Kubernetes

App Development

GOLDEN GUIDE TO
KUBERNETES APPLICATION DEVELOPMENT

By Matthew Palmer

Table of Contents

Kubernetes for app developers

Kubernetes is a system for running thousands of containers in an automated, declarative, repeatable, and understandable way. It's how the biggest tech companies in the world run their web services.

Kubernetes provides a standardized framework for deploying application code into production. Learning Docker and Kubernetes—as opposed to your company's ad-hoc build and deployment system—gives you tools that translate across languages, frameworks, and companies.

In the past, enterprises have had a centralized operations team that is responsible for getting application teams' code released and running in production. Kubernetes flips this model. Your company's operations team sets up a Kubernetes cluster, and effectively gives application teams the tools to be able to deploy to production themselves. This lets teams deploy their code as fits their own cadence, and avoids the need to coordinate releases across teams.

From an organizational perspective, Kubernetes empowers everyone to deploy to production and deploy smaller releases more frequently.

From a developer perspective, Kubernetes gives you the tools to create powerful, production-ready applications without needing to worry about the underlying physical infrastructure. Kubernetes consists of a set of deployment primitives—building blocks you use to construct your application's environment.

Kubernetes is a framework for running highly available, highly scaled web services. It gives you tools that work across languages, teams, and companies. You'll be able to deploy your applications more consistently, with less hassle, and with less worry about underlying infrastructure. Kubernetes lets you focus on your applications.

Kubernetes for app developers

Preamble

Contacting the author

You are encouraged to contact the author to ask questions, report issues, send feedback, or request a refund. Feel free to email me at **matt@matthewpalmer.net**, or tweet me at **@_matthewpalmer**.

The upgraded package

This book comes in two packages. There's the base package, which consists of just the book you're reading now. And there's the upgraded package, which is designed to be a complete preparation package for developers taking the CNCF Certified Kubernetes Application Developer exam. (Although I think it's really useful for anyone learning Kubernetes.)

The upgraded package gets you to the next level of Kubernetes expertise through:
- a thirty question, three hour practice exam in the style of the CNCF CKAD exam
- access to a private GitHub repository where you can view complete code samples, ask the author questions directly, and participate in a community of developers learning Kubernetes
- access to full source code, high-res diagrams, and solutions to exercises

If you originally purchased the base package and would like to upgrade, email the author at **matt@matthewpalmer.net**.

Work with Kubernetes

If you are a developer looking to join a forward-thinking organisation that uses Kubernetes to create powerful web applications, get in touch at **matt@matthewpalmer.net**.

For companies looking to increase their DevOps expertise, email me at **matt@matthewpalmer.net**. I'm available for consulting and training engagements, and also know a community of developers looking to join fast-moving organisations.

Reference materials

This chapter provides some reference materials for you to use as you progress throughout the book. If you'd like, you can print these out so that you can keep the complete Kubernetes picture in front of you as we cover each individual concept.

Physical implementation diagram
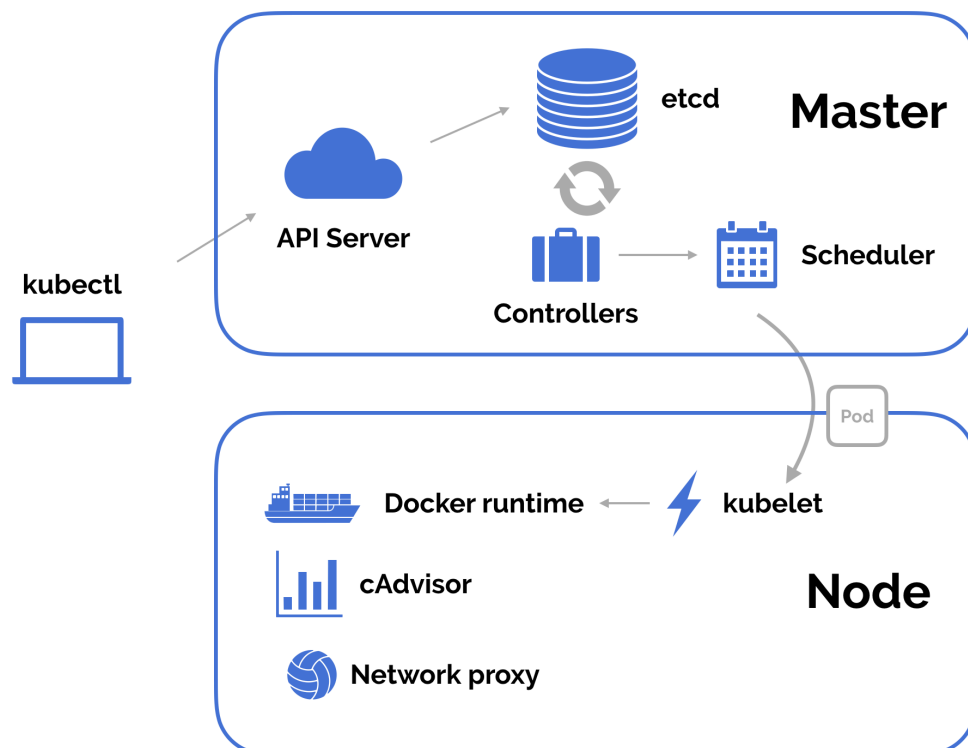
Conceptual diagram
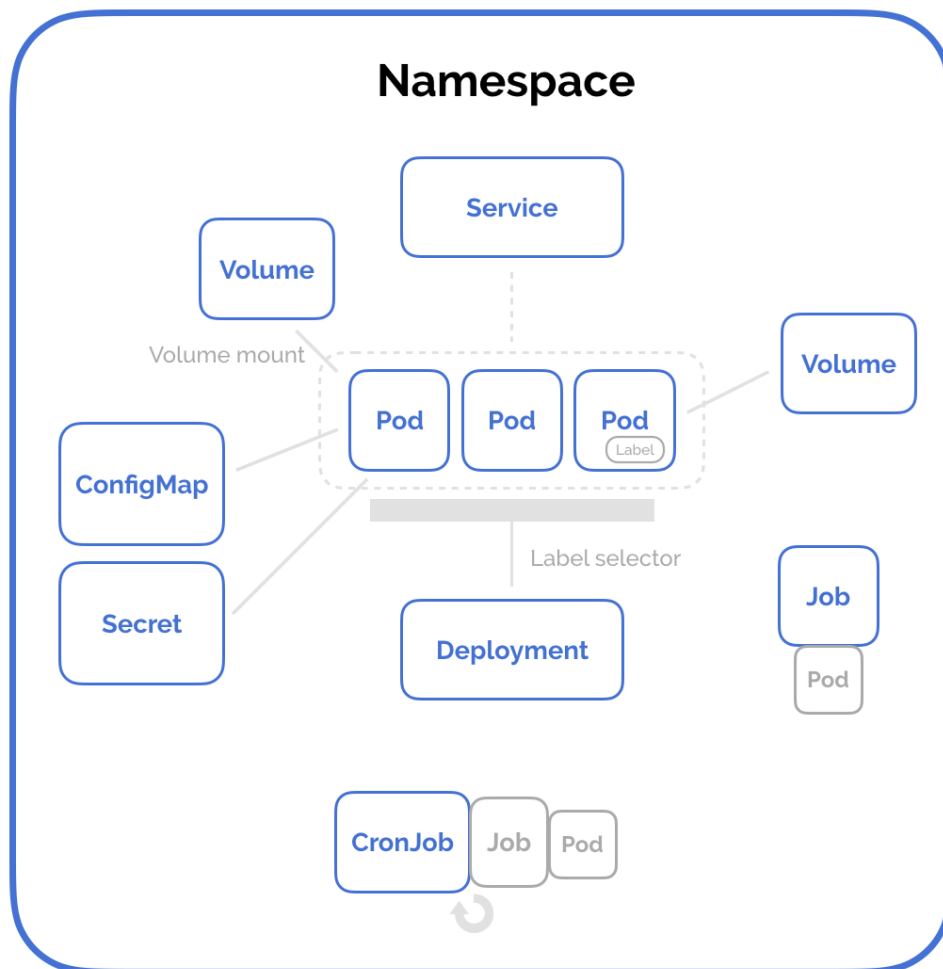
Glossary

kubectl
A command-line utility with commands to control the Kubernetes cluster

Minikube
A single-node Kubernetes cluster designed to be run locally while learning and developing on Kubernetes

Cluster
A cluster is a collection of computers coordinated to work as a single unit. In Kubernetes this consists of a master node and worker nodes.

master
A machine in the Kubernetes cluster that directs work to be done on the worker nodes.

etcd
A distributed key-value store used in Kubernetes to store configuration data for the cluster.

Controller
Controllers are responsible for updating resources in Kubernetes based on changes to data in etcd.

scheduler
A module in the Kubernetes master that selects which worker node a pod should run on, based on resource requirements.

node
A worker machine in the Kubernetes cluster, responsible for actually running pods.

kubelet
A process that runs on each worker node, and takes responsibility for containers that run on that node.

kube-proxy
A networking proxy that runs on each worker node, enables pod-to-pod, pod-to-service communication.

cAdvisor
A process that runs on each worker node that tracks monitoring and resource usage information.

YAML
A markup language used to define configuration for Kubernetes objects.

Containerization
The process of packaging an application, its dependencies, and its configuration into a single deployable unit.

Image
A template for how to create running instances of an application or system.

Container
A running instantiation of an image. Consists of the running process, a filesystem, and networking infrastructure.

Pod
The smallest object that Kubernetes interacts with. It is a layer of abstraction around the container, allowing for containers to be colocated and share filesystem and network resources.

Service
A Kubernetes object used to expose a dynamic set of pods to the network behind a single interface.

Deployment
A Kubernetes object that manages a set of pods as a single unit, controlling their replication, update, and rollback.

Annotation
Non-identifying key-value pairs that describe an object in Kubernetes, can be used to store long or structured data.

Label
Identifying, indexed key-value pairs that are used to group objects in Kubernetes.

Selector
The mechanism used to query objects based on their labels.

Namespace
A subdivision of a cluster that appears like the entire cluster. Used to avoid name collisions, isolate resources, and to apply resource quotas.

ConfigMap
An object used to inject configuration data into containers.

Secret
An object used to inject secret or sensitive configuration data into containers.

Liveness Probe
A process that checks if a container is still alive or if it needs to be restarted.

Readiness Probe
A process that checks if a container has started successfully and is ready to start receiving requests.

Security Context
Configures a pod or container to run with a particular user ID or filesystem group ID.

Sidecar pattern
A multi-container design pattern where another container runs alongside your main application and performs some task non-essential to the application container.

Adapter pattern

A multi-container design pattern where an adapter container massages the output or formatting of your main application so that it can be consumed by another party.

Ambassador pattern

A multi-container design pattern where the ambassador container proxies network requests to a third party. The main application makes requests to localhost, and the ambassador is responsible for forwarding those requests to the external service.

Volume

A piece of storage in a Kubernetes pod that lives for (at least) as long as the pod is alive. Analogous to a directory in your filesystem.

Volume mount

The mechanism by which a container gains access to a volume. The container declares a volume mount, and then it can read or write to that path as though it were a symbolic link to the volume.

Persistent Volume

A cluster-level storage resource where data is guaranteed to persist beyond the pod or container lifecycle.

Persistent Volume Claim

A request to access and use a persistent volume. Made by a pod via the `persistentVolumeClaim` volume type.

Job

A finite task that runs to completion and whose return code indicates success or failure. Kubernetes guarantees that a job will be re-run until it completes successfully.

CronJob

A Job that runs at a specific time or on a schedule. Analogous to the Unix utility cron.

Resource Quota

A namespace-level restriction on the count of objects or on the total resource usage of objects in the namespace.

User Account

An account used to authenticate a human user when they make a request to the Kubernetes API server.

Service Account

An account used to authenticate a process when it makes a request to the Kubernetes API server.

Network Policy

A restriction on the incoming or outgoing network traffic for a group of pods.

Installation

Installing Kubernetes and Docker
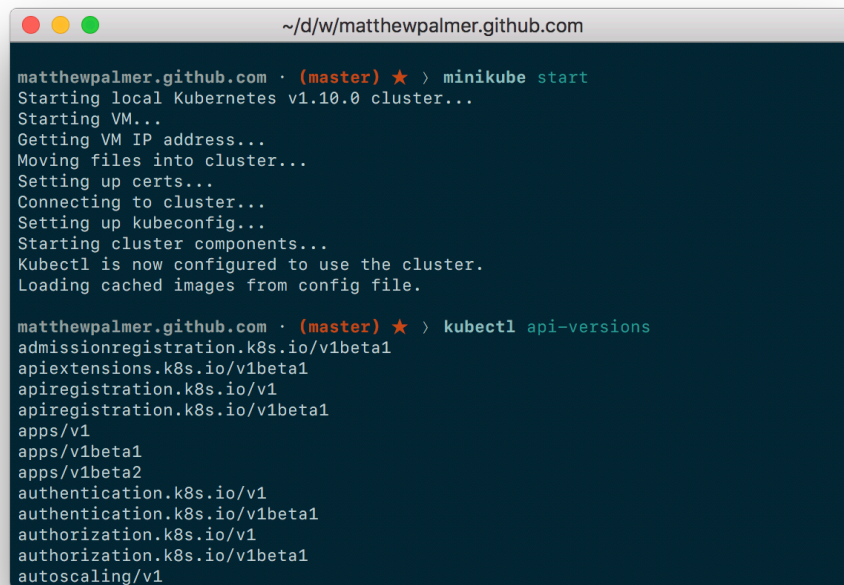
This installation process will guide you through running and accessing a Kubernetes cluster running on a Mac. You'll be using Homebrew, Docker for Mac, Minikube, VirtualBox, and `kubectl`.

