the ReplicaSet goodness such as background loops that make sure current state matches desired state etc.

> **Note:** You should not directly manage ReplicaSets that are created as part of a Deployment. You should perform all actions against the Deployment object and leave the Deployment to manage its own ReplicaSets.

When we need to push an update, we commit the changes to the **same Deployment manifest file** and rePOST it to the API server. In the background, Kubernetes creates a new ReplicaSet (now we have two) and winds the old one down at the same time that it winds the new one up. Net result: we get a smooth rolling update with zero downtime. And we can rinse and repeat the process for future updates - just keep updating that manifest file (which we should be storing in a version control system). Brilliant!

Figure 7.2 shows a Deployment with two revisions. The first is the initial deploy to the cluster, and the second is an update. We can see that the ReplicaSet associated with revision 1 has been wound down and no longer has any Pods. The ReplicaSet associated with revision 2 is now active and owns all of the Pods.

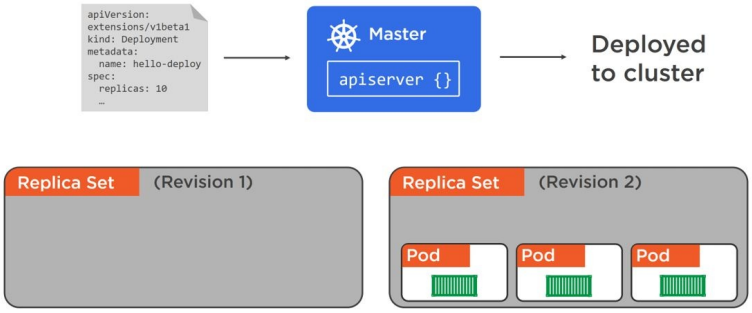


Figure 7.2

Rollbacks

As we can see in Figure 7.2, the old ReplicaSets stick around and don't get deleted. They've been wound down, so aren't managing any Pods, but they still exist. This makes them a great option for reverting back to previous versions.

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

Figure 7.3 shows the same app rolled back to revision 1.

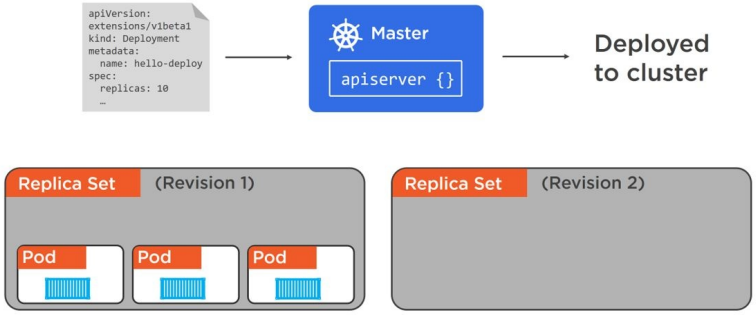


Figure 7.3

That's not the end though! There's built-in intelligence that allows us to say things like "wait X number of seconds after each Pod comes up before us mark it as healthy…". There's also readiness probes and all kinds of things. All-in-all, Deployments are lot better, and more feature rich, than the way we used to do things with ReplicationControllers.

With all that in mind, let's get our hands dirty and create our first Deployment.

How to create a Deployment

In this section, we're going to create a brand-new Kubernetes Deployment from a YAML file. We can do the same thing imperatively using the `kubectl run` command, but we shouldn't! The right way is the declarative way!

If you've been following along with the examples in the book, you'll have a ReplicaSet and a Service still running from the previous chapters. To follow with these examples, you should delete the ReplicaSet but keep the Service running.

```
$ kubectl delete rs web-rs
replicaset "web-rs" deleted
```

The reason we're leaving the Service running is to demonstrate the loose coupling between the Service object and the Pods it serves. We're going to deploy the same app again using the same port and labels. This means the Service we deployed earlier will select on the new Pods that we're about to create via the Deployment.

This is the Deployment manifest file we're going to use. The following examples call it `deploy.yml`.

> If you're using a version of Kubernetes prior to 1.8.0 you should use the `apps/v1beta1` API. If you are using 1.6.0 or earlier you should use `extensions/v1beta1`.

