MIRANTIS PRESS

Learn Kubernetes 5 Minutes at a Time

An introductory guide for developers

Eric Gregory

Learn Kubernetes 5 Minutes at a Time

by Eric Gregory

An introductory guide for developers

First edition

Welcome!

Kubernetes is notoriously challenging. All too often, developers are expected to learn this complex system on the run—whether because their organization has decided to migrate, or because they're starting (or seeking) a new position.

This book is for developers who need a crash course in Kubernetes. Each chapter is designed as a concise lesson that should fit into a short break or a lunch. For fast readers, many of these chapters may take only five minutes. You might complete a chapter a day and so learn the fundamentals of Kubernetes in under two weeks—or you might work through these chapters all over the course of a few hours.

Goals and non-goals

This book is *not* an introductory guide for operators. It is also not an exhaustive survey of Kubernetes for any role.

Our objective is to cover the essentials through hands-on exercises…*without* getting mired in details that aren't relevant to a developer at the beginning stages of a Kubernetes journey.

Prerequisites

This book assumes only a basic familiarity with containers and the Linux command line. If you have some experience building and running containers and you know how to navigate and manage files and directories on the command line, you should

be all set. If you're not familiar with containers, I recommend starting with ***Learn Containers 5 Minutes at a Time***, which is designed to lead directly into this book and is freely available from Mirantis Press at:

https://www.mirantis.com/resources/mirantis-container-5-minute/

We will be using Minikube for a local Kubernetes environment. It runs on Linux, Mac, or Windows. This edition was written using Minikube v1.26.1 and Kubernetes v1.24.3. The vast majority of exercises should work with other standard Kubernetes implementations, but your mileage may vary.

Our primary development language in this book will be JavaScript (Node.js), and while an understanding of JavaScript will be helpful, it is not required.

Conventions

Commands to be executed in the terminal will be indented, rendered in the `Consolas` typeface, and preceded by a bold `%` as below:

```
% minikube version
```

Terminal output will be rendered in white `Consolas` against a black background as below:

```
minikube version: v1.26.1
commit: 62e108c3dfdec8029a890ad6d8ef96b6461426dc
```

Throughout the chapters that follow, I will refer to a standard project directory tree that mirrors the one on the GitHub project for this book: https://github.com/ericgregory/kube-5mins

In each chapter, I will prompt you to create directories and files as they are necessary. If you would like to create your project directories up front, the tree is below. The project root, `kube-5mins`, may be wherever you wish on your system.

```
kube-5mins
├── 04-meet
├── 05-deployments
├── 06-services
├── 07-services-2
├── 08-storage
├── 09-secrets
├── 10-statefulsets
└── 11-scale
```

I hope you find this book useful for getting started with Kubernetes. Let's begin!

Table of Contents

Chapter 1
What is Kubernetes?

IN THIS CHAPTER...

— Understand the purpose of container orchestration
— Explain the function and history of Kubernetes

Before the advent of containerization, organizations deployed applications from either physical or virtualized servers, typically located either in the application owner's data center or the facilities of a hosting provider.

Traditional deployment from a **physical server** could be wasteful and slow to provision. If the application needed to scale up for increases in traffic, new machines and load balancing mechanisms would have to be set up manually—and often wouldn't use those compute resources efficiently, leaving processing power and storage paid for but unused.

The rise of **virtualization** went some way toward solving these efficiency and provisioning problems by simulating the full stack of application dependencies—from software libraries all the way down to the operating system kernel. Virtual machines (VMs) can be quickly replicated like any other piece of software, and multiple VMs can cohabitate on the same physical computer.

For some deployments, virtualization is an excellent solution. But for applications operating at scale, VMs can be sub-optimal in terms of both resource utilization and scalability. The problem lies in the *anatomy* of a virtual machine: every single instance of the VM is simulating the entire software stack, including the operating system. As you add more and more instances, that turns into a lot of duplicated effort—and spinning up an entire operating system every time you provision a new instance inevitably slows down the process.

By duplicating only the *unique* dependencies of a given application, **containers** provide a more svelte solution. Deploying containerized apps at scale can provide significant benefits:

- **Resource efficiency:** We can fit and run more containerized applications than comparable VMs on a given physical machine, making them more cost-effective.
- **Scalability:** Since containers are so relatively lightweight, they're faster to provision.
- **Resiliency:** If an instance crashes, the speed and efficiency of containers make it easier to immediately spin up a new instance.

Good news all around! But deploying complicated, large-scale containerized apps to production across many hosts poses

serious challenges. How do we coordinate tasks across all of those containers? How can containers across many different hosts communicate with one another? These challenges have given rise to a new class of software systems: **container orchestrators**.

What is Kubernetes?

Container orchestrators provide a unified, automated solution for networking, scaling, provisioning, maintenance, and many other tasks involved in deployment of containers. There are several notable container orchestrators with varying use-cases:

- **Swarm**: An orchestrator directly integrated with the Docker Engine and well-suited to smaller clusters where security is a top priority.
- **Apache Mesos**: An older orchestrator of both containerized and non-containerized workloads, originally developed at UC Berkeley and sometimes used for workloads centered around big data.
- **Kubernetes**: As of this writing, the industry's most widely-used container orchestrator is unquestionably the open source Kubernetes project that emerged from Google in 2014. Originally based on Google's internal **Borg** cluster manager, Kubernetes has been maintained under the auspices of the Cloud Native Computing Foundation (CNCF) since 2015.

Pronounced (roughly) "**koob-er-net-ees**" and sometimes shortened to **k8s** ("kay-eights", or "kates"), the project's name derives from the Ancient Greek κυβερνήτης, which means "captain" or "navigator."

If we understand software containers through the metaphor of a shipping container, then we can think of Kubernetes as the captain that guides those containers where they need to go.

That metaphor is illustrative, but incomplete. Kubernetes is more than a top-down giver of orders—it's also the substrate (or "underlying layer") that containerized applications run on. In this sense, Kubernetes isn't merely the ship's captain but *the ship itself*, a self-contained and potentially massive ship like an aircraft carrier or the starship *Enterprise*—a floating world that might host all kinds of activity.

An organization might run dozens or hundreds of applications or services on a Kubernetes **cluster**—meaning a set of one or more physical or virtual machines gathered into an organized software substrate by Kubernetes.

What's more, the system is designed to run on many different cloud providers, giving applications that run on Kubernetes a high degree of portability.

In these respects, Kubernetes is sometimes understood as an **operating system for the cloud**.

Who runs Kubernetes?

Kubernetes is designed to run enterprise applications at scale, and it has achieved widespread adoption for this purpose. The Cloud Native Computing Foundation's 2021 Cloud Native Survey[1] found that 96% of organizations among its respondents were either using or evaluating Kubernetes.

Within those organizations, there are generally two personas who interact with Kubernetes: **developers** and **operators**. These two roles have different priorities, needs, and usage patterns when it comes to Kubernetes:

- **Developers** need to be able to access resources on a Kubernetes cluster, build applications for a Kubernetes environment, and run those applications on the cluster.
- **Operators** need to be able to manage and monitor clusters and all of their attendant infrastructure, including storage and networking considerations.

There is some overlap between the skills and knowledge that each of these personas will require (and indeed, under some circumstances the same person might fulfill both roles)—developers and operators alike need a basic understanding of the anatomy of a Kubernetes cluster, for example.

[1] https://www.cncf.io/announcements/2022/02/10/cncf-sees-record-kubernetes-and-container-adoption-in-2021-cloud-native-survey/

Broadly speaking, however, this book will focus on what **developers** should know to use Kubernetes successfully. In the lessons that follow, we will learn how Kubernetes works, how to build apps for deployment on Kubernetes, and how to streamline the developer experience. By the end, we will have deployed multiple complex apps to our own cluster.

Our five minutes are up for today—next time, we'll get a Kubernetes cluster up and running.

Chapter 2
Setting Up a Kubernetes Learning Environment

IN THIS CHAPTER...

– Start a local cluster with Minikube
– Access your cluster with kubectl, Lens, and a web dashboard

A Kubernetes cluster can run in a wide variety of configurations according to its use case. For learning purposes, we're going to start with a **single-node cluster** on our local machine.

In the "real world," clusters will typically spread across multiple nodes (or machines), each of which will have dedicated roles like managing the cluster or running workloads. In *this* case, everything will happen on a single node. That will give us a relatively straightforward development environment in which we can figure out the fundamentals, before progressing to more complex configurations.

The first decision we need to make is which version of Kubernetes to install. Because Kubernetes is an open source system, it has given rise to a variety of alternative **distributions** with their own particular use-cases, just as the Linux kernel is the foundation of numerous Linux distributions. The **k0s** (pronounced "kay-zeros") project, for example, is a distribution designed for maximum resource efficiency, so it might run (and

scale) anywhere from Raspberry Pis to powerful servers in public or private clouds. The creators of Kubernetes distributions usually try to maintain full compatibility with "upstream" Kubernetes—the original, baseline project—so that users can utilize the full suite of open source tooling developed by the community.

To get started, we're going to use a distribution called **Minikube**, which is designed specifically for learning Kubernetes.

Installing Minikube

Minikube requires about 2GB of RAM to run, and it wants 20GB of disk space, which is the amount it assigns for cluster usage by default. (You can configure it to use as little as 2GB if you're short on space.) It works on macOS, Linux, or Windows, and depending on its configuration, it runs on a container, a virtual machine, a combination of the two, or bare metal. Most people will be running it through VMs, containers, or both. If you have Docker Desktop or Docker Engine installed, then you're ready to get started. (If not, install Docker Desktop[2] and return here.)

On an x86-64 machine running macOS with the Homebrew Package Manager, installation is a simple terminal command:

```
% brew install minikube
```

[2] https://www.docker.com/products/docker-desktop/

Users with other CPU architectures and operating systems will want to consult the official installation instructions[3] to download the version that is right for them.

Once Minikube is installed, make sure Docker is running, and then run the following command in the terminal:

```
% minikube start
```

You should see some friendly, emoji-illustrated status updates confirming that the system is running:

```
👍  Starting control plane node minikube in cluster
minikube
🚜  Pulling base image ...
🔄  Restarting existing docker container for "minikube" ...
🐳  Preparing Kubernetes v1.23.1 on Docker 20.10.12 ...
    ▪ kubelet.housekeeping-interval=5m
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
    ▪ Using image kubernetesui/dashboard:v2.3.1
    ▪ Using image
k8s.gcr.io/metrics-server/metrics-server:v0.4.2
    ▪ Using image kubernetesui/metrics-scraper:v1.0.7
🌟  Enabled addons: storage-provisioner, metrics-server,
default-storageclass, dashboard
🏄  Done! kubectl is now configured to use "minikube"
cluster and "default" namespace by default
```

[3] https://minikube.sigs.k8s.io/docs/start/

We can confirm that Kubernetes is running with another command:

```
% minikube kubectl get nodes
```

This will list the nodes associated with our Kubernetes cluster. We should get a result that looks something like this:

```
NAME       STATUS   ROLES                  AGE     VERSION
minikube   Ready    control-plane,master   5m26s   v1.23.1
```

All right—we have one node, and that makes sense, because we said that this would be a single-node cluster. But what's `kubectl`, and why am I using it?

For the answer, we need to break down an important element of Kubernetes development: **cluster access**.

Exercise: Three ways of looking at a cluster

Users—and software agents—ultimately conduct any interaction with a cluster through the Kubernetes application programming interface (API). But many of those interactions are mediated by other applications that consume the API.

Three of the most commonly used applications for accessing and interacting with the cluster are:

- **kubectl**: A command-line interface (CLI) for managing Kubernetes clusters; comes pre-packaged with most distributions. The proper pronunciation of this tool is a topic of spirited disagreement; I favor "koob-control," but plenty of people say "koob-C-T-L" or (heartwarmingly) "koob-cuddle."

- **The Kubernetes Web Dashboard**: A web-based graphical user interface (GUI) for interacting with clusters; comes pre-packaged with most distributions. Note that while it is suitable for learning purposes, the web dashboard can be a source of security vulnerabilities—Tesla, for example, had a cluster hijacked after the web dashboard was configured with elevated privileges and exposed to the Internet.

- **Lens**: A third-party, open source integrated development environment (IDE) for Kubernetes—supported by Mirantis—that enables users to interact with clusters at a granular level via a graphical user interface.

None of these three tools are bound to a given cluster. You can use any of them to interact with multiple clusters from the same machine. But for now, we're only concerned with our Minikube cluster.

To get a feel for these three methods of interacting with a cluster, we're going to launch a service—essentially, an abstraction for the delivery of a particular piece of software

functionality, as we saw in the last unit. In this case, our service will be an NGINX web server. Next, we're going to observe our service with each of our three methods of cluster access. Don't worry too much about the specifics of how we're launching our service for the time being—in the coming lessons, we'll examine that process in great detail. For now, we just want to get our feet wet interacting with a cluster.

A note on kubectl

Before we get started, we should briefly discuss a few of the ways you can run kubectl. As we've seen, you already have an installation of kubectl on your system—this is built into Minikube, and you need to preface the `kubectl` command with `minikube` if you want to use it. You're welcome to keep on doing this.

Alternatively, you can use an **alias**[4] or **symbolic link**[5] to map the command `minikube kubectl` to the more concise `kubectl`. Or if you prefer, you can install a separate instance of the tool[6] which you will activate with the command `kubectl`. The choice is yours—but note that we'll be using the simple command `kubectl` from here on out, so if you decide to use the Minikube version without creating an alias or symlink, you'll need to add `minikube` to the beginning of these commands.

In the terminal, enter the following command:

```
% kubectl create deployment nginx-test
--image=nginx --port=80
```

[4] https://en.wikipedia.org/wiki/Alias_(command)
[5] https://en.wikipedia.org/wiki/Symbolic_link
[6] https://kubernetes.io/docs/tasks/tools/

At this point, you probably have a pretty good guess as to what this is doing! We're creating a deployment of an application—NGINX—based on the official NGINX container image, and we're specifying that the container exposes port 80.

Now enter:

```
% kubectl expose deployment nginx-test
--type=NodePort --port=80
```

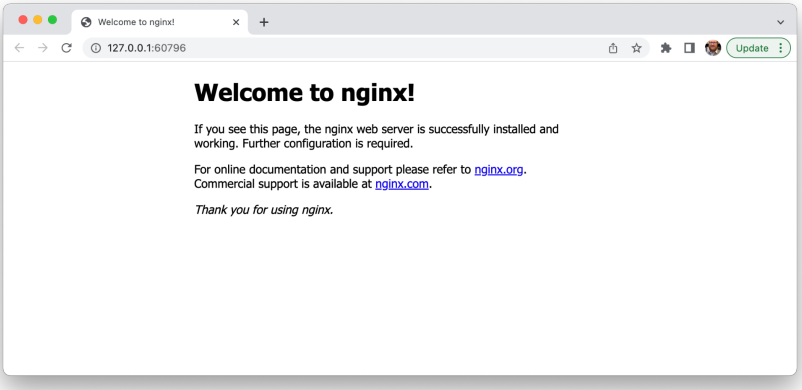
This creates a service from the deployment model we defined above; entering the command should return the result:

```
service/nginx-test exposed
```

Next enter:

```
% minikube service nginx-test
```

Minikube will create a network "tunnel" to make the new service accessible on our local machine, and automatically open the service in our web browser. We should see the NGINX welcome page here. (Note that your IP address and port may differ from the ones in the screenshot below.)



Stop the Minikube tunnel by pressing CTRL+C in the terminal. Even though we've stopped the tunnel, the service is still running in the cluster. We can get some basic information about it using kubectl:

```
% kubectl get services
```

If you want to save a few keystrokes, you can use `svc` instead of `services`. Either way, this should return something like the following, albeit with different IP addresses, and a different local port number:

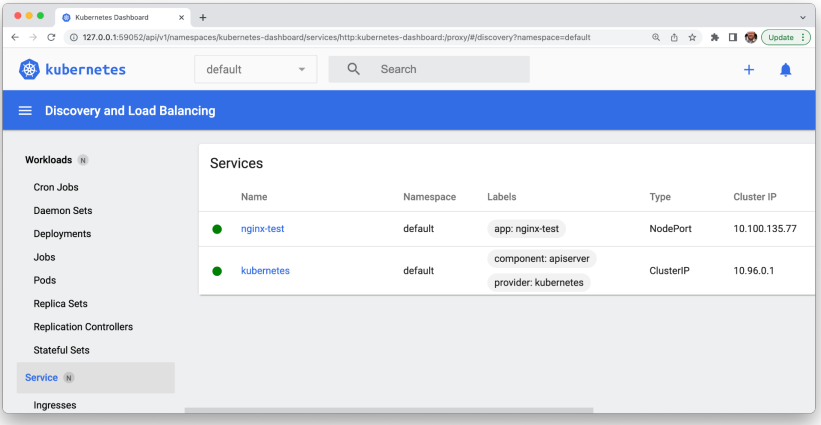
```
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP
nginx-test   NodePort    10.100.135.77   <none>        80:30218/TCP
```

Great. We see our `nginx-test` service, as well as the Kubernetes service itself.

We can also observe our services through the Kubernetes Web Dashboard. Minikube makes it easy to launch the dashboard from the command line—just enter:

```
% minikube dashboard
```

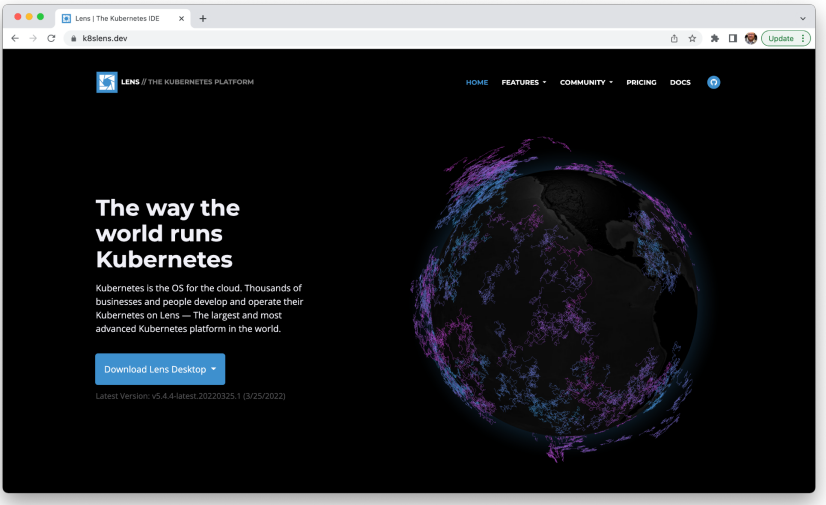
This command will start another process that needs to run continuously in the terminal. In the web browser, click on the **Service** tab in the left sidebar.



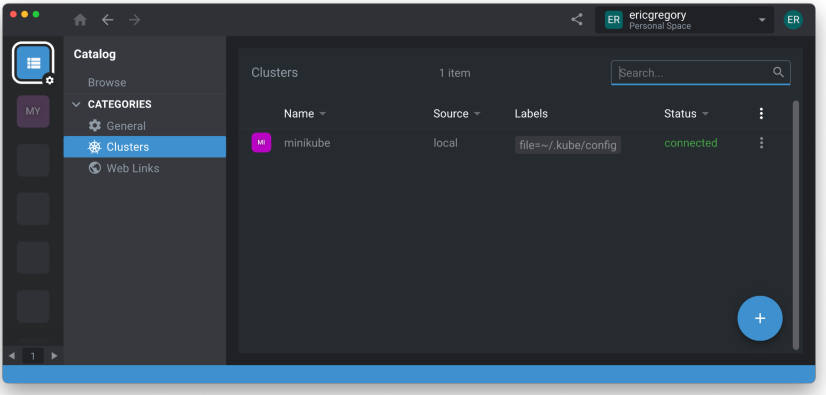
Here we can observe our services through a GUI.

Feel free to click around the Web Dashboard and explore. When you're finished, we'll observe our services from one more perspective.

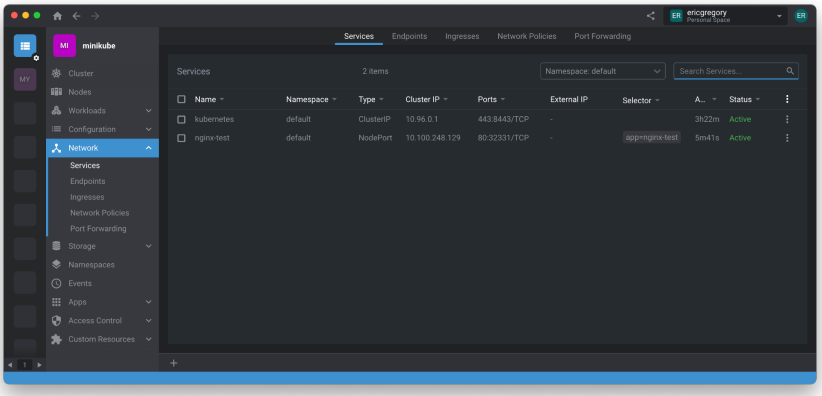
Navigate to k8slens.dev and download **Lens Desktop**.



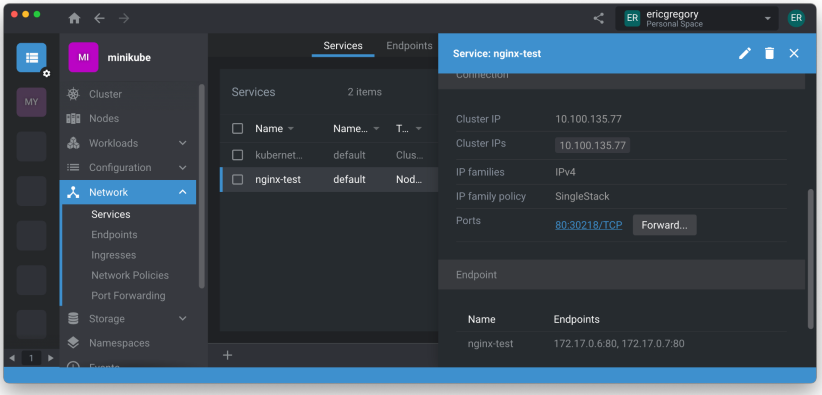
Run through the installer and account creation and open the application. It will automatically detect your Minikube cluster—click on “Catalog” in the left sidebar and then on the Minikube cluster to connect it to Lens.



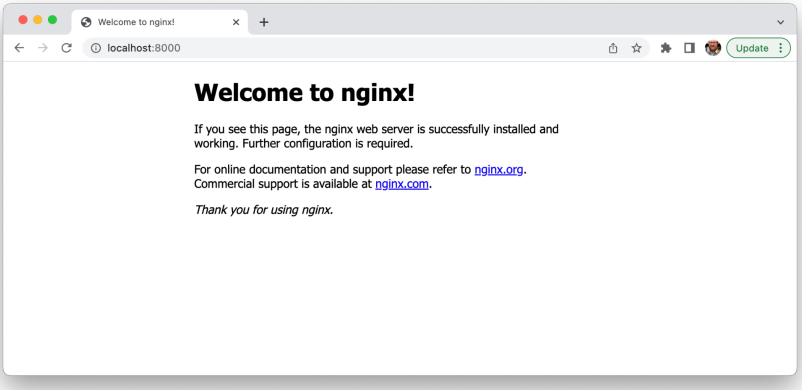
Once connected, open the Network dropdown from the left sidebar and click on Services.