These instructions written and updated for the latest version of macOS. If you're running Linux, you're probably better than me at installing software manually, and can figure out—with the help of the internet—how to install Kubernetes on your specific system.

Installation guide

The only pre-requisite for this guide is that you have [Homebrew](#) installed. Homebrew is a package manager for the Mac. You'll also need [Homebrew Cask](#), which you can install after Homebrew by running `brew tap caskroom/cask` in your Terminal.

1. Install [Docker for Mac](#). Docker is used to create, manage, and run our containers. It lets us construct containers that will run in Kubernetes Pods.
2. Install VirtualBox for Mac using Homebrew. Run brew cask install virtualbox in your Terminal. VirtualBox lets you run virtual machines on your Mac (like running Windows inside macOS, except for a Kubernetes cluster.)
3. Install `kubectl` for Mac. This is the command-line interface that lets you interact with Kubernetes. Run `brew install kubectl` in your Terminal.
4. Install [Minikube](#) via the [Installation > OSX instructions from the latest release](#). At the time of writing, this meant running the following command in Terminal: `curl -Lo minikube https://storage.googleapis.com/minikube/releases/v0.27.0/minikube-darwin-amd64 && chmod +x minikube && sudo mv minikube /usr/local/bin/`
5. Everything should work! Start your Minikube cluster with `minikube start`. Minikube will run a Kubernetes cluster with a single node. Then run `kubectl api-versions`.

```
  ● ● ●                  ~/d/w/matthewpalmer.github.com

matthewpalmer.github.com · (master) ★ ⟩ minikube start
Starting local Kubernetes v1.10.0 cluster...
Starting VM...
Getting VM IP address...
Moving files into cluster...
Setting up certs...
Connecting to cluster...
Setting up kubeconfig...
Starting cluster components...
Kubectl is now configured to use the cluster.
Loading cached images from config file.

matthewpalmer.github.com · (master) ★ ⟩ kubectl api-versions
admissionregistration.k8s.io/v1beta1
apiextensions.k8s.io/v1beta1
apiregistration.k8s.io/v1
apiregistration.k8s.io/v1beta1
apps/v1
apps/v1beta1
apps/v1beta2
authentication.k8s.io/v1
authentication.k8s.io/v1beta1
authorization.k8s.io/v1
authorization.k8s.io/v1beta1
autoscaling/v1
```

Come together

You've installed all these tools and everything looks like it's working. Let's review how these components relate.

- VirtualBox is a generic tool for running virtual machines. It runs other operating systems—Ubuntu, Windows, etc.—inside your host operating system, macOS.
- Minikube is a Kubernetes utility that runs a Kubernetes cluster on your machine. That cluster has a single node and has some unique features that make it more suitable for local development. Minikube tells VirtualBox to run. Minikube can use other virtualization tools—not just VirtualBox—however these require extra configuration.
- **kubectl** is the command line application that lets you interact with your Minikube Kubernetes cluster. It sends requests to the Kubernetes API server running on the cluster to manage your Kubernetes environment. **kubectl** is like any other command-line application that runs on your Mac—it just makes HTTP requests to the Kubernetes API on the cluster.

Interacting with Kubernetes

Kubernetes CLI

To interact with a Kubernetes cluster, you use the command line tool kubectl. This program—which you install when you set up your development environment before using Kubernetes—makes requests to the Kubernetes cluster to manipulate the state of what's running on your cluster.

kubectl is a powerful program used to accomplish a range of tasks in Kubernetes. The learning curve is steep. But kubectl's command line documentation (i.e. what you see when you run `kubectl <command> --help`) is some of the best of any tool. The documentation usually contains several examples that illustrate what you want to do. Learning to be comfortable navigating the kubectl command line documentation is worthwhile—it's often easier to read than the online documentation on kubernetes.io.

kubectl config and context

The first thing to understand about using kubectl is how it's configured, and how kubectl contexts work. By default, your kubectl utility should come preconfigured for you, but here's an overview of how it works.

The command `kubectl config` lets you interact with configuration settings for your kubectl client. `kubectl config` has three important operations.

- Add or remove clusters it can connect to
- Add or remove users to use when authenticating with these clusters
- Add, remove, or update contexts used by kubectl

A context is like a user profile for Kubernetes. A context defines a cluster to connect to, a user to use for connecting, and a default namespace for that cluster. Contexts are simple but useful. They help you switch between Kubernetes clusters quickly—otherwise you'd need to manually set your user and authentication method every time you wanted to work on a different cluster.

If you've set up your Kubernetes environment as per the previous chapter, you can run `kubectl config view` to view your current configuration settings and look at your current context. Later, if you're ever confused about settings you're using for each context, or which context you're in,

re-run **kubectl config view** to review your settings. The actual **kubectl config** file is a simple text file. located at **~/.kube/config**.

Let's take a look at what's inside your kubectl configuration by default. Run **kubectl config view** to see its contents, and we'll walk through each section.

```
$ kubectl config view
apiVersion: v1
kind: Config
preferences: {}

# Your current context
current-context: minikube

# Contexts – "user profiles" you can easily switch between.
# Each context consists of a cluster, an auth user,
# and a namespace to use by default.
# These users and clusters are defined below!
contexts:
- name: minikube
  context:
    cluster: minikube
    user: minikube
    # namespace: NAME – set a namespace for the context

# The list of clusters you have access to
clusters:
- name: minikube
  cluster:
    certificate-authority: /Users/mp/.minikube/ca.crt
    server: https://192.168.99.100:8443

# Users you can authenticate as
users:
- name: minikube
  user:
    client-certificate: /Users/mp/.minikube/client.crt
    client-key: /Users/mp/.minikube/client.key
```

Understanding kubectl commands

By now, you've interacted with kubectl a few times. In general, kubectl commands follow a formula.

`kubectl <command> <resource> <options>`

`<command>` is an action you do to a resource in the Kubernetes cluster, like `create`, `get`, `describe`, etc. You can view a list of all available commands by entering `kubectl --help`. The `--help` flag works on any given subcommand—simply append `--help` and kubectl will output useful documentation and examples for that command. Use it often!

`<resource>` is how you identify the entity you want the command to be executed on. You will see the resource identifier written in many different ways, and each resource type has an abbreviation (pod is po, service is svc, etc.). All of the following are equivalent, retrieving information about a pod called example-c7f6:

- `kubectl get pods example-c7f6` – the full identifier, useful because executing `kubectl get pods` will list all pods, in case you forget the name of the resource
- `kubectl get po example-c7f6` – uses the abbreviation alias po, meaning pod
- `kubectl get pod/example-c7f6` – most useful when copying and pasting between commands

`<options>` are as diverse and command-specific as you would expect, ranging from setting the number of replicas of a pod to specifying the input file for a secret key. You can access the range of options for a given command by entering `kubectl <command> --help`.

When learning Kubernetes for the first time, there are a few extraordinarily useful commands for exploring the Kubernetes API and YAML options without digging through online resources.

`kubectl <command> --help`

This is probably the most used Kubernetes command. As it does with most command line tools, it displays documentation and usage guidance for a given command. Kubernetes' help is excellent, it provides accurate—albeit short—descriptions, examples, and a thorough list of options.

`kubectl explain <object>`

`kubectl explain <object>.<sub-object>`
`kubectl explain <object> --recursive`
This command is extraordinarily useful for explaining the fields available to you when you are creating YAML files for your Kubernetes resources. For example, `kubectl explain pod.spec.volumes` gives you a description of what volumes mean for a pod, which `apiVersion` to use, plus all of the fields available for an entry in your volumes list.

Running `kubectl explain` on an object's nested fields is generally much faster, more concise, and more useful than any online documentation. While writing YAML configuration from scratch, it often solves problems faster than Google!

`kubectl explain <object> --recursive` is even more useful: it gives you a summary of every YAML field for an object or subobject recursively down the nested field hierarchy. `kubectl explain pod.metadata --recursive` shows you that you can provide annotations, labels, namespaces, and so on for a pod's metadata field.

`kubectl get <resource> -o yaml --export=true`
This command displays information about a given resource in YAML format, which you can then easily copy and paste into a new YAML document. As a result, I never really remember how to enter YAML from scratch. I simply do `kubectl run example-deployment --image=nginx` to create a deployment of a pod with the nginx image, then run `kubectl get pod/example-deployment-c7f6 -o yaml --export=true` to get the created pod in YAML format, and copy that into a new file after deleting irrelevant lines or entries with sensible defaults. You can also append the `--dry-run` flag to the `run` or `create` commands to see what the output would be without actually running the command on the cluster.

`kubectl get all`
`kubectl describe all`
This command displays information about every resource in your Kubernetes cluster, which can be extremely useful when you're not exactly sure what is even running in your cluster. `kubectl get` gives you a summary of each resource in the cluster, and `kubectl describe` provides detailed information.

Containers

Docker and containers

Containerization is packaging an application, its dependencies, and its configuration into a single unit. This unit is called an image. The image is then used as a template to construct live, running instances of this image. These running instances are called containers. A container consists of the image, a read-write filesystem, network ports, resource limits, and other runtime configuration. Docker is the most popular way to build images and run containers, and is what we use in this book.
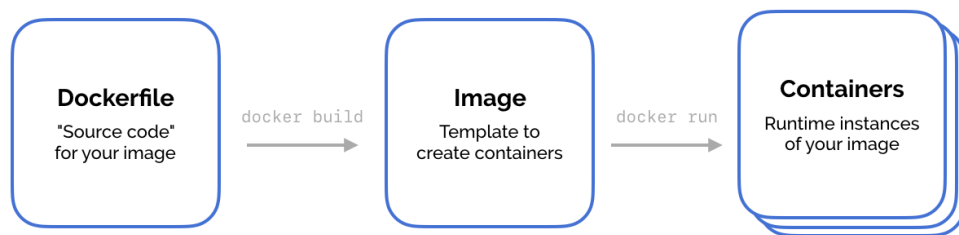
Consider a simple Node.js application that has not been containerized. If you were deploying this on a fresh virtual machine, you'd need to:

- install the operating system
- install the Node.js runtime
- copy your application's source code into place
- run your code using Node

Of that list, you really only take responsibility for your source code. When you deploy a new version of your application, you just swap out the old source code for the newer version. The operating system and Node.js stays in place.

When you package your application into a container, you take responsibility for everything you need to run your application—the OS, the runtime, the source code, and how to run it all. It all gets included into the image, and the image becomes your deployment unit. If you change your source code, you build a new image. When you redeploy your code, you instantiate a new container from the image.

Conceptually, this is great. Encapsulate everything your application needs into a single object, and then just deploy that object. This makes

deployment predictable and reproducible—exactly what you want for something that's typically outside an application developer's expertise.

But alarm bells might be ringing: why aren't these images huge and expensive to run? The image includes the whole operating system and the Node.js runtime!

Docker uses layers—read-only intermediate images—that are shared between final images. Each command used to generate the Docker image generates a new intermediate image via a delta—essentially capturing only what changed from the previous intermediate step. If you have several applications that call for the Ubuntu operating system in their Dockerfile, Docker will share the underlying operating system layer between them.

There are two analogies that might help depending on your familiarity with other technologies. React—the JavaScript framework—re-renders all your UI whenever your application's state changes. Like including an operating system in your application deployment, this seems like it should be really expensive. But React gets smart at the other end—it determines the difference in DOM output and then only changes what is necessary.

The other analogy is the git version control system, which captures the difference between one commit and the previous commit so that you can effectively get a snapshot of your entire project at any point in time. Docker, React, and git take what should be an expensive operation and make it practical by capturing the difference between states.

Let's create a Docker image to see how this works in practice. Start a new directory, and save the following in a file called **Dockerfile**.

```
# Get the Node.js base Docker image — shared!
FROM node:carbon

# Set the directory to run our Docker commands in
WORKDIR /app

# Copy your application source to this directory
COPY . .

# Start your application
CMD [ "node", "index.js" ]
```

Then, let's write a simple Node.js web server. Create the following in a file called **index.js**.

```
# index.js
var http = require('http');

var server = http.createServer(function(request, response) {
  response.statusCode = 200;
  response.setHeader('Content-Type', 'text/plain');
  response.end('Welcome to the Golden Guide to Kubernetes
Application Development!');
});

server.listen(3000, function() {
  console.log('Server running on port 3000');
});
```

In the directory, open a new shell and build the Docker image.

```
$ docker build . -t node-welcome-app
Sending build context to Docker daemon  4.096kB
Step 1/4 : FROM node:carbon
carbon: Pulling from library/node
1c7fe136a31e: Pull complete
ece825d3308b: Pull complete
06854774e2f3: Pull complete
f0db43b9b8da: Pull complete
aa50047aad93: Pull complete
42b3631d8d2e: Pull complete
93c1a8d9f4d4: Pull complete
5fe5b35e5c3f: Pull complete
Digest:
sha256:420104c1267ab7a035558b8a2bc13539741831ac4369954031e0142b565fb7b5
Status: Downloaded newer image for node:carbon
 ---> ba6ed54a3479
Step 2/4 : WORKDIR /app
Removing intermediate container eade7b6760bd
 ---> a8aabdb24119
Step 3/4 : COPY . .
 ---> 5568107f98fc
Step 4/4 : CMD [ "node", "index.js" ]
 ---> Running in 9cdac4a2a005
Removing intermediate container 9cdac4a2a005
 ---> a3af77202920
Successfully built a3af77202920
Successfully tagged node-welcome-app:latest
```

Now that we've built and tagged the Docker image, we can run a container instantiated from the image using our local Docker engine.

```
$ docker run -d -p 3000 node-welcome-app
a7afe78a7213d78d98dba732d53388f67ed0c3d2317e5a1fd2e1f680120b3d15

$ docker ps
CONTAINER ID   IMAGE             COMMAND          PORTS
a7afe78a7213   node-welcome-app  "node index.js"  0.0.0.0:32772->3000
```

The output of **docker ps** tells us that a container with ID **a7afe78a7213** is running the **node-welcome-app** image we just built. We can access this container using port 32772 on localhost, which Docker will forward to the container's port 3000, where our application server is listening.

```
$ curl 'http://localhost:32772'
Welcome to the Golden Guide to Kubernetes Application Development!
```

Pods

By this point, we've successfully used Docker to build an image, and then used Docker to run a container—an instantiation of that image. We could keep creating images and manually starting them up with Docker, but this would be laborious.

