

Kubernetes for Developers

William Denniss

MEAP



MANNING

sadfsdfasdasafdsadfsafdafds





**MEAP Edition
Manning Early Access Program
Kubernetes for Developers
Version 14**

Copyright 2023 Manning Publications

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing *Kubernetes for Developers*. I'm writing this book to help developers like you take their application and get it running on Kubernetes, whether it's your first web app, or you're moving an existing deployment onto Kubernetes.

Kubernetes is a highly capable platform, able to represent a wide range of application deployments and configurations, but this flexibility can make it seem vast and complex at first. My focus is to teach you the most important bits to get your application deployed, and to take advantage of everything it has to offer, like automated operations and rollouts to keep your application running and up to date. This book isn't intended as a general reference on Kubernetes, for that I recommend *Kubernetes in Action* also published by Manning, as well as the project documentation itself.

When you get to the end of Part 1, you should have all you need to know to confidently deploy your application into a Kubernetes cluster and keep it running. Part 2 then goes into some more advanced topics and troubleshooting for when you need to do something a little more complex, like retain state in an application, or even deploy your own database (if you really need to).

One of the reasons why I wanted to publish this book with Manning was their fantastic Author Online forum where you can give your feedback and ask questions. I've given feedback in these forums as a reader and was thrilled to see the authors adopt my suggestions, so now I'm hoping to do the same with your feedback. So please visit the [liveBook's Discussion Forum](#) and leave your comments.

—William Denniss

brief contents

PART 1

- 1 Kubernetes for Application Deployment*
- 2 Containerizing Apps*
- 3 Deploying to Kubernetes*
- 4 Automated Operations*
- 5 Resource Management*

PART 2

- 6 Scaling Up*
- 7 Internal Services and Load Balancing*
- 8 Node Feature Selection*
- 9 Stateful Applications*
- 10 Background Processing*
- 11 GitOps: Configuration as Code*
- 12 Securing Kubernetes*

1

Kubernetes for Application Deployment

This chapter covers

- The benefits of packaging applications in containers
- What makes Kubernetes an ideal platform to deploy containers with
- Deciding when to use Kubernetes

It's 5pm on a Friday, and the product you've been working on for the last year just went viral. You need to scale everything, and fast. Is your application and the platform it runs on capable of scaling 100x, ready to capture your success, or are you stuck with brittle code and an inflexible platform, meaning all your work will be for naught?

You built an amazing stateless application on a popular application platform capable of scaling rapidly, and everything is working great. Until one day, your business requirements shift, and suddenly you need to run a bespoke stateful application for some critical business data, or need to configure a nightly batch processing pipeline. Will these new workloads fit in seamlessly with your existing ones, or will you need to start over from scratch or patch together multiple disparate systems?

Kubernetes is fast becoming the industry standard for running workloads of all different shapes and sizes, and at different scales—for a reason. It enables you to launch containerized applications with the ability of scaling rapidly, while at the same time handling a variety of complex deployment patterns, from stateless applications to stateful databases, batch jobs with ephemeral storage, and so on. Invented by Google¹, open-sourced, and used by countless organizations like Spotify², CapitalOne³, and OpenAI⁴, Kubernetes is an open,

¹ <https://cloud.google.com/learn/what-is-kubernetes>

² <https://kubernetes.io/case-studies/spotify/>

³ <https://kubernetes.io/case-studies/capital-one/>

vendor agnostic and well proven platform that is to cloud deployments what Linux is to operating systems.

However, with great power comes... a bit of a learning curve. Being a general purpose platform capable of handling so many different deployment constructs, Kubernetes can be a daunting thing to learn. I'm here to tell you though that a) it's not as hard as people make it out to be, and b) it's worth learning. If you start with the basics and gradually layer on new constructs (which is how this book is structured), it's a lot more approachable. You can get a stateless app deployed with a few lines of YAML, and build your knowledge from there.

As a professional, when you are faced with a problem, like how best to deploy your applications, I believe the right answer isn't always to pick the simplest option available that can solve your immediate concern, but rather to invest your time to learn a system that can meet your needs both now and in the future. A platform that allows you to develop your skills and grow professionally as your requirements evolve. Kubernetes fits this bill. You can get up and running in hours with some simple deployments, while knowing there exists a wealth of functionality ready for you to learn and employ when, and if, you need it.

If you're already sold on the idea of Kubernetes, I suggest skipping to Chapter 2 to start building a docker image, and if you already know what a Docker container is and want to start deploying to Kubernetes, go right to Chapter 3. The rest of this chapter covers why Kubernetes and containers are proving so popular for application deployment.

1.1 Why Containers?

Kubernetes is a deployment platform for containers. All code deployed into Kubernetes, like your application, needs to be first packaged into a container. What are containers, and why bother with them at all?

Containers are the modern way to package and run applications. Unless you're running one application per host (which is pretty inefficient), you typically want some way to deploy multiple applications onto a machine, and/or collection of machines. What are the choices?

Before virtual machines (VMs), it was common to install each application into a different directory on a shared host, served each on a separate port. This presents a few problems in that the various applications need to cooperate with each other to some extent when it comes to sharing dependencies and the resources of the machine like CPU, memory and available ports. It can also be hard to scale: if you have one application that suddenly is receiving more traffic, how do you scale just that application, while leaving the others as they are?

More recently with VMs, the solution was to package each application into a virtual machine of its own. In this way, each application has its own operating environment, so dependencies can be isolated, and resources divided up and allocated. Since each virtual machine has the complexity of an individual host though, you now need to maintain the operating system and all packages for each application which has high overheads and is complex to maintain.

⁴ <https://kubernetes.io/case-studies/openai/>

Which brings us to containers. Containers are a way to package up just your application and its required dependencies, for hosting in an isolated environment much like a virtual machine, but without needing to install and manage an operating system with the application.

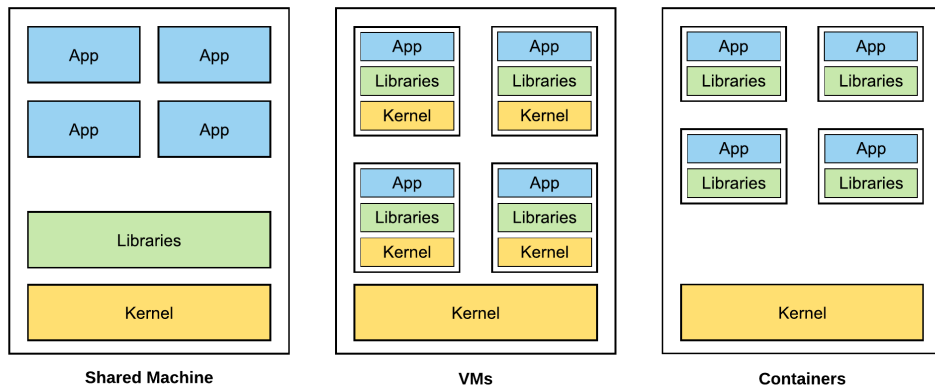


Figure 1.1 evolution of shared hosting architectures

Figure 1.1 illustrates the evolution of hosting services, from running multiple workloads on a single host, to running them on separate VMs, and finally containers. As you can see, containers provide many of the benefits of VMs but without the overheads of running another operating system kernel, making them the logical modern path forward.

1.1.1 Container Benefits

Some of the top reasons people choose containers are for the language flexibility (being able to run any language or environment on a container platform), lightweight isolation (protect your workloads from interfering with each other without using VMs), developer efficiency (bringing production closer to development, and allowing easy setup), and reproducibility (recording the steps used to create the environment in the container build file).

LANGUAGE FLEXIBILITY

Containers unbind you from language or library requirements from your deployment systems. You can bring any language, and update any package. No longer are you locked into specific languages and versions, or stuck with some outdated version of a critical dependency that shipped in the operating system years ago, as you might be on a traditional PaaS.

There are no shared libraries between two containers running on the same host, meaning the configuration of one will not interfere with the other. Need two different versions of Java, or some random dependency? no problem. This isolation extends beyond just the libraries of

the containers: each container can use a completely different base OS and package manager, for example one using Debian and apt-get, while another uses CentOS and rpm.

This flexibility makes it simpler to potentially string together a system from multiple services (a pattern known as microservices), each maintained by separate teams, with their own dependencies or languages.

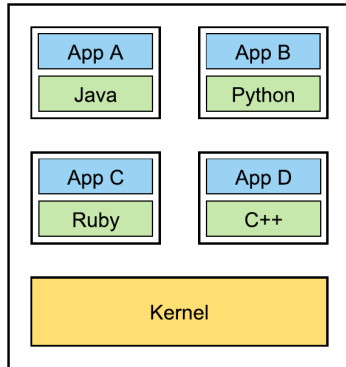


Figure 1.2 3 Containers with different languages sharing a host

ISOLATION WITHOUT OVERHEAD

In the past, to achieve isolation between multiple apps running on the same host, you would use virtual machines (VMs). VMs are heavier, both in image size and CPU / memory resource overhead, as the kernel and much of the OS is duplicated in each VM.

While containers are lighter than VMs, they still offer most of the same resource isolation benefits. You can limit your containers on Kubernetes to use only some of the resources of the host, and the system will restrict them from using more. This ultimately means you can pack more applications onto a single host, reducing your infrastructure costs.

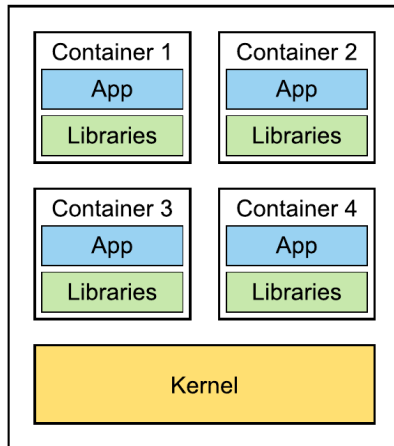


Figure 1.3 Four containers running on the same host, fully isolated but sharing the kernel.

DEVELOPER EFFICIENCY

What makes containers great for production by isolating dependencies also makes them great for development, as you can develop a myriad of applications on a developer machine without needing to configure the host with the dependencies of each.

In addition to developing Linux applications directly on Linux, with Docker you can use macOS or Windows workstations to develop a Linux container, without needing to create a version of the app that runs natively on those platforms, eliminating platform-specific configurations for development.

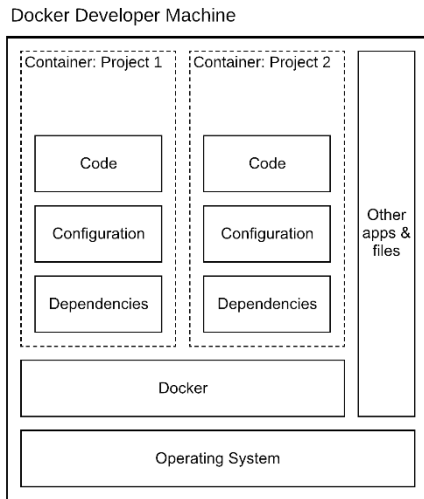


Figure 1.4 Developer Machine with two container-based projects

No longer do you need to have pages of setup instructions for developers to get started either, as setup is now as easy as install Docker, checkout the code, build and run. Working on multiple projects within a team or for different teams is now simple as well as each project is nicely isolated in its container without needing a particular host configuration.

With containers, your development and production app looks very similar, and can be the exact same container. No more development idiosyncrasies getting in the way like MacOS having a different MySQL library, or subtle differences in the way the code is packaged for production. Trying to diagnose a production issue? Download that exact container, run it against your development environment and see what's up.

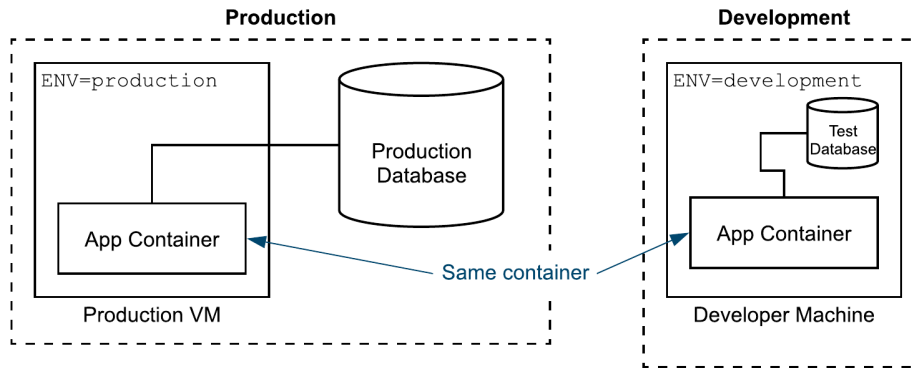


Figure 1.5 Same container being deployed in the production and development environments

REPRODUCIBILITY

Containers make it easier to reproduce your application environment as well. Imagine you have a VM on which your application is deployed, and you need to configure TLS for secure “https” connections. You SSH into the production host, add the TLS certificates to a folder. It didn’t work, so you add them to another folder. Soon they’re in 3 folders and it’s working so you don’t touch it. A year later, you need to update the TLS certificate. Can you remember how, and which of the 3 locations need to be updated?

Containers solve this. Rather than SSHing and tweaking the state, you would add the TLS certificate as a build step in the container. If it didn’t work, you’d tweak that build step until it does—but crucially only keeping the step (or steps) that actually do work. The files added in this step are also nicely isolated from the rest of the system, so essentially you’re capturing the delta—or differences—over the base system, just those modifications you needed to make. This means that a year later when you need to update the certificate, you just replace the certificate file, and re-run the container build, and it will put it in the right place.

Listing 1.1 pseudocode Dockerfile

```
Use the Debian OS
Copy and configure TLS certificate
Copy application
```

Above is a “pseudocode” example of a Dockerfile, that is, where the code to configure the container is expressed in plain English. In Chapter 2, we’ll present this same concept in the Docker script itself.

NOTE Docker as a tool for creating containers isn't perfect for reproducibility. Commands like "apt-get" to install a dependency operate on a live system, so you won't actually get the same output for the same input, as those dependent systems (like the apt-get repository) may have changed in between builds. Tools like Bazel, open sourced by Google, are designed to solve this problem and more, but come with their own complexities and are more recommended for sophisticated enterprise deployments. Despite this limitation, Docker's build system is still a heck of a lot more reproducible than trying to remember what you did a year ago when you SSH'd into that Linux box to fix an issue, and is good enough for most.

1.2 Why Kubernetes?

If containers sound like a great idea for packaging your application, you'll still need a way to actually run and manage those containers. Sure, you could just run a container or a handful of containers on each host, in much the same way it is possible run a bunch of different applications from folders, or VM images, but operating like this tends to create special snowflakes of machines, and limits your ability to scale due to the high-touch required to configure and manage hosts.

A better option is to have a shared pool (cluster) of machines (nodes), and use a so-called container orchestrator (like Kubernetes) to run your containers on this resource pool. In this way, machines are managed together as a group, where none need to be ascribed any special meaning. Should one fail, another will be there to pick up the slack. This pattern gets you out of the individual-machine business, and allows you to scale your application faster than your team size.

Previously systems that could flexibly orchestrate containers at scale were the domain of large companies. Kubernetes, and in particular managed Kubernetes offerings on public clouds, make this operations model accessible to deployments of all sizes, from a single container application running on 1 machine, to set of microservices each published by a different team running on a 15,000 machine behemoth.

Kubernetes also makes it easy to achieve *high availability* for your applications. As illustrated in Figure 1.6, we can deploy the same service across multiple availability zones, where the loss of an entire zone would not result in downtime. With a manual deployment system, this can be complex, but we can achieve such deployment patterns rapidly in Kubernetes by simply defining what we want to see (in this case, containers spread over multiple zones). Section 8.2.1 covers this.

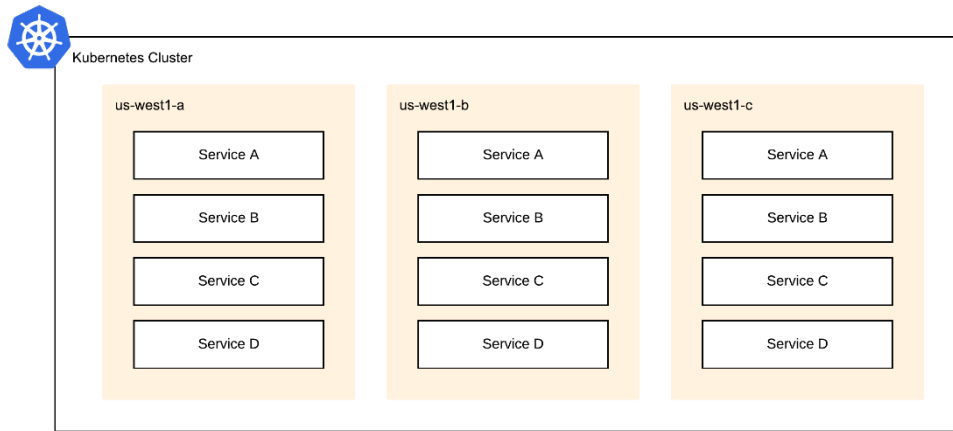


Figure 1.6 a Kubernetes cluster operating in 3 zones, managing 4 services

The best part is that updating a service in Kubernetes requires a single line of config to be changed, and Kubernetes will handle the roll out of the update to each of the zones for you, per your requirements. Updates to the Kubernetes platform itself happen in a similar, automated fashion (provided you are using a managed platform which handles this), where nodes are replaced gradually with updated versions, and your workload migrated to avoid downtime.

If your app isn't big enough to require a high-availability multi-zone deployment, fear not – Kubernetes can run at small scale too, with the added benefit that you can scale up when you need.

Kubernetes has gained popularity as it automates much of the operational aspects of scheduling and running containers on a pool of resources, and provides the level of abstraction to developers that seems to have hit the sweet spot. It isn't too low-level that you are worried about individual machines, but it doesn't go too high level either in ways that limit what workloads you can deploy.

1.2.1 Composable Building Blocks

In Kubernetes, containers are grouped into what are called "Pods". A Pod is simply a set of containers that get scheduled together and treated as a single unit. Quite often this is just a single container, but it could be multiple in the case where your app consists of multiple connected parts. Conceptually, the Pod is your application and its dependencies. Services are used to provide connectivity to groups of pods, both internally within the cluster, and externally. Figure 1.7 illustrates the resources of a typical app deployed to a Kubernetes cluster.

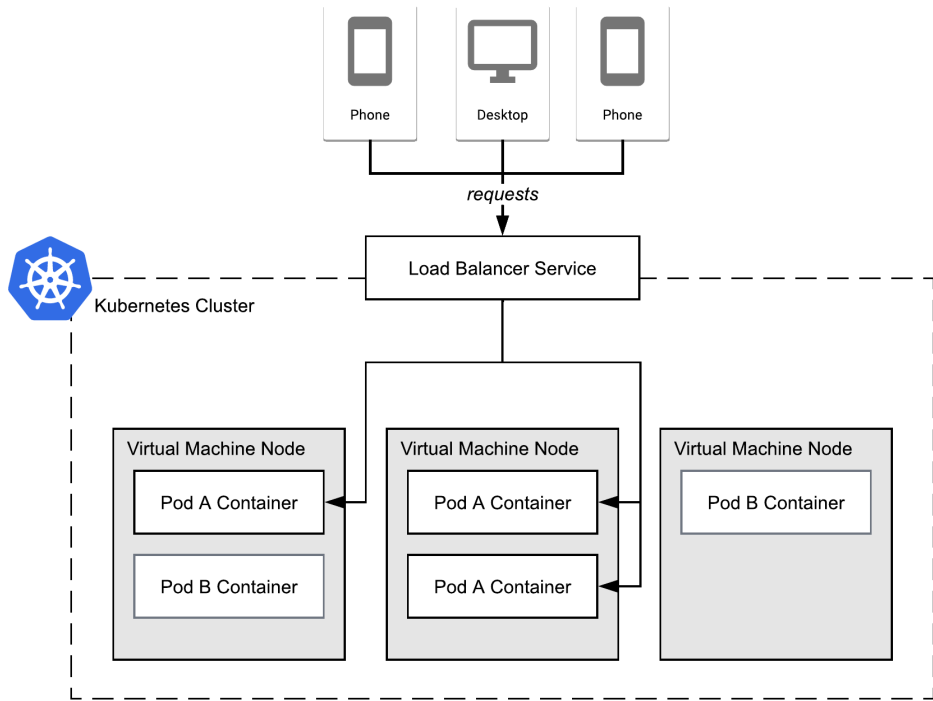


Figure 1.7 A Kubernetes cluster with several virtual machines running two different application containers, exposed with load balancing.

Kubernetes has several higher-order workload constructs, described throughout this book, that encapsulate Pods. For a stateless application, you will create a “Deployment” object that wraps the Pod definition (specifying your container versions) where you specify how many replicas (instances) you want. In all these cases, Kubernetes will do the heavy lifting of finding space in your cluster to place the Pods according to your requirements.

The range of workload types you can describe in Kubernetes configuration is wide and varied, and includes:

- Stateless applications
- Databases and other applications with persistent state
- Applications formerly configured in a VM
- A batch process you wish to run at a certain schedule.
- A batch task you want to run once, like training a Machine Learning (ML) model.

In all cases, the applications are containerized and grouped in Pods, and you describe to Kubernetes in configuration files how you want your workload to be run.

1.2.2 Features and Benefits

Here are some of the top reasons why people choose Kubernetes to deploy their containers.

AUTOMATED OPERATIONS

Provided you configure your deployment correctly, Kubernetes will automate various aspects of your deployment. Processes running on the node restart containers that crash, while liveness and readiness probes continue to monitor the container's health and ability to serve live traffic. Pod auto-scalers can be configured on your deployments to automatically increase the number of replicas based on metrics like CPU utilization.

Kubernetes itself doesn't repair compute node level issues. However, you can choose a managed platform that will provide such automation. Take, for example, the Autopilot mode of Google Kubernetes Engine (GKE): it automatically provisions the compute capacity for your pods, scaling up and down automatically as you change your replica count, and will repair and upgrade nodes as needed.

HIGH SCALE

No matter the size of your application, you will want to think about how it will scale. Whether you are deploying a huge enterprise application, or you are a bootstrapping startup, you will need a solution that can scale as you do. The time when you need to scale is not the time to start thinking about how you are going to scale!

It is hard enough to create a successful product; the last thing you want in your moment of success—when everyone is beating down your door trying to use your product—is for your application to go offline. In that moment and perhaps even in the months and years to come, you're likely not going to be able to completely rearchitect your application for scale.

Kubernetes can handle applications of any size. You can have a single-node cluster with a single CPU and a bunch of memory, or a multi-thousand node behemoth like the 10s of thousands of cores Niantic used to run Pokémon Go when it launched⁵.

Of course, your application itself will need to have properties that enable it to scale, and so will any dependencies, particularly database ones—but at least you can rest assured that your compute platform will scale as you do.

A WORKLOAD ABSTRACTION

Abstraction layers are great, until they aren't. It is a challenge to find tools that abstract away precisely those things you do not want to care about, without hiding details you do care about, but in my experience, Kubernetes comes the closest to achieving exactly that.

Infrastructure as a Service (IaaS) is a hardware-level abstraction. Rather than interacting with actual machines with spinning disks and network cards, you interact with an API that provides software that implements those same interfaces.

Kubernetes by comparison is a workload-level abstraction. Meaning that you describe your application in workload terms. For example, I have a server that needs to run in a distributed fashion; I have a database that requires certain disk volumes to be attached; I have a logging utility that needs to run on every node; or maybe I have a movie to render,

⁵ <https://cloud.google.com/blog/products/gcp/bringing-pokemon-go-to-life-on-google-cloud>

one frame at a time, on the cheapest resources available. All these deployment constructs and more can be represented natively in Kubernetes.

Kubernetes provides a layer above compute instances (VMs), freeing you from the need to manage or care about individual machines. You specify what resources your container needs: CPU, memory, disk, GPU, etc. A managed Kubernetes platform will also typically provision the compute capacity to handle your workloads. You don't need to worry about individual machines, but you can still do things that you would expect at a machine level, like write to a persistent local disk, tasks that until recently were often not possible at this level of abstraction.

The abstraction layer also remains quite clean, by not interfering with your application. Unlike many traditional Platform as a Service (PaaS) environments, Kubernetes does not modify how your app runs; for example, no code is injected or changed, and very little restrictions are placed on what your app can do. If the app can be run in a container, then it can likely be run on Kubernetes.

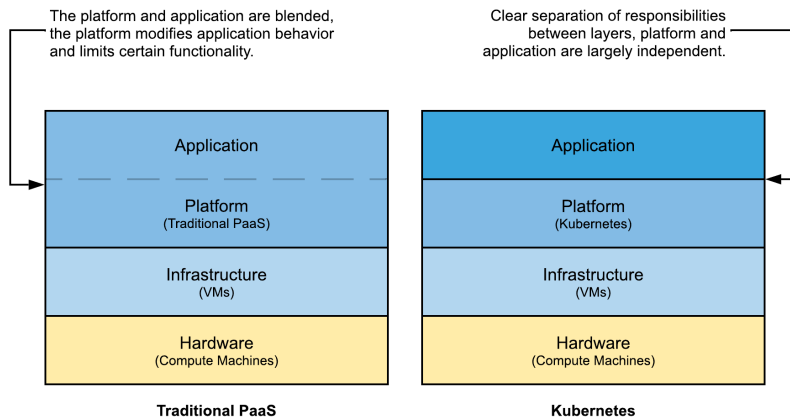


Figure 1.8 Illustration of the separation of concerns between the different compute layers

DECLARATIVE CONFIGURATION

Kubernetes uses a declarative resource model. You describe your workload in configuration (primarily YAML files), and the system seeks to enact your configuration and make it a reality. For example, if in the Deployment you specify that you want 3 replicas (copies) of your application, connected by a load balancer to the outside world, Kubernetes will find space in your cluster to run those 3 replicas, and attach a load balancer. Not only does Kubernetes place these replicas initially, but it will continue to monitor them and attempt to keep them running in the event of a crash or failure.

Declarative configuration is useful because it allows you to describe what your desired state is (e.g. run 3 copies of my application), and let Kubernetes do the work of actually

producing that state. As opposed to you issuing imperative commands (e.g. create 3 copies of my application), and doing the monitoring and adjusting yourself (like querying how many copies of my application are currently running and adjusting accordingly).

COST EFFICIENCY

Kubernetes takes the lowest-level compute building blocks—virtual machines—and makes them easy to manage. Whereas in the past, you might have assigned one app per virtual machine for maintenance reasons, Kubernetes allows you to efficiently host multiple instances of an app or apps on a single machine for high efficiency (so-called bin-packing). The combination of using commodity building blocks (raw compute nodes) with robust orchestration of the workloads often makes Kubernetes attractive from a price perspective.

Beyond bin-packing (running multiple services on one machine), resource pooling is another benefit of Kubernetes that improves efficiency. Your workloads can be configured in a way where they have a set amount of guaranteed resources, and when there's a usage spike, burst into the capacity that other containers have reserved but are not currently using.

EXTENSIBILITY

When you need to do something that Kubernetes can't, you can source or even write your own Kubernetes-style API to implement it. This isn't for everyone, and definitely isn't needed to deploy most workloads like stateless or stateful web applications, but it can be extremely handy when you need to add particular business logic, or some new construct that Kubernetes doesn't support. The Custom Resource Definition (CRD) object and operator patterns allow you to create your own Kubernetes-style APIs.

OPEN SOURCE

Kubernetes is open source, and available on all major clouds as a managed offering. Despite the proliferation of many different platforms, distributions, and installers, most such offerings have been certified under the Cloud Native Computing Foundation's certification program⁶ which offers several guarantees around workload portability and compatibility. In fact, the only way for a product to include the name "Kubernetes" in it (like "Google Kubernetes Engine") is to have formally passed these tests.

You can also run Kubernetes yourself from scratch. And if you do run Kubernetes yourself, then the quality of the code will matter to you. Not all open source is created equal. While open source does typically remove you from propriety lock-in, you might end up having to maintain it yourself (you use it, you own it, basically) unless there is a strong community. The exception is for large, well maintained open source projects, in the caliber of say Linux, where so many people depend on it, and so many people use it that you can rest assured you won't need to take over maintenance. Fortunately, Kubernetes as the leading open source container orchestrator fits into this category.

⁶ <https://www.cncf.io/certification/software-conformance/>

NOTE While it is possible to host Kubernetes yourself whether on a public cloud or on a cluster of Raspberry Pi's, I don't recommend this for production use (i.e. outside of learning how to manage a cluster) in most cases. Spend the time doing what you do best: building great applications, and let someone else handle the minutiae of running Kubernetes for you.

Beyond the project itself being open source, Kubernetes is surrounded by a vibrant community. There are open source tools for accomplishing pretty much anything, so you typically have the option to go with a managed service, or deploy an open source tool yourself. This is a break from proprietary-only marketplaces in PaaS systems of the past, where your only option for any type of component was a paid one. Do you get value from a managed monitoring tool? Use a proprietary product. Want to just manage it yourself? Go install open source Prometheus.

Kubernetes has a large and growing number of practitioners as well, so whatever the topic is, you should be able to find help on Stack Overflow, or in books like this one. *zed Workflows*

CUSTOMIZED WORKFLOWS

Kubernetes is very unopinionated about how you setup your own, or your company's, development workflows. Want a "git push to deploy" style workflow? There are a bunch of ways to do that, some with only minimal setup. Typically you'll start with a bunch of CI/CD building blocks which you assemble into your desired workflow, from simple push-to-deploy, to complex pipelines with admissions control, auto-injecting secrets and security scanning. The downside is that it's not quite as ready to use out of the box as say a traditional PaaS, but this book will show you it's not that hard to get started.

Particularly for larger teams, the flexibility provided by Kubernetes in this area is often a huge advantage. Companies with a central core platforms teams will create opinionated pipelines for their application developer (app dev) teams to use. The pipeline can be used to ensure certain development practices around things like security, resource usage, and so on.

1.2.3 Kubernetes vs Platforms as a Service (PaaS)

Another approach to application deployment is to use a Platform as a Service (PaaS). A PaaS makes it easy to deploy your application code at scale by taking care of a lot of the packaging and deployment aspects for you. As long as your application fits within the scope of what the PaaS offers in terms of languages, dependencies, how it handles state, etc., you can launch each application into the PaaS, and not worry about the machines underneath.

However, what happens when you need to highly customize your dependencies, like using a specific version of Java? Can you host a stateful backend alongside your stateless frontends? And is it cost effective when you have many applications, each needing many replicas? At a certain point, the limitations of a PaaS can be prohibitive, and once you move out of the PaaS world, you have to start over from scratch – a daunting prospect.

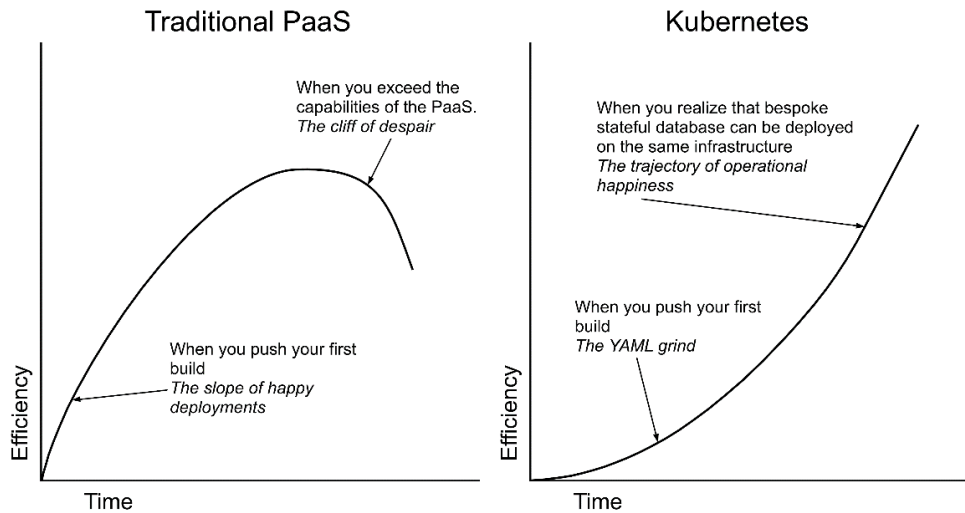


Figure 1.9 developer efficiency using a traditional PaaS and Kubernetes

A traditional PaaS is often fast to learn but slows down as you mature, and there's a potential cliff if you exceed the capabilities of the system and need to start from scratch. Kubernetes has a slower learning curve at the beginning, but expansive possibilities as you grow.

If you love your PaaS, then there may not be a need to move to Kubernetes. However, a common issue I've seen is that teams hit a certain level of complexity where their requirements exceed what the PaaS is capable of. One of the scariest things about being in that position is that you can't simply "break the glass" and assume more control yourselves. Often you'll need to re-architect the entire system, losing even the bits you were happy with, in order to build the new parts that you need. In this book, I'll show you how Kubernetes can run PaaS-type workloads with marginal added complexity over a dedicated PaaS, as well as how to run various other workload constructs like stateful workloads, background processing, and batch jobs which set you up for success in the future by enabling you to implement more complex product requirements.

A word on simplicity

I like to say: be wary of tools that make the simple easier, but the complex harder. Sure, it's nice when something helps you get up and running sooner, but is it leaving you in a good state, with the right knowledge and tools to get the job done? Kubernetes is easy enough to get started with, and powerful enough to serve your needs as you grow and expand. When choosing your platforms, prioritize making hard tasks possible over making simple tasks even easier.

Kubernetes will enable you to run a simple, 12-factor stateless application, migrate in a bespoke stateful custom application previously installed on a VM, or even run your own database. The abstraction layer doesn't limit what you can do, while still allowing you to get started using only the bits you need at first.

Some more modern PaaS support containers, so you could run there, and get the best of both worlds: the flexibility of containers, with the easy deployments. A downside of this is that even modern PaaS come with many restrictions on the types of workloads you can run. Can it, for example, run a stateful application with a block-based volume attached, as you might need if you have a legacy application to migrate, or you wish to run a bespoke database where no managed offering exists? I suggest carefully consider your current and future needs, and choose a platform that will be able to grow and scale as you do.

1.2.4 When (not) To Use Kubernetes

Like most tools, the goal of Kubernetes is to improve your efficiency, in this case managing your applications. It's best to ignore the hype and really consider whether Kubernetes will help or hinder your ability to run your service. Managed Kubernetes platforms exist to keep your cluster components running smoothly, but be aware that there is some overhead to running a general purpose platform like Kubernetes. Operational tasks include allocating CPU and memory resources to containers, updating deployments, configuring your network, and keeping everything up to date without disrupting your running services.

If you can predict the exact scope of your business requirements now and in the future and don't need the flexibility that Kubernetes provides, don't care about the vendor portability of the Kubernetes ecosystem, and can fit your application architecture neatly into the expectations of a more special-purpose platform, then go ahead and use it! Honestly, you'll probably have a simpler time of it.

I also don't recommend using Kubernetes for deployments of software where fully-managed equivalents exist. For example, why run a SQL database in Kubernetes when your cloud provider can do that for you? There are some exceptions where it becomes desirable to self-manage, but in general I believe if the managed service exists, use it!

Kubernetes *is* really good at a few things though, like running stateless applications at high density; mixing multiple workloads like a modern stateless application and legacy stateful monolith; migrating services from outdated systems to a unified platform; handling high-performance compute, like batch processing jobs for data analytics and machine learning; and of course running a bunch of microservices. In each of these cases, Kubernetes brings a lot to the table, by enabling high efficiency, unifying your hosting platform, automating your systems, and running your jobs.

Kubernetes does introduce a new level of management overhead which needs to be considered. There's a risk of simply replacing one problem with another if you take what you're doing (assuming it's working well) and throw it onto Kubernetes. Some cases where you may want to consider carefully would be: replacing a stateless system *if* it's already handling your scale and complexity; and moving standardized stateful workloads that have well-established deployment patterns like SQL databases. While you may see benefits in Kubernetes for such workloads, the advantages may not be as many, and so the trade-off needs to be more carefully considered.

To help decide, I suggest weighing up the benefits of moving to containers and unifying your compute platform around one deployment system suitable for varied workloads, with the added knowledge needed to administer Kubernetes. If what you're starting with is a bunch of services running on bespoke VMs in various stages of disrepair – it's likely not going to be a hard choice. Similarly, if you've outgrown your PaaS, or have a highly proficient team wanting to deploy faster with modern tools – go for it. But that MySQL cluster that's running like a charm on a custom clustering setup with 4 nines of reliability? Maybe that one's OK to leave for now.

Going to Kubernetes doesn't need to be an all or nothing decision. I'd suggest starting with those workloads that make the most sense and gradually migrate them as you and your team build up knowledge in operating Kubernetes.

1.3 Summary

- Containers are the modern way to run applications that achieves isolation between multiple applications running on the same host and does so with low overhead compared to virtual machines.
- Kubernetes is a deployment platform for containerized applications.
- Kubernetes has a bit of a learning curve, but it enables you to express a vast variety of deployment constructs and takes care of configuring infrastructure and keeping applications running.
- Hosted platforms (like Google Kubernetes Engine) take away the administrative burden of managing Kubernetes, allowing you to focus on your application deployments.
- Application developers focus on describing their app configuration in Kubernetes terms, after which the system is responsible for running it in the way you described.
- A key benefit of Kubernetes is that it allows you to grow as your needs evolve; you likely won't need to change platforms due to new requirements like an application needing to have its own local state.
- When you need to scale up due to increased demand, Kubernetes can help you do this in an efficient way.

2

Containerizing Apps

This chapter covers

- How to containerize apps
- Running your container locally
- Executing commands in the container context

Containerizing your application, that is, packaging your application and its dependencies into an executable container, is a required step before adopting Kubernetes. The good news is that containerizing your application has benefits beyond being able to deploy it into Kubernetes, it's a valuable step on its own right, as you're packaging up the application's dependencies and can then run it anywhere without needing to install those dependencies on the host machine.

Regardless of how you deploy your app, containerizing it means that your developers can start working on it locally using Docker, enabling them to get started on a new project with no setup beyond installing Docker. It provides easy context switching between the different applications developers are working on, as the environments are completely isolated. These properties make it a valuable way to improve developer productivity even if you don't end up deploying your app into production with containers (though you'll probably want to do that too).

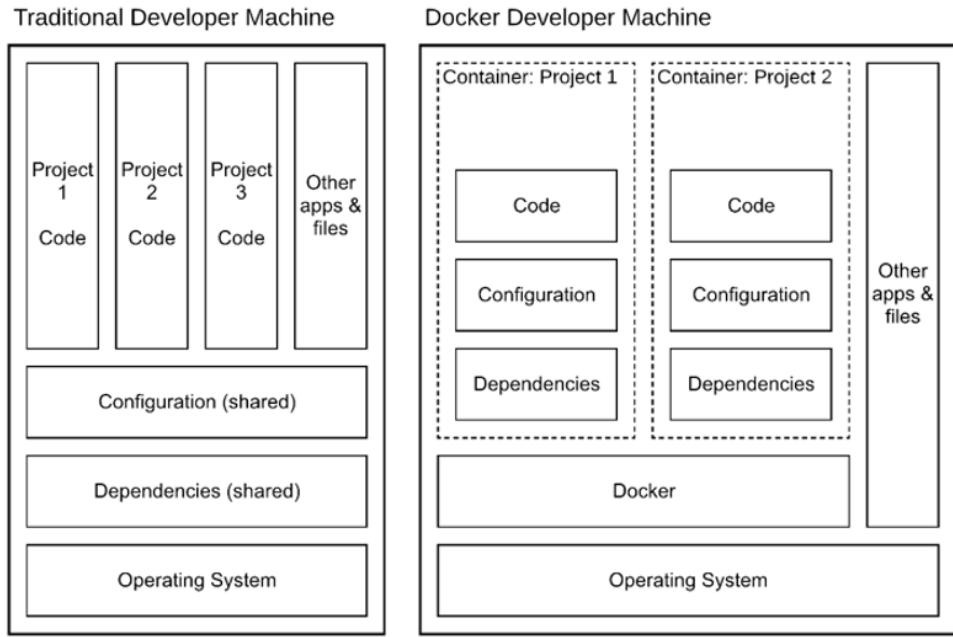


Figure 2.1 Comparison of multiple projects on a development machine with and without containerization

Having your application packaged into containers means that all your dependencies and configuration are captured by a container configuration file—the Dockerfile—rather than a mix of bash scripts, text based instructions, human memory and other non-standard configuration systems. It makes it possible to deploy multiple applications on a single host machine without worrying that they will interfere with each other, but with greater performance and less overhead than full virtualization.

2.1 Building Docker Containers

Let's take an application and put it in a container.

2.1.1 Developer Setup

Docker is distributed as a developer tool that is available for most platforms as Docker Desktop (<https://www.docker.com/products/docker-desktop>) which includes some convenient utilities including a local Kubernetes environment (covered in Chapter 3). For Linux (including Windows Subsystem for Linux) you can also install the Docker Engine standalone.

Mac

On Mac, simply install Docker Desktop.

WINDOWS

On Windows, I highly recommend first configuring the Windows Subsystem for Linux (WSL) (<https://learn.microsoft.com/en-us/windows/wsl/install>). WSL 2 is the one you want, so that Docker can use it as well. With WSL 2 installed, you can also install a distribution of Linux like Ubuntu (<https://apps.microsoft.com/store/detail/ubuntu/9PDXGNCFCSCZV>) which gives you a bash shell and is a convenient way to run the samples presented in this section.

Once WSL is configured, install Docker Desktop.

LINUX

For Linux, there is another option in addition to Docker Desktop, the Docker Engine. You can find instructions for various platforms, including Ubuntu, available here: <https://docs.docker.com/engine/install/ubuntu/>. Docker Engine is also an option when you use Linux via WSL.

2.1.2 Running Commands in Docker

To explore how Docker works before we build our own application container, we can bring up a containerized Linux shell in Docker like so:

```
$ docker run -it ubuntu bash
root@18e78382b32e:/#
```

What this does is download the base “ubuntu” image, start a container and run the bash command against it. The `-it` parameters make it an interactive bash terminal. Now we are in the container, and anything we run will happen in the container.

Since we’re going to be building an application on Ubuntu, let’s install the language package. I’m going to be using Python for many of the examples in this chapter, but the concept applies equally to any other language.

Run the following two commands in the container shell:

```
apt-get update
apt-get install -y python3
```

Now we can try out python interactively, for example:

```
# python3
>>> print("Hello Docker")
Hello Docker
>>> exit()
#
```

And we can capture that most basic of commands into our own python script:

```
# echo 'print("Hello Docker")' > hello.py
# python3 hello.py
Hello Docker
```

When you’re done playing around in this container, exit using `exit`.

The beauty of this, is that we installed Python and ran our Python command on the container, not on our local system.

The `docker run` command actually created a *container*, from our *image*. The image, `ubuntu`, is a prebuilt filesystem from which the container process runs in. When we exit our interactive session with the container, it will be stopped, but you can easily start it up again using `docker ps -a` to get the container id, `docker start CONTAINER_ID` to boot it, and `docker attach CONTAINER_ID` to reconnect our shell.

```
$ docker ps -a
CONTAINER ID   IMAGE      COMMAND                  CREATED          STATUS
c5e023cab033   ubuntu    "bash"                  5 minutes ago   Exited (0) 1 second ago

$ CONTAINER_NAME=c5e023cab033
$ docker start $CONTAINER_NAME
$ docker attach $CONTAINER_NAME
# echo "run more commands"
# exit
```

After running a lot of Docker containers, you'll end up with a pretty big list of stopped containers (and lots of hard drive space used). To clean up these images which typically you don't need to keep, at any time, run:

```
docker system prune -a
```

Container Image vs Container Instance

In Docker terminology, the container *image* is the file artifact (whether downloaded from a registry as in this section, or built locally), and the container instance (or just container) is an invocation of the container. In Kubernetes, configuration refers only to images, while container instances are created at runtime and are ephemeral in nature (they are deleted when the Pod is stopped). When using Docker locally, the instance concept is important, not least because every invocation creates a container instance that persists, so eventually you'll need to clean them up to recover the disk space.

With these steps, we now have a Linux environment on which we can use for testing and running random commands, all without needing to install anything (beyond Docker) on our local machine. Want two Linux container environments with a different config? No worries—just run another container!

If you've ever setup a VM before, you'll appreciate just how fast this is to setup! Containers are simple to create. As you'll see in the next section, they are also easy to build on, and expand.

2.1.3 Building our own Images

In the previous section, we started a Linux container, installed Python and created a simple python script which we ran in the container. Let's say we want to make this repeatable. That is, to capture the configuration of the container (installing Python), and our application (the python script) in our own container image. Such an image would be useful so we don't have to remember the steps we took, and also so that others can build our amazing application!

While this example uses only a simple python script, you can imagine that the application can be as large and complex as you want to make it. It doesn't just have to be Python

either—these steps work for any interpreted language (see section 2.1.7 for how to deal with compiled applications). Just substitute the Python configuration for whatever language you are using.

The process of building our container image so we can make a repeatable application deployment uses a configuration file known as a Dockerfile. The Dockerfile is a set of procedural instructions used to build your container. Think of it like a bash script that configures a virtual machine image with your app and its dependencies, only that the output is a container image.

Running the Examples

The example Docker applications and Kubernetes configuration listed in this book can be found in the source repository. Clone the repository and change into the root directory with the following.

```
git clone https://github.com/WilliamDenniss/kubernetes-for-developers.git
cd kubernetes-for-developers
```

The examples are arranged by folder, and section. For example, the code from Chapter 2 is in folder `Chapter02`, and the examples from section 2.1.3 are in a folder like `2.1.3_Docker`. Each code listing includes the path to the sample file so you can locate it.

The shell commands given start from the root sample code folder (`kubernetes-for-developers` if you cloned the repo per the above command), so just change back to that directory after running any example or exploring the code, and you should be ready to pick up where you left off, and follow the next example.

Starting with the basic Python program we created in the previous section:

Listing 2.1 Chapter02/2.1.3_Dockerfile/hello.py

```
print("Hello Docker")
```

To build our own container image setup with python and containing this script, you'll need to create a Dockerfile, pick a base container image to use as the starting point, and add your program. For now we'll start with the generic base image `ubuntu`, which provides a containerized Linux environment.

Here is a basic Dockerfile to capture our steps:

Listing 2.2 Chapter02/2.1.3_Dockerfile/Dockerfile

```
FROM ubuntu #A
RUN apt-get update #B
RUN apt-get install -y python3 #B
COPY . /app #C
WORKDIR /app #D
```

#A Specify base container image

#B Configure our environment

#C Copy the app to the container

#D Set the current working directory

Build this container, and name (tag) it “hello” like so:

```
cd Chapter02/2.1.3_Dockerfile/
docker build . -t hello
```

Once built we can run the `python3 hello.py` command on the container named “hello” like so:

```
$ docker run hello python3 hello.py
Hello Docker
```

Notice how the commands in our Dockerfile are essentially the same as the ones we used in the previous section. Rather than starting the `ubuntu` container image, we use it as our base image of the Dockerfile. Then we run the same two `apt-get` commands as before, to configure, install python, copy the our python script into the image, and specify the default working directory to indicate where commands will be run. Also notice that the command to run the code is still `python3 hello.py`, it’s just now prefixed to run in our new container image.

We’ve now encapsulated the environment we built, and our script, into a neat package that we can use and run ourselves and share with others. The wonderful thing about containers is that they encapsulate the configuration steps along with the program itself. The best part is that when the `ubuntu` base image is updated, we can rebuild our image by simply running that one build command again.

Compare this to installing Python on your host and running everything locally. For one, if you’d done that you would now have Python installed. You probably would be happy to have Python installed, but imagine a more complex application that brings with it dozens of tools and libraries. Do you really want all these on your system? Furthermore, what if you’re developing a few different applications, all with their own dependencies, or with the same dependency but requiring particular versions of that dependency, making it impossible to satisfy the dependencies of both applications simultaneously (a situation sometimes referred to as “dependency hell”).

Containers solve this, by isolating each application along with their dependencies in their own container images. You can happily work on multiple projects, share Dockerfiles with your development team, and upload container images to your production environment, all without messing up your developer machine.

To see how this looks for a different language, Ruby, the setup is fairly similar:

Listing 2.3 Chapter02-ruby/2.1.3_Dockerfile/hello.rb

```
puts "Hello Docker"
```

Listing 2.4 Chapter02-ruby/2.1.3_Dockerfile/Dockerfile

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y ruby
COPY . /app
WORKDIR /app
```

To run, the only difference is the command that is passed:

```
$ cd Chapter02-ruby/2.1.3_Dockerfile
$ docker build . -t hello_ruby
$ docker run hello_ruby ruby hello.rb
Hello Docker
```

2.1.4 Using Base Images

The above example uses the Linux container `ubuntu` as a base to configure our Linux-based app. Base images including `ubuntu` and other distributions such as `centos`, and the `alpine` base image are a good starting point for configuring any Linux-based app. However, for convenience the container community has created several more specific images designed for various languages and environments.

Instead of installing Python ourselves onto the `ubuntu` base image, we can just start with the `python` image, and save some steps. The added bonus is that these base images are generally created by experts, and thus are well configured to run Python apps.

Here's the same container, but starting with the `python` base image:

Listing 2.5 Chapter02/2.1.4_BaseImage/Dockerfile

```
FROM python:3
COPY . /app
WORKDIR /app
Simpler, right? Building and running it is the same as before:
$ cd Chapter02/2.1.4_BaseImage
$ docker build . -t hello2
$ docker run hello2 python3 hello.py
Hello Docker
```

What is a base image really?

The base image used in this example, `python`, is itself built with a `Dockerfile` and configures an environment with everything needed to run Python programs. For container images from Docker Hub, their `Dockerfile` sources are linked so you can see how they are composed. Base images often start with another base image, and so on, until one which starts with the completely empty container known as `scratch`.

If you're using Ruby instead of Python, setup is pretty similar, just using the `ruby` base image.

Listing 2.6 Chapter02-ruby/2.1.4_BaseImage/hello.rb

```
puts "Hello Docker"
```

Listing 2.7 Chapter02-ruby/2.1.4_BaseImage/Dockerfile

```
FROM ruby
COPY . /app
WORKDIR /app
To build and run:
$ cd Chapter02-ruby/2.1.4_BaseImage
$ docker build . -t hello_ruby2
$ docker run hello_ruby2 ruby hello.rb
Hello Docker
```

There exists more than operation system, and language-specific base images too. If you're using an environment like Apache, you can start with the `httpd` base image. Sometimes you'll have a situation where there are multiple base images that could serve as the base. The best rule of thumb is to pick the one that saves you the most configuration (and you can always crib from the `Dockerfile` of the one you didn't pick!).

Base images—or at least public examples that you can copy—exist for pretty much every common language, environment or open-source application. Before building your own from scratch, it is wise to search Docker Hub for a base image, or Google to see if someone has an example for your environment that you can use as a starting point.

2.1.5 Adding a Default Command

Typically, the command executed in the container (`python3 hello.py` in the earlier Python example) is the same each time. Rather than repeating it each time, you can specify that in the `Dockerfile` as well:

Listing 2.8 Chapter02/2.1.5_DefaultCommand/Dockerfile

```
FROM python:3
COPY . /app
WORKDIR /app
CMD python3 hello.py
To build and run this container, execute the following from the command line:
$ cd Chapter02/2.1.5_DefaultCommand
$ docker build . -t hello3
$ docker run hello3
Hello Docker
```

Unlike the other lines in the `Dockerfile` we've used so far, `CMD` is unique as it doesn't actually change how the container is built, but merely saves the default command that will be executed if you call `docker run` without a command specified. This doesn't stop you from overriding it and executing a different command at runtime.

With the command now specified in the `Dockerfile`, to build and run the Ruby version of this program it is also simply `docker run`.

```
$ cd Chapter02-ruby/2.1.5_DefaultCommand
$ docker build . -t hello_ruby3
$ docker run hello_ruby3
Hello Docker
```

2.1.6 Adding Dependencies

Most non-trivial applications will have their own dependencies, not included in the base image. To load those dependencies, you can run commands during the container build process to configure the image as you need. This was how we added Python to the Linux base image in the example above, and this method can be used to install all the dependencies that your application needs.

If your application establishes a database connection to a MariaDB database, here's how you might build your container:

Listing 2.9 Chapter02/2.1.6_Dependencies/Dockerfile

```
FROM python:3
RUN apt-get update #A
RUN apt-get install -y mariadb-client #A
COPY . /app
WORKDIR /app
CMD python3 hello.py
```

#A Configure your linux container with everything your app needs

The `python` base image is built from Debian, a distribution of Linux widely used for Containers, which uses the `apt-get` package manager, so we can use `apt-get` to install pretty much any other dependency we need.

You don't just have to use `apt-get` either. Say you have a service that's creating PDF files, and you need to include a Unicode font, you can build an image that includes Google's "Noto" free font like so:

Listing 2.10 Chapter02/2.1.6_Dependencies-2/Dockerfile

```
FROM python:3
RUN apt-get update
RUN apt-get install -y bsdtar #A
RUN mkdir -p ~/.fonts; cd ~/.fonts #B
RUN curl "https://noto-website-2.storage.googleapis.com/pkgs/Noto-hinted.zip" | bsdtar -
  xvf- #C
RUN fc-cache -f -v #D
COPY . /app
WORKDIR /app
CMD python3 hello.py
```

#A Install `bsdtar`

#B Create a new directory and `cd` into it. Notice how multiple commands can be combined on one line.

#C Download the font package and extract it

#D Install the fonts

It is common for containers to have many dependencies, and you can configure any part of the operating system you need to in this way, such as installing fonts, or TLS certificates.

2.1.7 Compiling Code in Docker

What about programs that need compilation, like Java, .NET, Swift, or C++? Obviously a `COPY` command will not suffice in the Dockerfile, unless you already have compiled binaries lying around.

Pre-compiling the application locally would be one option, but why not leverage Docker to compile your application as well! Let's re-implement our hello world example in Java, and compile it into our container.

Listing 2.11 Chapter02/2.1.7_CompiledCode/Hello.java

```
class Hello {
    public static void main(String[] args) {
        System.out.println("Hello Docker");
    }
}
```

Listing 2.12 Chapter02/2.1.7_CompiledCode/Dockerfile

```
FROM openjdk
COPY . /app
WORKDIR /app
RUN javac Hello.java #A
CMD java Hello
```

#A The compile command

The Dockerfile is similar to the previous ones: we start with the OpenJDK base image, and copy the app. In this case, however, we'll use the `RUN` command to build the app, prefaced with a `WORKDIR` directive to specify where this action (and subsequent actions) should be performed.

```
To build and run this example:
$ cd Chapter02/2.1.7_CompiledCode
$ docker build . -t compiled_code
$ docker run compiled_code
Hello Docker
```

Another example that compiles a server-side Swift application is given in the `Chapter02-swift/2.1.7_CompiledCode` folder. It can be built and run in the same way.

2.1.8 Compiling Code with a Multi-stage Build

Using `RUN` to compile code or perform other actions is a viable path; however, the drawback is that you end up configuring your container image with tools it needs to execute the `RUN` command. These tools end up in the final container image along with any source code.

For example, if you look at the image we created in the previous section, and run `ls`:

```
$ docker run compiled_code ls
Dockerfile
Hello.class
Hello.java
```

You'll see that the source code remains. Also, the Java Compiler (javac) is still present in the image, even though it will never be used again (we don't need the compiler when running our application).

This mixing of responsibilities of the container image—to both build and run—is less than ideal. Not only do all those extra binaries bloat the container image, but they also needlessly increase the attack surface area of the container (as any process running in the container now has a compiler to work with). You *could* clean up the container with a bunch of additional Docker commands (e.g., deleting the source code, uninstalling tools that are no longer needed) but it's not always practical, particularly if all these extra tools came from the base image.

A better way to solve this problem is to use a multi-stage container build. With a multi-stage build, we first configure a temporary container with everything needed to *build* the program, and then a final container configured with everything needed to *run* the program. This keeps the concerns separated and neatly isolated to their own containers.

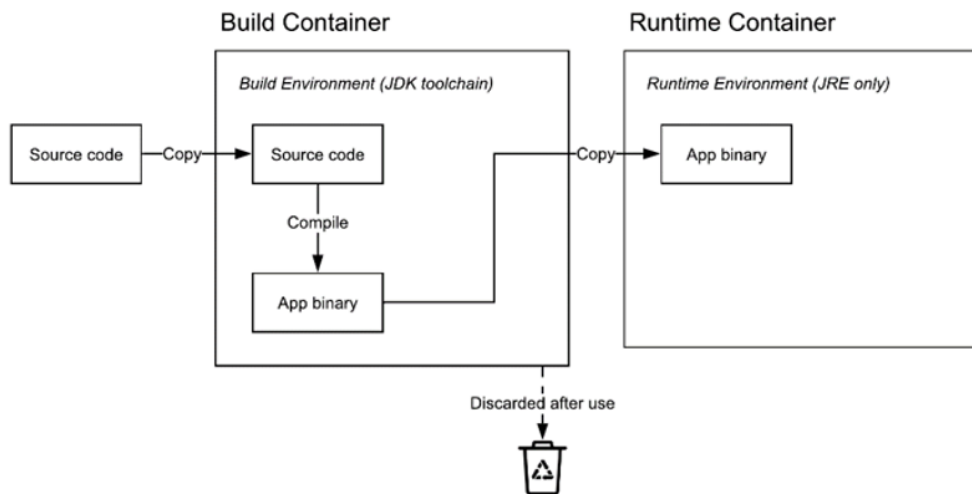


Figure 2.2 A multi-stage container build, where an intermediate container is used to build the binary

Let's rework the example in the previous section to be built using a multi-stage Dockerfile.

Listing 2.13 Chapter02/2.1.8_MultiStage/Dockerfile

```
FROM openjdk:11 AS buildstage #A
COPY . /app
WORKDIR /app
RUN javac Hello.java

FROM openjdk:11-jre-slim #B
COPY --from=buildstage /app/Hello.class /app/ #C
WORKDIR /app
CMD java Hello
```

#A The “build” container is named ‘buildstage’ and has the responsibility to build the code
 #B The runtime container uses a slimmed down base image, without the compile tools
 #C “--from=” is used to reference files from the build container.

As you can see from this example, there are what looks like two `Dockerfiles` in one (each beginning with a `FROM` command). The first is configured and built purely to compile the app, using the full OpenJDK base image which includes the java compiler, and the second has only what is needed to run the app and is built from the JRE base image, which only includes the Java runtime environment.

This `Dockerfile` produces as its final artifact, a production container that only contains the compiled Java class and dependencies needed to run it. The intermediate artifact of the first container that built the app is effectively discarded after the build completes (technically it’s saved in your docker cache, but no part is included in the final artifact that you would use in production).

```
To run this example:
$ cd Chapter02/2.1.8_MultiStage
$ docker build . -t compiled_code2
$ docker run compiled_code2
Hello Docker
```

Another example that compiles a server-side Swift application with the multi-stage build process is given in the `Chapter02-swift/2.1.8_MultiStage` folder. It can be built and run in the same way.

2.2 Containerizing a Server Application

The examples in the previous section were all simple programs that run once. This is a use-case for containers: for command line programs, batch workloads or even to serve requests in a Functions as a Service environment. One of the most common workloads to deploy in Kubernetes, however, is HTTP services—that is, an application that listens for, and processes, incoming requests, a.k.a. a web server.

A server application is no different to any other application from Docker’s perspective. There are a few differences to how you start and connect to the container owing to the fact that you likely want to keep the container running (so it can serve requests). You’ll also likely want to forward ports from your local machine so you can connect to it.

2.2.1 Containerizing an Application Server

Until now, the example program was a basic “Hello World” Python script. To demonstrate how to containerize HTTP servers, we’ll need something that is an HTTP server! The following code is an example of a bare-bones HTTP server in Python that returns the current date and time. Don’t worry too much about the code itself. This book is language agnostic, and the Python used here is purely an example. You can apply these principles to any HTTP server.

Listing 2.14 Chapter02/timeserver/server.py

```

from http.server import ThreadingHTTPServer, BaseHTTPRequestHandler
from datetime import datetime

class RequestHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/plain')
        self.end_headers()
        now = datetime.now()
        response_string = now.strftime("The time is %-I:%M %p, UTC.")
        self.wfile.write(bytes(response_string, "utf-8"))

def startServer():
    try:
        server = ThreadingHTTPServer(('', 80), RequestHandler)
        print("Listening on " + ":".join(map(str, server.server_address)))
        server.serve_forever()
    except KeyboardInterrupt:
        server.shutdown()

if __name__ == "__main__":
    startServer()

```

Containerizing this server application is very similar to the earlier command line program.

Listing 2.15 Chapter02/timeserver/Dockerfile

```

FROM python:3.10
ENV PYTHONUNBUFFERED 1
COPY . /app
WORKDIR /app
CMD python3 server.py

```

Containerizing your own application

If you're containerizing your own application, follow these generic steps:

Find an ideal base image, being one that provides as much of your configuration as possible. For a Ruby on Rails app, start with `ruby` and not the more generic `ubuntu`. For Django, use `python`, and so on.

Configure any application specific dependencies you need (via `RUN` statements, as we did above)

Copy your application

I find that Google is really your friend for this. Unless you're doing something new and exotic, someone's probably figured out and shared an example Dockerfile of how to configure an application using your framework. If you're using a popular framework like Django, Ruby on Rails, WordPress, Node.JS, or SpringBoot I can say with certainty that there are a lot of resources for you to draw on. Every application is different—your dependencies won't exactly match everyone else's all the time—but you can get a huge head start this way.

Now that we have our HTTP server application, we can build it like usual:

```

$ cd Chapter02/timeserver
$ docker build . -t timeserver

```

Running it is a little different this time, since we'll need to forward ports from the host machine to the container, so we can actually try this application in the browser. Let's forward port 8080 on our local machine to port 80 in the container that the application is listening on.

```
$ docker run -it -p 8080:80 timeserver
Listening on 0.0.0.0:80
```

Now you should be able to browse to <http://localhost:8080> and view the application. Or, with curl:

```
$ curl http://localhost:8080
The time is 1:30 PM, UTC.
```

The `-it` parameter (actually two parameters but normally used together) allows us to terminate by sending SIGTERM (often Ctrl/Command+C). This makes the typical developer loop of build, run, fix, repeat easy (run, Ctrl+C, fix, repeat). Alternatively you can run Docker in the background with `docker run -d -p 8080:80 timeserver`. Without using `-it`, you'll need to stop the process manually: `docker ps` to list the process, and `docker stop CONTAINER_ID` to stop it (or `docker stop $(docker ps -q)` to stop all running containers).

For a neat development loop, I like to use the following one-liner that will build and run the image in one go. When you need to rebuild, you can just hit Ctrl+C (or equivalent), press up to show the last used command, and enter, to do it all again. Just be sure to watch the console output for any errors during the build stage, as otherwise it will run the last built image.

```
$ docker build . -t timeserver; docker run -it -p 8080:80 timeserver
```

That's it! We now have a containerized application running in Docker. In the next section, 2.3, I cover how use Docker Compose to configure and run a local debug setup (useful if your application consists of a few different containers), and in the next chapter, how to deploy this web application into Kubernetes.

2.2.2 Debugging

If you're having trouble getting your app to run after configuring a Dockerfile, it can be useful to shell into in the container's environment to poke around and see what's going wrong. When the container is running using the previous instructions, you can shell into the running container from a new console window like so:

```
$ docker container ls
$ docker exec -it CONTAINER_ID sh
# ls
Dockerfile  server.py
# exit
$
```

You can run any other command other than `sh` too, for example on a Ruby on Rails project you might run `bundle exec rails console` here to directly bring up the rails console without an intermediate step.

I won't list out every docker command, as the docs do a great job at that, but another one I find especially useful for debugging is `docker cp`. It allows you to copy files between your host and the container. Here's an example:

```
$ docker cp server.py CONTAINER_ID:/app
Or to copy a file out of the container:
$ docker cp CONTAINER_ID:/app/server.py .
```

If you do fix anything through running commands via `exec`, or copying files, be sure to capture the change in the Dockerfile. The Dockerfile is your primary specification, not the container instance. If you rely on manual changes to the container instance, it's no better than the old "shell into a VM and change things" model that we're moving away from.

2.3 Using Docker Compose for Local Testing

At this point, we have built a container image and are ready to start using Kubernetes. If you like, skip ahead to the next chapter, and deploy this newly built container into Kubernetes right away, to the cloud or using a local test Kubernetes environment. This section covers how you can use Docker Compose for local container testing and development before deploying to Kubernetes.

In the previous section we booted our server application using docker, and forwarded ports to our host for testing. Using this approach for testing during development has a couple of drawbacks. You have to setup the ports to forward each time, and if you're developing an application with a few containers, it can be complex to get everything up and running, with the right ports forwarded, etc.

This is where Docker Compose comes in. Compose is a mini container orchestrator that can bring up and tear down multiple containers in a logical group, and preserve the runtime settings in between runs, which is useful for local testing.

To run the web server container from section 2.2.1 with compose, we can configure a `docker-compose.yml` file such as the following.

Listing 2.16 Chapter02/2.3_Compose/docker-compose.yml

```
services:
  web:
    build: ../timeserver #A
    command: python3 server.py #B
    ports: #C
      - "8080:80" #C
```

#A The path to the directory containing the docker container to be built

#B Command that will be run on the container. Can be skipped if your Dockerfile specifies `CMD`

#C Ports to forward to the container from the local machine. In this case, port 8080 on the local machine will be forwarded to the container's port 80.

To build and run the container:

```
cd Chapter02/2.3_Compose
docker-compose build
docker-compose up
```

When developing, I tend to run both these steps as one so I can create a tight rebuild loop.

