The Kubernetes Book

Nigel Poulton

The Kubernetes Book

Nigel Poulton

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

This version was published on 2017-07-12

Table of Contents

0: About the book

This is a book about how to get up-to-speed with Kubernetes as fast as possible. No prior knowledge required!

Along the way we'll cover some of the basics about containers. So… you'll get the most out of this book if you already know a thing or two about containers. You don't need to be a container expert, but if you know zero about containers you might want to check out my **Docker & containers: The Big Picture** online video training course - https://www.pluralsight.com/courses/docker-containers-big-picture. Yes you have to pay to take the course, but it's had well over 1,000 5-star reviews, so you'll be getting your money's worth. Alternatively you can check out my Docker book called "Docker for Sysadmins".

What will the book cover

In order to get you up-to-speed as fast as possible we won't be covering everything about Kubernetes. But… the things that we do cover, we'll cover well. This means you'll get all the theory and hands-on required to fully understand the things we'll learn.

We'll learn about ***Pods***, ***Replication Controllers***, ***Services***, and ***Deployments***. We'll touch on a few other things along the way, but most of our effort will be on these fundamental building blocks!

Text wrapping

I've tried really hard to get the commands and outputs to fit on a single line without wrapping! So instead of getting this…

```
$ docker service ps uber-service
ID                         NAME                 IMAGE                    NOD\
E              DESIRED STATE  CURRENT STATE               ERROR
7zi85ypj7t6kjdkevreswknys  uber-service.1       nigelpoulton/tu-demo:v2  ip-\
172-31-12-203  Running        Running about an hour ago
0v5a97xatho0dd4x5fwth87e5   \_ uber-service.1   nigelpoulton/tu-demo:v1  ip-\
172-31-12-207  Shutdown       Shutdown about an hour ago
31xx0df6je8aqmkjqn8w1q9cf  uber-service.2       nigelpoulton/tu-demo:v2  ip-\
172-31-12-203  Running        Running about an hour ago
```

… you *should* get this.

```
$ docker service ps web-fe
ID           NAME      IMAGE             NODE  DESIRED  CURRENT
817f...f6z  web-fe.1  nigelpoulton/...  mgr2  Running  Running 5 mins
a1dh...mzn  web-fe.2  nigelpoulton/...  wrk1  Running  Running 5 mins
cc0j...ar0  web-fe.3  nigelpoulton/...  wrk2  Running  Running 5 mins
```

> For best results you might want to flip your reading device into landscape mode.

In doing this I’ve had to trim some of the output from some commands. But you won’t be missing anything important!

That’s it!

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

The first thing to know is that Kubernetes came out of Google! The next thing to know is that in the summer of 2014 it was open-sourced and handed over to the Cloud Native Computing Foundation (CNCF). Since then it's gone on to become one of the most important container-related technologies in the world - probably second only to 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 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. Stories of Google 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 cam along - Google has been running its *search*, *gmail*, *GFS*, and others, 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 about managing containers at scale, and they're all related to Google.

This leads to some people thinking Kubernetes is an open source version of one 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 led to Kubernetes.*

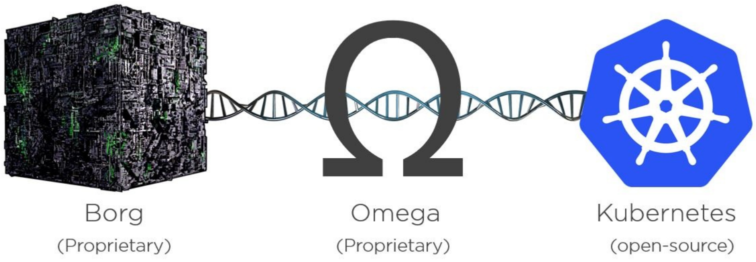


Figure 1.2

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's built on the foundations laid by Borg and Omega.

As things stand, Kubernetes is open source technology 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.

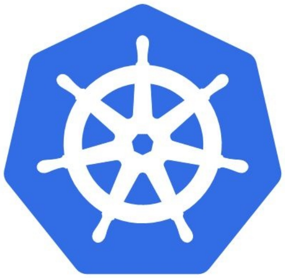


Figure 1.1

> **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. The rumor could be totally untrue, but I like the link to the *Borg* project :-D

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 typers like me ;-)

Kubernetes and the data center OS

As we said in the book's intro, I'm assuming you already have 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 of thumb, 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 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 computer** - a giant one.

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, Docker Swarm is another. 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 with a few 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 definitely here, and it's definitely 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 managing container workloads in them. And no, I don’t expect Kubernetes to take over your data center. But it will become a part of it - or there’s a good chance it will.

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

Chapter summary

Kubernetes is a leading cluster orchestrator that lets us manage containerized apps at scale, a bit like a data center scale OS. We give it an app to run and let it make all the hard decisions about where in the data center to run it and how to keep it running.

It came out of Google and is now open sourced under the Apache 2.0 license.

Disclaimer!

Kubernetes is a fast-moving project under active development. This means the things we’re learning about in this book will change. But don’t let that put you off - this is how the modern technology world works. If you snooze you lose! So don’t snooze.

After you’ve read 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 attempt to update this book approximately once per year.

2: Kubernetes principles of operation

In this chapter we’ll learn about the major things required to build a Kubernetes cluster and deploy a simple app. The aim of the game is to give your a solid big picture view before we 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 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 etc. Along comes the coach and he or she gives everyone a position and organizes them into a team. We go from Figure 2.1 to Figure 2.2.

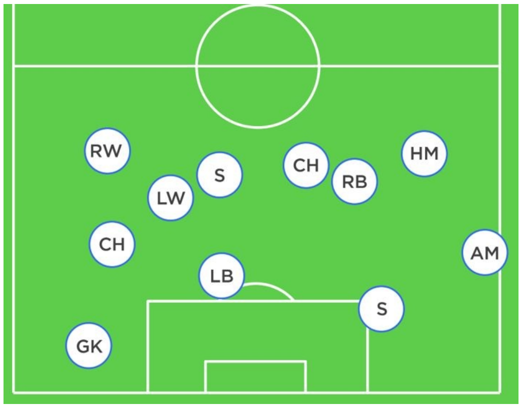


Figure 2.1

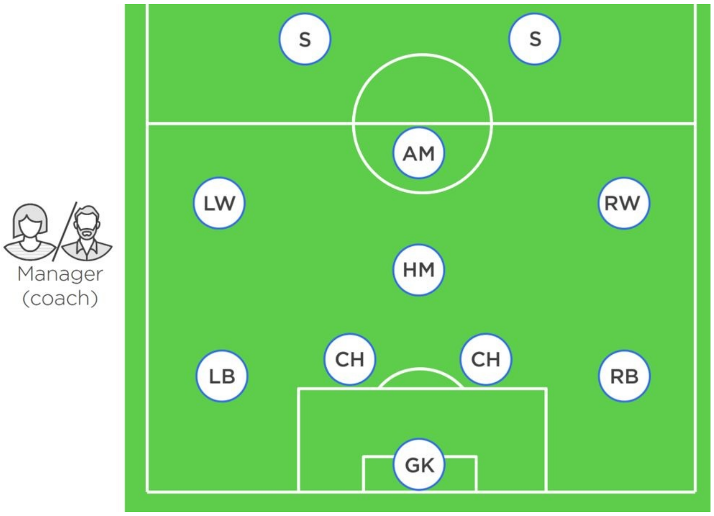


Figure 2.2

The coach also makes sure that the team maintains its shape and formation, and sticks to the team 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 exactly the same. Some might be load balancers, some might be web servers, some might be for logging etc. 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 what Kubernetes is doing “**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 an application is via something called a *Kubernetes 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 what our app should look like – what images to use, ports to expose, networks to join, how many replicas to run, how to perform updates, all that stuff. 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 runs continuous background checks to make sure that *Deployments* are running exactly as requested. If they're not, Kubernetes tries to fix the issue.

That's the big picture. Let's go 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.

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

Right now the control plane/master is a bit of a monolith. But don't be surprised if that changes in the future.

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.

Now 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 then the cluster store is its memory. The config and state of the cluster gets persistently stored here. It is the only stateful component of the cluster and is vital to its operation - no cluster store, no cluster!

At the time of writing, the cluster store is based on **etcd** - a distributed, consistent and watchable key-value store. Because 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 being to make sure the *actual 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, resource management etc.

Control Plane summary

Kubernetes *masters* run all of the cluster's control plane. Behind the scenes the 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).

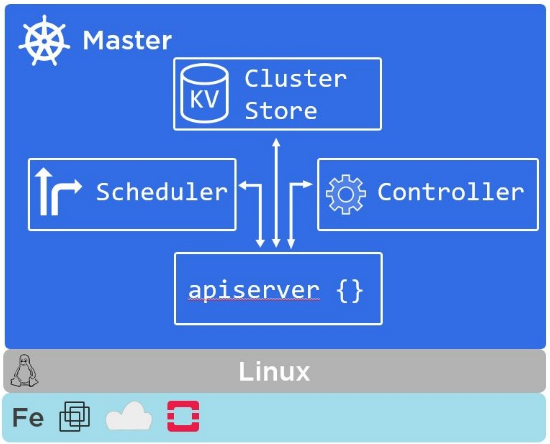


Figure 2.3

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*.