From here, you can click on the `nginx-test` service for more information.



You can click the **Forward** button to forward the container port to a port on your host machine—accomplishing roughly the same thing as the Minikube tunnel, but with a technique that can be generalized to any Kubernetes service…not just the ones running in a local Minikube environment.



Whew—we've done quite a bit today! We've installed Minikube and Lens, and we've explored three different ways to interact with a cluster. Before we finish, let's clean up. In the terminal, press CTRL+C to stop the dashboard process, then:

```
% kubectl delete service nginx-test
% kubectl delete deploy nginx-test
% minikube stop
```

```
✋  Stopping node "minikube"  ...
🛑  Powering off "minikube" via SSH ...
🛑  1 node stopped.
```

That's it for today. Next time, we'll break down the anatomy of a Kubernetes cluster.

Chapter 3
The Anatomy of a Kubernetes Cluster

IN THIS CHAPTER...

– Understand the core components of a cluster
– Explain how those components work together

A Kubernetes cluster is made up of **nodes**. A node is simply a compute resource: it can be either a virtual or physical computer.

Last time, we noted that our Minikube cluster is a single-node cluster, meaning that it consists of only one machine—typically a containerized or virtual machine running on our laptop or desktop.

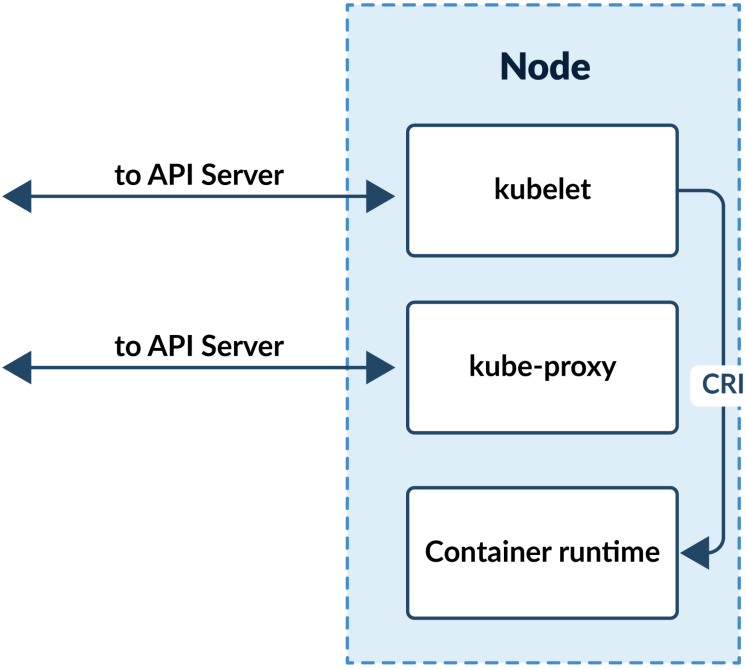
Any node might need to step up and run workloads at any time, so the components required to make that happen are found on every node. Those components include:

- **A container runtime**: This is the component that actually runs containers on a node. Today, Kubernetes supports a variety of runtimes, including containerd, CRI-O, Mirantis Container Runtime, and others.
- **kubelet**: A kubelet runs on every node and mediates between the control plane and the node in question, passing instructions to the container runtime—"Start

this workload!"—and then monitoring the status of the job. In order to facilitate the use of different runtimes, the kubelet communicates with the container runtime via a standardized Container Runtime Interface (CRI).

- **kube-proxy**: Containerized workloads will need to be able to communicate with one another and the world outside the cluster. In other words, we're going to have a lot of complicated networking going on! The kube-proxy component is a given node's on-site network expert, managing network rules and connections on the node.

We can visualize the fundamental components of a node like this:



Nodes in Kubernetes clusters are assigned roles. Start your Minikube cluster and then run the following command:

```
% kubectl get nodes
```

You should see one node, named `minikube`, because this is a single-node cluster.

```
NAME       STATUS   ROLES                  AGE     VERSION
minikube   Ready    control-plane,master   99s     v1.23.1
```

Note the roles listed here. The `minikube` node is fulfilling the roles of `control-plane` and `master`. What does this mean?

The **control plane** coordinates the assignment of workloads—meaning containerized applications and services—to other nodes. If there are no other nodes in the cluster, as in our case, then the control plane node has to roll up its sleeves and get its hands dirty.

In a real-world, production-grade Kubernetes deployment, you will almost certainly have many nodes dedicated to running workloads—these are sometimes called **worker nodes**. Additionally, you will likely have multiple nodes in the control plane. (We'll discuss this and other architectural patterns shortly.)

In such a scenario, the control plane is dedicated to ensuring that the current state of the cluster matches a pre-defined **specification** (or **spec**)—the *desired* state of the cluster. If our specification says that an instance of NGINX should be running, and the control plane learns that this isn't the case, it sets out to assign the job and resolve the discrepancy.

A note on terminology

So what about the `master` role? At the time of writing (as of release 1.24), it is interchangeable with `control-plane`. "Master" was the original term and is currently retained to maintain integrations between components that use the label, but the Kubernetes project plans to phase out the use of the word in this context. Eventually, the `get nodes` command above will only return `control-plane` for the node role.

This change reflects an ongoing conversation about the suitability of the term "master" in the tech industry and broader culture; many technologies are moving away from the use of "master" terminology just as they moved away from the use of "slave" metaphors several years ago. GitHub, for example, now encourages defining a "main" branch rather than a "master" branch.

The important thing to know when learning Kubernetes is that many third-party resources will discuss `master` nodes, though the term is no longer used in the Kubernetes docs. These older resources are referring to nodes with the `control-plane` role, which we can think of as members of a cluster's leadership committee. From here on out, we'll use "control plane" to refer to this role.

Ultimately, the control plane is a group of components that work together to make the actual state of the cluster match its spec. It responds to unexpected events out there in the real world (like a service going down), and it takes onboard any changes to the spec and assigns resources accordingly.

So what are these control plane components?

- **kube-apiserver**: Last lesson, we said that every interaction with the cluster from tools like kubectl or Lens is mediated by the Kubernetes API. The **API server** exposes that API. The API server is the hub at the center of activity on the cluster, and the only control plane component that communicates directly with nodes running workloads. It also mediates communications between the various elements of the control plane.
- **kube-scheduler**: Put simply, the scheduler assigns work. When we tell the API server that we want to run a workload, the API server passes that request to the scheduler. The scheduler, in turn, consults with the API server to check the cluster status and then matches the new job to a node that meets the relevant resource requirements—whether that means hardware specifications or policy constraints.
- **etcd**: By default, the Kubernetes cluster runs on an assumption of ephemerality: any node and any workload may be spun up or stopped at any moment and replaced with an identical instance. The processes

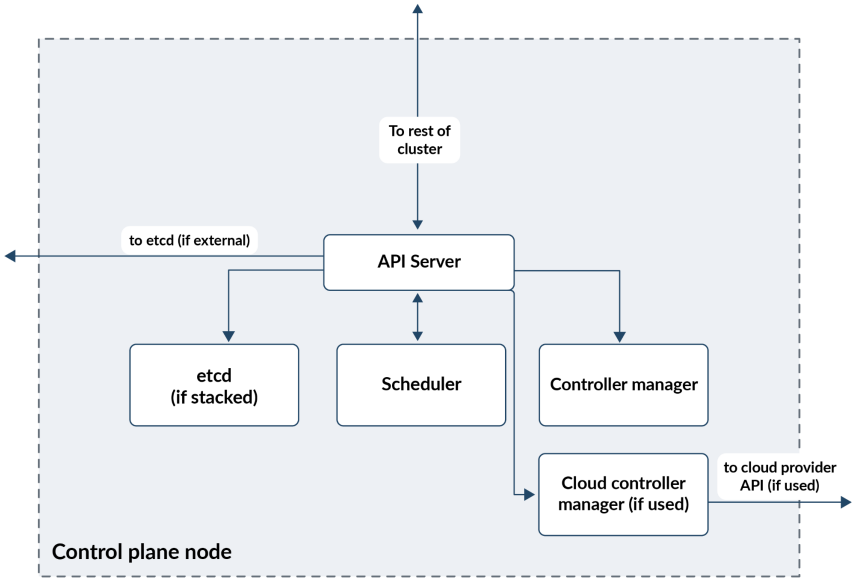
in the cluster are meant to match a specification, but they're not expected to persist. This is a philosophy of **statelessness**, and it resembles the operational assumptions of containers themselves. But even if you're running entirely stateless apps on your cluster, you will need to save and persist configuration and state data for the cluster, and etcd is the default means of doing so. It's a key-value store that may be located inside the cluster *or* externally. Only the API server communicates directly with etcd. Note that etcd may be used outside the context of Kubernetes as well; it's ultimately designed to serve as a distributed data store for relatively small amounts of data.

- **kube-controller-manager** and **cloud-controller-manager** are the watchdogs: they constantly monitor the state of the cluster, compare it to the spec, and initiate changes if the state and spec don't match. The **kube-controller-manager** focuses on the upkeep of workloads and resources within the cluster, while the **cloud-controller-manager** is essentially a translator who speaks the language of your cloud provider—dedicated to coordinating resources from your cloud infrastructure. The cloud-controller-manager is an optional component designed for clusters deployed on the cloud.

It's tempting to think of the control plane as one or more "boss" nodes—that's *essentially* true, but it's a bit of a simplification that obscures important details. For example, etcd may be

integrated into a control plane node *or* hosted externally. And as we've seen, a control plane node includes all of the components of a worker node. Typically, components of the control plane are initially installed on a single machine that doesn't run container workloads…but that's not a necessary or optimal configuration for every situation.

We can visualize a control plane node like this:



Kubernetes is a distributed system designed in large part for resiliency. But as you can imagine, the control plane *could* become a single point of failure, especially if it is consolidated on one machine. For this reason, the system offers a "High Availability" configuration, which is the recommended approach for production environments. A High Availability

(HA) cluster will include multiple replicas of the control plane node.

While one of the control plane nodes is the primary cluster manager, the replicas work constantly to remain in sync—and a load balancer can apportion traffic between the various replicas of the API server as needed. Similarly, if duplicates of etcd are sitting on multiple control plane nodes (or an external host), they form a collective that runs its own consensus mechanism to ensure uniformity between the members. These measures ensure that if a control plane node goes down, there are replicas available to pick up the slack.

Kubernetes architectural patterns

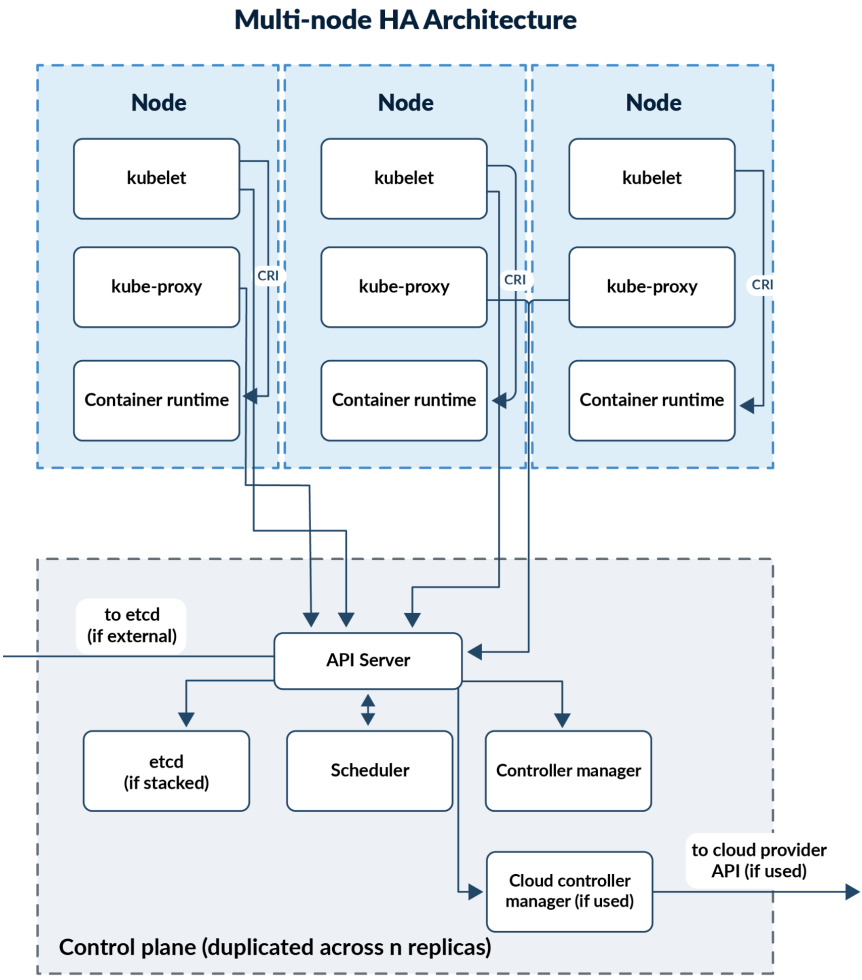
Before we move on, let's pause and take stock:

- A cluster is made up of one or more nodes.
- Any node can run container workloads.
- At least one node also has the components and role to act as the control plane and manage the cluster.
- A cluster configured for High Availability (that is, production-grade resilience) includes replicas of all control plane components.

Now, with all of these takeaways in place, let's look at the big picture. What are the major architectural patterns for a Kubernetes cluster?

- **Single node**: This pattern is what we have running right now via Minikube—a single cluster that fulfills the role of the control plane and runs any workloads we wish to run. This node walks and chews bubblegum at the same time: great for learning and development, but not suited for most production uses.
- **Multiple nodes with a single node control plane**: If we were to connect another node to our cluster as a dedicated worker node, then we would have a cluster with clear separation of responsibilities…but also a single point of failure in the control plane node.
- **Multiple nodes with High Availability**: When control plane components are replicated across multiple nodes, a cluster can achieve high levels of resiliency through strategic redundancy.

For each of the models above, there are two major sub-variants: you might include the etcd data store on the control plane node(s)—a model called “stacked etcd”—or you might host etcd externally. High Availability configurations are possible with both of these approaches to the data store.



As we've established, we're currently working with a single-node cluster. Let's get a little more information about that node and see how it correlates with what we've learned about Kubernetes architecture:

```
% kubectl describe node minikube
```

Read through the output and look for references to the components we've discussed. A few sections to note:

```
…
System Info:
  Machine ID:
8de776e053e140d6a14c2d2def3d6bb8
  System UUID:
45bc9478-1425-44bd-a441-bfe282c0ce3e
  Boot ID:
fbdac8bf-80eb-4939-86f1-fcf792c9468f
  Kernel Version:             5.10.76-linuxkit
  OS Image:                   Ubuntu 20.04.2 LTS
  Operating System:           linux
  Architecture:               amd64
  Container Runtime Version:  docker://20.10.12
  Kubelet Version:            v1.23.1
  Kube-Proxy Version:         v1.23.1
…
```

My Minikube cluster is running on a container, so the information on my system refers to those details. Here we also find version numbers for our universal node components like the container runtime, kubelet, and kube-proxy. Further down, we can find information on node resource usage:

```
  …
  kube-system    etcd-minikube
  kube-system    kube-apiserver-minikube
  kube-system    kube-controller-manager-minikube
  kube-system    kube-proxy-9rjzk
  kube-system    kube-scheduler-minikube
  …
```

Here we find a number of familiar faces: our instances of the API server, etcd, the scheduler, and the controller manager—the whole control plane team—as well as kube-proxy.

Now that we understand the architecture of a Kubernetes cluster, next time we'll take a look at some of the core abstractions that govern activity on the cluster, and which we can interact with through the Kubernetes API.

Chapter 4
Meet the Kubernetes API

IN THIS CHAPTER...

– Explore the Kubernetes API
– Understand and work with Pods

All of our interactions with Kubernetes run through the API server. That means we need to speak its language—to understand the abstractions that govern activity on the cluster.

If you're already experienced with RESTful API development, the terminology and fundamentals of the API should feel fairly comfortable—but if not, we should briefly define some fundamentals.

REST refers to "**Re**presentational **S**tate **T**ransfer." It refers to a pattern in which different software components communicate with one another—typically online—through a defined interface, using standard web protocols like HTTP as a medium to carry data payloads in formats like JSON.

In RESTful thinking, a **resource** is an endpoint—a URI—to which one may send a request and receive a predictable response according to the design of the API in question. Kubernetes expresses its core abstractions as **resource types**:

fundamental concepts that can be instantiated and defined on the cluster. If we had a resource type called a **Deployment** (spoiler: we do), and we created a specific manifestation of that resource type with defined characteristics, we would call that a **resource instance**.

Just think of resource *types* as "theory" and resource *instances* as "practice"—or for the philosophically inclined, resource types are Platonic forms, and instances are all the myriad manifestations of those abstracts in the physical world. But remember that ultimately, both the resource type and the resource instance are URI-accessible endpoints on the cluster.

Fun with semantics

The terminology can get a little messy—out in the real world, people will use "resource" to refer to both resource types and resource instances. We'll do the same thing in this book. But for the sake of clarity, we will adopt the convention of **capitalizing** the names of resources when we are talking about a **resource type**, and using **lowercase** when we are referring to an **instance** of that resource.

For example:

"This is important because the **Pod** is ephemeral." (*Here we are describing a characteristic of the resource type "Pod."*)

"We now have three **pods** on the cluster." (*In this case, we are talking about multiple instances of the resource.*)

Let's get a little more concrete. What kinds of resource types are we talking about here? We can use kubectl to retrieve a complete list. Start Minikube and then enter:

```
% kubectl api-resources
```

Here's an excerpt from the `NAME` column of the output:

```
NAME
…
namespaces
nodes
persistentvolumeclaims
persistentvolumes
pods
podtemplates
replicationcontrollers
resourcequotas
secrets
serviceaccounts
services
mutatingwebhookconfigurations
validatingwebhookconfigurations
customresourcedefinitions
apiservices
controllerrevisions
daemonsets
deployments
replicasets
statefulsets
…
```

By the end of this book, you will have a solid grounding in many of these resources. But as you can see in the terminal output,

the Kubernetes API is sprawling; in this book, we'll focus on the most essential Kubernetes resource types for developers.

Let's start with Kubernetes' elementary particle: the Pod.

What is a Pod?

We can ask Kubernetes itself. In the terminal, enter:

```
% kubectl explain pod
```

You should get the following answer:

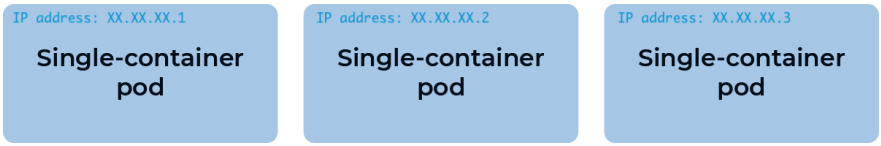
```
KIND:     Pod
VERSION:  v1

DESCRIPTION:
   Pod is a collection of containers that can run on a
   host. This resource is created by clients and scheduled
   onto hosts.
…
```

In other words, a **Pod** is a unit of one or more containers that share a network namespace and storage volumes while running on a Kubernetes node. It is also Kubernetes' most granular unit of scheduling for a containerized workload.

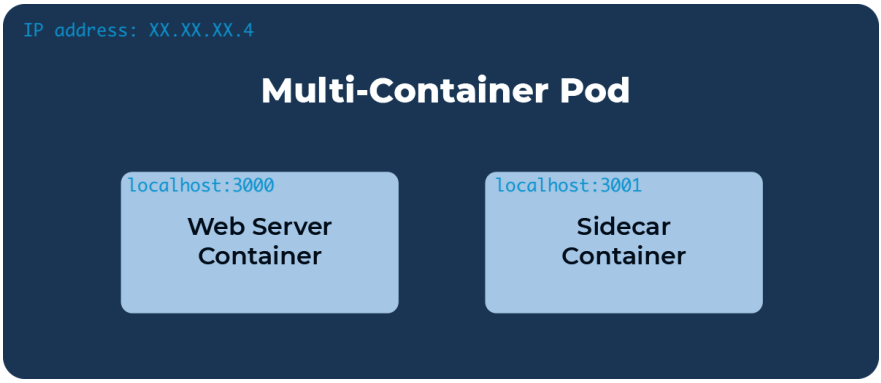
Up until now, we've used the general term "workload" to refer to the containerized processes that Kubernetes is orchestrating. As we've seen previously, the system assigns these workloads to worker nodes, the physical or virtual

machines that are part of the cluster. These are the "hosts" in the definition above. But Kubernetes adds a conceptual layer *separating* the node from the container, and that's the Pod.

Each pod is assigned a unique IP address, so containers *within* the pod can communicate with one another over localhost, and pods can communicate with one another at fixed addresses.



These two ideas—the abstraction of the Pod, and each pod having its own IP address—are fundamental, defining concepts for Kubernetes. They extend all the way back to the Borg project at Google, and they distinguish Kubernetes from other container orchestrators like Swarm. It is difficult to overstate the importance of the Pod; it contributes to the high scalability of Kubernetes, and it informs many of the networking patterns we will encounter going forward.



We've said that a pod consists of "one or more containers." The conventional wisdom on containers-per-pod is that **a given pod should include only one container if possible**, and multiple containers only when they are tightly coupled. Sometimes, your Kubernetes configuration may necessitate a multi-container pod—when using an Istio service mesh, for example, each pod will include a "sidecar" container in addition to the primary container. But we wouldn't expect to see more than two or three containers in most pods. (We'll talk more about service meshes later, but if you're ready to dive into the deep end now, you can check out *Service Mesh for Mere Mortals*[7].)

Testing the limits

"Sure," I hear you say, "I understand that we *should* run no more than a handful of containers per pod. But I'm a maverick! I want to test the limits. How many containers *can* we run in a pod?"

Short answer: There's no defined limit. As of version 1.23, the Kubernetes docs[8] note that Kubernetes is designed for a maximum of 300,000 total containers. *Technically*, you could cram all of them into one pod (assuming you could satisfy the underlying compute and storage requirements). But the docs also tell us that the system is designed for no more than 150,000 pods, and that ratio is instructive. Operating at the extremities of scale, we would have two containers per pod.

[7] https://info.mirantis.com/l/530892/2021-09-23/syxbh6
[8] https://kubernetes.io/docs/setup/best-practices/cluster-large/

By separating out the functionality of your containerized services into minimal viable units, you can scale those units—those services or microservices—independently. Say, for example, that you have a pod representing the core functionality of the auth service on your web app, and you suddenly experience a barrage of sign-ups. The cluster can scale up your auth service as needed without needlessly duplicating other, unrelated elements of the app.