```
docker-compose build; docker-compose up
```

With this simple configuration, there's no need to remember the specific docker command to boot and test the application—everything is stored neatly in the compose file. With this example, that mostly consists of some ports to forward, but this benefit will become apparent as you add more configuration and dependencies.

2.3.1 Mapping Folders Locally

Earlier we used `docker cp` to copy files into and out of container instances. One really useful feature of Compose is that you can actually map local folders right into the container. In other words, instead of the container having a copy of your application, it will actually just link to the same folder on your hard drive. During development this can be really handy, as it allows you to work on the files in the container right from your desktop, without needing to copy things back and forth, or rebuild the container.

Recall from the Dockerfile that our server app is copied into the `/app` directory within the container. What we want to do now is mount our local directory into the container at that same directory. This is done through volume binding.

Listing 2.17 Chapter02/2.3.1_VolumeMount/docker-compose.yaml

```
services:
  frontend:
    build: .
    command: python3 server.py
    volumes: #A
      - type: bind #A
        source: . #A
        target: /app #A
    environment:
      PYTHONDONTWRITEBYTECODE: 1 #B
    ports:
      - "8080:80"
```

#A Bind the local container build directory into the container's `/app` directory (matching the Dockerfile)

#B Set a new environment variable so that Python can be made to reload our source

With this volume binding, the files from our local machine are used instead of the ones copied when we built our container. When we update those files locally, the changes can be immediately read in the container without a rebuild. For interpreted languages like Python, Ruby and PHP, as well as markup like HTML and CSS, this means we can potentially have a setup where you just hit save in your editor, and reload the page in the browser for a really tight development loop where saving the source code triggers a code rebuild.

There's a catch, however. Once we run our python code, by default it isn't reloaded from the disk when there are changes, so while we can modify the source code it won't have any effect once the container is running. This will be true for many other build systems as well.

For compiled code, this may not be of much help. You could build the binary locally, replacing it in the container, but if you prefer to build everything through docker (or there is an architectural difference between your local environment and the container) then this won't

help. For compiled code I recommend using other developer tools like Skaffold¹ to give you a tight development loop.

Let's update the Python timeserver app to support reloading the code while it's running, and configure a local mount in Compose. The steps here will vary by language and framework. For Python we can use the reloading library to have our GET function reloaded from disk each time there is a new request.

Listing 2.18 Chapter02/2.3.1_VolumeMount/server.py

```
from reloading import reloading
from http.server import ThreadingHTTPServer, BaseHTTPRequestHandler
from datetime import datetime

class RequestHandler(BaseHTTPRequestHandler):
    @reloading #A
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/plain')
        self.end_headers()
        now = datetime.now()
        response_string = now.strftime("The time is %-I:%M %p, UTC.")
        self.wfile.write(bytes(response_string, "utf-8"))

def startServer():
    try:
        server = ThreadingHTTPServer(('', 80), RequestHandler)
        print("Listening on " + ":".join(map(str, server.server_address)))
        server.serve_forever()
    except KeyboardInterrupt:
        server.shutdown()

if __name__ == "__main__":
    startServer()
```

#A By adding the @reloading tag to our method, it will be reloaded from disk every time it runs so we can change our do_GET function while its running

Since we're using a new library, we'll need to add that dependency in the Dockerfile as well:

Listing 2.19 Chapter02/2.3.1_VolumeMount/Dockerfile

```
FROM python:3
RUN pip install reloading
ENV PYTHONUNBUFFERED 1
COPY . /app
WORKDIR /app
CMD python3 server.py
```

With our app configured to reload files from the disk, we can now run it with Compose as before:

¹ <https://skaffold.dev/>

```
$ cd Chapter02/2.3.1_VolumeMount
$ docker compose build; docker-compose up
Creating network "231_volumemount_default" with the default driver
Creating 231_volumemount_frontend_1 ... done
Attaching to 231_volumemount_frontend_1
```

As before, browse to the app at <http://localhost:8080/>. This time, open up the `2.3.1_VolumeMount/server.py` code, and make a change to the response. For example, we can change the response to 24 hour time by updating this line:

```
response_string = now.strftime("The time is %H:%M, UTC.")
```

Save the file in your editor, and reload the page. You should see the new text in the response! That's how you get a local development loop. In this example we had to make some code changes to make it work, but if you're using a standard development framework that will likely not be necessary, as you'll be able to configure it to perform reloads automatically.

```
$ curl http://localhost:8080
The time is 10:23, UTC.
```

Being able to map local folders into the container to create a development loop that is as fast as hitting save in your code editor, then reloading a page on the browser has to be one of my favorite features of Docker Compose, and containers in general. You have all the benefits of containers, where you don't need to mess around installing the developer tools locally, with the same efficiency as if you were running it locally without any build step.

The binding works both ways too. If you make any changes in the container within the bound volume, it will be reflected on your local disk. This can be useful when you want to run commands in the container and save their output. In fact, with this approach, you can completely avoid having the developer tools installed on your local machine at all. For example, a Rails developer will, from time to time, run `gem update rails` in their project directory to keep the framework up to date. With a volume binding you can run that in the container, and get the changed package list on your hard drive ready to commit to version control.

2.3.2 Adding Service Dependencies

In the case where your app is completely standalone, congratulations, you are done. The rest of the time, though, you'll likely have other services that you need to bring up alongside your application. These might be other separate servers that you build, or standard components that might be run by your cloud provider, like a database. In both cases, you can add these dependencies in Compose, to create a local development environment.

Compose or Kubernetes For Local Development?

Why use or consider Compose rather than Kubernetes itself to bring up dependent services for development?

Kubernetes certainly can be used for local development, and if you want to replicate your production environment it's the best option (Chapter 3 includes a section on local development). What makes Compose popular for this task, however, is its simplicity.

Compose is easy to setup for local use if you just need a handful of dependent services, which for many simple applications is the case. In production, where you're not just running a few single instance services on one machine, is a different (more complex) story, which is where Kubernetes comes in.

This duality means it's not uncommon to see Compose for local dev, and Kubernetes in production. It does mean that your runtime configuration is essentially duplicated, but this configuration has two separate purposes: development, and production, so it likely won't look identical even if it was all in Kubernetes. I suggest simply using whichever approach makes life as a developer easier.

Multiple services can easily be added to the Compose. These can reference standard images (in the case of a dependency like MySQL), or other projects on your computer. A common project structure is to have one root folder for all services, with each checked out in a sub-folder, and a docker-compose file that can reference them all.

Here's an example of a compose file with two containerized services: our app that's built locally, and a database instance using the public MySQL container. The demo app here doesn't actually use MySQL, but hopefully you can see how easy it is to add the dependencies that your app needs. You can add all the services you need here, including multiple locally built containers, and multiple external images.

Listing 2.20 Chapter02/2.3.2_MultipleServices/docker-compose.yml

```
services:
  frontend: #A
    build: ../timeserver #A
    command: python3 server.py
    environment: #A
      PYTHONDONTWRITEBYTECODE: 1 #A
    ports: #A
      - "8080:80" #A

  db:
    image: mysql:5.7 #B
    volumes: #B
      - db_data:/var/lib/mysql #B
    restart: always #B
    environment: #B
      MYSQL_ROOT_PASSWORD: super secret password #B
      MYSQL_DATABASE: my_database #B
      MYSQL_USER: dev_user #B
      MYSQL_PASSWORD: another secret password #B
volumes: #C
  db_data: #C
```

#A Our app that is built locally; you can have multiple locally built apps

#B A service running a public image; you can have multiple such services as well.

#C Volume definition for the development database so that it will persist between restarts

This is one of the key reasons to use Compose, rather than just Docker for local testing—the ability to bring up a complete testing environment, and tear it down all with a single command.

When configuring your application for local development, and for production, all configuration changes should be made by environment variables. Even a single environment variable that indicates “prod” or “dev” to select which configuration file to use can suffice. Configuration should not be baked into the container in such a way that you need to modify it between environments. This allows you to reuse the same container in all environments, and also means that you are testing the production artifact.

2.3.3 Faking External Dependencies

If to date you’ve been testing against remote dependencies (like a cloud storage API), now might be a good time to see if you can replace those remote dependencies with fakes. Fakes are lightweight implementations of the same API of the external dependency, which speed up development and testing by providing a local service.

In the past, you might have been constrained to finding a fake written in the same language as your application (for practical reasons, like not wanting to support multiple different environments for one project). One of the benefits of containers is that just like how you probably don’t care what language a cloud service you consume is written in, you no longer need to care about what language your fake is written in either as you don’t need to maintain the environment—it runs in its own container.

This also brings the opportunity for high-quality fakes that are really just lighter-weight implementations of the same API you’ll use in production. Just as in the earlier section we used real MySQL in a container (rather than a fake), you can use a real object storage provider to test against, even if you ultimately use a cloud provider service like Google Cloud Storage or AWS S3.

Taking the object storage example, say your application does cloud storage using S3-compatible APIs (e.g. with S3 itself, or one of the many object stores that support the API, like Google Cloud Storage). To setup a local fake for fast iteration, you could get a container like Adobe’s S3Mock², but with containers it’s equally easy to use a fully-fledged S3-compatible local storage solution like MinIO³. MinIO is not really a fake—you can deploy it into production for cases when you want to manage your own block storage service—but you can still use it as a high-quality fake and get benefits like a convenient UI.

The ubiquity of the S3 API for object storage

Like SQL standardized database query languages, S3’s API is surprisingly popular for object storage providers. For example, Google, Azure, and (of course) AWS all implement the S3 API, along with most other clouds and several bare metal storage options as well. The benefit of this ubiquity is you can easily switch between providers, and have several fakes to choose from to develop locally.

² <https://github.com/adobe/S3Mock>

³ <https://min.io/>

Earlier I discussed how containers make it easy to mix and match services all running in their own environments. Here we see how this ability can make development better as well. Rather than implementing a fake yourself, or finding a rudimentary fake for your environment, you can either use locally the same service as in production (such as with MySQL), or find a production-grade replacement for another cloud service that you use (like with MinIO subbing in for cloud object storage).

Let's add MinIO as another service to our Docker Compose file:

Listing 2.21 Chapter02/2.3.3_Fakes/docker-compose.yaml

```
services:
  storage: #A
    image: minio/minio
    command: minio server /data
    volumes:
      - storage_data:/data
    restart: always
    environment:
      MINIO_ACCESS_KEY: fakeaccesskey
      MINIO_SECRET_KEY: fakesecretkey
    ports:
      - "9000:9000"

  frontend:
    build: ../timeserver
    command: python3 server.py
    environment:
      PYTHONDONTWRITEBYTECODE: 1
      S3_ENDPOINT: http://storage:9000 #B
      S3_USE_PATH_STYLE: 1 #B
      S3_ACCESS_KEY_ID: fakeaccesskey #B
      S3_SECRET_ACCESS_KEY: fakesecretkey #B
    ports:
      - "8080:80"

volumes:
  db_data:
  storage_data:
```

#A The new storage mock

#B The application configured to use the new mock

Typically with services used as fakes, as we are using MinIO here, you can specify the access keys that it will use, then simply specify those same secrets to the application using environment variables.

2.4 Summary

- Containerization is essentially scripting your app's build, environment and configuration into a standardized format that can then be run with VM-like properties on a host, but without the VM overhead.
- Containerization is key step towards adopting Kubernetes, as this is the executable environment that Kubernetes supports.
- Not just for production, containers help developers work on multiple projects at once, without the environments conflicting and without needing complex setup instructions.
- The process of building our container image so we can make a repeatable application deployment uses a configuration file known as a Dockerfile, which contains a set of procedural instructions used to build your container.
- When building a Dockerfile, begin with the base image whose configuration is the most complete for your needs, then add the necessary default commands and dependencies.
- Multi-stage builds are useful for applications that require compilation before they are run.
- With a multi-stage container build, we first configure a temporary container with everything needed to *build* the program, and then a final container configured with everything needed to *run* the program. This keeps the concerns separated and neatly isolated to their own containers.
- Docker Compose is a lightweight container orchestrator that can give you a quick container-based development environment for multiple services.
- Mapping folders locally with Compose enables the editing of non-compiled applications in real time, for a tight development loop.
- During testing, containers bring the opportunity for high-quality fakes of external dependencies that are really just lighter-weight implementations of the same API you'll use in production, such as MySQL in a container or a real object storage provider.

3

Deploying to Kubernetes

This chapter covers

- **Kubernetes concepts related to specifying and hosting application deployments**
- **Deploying a containerized application to Kubernetes on a Cloud platform, with load balancing**
- **Updating deployments with new versions of the application container**
- **Running a version of Kubernetes locally for testing and development**

In the previous chapter, we covered how to containerize your application. If you stopped there, you would have a portable, reproducible environment for your app, not to mention a convenient developer setup, but you may have trouble scaling that app when you go to production.

For ultra-simple deployments where you don't mind running one container per VM, you might be able to deploy containers to VMs directly, then scale your VMs as needed. You'd get a few of the advantages of containers, like convenient packaging, but if like most you have a number of different services to deploy, you'll probably need something more flexible.

This is where a container orchestrator like Kubernetes comes in. Container orchestration is just a fancy way of saying tooling that handles the scheduling and monitoring of a bunch of different containers on a bunch of different machines. It allows you to work primarily in terms of your application deployment—the container, and its deployment attributes like how many replicas (instances) of the container there should be, requirements like high availability (spreading across failure domains), service networking, and so on—rather than needing to be overly concerned with the configuration of the underlying compute.

Being able to conveniently manage multiple services on a shared pool of compute resources gives you efficiency when running multiple applications, or adopting patterns like micro-services where the various parts of your application are deployed and managed separately. You can mix different types of deployments too, from a stateless application, to a

stateful database, batch jobs, and more—all without needing to worry too much about exactly which machine each container ends up actually running on.

3.1 Kubernetes Architecture

Kubernetes is an abstraction layer that sits at the workload level, on top of the raw compute primitives like VMs (or bare metal machines) and load balancers. VMs are referred to as “nodes”, arranged into a “cluster”. Containers (one or multiple) are grouped into a scheduling unit known as a “pod”. Networking is configured via “services”. Other higher order building blocks like “Deployments” exist to make pods even easier to manage. Let’s explore some of the basic building blocks of this architecture before deploying our first workload.

3.1.1 The Kubernetes Cluster

The Kubernetes Cluster is a collection of nodes, which are the compute instances on which the containers are run. Most commonly these are virtual machines, but can also be bare metal (non-virtualized) machines. Each of these nodes runs a special Kubernetes process called the “kubelet,” which is responsible for communicating with the control plane and managing the lifecycle of the containers which run on the nodes, and a container runtime responsible for loading and running the containers. Other than the operating system, the kubelet and the container runtime environment, the remaining processes, including your own workloads as well as some system components responsible for logging and monitoring, are run in containers, as shown in Figure 3.1.

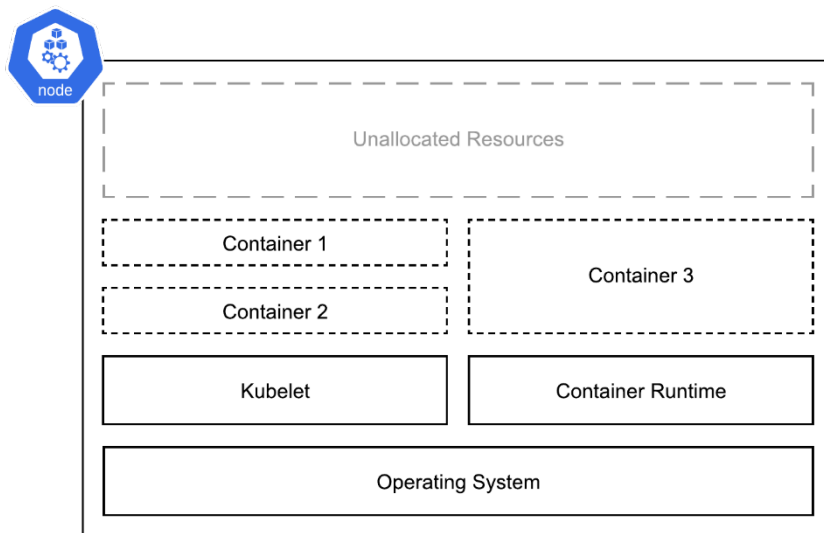


Figure 3.1 Processes running on a virtual machine, which Kubernetes calls a “node”

In the cluster, one or (when operating in high availability mode) multiples of these nodes have a special role (as shown in shown in Figure 3.2): running the Kubernetes orchestrator program itself. This special type of node, called the control plane, has the responsibility to:

- Run the API which you use to interact with the cluster (using tools such as the Kubernetes CLI tool)
- Store the state of the cluster
- Coordinate with all the nodes in the cluster to schedule (start, stop, restart) containers on them.

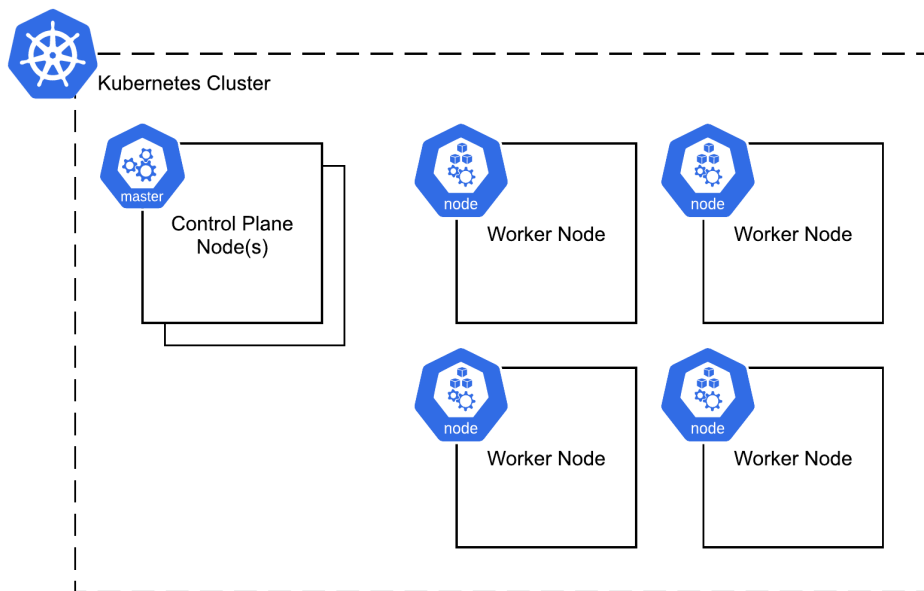


Figure 3.2 Self-managed Kubernetes cluster with the control-plane and worker nodes

In most cloud environments, the control plane is offered as a managed service. In such environments the control plane nodes are typically not visible to the user and the fact that the control plane may run on a node is implementation detail. In these environments, you'll typically think of the cluster as the managed control plane with worker nodes, as shown in Figure 3.3.

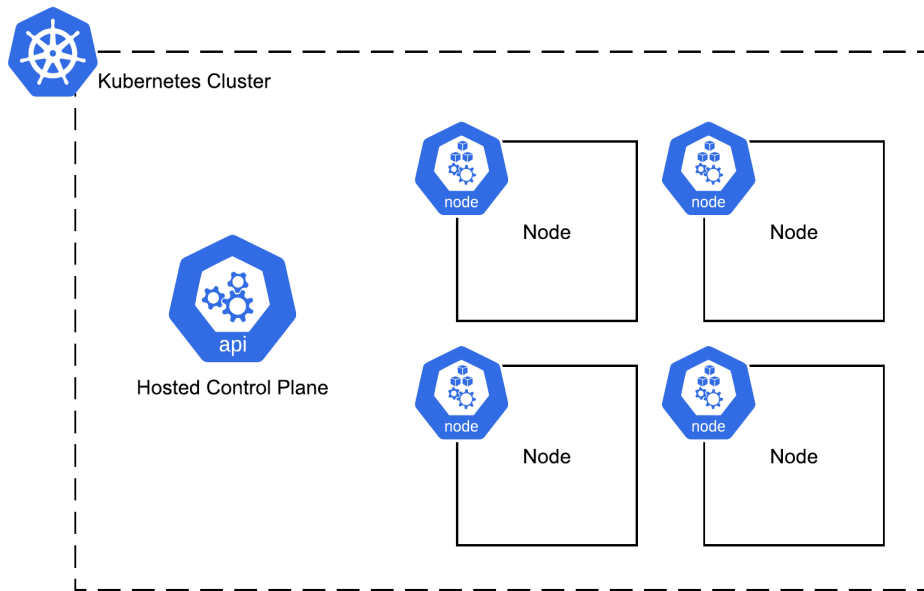


Figure 3.3 Cloud-hosted Kubernetes Cluster with nodes connecting to a hosted control plane.

Worker nodes (herein referred to simply as “nodes”) have the responsibility for managing the lifecycle of containers that run, including tasks such as starting and stopping containers. The control plane will instruct the node to run a certain container, but the actual execution of the container is then the responsibility of the node. The nodes also take some actions by themselves without needing to check in with the control plane, like restarting a container that has crashed, or reclaiming memory when the node is running low.

Collectively, the control plane and nodes form the “Kubernetes Cluster”, and provide the Kubernetes platform on which you can schedule your workloads. The cluster itself is provisioned and managed by whatever platform provider you use to run Kubernetes, which is responsible for creating the cluster resources like nodes. This book, aimed at developers, focuses primarily on *using* the Kubernetes cluster to run your workloads, rather than the platform provider tasks (which are more in the cloud provider domain) of offering this service to developers.

3.1.2 Kubernetes Objects

Once the cluster is created, you interact with Kubernetes primarily by creating, inspecting and modifying Kubernetes objects through the Kubernetes API. Each of these objects represent a particular deployment construct in the system. For example, there is an object that represents a group of containers (Pod), one that represents a deployment of pods

(Deployment), one for services, and so on. Even the node is represented as an object which you can query to view aspects of the current status like how much resources are being used.

To deploy a typical stateless web application into the cluster, you'll use three objects: the Pod, a Deployment (which encapsulates the Pod), and a Service.

Pod

The **Pod** is simply a collection of containers. Often this will just be a single container, but could be multiple in the case where tightly coupled containers need to be deployed together.

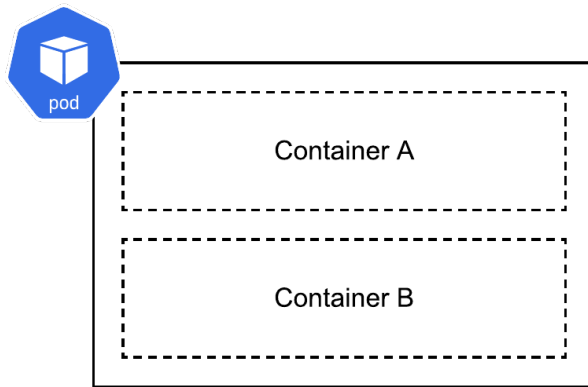


Figure 3.4 The Kubernetes pod. A pod can have one or many containers

The Pod is used as the primary scheduling unit in Kubernetes. Encompassing your application and its containers, it's the unit of compute that Kubernetes schedules onto nodes according to the resources you require. For example, if your workload requires 2 CPUs to run, you specify that in the Pod definition, and Kubernetes will find a machine with 2 available CPU resources.

How many containers to a Pod?

Except for in simple cases where a tightly coupled dependency exists between multiple containers, most containers are deployed individually with one container per Pod. Common situations where you might have multiple containers include so-called sidecars, where a second container is used for authorization, logging or some other function, and other situations where there is tight coupling between multiple containers, such that they benefit from being deployed together.

Were you to inspect the processes running on the node, you would not see the Pod itself, just a bunch of processes from the containers. The Pod itself is just a logical grouping of containers. It's Kubernetes that binds these containers together, ensuring that they share a common lifecycle: that they are created together, that if one fails they are restarted together, and that they are terminated together.

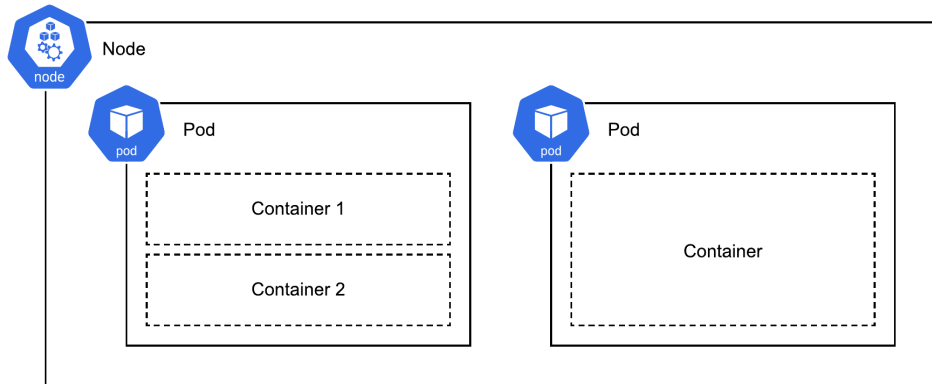


Figure 3.5 Multiple Pods running on a Node

DEPLOYMENT

While you can instruct Kubernetes to run Pods directly, this is rarely what you'll do. Applications crash, and machines fail, so Pods need to be restarted and/or rescheduled. Instead of directly scheduling Pods, it's better to wrap them into a higher order object that manages the Pod lifecycle.

For applications like web servers that need to run continuously, that object is a *Deployment*. Other options include a *Job* for running batch processes to completion, covered in a Chapter 10. In the deployment, you specify how many replicas of the Pod you wish to be running, and other information like how updates should be rolled out.

Like all objects in Kubernetes, a Deployment is a specification for the desired state of the system, which Kubernetes seeks to actuate. You can specify things like the number of replicas of your Pod, and (as we'll cover in later chapters) detailed requirements for how the Pods are spread across the cluster. Kubernetes continuously reconciles the observed state to the desired state while attempting to deliver what you requested. For example, were a Pod to become unavailable sometime after it was deployed like what would happen if the node it was running on failed, Kubernetes will observe that there are less Pods running than desired, and schedule new instances of the Pod to once again meet your requirements. These automated operations for scaling and repairing are the primary reason for using a Deployment to manage the lifecycle of a service, rather than running Pods directly.

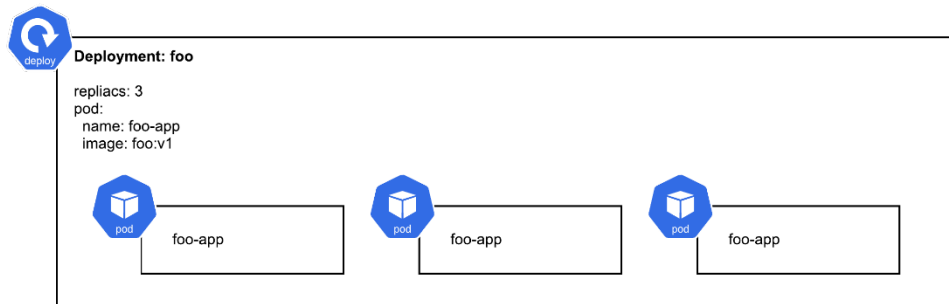


Figure 3.6 A deployment with 3 replicas of pod “foo-app”

SERVICE

Services are how you expose an application running on a set of Pods as a network service. A Service provides a single addressing mechanism and balances the load across the Pods. Services get their own IP address and DNS name which can be referenced by other Pods running within the cluster, and can also be assigned an external IP address.

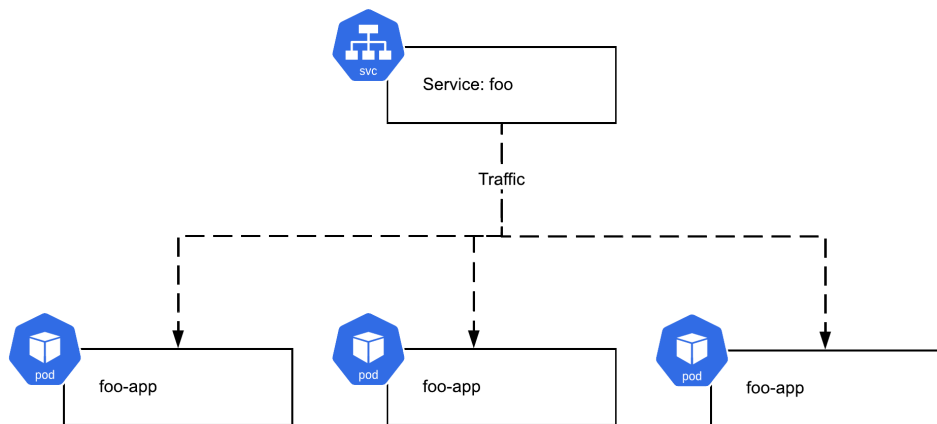


Figure 3.7 A Kubernetes Service

3.2 Deploying an Application

Let’s get started by deploying an application and making it available on the internet, and later update it with a new version. In other words, perform a basic application development,

release and update cycle with Kubernetes. To do this we'll be using the Kubernetes objects discussed in the prior section; a *Pod*, which will be managed by a *Deployment*, and exposed with a *Service*.

3.2.1 Creating a Cluster

Before deploying the application, you'll need a Kubernetes cluster to use. I recommend creating one on a public cloud, as it's less hassle to get set up *and* people can check out your creations immediately as you can share a public IP for any services you deploy. Many cloud providers have free trials to help reduce costs while learning.

Developing with a local Kubernetes cluster is another option, but there are some inherent differences between the environment of a local Kubernetes cluster and a cloud one, particularly around things like load balancing. I prefer to learn the environment that I can one day use in production, hence my suggestion while learning to pick a cloud provider and start with that.

PREFER A LOCAL CLUSTER? If you'd prefer to use a local distribution of Kubernetes, I've got you covered. Follow the steps in Section 3.4 (Local Development with Kubernetes) to get your `kubectl` command connected to a local cluster instead, then come back to Section 3.2.3 on Deploying to Kubernetes and continue. Just note that when you go to deploy your own locally built container image, there are some considerations that are outlined in Section 3.4 to ensure Kubernetes can find your image, and the way you access any services you create will be different (also outlined in that section) due to the lack of public load balancers.

At the end of the day, all you need to run just about every sample in this book is a Kubernetes cluster hosted somewhere, and the Kubernetes command line tool known as `kubectl`¹ authenticated to use that cluster, which any getting started guide should get you. The next two steps use Google Cloud, but I'll also include some instructions along the way on how to substitute the platform of your choice.

GOOGLE KUBERNETES ENGINE

Google Kubernetes Engine (GKE) was the first Kubernetes product to market and is a popular choice for trying out Kubernetes due to its maturity and ease of use. I work on the GKE team, and this is the platform that I know best, so it's the one I'll be using for the few places in this book where there are platform-specific requirements.

I've written this book to be applicable anywhere you find Kubernetes, and I expect that it will be useful for learning Kubernetes whether you're using GKE, OpenShift, AKS (Azure Kubernetes Engine), EKS (Elastic Kubernetes Service), or any one of the other Kubernetes platforms and distributions out there. There are a few places where the platform plays a role (like now, when creating a cluster), and in those instances I'll demonstrate the action with instructions for GKE, but I'll also be providing pointers on how to find the equivalents on other platforms.

¹ pronounced: "cube cuttle"

Creating a Kubernetes Cluster on Any Cloud

All you need to run the examples in this chapter after this setup section is the `kubectl` tool authenticated to the Kubernetes cluster of your choice. Creating and authenticating `kubectl` is the goal, and as you will see for GKE this can be done with two commands. You can substitute those commands for the equivalent cluster creation and authentication for the platform of your choice.

To run the following examples on any provider, follow the cluster creation guide for the provider of your choice, then continue to the section “Uploading Your Container”. Uploading containers is also another provider-specific action, but I’ve got you covered with some general tips there on how to get that done on any platform.

To start with GKE, you’ll need a Google Account (if you have a @gmail.com address, then you have a Google Account). Head over to <https://console.cloud.google.com/>, select your account, and review the terms. Activate your free trial if you have not already, or add billing info so you can run these samples (again, if you wish to run the samples locally, you can instead follow the steps in section 3.4 to get a local-only cluster).

With your account set up, head over to Kubernetes Engine in the console (direct link: <https://console.cloud.google.com/kubernetes>), and create a cluster. I recommend Autopilot mode, which takes care of the provisioning and management of nodes for you. With Autopilot, you can set a name, pick a region (as I’ve done in Figure 3.8), and leave the networking and advanced settings as the default.

Google Cloud Platform gke-autopilot-demo

← Create an Autopilot cluster HELP ASSISTANT

Create an Autopilot cluster by specifying a name and region. After the cluster is created, you can deploy your workload through Kubernetes and we'll take care of the rest, including:

- ✓ **Nodes:** Automated node provisioning, scaling, and maintenance
- ✓ **Networking:** VPC-native traffic routing for public or private clusters
- ✓ **Security:** Shielded GKE Nodes and Workload Identity
- ✓ **Telemetry:** Cloud Operations logging and monitoring

Name
kubernetes-for-developers

Region
us-west1

Networking

Define how applications in this cluster communicate with each other and how clients can reach them.

☒ Public cluster

☐ Private cluster

CREATE CANCEL Equivalent REST or COMMAND LINE

Figure 3.8 GKE Autopilot's cluster creation UI

Next, set up the command-line tools. You'll need the cloud provider CLI (in this case `gcloud`) to perform cluster operations like creating, and connecting to it, and `kubectl` for interacting with the Kubernetes API. Download the `gcloud` SDK at <https://cloud.google.com/sdk/install> and follow the installation instructions.

Once installed, run the `gcloud init` command to login. If you have more than one Google Account, be sure to select the same account that you created the cluster in earlier.

```
gcloud init
```

The Kubernetes CLI, `kubectl`, can be installed stand-alone (following the instructions at <https://kubernetes.io/docs/tasks/tools/>) or via `gcloud`. It doesn't matter how you install it, but since this example uses `gcloud`, we can conveniently use it to install `kubectl` like so:

```
gcloud components install kubectl
```

Once the cluster is ready and `gcloud` is configured, tap "Connect" in the UI, and copy and the `gcloud` command provided (as seen in Figure 3.9) into your shell to authenticate `kubectl`. That command will look like this:

```
gcloud container clusters get-credentials $CLUSTER_NAME --region REGION --project $PROJECT_NAME
```

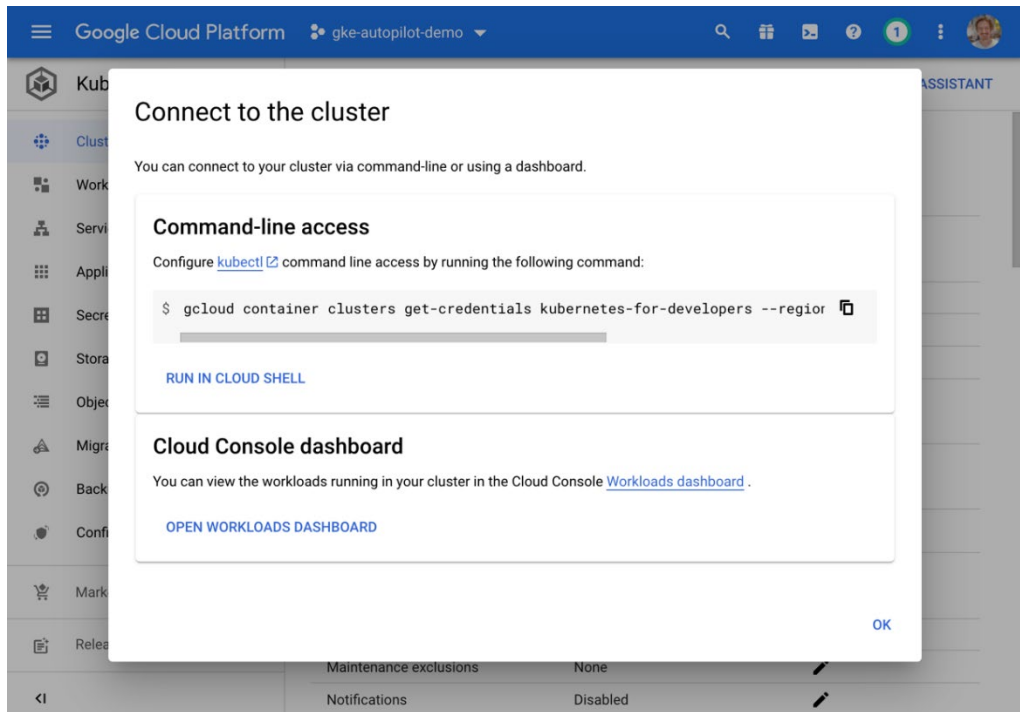


Figure 3.9 GKE's cluster connection UI

That command is the glue between the Google Cloud world and the Kubernetes one, and authenticates the `kubectl` CLI with the right credentials to be able to access your GKE cluster.

Creating Clusters in the CLI

Rather than using the UI, you can do both the creation and connection steps from the command line like so:

```
CLUSTER_NAME=my-cluster
REGION=us-west1
gcloud container clusters create-auto $CLUSTER_NAME --region $REGION
gcloud container clusters get-credentials $CLUSTER_NAME --region $REGION
```

With your cluster created, and `kubectl` authenticated, you're ready to get going with your first application! To make sure everything is working, run `kubectl get pods`. It should state that there are no resources (since we've yet to deploy any Pods).

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

If you get an error, it is likely that your cluster wasn't created or authenticated correctly – try repeating the above steps, or lookup the error message.

3.2.2 Uploading your Container

Till now, the containers we've created have been stored and run locally on your machine. Before you can deploy the container into Kubernetes running in the cloud, you'll need to upload your container image to a container registry. This is simply a place that stores the container image data, and provides a way for Kubernetes to fetch the image. Most registries support options for public images that anyone can use (like for open source projects and samples for a book), or private images which require authentication (which you will use for your own proprietary applications).

If you prefer, you can skip this step and use the publicly available image referenced by the examples below, but I'd recommend to build and upload your own container to use, so you can deploy your own applications when the time comes.

When choosing a container registry to use, DockerHub is a popular choice, particularly when it comes to public container images. This includes the base images (like the ones we used in the previous chapter), open source software like MariaDB, or perhaps your own software and demos you wish to share with the world. You can also access private container images from DockerHub (and other registries) from any Kubernetes platform, with a bit of extra configuration to set up the credentials.

The default choice for most users who wish to keep their images *private* is to use the container registry of your cloud provider, as this generally gives you efficiencies in terms of image pull time, reduced network data costs, and simpler authentication. For Google Cloud, that's Artifact Registry, on AWS it's Amazon Elastic Container Registry, on Azure it's Azure Container Registry, and so on.

Once you have chosen your preferred location, follow these steps to get your containers uploaded.

ACCOUNT SETUP

To get started, first create an **account** at your preferred provider if you don't have one already, and then create a **repository** where you'll be uploading the images.

For DockerHub, head over to <https://hub.docker.com/>, sign in, then hit "Create Repository".

For Artifact Registry, go to <https://console.cloud.google.com/artifacts>, and create a new Repository of type Docker in your desired location. Make a note of the path that is generated, which will look something like `us-docker.pkg.dev/my-project/my-repository`.

AUTHENTICATE

Next, you want to authenticate the `docker` command-line tool so it can upload images to your freshly created repository. Follow the instructions for your container registry to authenticate the docker command-line tool.

To do this DockerHub, you would run:

```
docker login
```

For Artifact Registry, recall the path of the repository you created earlier. Take the host portion of that path, for example `us-docker.pkg.dev`, and run the following command to install a credential helper to the docker tool so you can upload images there. You can run this multiple times, for each separate host you use.

```
HOST_NAME=us-docker.pkg.dev
gcloud auth configure-docker $HOST_NAME
```

Authenticate Docker with Any Cloud

Authenticating Docker with the Cloud of your choice is typically an easy operation: just find the equivalent command to configure docker with the needed credentials. The search query “authenticate docker with [your cloud provider] container registry” should do the trick!

TAG

When you build images, they are assigned a random hash-based name, like `82ca16cefe84`. Generally, it is a good idea to add your own tag that is somewhat meaningful so you can easily refer to your own images. In the previous chapter, we used these tags so we could run our images locally using nice names like `docker run timeserver` instead of `docker run 82ca16cefe84`.

When you upload containers into container registries, the tag takes on an additional meaning. You are required to tag the image with a name that follows a specific path convention dictated by the container registry in order for it to know which account and path to store the image in (and so that your local docker client knows which registry to upload it to). Tagging your image with a simple name like “timeserver” won’t work when you’re uploading to these repositories.

DockerHub uses the following convention:
`docker.io/$USERNAME/$REPOSITORY_NAME:$VERSION_TAG`

Where `$USERNAME` is your docker username, and `$REPOSITORY_NAME` is the name of the repository you created in DockerHub, and the version tag is an arbitrary string (typically including a number). Putting it together, in my own case where my username is “wdenniss” and my repository is “timeserver”, the string I get is `docker.io/wdenniss/timeserver:1`.

THE VERSION TAG The version tag is an unstructured string used to refer to the version of the image. The convention is to use the version number (potentially constructed as `major.minor.patch`) and optionally with a suffix, for example `2`, `2.1`, `2.1.5`, `2.1.5-beta`, etc. A special version tag `latest` can be used to refer to the most recent image when running containers, but don’t use `latest` when tagging the images for upload, as it’s applied automatically by the container repository.

Each repository has its own format. For Google Cloud’s Artifact Registry, the format consists of the following construction:

```
$LOCATION-docker.pkg.dev/$PROJECT_ID/$REPOSITORY_NAME/$IMAGE_NAME:$VERSION_TAG
```

After you create the Artifact Registry repository in the UI console, you should see the first portion of this string displayed, for example `us-docker.pkg.dev/wdenniss/ts`, which you can copy (or you can build the string using the formula above). To this prefix, append any image name and tag that you like, such as `timeserver:1`. Put it together, and you'll get something which for me looks like the following `us-docker.pkg.dev/wdenniss/ts/timeserver:1`.

Container Registry Tag Conventions

Every private container registry has its own magic string concatenation that you need to do to create the right tag, and they're all different. For example Azure provides the following example²: `<acrLoginServer>/hello-world:v1`, and AWS documents³ `<aws_account_id>.dkr.ecr.<region>.amazonaws.com/my-web-app`. One thing I'm sure about: make sure you follow the guidelines of whatever container registry you're using, otherwise Kubernetes won't know where to push the image. The search term I use is "[cloud provider] registry container tag name"

Once you've worked out the right image tag to use (which we'll refer to as `$IMAGE_TAG` in the remaining examples), you can go and tag any existing docker image for uploading. To upload one of the images we built in the earlier chapter to a container registry, you can reference the image from its previous tag and add a container registry tag (images can have multiple tags). Our example in section 2.2 was built with `docker build . -t timeserver`, so this image has the tag `timeserver` already, which means we can re-tag it for the container registry like so:

```
IMAGE_TAG=us-docker.pkg.dev/wdenniss/ts/timeserver:1
docker tag timeserver $IMAGE_TAG
You can view the resulting list of images like so:
$ docker images
REPOSITORY                                TAG      IMAGE ID      CREATED
timeserver                                latest   c07e34564aa0  2 minutes ago
us-docker.pkg.dev/wdenniss/ts/timeserver  1        c07e34564aa0  2 minutes ago
python                                    3.10     cf0643aafe49  1 days ago
```

You can also look up existing images and tag them based on Image ID (`docker tag $IMAGE_ID $IMAGE_TAG`), but I suggest tagging when you build to avoid confusion. In fact, I generally find it quicker to simply rebuild the image than try to find the right one tag after the fact.

To build and tag the sample container, replace `$IMAGE_TAG` with your own repository image name, and from the root sample directory run:

```
IMAGE_TAG=us-docker.pkg.dev/wdenniss/ts/timeserver:1
cd Chapter02/timeserver
docker build . -t $IMAGE_TAG
```

² <https://docs.microsoft.com/en-us/azure/container-registry/container-registry-get-started-azure-cli>

³ <https://docs.aws.amazon.com/AmazonECR/latest/userguide/docker-push-ecr-image.html>

PUSH

Once our repository is set up, docker is authenticated and our image tagged, you can push the image to the repository with:

```
docker push $IMAGE_TAG
```

The previous authentication step installed a helper into the docker configuration which enables Docker to speak with your cloud's container registry, whatever that may be. If you get a "permission denied" error, either you didn't authenticate docker correctly, or your image tag string construction is wrong. Verify that you authenticated docker to the appropriate repository, and set the correct image tag. Refer to the up to date docs for your chosen container registry for guidance.

If it goes well, you should see output like the following. Pay particular attention to the last line, which is where any authentication errors will be displayed.

```
$ docker push $IMAGE_TAG
The push refers to repository [us-docker.pkg.dev/wdenniss/ts/timeserver]
9ab1337ca015: Pushed
3eaafa0b4285: Layer already exists
a6a5635d5171: Layer already exists
8c25977a7f15: Layer already exists
1cad4dc57058: Layer already exists
4ff8844d474a: Layer already exists
b77487480ddb: Layer already exists
cd247c0fb37b: Layer already exists
cfd5c3bd77e: Layer already exists
870a241bfebd: Layer already exists
1: digest: sha256:edb99776ae47b1b16257f6743525a00fcc4f86c354c43a61397f7a9f1864afe7 size:
2425
```

Once the image has been uploaded, you're now ready to deploy your code into Kubernetes!

3.2.3 Deploying to Kubernetes

With a cluster created, and `kubectl` authenticated, we can deploy our first application. To do this, we'll create an aptly named Deployment object. Kubernetes uses declarative configuration, where you declare the state you want (like "I want 3 copies of my container running in the cluster") in a configuration file, then submit that config to the cluster, and Kubernetes will strive to meet the requirements you specified.

For the configuration file, most developers use YAML as it's easier to edit manually. JSON is another option (primarily used with automated access), and some configuration can be created imperatively (covered later in section 3.3). Here's a minimal Deployment specification for the Timeserver application from Chapter 2. It references a public container image built from the included sample app which I have uploaded to DockerHub. If you have your own image, such as one pushed to a container repository in the previous section, edit this file and replace my image with yours.

Listing 3.1 DeployingToKubernetes/deploy.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 3 #A
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:1 #B

```

#A How many Pod replicas (instances) to deploy

#B Which container image to deploy and run

This manifest will create three replicas of our container. Later we'll see how to configure a load balancer to split incoming requests over these three running instances. In this example of a minimalist deployment config, the three most important lines are the name, which is needed to inspect, modify and delete the deployment, the replica count and the container name. The rest is basically glue to make it all work (don't worry, I'll explain how the glue works as well).

The container image path is like a URL that references where to find the container. If you uploaded your container following the previous section, you already have this image path from that step. My container image with the docker.io prefix is available on DockerHub, a popular place to host public images including base images. One thing to note is that if you ever see an image path without a domain, like `ubuntu`, or `wdenniss/timeserver`, it's simply shorthand for images hosted on DockerHub.

So that's the deployment, let's go and create it in the cluster. From the root sample directory, run:

```

cd Chapter03/3.2_DeployingToKubernetes/
kubectl create -f deploy.yaml

```

This instructs Kubernetes to create the object defined by the configuration file. If you need to make changes once it's deployed (like changing the image version), you can make your changes locally, and update the deployment in the cluster with:

```

kubectl apply -f deploy.yaml
To observe the state of the deployment, run:
$ kubectl get deploy
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
timeserver 3/3      3            3           36s

```

This shows the state of the deployment. As mentioned earlier, the deployment is declarative statement of your desired requirements, for example, "3 replicas of this Pod". When you

created the deployment and the system returned a success response, this simply means that it accepted your deployment for scheduling—not that it had completed scheduling in the manner you desired. Querying the deployment with `get` will show you the current status such as how many of the Pods are ready to serve traffic (the number in the “Ready” column), and later when you update the deployment, how many of the pods are running the latest version during a roll out of a new version (the number in the “up to date” column).

To see more detail about the Pods which form your deployment, you can also query the Pods themselves:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
timeserver-6df7df9cbb-7g4tx	1/1	Running	0	68s
timeserver-6df7df9cbb-kjg4d	1/1	Running	0	68s
timeserver-6df7df9cbb-lfq6w	1/1	Running	0	68s

PENDING PODS If Pods show pending here, it may mean that your cluster doesn't have enough resources. In the case of a dynamically provisioned environment, simply waiting a minute or so is generally enough to see them scheduled. If they stay in Pending, review the “Stuck in Pending” advice below.

The `kubectl get pods` command returns the state of *all* pods in the active namespace, so once you have a lot of deployments, this might get a big jumbled. Instead, you can use a more verbose form where you pass the deployment's label (discussed earlier in this chapter) as a selector. Here's a complete example, using the label of our earlier example deployment.

```
$ kubectl get pods --selector=pod=timeserver-pod
```

NAME	READY	STATUS	RESTARTS	AGE
timeserver-6df7df9cbb-7g4tx	1/1	Running	0	2m13s
timeserver-6df7df9cbb-kjg4d	1/1	Running	0	2m13s
timeserver-6df7df9cbb-lfq6w	1/1	Running	0	2m13s

Once the Pod is “Running” we can interact with it! To connect to our fresh deployment and visit the server that we deployed before creating a public IP, we can simply forward a port from our local machine to the containers like so:

```
$ kubectl port-forward deploy/timeserver 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

This allows you to interact with the deployment from localhost, by browsing to <http://localhost:8080>. As you try out the containerized application, you can watch the log output in a new shell like so:

```
$ kubectl logs -f deploy/timeserver
Found 3 pods, using pod/timeserver-8bbb895dc-kg18l
Listening on 0.0.0.0:80
127.0.0.1 - - [09:59:08] "GET / HTTP/1.1" 200 -
```

The logs command, using the “-f” (follow) param, will stream the logs from one of the pods in the deployment. It's a good idea to log a statement to stdout in your own apps, as was done here with the “Listening on 0.0.0.0:80”, so that you can be assured the container really did start as expected.

TROUBLESHOOTING

Most actions you take in Kubernetes are not instant. Creating a Pod takes time, to provision new compute capacity (depending on what Kubernetes platform you are using), download the container from the container registry, and boot your container. If everything goes well, you should have running containers in a couple of minutes.

When things have succeeded, the Pods from your deployment will report a status (when queried with `kubectl get pods`) of "Running". You may see other statuses like "Pending" while it is waiting for capacity, and "ContainerCreating" once the container has been scheduled to your nodes and is booting. Confusing matters is that sometimes a Pod can get stuck in "Pending"—it's a bit of an ambiguous state—and there can be other errors. What follows is a list of common situations.

Image Pull Error (ErrImagePull / ErrImagePullBackoff)

This error means that Kubernetes was unable to download the container image. This typically means that either the image name was misspelt in your configuration, the image doesn't exist in the image repository, or your cluster doesn't have the required credentials to access the repository.

Check the spelling of your image, and verify that the image is in your repository. For a quick fix just to get the Deployment running, try a public container image like the one I've provided. Any fixes you make to your Deployment configuration can be applied using `kubectl apply -f deploy.yaml`.

STUCK IN PENDING If you see a pod stuck in the "Pending" state for more than a minute or so, it typically means that the Kubernetes scheduler is unable to find space on your cluster to deploy the images to. Often this can be resolved by adding additional resources to your cluster, like an extra or larger compute node.

You can see the details of the pending Pod by "describing" it, as follows:

```
kubectl get pods
kubectl describe pod $POD_NAME
```

Where `$POD_NAME` is one of the pods in the pending state. The "Events" section contains a list of any errors that Kubernetes has encountered. If you attempted to schedule a deployment and there were no resources available, you'll see a warning like "FailedScheduling". Here's the complete events section for a Pod that I attempted to schedule, but where there were not enough resources:

```
Events:
  Type      Reason            Age          From          Message
  ----      -
  Warning   FailedScheduling  26s (x2 over 26s)  default-scheduler  0/2 nodes are available:
  2 Insufficient cpu.
```

As long as at least one of your Pods is in the "Running" state, you don't need to worry for now, as your service should still run as long as there exists one Pod to answer requests, but if they are all pending, you'll need to take action—likely by adding more compute resources.

CRASHING (CRASHLOOPBACKOFF) Another common error is a crashing container. There can be various reasons for a crashing container, including that the container failed to start (possibly due to a configuration error, for example), or that the container crashes soon after starting.

For the purposes of Kubernetes deployments, a “crash” is any process that terminates, even one that terminates with a success exit code. Deployments are designed for long-running processes, not once-off tasks (Kubernetes does have a way to represent a Pod that should be scheduled to run as a once-off task, and that is the Job type, covered in a later chapter).

The occasional crash of a container in a Deployment-managed Pod like the ones we are deploying here is handled gracefully, by restarting it. In fact, when you run `kubectl get pods`, you can see how many times a container has been restarted. You can have a container that crashes every hour, and as far as Kubernetes is concerned that’s totally fine; it will keep restarting it and it will go on its merry way.

A container that crashes either instantly at boot or quickly after, however, is put into an exponential backoff loop, where rather than continuing to restart it continuously (consuming the resources of the system), Kubernetes introduces a delay between restart attempts that increases exponentially (i.e. 10s, then 20s, 40s, etc).

When a container crashes the first time it will have a status like `RunContainerError` (for a container that errored at start), or `Completed` for one that exited. Once the crash has been repeated a couple of times, the status will move to `CrashLoopBackOff`. The chances are, any container in the `CrashLoopBackOff` state has an issue that needs your attention. One possibility is that the container may exit when an external dependency (like a database) is not being met, in which case you should ensure that the external service is running and can be connected to.

To debug crashed containers, I’d always start with `kubectl describe pod $POD_NAME` like the earlier issues to view the events for clues there. The container’s logs are another good place to check. As described earlier, you can retrieve these with `kubectl logs -f $POD_NAME`. When dealing with crashing containers you may wish to view the logs from the *prior* instantiation of the container (before it was restarted after crashing), so as to see any error printed when it crashed, as this often will indicate the cause. To do that, add `--previous` (or just `-p`), to your log request.

```
kubectl logs -p $POD_NAME
```

3.2.4 The Podspec

It’s worth taking a moment to understand how the Deployment object is composed, since it actually encapsulates a Pod object which has its own specification. This is a pattern you will see repeated with other higher order workload types in Kubernetes like Job. It’s also relevant because the way that we expose the Deployment in a Service is actually by referencing the Pods, and not the Deployment.

When you create a Deployment of three replicas, what is actually happening is that you are instructing the Kubernetes Deployment controller to create and manage three Pods. The Deployment manages the lifecycle of these pods, including replacing them with newer versions when you update the Deployment with a new container, and rescheduling Pods that

get evicted due to planned or unplanned maintenance events. Figure 3.10 has a visual breakdown of this object composition.



Figure 3.10 Pod object embedded in the Deployment object

The Pod object template is referred to throughout the Kubernetes documentation as the Podspec. You can actually yank it out and run it by itself. To do so, you'll need to construct a new header specifying that this object is of kind "Pod" rather than "Deployment"; then you can copy the entire YAML under "template" into the root of the config. Like so.

Listing 3.2 ThePodspec/pod.yaml

```

apiVersion: v1
kind: Pod
metadata:
  name: timeserver
  labels:
    pod: timeserver-pod
spec:
  containers:
    - name: timeserver-container
      image: docker.io/wdenniss/timeserver:1

```

You can go ahead and create this Pod directly. Such pods are "unmanaged" by any Kubernetes controller. They will be rebooted if they crash, but if they are evicted due to causes such as an upgrade event or node failure, they won't be rescheduled. That's why typically you won't schedule the Pod directly, but rather will use a higher-order object like a Deployment, or (as we'll see in the later chapters) StatefulSet, Job and others.

THE UNIVERSAL PODSPEC One of the key takeaways of this object composition in Kubernetes is that every time you see a Podspec in an object like a deployment, know that it carries all the capabilities of a Pod. That means you can look at the document for Pod, and use any of the values within the pod template of the managed object.

The Podspec includes key information about your application, including the container or containers that comprises it. Each of these containers has its own name (so you can reference the individual containers in a multi-container pod), as well as the most important field: the container image path. There are a lot of optional fields as well, including some important ones to specify health checks and resource requirements, which are covered in the coming chapters.

There are some seemingly repetitive labels in the Deployment and its embedded Podspec. The Deployment's spec has a "selector -> matchLabels" section, and the Pod spec has a "metadata -> labels" section, both containing the same key value pair "pod: timeserver-pod". So, what's going on here?

Well, since the Pod object actually exists somewhat separately after creation (it is created as a separate object that is managed by the Deployment), we need a way to reference it. Kubernetes solves this by giving the pod a label (which is an arbitrary key/value pair), and then we reference (select) that same label from the Deployment. This is essentially the glue that binds the two objects together. It's easier to visualize in a diagram, so see figure 3.11.

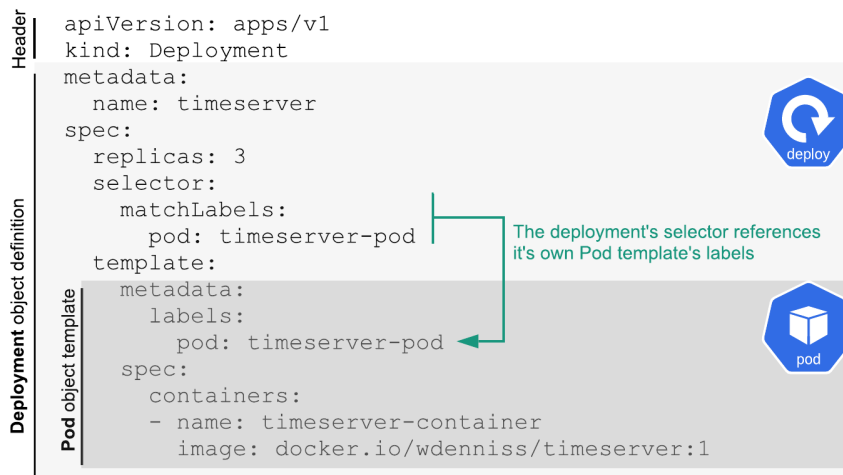


Figure 3.11 Relationship of the Deployment's selector, and the Pod template's labels

This may seem unnecessary, after all: can't Kubernetes do this object linking for us, since the Podspec is embedded in the Deployment? The reason why developers need to specify these labels manually is that you may reference the Pods which the Deployment manages directly using these labels through other systems. For example, in the next section where we configure a network Service, it references the pods directly, not the Deployment. The same is true for other concepts covered later in the book such as a Pod Disruption Budget. By specifying the label for your Pods, you will know what label to reference in these other systems. The Pod is the fundamental execution and scheduling unit in Kubernetes, and the Deployment is just one of many ways to create, manage, interact with Pods.

As for the key value label itself, it's completely arbitrary. You can use "foo: bar" for all Kubernetes cares. I used "pod: timeserver-pod", as I find it reads well when selecting Pods in other objects. A lot of documentation uses something like "app: timeserver". I have avoided repeating the name of the deployment ("timeserver") as the value of this label to avoid the misconception that the name of the Deployment has anything to do with the Pod label (since it doesn't).

So that's how the Deployment object is constructed with an embedded Podspec. I hope it's useful to understand this object composition, and how the Pod is referenced. In the next section we'll be exposing this Deployment to the world, which will reference the Pod by its labels.

3.2.5 Publishing your Service

With your container successfully deployed, no doubt you'll want to interact with it!

Each Pod is given its own cluster-local (internal) IP address which other Pods can use to communicate. It's possible to expose Pods directly on the internet as well (with the field "hostPort"), but unless you're writing a real-time game server, that's rarely what you'll do. Typically, and especially when Deployment is used, you will aggregate your Pods into a Service, which provides a single access point with an internal (and optionally external) IP, and will load balance requests across your pods. Even if you had a Deployment of a single Pod, you'll still want to create a Service to provide a stable address.

In addition to load balancing, Services keep track of which Pods are running and capable of receiving traffic. For example, while you may have specified three replicas in your Deployment, that doesn't mean that three replicas will be available at all times. There might only be two if a node is being upgraded, or there could be more than three while you're rolling out a new version of your Deployment. The Service takes care of this, and will only route traffic to running Pods (in the next chapter, we'll cover some key information you need to provide to make that work smoothly).

Services are used internally within the cluster to enable communication between multiple applications (a so-called microservice architecture), and offer convenient features such as service discovery for this purpose. This is covered in detail in Chapter 7. For now, let's focus on using a Service to expose your new application to the internet by specifying a "LoadBalancer" type service, to get it in the hands of end-users.

As with the deployment we'll start with a skeleton YAML configuration:

Listing 3.3 DeployingToKubernetes/service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: timeserver
spec:
  selector: #A
    pod: timeserver-pod #A
  ports:
    - port: 80 #B
      targetPort: 80 #C
      protocol: TCP #D
      type: LoadBalancer #E
```

#A Which Pods traffic to this service will be routed to

#B The port the service will be exposed on

#C The container's destination port that traffic will be forwarded to

#D The network protocol

#E Type of Service, in this case an external load balancer

The port list allows you configure which port to expose for users of the Service (`port`), and what port of the Pod that this traffic will be sent to (`targetPort`). This allows you to, say, expose a service on port 80 (the default HTTP port), and connect it to an application in a container running on port 8080.

Each Pod and Service in Kubernetes has its own internal cluster IP, so you don't need to worry about port conflicts between Pods. This means that you can run your application on whatever port you like (such as port 80 for a HTTP service), and use the same number for `port`, and `targetPort` for simplicity, as with the example above. If you do this, you can omit `targetPort` completely as the default is to use the `port` value.

All Services are given an internal, cluster-local IP address which Pods in the cluster can use. If you specify "type: LoadBalancer" as above, then an external IP address will be provisioned in addition.

Notice also that this service has a section named "selector", like our Deployment had. The Service doesn't reference the Deployment, and actually has no knowledge of the Deployment at all. Instead, it references the set of Pods that all have that label (which in this case will be the Pods created by our Deployment). Once again, it's easier to visualize so see Figure 3.12.

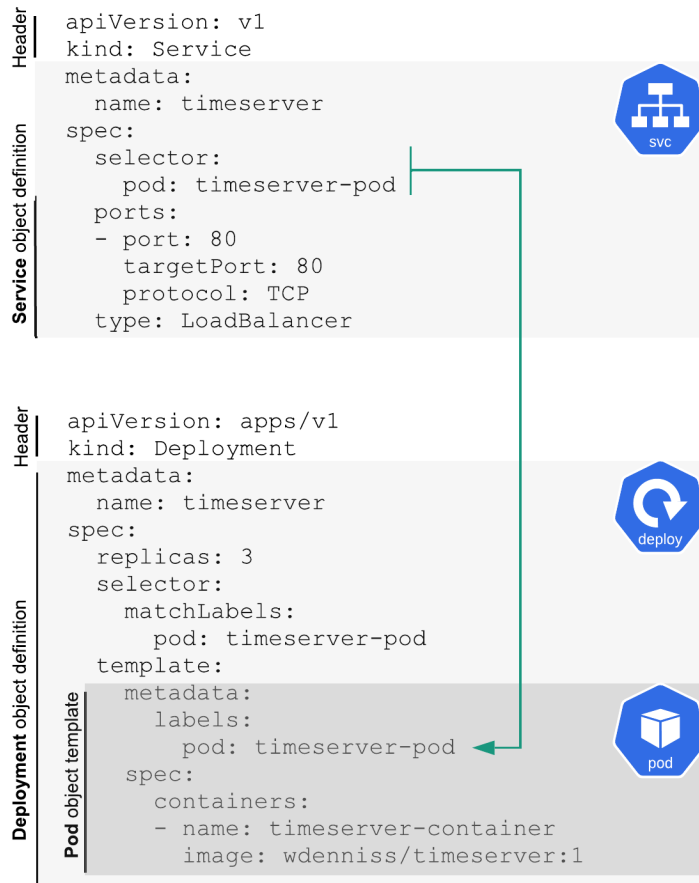


Figure 3.12 Relationship between the Service and the Pods it targets (selects)

Unlike in the Deployment object, the "selector" section has no "matchLabels" sub-section. They are, however, equivalent. Deployment is just using a newer, more expressive syntax in Kubernetes, but as used here, the selectors in the Deployment and in the Service are achieving the same result: specifying the set of pods that the object is referencing.

Create the Service object on your cluster with:

```
cd Chapter03/3.2_DeployingToKubernetes
kubectl create -f service.yaml
```

Notice how the creation command (`kubectl create`) is the same for the Deployment as the Service. All Kubernetes objects can be created, read, updated, and deleted (so-called CRUD

operations) with four `kubectl` commands: `kubectl create`, `kubectl get`, `kubectl apply` and `kubectl delete`.

To see the status of your Service, you can call `kubectl get` on the object type, like so:

```
$ kubectl get service
NAME         TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)        AGE
kubernetes   ClusterIP   10.22.128.1    <none>         443/TCP        1h
timeserver    LoadBalancer 10.22.129.13   203.0.113.16   80:30701/TCP   26m
```

Notice that your service is there (in this example, `timeserver`), as well as another service named `kubernetes`. You can ignore the `kubernetes` service if one is shown, as that's the Kubernetes API service itself running in your cluster. You can also specify just the service you're interested in with `kubectl get service $SERVICE_NAME`.

If the "External IP" in the output indicates "Pending", this just means it's waiting for the Load Balancer to come online. It's common for this to take a minute or two, so no need to rush to debug why it's Pending unless it's been that way for a while. Rather than repeating the above "get" command repeatedly, you can stream any changes to the status by adding the `--watch/-w` flag, i.e.: `kubectl get service -w`. Run that command, and within a couple of minutes, you should see output indicating that your service now has an external IP.

Note that to have an external IP provisioned, you must be running Kubernetes on a cloud provider, as the provider is provisioning an externally routable network load balancer behind the scenes. If you're developing locally, see section 3.4.3 on how to connect using tools like `kubectl port-forward`.

Once the IP comes online, try accessing the service by visiting the URL. In this example our external IP was `http://203.0.113.16` (but replace with your own external IP from `kubectl get service!`). Curl is great for testing HTTP requests from the command line (`curl http://203.0.113.16`); viewing it in a browser works just as well, too.

```
$ curl http://203.0.113.16
The time is 7:01 PM, UTC.
```

TROUBLESHOOTING

Unable to connect

Two common reasons for this: 1) the selector is incorrect, or 2) your ports are wrong. Triple check that the selector matches the labels in your deployment's Pod template. Verify that the target port is indeed the port your container is listening on (a boot-time debug message in the container printing the port can be a good idea to help verify this), and that you're connecting to the right port from your browser.

See if you can connect to one of your Pods directly on the `targetPort` using `kubectl's` port forwarding capability. If you can't connect to the Pod directly, then the issue is likely with the Pod, and if it does work then the issue could be an incorrect Service definition. You can set up a port forward to one of the Pods in the deployment like so:

```
kubectl port-forward deploy/$DEPLOYMENT_NAME $FROM_PORT:$TO_PORT
Where FROM_PORT is the port you'll use locally, and $TO_PORT is the targetPort that you
defined in your service. Using our example earlier, this would be:
kubectl port-forward deploy/timeserver 8080:80
```

Then browse to <http://localhost:8080>. This will select one of the Pods in the deployment automatically (bypassing the Service). You can also specify a specific pod to connect to directly with:

```
kubectl port-forward pod/$POD_NAME $FROM_PORT:$TO_PORT
```

External IP stuck in pending

It can take a little while to get an external IP, so give it a few minutes. Verify that your cloud provider will provision external IPs for Services of type LoadBalancer. Check the provider's documentation for any additional information around setting up load balancers in Kubernetes.