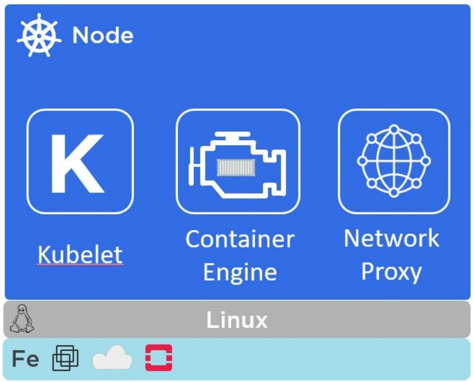


Figure 2.4

Kubelet

First and foremost is the *kubelet*. This is the main Kubernetes agent that gets installed on 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 let's 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.

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, though CoreOS rkt is also on the scene. In the case of Docker, Kubernetes talks natively to the Docker Remote API.

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 it's 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 tings 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 feed that to the Kubernetes API server
3. Kubernetes implements it on the cluster
4. 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, how to perform updates etc.

Once we've created our manifest we POST it to the Kubernetes API server. By default we do this with the `kubectl` command which sends the manifest to port 443 on the *master*.

Kubernetes inspects the manifest, identifies which controller to send it to (e.g. *deployments controller*) and records the config in the cluster store as part of the cluster's 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 *actual 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 where we issue lots of long commands telling Kubernetes how we want our app to be. Not only are manifest files simpler than long imperative commands, they also lend themselves to version control and self-documentation!

But the declarative desired state story doesn't end there. Things go wrong and things change, and when they do, the ***actual 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 in line. 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 of the *Pod* dies, the *actual state* will be reduced to 8 replicas, but the *desired state* will still be asking for 10. To rectify this Kubernetes might spin up two new replicas on other nodes in the cluster so that *actual state* returns to 10. The same will happen if we intentionally change the config by scaling the desired number of replicas up or down. We could even change the version (tag) of the image we want the web front-end to use - if we change *desired state* to request the `v2.01` tag while *actual state* is using the `v2.00` tag, Kubernetes will go through the process of updating the replicas to use the new tag specified in our updated *desired state*.

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 the cluster to look. This forms the basis of the clusters *desired state*. The Kubernetes control plane implements it and runs background reconciliation loops that are constantly checking that the *actual state* of the cluster matches the *desired state*. When they match, the world is a happy and peaceful place. When they don'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***.

Hypervisor Virtualization
VM
Atomic unit of scheduling
Container
Atomic unit of scheduling
Pod
Atomic unit of scheduling

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. Although the simplest model is to run a single container inside of a *Pod*, there are more 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 and shouldn’t be consider canonical.

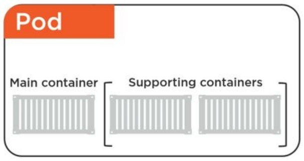


Figure 2.6

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 it, they all share the **same** *Pod* 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).

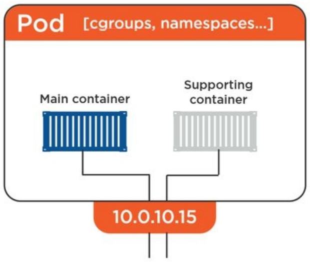


Figure 2.7

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

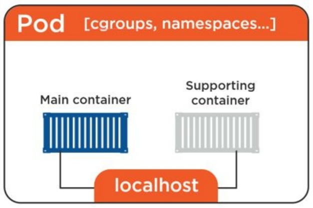


Figure 2.8

This means that multi-container *Pods* are ideal when you have requirements for tightly coupled containers – may be 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.

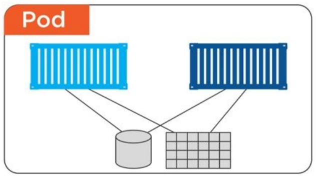


Figure 2.9 - Tightly coupled Pods

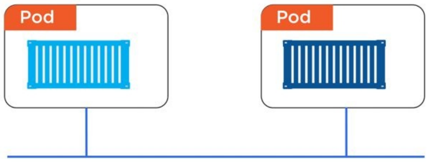


Figure 2.10 - Loosely coupled Pods

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 **don’t** 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.

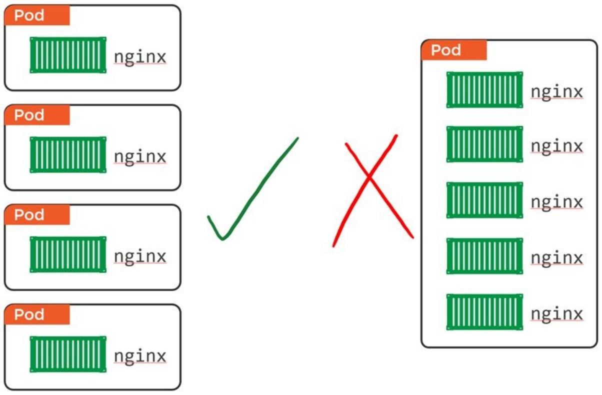


Figure 2.11 - Scaling with Pods

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. Either the entire *Pod* comes up and it gets 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. It's atomic!

A *Pod* can only exist on a single node. This means a single *Pod* cannot be spread over multiple nodes.

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.

This fit's the popular **pets vs cattle** model. 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 and nurse it back to life. Build your apps so that when their *Pods* die, 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 *Replication Controller* or *Deployment* (more on these later).

Deploying Pods via Replication Controllers

Before moving on to talk about *Services* we need to give a quick mention to *Replication Controllers (RC)*.

It’s been extremely popular to deploy *Pods* via higher level constructs called *Replication Controllers* (there is a ReplicationController object in the Kubernetes API).

As a higher level construct, a *Replication Controller* takes a *Pod* and adds features. As the names suggests, they take a *Pod* definition and deploy a desired number of *replicas* of it. They also instantiate a background loop that checks to make sure the right number of replicas are always running. – desired state vs actual state.

However, *Replication Controllers* are slowly being superseded by *Deployments*.

Services

We’ve just learned that *Pods* are mortal and can die. When this happens they get replaced with new *Pods* somewhere else in the cluster - with 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.

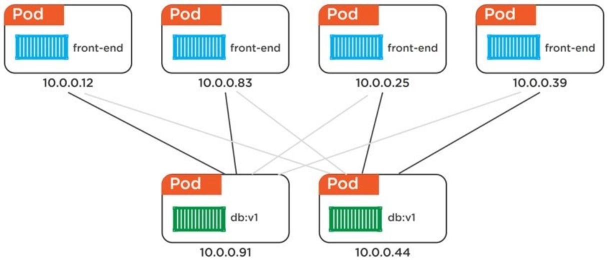


Figure 2.12

If we throw a service object in the middle, 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.

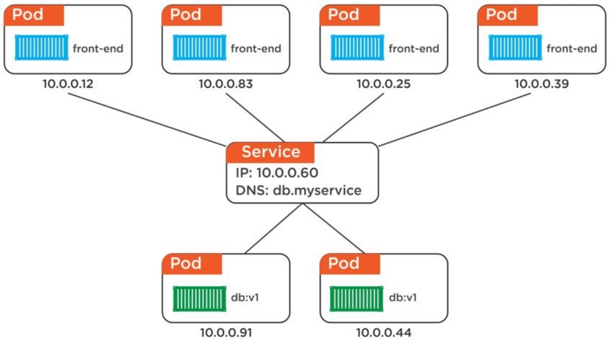


Figure 2.13

Digging in to a bit more detail, a *Service* is a fully fledged object in the Kubernetes API just like *Pods*, *Replication Controllers*, 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* – a stable networking abstraction point for multiple *Pods* that provides basic load balancing.

Connecting Pods to Services

The way that a *Pod* belongs to a *Service* is via labels.

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

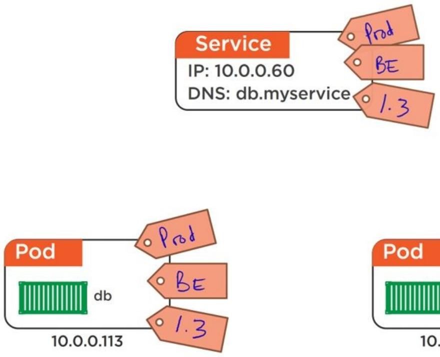


Figure 2.14

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.