Creating our first pods

We can launch a pod with a line in kubectl:

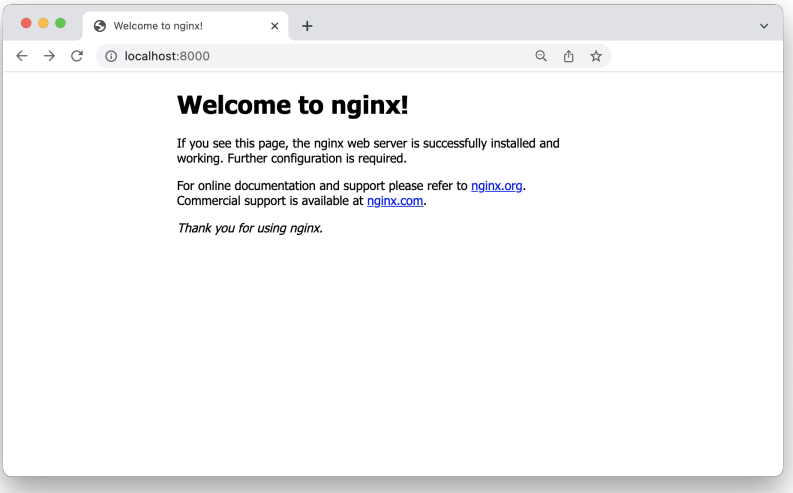
```
% kubectl run nginx --image=nginx
```

Here, we're starting a new pod named `nginx` based on the latest NGINX image in Docker Hub. We can see our pod running with…

```
% kubectl get pods
```

Suppose we want to see what our pod is serving. NGINX is running on port 80 within the pod. We can forward that port to localhost:8000 on our local machine with…

```
% kubectl port-forward pod/nginx 8000:80
```



Press CTRL-C to stop the port-forwarding process.

This may remind you of working with Docker—we have a powerful command line tool that allows us to spin up containerized apps quickly. But as with Docker, we need a way to define, store, and re-use operations more complicated than launching a simple NGINX container.

In Kubernetes, we can define the attributes of an object we intend to include in our cluster with a **manifest** in **YAML** format. (YAML stands for Yet Another Markup Language or YAML Ain't Markup Language, depending on who you ask.) YAML files use the .yml or .yaml file extensions, and a very simple one might look like this…

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: nginx-2
 labels:
   role: app
spec:
 containers:
   - name: web-server
     image: nginx
```

The YAML file is a list of key-value pairs. Often, values are nested objects indicated by hierarchical indentation. For example, the value for `metadata` above is a nested object containing two keys: `name` and `labels`.

Let's walk through this manifest:

- First, we establish the **API version** endpoint we're using.
- Next we note the **kind** of object we want to define: a Pod.
- Then we establish some **metadata**: the name of this pod object and labels that will help us refer to it later.
- Finally, the **spec** establishes that we want a pod running the NGINX web server.

These four top-level fields are required for every Kubernetes object manifest. When you submit the manifest to the API

server, it will cue the scheduler to find an appropriate node and then instruct the kubelet on that node to spin up a pod including an `nginx` container. The container runtime will use the image of the specified name found in the system's configured image registry. (Your Minikube setup uses Docker Hub.)

Create a directory for this chapter called `/04-meet/` and copy the YAML manifest above into a file named `pod.yml`. Navigate to `/04-meet/`, and you can launch a pod based on the manifest with…

```
% kubectl apply -f pod.yml
```

When you run `kubectl get pods` again, you should see both of the pods you've launched. Try port-forwarding the NGINX process from the `nginx-2` pod. Then delete the running pods:

```
% kubectl delete pod nginx
% kubectl delete pod nginx-2
```

JSON is YAML

YAML is a superset of JavaScript Object Notation (or JSON), which means that valid JSON can be included in YAML manifests—and indeed, a file full of JSON with the .yml or .yaml extension can *be* a valid YAML manifest.

If you're not familiar with JSON, don't worry about it for now—but if you are, it can provide some useful insight into what's happening in your YAML files. For example, the first line of the YAML manifest we just saw could be written:

```
{“apiVersion”: “v1”}
```

So why use YAML rather than JSON, especially given that the Kubernetes API server “speaks” JSON? Some people find YAML more readable, but the better answer is that YAML supports comments, which are as important in our manifests as in our code.

In review…

We’ve covered a lot of ground in this lesson:

- Kubernetes resources
- The Pod resource type
- YAML manifests

Finally, we got some practice working with individual pods. But if we want to *efficiently* create a service at scale, we don’t want to do it atom by atom. In practice, we don’t want to interact with individual pods very often at all, since they’re ephemeral by design. Instead, we need an abstraction to define our intent at a higher level.

Fortunately, Kubernetes gives us a resource to do just that: the Deployment. In the next lesson, we’ll discuss this essential tool for creating and managing groups of pods.

Chapter 5
Managing Workloads with Deployments

IN THIS CHAPTER...

– Understand the Deployment resource type
– Use a deployment to manage pods

In practice, you don't want to directly manage pods. Containers (and the pods that contain them) are meant to be ephemeral, and if you're working on Kubernetes, you're working at scale.

It is much more common to interact directly with a **Deployment**: a grouping of functionally identical pods that can be launched and managed together. These duplicate pods allow you to achieve the resiliency and scalability that Kubernetes is designed to foster: traffic may be balanced between them as needed, and if a pod goes down, it can be immediately replaced.

The Deployment resource type draws upon the Pod as well as another important abstraction: the **ReplicaSet**. This defines the number of duplicate pods the system should maintain—so if you specify that you intend for three replicas of an NGINX pod to be running, Kubernetes will work to ensure that this is the case. If a pod goes down, the system will start a new pod to meet the expectation of three replicas. A ReplicaSet is also used as a reference point by the **Horizontal Pod Autoscaler**

(HPA) when defining minimum and maximum numbers of replicas to run. (We'll talk more about the HPA in the final chapters of this book.)

It is perfectly possible to write a manifest for a ReplicaSet, just as it is possible to write a manifest for a Pod—but the beauty of the Deployment is that it joins those two resource types so you can create, delete, update, replicate, and otherwise manage groupings of pods through a single interface.

You might recall that we created a deployment with a kubectl command in our lesson on setting up a Kubernetes environment:

```
% kubectl create deployment nginx --image=nginx
--port=80
```

There, we issued our spec directly through kubectl. That works for a simple deployment, but for more complicated configurations, we may want to use a manifest. Let's have a look at a Deployment manifest. We can ask kubectl to give us a template:

```
% kubectl create deployment nginx --image=nginx
--port=80 -o yaml --dry-run=client
```

How is this different from the previous command? The `dry-run` argument prepares a request for the API server without actually submitting it. (You can replace the "`client`" option with "`server`" to submit it to the API server without

persisting the request to the cluster, which allows you to validate the request as a test.) We've also used the `-o` argument to output this practice request in the same YAML format we would like to use for our manifest. The result:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 creationTimestamp: null
 labels:
   app: nginx
 name: nginx
spec:
 replicas: 1
 selector:
   matchLabels:
     app: nginx
 strategy: {}
 template:
   metadata:
     creationTimestamp: null
     labels:
       app: nginx
   spec:
     containers:
     - image: nginx
       name: nginx
       ports:
       - containerPort: 80
       resources: {}
status: {}
```

In your project directory for this book, create a directory called `/05-deployments/` and inside create a blank file called `deployment.yml`. Open the file with your favorite code editor and copy over the kubectl output.

A few of these details—the `creationTimestamps` and `status`—aren't super-relevant to us right now; we can leave those to the API Server. But let's make some changes and additions to the fields that do interest us:

- On line 9, change the `replicas` spec to `5`
- On line 21, change the container image to `nginx:1.7.9`
- Add two new lines between lines 6 and 7. On the first of these new lines, at the same hierarchical level as line 6, add a new label: `tier: backend`. On the second new line, add a new label: `env: dev`

When you're finished, the manifest should look like this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 creationTimestamp: null
 labels:
   app: nginx
   tier: backend
   env: dev
 name: nginx
spec:
```

```
  replicas: 5
  selector:
    matchLabels:
      app: nginx
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.7.9
        name: nginx
        ports:
        - containerPort: 80
        resources: {}
status: {}
```

So what have we done here? We specified that we want:

- 5 replicas in this deployment
- a specific version number for our container image, rather than the latest version that is used by default
- Two new labels added to the deployment object's metadata.

Labels are string key-value pairs that help us organize and keep track of our objects. We can define both the key and the value; for example, we added a key "`tier`" and the value "`backend`."

Other deployments might use the same key and a different value like "`frontend`."

Ultimately, *every* field in the manifest is a key-value pair. Remember how `etcd` is a key-value data store? These are the keys and values that get stored. All the configuration and status data that define the workings of the cluster are ultimately broken down into pairs of keys and values, which the API server transmits either by the default JSON format or as Protocol Buffers[9] (aka Protobufs).

If we want to learn about the possible fields for a given object, we can use the `explain` functionality of kubectl again. For example:

```
% kubectl explain deployment
```

```
KIND:     Deployment
VERSION:  apps/v1

DESCRIPTION:
   Deployment enables declarative updates
   for Pods and ReplicaSets.

FIELDS:
   apiVersion      <string>
     APIVersion defines the versioned schema of this
     representation of an object. Servers should convert
     recognized schemas to the latest internal value, and
     may reject unrecognized values. More info:
     https://git.k8s.io/community/contributors/devel/
     sig-architecture/api-conventions.md#resources
```

[9] https://developers.google.com/protocol-buffers/

```
   kind      <string>
      Kind is a string value representing the REST resource
      this object represents. Servers may infer this from
      the endpoint the client submits requests to. Cannot
      be updated. In CamelCase. More info:
      https://git.k8s.io/community/contributors/devel/
      sig-architecture/api-conventions.md#types-kinds

   metadata  <Object>
      Standard object's metadata. More info:
      https://git.k8s.io/community/contributors/devel/
      sig-architecture/api-conventions.md#metadata

   spec <Object>
      Specification of the desired behavior of the
      Deployment.

   status <Object>
      Most recently observed status of the Deployment.
```

Note that the `FIELDS` section tells us what type of value each key takes—`kind` takes a string, `spec` takes an object, and so on. We can drill down further with the `explain` command:

```
% kubectl explain deployment.spec
```

```
KIND:     Deployment
VERSION:  apps/v1

RESOURCE: spec <Object>

DESCRIPTION:
Specification of the desired behavior of the Deployment.
```

```
DeploymentSpec is the specification of the desired behavior
of the Deployment.

FIELDS:
   minReadySeconds  <integer>
      Minimum number of seconds for which a newly created
      pod should be ready without any of its container
      crashing, for it to be considered available. Defaults
      to 0 (pod will be considered available as soon as it
      is ready)

   paused    <boolean>
      Indicates that the deployment is paused.

   progressDeadlineSeconds<integer>
      The maximum time in seconds for a deployment to make
      progress before it is considered to be failed. The
      deployment controller will continue to process failed
      deployments and a condition with a
      ProgressDeadlineExceeded reason will be surfaced in
      the deployment status. Note that progress will not be
      estimated during the time a deployment is paused.
      Defaults to 600s.

   replicas  <integer>
      Number of desired pods. This is a pointer to
      distinguish between explicit zero and not specified.
      Defaults to 1.

   revisionHistoryLimit   <integer>
      The number of old ReplicaSets to retain to allow
      rollback. This is a pointer to distinguish between
      explicit zero and not specified. Defaults to 10.

   selector  <Object> -required-
      Label selector for pods. Existing ReplicaSets whose
      pods are selected by this will be the ones affected
      by this deployment. It must match the pod template's
      labels.
```

```
   strategy  <Object>
      The deployment strategy to use to replace existing
      pods with new ones.

   template  <Object> -required-
      Template describes the pods that will be created.
```

You can use dot notation to dig deeper into each field. But for now, let's send our deployment manifest to the API server. From the directory where you've stored the manifest:

```
% kubectl apply -f deployment.yml
```

Check that your deployment is running—or if you prefer, you can check the status of the individual pods.

```
% kubectl get deployments
```

```
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   5/5     5            5           15s
```

```
% kubectl get pods
```

```
NAME                     READY   STATUS    RESTARTS   AGE
nginx-f7ccf9478-9f7nm    1/1     Running   0          15s
nginx-f7ccf9478-c6p8v    1/1     Running   0          15s
nginx-f7ccf9478-gtdmv    1/1     Running   0          15s
nginx-f7ccf9478-vskgd    1/1     Running   0          15s
nginx-f7ccf9478-wrhl9    1/1     Running   0          15s
```

Perfect. Now we can delete our deployment and stop Minikube.

```
% kubectl delete deployment nginx
% minikube stop
```

In this lesson, we got multiple replica pods up and running within the cluster using a Deployment. But these are ephemeral pods—we expect them to be destroyed and replaced. In the meantime, other pods would need to be able to connect to the NGINX backend we've started to define. So how can we ensure that churning instances of a hypothetical frontend can find churning instances of the backend?

That's where the Service resource type comes in. In the next chapter, we'll explain the Service abstraction, start to connect two simple services across separate pods, and learn a little more about Kubernetes networking along the way.

Chapter 6
Using Kubernetes Services, Part 1

IN THIS CHAPTER...

- Understand the Service resource type
- Explore the role of the Service in Kubernetes networking
- Create, containerize, and deploy an API service

So far, we've discussed two of the most important types of abstractions you'll use on a Kubernetes cluster: Pods and Deployments. Now we need an abstraction to help our containerized workloads communicate—both within the cluster and with the outside world. That abstraction is the **Service**.

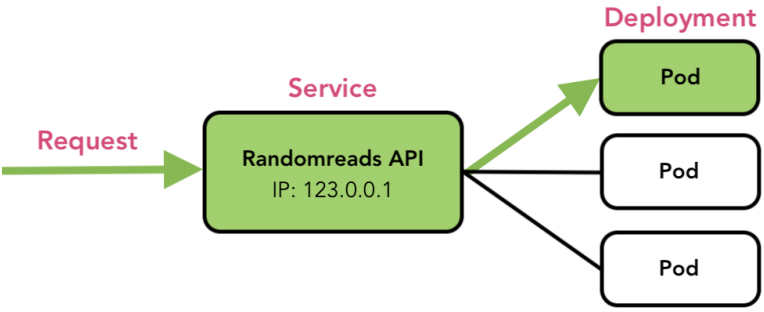
As we've seen, every pod has its own IP address. This IP address is accessible to any other pod on the cluster, regardless of the node on which it runs. The relative ease of pod-to-pod communication supports a microservices architecture—our microservices can talk to one another using standard protocols.

But there's one problem. Pods are ephemeral. We have to expect that a given pod might be deleted and replaced with a replica at any moment. If that pod is, say, serving an API, another microservice won't be able to reliably consume the API using the pod's dedicated IP address.

The Service abstraction solves that problem by connecting the deployment of a given piece of software to a service name that functions similarly to a domain name. In other words, the Service is an abstraction for the networking logistics required to deliver an app's functionality to clients—whether that means another pod or a client outside the cluster. In fact, a Service can even route to apps that aren't on your cluster at all—but we'll get into that next chapter.

Suppose we want to serve an API that generates random book recommendations. We'd follow these steps.

1. We deploy the recommendation API across three replica pods.
2. Now we want to deploy a web interface that consumes that API and serves the information to users. Instead of connecting to the recommendation API via a given replica's specific IP address, we'll be able to resolve a hostname like **http://randomreads-api** via `kube-dns`, Kubernetes' own Domain Name System (DNS).
3. That hostname directs the consumer's request to any pod currently running the recommendation API. If the pod in question is deleted, no problem: future requests will go to a different pod.



As you can see above, our Randomreads service—meaning this instance of the Service resource type—has its own IP address. When the cluster receives a request, it passes it on to the appropriate service, which in turn passes it to an appropriate pod that belongs to the specified deployment.

To understand services more clearly, let's build one from scratch.

Exercise: Communication Between Pods

For this exercise, we'll take the Randomreads API example and actually build it out.

OBJECTIVES

- Write and containerize an API to serve random book recommendations
- Write and containerize a web interface that consumes an API
- Run the API as a service
- Run the web interface as a service

I'll write this simple example with **Node.js**, a JavaScript runtime designed for server-side applications. You don't need to be proficient with Node to follow the exercise, though some familiarity will yield additional insight into what is going on here. I've chosen Node because JavaScript is one of the most widely used and understood languages, but most any language would do. One of the benefits of using an API-driven microservices approach is that different teams can use different languages according to suitability for the project at hand, available skill-sets, or simple preference. If you tend to work in another language and want to set yourself an additional challenge with this exercise, you may wish to try writing the API and web interface yourself.

If you'd like to follow along with my code (which is available in full on GitHub[10]), you'll need to install Node[11] on your system. Alternatively, you can simply read the code segments and deploy the container images I've already created, which are available on Docker Hub[12].

In your project directory for this book, create a new directory called `/06-services/` and a subdirectory called `/rr-api/`. Inside, initialize the project with `npm` and create a new file called `index.js`. Additionally, we're going to use the `express` module, so we'll go ahead and install that.

```
% npm init -y
```

[10] https://github.com/ericgregory/kube-5mins/tree/main/06-services/rr-api
[11] https://nodejs.org/en/download/
[12] https://hub.docker.com/repository/docker/ericgregory/randomreads-api

```
% touch index.js
% npm i express
```

Add the following code to `index.js`:

```
const express = require('express');
const app = express();

app.use(express.static('public'));

// This array stores a group of objects for books we
might recommend.

const bookstore = [
{ title: 'My Brilliant Friend', author: 'Elena
Ferrante'},
{ title: 'Piranesi', author: 'Susanna Clarke'},
{ title: 'The Summer Book', author: 'Tove Jansson'},
{ title: 'Middlemarch', author: 'George Eliot'},
{ title: 'Song of Solomon', author: 'Toni Morrison'},
{ title: 'The Tale of Genji', author: 'Lady Murasaki'}
]

// This function will generate a random integer.

function getRandomInt(max) {
   return Math.floor(Math.random() * max);
};

/*
This function requests a random integer with a range
defined by the length of the array.
Then it indexes the array and returns the appropriate
object.
*/
```

```
function randomizeBook() {
   let range = bookstore.length;
   let rng = getRandomInt(range);
   return bookstore[rng];
}

// The server responds to GET requests with a JSON
object for one of the books from the array.

app.get('/', (req, res) => {
   res.json(randomizeBook());
});

// The server is running on port 80, which is the
expected default.

app.listen(80, () => {
   console.log('The web server has started on port
80');
});
```

For the purposes of this example, our API will simply produce a randomized result from an array baked into the API server itself.

Now let's create a container image for our API server. We'll write a Dockerfile reproducing the setup we had to do for our local environment. Create a blank file called `Dockerfile` in your `/rr-api/` directory and add the following:

```
# Sets the base image
FROM node:18
```

```
# Establishes the working directory for your app
within the container
WORKDIR /usr/src/app

# Copies your package.json file and then installs
modules
COPY package*.json ./
RUN npm install

# Copies your project files and then runs the app
COPY . .
CMD [ "node", "index.js" ]
```

Save the file, then create a file called `.dockerignore` in the same directory, adding the following lines. (Doing this will prevent any accidental overwrites or confusion of Node modules when the build copies project files over to the container image.)

```
node_modules
npm-debug.log
```

Now build the image and publish it to Docker Hub:

```
% docker build . -t <Docker ID>/randomreads-api
% docker push <Docker ID>/randomreads-api
```

Now your image is available on Docker Hub. If you decided you'd rather use my image, it's available at

ericgregory/randomreads-api. Otherwise, simply enter your own Docker ID in the next command.

Make sure Minikube is running with `minikube start`. Now we'll create a deployment with three replicas:

```
% kubectl create deployment randomreads-api
--image=<Docker ID>/randomreads-api --port=80
--replicas=3
```

Before we expose the app as a Service, let's do a kubectl dry run and take a look at a YAML manifest for a service:

```
% kubectl expose deployment randomreads-api
--type=NodePort --port=80 -o yaml
--dry-run=client
```

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: randomreads-api
  name: randomreads-api
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: randomreads-api
  type: NodePort
status:
  loadBalancer: {}
```

Note that the `spec.selector` field uses the `app: randomreads-api` label to specify which pods should fulfill this service—in this case, pods from our `randomreads-api` deployment which include that `app` label.

Let's go ahead and expose the service for real.

```
% kubectl expose deployment randomreads-api
--type=NodePort --port=80
```

To see the API output from our browser, we can use a handy shortcut provided by Minikube. The following command returns (and usually opens) the URL of an exposed Service:

```
% minikube service randomreads-api
```

Ta-da! We've launched our first original service on Kubernetes! By now, you should have a much more specific understanding of what was happening under the hood when we started a Service back in our chapter on setting up a Kubernetes learning environment.

Now we still have several objectives before us. We need to write, containerize, and run our web interface in such a way that it consumes data from the `randomreads-api` on the cluster. But our five minutes are up for now. The next time we use our API Service, we'll be making a small but important change, so you can delete the `randomreads-api` deployment and service for now:

```
% kubectl delete service randomreads-api
% kubectl delete deployment randomreads-api
% minikube stop
```

In the next chapter, we'll finish our survey of services and get our web interface and API server working together across the cluster.

Chapter 7
Using Kubernetes Services, Part 2

IN THIS CHAPTER...

– Understand Service types
– Practice using labels and selectors
– Create, containerize, and deploy connected API and frontend services

In the last chapter, we began our survey of the **Service** abstraction in Kubernetes, which serves as a sort of domain name for workloads. Services help us connect users or apps to functionality that may be served across any number of identical, ephemeral pods.

In order to understand the Service resource type, we started an exercise in which we will connect two pieces of a simple book recommendation app across a cluster: a backend API and a web interface. In this chapter, we'll complete the exercise by launching our web interface.

Previously, we launched our Randomreads API service with a command in kubectl. We deleted the service at the end of the lesson, so we'll need to launch the service again—but this time, we're going to do it a little differently.

As before, we'll create our deployment with the following command. (Remember, if you wish to use a pre-created

container image rather than your own, you can substitute your `<Docker ID>` with mine, `ericgregory`.)

```
% kubectl create deployment randomreads-api
--image=<Docker ID>/randomreads-api --port=80
--replicas=3
```

This time, we're going to launch our API Service with a **YAML manifest**. In a new directory for this chapter called `/07-services-2/`, create a subdirectory called `/manifests/`. (The dedicated directory isn't strictly necessary—your manifests can be located anywhere—but it will help us keep organized.) Here, create a new file called `service.yml` and include the following:

```yaml
apiVersion: v1
kind: Service
metadata:
 labels:
   app: randomreads-api
   env: prod
   tier: backend
 name: randomreads-api
spec:
 ports:
 - port: 80
   protocol: TCP
   targetPort: 80
 selector:
   app: randomreads-api
 type: ClusterIP
```

Take special note of the **ClusterIP** type on the last line. This is where we're making a change—and now I have to make a confession…

In the first part of this exercise, we specified the **NodePort** type and didn't dwell on why. Here's the reason: I wanted to end that chapter with a satisfying, tangible sign that our API was running on the cluster. Using a NodePort Service type made it easy to see our API output in a web browser. That's nice for learning and development purposes, but it's not how we want our service to work in production.

So how *would* we want it to work? Well, there are four types of Services in Kubernetes. Let's consider them one by one.

ClusterIP

A ClusterIP Service has a virtual IP address accessible *within*—and only within—the Kubernetes cluster (hence the name). This is well-suited for many backend services that simply don't need to be available beyond the cluster, affording a layer of isolation from the wide world outside.