If you're running locally, or just want to try out the service without waiting for the External IP, you can forward a port on your machine to the service like so:

```
kubectl port-forward service/$SERVICE_NAME $FROM_PORT:$TO_PORT
```

3.2.6 Interacting with the Deployment

RUNNING ONE-OFF COMMANDS

Just as we can run one-off commands on the docker image using the `docker exec` command (covered in Chapter 2), we can also run one-off commands on our Pods with `kubectl exec`. A common command used to diagnose issues in the container is `sh` which will give you an interactive shell on the container (provided that `sh` is available in the container), and from there you can perform whatever other debugging steps you need to do inside the container.

Technically `exec` is run against a pod, but we can specify the deployment instead of a specific pod, and `kubectl` will select one pod at random to run the command on.

```
$ kubectl exec -it deploy/timeserver -- sh
# echo "Testing exec"
Testing exec
You can run any command on the container in this way, for example
$ kubectl exec -it deploy/timeserver -- echo "Testing exec"
Testing exec
```

COPYING FILES TO/FROM THE CONTAINER

Again, similar to `docker`, `kubectl` has a `cp` command allowing you to copy files between your system and the container. This command requires that the `tar` binary be present in your container image. This can be useful when you want to download your application logs or other diagnostic information.

```
kubectl cp $POD_NAME:/path/to/file .
You can also copy files in the other direction.
kubectl cp $FILE $POD_NAME:/path/
```

3.2.7 Updating your Application

Now that your application has been deployed and published to the world, no doubt you'll want to be able to update it!

Make a code change to the sample app, then build and push the container image to the container repository, with a new version tag. For example, if you used `us-docker.pkg.dev/wdenniss/ts/timeserver:1` before, your new image could be `us-docker.pkg.dev/wdenniss/ts/timeserver:2`. You can make this label anything you like, but it's a good convention to use version numbers.

Once the container image has been pushed to the repository (as we did in section 3.2.2), update your `deploy.yaml` file with the new image name.

```
For example (emphasis added):
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 3
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:2 #A
```

#A New image version

Save the file, and apply the change to your cluster with:

```
$ kubectl apply -f deploy.yaml
deployment.apps/timeserver configured
```

When you apply this change, an interesting thing happens. Remember how Kubernetes seeks constantly to actuate your requirements, driving the state it observes in the system to the state you required? Well, since you just declared that the deployment is now using the image with the version tag 2, and all the Pods are currently tagged 1, Kubernetes will seek to update the live state so that all Pods are the current version.

We can see this in action by running `kubectl get deploy`. Here's some example output:

```
$ kubectl get deploy
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
timeserver    3/3     1            3           10m
```

The "READY" column shows how many pods of the are serving traffic, and how many we requested. In this case, we have all 3 that are ready. The "UP-TO-DATE" column, however, indicates that only 1 of these Pods are the current version. This is because, rather than replacing all the pods at once, something that would cause some downtime to the application, by default pods are updated with a so-called rolling update strategy, that is, one or several at a time.

Rolling update and other rollout strategies are covered in detail in the next chapter, as well as important health checks that need to be configured to avoid glitches during the

rollout. For now it's enough to know that Kubernetes will actuate your changes and will replace the old v1 pods with the new v2 ones.

Once the "UP-TO-DATE" count is equal to the ready count, the rollout is complete. You can also observe the individual pods being created and replaced with `kubectl get pods`, which will show a list of all Pods in the deployment, both new and old.

MONITORING THE ROLLOUT

Since the output of the `kubectl get` commands displays the moment-in-time information, but the deployment is continuously changing, most operators will monitor the deployment in an automated way, avoiding the need to constantly re-run the same command.

Kubernetes includes one such option, the `--watch/-w` flag, which can be added to most `kubectl` commands. For example, `kubectl get pods -w`, and `kubectl get deploy -w`. When watch is specified, any changes to the status will be streamed to the console output.

The disadvantage of the watch flag is that it kind of jumbles the output. If you have many pods changing, you'll see line after line printed and it's easy to lose sight of the current state of the system. My preference is to use the Linux `watch` command instead. Unlike the watch flag, the watch command refreshes the entire output, optionally showing you what changed between the current and the last update. This command is available in most Linux distros, macOS, and the Windows Subsystem for Linux (WSL), and can be found wherever you get your packages.

When watch is installed, you can simply prepend it to any `kubectl` command, e.g.:

```
watch kubectl get deploy
My favorite watch flag is -d, which will highlight any changes, i.e.:
watch -d kubectl get deploy
```

With a terminal window (or tmux session window) opened for watching each command you can put together a real-time status dashboard with just `kubectl`.

WATCHING THE DEPLOYMENT

The `kubectl get deploy` and `kubectl get pods` commands above return all deployments and pods, respectively, in the current cluster namespace. As you make more deployments, you may want to specify just the resources you're interested in, like so:

```
kubectl get deploy $DEPLOYMENT_NAME
```

The name of the object can be found in the "name" field in the metadata section at the top of the file. Viewing all pods from a single deployment is a little more tricky; however, you can use the label selector to get the status of a set of Pods like so:

```
kubectl get pods --selector=pod=timeserver-pod
```

Where `pod=timeserver-pod` is the label selector specified in the deployment.

3.2.8 Cleaning Up

There are a number of ways to clean up the objects we've created in this chapter. You can delete by name. Since our deployment, service, and pod had the same name, we can do it all at once like so:

```
$ kubectl delete deploy,service,pod timeserver
deployment.apps "timeserver" deleted
service "timeserver" deleted
pod "timeserver" deleted
Or you can delete objects by referencing their configuration:
$ cd Chapter03
$ kubectl delete -f 3.2_DeployingToKubernetes
deployment.apps "timeserver" deleted
service "timeserver" deleted
$ kubectl delete -f 3.2.4_ThePodSpec
pod "timeserver" deleted
```

If after deleting, you change your mind, you can simply create them again (`kubectl create -f 3.2_DeployingToKubernetes`). That's the beauty of capturing your configuration in files: you don't need to remember any tweaks you made to the live state, because everything is updated first in the configuration.

3.3 Declarative Commands

Kubernetes offers two approaches for interacting with the system: declaratively, where you specify (declare) in configuration files the state that you want, and apply those configurations to the cluster; and imperatively, where you instruct the API one command (imperative) at a time to perform your wishes. The configuration driven declarative model is the approach that is strongly preferred by most practitioners (including myself), and what you'll most often encounter in a workplace.

In fact it's possible to create a Deployment without our container and expose it to the internet using purely imperative commands. For completeness, here's how you would do that.

Create the Deployment:

```
$ kubectl create deployment timeserver --image=docker.io/wdenniss/timeserver:1
deployment.apps/timeserver created
Create a Service of type LoadBalancer on Port 80 to expose this service:
$ kubectl expose deployment timeserver --type=LoadBalancer --port 80
service/timeserver exposed
```

Observe the result:

```
$ kubectl get deploy,svc
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/timeserver	1/1	1	1	4m49s

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/kubernetes	ClusterIP	10.22.128.1	<none>	443/TCP	5m27s
service/timeserver	LoadBalancer	10.22.130.202	<pending>	80:31215/TCP	31s

And to update the container in the deployment with a new version:

```
$ kubectl set image deployment timeserver timeserver=wdenniss/timeserver:2
deployment.apps/timeserver image updated
```

This option may look simpler at first brush, when compared to controlling Kubernetes using configuration files that are, frankly, a little verbose at times. However, there are good reasons to prefer the configuration-based approach. The first is reproducibility. Let's say you need to reproduce the configuration on another environment like production and staging, which is a pretty common use case. With the declarative approach you can just apply the same exact config in the new environment (with any needed tweaks). Whereas if you went the imperative route, you would need to remember the commands, perhaps storing them in a batch file.

It's also harder to make changes. With configuration files, if you need to change a setting you can just update the configuration and re-apply it, after which Kubernetes will dutifully carry out your wishes. With a command-based approach, each change is itself a different command: `kubectl set image` to change the image, `kubectl scale` to change the number of replicas, and so on. You also run the risk that the command could fail, like might occur due to a network timeout, whereas with configuration the changes will be picked up the next time you apply them.

Chapter 11 covers taking configuration files and treating them just as you do the source code for your application, a so-called GitOps or configuration as code methodology where imperative commands would not be an option at all.

If you encounter a system previously built with imperative commands, fear not as configuration can be exported from the cluster with `kubectl get -o yaml $RESOURCE_TYPE $RESOURCE_NAME`. When exporting configuration like this from the live cluster, though, there are some extraneous fields you'll need to remove. Fortunately, though, it's never too late to switch, as whether you use declarative or imperative commands, Kubernetes is still storing the object in the same way.

3.4 Local Kubernetes Environments

This chapter so far has used a cloud-based Kubernetes provider as the deployment environment. You can of course run Kubernetes locally as well. I made the choice to lead with a public cloud provider instead of a local development cluster to demonstrate deploying on Kubernetes, as I assume for most the goal is to publish your service and make it accessible beyond your own machine. Indeed, if you're following the examples in this chapter in order, then congratulations: you can now deploy your apps to the world using Kubernetes! In future chapters you'll learn how to operationalize them, scale them up and more.

Local Kubernetes development clusters, however, definitely have their place. They are useful during development when you want to rapidly deploy and iterate on code while running in a Kubernetes cluster, particularly when your application consists of several different services. They're a great place to try out and learn Kubernetes constructs without paying for a cloud service and are a convenient option for testing your deployment configuration locally.

There are a lot of differences using Kubernetes locally on a machine in a non-production grade environment with a fixed set of resources compared to a production-grade cloud

service with dynamic provisioning. In the cloud you can scale up massively using multiple machines spread over a geographical region, while your local machine has a fixed set of resources. In the cloud you can get a production-grade routable public IP for your service; not so much on your local machine. Due to these differences and many more, I believe learning directly in your target product environment is more efficient, hence the focus in this book on production-grade clusters. That being said, as long as you understand the differences then a local development cluster can be a useful tool indeed.

Do you need a Kubernetes cluster for application development?

There's no requirement to use Kubernetes during application development just because you use it for production deployment. A fairly common app development pattern I've observed is using Docker Compose (covered in Section 2.3) for local development and testing, with the resulting application deployed to Kubernetes for production.

Docker Compose works pretty well for development of apps with only a handful of inter-service dependencies. The downside is you need to define the application config twice (once for development with Compose, once for production in Kubernetes), but this overhead is minor for apps with only a few service dependencies. The upside is that Docker has some useful tools for development, in particular, being able to mount local folders into the container, which means for interpreted languages like Python and Ruby you can change code without a container rebuild. It's also simple to configure since you can skip all the production-related config like replica count and resource requirements (Chapter 4).

It's hard to understate the usefulness of Compose being able to mount your local app folder as a read/write volume. Edit code without a container rebuild, get output from commands you run in the container like log files, and perform database upgrades right in your development folder. Kubernetes does have some tools like to level the playing field here, like Skaffold, which gets you a tight development loop with Kubernetes (local or cloud) as the target, but Docker has a sterling reputation among developers for a reason.

I always say, use the best tool for the job. Decide whether a local Kubernetes cluster or a Docker Compose setup works best for application development and use what works for you. Even if you choose to use Compose for application development, you may still utilize a local Kubernetes cluster for deployment testing.

There are a bunch of options for running a local Kubernetes cluster. The two most popular are *Docker Desktop* and *Minikube*. In fact, if you have *Docker Desktop* installed, then you already have a local single-node Kubernetes cluster! *Minikube*, created by the Kubernetes project, is also trivial to set up, and offers a few more advanced options like multiple nodes, useful when you want to test more advanced Kubernetes constructs like pod spread policies and affinity (Chapter 5).

3.4.1 Docker Desktop's Kubernetes Cluster

Docker Desktop comes with its own single-node Kubernetes development environment. If you have Docker Desktop installed, then you already have a local Kubernetes environment. Follow the instructions at <https://docs.docker.com/desktop/kubernetes/> to get going in two simple steps:

1. Enable Kubernetes in Docker Desktop options, and ensure it's running
2. Using `kubectl`, switch contexts to the Docker Desktop cluster

DIFFERENT FLAVORS OF DOCKER Be aware that Docker's local Kubernetes option is packaged with the "Docker Desktop" product. If you are using Docker via the Docker Engine install on Linux, it does not have this functionality.

Once Docker Desktop is running with Kubernetes enabled, you can view the context and switch to it like so:

```
kubectl config get-contexts
kubectl config use-context docker-desktop
```

In fact, you can use these commands to switch to any cluster that you previously connected to, including a cloud service like the one used previously in this chapter. Any time you wish to switch clusters, simply run:

```
kubectl config get-contexts
kubectl config use-context $CONTEXT
```

I find those two commands a bit tedious to type when switching between clusters a lot, so I highly recommend the `kubectx` tool (<https://github.com/ahmetb/kubectx>) which makes it a lot quicker. To switch contexts with `kubectx`:

```
kubectx
kubectx $CONTEXT
```

If you have any issues with Docker Desktop, then the "Restart Kubernetes Cluster" and "Clean / Purge data" options, found in the debug menu, are your friend.

3.4.2 Minikube

Minikube is another great choice for testing locally and allows you to test more Kubernetes functionality by providing a multi-node environment. It's maintained by the open-source Kubernetes community. Follow the instructions at <https://minikube.sigs.k8s.io/docs/start/> to install Minikube for your system.

Once installed, to boot a virtual multi-node cluster (which I recommend, as it more closely resembles a production Kubernetes environment), run `minikube start`, and pass the number of nodes you desire like so:

```
minikube start --nodes 3
```

The `start` command will automatically configure `kubectl` to use the Minikube context, meaning any `kubectl` commands will operate on the minikube cluster. To change the context back to a different cluster, like your production cluster, use the `kubectl config` or `kubectx` commands described in the previous section.

Once Minikube is running, you can go ahead and use it like a regular Kubernetes cluster following the instructions in this chapter. Before you start using it, to verify that things are running as expected, run `kubectl get nodes` to check that you can connect to the cluster.

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
minikube	Ready	control-plane	4m54s	v1.24.3
minikube-m02	Ready	<none>	4m32s	v1.24.3
minikube-m03	Ready	<none>	3m58s	v1.24.3

If you're done using Minikube and want to get your machine's CPU and memory resources back, run `minikube stop`. To delete all the data and allow you to create a new minikube cluster next time with different settings (like a different node count):

```
minikube delete
```

3.4.3 Using your Local Kubernetes Cluster

With `kubectl` set up to point to your preferred local Kubernetes cluster, you can deploy your application locally using the same `kubectl` commands shown earlier in this chapter. Two important differences, however, will be in how you expose and access services, and how you reference container images built locally.

To deploy the sample application from this chapter, from the sample root directory run:

```
$ cd Chapter03/3.2_DeployingToKubernetes
$ kubectl create -f .
deployment.apps/timeserver created
service/timeserver created
```

Benefit of declarative configuration

Throughout this book, the examples are given using declarative configuration rather than imperative commands. In other words, to create a deployment, we first create the configuration of the deployment, then apply it, as opposed to using `kubectl` to create the deployment directly.

One of the many benefits of this approach is it means you can test out your configuration locally, then deploy it confidently to production later, without needing to remember a bunch of one-off commands. Notice how we can deploy the same configuration files against the local cluster as we did against the production cluster. Neat!

ACCESSING THE SERVICE

Unlike when developing on a cloud Kubernetes provider, when creating a `LoadBalancer` service locally, you won't get an external IP.

For Docker Desktop, Minikube, and in fact any Kubernetes cluster, you can also use `kubectl` to forward ports from your local machine to the Service inside the cluster. This is useful for testing against a local Kubernetes cluster, and also debugging your cloud cluster. The following command exposes the Service locally:

```
kubectl port-forward service/$SERVICE_NAME $FROM_PORT:$TO_PORT
```

Where `FROM_PORT` is the port you'll access the service on locally, and `TO_PORT` is the IP of the Service. For our demo, choosing 8080 as a high level port, the command can look like:

```
kubectl port-forward service/timeserver 8080:80
```

You can then browse to <http://localhost:8080> to connect to the service. There are a range of useful flags⁴ for `port-forward`, including `--address 0.0.0.0` to bind to all network interfaces so you can access the forwarded service from other devices on the network (if your firewall allows it). Port forwarding is also useful to debug services running on a cloud Kubernetes service.

Minikube offers an additional way⁵ to route traffic to your Service. They can be accessed with:

```
minikube service $SERVICE_NAME
For the sample in the earlier section, that would be:
minikube service timeserver
```

Accessing Kubernetes Services locally from Docker

Are you running a Service in Kubernetes that you want to access directly from a Docker container running outside of a Kubernetes for some reason, like if you're doing some rapid iterating in Docker, and want to access an established Service in Kubernetes?

The solution is easy. Forward the service so that the port is open on your local machine as described above. You can then reference it in containers running directly in Docker using the host `host.docker.internal` on whatever port you forwarded. `host.docker.internal` is how containers can talk to services on the local machine, and since you forwarded the port to your local machine the connection can go through.

For example, if you deploy Redis in Kubernetes (see Chapter 9), and forward the ports like so: `kubectl port-forward service/timeserver 6379:6379`, and then want to connect to it from a local container in Docker running python, using the `redis-py` library, you can do that like so:

```
redis.Redis(host='host.docker.internal', port= '6379')
```

Happy coding!

DEPLOYING LOCAL IMAGES

By default, a local Kubernetes cluster will attempt to pull container images from the internet—behaving just like a production Kubernetes cluster. For public images like `ubuntu` or my sample image `docker.io/wdenniss/timeserver`, everything will just work. But for your own images built locally, you'll need to supply them to the local cluster. Of course you could upload them to a public container registry as you would for production, whereby your local cluster will pull them like in production.

Uploading every image you build during development, however, is a bit of a hassle. It slows down your development as you wait for the push and pull, and unless you're using public images, you'll need to provision credentials so your local cluster can access them (a step that is typically done for you when you're pulling private images from the container registry of your Kubernetes provider).

To get your local cluster to use a local image, you need to make two changes to your Kubernetes deployment configuration. Firstly, add the `imagePullPolicy` parameter, set to

⁴ <https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#port-forward>

⁵ <https://kubernetes.io/docs/setup/learning-environment/minikube/#services>

Never, and secondly refer to your image using its local image name without any repository prefix.

The path for locally built images is simply their repository and version tag, with no repository URL prefix. If you've built an image with `docker build . -t timeserver` as we did in Chapter 2, you would reference this in your Pod spec as `image: timeserver:latest` in your config file (using "latest" as the version tag will give us the most recently built image). Run `docker images` to view a list of available local images. Here's an example of a deployment referencing this locally built image:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 3
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: timeserver:latest #A
          imagePullPolicy: Never #B
```

#A Reference to a locally-available image

#B Image pull policy prevents Kubernetes from attempting to fetch this local image remotely

IMAGE PULL POLICY Only apply the `imagePullPolicy: Never` configuration to images you plan to provide locally. You don't want to set this on remote images, as they won't be pulled and will error with a `ErrImageNeverPull` status. If you see that error, it means the image isn't available locally yet the deployment was configured to use a local image.

There is one more step if you're using Minikube. While Docker Desktop has access to all the images you built locally with docker, Minikube does not (it has its own independent container runtime, and doesn't share images with your local install of docker). To push local images you want to use into Minikube. Simply run the following command:

```
minikube image load $REPOSITORY:$TAG
For example:
minikube image load timeserver:latest
Now then apply your changes with kubectl as before:
kubectl apply -f deploy.yaml service.yaml
```

3.5 Summary

- A Kubernetes cluster consists of a control plane and nodes on which your containers are run
- You interact with the container via the Kubernetes API, typically with the command line tool `kubectl`
- To deploy your own application to Kubernetes, first upload the container image to a container repository
- Workloads are specified using objects such as a Deployment, which encapsulates a Pod, which defines your containers
- Services are used to create network endpoints and expose containers to the internet
- Pods are referenced by other objects such as Deployments and Services with labels
- Kubernetes uses declarative configuration, typically YAML formatted configuration files
- You specify your requirements through configuration, and the Kubernetes controller seeks to actuate and fulfill them continuously
- Updating the application is as simple as modifying the configuration with the new container version and applying the change to the cluster
- Kubernetes will compare changes across configuration versions and actuate any changes specified

4

Automated Operations

This chapter covers

- Creating long-lasting, reliable application deployments
- Have Kubernetes keep your applications running without your intervention
- Updating applications without downtime

Kubernetes can automate many operations like restarting your container if it crashes, or migrating your application in the case of hardware failure, all which work to make your deployment more reliable without you needing to monitor it 24/7. These automated operations are one of the key value propositions of Kubernetes, and an essential step to taking full advantage of everything it has to offer.

Kubernetes can also help you update your application without outages and glitches by booting the new version of the application and monitoring its status to ensure it's ready to serve traffic, before removing the old version.

To help Kubernetes help keep your application running without downtime during normal operations and upgrades, you need to provide certain information about the state of your application with a process known as health checks. In the next section we'll go through adding the various health checks to your application, and in the later section, how these can be used with Kubernetes' in-built roll-out strategies to update your application without glitches or downtime.

4.1 Automated Uptime with Health Checks

There are some conditions that Kubernetes can detect and repair on its own. If your application crashes, Kubernetes will restart it automatically. Likewise, if the node running your container were to fail or be removed, Kubernetes will notice that your Deployment is missing replicas, and boot new replicas on available space in the cluster.

But what about other types of application failure like a hung process, a web service that stops accepting connections, or an application that depends on an external service when that service becomes inaccessible? Kubernetes can gracefully detect and attempt to recover from all these conditions, but it needs you to provide signals on the health of your application and whether or not it is ready to receive traffic. The process used to provide these signals are named health checks, which Kubernetes refers to as liveness and readiness probes.

Since Kubernetes can't know what it means for each and every service that runs on the platform to be down or up, ready or unready to receive traffic—apps must themselves implement this test. Simply put, the probe queries the container for its status, the container checks its own internal state, and returns a success code if everything is good. If the request times out (e.g. if the application is under too much load), or the container itself determines that there's a problem (such as with a critical dependency), the probe is considered a fail.

4.1.1 Liveness and Readiness Probes

In Kubernetes, the health of a container is determined by two separate probes: *liveness* that determines if the container is running, and *readiness* which indicates when the container is able to receive traffic. Both probes use the same techniques to perform the checks, but how Kubernetes uses the result of the probe is different.

	Liveness	Readiness
Semantic meaning	Is the container running?	Is the container ready to receive traffic?
Implication of probe failures exceeding threshold	Pod is terminated and replaced.	Pod is removed from receiving traffic until the probe passes.
Time to recover from failed probe	Slow; Pod is rescheduled on failure and needs time to boot.	Fast; Pod is already running and can immediately receive traffic once the probe passes.
Default state at container boot	Passing (Live).	Failing (Unready).

There are a few reasons for having the two probe types. One is the state at boot. Note how the Liveness probe starts in the passing, or live, state (container is assumed to be live, until the pod proves otherwise), whereas the Readiness probe starts in the unready state (container is assumed to not be able to serve traffic until it proves it can).

Without a readiness check, Kubernetes has no way to know when the container is ready to receive traffic, so it has to assume it's ready the moment the container starts up, and it will be added to the Service's load balancing rotation immediately. Most containers take tens of seconds, or even minutes to start up—so sending traffic right away would result in some traffic loss during startup. The readiness check solves this by only reporting "Ready" when the internal tests are passing.

Likewise, with a Liveness check, the conditions that require a container restart may be different to those which indicate the container is not ready to receive traffic. The best example is a container that is waiting for an external dependency, like a database connection. Until the container has the database connection, it should not be serving traffic (therefore is "Unready"), but internally the container is good to go. You don't want to replace

this container too hastily so that it has enough time to establish the database connection which it depends on.

Another reason for having two types of probes is the sensitivity and recovery times. Readiness checks are typically tuned to quickly remove the Pod from the load balancer (as this is a fast, and cheap operation to initiate) and add it back when the check is passing again, whereas Liveness checks are often tuned to be a little less hasty as the time needed to re-create a container is longer.

4.1.2 Adding a Readiness Probe

For a web service, a rudimentary health check could simply test “is the service serving traffic?” Before building a dedicated health check endpoint for your service, you could just find any endpoint on the service that returns a HTTP 200 status code, and use it as the health check.

If the root path returns HTTP 200 on all responses, you can even just use that. As it happens, the examples in the book do that, so the following readiness probe will work just fine.

Listing 4.1 Chapter04/4.1.2_Readiness/deploy.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 3
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:1
          readinessProbe:
            initialDelaySeconds: 15 #A
            periodSeconds: 30 #B
            httpGet: #C
              path: / #C
              port: 80 #C
              scheme: HTTP #C
            timeoutSeconds: 2 #D
            failureThreshold: 1 #E
            successThreshold: 1 #F
```

#A After an initial delay

#B Every 30 seconds

#C Perform this HTTP request

#D Timing out after 2 seconds

#E And consider 1 error response to indicate the container is not ready

#F And consider 1 successful response to indicate the container is ready after being considered unready

From the root directory, update the timeserver Deployment with:

```
cd Chapter04/4.1.2_Readiness
kubectl apply -f deploy.yaml
```

Now, any time the container fails to respond to the readiness check, that Pod will be temporarily removed from the service. Say you have 3 replicas of a Pod, and one of them fails to respond, then any traffic to the Service (which could be a service exposed externally, or a private one available only to other Pods in the cluster), will be routed to the remaining 2 healthy Pods. Once the Pod returns success (a HTTP 200 response in this case), it will be added back into service.

This is particularly important during updates, as you don't want Pods to be receiving traffic while they are booting (as these requests will fail). With correctly implemented readiness checks, you can get zero-downtime updates, as traffic is only routed to those Pods which are ready, and not ones in the process of being created.

Observing the Difference

If you want to see the difference between having a readiness check and not with your own experimentation, try the following test.

In one shell window, create a deployment without a readiness check (let's use the one from Chapter 3).

```
cd Chapter03/3.2_DeployingToKubernetes
kubectl create -f .
```

Wait for the service to be assigned an External IP:

```
kubectl get svc -w
```

Now setup a watch on the service endpoint in a separate console window:

```
watch -n 0.25 -d curl "http://[YOUR_IP]"
```

Back in the first window, trigger a rollout:

```
kubectl rollout restart -f deploy.yaml
```

As the pods restart, you should see some intermittent connection issues in the curl window

Now update the deployment with a readiness check (like the one in this section), and apply:

```
cd ../../Chapter04/4.1.2_Readiness
kubectl apply -f deploy.yaml
```

This time, since the deployment has a readiness check, you shouldn't see any connection issues on the curl window.

4.1.3 Adding a Liveness Probe

Liveness probes have the same specification as readiness, but are specified with the key `livenessProbe`. How the probes are *used* on the other hand is quite different. The result of

the readiness probe governs whether the Pod receives traffic, whereas a failing liveness probe will cause the Pod to be restarted (once the failure threshold is met).

The readiness check we added to our deployment in the previous section was rudimentary in that it just used the root path of the service rather than a dedicated endpoint. We can continue that practice for now, and use the same endpoint from the readiness probe for the liveness probe in the following example, with minor changes to increase the failure tolerance. Since the container gets restarted when the liveness probe fails the threshold, and can take some time to come back, we don't want the liveness probe setup on a hair trigger. Let's add a liveness probe to our deployment which will restart it if it fails for 180 seconds (6 failures at a 30s interval).

Listing 4.2 Liveness/deploy.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 3
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:1
          readinessProbe:
            initialDelaySeconds: 15
            periodSeconds: 30
            httpGet:
              path: /
              port: 80
              scheme: HTTP
            timeoutSeconds: 2
            failureThreshold: 1
            successThreshold: 1
          livenessProbe: #A
            initialDelaySeconds: 30 #B
            periodSeconds: 30 #C
            httpGet: #D
              path: / #D
              port: 80 #D
              scheme: HTTP #D
            timeoutSeconds: 5 #E
            failureThreshold: 10 #F
            successThreshold: 1 #G
```

#A Specify a liveness probe this time
 #B After an initial delay of 30 seconds
 #C Every 30 seconds
 #D Perform this HTTP request

```
#E Timing out after 5 seconds (more tolerant than the readiness check)
#F And consider 10 error responses in a row to indicate the container is not ready
#G And consider 1 successful response to indicate the container is ready after being considered unready
```

Update the timeserver Deployment with these latest changes:

```
cd Chapter04/4.1.3_Liveness
kubectl apply -f deploy.yaml
```

Now, your Deployment has a readiness and liveness probe. Even these rudimentary probes improve the reliability of your deployment drastically, and if you stop here it's probably enough for a hobby application. The next section details some further design considerations to bulletproof your probes for production use.

4.1.4 Designing Good Health Checks

While using an existing endpoint as we did in the previous two sections as the health check path is better than nothing, it's generally better to add dedicated health check endpoints to your application. These health checks should implement the specific semantics of readiness and liveness, and be as lightweight as possible. Without understanding the semantic differences between liveness and readiness you could see instability due to excessive restarts and cascading failures, and if you're re-using some other endpoint, chances are it's heavier-weight than needed—why pay the cost of rendering an entire HTML page when a simple HTTP header response would suffice?

When creating the HTTP endpoints to implement these checks, it's important to take into account any external dependencies being tested. Generally, you don't want external dependencies to be checked in the liveness probe, rather it should test only whether the container itself is running (assuming your container will retry the connections to its external connections). This is because there's not really any value in restarting a container that's running just fine, and only because it can't connect to another service which is having trouble. This could cause unnecessary restarts that creates churn and could lead to cascading failures, particularly if you have a complex dependency graph. There is an exception to this principle of not testing dependencies in liveness probes which I cover later in the section.

Since the liveness is only testing whether or not the server is responding, the result can and should be extremely simple, generally just a HTTP 200 status response, even one with no response body text. If the request can get through to the server code, then it must be "live", and this is good enough.

For Readiness probes on the other hand, it's generally desirable that they test their external dependencies (like a database connection). This is useful because if you have say 3 replicas of a Pod, and only 2 of them can connect to your database, it makes sense to only have those 2 fully functional pods in the load balancer rotation. One way to test the connection is to lookup a single row from the database in your readiness check.

Is pseudocode such a database connection check could look something like:

```
result = sql.execute("SELECT id FROM users LIMIT 1;")
if result:
    http_response(200, "Ready")
else:
    http_response(503, "Not Ready")
```

Performing a simple SQL query should be enough to ensure that the database is both connected, and responsive. Rather than using a SELECT query, you could perform any other database operation, but I personally like the legitimacy of a SELECT statement as if this works, then I'm confident the other queries will work too.

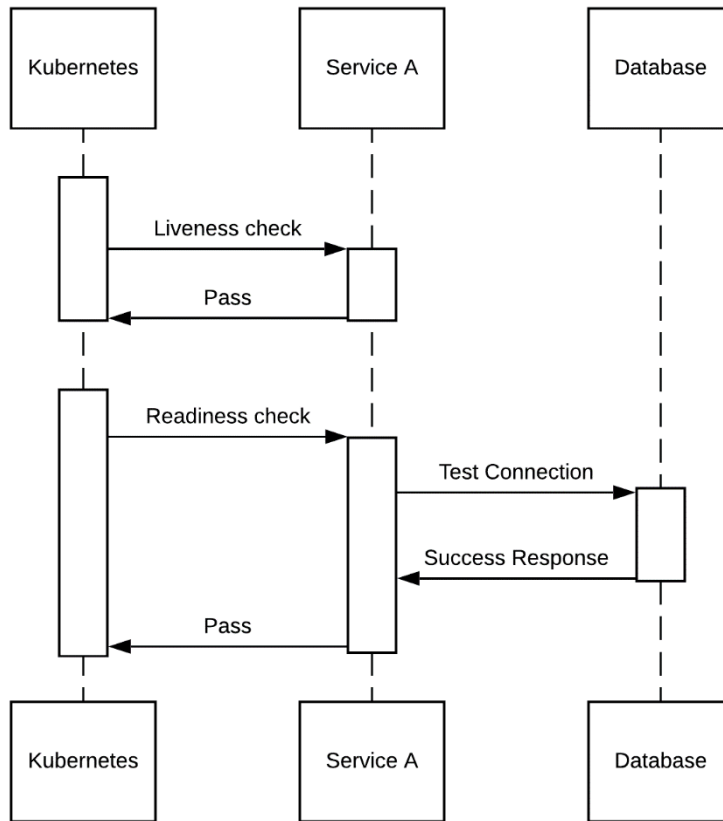


Figure 4.1 Liveness and Readiness checks and external dependencies

The python timeserver example app doesn't have a Database dependency, but let's refactor the code to include specific `/healthz` and `/readyz` paths as is best practice to have dedicated endpoints for these probes.

Listing 4.3 timeserver2/server.py

```

from http.server import ThreadingHTTPServer, BaseHTTPRequestHandler
from datetime import datetime

class RequestHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        match self.path:
            case '/':
                now = datetime.now()
                response_string = now.strftime("The time is %-I:%M %p, UTC.")
                self.respond_with(200, response_string)
            case '/healthz':
                self.respond_with(200, "Healthy")
            case '/readyz':
                dependencies_connected = True
                # TODO: actually verify any dependencies
                if (dependencies_connected):
                    self.respond_with(200, "Ready")
                else:
                    self.respond_with(503, "Not Ready")
            case _:
                self.respond_with(404, "Not Found")

    def respond_with(self, status_code: int, content: str) -> None:
        self.send_response(status_code)
        self.send_header('Content-type', 'text/plain')
        self.end_headers()
        self.wfile.write(bytes(content, "utf-8"))

def startServer():
    try:
        server = ThreadingHTTPServer(('', 80), RequestHandler)
        print("Listening on " + ":".join(map(str, server.server_address)))
        server.serve_forever()
    except KeyboardInterrupt:
        server.shutdown()

if __name__ == "__main__":
    startServer()

```

Updating our Deployment configuration for these new endpoints, we get:

Listing 4.4 GoodHealthChecks/deploy.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 3
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:2
          readinessProbe:
            initialDelaySeconds: 15
            periodSeconds: 30
            httpGet:
              path: /readyz #A
              port: 80
              scheme: HTTP
            timeoutSeconds: 2
            failureThreshold: 1
            successThreshold: 1
          livenessProbe:
            initialDelaySeconds: 30
            periodSeconds: 30
            httpGet:
              path: /healthz #A
              port: 80
              scheme: HTTP
            timeoutSeconds: 3
            failureThreshold: 3 #B
            successThreshold: 1
```

#A Updated endpoints

#B Now the liveness probe is lightweight, we can tighten up the failure threshold

Your own application may have more complex readiness and liveness logic. While the healthz endpoint here probably works for many HTTP applications (simply testing that the HTTP server is responding to requests is sufficient), every application with dependencies like databases should define their own readiness test to determine if your application is truly ready to serve user requests.

4.1.5 Rescheduling Unready Containers

The previous section detailed the standard way to setup liveness and readiness checks in Kubernetes, and only verifying service dependencies in the readiness check. There is one problematic condition by not testing dependencies in the liveness check. By separating the

concerns into readiness: “is the container ready to receive traffic” and Liveness: “is the container running”, there could be a condition where the container is running, but due to a bug in the container’s retry logic, the external connections will never be resolved. In other words, your container could be stay unready forever, something that a restart might resolve.

Recall that we don’t test readiness in the liveness check, as this could cause the Pod to be recreated too quickly and not give any time for the external dependencies to resolve, but it might still make sense to have this Pod be recreated if it stays unready for too long. Sometimes it’s best just to turn it off and on again.

Unfortunately, Kubernetes doesn’t have a way to express this logic, but it’s easy enough to add it to our own Liveness check so that it will fail if the Pod doesn’t come ready in a certain time. You can simply record the time of each Readiness success response, and then fail your liveness check if too much time has passed (for example, 5 minutes).

Here is a simple implementation of this logic into the timeserver container:

Listing 4.5 timeserver3/server.py

```
from http.server import ThreadingHTTPServer, BaseHTTPRequestHandler
from datetime import datetime, timedelta

last_ready_time = datetime.now() #A

class RequestHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        global last_ready_time

        match self.path:
            case '/':
                now = datetime.now()
                response_string = now.strftime("The time is %-I:%M %p, UTC.")
                self.respond_with(200, response_string)
            case '/healthz':
                if (datetime.now() > last_ready_time + timedelta(minutes=5)): #B
                    self.respond_with(200, "Not Healthy") #B
                else: #B
                    self.respond_with(200, "Healthy") #B
            case '/readyz':
                dependencies_connected = True
                # TODO: actually verify any dependencies
                if (dependencies_connected):
                    last_ready_time = datetime.now() #C
                    self.respond_with(200, "Ready")
                else:
                    self.respond_with(503, "Not Ready")
            case _:
                self.respond_with(404, "Not Found")

    def respond_with(self, status_code: int, content: str) -> None:
        self.send_response(status_code)
        self.send_header('Content-type', 'text/plain')
        self.end_headers()
        self.wfile.write(bytes(content, "utf-8"))

def startServer():
    try:
```

```

server = ThreadingHTTPServer(('', 80), RequestHandler)
print("Listening on " + ":".join(map(str, server.server_address)))
server.serve_forever()
except KeyboardInterrupt:
    server.shutdown()

if __name__ == "__main__":
    startServer()

```

#A The last ready time is initialized at the current time to allow for 5 minutes since startup

#B Each time the readiness passes, the time is updated

#C If 5 minutes have passed since the last successful readiness result (or since boot), fail the liveness check.

Having the liveness check fail eventually if the container never becomes ready gives it a chance to restart. Now we have the best of both worlds, we don't test the external dependencies in every liveness check, but we do in the readiness one. That means our container won't receive traffic when its dependencies are not connected, but it's not rebooted either, giving it some time to self-heal. But if, after 5 minutes, the container is still not Ready, it will fail the liveness and be restarted.

An alternative approach to achieve this (restarting the container after a prolonged period of unreadiness) is to use the readiness endpoint for both Liveness and Readiness probes, but with different tolerances (e.g. the Readiness would fail after 30 seconds, but Liveness only after 5 minutes). This approach still gives the container some time to resolve any inter-dependent services, before eventually rebooting in the event of continued downtime which may indicate a problem with the container itself. This is not technically idiomatic Kubernetes, as you're still testing dependencies in the Liveness check, but it gets the job done.

In conclusion, these two probes are incredibly important to giving Kubernetes the information it needs to automate the reliability of your application. Understanding the difference between them, and implementing appropriate checks that take into account the specific details of your application is crucial.

4.1.6 Probe Types

To now, the examples have assumed a HTTP service, and the probes therefore were implemented as HTTP requests. Kubernetes can be used to host many different types of services, as well as batch jobs with no service endpoints at all. Fortunately, there are a number of ways to expose health checks:

HTTP. Recommended for any container that provides a HTTP service, the service exposes an endpoint, such as `/healthz`. A HTTP 200 response indicates success, any other response (or timeout) indicates a failure.

TCP. Recommended for TCP-based services other than HTTP (for example, a SMTP service). The probe succeeds if the connection can be opened.

```

readinessProbe:
  initialDelaySeconds: 15
  periodSeconds: 30
  tcpSocket: #A
    port: 25 #A
  successThreshold: 1
  failureThreshold: 1

```

#A TCP probe specification

Bash Script. Recommended for any container not providing a HTTP or TCP service such as batch jobs that don't run service endpoints. Kubernetes will execute the script you specify, allowing you to perform whatever tests you need. A non-zero exit code indicates failure.

4.2 Updating Live Applications

Once you've implemented Readiness checks, you can now roll out changes to your application without downtime. Kubernetes uses the Readiness check during updates to know when the new Pod is ready to receive traffic, and to govern the rate of the rollout according to parameters you set. There are several different rollout strategies to choose from, each with their own characteristics.

4.2.1 Rolling Update Strategy

The default zero-downtime update strategy offered by Kubernetes is a "rolling update". In a rolling update, Pods with the new version are created in groups (the size of which is tunable). Kubernetes waits for the new group of Pods to become available, then terminates the same number of Pods running the old version, repeating this until all Pods are running the new version.

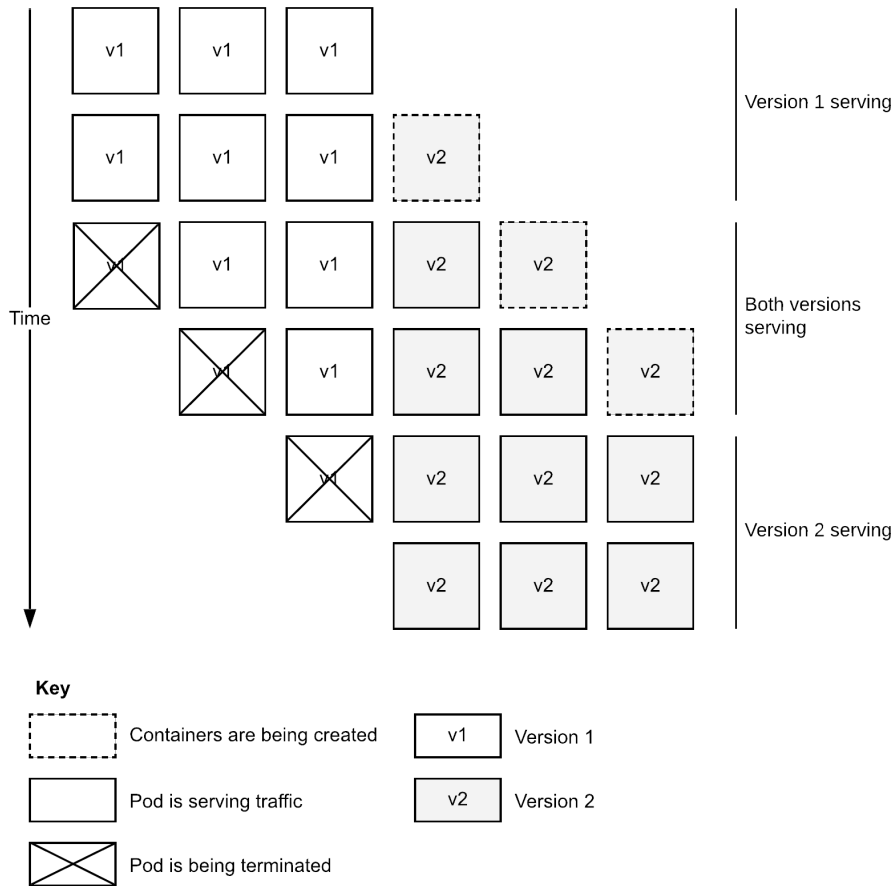


Figure 4.2 Pod status during a rolling update. With this strategy, requests can be served by either the old or the new version of the app until the rollout is complete.

The goal of such a strategy is two-fold:

- Provide continuous uptime during the rollout
- Use as few extra resources as possible during the update

Importantly, with this strategy the two versions of your application (old, and new) need to be able to co-exist as they will both be running for a time. That is, your backend or any other dependencies must be able to handle these two different versions, and that users may get alternating versions when they make different request. Imagine reloading the page and seeing the new version, reloading it and seeing the old version again. Depending on how many replicas you have, a rolling update can take a while to complete (and therefore, any rollback can also take a while).

Let's configure our deployment to use the rolling update strategy:

Listing 4.6 RollingUpdate/deploy.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 3
  selector:
    matchLabels:
      pod: timeserver-pod
  strategy:
    type: RollingUpdate #A
    rollingUpdate: #B
    maxSurge: 2 #B
    maxUnavailable: 1 #B
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:1
```

#A rolling update strategy

#B optional configuration

The options `maxSurge` and `maxUnavailable` can be used to govern how quickly the rollout happens.

MAXSURGE

"maxSurge" governs how many more Pods you're willing to create during the rollout. For example, if you set a replica count of 5, and a `maxSurge` of 2, then it may be possible to have 7 Pods (of different versions) scheduled.

The tradeoff is that the higher this number is, the faster the rollout will complete, but the more resources it will (temporarily) use. If you're highly optimizing your costs, you could set this to 0. Alternatively, for a large deployment you could temporarily increase the resources available in your cluster during the rollout by adding nodes, and removing them when the rollout is complete.

MAXUNAVAILABLE

"maxUnavailable" sets the maximum number of Pods that can be unavailable during updates (percentage values are also accepted, and are rounded down to the nearest integer). If you've tuned your replica count to handle your expected traffic, you may not want to set this value much higher than zero, as your service quality could degrade during updates.

The trade off here is that the higher the value, the more Pods can be replaced at once and the faster the rollout completes, while reducing the number of Ready pods temporarily that are able to process traffic.

Given that a rollout could coincide with another event that lowers availability like a node failure, for production workloads I would recommend setting this to 0. The caveat is that if you set it to 0, and your cluster has no schedulable resources, the rollout will get stuck and you will see Pods in the “Pending” state until resources become available. When the `maxUnavailable` is 0, `maxSurge` cannot also be zero (the system needs to surge, i.e. add create new pods, that are by definition not ready as they are booting).

RECOMMENDATION

Rolling update is a good go-to strategy for most services. For production services, `maxUnavailable` is best set to 0. `maxSurge` should be at least 1, or higher if you have enough spare capacity and want faster rollouts.

DEPLOYING CHANGES WITH ROLLING UPDATE

Once your deployment is configured to use Rolling Update, deploying your changes is as simple as updating the deployment manifest, for example with a new container version, and applying the changes with `kubectl apply`. Most changes made to the Deployment including Readiness and Liveness checks are also versioned, and will be rolled out just like a new container image version.

4.2.2 Replacement Strategy

Another approach, some might say the old fashion approach, is to cut the application over directly—delete all Pods of the old version, and schedule replacements of the new version. Unlike the other strategies discussed here, this is *not* zero-downtime. It will almost certainly result in some unavailability. With the right readiness checks in place, this downtime could be as short as the time to boot the first Pod, assuming it can handle the client traffic at that moment in time.

The benefit of this strategy is does not require compatibility between the new version and the old version (since the two versions won’t be running at the same time), nor does it require any additional compute capacity at all (since it’s a direct replacement).

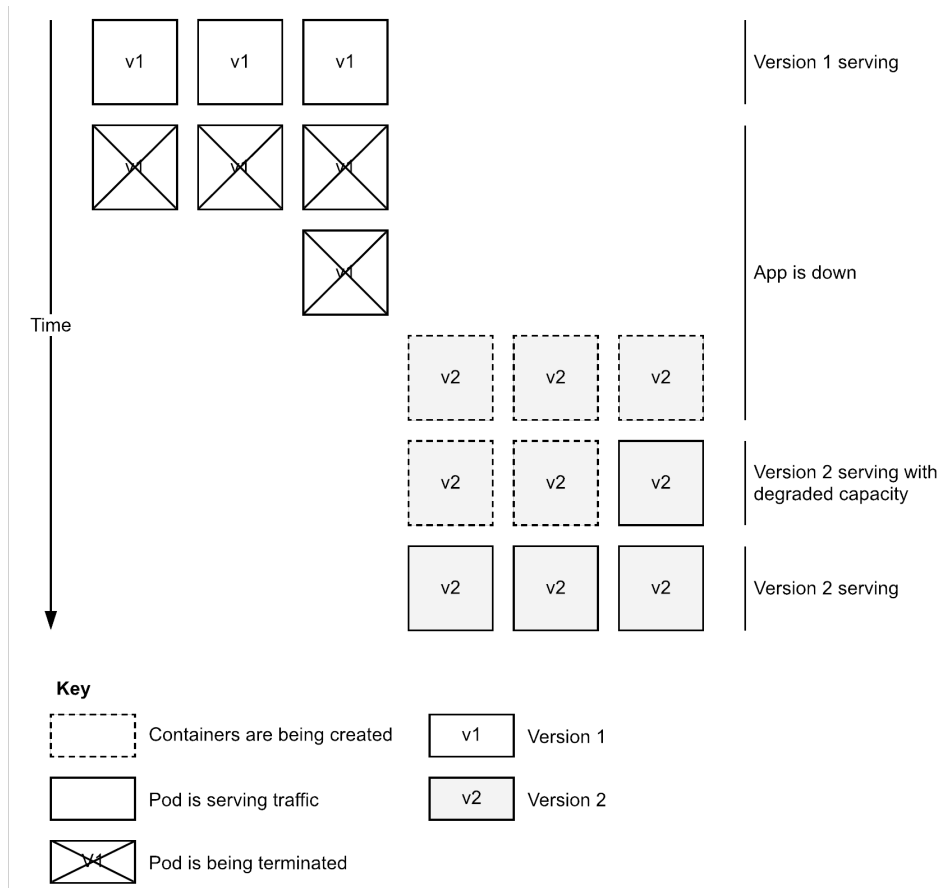


Figure 4.3 pod status during a rollout with the replacement strategy. During this type of rollout, the app will experience a period of total downtime, and a period of degraded capacity.

For development and staging, this strategy may be useful to avoid needing to slightly overprovision compute capacity to handle rolling updates, and for its speed, but otherwise should generally be avoided.

```
strategy:
  type: Recreate
```

Changes to the Deployment manifest can be applied as with Rolling Update, using `kubectl apply`.

4.2.3 Blue / Green Strategy

The Blue/Green strategy is a rollout strategy where the new application version is deployed alongside the existing version. These versions are given the names “blue” and “green”. When the new version is fully deployed, tested, and ready to go – the service is cut over. If there’s a problem, it can be immediately cut back, and after a time if everything looks good, the old version can be removed. Unlike the prior two strategies, the old version remains ready to serve and is removed only when the new version is validated (and often with a human decision involved).

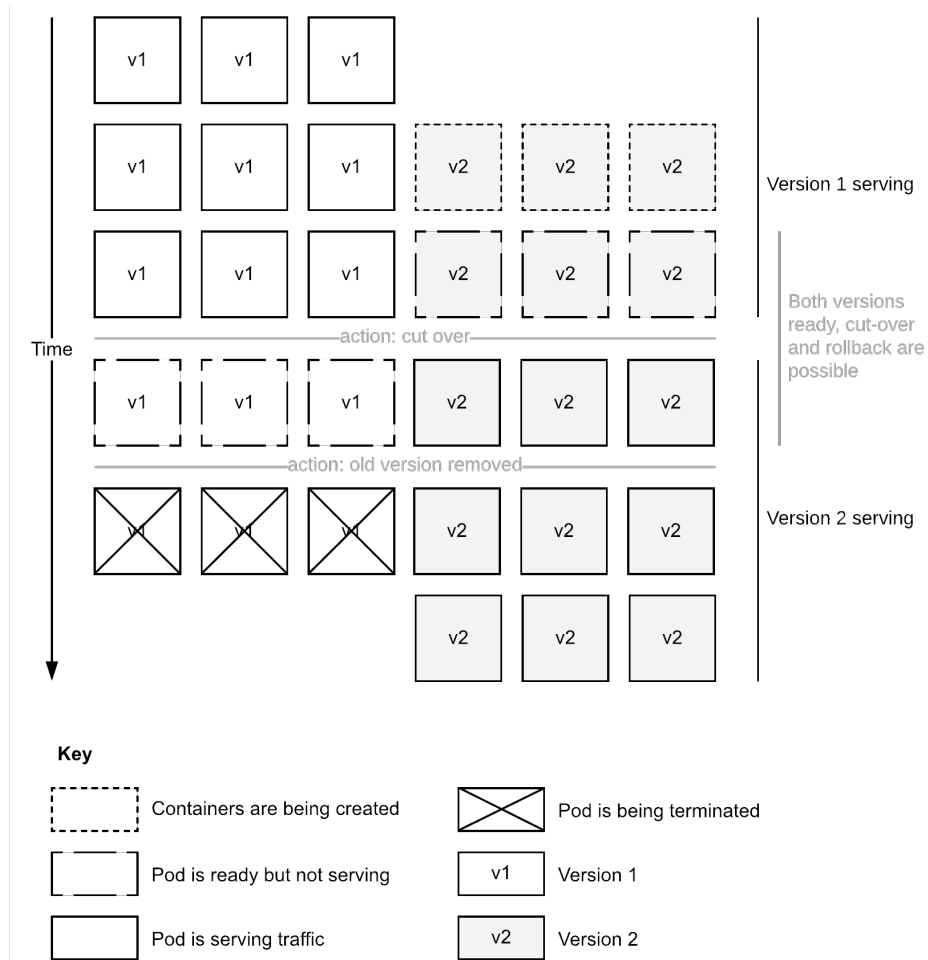


Figure 4.4 pod status during a blue / green rollout. Unlike the previous strategies, there are two action points where other systems, potentially including human actors make decisions.

The benefits of this strategy is:

- Only one version of the app is running at a time, for a consistent user experience
- The rollout is fast (within seconds)
- Rollbacks are similarly fast

The drawbacks are:

- Temporarily consumes double the compute resources
- Not supported directly by Kubernetes Deployments.

This is an advanced rollout strategy, popular with large deployments. There are often several other processes included. For example, when the new version is ready—it can be tested first by a set of internal users, followed by a percentage of external traffic prior to the 100% cut-over—a process known as canary analysis. After the cut-over, there is often a period of time where the new version continues to be evaluated, prior to the old version being scaled down (this could last days). Of course, keeping both versions scaled up doubles the resource usage, with the trade off that near-instant rollbacks are possible during that window.

Unlike the prior two strategies—rolling update and replace—there is no native Kubernetes support for blue green. Typically users will use additional tooling to help with the complexities of such a rollout, native ones for Kubernetes include Istio for being able to split traffic at a fine-grained level, and Spinnaker to help automate the deployment pipeline with the canary analysis and decision points. If you prefer to do a blue green style deployment without some of the fancier features like canary analysis provided by such tooling, then you can perform a blue / green rollouts directly in Kubernetes using multiple deployments.

IMPLEMENTING BLUE/GREEN IN KUBERNETES

Despite the lack of native support, it is possible to perform a blue/green rollout in Kubernetes. Without the aforementioned tools to help with the pipeline and traffic splitting, it is a slightly manual process, and missing some benefits like being able to do canary analysis on a tiny percentage of production traffic, but that doesn't mean it's hard to implement.

Recall the Deployment and a Service we deployed in Chapter 3. Employing a blue green strategy for this application simply requires having one extra deployment. Duplicate the deployment, and suffix both deployment's *filename*, *metadata name*, and the Pod template's *labels* with "-blue", and "-green" respectively. You can then direct traffic from your Service by selecting either the "-blue", or the "-green" labels.

The update strategy you would specify in the Deployment configuration in this case is the `Recreate` strategy. Since only the Pods in the inactive deployment are updated, deleting all the old version and creating Pods with the new version won't result in downtime, and is faster than a rolling update.

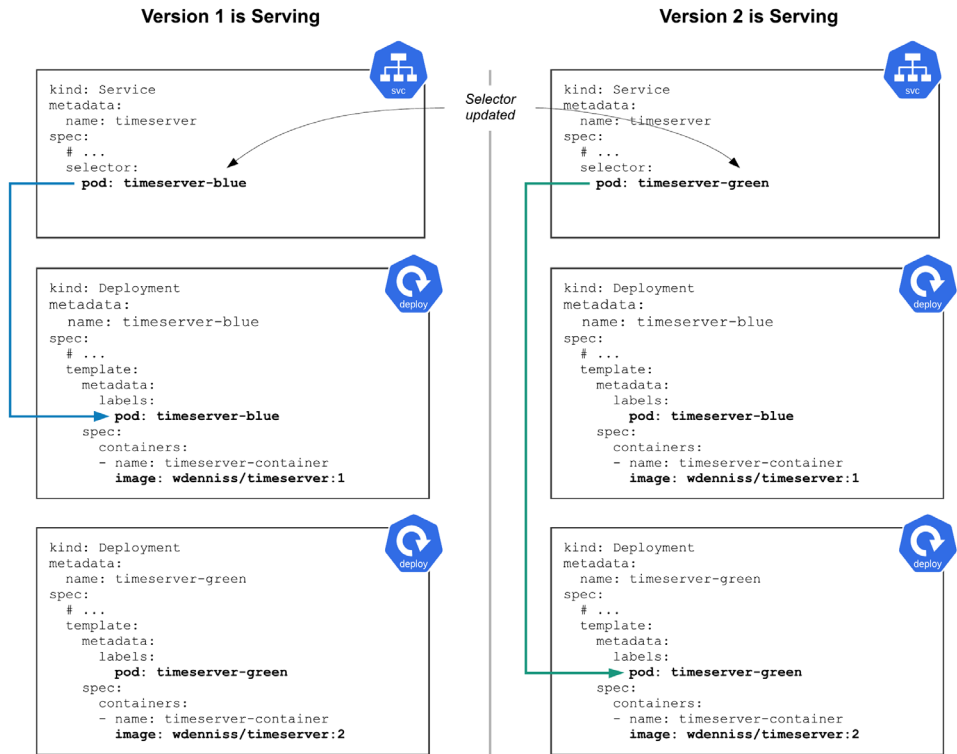


Figure 4.5 One service with a “blue” and “green” deployment with different versions of the same application. The service’s selector is used to route traffic to the live version.

In this two-deployment system, one version is live, and one is non-live at any given time. The Service is selecting the live deployment with the label selector.

The steps to rollout a new version with blue green are the following:

1. Identify the non-live deployment (the one not selected by the Service)
2. Update the image path of the non-live deployment with the new container image version
3. Wait until the Deployment is fully rolled out (`kubectl get deploy`)
4. Update the Service’s selector to point to the new version’s Pod template’s labels

The update steps are performed by modifying the YAML configuration for the resource in question, and applying the changes with `kubectl apply`.

The next time you want to rollout a change to this application, the steps are the same, but the colors are reversed (if blue was live for the last update, green will be live next time).

As mentioned, this strategy doubles the number of pods used by the deployment which will likely impact your resource usage. To minimize resource costs, you can scale the non-live

deployment to zero when you're not currently doing a rollout, scaling it back up to match the live version when you're about to do a rollout. You'll likely need to adjust the number of nodes in your cluster as well. See Chapter 5 for how to perform these steps.

4.2.4 Choosing a Rollout Strategy

For most deployments, one of the built-in rollout strategies should suffice. Use `RollingUpdate` as an easy way to get zero-downtime updates on Kubernetes. To achieve the zero downtime or disruption, you will also need to have a readiness check implemented otherwise traffic can be sent to your container before it has fully booted. You need to consider that two versions of your application can be serving traffic simultaneously and design attributes like data formats with that in mind. Being able to support at least the current and previous version is good practice generally anyway, as it also allows you to rollback to the previous version if something goes wrong.

`Replace` is a useful strategy when you really don't want two application versions running at the same time, and things like legacy single-instance services where only one copy can exist at a time.

Blue/green is an advance level strategy that requires additional tooling or processes, but with the advantage for near-instant cut-overs while offering the best of both worlds in that only one version is live at a time, but without the downtime of the `Replace` strategy. I recommend getting started with the in-built strategies but keep this one in mind for when you need something more.

4.3 Summary

- Kubernetes provides many tools to help you keep your deployments running and updated
- It's important to define health checks so that Kubernetes has the signals it needs to keep your application running by rebooting containers that have gotten stuck or that are non-responsive.
- Liveness probes are used by Kubernetes to know when your application needs restarting
- The readiness probe governs which replicas receive traffic, which is particularly important during updates to prevent dropped packets
- Kubernetes can also help update your application without downtime
- `RollingUpdate` is the default rollout strategy in Kubernetes, giving you a zero-downtime rollout while using minimal additional resources
- `Recreate` is an alternative rollout strategy that does an in-place update with some downtime but no additional resource usage
- Blue/green is a rollout strategy that isn't directly supported by Kubernetes but can still be performed using native Kubernetes constructs.
- Blue/green offers some of the highest quality guarantees, but is more complex, and temporarily doubles the resources needed by the deployment.

5

Resource Management

This chapter covers

- **How Kubernetes allocates the resources in your cluster**
- **Configuring your workload to request just the resources it needs**
- **Overcommitting resources to improve your performance-to-cost ratio**
- **Balancing Pod replica count with internal concurrency**

Chapter 2 covered how Containers are the new level of isolation each with their own resources, and Chapter 3 that the schedulable unit in Kubernetes is a Pod (which itself is a collection of containers). This chapter covers how Pods are allocated to machines based on their resource requirements, as well as the information that you need to give the system so that your Pod will receive the resources that it needs. Knowing how Pods are allocated to Nodes helps you make better architectural decisions around resource requests, bursting, overcommit, availability, and reliability.

5.1 Pod Scheduling

The Kubernetes scheduler performs a resource-based allocation of Pods to Nodes, and is really the brains of the whole system. When you submit your configuration to Kubernetes (as we did in Chapter 3 and 4), it's the scheduler that does the heavy lifting of finding a Node in your cluster with enough resources, and tasks the Node with booting and running the containers in your Pods.

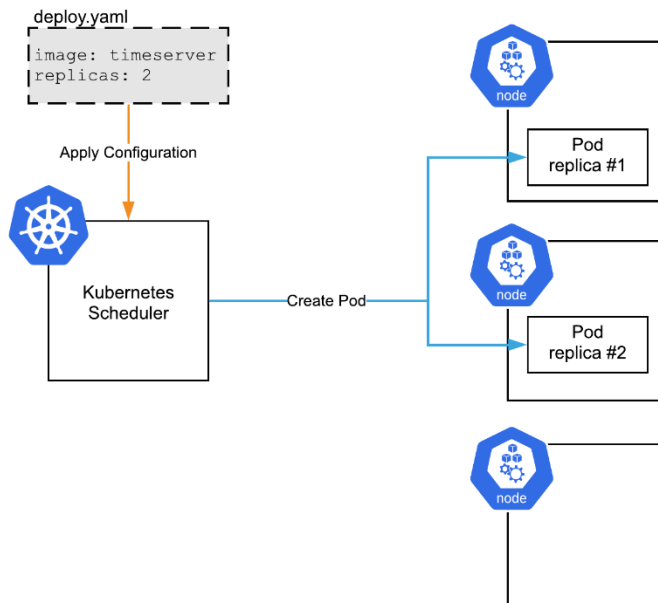


Figure 5.1 In response to the user applying configuration, the scheduler issues a Create Pod command to the node.

The scheduler's work doesn't stop there either. In the case of the Deployment object (what we've been using in the book so far), it continuously monitors the system with the goal to make the system state what you requested it to be. In other words, if your deployment requested 2 replicas of your Pod, the scheduler doesn't just create those replicas then forget about it, it will keep verifying that there are still 2 replicas running. If something were to happen (for example, say a node disappeared due to some failure), it would attempt to find a new place to schedule the Pod so that your desired state (2 replicas in this case) is still met.

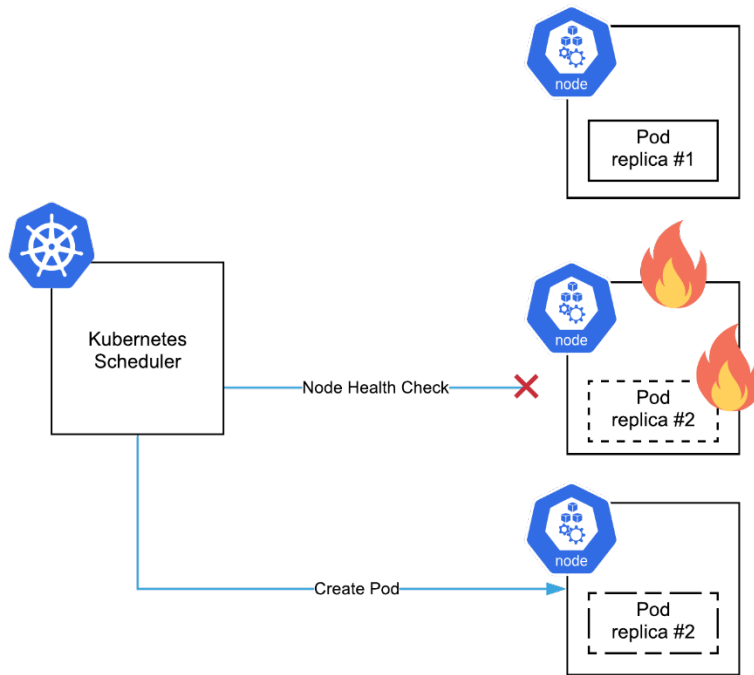


Figure 5.2 One of the nodes develops an issue. Health checks from the Kubernetes control plane fail, so the Scheduler issues a Create Pod command to a healthy node.

This recreation of Pods due to node failures by the scheduler is separate behavior to the Pod restarts we covered in the last chapter. Pod restarts due to liveness or readiness failures are handled locally on the node by the kubelet, whereas the Scheduler is responsible for monitoring the health of the nodes, and reallocating Pods when issues are detected.

Since each Node in the cluster is constrained by the resources that it has, and Pods may have differing resource requirements themselves, an important responsibility of the scheduler is finding the enough room to run your Pods. It considers multiple scheduling dimensions when considering where to place the containers of your Pod in the cluster whether for the first time they're deployed, or in response to disruption like the one illustrated in Figure 5.2.

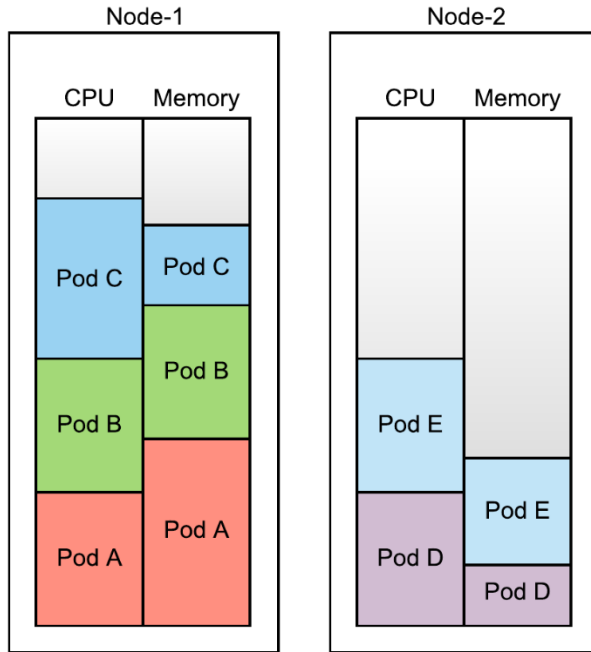


Figure 5.3 5 containers allocated on 2 nodes based on their resource needs

It is the scheduler that has the task of finding the right place in your cluster to fit the Pod, based on its resource requirements and (as we'll cover later in this chapter) any other placement requirements. Any Pods that can't be placed on the cluster will have the status "Pending" (see "Stuck in Pending" in Chapter 3 if you have Pods that remain in this status for too long).