Docker wasn't designed to coordinate running hundreds of containers across multiple computers. Its responsibility is to build images and run containers—it's a container runtime.

This is where Kubernetes comes in.

Kubernetes is a container orchestration system—it automates the deployment and scaling of containers. Kubernetes' responsibility is to manage hundreds of containers across many computers. It takes control of their uptime, networking, storage, and scheduling. When it needs to actually run a container, Kubernetes leaves that responsibility to the container runtime. The most popular container runtime is Docker, which is what we use in this book, but Kubernetes also supports other container runtimes like rkt and containerd.

Rather than working with containers directly, Kubernetes adds a small layer of abstraction called a pod. A pod contains one or more containers, and all the containers in a pod are guaranteed to run on the same machine in the Kubernetes cluster. Containers in a pod share their networking infrastructure, their storage resources, and their lifecycle.

In the previous chapter, we ran our Node.js web server using the `docker run` command. Let's do the equivalent with Kubernetes.

Creating a Kubernetes pod

We're going to create a Dockerfile that defines a Docker image for a simple web server that runs on Node.js. We'll use Docker to build this
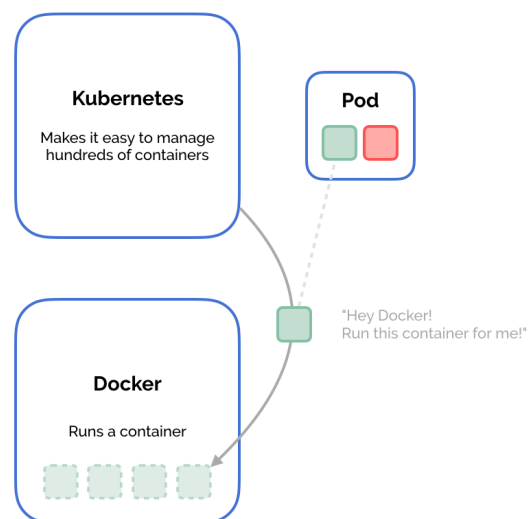
image. This is essentially the same as what we did in the previous chapter. But instead of using `docker run` to create a container running the image, we'll define a pod in Kubernetes that uses the image. Finally, we'll create the pod in Kubernetes, which runs the container for us. Then we'll access the pod and make sure our container is running.

First, let's create a simple web server using Node.js. When a request is made to localhost:3000, it responds with a welcome message. Save this in a file called **index.js**.

```
# index.js
var http = require('http');

var server = http.createServer(function(request, response) {
  response.statusCode = 200;
  response.setHeader('Content-Type', 'text/plain');
  response.end('Welcome to the Golden Guide to Kubernetes
Application Development!');
});

server.listen(3000, function() {
  console.log('Server running on port 3000');
});
```

Next, we'll create a **Dockerfile**—the file that gives Docker instructions on how to build our Docker image. We start from the Node.js image, mix in our **index.js** file, and tell the image how to start containers when they are instantiated.

```
# Dockerfile
FROM node:carbon
WORKDIR /app
COPY . .
CMD [ "node", "index.js" ]
```

Now we've got to build this image. Take note that we need to make sure this image is available to the Docker engine in our cluster. If you're running a cluster locally with Minikube, you'll need to configure your Docker settings to point at the Minikube Docker engine rather than your local (host) Docker engine. This means that when we do docker build, the image will be added to Minikube's image cache and available to

Kubernetes, not our local one, where Kubernetes can't see it. You will need to re-run this command each time you open a new shell.

```
$ eval (minikube docker-env)
# This command doesn't produce any output.
# What does it do? It configures your Docker settings
# to point at Minikube's Docker engine, rather than the local one,
# by setting some environment variables.
# set -gx DOCKER_TLS_VERIFY "1";
# set -gx DOCKER_HOST "tcp://192.168.99.100:2376";
# set -gx DOCKER_CERT_PATH "/Users/matthewpalmer/.minikube/certs";
# set -gx DOCKER_API_VERSION "1.23";

$  docker build . -t my-first-image:1.0.0
Sending build context to Docker daemon  4.096kB
Step 1/4 : FROM node:carbon
 ---> ba6ed54a3479
Step 2/4 : WORKDIR /app
 ---> Using cache
 ---> 3daa6d2e9d0b
Step 3/4 : COPY . .
 ---> c85c95b4a4be
Step 4/4 : CMD [ "node", "index.js" ]
 ---> Running in 2e68c5316ed9
Removing intermediate container 2e68c5316ed9
 ---> 4106fb401625
Successfully built 4106fb401625
Successfully tagged my-first-image:1.0.0
```

The `-t` flag specifies the tag for an image—by convention it's the name of your image plus the version number, separated by a colon.

Now that we've built and tagged an image, we can run it in Kubernetes by declaring a pod that uses it. Kubernetes lets you declare your object configuration in YAML or JSON. This is really beneficial for communicating your deployment environment and tracking changes. If you're not familiar with YAML, it's not complicated—search online to find a tutorial and you can learn it in fifteen minutes.

Save this configuration in a file called pod.yaml. We'll cover each field in detail in the coming chapters, but for now the most important thing to note is that we have a Kubernetes pod called `my-first-pod` that runs the `my-first-image:1.0.0` Docker image we just built. It instantiates this image to run in a container called `my-first-container`.

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: my-first-pod
spec:
  containers:
    - name: my-first-container
      image: my-first-image:1.0.0
```

While you can create things directly on the command line with kubectl, one of the biggest benefits of Kubernetes is that you can declare your deployment environments explicitly and clearly through YAML. These are simple text files that can be added to your source control repository, and changes to them can be easily tracked throughout your application's history. For this reason, we prefer writing our Kubernetes resources' configuration into a YAML file, and then creating those resources from the file.

When working with resources declared in a YAML file in Kubernetes, there are several useful commands. All of them use the **-f** argument followed by the path to the file containing our YAML.

kubectl create -f <filename>
This command explicitly creates the object declared by the YAML.

kubectl delete -f <filename>
This command explicitly deletes the object declared by the YAML.

kubectl replace -f <filename>
This command explicitly updates a running object with the new one declared by the YAML.

kubectl apply -f <filename or directory>
This command uses declarative configuration, which essentially gives Kubernetes control over running create, delete, or replace operations to make the state of your Kubernetes cluster reflect the YAML declared in the file or directory. While using **kubectl apply** can be harder to debug since it's not as explicit, we often use it instead of **kubectl create** and **kubectl replace**.

Choose either **create** or **apply**, depending on which makes more sense to you, and create the pod.

```
$ kubectl create -f pod.yaml
pod "my-first-pod" created

$ kubectl get pods
NAME                     READY     STATUS         RESTARTS    AGE
my-first-pod             1/1       Running        0           11s
```

Great! Our pod looks like it's running. You might be tempted to try to access our container via http://localhost:3000 like we did when we ran the container directly on our local Docker engine. But this wouldn't work.

Remember that Kubernetes has taken our pod and run its containers on the Kubernetes cluster—Minikube—not our local machine. So to access our Node.js server, we need to be inside the cluster. We'll cover networking and exposing pods to the wider world in the coming chapters.

The **kubectl exec** command lets you execute a command in one of the containers in a running pod. The argument **-it** allows you to interact with the container. We'll start bash shell on the container we just created (conceptually, this is kind of similar to SSHing in to your pod). Then we'll make a request to our web server using **curl**.

```
$ kubectl exec -it my-first-pod -c my-first-container bash
root@my-first-pod:/app # curl 'http://localhost:3000'
Welcome to the Golden Guide to Kubernetes Application Development!
```

We've successfully created a Docker image, declared a Kubernetes pod that uses it, and run that pod in a Kubernetes cluster!

Common "dummy" images

While you're learning Kubernetes, tutorials conventionally use a few "dummy" or "vanilla" Docker images. In practice, your images will be real, production Docker images custom to your application. In this book, we use these as a placeholder for your real application.

Here are a few of the most common "dummy" images you'll see throughout this book and tutorials online.

- **alpine** – a Linux operating system that is very lightweight but still has access to a package repository so that you can install more packages. Often used as a lightweight base for other utilities and applications.
- **nginx** – a powerful, highly performant web server used in many production deployments. It's widely used as a reverse proxy, load balancer, cache, and web server. It's often used when tutorials need a standard web server.
- **busybox** – a very space-efficient image that contains common Unix utilities. It's often used in embedded systems or environments that need to be very lightweight but still useful.
- **node** – an image of the Node.js JavaScript runtime, commonly used for web applications. You'll also see variants of node based on different base operating systems via different tags. For example, **node:8.11.3–alpine** and **node:8.11.3–jessie** are two variants of **node:8.11.3** that use the alpine and jessie Linux distros respectively
- Similarly, there are Docker images for php, ruby, python, and so on. These have variants that you can use via different tags, similar to the node image.

Now let's take a step back and take in the bigger picture of how Kubernetes works.

Kubernetes System Overview

Kubernetes helps you run containers across many machines. Kubernetes manages how containerized applications run, communicate, and share resources across these machines. It will scale, gracefully update, and restart containers. Kubernetes also accounts for different workflows for your containers—some containers will want to be replicated multiple times, some will want to run only once, and some might need to run every six hours. Kubernetes manages all of this for you.

Kubernetes unlocks the ability for you to treat many machines as a single resource—Kubernetes takes control of the cluster so that you can focus on your application. If a machine within the cluster dies while one of your containers is doing something, Kubernetes will get a new machine running, migrate your container onto it, and make sure your container runs properly.

In reality, Kubernetes doesn't work directly with containers. It adds one layer of abstraction around the container called a pod. This abstraction has many benefits—described in the coming chapter—but the conceptual model is the same: Kubernetes runs pods using a cluster of computers.

Kubernetes is an operating system for a cluster of computers. This operating system can be used for many workloads, but is particularly suited for running robust, highly available, scaled web services. You can think of containers like you think of processes in your computer's operating system, where Kubernetes assumes the role of the OS. The key difference is that containers let you encapsulate all of your application's requirements into a single, reproducible unit. This is a powerful abstraction for building better web applications.

Components

As an application developer, you need to have a conceptual understanding of how Kubernetes works.

Kubernetes works across multiple machines—a cluster of computers. One of these machines is the 'master', and all the other machines are the 'nodes'.

The master

The master machine is responsible for making sure that the other nodes are alive and running what you've told them to run. Several key components live inside the master: the configuration database, the API server, the controller manager, and the scheduler.

The configuration database is essential to Kubernetes, and to understanding how Kubernetes works. This database stores the actual state of the entire cluster at any point in time. It also stores the desired state of the cluster, based on your requests. This configuration database is implemented using `etcd`, a distributed, reliable key-value store that runs on the master and every node.

Other components watch this store for changes—if the actual state doesn't match the desired state, Kubernetes controllers will make changes in the system to reconcile them.

You ask two natural questions based on this design pattern. How did the current state of the cluster and the ideal state diverge? How do I get to the ideal state? Everything else you need to understand flows from answering these two questions.

First—how did the current state and the ideal state diverge? Two ways: the actual state changed, or the ideal state changed.

The ideal state of the cluster changes because you, the application developer, request it, or because one of the nodes in your cluster requests it. Kubernetes is told to run a new pod, scale a deployment, or maybe to add more storage. The request is made through the API server (`kube-apiserver`), a process inside the master that gives access to the Kubernetes API via a HTTP REST API. The API Server will update `etcd` to reflect what the new ideal state is.

The actual state of the cluster might change without you requesting it because a node dies, a container crashes, or some other failure. This is Kubernetes' power—in the event of a failure, Kubernetes can see what the state of the system is *meant* to be, and tries to get back to that state.

Second—how do I get to the ideal state? Kubernetes has a group of *controllers* whose job it is to make the actual cluster state match the ideal state. An individual controller typically takes responsibility for a resource type—pods, nodes, namespaces, etc.—and then runs in an infinite loop.