In practice, this is probably the Service type you will use most often. It's also the default Service type.

NodePort

Of course, sometimes we *do* want to be able to access a service from outside our cluster. Perhaps we're exposing a service to outside users or clients, or perhaps we simply want to test a

piece of functionality in development. The NodePort Service type is one way to do this, but it's not ideally suited to production.

When you create a NodePort Service, every node on your cluster will listen for requests on a given port. External requests can reach the Service at the IP address of a node followed by the specified port:

```
<IP address of the node>:<NodePort>
```

The NodePort API we launched last lesson, for example, had an address like:

```
http://127.0.0.1:61304/
```

When requests come in, the Service will route them to a ClusterIP, which in turn will route the request to an appropriate Pod. So a NodePort Service isn't so much an alternative to ClusterIP as an addition.

When you expose a NodePort Service, *all* of your nodes will need to have a port open to the outside world for each service. If you're running multiple services, every node will need to open a port for each. This is fine for development and learning, but it's a blunt approach to the problem of Service exposure that doesn't scale well and won't help you route traffic intelligently between your nodes.

For *that*, you'll need to use…

LoadBalancer

The LoadBalancer Service type is designed to help your cluster efficiently apportion external traffic across nodes. It will help you operate at scale and it's appropriate for production.

When your cluster's load balancer receives an external request, it selects an appropriate node and routes the request to that node via an internally available NodePort Service, which routes the request to a ClusterIP, and then finally to an appropriate pod.

(So again, we're building on top of the other types, not replacing them.)

Notably, **the LoadBalancer type depends on external load balancing solutions from a cloud provider**. LoadBalancer is designed for clusters on clouds like AWS, Azure, or Google Cloud, all of which have their own tools for load balancing. As we'll see in a moment, Minikube also has its own way of handling LoadBalancer Services.

This extensible approach follows directly from the design philosophy of Kubernetes, which aims for flexibility and portability. The particulars of a given load balancer implementation are abstracted away. For the most part, all developers have to do is interact with the service. Your code and your manifests will be the same regardless of which cloud provider or load balancing tool the system is using.

ExternalName

Now, what if we wanted to create a service that routed *not* to a workload on our cluster, but to some external application? There's a Service type for that: ExternalName.

Suppose that we wanted an application running on our cluster to be able to resolve requests to an external API using a consistent alias. An ExternalName Service enables us to do just that, specifying an external CNAME like `example-app.com`.

There is an important caveat here, however: ExternalName Services don't communicate over HTTP or HTTPS very well, since the target names in request headers sent from your services won't match the destination domain.

Building the web client

Now that we've surveyed our Service types, we understand why **ClusterIP** is the best choice for our Randomreads API: it only needs to be accessible to the web client, which is also part of the cluster. Let's go ahead and start our deployment from the YAML manifest.

```
% kubectl apply -f service.yml
```

Now the API is running again, and it's available to other pods within the cluster via the `randomreads-api` domain. Let's build our web client.

We'll use the same tools as last time, Node.js and the `express` module, but remember that we could just as readily use a different toolkit here—if we were working on a different team than the developers of the backend API, we wouldn't be beholden to their favored languages and frameworks. In this case, however, we'll stay consistent for simplicity's sake.

As before, if you would rather skip the coding and deploy from my container image, you're welcome to do so; likewise, for an additional challenge, you may wish to recreate the frontend with the language of your choice. My code is on GitHub[13], and the container image for the web client is on Docker Hub[14].

In your `/07-services-2/` directory, create a new subdirectory called `/rr-web/`. Then, inside your new directory, initialize the project, create a new file called `index.js`, and install the `express` and `express-handlebars` modules:

```
% npm init -y
% touch index.js
% npm i express express-handlebars
```

Add the following code to `index.js`:

```
/*
Below we're requiring our two dependencies and setting
a very important constant: the API endpoint that we
wish to consume. We're simply using the name of our
randomreads-api Service.
```

[13] https://github.com/ericgregory/kube-5mins/tree/main/07-services-2/rr-web
[14] https://hub.docker.com/repository/docker/ericgregory/randomreads-web

```
*/

const express = require('express');
const { engine } = require('express-handlebars');
const ENDPOINT = 'randomreads-api'

const app = express();

app.use(express.static('public'));

/*
Express-Handlebars gives us a super-simple view engine
to render our webpage. Here we're configuring it to
look for webpage files in a directory called 'views'
and to use a wrapper file called 'main' as the default
layout.
*/

app.engine('handlebars', engine({
   helpers: {
       isCompleted: function (status) {
           if (status == "completed") {
               return true
           } else {
               return false
           }
       },
   },
   defaultLayout: 'main',
}));
app.set('view engine', 'handlebars');
app.set('views', './views');

/*
Now we've reached the juicy stuff. This will trigger
when a GET request is made to the server. Here we're
```

```
using fetch (an experimental but functional feature in
Node at the time of this writing in July 2022), and
we're using it to grab the JSON available at the API
endpoint we defined up top. Then we're funneling the
values inside the JSON payload into two variables that
we can utilize when we build the index page.
*/

app.get('/', async (req, res) => {
   fetch(`http://${ENDPOINT}/`)
       .then(response => response.json())
       .then(data => res.render('index', {
           title: data.title,
           author: data.author
       }
       ));
});

// Like the API, the web server is running on port 80,
which is the expected default.

app.listen(80, () => {
   console.log('The web server has started on port
80');
});
```

Now we'll create a new directory for `/views/` (with a `/layouts/` subdirectory inside). Here we'll add two very simple files.

```
% mkdir views
% mkdir views/layouts
% touch views/index.handlebars
% touch views/layouts/main.handlebars
```

Add the following code to `main.handlebars`:

```
<html>
<body>
{{{body}}}
</body>
</html>
```

(Told you it was a simple file.)

Now add this code to `index.handlebars`:

```
<h1>Randomreads</h1>
Need a read? How about...
<br> 
<div>
        <em>{{title}}</em>
        <br>
        by {{author}}
</div>
```

Again, pretty simple. Here we're making use of the `title` and `author` variables.

Our web client is done. Now we can write a Dockerfile (review the last chapter if you need a refresher) then build and publish a container image the same way we did before:

```
% docker build . -t <Docker ID>/randomreads-web
% docker push <Docker ID>/randomreads-web
```

If you prefer to use my image for the next step, it's available at ericgregory/randomreads-web. Create a deployment with three replicas:

```
% kubectl create deployment randomreads-web
--image=<Docker ID>/randomreads-web --port=80
--replicas=3
```

Now both of our deployments are running on the cluster. All that remains is to expose the service for the web client.

Understanding labels and selectors

Return to your existing `service.yml` file (located at `/07-services-2/manifests/`). We're going to make the following changes to the manifest:

- Our new service will be called `randomreads-web`
- The "`app`" label will have the value "`randomreads-web`"
- The "`env`" label will have the value "`dev`"
- The "`tier`" label will have the value "`frontend`"
- The app selector will point to the `randomreads-web` deployment, which it will find using the label on that deployment
- The Service type will be `LoadBalancer`

With the changes incorporated, your `service.yml` file should look like this:

```
apiVersion: v1
kind: Service
metadata:
 labels:
   app: randomreads-web
   env: dev
 name: randomreads-web
spec:
 ports:
 - port: 80
   protocol: TCP
   targetPort: 80
 selector:
   app: randomreads-web
 type: LoadBalancer
```

Now go ahead use kubectl to add this service to the cluster:

```
% kubectl apply -f service.yml
```

Note that we're using the *same* file we used before, but we're creating an entirely new service. The `service.yml` file is totally decoupled from any particular entity on the cluster; it's simply a vehicle for our intentions.

The `env` and `tier` labels in these manifests are completely arbitrary keys–they don't carry any inherent meaning to the system, but *we* can use them to organize and manage services. Imagine that `randomreads-web` is a dev deployment, not yet ready for production. Labels could help us keep track of which services are operating in which environments.

Labels are simple but powerful tools that are essential to how Kubernetes works. Suppose we want information on all of the services that are part of our overall app's backend. We can use **selectors** to identify those services via labels (specified here using the `-l` argument):

```
% kubectl get services -l env=dev
```

```
NAME              TYPE          CLUSTER-IP     EXTERNAL-IP  PORT(S)       AGE
randomreads-web   LoadBalancer  10.103.25.143  127.0.0.1    80:31126/TCP  2m
```

What if we only want to see backend services in production?

```
% kubectl get services -l tier=backend,env=prod
```

```
NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
randomreads-api   ClusterIP   10.106.116.211   <none>        80/TCP    14m
```

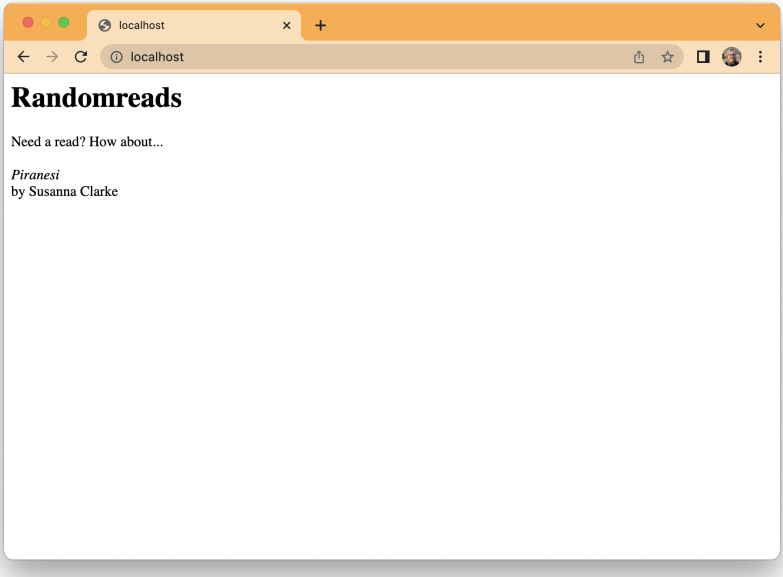
The same technique works for pods or deployments. We can use selectors to organize and select resources all throughout the cluster.

All right, that's enough delay. We've deployed our frontend service, so let's see if everything works correctly! Minikube enables you to access LoadBalancer services with the following command:

```
% minikube tunnel randomreads-web
```

This will start a running process in your current terminal tab (and may require sudo permission). Once you start the tunnel,

you should be able to access the web client on your local machine's `localhost`:



Et voilà! We have two services connected across our Kubernetes cluster, one of which is available externally.

That's it for today. Stop Minikube—your services and deployments will be automatically deleted.

```
% minikube stop
```

We've spent the last two lessons building apps specifically for Kubernetes, but a great deal of cloud native development entails taking monolithic applications *apart*. In the next lesson, we'll start decomposing an actual monolith—and what's more, it'll be a *stateful* monolith. To run a persistent application on our cluster, we'll need to explore how volumes work in Kubernetes.

Chapter 8
Persistent Data and Storage

IN THIS CHAPTER...

- Learn the fundamental of persistent data on Kubernetes
- Understand the PersistentVolume, PersistentVolumeClaim, and StorageClass resource types
- Practice running a database server on Kubernetes

If you're a developer whose organization is moving to Kubernetes, there's a pretty good chance that you're going to spend some time decomposing existing **monolithic applications** into **microservices** that take advantage of Kubernetes' capabilities.

In the next few lessons, we're going to work step-by-step to break down a monolith into dedicated Services running on Kubernetes. Specifically, we will decompose a simple To Do app with a web client. This will give us a hands-on, project-based approach to learning major concepts in Kubernetes development such as Volumes, Secrets, and Ingress.

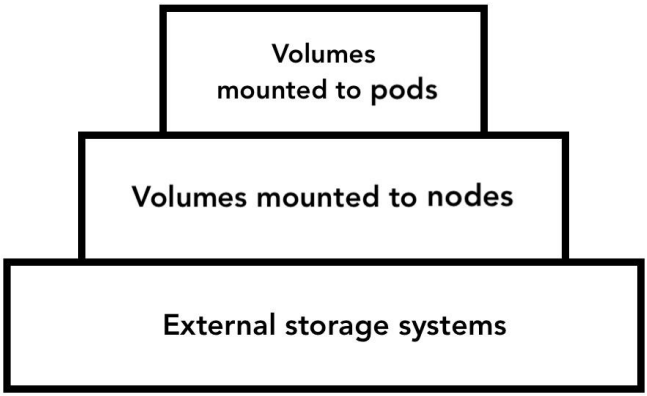
The monolithic To Do app is built with Node.js, using a MySQL database to store task data. We'll start with one of the thornier challenges of decomposition: how to persist data in Kubernetes.

The pyramid of persistent data

In Kubernetes, containers are typically presumed to be ephemeral and immutable, which is to say that we expect any given container to be short-lived, replaceable, and unchanging in its contents. For **stateless** applications—apps that don't store or modify changeable data—this is perfect.

But most apps are ***stateful***: they change into different states as users or clients create, update, or delete data that may persist over time. This is a little out of step with the philosophy and assumptions of Kubernetes, but it's a fundamental requirement that the system has to be able to handle.

And so it can! There are various ways in which Kubernetes can persist data, at varying degrees of persistence and complexity. You can think of these approaches as steps on a pyramid, with the duration and reliability of data persistence running more or less inverse to the complexity of the solution.



Let's start at the top of the pyramid and decide which solution will be most suitable for our purposes.

Volumes mounted to pods

Containers within a pod can mount a writable volume at the *Pod* layer, similar to the way a Docker container might mount a volume on a host system. This approach is relatively simple—and has its uses—but your data won't be very durable. If the container is replaced *within* the pod—for example, if it has failed and been automatically replaced—the new container will still have access to the volume. But you're never going to store, say, user account data this way, because as soon as the *pod* is erased, any data the container has written will be gone. Pod-mounted volumes are most suited to temporary caches and other short-term uses that you don't expect to last any longer than a given pod—for example, this technique can be used to mount a Secret to a pod (which we'll discuss in depth in the next lesson).

Volumes mounted to nodes

If you need data to persist beyond the lifespan of a particular Pod, you might consider using a volume mounted at the *node* level. Now persistent data can be made available to any pod running on that node, but the limitation should be immediately clear: only pods on that node can access the data.

In a single-node cluster, or with pods that are limited to a dedicated node, node-mounted volumes can be a great fit. But if you're running the same pods across multiple nodes, that's a

problem—and for High Availability production apps, you really want to be able to run across a multi-node cluster.

External storage systems

External storage systems can enable us to store data that is available clusterwide. Excellent! But dynamically apportioning storage across a multi-node cluster is a complex job that varies somewhat according to the underlying infrastructure. So this is another area where Kubernetes outsources the job to plugins and third-party tooling, whether for cloud providers like Azure and AWS or components like **NFS** or **Ceph** configured on-prem.

External storage systems like these connect to Kubernetes by way of the **Container Storage Interface (CSI)**. This provides a standard interface between different types of storage systems—including open and closed source solutions from various vendors—and the storage abstractions in the Kubernetes API. This design is similar to the Container Runtime Interface (CRI) model for runtimes, which we discussed in Chapter 3.

As a specification, the CSI is itself external[15]—and not limited—to the Kubernetes core, managed as an independent project and used by other container orchestrators such as Mesos and Nomad.

But back to Kubernetes. What was that about those storage abstractions in the API? When we're working with Kubernetes, we need a standardized way to specify our requirements for

[15] https://github.com/container-storage-interface/spec

persistent storage. Kubernetes provides this with API resources called **PersistentVolumes**, **PersistentVolumeClaims**, and **StorageClasses**. We'll break them down one by one.

Introducing PersistentVolumes and PersistentVolumeClaims

If we run `kubectl explain persistentvolume`, we get this description:

```
PersistentVolume (PV) is a storage resource provisioned by
an administrator. It is analogous to a node.
```

What does it mean to be "analogous to a node"? Simply that the **`PersistentVolume` (PV)** is a system resource for storage in the same way that a node is a system resource for compute. `PersistentVolumes` are administrator-facing objects: system resources to be defined for others' use.

As a developer, you wouldn't typically create them, just as you probably wouldn't add a node to the cluster. But you would *consume* storage, and to do so you would make a **`PersistentVolumeClaim`** (PVC)—a request to provision available storage resources. This is essentially your app's voucher for storage utilization.

Depending on the specific requirements of your PVC and the resources available on your cluster, the provisioning of your storage may be **static** or **dynamic**.

With **static provisioning**, storage is a zero-sum resource, correlated directly with actually existing PersistentVolumes created manually by an administrator. It's as though you walked up to a hotel and requested a room; there is a limited, defined set of rooms, and they are either occupied or available.

Administrators may also give you the option of **dynamic provisioning**. In this case, the system may automatically create PersistentVolumes based on your requirements and available resources, using a specified storage solution and predetermined parameters. If the hotel doesn't have any vacancy, no problem; they've partnered with an RV company and your room will be here shortly.

To understand how dynamic provisioning works, we'll need to dive into the third major storage API resource for Kubernetes: StorageClass.

Dynamic provisioning with Storage Classes

Much as Deployments provide an abstraction for managing Pods, the **StorageClass** API resource sits over, manages, and dynamically creates PersistentVolumes.

Using a StorageClass is *necessary* to dynamically provision storage, and in general it is the preferable target for storage requests via PersistentVolumeClaims. In the same way that you typically want to interact with a Deployment rather than

an individual Pod, you will typically prefer to interact with StorageClasses (SCs) rather than individual PVs.

Why use this higher level of abstraction? StorageClasses enable you to provision PVs for *particular usages*: storage with a triplicate backup policy, for example, or higher-latency cold storage. In other words, you're defining a *class* of *storage* (just what it says on the tin!), instances of which can be spun up as needed and at scale.

With Minikube running, we can take a look at StorageClasses on our cluster:

```
$ kubectl get sc
```

```
NAME                 PROVISIONER
standard (default)   k8s.io/minikube-hostpath
```

Wait—we haven't created any StorageClasses, have we?

Not yet! But this is typical. You would find a `standard (default)` StorageClass (and likely a couple more options) on a cloud-hosted Kubernetes cluster as well. If you were using AWS or Azure rather than local Minikube, you would find a different value in the `PROVISIONER` field. `PROVISIONER` refers to the CSI plugin through which Kubernetes will be connecting dynamically generated PVs to whatever external storage system you are using.

Another field we should note is the `reclaimPolicy`, which is set to `Delete`. This means that storage volumes and PVs provisioned by a PersistentVolumeClaim to this StorageClass will persist only as long as the PVC: once it's deleted, the data will be deleted, too. This is the default for a StorageClass without a defined reclaimPolicy, and it makes sense from a data security and resource efficiency standpoint: it's biased toward the deletion of unnecessary or unused artifacts. From a data *persistence* standpoint, it's a risky policy, and if you want to avoid that risk, you'll need to set the `reclaimPolicy` to `Retain` in your bespoke SCs. We'll see how to do that in a moment. (We'll discuss the other fields in the above output—`volumeBindingMode` and `allowVolumeExpansion`—as well.)

Take note: The `standard` SC is intended for quick experiments and debugging—in production, you would want a deliberately defined SC aligned with the specific requirements of your app. While we *could* use it for a learning exercise, we'll act as our own operator and create a new StorageClass, which our app will access by means of a PersistentVolumeClaim.

Exercise: Running MySQL on Kubernetes

We said up top that our objective here is to decompose a To Do app built on Node.js and MySQL, using a MySQL database to store tasks. You can check out the code for our starting monolith on GitHub, but we won't be using it much today. The

MySQL server is already pretty separate from the rest of the app, so that makes a good candidate to migrate first.

I see your hand in the back of the room: "Wait, *should* we run MySQL on Kubernetes?" That's a good question! We'll address it at the end of the lesson. But for now we'll focus on how to go about it.

First, we'll create a container image that fits the requirements of our app. In this case, we want to deploy a MySQL server with a dedicated database and credentials ready to go. We'll accomplish this by writing a Dockerfile that builds on top of the official MySQL base image. You can store your Dockerfile anywhere, but for organization's sake we will write it in our `/kube-5mins/` project directory, in a subdirectory for this chapter called `/08-storage/`.

```
FROM mysql:8
ENV MYSQL_ROOT_PASSWORD=octoberfest
ENV MYSQL_DATABASE=todo_db
```

In this Dockerfile, we're…

- **Building on top of the MySQL base image**. It's a good idea to specify a version number rather than defaulting to the latest image. Future versions of a container image may introduce breaking changes, so by specifying a specific version, you stay in control of the image your pods are using (and can make upgrades more deliberately).

- **Establishing a root password and database through environment variables**. Note: information in Docker images isn't encrypted or particularly secure by default, and this means you would never want to store this Dockerfile *or* your image in a public repository! **Indeed, in production, you wouldn't want to set a sensitive password directly via an environment variable at all**. We'll discuss why—and a more secure approach—next lesson, but for now, just remember that we're cutting a corner here, and you would ***never*** want to upload an image like this to Docker Hub in the real world. Instead, you'd want to use a private image registry.

Now we'll upload our image to Docker Hub. Remember, **we wouldn't use Docker Hub in practice**! Imagine, instead, that we're pushing the image to a private registry that has been set up by our cluster administrator. The registry might be hosted externally by a cloud provider or it might be hosted within the cluster using tools like Mirantis Secure Registry or Harbor.

```
% docker build . -t <Docker ID>/todo-mysql
% docker push <Docker ID>/todo-mysql
```

Start Minikube with `minikube start`. Now it's time to create our `StorageClass`. In the same storage directory as your Dockerfile, create a YAML file called `sc.yml` and add the following:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
 name: sc-local
```

```
provisioner: k8s.io/minikube-hostpath
parameters:
 {}
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: false
```

In our manifest for the StorageClass, we specify…

- A **`name`** to help match the StorageClass with relevant claims.
- A **`provisioner`** of **`k8s.io/minikube-hostpath`**. This points to the Minikube-specific driver that enables us to provision storage space from the Minikube instance's host.
- **An empty `parameters` field**, which I've included with a blank object value here because it's an important field in real-world implementations. This is where an operator might set parameters specific to a given vendor's storage solution. The vendor will provide documentation for using this field.
- A **`reclaimPolicy`** of **`Delete`**: Alternatively, we could set this to `RETAIN`. If we did so, the system would preserve data created by a given PVC even once that PVC is deleted, but it wouldn't be accessible by future PVCs, and we would have to manually clean up the data left behind.
- A **`volumeBindingMode`** of **`Immediate`**. This controls the timing of the creation and binding of new PersistentVolumes with a PVC that makes a claim

against the StorageClass. In this case, as soon as a PVC is submitted, the StorageClass will provision and bind storage to the claim. That's nice and fast, but not without its complications! The `Immediate` binding process can all happen before the system has any knowledge of particular Pod requirements, so the SC could bind storage that isn't actually suitable for the consuming app we have in mind—it might be on the wrong node, for example. If we want to avoid that kind of snafu, we can use a `volumeBindingMode` value of `WaitForFirstConsumer`—this will wait until an actual, real-life Pod tries to consume storage via a PVC to provision and bind the volume. Not all provisioners support that option, however, and our `minikube-hostpath` provisioner is one that does not.

- An **`allowVolumeExpansion`** of **`False`**. This defines whether or not the size of the PersistentVolumes created by this SC may be changed (by editing the requesting PVC) after the initial provisioning. It accepts a Boolean value.

Now we'll create our StorageClass:

```
% kubectl apply -f sc.yml
```

Once the SC is created, we can check on its status:

```
% kubectl get sc
```

```
NAME                 PROVISIONER                RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
sc-local             k8s.io/minikube-hostpath   Delete          Immediate           false                  15s
standard (default)   k8s.io/minikube-hostpath   Delete          Immediate           false                  78m
```

Our SC is available, so now we'll create a claim.

In a new YAML file called `pvc.yml`:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: tododb-claim
spec:
 storageClassName: sc-local
 accessModes:
   - ReadWriteOnce
 resources:
   requests:
     storage: 1Gi
```

In the spec, we're invoking our new StorageClass by name and then defining some requirements for the PV we're requesting:

- An **`accessMode`** of **`ReadWriteOnce`**. This means that one node can access a volume at a time (though multiple pods on that node can access the volume simultaneously). This is the default access mode and the only one universally supported by different PV storage solutions. Other access modes include:
    - **`ReadOnlyMany`**: Multiple nodes can read from the same volume simultaneously.
    - **`ReadWriteMany`**: Multiple nodes can read and write to the same volume simultaneously.
    - **`ReadWriteOncePod`**: Only one *pod* can access a volume at a time.