5.1.1 Specifying Pod Resources

You give the scheduler the information it needs to make scheduling decisions by specifying the resource requests in your deployment manifest (and other workload types that have an embedded Pod specification). So far, the examples in this book have not specified their resource requirements, but for production grade deployments this needs to be added.

A Pod that needs 20% of the time for one CPU core, and 200MiB of memory would be specified like so:

Listing 5.1 PodResources/A_deploy_with_requests.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 3
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:1
          resources:
            requests: #A
              cpu: 200m #A
              memory: 250Mi #A

```

#A The resource requests of this deployment

The “200m” in the example here represents 200 milli-cores, that is, 20% of 1 core. You can also use floating point numbers, e.g. “0.2”, however it’s very common among Kubernetes practitioners to use milli-cores. The “Mi” suffix for memory indicates Mebibytes, with “Gi” indicating Gibibyte (powers of 1024), while “M” and “G” indicate Megabyte and Gigabyte (powers of 1000).

These values are extremely important, as it gives Kubernetes the information it needs to match the Pod requirements to the node capacity. Say you have 3 Pods that have really high requirements, and 3 with low requirements, you would want Kubernetes to ensure each Pod has the resources it needs, and not place them randomly without regarding their needs (for example, by placing the 3 high requirement pods together on a node where they would each be starved of resources).

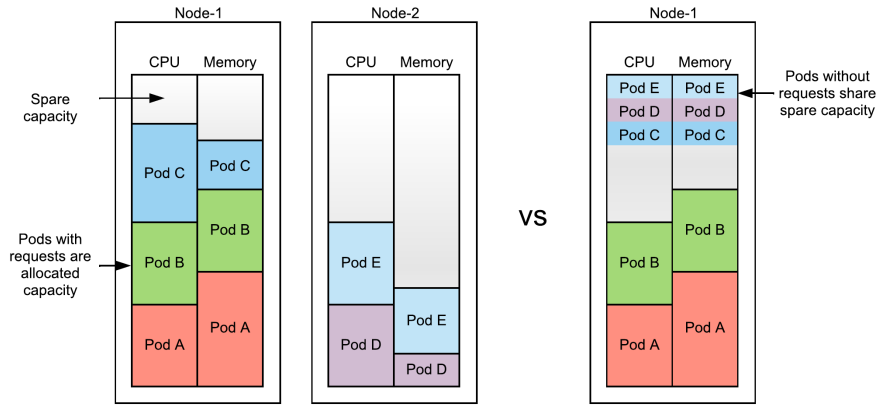


Figure 5.4 Comparison of Pod allocation when all Pods have resource requests, and when only some do. Pods without resource requests share the spare capacity on a best effort basis, without regards to their actual needs.

This may sound fairly simple so far – we’re just pairing the requests with the resources. It would be simple too, if it were not for the ability to *burst*, that is, consume more resources than you requested. Much of the time a process may not need all the resources it asked for, wouldn’t it be good if the other Pods on the node could use that capacity on a temporary basis?

Kubernetes represents this burst capacity with *limits*. A Pod declares both the resources it requests which is used for scheduling, and the limits which is used for constraining the resources being used.

Listing 5.2 PodResources/B_deploy_with_requests_limits.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 3
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:1
          resources:
            requests:
              cpu: 200m
              memory: 250Mi
            limits: #A
              cpu: 300m #A
              memory: 400Mi #A

```

#A The resource limits of this deployment, the pod can use up to 0.3 of a CPU core, and 400 MiB memory

The sample configuration above specifies both requests and limits. It is best practice to have both requests and limits set for all your deployments. But how do you determine what to set them at? Read on to understand how the requests and limits interact to form a quality of service class, and how you can measure your application's performance to determine what values to set.

5.1.2 Quality of Service

Having limits higher than requests, or not set at all, introduces a new problem, and that is what to do when all these Pods are consuming too many resources (most commonly, too much memory), and they need to be evicted to reclaim the resource. Kubernetes performs a stack rank of importance when choosing which pods to remove first.

Kubernetes traditionally classified Pods into 3 quality of service classes in order to rank their priority. These are no longer directly used in the eviction algorithm, but I'll present them here as there are a lot of people and documentation that use the terms, and it's still a reasonable way to consider the quality of service a Pod will receive.

GUARANTEED CLASS

Guaranteed class pods are where the limits are set equal to the requests. This is the most stable configuration as the pod is guaranteed the resources it requested, no more, no less. If your Pod has multiple containers, they all must meet this requirement for the Pod to be considered Guaranteed.

Guaranteed class Pods are always guaranteed to have the same resources available under varying conditions, and they won't be evicted from the node as it's not possible for them to use more resources than they were scheduled for.

BURSTABLE CLASS

Burstable class Pods on the other hand have limits set higher than requests, and are able to "burst" temporarily provided there are resources available (i.e. from other Pods not using all of their requests, or unallocated space on the node).

You need to be careful with these Pods as there can be some unforeseen consequences such as accidentally relying on the bursting. Say a Pod lands on an empty node and can burst to its heart's content. Then sometime later, it gets rescheduled onto another node with less resources, the performance will now be different. So, it's important to test burstable pods in a variety of conditions.

A Pod with multiple containers is considered Burstable if it doesn't meet the criteria for Guaranteed, and if any of the containers has a request set.

BEST EFFORT

Pods without any requests or limits set are considered "best effort" and are scheduled wherever Kubernetes wishes. This is the lowest of the classes, and I strongly recommend against using this pattern. You can achieve a similar result with the burstable class by setting really low requests, and that is more explicit than just closing your eyes and hoping for the best.

When thinking about the stability for your pods, it's always best to at least set the resource requests to a high enough value that gives them enough resources to run, and avoid not setting any resource requests at all. High priority, critical workloads should always have limits set to their requests for guaranteed performance.

5.1.3 Evictions, Priority and Preemption

In times of resource contention (too many Pods trying to burst at once), Kubernetes will reclaim resource by removing (through a process known as evicting) pods that are using resources beyond their requested allocation. This is why it's so important to have a Pod's resources adequately specified.

EVICTON

Guaranteed class pods are never evicted in times of resource contention, so for a bulletproof deployment, always set your Pods' limits equal to their requests to define them as guaranteed. The rest of this section discusses how the non-guaranteed Pods are ranked when considering eviction, and how you can influence the ordering.

When looking for Pods to evict, Kubernetes first considers those Pods which are using more resources than their requests, sorts them by their priority number, and finally by how many more resources (of the resource in contention) that the Pod is using beyond what it requested. By default, all Pods have the same priority number (zero). When all Pods have the same priority, the amount of usage above requests is what's used to rank, as shown in the following diagram.

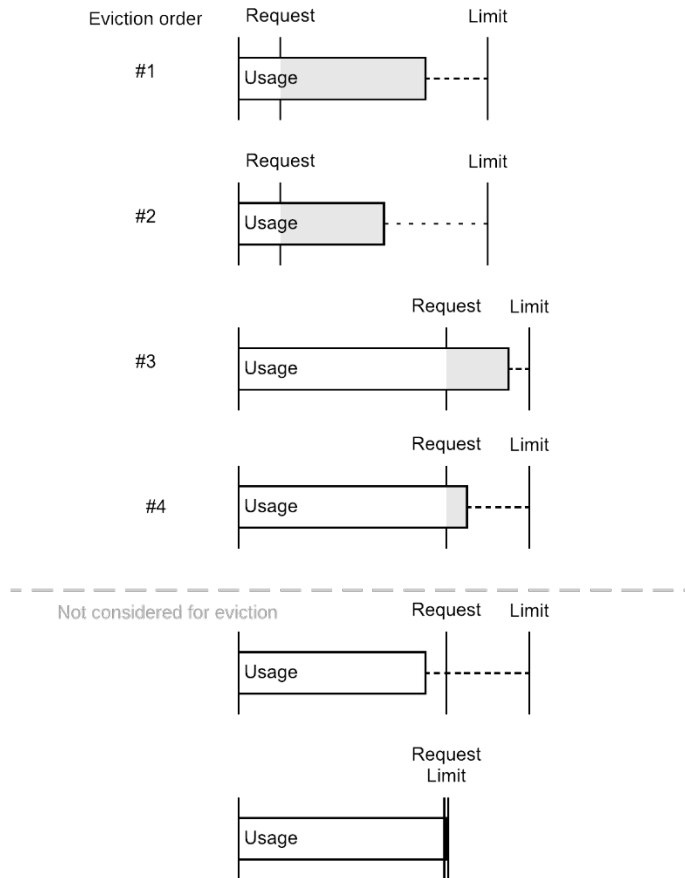


Figure 5.5 Eviction order for Pods of equal priority

“EJECTED” ERROR STATUS

If you query your Pods and you see a status of “Evicted” it indicates that the scheduler evicted your pod because it was using more resources than it requested. To resolve, increase the resources requested by your containers, and review whether you need to add more compute capacity to your cluster as a result.

PRIORITY

Priority is just an integer number (between 0 and 1,000,000,000) that you can assign to Pods (via a “priority class”) to change the ranking. The following diagram shows the eviction order if priority numbers were assigned to Pods from the previous diagram. As you can see,

the eviction is first sorted by the priority, then how much the usage is above requests. Pods that are not using more than their requests are not at risk of eviction, regardless of priority.

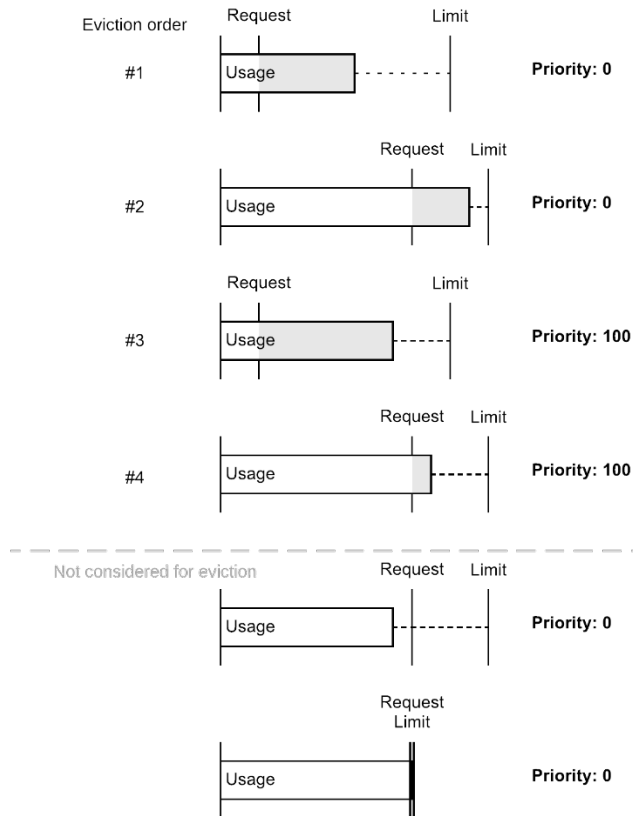


Figure 5.6 Eviction order of Pods with multiple priority values

To create your own priority level, you need to first create a `PriorityClass` object:

Listing 5.3 Priority/A_priority/priorityclass_high-priority.yaml

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000000 #A
preemptionPolicy: Never #B
globalDefault: false #C
description: "Critical services."
```

#A The priority integer

#B The 'Never' Preemption policy means that lower-priority Pods won't be booted if there's no space to schedule this Pod (see below for other options)

#C Whether this priority class should replace the default "0"

Then assign it to a Pod, like so:

Listing 5.4 Priority/A_priority/deploy.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 3
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      priorityClassName: high-priority #A
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:1
          resources:
            requests:
              cpu: 200m
              memory: 250Mi
```

#A The priority class to use for this deployment

The priority number is also used during scheduling. If you have many Pods waiting to be scheduled, the scheduler will schedule the highest priority Pod first. Using priority to govern the scheduling order is particularly useful for ranking which batch jobs should execute first (batch jobs are covered in a later chapter).

PREEMPTION

When used by itself, priority is useful to rank workloads so that more important workloads are scheduled first and evicted last. There can be a situation however where the cluster does not have enough resources for a period of time and high-priority Pods are left stuck in Pending while low priority ones are already running.

If you'd rather have higher priority workloads proactively bump lower priority ones rather than waiting for capacity to free up, you can add preemption behavior by changing the `preemptionPolicy` field in your `PriorityClass` as shown.

Listing 5.5 Priority/B_preemption/priorityclass_high-priority-preemption.yaml

```

apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority-preemption
  value: 1000000
preemptionPolicy: PreemptLowerPriority
globalDefault: false
description: "Critical services."

```

Pods that belong to Deployments that are removed from the node due to eviction, or preemption are not forgotten about. These Pods are returned to the “Pending” state and will be re-scheduled on the cluster when there are enough rooms.

WHEN TO USE PRIORITY AND PREEMPTION

Priority and preemption are useful features of Kubernetes and are important to understand due to the impact on eviction and scheduling. Before spending too much time configuring all your Deployments with Priority, I would prioritize ensuring that your Pod requests and limits are appropriate as that is the most important configuration.

Priority and preemption really come into play when you’re juggling many deployments and looking to save money by squeezing every ounce of compute out of your cluster by overcommitting where you need a way to signal which of your Pods are more important to resolve the resource contention. I wouldn’t recommend starting with this design as you’re just adding complexity. The simpler way to get started is to allocate enough resources to schedule all your workloads amply, and fine tune things later to squeeze some more efficiency out of your cluster.

Once again, the simplest way to guarantee the performance of your critical services is to set the resource requests appropriately, and have enough nodes in your cluster for them all to be scheduled.

5.2 Calculating Pod Resources

In the previous section we discussed why it’s important to set appropriate resource requests and limits for pods for the most reliable operational experience. But how do you determine what the best values are? The key is to run and observe your Pods.

Kubernetes ships with a resource usage monitoring tool out of the box `kubectl top`. You can use it to view the resources used by Pods and nodes. We’ll be focusing on Pods as that’s what we need to know to set the right resource request.

First, deploy your Pod with an excessively high resource request. This may even be a Pod that’s already in production—after all, it’s generally OK for performance (though not always for budget) to overestimate your resources needed. The goal of this exercise is to start high, observe the Pod’s actual usage, then pair the requests back to provision the right resources and avoid wastage.

Until you have a good feel for how many resources the Pod needs, it may be best to leave the limits unset (allowing it to use all the spare resources on the node). This doesn’t completely solve the need to set *some* resource requests, as you would prefer to be allocated dedicated capacity above what you need at first.

Listing 5.6 ResourceUsageTest/deploy.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 3
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:1
          resources: #A
            requests:
              cpu: 200m
              memory: 250Mi

```

#A The Pod under test. Resource limits are not set so we can analyze usage.

Run `kubectl top pods` (you may need to wait a minute or two for the data to be available), and note the startup resource usage, particularly memory. It's useful to have a snapshot of what resources the Pod needs to boot, as this is the lower bound if you choose to use Burstable quality of service.

Minikube Metrics

If you're using Minikube and get an error like `error:Metrics API not available`, you can enable metrics with `minikube addons enable metrics-server`.

Now, direct enough load at the Pod to simulate a real world usage. Performance tools like Apache Bench (installed with Apache⁴) can help here. An example Apache Bench command that will generate 10k requests total using 20 threads is below. You typically want to run this test for a while (say, 5 minutes) to make it simpler to observe the high water mark.

```
ab -n 10000 -c 20 https://example.com
```

The ideal situation would be to observe a Pod receiving normal production load. To avoid impacting production, you can temporarily overprovision your capacity (by setting high resource requests) and adding extra nodes to your cluster. Once you have a good measure of the actual needs, you can later tune the requests and right-size the cluster.

⁴ <http://httpd.apache.org/docs/2.4/install.html>

With your Pod under load either generated or production, run `kubectl top pods` again (remembering that it can take a minute or two to reflect the latest values, so keep your load simulation running).

Now you should have values like so:

Table 5.1 Memory and CPU usage from startup and under load

	Memory	CPU
Startup	200MiB	2%
Under Normal Load	400MiB	2%

It may be useful to repeat this process a couple more times and get values for your Pod under different loads (e.g. low, normal and high traffic), and timeframes. Multiple timeframes (e.g. directly after boot, 1 hour after boot, 1 day after boot) are useful to account for potential growth in usage (e.g. memory leaks).

So you might end up with something like:

Table 5.2 Memory and CPU usage after testing

	Memory	CPU
Startup	400MiB	2%
Under Normal Load	500MiB	20%
Under High Load	503MiB	25%
After 1 hour	505MiB	21%
After 1 day	600MiB	20%

5.2.1 Setting Memory Requests and Limits

With this data in hand, how should you set your resource requests? For starters, you now have an absolute lower bound for your memory: 400Mb. Since you're only ever guaranteed to get your resource request, and you know your Pod uses 400Mb under load, setting it any lower will likely cause your pod to be *OOMKilled* (terminated for being out of memory). You may not see it right away if you have a higher resource limit set, but you don't want to rely on spare capacity when you know you'll need it.

Does that make 400Mb the right request? Probably not. Firstly, you definitely want to have a buffer, say 10%. Also, you can see that after an hour, 505Mb was used, so this might be a better starting lower bound (before accounting for the buffer). Does it need to be 600Mb though? We saw that after a day the Pod needed that much, possibly due to a leak somewhere. This depends. You certainly could set this higher limit, then you could have some confidence that your pod could run for a day, but also thanks to Kubernetes' automatic restarting of crashed containers, having the system reboot a leaky process after a day to reclaim memory may be OK, or even desirable.

When Memory Leaks are OK

Instagram famously² disabled the garbage collection in Python for a 10% CPU improvement. While this is probably not for everyone, it's an interesting pattern to consider. Does it really matter if a process gets bloated over time and is rebooted, if it all happens automatically, and there are thousands of replicas? Maybe not.

Kubernetes automatically restarts crashed containers (including when that crash is due to the system removing them due to an out of memory condition), making it fairly easy to implement such a pattern. I wouldn't recommend this without thorough investigation, but I do think it indicates that if your application has a slow leak that it may not be your highest priority bug to fix.

Importantly, you need to make sure you at least give the container enough resources to boot and run for a time, otherwise you could get caught in a OOMKill crashloop which is no fun for anyone. Having enough replicas (covered in the next section) is also important to avoid a user visible failure.

Using the data you gathered, find the lower bound by looking at the memory usage of your Pod under load, and add a reasonable buffer (at least 10%). With this example data, would have picked 505MB * 1.1 = 555Mb. You know it's enough to run the Pod under load for at least an hour, with a bit to spare. Depending on your budget, and risk profile you can tune this number accordingly (the higher it is, the lower the risk, but higher the cost).

So, requests need to at least cover the stable state of the Pod. What about the memory *limit*? Assuming your data is solid and covered all use-cases (i.e. there's no high-memory code path that didn't execute while you were observing), I wouldn't set it too much higher than the 1-day value. Having an excessive limit (say, twice as high as the limit or greater) doesn't really help much since you already measured how much memory your Pods need over the course of a day, and if you do have a memory leak it may be better for the Pod to simply be restarted by the system when the limit is hit, than to be allowed to grow excessively.

An alternative is to simply set the limit equal to the request, for the guaranteed QoS class. This has the advantage of giving your Pod constant performance regardless of what else is running on the node. In this case, you should give the Pod a little extra resource buffer, since the Pod will be terminated the moment it exceeds its requested amount.

5.2.2 Setting CPU Requests and Limits

Unlike Memory, CPU is *compressible*. What this means is that if you don't get the CPU resources you need, the application just runs slower. Whereas if it doesn't get the memory it needs, it will crash. You still likely want to give the application the CPU it needs, otherwise performance will decrease, but there's not as much need to have a buffer as there is with memory.

In the example data for our application shown in Table 2 above, we can see that the stable state is about 20% of CPU. That would seem to be a good starting point for your CPU requests. If you want to save money and are OK with degrading performance, you could set it a little lower.

² <https://instagram-engineering.com/dismissing-python-garbage-collection-at-instagram-4dca40b29172>

For the CPU **limit**, this is an area where Kubernetes can improve your resource efficiency as you can set a limit higher than your requests to enable your application to consume the unused cycles on the node if it needs to burst. As with memory, Kubernetes only guarantees the requested CPU, but often it's nice to allow the pod to take advantage of unused capacity on the node to run a bit faster. For web applications that spend a lot of time waiting on external dependencies (e.g. waiting for a database to respond), there is often spare CPU capacity on the node which the active request could take advantage of.

As with memory, the downside of setting limits to higher than requests (i.e. the Burstable QoS class), is it means that your performance won't be constant. A burstable Pod running on an empty node will have a lot more resources than on a node packed with Pods. While generally it's nice to be able to handle bursts in traffic by consuming the unused capacity on the node, if constant performance is important, setting limits equal to requests may be preferable.

5.2.3 Reducing Costs by Overcommitting CPU

One strategy to reduce costs is to use the property that CPU is compressible to overcommit the node. This is achieved by setting the CPU request to a low value (lower than what the Pod actually needs), and therefore cramming more Pods onto the node than you could if you were to set the CPU request to the actual usage.

This saves money but has the obvious performance drawback. However, in the case of workloads that are considered very *bursty* it can be a very desirable strategy. Let's say you are hosting hundreds of low-traffic applications. Each may only get a few requests an hour, only needing the CPU for that time. For such a deployment, each application could have a CPU request of 1% (allowing 100 to be scheduled on a single core), and limit of 25% (allowing it to temporarily burst up to a quarter of a core).

The key for making an overcommitment strategy like this work is that you need to be intimately aware of what else is running on the machine. If you are confident that most of the websites will be idle most of the time this could work, but it may break down if all 100 suddenly need all the resources (unless 1% of a core is actually enough). This kind of violates one of the isolation properties of containers: now you need to be aware of and plan the make-up of the node accordingly. However, it can be done.

The safest approach of course is not to overcommit at all. A sensible compromise is to not overcommit too much. Giving Pods a little extra CPU (through setting their resource limits higher) can help reduce latency in an opportunistic fashion, but set your CPU resource requests high enough to handle a reasonable base load so that this excess capacity isn't being relied on.

5.2.4 Balancing Pod Replicas and Internal Pod Concurrency

Now you have a handle on how resource requests influence how your Pods are scheduled, and the resources they get, it's worth considering concurrency within the Pod. A Pod's concurrency (e.g. how many processes/threads of the application are running) influences the resource size, and there is a trade-off of efficiency for durability by using concurrency within the Pod over Pod replicas.

If you're coming from an environment where installations of your application were expensive, either in monetary cost for servers, or time to configure instances, your application will likely have a lot of internal concurrency configured through the use of threads and/or forks, often described as the number of "workers" used to handle incoming requests concurrently.

Concurrent workers still have advantages in the Kubernetes world due to their resource efficiency. I wouldn't take a Pod that currently had 10 workers, and instead deploy 10 replicas with 1 worker each. The container's internal concurrency is very efficient memory wise, as forks share some of the memory used by the application binary, and threads share even more. CPU is also pooled between workers which is useful as a typical web application spends a lot of time waiting on external dependencies, meaning there is often spare capacity to handle many requests at once.

Balancing the benefits of workers is the fact that the more replicas of a Pod you have, the more durable it is. For example, if you have 2 replicas of a pod, with 18 workers each to handle a total of 36 concurrent connections, then if one of those Pods were to crash (or be restarted because it failed the health check you setup in Ch.4), half your capacity would be offline before the Pod restarts. A better approach might be to have 6 Pods replicas with 6 workers each, still maintaining some inter-container concurrency while adding some redundancy.

To strike the right balance, a simple heuristic like the following can be used. Consider the total number of workers you need to serve your users, and of those how many can be offline at any given time without noticeable user impact. Once you've calculated how many can be offline—using our previous example, say 16% of the 36 workers could be offline before issues are noticed—then the most number of workers you can concentrate in a single Pod is 16%, or 6. If you find this results in more replicas than you need, increase the total number of workers so you can have more per Pod.

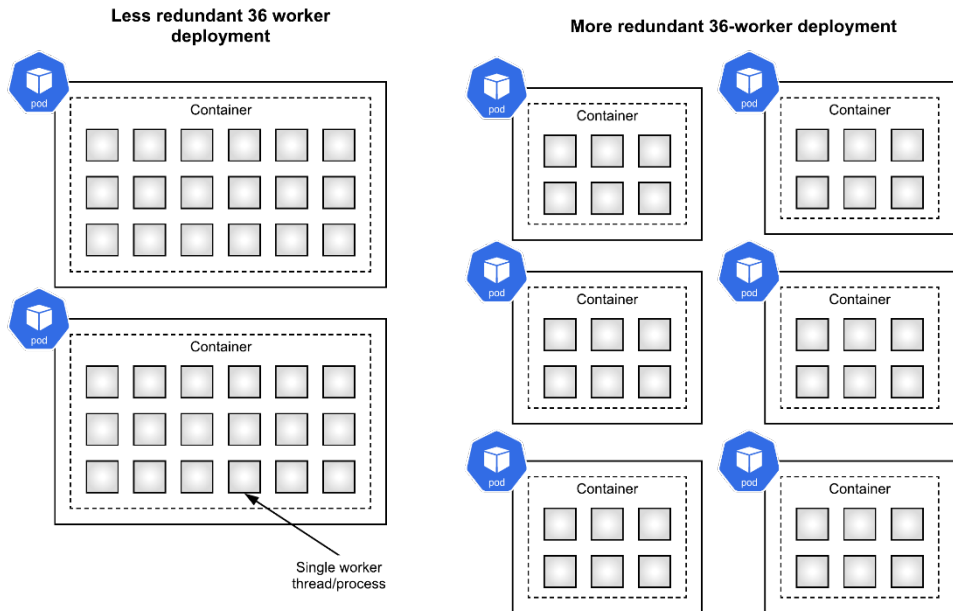


Figure 5.7 Comparison of two possible deployments for a total of 36 workers

In short, the more Pod replicas you have, the safer the design, but the less efficient in terms of resource usage, so it's worth considering this and balancing your own availability and resource requirements.

After balancing the number of Pod replicas you have, another important attribute to increase availability is to ensure that your Pods are spread across multiple nodes. After all, if you design for multiple replicas, but all those replicas run on the same node, you're still at risk from a single point of failure if that node were to become unhealthy. Fortunately, most Kubernetes platforms (including GKE) enable default Pod spreading policies that will spread Pods over all available nodes, and across multiple zones (in the case of a regional cluster). To get this default behavior, it's generally enough to ensure that you have a bunch of nodes in your cluster in different zones. If you want to dig more into node placement, and Pod Spread Topologies, Chapter 8 has you covered.

5.3 Summary

- The Kubernetes scheduler lies at the core of the system and does the heavy lifting of finding the right home for your deployment's Pods on your infrastructure.
- The scheduler will try to fit as many containers as it can on a given node, provided Pods have resource requests set appropriately
- Kubernetes uses the Pod's resource requests and limits to govern how resources are allocated, overcommitted, and reclaimed
- Overcommitting resources using bursting can save resources but introduces performance variability
- The specification of requests and limits by your workloads sets the quality of service they receive
- When designing your workloads, there is an availability/resource-usage trade-off between the replica count and the Pod's internal thread/process worker count
- Most platforms enable Pod spreading by default, to ensure that replicas are not generally placed on the same node, thus avoiding a single point of failure. Make sure you have a few nodes in your cluster to achieve higher availability.

6

Scaling Up

This chapter covers

- Scaling Pods and Nodes manually
- Using CPU utilization and other metrics to dynamically scale Pod replicas
- How some Kubernetes platforms can be configured to add and remove nodes automatically based on the resources your Pods require
- Using low priority “balloon” pods to provision burst capacity
- Architecting apps so that they can be scaled

Now that we have the application deployed and have health checks in place to keep it running without intervention, it’s a good time to look at how you’re going to scale up. I’ve named this chapter “scaling up”, as I think everyone cares deeply about whether their system architecture can handle being scaled up when your application becomes wildly successful and you need to serve all your new users, but don’t worry I’ll also cover scaling down so you can save money during the quiet periods.

The goal is ultimately to operationalize our deployment using automatic scaling, that way we can be fast asleep, or relaxing on a beach in Australia, and our application can be responding to traffic spikes dynamically. To get there, we’ll need to ensure that the application is capable of scaling, understand the scaling interactions of Pods and Nodes in the Kubernetes cluster, and the right metrics to configure an autoscaler to do it all for us.

6.1 Scaling Pods and Nodes

Getting your application containerized and deployed on Kubernetes is a great step towards building an application deployment that is capable of scaling and supporting your growth. Let’s now go over how to actually scale things up when that moment of success arrives, and the traffic increases (and scale things down to save some money in the quiet periods).

In Kubernetes, there are essentially two resources that you need to scale: your application (Pods), and the compute resources they run on (Nodes). What can make life a bit complicated is that the way you scale these resources is separate, even though the requirements (e.g. more application capacity) are somewhat correlated. Unless you're using a fully automated Kubernetes platform it's not enough to just scale Pods as they'll run out of compute resources to run on, nor is it enough to scale up Nodes alone as that just adds empty capacity. Scaling both in unison and at the correct ratio is what's needed. Fortunately there are some tools to make your life easier which I'll cover below.

To handle more traffic to your application, you'll need to increase the number of Pod replicas. Starting with the manual approach, you can achieve this by updating your deployment configuration with the desired number of replicas.

Listing 6.1 Replicas/deploy.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 6 #A
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:1
          resources:
            requests:
              cpu: 200m
              memory: 250Mi
```

#A The replicas field specifies how many duplicate copies of your Pod that you wish to be running

As usual, you can apply changes you make to config with `kubectl apply -f deploy.yaml`.

Kubectl also offers a convenient imperative command that can achieve the same result:

```
kubectl scale deployment YOUR_DEPLOYMENT --replicas=6
```

Where the nodes come in, is that if you try to add too many Pod replicas, you'll run out of space in your cluster for those Pods to be scheduled. You'll know when you've run out of room when you run `kubectl get pods`, and a bunch are listed as "Pending".

Pods can be in "Pending" for a number of reasons, the most common of which is a lack of resources. Essentially the lack of resources is an unsatisfied condition, and the Pod remains "Pending" until the condition can be satisfied. There can be other unsatisfied conditions as well if the Pod has dependencies (like requiring to be deployed on a node with another Pod that hasn't been created). To disambiguate, describe the pod with `kubectl describe pod`

`POD_NAME`, and look at the events. If you see an event such as “FailedScheduling” with a message like “Insufficient CPU”, you likely need to add more nodes.

Nodeless Kubernetes

I’d like to take a moment to cover Nodeless Kubernetes platforms. It is my opinion that the ideal cloud Kubernetes platform is one where the operator doesn’t really need to care a whole lot about Nodes. After all, if you’re using the Cloud, why not have a platform that provisions the Node resources that are needed based on the pod’s requirements so you can focus more on creating great applications?

In my job as a Product Manager at Google Cloud, this is exactly the product I built with my team. We called it *GKE Autopilot*. My design for this experience is one of a platform where nodes are outside of the developer’s concern. Define your Kubernetes resources like Deployments, StatefulSet, and Jobs, set the appropriate CPU and memory resource requests, and Autopilot will provision the compute needed to run your Pods, and manage that compute capacity for you. This both improves developer efficiency (no need to define your computing requirements twice; in the Pod and the Node), and operational efficiency (the platform operator takes on more of the Node operational burden).

One thing that sets *Autopilot* apart from the pack is that the Kubernetes node concept retains some relevance. Much of the node-related scheduling logic (like spread topologies, affinity, and anti-affinity covered in Chapter 5) is relevant and can still be used. *Autopilot* is “nodeless” in the sense that you no longer need to worry about how nodes are provisioned or managed, but it doesn’t completely abstract away or hide nodes. After all, there *is* a machine somewhere running your code, and this can have physical relevance around things like failure domains or wishing to co-locate pods for reduced latency.

I believe *Autopilot* has a best-of-both worlds design that gives you the node level controls that you need, while still removing the burden of operating and administering those nodes. No need to care any more about how many nodes you have, their size and shape, whether they are healthy, and whether they are sitting idle or underused.

If you are using GKE Autopilot or a platform like it, you can basically ignore everything in this chapter that talks about scaling *Nodes*, and focus purely on scaling *Pods*. Scaling pods manually, or automatically with a `HorizontalPodAutoscaler` works in *Autopilot* to also provision the necessary node resources, without the need to pair it with another system that scales nodes.

To scale the nodes, you’ll need to consult your Kubernetes provider’s platform documentation, as Kubernetes itself doesn’t orchestrate nodes. In the case of GKE, if you use Autopilot, nodes are provisioned automatically, and you can skip right ahead to Section 6.2. For GKE’s Standard mode of operation, the command looks like this:

```
gcloud container clusters resize cluster-name --node-pool pool-name \
--num-nodes $NODE_COUNT
```

Scaling down is performed with the same commands. When you scale down the nodes, depending on your provider, you should be able to run the same command as to scale up, and the cluster will first cordon and drain the nodes (to prevent new Pods being scheduled on them, and give running Pods’ time to shutdown gracefully and be re-created on other nodes). Alternatively, you can manually cordon, drain, and remove nodes with the following commands:

```
kubectl get nodes # get a list of node names
kubectl cordon node $NODE_NAME # prevent scheduling on the node
kubectl drain node $NODE_NAME # remove running pods from the node
kubectl delete node $NODE_NAME
```

These manual steps are also useful if you have a node that you found was not behaving correctly, and you want it gone.

So, this is how you scale Pods and Nodes by hand. Read on to learn how you can automate both these operations with Horizontal Pod Autoscaling to scale pods, and Cluster Autoscaling to scale Nodes (for Cloud providers that offer it).

6.2 Horizontal Pod Autoscaling

Scaling the number of Pod replicas of your application in Kubernetes is referred to as horizontal pod autoscaling. It's "horizontal" as you're increasing the number of replicas in order to serve increased traffic, rather than "vertical" which implies increasing the resources available to each replica instead. Generally, to scale up a system, it's horizontal scaling that you want.

Kubernetes includes a feature called the Horizontal Pod Autoscaler (HPA), a system whereby you specify a Pod *metric* like CPU usage to observe and target, along with some scaling limits (minimum and maximum replicas). The HPA will then aim to satisfy your metric by creating and removing Pods. In the case of CPU, say your target is 20% CPU utilization, this means that the HPA will add replicas when your average utilization (across all pods) goes above 20% (of what the Pod requested in its resource requests), and remove them when it goes below 20%. These actions are subject to a minimum and maximum limit you provide, as well as cooldown periods to avoid too much churn.

We can create a HPA for our deployment like so:

Listing 6.2 HPA/hpa.yaml

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: timeserver
spec:
  minReplicas: 1 #A
  maxReplicas: 10 #B
  metrics:
  - resource:
      name: cpu
      target:
        averageUtilization: 20 #C
        type: Utilization
    type: Resource
  scaleTargetRef: #D
    apiVersion: apps/v1 #D
    kind: Deployment #D
    name: timeserver #D
```

#A minimum number of replicas
 #B maximum number of replicas
 #C The CPU utilization target (will create more replicas when the Pod CPU utilization is higher than this value)
 #D The deployment that will be scaled

You can also create it imperatively. As always, I prefer the config approach as it makes it easier to edit things later. But here is the equivalent imperative command for completeness:

```
kubectl autoscale deployment timeserver --cpu-percent=20 --min=1 --max=10
```

To test this, we'll need to make the CPU really busy. Let's add a really CPU-intensive path to our timeserver application: calculating pi.

Listing 6.3 /timeserver4/pi.py

```
from decimal import *

# Calculate pi using the Gregory-Leibniz infinity series
def leibniz_pi(iterations):

    precision = 20
    getcontext().prec = 20
    piDiv4 = Decimal(1)
    odd = Decimal(3)

    for i in range(0, iterations):
        piDiv4 = piDiv4 - 1/odd
        odd = odd + 2
        piDiv4 = piDiv4 + 1/odd
        odd = odd + 2

    return piDiv4 * 4
```

Listing 6.4 /timeserver4/server.py

```
from pi import *

# ...

case '/pi':
    pi = leibniz_pi(1000000)
    self.respond_with(200, str(pi))
```

Showing just new function (HTTP route) added to server.py.

Here's a revised deployment that references this new version. To work correctly with the HPA, it's important to set resource requests which we added in Chapter 5 and are present here.

Listing 6.5 HPA/deploy.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 1
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:4
          resources:
            requests:
              cpu: 250m
              memory: 250Mi

```

We can now create the deployment, service and the HorizontalPodAutoscaler:

```

$ cd Chapter06/6.2_HPA
$ kubectl create -f deploy.yaml -f service.yaml -f hpa.yaml
deployment.apps/timeserver created
service/timeserver created
horizontalpodautoscaler.autoscaling/timeserver created
$ kubectl get svc -w

```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.22.128.1	<none>	443/TCP	6m35s
timeserver	LoadBalancer	10.22.131.179	<pending>	80:32650/TCP	18s
timeserver	LoadBalancer	10.22.131.179	34.83.118.233	80:32650/TCP	26s

While you're waiting for the external IP to be provisioned you can start watching the CPU utilization of your Pods with the following command (I'd suggest putting it in a new window):

```
kubectl top pods
```

Once you have the external IP, generate some load on the endpoint. Apache Bench (which you can install on most systems) works well for this. The following command will send 50 requests simultaneously to our endpoint, until 10k have been sent—that should do it!

```
ab -n 10000 -c 5 http://$EXTERNAL_IP/pi
```

You can watch the status of the deployment with the following:

```
kubectl get pods -w
```

To watch all resources using a single command (which kubectl can't do by itself), the Linux `watch` command is convenient:

```
watch -d kubectl get deploy,hpa,pods
```

If all goes correctly, you should observe the CPU utilization increase as visible with `kubectl top pods`, and more Pods replica being created. Once you stop sending load to the endpoint (e.g. by interrupting `ab`, or waiting for it to finish), you should observe these replicas gradually being removed. Generally the removal of replicas is slower than the addition, so that the system quickly scales up to meet demand, and cautiously scales down so that there's a bit of spare capacity in case the demand is spikey.

Here's what it looked like for my sample run:

```
$ kubectl get deploy,hpa,pods
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/timeserver	2/6	6	2	7m7s

NAME	MAXPODS	REPLICAS	AGE	REFERENCE	TARGETS	MINPODS
horizontalpodautoscaler.autoscaling/timeserver	6	6	7m7s	Deployment/timeserver	100%/30%	1

NAME	READY	STATUS	RESTARTS	AGE
pod/timeserver-b8789946f-2b969	1/1	Running	0	7m7s
pod/timeserver-b8789946f-fzbnk	0/1	Pending	0	96s
pod/timeserver-b8789946f-htwn	1/1	Running	0	96s
pod/timeserver-b8789946f-vvnhj	0/1	Pending	0	96s
pod/timeserver-b8789946f-xw9zf	0/1	ContainerCreating	0	36s
pod/timeserver-b8789946f-zbw9	0/1	ContainerCreating	0	36s

The HPA autoscaler shown here worked pretty well using the CPU metric, but there's a catch—your workload may not be CPU bound. Unlike the CPU-intensive request used in the demo, many HTTP services spend a lot of time waiting on external services like databases. These deployments may need to scale using other metrics like the number of requests per second hitting the service, rather than the CPU utilization. Kubernetes offers two built in metrics: CPU (demonstrated above), and memory. It doesn't directly support metrics like requests per second, but it can be configured by using custom and external metrics exposed by your monitoring service. The next section covers this situation.

What about *Vertical* Pod Autoscaling?

Vertical Pod Autoscaling (VPA) is a concept whereby Pods are scaled vertically by adjusting their CPU and memory resources. Implementations of VPA in Kubernetes achieve this by observing the Pods resource usage, and dynamically changing the Pod's resource requests over time. Kubernetes doesn't offer a VPA implementation out of the box, although an open source implementation is available¹, and cloud providers including GKE offer their own versions.

As a VPA can determine a Pod's resource requests automatically, it could save you some effort and provide some resource efficiency. It's also the right tool for the job if you need the Pod's resource requests to be adjusted dynamically over time (for Pods who have resource requirements that fluctuate widely).

Using a VPA adds its own complexity and may not always play nice with the HPA. I would focus first on setting appropriate pod resource requests, and the horizontal scaling of replicas.

¹ <https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler>

6.2.1 External Metrics

One popular scaling metric is Requests per Second (RPS). The basis of using RPS metrics for scaling is that you measure how many requests an instance of your application can serve every second (the replica's capacity). Then, you divide the current number of requests by this amount, and voila, you have the number of replicas needed.

$$\text{replica_count} = \text{RPS} \div \text{replica_capacity}$$

The benefit of the RPS metric is that if you are confident of your application to handle the RPS that you tested it for, then you can be confident that it should be able to scale under load as it's the auto-scaler's job to provision enough capacity.

In fact, even if you're not doing *automatic* scaling, this is still a really good way to plan your capacity. You can measure the capacity of your replicas, project your traffic and increase your replicas accordingly. But since this is Kubernetes, we can set this up as an auto scaler.

Now in this case, we'll be using the "external metric" property of the HPA. One issue with this is that the metric, as its name suggests, it is sourced from outside the cluster. So, if you're using a different monitoring solution than the one I use in my example, you'll need to look up what the relevant RPS metric is. Fortunately, this is a pretty common metric, and any monitoring solution worth its salt will offer it.

In prior chapters we discussed a few different ways to get traffic into your cluster, via a so-called L3 LoadBalancer which operates at a TCP/IP level, and a so-called L7 ingress which operates at the HTTP level. As "requests" are a HTTP concept, you'll need to be using an Ingress to get this metric. Ingress is covered in the next chapter in depth, for now it's enough to know that this object sees and inspects your HTTP traffic, thus can expose a metric for the number of requests you are getting.

For this example, we will use the same Deployment but expose it on an Ingress, via a NodePort service, instead of a LoadBalancer.

Listing 6.6 ExternalMetricGCP/service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: timeserver-internal
spec:
  selector:
    pod: timeserver-pod
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
  type: NodePort
```

Listing 6.7 ExternalMetricGCP/ingress.yaml

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: timeserver-ingress
spec:
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: timeserver-internal
            port:
              number: 80

```

Now if you're using Google Cloud, the following HPA definition can pick up the RPS metric from the L7 ingress, once you replace the forwarding rule name with your own.

Listing 6.8 ExternalMetricGCP/hpa.yaml

```

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: timeserver-autoscaler
spec:
  minReplicas: 1
  maxReplicas: 6
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: timeserver
  metrics:
  - type: External
    external:
      metric:
        name: loadbalancing.googleapis.com|https|request_count
        selector:
          matchLabels:
            resource.labels.forwarding_rule_name: "k8s2-fr-21mgs2f1-default-timeserver-
            ingress-s2j1ki7x"
      target:
        type: AverageValue
        averageValue: 5

```

The `forwarding_rule_name` is how the metric server knows which ingress object you're talking about. You can omit the `metricSelector` completely, but then it will match on all ingress objects, probably not what you want.

Unfortunately this forwarding rule name is a platform-specific resource name, and not the Kubernetes object name (and in this example, that name is set automatically by GKE). To discover the platform resource name, you can describe your ingress object like so:

```
$ kubectl describe ingress
Name:          timeserver-ingress
Namespace:     default
Address:       34.110.232.121
Default backend: default-http-backend:80 (10.22.0.202:8080)
Rules:
  Host      Path  Backends
  ----      -
  *         /    timeserver-internal:80 (10.22.0.130:80,10.22.0.131:80,10.22.0.196:80 + 1
more...)
Annotations:  ingress.kubernetes.io/backends:
               {"k8s-be-32730--a52250670846a599": "HEALTHY", "k8s1-a5225067-default-
timeserver-internal-80-6284288f": "HEALTHY"}
               ingress.kubernetes.io/forwarding-rule: k8s2-fr-21mgs2f1-default-timeserver-
ingress-s2j1ki7x
               ingress.kubernetes.io/target-proxy: k8s2-tp-21mgs2f1-default-timeserver-
ingress-s2j1ki7x
               ingress.kubernetes.io/url-map: k8s2-um-21mgs2f1-default-timeserver-ingress-
s2j1ki7x
Events:
  Type     Reason      Age           From                      Message
  ----     -
  Normal   Sync        6m28s (x31 over 5h6m)  loadbalancer-controller  Scheduled for sync
```

Another way to query this information which is important if you are configuring automated tooling, is to understand where the data is within the object structure, and use the JsonPath format of kubectl.

```
$ kubectl get ingress -
-o=jsonpath="{.items[0].metadata.annotations['ingress\.kubernetes\.io\/forwarding-
rule']}"
k8s2-fr-pqby6yf-default-pluscode-ingress-hrsjzvz
```

(How I built that JsonPath expression is to first query the `-o=json` version of the ingress, then figure out the path through a combination of looking at the JsonPath Docs, Stack Overflow, and trial and error).

Once you have the objects ready, there's one last step which is to install some glue that gives the HPA access to the metrics. You may want to check the latest instructions², at the time of writing it's a one-liner:

```
kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/k8s-
stackdriver/master/custom-metrics-stackdriver-
adapter/deploy/production/adapter_new_resource_model.yaml
```

With our Deployment, (internal NodePort) Service, Ingress, HPA and metric adapter all configured, we can now try it out!

Generate some requests to the ingress like so (replacing the IP of your ingress, obtained via `kubectl get ingress`):

² <https://github.com/GoogleCloudPlatform/k8s-stackdriver/tree/master/custom-metrics-stackdriver-adapter>

```
ab -n 100000 -c 100 http://34.110.232.121/
```

And in a separate window, observe the scale out:

```
$ kubectl get hpa,ingress,pods
```

NAME	MAXPODS	REPLICAS	AGE	REFERENCE	TARGETS	MINPODS
horizontalpodautoscaler.autoscaling/timeserver	6	4	5d1h	Deployment/timeserver	94%/30%	1

NAME	AGE	CLASS	HOSTS	ADDRESS	PORTS
ingress.networking.k8s.io/timeserver-ingress	30h	<none>	*	34.110.232.121	80

NAME	READY	STATUS	RESTARTS	AGE
pod/timeserver-b8789946f-8dpmg	1/1	Running	0	5h51m
pod/timeserver-b8789946f-gsrt5	0/1	ContainerCreating	0	110s
pod/timeserver-b8789946f-sjvqb	1/1	Running	0	110s
pod/timeserver-b8789946f-vmhsw	0/1	ContainerCreating	0	110s

One thing you may notice already is that it's already easier to validate that the system is performing more as expected. Apache Bench allows you to specify concurrent requests, you can see how long they take (and therefore calculate the RPS) and look at the number of replicas to see if it's right. This was a bit harder with the CPU metric where to test you might have just tried to make the pod as busy as possible. This property of scaling based on *user requests* is one reason why this is a popular metric to use.

OBSERVING AND DEBUGGING

To see what the HPA is doing, you can run `kubectl describe hpa`. Pay particular attention to the "ScalingActive" condition. If it is `False`, then it likely means that your metric is not active which can be for a number of reasons: a) the metric adapter wasn't installed (or isn't authenticated), b) your metric name or selector is wrong, or c) there just isn't any metrics available yet. Note that even with the correct configuration, you will see `False` when there is no data (for example there are no requests), so be sure to send some requests to the endpoint and wait a minute or two for the data to come through before investigating further.

AVERAGEVALUE VS VALUE

In the example above, we used `targetAverageValue`. `targetAverageValue` is the target *per pod* value of the metric. `targetValue` is an alternative, which is the target absolute value. As the RPS capacity is calculated at a per-pod level, it's `targetAverageValue` we want.

OTHER METRICS

Another popular external metric when dealing with Batch Jobs (covered in a later chapter) is the PubSub queue length. PubSub is a queuing system that allows you to have a queue of work that needs to be performed, and you can set up a workload in Kubernetes to process that queue. For such a setup, you may wish to react to the queue size by adding and removing Pod replicas (workers that can process the queue). You can find a fully worked

example on the GKE docs³ for this, essentially it boils down to a HPA that looks like the one above, just with a different metric:

```
metricName: pubsub.googleapis.com|subscription|num_undelivered_messages #A
metricSelector:
  matchLabels:
    resource.labels.subscription_id: your-subscription #B
```

#A the metric name

#B the resource for the metric, in this case the PubSub's GCP resource name

OTHER MONITORING SOLUTIONS

External metrics is something you should be able to configure for *any* Kubernetes monitoring system. While the worked example given above was using Cloud Monitoring on GCP, the same principles should apply if you're using Prometheus, or another cloud. To get things going, you'll need to determine a) how to install the metric adapter for your monitoring solution b) what the metric name is in that system, and c) the right way to select the metric resource.

6.3 Node Autoscaling & Capacity Planning

6.3.1 Cluster Autoscaling

Cluster autoscaling is not part of Kubernetes, but is a common provider-provided feature that scales nodes for you, allowing you to focus just on your application and how many replicas it has. As this is a platform-specific feature, the exact implementation will vary (and not all providers offer it). Search for "*product name* Cluster Autoscaler" to find the relevant docs.

In the case of GKE, if you use the Autopilot mode of operation clusters have built-in node provisioning and autoscaling, no further configuration required. For GKE's Standard mode of operation, you can configure autoscaling when creating a node pool, or update an existing node pool.

When using cluster autoscaling, you can focus on scaling your own workloads, having the cluster respond automatically. This is really convenient, as it can solve the "Pending" pods problem, both when scaling existing workloads and deploying new ones. Do read the specific implementation details of your provider though to understand what cases are not covered (like how pods that are too big to fit on current node configurations are handled).

³ https://cloud.google.com/kubernetes-engine/docs/tutorials/autoscaling-metrics#pubsub_8

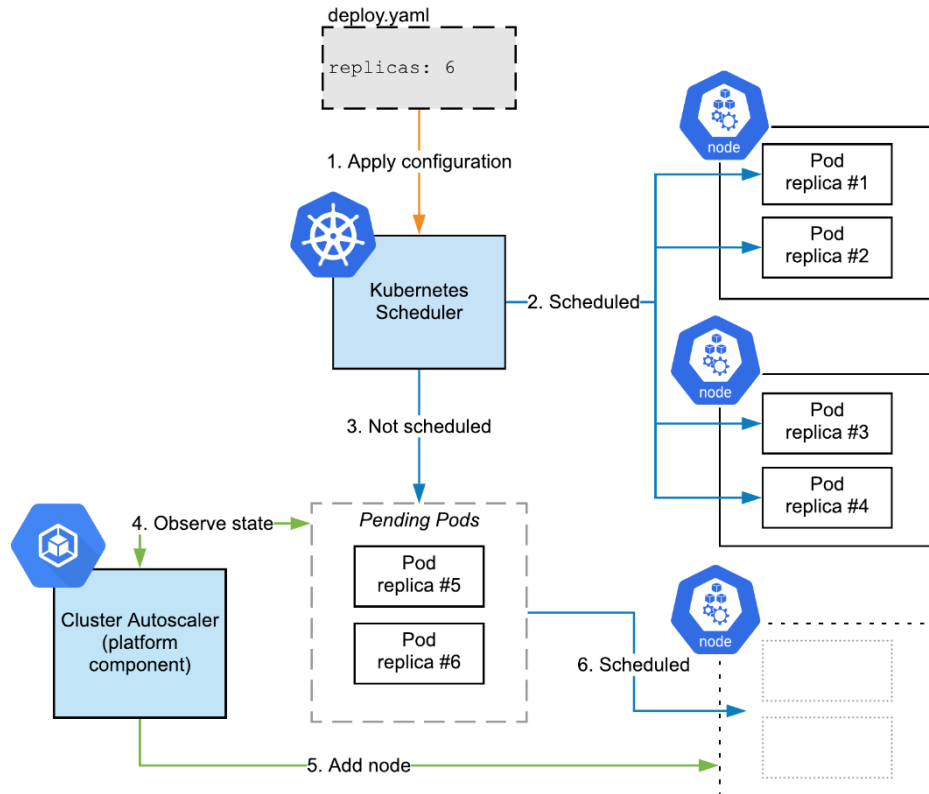


Figure 6.1 The Cluster Autoscaler watches for Pending pods, and creates new nodes if needed

Traditional cluster autoscalers may only add new nodes of an existing predefined configuration, requiring you to define each possible node type you wish to use, so be sure to read the docs. GKE can add new nodes of any type if you use Autopilot (no configuration needed, that's how it works out of the box), or Standard with Node Auto Provisioning configured.

Cluster autoscaling and other provider tools that can add and remove nodes automatically make your life easier by allowing you to mostly ignore nodes, and focus purely on your own Pods. When paired with Pod-based scaling like the HorizontalPodAutoscaler, you can have a fairly hands-off, automated deployment.

6.3.2 Spare Capacity with Cluster Autoscaling

One of the drawbacks of autoscaling nodes compared to manually adding nodes is that sometimes the Autoscaler can tune things a little *too* well and result in no spare capacity. This can be great for keeping costs down, but it makes it slower to start new Pods, as capacity needs to be provisioned before the Pod can startup.

Adding new Nodes then starting the Pod is slower than adding new Pods to existing Nodes. Nodes have to be provisioned and booted, while Pods that get scheduled onto existing nodes just have to pull the container and boot (and if the container is already in the cache, they can even start booting right away). As shown in the figure below, the newly scheduled Pod must wait for capacity to be provisioned before it can begin booting.

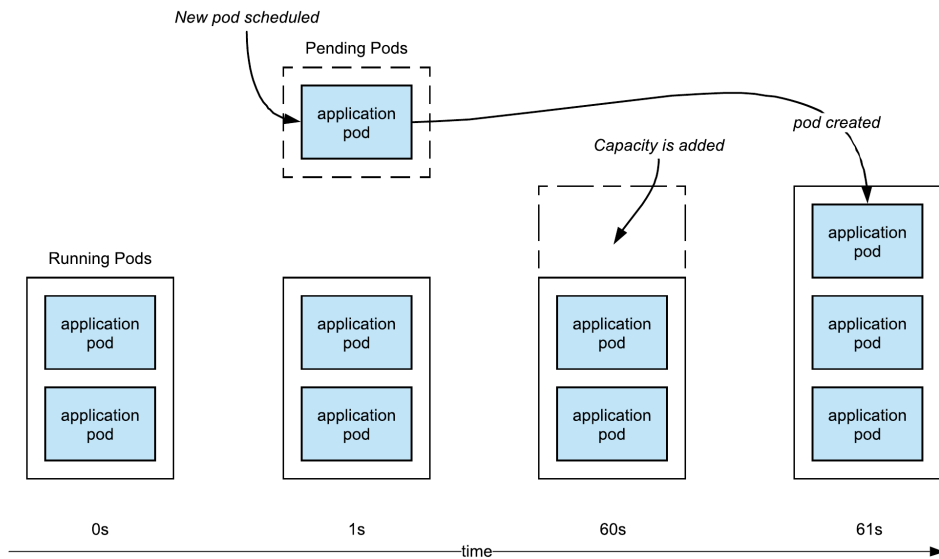


Figure 6.2 Dynamically adding capacity with autoscaling to accommodate newly scheduled pods

One way to solve both these problems, while still keeping your autoscaler is to use a low priority balloon pod. This is a Pod that does nothing itself, other than to reserve capacity (keeping additional nodes up and running on standby). This Pod's priority is low so that when your own workloads scale up, they can preempt this pod and use the node capacity.

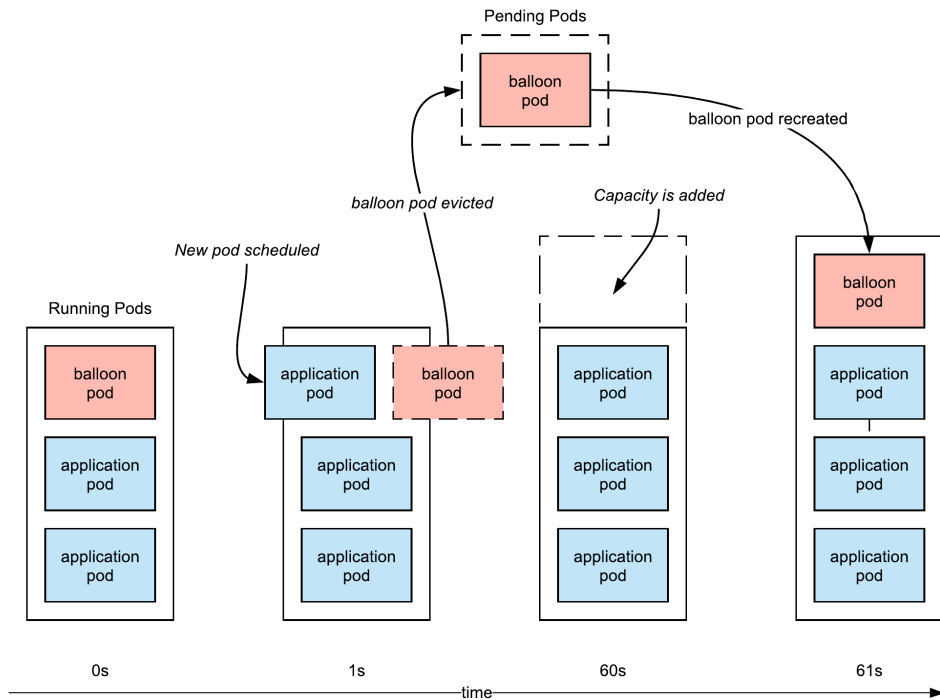


Figure 6.3 Autoscaling with a balloon pod allowing for rapid booting of new pods using spare capacity

To create our “balloon” pod deployment, first we’ll need a `PriorityClass`. This priority class should have the lowest possible priority (we want every other priority class to preempt it).

Listing 6.9 BalloonPod/balloon-priority.yaml

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: balloon-priority
value: -10
preemptionPolicy: Never
globalDefault: false
description: "Balloon pod priority."
```

Now we can create our “do nothing” container deployment like so:

Listing 6.10 BalloonPod/balloon-deploy.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: balloon-deploy
spec:
  replicas: 10 #A
  selector:
    matchLabels:
      app: balloon
  template:
    metadata:
      labels:
        app: balloon
    spec:
      priorityClassName: balloon-priority #B
      terminationGracePeriodSeconds: 0 #C
      containers:
      - name: ubuntu
        image: ubuntu
        command: ["sleep"] #D
        args: ["infinity"] #D
        resources:
          requests:
            cpu: 200m #E
            memory: 250Mi #E

```

#A how many replicas do you want? This, with the cpu and memory requests determines the size of the headroom capacity provided by the balloon pod

#B Use the priority we just created

#C We want this Pod to shutdown immediately with no grace period

#D This is our “do nothing” command

#E The resources that will be reserved by the balloon pod. This should be equal to the largest Pod you wish to replace this Pod

When creating this yourself, consider the number of replicas you need, the size (memory and CPU requests) of each replica. The size should be at least the size of your largest regular pod (otherwise, your workload may not fit in the space when the balloon pod is preempted). At the same time, don't increase the size too much – it would be better to use more replicas, than replicas that are much larger than your standard workloads pods if you wish to reserve extra capacity.

For these balloon pods to be preempted by other pods that you schedule, those pods will need to have a priority class that both has a higher value, and not have a `preemptionPolicy` of `Never`. Fortunately, the default priority class has a value of 0, and a `preemptionPolicy` of `PreemptLowerPriority`, so by default all other Pods will displace our balloon pod.

To represent the Kubernetes default as its own priority class, it would look like the following. As you don't actually need to change the default, I wouldn't bother configuring this. But if you're creating your own priority classes, you can use this as the reference (just don't set `globalDefault` to `true`, unless that's what you really intend). Once again: for the balloon pod preemption to work, be sure not to set `preemptionPolicy` to `Never`.

Listing 6.11 BalloonPod/default-priority.yaml