```
while true:
      ideal_state = get_ideal_state()
      actual_state = get_actual_state()

      if actual_state != ideal_state:
            make_changes()
```

The component inside Kubernetes master that manages these infinitely-looping controllers is the controller manager (**kube-controller-manager**). These controllers communicate with the API server to create, update, and delete the resources they manage so that the cluster gets back to its ideal state.

The other core component that runs in master is the scheduler. **kube-scheduler** determines which node should run a pod. It finds new pods that don't have a node assigned, looks at the cluster's overall resource utilisation, hardware and software policies, node affinity, and deadlines, and then decides which node should run that pod.

The master maintains the actual and desired state of the cluster using **etcd**, lets users and nodes change the desired state via the **kube-apiserver**, runs controllers that reconcile these states, and the **kube-scheduler** assigns pods to a node to run.

The nodes

While there is only one master machine, there are many node machines to run pods on. These are the workers—the machines where containers are deployed and actual work is done. Every node in a Kubernetes cluster has a container runtime, a Kubernetes node agent, a networking proxy, and a resource monitoring service.

The container runtime is responsible for actually running the containers that you specify. This is Docker or rkt, and works just like running a container on your local machine with **docker run**.

The agent that Kubernetes uses to manage the lifecycle of a node is called **kubelet**. **kubelet** is a process that runs on each node that takes responsibility for the state of that node. It starts and stops containers as directed by the master, and ensures that its containers remain healthy. It

will also track the state of each of its pods, and if a pod is not in its desired state, it will be redeployed. **kubelet** must also relay its health to the **master** every few seconds. If the master sees that a node has failed (i.e. **kubelet** has failed to report that the node is healthy), controllers will see this change and relaunch pods on healthy nodes.

Master

Config store

API Server

Controller manager

Controllers

Scheduler

Node	**Node**	**Node**
Docker runtime	Docker runtime	Docker runtime
kubelet	kubelet	kubelet
Network proxy	Network proxy	Network proxy
Monitoring	Monitoring	Monitoring
Pod 1 — Pod 2	Pod 1 — Pod 2	Pod 1 — Pod 2
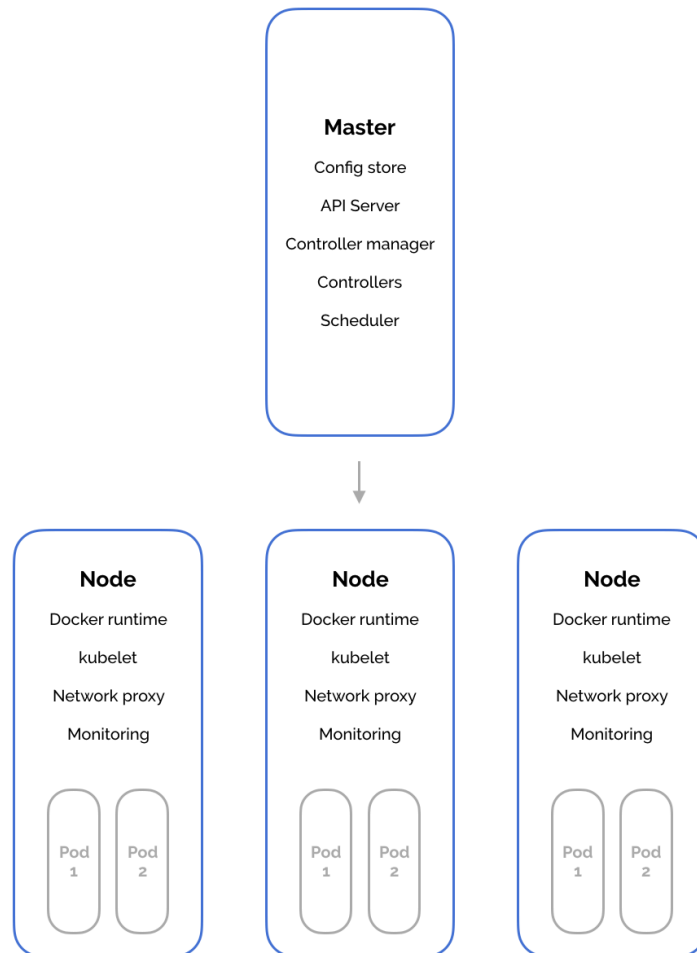
Every Kubernetes node also requires a networking proxy (**kube-proxy**) so that it can communicate with other services in the cluster and the master node. This process is responsible for routing networking traffic to the relevant container and other networking operations.

Every Kubernetes node also runs **cAdvisor**, a simple agent that monitors the performance and resource usage of containers on the node.

These four components are necessary for a Kubernetes node to exist. However, it's not doing anything useful! The last and most important

aspect of a node are its pods—the groups of containers that actually run
your application.

Objects

Your Kubernetes environment is defined by a collection of objects.
Kubernetes provides many types of objects, all with unique
characteristics, that you combine to perform specific tasks.

The two most basic attributes of
any Kubernetes object are its
`apiVersion`—the Kubernetes API
version it's from, and `kind`—the
type of object it is, like a class in
object-oriented programming.

An object also has metadata. It
contains descriptive information
about the object itself. In the case
of Kubernetes objects, this is the
object's name, its labels, and its
annotations.

> ### A Kubernetes Object
>
> **Version**   *v1, v1beta1, …*
>
> **Kind**   *Pod, Service, …*
>
> ---
>
> **Metadata**
> *Name*
> *Labels*
> *Annotations*
>
> ---
>
> **Spec**
> *Your desired state*
>
> **Status**
> *Kubernetes' actual state*

The meat of an object are its two
properties `spec` and `status`. `spec`
is how you define your object—it tells Kubernetes the desired state of
your object. `status` is reality—it's where Kubernetes stores how your
object is actually running. If the actual state (`status`) differs from the
ideal state (`spec`), Kubernetes will act to get that object to its ideal state.

All of this configuration information is stored inside etcd, and if the actual
state of Kubernetes diverges from the state declared by this
configuration, work will be done to reconcile them.

The following example explains each field on a pod, the simplest object
in Kubernetes that runs containers.

```
# Which version of Kubernetes this object comes from
apiVersion: v1

# What type of object am I?
kind: Pod

# Metadata —  Meta-information about the object itself
metadata:
  name: nginx-pod

  # (Labels and annotations will be explained later!)
  labels:
    key: value
  annotations:
    otherkey: othervalue

# Spec — what you want your object to be
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```

The `apiVersion` field

Kubernetes' documentation has a great shortcoming: the `apiVersion` field. Nowhere does anyone explain what it means, what your options are, and what to use, when. As you explore Kubernetes, you'll see different values like `v1`, `apps/v1`, `extensions/v1beta1`, and so on.

What is the correct value for this field? It depends! The most up-to-date version will change as Kubernetes releases new versions, and different objects move through alpha, beta, and into different API groups. To help you navigate the changes to the apiVersion field, I've created an [up-to-date web page that lists the correct apiVersion for each object](#).

Pods in detail

Overview

The most important object in Kubernetes is the pod. It's also the simplest. A pod is what you run when you run something on Kubernetes—it's the small, simple building block that you deploy.

In reality, a pod is one level of abstraction around a container. While pods and containers are different—and you need to understand the difference—they are close siblings. Understanding containers is essential to understanding pods.

A container is an enclosed, self-contained execution environment, much like a process in an operating system. A pod encapsulates a group of these containers, adds storage resources, networking configuration, and rules for running the containers.

While you can run multiple containers in a pod, in reality you often use one container per pod. When designing your pods, don't force multiple containers into the same pod—we will discuss a few of the specific reasons to have multiple containers in a pod in this chapter.
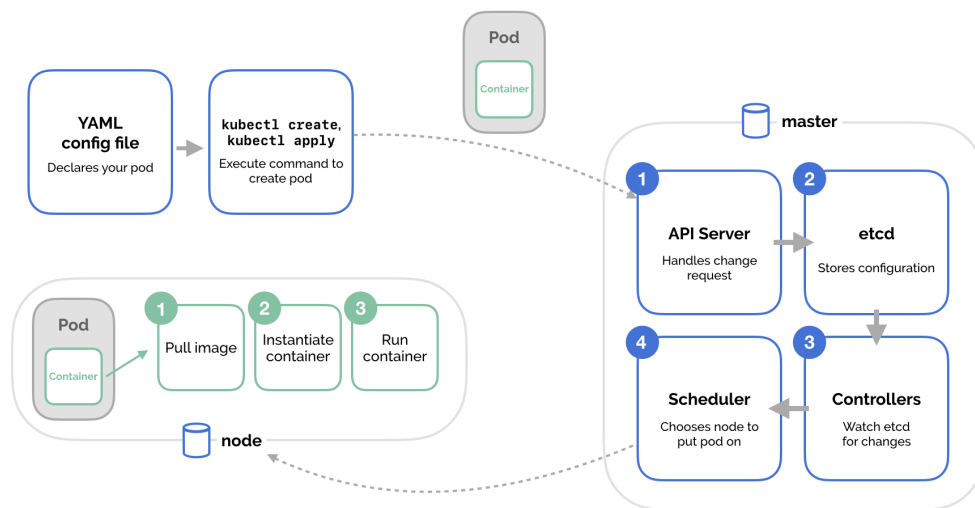
Pod lifecycle

In its lifetime, a pod transitions through several stages. A pod's current stage is tracked in the `status` field, which has a subfield called `phase`. When a pod's state changes, the kubelet process on that node updates

the `phase` field in the pod's etcd entry. A pod has one of several phases: Pending, Running, Succeeded, Failed, or Unknown.

Pending
The API Server has validated the pod and created an entry for it in etcd, but its containers haven't been created or scheduled yet.

Running
All of a pod's containers have been created, the pod has been scheduled on a node in the cluster, and it has running containers.

Succeeded
Every container in the pod has finished executing without returning errors or failing. The pod is considered to have completed successfully. None of the containers are going to be restarted.

Failed
Every container has finished executing, but some of them have exited with a failure code.

Unknown
The pod's status could not be obtained.

A pod's `phase` field provides a high-level view of where it is up to in transitioning through its lifecycle. Perhaps more practical, though, is the `status` column that is shown when you run `kubectl get pods`. This column combines information from all aspects of the pod's `status` field— not just the `phase`—to provide a useful view of what has happened to the pod.

In this example, a deployment with an invalid image has been created. Let's look at the information Kubernetes provides about the failing pods' lifecycle.

```
$ kubectl get pods -o yaml
# ... truncated to `status`
status:
    conditions:
    containerStatuses:
    - image: fds982r3fds
      imageID: ""
      lastState: {}
      name: failed-deployment
      ready: false
      restartCount: 0
      state:
        waiting:
          message: Back-off pulling image "fds982r3fds"
          reason: ImagePullBackOff
    phase: Pending

$ kubectl describe pods
# ... truncated to Events
Events:
  Type      Reason      Message
  ----      ------      -------
  Normal    Scheduled   Successfully assigned fd1 to minikube

  Normal    Pulling     pulling image "fds982r3fds"

  Warning   Failed      Failed to pull image "fds982r3fds"

  Warning   Failed      Error: ErrImagePull

  Warning   Failed      Error: ImagePullBackOff

  Normal    BackOff     Back-off pulling image "fds982r3fds"
```

In the output of `kubectl get pods -o yaml`, we see the pod is in the
Pending phase of its lifecycle—it has been accepted by Kubernetes but
hasn't had any of its containers created yet or been successfully
scheduled. In the event logs from `kubectl describe pods`, Kubernetes
has tried to schedule the pod but failed to get its image—the image has
an invalid name that can't be found anywhere. This gets reflected into the
pod's `status.containerStatuses[0].state` field, which captures the
last error from Kubernetes.

Advanced configuration

Custom command and arguments

It's useful, especially while learning Kubernetes, to create containers that
use a custom command rather than having to create a complete Docker
image to illustrate and test a simple concept. It's something we do
frequently in this book.

In Kubernetes, this is accomplished via the **command** and **args** fields in your container's YAML spec.

command replaces the image's **ENTRYPOINT**, the command that is executed to start your container. If you are overriding **ENTRYPOINT** in a YAML file, it's common that this will point to a shell executable like **/bin/sh so** that you can execute another command in that shell.

args replaces the image's **CMD**. This list of arguments is passed to the command specified in the previous field. Often in this book, you'll see args set to **["-c", "<some command>"]**. This allows us to define shell commands in YAML that are executed when the container runs. Since pods are only alive for as long as the first command is running, these will often run in an infinite loop so that we can inspect the running containers later. This is a useful debugging tool in case you don't have a suitable Docker image to deploy.

```
kind: Pod
apiVersion: v1
metadata:
  name: custom-command-pod
spec:
  containers:
    - name: command-container
      image: alpine
      command: ["/bin/sh"]
      args: ["-c", "while true; do date; sleep 5; done"]
```

```
$ kubectl apply -f custom-command.yaml
pod "custom-command-pod" created

$ kubectl logs custom-command-pod
Tue Jul  3 09:58:15 UTC 2018
Tue Jul  3 09:58:20 UTC 2018
Tue Jul  3 09:58:25 UTC 2018
```

Environment variables

When running your application, you're likely to want to use environment variables to dynamically inject configuration settings into your containers. While it's also recommended to use ConfigMaps to accomplish this, which are covered in a later chapter, you can also set these values directly in YAML.