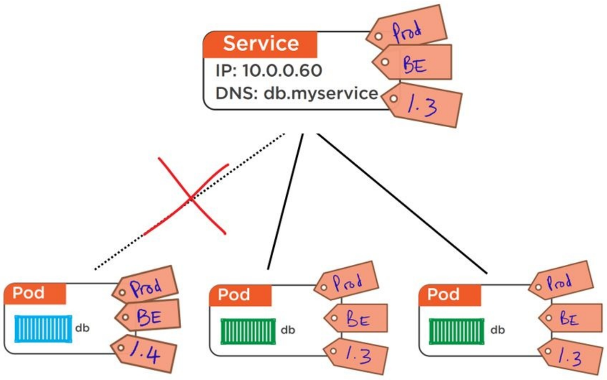


Figure 2.15

One final thing about *Services*. *Services* only send traffic to healthy *Pods*. This means if your *Pods* are failing health-checks they will not receive traffic form the *Service*.

So yeah… *Services* provide stable IP addresses and DNS names to the unstable world of *Pods*!

Deployments

Earlier in the chapter we said that we don’t normally operate on *Pods* directly. Instead, we normally operate on them via higher level objects like *Replication*

Controllers and *Deployments*. We’ll finish this chapter off by looking briefly at Kubernetes *Deployments*.

Deployments are declarative

At a very conceptual level, *Deployments* are all about being *declarative*. This is just a fancy way of saying; we give the cluster a manifest file explaining how we want things to look, and the cluster makes it happen.

This declarative model is easily the best way to manage modern applications and clusters, especially in production environments. For starters, it’s self documenting, it’s self-versioning, and it’s great for repeatable deployments – spec-once-deploy-many. This is the *gold standard* for production environments. It’s also transparent and easy get your head around, which is vital for cross-team collaboration and getting new hires up to speed etc.

But there’s more… The declarative model makes rollouts and rollbacks insanely simple!

As of Kubernetes version 1.2 *Deployments* are now first-class REST objects in the Kubernetes API. This means we define them in YAML or JSON manifest files, and we deploy them by POSTing these manifests to the API server.

In the same way that *Replication Controllers* add features and functionality around *Pods*, *Deployments* add features and functionality around *Replication controllers*. Though with *Deployments* we actually use something called a *Replica Set* instead of a *Replication Controller*. But don’t get hung up on this, just think of a *Replica Set* as a next generation *Replication Controller*.

Back to *Deployments*… the new things they add on top of *Replica Sets* include a powerful update model, and super simple version-controlled rollbacks.

Deployments and updates

Rolling updates are a core feature of Kubernetes *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 deployed is not working.

Finally, rollbacks are super simple!

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

3: Installing Kubernetes

In this chapter we'll take a look at some of the ways to install Kubernetes.

We'll look at:

- 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 of most use.

Secondly, Kubernetes is a fast-moving project. This means that some of what we'll discuss here will change and start to feel out of date. Unfortunately I don't have a magic bullet to stop that happening. But even when this does happen, there's a good chance that a lot of what you'll learn here will still apply and you won't have wasted your time.

Minikube

Minikube is great if you're curious about Kubernetes and want to spin something up on your laptop to play around with. It's also 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 Minikube is similar to Docker for Mac and Docker for Windows under the hood. On the left is the high-level Minikube architecture and on the right is the same for *Docker for Macn* or *Docker for Windows*.

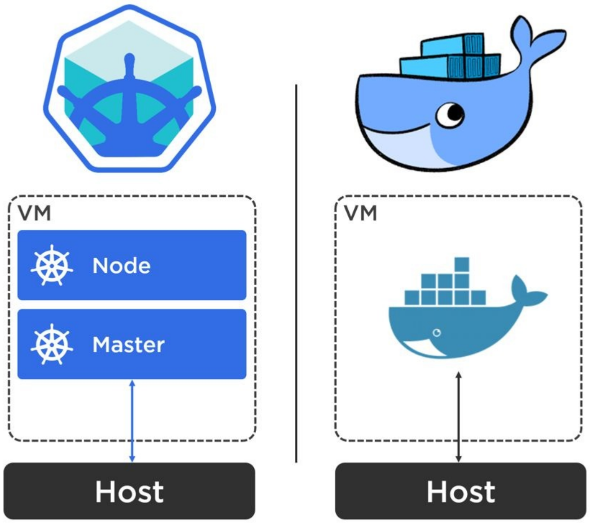


Figure 3.1

At a high level, you download the Minikube installer and type `minikube start`. This creates a local virtual machine and spins up Kubernetes cluster inside of it. It also sets up your local environment so that you can access the cluster directly from the shell of your laptop (without opening a shell inside 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 VM also runs the container runtime. At the time of writing this default's to Docker, though you can specify **rkt** instead if you require. In fact there's a good chance that **rkt** will replace **Docker** as the default container runtime at some point in the future.

This architecture is shown in Figure 3.2.

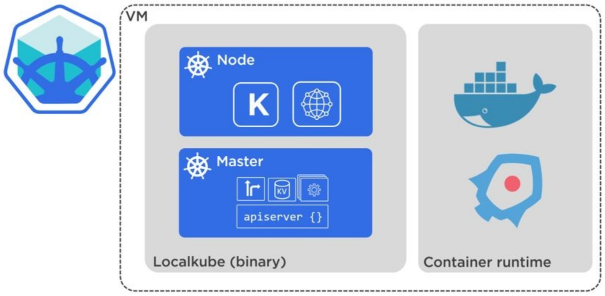


Figure 3.2

Finally, outside of the VM we have `kubectl` the Kubernetes client. The Minikube installer configured this to issue commands to the cluster inside of the VM.

However, the architecture isn’t too important. 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:"6"...
   ```

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.

```
$ 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-xhyve/bin\
/docker-machine-driver-xhyve
```

4. Allow it 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/docker\
-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!

Use `kubectl` to verify the Minikube install

You can now use `kubectl` from the shell of your Mac to manage your Minikube cluster.

The following command verifies that the kubectl context is set to Minikube (this means `kubectl` commands will be sent to the Minikube cluster).

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

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    v1.5.3
```

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 also tear it down with a single `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 properly - 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 let’s 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 the following command to start a Minikube cluster running Kubernetes version 1.6.0.

```
$ minikube start \
  --vm-driver=xhyve \
  --kubernetes-version="v1.6.0"

  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    v1.6.0
```

Bingo!

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

Installing Minikube on Windows

At the time of writing, Minkube on Windows is a lot newer than Minikube on Mac. This means the install method shown here is probably going to get a lot slicker very soon!

Before installing Minikube let’s install the `kubectl` client.

1. Point your browser to the following URL to download the `kubectl` executable.

   https://storage.googleapis.com/kubernetes-release/release/1.5.3/bin/windows/amd64/kubectl.exe

   If you look at the URL above you will see that it has the kubectl version embedded. Feel free to change this to the particular version you want to download

2. Once the download is complete copy the executable file to your system’s %PATH%.

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.
3. Start the installer and click through the wizard accepting the default options.
4. Verify the Minikube version with the following command.

   ```
   > minikube version
   minikube version: v0.17.1
   ```

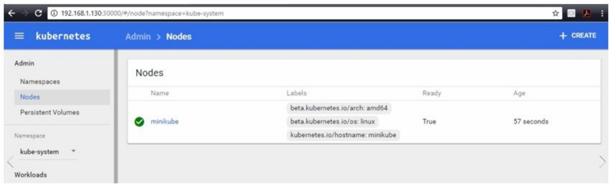
5. Use the following command to start a local Minikube instance.