```

apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: default-priority
value: 0
preemptionPolicy: PreemptLowerPriority
globalDefault: true
description: "The global default priority. Will preempt the balloon pods."

```

Balloon Pods encapsulated in a Deployment like this are useful for providing constant scaling headroom, giving you a defined amount of capacity ready for quick scheduling. You can also encapsulate them in a Job, or run as a standalone Pod for alternative ways to provision capacity.

6.4 Building Your App to Scale

Scaling your application up is only part of the equation. The application itself needs to build with scaling in mind. Even though you may not be at the point of your growth where you need to worry about these issues, I believe that **the time when you need to scale, is not the time to design how you're going to scale.**

When your application is one with unpredictable growth (for example, a startup with potentially unlimited users), you really want to plan ahead to avoid the “success failure” scenario. This is where, in a breakout moment of your success, the app fails because it couldn’t handle the scale. Since you don’t know when this breakout moment will be, you need to have designed for this ahead of time. Not every startup will have a breakout moment, but if yours does, you want to be ready to capitalize on the opportunity otherwise it could all be for naught.

Fortunately, by choosing Kubernetes to orchestrate your containers, you are starting with a really solid foundation for a scalable app. There are some other factors to keep in mind when designing the application which are largely independent to Kubernetes. Most scalable design principles apply to both Kubernetes and non-Kubernetes environments, but I’ll cover a few best practices that are worth keeping in mind to build a scalable app on Kubernetes. Keeping some scalability principles in mind as you develop your application could matter in the future when your breakout moment arrives, and you need to scale it to the moon.

6.4.1 Avoiding State

One of the most important aspects to being able to scale, is avoiding local state in your applications. A stateless design is where each replica (instance) of your application that’s running can serve any incoming request, without reference to any data stored locally on any other instance. Local “ephemeral” storage can be used for temporary data processing, as long as it’s not shared between replicas and doesn’t need to be available for the next request that comes in.

The most important of the twelve factors

This property of the application being stateless is, I believe, the most important factor in the popular “The Twelve-Factor App” design methodology (<https://12factor.net/processes>). Stateless apps are easier to scale and maintain as each instance can independently serve any request.

Unlike with a classical host machine, in Kubernetes all data written to disk by the container is ephemeral (deleted when the container is terminated or restarted) by default. It is possible to create stateful applications using Persistent Volumes, and the StatefulSet construct (see Chapter 9), but by default containers are treated as stateless, and you generally want to keep it that way so that you can scale.

Rather than storing state on disks that you manage in Kubernetes, use external data stores to store data instead, like SQL and NoSQL databases for structured data, object storage for files, and memory databases like Redis for session state. To support your ability to scale, choose managed services (rather than self-hosting), and ensure the services you choose can handle your potential growth.

This is not to say that all state is bad. After all, you need *somewhere* to store your state, and sometimes this needs to be a self-hosted application. When you do create such an application, be sure to choose highly scalable solutions, like a popular open source solution with a track record of success (for example, Redis).

Relational Database Gotchas

If you use a relational database like MySQL or PostgreSQL to store data, then there are more than a few potential pitfalls worth paying attention to.

Taming your queries

It goes without saying that inefficient queries will give you inefficient scaling, slowing down as the amount of data increases and the number of requests increase. To keep things under control, I recommend logging and analyzing your queries, and starting early in the development process (you don’t want to wait until your app is a hit to look at the queries!).

You can’t improve what you don’t measure, so logging the queries a request performs and their run time is the most important first step. Look for requests that generate a lot of queries, or slow queries, and start there.

Both MySQL and PostgreSQL support the `EXPLAIN` command which can help analyze specific queries for performance. Common tactics to improve performance include adding indices for commonly searched columns, and reducing the number of JOINS you need to perform. MySQL’s documentation *Optimizing SELECT Statements*⁴ goes into great detail on many different optimization tactics.

Avoid N+1 queries

Even if your queries are super-efficient, each individual query you make to the database has overhead. Ideally each request your application processes should perform a constant number of queries, regardless of how much data is displayed.

⁴ <https://dev.mysql.com/doc/refman/8.0/en/select-optimization.html>

If you have a request that renders a list of objects, you ideally want to serve this without generating a separate query for each of those objects. This is commonly referred to as the N+1 query problem (as when the problem occurs, there is often 1 query to get the list, and then one for each item [N items] in the list).

This anti-pattern is particularly common with systems that use object-relational mapping (ORM), and feature lazy loading between parent and child objects. Rendering the child objects of a one-to-many relationship with lazy loading typically results in N+1 queries (1 query for the parent, N queries for the N child objects), which will show up in your logs. Fortunately there is normally a way with such systems to indicate upfront that you plan to access the child objects so that the queries can be batched.

Such N+1 query situations can normally be optimized into a constant number of queries, either with a JOIN to return the child objects in the list query, or two queries: one to get the record set, then a second to get the details for the child objects in that set. Remember: the goal is to have a small constant number of queries per request, and in particular, the number of queries shouldn't scale linearly with the number of records being presented.

Use read replicas for SELECT queries

One of the best ways to reduce the strain on your primary DB, is to create a read replica. In cloud environments, this is often really trivial to setup. Send all your read queries to your read replica (or replicas!) to keep the load off the primary "read/write" instance.

To design your application with this pattern in mind before you actually need a read replica, you could have two database connections in your application to the same database, using the second to simulate the read replica. Setup the "read only" connection with its own user that only has read permissions. Later when you need to deploy an actual read replica, you can simply update the instance address of your second connection, and you're good to go!

Incrementing primary keys

If you really hit it big, you may end up regretting using incrementing primary keys. They're a problem for scaling, as they assume a single writable database instance (inhibiting horizontal scaling) and require a lock when inserting which impacts performance (i.e. you can't insert two records at once).

This is really only a problem at very large scale, but worth keeping in mind as it's harder to rearchitecting things when you suddenly need to scale up. The common solution to this is global UUIDs (e.g. 8fe05a6e-e65d-11ea-b0da-00155d51dc33), a 128bit number commonly displayed as a hexadecimal string, which can be uniquely generated by any client (including code running on the user's device).

When Twitter needed to scale up, they opted instead to create their own global incrementing ids to retain the property that they are sortable (i.e. that newer tweets have a higher ID number), which you can read about in their post *Announcing Snowflake*⁶.

On the other hand, might prefer to keep incrementing primary keys for aesthetic reasons, like when the record ID is exposed to the user (as in the case of a tweet ID), or for simplicity. Even if you plan to keep your incrementing primary keys for a while, one step you can still take early on is not using auto incrementing primary keys in places where they wouldn't add any value, like say a user session object—maybe not every table needs an incrementing primary key.

6.4.2 Microservice Architectures

One way to build up your application is by splitting services into multiple services, often described as using a microservice architecture. This is basically just creating several internal services to perform separate tasks, and using remote procedure calls (a HTTP request, essentially) to call those functions from other services. This contrasts to the "monolith" service design approach, of having the complete program logic in a single container.

⁶ https://blog.twitter.com/engineering/en_us/a/2010/announcing-snowflake.html

While there are some benefits to splitting up a monolith into multiple smaller services, there are some drawbacks as well, so I don't advocate using a microservice architecture just for the sake of it. Benefits including being able to use different programming languages for each service, being able to develop them independently (for example, by separate teams), and independent scaling. Drawbacks include more complex debugging and integration testing, as you now have more components, and need a way to trace requests through the system.

Microservice vs Monolith

Should you build microservices, or a monolith? For the sake of this debate, which I'm not going to litigate that in this book, let me share two views on the topic and let you judge for yourself.

David Heinemeier Hansson (DHH) writes in his post *The Majestic Monolith* (<https://m.signalnoise.com/the-majestic-monolith/>) that microservices is for large tech companies, and most smaller teams are better served by a monolith. His argument is that while microservices can have advantages in certain situations, it's not always clear cut, and the overhead—particularly for smaller teams—is not worth it.

James Lewis and Martin Fowler in their essay on *Microservices* (<http://martinfowler.com/articles/microservices.html>) lay out a well thought and balanced view on microservices. Benefits highlighted include a product mentality where internal teams focus on building and managing their own components, a decentralized approach that allows teams to make their own architectural decisions.

Whether you go all in on microservices or not, the key point I want to focus on here is that if you have multiple services, you can scale them separately. This is true of course even if you have just a single internal service in addition to your main application—there's no need to make *every* endpoint its own service to benefit from this architecture. For example, say you have a web application that mostly serves HTML and JSON requests, but has one endpoint that does some real-time graphics work that uses a more memory than your average request. It might be worth creating a separate Deployment (even one using the same container) to serve the graphics endpoint so you can scale it separately, and also isolate it a bit.

There are a couple of ways to do this too. You can have a single front end, that calls the internal service, as illustrated in figure 6.4, or you can have end users connect to this new service directly, as shown in figure 6.5.

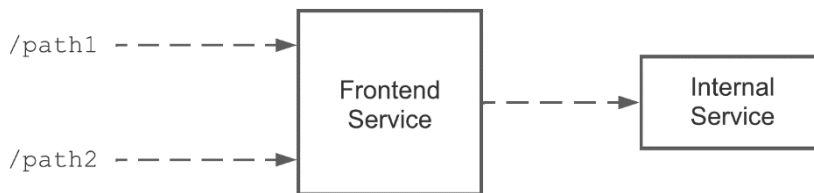


Figure 6.4 Two HTTP paths being served by the same frontend that communicates with an internal service.

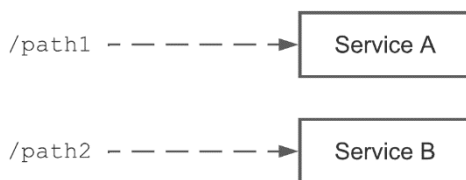


Figure 6.5 Two paths being served by separate services

Whether you are going all-in on microservices, splitting off a single service to be handled by its own scalable deployment, using multiple programming languages, or running internally developed and open-source software to provide your application, you will end up creating *internal services* in Kubernetes. Internal services are Kubernetes Services that are provisioned with a private cluster IP address, and are called by other services in the cluster in order to deliver this architecture. The next chapter covers how to configure such internal services.

6.4.3 Background Jobs

Another important factor to help you scale is to avoid having any heavy processing inline. For example, let's say you have an endpoint that returns a thumbnail of an image, and will generate the thumbnail if it doesn't exist in the cache. You can place this logic inline, where the user requests the thumbnail, and the service responds by returning the thumbnail from the cache, or generating one if the cache is empty. The problem with such a design is that serving the thumbnail from the cache should be very fast, while creating the thumbnail is not. If a lot of requests come in all needing to create a thumbnail, the server could slow down or crash. Plus, it's hard to scale, because some requests are lightweight, and others are really heavy. You could scale this service up, but still be unlucky and have your load balancer direct all the heavy requests at a single instance.

The solution to this is to use the background job pattern, covered in detail in Chapter 10. Essentially, when the heavy processing is needed, rather than doing it inline, you schedule a

job and return a status code to the client indicating it should retry the request. There is a container configured to process this job queue, and which can be scaled accurately based on the current queue length. So, the request comes in, resulting in a cache miss and a queued job. If things go well, when the client automatically retries the request after a short time, the thumbnail will have been processed by the background queue, and be ready for serving. Similar end result for the user, a little extra work to build a background queue and a client with retry logic, but much better scalability.

6.5 Summary

- Kubernetes is well suited to help you scale, some of the largest applications out there run on Kubernetes.
- To make the most of this architecture, design your application at the get-go so that it can scale horizontally
- The `HorizontalPodAutoscaler` can be used to provision new Pods as needed, working together with the Cluster Autoscaler for a complete auto-scaling solution
- You're not just confined to CPU metrics, but can scale your Pods based on any metric exported by your monitoring solution
- The Cluster Autoscaler technique (if supported by your provider) can be used to provision new nodes as needed
- Balloon pods can be used to add capacity headroom even while autoscaling
- Considering splitting your application into microservices, or simply hosting multiple deployments of the same application to allow for separate scaling groups

7

Internal Services and Load Balancing

This chapter covers

- Creating internal services
- How packets are routed in Kubernetes between virtual IP addresses of pods and services
- Discovering the IP address of internal services
- Configuring HTTP Load Balancers with Ingress
- Provisioning TLS certificates to create HTTPS endpoints

Internal services are a way to scale how you develop and service your application by splitting your application into multiple smaller services. These individual services can be on different development cycles (possibly by different teams) and use completely different languages and technology from each other. After all, as long as you can containerize it, you can run it in Kubernetes. No longer do you need to worry whether your application deployment platform can run what you need it to run.

In this chapter, we'll look at how to configure and discover internal services in the cluster, as well as how Kubernetes gives each of these a cluster-local IP address and implements internal network routing to make them accessible from each other. We'll also look at how you can expose multiple services on a single external IP using Ingress, and how it can handle TLS termination so you can offer https endpoints for your application, without needing to configure TLS certificates in your applications.

7.1 Internal Services

There are many reasons to create services that are completely internal to your cluster. Perhaps you've adopted a microservice architecture, or you're integrating an open source service, or you just want to connect two applications together that are written in different languages.

In Chapter 3, I introduced Services of the type LoadBalancer as the way to get external traffic on a public IP. Services are also be used to connect internal services, but using private IP addresses. Kubernetes supports a few different Service types, the two used for internal services are ClusterIP, and NodePort.

ClusterIP gives you a virtual IP address in the Kubernetes cluster. This IP is addressable from any Pod within your cluster (like in this example, your main application). NodePort on the other hand reserves a high-level port number on each node in the cluster, allowing you to access it from any Pod using localhost, and the service IP. In both cases, Kubernetes provides the networking setup to proxy requests to the Pods that back the service.

7.1.1 Kubernetes Cluster Networking

Now might be a good time for a quick primer on Kubernetes networking. In Kubernetes, each Node, Pod and Service gets its own internal IP (on a private network). In the case of the Node, these IPs are directly routable on the private network, being assigned to a NIC (Network Interface Card). Nodes may also have a public IP routable on the public internet, but that isn't generally used for cluster-internal traffic.

In the case of Pods and Services, their internal IPs are virtual (there is no NIC, unlike for the nodes), and networking glue (provided by Kubernetes using either iptables or IPVS) is installed on each node to seamlessly route traffic via the node's NIC. This topic alone could fill several chapters; what you primarily need to know for deploying applications and consuming internal services is that **Pods and Services have virtual IPs that are usable within the cluster**. This means that each Pod has it's own port ranges to allocate too: you don't have to worry about port conflicts between Pods, in the same way you may normally when running several applications or containers on a host.

When a request is made from a Pod to a service over the ClusterIP or NodePort, that request is first handled by the networking glue on the node, which has an updated list from the Kubernetes control plane of every Pod that belongs to that service (and which nodes those Pods are on). It will pick one of the Pod IPs at random and route the request to that Pod via its node. Fortunately, all this happens quite seamlessly, your app can simply make a request such like HTTP GET using the IP of the service, and everything behaves as you'd expect.

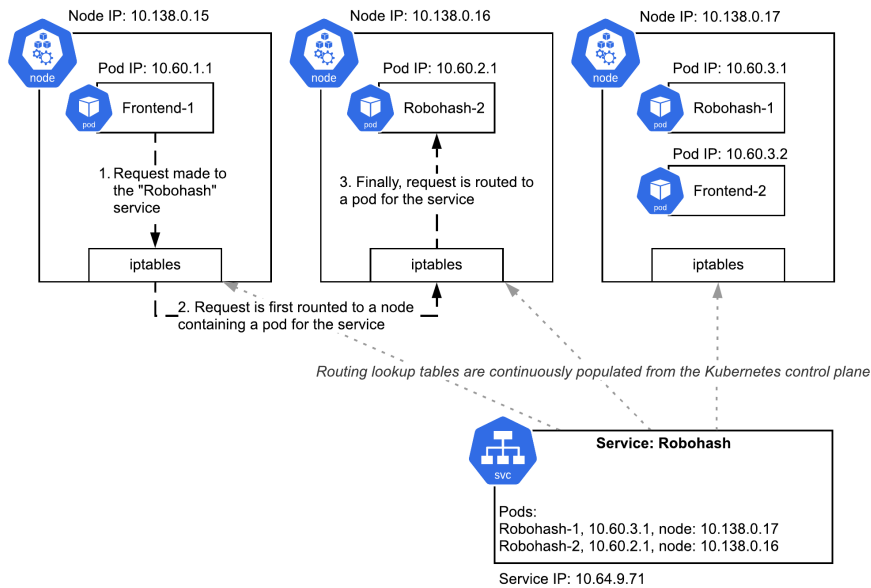


Figure 7.1 IP routing for an internal service named “Robohash”. Frontend-1 Pod makes an internal request to the service. The iptables routing glue on the node has a list of Pods for the service which is supplied by the Kubernetes control plane, and selects the Pod named “Robohash-2” at random. The request is then routed to that Pod.

What this all means is that when it’s time for you to deploy an internal service, you can achieve this by creating a Service, of type ClusterIP thereby obtaining an IP address that the other services in your cluster (like your app’s frontend) can communicate with seamlessly. This IP address automatically balances the load between all Pod replicas of the internal service. You don’t need to worry about all the networking glue that makes this possible, but hopefully this section has given you at least a superficial understanding on how it works.

7.1.2 Creating an Internal Service

Now that you hopefully understand a bit about how Kubernetes networking works under the hood, let’s build an internal service that can be used by other Pods in the cluster.

As an example, let’s deploy a new internal service to our app. For this I’m going to use a neat open source library called “Robohash” that can generate cute robot avatars for users based on a hash (like a hash of their IP). For your own deployments, internal services can be things as simple as avatar generators, other parts of your application, or even entire database deployments.

Here’s the deployment for this new container:

Listing 7.1 InternalServices/robohash-deploy.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: robohash
spec:
  replicas: 1
  selector:
    matchLabels:
      app: robohash
  template:
    metadata:
      labels:
        app: robohash
    spec:
      containers:
        - name: robohash-container
          image: wdenniss/robohash:v1

```

This time, instead of exposing this service to the world with a Service of type LoadBalancer, we'll keep it internal with a NodePort service. Why NodePort type instead of ClusterIP? Well, it turns out each NodePort type service also gets a cluster IP, so it's the best of both worlds. Plus, NodePort services can be used with Ingress (covered in the next section), while ClusterIP ones cannot, so that's another advantage of choosing NodePort as the default. I setup my internal services typically with type NodePort, but will still often reference them via their assigned cluster IP as I find it a closer analogy to how you would host multiple services outside of Kubernetes.

Here's the internal service definition for our robohash deployment:

Listing 7.2 InternalServices/robohash-service.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: robohash-internal
spec:
  selector:
    app: robohash
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
  type: NodePort

```

Since this isn't a LoadBalancer type service like we used in Chapter 3, it doesn't have an external IP. If you want to test it, you can use `kubectl port-forward` like so:

```
kubectl port-forward service/robohash-internal 8080:80
```

Now you can browse to <http://localhost:8080> on your local machine and check out the service. To generate a test avatar, try something like <http://localhost:8080/example>. You'll get an auto-generated robot avatar like:



Figure 7.2 Example robot avatar (attribution: Robohash.org)

Next, let's use this internal service from another service—our front end—and build out our microservice architecture!

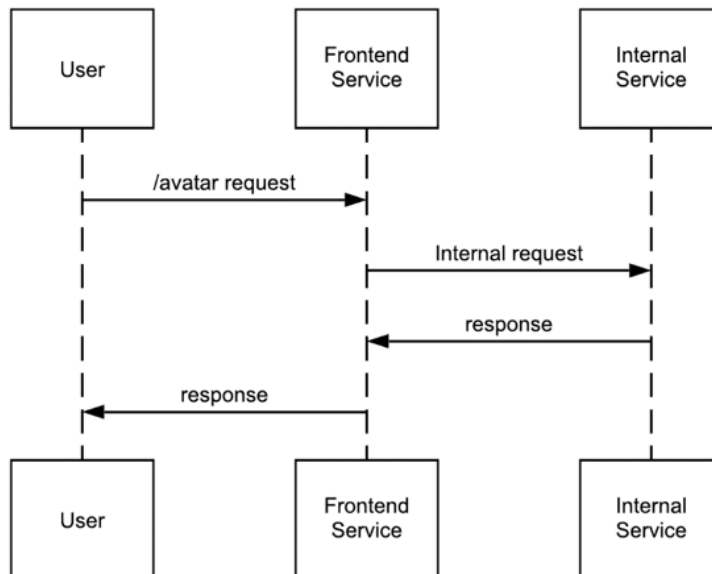


Figure 7.3 UML sequence diagram of a simple microservice configuration

To access this internal service from other Pods, you can reference its cluster IP. As the type is NodePort, you can also access it over localhost on the assigned port. To view the cluster IP and ports assigned, query the service like so:

```
$ kubectl get service
NAME          TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)
robohash-service  NodePort    10.63.254.218 <none>         80:32473/TCP
```

In this case, you can query the service from other pods on the given cluster IP (seen as 10.63.254.218 in the output above), such as by making a HTTP GET request to `http://10.63.254.218/example`. Or, using the node port of the service (seen as 32473 in the output above) you can also call `http://localhost:32473/example` from any node in the cluster. These paths only work from other Pods inside the cluster. There's no difference in the outcome if you use cluster IP or the node port to call the service from other Pods, but from here on I'll use the cluster IP. I personally prefer to think about services mapping to IPs, as traditionally "localhost" meant the service was running on the same host, but it really doesn't matter which you choose.

7.1.3 Service Discovery

In the previous example we used `kubectl get service` to look up the internal cluster IP address assigned to our service. While you could simply take that IP address and hardcode it into your application, doing this isn't great for portability. You may wish to deploy the same application in a few different places, like locally on your development machine, in a staging environment and in production (how to set up these different environments is covered in the next chapter), and this approach will mean you need to update your code every time.

Better is to discover the IP address dynamically in from the Pod that needs to call the service, much like we discovered the IP address using `kubectl`. Kubernetes offers Pods two ways to perform service discovery, using a DNS lookup, or an environment variable. The DNS lookup works cluster-wide, while the environment variable is only for Pods within the same namespace.

SERVICE DISCOVERY USING ENVIRONMENT VARIABLES

Kubernetes automatically creates an environment variable for each service and populates it with the Cluster IP, and makes this available in every Pod that is created after the service is created. The variable follows a naming conversion whereby our example `robobhash-internal` service gets the environment variable `ROBOHASH_INTERNAL_SERVICE_HOST`.

Rather than figuring out the correct conversion, you can view a list of all such environment variables available to your Pod by running the `env` command on you Pod with `exec`, like so (with truncated output):

```
$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
robobhash-6c96c64448-7fn24         1/1     Running   0           2d23h

$ kubectl exec robobhash-6c96c64448-7fn24 -- env
ROBOHASH_INTERNAL_PORT_80_TCP_ADDR=10.63.243.43
ROBOHASH_INTERNAL_PORT_80_TCP=tcp://10.63.243.43:80
ROBOHASH_INTERNAL_PORT_80_TCP_PROTO=tcp
ROBOHASH_INTERNAL_SERVICE_PORT=80
ROBOHASH_INTERNAL_PORT=tcp://10.63.243.43:80
ROBOHASH_INTERNAL_PORT_80_TCP_PORT=80
ROBOHASH_INTERNAL_SERVICE_HOST=10.63.243.43
```

The benefit of this approach is that it's extremely fast. Environment variables are just string constants, with no dependencies external to the pod itself. It also frees you to specify any DNS server you like to serve the other requests of the pod (e.g., 8.8.8.8).

The downside is that only those services in the same namespace of the Pod are populated in environment variables, and that ordering matters: the service must be created before the Pod, for the Pod to get the service's environment variable.

If you find yourself in a situation where you need to restart your Pods to pick up changes to the service, you can do this with the following command (no change to the Pod needed):

```
kubectl rollout restart deployment DEPLOYMENT_NAME
```

One common way to reference these variables is to provide the complete service HTTP endpoint of the internal service in its own environment variable defined in the Deployment. This allows your container to be even more portable and be able to run outside of Kubernetes (e.g. in Docker Compose). Here's how you can embed the value of the automatically generated environment variable (ROBOHASH_INTERNAL_SERVICE_HOST) in your own custom environment variable (AVATAR_ENDPOINT) which your application will ultimately consume:

Listing 7.3 InternalServices/timeserver-deploy-env.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 1
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:1
          env:
            - name: AVATAR_ENDPOINT
              value: http://$(ROBOHASH_INTERNAL_SERVICE_HOST)
```

Using this additional layer of indirection, where our custom environment variable references the Kubernetes one is useful as now we can run this container standalone in docker (just populate AVATAR_ENDPOINT with the endpoint of the internal service wherever it's running), or switch to DNS-based lookups.

In summary, environment variable discovery has a few advantages:

- Super fast (they are string constants)
- No dependency on other the DNS Kubernetes component

And some disadvantages:

- Only available to pods in the same namespace
- Pods must be created after the service is created

SERVICE DISCOVERY USING DNS

The other way to discover services is via the cluster's internal DNS service. For services running in a different namespace to the pod, this is the only option for discovery. The service name is exposed as a DNS host, so you can simply do a DNS lookup on `robohash-internal` (or use `http://robohash-internal`) as your http path, and it will resolve. When calling the service from other namespaces, append the namespace, e.g. `robohash-internal.default` to call the the service `robohash-internal` in the `default` namespace.

The only downside to this approach is that it's a little slower to resolve that IP address, as a network request is needed. In many Kubernetes clusters, this network service will be running on the same node so it's pretty fast, in others it may require a hop to the DNS service running on a different node, or a managed DNS service, so be sure to cache the result.

Since we previously made the endpoint URL an environment variable of the deployment, we can easily update the variable, this time giving it the service name (`http://robohash-internal`). The complete deployment will look like so:

Listing 7.4 InternalServices/timeserver-deploy-dns.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 1
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:5
          env:
            - name: AVATAR_ENDPOINT
              value: http://robohash-internal
```

In summary, DNS-based service discovery has a few advantages:

- Can be called from any namespace in the cluster
- No ordering dependencies

And some disadvantages:

- Slightly slower than using an environment variable (which is a constant)
- Dependency on the internal DNS service

So that is two ways how our front-end service can discover the internal service's internal Pod IP, rather than having that IP address hardcoded. Since these discovery methods are Kubernetes-specific, it's recommended to supply the path as an environment variable to the container as we did in the example, so then you can easily supply a completely different path when running the container outside of Kubernetes.

PUTTING IT ALL TOGETHER

Let's make a call from the timeserver app to our internal robohash service on a new endpoint /avatar. All this new endpoint does is to read an image from the internal service, and return it.

Listing 7.5 /timeserver5/server.py

```
import urllib.request
import os
import random

# ...

case '/avatar':
    url = os.environ['AVATAR_ENDPOINT'] + "/" + str(random.randint(0, 100))
    try:
        with urllib.request.urlopen(url) as f:
            data = f.read()
            self.send_response(200)
            self.send_header('Content-type', 'image/png')
            self.end_headers()
            self.wfile.write(data)
    except urllib.error.URLError as e:
        self.respond_with(500, e.reason)

# ...
```

Now that our application actually uses the internal service, we can deploy it all to Kubernetes.

```
$ cd Chapter07/7.1_InternalServices
$ kubectl create -f robohash-deploy.yaml
$ kubectl create -f robohash-service.yaml
$ kubectl create -f timeserver-deploy-dns.yaml
$ kubectl create -f timeserver-service.yaml
$ kubectl get svc/timeserver
NAME         TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
timeserver   LoadBalancer  10.22.130.155    34.127.43.33     80:32131/TCP     4m25s
$ open "http://34.127.43.33/avatar"
```

Wait for the external IP to be provisioned, then try out the /avatar URL. You should be greeted with a robot avatar.

We are now using microservice architecture for the demo! Using this technique, you can have multiple internal services that can be deployed and managed separately (perhaps by different teams). You can add in separate services using open source tooling, or simply bring together different components of your application written in different languages.

7.2 Ingress: HTTP(S) Load Balancing

So far in the book we’ve been using Services of type LoadBalancer. This provides you a so-called level 3 (L3) load balancer, which balances requests at the network level, and can work with a variety of protocols (TCP, UDP, SCTP). You configure the Service with your desired protocol and port, and you get an IP that will balance traffic over your Pods. If you expose a HTTP service over a LoadBalancer, you need to implement your own TLS termination handling (configuring certificates, and running a HTTPS endpoint), and all traffic to that endpoint will get routed to one set of Pods (based on the matchLabels rules). There is no option for exposing two or more separate services directly on the same load balancer (though one can proxy requests to the other internally).

When you are publishing a HTTP app specifically, you may get more utility from a so-called level 7 (L7) load balancer, which balances at the HTTP request level and can do more fancy things like terminate HTTPS connections (meaning it will handle the HTTPS details for you), and perform path-based routing, so you can serve a single domain host with multiple services. In Kubernetes, a HTTP load balancer is created with an Ingress object.

Ingress lets you place multiple internal services behind a single external IP with load balancing. You can direct HTTP requests to different backend services based on their URI path (/foo, /bar), hostname (foo.example.com, bar.example.com), or both. The ability to have multiple services running on a single IP (and potentially serving different paths under a single domain name) is unique to Ingress, because if you’d exposed them with individual Services of type LoadBalancer like in the earlier chapter, the services would have separate IP addresses, necessitating separate domains (e.g. one Service hosted on foo.example.com, and the other on bar.example.com).

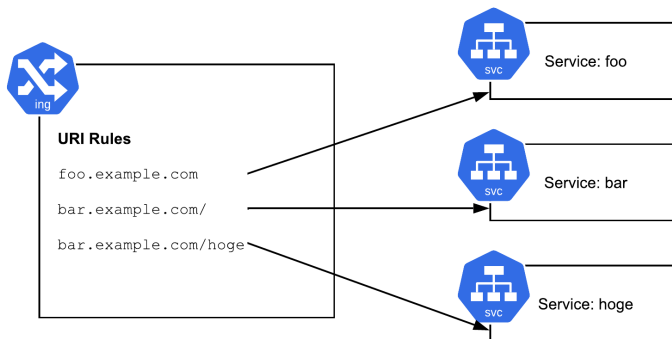


Figure 7.4 the ingress' rule list, or "URL Map" allows one HTTP load balancer to handle the traffic for multiple services.

The property of Ingress being able to place multiple services under one host is useful when scaling up your application. When you need to break up your services into multiple services for developer efficiency (teams wanting to manage their own deployment lifecycle), or

scaling (being able to scale aspects of the application separately), you can use Ingress to route the requests while not altering any public-facing URL paths. For example, let's say that your application has a path that is a particularly CPU-intensive request. You might wish to move it to its own service to allow it to be scaled separately. Ingress allows you to make such changes seamlessly to end users.

Here's an example Ingress where the routes are served by different backends. In this example, we'll run an internal Timeserver service on the root path (/), and we'll expose the internal Robohash service from the previous section on /robohash.

Listing 7.6 Ingress/ingress_path.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: timeserver-ingress
spec:
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: timeserver-internal
            port:
              number: 80
      - path: /robohash
        pathType: Prefix
        backend:
          service:
            name: robohash-internal
            port:
              number: 80
```

Here's a variation using different hosts. Each of these hosts can also have multiple paths using the format above.

Listing 7.7 Ingress/ingress_host.yaml

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: timeserver-ingress
spec:
  rules:
    - host: timeserver.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: timeserver-internal
                port:
                  number: 80
    - host: robohash.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: robohash-internal
                port:
                  number: 80

```

The Services referenced in the Ingress object are the same Services discussed in Ch. 03, and in the prior section with one important consideration, they are all of type `NodePort`.

Listing 7.8 Ingress/timeserver-service-internal.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: timeserver-internal
spec:
  selector:
    pod: timeserver-pod
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
  type: NodePort

```

Ingress objects can be configured to perform exact matching (only requests exactly matching the path given will be routed to that service), or prefix matching (all requests matching the path prefix will be routed) with the `pathType` property. I'm not going to go into a lot of detail here, as the official docs do a great job. One facet worth reproducing is the rule on multiple matches:

In some cases, multiple paths within an Ingress will match a request. In those cases precedence will be given first to the longest matching path. If two paths are still equally matched, precedence will be given to paths with an exact path type over prefix path type.

Ingress – The Kubernetes Authors⁴

As you may have seen in the first example I presented, there was a path for `/`, and a second one for `/foo`. A request to `/foo` will be routed to the second service, even though it also matched the first path. If you've used other routing mechanisms in the past (like Apache URL rewriting), often the preference would go to the *first* rule matched. Not so in Kubernetes, where the longer matched rule gets preference. This design in Kubernetes is convenient, as it matches well with developer intent.

Deploying this example:

```
$ cd Chapter07/7.2_Ingress
$ kubectl create -f robohash-deploy.yaml
deployment.apps/robohash created
$ kubectl create -f robohash-service.yaml
service/robohash-internal created
$ kubectl create -f timeserver-deploy-dns.yaml
deployment.apps/timeserver created
$ kubectl create -f timeserver-service-internal.yaml
service/timeserver-internal created
$ kubectl create -f ingress_path.yaml
ingress.networking.k8s.io/timeserver-ingress created
will@macbook-pro 7.2_Ingress % kubectl get ing -w
NAME          CLASS    HOSTS    ADDRESS    PORTS    AGE
timeserver-ingress <none>   *       *          80       4s
timeserver-ingress <none>   *       34.110.232.121 80       100s
```

Once your ingress has an IP, you can browse to it. One thing to note is that the resources backing the ingress may take a bit of extra time to be provisioned. Even if you have the IP address, and you browse to it, you may see a 404 error for a time. I suggest trying again in about 5 minutes to give the cloud provider time to update the ingress.

To debug issues with the ingress, you can use `kubectl describe`. Here's what I see when I described the ingress shortly after it had an IP, and then again after waiting a few minutes.

⁴ <https://kubernetes.io/docs/concepts/services-networking/ingress/#multiple-matches>

```
$ kubectl describe ing
Name:          timeserver-ingress
Namespace:     default
Address:       34.110.232.121
Default backend: default-http-backend:80 (10.22.0.130:8080)
Rules:
  Host      Path  Backends
  ----      -
  *
          /      timeserver-internal:80 (10.22.0.135:80)
          /robobash robobash-internal:80 (10.22.1.4:80)
Annotations:  ingress.kubernetes.io/backends:
               {"k8s-be-32730--a52250670846a599": "Unknown", "k8s1-a5225067-default-robobash-internal-80-fa41c363": "Unknown", "k8s1-a5225067-default-timeser...
               ingress.kubernetes.io/forwarding-rule: k8s2-fr-21mgs2f1-default-timeserver-ingress-s2j1ki7x
               ingress.kubernetes.io/target-proxy: k8s2-tp-21mgs2f1-default-timeserver-ingress-s2j1ki7x
               ingress.kubernetes.io/url-map: k8s2-um-21mgs2f1-default-timeserver-ingress-s2j1ki7x
Events:
  Type    Reason    Age      From                      Message
  ----    -
  Normal  Sync      80s      loadbalancer-controller  UrlMap "k8s2-um-21mgs2f1-default-timeserver-ingress-s2j1ki7x" created
  Normal  Sync      78s      loadbalancer-controller  TargetProxy "k8s2-tp-21mgs2f1-default-timeserver-ingress-s2j1ki7x" created
  Normal  Sync      70s      loadbalancer-controller  ForwardingRule "k8s2-fr-21mgs2f1-default-timeserver-ingress-s2j1ki7x" created
  Normal  IPChanged 70s      loadbalancer-controller  IP is now 34.110.232.121
  Normal  Sync      66s (x4 over 2m46s)      loadbalancer-controller  Scheduled for sync

$ kubectl describe ing
Name:          timeserver-ingress
Namespace:     default
Address:       34.110.232.121
Default backend: default-http-backend:80 (10.22.0.130:8080)
Rules:
  Host      Path  Backends
  ----      -
  *
          /      timeserver-internal:80 (10.22.0.135:80)
          /robobash robobash-internal:80 (10.22.1.4:80)
Annotations:  ingress.kubernetes.io/backends:
               {"k8s-be-32730--a52250670846a599": "HEALTHY", "k8s1-a5225067-default-robobash-internal-80-fa41c363": "HEALTHY", "k8s1-a5225067-default-timeser...
               ingress.kubernetes.io/forwarding-rule: k8s2-fr-21mgs2f1-default-timeserver-ingress-s2j1ki7x
               ingress.kubernetes.io/target-proxy: k8s2-tp-21mgs2f1-default-timeserver-ingress-s2j1ki7x
               ingress.kubernetes.io/url-map: k8s2-um-21mgs2f1-default-timeserver-ingress-s2j1ki7x
Events:
  Type    Reason    Age      From                      Message
  ----    -
  Normal  Sync      4m48s    loadbalancer-controller  UrlMap "k8s2-um-21mgs2f1-default-timeserver-ingress-s2j1ki7x" created
  Normal  Sync      4m46s    loadbalancer-controller  TargetProxy "k8s2-tp-21mgs2f1-default-timeserver-ingress-s2j1ki7x" created
```

Normal	Sync	4m38s	loadbalancer-controller	ForwardingRule "k8s2-fr-21mgs2f1-default-timeserver-ingress-s2j1ki7x" created
Normal	IPChanged	4m38s	loadbalancer-controller	IP is now 34.110.232.121
Normal	Sync	2m44s (x6 over 6m14s)	loadbalancer-controller	Scheduled for sync

Notice how the annotation changed from "Unknown", to "HEALTHY". After that, I was able to visit the IP.

Cost Saving Tip: Saving IPs with Ingress

A benefit of Ingress is that by using host-based routing, you can host several services all with the same external IP address. The ingress inspects the `Host` header in the HTTP request, and routes traffic accordingly. This contrast with Services of type `LoadBalancer` which each get their own IP address assigned and perform no packet inspection or routing.

Cloud providers often charge based on "load balancing rules", which roughly translates into how many load balancing external IP addresses are assigned. By using an ingress to combine several services into one, rather than each being exposed with it's own IP, you can likely save money.

If your cloud provider groups HTTP Loadbalancers (Ingress) and Network Load Balancers (service of type `LoadBalancer`) separately, and has a minimum rule fee (like Google Cloud does as the time of writing), you may want exclusively use one or the other until you need more than the minimums.

Another trick, but one I don't recommend, is running your own *Ingress controller*. This technique (not covered in this book), means deploying an open source component as a `LoadBalancer` to implement the Kubernetes Ingress functionality, overriding the default implementation of your cloud provider. This approach means that the Ingress objects and `LoadBalancer` objects get treated as the same rule types for billing which can save money if you need both, but there's a sacrifice which is that you now need to manage this component yourself. Are you an expert at debugging Kubernetes Ingress controllers? Better to go all-in using standard Ingress objects, or stick with pure `LoadBalancers` if you need to save money, in my experience.

7.2.1 TLS

Another useful property of Ingress is that it will perform the TLS encryption for you. Modern web applications are typically hosted as secure HTTPS applications with TLS which is important for security, but comes with some overhead for the application server. Depending on the server middleware you are using, you may see performance gains by letting the Ingress load balancer handle the TLS connection (a so-called TLS terminator), and communicate on to the backend only over HTTP (via a secure network like that of your cloud provider, of course). If you prefer, the Ingress can re-encrypt traffic and connect to your services over HTTPS, but there is no option to pass the unmodified encrypted traffic directly through from the client to the backend (for that, you'd use a Service of type `LoadBalancer`, like we did in Chapter 3).

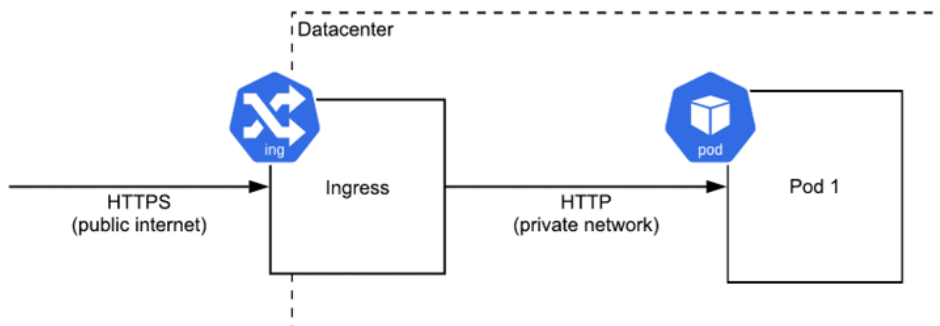


Figure 7.5 The Ingress terminates HTTPS (TLS) traffic, and can forward it to the serving Pods over plain-HTTP or HTTPS connections.

Now that the Ingress is terminating your TLS connections, you'll need to set it up with certificates. If like me you've done this a few times on different systems, you might be dreading this step. Fortunately, Kubernetes makes it a breeze!

You just need to import your certificate and key as a Kubernetes Secret, then reference that secret in your Ingress configuration. A Kubernetes Secret is simply a data object in your cluster used to contain things like TLS keys.

To do this, normally you would follow the instructions of your certificate authority to create a certificate, the end product of which would include the two files we need: the private key that you created, and the certificate issued by the certificate authority.

For demonstration purposes we can create our own self-signed certificate in lieu of a trusted certificate. Note that while this will provide the same encryption for the connection, there is no identity verification, and you'll see scary messages in your browser. The following commands will create such a certificate.

```
$ create a private key
openssl genrsa -out example.key 2048

$ create a certificate request for 'example.com'
openssl req -new -key example.key -out example.csr \
  -subj "/CN=example.com"

$ self-issue an untrusted certificate
openssl x509 -req -days 365 -in example.csr -signkey \
  example.key -out example.crt
```

Once you have your private key and certificate (whether you created them with the above instructions, or by following the instructions of your certificate authority), you can now create the Kubernetes secret like so:

```
kubectl create secret tls my-tls-cert \
  --cert example.crt --key example.key
```

You may notice the imperative `kubectl create` command here. This is one of the few times I recommend using an imperative command rather than defining the configuration in a file because it's simpler than creating the object manually and base64-encoding all the data. If you want to see the config that got created with this command, you can easily view it with `kubectl get -o yaml secret my-tls-cert`.

The final step is to reference this secret in our ingress.

Listing 7.9 TLS/ingress_tls.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: timeserver-tls
spec:
  tls:
    - secretName: my-tls-cert
  rules:
    - host: example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: timeserver-internal
                port:
                  number: 80
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: robohash-internal
                port:
                  number: 80
```

With the deployment and services created in the previous section, we can create this new ingress with a TLS secret like so:

```
$ kubectl create -f ingress_tls.yaml
ingress.networking.k8s.io/timeserver-tls created
$ kubectl get ing
NAME          CLASS    HOSTS          ADDRESS          PORTS    AGE
timeserver-tls <none>    example.com    34.110.232.121   80, 443   9m15s
$ open "https://34.110.232.121"
```

Remember that the provisioning step can take a while, even from the point where the ingress has received an IP. As this is a self-signed certificate, you will see some scary warnings in the browser.

To test out the domain name route in this ingress ("example.com" in the example), you'll need to configure the DNS for the domain you've used with the IP of the ingress. To test locally, you can also edit your local hosts file and add the IP and domain name (to find instructions on how to do that, a Google search for "update hosts file in <your operating system version>" should do the trick!). The IP of the ingress can be found with `kubectl get ingress`.

Here's what my ingress object looks like, and the entry I added to my local hosts file:

```
$ kubectl get ingress
NAME                CLASS      HOSTS          ADDRESS          PORTS          AGE
timeserver-ingress <none>    example.com    203.0.113.2     80, 443        82m

$ cat /etc/hosts
# ...
203.0.113.2 example.com
```

Now, provided that you've configured your host, you should be able to browse to <https://example.com>. If you generated a self-signed certificate, you'll get a scary browser error which in this case is fine to click through. To actually publish your service to the world, you'll want to go back and request a certificate from an actual certificate authority, and use that to create the TLS secret instead.

Once again, the nice thing about Kubernetes is that all this configuration is in the form of Kubernetes objects (rather than random files on a host VM), making it straight forward to reproduce the environment elsewhere.

Using GKE? Try Managed Certificates

The above instructions are for adding a tried-and-true CA certificate to your Kubernetes Ingress object. If you're using GKE, and want an even simpler approach you can use a managed certificate instead.

With a managed certificate, you skip the CA signing step, and the copying of your private key and certificate into Kubernetes as a Secret. Instead, you need to first prove ownership of the domain to Google (this is done in the Google Cloud console), create a GKE-specific `ManagedCertificate` object listing the (sub)domains you wish to provision certificates for, then reference that object in your ingress. Google will then provision and manage the certificates automatically. It's all pretty straight forward, so I'll let the official docs² be your guide for this one.

7.3 Summary

- Kubernetes offers several tools to create, discover, connect and expose multiple services, for when your requirements exceed what can be hosted in a single container.
- Internal services are a way to connect together a wide range of workloads that can be written in a different language, be on a different release schedule or simply need to scale independently
- Internal services can be exposed on a private cluster IP that allows them to be called from other services in the cluster
- Kubernetes offers two forms of service discovery to find these internal service IPs: environment variables and DNS
- Ingress can be used to expose multiple internal services to the internet using a single IP, with routing performed by path and/or host name

² <https://cloud.google.com/kubernetes-engine/docs/how-to/managed-certs>

- Ingress is a “L7” HTTP load balancer that can also handle your HTTPS connections and TLS termination
- By performing TLS termination at the load balancer layer, you can save configuration effort in your application and reduce CPU overhead

8

Node Feature Selection

This chapter covers:

- **Selecting nodes with specific hardware properties**
- **Using taints and tolerations to prevent scheduling by default on special nodes**
- **Keeping workloads separated on discrete nodes**
- **Avoiding a single point of failure with a highly available deployment strategy**
- **Targeting and avoiding specific groups of nodes for deployments**

So far this book has treated the compute nodes in the cluster—the machines responsible for actually running your containers—as equal. Different Pods may request more or less CPU, but they're all running on the same type of nodes under the hood.

One of the fundamental properties of cloud computing is that even when you're using an abstract platform that takes care of much of the low-level compute provisioning for you (as Kubernetes platforms are capable of doing), you may still care to some extent about the servers that actually running your workloads. Serverless is nice, but at the end of the day, the workload is running on a computer, and you can't always escape the properties of that machine, nor do you always want to.

This is when node features selection comes in. In a managed platform, including GKE, there are dozens of different hardware and operational properties of the nodes. The node CPU can be of the x86 architecture, or Arm. It might be AMD, or Intel. Nodes can have expensive hardware like GPU attached, if you need it, or they can be run in low-priced Spot provisioning mode saving you money, at the risk of workload disruption. You may not always need to care about these elements, but when you do it can be useful (like saving money with Spot), or critical (needing a GPU for a machine learning inference workload).

Another impact of nodes in Kubernetes is that Kubernetes runs multiple Pods on the same node (so-called binpacking). This is great for cost reasons, hosting multiple containers on the same host and is especially useful when you use Bursting to run several workloads in a manner

where they can burst into each other's reserved capacity. The downside of course is the potential for single-points of failure. Fortunately, Kubernetes ships with some built-in methods to avoid concentrations of the same pod on a node, in the form of Pod Spread Topology. In this chapter, you'll learn how to select nodes for their features, group pods together, or spread them apart.

8.1 Node Feature Selection

Not all compute nodes are equal. You may have workloads that require additional hardware like higher-performance CPUs, GPUs, or properties like running in provision model like "Spot" with lower costs, but the chance of abrupt termination. Some nodes run Linux while others run Windows. Some CPUs are the x86, others are Arm and so on. Just as in the past we might place workloads on machines with specific features, we can do the same in Kubernetes through node selection and affinity.

8.1.1 Node Selectors

The way these node features are differentiated in Kubernetes is through node labels. And the way you target specific node features from your pods is with node selection and/or node affinity. Node selection and affinity are simply ways to express the desired labels (and therefore features) of the nodes that your Pods require.

Take for example a Pod that needs to run on an Arm node. Arm nodes are labelled with the well-known label `kubernetes.io/arch: arm64` (well-known labels are those that are defined in the open source and should be constant across different providers). We can use a `nodeSelector`, or node affinity to target that label, and ensure our Pod will only run on an Arm node.

The following deployment selects the arm64 architecture, to prevent the Pod being scheduled on any other type of architecture:

Listing 8.1 NodeSelection/deploy_nodeselector.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 1
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      nodeSelector:
        kubernetes.io/arch: arm64
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:3
```

A more verbose way to express the exact same requirement is through a node affinity.

Listing 8.2 NodeSelection/deploy_nodeaffinity.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 1
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: kubernetes.io/arch
                    operator: In
                    values:
                      - arm64
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:3
```

These previous two deployment configurations will achieve the exact same result: a pod placed only on an Arm node. The advantage of the nodeAffinity, and the reason you would use it is that it allows for more expressive logic, about which I will go into more detail in the next section.

Requiring these feature-related node labels in your Podspecs is the first step, but you need a way to actually have nodes provisioned with that functionality (i.e. having the labels you are selecting). As always, the provisioning of nodes and their associated features are done at the platform level. If you are using a fully-managed service like GKE Autopilot, then simply specifying your nodeSelector with your feature labels is enough to get a node that has those capabilities, provided the capability is offered by the platform. On a more traditional Kubernetes platform, you would need to provision nodes that will have those features out of band, for example by creating a node pool or node group with the desired properties.

To find out what capabilities are supported, the provider's docs are best but if you have a node in the cluster with the desired properties you can also inspect it and see what labels are available for selection.

Kubect1 describe nodes

Here is the relevant output from GKE for a node that was created as a spot VM.

```

Name:          gke-cluster-1-pool-1-c7d036dd-hmvt
Roles:         <none>
Labels:        beta.kubernetes.io/arch=amd64
               beta.kubernetes.io/instance-type=e2-standard-8
               beta.kubernetes.io/os=linux
               cloud.google.com/gke-nodepool=pool-1
               cloud.google.com/gke-os-distribution=cos
               failure-domain.beta.kubernetes.io/region=us-central1
               failure-domain.beta.kubernetes.io/zone=us-central1-c
               kubernetes.io/arch=amd64
               kubernetes.io/hostname=gke-cluster-1-pool-1-c7d036dd-hmvt
               kubernetes.io/os=linux
Annotations:   container.googleapis.com/instance_id: 7069679921314020139
               node.alpha.kubernetes.io/ttl: 0
               volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp: Wed, 15 Jul 2020 15:41:18 -0700

```

8.1.2 Node Affinity and Anti-Affinity

NodeAffinity is very expressive, and can do a lot more than require a list of labels like nodeSelector. With the “In” operator for example, you can specify a list of possible values. Let’s say that you want to select either x86 or Arm as the architecture, you can do that using node affinity, by providing a list with the In operator.

Listing 8.3 NodeAffinity/deploy_nodeaffinity_multi.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 1
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: kubernetes.io/arch
                    operator: In
                    values:
                      - arm64
                      - amd64
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:3

```

Note that you cannot express this “OR” logic using nodeSelector. NodeSelector conditions are ANDed together, and since each label can only have 1 key, it isn’t valid to select on two

different keys for the same label (as this can never be satisfied). For example, the following is NOT a valid replacement for the `In` operator used above:

```
# Invalid
nodeSelector:
  kubernetes.io/arch: arm64
  kubernetes.io/arch: amd64
```

The **operator** logic is also very expressive and can be used to turn the expression into one of anti-affinity (i.e. avoid nodes with the given labels) with the opposite of `In`: `NotIn`. Supported operators include

<code>In</code>	The value of the node label is one of the options given.
<code>NotIn</code>	The value is not present in the list you supply.
<code>Exists</code>	The label key is present on the node (with any value)
<code>DoesNotExist</code>	The label key is not present on the node
<code>Gt</code>	The value given is greater than that which is in the node label.
<code>Lt</code>	The value given is less than that which is in the node label.

PREFERRED NODE AFFINITY

Another benefit of `nodeAffinity` is that you can create “preferred” rather than “required” rules to express a set of preferences. For example, if your container is multi-architecture, and can run on x86 or Arm, but you prefer to use Arm if possible (for example, for cost reasons), then you can express that as follows.

Listing 8.4 NodeAffinity/deploy_nodeaffinity_preferred.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 1
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 100
          preference:
            matchExpressions:
              - key: kubernetes.io/arch
                operator: In
                values:
                  - arm64
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:3

```

Caution with preferred affinity

This `preferredDuringSchedulingIgnoredDuringExecution` logic may sometimes yield surprising results. While the preference ordering works when you have existing unallocated capacity on nodes, the way it interacts with cluster autoscaling when there is no unallocated capacity of the preferred type and a new node is needed might be contrary to what you prefer. For example, in the event that there is any unallocated capacity on existing nodes in your cluster, even if it is of the dis-preferred type, Kubernetes will actually schedule the Pod there first, before the platform kicks in to add new nodes of your preferred type.

The reason for this is that the Kubernetes scheduler, responsible for placing Pods on nodes and the platform Autoscaler (a common platform component responsible for adding new nodes) are operating somewhat separately. The way a typical node Autoscaler works at the platform level is to look for Pending pods that can be scheduled if more capacity was added. But since the Kubernetes scheduler kicks in first and places the pod on the dis-preferred but available capacity, the Autoscaler doesn't have a chance to act.

My advice when using a cloud provider is that you can generally just require the functionality you need, and rely on the fact that they will have capacity to serve those needs. An alternative method to reduce the instances of dispreferred capacity being used is to minimize the amount of dispreferred capacity that the workload can use by separating it from other workloads in the cluster. To do this, you can use a required selector to a set of nodes that are tainted, just like earlier in the chapter, except in this case the nodes are tainted just to keep the workload separate. (Tainted nodes are described in the next section.)

8.1.3 Tainting Nodes to Prevent Scheduling by Default

Another common requirement when you have a group of nodes that have special characteristics is that Pods should not be scheduled by default on these nodes. Take Arm for example: since Arm is relatively new and not all container images yet support it, you may want to configure your cluster so that Arm nodes will not be used for scheduling by default, unless the workload expressly indicates support. Other examples include when you have a Node with special hardware like a GPU that you need to reserve only for Pods that will use this hardware, and when you have Spot compute that can be shutdown abruptly, which not all workloads may respond well to.

While you *could* annotate every other Pod to avoid such nodes using node anti-affinity (that is, a node affinity rule with the `NotIn` operator), that is laborious, so Kubernetes has the ability to so-called “taint the node” to prevent Pods being scheduled on it *by default*. How it works is that you “taint” nodes that have special characteristics and shouldn’t be scheduled on by default, then you “tolerate” this taint in the Podspec of just those workloads that are OK to run on these nodes.

By way of example, we can taint nodes individually to see the effect. This isn’t how you normally do it in production, but is a decent way to experiment. For this demo, we can use minikube from Chapter 3, and taint one of the nodes as follows:

```
minikube create --nodes 3
kubectl get nodes

kubectl taint nodes NODE_NAME spot=true:NoSchedule
kubectl taint nodes NODE_NAME spot-
```

In this example, `spot=true` is the name we gave to the taint, and is used later when marking Pods as able to tolerate this taint. The `NoSchedule` which follows indicates the desired behavior of the effect of this taint (being that Pods without the toleration should not be scheduled). There are alternatives to the `NoSchedule` behavior, but I do not recommend them. `PreferNoSchedule` is an option that creates a soft rule which may sound useful, but if your primary goal is to avoid scheduling pods on classes of nodes, a soft rule would not achieve that and may make it harder to debug. Sometimes it’s better to have an unscheduled Pod that you need to allocate resources for, than having it scheduled on your special tainted machines and cause other unspecified issues.

When you’re operating a hosted Kubernetes service, it’s unlikely that you’ll be tainting nodes individually like in the above example. That’s because generally a taint applies to a group of nodes (ones that share the same characteristic like Arm, or Spot, or GPU), and because nodes are regularly replaced during upgrade or repair events. Look for the platform provider’s API that allows you to taint groups of nodes, so that the taint will be applied to all nodes in the group, and persist during upgrades.

Node Tainting In GKE

For GKE Autopilot node tainting is completely automatic. When you select for a particular (non-default) feature like Spot compute, or Arm, the nodes that are provisioned are automatically tainted. Conveniently the Pods are also modified to tolerate the automatic taint, so all you need to do is select for the feature. This automatic modification of the Pods is done through an admission controller (which are covered in Chapter 11) that is installed and maintained by the platform.

In GKE Standard, you can taint node pools when you create them. For example, you if you're creating a spot node pool, you can configure all the nodes to be tainted as follows

```
gcloud container node-pools create NODE_POOL_NAME --cluster CLUSTER_NAME \
  --spot --node-taints spot=true:NoSchedule
```

When you taint node pools like this, it would normally be in cases where you have non-tainted (e.g. non-spot) nodes. If your entire cluster was to consist of spot nodes, the taint would not normally be used.

Once you have tainted nodes, if you now schedule a workload, you'll notice that it won't be scheduled on these nodes (use `kubectl get pods -o wide` to see which nodes the Pod lands on). In order to make the workload schedulable on the node you just tainted, the workload will need to be updated.

Listing 8.5 Taints/deploy_tolerate_spot.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 3
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      tolerations:
        - key: spot
          value: "true"
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:1
```

The toleration won't cause the Pod only to be scheduled on the tainted node; it only allows it to be. Where the Pod is scheduled will be determined based on a few other factors, like available capacity. Thus, Pods with the toleration can land on untainted nodes, as well as nodes with taints that they tolerate, as shown in Figure 5.11.

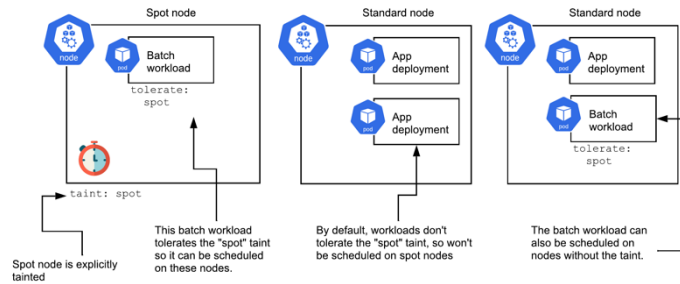


Figure 8.1: This cluster has a lower-availability Spot VM, and 2 standard nodes.

Commonly, you will combine both taints and tolerations with NodeSelectors or Node Affinity to ensure that a particular set of Pods and only that set of Pods will be run on nodes in question. A good example of where that matters is for GPU workloads: these workloads must only be run on a node with a GPU, and you don't want non-GPU workloads taking up that valuable space, as shown in figure 5.12

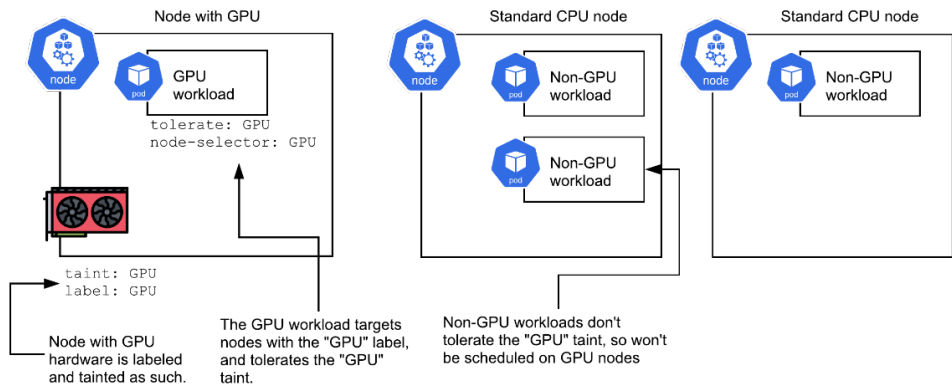


Figure 8.2: This cluster has a special node with a GPU, and 2 standard nodes.

The GPU node is tainted to prevent standard workloads from being scheduled on it. The GPU workload tolerates the taint, so can be scheduled on the GPU node, and uses a nodeSelector to ensure it is *only* scheduled on this node.

TOLERATING ALL TAINTS

Some workloads, most commonly those deployed as DaemonSets (covered in Chapter 11), need to run on every node, and must be designed to handle all the configuration of the cluster. Such workloads typically tolerate all taints, as the following example demonstrates.

Listing 8.6 Taints/daemonset_tolerate_all_taints.yaml

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: example-ds
spec:
  selector:
    matchLabels:
      pod: example-pod
  template:
    metadata:
      labels:
        pod: example-pod
    spec:
      tolerations:
        - effect: NoExecute
          operator: Exists
        - effect: NoSchedule
          operator: Exists
      containers:
        - image: ubuntu
          command: ["sleep", "infinity"]
          name: ubuntu-container
```

Just be aware that when you do this, your Pod will actually need to run on all node types that may exist in the cluster now and in the future. This can be an issue when adding a feature like Arm nodes that requires containers to be specifically built for Arm. When such cases occur, you can combine the various techniques in this chapter to achieve the result of being schedulable on all nodes regardless of taints, *except* those with a specific label, such as Arm. This is done through combining the toleration, and a node-anti affinity rule, shown in the next example.

Listing 8.7 Taints/daemonset_tolerate_antiaffinity.yaml

```

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: example-ds
spec:
  selector:
    matchLabels:
      pod: example-pod
  template:
    metadata:
      labels:
        pod: example-pod
    spec:
      tolerations:
        - effect: NoExecute
          operator: Exists
        - effect: NoSchedule
          operator: Exists
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: kubernetes.io/arch
                    operator: NotIn
                    values:
                      - arm64
      containers:
        - image: ubuntu
          command: ["sleep", "infinity"]
          name: ubuntu-container

```

8.1.4 Workload Separation

Another use for taints, tolerations and node selectors is to separate workloads. So far the use-cases for node selection we've covered were around feature-based selection: requiring the Arm architecture, Spot compute, GPU nodes and the like.

Node selection isn't limited to node features, and also can be used to separate workloads from each other on nodes. While you can use pod anti-affinity (covered in the next section) to prevent particular pods being co-located, sometimes it helps just to keep workloads on their own dedicated groups of nodes.

One requirement for this I heard was from a customer who uses a cluster for a lot of batch workloads, consisting of a deployment of a coordinator that schedules the work, and the Pods for the workloads themselves. They preferred to keep the Pods for these two roles separate, so that any autoscaling of the nodes for the work Pods doesn't impact that of the coordinator Pods. Another example is for the noisy neighbor problem, where two Pods can potentially compete for resources on the node and would be better if separated.

Yet another example might be where you simply want your workload to land on a "fresh" node. If you give the workload a fresh workload separation label/toleration then you're forcing the platform to create a new node for it, which is generally on the current version of the cluster and therefore less likely to be disrupted with an upgrade in the near future.

To achieve workload separation, we can combine several of the techniques used so far along with a custom node label. The node gets a label and a taint, and the workload a toleration and selector for that label, which together means the workload will be scheduled on a group of nodes by itself (potentially shared with other workloads with the same selector and toleration).

Here is an example Deployment with an arbitrary toleration and node selector to achieve workload separation. For convenience, we will use the same key-value pair ("group=1") for both elements, though note that they are separate concepts in Kubernetes.

Listing 8.8 WorkloadSeparation/deploy_team1.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver1
spec:
  replicas: 5
  selector:
    matchLabels:
      pod: timeserver1-pod
  template:
    metadata:
      labels:
        pod: timeserver1-pod
    spec:
      tolerations:
        - key: group
          operator: Equal
          value: "1"
          effect: NoSchedule
      nodeSelector:
        group: "1"
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:1
```

And for demonstration we can make a copy of this deployment using "group=2" as the key/value for the toleration and selector:

```
tolerations:
  - key: group
    operator: Equal
    value: "2"
    effect: NoSchedule
nodeSelector:
  group: "2"
```

To have these deployments deployed on discrete sets of nodes, we'll need to have nodes that are tainted to prevent other pods landing on them, and labeled so our deployments can target them. If you miss the label, then these Deployments won't ever be scheduled as there won't be a node that meets the nodeSelector requirement. If you label, but don't taint the nodes, then these workloads will schedule, and be separated from each other (by way of the node selector), but other random Pods might land on them as well since there is no taint to keep them away.

Workload Separation on GKE Autopilot

If you deploy the above workload on GKE Autopilot, nodes with the requested labels and taints will be provisioned automatically! That's because this operationally nodeless platform is actuating on your Pod's requirements, and providing nodes that match, so there's nothing more you need to do. In a traditional Kubernetes platform where you are managing nodes (like GKE Standard, or minikube), you'll need to create nodes with these properties yourself.

In Kubernetes environments where you manage the nodes, you'll need to provide nodes with the correct taints and labels, to achieve the workload separation. Using minikube to demonstrate, we can taint and label nodes directly. Just note that on a managed platform, you typically operate on nodes at a "node pool" or group level, and would use a platform API to provide the nodes, so look for the label and taint parameters in that API.

```
$ minikube create --nodes 3 #A

$ kubectl get nodes #B
NAME          STATUS    ROLES          AGE   VERSION
minikube      Ready    control-plane  67s   v1.24.3
minikube-m02  Ready    <none>         46s   v1.24.3
minikube-m03  Ready    <none>         24s   v1.24.3

$ kubectl taint nodes minikube-m02 group=1:NoSchedule #C
$ kubectl label node minikube-m02 group=1 #C

$ kubectl taint nodes minikube-m03 group=2:NoSchedule #D
$ kubectl label node minikube-m03 group=2 #D
```

#A create a new minikube cluster

#B view the nodes

#C taint and label the m02 node for group 1

#D taint and label the m03 node for group 2

Both the taint and the label are required (as is the matching toleration and nodeSelector in the deployment) as they serve different purposes. The taint prevents all but those workloads that tolerate the taint from landing on it, while the label can be used to ensure the workload doesn't land on any other nodes (such as nodes without any taints). For convenience, I used the same key/value pair for both the taint and the label (e.g. "group=1"), but this doesn't have to be the case.

With our cluster configured, we can deploy our workload separated deployments and watch the result. Pay particular attention to which node the pods land on.

```
$ kubectl create -f Chapter08/8.4.1_WorkloadSeparation
deployment.apps/timeserver1 created
deployment.apps/timeserver2 created
```

```
$ kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	NODE
timeserver1-75b69b5795-9n7ds	1/1	Running	0	2m2s	minikube-m02
timeserver1-75b69b5795-kft64	1/1	Running	0	2m2s	minikube-m02
timeserver1-75b69b5795-mnc4j	1/1	Running	0	2m2s	minikube-m02
timeserver1-75b69b5795-msg9v	1/1	Running	0	2m2s	minikube-m02
timeserver1-75b69b5795-r8r9t	1/1	Running	0	2m2s	minikube-m02
timeserver2-6cbf875b6b-6wm7w	1/1	Running	0	2m2s	minikube-m03
timeserver2-6cbf875b6b-dtnhm	1/1	Running	0	2m2s	minikube-m03
timeserver2-6cbf875b6b-fd6vh	1/1	Running	0	2m2s	minikube-m03
timeserver2-6cbf875b6b-q6fk8	1/1	Running	0	2m2s	minikube-m03
timeserver2-6cbf875b6b-zvk72	1/1	Running	0	2m2s	minikube-m03

Once you're done with the minikube cluster, you can delete all traces of it like so:

```
$ minikube delete
```

8.2 Placing Pods

In the previous section, we covered the balance between replicas and internal concurrency. While it's good having a bunch of Pod replicas for when one fails its health checks, or has a memory leak and needs to be restarted, there are some other considerations, including *where* those Pods are placed.

If you have 10 replicas of a Pod, but they're all on a single node, then you would be impacted by the failure of that node. And expanding on this, using typical cloud topologies, if all your *nodes* are in a single availability zone, then you're at risk of a zone outage. How much time and money you should spend guarding against these conditions is a choice you need to make based on your own production guarantees and budget, since the sky is the limit.

I will focus this section on some sensible and affordable strategies for spreading your Pods on the nodes you already have, as this is something you can do for free, and it gets you some additional availability.

8.2.1 Building Highly Available Deployments

So far we've talked about how resource requests are used to allocate pods to nodes. However, there are other dimensions to consider. To make your application highly available, it is desirable that the replicas don't all end up on the same node. Say you have a small Pod, 100mCPU, 100MiB, and 3 replicas. These 3 replicas could easily all fit on the same node. But then, if that node were to fail, the deployment would be offline.

Better would be to have the scheduler spread these pods out across your cluster! Fortunately, Kubernetes has a built-in way to achieve this called a "topology spread constraint". A topology spread constraint aims to spread your nodes across a failure domain, such as the node, or even a whole zone, and multiple can be specified, so you can spread across both nodes and zones, or any other failure domains defined by your provider.

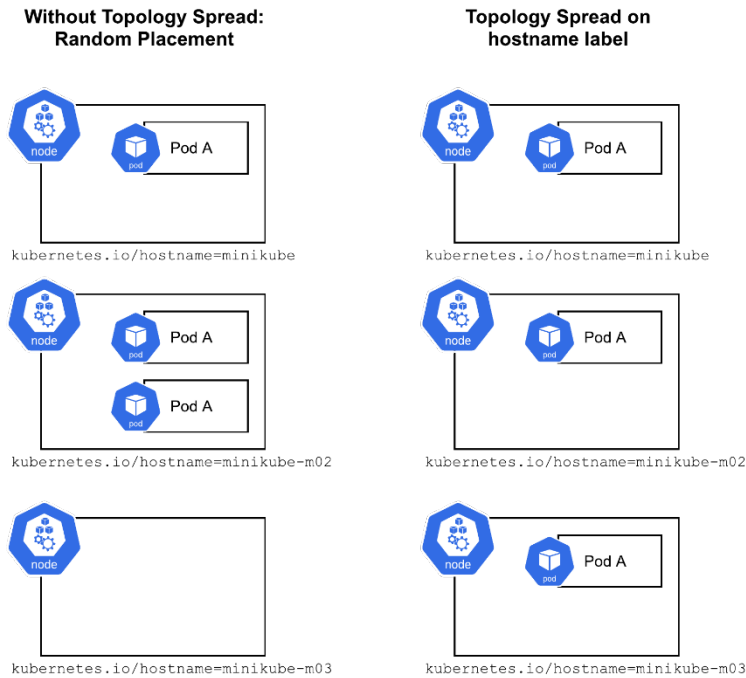


Figure 8.3: deployment pod placement with and without topology constraints

NOTE Many Kubernetes providers have some default topology spread for deployments—including GKE. If you trust the default settings to do the right thing in most cases, feel free to skip past this section. I've included this information regardless, as I find it helps to know why things work the way they do, so I think it's important to understand why Pods get spread over nodes. It is also possible to use the techniques in this chapter to modify the default policies, such as to impose something stricter say for a mission-critical deployment, and to apply topology spreads to objects that don't get them by default like Jobs (covered in a later chapter).

To override the spread topology for a particular deployment, you can add the `topologySpreadConstraints` key, as I've done in the following example.

Listing 8.9 TopologySpread/deploy_topology.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 3
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      topologySpreadConstraints: #A
      - maxSkew: 1 #B
        topologyKey: kubernetes.io/hostname #C
        whenUnsatisfiable: ScheduleAnyway #D
        labelSelector: #E
          matchLabels: #E
            pod: timeserver-pod #E
      containers:
      - name: timeserver-container
        image: docker.io/wdenniss/timeserver:1
        resources:
          requests:
            cpu: 200m
            memory: 250Mi
          limits:
            cpu: 300m
            memory: 400Mi

```

#A the topology constraint added

#B the maximum number of replica imbalance

#C the node label to use for the topology

#D The behavior to use when it's not possible to satisfy the topology requirement

#E another label selector set to this template's metadata label

In this example we're targeting the `kubernetes.io/hostname` topology using the `topologyKey` setting, which really means that Kubernetes will consider all nodes labelled with the same value for the `kubernetes.io/hostname` key to be equal. Since no two nodes should be labelled with the same hostname, this yields a node-level spreading target.

For this configuration to work, and I cannot stress this enough, you must ensure that the nodes in your cluster actually have the label specified in `topologyKey` (`kubernetes.io/hostname` in my example). There are some well-known labels¹, like the one I'm using here, but there is no guarantee that your Kubernetes platform will use it. So, verify by running `kubectl describe node` and look at the Labels that your nodes have.

Going over the rest of the settings, in the example I've used a `maxSkew` of 1, the smallest possible skew, which means there can be at most 1 level of unbalance (which means any node can have at most 1 more pod than the other nodes).

¹ <https://kubernetes.io/docs/reference/kubernetes-api/labels-annotations-taints/>

`whenUnsatisfiable` governs what happens when the constraint can't be satisfied (say that a node is completely full with other pods). The choices are `ScheduleAnyway` and `DoNotSchedule`, whose behavior is self explanatory. `DoNotSchedule` is helpful when testing as it makes it easier to see when the rule is working, but for production `ScheduleAnyway` is going to be safer. While `ScheduleAnyway` makes the rule a "soft" rule, Kubernetes will still do its best to meet your requirements, which I think is better than leaving the replica unscheduled altogether, especially when our goal is higher availability of our replicas!