- **Storage capacity** of 1 gibibyte

Apply the claim, and then run `kubectl get` on the PV again:

```
% kubectl apply -f pvc.yml
% kubectl get pv
```

```
NAME         CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                 STORAGECLASS   AGE
pvc-a8b...   1Gi        RWO            Delete           Bound    default/tododb-claim  sc-local       33s
```

Remember: We used the Immediate volumeBindingMode, so this PVC has provisioned and bound to a volume without being consumed by an actual Pod.

Now we're ready to deploy our MySQL server. In the same storage directory, create a new file called `todo-mysql.yml`:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 labels:
   app: todo-mysql
 name: todo-mysql
spec:
 replicas: 1
 selector:
   matchLabels:
     app: todo-mysql
 template:
   metadata:
     labels:
       app: todo-mysql
   spec:
     volumes:
       - name: todo-volume
```

```
        persistentVolumeClaim:
          claimName: tododb-claim
    containers:
    - image: ericgregory/todo-mysql
      name: todo-mysql
      ports:
      - containerPort: 80
      volumeMounts:
      - mountPath: "/var/lib/mysql"
        name: todo-volume
```

This Deployment manifest includes a `volumes` section in the spec, connecting a container-level volume mount named `todo-volume` with our `tododb-claim` PVC.

In the container spec, we also specify a `mountPath`: the directory within the container to associate with the volume, where we wish to write data.

Now create the deployment:

```
% kubectl apply -f todo-mysql.yml
```

It will take a moment for the deployment to be ready. You can run `kubectl get pods` to check the status of the pod being generated.

Once the pod is ready, we can hop into it using the `kubectl exec` command. This works much the same way as `docker exec`. In theory, we don't want to be handling individual pods manually, but this is a good way to check in on our container and storage. (Besides, practice is often messier than

theory—while we'd *prefer* not to handle individual pods manually, more than one developer has found themselves debugging by checking logs within a pod.)

Copy the name of the individual pod from the output of `kubectl get pods`, and then use that with `kubectl exec`. (Your pod name will differ slightly from mine below.)

```
% kubectl exec --stdin --tty
todo-mysql-5798c74978-dksct -- /bin/bash
```

Now we're inside the container—you can tell by the **`bash4.4#`** at the beginning of your terminal line, cluing you in to where you are. (Note that you shouldn't type `bash4.4#` in the command below—that merely indicates the context for the action, like the **%** before terminal commands.) Go ahead and open MySQL:

```
bash4.4# mysql -u root -p
```

On the prompt, enter the password we set: `octoberfest`

Now we should be in MySQL, within the container; your terminal line should begin with mysql and an angle bracket. (As above, you shouldn't type **`mysql>`** in the commands below.) From here, let's access the database we created:

```
mysql> USE todo_db;
```

Excellent, the database is there! We'll create a test table in the database:

```
mysql> CREATE TABLE IF NOT EXISTS test123 ( \
    -> id INTEGER PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.04 sec)

mysql> SHOW TABLES;
+------------------+
| Tables_in_todo_db |
+------------------+
| test123          |
+------------------+
1 row in set (0.00 sec)
```

Now we'll exit MySQL and the container by typing `exit` twice.

The last step is to test our work: Will our `test123` table persist if the pod is deleted and replaced? Let's see!

```
% kubectl delete pod todo-mysql-5798c74978-dksct
% kubectl get pods
```

```
NAME                          READY   STATUS    RESTARTS   AGE
todo-mysql-5798c74978-8b28k   1/1     Running   0          6s
```

All right, after we deleted the original pod, it was immediately replaced by a new one. (Remember that your pod names will differ slightly from mine.) Let's peek inside the new pod.

```
% kubectl exec --stdin --tty
todo-mysql-5798c74978-8b28k -- /bin/bash
```

```
bash-4.4# mysql -u root -p
Enter password: octoberfest
mysql> USE todo_db;
mysql> SHOW TABLES;
```

```
+------------------+
| Tables_in_todo_db |
+------------------+
| test123          |
+------------------+
1 row in set (0.01 sec)
```

Excellent! Everything is working just the way we hoped.

Before we finish for the day, I bracketed an important question earlier, and now we should return to it: *Should* we run a database on Kubernetes?

Not long ago, the conventional wisdom would have said, "Absolutely not—Kubernetes is for stateless apps." But the ecosystem is constantly evolving, and these days the answer is more complicated.

The resources for stateful apps within Kubernetes have matured, including an important abstraction that can help to simplify the management of stateful applications: an API resource called **StatefulSets**. (We'll make use of StatefulSets soon, when we start to bring all the pieces of our application together.) On top of the resources in the Kubernetes core, there are more third-party extensions tools than ever before—and as a result, lots and lots of folks are running lots and lots of databases on Kubernetes clusters.

But should *you*? The annoying-but-true answer is: It depends!

There's no doubt that stateful applications introduce a significant amount of complexity to Kubernetes deployments. But there's also an enormous appeal to running everything in the cluster and using the same deployment models and processes for every piece of an app.

When it comes to using databases with Kubernetes, you have a lot of options. Some organizations use external, managed database services and simply keep the stateful elements of their services off the Kubernetes cluster entirely. Others use PersistentVolumes backed by external services. Still others run stateful apps entirely on the cluster, whether using purpose-built cloud native database technologies like **CockroachDB** or extensions for established tools like MySQL, PostgreSQL, and others. (More on those in a future lesson.)

In order to choose an approach, you'll want to ask yourself questions like:

- How important is scaling for this database?
- Am I migrating existing data? How complicated would that migration be?
- How fault-tolerant are my options?
- What are the security and compliance requirements for this data?

The right answer will depend on your use case, and it might well differ from workload to workload.

That's it for today. Stop Minikube as usual. In the next lesson, we'll focus on Kubernetes Secrets, and get our To Do app's API server running with an eye toward security.

Chapter 9

Secrets with Environment Variables and Volume Mounts

IN THIS CHAPTER...

— Learn the uses (and limitations) of Kubernetes Secrets
— Implement a Secret via environment variable and volume mount
— Manage Secrets with Lens

In the last chapter, we used a PersistentVolume to store data in Kubernetes—specifically, in a containerized MySQL database. But we cut a corner by configuring the password for our database via an environment variable.

That's an insecure approach that we would never want to use in production.

Why? Environment variables are readily visible within their containers, within Kubernetes logs, in the manifests, and to anyone on the cluster with the ability to view the spec.

If an attacker gains access to any of these, they will have access to the database's root password as well.

Sub-optimal, to say the least!

Instead of configuring a password *directly* through an environment variable, we can use **Kubernetes Secrets**. While they aren't the be-all and end-all of Kubernetes security, Secrets give us a way to add an additional layer of protection for sensitive data.

As we continue our overall project of decomposing a monolithic To Do app into a set of cloud native services, our next step will be to implement and connect our database and API server with Secrets.

Along the way, we'll learn exactly what Kubernetes Secrets are and how they work.

What are Kubernetes Secrets?

Kubernetes Secrets are confidential credentials, stored by default in `etcd` and available for authorized entities to use in authentication throughout the system.

Essentially, Secrets are placeholders for sensitive information. Once we've established a Secret in our cluster, we can use that Secret to inject, say, the value of a "password" environment variable for a database container, rather than specifying the password itself.

Secrets can hold either plaintext or base64-encoded data. The latter is a bit more secure, preventing someone in the same room from seeing a password value at a glance, but you shouldn't mistake this encoding for encryption.

A word of warning

Before we go any further, I should repeat that Secrets are not the be-all end-all of Kubernetes security. Once a container has consumed a Secret via environment variable, the value is freely available within the container. The value is also stored **without encryption** in `etcd`. This is not to say you shouldn't use Secrets, but rather that you should be aware of their limitations, and they should be only one part of your overall security strategy. We'll discuss some ways to harden your security further in our final lesson.

You might very reasonably ask: If Secrets come with so many caveats, why do we bother to use them? What are they for? The simple answer is that the Secret is a **mechanism for abstracting credentials**, which then allows us to programmatically utilize and manage those credentials. With a password captured as a Secret, we can define which users, namespaces, or Pods can access that password. The Secret is intended for no more and no less—it isn't *about* encryption, but rather conceptual modeling and access control. As a result, it provides a way to integrate external solutions for better security.

The two most common ways for Pods to consume Secrets are through environment variables and as files injected into the pod through a volume.

- **Injecting a Secret as an environment variable** makes the value available to everything inside the pod. That can be useful in that multiple elements of your application can access it with ease, but it also means that there is risk of the value being exposed via logs or other processes.

- **Injecting a Secret by means of a file in a volume** can be more secure in the scope of the pod, since you can make the file only selectively available to components within a container. But it's also now only as secure as the volume where it's being stored.

The approach you use may depend on the application you're working with. In the case of our MySQL server image, we will use an environment variable, because this is how the app is designed to consume credentials. But if we're designing our own service, we might choose to use the volume-based approach. We'll use both techniques in today's exercise.

How to Create a Kubernetes Secret

As with other Kubernetes objects, we can create a Secret straight from kubectl. Enter the following command:

```
% kubectl create secret generic test-secret
--from-literal=password=octoberfest
```

In this command, we have…

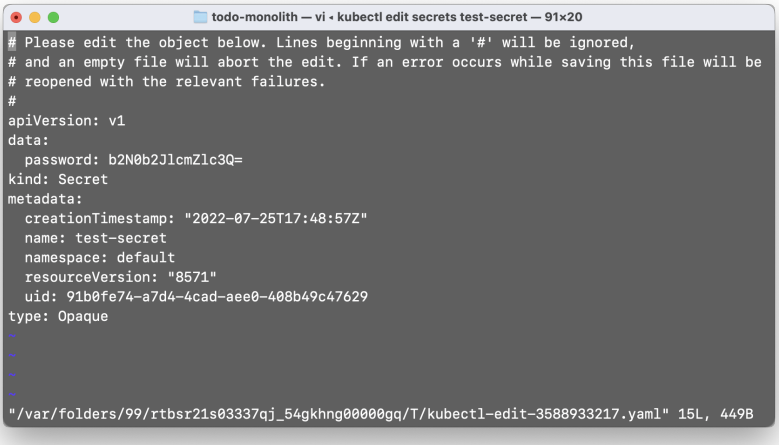
- Used the standard `kubectl create` command, and we've used it to create a Secret
- Specified that we want a `generic` Secret (which I'll explain in a moment)
- Named our Secret "`test-secret`"

- Defined a plaintext password value of "`octoberfest`" to store within the Secret using the `--from-literal` flag

Suppose we want to change some of the details of our Secret. We can do so with…

```
% kubectl edit secrets test-secret
```

This uses your default command line editor to open the Secret file. On my system, I'm viewing the Secret contents with `vi`:



A couple items to note here:

- The **`password`** value was automatically base64-encoded. Go ahead and copy this value—we'll use it in a moment.
- The value of the **`type`** field is **`Opaque`**.

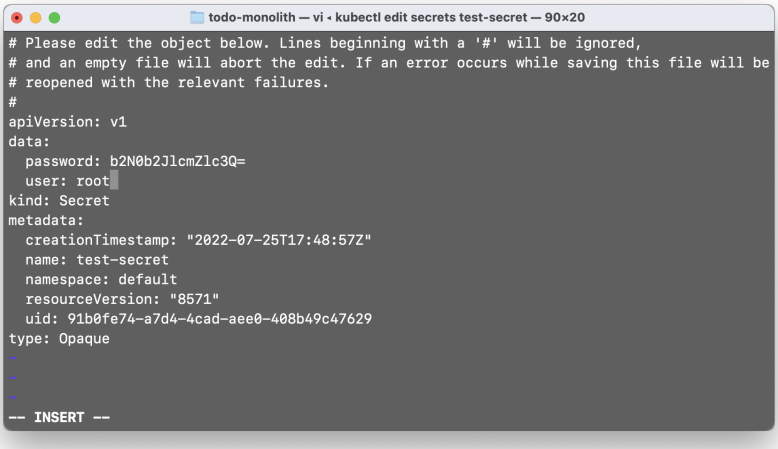
"Opaque" is the default Secret type. When we told kubectl that we wanted a generic Secret, this was what we meant: We want a Secret of the default type. The word "opaque" doesn't mean a whole lot here—this type could just as easily have been called "generic" or "standard" or "default." The word *opaque* is simply assuring us that the value of the Secret is only available to the users, Pods, and namespaces with access…in other words, that it's doing the basic job of a Secret.

There are several other Secret types available that impose additional constraints on configuration or usage and are pre-defined for specific use-cases. We won't be using these today, but they're worth noting for future reference:

`kubernetes.io/service-account-token`	ServiceAccount token
`kubernetes.io/dockercfg`	serialized `~/.dockercfg` file
`kubernetes.io/dockerconfigjson`	serialized `~/.docker/config.json` file
`kubernetes.io/basic-auth`	Basic authentication credentials
`kubernetes.io/ssh-auth`	SSH authentication credentials
`kubernetes.io/tls`	TLS client or server data
`bootstrap.kubernetes.io/token`	bootstrap token data

Some of these, such as the bootstrap token type, are used by the Kubernetes system itself. Additionally, you can define your own Secret type using a non-empty string as the `type` value.

Let's make some changes to our Secret! If you're using `vi` for your text editor, you can press `i` to enter `INSERT` mode, then add an indented line below the `password` field. Here, we'll store a `user` value of "`root`" in plaintext, just to demonstrate both editing and plaintext storage. Press `ESC` and then `:w` to write/save the file, and then `:q` to quit `vi`.



If you edit the Secret again, you can see that the `user` field is still there.

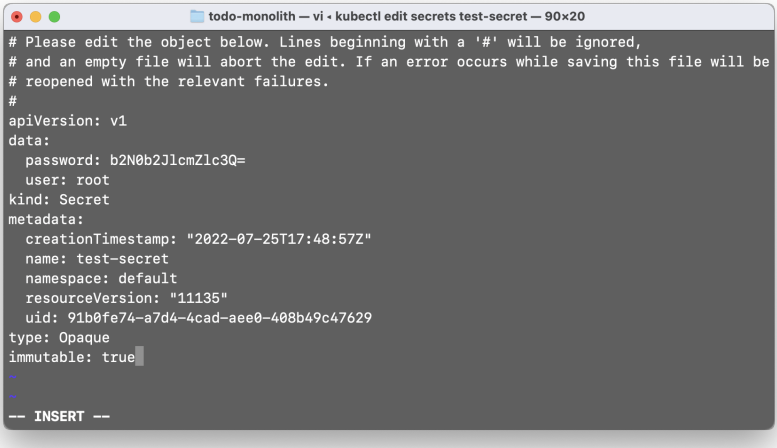
Encoding vs. encryption

A moment ago, I suggested that you copy the base64-encoded password. Try running the following command:

```
% echo 'b2N0b2JlcmZlc3Q=' | base64 --decode
```

Remember, anyone who gets hold of your encoded Secret can run that command, too. This is encoding, not encryption.

Now, it's easy to imagine that you might not *want* all of your Secrets to be readily editable, not least because someone might accidentally modify them. In such a case, you can make a Secret **immutable**. To do this, simply add an `immutable` field set to `true` in the Secret file:



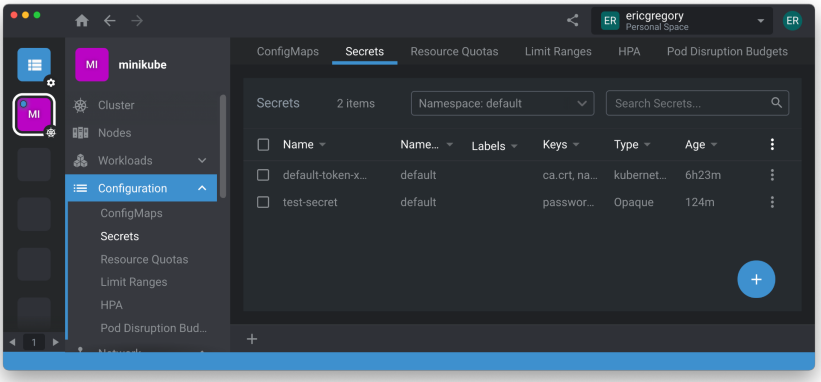
Note: Once you save the file, this particular Secret will no longer be editable (as per your request!). To make any changes, you'll need to delete this Secret and create a new one.

Immutability not only protects a Secret from accidental modification; it can also help improve performance across your cluster, since an operator can allow the API server to stop watching for changes to immutable Secrets.

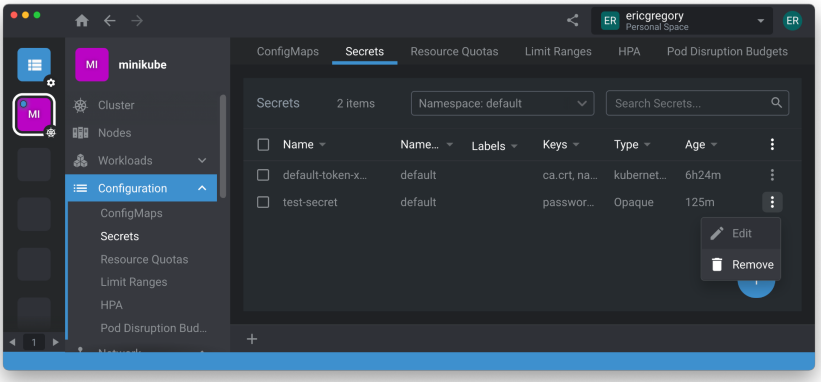
Using Lens to manage Secrets

Now, we've seen Secret creation and management through `kubectl`, but that's not the only way to get the job done. Many users will find it easier to use a graphical user interface.

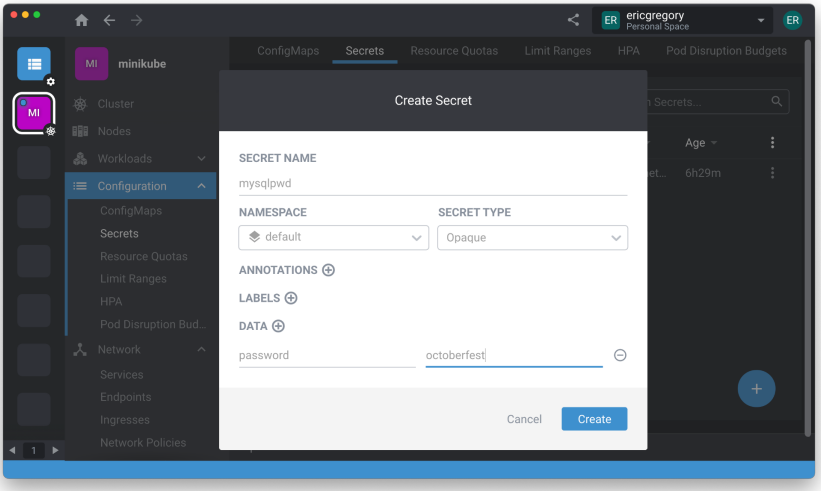
Back in Chapter 2, we downloaded and learned to use **Lens**, which serves as a GUI and management environment for Kubernetes. Let's open Lens now, connect to our Minikube cluster, and navigate to the **Secrets** tab (under **Configuration**).



Here we can see a default token used by the system and the now-immutable `test-secret` we created. Click on the **vertical ellipsis icon** (⋮) for `test-secret` and select **Remove** to delete the Secret.



Now let's create a new Secret. Click the blue plus (+) icon in the lower right corner. Name your new Secret **`mysqlpwd`**. Click the plus icon beside **Data** to add a new field, with **`password`** as the name and **`octoberfest`** as the value. Click **Create**.



Now the Secret is saved in our cluster. If you return to the command line and run `kubectl get secrets`, you'll see `mysqlpwd` there with one datum, the password.

```
% kubectl get secrets
```

```
NAME                  TYPE                                  DATA   AGE
default-token-x5n8f   kubernetes.io/service-account-token   3      6h35m
mysqlpwd              Opaque                                1      2m2s
```

Exercise: Keep it Secret, Keep it Safe-ish

Now it's time to use this Secret. To get our MySQL server running, we're going to use a single, lengthy YAML file that synthesizes much of what we've learned over the past few lessons. (You can find a copy of the YAML file on GitHub[16].) Create a new directory for this chapter called `/09-secrets/` and a subdirectory called `/manifests/`, then inside create a new YAML file called `todo-mysql.yml` with the code below.

There's a lot going on here, but don't worry—we'll go through the manifest step-by-step:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
 name: sc-local
provisioner: k8s.io/minikube-hostpath
parameters:
 {}
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: false

---

apiVersion: v1
kind: Service
metadata:
 name: todo-mysql
```

[16] https://github.com/ericgregory/kube-5mins/blob/main/09-secrets/manifests/todo-mysql.yml

```
 labels:
   app: todo-mysql
spec:
 type: ClusterIP
 selector:
   app: todo-mysql
 ports:
   - port: 3306
     protocol: TCP
 clusterIP: "None"

---
apiVersion: apps/v1
kind: StatefulSet
metadata:
 name: todo-mysql
spec:
 selector:
   matchLabels:
     app: todo-mysql
 serviceName: todo-mysql
 replicas: 1
 template:
   metadata:
     labels:
       app: todo-mysql
   spec:
     terminationGracePeriodSeconds: 10
     containers:
     - name: todo-mysql
       image: mysql:8
       ports:
       - containerPort: 3306
```

```
        name: todo-mysql
      env:
      - name: MYSQL_ROOT_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysqlpwd
            key: password
      - name: MYSQL_DATABASE
        value: "todo_db"
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: todo-volume
volumeClaimTemplates:
- metadata:
    name: todo-volume
  spec:
    storageClassName: sc-local
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 1Gi
```

- The first thing to note is that **we're defining multiple Kubernetes resources in a single manifest**, separating the individual specifications with three dashes. That's a good option to have—it gives us the choice to define related or inter-dependent resources all at once.
- We're defining a **StorageClass** first, exactly as we did last lesson.
- Next we define a simple **Service** for our MySQL server as we've done previously.