   ```
   > minikube start \
     --vm-driver=hyper-v \
     --kubernetes-version="v1.6.0"

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

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



Nodes

Name	Labels	Ready	Age
minikube	beta.kubernetes.io/arch: amd64 beta.kubernetes.io/os: linux kubernetes.io/hostname: minikube	True	57 seconds

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.

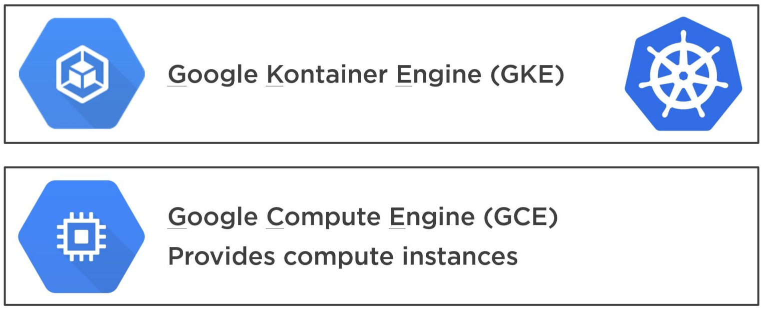

Google Kontainer Engine (GKE)
Google Compute Engine (GCE)
Provides compute instances

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 Kubernetes cluster already packaged ready for us consume.

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 in the book - 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 your Google Cloud Platform (GCP) project, open the navigation pane on the left hand side and select `Container Engine` from under the `COMPUTE` section.
2. Make sure that `Container clusters` is selected in the left-hand navigation pane and click `Create a container cluster`.

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

1. Enter a `Name` and `Description` for the cluster.
2. 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.
3. Select the `Machine type` that you want your cluster nodes to be based on.
4. Under `Node image` chose `COS`. This is the Container Optimized OS image based on Chromium OS. It supersedes the older `container-vm` image.
5. In the `Size` field 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.
6. 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.

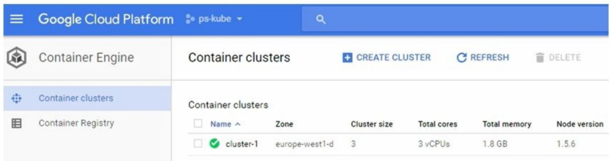
Your cluster will now be created!

Exploring GKE

Now that you have a cluster built 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 1.5.6 of Kubernetes.

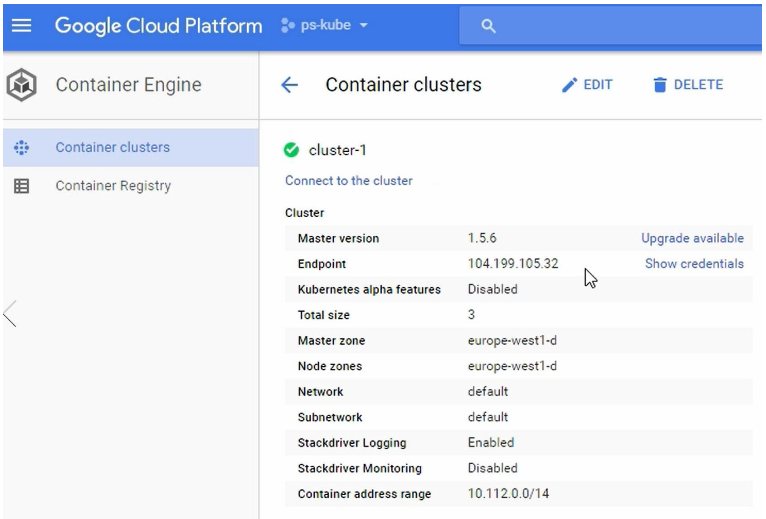


Container clusters

Name ^	Zone	Cluster size	Total cores	Total memory	Node version
✓ cluster-1	europe-west1-d	3	3 vCPUs	1.8 GB	1.5.6

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.



✓ cluster-1

Connect to the cluster

Cluster

Master version	1.5.6	Upgrade available
Endpoint	104.199.105.32	Show credentials
Kubernetes alpha features	Disabled	
Total size	3	
Master zone	europe-west1-d	
Node zones	europe-west1-d	
Network	default	
Subnetwork	default	
Stackdriver Logging	Enabled	
Stackdriver Monitoring	Disabled	
Container address range	10.112.0.0/14	

Figure 3.6

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

Run the `gcloud container clusters list` command from within the Cloud Shell. 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
clus1  europe..  1.5.6   104.199...  1.5.6     3      RUNNING
```

The output above has been clipped to make it more readable on small handheld devices.

If you click the `Connect to the cluster` link in the web UI you get presented with the commands needed to configure kubectl etc 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     v1.5.6
gke-cluster...   Ready      6m     v1.5.6
gke-cluster...   Ready      6m     v1.5.6
```

Congratulations! You now know how to create a production-worthy 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 publically 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/$(curl \
   -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin\
   /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/kops-l\
   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 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!

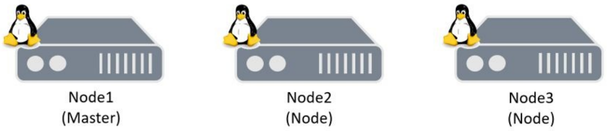


Figure 3.7

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 node\
as root:

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

The command pulls all required images and creates all of the required system *Pods* etc. The output of the command gives you a few more commands that you need to run in order 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 different ways and on various different 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). If you need to deploy some work in those worlds, you stamp it out in a VM.

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

In the Kubernetes world, it's the *Pod*!

Hypervisor Virtualization
VM
Atomic unit of scheduling
Container
Atomic unit of scheduling
Pod
Atomic unit of scheduling

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

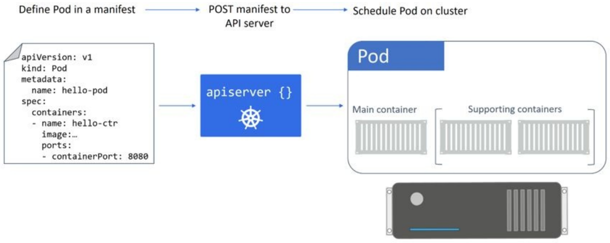
Pods sit somewhere in between a container and a VM. They're bigger, and arguably more high level than a container, but they're a lot smaller than a VM.

Under the covers a *Pod* contains one or more containers. More often than not a *Pod* only has one container, but multi-container *Pods* are definitely a thing –

they’re great if you need to tightly couple containers.

How do we deploy Pods

To deploy a *Pod* to a Kubernetes cluster you 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 *node*. Whether or not the *Pod* has one or more containers is defined in the manifest file.



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

Figure 4.2

But let’s dig a bit deeper…

The anatomy of a Pod

Every *Pod* gets its own IP. Even if a *Pod* is a multi-container *Pod*, the *Pod* still gets a **single IP**. Figure 4.3 shows two *Pods* each with their own IP. Even though one of the *Pods* is hosting two containers it still only gets a single IP.

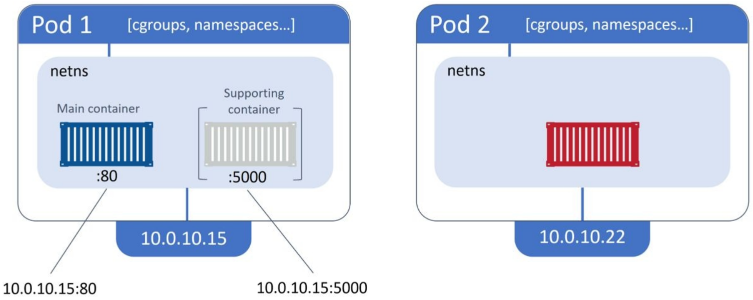


Figure 4.3

In this example we can access the individual containers in Pod 1 via their respective port numbers (80 and 5000).

Digging a little deeper we can see network namespaces playing a role. Every container inside a *Pod* shares a single network namespace that is owned by the *Pod*. This means the *Pod* actually sees not only a single IP, but also a single shared `localhost` adapter, and a single shared set of ports.

But it’s more than just networking. All containers in a *Pod* share the same cgroup limits, they have access to the same volumes, the same memory, the same IPC namespaces and more. The *Pod* holds all the namespaces - any containers they run just join them and share them.

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* network. If you read the chapter on installing Kubernetes you’ll have seen how we created a *Pod* network at the end of the manual install section. Every *Pod* gets it’s own routable IP on that network. This means every *Pod* can talk directly to every other *Pod* using IPs (Figure 4.4). There’s no need to mess about with things like nasty port mappings.

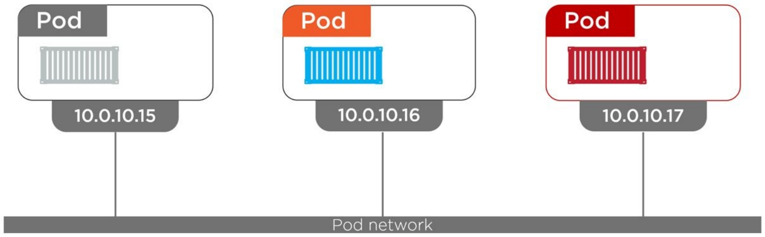


Figure 4.4 Inter-pod communication

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

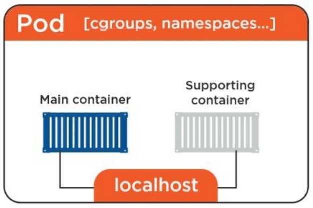


Figure 4.5 Intra-pod communication

If you do need to make multiple containers available outside of the *Pod* you do this by exposing them on individual ports. Each container needs its own port and no two containers can 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 etc. The *Pod* is at the center of the Kuberverse!

Atomic deployment of Pods

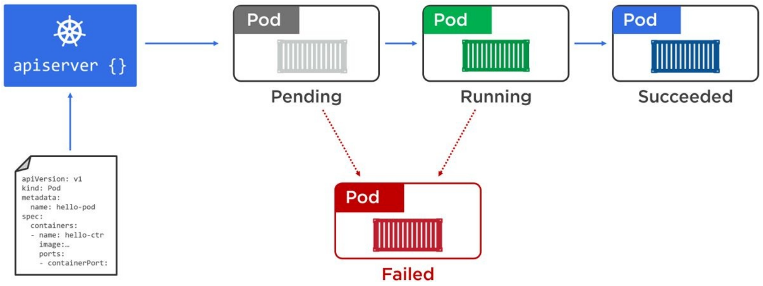
When we deploy a *Pod* it's an all or nothing job. There's no such thing as a half-deployed *Pod*. All of the containers and other aspects of a *Pod* come up and the *Pod* becomes available, or they don't come up and the *Pod* fails. For example, you can never have a situation where you have a multi-container *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 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!

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 it gets scheduled to a 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* stays in this *pending* state until all of its resources such as containers and the likes 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 there's a reason that a *Pod* can't start, it can remain in *pending* state or potentially go straight to the *failed* state. This is all shown in Figure 4.6.



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

Figure 4.6 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* model - *Pods* should be considered 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 exactly the same config, but a different ID and IP etc - magically appears and takes its place.

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-pod containers are ideal for containers that need to be tightly coupled - may be they need to share memory or volumes.
3. *Pods* get scheduled on nodes – you can't ever end up with a single *Pod* spanning 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 4.7

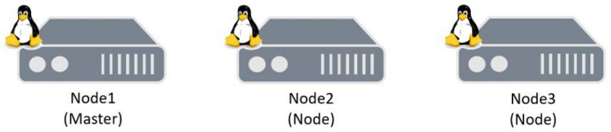


Figure 4.7

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 and 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 simple *Pod* manifest file (called pod.yml):

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hello-pod
  labels:
    zone: prod
    version: v1
spec:
  containers:
  - name: hello-ctr
    image: nigelpoulton/pluralsight-docker-ci: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`.

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 is stable. If you're working with some of the newer constructs such as *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 asking to deploy a *Pod*. As we progress through the book we'll populate this with various different values – *Replication Controllers*, *Services*, *Deployments*, 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 and you can pretty much add anything you need. We'll talk more about labels later.

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, calling it "hello-ctr", basing it off the `nigelpoulton/pluralsight-docker-ci: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.

Deploying Pods form a manifest file

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

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(s) to start the container in the *Pod*.

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

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

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

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, you’ve deployed a simple pod-based app.

Deploying Pods via Replication Controllers

It’s not very common to deploy *Pods* directly. It’s much simpler and a lot more powerful to deploy them via higher level objects such as *Replication Controllers* and *Deployments*. In this section we’ll take a look at deploying *Pods* via *Replication Controllers*.

Replication Controllers wrap around *Pods* and introduce **desired state**.

Let’s assume we’re deploying an application and require 5 replicas of a particular *Pod* running at all times. Instead of defining the *Pod* 5 times and individually deploying 5 copies of it, we’d define it as part of a *Replication*

Controller. This *Replication Controller* is defined as a simple YAML or JSON manifest and POSTed to the API server. However, as part of the *Replication Controller* manifest, we define the need for 5 replicas. This is recorded in the cluster store as part of the clusters *desired state*. Kubernetes then runs continuous background loops that are always checking to make sure that the *actual state* of the cluster matches the *desired state* (5 replicas of the *Pod* running).

Let’s take a look at how to deploy the same *Pod* as part of a Replication Controller.

If you’ve been following along with the examples you should delete the “hello-pod” *Pod* that we previously deployed.