The last field is a `labelSelector` with our friend `matchLabels` (covered in section 3.X). It's frustrating that Kubernetes doesn't have a simple self-reference here (i.e. why do you even need this at all since it's already embedded in the Pod's specification?), but basically this `matchLabels`, the earlier `matchLabels` and the metadata label (`pod: timeserver-pod`) that they both refer to must match, and be unique for this deployment.

With that, let's go ahead and deploy this example, and verify that the topology is what we expected! To demo this, we'll need a cluster with a few nodes, and one without any default spreading behavior. GKE comes with default node and zonal spreading, so this setting isn't needed on that platform (but in any case it's good to understand that this is what's happening behind the scenes, or if you need to fine-tune the behavior). To try this out and see the differences between various topologies, I suggest minikube configured with 3 nodes.

```
minikube start --nodes 3
kubectl create -f deploy_topology.yaml
kubectl get pods -o wide
```

Looking at the NODE column, you should see 3 separate nodes (assuming you have 3 nodes in the cluster).

```
> kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
pluscode-754665f45f-6n6bc	1/1	Running	0	39s	10.244.2.46	minikube-m03
pluscode-754665f45f-dkvvp	1/1	Running	0	39s	10.244.0.27	minikube
pluscode-754665f45f-dmqpq	1/1	Running	0	39s	10.244.1.47	minikube-m02

Figure 8.4: Deployment with `topologySpreadConstraints`, with the unique nodes highlighted

NOTE Topology spread is a scheduling-time constraint, in other words it's considered only when Pods are placed onto nodes. Once all replicas are running, if the topology changes (e.g. a node is added), the running Pods will not be moved. If needed, you can redeploy the Pods by making a change to the deployment, which will apply the scheduling rules again, so any topology changes would then be considered.

To compare, deploy the same deployment but without the `topologySpreadConstraints`, you'll notice that pods can be grouped up. Of course, due to chance they may be spread out anyway, in which case try deleting the deployment and redeploying. If you observe that the Pods are following a topology without one being explicitly set, then there's likely a default on the cluster, as in the case with GKE.

SPREADING ACROSS ZONES

`topologySpreadConstraints` can be used with any node label, so another common strategy is to spread across zones (if you have a multi zone cluster). For this you can repeat the earlier example, but using a zone-based key with `topology.kubernetes.io/zone` being the standardized “well known” key (but again, do check that your nodes actually have this label otherwise it will have no effect). Multiple topologies can be specified in the array provided to `topologySpreadConstraints`, so you can have both a node and zonal spread.

8.2.2 Collocating Interdependent Pods

In some cases, you may have tightly coupled Pods where it’s desirable to have them be present on the same physical machine. Services that are particularly “chatty” (i.e. they make a lot of inter-service procedure calls) are often candidates for this type of architecture. Say that you have a front end, and a backend and they communicate a lot between each other. You may wish to pair them on nodes together to reduce network latency and traffic.

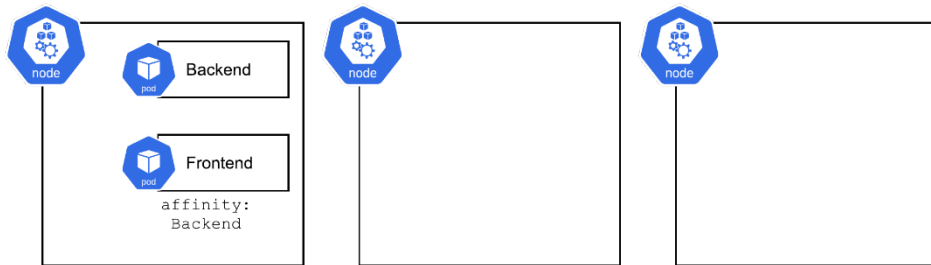


Figure 8.5: 3 frontend pods scheduled on the same node as the backend pod using pod affinity

This deployment construct can be achieved through Pod affinity rules. Essentially one of the deployments, using the above example perhaps the frontend, gets a rule that tells the scheduler “only place this Pod on nodes which have a backend Pod”.

Let’s say we have the following “backend” deployment:

Listing 8.10 CoLocation/backend.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: mariadb
spec:
  replicas: 1
  selector:
    matchLabels:
      pod: mariadb-pod
  template:
    metadata:
      labels:
        pod: mariadb-pod
    spec:
      containers:
        - name: mariadb-container
          image: mariadb
          env:
            - name: MARIADB_RANDOM_ROOT_PASSWORD
              value: "1"

```

There is nothing special about this deployment at all, it follows the same pattern we've been using. This Pod will be placed on any available space in the cluster.

Now, for the "frontend" deployment where we want to require it to be placed on nodes with instances of a Pod from the backend deployment, we can use the following configuration.

Listing 8.11 CoLocation/frontend.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 3
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:1
      affinity: #A
      podAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: pod
                  operator: In
                  values:
                    - mariadb-pod
            topologyKey: "kubernetes.io/hostname"

```

#A Pod affinity rule

This specification requires that the scheduler locate this Pod on a node within the specified topology that has an existing pod with the label "app: mariadb". As the topology in the example is a node topology (using the well-known label for hostname), this means that the app will only be scheduled onto a node that has the target Pod. If a zonal topology was used (using the well-known label for zone, as discussed in 8.2.1), then the Pod would be placed on any node in the zone that has an existing pod with the target label.

To make this co-location a "soft" (or best-effort) requirement so that your Pods will still be scheduled, even if the requirement can't be satisfied, the `preferredDuringSchedulingIgnoredDuringExecution` can be used instead of `requiredDuringSchedulingIgnoredDuringExecution`.

As you can see, Kubernetes is really flexible, allowing you to make scheduling rules binding or just guidelines, and specify your preferred topology in a myriad of ways (with node and zonal being two common choices). It's easy to get bamboozled with the choice. For most deployments, I would advise **not** using Pod affinities at the outset, but rather keep these techniques in your back pocket, and applying them when you have specific issues you wish to resolve (e.g. like wanting to co-locate Pods on a single node to reduce inter-service latency).

8.2.3 Avoiding Certain Pods

In the earlier section "Building Highly Available Deployments", I covered how you can use topology spread to spread out Pods from the same deployment to avoid single points of failure. What about Pods that are related (so you want them spread out) but are deployed separately? As an example, imagine you have a deployment for a backend service, and a separate deployment for a caching service, and would prefer they be spread out.

For this, you can use pod anti-affinity. This is simply throwing the pod affinity rule from the previous section into reverse so that the Pods will be scheduled on other nodes (or the topology of your choice).

Listing 8.12 PodAntiAffinity/frontend.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 3
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:1
      affinity:
        podAntiAffinity: #A
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: pod
                    operator: In
                    values:
                      - mariadb-pod
              topologyKey: "kubernetes.io/hostname"

```

#A the pod affinity rule from the previous example is reversed, so now this Pod will explicitly avoid nodes that have a pod with the “app: mariadb” label.

All these things can act together too, so you can have a topology spread that seeks to keep pods apart. Just be careful that your rules *can* actually be satisfied; otherwise you’ll end up with unscheduled Pods. As with affinities in the previous section, you can also use “soft” rules by specifying `preferredDuringSchedulingIgnoredDuringExecution` instead of `requiredDuringSchedulingIgnoredDuringExecution`. When doing this, you might want to test it first with the required rule to ensure you have your `labelSelector` setup correctly, before relaxing the rule. Section 5.4.4 has some more debugging tips for setting these rules.

8.3 Debugging Placement Issues

Pod placement is a pretty complex topic, so don’t be surprised if you encounter bugs. The most common issue occurs when you require the existence of a label that none of your nodes have (or in the case of a nodeless platform, a label for a feature that isn’t supported by the platform)! Such pods will never be scheduled. Here are some of the common issues you might encounter, and how to solve them:

PLACEMENT RULES DON’T APPEAR TO WORK

If your placement rules don’t appear to work in testing, the first thing I’d suggest is ensure you are not using any “soft” placement rules. These rules mean that the scheduler basically

ignores your rule when it can't be satisfied, which isn't so great for testing. It's better to verify that all your rules are working before relaxing them by changing them to soft rules.

Use a small cluster with only a couple of nodes, no soft rules, and you should be able to observe the effect of the placement features. Verify that the rules are enforced by intentionally attempting to schedule Pods that would violate the rules. Their status should be "Pending" because the constraints can't be satisfied.

PODS ARE PENDING

Pods that display as the "Pending" state mean that the scheduler can't find a suitable place for them. In Chapter 3, we discussed this error in the context of the cluster not having enough resources to place the Pod. Once you configure your placement rules, it's possible the Pod can't be scheduled because the rules can't be satisfied. To find out what the reason is (i.e. which rule couldn't be satisfied), *describe* the pod. Note, that you need to do this at a Pod level – the deployment itself won't show any error messages (although it will indicate that the desired number of replicas isn't met).

```
kubectl get pods
kubectl describe pod POD_NAME
```

Example abridged output:

Events:				
Type	Reason	Age	From	Message
----	-----	----	----	-----
Warning	FailedScheduling	4s	default-scheduler	0/1 nodes are available: 1 node(s) had taints that the pod didn't tolerate. #A

#A There are no nodes available without taints. Either add a toleration to this pod, or add more untainted nodes

Warning	FailedScheduling	17s (x3 over 90s)	default-scheduler	0/1 nodes are available: 1 node(s) didn't match pod affinity/anti-affinity, 1 node(s) didn't match pod anti-affinity rules. #A
---------	------------------	-------------------	-------------------	--

#A The Pod's affinity or anti-affinity rules could not be satisfied. Review and revise the rules.

8.4 Summary

The Kubernetes scheduler lies at the core of the system and does the heavy lifting of finding the right home for your deployment's Pods on your infrastructure.

- Topology spread policy can be used to spread your Pod replicas out, and many platforms will have a default spread policy
- Pods that benefit from being in proximity to each other can be co-located with pod affinity
- To spread related but different Pods, anti-affinity rules can be applied
- Taints can be used to prevent workloads landing on certain types of nodes by default
- Tolerations and node selectors can be used to target specific workloads at tainted nodes

9

Stateful Applications

This chapter covers

- How to attach persistent disk storage to Pods
- How the Kubernetes concepts of volumes, PersistentVolumes, PersistentVolumeClaims and StorageClasses interact to give you stateful capabilities
- Creating a simple single-Pod deployment with persistent state
- Deploying a complex multiple-Pod stateful application with multiple roles
- Migrating and recovering data by re-linking Kubernetes objects to the underlying disk resources

Stateful applications, ones that have attached storage, finally have a home with Kubernetes. While stateless applications are lauded for their high scalability, and not needing to manage attached storage helps with that, they are not the only way to deploy software. Whether you're deploying a sophisticated database or are migrating an old stateful application from a VM, Kubernetes has you covered.

Using Persistent Volumes, you can attach stateful storage to any Kubernetes Pod. When it comes to multi-replica deployments, just as Kubernetes offers Deployment as a high-level construct for managing a stateless application, StatefulSet exists to provide high-level management of stateful applications.

9.1 Volumes, Persistent Volumes, Claims and Storage Classes

To get started with storing state in Kubernetes, there are a few concepts around volume (disk) management to cover before moving on to the higher level StatefulSet construct. Just like nodes are the Kubernetes representation of a virtual machine, Kubernetes has its own representation of disks as well.

In this section we will deploy a MariaDB database, and attach an external disk mounted at `/var/lib/mysql`, which is where MariaDB stores its data. You can replace this MariaDB container with your own single-pod deployment that requires a persistent disk.

9.1.1 Volumes

Kubernetes offers functionality to Pods that allows them to mount a volume. What's a *volume*? The docs describe it like so:

At its core, a volume is just a directory, possibly with some data in it, which is accessible to the containers in a pod. How that directory comes to be, the medium that backs it, and the contents of it are determined by the particular volume type used.

Volumes, the Kubernetes Authors

Kubernetes ships with some built in volume types, and others can be added by your platform administrator via storage drivers. Some types you may encounter frequently are `emptyDir`, an ephemeral volume tied to the lifecycle of the node, `ConfigMap`, which allows you specify files in Kubernetes manifests and present them to your application as file on disk, and cloud provider disks for persistent storage.

EMPTYDIR VOLUMES

The built-in volume type `emptyDir` is an ephemeral volume that is allocated on space from the node's boot disk. If the Pod is deleted, or moved to another node, or the node itself becomes unhealthy, all data is lost. So what's the benefit?

Pods can have multiple containers, and `emptyDir` mounts can be shared between them. So when you need to share data between containers, you would define an `emptyDir` volume, and mount it in each container in the Pod. The data is also persisted between container *restarts*, just not all the other events I mentioned earlier. This is useful for ephemeral data such as that of an on-disk cache where it is beneficial if the data was preserved between Pod restarts, but where long term storage isn't necessary.

Listing 9.1 Volume/emptydir_pod.yaml

```

apiVersion: v1
kind: Pod
metadata:
  name: emptydir-pod
  labels:
    pod: timeserver-pod
spec:
  containers:
    - name: timeserver-container
      image: docker.io/wdenniss/timeserver:1
      volumeMounts:
        - name: cache-volume #A
          mountPath: /app/cache/ #A
  volumes:
    - name: cache-volume #B
      emptyDir: {} #B

```

#A The mount path

#B The volume definition

Why on earth is this called `emptyDir`? Because the data is stored in an initially empty directory on the node. It's a misnomer in my opinion, but what can you do.

As a practical example, see section 9.2.2 where `emptyDir` is used to share data between two containers in the same Pod, where one of them is an “init container” that runs first and can perform setup steps for the main container.

CONFIGMAP VOLUME

ConfigMap is a useful Kubernetes object. You can define key/value pairs and reference them in other object declarations. You can also use them to store entire files! Typically these files would be configuration files like `my.cnf` for MariaDB, `httpd.conf` for Apache, `redis.conf` for Redis and so on. You can mount the ConfigMap as a volume, which allows the files it defines to be read from the container. ConfigMap volumes are read-only.

This technique is particularly useful for defining a configuration file for use by a public container image, as it allows you to provide configuration without needing to extend the image itself. For example, to run Redis you can reference the official Redis image, and just mount your config file using ConfigMap wherever Redis expects it, no need to build your own image just to provide this one file.

See section 9.2.1 and 9.2.2 for examples of configuring Redis with a custom configuration file specified via a ConfigMap volume.

CLOUD PROVIDER VOLUMES

More applicable for building *stateful* applications (where you don't typically want to use ephemeral or read-only volumes), is mounting disks from your cloud provider as volumes. Wherever you are running Kubernetes, your provider should have supplied drivers into the cluster that allow you to mount persistent storage, whether that's NFS or block-based (often, both).

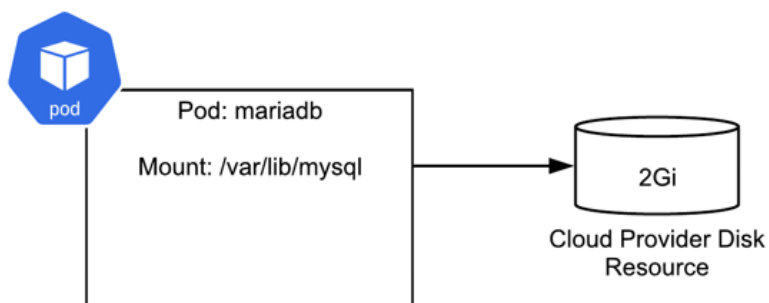


Figure 9.1: Pod with a mounted cloud provider volume

By way of example, here is the specification for a MariaDB Pod running in GKE mounting a GCE persistent disk for its storage.

Listing 9.2 Volume/mariadb_pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: mariadb-demo
  labels:
    app: mariadb
spec:
  affinity:
    nodeAffinity: #A
    requiredDuringSchedulingIgnoredDuringExecution: #A
    nodeSelectorTerms: #A
    - matchExpressions: #A
      - key: topology.kubernetes.io/zone #A
        operator: In #A
        values: #A
      - us-west1-a #A
  containers:
    - name: mariadb-container
      image: mariadb:latest
      volumeMounts: #B
      - mountPath: /var/lib/mysql #B
        name: mariadb-volume #B
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: "your database password"
  volumes:
    - name: mariadb-volume
      gcePersistentDisk:
        pdName: mariadb-disk #C
        fsType: ext4
```

#A Node affinity targeting the zone where our disk exists, so that the Pod will be created in the same zone.

#B The directory where we want to mount the disk in our container.
 #C The name of the persistent disk GCP resource.

Unlike the more automated and cloud agnostic approaches we'll cover next, this method is tied to your cloud provider, and requires manual creation of the disk. You need to ensure a that a disk with the name specified exists (which you need to create out-of-band, i.e. using your cloud provider's tools), and that both the disk and the pod are in the same zone. In this example, I use `nodeAffinity` to target the zone of the disk, which is important for any Kubernetes cluster that exists in multiple zones (otherwise your Pod could be scheduled on a different zone to that of the disk).

Creating the disk used by this example out-of-band can be achieved using the following command.

```
gcloud compute disks create --size=10GB --zone=us-west1-a mariadb-disk
```

Since we're creating this disk manually, pay close attention to the location where the resource is being created. The zone in the previous command, and the zone set via the `nodeAffinity` configuration needs to match. If you see your Pod stuck in "Container Creating", inspect your event log for the answer. Here's a case where I hadn't created the disk in the right project:

```
$ kubectl get events -w
0s          Warning   FailedAttachVolume
             pod/mariadb-demo          AttachVolume.Attach failed
             for volume "mariadb-volume" : GCE persistent disk not found: diskName="mariadb-disk"
             zone="us-west1-a"
```

The downside to mounting volumes directly is that the disks need to be created outside of Kubernetes, which means:

- a) The user creating the Pod must have permissions to create the disk (which is not always the case)
- b) Steps exist outside of Kubernetes configuration that need to be remembered and ran manually
- c) The volume descriptors are platform-dependent, so this Kubernetes YAML is not portable and won't work on another provider.

Naturally, Kubernetes has a solution to this lack of portability. By using the volume abstraction provided by Kubernetes, you can simply request the disk resources you need, and have them provisioned for you with no need to perform any out-of-band steps. Read on.

9.1.2 Persistent Volumes and Claims

To provide a way to manage persistent volumes in a more platform-agnostic way, Kubernetes offers higher level primitives around volumes, known as persistent volumes and persistent volume claims.

Instead of linking to the volume directly, the pod references a `PersistentVolumeClaim` object which defines the disk resources that the Pod requires in platform-agnostic terms (for example: "1 gigabyte of storage"). The disk resources themselves are represented in Kubernetes using a `PersistentVolume` object, much like how Nodes in Kubernetes represent

the virtual machine resource. When the `PersistentVolumeClaim` is created, Kubernetes will seek to provide the resources requested in the claim by creating or matching it with a `PersistentVolume` and binding the two objects together. Once bound, the persistent volume and claim which now reference each other, typically remain linked until the underlying disk is to be deleted.

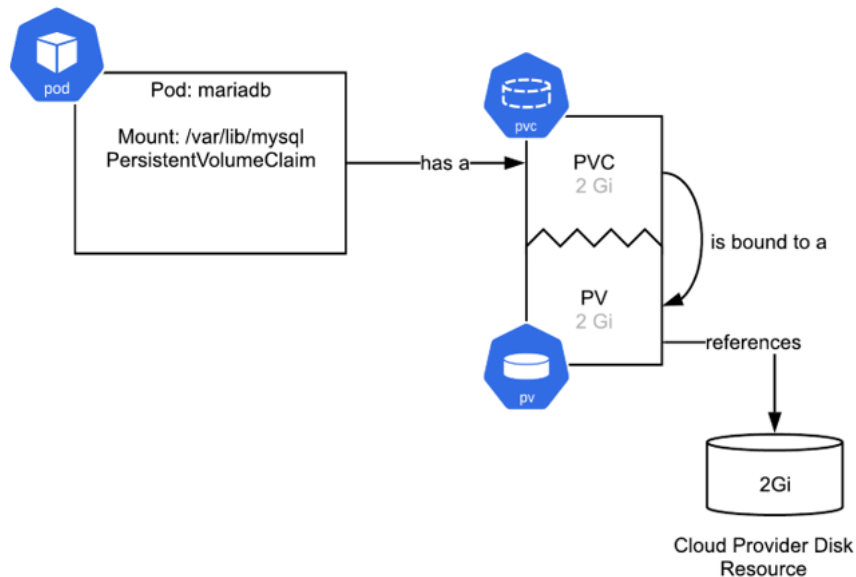


Figure 9.2: Pod that references a `PersistentVolumeClaim` that gets bound to a `PersistentVolume` which references a disk

This behavior of having the claim which requests resources, and a representation of those resources is similar to how a pod requests compute resources like CPU and memory, and the cluster finds a Node that has these resources to schedule the pod on. And like with a Pod's CPU and memory capacity requests, it now means that the storage requests are defined in a platform independent manner. Unlike using the cloud provider disk directly, when using the `PersistentVolumeClaim`, your Pods can be deployed anywhere, provided the platform supports persistent storage.

Let's rewrite our Pod from the previous section to use an `PersistentVolumeClaim` to request a new persistent volume for our pod.

Listing 9.3 PersistentVolume/pvc-mariadb.yaml

```

apiVersion: v1
kind: Pod
metadata:
  name: mariadb-demo
  labels:
    app: mariadb
spec:
  containers:
    - name: mariadb-container
      image: mariadb:latest
      volumeMounts:
        - mountPath: /var/lib/mysql
          name: mariadb-volume
      resources: #A
        requests: #A
          cpu: 1 #A
          memory: 4Gi #A
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: "your database password"
  volumes:
    - name: mariadb-volume
      persistentVolumeClaim:
        claimName: mariadb-pv-claim
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mariadb-pv-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources: #B
    requests: #B
      storage: 2Gi #B

```

#A the compute resources that the pod is requesting (see Chapter 5)

#B the disk resources that the pod is requesting

In the `PersistentVolumeClaim` definition, we're making a request for 2Gi of storage, and specifying the desired `accessMode`. The `ReadWriteOnce` access mode is for a volume that behaves like a traditional hard drive where your storage is mounted to a single pod for read write access and is the most common. The other choices for `accessMode` are `ReadOnlyMany` which can be used to mount a volume of existing data that's shared across many pods, and `ReadWriteMany` for mounting file storage (like NFS) where multiple pods can read/write at the same time (a fairly special mode, only supported by a few storage drivers). In this chapter the goal is stateful applications backed by traditional block-based volumes, so `ReadWriteOnce` is used throughout.

If your provider supports dynamic provisioning, a `PersistentVolume` backed by a disk resource will be created to fulfil the storage requested by the `PersistentVolumeClaim`, after which the `PersistentVolumeClaim` and `PersistentVolume` will be bound together. The dynamic provisioning behavior of the `PersistentVolume` is defined through the `StorageClass`

(which we cover in the next section). GKE and almost every provider supports dynamic provisioning, and will have a default storage class, so the above Pod can be deployed pretty much anywhere.

In the rare event that your provider *doesn't* have dynamic provisioning, you (or the cluster operator/admin) will need to manually create a `PersistentVolume` yourself with enough resources to satisfy the `PersistentVolumeClaim` request. Kubernetes still does the matchmaking of linking the claim to the volume.

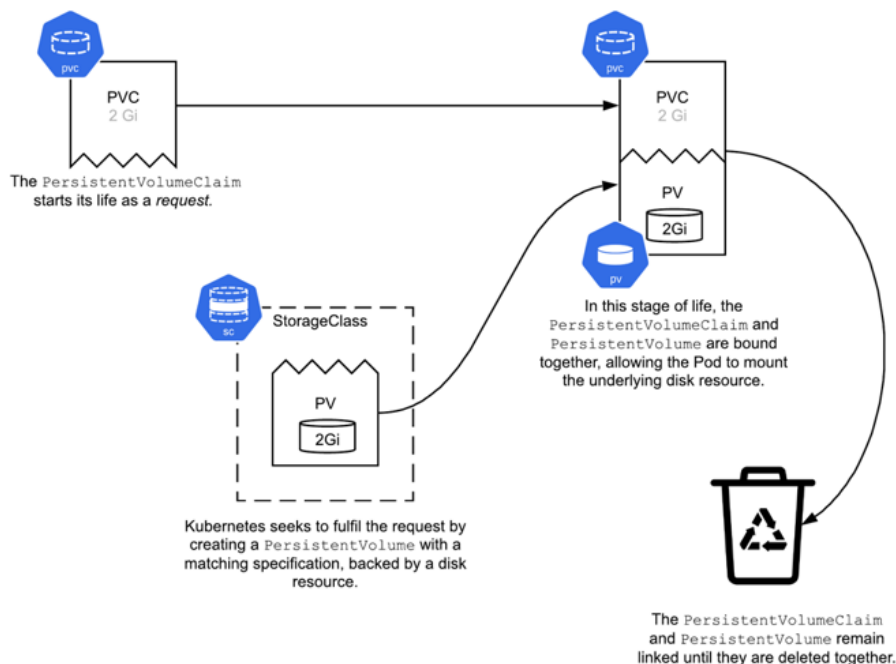


Figure 9.3: The lifecycle of a `PersistentVolumeClaim` and `PersistentVolume` in a dynamically provisioned system.

The `PersistentVolumeClaim` as defined in the above example can be thought of as a *request* for resources. The claim part really happens later, when it is matched with, and bound to, a `PersistentVolume` resource (and both resources are linked to each other). Essentially the `PersistentVolumeClaim` has a lifecycle that starts as a request, and becomes a claim when bound.

We could leave it there, but since your precious data will be stored on these disks, let's dig in to see just how this binding works. If we query the YAML of the above `PersistentVolumeClaim` *after* it's bound, you'll see that it's been updated with a `volumeName`.

This `volumeName` is the name of the `PersistentVolume` that it was linked to, and now claims. Here's what it looks like (with some superfluous information omitted for readability):

```
$ kubectl get -o yaml pvc/mariadb-pv-claim
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mariadb-pv-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
  storageClassName: standard-rwo
  volumeMode: Filesystem
  volumeName: pvc-ecb0c9ed-9aee-44b2-a1e5-ff70d9d3823a #A
status:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 2Gi
  phase: Bound
```

#A the `PersistentVolume` which this object is now bound to

We can query the `PersistentVolume` named in this configuration with `kubectl get -o yaml pv NAME`, and we'll see that it links right back to the PVC. Here is what mine looked like.

```
$ kubectl get -o yaml pv pvc-ecb0c9ed-9aee-44b2-a1e5-ff70d9d3823a
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pvc-ecb0c9ed-9aee-44b2-a1e5-ff70d9d3823a
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 2Gi
  claimRef:
    apiVersion: v1
    kind: PersistentVolumeClaim
    name: example-pv-claim #A
    namespace: default #A
  csi:
    driver: pd.csi.storage.gke.io
    fsType: ext4
    volumeAttributes:
      storage.kubernetes.io/csiProvisionerIdentity: 1615534731524-8081-
        pd.csi.storage.gke.io
      volumeHandle: projects/gke-autopilot-test/zones/us-west1-b/disks/pvc-ecb0c9ed-9aee-
        44b2-a1e5-ff70d9d3823a #B
  persistentVolumeReclaimPolicy: Delete
  storageClassName: standard-rwo
  volumeMode: Filesystem
status:
  phase: Bound #C
```

- #A The PersistentVolumeClaim that this PersistentVolume is bound to
- #B The pointer to the underlying disk resources
- #C The status is now “bound”

It helps to visualize this side by side, so here it is:

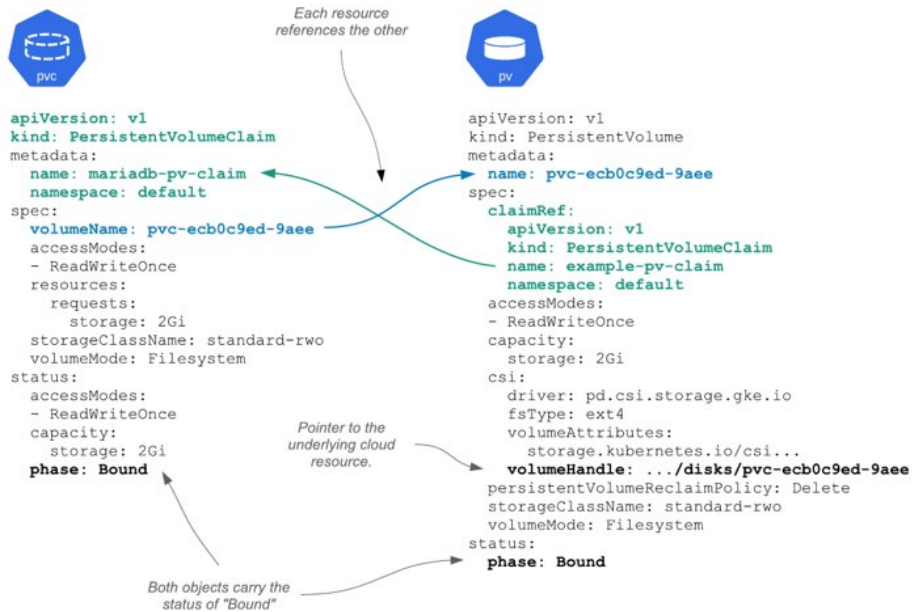


Figure 9.4: The PersistentVolumeClaim and PersistentVolume after the latter was provisioned, and they were bound together

The `PersistentVolumeClaim` has really undergone a metamorphosis here, going from a request for resources, to being a claim for a specific disk resource that exists (and will contain your data). This is not really like any other Kubernetes object I can think of. While it's common to have Kubernetes add fields and perform actions on the object, few change like these do, starting as a generic request for and representation of storage, and end up as a bound stateful object.

There is one exception to this typical lifecycle of a `PersistentVolumeClaim`, which is when you have *existing* data that you wish to mount into a Pod. In that case, you create the `PersistentVolumeClaim` and the `PersistentVolume` objects already pointing at each other, so they are bound immediately at birth. This scenario is discussed in section 9.3 on migrating and recovering disks, including a fully worked data recovery scenario.

9.1.3 Storage Classes

So far, we’ve relied on the default dynamic provisioning behavior of the platform provider. But what about if we want to change what type of disks we get during the binding process, or what happens to the data if the `PersistentVolumeClaim` is deleted? That’s where storage classes come in.

Storage classes are way to describe the different types of *dynamic* storage that are available to be requested from `PersistentVolumeClaims`, and how the volumes that are requested in this way should be configured.

Your Kubernetes cluster probably has a few defined already, let’s view them (some columns have been stripped for readability):

```
$ kubectl get storageclass
```

NAME	PROVISIONER	RECLAIMPOLICY
premium-rwo	pd.csi.storage.gke.io	Delete
standard	kubernetes.io/gce-pd	Delete
standard-rwo (default)	pd.csi.storage.gke.io	Delete

When we created the pod in the previous section with a `PersistentVolumeClaim`, the default storage class (`standard-rwo` in this case) was used. If you go back and look at the bound `PersistentVolumeClaim` object, you’ll see this storage class in the configuration.

This is a pretty good start, and you may not need to do much here. There is one thing that I do suggest you consider customizing. You may notice if you read the “reclaim policy” column above and that it says `Delete`. What this means is that if the persistent volume claim is deleted the bound persistent volume and the disk resource that backs it will also be deleted. If your stateful workloads are mostly just caching services this might be fine, but if instead your workloads store unique and precious data this default behavior is probably not ideal.

Kubernetes also offers a `Retain` reclaim policy which means that the underlying disk resource will not be deleted on the deletion of the persistent volume claim allowing you to bind it to a new persistent volume and persistent volume claim potentially even one that you create in a completely separate cluster.

To build our own storage class, it’s simplest to start with the system one. We can export it as follows. If your list is different to mine, replace `standard-rwo` with your default or preferred storage class:

```
kubectl get -o yaml storageclass standard-rwo > storageclass.yaml
```

Now we can customize and set the all-important `Retain` reclaim policy. Since we want to create a new policy it’s also important to give it a new name, and strip the `uid` and `selfLink` metadata fields. After performing those steps, this is what I get:

Listing 9.4 StorageClass/storageclass.yaml

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
  name: example-default-rwo
parameters:
  type: pd-balanced
provisioner: pd.csi.storage.gke.io
reclaimPolicy: Retain #A
volumeBindingMode: WaitForFirstConsumer
allowVolumeExpansion: true

```

#A The reclaim policy is set to Retain

Before we can apply this config, we need to mark the current default as non-default. You can edit with `kubectl edit storageclass standard-rwo`, or patch it with the following one-liner. Again, replace `standard-rwo` with whatever the name of your default class is.

```

kubectl patch storageclass standard-rwo -p '{"metadata":
{"annotations":{"storageclass.kubernetes.io/is-default-class":"false"}}}'

```

Then create our new storage class, with `kubectl create -f storageclass.yaml`.

If you have different types of persistent data in your cluster, say precious data and also caching data that can be re-created, you could have two storage classes configured to represent each “class” of storage (precious data, and cache data). You can then reference them manually using the `storageClassName` field in your `PersistentVolumeClaim` objects. The examples in this chapter however will all use the default `StorageClass`, whatever you (or your cloud provider) set that to be.

9.1.4 Single-pod Stateful Workload Deployments

Combining these concepts, we can provision a 1-replica stateful workload by simply enclosing our Pod into a Deployment. The benefit of using a Deployment even for a single-replica pod is that if the pod is terminated it will be recreated.

Listing 9.5 Deployment_MariaDB/mariadb-deploy.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: mariadb-demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mariadb
  template: #A
    metadata:
      labels:
        app: mariadb
    spec:
      containers:
        - name: mariadb-container
          image: mariadb:latest
          volumeMounts:
            - mountPath: /var/lib/mysql
              name: mariadb-volume
          resources:
            requests:
              cpu: 1
              memory: 4Gi
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: "your database password"
      volumes:
        - name: mariadb-volume
          persistentVolumeClaim:
            claimName: mariadb-pvc
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mariadb-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi

```

#A Our pod template spec is identical to the one shown in Section 9.1.2

So there we have it this is a single pod deployment of a MariaDB database with an attached disk that won't be deleted even if this entire Kubernetes cluster is deleted, thanks to the `Retain` policy in the storage class we created in the prior section.

If you want to give this database a spin, create a service for it (see the `service.yaml` file in the same directory). Once the service is created, you can connect to the database from a local client, or you can try out the containerized phpMyAdmin (see Bonus/phpMyAdmin in the code repository that accompanies the book for a sample configuration of that).

Running Databases in Kubernetes

Before you take the plunge to manage your own like MariaDB in Kubernetes like this, you probably want to look for managed solution with your cloud provider. I know it's tempting just to deploy in Kubernetes because it's fairly easy to create such a database, as I demonstrated, but the operational cost comes later when you have to secure, update and manage it. Generally, I recommend reserving the stateful workload functionality of Kubernetes for customized or bespoke services, or services that your cloud provider doesn't offer as a managed offering.

As demonstrated in this section, we can make our workloads stateful by attaching volumes, using `PersistentVolumeClaims`. Using Pod and Deployment objects for this however, constrains us to single replica stateful workloads. This might be enough for some, but what if you have a sophisticated stateful workload like Elasticsearch or Redis with multiple replicas? You could try and stitch together a bunch of deployments, but fortunately Kubernetes has a high-level construct that is designed to represent exactly this type of deployment called the stateful set.

9.2 StatefulSet

We've seen how persistent storage can be added to a pod in Kubernetes—a useful feature because Pods are the basic building block in Kubernetes, and they are used in many different deployment constructs like Deployments (Chapter 3) and Jobs (Chapter 10). Now you can add persistent storage to any of them and build stateful pods wherever you need, provided that the volume specification is the same for all instances.

The limitation of the deployment constructs like Deployment that all pods share the same specification is an issue for `ReadWriteOnce` volumes. These traditional block storage volumes can only be mounted by a single instance. This is OK when there is only one replica in your Deployment, but it means that if you create a second replica, that Pod will fail to be created as the volume is already mounted.

Fortunately, Kubernetes has a high-level construct that makes our lives easier when we need multiple Pods where they each get their own disk (a highly common pattern). Just like Deployment is a high-level construct for managing (normally stateless) continuously running services, StatefulSet is a high-level construct for managing stateful services.

StatefulSet has a few helpful properties for building such services. You can define a volume template instead of a referencing a single volume in the pod spec and Kubernetes will create a new persistent volume claim for each pod which solves the issue when using Deployment, where each instance shared the same persistent volume claim. StatefulSet assigns each pod a stable identifier which is linked to a particular persistent volume claim and provides ordering guarantees for deployment, scaling and updates. Now you can create multiple Pods and coordinate them by using this stable identifier to assign each a different role.

9.2.1 Deploying StatefulSet

Putting this into practice, let's look at two popular stateful workloads: MariaDB and Redis, and how to deploy them as a StatefulSet. At first we'll stay with a single-pod StatefulSet which is

the simplest to demonstrate as there are not multiple roles to worry about. The next section will add additional replicas with different roles to fully use the power of StatefulSet.

MARIADB

First, let's convert the single pod MariaDB deployment we created in the previous section to one using StatefulSet and take advantage of the persistent volume claim template to avoid the need to create a separate PersistentVolumeClaim object ourselves.

Listing 9.6 StatefulSet_MariaDB/mariadb-statefulset.yaml

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mariadb
spec:
  selector: #A
    matchLabels: #A
      app: mariadb-sts #A
  serviceName: mariadb-service #B
  replicas: 1
  template:
    metadata:
      labels:
        app: mariadb-sts
    spec:
      terminationGracePeriodSeconds: 10 #C
      containers:
        - name: mariadb-container
          image: mariadb:latest
          volumeMounts:
            - name: mariadb-pvc #D
              mountPath: /var/lib/mysql #D
          resources:
            requests:
              cpu: 1
              memory: 4Gi
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: "your database password"
      volumeClaimTemplates: #E
        - metadata: #E
            name: mariadb-pvc #E
          spec: #E
            accessModes: #E
              - ReadWriteOnce #E
            resources: #E
              requests: #E
                storage: 2Gi #E
---
apiVersion: v1 #F
kind: Service #F
metadata: #F
  name: mariadb-service #F
spec:
  ports:
    - port: 3306
  clusterIP: None
```

```
selector:
  app: mariadb-sts
```

#A StatefulSet uses the match labels pattern as Deployments, which discussed in Chapter 3

#B This is a link to the headless service which is defined at the bottom of the file.

#C StatefulSet requires that a graceful termination period be set. This is the number of seconds that the Pod has to exit on its own before being terminated.

#D The volume claim matching this name is defined in the volumeClaimTemplates section

#E Unique to StatefulSet, we can define a template of a PersistentVolumeClaim just like we define the template of the pod replicas. This template is used to create the PersistentVolumeClaims, associating one to each of the pod replicas.

#F The headless services for this StatefulSet (required)

So what changed compared to the Deployment of a MariaDB pod with an attached persistent volume in 9.1.4? Other than the usual metadata differences, there are two key changes.

The first difference is how the `PersistentVolumeClaim` is configured. In 9.1.4 when used with a Pod in a deployment, this was a standalone object. With StatefulSet, this is rolled into the definition itself much like how a Deployment has a Pod template. What this means is that each `PersistentVolume` claim will use this template. You end up with the same result, but don't have to define a separate object (which is useful with this example, and essential when creating one with more than 1 replica). Here they are side by side:

Table 9.1 PersistentVolumeClaim vs volumeClaimTemplates

<pre>apiVersion: v1 kind: PersistentVolumeClaim metadata: name: mariadb-pvc spec: accessModes: - ReadWriteOnce resources: requests: storage: 2Gi</pre>	<pre>... volumeClaimTemplates: - metadata: name: mariadb-pvc spec: accessModes: - ReadWriteOnce resources: requests: storage: 2Gi</pre>
--	---

If you query the persistent volume claims after creating the StatefulSet, you'll see this claim (with some columns removed for readability):

```
$ kubectl get pvc
NAME                                STATUS  VOLUME    CAPACITY  ACCESS MODES
mariadb-pvc-mariadb-0              Bound   pvc-71b1e  2Gi       RWO
```

The only difference is that the one created with the template has the pod name (`mariadb-0` in the case of the first pod) appended to it, so instead of being `mariadb-pvc` (the name of the claim template), it's `mariadb-pvc-mariadb-0` (the claim template name, and pod name combined).

The second difference is a service that is referenced in the StatefulSet with the `serviceName: mariadb-service` line, and defined as so:

```
apiVersion: v1
```

```
kind: Service
metadata:
  name: mariadb-service
spec:
  ports:
    - port: 3306
  clusterIP: None
  selector:
    app: mariadb-sts
```

This Service is a bit different to the ones presented in the book so far, as it's what's known as a headless service (thanks to the `clusterIP: None` in the specification). That means that unlike every other Service we created so far there is no service endpoint with load balancing. The service has no IP, external or internal. It exists so that the pods in the StatefulSet can get their own network identities so that *they* can be addressed. Remember that each pod in the stateful set is unique (unlike with Deployments), and is given an incrementing integer value known as the ordinal, so it's common to address them individually and directly. This is opposed to Pods in a Deployment where each pod is given a random id. If a Pod in a Stateful set is recreated (like when moved from one node to another during an update) it will retain the same identifying ordinal, whereas Pods that are replaced in a Deployment are assigned a new random name.

Pods can be addressed at `STATEFULSET_NAME-POD_ORDINAL.SERVICE_NAME`

In this example, our single pod can be referenced using the DNS address: `mariadb-0.mariadb-service`. From outside the namespace, you can append the namespace e.g.:

```
mariadb-0-mariadb-service.production.svc
```

REDIS

Another example we can use is Redis. Redis is a very popular deployment in Kubernetes, and has many different possible uses, often for caching and other real-time data storage and retrieval needs.

For this example, let's imagine the caching use-case where the data isn't super precious. You still want to persist the data to disk (to avoid rebuilding the cache in the event of a restart), but there's no need back it up. What follows is a perfectly usable single-pod Redis deployment for Kubernetes for such applications.

To configure Redis, let's first define our config file which we can mount as a volume in the container:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: redis-config
data:
  redis.conf: |
    bind 0.0.0.0 #A
    port 6379 #B
    protected-mode no #C
    appendonly yes #D
    dir /redis/data #E
```

#A bind to all interfaces so that other Pods can connect

#B the port to use
 #C disable protected mode so that other Pods in the cluster can connect without a password
 #D enable the append log to persist data to disk
 #E specify the directory

See the code annotation for what each of the configuration options do. The important thing is that we're persisting the Redis state to the `/redis/data` directory, so it can be reloaded if the Pod is re-created.

This example does not configure authentication for Redis, which means that every Pod in the cluster will have read/write access. If you take this example and use it for in a production cluster, please consider how you wish to configure the cluster.

Now let's go ahead and create a StatefulSet that will reference this config, and mount the `/redis/data` directory as a PersistentVolume:

Listing 9.7 StatefulSet_Redis_SinglePod/redis-statefulset.yaml

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  selector:
    matchLabels:
      app: redis-sts
  serviceName: redis-service
  replicas: 3
  template:
    metadata:
      labels:
        app: redis-sts
    spec:
      terminationGracePeriodSeconds: 10
      containers:
        - name: redis-container
          image: redis:latest
          command: ["redis-server"]
          args: ["/redis/conf/redis.conf"]
          volumeMounts:
            - name: redis-configmap-volume
              mountPath: /redis/conf/
            - name: redis-pvc
              mountPath: /redis/data
          resources:
            requests:
              cpu: 1
              memory: 4Gi
      volumes:
        - name: redis-configmap-volume
          configMap:
            name: redis-config
  volumeClaimTemplates:
    - metadata:
        name: redis-pvc
      spec:
        accessModes: [ "ReadWriteOnce" ]
        resources:
```

```

      requests:
        storage: 1Gi
---
apiVersion: v1
kind: Service
metadata:
  name: redis-service
spec:
  ports:
    - port: 6379
  clusterIP: None
  selector:
    app: redis-sts

```

Compared to the MariaDB StatefulSet, it's a similar setup, other than the application specific differences, like the different ports used, the container image of course and the mounting of the config map into `/redis/conf`.

So that's a 1-replica StatefulSet. It's a bit more convenient than using a Deployment for such a workload, as Kubernetes can take care of creating the PersistentVolumeClaim automatically.

If you delete the StatefulSet object, the PersistentVolumeClaim object will remain. If you then re-create the StatefulSet, it will re-attach to the same PersistentVolumeClaim, so no data is lost. Deleting the PersistentVolumeClaim object itself *can* delete the underlying data though, depending on how the storage class is configured. If you care about the data being stored (e.g. it's more precious than a cache that can be re-created), be sure to follow the steps in Section 9.1.3 to setup a StorageClass that will retain the underlying cloud resources if the PersistentVolumeClaim object is deleted for whatever reason.

As previously, we can scale this StatefulSet but that would simply duplicate the Pods. In some cases this may be desirable, but for the Redis deployment defined here, it would just give us more individual Redis instances. The next section goes into detail about how to setup a multiple pod architecture within a single StatefulSet, where each unique Pod is configured differently, based on the ordinal of the pod.

9.2.2 Deploying a Multi-Role StatefulSet

The real power of stateful set comes into play when you need to have multiple pods. When designing an application that will use stateful set, pod replicas within the stateful set need to know about each other and communicate with each other as part of the stateful application design. This is the benefit though of using stateful set because each of the pods gets a unique identifier in a set known as the ordinal. You can use this uniqueness and guaranteed ordering to assign different roles to the different unique Pods in the set and associate the same persistent disk through updates and even deletion and recreate.

For this example we'll take the single pod Redis StatefulSet from the previous section and convert it to a three-pod setup with two separate roles. In Redis parlance, one will be assigned the role of "master", the remaining as "replicas" (not to be confused with replicas in Kubernetes parlance used in Deployment and StatefulSet to mean pod instances).

Building on the example in the previous section, we'll add a second file to our configuration directory for the replica role. I've chosen Redis for the demo here as it is one of the most popular stateful applications to run on Kubernetes.

Listing 9.8 StatefulSet_Redis_Replicated/config_map.yaml: ConfigMap containing two Redis Configuration files, one for each role

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: redis-config
data:
  master.conf: |
    bind 0.0.0.0
    port 6379
    protected-mode no
    appendonly yes
    dir /redis/data
  replica.conf: |
    replicaof redis-0.redis-service 6379
    bind 0.0.0.0
    port 6379
    protected-mode no
    appendonly yes
    dir /redis/data
```

ConfigMaps are simply a convenient way for us to define two configuration files (one for each of the two roles). We could equally build our own container using the Redis base image, and put these two files in there. But since this is the only customization we need, it's simpler to just define them here, and mount them into our container.

Next, we'll update the StatefulSet deployment to use an init container (a container that runs during the initialization of the Pod) to set the role of each pod replica. The script that runs in this init container looks up the ordinal of the pod being initialized to determine its role and copies the relevant configuration for that role (recall that a special feature of StatefulSets is that each pod is assigned a unique ordinal). We can arbitrarily use the ordinal value of 0 to designate the master Redis pod, while assigning the remaining pods the replica role.

This technique can be applied to a variety of different stateful workloads where you have multiple roles. If you're looking for MariaDB, there's a great guide¹ provided with the Kubernetes docs.

¹ <https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/>

Listing 9.9 StatefulSet_Redis_Replicated/redis-statefulset.yaml: Multi-role Redis StatefulSet

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  selector:
    matchLabels:
      app: redis-sts
  serviceName: redis-service
  replicas: 3
  template:
    metadata:
      labels:
        app: redis-sts
    spec:
      terminationGracePeriodSeconds: 10
      initContainers:
        - name: init-redis
          image: redis:latest
          command:
            - bash
            - "-c"
            - |
              set -ex
              # Generate server-id from pod ordinal index.re
              [[ `hostname` =~ -([0-9]+)$ ]] || exit 1
              ordinal=${BASH_REMATCH[1]}
              echo "ordinal ${ordinal}"
              # Copy appropriate config files from config-map to emptyDir.
              mkdir -p /redis/conf/
              if [[ $ordinal -eq 0 ]]; then
                cp /mnt/redis-configmap/master.conf /redis/conf/redis.conf
              else
                cp /mnt/redis-configmap/replica.conf /redis/conf/redis.conf
              fi
              cat /redis/conf/redis.conf
          volumeMounts:
            - name: redis-config-volume
              mountPath: /redis/conf/
            - name: redis-configmap-volume
              mountPath: /mnt/redis-configmap
      containers:
        - name: redis-container
          image: redis:latest
          command: ["redis-server"]
          args: ["/redis/conf/redis.conf"]
          volumeMounts:
            - name: redis-config-volume
              mountPath: /redis/conf/
            - name: redis-pvc
              mountPath: /redis/data
          resources:
            requests:
              cpu: 1
              memory: 4Gi
      volumes:
        - name: redis-configmap-volume

```

```

    configMap:
      name: redis-config
    - name: redis-config-volume
      emptyDir: {}
  volumeClaimTemplates:
    - metadata:
        name: redis-pvc
      spec:
        accessModes: [ "ReadWriteOnce" ]
        resources:
          requests:
            storage: 1Gi
---
apiVersion: v1
kind: Service
metadata:
  name: redis-service
spec:
  ports:
    - port: 6379
  clusterIP: None
  selector:
    app: redis-sts

```

There's a bit to unpack here. The main difference to our single-instance Redis StatefulSet is the presence of an init container. This init container, as its name suggests, runs during the initialization phase of the Pod. It mounts two volumes, the ConfigMap, and a new volume "redis-config-volume".

```

  volumeMounts:
    - name: redis-config-volume
      mountPath: /redis/conf/
    - name: redis-configmap-volume
      mountPath: /mnt/redis-configmap

```

The `redis-config-volume` is of type `emptyDir`, which allows data to be shared between containers, but does not persist data if the pod is rescheduled (unlike `PersistentVolume`). All we're using this `emptyDir` volume is to store the config, and this is ideal for that.

```

  command:
    - bash
    - "-c"
    - |
      set -ex
      # Generate server-id from pod ordinal index.
      [[ `hostname` =~ -([0-9]+)$ ]] || exit 1
      ordinal=${BASH_REMATCH[1]}
      # Copy appropriate config files from config-map to emptyDir.
      mkdir -p /redis/conf/
      if [[ $ordinal -eq 0 ]]; then
        cp /mnt/redis-configmap/master.conf /redis/conf/redis.conf
      else
        cp /mnt/redis-configmap/replica.conf /redis/conf/redis.conf
      fi

```

The bash command that the container runs copies the master, or replica config from the configmap volume (mounted at `/mnt/redis-configmap`) to this shared emptyDir volume (mounted at `/redis/conf`), depending on the ordinal number of the pod. That is, if the pod is “redis-0” the `master.conf` file is copied, for the rest, `replica.conf` is copied.

The main container then mounts the same “redis-config-voulme” at `/redis/conf`, and the redis process is started and told to use whatever configuration resides at `/redis/conf/redis.conf`.

9.3 Migrating/Recovering Disks

Now I know what you’re thinking: can I really trust Kubernetes with my precious data? There’s a bit too much magic going on here, how can I be confident that my data is safe, and recoverable if the Kubernetes cluster goes away?

Time to build some confidence. Let’s create a stateful workload in Kubernetes. Then completely delete every Kubernetes object associated with it and try to recreate that workload from scratch, relinking it to the underlying cloud disk resources.

One thing to be very aware of is that commonly by default, volumes that Kubernetes creates are deleted if you delete the associated bound `PersistentVolumeClaim` because they are configured with the `Delete` `retainPolicy`. With this policy set in the `StorageClass`, deleting the `StatefulSet` doesn’t itself delete the `PersistentVolumeClaim`, which is good (forcing admins to manually clean up the `StatefulSet`’s `PersistentVolumeClaims` if they do wish to delete them, but avoiding accidental delete). But, deleting the `PersistentVolumeClaim` objects *will* delete the underlying disk resources and it’s not that hard to do (e.g. by passing `--all` to the relevant `kubect1 delete` command).

So if you value your data, the first thing is to make sure the `StorageClass` that’s used when creating the disks for your precious data has its `reclaimPolicy` set to `Retain`, not `Delete`. This will preserve the underlying cloud disk when the Kubernetes objects are deleted allowing you to manually re-create the `PersistentVolumeClaim` / `PersistentVolume` pairing in the same, or a different cluster (which I will demonstrate). To run this experiment using the configuration provided, follow the steps in 9.1.3 to set a default `StorageClass` to be configured as `Retain`, so that the underlying cloud disk resource won’t be deleted. Note that changes to the `reclaimPolicy` only apply for disks created *after* the change. For any existing `PersistentVolumes`, you’ll need to update them manually.

With our `reclaimPolicy` set correctly we can now deploy Redis, and add some data to our newly created data volumes. First, deploy the example from 9.2.2 . Once it’s running, let’s add some data which we can use to validate our ability to recover our Kubernetes `StatefulSet` after we delete it. To add data, exec into the master pod and run the `redis-cli` tool. You can do both with the following command:

```
kubect1 exec -it redis-0 -- redis-cli
```

Next, we’ll add some data to Redis. If you’ve not used Redis before, don’t worry about this—we’re just adding some trivial data so as to prove that we can recover it.

```
127.0.0.1:6379> SET capital:australia "Canberra"
OK
127.0.0.1:6379> SET capital:usa "Washington"
OK
```

If you like, at this point you can delete the StatefulSet, and re-create it, then exec back into the CLI and test the data. Here's how:

```
$ kubectl delete -f redis-statefulset.yaml; kubectl create -f redis-statefulset.yaml
service "redis-service" deleted
statefulset.apps "redis" deleted
service/redis-service created
statefulset.apps/redis created

$ kubectl exec -it redis-0 -- redis-cli
127.0.0.1:6379> GET capital:usa
"Washington"
```

This works (the data is persisted) because when the StatefulSet is re-created, it references the same PersistentVolumeClaim which has our data for Redis to load when it boots, and so Redis picks off right where it left off.

Good so far. Now let's take a more drastic step and delete the PVC and the Volume, and attempt to re-create. The re-creation can optionally be done in a completely new cluster if you like, to simulate the entire cluster being deleted (just be sure to use the same cloud region so the disk can be accessed).

Before we delete those objects though, let's save their configuration. This isn't strictly necessary, you certainly *can* re-create them from scratch if needed, but it will help save some time.

```
$ kubectl get pvc,pv
$ kubectl get -o yaml persistentvolumeclaim/redis-pvc-redis-0 > pvc.yaml
$ kubectl get -o persistentvolume/PV_NAME > pv.yaml
```

Now, the nuclear option: delete the stateful set and the pvc.

```
$ kubectl delete pvc,pv --all
```

Due to the `Retain` policy on the `StorageClass` (I hope you did use a storage class with `Retain` as instructed, otherwise that data is gone!), the cloud disk resource will still exist. Now it's just a matter of manually creating a PV to link to that disk, and a PVC to link to that.

Here's what we know:

- We know (or can find out) the name of the underlying disk resource in our cloud provider
- We know the name of the PVC that the StatefulSet will consume (`redis-pvc-redis-0`)

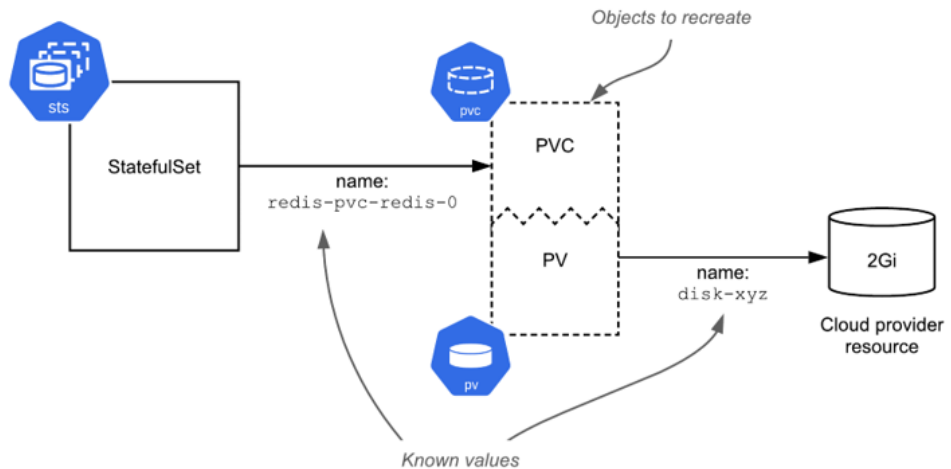


Figure 9.5: the known values, and the objects we need to recreate

So what we need to do is create a PVC with the name `redis-pvc-redis-0` that is bound with a PV that references the disk. Importantly, the PVC needs to name the PV, and the PV needs to define the bound PVC, otherwise the PVC could bind a different PV, and the PV could be bound by a different PVC.

Creating the objects from our saved config with `kubectl create -f pv.yaml` and `kubectl create -f pvc.yaml` unfortunately won't work. This is because that configuration also exported the *state* of the binding, and unique identifiers that don't carry over when you delete and create the object from config. If you try this, you'll see that the PVC Status is `Lost`, and the PV status is `Released`. Not what we want.

To fix this, we just need to remove the binding status and the `uids`:

Edit the PersistentVolume (the configuration we exported to `pv.yaml`) and make two changes:

- Remove the `uid` field from the `claimRef` (the `claimRef` is the pointer to the PVC, the issue is that the PVC's `uid` has changed)
- Set the `storageClassName` to the empty string `""` (we're manually provisioning and don't want to use a `storageClass`).

Edit to the PVC (`pvc.yaml` exported config) and make 2 changes there:

- Delete the annotations such as `pv.kubernetes.io/bind-completed: "yes"` (this PVC needs to be re-bound and this annotation will prevent that)
- Set the `storageClassName` to the empty string `""` (same reason as above)

Alternatively, if you're re-creating this config from scratch, the key is that the `volumeName` of the PVC needs to be set to that of the PV, the `claimRef` of the PV needs to reference the PVC's name and namespace, and both have the `storageClassName` of "".

Again, it's easier to visualize side by side. This figure is based on the configuration I exported when I ran this test, and removed the fields as documented above.

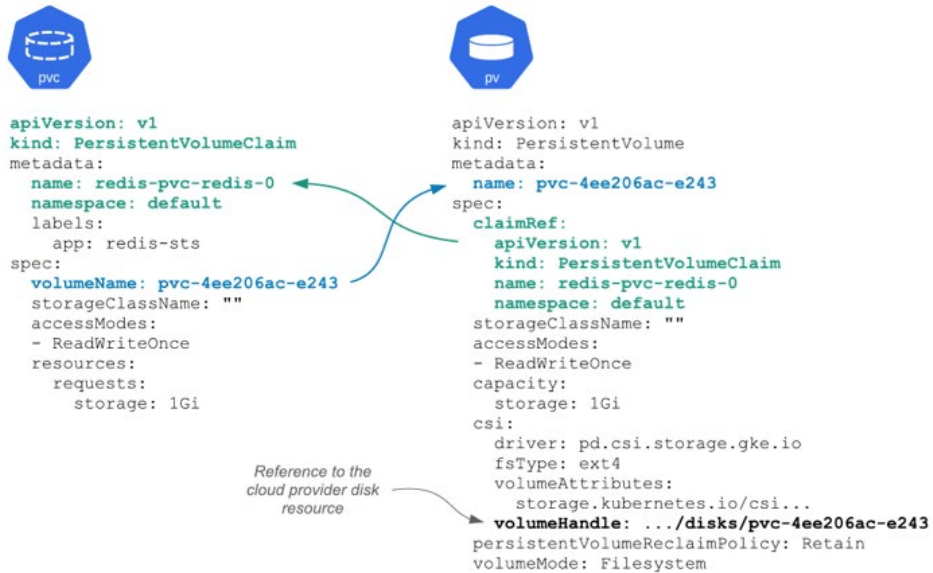


Figure 9.6: Pre-linked PVC and PV objects

Once prepared, you can create both configuration files in the cluster then inspect their status with `kubectl get pvc,pv`.

If it goes correctly, the status for both objects should read "Bound". If instead one or both are listed as Pending or Released, go back and check that they are linked correctly with all the information needed, and without any extra information. Yes, unfortunately, this is a bit of a pain, but it is possible to rebind these objects, provided that the underlying cloud resource is still there (which it will be, since you used the Retain policy on your StorageClass didn't you).

This is what success looks like (with some columns removed for readability):

```
$ kubectl get pvc
NAME                STATUS    VOLUME
redis-pvc-redis-0   Bound    pvc-f0fea6ae-e229

$ kubectl get pv
NAME                RECLAIM POLICY    STATUS    CLAIM
pvc-f0fea6ae-e229   Retain            Bound    default/redis-pvc-redis-0
```

Once you have your manually created PVC and PV objects, it's time to re-create the StatefulSet. As we tested earlier when we deleted and re-created the StatefulSet, as long as the PVC exists with the expected name, it will be re-attached to the Pods of the StatefulSet. The name of the PVCs that the StatefulSet is deterministic, so when we recreate the StatefulSet, it will see the existing PVC objects and reference them, rather than creating new ones. Basically, everything should just work as we re-create these objects using the same names as before.

Notice in this example, that despite the fact the StatefulSet has 3 PVCs and therefore 3 associated disks, we only manually recovered 1 disk, that of the master. This is because the replicas will automatically be recreated.

Once the StatefulSet is deployed, let's exec into one of those replicas and see if our data is still there:

```
$ kubectl create -f redis-statefulset.yaml
service "redis-service" deleted
statefulset.apps "redis" deleted
service/redis-service created
statefulset.apps/redis created

$ kubectl exec -it redis-1 -- redis-cli
127.0.0.1:6379> GET capital:australia
"Canberra"
```

If you can read back the data we wrote to Redis earlier, the congratulations! You've recovered the StatefulSet from scratch.

I hope this has given you some confidence about the persistence of the data when the `Retain` policy is used. As demonstrated, you can completely delete all the objects (heck, even the entire cluster), and re-create all the links from scratch. It's a bit laborious, but it's possible. To reduce the toil, it's advisable (but not essential) to export the config for our PVC and PV objects and store them in our configuration repository, to make it faster to recreate these objects in the future.

9.4 Summary

One of the strengths of Kubernetes is that it can deftly handle deployments of stateful applications. It's not confined only to stateless applications, as some other application platforms are. There are several tools at your disposal to confidently host stateful applications, including:

- The ability to mount disks from your cloud provider directly into Pods
- Creating simple 1-Pod stateful deployments
- The complex lifecycle of the `PersistentVolume` and `PersistentVolumeClaim` objects including the metamorphosis they go through as they are bound together into a single logical object
- Configuring `StorageClasses` to enable dynamic storage with your preferred configuration options (most importantly, the option to retain the cloud provider disk resources should the Kubernetes objects be deleted)
- Deploying simple 1-Pod stateless applications using `StatefulSet` or even just as a `Deployment`
- The benefit that the Kubernetes `StatefulSet` offers stateful deployments, being that each pod and their associated disks are assigned a specific number (ordinal) so that when pods are re-created, the disk is re-mounted to the pod with the same ordinal.
- Using the properties of `StatefulSet` to deploy more complex multi-pod stateful applications that employ different roles (such as a master copy and replicas) using the ordinal number of the Pod
- How to re-create bound `PersistentVolume` and `PersistentVolumeClaim` object pairs to reference existing disks (for example, when recovering a deployment or migrating between two clusters)

10

Background Processing

This chapter covers

- **How to process background tasks in Kubernetes**
- **The Kubernetes Job and CronJob objects**
- **When to use (and not use) Job objects for your own batch processing workloads**
- **Creating a custom task queue with Redis**
- **Implementing a background processing task queue with Kubernetes**

In the prior chapters we looked at developing services that are exposed on an IP address, whether it's an providing an external service on a public address, or an internal service on a cluster local IP. But what about all the other computation that you may need to do that isn't directly part of a request-response chain, like resizing a bunch of images, sending out device notifications, processing financial data, or rendering a movie one frame at a time? Background tasks are all the compute processes that take an input and produce an output without being part of the synchronous processing of requests in the way services are.

You can process background tasks using Deployment, or the Kubernetes Job object. Deployment is ideal for a continuously running task queue like the one most web applications run for tasks like image resizing. The Kubernetes Job construct is great for running one-off maintenance tasks, periodic tasks (via CronJob), and processing a batch workload when there is a set amount of work to complete.

Terminology: task or job

Practitioners routinely use the terms “task” and “job” interchangeably when referring to a background computation for example “job queue” and “task queue”, “background job”, and “background task”. Since Kubernetes has an object named Job, to reduce ambiguity I will always sentence-case “Job” when referring to the object itself, and will use the word “task” (like “background task” and “task queue”) when referring to the general concept of background processing, however it is implemented.

This chapter covers using both Deployment and Job for background task processing. By the end, you’ll be able to configure a continuous background task processing queue for your web application, defined batch workloads with an end-state, as well as period and one-off tasks, all in Kubernetes.

10.1 Background Processing Queues

Most web applications deployed in the wild have a major background task processing component in order to handle processing tasks that can’t be completed in the short HTTP request/response time window. User research conducted by Google observes that the longer the page load is, the higher the chance the user will “bounce” (i.e. leave the page and go somewhere else), so it’s generally a mistake to try and do any heavy lifting while the user is waiting, and instead put that task on a background queue, and keep the user apprised of the progress. Page load speeds need to be on everyone’s mind, from the front end developers to the backend; it’s a collective responsibility.

The probability of bounce increases 32% as page load time goes from 1 second to 3 seconds.

Google/SOASTA Research, 2017.¹

There’s a lot that goes into the time it takes to load the page, and many aspects like image sizes and JavaScript are out of scope for Kubernetes. A relevant metric to consider when looking at your workload deployments in Kubernetes is “time to first byte” or TTFB. This is the time it takes for your web server to complete its processing of the request and the client to start downloading the response. To achieve a low overall page loading time, it’s critical to reduce the TTFB time and respond in sub-second times. That pretty much rules out any kind of data processing that happens “inline” as part of the request. Need to create a ZIP file to serve to a user, or shrink an image they just uploaded? Best not to do it in the request itself.