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: environment-variables
spec:
  containers:
    - name: env-var-container
      image: nginx:1.7.9
      # The env field lets you set environment variables
      # for the container
      env:
      - name: BASE_URL
        value: "https://api.company.com/"
      - name: CONNECTION_STRING
        value: "mongodb://localhost:27017"
```

Liveness and Readiness

Many containers, especially web services, won't have an exit code that accurately tells Kubernetes whether the container was successful. Most web servers won't terminate at all! How do you tell Kubernetes about your container's health? You define container probes.

Container probes are small processes that run periodically. The result of this process determines Kubernetes' view of the container's state—the result of the probe is one of **Success**, **Failed**, or **Unknown**.

You will most often use container probes through Liveness and Readiness probes. Liveness probes are responsible for determining if a container is running or when it needs to be restarted. Readiness probes indicate that a container is ready to accept traffic. Once all of its containers indicate they are ready to accept traffic, the pod containing them can accept requests.

There are three ways to implement these probes. One way is to use HTTP requests, which look for a successful status code in response to making a request to a defined endpoint. Another method is to use TCP sockets, which returns a failed status if the TCP connection cannot be established. The final, most flexible, way is to define a custom command, whose exit code determines whether the check is successful.

Readiness Probes

In this example, we've got a simple web server that starts serving traffic after some delay while the application starts up. If we didn't configure our readiness probe, Kubernetes would either start sending traffic to the

container before we're ready, or it would mark the pod as unhealthy and never send it traffic.

Let's start with a readiness probe that makes a request to http://localhost:8000/ after five seconds, and only looks for one failure before it marks the pod as unhealthy.

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: liveness-readiness-pod
spec:
  containers:
    - name: server
      image: python:2.7-alpine

      # Check that a container is ready to handle requests.
      readinessProbe:
        # After five seconds, check for a
        # 200 response on localhost:8000/
        # and check only once before thinking we've failed.
        initialDelaySeconds: 5
        failureThreshold: 1
        httpGet:
          path: /
          port: 8000

      # This container starts a simple web
      # server after 45 seconds.
      env:
        - name: DELAY_START
          value: "45"
      command: ["/bin/sh"]
      args: ["-c", "echo 'Sleeping...'; sleep $(DELAY_START); echo 'Starting server...'; python -m SimpleHTTPServer"]
```

Start the pod, and we'll inspect its event log—notice how Kubernetes thinks the pod is unhealthy. In reality, we just have an application that takes a little while to boot up!

```
$ kubectl apply -f liveness-readiness.yaml
pod "liveness-readiness-pod" created

$ kubectl describe pod liveness-readiness-pod
Events:
  Normal   Scheduled             Successfully assigned
  Normal   SuccessfulMountVolume MountVolume.SetUp succeeded
  Normal   Pulled                Container image already present
  Normal   Created               Created container
  Normal   Started               Started container
  Warning  Unhealthy             Readiness probe failed
```

Liveness Probes

Another useful way to customize when Kubernetes restarts your applications is through liveness probes. Kubernetes will execute a container's liveness probe periodically to check that the container is running correctly. If the probe reports failure, the container is restarted. Otherwise, Kubernetes leaves the container as-is.

Let's build off the previous example and add a liveness probe for our web server container. Instead of using an `httpGet` request to probe the container, this time we'll use a command whose exit code indicates whether the container is dead or alive.

```
kind: Pod
apiVersion: v1
metadata:
  name: liveness-readiness-pod
spec:
  containers:
    - name: server
      image: python:2.7-alpine

      # Check that a container is ready to handle requests.
      readinessProbe:
        # After thirty seconds, check for a 200
        # response on localhost:8000/
        # and check four times before thinking we've failed.
        initialDelaySeconds: 30
        failureThreshold: 4
        httpGet:
          path: /
          port: 8000

      # Check that a container is still alive and running
      livenessProbe:
        # After sixty seconds, start checking
        # periodically whether we have an index.html file.
        # Wait five seconds before repeating the check.
        # This file never exists so the container restarts.
        initialDelaySeconds: 60
        periodSeconds: 5
        exec:
          command: ["ls", "index.html"]

      # This container starts a simple
      # web server after 45 seconds.
      env:
        - name: DELAY_START
          value: "45"
      command: ["/bin/sh"]
      args: ["-c", "echo 'Sleeping...'; sleep $(DELAY_START);
echo 'Starting server...'; python -m SimpleHTTPServer"]
```

Now, delete and recreate the new pod. Then, inspect its event log. We'll see that our changes to the readiness probe have worked, but then the liveness probe finds that the container doesn't have the required file so the container will be killed and restarted.

```
# (This will take around 90 seconds before displaying the complete log.)
$ kubectl describe pod liveness-readiness-pod
Events:
  # ... prior events
  Normal   Created   2s (x2 over 1m)  Created container

  # Notice how our readiness probe only failed twice --
  # at 30s and 40s, but then was successful because our
  # web server started up after 45s.
  Warning  Unhealthy 2s (x2 over 1m)  Readiness probe failed

  # After the pod becomes ready, Kubernetes starts doing liveness
  # probs. Our liveness probe fails, so the container is
  # killed and restarted.
  Normal   Killing   2s               Killing container.
                                      Container failed liveness probe.
  Normal   Started   1s (x2 over 1m)  Started container
```

Security Contexts

A little-used but powerful feature of Kubernetes pods is that you can declare its security context, an object that configures roles and privileges for containers. A security context can be defined at the pod level or at the container level. If the container doesn't declare its own security context, it will inherit from the parent pod.

Security contexts generally configure two fields: the user ID that should be used to run the pod or container, and the group ID that should be used for filesystem access. These options are useful when modifying files in a volume that is mounted and shared between containers, or in a persistent volume that's shared between pods. We'll cover volumes in detail in a coming chapter, but for now think of them like a regular directory on your computer's filesystem.

To illustrate, let's create a pod with a container that writes the current date to a file in a mounted volume every five seconds.

```
kind: Pod
apiVersion: v1
metadata:
  name: security-context-pod
spec:
  securityContext:
    # User ID that containers in this pod should run as
    runAsUser: 45
    # Group ID for filesystem access
    fsGroup: 231
  volumes:
    - name: simple-directory
      emptyDir: {}
  containers:
    - name: example-container
      image: alpine
      command: ["/bin/sh"]
      args: ["-c", "while true; do date >> /etc/directory/
file.txt; sleep 5; done"]
      volumeMounts:
        - name: simple-directory
          mountPath: /etc/directory
```

```
$ kubectl apply -f security-context.yaml
pod "security-context-pod" created

$ kubectl exec security-context-pod -- ls -l /etc/directory
-rw-r--r--    1 45       231             58 Jul  3 09:45 file.txt
```

Notice that the created file has the user ID 45 and the group ID 231, both taken from the security context. This is useful for auditing, reviewing where files come from, and also to control access to files and directories.

Multi-container pod design patterns

Pods will often only have one container—this is good. One of the downsides of Kubernetes introducing the abstraction of pods is that it makes it seem like you should put multiple containers in a pod to use Kubernetes correctly. Don't fall for this trap. In fact, the opposite is true: single-container pods are easier to build, deploy, and debug. Containers grouped together in a pod are guaranteed to run on the same physical node, this allows container-to-container coordination of restart policies, networking infrastructure, and filesystem resources. You don't always need to use multi-container pods.

When should you combine multiple containers into a single pod? When the containers have the exact same lifecycle, when the containers share filesystem resources, or when the containers must run on the same node. The most common scenario is that you have a helper process that needs to be located and managed on the same node as the primary container.

Another reason to combine containers into a single pod is for simpler communication between containers in the pod. These containers can make network requests via localhost, they can communicate through shared volumes (writing to a shared file or directory) and they can use inter-process communication (semaphores or shared memory).

There are three common patterns to use when combining multiple containers into a single pod: the sidecar pattern, the adapter pattern, and the ambassador pattern.
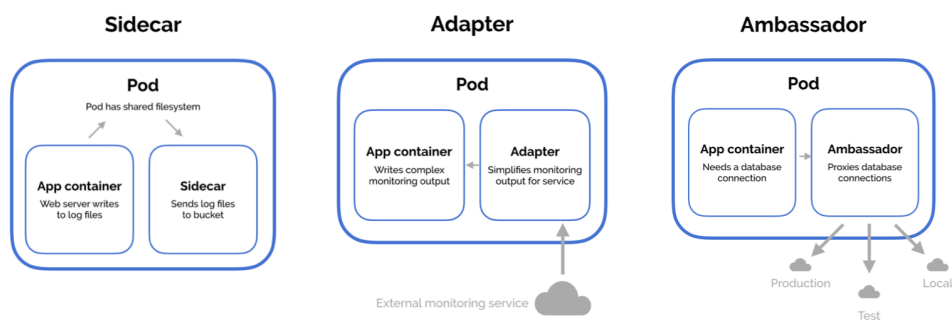
Sidecar pattern

The sidecar pattern consists of a main application—i.e. your web application—plus a helper container with a responsibility that is useful to your application, but is not necessarily part of the application itself. The most common sidecar containers are logging utilities, sync services, watchers, and monitoring agents. It wouldn't make sense for a logging container to run while the application itself isn't running, so we create a pod composed of the main application and the sidecar container. Moving the logging work to another container means that any faults are isolated to that container—failure won't bring down the main application.

Here's an annotated example of a pod using the sidecar pattern.

```yaml
# This pod defines a main application container which writes
# the current date to a log file every five seconds.

# The sidecar container is nginx serving that log file.
# (In practice, your sidecar is likely to be a log collection
# container that uploads to external storage.)

# Once the pod is running:
#
#    (Connect to the sidecar pod)
#    kubectl exec pod-with-sidecar -c sidecar-container -it bash
#
#    (Install curl on the sidecar)
#    apt-get update && apt-get install curl
#
#    (Access the log file via the sidecar)
#    curl 'http://localhost:80/app.txt'
kind: Pod
apiVersion: v1
metadata:
  name: pod-with-sidecar
spec:
  # Create a volume called 'shared-logs' that the
  # app and sidecar share.
  volumes:
  - name: shared-logs
    emptyDir: {}

  # In the sidecar pattern, there is a main application
  # container and a sidecar container.
  containers:

  # Main application container
  - name: app-container
    # Simple application: write the current date every 5s.
    image: alpine
    command: ["/bin/sh"]
    args: ["-c", "while true; do date >> /var/log/app.txt; sleep 5;done"]

    # Mount the pod's shared log file into the app container.
    volumeMounts:
    - name: shared-logs
      mountPath: /var/log

  # Sidecar container
  - name: sidecar-container
    # Simple sidecar: display log files using nginx.
    image: nginx:1.7.9
    ports:
      - containerPort: 80

    # Mount the pod's shared log file into the sidecar
    # container. In this case, nginx will serve the files
    # in this directory.
    volumeMounts:
    - name: shared-logs
      mountPath: /usr/share/nginx/html
```

Adapter pattern

The adapter pattern is used to standardize and normalize application output or monitoring data for aggregation. As a simple example, we have a cluster-level monitoring agent that tracks response times. Say we have a Ruby application in our cluster that writes request timing in the format `[DATE] – [HOST] – [DURATION]`, while a different Node.js application writes the same information in `[HOST] – [START_DATE] – [END_DATE]`.

The monitoring agent can only accept output in the format `[RUBY|NODE] – [HOST] – [DATE] – [DURATION].` We could force the applications to write output in the format we need, but that burdens the application developer, and there might be other things depending on this format. The better alternative is to provide adapter containers that adjust the output into the desired format. Then the application developer can simply update the pod definition to add the adapter container and they get this monitoring for free.

The following example of the adapter pattern defines a main application container that writes the current date and system usage information to a log file every five seconds. The adapter container reformats this output into the format required by a hypothetical monitoring service.

```yaml
#   (Take a look at what the application is writing.)
#   cat /var/log/top.txt
#
#   (Take a look at what the adapter has reformatted it to.)
#   cat /var/log/status.txt
kind: Pod
apiVersion: v1
metadata:
  name: pod-with-adapter
spec:
  volumes:
  - name: shared-logs
    emptyDir: {}

  containers:

  # Main application container
  - name: app-container
    # This application writes system usage information
    # (`top`) to a status file every five seconds.
    image: alpine
    command: ["/bin/sh"]
    args: ["-c", "while true; do date > /var/log/top.txt && top
-n 1 -b >> /var/log/top.txt; sleep 5;done"]

    # Mount the pod's shared log file into the app
    # container. The app writes logs here.
    volumeMounts:
    - name: shared-logs
      mountPath: /var/log

  # Adapter container
  - name: adapter-container
    # This sidecar container takes the output format of
    # the application, simplifies and reformats it for
    # the monitoring service to come and collect.
    # In this example, our monitoring service requires
    # status files to have the date, then memory usage,
    # then CPU percentage each on a new line.

    # Our adapter container will inspect the contents of
    # the app's top file, reformat it, and write the correctly
    # formatted output to the status file.
    image: alpine
    command: ["/bin/sh"]

    # A long command doing a simple thing: read the
    # `top.txt` file, adapt it, and write it to status.txt
    args: ["-c", "while true; do (cat /var/log/top.txt | head -1
> /var/log/status.txt) && (cat /var/log/top.txt | head -2 | tail
-1 | grep
 -o -E '\\d+\\w' | head -1 >> /var/log/status.txt) && (cat /var/
log/top.txt | head -3 | tail -1 | grep
-o -E '\\d+%' | head -1 >> /var/log/status.txt); sleep 5; done"]

    # Mount the pod's shared log file into the adapter
    # container.
    volumeMounts:
    - name: shared-logs
      mountPath: /var/log
```

Ambassador pattern

The ambassador pattern is a useful way to connect containers with the outside world. An ambassador container is essentially a proxy that allows other containers to connect to a port on localhost while the ambassador container proxies these requests to the external world.

One of the best use-cases for the ambassador pattern is for providing access to a database. When developing locally, you probably want to use your local database, while your test and production deployments want different databases again. Managing which database you connect to could be done through environment variables, but will mean your application changes connection URLs depending on the environment. A better solution is for the application to *always* connect to localhost, and let the responsibility of mapping this connecting to the right database fall to an ambassador container. In other situations, the ambassador could be sending requests to different shards of the database—the application itself still doesn't need to worry.

Readers with the upgraded book package can access an advanced example of the Ambassador design pattern in the `code/multi-container/ambassador` directory in the GitHub repository.

Labels, selectors, annotations

Kubernetes provides multiple tools for keeping track of resources in your cluster. When you first start with Kubernetes, these might seem like small, irrelevant topics. But, as we'll see in the next chapter, using labels and selectors is crucial for implementing deployments and services correctly.

Namespaces

The first useful tool for managing access to resources in your cluster is namespaces. Namespaces allow you to subdivide a single physical cluster into several virtual clusters, each of which looks like the original but is completely isolated from the others. One of these virtual clusters is called a namespace.