```
$ kubectl delete pods hello-pod
pod "hello-pod" deleted
```

We’ll be using the following simple *Replication Controller* manifest for the examples that follow:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: hello-rc
spec:
  replicas: 10
  selector:
    app: hello-world
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-ctr
        image: nigelpoulton/pluralsight-docker-ci:latest
        ports:
        - containerPort: 8080
```

The first thing you should notice is that we have the same 4 top-level constructs: `apiVersion`, `kind`, `metadata`, and `spec`.

Replication Controllers are fully defined in the `v1` API. For `kind` we’re telling it to deploy a *Replication Controller*. We give it a name in the `metadata` section. Then we define it in the `spec` section.

The first thing to note about the `spec` section is the request for 10 replicas (`replicas: 10`). This gets recorded in the cluster store as part of the clusters *desired state*.

The `spec` also specifies a selector `app=hello-world`.

The `template` section is effectively an embedded *Pod* spec that defies a single-container *Pod* and includes the same `app=hello-world` label specified in the *Replication Controllers* `selector` above.

Deploy the *Replication Controller* by saving it as `rc.yml` and POSTing it to the API server using `kubectl`.

```
$ kubectl create -f rc.yml
replicationcontroller "hello-rc" created
```

You can use the normal `kubectl get` and `kubectl describe` commands to inspect the *Replication Controller*.

```
$ kubectl get rc
NAME       DESIRED   CURRENT   READY   AGE
hello-rc   10        10        10      36s
```

The output above shows that all 10 replicas are up and ready.

Chapter summary

In this chapter we learned that the smallest unit of deployment in the Kubernetes world is the *Pod*, and that deploying a *Pod* is an all-or-nothing atomic transaction.

We saw how to define a *Pod* as a YAML manifest file and POST it to the API server to be scheduled on the cluster. However, we learned that we usually deploy *Pods* via a higher level construct such as a *Replication Controller*. These add *desired state* to the game.

5: Kubernetes Services

In the previous chapter we launched a simple pod-based app and added a *Replication Controller* to the mix. But we 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:

- What is a *Service* and how do they work.
- Working with *Services*.
- 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.

What is a Kubernetes Service and do they work

To set the scene we need to know 3 things about Kubernetes *Service* objects.

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*, *Replication controller*, or *Deployment*, a Kubernetes ***Service*** is an object in the API that we define in a manifest (JSON or YAML) 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 mean that we can use a *Service* to provide stable networking to sets of *Pods*.

Figure 5.1 shows a simple pod-based application deployed via a *Replication Controller*. It also shows a client (which could be another component of the app) that does not have a reliable network endpoint to access the *Pods* through - remember that *Pod* IPs are unreliable.

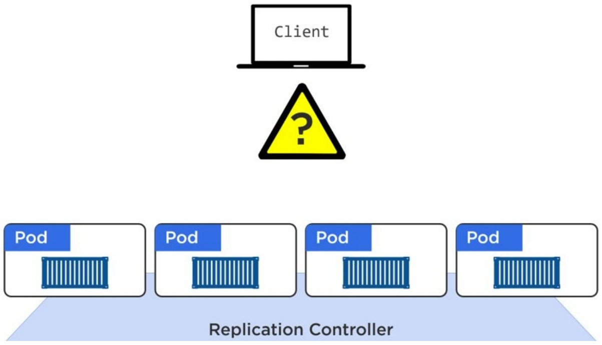


Figure 5.1

Figure 5.2 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*.

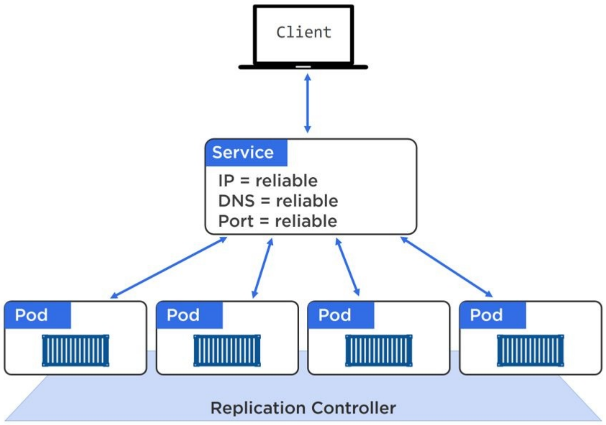


Figure 5.2

This means that the *Pods* behind a *Service* can chop and change as much as they need - they can scale up and down, they can fail, they can be updated… but the *Service* in front of them never changes!

We link *Pods* to *Services* via *labels* and *selectors*. Figure 5.3 shows an example where 3 *Pods* are labelled as `zone=prod` and `version=1`. The *Service* object in the diagram is linked to these *Pods* via a property called a *selector* - if a *Service* object's *selector* is populated with the same values as the labels on a *Pod* the *Service* will be linked to the *Pod*. In Figure 5.3 the *Service* is linked to all three

Pods and will provide a stable networking endpoint for them and load balance across them.

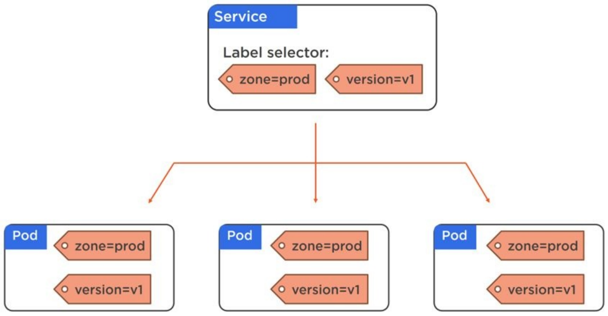


Figure 5.3

For a *Service* to match a set of *Pods* it only needs to have some of the *Pods* labels included in its *selector*. However, for a *Pod* to match a *Service* the *Pod* must match all of the values in the *Service's selector*. If that sounds confusing see the examples shown in Figure's 5.4 and 5.5.

Figure 5.4 shows an example where the *Service* does not match any of the *Pods* because the *Service* is looking to match on two labels and the *Pods* only contain one. This is a Boolean `AND` operation.