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: hello-deploy
spec:
  replicas: 10
  selector:
    matchLabels:
      app: hello-world
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-pod
        image: nigelpoulton/k8sbook:latest
        ports:
        - containerPort: 8080
```

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

Right at the very top, we're specifying the API version to use. Deployment objects are relatively new at the time of writing, so they're not in the `v1` API, and require us to use relatively new versions. The example assumes Kubernetes 1.8.0 or later with the `apps/v1beta2` API. For version of Kubernetes prior to 1.8 you should use `apps/v1beta1`, and for versions prior to 1.6.0 you should use `extensions/v1beta1`.

Next, we're using the `.kind` field to tell Kubernetes we're defining a Deployment - this will make Kubernetes send this to the deployment controller on the control plane.

After that, we give it a few properties that we've already seen with ReplicaSets. Things like a name and the number of Pod replicas.

In the `.spec` section we define a few Deployment-specific parameters, and in the `.spec.template` section give it a Pod spec. We'll come back and look at some of these bits later.

To deploy it we use the `kubectl create` command.

```
$ kubectl create deployment -f deploy.yml
deployment "hello-deploy" created
```

If you want, you can omit the optional `deployment` keyword because Kubernetes can infer this from the `.kind` field in the manifest file.

Inspecting a Deployment

We can use the usual `kubectl get deploy` and `kubectl describe deploy` commands to see details.

```
$ kubectl get deploy hello-deploy
NAME          DESIRED   CURRENT  UP-TO-DATE  AVAILABLE   AGE
hello-deploy  10        10       10          10          1m
```

```
$ kubectl describe deploy hello-deploy
Name:         hello-deploy
Namespace:    default
Labels:       app=hello-world
Selector:     app=hello-world
Replicas:               10 desired | 10 updated | 10 total ...
StrategyType:           RollingUpdate
MinReadySeconds:        10
RollingUpdateStrategy:  1 max unavailable, 1 max surge
Pod Template:
  Labels:               app=hello-world
  Containers:
```

```
        hello-pod:
          Image:        nigelpoulton/k8sbook:latest
          Port:         8080/TCP
<SNIP>
```

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

As we mentioned earlier, Deployments automatically create new ReplicaSets. This means we can use the regular `kubectl` commands to view these.

```
$ kubectl get rs
NAME                  DESIRED   CURRENT   READY   AGE
hello-deploy-7bbd...  10        10        10      5m
```

Right now, we only have one ReplicaSet. This is because we've only done the initial rollout of the Deployment. However, we can see it gets the same name as the Deployment appended with a hash of the Pod template from the manifest file.

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

Accessing the app

In order to access the application from a stable IP or DNS address, or even from outside the cluster, we need the usual Service object. If you're following along with the examples you will have the one from the previous chapter still running. If you don't, you will need to deploy a new one.

Only perform the following step if you do not have the Service from the previous chapter still running.

```
$ kubectl create svc svc.yml
service "hello-svc" created
```

The manifest file for the Service is shown below:

```
  apiVersion: v1
  kind: Service
  metadata:
    name: hello-svc
    labels:
      app: hello-world
  spec:
    type: NodePort
    ports:
```

```
    - port: 8080
      nodePort: 30001
      protocol: TCP
    selector:
      app: hello-world
```

Accessing the app is the same as before - a combination of the DNS name or IP address of one of the nodes in the cluster, coupled with the `NodePort` value of the Service. For example, `54.246.255.52:30001` as shown in Figure 7.4.

If you are using Minikube, you should use the IP address of the Minikube, which you can get using the `minikube ip` command.

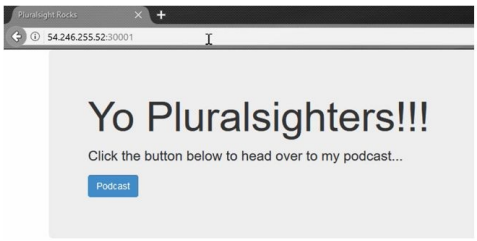


Figure 7.4

How to perform a rolling update

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

The first thing we need to do is update the image tag used in the Deployment's manifest file. The initial version of the app that we deployed used an image tagged as `nigelpoulton/k8sbook:latest`. We'll update the `.spec.template.spec.containers.` section of the Deployment's manifest file to reference a different image called `nigelpoulton/k8sbook:edge`. This will ensure that next time the Deployment manifest is POSTED to the API server, the entire Deployment will be updated to run the new `edge` image.

The following is an updated copy of the `deploy.yml` manifest file (the only change is to `.spec.template.spec.containers.image` - the third line from bottom).

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: hello-deploy
```

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

> **Warning:** The images used in this book are not maintained and will be full of vulnerabilities and other security issues.

Let's take a look at some of the update related settings in the manifest before starting the update.

The `.spec` section of the manifest contains all of the configuration specific to the Deployment. The first value of interest is `.spec.minReadySeconds`. We've set this to 10, telling Kubernetes to wait for 10 seconds after updating each Pod before moving on and updating the next. This is useful for throttling the rate at which an update occurs - longer waits give us a chance to spot problems before all of the Pods have been updated.