- Next we're adding a new kind of resource: an API resource called a **StatefulSet**.

Don't worry too much about the StatefulSet at the moment—it's a big, important topic that we're going to explore in depth in the next lesson. For now, just note that it is an API resource roughly analogous to a Deployment. The `container` spec in the StatefulSet manifest is similar to the container spec in a Deployment manifest, and we should zoom in on it for a moment because there's some important work going on here to implement our Secret:

```
...
    containers:
     - name: todo-mysql
       image: mysql:8
       ports:
       - containerPort: 80
         name: tcp
       env:
       - name: MYSQL_ROOT_PASSWORD
         valueFrom:
           secretKeyRef:
             name: mysqlpwd
             key: password
       - name: MYSQL_DATABASE
         value: "todo_db"
       volumeMounts:
       - mountPath: /var/lib/mysql
         name: todo-volume
...
```

We're using the `env` field to define environment variables for `MYSQL_DATABASE` and `MYSQL_ROOT_PASSWORD`, which the MySQL container image is designed to utilize. We've defined the values of those variables in two different ways—directly, in the case of `todo_db`, and through our `mysqlpwd` Secret for the root password. We've defined *that* with the `secretKeyRef` field, the value of which is an object identifying the Secret's name and the data from the Secret that we wish to use.

Let's use this YAML manifest to create our Storage Class, Service, and StatefulSet all at once. From the `/manifests/` directory where our new `todo-mysql.yml` file is stored:

```
% kubectl apply -f todo-mysql.yml
```

Mounting Secrets from volumes

Next we'll launch our To Do app's backend service. The backend will access the MySQL server using the root password, which it will consume via the Secret mounted from a volume.

There's no need to completely decompose our monolith into frontend and backend at this stage. At this point, we'll stand up a skeleton to perform the bare essentials before moving on to the next steps of decomposition. For our purposes here, the essentials include:

- Mounting a Secret from a volume
- Connecting to and authenticating with the MySQL server

In your `/09-secrets/` directory, create a new folder called `/5min-dbcheck/`. Within that directory, run:

```
% npm init -y
% npm install --save mysql2
```

In your `index.js` file, add the following:

```
var mysql = require('mysql2');
var fs = require('fs');

const MYSQLPWD =
fs.readFileSync("secrets/password", 'utf8');

var con = mysql.createConnection({
   host: "todo-mysql",
   user: "root",
   password: `${MYSQLPWD}`
});

var minutes = 5;
var interval = minutes * 60 * 1000;

setInterval(function() {
 console.log("Performing 5 minute check...");
 con.connect(function(err) {
   if (err) throw err;
   console.log("Connected!");
 });
}, interval);
```

- We're using Node's `fs` (fileSync) module to read a password file that will soon be stored in a `/secrets/` directory within container instances of the application.
- The value of the fileSync is assigned to the constant `MYSQLPWD`, which we're interpolating into the password field for our MySQL connection configuration.
- We're setting the host field in our MySQL connection configuration to the service hostname of our MySQL server: `todo-mysql`
- Finally we're creating a function that runs at five minute intervals, connecting to the database and producing a console message if successful

Now let's get this running on the cluster. If you'd like to use my code or container image, they are available on GitHub[17] and Docker Hub[18] respectively. To build an image locally, use the standard Dockerfile we've used for Node applications so far:

```
# Sets the base image
FROM node:18
# Establishes the working directory for your app
within the container
WORKDIR /usr/src/app
# Copies your package.json file and then installs
modules
COPY package*.json ./
RUN npm install
```

[17] https://github.com/ericgregory/kube-5mins/tree/main/09-secrets/5min-dbcheck
[18] https://hub.docker.com/repository/docker/ericgregory/5min-dbcheck

```
# Copies your project files and then runs the app
COPY . .
CMD [ "node", "index.js" ]
```

Build the container image and then upload it to Docker Hub:

```
% docker build . -t <Docker ID>/5min-dbcheck
% docker push <Docker ID>/5min-dbcheck
```

Return to the root project directory for this chapter, where you've been storing your manifests. Create a new Deployment manifest called `5min-dbcheck.yml` that contains the following:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 labels:
   app: 5min-dbcheck
 name: 5min-dbcheck
spec:
 replicas: 1
 selector:
   matchLabels:
     app: 5min-dbcheck
 template:
   metadata:
     labels:
       app: 5min-dbcheck
   spec:
     containers:
     - image: ericgregory/5min-dbcheck:latest
```

```
    name: 5min-dbcheck
    ports:
    - containerPort: 80
    volumeMounts:
    - name: secrets
      mountPath: "usr/src/app/secrets"
  volumes:
    - name: secrets
      secret:
        secretName: mysqlpwd
        defaultMode: 0400
```

- For the image in the `container` spec, replace my name with your own Docker ID (or simply use my image).
- Everything here should be familiar except for the `volumeMounts` field for the `container` spec and the particulars of the `volumes` spec.
    - The `name` field under `volumeMounts` identifies the storage volume that will be mounted to this container (pointing to the volume defined in the `volumes` spec below).
    - The `mountPath` identifies where in the container filesystem the volume will be mounted.
- In the `volumes` spec…
    - We name our new volume.
    - The `secret` field defines this volume as the bespoke, RAM-backed volume type dedicated to Secrets.

- We refer to the Secret we want to use by name and then define its `defaultMode` as `0400` (the Linux permission notation for Read access by owner), which means the Secret file(s) created by the volume mount will have a permission of `0400`.

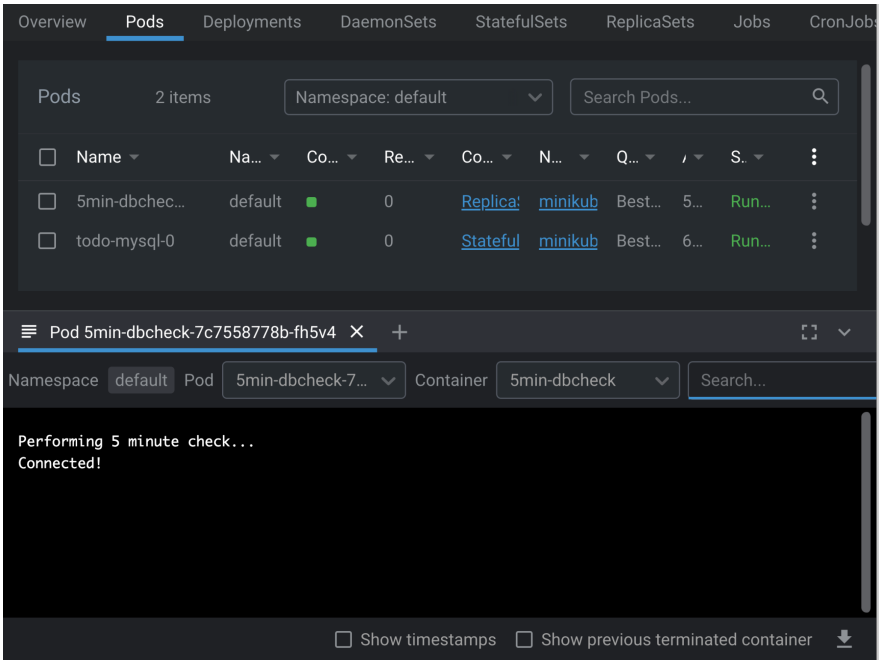
Launch the new deployment:

```
% kubectl apply -f 5min-dbcheck.yml
```

Now we can check our new app's logs to verify whether it has mounted the Secret and connected to the database server successfully. We can do this either using `kubectl exec` as before or by using the **Logs** button in Lens.

I'll use Lens—under the **Workloads > Pods** tab, select the 5min-dbcheck Pod. At the top of the Pod details pane, select the **Logs** icon.

When the app conducts its connectivity check, you should see "Connected!"



We've accomplished quite a bit this lesson:

- Using Secrets as environment variables
- Using Secrets as volume mounts

Next time, we'll learn more about the StatefulSet API resource and how it acts as a manager for stateful applications, and then learn how to create a scalable database server.

Chapter 10

StatefulSets, Custom Resources, and Operators

IN THIS CHAPTER...

- Understand the StatefulSet resource type, how it works, and why it's such an important resource for running stateful components like databases
- Deploy a scalable MySQL server using a custom resource and operator

The StatefulSet API resource is an abstraction for managing stateful applications on Kubernetes. It is roughly analogous to a Deployment, but tailored to *stateful* rather than stateless processes.

In the last lesson, we skipped over the whys and wherefores of StatefulSets to focus on implementing Secrets. Today, we'll take a look at some of the same YAML markup, but zero in on StatefulSets instead.

In a new directory for this lesson named `/10-statefulsets/manifests/`, create a YAML file called `todo-mysql.yml` and copy the manifests below. You can also find a copy in the GitHub repository[19] for this lesson.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
```

[19] https://github.com/ericgregory/kube-5mins/blob/main/10-statefulsets/manifests/todo-mysql.yml

```
metadata:
 name: sc-local
provisioner: k8s.io/minikube-hostpath
parameters:
 {}
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: false

---

apiVersion: v1
kind: Secret
metadata:
 name: mysqlpwd
data:
 password: b2N0b2JlcmZlc3Q=

---

apiVersion: v1
kind: Service
metadata:
 name: todo-mysql
 labels:
   app: todo-mysql
spec:
 type: ClusterIP
 selector:
   app: todo-mysql
 ports:
   - port: 3306
     protocol: TCP
```

```
 clusterIP: "None"

---
apiVersion: apps/v1
kind: StatefulSet
metadata:
 name: todo-mysql
spec:
 selector:
   matchLabels:
     app: todo-mysql
 serviceName: todo-mysql
 replicas: 1
 template:
   metadata:
     labels:
       app: todo-mysql
   spec:
     terminationGracePeriodSeconds: 10
     containers:
     - name: todo-mysql
       image: mysql:8
       ports:
       - containerPort: 3306
         name: todo-mysql
       env:
       - name: MYSQL_ROOT_PASSWORD
         valueFrom:
           secretKeyRef:
             name: mysqlpwd
             key: password
       - name: MYSQL_DATABASE
         value: "todo_db"
```

```
        volumeMounts:
        - mountPath: /var/lib/mysql
          name: todo-volume
  volumeClaimTemplates:
  - metadata:
      name: todo-volume
    spec:
      storageClassName: sc-local
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

This mega-manifest includes all of the Kubernetes resources for our To Do app's **MySQL server**. It differs from the similar YAML file we used previously only in that it includes a definition for the `mysqlpwd` Secret. The last section of the manifest defines our **StatefulSet**.

The StatefulSet manifest should feel familiar—it looks a lot like a Deployment manifest! Instead of the `volume` field under a Deployment's `template` spec, we define a `VolumeClaimTemplate` under the overall StatefulSet spec to describe how the workload will consume storage. In this case, that means connecting to our `sc-local` Storage Class, specifying a ReadWriteOnce access mode, and defining a one gibibyte storage request.

Take note of the fact that we're defining a *template* for PersistentVolumeClaims here, which Kubernetes will use to

create a unique PVC for each Pod managed by the StatefulSet. That will be important later.

Getting a StatefulSet up and running

Make sure Minikube is running, then from the directory where you've saved `todo-mysql.yml`, run:

```
% kubectl apply -f todo-mysql.yml
```

As soon as the resources are created, you should be able to see them in Lens. You can check **Pods** under the **Workloads** menu to see if your automatically generated `todo-mysql-0` Pod is running yet, or check with:

```
% kubectl get pods
```

Note the name of the pod: `todo-mysql-0`. This is a key difference between a StatefulSet and a Deployment:

- The **StatefulSet** maintains a primary pod at ordinal number 0, using a predictable name that applications can connect to reliably if needed. Replicas are duplicated from this primary pod to facilitate an orderly flow of persistent data.
- **Deployments**, by contrast, use randomized addenda to their names for individual instances, and are not designed with persistent data in mind.

Now we'll use `kubectl exec` to ensure that the StatefulSet is working correctly. Since our pod is named using an ordinal, we can jump right into it without searching for its name:

```
% kubectl exec --stdin --tty todo-mysql-0 --
/bin/bash
```

Now that we're inside the container, start MySQL:

```
bash4.4# mysql -u root -p
```

Enter our root password:

```
octoberfest
```

From here, we can access the database we created:

```
mysql> USE todo_db;
```

Now let's create the database table that our To Do app will use:

```
mysql> CREATE TABLE IF NOT EXISTS Todo (task_id
int NOT NULL AUTO_INCREMENT, task VARCHAR(255)
NOT NULL, status VARCHAR(255), PRIMARY KEY
(task_id));
```

Press return.

```
Query OK, 0 rows affected (0.04 sec)
```

We can check to confirm that our table has been created:

```
mysql> SHOW TABLES;
```

```
+------------------+
| Tables_in_todo_db |
+------------------+
| Todo             |
+------------------+
1 row in set (0.00 sec)
```

Now let's manually add an item to the table—in this case, a simple 'hello' message:

```
mysql> INSERT INTO Todo (task, status) VALUES
('Hello','ongoing');
```

Check the table for the new item:

```
mysql> SELECT * FROM Todo;
```

```
+---------+-------+---------+
| task_id | task  | status  |
+---------+-------+---------+
|       1 | Hello | ongoing |
+---------+-------+---------+
1 row in set (0.00 sec)
```

Exit MySQL and the container by typing `exit` twice.

Question time: if we create a replica, do you think our data will persist across replicas of the pod? Let's see! We can add replicas imperatively using kubectl:

```
% kubectl scale statefulsets todo-mysql
--replicas=2
```

```
% kubectl get pods
```

```
NAME           READY   STATUS    RESTARTS   AGE
todo-mysql-0   1/1     Running   0          11m
todo-mysql-1   1/1     Running   0          6s
```

Now let's hop into the new replica.

```
% kubectl exec --stdin --tty todo-mysql-1 --
/bin/bash
bash-4.4# mysql -u root -p
Enter password: octoberfest
mysql> USE todo_db;
mysql> SHOW TABLES;
```

```
Empty set (0.00 sec)
```

If you guessed that the data wouldn't persist across pods, well done. Go ahead and exit, then delete the running StatefulSet and Service.

```
mysql> exit
bash-4.4# exit
% kubectl delete statefulset todo-mysql
% kubectl delete service todo-mysql
```

Let's talk about *why* the data isn't persisting. One simple reason is the design of the StatefulSet.

When we created a new replica, Kubernetes used the volumeClaimTemplate in the StatefulSet spec to create a new PersistentVolumeClaim and bind a newly apportioned chunk of

storage. In practical terms, this is a new database instance with the same credentials and pre-set todo_db database as the first instance.

"Replica" is a bit of a misnomer for this new pod. It's a **unique instance**, with **unique storage** and a **unique hostname**. Moreover, the volume claim is *renewable*, and the hostname is *predictable* on account of the appended ordinal number. If this pod should crash, a new one will be generated with the same ordinal number, and that new instance will be able to take up the existing PVC, so that the pod has access to the same data.

In short, the pod achieves **stable identity** in terms of both storage and networking.

This is all by design. The StatefulSet is meant to serve as a building block for statefulness on Kubernetes. By enabling us to manage **predictably unique** workloads, the StatefulSet gives us a key ingredient for scaling stateful applications, including databases. We shouldn't expect to run a production-grade database with a StatefulSet alone, but it provides a building block and common abstraction for stateful applications.

Horizontally scalable databases

In the context of databases, "horizontal scaling" means expanding capacity across multiple nodes or instances rather than "vertically" raising the ceiling on one big node.

Distributed databases tend to use a couple of patterns—often in conjunction—to scale data stores horizontally:

- **Replication**: in which the database copies an entire data-set and coordinates updates among the replicas. The primary goals here are to reduce latency, if a given database instance is subject to many simultaneous requests, and to foster resilience. This might be used for smaller volumes of data or to facilitate back-ups.
- **Sharding**: in which the database divides the data-set between a number of database nodes—or "shards"—and coordinates queries between them. Sharding is a different approach to increasing speed and a way to more efficiently store large data-sets that may be costly or difficult to store on one node, let alone multiple replicas. This is the typical approach for large distributed data-sets.

Many databases, including MySQL, support **manual sharding**—handcrafting the joins between different sections of the data-set—but that's exactly as excruciating as it sounds. No matter how our cloud native database handles distributed data, we *definitely* want it to operate in an automated way.

Here are a few popular open source databases used widely with Kubernetes:

- **CockroachDB**: A distributed SQL database designed specifically for Kubernetes, focusing on resilience, scalability, and ease of use within Kubernetes. Open

source and available self-hosted or managed through creator Cockroach Labs, this is considered a "NewSQL" database in that it combines an old-school relational and SQL-compatible approach with new-school distributed and cloud native architecture.

- **Cassandra**: While it precedes Kubernetes (and therefore isn't tailored to it specifically), Apache's Cassandra is a distributed "NoSQL" (non-relational, or "not only SQL") database system with a mature set of drivers, designed to pass data quickly between its own database nodes. Cassandra may be self-hosted or used as a service managed by a variety of vendors including the major public cloud providers.
- **MongoDB**: MongoDB is a NoSQL database that precedes Kubernetes and is designed to use JSON-like data objects. MongoDB can be self-hosted or managed by MongoDB using the MongoDB Atlas service, or as a managed offering from other vendors.

In their Kubernetes implementations, all of these databases are built on StatefulSets, and all of them require some additional logic to work as intended. But as I said before, a StatefulSet isn't sufficient to run a production-grade database—especially not one that needs to juggle concurrent requests across instances and ensure data fidelity. For those kinds of complex tasks, Kubernetes relies on two important avenues of extensibility: custom resources and operators.

Introducing custom resources and operators

The Kubernetes creed—declarative portability for All the Things!—dictates that we'd really like our extensions to work as standard, portable Kubernetes resources. You can define custom resources with **Custom Resource Definitions** (CRDs), which are simply YAML manifests for creating new Kubernetes resource types.

Suppose we want to replicate instances of the MySQL server to reduce latency. This is only possible for reads—if we try to do the same thing for writes, we introduce problems like **write skew**, wherein two transactions simultaneously attempt to change a value. But we can accomplish our goal with the help of a CRD and an operator.

Today, we're going to use a CRD developed by Oracle, owners and sponsors of MySQL. Ultimately, this will work together with an operator and use the StatefulSet to manage pods with a primary-secondary model—enabling us to scale horizontally. You can take a look at the entire CRD manifest[20] if you wish, but we'll just peek at a representative snippet:

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
 name: innodbclusters.mysql.oracle.com
spec:
 group: mysql.oracle.com
 versions:
```

[20] https://raw.githubusercontent.com/mysql/mysql-operator/trunk/deploy/deploy-crds.yaml

```
    - name: v2
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          required: ["spec"]
          properties:
            metadata:
              type: object
              properties:
                name:
                  type: string
                  maxLength: 40
            spec:
              type: object
              required: ["secretName"]
              properties:
                secretName:
                  type: string
                  description: "Name of a generic
type Secret containing root/default account
password"
            ...
```

After defining the metadata for this new resource called `innodbclusters.mysql.oracle.com`, the spec goes on to start defining the properties that future manifest-writers will use to define their own instances of this resource. I've only excerpted one property out of the many included in the full file: `secretName`, which accepts a string as its value. The definition

here also includes a description that can be accessed via `kubectl describe`, Lens, or other Kubernetes API clients.

Let's apply our custom resource—not from our local machine, this time, but from the web:

```
% kubectl apply -f
https://raw.githubusercontent.com/mysql/mysql-o
perator/trunk/deploy/deploy-crds.yaml
```

Next, we're going to deploy the **MySQL Operator for Kubernetes**[21], also developed by Oracle. An **operator** is an application that runs on the cluster and manages a custom resource. This is ultimately just an application pattern intended to automate some of the tasks of a human operator. This operator is built out of a collection of Kubernetes resources including a deployment running the `mysql-operator` container image. You can find operators for other database systems at Operator Hub[22], which serves as a central location for finding Operators of all sorts, including but not limited to databases.

Apply the operator manifest as well, and then check that it's running:

```
% kubectl apply -f
https://raw.githubusercontent.com/mysql/mysql-o
perator/trunk/deploy/deploy-operator.yaml
```

[21] https://raw.githubusercontent.com/mysql/mysql-operator/trunk/deploy/deploy-operator.yaml
[22] https://operatorhub.io/?category=Database

```
% kubectl get deployment mysql-operator
--namespace mysql-operator
```

```
NAME             READY   UP-TO-DATE   AVAILABLE   AGE
mysql-operator   1/1     1            1           77s
```

With our CRD and operator in place, it's time for another manifest to deploy our set of MySQL servers. Let's see how much of a mega-manifest we're working with this time:

```
apiVersion: v1
kind: Secret
metadata:
 name: mysqlpwd
data:
 rootHost: JQ==
 rootPassword: b2N0b2JlcmZlc3Q=
 rootUser: cm9vdA==

---

apiVersion: mysql.oracle.com/v2
kind: InnoDBCluster
metadata:
 name: todo-mysql
spec:
 secretName: mysqlpwd
 instances: 3
 router:
   instances: 1
 tlsUseSelfSigned: true
```

Hey, that's downright manageable! Since our StorageClass is already in place, we're defining two resources:

- Our Secret, which we're revising slightly to the format expected by the custom resource. I have the credential fields base64-encoded here.
- An instance of the custom resource `InnoDBCluster`, which abstracts away all of its constituent components. (We'll take a peek at some of those in a moment.)

Meanwhile, we only need to specify values for a handful of InnoDBCluster's fields: the Secret name, how many database server instances we want, and how many routers we want. The router will help coordinate traffic between the three replicas: to ordinal zero (`todo-mysql-0`) when a write comes over the transom, and to any ordinal for a read. This is the primary-secondary model discussed earlier, and the StatefulSet's unique, predictable hostnames make it possible.