As a result of this, the common pattern is to run a continuous background processing queue. The web application hands off tasks it can’t do inline like process data, which gets picked up by the background queue. The web application might show a spinner or some other UI affordance while it waits for the background queue to do its thing, or may email the user when the results are ready, or simply prompt the user to come back later. How you architect

¹ <https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/page-load-time-statistics/>

your user interaction is up to you. What we'll cover here is how to deploy this kind of background processing task queue in Kubernetes.

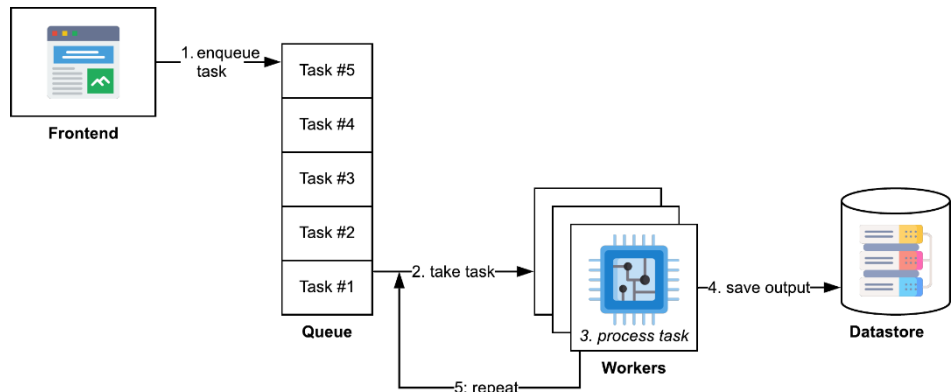


Figure 10.1: Front-end web server with a background task queue

Recall that a Deployment (as covered in chapter 3) is a workload construct in Kubernetes that's purpose is to maintain a set of continuously running Pods. For background task processing queues, you need a set of continuously running pods to serve as your task workers. So that's a match! It doesn't matter that the Pods in the Deployment won't be exposed with a Service, the key is that you want at least one worker to be continuously running. You'll be updating this Deployment with new container versions and scaling it up and down just like with a deployment that serves your frontend requests, so everything we've learnt so far can be applied equally to a Deployment of background task workers.

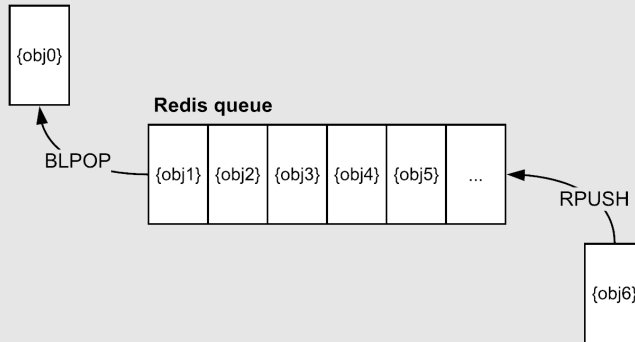
10.1.1 Creating a custom task queue

The worker Pods that you deploy in your deployment have a simple role: take an input and produce an output. But where do they get the input from? For that, you'll need a queue on which other components of the application can add tasks. This queue will store the list of pending tasks, which the worker Pods will process. There are a bunch of off-the-shelf solutions for background queues (some which I mention in section 10.1.2), but to best understand how these work, let's create our own!

For the queue data store, we'll be using the same Redis deployment we created in the previous chapter. Redis includes built-in support for queues, making it perfect for this task (and many off-the-shelf solutions also use Redis). The design of our work system is pretty straightforward: the web application role enqueues tasks to Redis (we can emulate this role by manually adding tasks), and worker Pods from our Deployment pop the tasks, perform the work and waits for the next one.

Queues in Redis

Redis has several convenient data structures out of the box. The one we're using today is a queue. There are two functions we'll be using with this queue structure to get FIFO (first-in, first-out) ordering which is typical of a background queue (processing items in the order they are added): `RPUSH` to add items to the back of the queue, and `BLPOP` to pop items from the front of the queue and block if none are available.



If you think of the queue going from right to left, where the rightmost item is at the back of the queue, and the leftmost item is the front, then the “L” and “R” function prefixes will make sense (`RPUSH` to push an object on the right, and `BLPOP` to blocking pop the leftmost item). The further “B” prefix refers to the blocking form of the function (in this case the blocking version of `LPOP`), which will cause it to wait for an item in the event the queue is empty rather than returning right away with `nil`. We could simply use `LPOP` in our own retry loop, but it's useful to block on the response to avoid a “busy wait” which would consume more resources. This way we can leave that task to Redis as well.

As a concrete but trivial example, our task will take as input an integer n , and calculate π using the Leibniz series formula² with n iterations (the more iterations when calculating π in this way, the more accurate the end result). In practice your task will likely be a lot more complex and complete whatever arbitrary processing you need it to do, and will take as input a URL, or dictionary of key/value parameters. The concept is the same.

CREATING A WORKER CONTAINER

Before we get to the Kubernetes deployment, we'll need to create our worker container. I'll use Python again for this sample, as we can implement a complete task queue in a few lines of Python. The complete code for this container can be found in the `Chapter10/worker` folder in the source code that accompanies this book. It consists of 3 python files, presented as follows: `pi.py` contains our work function.

² https://en.wikipedia.org/wiki/Leibniz_formula_for_pi

This is where the actual computation happens. It has no concept of being in a queue, it just does the processing. In your own case, you'd replace this with whatever computation you need to do, e.g. creating a ZIP file, or compressing an image.

Listing 10.1 /pi_worker/pi.py

```
from decimal import *

# Calculate pi using the Gregory-Leibniz infinity series
def leibniz_pi(iterations):

    precision = 20
    getcontext().prec = 20
    piDiv4 = Decimal(1)
    odd = Decimal(3)

    for i in range(0, iterations):
        piDiv4 = piDiv4 - 1/odd
        odd = odd + 2
        piDiv4 = piDiv4 + 1/odd
        odd = odd + 2

    return piDiv4 * 4
```

Then, we have our worker implementation that will take the object at the head of the queue with the parameters of the work needed to be done, and perform the work by calling the `leibniz_pi` function above. For your own implementation, the object that you queue just needs to contain the relevant function parameters for the task, like the details of the ZIP file to create, or image to process.

Listing 10.2 /pi_worker/pi_worker.py

```
import os
import redis
from pi import *

redis_host = os.environ.get('REDIS_HOST')
assert redis_host != None
r = redis.Redis(host=redis_host, port='6379', decode_responses=True)

print("starting")
while True:
    task = r.blpop('queue:task') #A
    iterations = int(task[1])
    print("got task: " + str(iterations))
    pi = leibniz_pi(iterations) #B
    print(pi)
```

#A Pop the next task (and block if there are none in the queue)

#B Perform the work

To pop our Redis-based queue, we use the Redis `BLPOP` command which will get the first element in the list, and block if the queue is empty (thus, wait for more tasks to be added). There is more we would need to do to make this production-grade, such as add signal handling for when the Pod is terminated (covered in section 10.1.2), but this is enough for now.

Lastly, we have a little script to add some work to this queue. In the real world, you will queue tasks as needed (by calling `RPUSH` with the task parameters), such as in response to user actions (e.g. queuing the task to resize an image in response to the user uploading an image). For our demonstration, we can seed our task queue with some random values. The following code will create 100 sample tasks using a random value for our task input integer (with a value in the range 1 to 10 million). Here's some python code to add some tasks to our queue.

Listing 10.3 /pi_worker/add_tasks.py

```
import os
import redis
import random

redis_host = os.environ.get('REDIS_HOST')
assert redis_host != None
r = redis.Redis(host=redis_host, port='6379', decode_responses=True) #A

random.seed()
for i in range(0, 10):
    rand = random.randint(10,100)
    iterations = rand * 100000
    r.rpush('queue:task', iterations)
    print("added task: " + str(iterations)) #B

print("queue depth", str(r.llen('queue:task')))
print ("done")
```

#A Connect to our Redis service

#B Add the task with the "iterations" value to our 'queue:task' list

The `rpush` python command used above maps to the `RPUSH` command³ which adds the given value (in our case, an integer) to the list specified with the key (in our case the key is "queue:task"). If you've not used Redis before, you might be expecting something more complex, but this is all that's needed to create a queue there's no pre-configuration or schemas needed.

Bundling these python scripts into a container is pretty simple. We can use the official python base image, and add the Redis dependency (see chapter 2 if you need a refresher on how to build such containers). For the default container entry point, we'll start our worker loop with `python3 pi_worker.py`.

Listing 10.4 /pi_worker/Dockerfile

```
FROM python:3
RUN pip install redis
COPY . /app
WORKDIR /app
CMD python3 pi_worker.py
```

³ <https://redis.io/commands/rpush>

With our Python worker container created, we can now get to the fun part of deploying it to Kubernetes!

DEPLOYING TO KUBERNETES

Here's what the Kubernetes architecture looks like, we have the StatefulSet that runs Redis, and the Deployment which runs the worker pods. There is also the web application role which adds the tasks, but we'll just do that manually for this example.

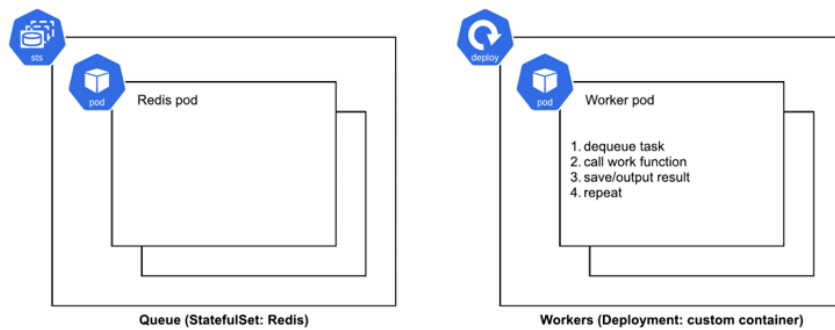


Figure 10.2: Kubernetes architecture of the background processing task queue

Our worker pods will be deployed in a hopefully now-familiar Deployment configuration (see chapter 3). We'll pass in the location of our Redis host using an environment variable, that references the internal service host (as per section 7.1.3):

Listing 10.5 /10.1.1_TaskQueue/deploy_worker.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: pi-worker
spec:
  replicas: 2
  selector:
    matchLabels:
      app: pi
  template:
    metadata:
      labels:
        app: pi
    spec:
      containers:
        - name: pi-container
          image: docker.io/wdenniss/pi_worker:v1
          env:
            - name: REDIS_HOST #A
              value: redis-0.redis-service #A
            - name: PYTHONUNBUFFERED #B
              value: "1" #B

```

#A The Kubernetes service host name of the master Redis pod

#B Env variable to instruct Python to output all print statements immediately

Notice how there's nothing special at all about this deployment when compared to the deployments we've seen so far that are exposed with services. It's just a bunch of Pods that we happen to have given the role of task workers.

Our worker pods are expecting a Redis instance, so let's deploy that first. We can use the one from chapter 9, the solutions in the `9.2.1_StatefulSet_Redis_SinglePod` and `9.2.2_StatefulSet_Redis_Replicated` folders both work for our purposes here. From the code sample root folder, simply run:

```

$ kubectl create -f Chapter09/9.2.2_StatefulSet_Redis_Replicated
$ kubectl get pods

```

NAME	READY	STATUS	RESTARTS	AGE
redis-0	1/1	Running	0	20s
redis-1	1/1	Running	0	13s
redis-2	0/1	Init:0/1	0	7s

Now create our worker deployment.

```

kubectl create -f Chapter10/10.1.1_TaskQueue/deploy_worker.yaml

```

Finally, verify everything is working correctly: you should see 5 running Pods.

```

$ kubectl get pods

```

NAME	READY	STATUS	RESTARTS	AGE
pi-worker-55477bdf7b-7rmhp	1/1	Running	0	2m5s
pi-worker-55477bdf7b-ltcsd	1/1	Running	0	2m5s
redis-0	1/1	Running	0	3m41s
redis-1	1/1	Running	0	3m34s
redis-2	1/1	Running	0	3m28s

Watching Rollout Progress

The `get` commands I show here like `kubectl get pods` give you a point-in-time status. Recall from chapter 3 that there are 2 great options for watching your rollout, you can append `-w` to `kubectl` commands which is Kubernetes' in-built watching option, for example `kubectl get pods -w`, or you can use my favorite, the Linux `watch` command. I use it like so: `watch -d kubectl get pods` which will refresh the status every 2 seconds, and highlight changes. You can also customize the refresh rate. To keep things simple in the book, I won't add watches to every command I share, but remember that they are available for use.

Now that our app is deployed, we can look at the logs to see what it's doing. Unfortunately, there's no built-in way in Kubernetes to stream logs from multiple pods like our 2 workers at the same time, but we can randomly pick one and follow its logs like so:

```
$ kubectl logs -f deployment/pi-worker
Found 2 pods, using pod/pi-worker-55477bdf7b-7rmhp
Starting
```

If you want to view the logs for all pods in the deployment (but not stream them), that can also be done by referencing the labels from the deployment, in our case the `app=pi` label, like so:

```
$ kubectl logs --selector app=pi
starting
starting
```

However you view the logs, we can see that the Pod has printed "starting", and nothing else, which is because our Pod is waiting on tasks to be added to the queue. Let's add some tasks for it to work on.

ADDING WORK TO THE QUEUE

Normally, it will be the web application or another process that will be adding work for the background queue to process. All that the web application need do is call `redis.rpush('queue:task', object)` **with the object that represents the tasks.**

For this example, we can run the `add_tasks.py` script that we included in our container (documented above) for scheduling some tasks. We can execute a one-off command on the container in one of our pods like so:

```
$ kubectl exec -it deploy/pi-worker -- python3 add_tasks.py
added task: 9500000
added task: 3800000
added task: 1900000
added task: 3600000
added task: 1200000
added task: 8600000
added task: 7800000
added task: 7100000
added task: 1400000
added task: 5600000
queue depth 8
done
```

Note that when we pass in `deploy/pi-worker` here, `exec` will pick one of our Pods randomly to run the actual command on (this can even be a Pod in the Terminating state, so be careful!). You can also run the command directly on the pod of your choice with `kubectl exec -it POD_NAME python3 add_tasks.py`.

VIEWING THE WORK

With tasks added to the queue, we can observe the logs of our worker pods to see how they're doing.

```
$ kubectl logs -f deployment/pi-worker
Found 2 pods, using pod/pi-worker-54dd47b44c-bjccg
starting
got task: 9500000
3.1415927062213693620
got task: 8600000
3.1415927117293246813
got task: 7100000
3.1415927240123234505
```

This worker is getting the task (calculate Pi with n iterations of the Gregory-Leibniz infinity series algorithm) and performing the work.

10.1.2 Signal Handling in Worker Pods

One thing to note is that the above worker implementation has no SIGTERM handling, which means it won't shutdown gracefully when the Pod needs to be replaced. There are a lot of reasons why a Pod might be terminated, including if you update the deployment, or the Kubernetes node is upgraded, so this is a very important signal to handle.

In Python we can fix this with a SIGTERM handler that will instruct our worker to terminate once it finishes its current task. We'll also add a timeout to our queue-pop call so the worker can check the status more frequently. For your own work, look up how to implement SIGTERM signal handling in your language of choice.

Let's add termination handling to shut down the worker when SIGTERM is received:

Listing 10.6 /pi_worker2/pi_worker.py

```

import os
import signal
import redis
from pi import *

redis_host = os.environ.get('REDIS_HOST')
assert redis_host != None
r = redis.Redis(host=redis_host, port= '6379', decode_responses=True)

running = True #A

def signal_handler(signum, frame): #B
    print("got signal") #B
    running = False #B

signal.signal(signal.SIGTERM, signal_handler) #B

print("starting")
while running:
    task = r.blpop('queue:task', 5) #C
    if task != None:
        iterations = int(task[1])
        print("got task: " + str(iterations))
        pi = leibniz_pi(iterations)
        print (pi)

```

#A Add a running state variable instead of looping indefinitely

#B Register a signal handler to set the running state to false when SIGTERM received

#C Pop the next task, but only wait for 5 seconds so the loop can terminate. Returns "None" if there are no tasks in the queue after 5 seconds.

And deploy this revision in an updated Deployment, specifying the new image along with `terminationGracePeriodSeconds` to request 2 minutes to handle that SIGTERM by wrapping up the current work and exiting.

Listing 10.7 TaskQueue2/deploy_worker.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: pi-worker
spec:
  replicas: 2
  selector:
    matchLabels:
      app: pi
  template:
    metadata:
      labels:
        app: pi
    spec:
      containers:
        - name: pi-container
          image: docker.io/wdenniss/pi_worker:v2
          imagePullPolicy: Always
          env:
            - name: REDIS_HOST
              value: redis-0.redis-service
            - name: PYTHONUNBUFFERED
              value: "1"
          resources:
            requests:
              cpu: 250m
              memory: 250Mi
      terminationGracePeriodSeconds: 120

```

Together, the signal handling in the pod and the termination grace period means that this Pod will stop accepting new jobs once it receives the SIGTERM, and will have 120s to wrap up any current work. Adjust the `terminationGracePeriodSeconds` as needed for your own workloads.

There's a few more things we didn't consider here. For example. If the worker were to crash while processing a task, then that task would be lost (as it was removed from the queue, but not completed). Also there's only minimal observability and other functions. The goal of the above sample is not to provide a complete queue system, but rather to demonstrate conceptually how they work. You could continue to implement fault tolerance and other functionality or adopt an open source background task queue and have it do that for you. That choice is yours.

10.1.3 Scaling Worker Pods

Scaling the worker pods is the same technique for any Deployment, as covered in Chapter 6. You can set the replica count manually, or with a Horizontal Pod Autoscaler (HPA). Since our example workload is CPU-intensive, the CPU metric works well for scaling using a HPA, so let's set one up now.

Listing 10.8 HPA/hpa.yaml

```

apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: pi-worker-autoscaler
spec:
  minReplicas: 2
  maxReplicas: 10
  targetCPUUtilizationPercentage: 20
scaleTargetRef:
  apiVersion: apps/v1
  kind: Deployment
  name: pi-worker

```

This will scale our deployment to between 2 and 10 Pods, aiming for each Pod to be using less than 20% of a CPU.

HPA uses Absolute CPU utilization numbers

Note that this utilization target isn't expressed as a percentage of the Pod's total requested CPU, but rather an absolute number in CPU units. Since our Pod deployment (as revised in 10.1.2) requests 250mCPU (i.e. 25% of a core), a 20% target in the HPA means that a scale up will occur when the average Pod is consuming greater than 20% of a core (200mCPU). In relative terms, this works out to 80% of its total requested capacity—being 20% of a core actual usage divided by 25% of a core requested, but don't make the mistake of specifying that relative percentage as your `targetCPUUtilizationPercentage`.

Create the HPA like so:

```
kubectl create -f kubectl create -f Chapter10/10.1.3_HPA
```

With the HPA in place, you can repeat the add tasks command from above, and watch the HPA do its thing. The `kubectl get` command supports multiple resource types, so you can run `kubectl get pods,hpa` (which I prefix with the Linux `watch` command, naturally) in order to observe all the components interacting.

```
$ kubectl exec -it deploy/pi-worker -- python3 add_tasks.py
```

```
$ kubectl get pods,hpa
```

NAME	READY	STATUS	RESTARTS	AGE
pod/pi-worker-54dd47b44c-22x9b	1/1	Running	0	2m42s
pod/pi-worker-54dd47b44c-9wppc	1/1	Running	0	2m27s
pod/pi-worker-54dd47b44c-bjccg	1/1	Running	0	13m
pod/pi-worker-54dd47b44c-f79hx	1/1	Running	0	2m42s
pod/pi-worker-54dd47b44c-fptj9	1/1	Running	0	2m27s
pod/pi-worker-54dd47b44c-hgbqd	1/1	Running	0	2m27s
pod/pi-worker-54dd47b44c-lj2bk	1/1	Running	0	2m27s
pod/pi-worker-54dd47b44c-wc267	1/1	Running	0	2m10s
pod/pi-worker-54dd47b44c-wk4dg	1/1	Running	0	2m10s
pod/pi-worker-54dd47b44c-x2s4m	1/1	Running	0	13m
pod/redis-0	1/1	Running	0	56m
pod/redis-1	1/1	Running	0	56m
pod/redis-2	1/1	Running	0	56m

NAME	MINPODS	MAXPODS	REPLICAS	AGE	REFERENCE	TARGETS
horizontalpodautoscaler.autoscaling/pi-worker-autoscaler	2	10	10	3m46s	Deployment/pi-worker	66%/20%

10.1.4 Open Source Task Queues

So far, we’ve been building our own task queue. I find it best to get hands on to understand how things work. However, you likely don’t need to implement a task queue yourself from scratch, since others have done the work for you.

For Python, RQ (<https://python-rq.org/>) is a popular choice that allows you to basically enqueue a function call with a bunch of parameters. No need to even wrap this function in an object that implements a particular abstract method.

For Ruby developers, Resque (<https://github.com/resque/resque>), created by the team at GitHub, is a popular choice. Tasks in Resque are simply ruby classes that implement a `perform` method. The Ruby on Rails framework makes Resque particularly easy to use with its *Active Job* framework that allows Resque (among other task queue implementations) to be used as the queuing backend.

Before going out and building your own queue, I’d recommend looking at these options and more. If you have to build something yourself, or the off-the-shelf options just don’t cut it, then I hope you saw through the earlier examples that it’s at least pretty straightforward to get started.

10.2 Jobs

Kubernetes offers a way to define a finite set of work to process with the *Job construct*. Both Job and Deployment can be used for handling “batch jobs” and background processing in Kubernetes in general. The key difference is that Job is designed to process a finite set of work, and can potentially be used without needing a queue data structure like Redis, while Deployment (as described in section 10.1) is for a continuously running background queue that will need some kind of queue structure for coordination. You can also use Jobs to run one-off and periodic tasks like maintenance operations, which wouldn’t make sense in a Deployment (which would restart the Pod once it finishes).

You may be wondering why a separate construct is needed in Kubernetes to run something once, since stand-alone Pods could do that as well. While it's true that you can schedule a Pod to perform a task and shutdown once its complete, there is no controller to ensure that the task actually completes. For example, if the Pod was evicted due to a maintenance event before it had a chance to complete. Job adds some useful constructs around the Pod to ensure that the task will complete (by rescheduling it if it failed or was evicted), as well as the potential to track multiple completions and parallelism.

At the end of the day, Job is just another higher-order controller in Kubernetes for managing pods, like Deployment and StatefulSet. All three create Pods to run your actual code, just with different logic around scheduling and management provided by the controller. Deployments are for creating a set of continuously running Pods, StatefulSet for pods that have a unique ordinal and can attach disks through persistent volume templates, and Jobs for Pods that should run to completion (potentially multiple times).

10.2.1 Running one-off tasks with Jobs

Jobs are great for running one-off tasks. Let's say you want to perform a maintenance task like clearing a cache, or anything else that is essentially just running a command in your container. Instead of using `kubectl exec` on an existing Pod, you can schedule a Job to run the task as a separate process with its own resources, ensure that the action will complete as requested (or report a failure status), and make it easily repeatable.

Exec should really only be used for debugging running Pods. If you use exec to perform maintenance tasks, then your task is sharing the resources with the Pod which isn't great—the pod may not have enough resources to handle both, and you are impacting the performance. By moving tasks to a Job, they get their own Pod, with their own resource allocation.

Configuration as code for maintenance tasks

Throughout this book, I've been espousing how important it is to capture everything in configuration. By capturing routine maintenance tasks as Jobs, rather than having a list of shell commands to copy/paste, you're building repeatable configuration. If you follow the GitOps approach whereby production changes go through git (as covered in chapter 8), then your maintenance tasks can go through your usual code review process in order to be rolled out into production.

In the previous section we needed to execute a command in the container to add some work to our queue, and we used `kubectl exec` on an existing pod to run `python3 add_tasks.py`. Let's upgrade the process of adding work to be a proper Job with its own Pod. The following Job definition can be used to perform the `python3 add_tasks.py` task on our container named `pi`.

Listing 10.9 Job/job_addwork.yaml

```

apiVersion: batch/v1
kind: Job
metadata:
  name: addwork
spec:
  backoffLimit: 2
  template:
    spec:
      containers:
      - name: pi
        image: docker.io/wdenniss/pi_worker:v2
        command: ["python3", "add_tasks.py"]
        env:
        - name: REDIS_HOST
          value: redis-0.redis-service
        - name: PYTHONUNBUFFERED
          value: "1"
      restartPolicy: Never

```

The `spec` within the `template` within the `spec` pattern may look familiar, and that's because this object embeds a Pod template just as Deployment and StatefulSet do. All the parameters of the Pod can be used here like resource requests and environment variables, with only a couple of exceptions for parameter combinations that don't make sense in the Job context.

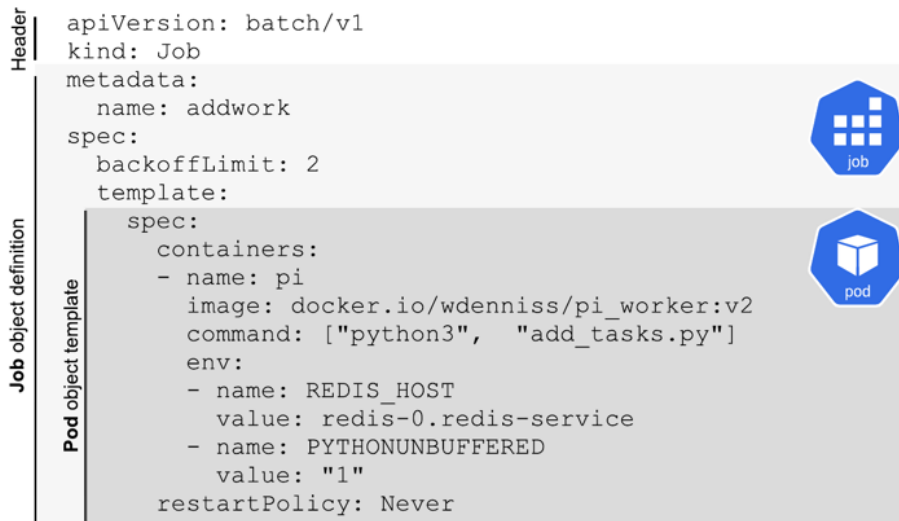


Figure 10.3: Object composition of Job

Our Podspec for the Job has the same environment variables as our Podspec from the deployment. That's the great thing about Kubernetes object composition, the specification is

the same wherever the Pod is embedded. The other changes are the `restartPolicy`, and the `backoffLimit`.

The Pod restart property, a property of the Podspec embedded in the Job, governs whether or not kubelet on the node will try to restart a container that existed with a non-success result in-place. For Jobs, this can be set to `OnFailure` to restart failed containers, or `Never`, which won't. The Always option doesn't make sense for Jobs, as this would restart a succeeded Pod, which is not what Jobs are designed to do (that's more in the domain of Deployment).

The backoff limit is part of the Job, and determines how many times to try and run the Job. This encompasses both crashes, but also node failures. For example, if you set this value to 6, and the Job crashes twice on a node, then that node is removed, the Job controller will reschedule the Pod elsewhere and try it 4 more times. Some practitioners like to use `Never` when debugging as it's easier to see all the failed Pods and query their logs.

Create the Job like any other Kubernetes object, and then observe the progress like so:

```
$ kubectl create -f Chapter10/10.2.1_Job/job_addwork.yaml
$ kubectl get job,pods
```

NAME	COMPLETIONS	DURATION	AGE
job.batch/addwork	1/1	3s	9s

NAME	READY	STATUS	RESTARTS	AGE
pod/addwork-99q5k	0/1	Completed	0	9s
pod/pi-worker-6f6dfdb548-7krpm	1/1	Running	0	7m2s
pod/pi-worker-6f6dfdb548-pzxq2	1/1	Running	0	7m2s
pod/redis-0	1/1	Running	0	8m3s
pod/redis-1	1/1	Running	0	7m30s
pod/redis-2	1/1	Running	0	6m25s

If the Job succeeds, we can watch our worker pods which should become busy with newly added work. If you deployed the HPA earlier, then you'll soon see new containers created, as I did here:

```
$ kubectl get pods,hpa
```

NAME	READY	STATUS	RESTARTS	AGE
pod/addwork-99q5k	0/1	Completed	0	58s
pod/pi-worker-6f6dfdb548-7krpm	1/1	Running	0	7m51s
pod/pi-worker-6f6dfdb548-h6pld	0/1	ContainerCreating	0	8s
pod/pi-worker-6f6dfdb548-pzxq2	1/1	Running	0	7m51s
pod/pi-worker-6f6dfdb548-qpgxp	1/1	Running	0	8s
pod/redis-0	1/1	Running	0	8m52s
pod/redis-1	1/1	Running	0	8m19s
pod/redis-2	1/1	Running	0	7m14s

NAME	MINPODS	MAXPODS	REPLICAS	AGE	REFERENCE	TARGETS
horizontalpodautoscaler.autoscaling/pi-worker-autoscaler	2	10	2	2d4h	Deployment/pi-worker	100%/20%

One thing to note about Jobs is that whether it has completed or not, you won't be able to schedule it again with the same name (i.e. to repeat the action) without deleting it first. That's because, even though the work is now finished, the Job object still exists in Kubernetes. You can delete like any object:

```
$ kubectl delete -f Chapter10/10.2.1_Job/job_addwork.yaml
```

To recap, Job is for when you have some task or work to complete. Our example was to execute a simple command, however this could equally have been a long and complex computational task. If you need to run a one-off background process, simply containerize it, define it in the Job and schedule. When the Job reports itself as “Complete”, the work is done.

Two parameters which we didn’t use to run a one-off task are `completions`, and `parallelism`. These parameters allow you to process a batch of tasks using a single Job object description, which is covered in 10.3. Before we get to that, let’s look how to schedule Jobs at regular intervals.

10.2.2 Scheduling Tasks with Cron Jobs

In the previous section, we took a command that we had executed manually on the cluster and created a proper Kubernetes object to encapsulate it. Now any developer on the team can run perform that task by creating the Job object rather than needing to remember a complex `exec` command.

What about tasks that you need to run repeatedly on a set interval? Kubernetes has you covered with CronJob. CronJob encapsulates a Job object and adds a frequency parameter that allows you to set a daily or hourly (or any interval you like) frequency to run the Job. This is a popular way to schedule tasks like a daily cache cleanup and the like.

Listing 10.10 CronJob/cronjob_addwork.yaml

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: addwork
spec:
  schedule: "*/5 * * * *"
  jobTemplate:
    spec:
      backoffLimit: 2
      template:
        spec:
          containers:
            - name: pi
              image: docker.io/wdenniss/pi_worker:v2
              command: ["python3", "add_tasks.py"]
              env:
                - name: REDIS_HOST
                  value: redis-0.redis-service
                - name: PYTHONUNBUFFERED
                  value: "1"
          restartPolicy: Never
```

You might notice that we just copied the entire specification of the Job (i.e. the `spec` dictionary) in the previous section under this CronJob’s `spec` dictionary as the `jobTemplate` key, and added an extra `spec`-level field named `schedule`. Recall that the Job has its own template for the Pods that will be created (which also have their own `spec`).

So, the CronJob embeds a Job object, which in turn embeds a Pod. I find it helpful to visualize this through object composition, so take a look at the following figure.

preferred expression, rather than waiting a week to verify that the Job you wanted to run weekly actually ran.

```
$ kubectl create -f Chapter10/10.2.2_CronJob/cronjob_addwork.yaml
# kubectl get cronjob,job
NAME                                SCHEDULE      SUSPEND    ACTIVE    LAST SCHEDULE    AGE
cronjob.batch/addwork              */5 * * * *   False      0         <none>           58s
```

Wait a couple of minutes (for this example, the Job is created every 5 minutes, i.e. :00, :05, etc), then you can see the Job, and the Pod that it spawned.

```
$ kubectl get cronjob,job,pods
NAME                                SCHEDULE      SUSPEND    ACTIVE    LAST SCHEDULE    AGE
cronjob.batch/addwork              */5 * * * *   False      0         2m38s            3m11s

NAME                                COMPLETIONS   DURATION    AGE
job.batch/addwork-27237815          1/1           107s        2m38s

NAME                                READY    STATUS    RESTARTS   AGE
pod/addwork-27237815-b44ws          0/1      Completed 0           2m38s
pod/pi-worker-6f6dfdb548-5czkc      1/1      Running   5           14m
pod/pi-worker-6f6dfdb548-gfkcq      1/1      Running   0           7s
pod/pi-worker-6f6dfdb548-pl584      1/1      Running   0           7s
pod/pi-worker-6f6dfdb548-qpgxp      1/1      Running   5           25m
pod/redis-0                          1/1      Running   0           14m
pod/redis-1                          1/1      Running   0           33m
pod/redis-2                          1/1      Running   0           32m
```

CronJob will spawn a new Job on a schedule (which in turn, spawns a new Pod). You can inspect these historic jobs, as they remain with the “Complete” status. The `successfulJobsHistoryLimit` and `failedJobsHistoryLimit` options in the `CronJobSpec`⁴ can be used to govern how many of those historic Jobs will be kept.

Timezones

Be aware that the cron job will run on the timezone of your cluster, which for many platforms like GKE will be UTC . The timezone used is that of the system Kubernetes controller component, which runs on the master. If you're on a managed platform, it may not be possible to query the master node directly, but it is possible to check nodes which likely are the same. Here's how to create a one-off Pod to run the Linux `date` command and then exit.

```
$ kubectl run date --restart=Never -it --rm --image ubuntu date +%Z
UTC
pod "date" deleted
```

There exists a prefix that can be added `CRON_TZ=<timezone>` (like `CRON_TZ=UTC 0 16 * * 1`), however as of Kubernetes 1.21 it doesn't form part of the official Kubernetes API contract, and thus isn't recommended.

⁴ <https://kubernetes.io/docs/reference/kubernetes-api/workload-resources/cron-job-v1/#CronJobSpec>

10.3 Batch task processing with Jobs

What if you have a batch set of work that you want to process as a regular, or one-off event? As covered in section 10.1, if a *continuously running* task queue is what you want, then actually Deployment is the right Kubernetes object. But, if you have a finite batch of work to process, then Job is the ideal object to use.

If you have a dynamic work queue database like we did in section 10.1, but want your workers to shutdown completely when the queue is empty, that's something that Job can do. With a Deployment, you need a separate system (like a HorizontalPodAutoscaler) to scale the worker Pods up and down, for example when there is no more work in the queue. When using Job, the worker Pods themselves can signal to the Job controller when the work is complete and they should be shutdown, and the resources reclaimed.

Another way to use Job is to run it on a static work queue in such a way where a database is not needed at all. Let's say you know you need to process 100 tasks in a queue, you could run the Job 100 times. The catch of course is that each Pod instantiation in the Job series needs to know which of those 100 tasks to run on, which is where the indexed job comes in.

In this section, I'll cover both the dynamic and static approach to task processing.

10.3.1 Dynamic Queue Processing with Jobs

Let's redesign the dynamic queue from section 10.1 to use Job instead of Deployment. Both Kubernetes objects allow the creation of multiple Pod workers, and both will recreate Pods in the event of failure. The key difference is that when a Pod managed by a Job exits with a "success" code (`exit 0` in Unix terms), it indicates to Kubernetes that the whole queue's work is complete, and no new Pod workers will be created (while the running ones can wrap up their work).

It's this property of Job that allows the individual workers to signal when the work is finished that is the useful property of Job. If you're using a dynamic Kubernetes environment such as one with autoscaling (including my own product GKE Autopilot), then Job allows you to "set and forget" the work, where you schedule it and once it's done the resource consumption goes to zero (which on a dynamic platform like Autopilot, means zero cost).

To work correctly in a Job, we need to adapt our task worker from the previous section to signal success (by terminating with the success status) when the queue becomes empty. Here's what our revised worker code looks like:

Listing 10.11 /pi_worker3/pi_worker.py

```

import os
import signal
import redis
from pi import *

redis_host = os.environ.get('REDIS_HOST')
assert redis_host != None
r = redis.Redis(host=redis_host, port= '6379', decode_responses=True)

running = True

def signal_handler(signum, frame):
    print("got signal")
    running = False

signal.signal(signal.SIGTERM, signal_handler)

print("starting")
while running:
    job = r.blpop('queue:task', 5)
    if task != None:
        iterations = int(task[1])
        print("got task: " + str(iterations))
        pi = leibniz_pi(iterations)
        print (pi)
    else: #A
        if os.getenv('COMPLETE_WHEN_EMPTY', '0') != '0': #A
            print ("no more work") #A
            running = False #A

```

#A When configured with COMPLETE_WHEN_EMPTY, exit with success **when there are no jobs**

With our worker container setup to behave correctly in the Job context, we can now create a Kubernetes Job to run it. Whereas in the deployment we use the `replica` field to govern the number of pods that are running at once, with Job it's the `parallelism` parameter, which basically governs the same thing.

Listing 10.12 JobWorker/job_worker.yaml

```

apiVersion: batch/v1
kind: Job
metadata:
  name: jobworker
spec:
  backoffLimit: 2
  parallelism: 2
  template:
    spec:
      containers:
      - name: pi
        image: docker.io/wdenniss/pi_worker:v3
        imagePullPolicy: Always
        env:
        - name: REDIS_HOST
          value: redis-0.redis-service
        - name: PYTHONUNBUFFERED
          value: "1"
        - name: COMPLETE_WHEN_EMPTY
          value: "1"
      restartPolicy: Never

```

If you compare this to the code listing in 10.1 for Chapter10/10.1.1_TaskQueue/deploy_worker.yaml, you'll notice that the PodSpec is identical other than the addition of the COMPLETE_WHEN_EMPTY flag.

Clean up

If you are running the previous samples, clean them by deleting the deployment and the CronJob if one was used.

```

$ kubectl delete -f 10.1.2_TaskQueue2
deployment.apps "pi-worker" deleted

```

```

$ kubectl delete -f 10.2.1_Job
job.batch "addwork" deleted

```

```

$ kubectl delete -f 10.2.2_CronJob
cronjob.batch "addwork" deleted

```

Since our Redis-based queue may have some existing jobs, you can reset it as well using the LTRIM Redis command.

```
kubectl exec -it pod/redis-0 -- redis-cli ltrim queue:task 0 0
```

You can also run the redis-cli interactively if you prefer

```

$ kubectl exec -it pod/redis-0 -- redis-cli
127.0.0.1:6379> LTRIM queue:task 0 0
OK

```

Let's take this Job for a spin. First, we can add some work to our Redis queue, like before.

```
$ kubectl create -f 10.2.1_Job
job.batch/addwork created
```

```
$ kubectl get job,pod
```

NAME	COMPLETIONS	DURATION	AGE
job.batch/addwork	0/1	19s	19s

NAME	READY	STATUS	RESTARTS	AGE
pod/addwork-19fgg	0/1	ContainerCreating	0	19s
pod/redis-0	1/1	Running	0	19h
pod/redis-1	1/1	Running	0	19h
pod/redis-2	1/1	Running	0	19h

Once this "addwork" Job has completed, we can run our new Job queue to process the work. Unlike previously, the order matters here since the Job workers will exit if there is no work in the queue, so make sure that "addwork" completed before you run the Job queue. Observe the status like so:

```
$ kubectl get job,pod
```

NAME	COMPLETIONS	DURATION	AGE
job.batch/addwork	1/1	22s	36s

NAME	READY	STATUS	RESTARTS	AGE
pod/addwork-19fgg	0/1	Completed	0	37s
pod/redis-0	1/1	Running	0	19h
pod/redis-1	1/1	Running	0	19h
pod/redis-2	1/1	Running	0	19h

Once we see "Completed" on our addwork task, we can go ahead and schedule the Job queue.

```
$ kubectl create -f 10.2.4_JobWorker
job.batch/jobworker created
```

```
$ kubectl get job,pod
```

NAME	COMPLETIONS	DURATION	AGE
job.batch/addwork	1/1	22s	3m45s
job.batch/jobworker	0/1 of 2	2m16s	2m16s

NAME	READY	STATUS	RESTARTS	AGE
pod/addwork-19fgg	0/1	Completed	0	3m45s
pod/jobworker-swb6k	1/1	Running	0	2m16s
pod/jobworker-tn6cd	1/1	Running	0	2m16s
pod/redis-0	1/1	Running	0	19h
pod/redis-1	1/1	Running	0	19h
pod/redis-2	1/1	Running	0	19h

What should happen next is that when the queue is empty, the workers will process 1 more task, then exit. If you want to monitor the queue depth, so as to know when the work should wrap up, you can run LLEN on the Redis queue to get the current length.

```
$ kubectl exec -it pod/redis-0 -- redis-cli llen queue:task
(integer) 0
```

When it's zero, you should observe the Pods entering the "Completed" state. Note that they won't enter this state right away, but rather after they wrap up the last task they are processing.

```
$ kubectl get job,pod
```

NAME	COMPLETIONS	DURATION	AGE
job.batch/addwork	1/1	22s	3m45s
job.batch/jobworker	0/1 of 2	2m16s	2m16s

NAME	READY	STATUS	RESTARTS	AGE
pod/addwork-19fgg	0/1	Completed	0	10m09s
pod/jobworker-swb6k	1/1	Completed	0	8m40s
pod/jobworker-tn6cd	1/1	Completed	0	8m40s
pod/redis-0	1/1	Running	0	19h
pod/redis-1	1/1	Running	0	19h
pod/redis-2	1/1	Running	0	19h

10.3.2 Static Queue Processing with Jobs

Instead of using a dynamic queue like Redis as we did in the previous section to store the task list, there are a number of ways to run Jobs with a static queue. When using a static queue, the queue length is known ahead of time and is configured as part of the Job itself, and a new Pod is created for each task. Instead of having task workers running until the queue is empty, you are defining upfront how many Pods to create.

The main reason for doing this is to avoid modification of the container to pull tasks from the dynamic queue. The drawback is that there is generally additional configuration on the Kubernetes side. It essentially shifts the configuration burden from the worker container to Kubernetes objects.

Note that even if you have the requirement that you can't modify the container which performs the work, this doesn't mean you have to use a static queue. You have multiple containers in a Pod, and have one container that performs the dequeuing, passing the parameters on to the other container.

So how do you represent this queue in Kubernetes configuration exactly? There are a few different options, three of which I'll outline here.

Static queue using an index

Indexed jobs are the most interesting static queue option in my opinion. Usable when you know ahead of time how many tasks to process, and the task list is one that is easily indexed. One example is rendering a movie. You know the number of frames (queue length) and can easily pass each instantiation the frame number (i.e. index into the queue) of the frame to render.

Kubernetes will run the Job the total number of times (completions) you specify, creating a Pod for each task. Each time it runs, it will give the Job the next index (supplied in the environment variable `$JOB_COMPLETION_INDEX`). If your work is naturally indexed, for example rendering frames in an animated movie, this works great! You can easily instruct Kubernetes to run the job 30,000 times (render 30,000 frames), and it will give each pod the frame number. Another obvious approach is to give each job the full list of work using some data

structure (e.g. an array of tasks encoded in YAML or just plain text, one per line), and Kubernetes supplies the index. The Job can then lookup the task in the list using the index.

At the time of writing, this option is still in beta, and so may or may not exist in the future in the form presented here.

Alpha and Beta Features

Alpha features in Kubernetes are ones in active development, and can be changed and removed at any time. Cloud providers generally don't offer them in production clusters. I will probably regret including this Alpha status workload pattern in the book, and you'll notice I've not included any other alpha objects—I just think this one is too interesting not to mention.

Alpha features can be dropped or changed without notice (i.e. they can exist in one version, but not the next). Beta features are only a little better, and can be removed after 3 releases (which can happen quicker than you realize—time goes fast!). Stay far, far away from Alpha features until they graduate, it's not worth the tears. For beta features, I know it's irresistible to use them when they can solve your use-case perfectly, and given how "young" Kubernetes still is, it may be unavoidable, just follow the releases careful and watch for changes or removal.

Here's an example configuration that simply outputs the frame number. You can sub in the actual movie rendering logic yourself.

Listing 10.13 IndexedJob/indexed_job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: echo
spec:
  completions: 5 #A
  parallelism: 3 #B
  completionMode: Indexed #C
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: 'worker'
          image: 'docker.io/library/busybox'
          command: ["echo", "render frame: $JOB_COMPLETION_INDEX"]
```

#A the number of times to run the job (upper bound of the index)

#B number of worker pods to run in parallel

#C Run in indexed mode, passing in JOB_COMPLETION_INDEX

Kubernetes supports the ability for you to essentially define your queue length and each Pod can get an index into that queue. For example, say you want to render a movie with 100,000 frames. You can set your "completions" (the number of times you want to run the job successfully) to 100,000. This will create (over time) 100,000 pods, and set the environment variable JOB_COMPLETION_INDEX in each pod with the index (0, 1, 2 and so on).

Your application can either use this environment variable directly, or you can use an init container to take the index and perform any configuration steps needed for the main container to perform the work, for example by building a script that will be run.

Static queue with a message queue service

Another approach that doesn't require modification is to populate a message queue, and have each Pod pull the work from that. As they can be configured to get the required parameters via Kubernetes configuration, it's possible to build a job where the container is unaware of the queue. It's still "static" since you have to declare upfront how many tasks there are, and run one worker pod per task, but it also requires a data structure (the message queue).

The Kubernetes docs do a great job of demonstrating this approach using RabbitMQ for the message queue here: <https://kubernetes.io/docs/tasks/job/coarse-parallel-processing-work-queue/>

Static queue via scripting

Another option is to use scripting to simply create a separate job for each task in the queue. Basically, if you have 100 tasks to complete, you'd setup a script to "expand" your task definition to create 100 Jobs giving each the input data it needs. This is personally my least favorite option as it's a bit unwieldy to manage. Imagine you queue all this work up and then want to cancel it? Instead of just deleting a single Job as in all the other examples in this section, you'd have to delete 100, so you'd likely need more scripting to do that, and on it goes.

Again, the Kubernetes docs have a good demo of this, so if it interests you, check it out here: <https://kubernetes.io/docs/tasks/job/parallel-processing-expansion/>

10.4 Liveness Probes for Background Tasks

Just like containers that serve HTTP traffic, containers that perform tasks, whether configured in a Deployment or Job should have liveness probes. A liveness probe helps protect you from coding errors like the inability to recover from the failure of an external dependency that causes your process to hang. The kubelet will automatically restart crashed containers (subject to how you configure your restart policy), but it has no way of knowing if your process has hung, or is performing as expected without a liveness probe.

You may recall from Chapter 4 (Zero Downtime Operations) that Kubernetes uses the information from Liveness and Readiness probes to make restart decisions on workloads. This is still true for background tasks, but with a few differences. Readiness as a concept isn't generally relevant for batch jobs as they don't serve traffic (so they can't therefore be ready, or not ready to receive traffic), but liveness still applies. We can use the same liveness probe concept that we used to detect stuck serving containers, to detect stuck tasks.

Whereas for a HTTP service we can use an endpoint that returns a success response for the liveness, and a TCP service where liveness can be indicated by successfully opening a port, batch workloads generally have neither, which leaves us with the third option for liveness probes: a command. You can provide any command, and if it exits with success, it indicates the task is live.

But what command can you run that would verify that the process has not hung? One approach is for the task to write the current timestamp to a file periodically, and simply check the recency of that timestamp using the liveness command.

LIVENESS BASH SCRIPT FOR BACKGROUND TASKS

In the application code, we have a function that writes the current Unix-format timestamp to a file. In this example, the code is Ruby – but the same pattern could be implemented in any language with ease.

```
def self.log_process_date
  fileName = "process.date"
  aFile = File.new(::Rails.root.to_s + "/log/" + fileName, "w")
  contents = Time.now.to_i.to_s
  aFile.write(contents)
  aFile.close
end
```

During normal operation of the background task, the `log_process_date` method will be called at various points. This should happen pretty frequently.

For the readiness command, we can run a bash script that can read and check this file. The goal of this script is simply to return the success exit code (0) when the timestamp file exists and is not stale. And exit with a non-zero code on any errors. This script is included in the application's container. It takes the timestamp file as an input parameter.

script/process_liveness.sh:

```
#!/bin/bash

# Liveness probe for batch process
# The process writes a logfile every time it runs with the current Unix timestamp.

# Usage: process_liveness.sh <path_to_file>
# The file must contain only the latest date as a Unix timestamp and no newlines

if ! rundate=$(<$1); then #A
  echo >&2 failed #A
  echo "no logfile" #A
  exit 1 #A
fi #A

curdate=$(date +%s) #B

diff=$((curdate-rundate)) #C

if [ $diff -gt 300 ] #D
then #D
  echo "too old" #D
  exit 100 #D
fi #D

exit 0 #E
```

#A Read the timestamp file from the input (parameter \$1), and exit with an error if it doesn't exist

#B Get the current timestamp from the system

#C Compare the two timestamp

#D Return an error status code if the process timestamp is older than 300 seconds
 #E Return a success status code

Finally, we write a liveness probe to call this script.

deploy.yaml

```
livenessProbe:
  initialDelaySeconds: 600 #A
  periodSeconds: 30
  exec: #B
    command: ["/script/process_liveness.sh", "log/process.date"] #B
  successThreshold: 1
  timeoutSeconds: 1
```

#A The initial delay is long, as we need time for the app to boot and write the first timestamp

#B The command to run

10.5 Summary

Kubernetes has a few different options for handling background queues and batch jobs.

- Deployments can be used to build a continuously running job queue, using a queue data structure like that offered by Redis for coordination.
- The background processing that many websites run to offload heavy requests would typically be run as a Deployment
- Kubernetes also has a dedicated Job object for running tasks
- Jobs can be used for one-off tasks, such as a manual maintenance task
- CronJob can be used to schedule Jobs to run, for example a daily cleanup task
- The Job object also supports batch jobs through the completions and parallel configuration
- Unlike a Deployment-based background queue, Job can be used to schedule work on a static queue, avoiding the need for a queue data structure
- Liveness checks are still relevant for Pods that process background tasks (to detect stuck/hung processes), and can be configured using an exec liveness check

11

GitOps: Configuration as Code

This chapter covers:

- Using namespaces and configuration to replicate environments
- Benefits of treating Kubernetes deployment configuration like source code
- Using `git pull` (merge) requests to drive operations
- Handling secrets without storing them in plain text in version control

You may have noticed in this book so far that we've been writing a lot of YAML configuration. It is possible to interact with most Kubernetes objects without writing configuration files (using imperative `kubectl` commands), and these are arguably easier to learn. So why did I exclusively use the declarative configuration-based approach? One reason is so that now as we take the app to production, we can treat our configuration like we do our code.

Another reason is it allows us to easily spin up multiple environments with the same configuration. Let's say you want a staging and a production environment that are close to identical. With your deployment represented in configuration files it's possible to replicate the environments easily. Kubernetes has a namespaces feature which makes this possible without needing to worry about name collisions.

11.1 Production and Staging Environments using Namespaces

As you prepare your application for production, you'll likely want to create a staging environment where changes can be tested before the live production application is updated. Kubernetes makes this easy with namespaces.

Namespaces as their name suggests provide name uniqueness within a single logical space. So you can setup a "production" namespace and a "staging" namespace, and have the deployment object "foo-deployment" and "foo-service" in each of them. This avoids the need to excessively modify your configuration for the different environments, like creating "foo-staging-deployment" and "foo-staging-service", and provides some protection against

accidental changes as by default, `kubectl` commands only apply to the namespace that's currently active.

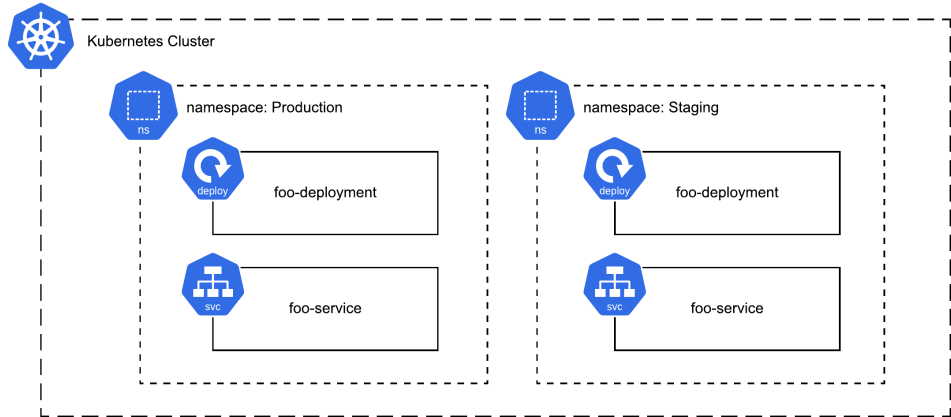


Figure 11.1: A Kubernetes cluster with two namespaces. Note that the deployment and service objects in these namespaces have the same name (and potentially, same configuration).

The main configuration differences between your production and staging are typically things like the scale (number of replicas), and any external service credentials as these external services are not included in the namespace.

To create a namespace you can run:

```
# Create the namespace "staging"
kubectl create namespace staging
```

To interact with this namespace, you can either add `--namespace staging` (or `-n staging` for short) to every `kubectl` command you run, or change the `kubectl` context so that all commands will run in this new namespace. I highly recommend the latter, as you don't want to forget the `-n` flag and accidentally run a command in the wrong namespace. Better to switch contexts each time.

```
# List the available namespaces
kubectl get namespace

# Set the context to the staging namespace
kubectl config set-context --current --namespace=staging
```

You may notice with the first command, that Kubernetes has a few namespaces out of the box. `kube-system` is where the system pods go, best not to touch this one unless you know what you're doing. `default` is the default user namespace, but I recommend creating your own dedicated ones, one for each application environment.

I find the context setting command above tedious, and highly recommend setting up your shell with a utility to make it simpler to switch. The one I use is `kubectx` + `kubens`, written by my colleague Ahmet Alp Balkan, available at <https://github.com/ahmetb/kubectx>. With `kubens` installed, you can instead run:

```
# List namespaces
kubens

# Switch to the staging namespace
kubens staging
```

The other included utility, `kubectx`, can be used to quickly switch between different clusters entirely. These scripts are just shorthand for the longer `kubectl config set-context` commands, so you can go back to using `kubectl` as normal once the context is set.

11.1.1 Deploying to our new namespace

Once you have the namespace created, you can deploy your application easily from configuration. This is why this book has been using configuration files in every case. Instead re-running a bunch of imperative commands to recreate your deployment, you can simply run the following command from the folder with your configuration:

```
# Deploy/update all configuration in this folder
kubectl apply -f .
```

And if you make any changes (or create another namespaced environment), you can just re-run that command to roll out your changes.

In fact, creating new environments with namespaces in Kubernetes is so trivial to configure, that if you were sharing a single staging environment in the past with other platforms, you may see some benefit to having a lot of different environments (e.g., one per developer, one for staging, another for integration testing, etc). Generally, the namespaces are free, although the compute resources used by your pods are not.

11.1.2 Syncing Mutations from the Cluster

But what about any changes that were made imperatively, outside of configuration? Perhaps you scaled a deployment with `kubectl scale`, or changed the image with `kubectl set-image`, or created a deployment with `kubectl run`. It happens, I won't judge.

Kubernetes lets you view, and export configuration with the `--output` parameter (`-o` for short) on any `get` request.

For example, to get the latest YAML configuration for a deployment:

```
# View the deployment as YAML
kubectl get deploy your-deployment -o yaml

# Pipe the deployment YAML config to a file
kubectl get deploy your-deployment -o yaml > your-deployment.yaml
```

The catch is that Kubernetes adds a lot of extra fields that you don't really want in your on-disk configuration, like status messages, etc. There used to be a handy `--export` option

which would strip these, but sadly it's deprecated (depending on when you read this, it may still work). So it's a bit of an art to figure out which lines you can delete, and which you need to keep. But you can compare the YAML files you get in this way to the ones in this book to see which lines are important.

If you plan to use the configuration in multiple namespaces (which is common), you will definitely want to delete the `metadata -> namespace` field. Removing this will allow you to deploy the configuration in the current namespace (while keeping it will mean any changes will update the object in whatever namespace was specified). The danger I see in keeping the namespace is you might accidentally have some configuration in your staging folder set to the production namespace. Section 7.3 below discusses some tactics on safety around rollouts to different namespaces, but it relies on *not* specifying the namespace in resource objects directly.

Other fields to consider for removal for cleanliness are from the `metadata` section, the fields `uid`, `resourceVersion`, `selfLink`, `creationTimestamp`, and the entire `status` section. These fields won't prevent you from reusing the configuration in other namespaces or clusters, but don't really have meaning outside their deployed context, so best keep it out of version control to avoid confusion.

11.2 Configuration as Code the Kubernetes Way

When you have a bug in your code, you may inspect the version history to see when the code was changed and might even roll back a commit to get back to the previously working state. When you treat your configuration as code (by committing it to your version control system), you can perform similar actions, but with your production systems.

If you have a peer review process for code that's submitted, the same process should apply to configuration. After all, the configuration impacts the running system just as much as the code does. Peer review on configuration repositories can help to catch errors before they are rolled out. If you fat-finger the wrong number of replicas, hopefully your colleague might notice.

You'll find this pattern used at all major internet companies. Most Google services for example are developed and deployed out of a single code repository¹, so the service configuration sits right beside the code. The exact same code review practices are followed for code, and for services, although the list of owners (the engineers can approve the changes for merging) will differ.

There's no obligation to store the configuration in the same repository as the code like Google though, this is mostly a matter of taste (and endless technical debate). The model I'll present here for storing Kubernetes configuration in Git is just an example which I've found works for me, but you should adapt it to your own engineering practices.

I use a single Git repository to represent all the Kubernetes objects deployed in a single cluster. In this repo is a folder for each Kubernetes namespace, and in those folders are the YAML files for the objects in the namespace. An alternative is to use a separate branch for each namespace, which has some nice properties like being able to merge changes as they go from staging to production, but as there are likely some changes you *don't* want to merge, it

¹ <https://research.google/pubs/pub45424/>

can get messy (you wouldn't want to accidentally merge your staging-only changes to production).

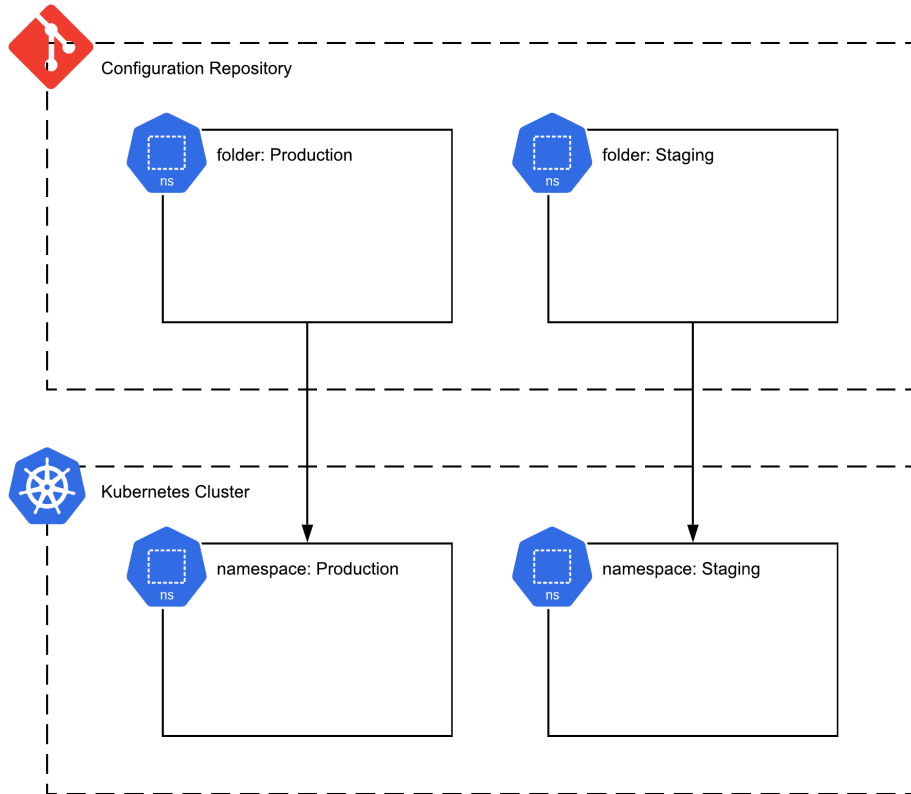


Figure 11.2: Git repository folder structure and the relationship to Kubernetes namespaces

Here's an example directory layout:

```
/_debug #A
/_cluster #B
/staging #C
/production #C
```

#A A directory for any debug scripts you wish to store for all developers

#B The cluster configuration, for example the namespace configuration files. These are only used during cluster creation

#C The environment configuration folders

Each directory in this repository is mapped to a Kubernetes namespace. The beauty of such a 1:1 mapping is that it allows you to confidently execute a `kubectl apply -f .` command to rollout all changes in the directory to the active namespace. Cloning the environment is as simple as duplicating the entire folder (and then deploying it to its own namespace).

It's common—particularly for smaller scale deployments—to share a cluster with multiple environment namespaces for cost reasons. This avoids extra charges for the control plane, and allows workloads to pool compute resources on the nodes. As the deployments get larger, it may be desirable to separate environments into their own clusters to provide an extra level of access control and resource isolation. The good news is that the configuration repository doesn't care where these namespaces are, it's totally fine for them to exist in different clusters.

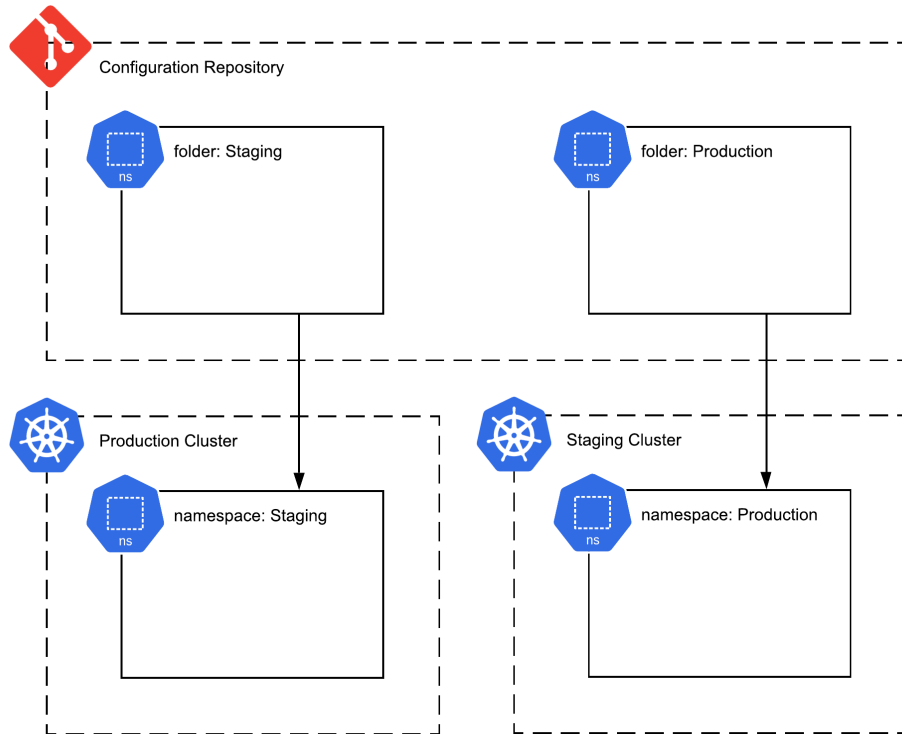


Figure 11.3: configuration repository with environments in multiple clusters

Now that your resource repository is setup, the work process looks like this: make changes to the deployed environment to your configuration, commit those changes, then update the live state by setting the current namespace with `kubectl`, then running `kubectl apply -f .` on the matching directory.

With this, you're following a "Configuration as Code" pattern, but there is more you can do. One danger with the setup as described so far is that you can accidentally rollout the configuration from one folder to the wrong namespace. The next sections cover how to roll out safely and avoid this problem, and how to up level to a full "GitOps" style process where the `git push` on the configuration repository triggers the rollout automatically.

11.3 Rolling Out Safely

With your configuration as code repository setup, there is now a question of how best to roll out the changes in the repo.

Sure, you can simply checkout the repository and run `"kubectl apply -f ."` as we did earlier, but this can be dangerous. You could accidentally deploy the wrong configuration into the wrong namespace. Since we're reusing object names in multiple environments, this could be quite bad indeed. Also, there's nothing to stop you running any other commands that don't require the configuration to be committed to the repository.

To tackle the "wrong namespace" issue, what I recommend is to have some guardrails in place to avoid accidentally deploying the wrong configuration to the wrong namespace. Instead of simply running `kubectl apply -f .` as we did earlier, wrap it up in a script that performs a check to ensure you're deploying into the right namespace. If we name our folders the same as the namespace, then the check is simple: if current namespace is equal to the folder name, deploy.

Here's an example script that compares the current directory name to the current namespace, and exits with an error status if they mismatch.

gitops_check.sh

```
#!/bin/bash

CURRENT_DIR=`echo "${PWD##*/}"`
CURRENT_NAMESPACE=`kubectl config view --minify -
o=jsonpath='{.contexts[0].context.namespace}'`

if [ "$CURRENT_DIR" != "$CURRENT_NAMESPACE" ]; then
    >&2 echo "Wrong namespace (currently $CURRENT_NAMESPACE but $CURRENT_DIR expected)"
    exit 1
fi

exit 0
```

You can then use this in any other scripts, like the following rollout script:

rollout.sh

```
#!/bin/sh

if [ $(../gitops_check.sh; echo $?) != 0 ]; then exit 1; fi

kubectl apply -f .
```

A full "git ops" folder structure including these scripts is provided in the samples in `Chapter11/gitops`

This isn't the only option of course. Another approach would be to set the desired namespace in your rollout script and then deploy (just be sure if the "set namespace" step fails, the whole thing will bail out).

For these scripts to work though, you'll need to ensure that none of your configuration files specify a `metadata -> namespace` field directly. If they have a namespace set, it will ignore the current context, so the above script won't prevent updates in that case.

To really do GitOps properly though, you'll want to solve the other issue and that is to guarantee that the configuration deployed is what is in the repository. The best way to solve that is to remove the human from the loop completely, and configure a *deployment pipeline*, or a *gitops operator*.

11.3.1 Deployment Pipelines

A deployment pipeline is simply a set of functions that run based on a code repository trigger. For example: "when a code is pushed to the configuration repository, deploy the configuration to a Kubernetes cluster".