We also have a nested `.spec.strategy` map that tells Kubernetes we want this Deployment to:

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

As the desired state of the app demands 10 replicas, `maxSurge: 1` means we will never have more than 11 Pods in the app during the update process.

With the updated manifest ready, we can initiate the update with the `kubectl apply` command.

```
$ kubectl apply -f deploy.yml --record
Warning: kubectl apply should be used...
```

```
deployment "hello-deploy" configured
```

The update will take some time to complete. It will iterate one Pod at a time, wait 10 seconds after each, and it has to pull down the new image on each node.

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

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

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

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

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

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

The output above shows the update as complete - 10 Pods are up to date. You can get more detailed information about the state of the Deployment with the `kubectl describe deploy` command - this will include the new version of the image in the `Pod Template` section of the output.

If you've been following along with the examples you'll be able to hit `refresh` in your browser and see the updated contents of the web page (Figure 7.5).

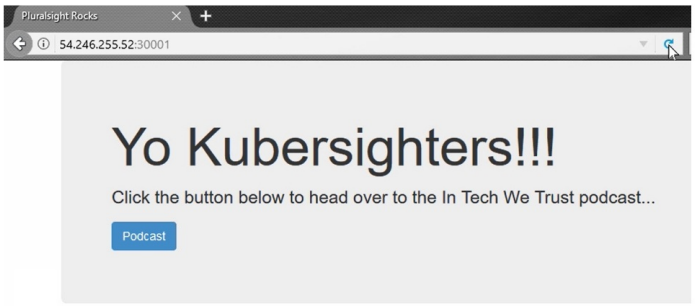


Figure 7.5

How to perform a rollback

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

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

Revision `1` was the initial deployment that used the `latest` image tag. Revision `2` is the rolling update that we just performed, and we can see that the command we used to invoke the update has been recorded in the object's history. This is only there because we used the `--record` flag as part of the command to invoke the update. For this reason, it is highly recommended to use the `--record` flag.

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

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

The output above shows that the ReplicaSet for the initial revision still exists (`hello-deploy-7bbd...`) but that it has been wound down and has no associated replicas. The `hello-deploy-6bc8...` ReplicaSet is the one form the latest revision, and is active with 10 replicas under management. However, the fact that the previous version still exists, makes rollbacks extremely simple.

The following example uses the `kubectl rollout` command to roll our application back to revision 1.

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

Although it might look like the rollback operation is instantaneous it is not. It follows the same rules set out in the Deployment manifest - `minReadySeconds: 10`, `maxUnavailable: 1`, and `maxSurge: 1`. You can verify this and track the progress with the `kubectl get deploy` and `kubectl rollout` commands shown below.

```
$ kubectl get deploy hello-deploy
NAME          DESIRED  CURRNET  UP-TO-DATE  AVAILABE  AGE
hello-deploy  10       11       4           9         45m
$
$ kubectl rollout status deployment hello-deploy
Waiting for rollout to finish: 6 out of 10 new replicas have been updated..
Waiting for rollout to finish: 7 out of 10 new replicas have been updated..
Waiting for rollout to finish: 8 out of 10 new replicas have been updated..
Waiting for rollout to finish: 1 old replicas are pending termination...
Waiting for rollout to finish: 9 of 10 updated replicas are available...
^C
```

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

Chapter summary

In this chapter, we learned that *Deployments* are the latest and greatest way to manage Kubernetes apps. They build on top of Pods and ReplicaSets by adding mature and configurable updates and rollbacks.

Like everything else, they're objects in the Kubernetes API and we should be looking to work with them declaratively.

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

8: What next

Hopefully you're up-to-speed and comfortable working with Kubernetes!

Taking your journey to the next step couldn't be easier! It's insanely easy to spin-up infrastructure and workloads in the cloud where you can build and test Kubernetes until you're a world authority! I'm a huge fan of Play with Kubernetes (PWK) for testing and playing around!

If you feel trying another style of learning, you can also head over to my video training courses at Pluralsight. If you're not a member of Pluralsight then become one! Yes, it costs money. But it's excellent value for money! And if you're unsure… there's always have a free trial where you can get access to my courses for free for a limited period.

I'd also recommend you hit events like KubeCon and your local Kubernetes meetups.

Feedback

Massive thanks for reading my book. I genuinely hope you loved it. And if you're an *old dog*, I hope you learned some new tricks!

Time for me to ask for a favour (that's how we spell it in the UK)… It takes an insane amount of effort to write a book. So, do me a huge favour and give the book a review on Amazon. It'll take you two minutes, and I'd really appreciate it!*

Also…. feel free to hit me on Twitter.

That's it. Thanks again for reading my book, and good luck driving your career forward!!

Table of Contents