Copy the manifest above into a file called `inno.yml` in your `/stateful/` project directory. (Alternatively, it's available on GitHub[23].) Then apply the manifest:

```
% kubectl apply -f inno.yml
```

If you wish, you can watch the database server instances spin up from the command line:

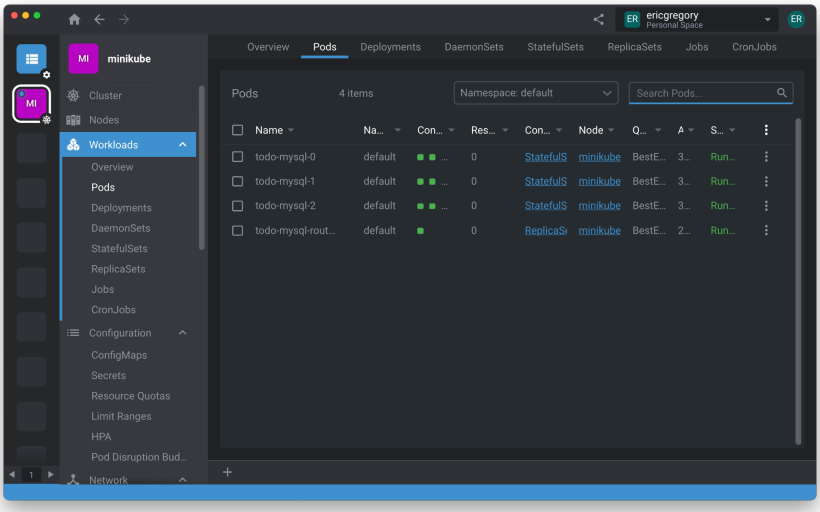
```
% kubectl get innodbcluster --watch
```

[23] https://github.com/ericgregory/kube-5mins/blob/main/10-statefulsets/manifests/inno.yml

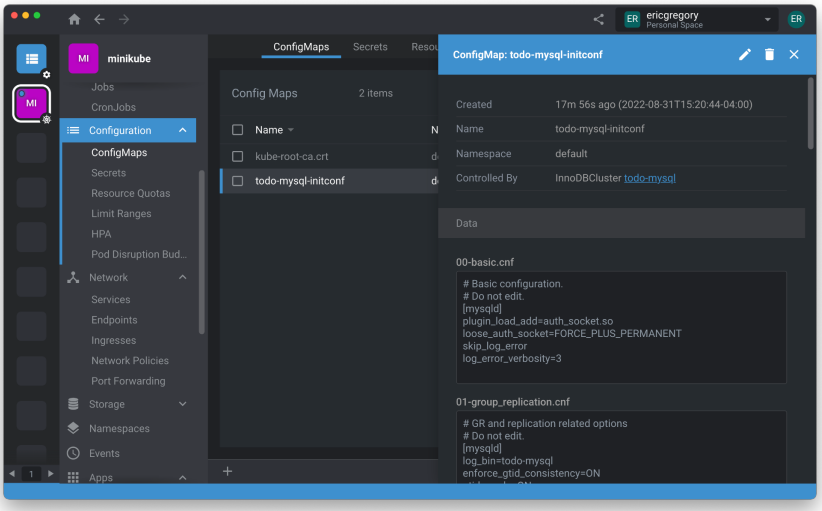
It'll take a moment, but all instances should reach online status before long:

```
NAME          STATUS   ONLINE   INSTANCES   ROUTERS   AGE
todo-mysql    ONLINE   2        3           1         48s
todo-mysql    ONLINE   2        3           1         53s
todo-mysql    ONLINE   3        3           1         58s
```

Once the instances are up, open Lens and take a look at your pods. You'll see we have three database server replicas (managed by a StatefulSet) and one stateless database router.



Let's have a look at a couple more components to get a sense for how this contraption is working. Under **Configuration** in the Lens menu, select **ConfigMaps** and select the entry called `todo-mysql-initconf`.



The ConfigMap API resource is an abstraction that gives us a way to separate configuration details from their subject container images. A Pod or Deployment or StatefulSet can then access the details from the ConfigMap through an environment variable, volume-mounted config file, or command line argument.

This is useful especially when configuration details might be re-used—or when they might differ according to the circumstances. When Pods in the same StatefulSet (or Deployment) initialize, they can use these configuration files conditionally, and this is another place where a StatefulSet's fixed ordinals—which initialize one-by-one, in sequence—can come in really handy. As a `todo-mysql` pod comes online, it can check to see where it stands in the ordinal line-up; if it's first, it can use the configuration files associated with a writable

primary instance. If it's later in the sequence, it can configure itself as a read-only replica.

But does our horizontally-scaling set of database servers actually work as intended? Let's see! Once again, we will:

- Hop into the ordinal-zero pod
- Log in to MySQL
- Create a Todo table in the todo_db database
- Add an item in the table

Working with multi-container pods

Note that the opening `kubectl exec` command looks a little different than before. That's because we're jumping into a **multi-container pod**, so we need to specify which container we want to access with the `-c` argument. In Lesson 2, I mentioned that some applications break the one-container-per-pod ideal and use "sidecar" containers to manage some extra bit of functionality. Here's one such pod in the wild.

```
% kubectl exec --stdin --tty -c mysql
todo-mysql-0 -- /bin/bash
bash4.4# mysql -u root -p
Enter password: octoberfest
mysql> CREATE DATABASE IF NOT EXISTS todo_db;
mysql> USE todo_db;
mysql> CREATE TABLE IF NOT EXISTS Todo (task_id
int NOT NULL AUTO_INCREMENT, task VARCHAR(255)
```

```
NOT NULL, status VARCHAR(255), PRIMARY KEY
(task_id));
mysql> INSERT INTO Todo (task, status) VALUES
('Hello','ongoing');
mysql> SELECT * FROM Todo;
```

```
+---------+-------+---------+
| task_id | task  | status  |
+---------+-------+---------+
|       1 | Hello | ongoing |
+---------+-------+---------+
1 row in set (0.00 sec)
```

Exit MySQL and the container by typing `exit` twice. Moment of truth—let's see if our write has carried over to a replica.

```
% kubectl exec --stdin --tty -c mysql
todo-mysql-1 -- /bin/bash
bash4.4# mysql -u root -p
Enter password: octoberfest
mysql> USE todo_db;
mysql> SELECT * FROM Todo;
```

```
+---------+-------+---------+
| task_id | task  | status  |
+---------+-------+---------+
|       1 | Hello | ongoing |
+---------+-------+---------+
1 row in set (0.00 sec)
```

Excellent!

This kind of simple, wholesale replication is by no means a standard approach to scaling databases—but it's *one* use-case for a StatefulSet, and it gives us some key insight into the resource's uses and workings. The StatefulSet is a deceptively powerful resource with enormous flexibility:

- It can drive full data-store replication via sequential pod initialization, as with the InnoDB CRD and MySQL Operator.
- It can be used to facilitate sophisticated architectures in which each pod serves as a unique node in a database cluster, as in the case of CockroachDB.

Medium and large organizations—the ones for which Kubernetes is best-suited—will be using many different database systems for many different ends. The StatefulSet is a key API resource that makes each of those different approaches possible.

That brings us to a close for today. Run `exit` twice and shut down your cluster with `minikube stop`. (We'll use the running resources next time, and they should launch again when you restart Minikube.) Next time, we'll use what we've learned here to finalize the decomposition of our monolithic To Do app into a scalable, cloud native architecture.

Chapter 11
A Stateful Web App at Scale

IN THIS CHAPTER...

– Complete the decomposition of the To Do app
– Run the decomposed To Do app on Kubernetes
– Understand the Ingress and Gateway resource types

This lesson will conclude our decomposition project. By the end, we'll have an application broken out into three services: a MySQL server, an API server, and a web interface.

We're not decomposing and Kubernetizing this app for kicks; we're migrating to this substrate for its availability and resilience. If demand for our application spikes, Kubernetes can provision (and then load balance) replicas to handle the load and avoid a bottleneck. Breaking an application down into independent functionalities can make the process faster and more efficient by *only* provisioning the specific components under demand.

In order to understand scaling, we need to explore how Kubernetes manages incoming traffic and how it runs applications at scale.

Our agenda for this lesson, then, is as follows:

- Decompose and deploy the To Do app
    - Deploy the MySQL server and storage resources
    - Build the API server
    - Build the web client
    - Deploy the services to Kubernetes
- Use the Horizontal Pod Autoscaler
- Understand the Ingress and Gateway resources

Deploying the MySQL server and storage resources

Start Minikube and check that your MySQL pods from the previous lesson are running on the cluster. As a reminder, we're using the InnoDBCluster custom resource and MySQL Kubernetes Operator to run a horizontally-scalable set of replica MySQL servers.

```
% minikube start
% kubectl get pods
% kubectl get pods
```

```
NAME                                READY   STATUS    RESTARTS        AGE
todo-mysql-0                        2/2     Running   4 (4m ago)      19h
todo-mysql-1                        2/2     Running   4 (3m12s ago)   19h
todo-mysql-2                        2/2     Running   4 (3m12s ago)   19h
todo-mysql-router-8445df87bd-6sjjh  1/1     Running   7 (116s ago)    19h
```

(If the Pods are not present, follow the instructions under **Introducing custom resources and operators** in the previous chapter and then return here.)

The stateful components of our web app should be up and running. Now let's connect the API server and the web client.

Building the API server

First, we'll implement our API server. In our new services-based architecture, this component will be responsible for handling requests from the web client and interfacing with the database accordingly. For our To Do app, that means handling requests to…

- Read the tasks from the database
- Add a new task
- Update the status of a task
- Delete a task

Create a directory called `/11-scale/` for this chapter, and then a new subdirectory called `/todo-api/`. Initialize the project:

```
% npm init -y
```

This creates a `package.json` file that holds the organizing metadata for your app. Now create a file called `index.js` and add the following:

```
const express = require('express');
const bodyParser = require('body-parser');
const con = require('./models/taskModel');
```

```
const app = express();

app.use(bodyParser.urlencoded({
   extended: true
}));

app.use(bodyParser.json());

// Read tasks handler

app.get('/', (req, res) => {
   let query = `SELECT * FROM Todo`;
   let items = []
   con.execute(query, (err, result) => {
       if (err) throw err;
       items = result
       console.log(items)
       res.json(items);
       })
   });

// Status update handler

app.post('/update', (req, res) => {
   let intCheck = req.body.id * 1;
   if (Number.isInteger(intCheck)) {
   console.log(req.body)
   let query = "UPDATE Todo SET status='" +
req.body.status + "' WHERE task_id=" +
req.body.id;
   con.execute(query, (err, result) => {
       if (err) throw err;
```

```
        console.log(result)
    });
} else {
    console.log('There was a problem');
};
});

// Delete handler

app.post('/delete', (req, res) => {
let intCheck = req.body.id * 1;
if (Number.isInteger(intCheck)) {
    console.log(req.body)
        let query = "DELETE FROM Todo WHERE
task_id=" + req.body.id
        con.execute(query, (err, result) => {
          if (err) throw err;
          console.log(result);
    })
} else {
    console.log('There was a problem');
}
});

// New task handler

app.post('/', (req, res) => {
    console.log(req.body.task);
    let query = "INSERT INTO Todo (task, status)
VALUES ?";
    data = [
        [req.body.task, "ongoing"]
    ]
```

```
    con.query(query, [data], (err, result) => {
        if (err) throw err;
        console.log(result);
    })
});

// port where app is served

app.listen(80, () => {
    console.log('The API server has started on
port 80');
});
```

You may wish to compare this to the `index.js` file in the GitHub repo for the original monolith[24]. While the structure is quite similar, you'll notice some important differences:

- We're using fewer modules, since we no longer need to worry about rendering the web client.
- Several routes are now simple, task-specific addresses: `/update` to update, `/delete` to delete.

As in the monolith, we've broken out the database connection into a separate module called `taskModel`. We've referred to it in our configuration at the top of `index.js`—now we need to create it. Make a new directory within `/todo-api/` called `/models/` and a new file there called `taskModel.js`.

[24] https://github.com/ericgregory/todo-monolith

In the last chapter, we created a skeleton for this API server with a simple Node app that connected to the database and verified the connection. You can copy the code from that skeleton to `taskModel.js` and add the following changes:

```
var mysql = require('mysql2');
var fs = require('fs');

const MYSQLPWD =
fs.readFileSync("secrets/password", 'utf8');

var con = mysql.createConnection({
 host: "todo-mysql",
 user: "root",
 password: `${MYSQLPWD}`,
 database: "todo_db"
});

con.connect(function (err) {
   if (err) throw err;
   console.log("Connected to the database!");
   let query = "CREATE TABLE IF NOT EXISTS Todo
(task_id int NOT NULL AUTO_INCREMENT, task
VARCHAR(255) NOT NULL, status VARCHAR(255),
PRIMARY KEY (task_id))";
   con.query(query, (err, result)=>{
     if (err) throw err;
     console.log(result)
 })
});

module.exports = con;
```

We didn't have to change much here:

- We specified a database in the database connection configuration on line 10.
- On lines 13-21, we connect to our database and ensure that the `Todo` table exists
- Line 23 exports the module so the rest of the app can use it

From the `/todo-api/` directory:

```
% npm i mysql2 express body-parser
```

This command uses the npm package manager to install three modules in your project: the `mysql2` driver, the `express` web app framework, and `body-parser`, which will help us parse JSON. (As an alternative to the command above, you can simply include those three modules as dependencies in your `package.json` file.)

In the same directory, we'll create our standard Dockerfile for Node apps:

```
# Sets the base image
FROM node:18
# Establishes the working directory for your app
within the container
WORKDIR /usr/src/app
```

```
# Copies your package.json file and then installs
modules
COPY package*.json ./
RUN npm install
# Copies your project files and then runs the app
COPY . .
CMD [ "node", "index.js" ]
```

Build the container image and then upload it to Docker Hub:

```
% docker build . -t <Docker ID>/todo-api
% docker push <Docker ID>/todo-api
```

That's it for our API server. We're two-thirds of the way through our decomposition. Let's tackle the web client.

Building the web client

Structurally, the web client will look quite similar to the `index.js` file of the monolith, or the Randomreads web client from Chapter 7. It will use the `express` and `express-handlebars` modules to serve the website, and it will use a combination of `express` and the Fetch API to route HTTP requests to the API server.

In your `/11-scale/` project directory, create a new directory called `/todo-web/`. Initialize the project:

```
% npm init -y
```

Create an `index.js` file and add the following:

```
const express = require('express');
const { engine } = require('express-handlebars');
const bodyParser = require('body-parser');
const ENDPOINT = 'todo-api'
const app = express();

app.use(express.static('public'));

app.engine('handlebars', engine({
   helpers: {
       isCompleted: function (status) {
           if (status == "completed") {
               return true
           } else {
               return false
           }
       },
   },
   defaultLayout: 'main',
}));
app.set('view engine', 'handlebars');
app.set('views', './views');

app.use(bodyParser.urlencoded({
   extended: true
}));

app.use(bodyParser.json());
```

The lines above will include and initialize our major module dependencies like `express` and `express-handlebars`.

If you compare this to the original monolithic `index.js`, you'll see that here we establish an ENDPOINT constant up top and assign it the hostname we will use for the API server. Now we can simply interpolate the constant whenever we want to communicate with the API server (and if we need to change the hostname, we only need to change it once).

Now add the following below the previous lines in `index.js`:

```
// Upon GET request, send a GET request to the
API server

app.get('/', async (req, res) => {
   fetch(`http://${ENDPOINT}/`)
       .then(response => response.json())
       .then(data => res.render('index', {
           items: data
       }
       ));
});

// Redirect POST requests to the API

app.post('/', (req, res) => {
   const data = { task: `${req.body.task}` };
   fetch(`http://${ENDPOINT}/`, {
       method: 'POST',
       body: JSON.stringify(data),
```

```
        mode: 'cors',
        headers: {
            'Content-Type': 'application/json',
        },
    })
        .then(res.redirect('/'))
});

// Route updates

app.get('/:status/:id', (req, res) => {
    const data = { id: `${req.params.id}`, status:
`${req.params.status}`};
    fetch(`http://${ENDPOINT}/update`, {
        method: 'POST',
        body: JSON.stringify(data),
        mode: 'cors',
        headers: {
            'Content-Type': 'application/json',
        },
    })
        .then(res.redirect('/'))
});

// Route deletions

app.get('/:id', (req, res) => {
    const data = { id: `${req.params.id}` };
    fetch(`http://${ENDPOINT}/delete`, {
        method: 'POST',
        body: JSON.stringify(data),
        mode: 'cors',
        headers: {
```

```
            'Content-Type': 'application/json',
        },
    })
        .then(res.redirect('/'))
});

// Port where app is served

app.listen(80, () => {
    console.log('The web server has started on
port 80');
});
```

When we receive a GET request to the base route, we use the Fetch API to send a GET request to the API server, then take the task data we receive and return it to the frontend.

When we receive requests to the specified routes for task additions, status updates, and deletions, we use the Fetch API to send JSON payloads via POST request to the API server. This passes on the relevant information to update the database; we then redirect to the base route and see the homepage updated to reflect the latest changes.

The routes for updates and deletions resemble those used by the monolith more than in the API server. The web client uses the same route handling paths for individual task IDs and status updates as the monolith, but the API server is decoupled and can use its own simpler routes.

For the website itself, we'll need to create a `/views/` folder in the project directory. Here, we can simply copy the `index.handlebars` and `/layouts/main.handlebars` files over from the monolith.

Here's `todo-web/views/index.handlebars`:

```
<div class="container">

<h1>todos</h1>
  <form method="POST" action="/" >
     <input type="text" name="task" id="task"
placeholder="What do you need to do? &#8629"
autocomplete="off" required>
  </form>
</div>

<div id="tasks_container">

        {{#each items}}
        <p id="item_style">
        {{#if (isCompleted status)}}
        <span class="completed">{{task}}</span>
        <a href="/{{task_id}}"
id="emoji">&#128465;</a><a
href="/ongoing/{{task_id}}"
id="emoji">&#10062;</a>
        {{else}}
        <span class="ongoing">{{task}}</span>
        <a href="/{{task_id}}"
id="emoji">&#128465;</a><a
```

```
href="/completed/{{task_id}}"
id="emoji">&#9989;</a>
        {{/if}}
        </p>
        {{/each}}
</div>
```

And here's the layout wrapper `todo-web/views/layouts/main.handlebars`:

```
<!DOCTYPE html>
<html>
  <head>
     <title>Todo App</title>
     <link rel="stylesheet" type="text/css"
href="/styles/index.css">
     <link rel="preconnect"
href="https://fonts.googleapis.com">
     <link rel="preconnect"
href="https://fonts.gstatic.com" crossorigin>
     <link
href="https://fonts.googleapis.com/css2?family=Ro
boto&display=swap" rel="stylesheet">
     <meta charset="UTF-8">
  </head>
  <body>
     {{{body}}}
  </body>
</html>
```

A final piece of housekeeping—we need to create a `/public/` folder inside `/todo-web/` to hold a couple of assets: a stylesheet and a favicon. You can grab those from this project's GitHub page[25].

From the `/todo-web/` directory:

```
% npm i express express-handlebars body-parser
```

(Or include those three modules as dependencies in your `package.json` file.)

Write our standard Dockerfile in the same directory:

```
# Sets the base image
FROM node:18
# Establishes the working directory for your app
within the container
WORKDIR /usr/src/app
# Copies your package.json file and then installs
modules
COPY package*.json ./
RUN npm install
# Copies your project files and then runs the app
COPY . .
CMD [ "node", "index.js" ]
```

Build the container image and then upload it to Docker Hub:

```
% docker build . -t <Docker ID>/todo-web
```

[25] https://github.com/ericgregory/kube-5mins/tree/main/11-scale/todo-web/public

```
% docker push <Docker ID>/todo-web
```

Our decomposition is complete. All that remains is to deploy the API server and web client to Kubernetes.

Deploying your services to Kubernetes

Create a manifest in `/11-scale/manifests/` called `todo-app.yml`:

```
apiVersion: v1
kind: Service
metadata:
 labels:
   app: todo-api
 name: todo-api
spec:
 ports:
 - port: 80
   protocol: TCP
   targetPort: 80
 selector:
   app: todo-api
 type: ClusterIP
---
apiVersion: apps/v1
kind: Deployment
metadata:
 labels:
   app: todo-api
 name: todo-api
spec:
```

```
 replicas: 1
 selector:
   matchLabels:
     app: todo-api
 template:
   metadata:
     labels:
       app: todo-api
   spec:
     containers:
     - image: ericgregory/todo-api:latest
       name: todo-api
       ports:
       - containerPort: 80
       volumeMounts:
       - name: secrets
         mountPath: "usr/src/app/secrets"
     volumes:
       - name: secrets
         secret:
           secretName: mysqlpwd
           defaultMode: 0400

---

apiVersion: v1
kind: Service
metadata:
 labels:
   app: todo-web
 name: todo-web
spec:
 ports:
```

```
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: todo-web
  type: LoadBalancer
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: todo-web
  name: todo-web
spec:
  replicas: 1
  selector:
    matchLabels:
      app: todo-web
  template:
    metadata:
      labels:
        app: todo-web
    spec:
      containers:
      - image: ericgregory/todo-web:latest
        name: todo-web
        ports:
        - containerPort: 80
```

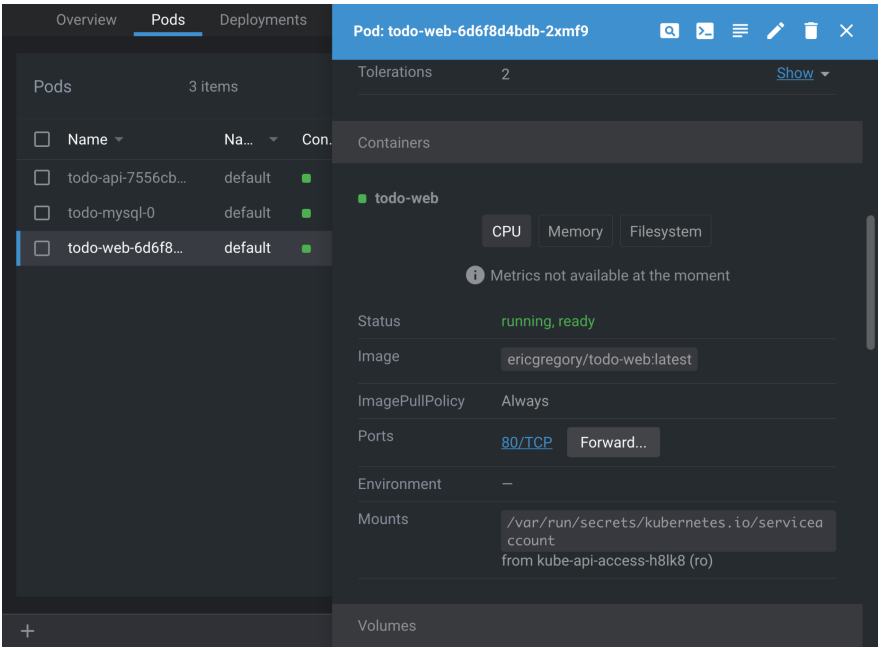
Here we're defining Service and Deployment resources for both the API server (`todo-api`) and the web client (`todo-web`). At this point, everything in this manifest should be familiar.

- Instances of `todo-api` access the Secret containing the database password via volume mounts, and are served by means of a ClusterIP Service—so they are directly accessible only on the cluster.
- Instances of `todo-web` are served by a Service with the LoadBalancer type, so traffic to the web client can be apportioned between instances by a load balancer.