Why subdivide a cluster like this? You can define resource limits on each namespace, which prevents users in that namespace from exceeding CPU, disk, or object limits. Namespaces also isolate resources, which means that resources created by one team in your company won't be affected by resources created by a different team.
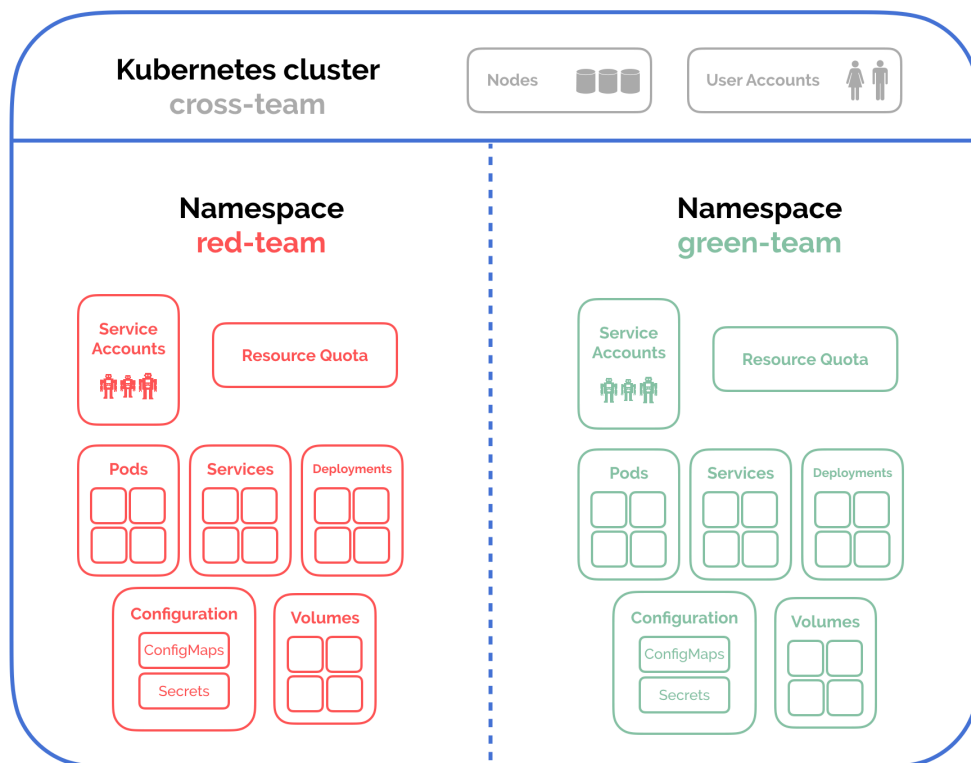
While namespaces may not be necessary if you're learning Kubernetes locally, readers studying for the CNCF Certified Kubernetes Application Developer exam need to understand and use them. Many questions will ask you to operate in a specific namespace.

Using namespaces

If you use namespaces very rarely, or you want to be explicit when using them, simply append the `--namespace=NAMESPACE_NAME` argument to the kubectl command you want to execute. This is a simple, explicit, and easy to remember approach.

Another option is to save the namespace for a kubectl context to your kubectl configuration file. This is done by executing `kubectl config set-context CONTEXT_NAME --namespace=NAMESPACE_NAME`. This could be the right approach for you if you are frequently switching kubectl contexts and have different namespace names depending on the context.

Labels

Kubernetes lets you associate key-value pairs to objects to help you identify and query them later. These key-value pairs are called labels. While they might not seem important at first, they're vital for grouping objects in Kubernetes. For example, if you want to group pods together into a service, you need labels and a label selector. In your system, expect that many objects will carry the same labels.

Set labels for an object inside the `labels` dictionary in the object's `metadata` property. Like dictionary entries in programming languages, Kubernetes labels consist of both a key and a value.

```
apiVersion: v1
kind: Pod

# Metadata — Meta-information about the object itself
metadata:
  name: my-pod
  # Define labels for the pod
  labels:
    release: stable
    environment: dev
    tier: backend
    region: us

spec:
  # ...
```

Labels are meant to specify small, specific properties of an object, such as its release stream or which layer of the application it fits in. They will be indexed and tracked by Kubernetes for querying, sorting, and filtering—keep them short and sharp.

View labels for a resource by appending the `--show-labels` argument to a `kubectl get` command, for example `kubectl get all --show-labels` lists all the objects in Kubernetes, including their associated labels.

```
$ kubectl get all --show-labels
NAME                 LABELS
service/kubernetes   component=apiserver,provider=kubernetes
```

Selectors

Selectors are the counterpart to labels that make labels useful. They are the mechanism for performing queries based on objects' labels. Selectors let you select a group of objects based on the key-value pairs that have been set as their labels. You can think of selectors as the `WHERE` part of a `SELECT * from pods WHERE <labels> = <values>`. You use label selectors from the command line or in an object's YAML when it needs to select other objects.

Querying on the command line

To query objects in Kubernetes based on their labels, use the **-l** or **--selector** argument followed by your label query.

Label queries specify a key and value to use for matching objects. You can query for things that have that value (the **=** operator) or things that do not have that value (the **!=** operator). Combine multiple filters using a comma (**,**), which is the equivalent of **&&** in programming languages. Label queries can filter on a set of values using the **in**, **notin**, and **exists** keywords.

```
# Equals, not equals, and multiple filters
kubectl get all --selector='provider = kubernetes'
kubectl get all --selector='provider != kubernetes'
kubectl get all \
  --selector='provider = kubernetes, component = apiserver'

# Set-based filters
kubectl get all --selector='provider in (kubernetes,utils,mail)'
```

Selectors in YAML

For application developers, there are two important uses for label selectors: in services and in deployments.

We'll cover services in depth in a coming chapter, but one of the most common uses of labels and selectors is to group pods into a service. The **selector** field in a service's **spec** defines which pods receive requests sent to that service. The service runs a query using the key-value label pairs specified in this field to find pods it can use. If your pods' labels are incorrectly defined, your services won't send traffic to the correct pods!

In the following example, the service will route requests to pods that have **key** equal to **value**, and **key2** equal to **value2**. Using the command-line syntax, this is equivalent to **kubectl get pods --selector='key = value, key2 = value2'**.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  # ...
spec:
  selector:
    key: value
    key2: value2
```

Annotations

Labels and selectors have a functional purpose. Labels are key-value pairs set on an object. Selectors are queries that search for objects based on those key-value pairs. Because they're used in queries and sorting, labels need to be short and simple.

But what if we need to store detailed, non-identifying metadata about our objects? Use annotations.

Annotations are also key-value pairs, however annotations won't be indexed or queried on, and they'll never be used to identify your objects. Annotations can store large amounts of data, potentially in a structured format. They are useful for storing version numbers, contributor lists, contact details, short release descriptions, links to logs and dashboards, and debugging information.

Like labels, annotations are defined in an object's `metadata` field.

```
apiVersion: v1
kind: Pod

metadata:
  name: my-pod
  labels:
    # ...
  annotations:
    commit: 6da32faca
    logs: 'http://logservice.com/oirewnj'
    contact: 'Matthew Palmer <matt@matthewpalmer.net>'

spec:
  # ...
```

Deployments

Overview

Deployments in Kubernetes let you manage a set of identical pods through a single parent object—the deployment. If you have a pod that you want many copies of—say an application server that needs to scale up with increased load—deployments are the right tool. They let you create that pod so that it can be replicated, updated without downtime, and the number of copies scaled up.

Through the deployment's YAML configuration, you tell Kubernetes the number of copies of a pod you want running. If a pod fails, a node dies, or you increase the ideal number of copies, Kubernetes will work to make the deployment match your declared ideal state.

Deployments are important in Kubernetes. As an application developer, having Kubernetes manage the scaling and replication of your web apps is probably the main appeal. We'll walk through how deployments work through a command-line example, and then move into the nitty-gritty of declaring a deployment in YAML.

To start, we'll create a deployment through the command line.

```
$ kubectl run nginx-deployment --image=nginx --replicas=3
deployment.apps "nginx-deployment" created
```

This command tells Kubernetes that we want a deployment called nginx-deployment, to run the nginx Docker image, and that we want this to happen on three pods.

We can then query the state of the deployment.

```
$ kubectl get deployment nginx-deployment
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3            3           5m
```

Kubernetes has created three replicas of our nginx pod, and all of them are available. Notice the **DESIRED**, **CURRENT**, and **AVAILABLE** columns—if any of these contain different numbers, Kubernetes will enact changes to reconcile the current state with the desired state.
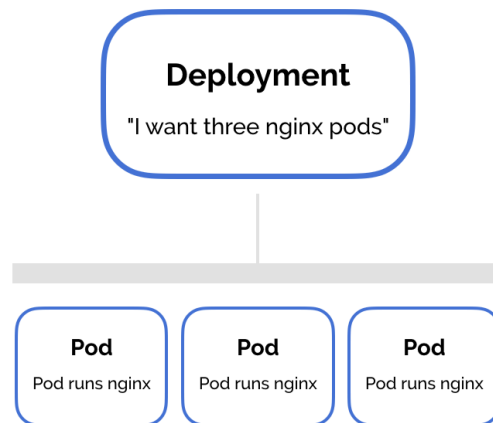
Now simulate one of the application pods failing, and watch the Kubernetes deployment created a new pod in its place.

```
$ kubectl get pods
NAME                                READY    STATUS         RESTARTS    AGE
nginx-deployment-6b979c859f-bvdjn   1/1      Running        0           5m
# ...

$ kubectl delete pod nginx-deployment-6b979c859f-bvdjn
pod "nginx-deployment-6b979c859f-bvdjn" deleted

$ kubectl get pods
NAME                                READY    STATUS
nginx-deployment-6b979c859f-7bcxh   0/1      ContainerCreating
nginx-deployment-6b979c859f-bvdjn   0/1      Terminating
# ...

$ kubectl get pods
NAME                                READY    STATUS
nginx-deployment-6b979c859f-7bcxh   1/1      Running
#...
```

After we killed the pod, a new pod was immediately scheduled by Kubernetes with the ContainerCreating status. After a few seconds, the Terminating pod was completely removed. The new pod is ready, and moves into the Running status. Our deployment has automatically

reconciled the actual system state with the desired state we declared to Kubernetes when we created the deployment.

Deployment YAML

One of the main appeals of Kubernetes is that you can declare your desired system state through configuration files which can be tracked, versioned, and easily understood. Like pods, deployments are declared through YAML files.

At first, a deployment's YAML file looks more confusing than the example YAML configuration we've seen so far. But this is only because it nests a pod specification inside the deployment's configuration—the `deployment.spec.template` field is in fact the same configuration we've seen for all the pods we've created so far.

```yaml
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: nginx-deployment
spec:
  # A deployment's specification really only
  # has a few useful options

  # 1. How many copies of each pod do we want?
  replicas: 3

  # 2. How do want to update the pods?
  strategy: Recreate

  # 3. Which pods are managed by this deployment?
  selector:
    # This matches against the labels we set on the pod!
    matchLabels:
      deploy: example

  # Where deployments start to look intimidating. In reality,
  # this template field is a regular pod configuration!
  template:
    metadata:
      # Set labels on the pod.
      # This is used in the deployment selector.
      labels:
        deploy: example
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
```

Rolling updates and rolling back

Other than providing seamless replication and declarative system states, using deployments gives you the ability to update your application with zero downtime and track multiple versions of your deployment environment.

When a deployment's configuration changes—for example by updating the image used by its pods—Kubernetes detects this change and executes steps to reconcile the system state with the configuration change. The mechanism by which Kubernetes executes these updates is determined by the deployment's `strategy.type` field.

Recreate

With the `Recreate` strategy, all running pods in the deployment are killed, and new pods are created in their place. This does not guarantee zero-downtime, but can be useful in case you are not able to have multiple versions of your application running simultaneously.

RollingUpdate

The preferred and more commonly used strategy is `RollingUpdate`. This gracefully updates pods one at a time to prevent your application from going down. The strategy gradually brings pods with the new configuration online, while killing old pods as the new configuration scales up.

When updating your deployment with `RollingUpdate`, there are two useful fields you can configure.

`maxUnavailable` effectively determines the minimum number of pods you want running in your deployment as it updates. For example, if we have a deployment currently running ten pods and a `maxUnavailable` value of 4. When an update is triggered, Kubernetes will immediately kill four pods from the old configuration, bringing our total to six. Kubernetes then starts to bring up the new pods, and kills old pods as they come alive. Eventually the deployment will have ten replicas of the new pod, but at no point during the update were there fewer than six pods available.

`maxSurge` determines the maximum number of pods you want running in your deployment as it updates. In the previous example, if we specified a

`maxSurge` of 3, Kubernetes could immediately create three copies of the new pod, bringing the total to 13, and then begin killing off the old versions.

Rolling back

A deployment's entire rollout and configuration history is tracked in Kubernetes, allowing for powerful undo and redo functionality. You can easily rollback to a previous version of your deployment at any time.

This is managed through the `kubectl rollout` command, which is easy-to-use and has self-explanatory commands. The command-line help information for this command is quite good, and provides useful examples—you can access it via `kubectl rollout --help`. We'll use this command in the following example.

Using rolling updates and rolling back

Let's start with a simple deployment that has five pods running the `nginx:1.7.9` image.

```yaml
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: nginx-deployment
spec:
  replicas: 5
  strategy:
    type: RollingUpdate
    rollingUpdate:
      # Allow up to seven pods to be running
      # during the update process
      maxSurge: 2
      # Require at least four pods to be running
      # during the update process
      maxUnavailable: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
```

Now create the deployment, and check out the deployment revision log using `kubectl rollout`.