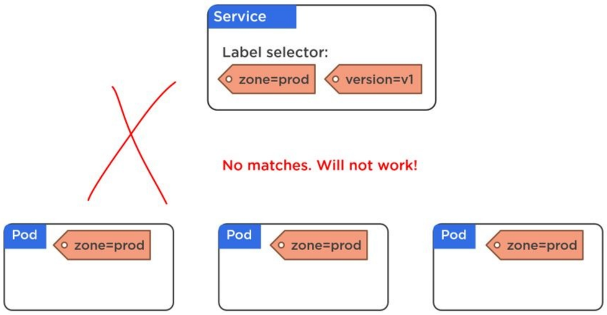


Figure 5.4

Figure 5.5 shows an example that does work because the *Service* is looking to match on two labels and the *Pods* have both. It does not matter that the *Pods* have additional labels that the *Service* does not include in its *selector*.

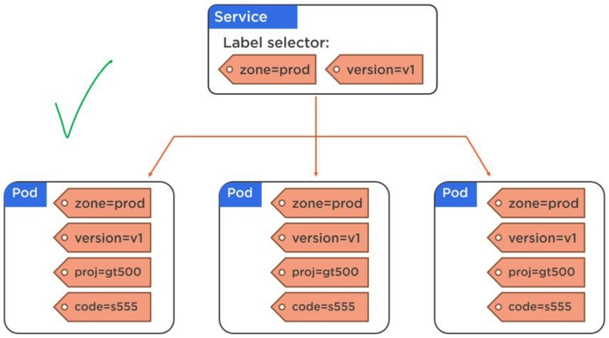


Figure 5.5

The following excerpts from a *Service* YAML and *Replication Controller* YAML show how *labels* and *selectors* are implemented.

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
```

rc.yml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: hello-rc
spec:
  replicas: 10
  selector:
    app: hello-world
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-ctr
        image: nigelpoulton/pluralsight-docker-ci:latest
        ports:
        - containerPort: 8080
```

In the example files you can see that the *Service* (svc.yml) has a *selector* with a single value `app=hello-world`, and the *Replication Controller* has a single *label* with the same value `app=hello-world`. This means that the *Service* will match all 10 replicas that the *Replication Controller* will deploy, and will provide them a stable networking endpoint and load balance over them all.

As *Pods* come and go (scaling up and down, failing, rolling updates etc.) the *Service* is dynamically updated so that it always knows about the latest set of pods that match its *selector*. There are two major constructs that make this happen - the *Service's selector* and something 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 selector*. This ensures that the *Service* is kept up-to-date as *Pods* come and go. This works because Kubernetes is continuously evaluating the *Service's selector* and matching it with live *Pods*.

The *Endpoint* object also has it's own API endpoint that Kubernetes-native apps can monitor in order to access the latest list of matching *Pods*. Non-native Kubernetes apps can use the *Service*'s 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 virtual IP. 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* virtual IP. This means that 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* and applications 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 that 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 our *nodes* in the cluster that are hosting our *Pods*. Then we create a *Service* object that's associated with the *Pods* in our app. That *Service* object has its own unique port on every node in the cluster – the port number that the *Service* uses is the same on every cluster node. This means that traffic from outside of the cluster can hit any node on that port and get through to our application (Pods).

Figure 5.6 shows an example where 3 *Pods* are being exposed by a *Service* on port `30050` on every node in the cluster. In step 1 an external client hits **Node2** in the cluster 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 that has an always-up-to-date list of *Pods* in the *Service*. Step 4 shows the client being directed to **Pod1**.

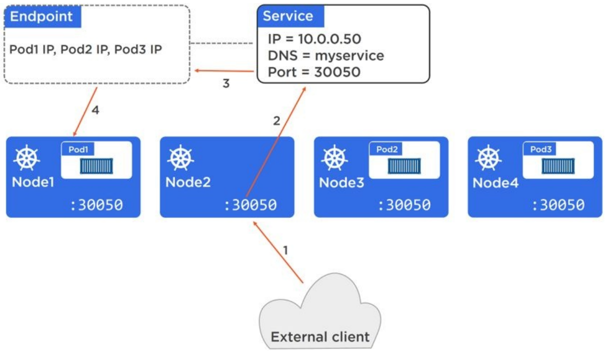


Figure 5.6

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 a bit 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 (not-so-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 kubelets (nodes) to use it for DNS.

It constantly watches the API server for new *Services* and registers each new one in DNS. This results in *Services* getting DNS names that are resolvable by other objects in the cluster.

The other way is through environment variables. Every *Pod* also gets a set of environment variables that assist with name resolution - in case you're not using DNS. However, these are inserted into the *Pod* when the *Pod* is created, meaning if you add objects **after** you create a *Pod*, the *Pod* will not know about them. This is one of the reasons that DNS is the preferred method.

Summary of Service theory

Services are all about providing a stable networking endpoint for *Pods*. We connect them to *Pods* using labels and selectors, and they load balance requests across all *Pods* behind them. They also map to a cluster-wide port that we can use for access from outside of the cluster.

Working with Services

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

We'll add a simple to *Service* to a simple pod-based app, and we'll show how to do it in two ways:

- The imperative way
- The declarative way

The imperative way

In the earlier chapters we've deployed a pod-based app via a *Replication Controller*.

The following `kubectl` command shows how we can create a *Service* and link it to our existing pod-based app.

```
$ kubectl expose rc hello-rc \
  --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 `rc hello-rc` is telling Kubernetes to expose the *Replication Controller* that we called “hello-rc”. `--name=hello-svc` tells Kubernetes that we want to call this *Service* “hello-svc”. `--target-port=8080` is telling the *Service* which port our app is listening on (this is **not** the cluster-wide port that we’ll access the *Service* on). `--type=NodePort` is telling Kubernetes we want a cluster-wide port for the *Service*.

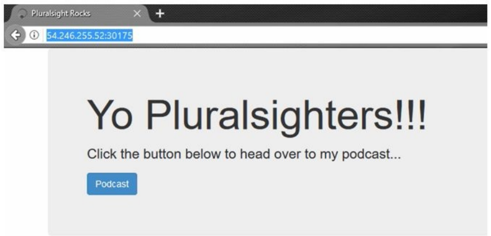
Once the *Service* is created you can inspect it with the `kubectl describe svc hello-svc` command.

```
$ 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*
- `Port` is the port that our app is configured to work on
- `NodePort` is the cluster-wide port
- `Endpoints` is the dynamic list of *Pods* that currently match the *Service’s 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 publically routable IP if you will be accessing via the internet). Figure 5.7 shows a web browser accessing a cluster node with an IP address of `54.246.255.52` on the cluster-wide port `30175`.



Yo Pluralsighters!!!

Click the button below to head over to my podcast...

Figure 5.7

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 in a more declarative way using a YAML manifest file. In order 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

In this section of the chapter we're going to see how to deploy the same *Service* in a declarative way - defining your desired state in a manifest file. The declarative way is the *Kubernetes way* and is preferred.

A Service manifest file

We'll use the following *Service* manifest file to deploy the same *Service* that we deployed in the previous section.

```
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. I expect them to exist in future versions such as `v2`.

The `kind` field is telling the API server that this manifest is defining a *Service* and should be passed to the *Service* controller.

The `metadata` section defines some key-value pairs that we'll use later.

The `spec` section is where we actually define the *Service*. 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, we're telling it to select on all *Pods* that have the `app=hello-world` label. That means the *Service* will provide stable networking and load balance across all *Pods* with the label `app=hello-world`.

The three common *ServiceTypes* are:

- `ClusterIP` This will give the *Service* a stable IP address internally within the cluster and is the default. 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. This makes the *Service* available outside of the cluster.
- `LoadBalancer` This builds on top of `NodePort` and integrates with cloud-native load-balancers.

The manifest file 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
```

The command above 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. In the example above we used a manifest file called `svc.yml`.

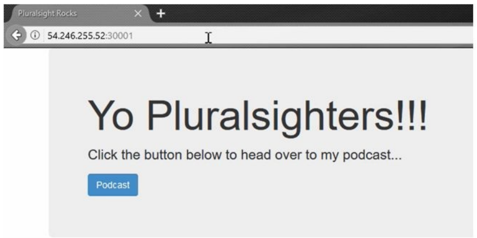
You can inspect the *Service* with the usual `kubectl get` and `kubectl describe` commands.

```
$ kubectl get svc hello-svc
NAME         CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
hello-svc    100.70.40.2     <nodes>       8080:30001/TCP   8s
Kubernetes   100.64.0.1      <none>        443/TCP          3d

$ 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* on port `30001` across the cluster. This means we can point a web browser to that port on any node in the cluster and get to our *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 group rules allow the traffic).

Figure 5.8 shows a web browser accessing our app via a cluster node with an IP address of `54.246.255.52` on the cluster-wide port `30001`.



Yo Pluralsighters!!!

Click the button below to head over to my podcast...

Figure 5.8

Endpoint objects

Earlier in the chapter we said that every *Service* gets an *Endpoint* object. 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 (*Endpoints* get the same name as the *Service* they relate to).