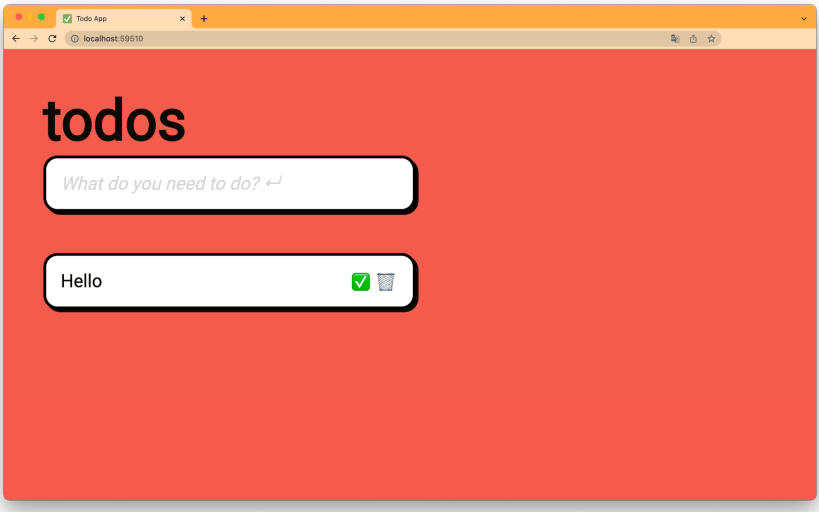
Deploy with kubectl:

```
% kubectl apply -f todo-app.yml
```

In Lens, click the **Overview** tab and select **Pods**. Choose the `todo-web` instance (it will have a random alphanumeric code appended to the end) and scroll down to the **Forward** button.



This will create a port forward to your local machine. Click the button and press Start to “Open in browser.”



Your application is running on Kubernetes. Test it out—adding tasks, deleting tasks, and updating status should all work as expected. Well done!

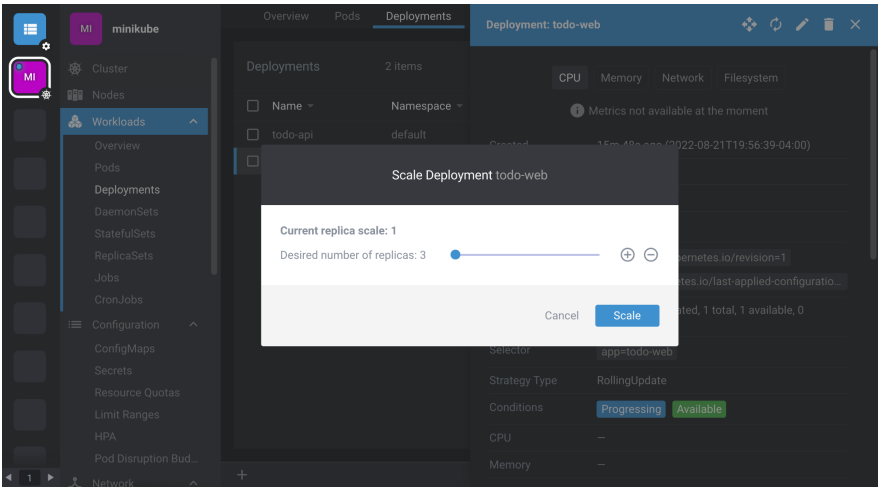
Matters of scale

Our manifests specified one replica, or one instance, of the `todo-api` and `todo-web` workloads. Accordingly, we have one pod for each.

But suppose we wanted to scale up and run more instances. This can be managed through the Deployment resource instances for the API server and web client, and the custom

InnoDBCluster resource for the database server (which in turn manages a StatefulSet).

In Lens, click on the **Deployments** tab and select a deployment. In the upper-right corner, you'll see an icon with four outward-facing arrows. Click it to bring up the **Scale** pane. Here you can use a slider to increase or decrease the number of replicas. Try changing it to **3** and clicking **Scale**.



If you check the Pods pane, you'll find that the desired replicas have been added.

In addition to manual scaling, Kubernetes can automatically scale a Deployment (or StatefulSet) using a core Kubernetes resource called the **Horizontal Pod Autoscaler**. This resource monitors a pre-defined metric that is pertinent to your application (latency, say, or utilization) and adjusts up or down

within defined constraints, striving to remain as near as possible to your defined optimum value.

Suppose we want our API server to autoscale such that there are always a minimum of two replicas and a maximum of five, while trying to remain as close as possible to a target CPU utilization percentage of 80%.

We can express this **declaratively** through a YAML manifest as below:

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
 name: todo-api
spec:
 maxReplicas: 5
 minReplicas: 2
 scaleTargetRef:
   apiVersion: apps/v1
   kind: Deployment
   name: todo-api
 targetCPUUtilizationPercentage: 80
```

We can also express our intention **imperatively** through kubectl:

```
% kubectl autoscale deployment todo-api --min=2
--max=5 --cpu-percent=80
```

Run the command above and check your running Pods using either Lens or `kubectl get pods`. While system resources will vary, you'll likely find that a new replica of `todo-api` has been added.

Ingress and Gateway

Most of our application's traffic during this lesson took place within the cluster—with the exception of our accessing the web client through a port forward. We could have achieved a similar result by using Minikube's service command (in this case, it would have been `minikube service todo-web`) as we've done in the past.

These are both "hacky" means of getting to a cluster running on the cluster from outside. But a real-world app will need a more reliable, systematic means of managing "north-south" traffic, meaning traffic in and out of the cluster. Before we finish for the day, we should discuss some of the ways a cluster administrator might approach this issue.

Ingress

The first approach is using the **Ingress** API. With the Ingress API, you can use **Ingress objects** to define rules for routing traffic from outside the cluster to services inside the cluster, and back out again.

Of course, we saw in previous lessons that it is possible to access services within the cluster using Service types such as

NodePort or LoadBalancer (paired with a cloud provider's load balancer). But both approaches are unwieldy. The Ingress object is an abstraction for, well, *ingress*, enabling you to define and manage a path inside—along with often-complex routing rules that can be used by multiple Services, instead of maintaining all of those Services' load balancer implementations individually.

These Ingress resources are then handled by an **Ingress controller**, which an operator will have installed. Ingress controllers may be platform-specific, like Azure's Ingress controller, or generic, like the NGINX Ingress Controller. (In a cloud setting, operation of an Ingress controller will naturally cost money.) Minikube comes with an Ingress add-on, in fact, which is really just an implementation of the NGINX Ingress Controller.

If you'd like to dig deeper on Ingress, you can learn more from the Ingress page in the Kubernetes docs[26].

Gateway

A newer alternative to Ingress is the **Gateway** API. Originally conceived as a successor to Ingress, the Kubernetes maintainers now describe Gateway as a "superset of Ingress functionality," adding more fine-grained and standardized options. Like Ingress, Gateway is a resource you can use to provide specifications for a vendor-provided Gateway controller. The Gateway implementation sits at the door to

[26] https://kubernetes.io/docs/concepts/services-networking/ingress/

your cluster and routes requests from outside to an appropriate destination inside.

But where Gateway *builds* on Ingress is its standardization and its expressivity. Previously, functionalities like weighted traffic routing or routing across namespaces required heavy annotation and custom solutions—with Gateway, you can be a lot more standard and portable in implementations of those features. Additionally, Gateway explicitly supports HTTP, TCP, UDP, and TLS, where Ingress by default really only plays nice with HTTP. As part of the Gateway API, routes with these protocols can now be explicitly modeled using route objects such as `HTTPRoute`.

Gateway is a welcome addition for developers, since it provides some new tools and means that, ideally, you can utilize more standardized routes, defined now through a Route resource, without having to worry as much about the particularities of Gateway or Ingress configurations. The YAML for a Route might look like this:

```
apiVersion: gateway.networking.k8s.io/v1beta1
kind: HTTPRoute
metadata:
 name: httproute-example
spec:
 parentRefs:
 - name: todo-org
```

The parentRefs parameter defines the Gateways to which this route is attached—in this case, the gateway maintained by a hypothetical ToDo organization. Now the route is decoupled from the gateway, and if you need to make any configuration changes, you can simply do that at the Gateway level.

Gateway is a relatively *recent* addition—the Gateway, GatewayClass, and HTTPRoute resources graduated to beta in July 2022. Notably, the TCPRoute, TLSRoute, and UDPRoute resources are still in alpha.

You can learn more at the Kubernetes Gateway API Reference[27].

In review...

Our decomposition is complete, and we've discussed two major approaches to north-south traffic.

In our final chapter, we'll recommend some next steps into more advanced Kubernetes usage, with tips for further study on topics like security, logging and monitoring, and CI/CD.

[27] https://gateway-api.sigs.k8s.io/api-types/gateway/

Chapter 12
Taking Your Next Steps with Kubernetes

IN THIS CHAPTER...

– Understand fundamentals of application packaging, CI/CD, and security
– Explore avenues for further learning

This series has aimed to give you the fundamental grounding you need to get started with Kubernetes as a developer. But there is much more to learn as you progress into more advanced Kubernetes usage, and there are many different directions you can take your study from here.

Many of those paths involve branching out into the wider Kubernetes ecosystem, including open source projects—many maintained by the Cloud Native Computing Foundation—and vendor offerings. But it's a big world out there! Where should you start? In this final chapter, we'll outline some major topics you may wish to pursue going forward as you continue your Kubernetes journey.

Helm and Kustomize for application packaging and templating

For all its myriad abstractions, Kubernetes doesn't really include an abstraction for a **big-picture application** like our To

Do app—which was, of course, quite a *small* web app in the scheme of things. Kubernetes thinks in terms of the individual components that work together to comprise a larger application.

That can be challenging, especially when a lot of configuration goes into a single component of the app. In the course of our decomposition project, we needed to configure manifests for…

- Three Services
- Two Deployments
- A StatefulSet
- A StorageClass
- A Secret

…and that was without getting into resources for more advanced production usage. YAML manifests can add up quickly, and important configuration details may be spread out between them.

Helm is an open source **package manager** for Kubernetes, and it goes some way toward addressing the problem of configuration sprawl by bundling and templatizing the various YAML manifests for an application, making it easier to install, manage, and publish your apps.

Helm is maintained outside of the Kubernetes core. By default, it is used as a command line tool, similar to but completely

separate from kubectl. You can also use Helm with a graphical user interface through Lens.

With Helm, you can install, manage, and delete or roll back pre-packaged "charts" (Helm's name for packages) through the abstraction of the "release"—the resource that represents the application as a whole.

The second major reason you might use Helm is that it enables you to create and use **templated packages**. That means the creators of a package can write their charts with parameterized "blanks" that a user can fill in with their own information. For example, if we were to create a chart for the app we wrote that checks database connectivity every five minutes, we might make the database server hostname a user-fillable parameter. That way, users can quickly and easily specify the hostname for their own database server when they install the app.

Helm is frequently framed as being in competition with **Kustomize**, a templating engine built into the Kubernetes core.

Both provide solutions to YAML configuration sprawl with some degree of templating functionality. But the templating in Kustomize is much more granular, enabling you to create multiple "patches" for the same configuration files, "kustomizing" them for different environments or needs. Though there is some overlap in their functionality, Helm and Kustomize have different aims and use-cases and are best understood as complementary tools.

Further learning:

- This **hands-on tutorial** is a good place to start learning about **Kustomize**: https://www.mirantis.com/blog/introduction-to-kustomize-part-1-creating-a-kubernetes-app-out-of-multiple-pieces
- To learn more about **Helm**, you can check out my **video tutorial** that walks through installing and publishing charts, as well as best practices for Helm usage: https://www.youtube.com/watch?v=ZLHoxSmsgkU&t

Developer workflow and CI/CD

By this point, you will have gathered that code may go through quite a few steps before being deployed to Kubernetes. Depending on the project and your Kubernetes infrastructure, it might go through steps including:

- Writing code
- Committing to source control
- Building and pushing container images
- Packaging in Helm charts

Over the course of this series, we've been going through these steps manually and then deploying to our local developer cluster to verify that our code is working on Kubernetes. In real-world use, we might then push our code to our production cluster.

This is one possible Kubernetes development workflow, and it brings a major benefit: by using containers on the dev side, you can achieve parity between your dev and prod (and any other) environments.

But I'll wager you can see the pain-points immediately. As soon as you start debugging with any intensity, you'll be going through your build steps—including, at bare minimum, container build steps—over and over again. And you may be further tripped up by local container configuration issues.

An alternative workflow moves build steps off of the developer's machine and out into a standardized pipeline that runs to the production cluster. This is **continuous integration (CI)**–or the automation of build steps—and **continuous delivery (CD)**, and it can be particularly impactful when working with Kubernetes.

In this approach, developers work on their local machines and publish code changes to a git repository. The code is automatically containerized and packaged and then deployed to a hosted cluster—perhaps a dedicated dev cluster, or a developer namespace on a larger cluster. (We'll talk more about namespaces in a moment.)

This can bring much less friction on the dev side. It is also much more *standardized*—and as a result, potentially more secure. But it depends, of course, on a CI/CD system. It is possible to plug in several popular generic CI/CD systems such as CircleCI or GitHub Actions. There are also a handful of common

Kubernetes-specific solutions that may be installed on the cluster itself:

- **Jenkins X** (https://github.com/jenkins-x) brings the longstanding Jenkins project to Kubernetes, where it uses git as a source of truth to drive continuous integration and delivery.
- **Argo** (https://argoproj.github.io) is a widely used suite of CI/CD tools for Kubernetes, also utilizing git as a source of truth. While it covers many use cases, at the time of this writing there are several notable security advisories (https://github.com/argoproj/argo-cd/security/advisories) for ArgoCD that readers should be aware of.

Now, there can be a downside to this approach—you're potentially missing out on the environmental parity you had developing on a local cluster. Once again, you're haunted by the old specter of code running differently on your machine than in production—a problem that containers were meant to have solved.

Fortunately, there are open source tools to address this problem. **Lagoon** is an application delivery platform for Kubernetes that not only delivers local code to production but also uses Docker to maintain a local development environment with identical images and service configurations to the production environment. Lagoon is particularly tailored to web applications that can be especially challenging to develop on Kubernetes—and fits into a larger CI/CD pipeline.

Further learning:

- Read the **Getting Started guide for Lagoon**: https://docs.lagoon.sh

Kubernetes security

When you introduce a complex substrate layer like Kubernetes, you inevitably expand your "attack surface," or the available avenues of attack in your applications and infrastructure. This necessitates security controls within your system as well as new ways of thinking about security.

Let's break our quick tour of Kubernetes security down into four general topics:

- Isolation and access control
- CI/CD and security
- Service meshes
- Observability

Isolation and role-based access control

Just as Linux namespaces can provide the process isolation underlying containers, **Kubernetes namespaces** enable you to completely isolate environments on the same cluster from one another, effectively creating a set of virtual clusters. This has a range of uses—from isolating sensitive workloads to defining

dev/test/prod environments to creating distinct cluster environments for different users or teams.

You're always working within a namespace on Kubernetes—by default, resources are deployed to the default namespace. If you start Minikube, you can get a look at other namespaces on your cluster with…

```
% kubectl get namespaces
```

…or by selecting the **Namespaces** tab in Lens. You'll notice that there are dedicated namespaces like `kube-system` for system components, and that makes sense—those system components are running as resources on the cluster, but most users won't need or want access to them. Putting them on a separate namespace introduces a layer of separation and keeps those components out of the way. Namespaces can also be defined with strict rules determining who can do what within the environment.

As developers, we probably won't have to worry about creating namespaces very often, but we'll need to be able to *use* them. Fortunately, that's a simple matter of **context-setting**. If I know I need to use a namespace called `development` where I've been assigned the user ID `ericgregory`, I can run the following command with kubectl:

```
% kubectl config set-context dev
--namespace=development --cluster=minikube
--user=ericgregory
```

```
% kubectl config use-context dev
```

Here I'm creating a new context called `dev`—specifying the relevant namespace, user ID, and cluster name—and then switching over to that context.

We don't actually need kubectl to change this configuration—all the command is doing is updating our `kubeconfig` file, which configures the connection between clients like kubectl or Lens and the cluster(s). You can *view* this file by running `kubectl config view`, and you can typically *find* your kubeconfig by navigating to your local user directory and looking for a hidden directory called `.kube`. (There's nothing special about this directory; you can place it anywhere and set the `KUBECONFIG` environment variable to that location.)

On Linux and Mac, you can use this quick command from the local user directory to open the kubeconfig with `nano`:

```
% nano ./.kube/config
```

Now you might ask, "What's this about user IDs?"

In addition to namespaces, the second important tool operators can use to manage permissions on the cluster is **role-based access control (RBAC)**. When in use, RBAC enables operators to create roles on the system and then associate those roles with individual users and service accounts. Effective RBAC implementations will define role permissions

and policies on the principle of least privilege, so that any given user account (or service account) has only the access required to do its job.

CI/CD and security

You may have heard discussion of security "shifting left." This expression means that, relative to more traditional conceptions of security that emphasized perimeter defenses for a production application, some of today's most important security considerations take place earlier in the build-and-deployment process—further to the "left" on the development pipeline.

These are typically questions of "supply-chain security," which deals with the components—often open-source—that enter a project at the development stage. Vulnerabilities or malware may be present in container images you download from Docker Hub or modules you download from a package manager like npm and incorporate into your code. From there, the vulnerability or malware may worm its way into your cluster.

If you were developing without a CI/CD system, it would be a good idea for you to manually scan your container images for known vulnerabilities (perhaps using a tool like Snyk) and otherwise investigate your dependencies and their associated security advisories. That's a heavy "shift left" of responsibilities.

More likely, you *will* be using a CI/CD system, which has an important part to play in security. A security-conscious CI/CD process should…

- Draw images from a private container image registry with base images approved for developer use
- Automatically scan images for known vulnerabilities during the build process
- Use container image signing to verify image provenance

These functionalities will generally require cluster components to support them—for example, a private image registry such as Mirantis Secure Registry and a container runtime such as Mirantis Container Runtime will need to integrate with one another and with a CI/CD system to facilitate a secure, automated workflow.

Service meshes

Some Kubernetes clusters will use an external component called a **service mesh** to manage network traffic within the cluster—what is sometimes known as "east-west" traffic. Service meshes are a topic both deep and wide—so much so that Mirantis Press has published an entire freely available book on the topic, ***Service Mesh for Mere Mortals***.

There are many potential benefits to service meshes, including latency improvements and simplified service discovery, but the

one I want to mention here is **encrypted pod-to-pod and service-to-service communication**.

With vanilla, out-of-the-box Kubernetes, east-west traffic is unencrypted. You may have flagged that in the course of our decomposition project—our `todo-web` and `todo-api` services communicated with old-fashioned, unencrypted HTTP. Under ideal circumstances, that might be fine, but if a malicious actor gains access to the cluster, this gives them a foothold to gather data and even escalate permissions. Service meshes can enable us to use TLS. They can also contribute to our final security topic…

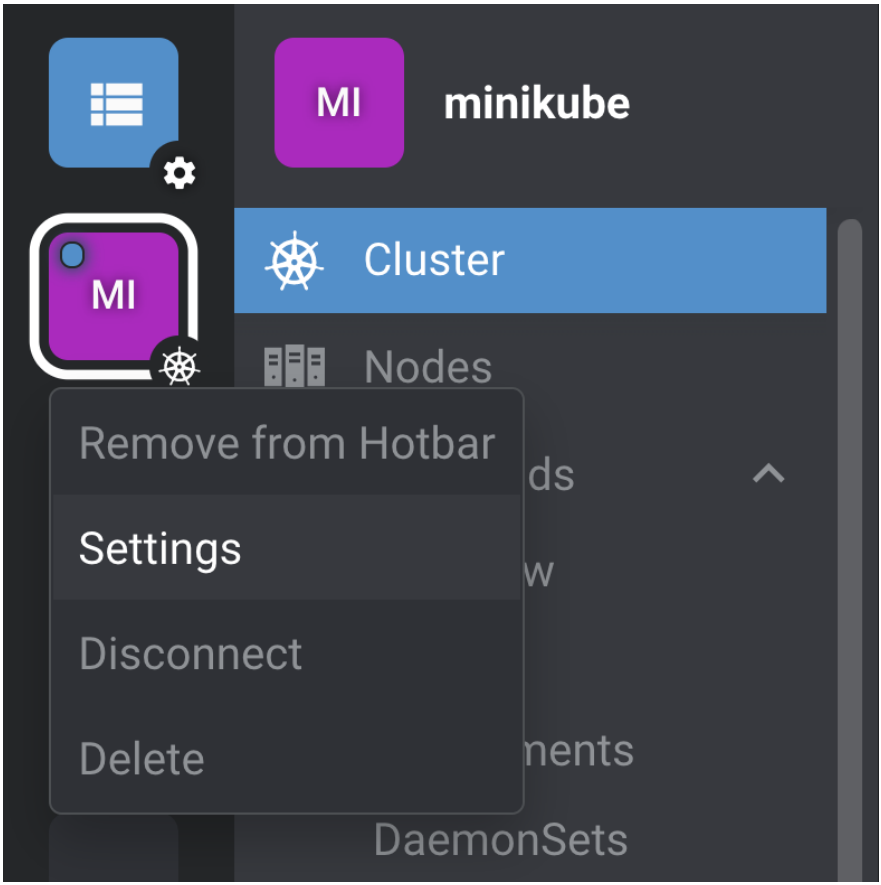
Observability

I said before that cloud native security involves a shift left to earlier stages of the development cycle. But that's not the only change in mindset—Kubernetes also compounds the importance of observability, including monitoring and logging. In a system as complex as Kubernetes, it's important for organizations to prepare for the eventuality of malicious software getting onto the cluster. Robust monitoring and logging systems can help teams detect when something untoward is happening on the cluster.

As I mentioned previously, service meshes can help with observability, but there are many dedicated open source tools as well. Two of the most popular are:

- **Fluentd + ElasticSearch**: Vanilla Kubernetes' logging functionality is pretty rudimentary, simply collecting and storing container runtime logs.
    - The open source **Fluentd** collects much richer logs, acting as a "unified logging layer" and consolidating logs from components like databases, web servers, and cloud providers.
    - **ElasticSearch** serves as a data store for those logs.
    - Consolidated logging is obviously helpful not only for security, but for debugging, system optimization, and other purposes.
- **Prometheus**: This popular CNCF-sponsored project is the standard for Kubernetes monitoring, providing rich, query-able metrics and alerting (and its own API endpoints for extension and integration).

Prometheus underlies the monitoring features in **Lens**, but you'll need to manually enable it. To set up metrics, right click on the cluster icon in Lens' left-hand menu and select **Settings**.



On the **Settings** pane, select **Lens Metrics** to enable a new instance. Prometheus will take a minute or two to start. Press ESC to exit the Settings pane. Lens will connect to your Prometheus and cluster metrics will automatically appear on your dashboard.

Further learning:

- The U.S. National Security Agency (NSA) and Cybersecurity Infrastructure Security Agency (CISA) maintain a **Kubernetes Hardening Guide** with detailed guidance for Kubernetes security. At the time of this writing, the most recent update was March 2022: https://media.defense.gov/2021/Aug/03/2002820425/-1/-1/0/CTR_Kubernetes_Hardening_Guidance_1.1_20220315.PDF

- The Cloud Native Computing Foundation maintains a **Cloud Native Security Whitepaper** with extensive guidance on Kubernetes security practices: https://github.com/cncf/tag-security/tree/main/security-whitepaper

- ***Service Mesh for Mere Mortals*** by Bruce Basil Matthews is a free, full-length ebook from Mirantis Press with hands-on exercises using the Istio service mesh: https://www.mirantis.com/resources/service-mesh-for-mere-mortals/

Conclusion

That ends our whistlestop tour of Kubernetes for developers. I hope this has helped you build a basic grounding in cloud native development that will serve you well in your next steps.

Wherever your Kubernetes journey takes you, good luck!