```
$ kubectl apply -f deployment.yaml
deployment.extensions "nginx-deployment" created

$ kubectl rollout history deployment
deployments "nginx-deployment"
REVISION   CHANGE-CAUSE
1          <none>
```

We can see that the first version of our deployment has been deployed
and rolled out.

Let's update our deployment to use a more modern version of nginx, the
Docker image nginx:1.15.0. Watch as Kubernetes scales down the old
replicas of the pod, and brings online replicas of the pod that has the
updated version of nginx.

```
$ kubectl apply -f deployment-update.yaml
deployment.extensions "nginx-deployment" configured

$ kubectl rollout history deployment
deployments "nginx-deployment"
REVISION   CHANGE-CAUSE
1          <none>
2          <none>

$ kubectl get deployments
NAME              DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment  5         6         5            4           22s

$ kubectl rollout status deployment nginx-deployment
deployment "nginx-deployment" successfully rolled out

$ kubectl get deployments
NAME              DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment  5         5         5            5           33s

$ kubectl describe deployments nginx-deployment
Events:
Scaled up replica set nginx-deployment-598744d4b8 to 5
Scaled up replica set nginx-deployment-d4779896 to 2
Scaled downreplica set nginx-deployment-598744d4b8 to 4
Scaled up replica set nginx-deployment-d4779896 to 3
Scaled downreplica set nginx-deployment-598744d4b8 to 3
Scaled up replica set nginx-deployment-d4779896 to 4
Scaled downreplica set nginx-deployment-598744d4b8 to 2
Scaled up replica set nginx-deployment-d4779896 to 5
Scaled downreplica set nginx-deployment-598744d4b8 to 1
Scaled down replica set nginx-deployment-598744d4b8
```

But what if we discover this new version has a problem? We should
perform a roll back to restore the last version of the deployment. **kubectl**

`rollout` makes it simple to inspect your deployment history, and to roll back to a previous version.

Let's first inspect our current revision, then compare it to the previous revision, and finally undo the new deployment.

```
$ kubectl rollout history deployment
deployments "nginx-deployment"
REVISION  CHANGE-CAUSE
1         <none>
2         <none>

$ kubectl rollout history deployment
    nginx-deployment --revision=2
deployments "nginx-deployment" with revision #2
Pod Template:
  Labels:       app=nginx
        pod-template-hash=80335452
  Containers:
   nginx:
    Image:      nginx:1.15.0
    Port:       <none>
    Host Port:  <none>
    Environment:        <none>
    Mounts:     <none>
  Volumes:      <none>

$ kubectl rollout history deployment
    nginx-deployment --revision=1
deployments "nginx-deployment" with revision #1
Pod Template:
  Labels:       app=nginx
        pod-template-hash=1543008064
  Containers:
   nginx:
    Image:      nginx:1.7.9
    Port:       <none>
    Host Port:  <none>
    Environment:        <none>
    Mounts:     <none>
  Volumes:      <none>

$ kubectl rollout undo deployment
    nginx-deployment --to-revision=1
deployment.apps "nginx-deployment"
```

Scaling and autoscaling

Kubernetes makes adjusting the number of replicas in your deployments easy. You can scale a deployment manually using kubectl, or automatically based on resource usage with the Horizontal Pod Autoscaler resource.

Manual scaling

To manually scale the previous deployment up to eight replicas, use the **`kubectl scale`** command.

```
$ kubectl scale deployment
    nginx-deployment --replicas=8
deployment.extensions "nginx-deployment" configured

$ kubectl rollout history deployment
deployment.extensions "nginx-deployment" scaled

$ kubectl get deployments
NAME              DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment  8         8         8            8           15m
```

Autoscaling

The Horizontal Pod Autoscaler is a new resource in modern versions of Kubernetes that manages the number of replicas in your deployment automatically. The autoscaler automatically scales the number of pods based on resource utilization. This was originally limited to tracking CPU utilization, but recently gained the ability to scale based on memory usage and custom metrics (one common use-case is to scale pods based on the number of items in a work queue).

Autoscaling is an advanced, new feature whose configuration settings are changing quickly, so anything written in this guide is likely to be outdated soon. When researching online, also take into account your production Kubernetes cluster version—the autoscaling features available to you might be different than you expect.

Services

Overview

Now, forget everything you just learned about deployments.

Most introductions try to merge together the topic of deployments and services since they go together so well. However, it's useful to understand that the two are unrelated, isolated components each with distinct responsibilities.

A deployment is responsible for keeping a set of pods running. A service is responsible for enabling network access to a set of pods. We could avoid using a service by enabling network access to each pod manually, however when pods are created and destroyed (say, by a deployment), we'll have to re-enable networking for each of them. That's not ideal.

Services let you define networking rules for pods based on their labels. Whenever a pod with a matching label appears in the Kubernetes cluster, the service will detect it and start using it to handle network requests made to the service.

A service's **type** determines how the service is exposed—whether it is only accessible within the cluster, accessible through a static port on each node (effectively enabling external access through this port), or using a load balancer from a cloud provider.

To illustrate how a service works, we'll manually create a few pods, and then enable external network access to them via a service.

Each pod's container will simply output the pod's hostname when a request is made to port 3000. This will let us see which pod our service has routed the network request to.

Network request
to service

Service
Forward network request
to a pod matching the selector

```
"select pod where
  label = abc;"
```

| Pod | Pod | Pod |
| Pod | Pod | Pod |
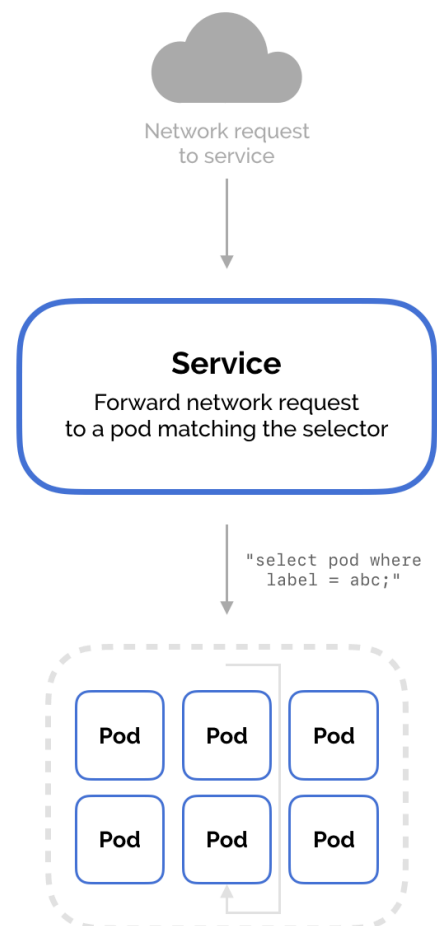
First, we'll create the service that will forward network requests to our hostname pods.

```
kind: Service
apiVersion: v1
metadata:
  name: hostname-service
spec:
  # Expose the service on a static port on each node
  # so that we can access the service from outside the cluster
  type: NodePort
  selector:
    app: echo-hostname
  ports:
    # Three types of ports for a service
    # nodePort   - a static port assigned on each the node
    # port       - port exposed internally in the cluster
    # targetPort - the container port to send requests to
    - nodePort: 30163
      port: 8080
      targetPort: 80
```

Create this service with `kubectl apply -f service.yaml`.

Now, we'll create two pods that the service will use to handle network traffic. Pay particular attention to the relationship between the service's selector field and the pods' labels—this is how the service discovers the pods it can use.

```
# Create two pods —— one for version 1.0.1 and
# one for 1.0.2 of our Docker image. They serve
# slightly different output so we can tell which
# pod is handling which request.

kind: Pod
apiVersion: v1
metadata:
  name: hostname-pod-101
  labels:
    # These labels are used by the service to select
    # pods that can handle its network requests
    app: echo-hostname
    app-version: v101
spec:
  containers:
    - name: nginx-hostname
      image: kubegoldenguide/nginx-hostname:1.0.1
      ports:
        # nginx runs on port 80 inside its container
        - containerPort: 80
---
kind: Pod
apiVersion: v1
metadata:
  name: hostname-pod-102
  labels:
    app: echo-hostname
    app-version: v102
spec:
  containers:
    - name: nginx-hostname
      image: kubegoldenguide/nginx-hostname:1.0.2
      ports:
        - containerPort: 80
```

Create these two pods with `kubectl apply -f pods.yaml`.

How is a request to a service routed through Kubernetes?

Once everything has finished being created, let's inspect what Kubernetes has done behind the scenes.

```
$ kubectl get all
NAME                     READY     STATUS    IPNODE
pod/hostname-pod-101     1/1       Running   172.17.0.2
pod/hostname-pod-102     1/1       Running   172.17.0.5

NAME                        TYPE          CLUSTER-IP        EXTERNAL-IP
service/hostname-service    NodePort      10.100.213.239   <none>

                            PORT(S)       AGE           SELECTOR
                            8080:30163/TCP 46m                      app=echo-hostname

$ minikube dashboard --url
http://192.168.99.100:30000   # 192.168.99.100 is the cluster's IP
```

To verify that everything's working, let's make a few requests to the cluster's IP and the service's external port. We'll explain how this works soon, but first, run the request a few times and make sure that the service is routing requests between the two different pods.

```
# Remember to replace 30163 with the port specified
# after the ':' in the PORT(s) column of your service
$ curl 'http://192.168.99.100:30163'
Hostname: dc534047acbb, date: Sat Jun 16 21:57:15 UTC 2018

$ curl 'http://192.168.99.100:30163'
Hostname: dc534047acbb
```

Now that we know it's working, let's backtrack and understand what happens when we're making these requests.

What path does a successful request take?

1. A request comes from outside the cluster to the cluster's IP address, in this case 192.168.99.100. The request is made to port 30163, the port that was assigned due to the service having type NodePort. (We manually specified a `nodePort`, but Kubernetes can also do this automatically.)

2. The request, once inside the cluster, is routed to the service's `clusterIP` address, 10.100.213.239. This address is the internal-facing IP for the service. It is made on port 8080, defined in the service's YAML.

Request to service
e.g. from your shell

Service's **nodePort**
(automatically assigned) ·········· :30163

Cluster

Cluster
- Access via cluster's IP address
  or domain name, e.g. 192.168.99.100
- Cluster port available because service
  is of type NodePort

Service's **port** ·········· :8080

Service

Service
- Access only within the cluster via the
  service's clusterIP
- Available at 10.100.213:8080
  or hostname-service.default:8080
- Service exposes port 8080,
  targets pod port 80

Pod's **containerPort**
Service's **targetPort** ·········· :80 ⟋ ⟍ :80

Pod **Pod**

Pod
- Access via the pod's IP address inside
  the cluster
- Pod exposes port 80 (which nginx uses
  by default)

3. The service chooses a pod to route the request to based on its label selector, in this case any pod with the label `app` set to `echo-hostname`.
4. Once a pod is selected (say, the first pod with IP address 172.17.0.2), this request is sent to port 80 on that pod. Port 80 is defined in the service's `targetPort`, and also in the container's `containerPort`.
5. The request hits the pod, which sends a response.

If you are accessing the service from outside the cluster (say, using curl from a normal shell on your computer):

- 10.100.213.239:8080, 10.100.213.239:30163, or anything using the 10.100.213.239 IP address will not work. This is because that IP address is internal to the cluster—it will only work from a pod (or directly on a node) inside the cluster.
- You need to find the cluster's IP address, and access the service through that. Running a local Kubernetes cluster with minikube, you can use `minikube dashboard --url` to find the IP address of your cluster. In the default scenario, this is 192.168.99.100.
- The only address that works for accessing the service from outside the cluster is 192.168.99.100:30163. This is the cluster's IP address and the external NodePort that was set up when Kubernetes created the service.

If you are accessing the service from inside the cluster—for example from a pod—you can use the service's IP address (from the cluster IP column in `kubectl get service`) and the port we defined for the service.

- 10.100.213.239:8080 hits the service, then routes requests to pods matching its label selector
- The URL `hostname-service.default` will also hit the service, since Kubernetes creates DNS records for this in the background
- You can access a pod directly after retrieving its IP address from `kubectl get pods -o wide`, and using port 80—the containerPort we specified in the pod's YAML

If it helps you to enumerate the scenarios, this table summarizes whether a request to each address would succeed, and the reasons why.

| Origin	Address	Works?	Why?
Outside cluster	192.168.99.100:30163	Yes	Uses cluster IP address and service's nodePort from outside the cluster
Outside cluster	192.168.99.100:8080	No	Uses internal service port from outside the cluster
Outside cluster	10.100.213.239:*	No	Uses internal service IP address from outside the cluster
Inside cluster	10.100.213.239:8080	Yes	Uses internal service IP address and internal port from inside the cluster
Inside cluster	10.100.213.239:30163	No	Uses external nodePort on with an internal IP address
Inside cluster	192.168.99.100:*	No*	May not work because it uses the cluster's IP address from inside the cluster. Depends on what the IP address is and firewall configuration.
Inside cluster	172.17.0.2:80	Yes	Accesses one pod directly using its cluster-internal IP address and containerPort

Deployments and Services Together

Kubernetes provides deployments, which let us automatically manage the creation, deletion, and replication of pods. It also provides services, which lets us dynamically route network requests to a set of pods. Naturally, these two components fit well together.
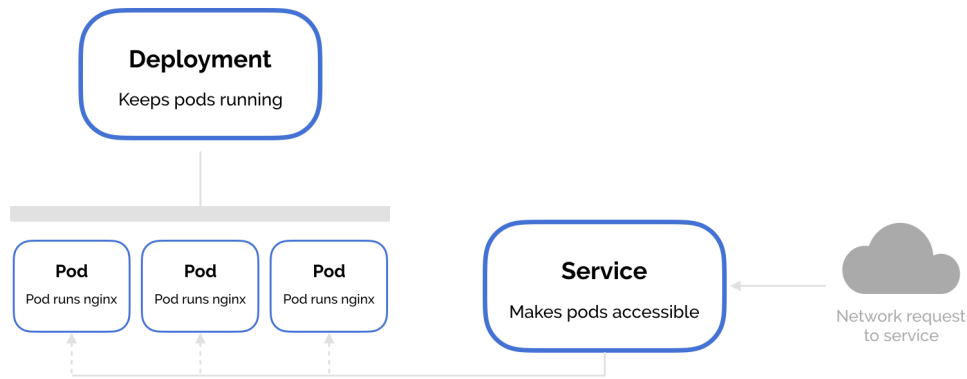
Notice that there's no connection between deployments and services—the two exist independently! This is a really useful decoupling. One thing it enables is canary deployments. You add a new deployment version of a pod and run it alongside your existing deployment, but have both deployments handle requests to the service. Once deployed, you can verify the new deployment on a small proportion of the service's traffic, and gradually scale up the new deployment while decreasing the old one.

Let's create a deployment and service that work together. We'll create two deployments of the nginx-hostname application from the previous example. The first deployment is the original application version, 1.0.1, scaled to three replicas, and the second deployment is a single replica of version 1.0.2 of the application.
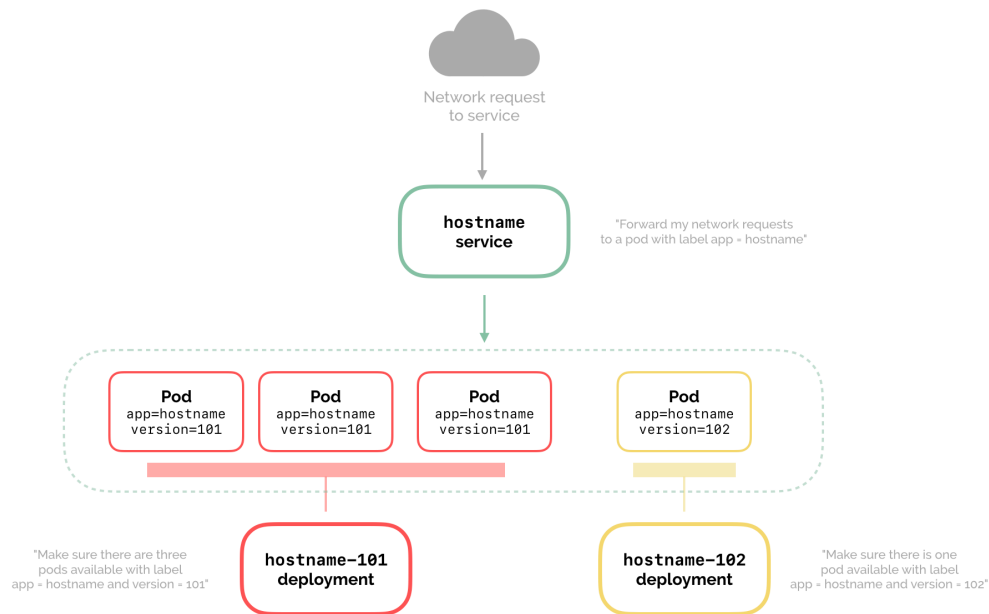
Create the first deployment, and pay particular attention to its labels.