```
$ 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

The preferred method of deploying *Services* is the declarative way using manifest files. You can deploy new *Services* that work with *Pods* and *Replication Controllers* that are already deployed and in use. Each *Service* gets an *Endpoint* object that maintains an up-to-date list of *Pods* that match the *Service*.

Real world example

Although everything we've learned so far is cool and interesting, the important question is how does it bring value? 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 5.9 shows a simple business app deployed on a Kubernetes cluster as a bunch of *Pods* managed by a *Replication Controller*. As part of it we've got a *Service* selecting on *Pods* with labels that say `app=biz1` and `zone=prod` (notice how the *Pods* include two of the same labels as the *Service selector*). The application is up and running.

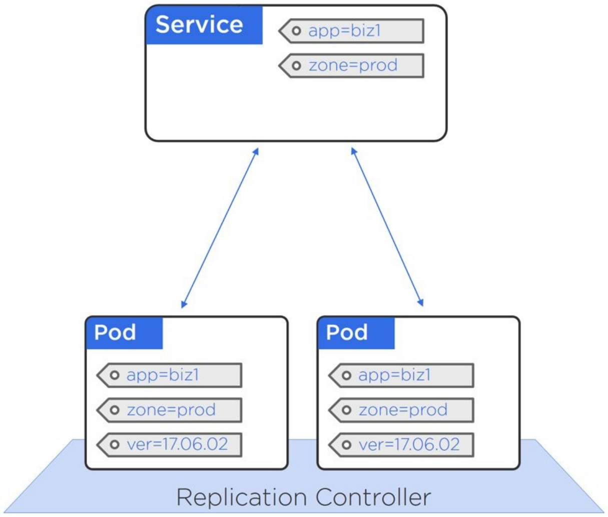


Figure 5.9

Now 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 5.10.

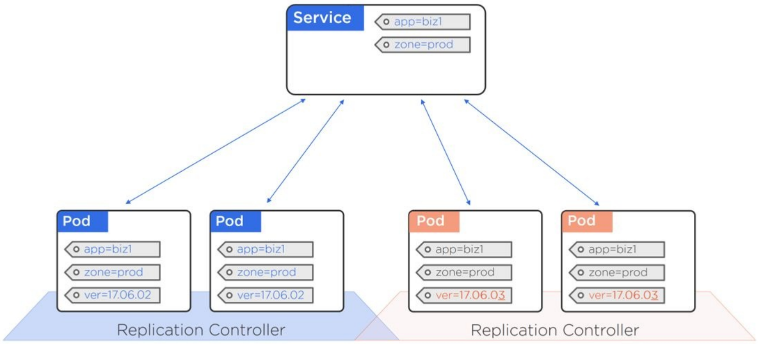


Figure 5.10

The updated *Pods* are deployed through their own *Replication Controller* and are labeled so that they match the existing *Service’s* label selector. This means that 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 watched and the *Endpoint* object and VIP load balancing are constantly being updated with new matching *Pods*.

To force all traffic to the updated *Pods* 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 now getting the new version (Figure 5.11).

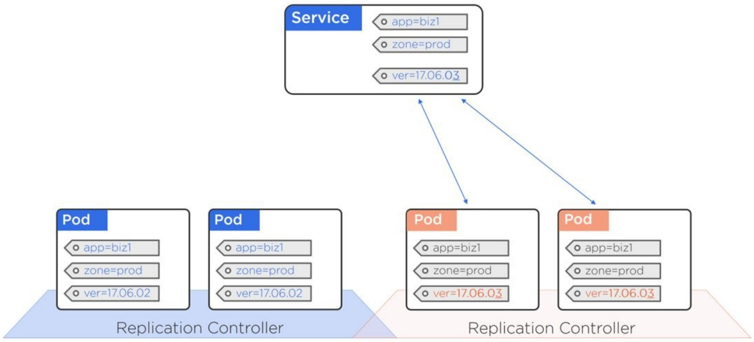


Figure 5.11

However, the old version still exists - we’re just not using it any more. This means that if we experience an issue with the new version we can switch back to the previous version by simply changing the label selector to `version=17.06.02`. See Figure 5.12.

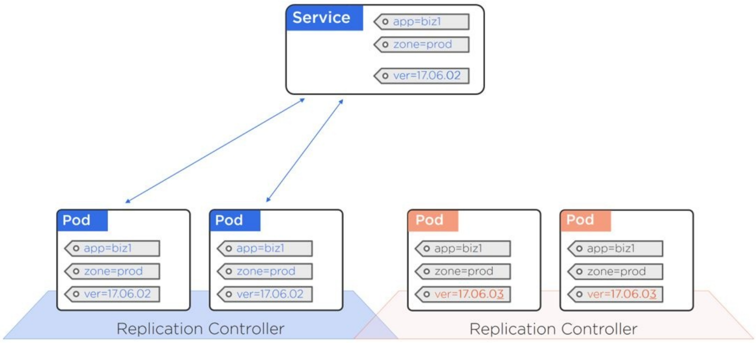


Figure 5.12

This functionality can obviously 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 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.

6: Kubernetes Deployments

In this chapter we’ll see how Kubernetes *Deployments* build on everything we’ve learned, and adds mature rolling update and rollback capabilities.

We’ll split the chapter up as follows:

- Deployment 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. Then we wrap them in a *Replication Controller* that brings scalability, resiliency, and desired state. *Deployments* take this to the next level by wrapping around *Replication Controllers* (actually Replica Sets) and adding rolling updates and rollbacks.

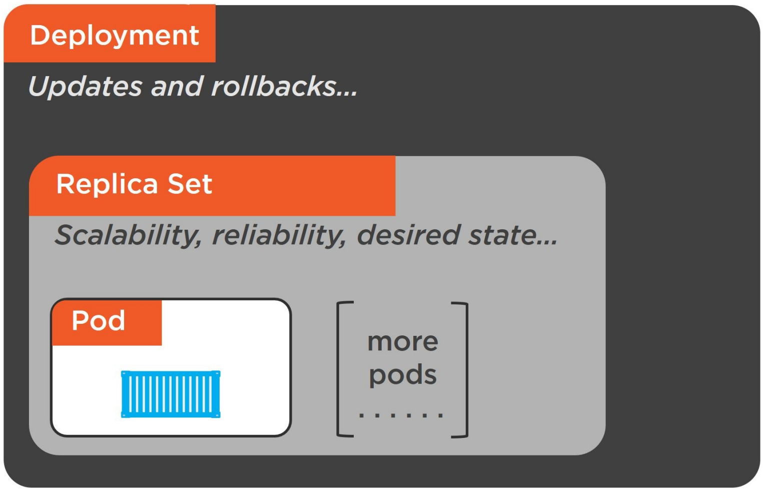


Figure 6.1

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 and feed them to the deployment controller via the API server.

A second ago we mentioned something called a *Replica Set*. *Deployments* don't actually wrap around *Replication Controllers*, they actually wrap around something newer called a *Replica Set*. There are some minor differences, but for all intents and purposes they're the same thing and we should just think of *Replica Sets* as next generation *Replication Controllers*. The fact that we deal directly with the *Deployment* and not with the *Replica Set* also means that we don't need to care too much about them.

To recap; *Deployments* manage *Replica Sets*, and *Replica Sets* manage *Pods*.

Rolling updates in Kubernetes

Before we had *Deployments* we'd deploy our apps via *Replication Controllers*. To update the app we'd create a new *Replication Controller* with another name and may be a different version label. Then we'd do a `kubectl rolling-update` and tell it to use the manifest file for the new *RC*. Kubernetes would then take care of the update.

And it 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. Roll-backs were a bit basic, and there wasn't any proper audit trailing.

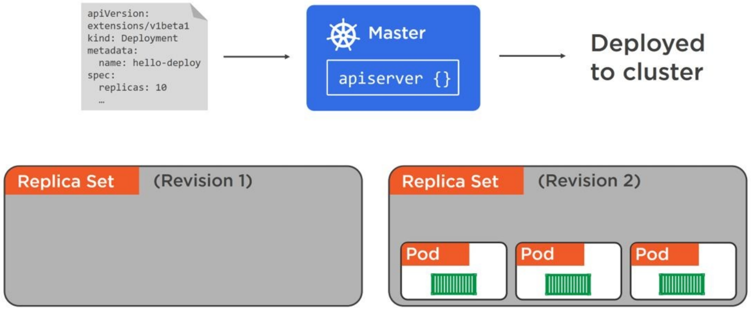
Deployments are better.

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

Behind the scenes, you get a *Replica Set* and a bunch of *Pods* – the *Deployment* takes care of creating all of that for you. Plus you get all of the background loops that make sure actual state meets desired state. All the usual stuff.

When you need to push an update, you make the changes to the **same *Deployment* manifest file** and push it to the API server again. In the background Kubernetes creates another *Replica Set* (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 you can rinse and repeat the process for future updates - just keep updating that manifest file which you can manage in a proper source code repository. Brilliant!