Using pipelines guarantees that the configuration being deployed matches what was committed. If the operator needs to make additional changes after the deployment (for example to correct an error), they make them in the configuration code repository like normal, push the code and trigger the pipeline rollout once again.

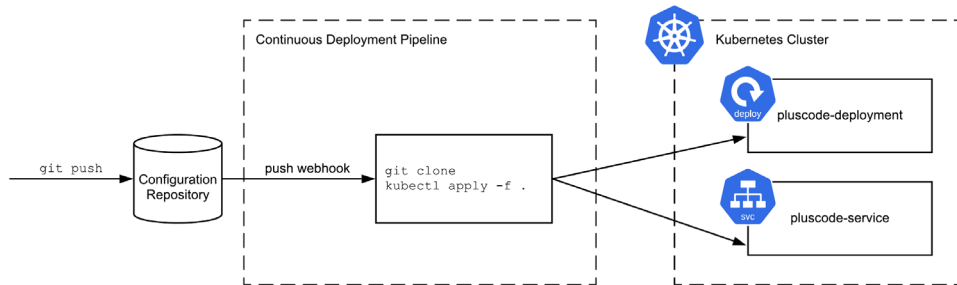


Figure 11.4: A continuous deployment pipeline to Kubernetes

Now that your pipeline is configured, you can push to producing by merging a code on your git repo, i.e. git-driven operations, or "GitOps". The key is to not make any changes on the cluster directly; all changes go through the configuration repository and the continuous deployment pipeline.

CONTINUOUS DEPLOYMENT WITH CLOUD BUILD

To implement such a deployment pipeline in practice, there are many products on the market. For GKE users, one option is Cloud Build. You can set up a trigger so that when your configuration repository is pushed, it will run `"kubectl apply -f ."`.

Here's how to set it up: 1) configure IAM permissions² for the Cloud Build service account (to give it permission to act on your GKE cluster), 2) create a new trigger (set to fire when your configuration repository is pushed), and 3) add a Cloud Build configuration file to your repository, and reference it in the trigger.

cloudbuild-deploy.yaml

```
steps:
- name: 'gcr.io/cloud-builders/kubectl'
  id: Deploy
  args:
  - 'apply'
  - '-f'
  - '$FOLDER'
  env:
  - 'CLOUDSDK_COMPUTE_ZONE=us-west1-a'
  - 'CLOUDSDK_CONTAINER_CLUSTER=my-cluster'
```

This is just scratching the service of continuous delivery, if you're using Cloud Build, then you could consult the excellent guide *GitOps-style continuous delivery with Cloud Build*³, which goes further and sets up a complete end to end CI/CD flow.

CONTINUOUS RECONCILIATION

The method described here can be further improved by using a GitOps operator. This is a code loop that runs in the cluster and constantly reconciles what is running in the cluster to what is present in the configuration repository. The end result is similar to the event-driven pipeline described above, with the advantage that it can perform additional reconciliation if there is ever a divergence, while the pipeline relies on the push event to trigger. Flux⁴ is one such GitOps operator.

11.4 Secrets

A git repo is a great place to store your Kubernetes configuration, but there is some data that probably shouldn't be stored there: secret values like database passwords and API Keys. If such secrets are embedded in the code itself, or in environment variables, it means that anyone with access to your source code will have the secret. An improvement would be that only those who can access your production system would have access to this data. You can go further of course, but in the context of this chapter on GitOps, I'll focus on how to separate your secrets from your code and config repositories.

Kubernetes actually has an object for storing secrets, aptly named Secrets. These objects are a way to provide information such as credentials and keys to workloads, in a way that separates them from the configuration of the workload itself.

² <https://cloud.google.com/build/docs/securing-builds/configure-access-for-cloud-build-service-account>

³ <https://cloud.google.com/kubernetes-engine/docs/tutorials/gitops-cloud-build>

⁴ <https://fluxcd.io/>

11.4.1 String-based (Password) Secrets

If you've been embedding secrets like passwords in plain environment variables in the deployment configuration, now would be a good time to migrate them to Secrets. Let's say we have a secret with a value of `secret_value` (in reality, this might be a key obtained from your cloud provider).

We can encapsulate our `secret_value` into a Kubernetes Secret object like so:

Listing 11.1 StringSecrets/secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: secrets-production
type: Opaque
stringData:
  SECRET_KEY: secret_value
  ANOTHER_KEY: another_secret_value
```

Secrets can be provided to Pods in two ways: as a file, and as an environment variable. You would use the file method for secret data that your application will access as a file (such as a private SSL key), and the environment variable for items like database passwords. Since our secret above is a simple string, we'll use the environment variable method here:

Listing 11.2 StringSecrets/deploy.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 1
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:4
          env:
            - name: AVATAR_ENDPOINT #A
              value: http://robohash-internal #A
            - name: SECRET_KEY #B
              valueFrom: #B
                secretKeyRef: #B
                  name: secrets-production #B
                  key: SECRET_KEY #B
```

#A A normal (non-secret) environment variable

#B An environment variable populated with the value of the secret

To verify that everything worked correctly, create the deployment and secret. Run `kubectl get pods`, grab one of the pod names, and run:

```
$ kubectl exec POD_NAME exec -- env
SECRET_KEY=secret_value
```

You should see the secret in the list. Our application now has access to our `SECRET_KEY` environment variable.

11.4.2 Base64 Encoded Secrets

At the time of writing, the Kubernetes docs and many other materials demonstrate secrets where the value is base64 encoded (using the `data` key, rather than `stringData`). This isn't done for security (base64 is an encoding, not encryption), but rather so that you can represent data that wouldn't be compatible YAML.

I find that this mostly serves to obfuscate the data without adding much value. However, if you have a string that you can't represent in YAML, or are storing a file, then you would want to base64 encode the data. Here's an equivalent representation of the secret shown earlier:

Listing 11.3 Base64Secrets/secret-base64.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: secrets-production
type: Opaque
data:
  SECRET_KEY: c2VjcmV0X3ZhbHVlCg==
```

To do the encoding, on any *nix system you can do the following:

```
$ echo "secret_value" | base64
c2VjcmV0X3ZhbHVlCg==

$ echo "c2VjcmV0X3ZhbHVlCg==" | base64 -D
secret_value
```

You can include both `data` and `stringData` in the same configuration file, if you have some values that need base64 encoding, and others that don't. You can also store multiple secrets in each Kubernetes Secret object (one per line). Here's an example that defines 3 secrets, 2 using plain text, and a third using base64.

Listing 11.4 Base64Secrets/secrets-multiple.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: secrets-production
type: Opaque
stringData:
  SECRET_KEY: secret_value
  ANOTHER_KEY: another_value
data:
  ENCODED_KEY: VGhpcyBzdHJpbmckbWlnaHQgYmUgaGFyZCB0byByZXByZXNlbnQgaW4gWUFNTCDwn5iFCg==
```

If you are retrieving secrets from the server, those are stored in base64, so you will need to decode to get the plain-text values.

I personally have one secret object for each of my namespaces, e.g. `myapp-secrets-production`, `myapp-secrets-staging`, each with multiple secrets, but I store them in a separate repo to the rest of my config (in Section 8.4.4 below, I'll discuss some options for how to store secrets apart from your main configuration repository, while still using a Gitops approach).

11.4.3 File-based Secrets

Sometimes you'll be dealing with secrets that you want to access from your application as files, rather than strings from environment variables.

Kubernetes has you covered here as well. Creating the secret is actually the same, but I'll provide a fresh example of a multi-line text file, since how such data is represented in YAML has some nuance.

Say we have a private key to store. Here's one I generated using `openssl genrsa -out example.key 256` (normally you'd use a 2048 bit key or higher, but for brevity, I'll use 256).

```
-----BEGIN RSA PRIVATE KEY-----
MIGsAgEAAiEA4TneQFg/UMsVGrAvsm1wkonC/5jX+ykJAMeNffn1PQkCAwEAAQIh
ANGcs+MgClkXFQAP0SSvmJRmnRze3+zgUbN+u+rrYNR1AhEA+K0ghKRgK1zVn0xw
q1tgTwIRAOfb8LCVNf6FAdD+bJGwHycCED6Yzf01s0NZBQiAWAf6Am8CEQDIEXI8
fVSNHmp108UNZcNLAhEA3hHFV5jZppEHHHLy4F9Dnw==
-----END RSA PRIVATE KEY-----
```

This can be represented in YAML, the following way. Note the all-important pipe character which will preserve the line endings (the config itself cannot have line breaks, this representation preserves the linebreaks in the value itself).

Listing 11.5 FileSecrets/secret_file.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-files
type: Opaque
stringData:
  example.key: |
    -----BEGIN RSA PRIVATE KEY-----
    MIGsAgEAAiEA4TneQFg/UMsVGrAvsm1wkonC/5jX+ykJAMeNffn1PQkCAwEAAQIh
    ANGcs+MgClkXFQAP0SSvmJRmnRze3+zgUbN+u+rrYNR1AhEA+K0ghKRgK1zVn0xw
    q1tgTwIRAOfb8LCVNf6FAdD+bJGwHycCED6Yzf01s0NZBQiAWAf6Am8CEQDIEXI8
    fVSNHmp108UNZcNLAhEA3hHFV5jZppEHHHLy4F9Dnw==
    -----END RSA PRIVATE KEY-----
```

If you're tired of wrestling with YAML syntax by now, you can base64 encode the file data instead using `cat example.key | base64`, and represent it like so (with the data truncated for readability). Note that the entire base64 string is placed on one line (no line breaks!).

Listing 11.6 FileSecrets/secret_file_base64.yaml

```

apiVersion: v1
kind: Secret
metadata:
  name: secret-files
type: Opaque
data:
  example.key: LS0tLS1CRUdJTiBSU0EgU...SBLRVktLS0tLQo=

```

It's a bit tedious creating these configuration files for secrets by hand. A more automated approach is to use `kubectl` to create the files for you. The following command will create the same functional output (note that the base64 string is truncated for readability).

```

$ kubectl create secret generic secret-files --from-file=example.key=./example.key --dry-run=client -o yaml

```

```

apiVersion: v1
data:
  example.key: LS0tLS1CRUdJTiBSU0EgU...SBLRVktLS0tLQo=
kind: Secret
metadata:
  creationTimestamp: null
  name: secret-files

```

The `--dry-run=client -o yaml` part means that you won't actually create the secret on the server, and instead will output it as YAML (for you to place in a configuration file, to be later applied to the server with `kubectl apply -f filename.yaml`). Omitting the `dry-run`, would create the secret directly on the cluster (the imperative style of creating Kubernetes objects). In fact, every example given in this section could have been written as an imperative `kubectl` command, but as I hopefully convinced you in section 7.2, there are durable benefits to a declarative, configuration-driven approach to operating your cluster.

Once created, you can mount all the secrets in the Secret into a folder of your choice. The below example mounts our `secret-files` secret into the folder `/etc/config`. Each of the data keys is mounted as its own file. In our case, there was only one: `example.key`.

Listing 11.7 FileSecrets/deploy.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 1
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:4
          volumeMounts:
            - name: secret-volume
              mountPath: "/etc/config"
              readOnly: true
      volumes:
        - name: secret-volume
          secret:
            secretName: secret-files

```

To verify that everything worked correctly, create the deployment and secret. Run `kubectl get pods`, grab one of the pod names, and use `exec` to list the directory. You should see our file "example.key":

```

$ kubectl exec POD_NAME -- ls /etc/config
example.key

```

Now you can point your code to this file, just like any other file on the system.

11.4.4 Secrets and GitOps

Using Secrets is only one part of the equation. Now you'll need to figure out how to store them. If you place them in the same configuration repository, you may as well have just used plain environment variables and skipped the step in the previous section.

There's no silver bullet to this problem, but here are a few ideas, presented in increasing order of complexity.

SEPARATE REPOSITORY

A simple option is to have a separate configuration repository for your secrets with fewer users granted access than your regular repos. You still have all the benefits of configuration as code (code reviews, rollback, etc), but can limit the audience. If you operate a repository with granular access control, you could place the secrets in an access-controlled folder of that repo.

One logical place for this repository to be located would be together with your production resources at your cloud provider, with the same access control as your production environment. Since anyone with access to your production environment has the secrets

anyway, this model doesn't provide any additional access if someone compromises the account.

SEALED SECRETS

The Sealed Secrets⁵ project has an interesting approach; it encodes all of your secrets with a master secret. While you still end up with the same problem of where to store the master secret, it means that the encoded secrets can be included in the main configuration repository with all the benefits that this entails, like rollback.

SECRETS SERVICE

Another option is to run a separate service that can inject secrets into your cluster. Vault by HashiCorp⁶ is a very popular implementation of this concept, and is available as open source if you wish to run it yourself.

11.5 Summary

Aside from being able to express a wide range of deployment constructs, one of the big advantages of Kubernetes is the declarative design, how you can *declare* what you want in configuration files, and have Kubernetes take care of the rest. To take full advantage of this ability:

- Use Namespaces to separate different environments like production and staging, and different applications
- Always update your configuration first, then apply it to the cluster—don't operate directly on the cluster
- Store this configuration in a code repository, and treat it like code (with peer reviews if you do them, etc)
- Optionally, configure a continuous deployment pipeline driven by git pushes of the config repo (so-called "GitOps").
- Get secrets out of your application config, and into Kubernetes Secret objects. Store them in a way that limits access.

⁵ <https://github.com/bitnami-labs/sealed-secrets>

⁶ <https://www.vaultproject.io/>

12

Securing Kubernetes

This chapter covers

- Keeping your cluster up to date and patched against vulnerabilities
- Managing update-related disruptions with a PodDisruptionBudget
- Using DaemonSets to deploy an agent to every node
- The security configuration of Pods
- Building and running containers as the non-root user
- Using Admission controllers to modify and/or validate Kubernetes objects
- Using the built-in Pod Security Admission to enforce security standards in namespaces
- Controlling user access to namespaces with RBAC

So far, the book has focused on deploying various different types of software into Kubernetes clusters. In this last chapter, I'll cover some key topics when it comes to keeping everything secure. Security is a huge area in general, and Kubernetes is no exception. If you deploy code to a Kubernetes cluster managed by another team, then lucky you, you may not need to worry about some of these topics. For developers who are also responsible for operations, or for cluster operators themselves, securing and updating the cluster is a key responsibility.

In addition to keeping your cluster up to date, handling disruption, deploying node agents and building non-root containers, this chapter walks you through the process of creating a dedicated namespace for a team of developers, and how access can be granted specifically to that namespace. This is a pretty common pattern I've observed in companies where several teams share clusters.

12.1 Staying up to date

Kubernetes has a large surface area. There's the Linux kernel, the Kubernetes software running on the control plane and user nodes, then there's your own containers and all their

dependencies like a base image. All this means there's a lot to keep up to date and protected against vulnerabilities.

12.1.1 Cluster and Node Updates

One critical task for a Kubernetes operator is to ensure that your cluster and nodes are up to date. This helps mitigate against known vulnerabilities in Kubernetes, and the operating system of your nodes.

Unlike most of the topics discussed in this book so far, the updating of clusters and nodes is actually not part of the Kubernetes API. It sits at the platform level, so you'll need to consult the docs for your Kubernetes platform. Fortunately, if you're using a managed platform this should be straight forward. If you're running Kubernetes the hard way via a manual installation on VMs (which I don't recommend), these updates will be a significant burden, as you are now the one offering the Kubernetes platform.

UPDATING GKE

In the case of GKE, staying up to date is easy. Simply enroll in one of the three release channels: Stable, Regular or Rapid. Security patches are rolled out to all channels quickly, what differs is how soon you get other new features of both Kubernetes and the GKE platform.

When enrolled in a release channel, both the cluster version and nodes are automatically kept up to date. The older "static version" (available only for the Standard mode of operation) option is not recommended, as you need to keep on top of the updates manually.

12.1.2 Updating Containers

Keeping the Kubernetes cluster up to date isn't the only updating you'll need to do. Security vulnerabilities are often found in components of base images like ubuntu. As your containerized application is built on these base images, it can inherit vulnerabilities that exist in them.

The solution is to rebuild and update your containers regularly (and especially, if any vulnerabilities are found in the base images you use). Many developers and enterprises employ vulnerability scanners (often known as "CVE scanners" after the Common Vulnerabilities and Exposures system where known vulnerabilities are documented) to look through built containers to see if any reported vulnerabilities exist in them, in order to prioritize rebuilds and rollouts.

When updating your containers, be sure to specify the base image which contains the latest fixes. Typically, this can be achieved by only specifying the minor version of the base image you're using, rather than the specific patch version. You can use the "latest" tag to achieve this, but then you might get some unwanted feature changes.

For example, take the python base image⁴. For any given version of python (say 3.10.2), you have a bunch of different options: 3.10.2-bullseye, 3.10-bullseye, 3-bullseye, bullseye (bullseye refers to the version of Debian it uses). You can also use latest. For

⁴ https://hub.docker.com/_/python

projects that follow semantic versioning (semver) principles, I would typically recommend going with the `major.minor` version, in this example `3.10-bullseye`. This allows you to get patches to the 3.10 version automatically, while avoiding breaking changes. The downside is that you need to pay attention to when the support drops for 3.10, and migrate. Going with the major version instead, i.e. `3-bullseye` in this example, would give you longer support but with slightly more risk of breakages (in *theory* with semver you should be safe to use the major version as changes should be backwards compatible, but in practice I find it safer to go with the minor version). Using `latest` while great from a security perspective, is typically not recommended due to the extremely high risk of breakage.

However you configure your docker file, the key principle is to rebuild often, reference base images that are up to date and patched, rollout updates to your workloads frequently, and employ CVE scanning to look for containers that are out of date.

REDUCING CONTAINER UPDATES

A further mitigation to reduce potential vulnerabilities in application containers is to build extremely lightweight containers that contain only the absolute minimum needed to run your application, being your application and its dependencies. Using a typical base image like ubuntu includes a package manager, and various software packages which make life easy, but also increase the attack surface area. The less code there is in your container from other sources, the less you'll need to update it due to vulnerabilities found in that code, and the less bugs you can potentially be exposed to.

The code sample in section 2.1.8 earlier in the book employed this principle by using one container to build your code, and another to run the code. To reduce the potential attack surface, the key is to pick the slimmest possible runtime base image for the second stage of the container build. Google has an open source project [distroless](https://github.com/GoogleContainerTools/distroless)² to assist with providing super-lightweight runtime containers.

Here is the distroless project's example of a building Java container, referencing the Google-provided distroless image in the second step:

Listing 12.1

<https://github.com/GoogleContainerTools/distroless/tree/main/examples/java/Dockerfile>

```
FROM openjdk:11-jdk-slim-bullseye AS build-env
COPY . /app/examples
WORKDIR /app
RUN javac examples/*.java
RUN jar cfe main.jar examples.HelloJava examples/*.class

FROM gcr.io/distroless/java11-debian11
COPY --from=build-env /app /app
WORKDIR /app
CMD ["main.jar"]
```

² <https://github.com/GoogleContainerTools/distroless>

12.1.3 Handling Disruptions

With all this updating, you might be wondering: what happens to my running workloads!? It's inevitable that as you update Pods will be deleted and recreated. This can obviously be very disruptive to the workloads running in those Pods, but fortunately Kubernetes has a number of ways to reduce this disruption and potentially eliminate any ill-effects.

READINESS CHECKS

Firstly, if you've not setup Readiness checks, now is the time to go back and do that as it's absolutely critical. Kubernetes relies on your container reporting when it's ready, and if you don't do that it will assume it's ready the moment the process starts running which is likely *before* your application has finished initializing and is actually ready to serve production traffic. Chapter 4 covers this topic. The more your Pods are moved around such as during updates, the more requests will error out hitting Pods that are not ready, unless you implement proper readiness checks.

SIGNAL HANDLING AND GRACEFUL TERMINATION

Just as Readiness checks are used to determine when your application is ready to start, graceful termination is used by Kubernetes to know when your application is ready to stop. In the case of a Job which may have a process that takes a while to complete, you may not want to simply terminate that process if it can be avoided. Even web applications with short-lived requests can suffer from abrupt termination that causes requests to fail.

To prevent these issues, it's important to handle SIGTERM events in your application code to start the shutdown process, and set a graceful termination window (configured with `terminationGracePeriodSeconds`) long enough to complete the termination. Web applications should handle SIGTERM to shutdown the server once all current requests are completed, and batch jobs would ideally wrap up any work they are doing, and not start any new tasks.

In some cases, you may have a Job performing a long running task that if interrupted would lose its progress. In these cases, you may set a very long graceful termination window, accept the SIGTERM but simply continue to attempt to finish the current task. Managed platforms may have a limit on how long the graceful termination window can be for system-originated disruption.

Section 10.1.2 in the previous chapter has examples of SIGTERM handling and `terminationGracePeriodSeconds` configuration in the context of Jobs. The same principles apply to other workload types.

ROLLING UPDATES

When you update the containers in a Deployment or a StatefulSet (for example to update the base image), the rollout is governed by your rollout strategy. Rolling update, covered in Chapter 4, is a recommended strategy here to minimize disruption when updating workloads by updating Pods in batches, while keeping the application available. For Deployments, be sure to configure your `maxSurge` parameters of the Deployment which will do a rollout by temporarily increasing the pod replica count (safer for availability than reducing it).

POD DISRUPTION BUDGETS

When nodes are updated, this process does *not* go through the same Deployment rollout process. Here's how it works: firstly, the node is cordoned to prevent new Pods being deployed on it. Then the node is drained, whereby Pods are deleted from this node, and recreated on another node. By default, Kubernetes will delete all Pods at once from the node and schedule them to be created elsewhere. Note that it does *not* first schedule them to be created elsewhere, then delete them (although this is a feature I think we should add to Kubernetes). If multiple replicas of a single Deployment are running on the same node, this can cause unavailability.

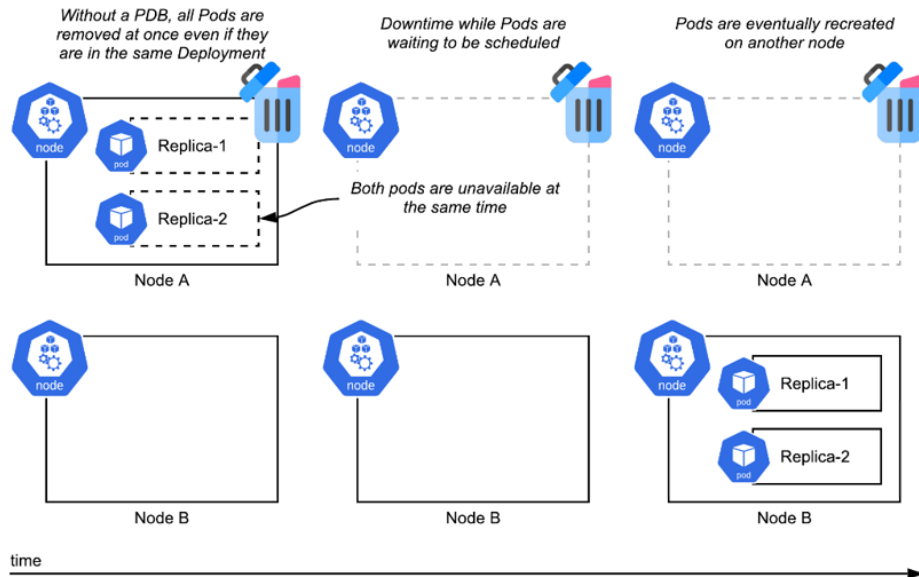


Figure 12.1: node deletion without pod disruption budgets. All the Pods on the node will become unavailable at once.

To solve the problem where draining a node may reduce the availability (meaning running replicas) of your deployments, should multiple pods from the same deployment be on that node, Kubernetes has a feature called Pod Disruption Budgets (PDBs). PDBs allow you to inform Kubernetes how many pods, or what percentage of your pods you are willing to have unavailable, for your workload to still function as you designed it.

Listing 12.2 PDB/pdb.yaml

```

apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: timeserver-pdb
spec:
  maxUnavailable: 1
  selector:
    matchLabels:
      pod: timeserver-pod

```

Deploying this into your cluster will ensure that at no time during disruptions will more than 1 of your Pods be unavailable. An alternative configuration uses `minAvailable` to set how many replicas you need. I prefer `maxUnavailable`, as it works better with scaling (if you use `minAvailable`, you may need to scale that value along with your replica count to retain the desired minimum availability, which is just extra work).

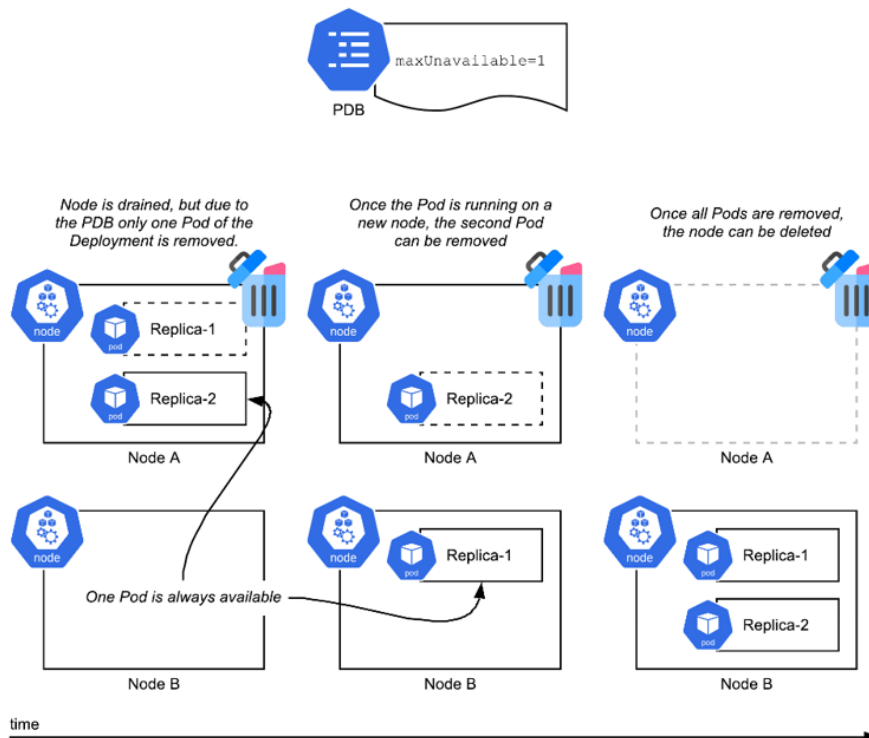


Figure 12.2: with a PDB, Kubernetes will wait for the required number of pods in a Deployment to be available before deleting others, reducing the disruption

The process of handling disruptions with a PDB is somewhat similar to how a rolling update avoids taking out too many pods at the same time. To ensure your application stays available during deployment updates that you initiate, and disruptions initiated by cluster updates, you'll need to have both the rolling update, and the PDB configured.

12.2 Deploying Node Agents with DaemonSet

This book has covered a bunch of high-order workload constructs that encapsulate Pods with particular objectives, like Deployment for application deployments, StatefulSet for database deployments, and CronJob for period tasks. DaemonSet is another workload type that allows you to run a Pod on every node.

When would you need that? It's almost entirely for cluster operational reasons, like logging, monitoring, and security. As an application developer, DaemonSet is generally not your go-to deployment construct. Due to the ability to expose services internally on a cluster IP, any Pod in your cluster can talk to any service you create, you don't need to run services on every node to make them available. And if you need to be able to connect to a service on localhost, you can do that virtually with NodePort. DaemonSets are generally for when you need to perform operations at a node level, like reading load logs, or observing performance, putting them squarely in the administrative domain.

DaemonSets are typically how logging, monitoring and security vendors deploy their software. This software performs actions like reading logs off the node and uploading it to a central logging solution, querying the kubelet API for performance metrics (like how many pods are running, their boot times, etc.), and for security, such as monitoring container and host behaviors. These are all examples of Pods that need to be on every node to gather the data they need to provide their solution.

The typical cluster will have a few DaemonSets running in kube-system, such as this abridged list from a GKE Autopilot cluster that provides functionality like logging, monitoring and cluster DNS.

```
$ kubectl get daemonset -n kube-system
NAMESPACE      NAME
kube-system    filestore-node
kube-system    fluentbit-gke
kube-system    gke-metadata-server
kube-system    gke-metrics-agent
kube-system    kube-proxy
kube-system    metadata-proxy-v0.1
kube-system    netd
kube-system    node-local-dns
kube-system    pdcsi-node
```

Typically, application developers will not be creating DaemonSets directly, but rather be using off the shelf ones from vendors. By way of example though, here is a simple DaemonSet that reads logs from the node into standard output (stdout).

Listing 12.3 DaemonSet/logreaderds.yaml

```

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: logreader
spec:
  selector:
    matchLabels:
      ds: logreaderpod
  template:
    metadata:
      labels:
        ds: logreaderpod
    spec:
      containers:
        - image: ubuntu
          command: ["tail", "-f", "/var/log/kube-proxy.log"]
          name: logreadercontainer
          resources:
            requests:
              cpu: 50m
              memory: 100Mi
              ephemeral-storage: 100Mi
          volumeMounts:
            - name: logpath
              mountPath: /var/log
              readOnly: true
      volumes:
        - hostPath:
            path: /var/log
            name: logpath

```

To create the DaemonSet:

```

$ kubectl create -f 12.2_DaemonSet/logreader.yaml
daemonset.apps/logreader created

```

Once the Pods are ready, we can tail the output and we'll see the output from the node's kube-proxy.log file

```

$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
logreader-2nbt4     1/1     Running   0           4m14s
$ kubectl logs -f logreader-2nbt4
I0125 01:39:28.576504      1 proxier.go:845] "Syncing iptables rules"
I0125 01:39:28.615108      1 proxier.go:812] "SyncProxyRules complete"
      elapsed="38.697941ms"

```

This is just a trivial example. In practice, you will likely encounter DaemonSets when deploying logging, monitoring, and security solutions from vendors.

12.3 Pod Security Context

The PodSpec has a `securityContext` property where the security attributes of the Pod, and its containers are defined. If your Pod needs to perform some kind of administrative function

(for example, perhaps it's part of a DaemonSet which is doing a node-level operation), it's here where you would define the various privileges it needs. For example, here is a Pod in a DaemonSet that requests privilege (root access) on the node:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: admin-workload
spec:
  selector:
    matchLabels:
      name: admin-app
  template:
    metadata:
      labels:
        name: admin-app
    spec:
      containers:
        - name: admin-container
          image: ubuntu
          securityContext:
            privileged: true
```

As a developer of a non-administrative application that run on Kubernetes, you will more likely be using these properties to *limit* what functions your Pod can use in order to reduce risk. Contrasting the above example, here is the PodSpec for a Pod with locked-down privileges that runs as the non-root user and cannot elevate privileges.

```
apiVersion: v1
kind: Pod
metadata:
  name: timeserver-demo
  labels:
    app: timeserver
spec:
  containers:
    - name: timeserver-container
      image: wdenniss/timeserver2:latest
      securityContext:
        runAsNonRoot: true
        runAsUser: 1001
        allowPrivilegeEscalation: false
      capabilities:
        drop:
          - ALL
```

By default, any Pod is free to request whatever capabilities it wants, even root access (unless your Kubernetes platform restricts this, as some nodeless platforms do). As the cluster operator, this may be something you want to restrict as it basically means that anyone with kubectl access to the cluster has root privileges. Furthermore, there are some other recommended principles for hardening clusters, like not running containers as the root user (which is distinct from having root on the node), something that is enforced by the `runAsNonRoot: true` configuration in the prior example.

The remainder of this chapter will cover these topics, starting with how to build containers so they don't need to run as the root user, and how as a cluster administrator you can force users of the cluster to adopt this and other desired security settings.

12.4 Non-Root Containers

One common security recommendation when deploying containers is to not run them as the root user. The reason for this is that despite all the fancy packaging, Linux containers are basically just processes that run on the host with sandboxing technology applied (like Linux cgroups and namespaces). If your container is built to run using the root user (which is the default), then it actually runs as root on the node, just sandboxed. The container sandboxing means that the process doesn't have the power of root, but it's still running under the root user. The issue with this configuration is that while the sandboxing prevents the process from having root access, if there is ever a "container escape" vulnerability due to bugs in the underlying Linux containerization technology, the sandboxed container process can gain the same privileges as the user it's running, meaning if the container is running as root a container escape would give full root access on the node, which is not so good.

Since Docker runs all processes as root by default, this means that any container escape vulnerabilities can present an issue. While such vulnerabilities are fairly rare, they do occur, and for the security principle known as "defense in depth", it's best to protect against it. Defense in depth means that even though container isolation offers protection of the host in the event your application is breached, ideally you would have further layers of defense in case *that* protection is breached. In this case, defense in depth means running your containers as the non-root user, so in the event an attacker can breach your container and take advantage of a container escape vulnerability in Linux, they still wouldn't end up with elevated privileges on the node, and would need to string together a yet another vulnerability to elevate their privileges making for 3 layers of defense (your application, Linux containerization, and Linux user privileges).

WHY DOCKER DEFAULTS TO THE ROOT USER You may be wondering, if it's the best practice to not run container processes as root, why then Docker defaults to the root user when building containers. The answer is developer convenience. It's convenient to act as the root user in a container, as you can use privileged ports (those with numbers below 1024, like the default HTTP port 80), and don't have to deal with any folder permission issue. As you'll see below, building and running containers with the non-root user can introduce some errors that need to be worked through. If you adopt this principle from the start however, you may not find it so difficult to fix these issues as they arise, and the payoff is adding one more layer of defense into your system.

Preventing containers from running as the root user is simple in Kubernetes, although the problem (as we'll see below) is that not all containers are designed to run this way and may fail.

You can annotate your Pods in Kubernetes to prevent them running as a root user. So, to achieve the goal of "not running as root", the first step is to simply add this annotation! If you're configuring a Kubernetes cluster for a wider team (or if you're a member of that team

using such a configured cluster), a Kubernetes admission controller can be used to automatically add this annotation to every pod. The end result is the same, so for this demo we'll just add it manually. The following deployment enforces the best practice to prevent containers running as root.

Listing 12.4 NonRootContainers/1_permission_error/deploy.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 1
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:6
          securityContext:
            runAsNonRoot: true
```

Unfortunately, we're not done because the container itself doesn't configure a non-root user to run as. If you try and create this deployment, Kubernetes will enforce the securityContext and won't let the container run as root. Below is the truncated output you'll see if you try and run this deployment.

```
$ kubectl get pods
NAME                                READY   STATUS                    RESTARTS   AGE
timeserver-pod-fd574695c-5t92p      0/1     CreateContainerConfigError 0           34s
$ kubectl describe pod timeserver-pod-fd574695c-5t92p
Name:                                timeserver-pod-fd574695c-5t92p
Events:
  Type     Reason      Age          From          Message
  ----     -
Warning   Failed      10s (x3 over 23s)  kubelet       Error:
            container has runAsNonRoot and image will run as root (pod: "timeserver-pod-fd574695c-5t92p_default(8cfaf4a9-c2e6-4dde-b506-14ab04444f50)", container: timeserver-container)
```

To resolve, you need to configure the user that the Pod will be run under. Root is always user 0, so we just need to set any other user number, I'm going to pick user 1001. This can either be declared in the Dockerfile with `USER 1001`, or in the Kubernetes configuration with `runAsUser: 1001`. When both are present, the Kubernetes configuration takes priority, similar to how the `command` parameter in a Kubernetes deployment overrides `CMD` if present in the Dockerfile.

Here's the Dockerfile option:

```
FROM python:3
COPY . /app
WORKDIR /app
RUN mkdir logs
CMD python3 server.py
USER 1001
```

Or, in the Deployment:

Listing 12.5 NonRootContainers/2_fixed/deploy.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: timeserver
spec:
  replicas: 1
  selector:
    matchLabels:
      pod: timeserver-pod
  template:
    metadata:
      labels:
        pod: timeserver-pod
    spec:
      containers:
        - name: timeserver-container
          image: docker.io/wdenniss/timeserver:7
          securityContext:
            runAsNonRoot: true
            runAsUser: 1001
```

Both approaches work, but what I recommend is to configure it on the Kubernetes side as this is better for keeping your development and production environments separate. If you specify the run-as user in the Dockerfile and want to run your container locally outside of Kubernetes and try to mount a volume, you'll hit a snag like Docker issue #2259 which prevents you from mounting a volume as a user other than root³, a 7+ year old issue. Since the original security concern to not run containers as root is only related to production, why not relegate this whole "run as non-root" concern to production as well! Fortunately, it's easy to let your container that runs as root in Docker for maximum convenience, and non-root in production in Kubernetes for better defense in depth.

The above configuration is enough to run our container as non-root. Provided that the container is capable of running as non-root then your job is done. Most public, well-known containers should be designed to run as non-root, but this likely isn't the case for your own containers.

In the case of our example container it wasn't designed to run as non-root and will need to be fixed. Two major differences when running the container as non-root are that you can't listen on privileged ports, those between 1 and 1023, and you don't have write access by default to container writable layer (meaning, by default you can't write any files). This would

³ <https://github.com/moby/moby/issues/2259>

be a problem for the Timeserver sample app (first introduced in Chapter 2, section 2.2), which listens on port 80, and writes a logfile to `/app/logs`.

UPDATING CONTAINERS TO RUN AS NON-ROOT

If you were to create the deployment above (listing X), you will see that there is no config error when deployed, but the container itself is crashing. When your container starts crashing after you change the user it runs as non-root, it's probably a permission error related to that. Before you start debugging the non-root user errors, be sure your container runs fine as root, otherwise the issue could be something completely unrelated.

The steps to debug permission issues for containers running as non-root will vary, but let's walk through how to find and fix these two common errors with our example app. The following is the output and truncated logs that I see for this crashing container.

```
$ kubectl get pods
NAME                                READY   STATUS              RESTARTS   AGE
timeserver-demo-774c7f5ff9-fq94k    0/1     CrashLoopBackOff    5 (47s ago) 4m4s
$ kubectl logs timeserver-demo-76ddf6d5c-7s9zc
Traceback (most recent call last):
  File "/app/server.py", line 23, in <module>
    startServer()
  File "/app/server.py", line 17, in startServer
    server = ThreadingHTTPServer(('',80), RequestHandler)
  File "/usr/local/lib/python3.9/socketserver.py", line 452, in __init__
    self.server_bind()
  File "/usr/local/lib/python3.9/http/server.py", line 138, in server_bind
    socketserver.TCPServer.server_bind(self)
  File "/usr/local/lib/python3.9/socketserver.py", line 466, in server_bind
    self.socket.bind(self.server_address)
PermissionError: [Errno 13] Permission denied
```

Fortunately, the port issue in Kubernetes is an easy fix without any end-user impact. We can change the port that the container uses, while keeping the standard port 80 for the load balancer. First let's update the port used by the container:

Listing 12.6 /timeserver7/server.py

```
//...

def startServer():
    try:
        server = ThreadingHTTPServer(('',8080), RequestHandler)
        print("Listening on " + ":".join(map(str, server.server_address)))
        server.serve_forever()
    except KeyboardInterrupt:
        server.shutdown()

if __name__ == "__main__":
    startServer()
```

If we're changing ports in the application, then we'll need to update our Kubernetes service configuration to match the new port by updating the `targetPort`. Note that we do **not** need to change the actual port of the Service. The Service glue is provided by Kubernetes, and doesn't run as a particular user so it can use ports below 1024.

Listing 12.7 NonRootContainers/2_fixed/service.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: timeserver
spec:
  selector:
    pod: timeserver-pod
  ports:
    - port: 80
      targetPort: 8080
      protocol: TCP
  type: LoadBalancer

```

Once the socket issue is fixed, and we re-run the application, another error will be encountered when the app attempts to write to the log file on disk. This error doesn't stop the app starting, but is encountered when a request is made. Looking at those logs, we see:

```

$ kubectl logs timeserver-demo-5fd5f6c7f9-cxzrb
10.22.0.129 - - [24/Mar/2022 02:10:43] "GET / HTTP/1.1" 200 -
Exception occurred during processing of request from ('10.22.0.129', 41702)
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/socketserver.py", line 683, in process_request_thread
    self.finish_request(request, client_address)
  File "/usr/local/lib/python3.10/socketserver.py", line 360, in finish_request
    self.RequestHandlerClass(request, client_address, self)
  File "/usr/local/lib/python3.10/socketserver.py", line 747, in __init__
    self.handle()
  File "/usr/local/lib/python3.10/http/server.py", line 425, in handle
    self.handle_one_request()
  File "/usr/local/lib/python3.10/http/server.py", line 413, in handle_one_request
    method()
  File "/app/server.py", line 11, in do_GET
    with open("logs/log.txt", "a") as myfile:
PermissionError: [Errno 13] Permission denied: 'logs/log.txt'

```

If you see a permission denied error when running as non-root when writing a file, it's a clear sign that your folder permissions have not been setup correctly for non-root users.

The simplest way to solve this is to set the group permissions on the folder in question. I like using the group permissions, as we can use the same group (being group 0) for running locally using Docker, and deploying in production to Kubernetes without environment-specific changes in the Dockerfile. Let's update the Dockerfile to give write access to group 0:

Listing 12.8 /timeserver7/Dockerfile

```

FROM python:3.10
ENV PYTHONUNBUFFERED 1
COPY . /app
WORKDIR /app
RUN mkdir logs
RUN chgrp -R 0 logs \
    && chmod -R g+rwX logs
CMD python3 server.py

```

If you want to run the container in Docker using a non-root user, you can also override this at runtime, like so: `docker run --user 1001:0`.

So there we have it, our container now runs happily as the non-root user. If you want to see all the changes made, compare the folder `Chapter12/timeserver6` to `Chapter12/timeserver7` for the app changes, and `Chapter12/12.4_NonRootContainers/1_permission_error` to `Chapter12/12.4_NonRootContainers/2_fixed` for the configuration.

12.5 Admission Controllers

In the previous section we added `runAsNonRoot` to our Pod to prevent it from ever running as root, but we did it manually. If this is a setting we want for all Pods, ideally we'd be able to configure the cluster to reject any Pod without this configuration, or even just add it automatically.

This is where admission controllers come in. Admission controllers are bits of code that are executed via webhooks when you create an object (like with `kubectl create`). There are two types: validating, and mutating. Validating admission webhooks can accept or reject the Kubernetes object, for example, rejecting Pods without `runAsNonRoot`. Mutating admission webhooks can change the object as it comes in, for example setting `runAsNonRoot` to true.

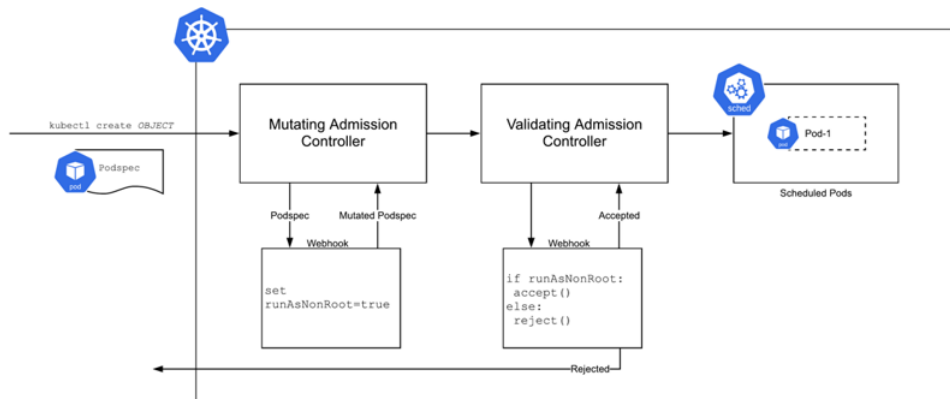


Figure 12.3: admission process of a pod that gets scheduled

You can write your own admission controllers to implement the behavior you desire, but depending on what you're hoping to achieve, you may not need to as Kubernetes ships with an admission controller out of the box, and others may be available as commercial or open source deployments.

12.5.1 Pod Security Admission

Writing admission controllers is no walk in the park. You need to configure certificates, build an application that can be setup as webhook which conforms to the request/response API of Kubernetes, and have a development process to keep it up to date as Kubernetes changes (which it does fairly frequently). The good news is that the typical end-developer running on Kubernetes would not need to write their own admission controllers. You'll typically use those from third party security vendors, and those included in Kubernetes.

Kubernetes offers included admission controllers for enforcing security policies like `runAsNonRoot`. Prior to Kubernetes 1.25, `PodSecurityPolicy`⁴ served this purpose but never left beta and was removed. Pod Security Admission is the current recommended way to enforce security policies via an admission controller. It will be enabled by default from Kubernetes 1.23 (starting as a beta feature), but fortunately you can deploy it manually⁵ into clusters running older version of Kubernetes, or where the feature wasn't enabled by the platform operator.

Pod Security Admission Beta

I've avoided discussing beta features as much as possible in the book, due to potential churn, and at the time of writing Pod Security Admission is beta in Kubernetes 1.23. Given its availability as a stand-alone deployment (which has its own lifecycle outside of Kubernetes), I hope that this section will have a bit of longevity. Even if this feature never makes it to GA as a built-in admission controller, you could always install the admission webhook directly, and the content here will apply. I'm also hoping that since this is the second bite at the apple when it comes to security admission controllers, this one might actually make it to GA.

INSTALLING POD SECURITY ADMISSION

First, verify if the `pod-security-webhook` is installed on your cluster. If it shows up in the list of `ValidatingWebhookConfigurations`, then you don't need to install it.

```
$ kubectl get ValidatingWebhookConfiguration | grep pod-security-webhook.kubernetes.io
pod-security-webhook.kubernetes.io          2          26h
```

If you need to install it, then you can in the following way:

```
$ git clone https://github.com/kubernetes/pod-security-admission.git
$ cd pod-security-admission/webhook
$ make certs
$ kubectl apply -k .
```

It will take a moment to install, and in that time you won't be able to schedule any Pods.

POD SECURITY POLICIES

Pod Security defines⁶ three security policy levels that apply at a namespace level. These are:

⁴ <https://kubernetes.io/docs/concepts/policy/pod-security-policy/>

⁵ <https://kubernetes.io/docs/concepts/security/pod-security-admission/#webhook>

⁶ <https://kubernetes.io/docs/concepts/security/pod-security-standards/>

- **Privileged** – Pods have unrestricted administrative access, and can gain root access to nodes.
- **Baseline** – Pods cannot elevate privileges to gain administrative access, while still retaining most functionality.
- **Restricted** – Implements current best practices for hardening (i.e. defense in depth), adding additional layers of protection over the baseline profile, including restricting running as the root user.

Basically: privileged should be used only for system workloads. Baseline offers a good balance of security, and compatibility, while restricted offers additional defense in depth at a cost of some compatibility (such as needing to ensure all containers can run as non-root, per section X above).

CREATING A NAMESPACE WITH POD SECURITY

In keeping with the running example of this chapter, and to implement the most secure profile, let's create a namespace with the *restricted* policy. This will require pods to run as a user other than root (per the docs⁷ with the restricted policy "containers must be required to run as non-root users"), and will enforce several other security best practices as well.

To start, create a new namespace with the restricted policy. We'll call this namespace "team1", as it can be the place for a theoretical "team1" to deploy their code to.

Listing 12.9 PodSecurityAdmission/namespace.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: team1
  labels:
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/enforce-version: v1.23
```

These two labels set the policy we want to enforce, and the version of the policy that will be enforced. While you can skip the version label, there is a good reason not to as the definition of what the policy actually enforces may evolve as new security risks are uncovered. If we don't reference the version, then we'll get whatever the current policy is, which could break existing workloads as that policy is updated. Ideally you would test the newer policy versions in a staging namespace or cluster to validate them first, before updating the enforce-version in your production environment.

Let's create this namespace:

```
$ kubectl create -f 12.3.1_PodSecurityAdmission/namespace.yaml
$ kubectl config set-context --current --namespace=team1
```

Now if we try to deploy a Pod from chapter 3 that doesn't set `runAsNonRoot` the pods will be rejected.

⁷ <https://kubernetes.io/docs/concepts/security/pod-security-standards/#restricted>

```
$ kubectl create -f Chapter03/3.2.4_ThePodspec/pod.yaml
Error from server (Forbidden): error when creating "Chapter03/3.2.4_ThePodspec/pod.yaml":
admission webhook "pod-security-webhook.kubernetes.io" denied the request: pods
"timeserver" is forbidden: violates PodSecurity "restricted:v1.23":
allowPrivilegeEscalation != false (container "timeserver-container" must set
securityContext.allowPrivilegeEscalation=false), unrestricted capabilities
(container "timeserver-container" must set
securityContext.capabilities.drop=["ALL"]), runAsNonRoot != true (pod or container
"timeserver-container" must set securityContext.runAsNonRoot=true)
```

If we add the appropriate `securityContext` to satisfy the Pod Security admission policy, and use the updated container that is designed to run as root from the previous section, then our pod will be admitted.

Listing 12.10 PodSecurityAdmission/nonroot_pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: timeserver-pod
spec:
  containers:
  - name: timeserver-container
    image: wdenniss/timeserver:7
    securityContext:
      runAsNonRoot: true
      allowPrivilegeEscalation: false
      runAsUser: 1001
      capabilities:
        drop:
        - ALL
```

Debugging Pod Admission Rejects for Deployments

The two examples in this section used Pods, rather than Deployments. The reason I did that is it's easier to debug when the Pod admission is rejected. Once you confirm it's working as expected, you can always embed the PodSpec in the Deployment of your choice.

Unfortunately, if you create a Deployment that violates these constraints, you won't see an error printed on the console like for my example when I tried to create the Pod directly. This is an unfortunate fact of Kubernetes' implementation of Deployment. Creating the Deployment object itself succeeds, so you don't see an error on the console. However, when the Deployment then goes to create its Pods, they will fail. Also, since the Deployment actually creates an object called a ReplicaSet under the hood to manage Pods of a particular version of the deployment, you won't even find this error if you describe the Deployment object, but rather need to inspect its ReplicaSet.

I've not mentioned ReplicaSet yet in the book as it's basically implementation detail. Basically a ReplicaSet is a workload construct that manages a set of Pods. Deployment uses them by creating a new ReplicaSet for each version you deploy. So when you're doing a rolling update, the deployment will actually have 2 replica sets, one for the old version, one for the new, and these are scaled gradually to achieve the rolling update. Normally this implementation detail doesn't matter—which is why I didn't spend any time on it in the book—but here is one of the few times it does, since the replicaset is where this particular error is hidden.

So this is... not exactly an ideal UX, but here's how to debug this type of issue. Normally when you create a Deployment, it will create Pods. If you type `kubectl get pods`, you should see a bunch of Pods. Now, those pods may not always be Ready—there are a bunch of reasons why they might be pending (and in some cases, may get stuck in Pending forever), but there would still normally at least be the pod objects there in some state. If when you call `kubectl get pods`, you don't see any Pod objects at all for your Deployment, it could mean that those pods were rejected during admission, which is why there are no objects.

Since it's the ReplicaSet owned by the Deployment that actually creates the pods, you need to describe the ReplicaSet to see the error with `kubectl describe replicaset` (`kubectl describe rs` for short). Here's an example, with the output truncated to show the error message of interest:

```
$ kubectl create -f Chapter03/3.2_DeployingToKubernetes/deploy.yaml
$ kubectl get deploy
NAME          READY    UP-TO-DATE    AVAILABLE    AGE
timeserver    0/3      0              0             12s
$ kubectl get pods
No resources found in myapp namespace.
$ kubectl get rs
NAME                                DESIRED    CURRENT    READY    AGE
timeserver-5b4fc5bb4                3          0          0        31s
$ kubectl describe rs
Events:
  Type          Reason          Age           From              Message
  ----          -
Warning        FailedCreate     36s           replicaset-controller Error creating:
admission webhook "pod-security-webhook.kubernetes.io" denied the request: pods
"timeserver-5b4fc5bb4-hvqcm" is forbidden: violates PodSecurity "restricted:v1.23":
allowPrivilegeEscalation != false (container "timeserver-container" must set
securityContext.allowPrivilegeEscalation=false), unrestricted capabilities
(container "timeserver-container" must set
securityContext.capabilities.drop=["ALL"]), runAsNonRoot != true (pod or container
"timeserver-container" must set securityContext.runAsNonRoot=true)
```

BALANCING SECURITY WITH COMPATIBILITY In this section we used the example of the restricted pod security profile, and configured our container to be able to run as the non-root user. Hopefully this has given you the confidence to be able to run containers in a highly secure manner. While this is the best practice, and may be required in situations like regulated industries, there is a clear tradeoff with ease of development and it may not always be practical. Ultimately it's up to you, your security team (and maybe, your regulators) to determine what security profile you're happy with. I'm not necessarily recommending every single Kubernetes deployment should be into a namespace with the "restricted" profile. I would suggest that you should be using "baseline" for every non-administrative workload you deploy in your cluster, as it helps protect your cluster in the event that one of your containers is compromised, and shouldn't cause any incompatibility with the average app. Administrative workloads that need the "privileged" profile should be run in their own namespaces, separate to common workloads.

12.6 Role-based Access Control (RBAC)

Let's say that you have a requirement for Pods to runAsNonRoot (section 12.4), and setup an admission controller to force this using Pod Security Admission (section 12.5). This sounds great, provided you trust all the users of your cluster not to mess anything up and remove those restrictions whether accidentally or on purpose. To actually enforce the requirements of your admission controller, and create a tiered user permission setup with roles like "platform operator" (who can configure namespaces and controller), and "developer" (who can deploy to namespaces, but not remove admission controllers), you use role-based access control (RBAC).

RBAC is a way to control what access users of the cluster have. One common setup is to give developers in a team access to a particular namespace in the cluster, with all the desired Pod Security configured. This gives them the freedom to deploy whatever they like within the namespace, provided it conforms to the security requirements that's been set. This way it's still DevOps, developers are the ones doing the deployments, just with some guardrails in place.

RBAC is configured through two configuration types at a namespace level: Role, and RoleBinding. Role is where you define a particular role for a namespace, like the "developer" role. RoleBinding is where you assign this role to subjects in your cluster, i.e. your developer identities. There are also cluster-level versions of these being ClusterRole and ClusterRoleBinding which behave identically to their namespace level counterparts, except that they grant access at a cluster level.

NAMESPACE ROLE

In the Role, you specify the API group(s), the resource(s) within that group, and the verb(s) which you are granting access to. Access is additive, there is no subtractive option, so everything you define grants access. Since our goal is to create a Role which gives the developer access to do pretty much everything within their namespace, *except* to modify the namespace itself and remove the Pod Security annotation, the following is a Role that can achieve that:

Listing 12.11 RBAC/role.yaml

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: developer-access
  namespace: team1
rules:
  - apiGroups:
    - "" #A
    resources:
    - namespaces #B
    verbs: ["get"] #B
  - apiGroups: #C
    - "" #C
    resources: #C
    - events #C
    - pods #C
    - pods/log #C
    - services #C
    - secrets #C
    - configmaps #C
    - persistentvolumeclaims #C
    verbs: ["*"] #C
  - apiGroups:
    - apps #D
    - autoscaling #E
    - batch #F
    - networking.k8s.io #G
    - policy #H
    resources: ["*"]
    verbs: ["*"]

```

#A The empty string here indicates the core API group

#B Allow developers to view the namespace resource, but not edit it

#C Grant developers full access on core workload types

#D 'apps' includes resources like Deployment

#E 'autoscaling' includes resources like the HorizontalPodAutoscaler

#F 'batch' includes the Job workloads

#G 'networking.k8s.io' is needed so developers can configure Ingress

#H 'policy' is required for configuring PodDisruptionBudgets

This Role grants access to the team1 namespace, and allows the user to modify Pods, Services, Secrets and ConfigMaps within the core API grouping, and all resources in the apps, autoscaling, batch, networking.k8s.io and policy groupings. This particular set of permissions will let the developer deploy nearly every YAML file in this book, including Deployments, StatefulSets Services, Ingress, HPA Autoscaling, Jobs and more. Importantly, the "namespaces" resource is not listed in the core API group (the empty string group), so the user won't be able to modify the namespace.

To grant this role to our developer, we can use a RoleBinding where the subject is our user.

Listing 12.12 RBAC/rolebinding.yaml

```

kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: developerA
  namespace: team1
roleRef:
  kind: Role
  name: developer-access
  apiGroup: rbac.authorization.k8s.io
subjects:
# Google Cloud user account
- kind: User
  name: developerA@gmail.com #A

```

#A Set this to be the identity of your developer. For GKE, this is a Google User who has “Kubernetes Engine Cluster Viewer” IAM role access to the project

Note that the acceptable values within the User subject are governed by your Kubernetes platform, and any identity systems you have configured. With Google Cloud, the name here can be any Google user. RBAC authorizes the user to the actions specified in the Role, however in order to authenticate the user, they need the `clusters.get` permission which is configured outside of Kubernetes (the “Kubernetes Engine Cluster Viewer” role allows users to authenticate to the cluster, without granting any further permission, after which RBAC can be used). The exact steps here will vary depending on your platform provider.

AUTHENTICATION VS AUTHORIZATION Authentication (AuthN) is the means with which the user presents their identity credentials to the system. In this case, being able to authenticate to the cluster means that the user can retrieve credentials to access the cluster via `kubectl`. Authorization (AuthZ) is the process to grant users access within the cluster. Depending on your platform’s IAM system, it should be possible to allow users to authenticate to the cluster (e.g. get `kubectl` credentials) but then not actually be able to perform any action (no authorization). You can then use RBAC to grant the precise authorization you want. In the case of GKE, granting users the “Kubernetes Engine Cluster Viewer” role in the IAM permissions (outside of Kubernetes) will allow them to Authenticate, after which you can authorize them to access specific resources using RBAC, and the examples shown here. Again, depending on your particular Kubernetes platform, it’s possible (as is the case with GKE) that some IAM roles will also grant the user authorization to some resources in addition to whatever RBAC rules you have here (the project-wide “Viewer” role is one such example in GKE, that will allow users to view most of the resources in the cluster without needing specific RBAC rules to do so).

As the cluster administrator, create the namespace, and these two objects:

```

$ kubectl create ns team1
namespace/team1 created
$ kubectl create -f role3.yaml
krole.rbac.authorization.k8s.io/developer-access created
$ kubectl create -f binding.yaml
rolebinding.rbac.authorization.k8s.io/developerA created

```

With this role and binding deployed in the cluster, our developer user should be able to deploy most of the code in this book in the “team1” namespace, but specifically not be able to

change any other namespaces, nor edit the team1 namespace itself. For a meaningful experiment, you'll need to set as the User subject in the RoleBinding an actual user, for example a test developer account.

Switch to the test developer account, by authenticating to the cluster as the user specified in the Subject. Once authenticated as our developer user, and try to deploy something into the default namespace, it should fail:

```
$ kubectl config set-context --current --namespace=default
$ kubectl create -f Chapter03/3.2.3_DeployingToKubernetes/deploy.yaml
Error from server (Forbidden): error when creating
    "Chapter03/3.2.3_DeployingToKubernetes/deploy.yaml": deployments.apps is forbidden:
    User "example@gmail.com" cannot create resource "deployments" in API group "apps" in
    the namespace "default": requires one of ["container.deployments.create"]
    permission(s).
```

Switching the context to the team1 namespace, for which we configured with the Role above, we should now be able to create the deployment.

```
$ kubectl config set-context --current --namespace=team1
Context "gke_project-name_us-west1_cluster-name" modified.
$ kubectl create -f Chapter03/3.2_DeployingToKubernetes/deploy.yaml
deployment.apps/timeserver created
```

While this developer can now deploy things in the namespace, if they try to edit the namespace to gain the privileged Pod Security level, it will be restricted to the lack of edit permission on the namespace resource

```
$ kubectl label --overwrite ns team1 pod-security.kubernetes.io/enforce=privileged
Error from server (Forbidden): namespaces "team1" is forbidden: User "example@gmail.com"
    cannot patch resource "namespaces" in API group "" in the namespace "team1":
    requires one of ["container.namespaces.update"] permission(s).
```

CLUSTER ROLE

So far, we've setup a Role and RoleBinding to give a developer access to a particular namespace. With this Role, they can deploy most of the configuration in this book. There are however a couple of things they won't be able to do, and that is create a PriorityClass (chapter 6), create a StorageClass (chapter 9), or list the PersistentVolumes in the cluster (chapter 9). Those resources are considered cluster-wide objects, so we can't amend the namespace-specific Role we created earlier. Instead, we'll need a separate ClusterRole and ClusterRole binding to grant this additional access.

Figuring out what permissions to grant

I've done the leg work here to provide a Role that cover all the needed permissions to deploy the code in the book, but this may be missing permission that you need to grant developers. To figure out which groups, resources and verbs you need to grant you can consult the API docs. When debugging permission errors, say a developer is complaining that they don't have access they need you can simply inspect the error message, for example the following:

```
$ kubectl create -f balloon-priority.yaml
Error from server (Forbidden): error when creating "balloon-priority.yaml":
priorityclasses.scheduling.k8s.io is forbidden: User "example@gmail.com" cannot
create resource "priorityclasses" in API group "scheduling.k8s.io" at the cluster
scope: RBAC: clusterrole.rbac.authorization.k8s.io "developer-cluster-access" not
found
requires one of ["container.priorityClasses.create"] permission(s).
```

To add this to the Role, we can see that the group is scheduling.k8s.io, the resource is "priorityClasses", and the verb is "create", and thus form our Role based on that.

Here is a ClusterRole to provide the additional permissions needed to create StorageClass and PriorityClass objects:

Listing 12.13 RBAC/clusterrole.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: developer-cluster-access
rules:
- apiGroups: #A
  - scheduling.k8s.io #A
  resources: #A
  - priorityclasses #A
  verbs: ["*"] #A
- apiGroups: #B
  - storage.k8s.io #B
  resources: #B
  - storageclasses #B
  verbs: ["*"] #B
- apiGroups: #C
  - "" #C
  resources: #C
  - persistentvolumes #C
  - namespaces #C
  verbs: ["get", "list"] #C
```

#A Grant developer access to modify all PriorityClasses in the cluster

#B Grant developer access to modify all StorageClasses in the cluster

#C Grant developer read-only access to view and list PersistentVolumes and Namespaces

And the ClusterRoleBinding which looks very similar to the RoleBinding used earlier:

Listing 12.14 RBAC/clusterrolebinding.yaml

```

kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: developerA
  namespace: team1
roleRef:
  kind: Role
  name: developer-access
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: User
  name: example@gmail.com #A

```

#A Set this to be the identity of your developer. For GKE, this is a Google User who has “Kubernetes Engine Cluster Viewer” IAM role access to the project

With these additional cluster roles and bindings, our developer should be able to perform every action in this book.

Identity Federation

For RBAC to be able to reference your developer identities as Users and Groups, your cluster needs to understand the concept of your developer’s identities. In the case of GKE, it natively understands Google users in the User field, and also Google groups, when Google Groups for RBAC (a proprietary feature) is enabled. Depending on your platform and your corporate identity provider you may have this access already, or you may need to set it up. This setup is outside the scope of this book, what you probably want to do is configure OpenID Connect (OIDC) integration so that RBAC can reference identities provided by OIDC.

When using an identity system plugin that can offer Groups support, instead of needing to list every “User” as a subject of our role bindings, we can specify a single Group instead.

APPLYING THE POD SECURITY

Previously, we created the namespace without using Pod Security. If we go back and configure the namespace with the Pod Security labels from the previous section, it would have the effect of locking down this namespace to the “restricted” profile, and thanks to RBAC, our developer would not be able to modify that restriction. Mission accomplished.

RBAC FOR SERVICEACCOUNTS

In the examples in this section, we used RBAC with the “User” subject, that’s because our developers are actual human users of our cluster. Another common use-case for RBAC is to grant access to services, that is, the code that runs in the cluster.

Let’s say you have a Pod that belongs to a Deployment in the cluster which needs to access the Kubernetes API, like perhaps it’s monitoring the pod status of another deployment. To give this machine user access, you can create a Kubernetes “ServiceAccount”, and then reference this in the subject of your RBAC binding instead of a user.

You may see some documentation that sets up “ServiceAccounts” for human users, where the user then downloads the certs of the service account to interact with Kubernetes. While this is one way to configure your developers and bypasses the need to setup identity federation, it is not recommended as it sits outside of your identity system. For example, if say the developer resigned and their account was suspended in the identity system, the tokens they downloaded for the ServiceAccount would continue to be valid. It’s better to properly configure identity federation and only use “User” subjects for human users, so that if the user is suspended their Kubernetes access will also be revoked. Once again, managed platforms like Google Cloud make this integration easy, for other platforms you may need to do a bit of setup to get it working.

Kubernetes ServiceAccounts are utilized when you have a Pod inside the cluster that needs its own access to the Kubernetes API. For example, say you want to create a Pod to monitor another Deployment, you can create a ServiceAccount to use as the subject of the RoleBinding, and assign that service account to the Pod. The Pod can then utilize that credential when making API calls, including with kubectl.

12.7 Summary

One of the most important things you can do as a developer using Kubernetes is to keep your cluster, and your code up to date. This can be time consuming, but is aided by choosing a well managed platform, and using a CI/CD system to build your code frequently. Additionally, Kubernetes offers several security-focused configuration options to help lock down your cluster, particularly when dealing with large teams of developers with the separation of the cluster operator, and developer roles.

- Kubernetes is a large and complex system with a huge surface area
- It’s important to keep your cluster and its nodes up to date to mitigate against security vulnerabilities
- Docker base images also introduce their own attack surface area, requiring monitoring and updating of deployed containers
- Using the smallest possible base image can help to reduce this surface area, decreasing the frequency of application updates to mitigate security vulnerabilities
- DaemonSets are another workload type, along with Deployment, StatefulSet and Job, commonly used to configure logging, monitoring, and security software in the cluster
- The Pod Security Context is how Pods are configured to have elevated, or restricted permissions
- Admission Controllers can be used to make changes to Kubernetes objects as they are created, and also enforce requirements including around the Pod Security Context
- Kubernetes ships with an Admission Controller named Pod Security Admission to enable to you to enforce security profiles like baseline, for mitigating most known attacks and restricted to enforce security best practices on pods
- RBAC is a role-based permission system that allows users with the cluster administrator role to grant fine-grained access to developers in the system, for example restricting them to a given namespace, and preventing them from altering admission controller rules (like Pod Security)