Figure 6.2 shows a *Deployment* with two revisions. The first revision was the initial deploy to the cluster and the second revision is an update that has been made. You can see that the *Replica Set* associated with revision 1 has been wound down and no longer has any *Pods*. The *Replica Set* associated with revision 2 is now active and owns all of the *Pods*.



```
apiVersion:
extensions/v1beta1
kind: Deployment
metadata:
  name: hello-deploy
spec:
  replicas: 10
  …
```

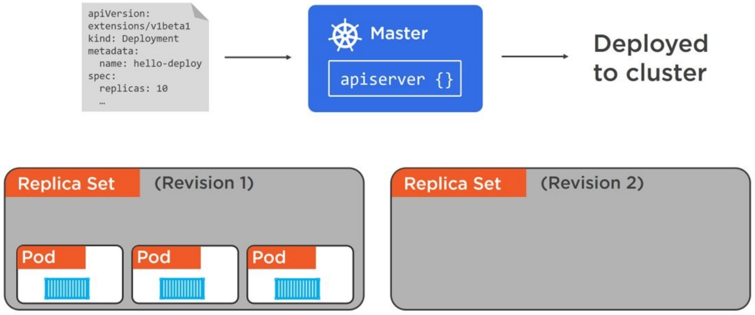
Figure 6.2

Rollbacks

As we can see in Figure 6.2, the old *Replica Sets* stick around and don't get deleted. Obviously they've been wound down so aren't managing any *Pods* any more, but their definitions still exist. This makes them a great way to revert to previous versions.

This means that to rollback we just do the opposite of a rolling update - we wind up one of the old *Replica Sets* and wind down the current one. Simple!

Figure 6.3 shows the same app rolled back to revision 1.



```
apiVersion:
extensions/v1beta1
kind: Deployment
metadata:
  name: hello-deploy
spec:
  replicas: 10
  …
```

Figure 6.3

There's more as well. There's intelligence built in so that you can say things like "wait X number of seconds after each *Pod* comes up before you 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 *Replication Controllers*.

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. You can obviously do the same imperatively using the `kubectl run` command, but the preferred way is the declarative method.

If you've been following along with the examples in the book you'll have a *Replication Controller* and a *Service* running. To follow with these examples you'll need to delete the *Replication Controller* but keep the *Service* running.

```
$ kubectl delete rc hello-rc
replicationcontroller "hello-rc" deleted
```

The reason we can leave the *Service* running is that we're going to deploy the same app again using the same port and labels. This means the *Service* we already have deployed will work with the new *Pods* we'll deploy via the *Deployment*.

This is the `deploy.yml` *Deployment* manifest file we're going to use:

```
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/pluralsight-docker-ci: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 `apiVersion: apps/v1beta1`. This is because *Deployment* objects are relatively new at the time of writing. As a result they’re not defined in the `v1` API, so we need to specify this beta API extension (for versions prior to 1.6.0 you should use `extensions/v1beta`). As and when the `v2` and later APIs comes out, you should start using them - *Deployments* will be in there as well.

Next we’re specifying `kind: Deployment` to tell Kubernetes we’re defining a *Deployment* - this will make Kubernetes send this to the deployment controller.

After that, we give it a few properties that we’ve already seen with *Replication Controllers*. Things like giving it a name and telling it to log 10 replicas as part of the desired state.

After that, we define a few *Deployment* specific parameters and give it a *Pod* spec. We’ll come back and look at these bits later.

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

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

If you want, you can omit the optional `deployment` keyword because we defined `kind: Deployment` in the manifest file.

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          1s
```

```
$ 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:        0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
```

```
Pod Template:
  Labels:               app=hello-world
  Containers:
        hello-pod:
          Image:        nigelpoulton/pluralsight....
          Port:         8080/TCP
<SNIP>
```

The command outputs above has been trimmed for readability.

As we mentioned earlier, *Deployments* automatically create new *Replica Sets* which we can also see using the usual commands.

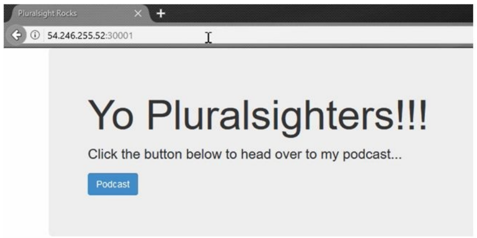
```
$ kubectl get rs
NAME                  DESIRED    CURRENT   READY    AGE
hello-deploy-4872...  10         10        10       43s
```

Right now we only have one *Replica Set*. 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 part of the manifest file.

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

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 (see previous chapter).

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 6.4.



Yo Pluralsighters!!!

Click the button below to head over to my podcast...

Figure 6.4

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 *Depoloyment* manifest file. The initial version of the app that we deployed used an image tagged as `nigelpoulton/pluralsight-docker-ci:latest`. We will update this in the manifest file to be `nigelpoulton/pluralsight-docker-ci:edge`. This is the only change we need to make to ensure the updated version of the app will run the newer image tagged as `edge`.

The following is an updated copy of the `deploy.yml` manifest file (the only change is on 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/pluralsight-docker-ci:edge
        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.

In the `spec` section we have the `minReadySeconds: 10` map. This is going to tell 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 the update occurs - longer waits give you a chance to spot problems before all of your *Pods* have been updated.

We also have a nested `strategy` map that tells Kubernetes we want this Deployment to update using the `RollingUpdate` strategy, to only ever have one *Pod* unavailable (`maxUnavailable: 1`) and to never go more than one *Pod* above the desired state (`maxSurge: 1`). As the current 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.

Now that we have our 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.

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

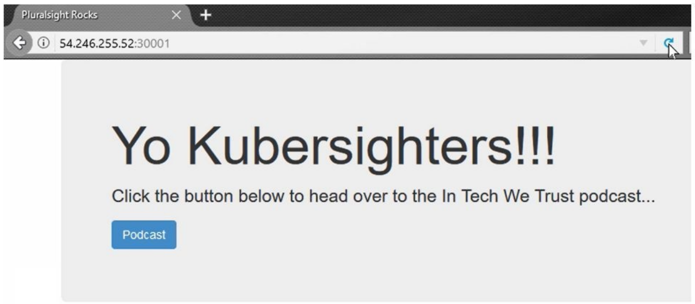
```
$ kubectl rollout status deployment hello-deploy
Waiting for rollout to finish: 4 out of 10 new replicas...
Waiting for rollout to finish: 4 out of 10 new replicas...
Waiting for rollout to finish: 5 out of 10 new replicas...
^C
```

Once the update is complete you 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 6.5).



Yo Kubersighters!!!

Click the button below to head over to the In Tech We Trust podcast...

Figure 6.5

How to perform a rollback

A moment ago we used the `--recrod` flag on the `kubectl apply` command that we used to perform the rolling update. This is an important flag that will maintain a revision history of the *Deoployment*. 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 - 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 this flag.

Earlier in the chapter we also said that updating a *Deployment* created a new *Replica Set* and that the previous *Replica Sets* were not deleted. We can verify this with a `kubectl get rs`.

```
$ kubectl get rs
NAME                DESIRED  CURRENT  READY  AGE
hello-deploy-47...  0        0        0      10m
hello-deploy-93...  10       10       10     2m
```

The output above shows that the *Replica Set* for the initial revision still exists (`hello-deploy-47...`) but that it has been wound down and has no associated *Pods*. The `hello-deploy-93...` *Replica Set* associated with the latest revision is active with 10 replicas under management. However, the fact that the previous version of the *Replica Set* 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       2           10        11m
$
$ kubectl rollout status deployment hello-deploy
Waiting for rollout to finish: 2 out of 10 new replicas...
Waiting for rollout to finish: 3 out of 10 new replicas...
Waiting for rollout to finish: 3 out of 10 new replicas...
^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 our apps in Kubernetes. They provide everything that older *Replication Controllers* provide, plus they add mature and configurable update and rollback capabilities.

Like everything else, they’re objects in the Kubernetes API and we should be looking to work with them declaratively. If you’re using the `v1` API you will need to use the beta1 extensions, but when the `v2` API comes out you should be able to use that.

We learned that *Deployments* use *Replica Sets* instead of *Replication Controllers*. When we perform updates with the `kubectl apply` command older versions of *Replica Sets* get wound down, but stick around making it easy for us to perform rollbacks.

7: What next

Hopefully you’re now comfortable talking about Kubernetes and working with it.

Taking your journey to the next step is simple in today’s world. 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!

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 its definitely a service where you get value for your money! And if you’re unsure… they always have a free trial period 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

A massive thanks for reading my book. I hope you loved it!

Now time for me to ask for a favour… *it takes an insane amount of effort to write a book and the money is rubbish! So do me a huge favour and give the book some stars on Amazon, I’d really appreciate it!*

Feel free to hit me on Twitter as well, but stars on Amazon are what I dream of at night ;-)

@nigelpoulton

Thanks again for reading my book and good luck driving your career forward!!