```
# Deployment 1: three replicas of v1.0.1 of the app
# Deployment 2: one replica of v1.0.2 of the app
# Service: any pod with the label `app: hostname`
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: hostname-101-deployment
spec:
  replicas: 3
  selector:
    # Like saying "Make sure there are three pods running
    # with the label app = hostname and version = v101"
    matchLabels:
      app: hostname
      version: v101
  template:
    metadata:
      labels:
        # The `app` label is used by both the service
        # and the deployment to select the pods they operate on.
        app: hostname
        # The `version` label is used only by the deployment
        # to control replication.
        version: v101
    spec:
      containers:
        - name: nginx-hostname
          image: kubegoldenguide/nginx-hostname:1.0.1
          ports:
            - containerPort: 80
```

Create the second deployment, which differs only in its image version, version label, and replica count.

```
# Second deployment: a single replica of v1.0.2
# of the application
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: hostname-102-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hostname
      version: v102
  template:
    metadata:
      labels:
        app: hostname
        version: v102
    spec:
      containers:
        - name: nginx-hostname
          image: kubegoldenguide/nginx-hostname:1.0.2
          ports:
            - containerPort: 80
```

Now, create the service, which will forward network requests to the service to any pod with the label app set to hostname. In our case, we have four such pods created via our deployments—three of them with the original application version, and one with the new application version.

```
# Expose both deployments through a single service!
kind: Service
apiVersion: v1
metadata:
  name: hostname-service
spec:
  selector:
    # Equivalent to "select any pod where the
    # label app = hostname"
    app: hostname
  type: NodePort
  ports:
    - port: 8080
      targetPort: 80
```

We've now created three replicas of the first version of our application, a single replica of the second version, and exposed all of these to the network through a single unified interface. Let's access our deployed

application, and watch how the Kubernetes service automatically load balances our request between the running pods.

```
$ kubectl get services
NAME              TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)
hostname-service  NodePort   10.101.23.136  <none>        8080:30412
kubernetes        ClusterIP  10.96.0.1      <none>        443/TCP

$ curl 'http://192.168.99.100:30412'
Hostname: dc534047acbb, date: Sat Jun 16 21:57:15 UTC 2018

$ curl 'http://192.168.99.100:30412'
Hostname: dc534047acbb
```
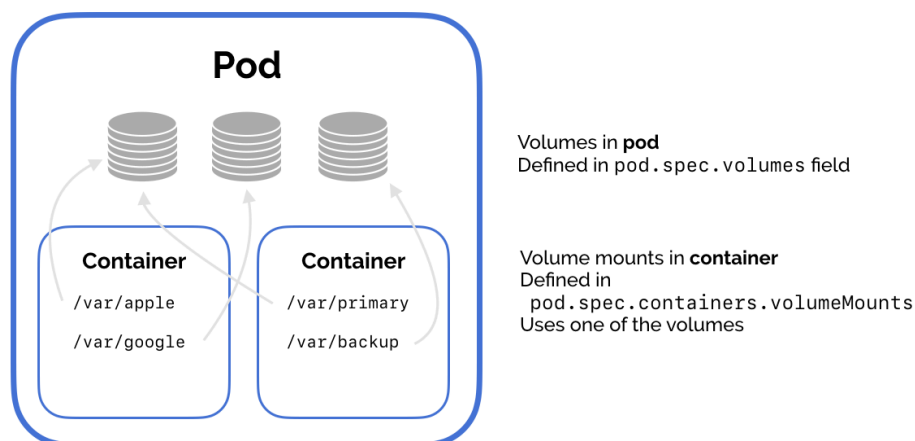
Storage

Volumes

Your containers can read and write to their filesystem any time they like. However, this filesystem is temporary and isolated—everything you've written will be lost if the container is restarted.

How does Kubernetes solve this so you can keep data across container restarts? Pods can define their volumes—filesystems that exist as long as the pod lives. These are mounted into a container, which provides a directory the container can read and write to.

A volume provides storage to your pod and the containers running inside it. The storage will live for as long as the pod lives—data will survive across container restarts but will be lost when the pod is destroyed. Further, this storage can be shared across containers, so if two containers are running in the same pod, they will be able to share files using the volume.

A volume is essentially a directory. Volumes are powerful because they can be provided by different sources, use different storage mediums, and have different default data, all while still appearing as a regular filesystem directory to the pod and its containers.

Within the container, volumes are mounted at a specific path. It's like a symbolic link from the container to the volume within the pod. A volume can also define permissions around read and write access. From there, the container can use that directory however they like.

Volumes in **pod**
Defined in `pod.spec.volumes` field

Volume mounts in **container**
Defined in
` pod.spec.containers.volumeMounts`
Uses one of the volumes

There are two sides to using a volume. Step one: the pod defines a volume. Step two: the container mounts that volume in its filesystem. The container then treats that volume as a regular directory.

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: simple-volume-pod
spec:
  # Volumes are declared at the pod level. They share its
  # lifecycle and are communal across containers inside it.
  volumes:
    # Volumes have a name and configuration
    # based on the type of volume.
    # In this example, we use the emptyDir volume type
    - name: simple-vol
      emptyDir: {}

  # Now, one of our containers can mount this volume
  # and use it like any other directory.
  containers:
    - name: my-container
      volumeMounts:
        # name must match the volume name set at the pod level
        - name: simple-vol
          # Where to mount this directory in our container
          mountPath: /var/simple

      # Now that we have a directory mounted at /var/simple,
      # write to a file inside it!
      image: alpine
      command: ["/bin/sh"]
      args: ["-c", "while true; do date >> /var/simple/file.txt;
sleep 5; done"]
```

Types of Volumes

In the above example, we used the simple `emptyDir` volume type. There are dozens of types of volumes you can use—each with different settings —that you can use as-needed. Every application developer uses a few key volume types in their career.

emptyDir

An emptyDir volume is an empty directory created in the pod. Containers can read and write files in this directory. The directory's contents are removed once the pod is deleted. This is often the right volume type for

general-purpose use, where you want the files to exist beyond container restarts and boundaries, but you don't need the files forever.

`nfs`

nfs volumes are network file system shares that are mounted into the pod. An nfs volume's contents might exist beyond the pod's lifetime—Kubernetes doesn't do anything in regards to the management of the nfs—so it can be a good solution for sharing, pre-populating, or preserving data.

`hostPath`

hostPath volumes directly mount the node's filesystem into the pod. While these are often not used in production since—as developers—we prefer not to worry about the underlying node, they can be useful for getting files from the underlying machine into a pod.

`configMap` and `secret`

configMap and secret are two essential volume types. In fact, they each have their own chapter in this book. They are useful for inserting configuration and secret data into your containers. When you read that chapter on secrets and configuration, it's worth remembering that the functionality is powered by volumes, and that they work just like any other volume!

Persistent Volumes

volumeMount	volume type: pvc	Gives pod control of persistent volume	Persistent storage resource owned by cluster
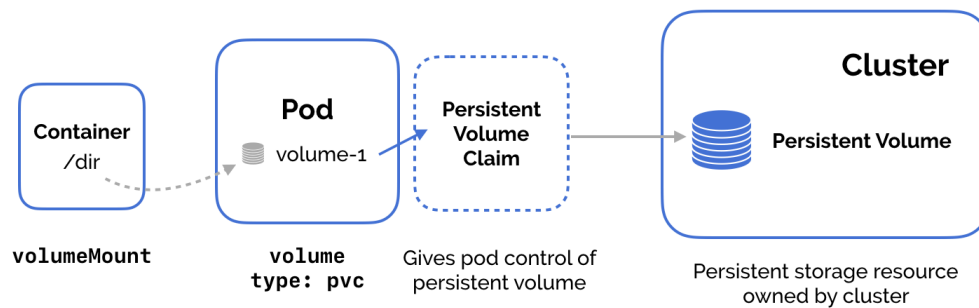
Persistent Volumes

A persistent volume is a piece of storage in your Kubernetes cluster, often provisioned by a Kubernetes administrator. Its defining characteristic is that it is storage that persists beyond containers, pods, and node restarts. In the same way that a node is a resource, a persistent volume is also a resource: it can be consumed by pods via Persistent Volume Claims.

```
# Declares a persistent volume is available for
# pods to use as a long-term storage solution.
# This persistent volume is backed by your local
# minikube cluster.
kind: PersistentVolume
apiVersion: v1
metadata:
  name: my-persistent-volume
spec:
  capacity:
    storage: 128M

  # The list of ways the persistent volume can be mounted
  accessModes:
    - ReadWriteOnce

  # Configuration settings for the persistent volume.
  # In this case, we're going to store the data on minikube.
  hostPath:
    path: /data/my-persistent-volume/
```

Persistent Volume Claims

A persistent volume claim is a request made by a user (i.e. a pod or other workload) for a persistent volume resource. A persistent volume claim consumes persistent volume resources based on the size and access mode required by the claim. Pods then use that persistent volume claim

as a volume, and the pod's containers mount that volume into their filesystem.

```yaml
# Declares a claim to use the persistent volume
# defined in the previous YAML.
# The pod in will eventually use this
# persistent volume claim when mounting the volume.
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-small-pvc
spec:
  # "Give me a persistent volume with at least
  # 64MB of space where I can read and write."
  resources:
    requests:
      storage: 64M
  accessModes:
    - ReadWriteOnce
```

When it comes to use the persistent volume, a pod does so via the **persistentVolumeClaim** volume type. From there, it works like any other volume!

```yaml
# Notice that if you delete and recreate this
# pod, your previous files will persist on the
# minikube cluster!
kind: Pod
apiVersion: v1
metadata:
  name: my-pv-user-pod
spec:
  volumes:
    # This volume is of type persistentVolumeClaim -- i.e.
    # we need a persistent volume.
    - name: a-persistent-volume
      persistentVolumeClaim:
        # Must match claim name from the PVC YAML
        claimName: my-small-pvc

  # Mount the volume into the container and use it
  containers:
    - name: pv-user
      volumeMounts:
        - name: a-persistent-volume
          mountPath: /var/forever

      image: alpine
      command: ["/bin/sh"]
      args: ["-c", "while true; do date >> /var/forever/
file.txt; sleep 5; done"]
```

Lifecycle

Using a persistent volume in your Kubernetes cluster follows the following lifecycle.

1. **Provisioning**

   Persistent volumes can be dynamically created by the cluster as it receives new requests, or statically created where an admin sets up a fixed number of persistent volumes of each type.

2. **Binding**

   The Kubernetes master watches for persistent volume claims that are lodged. When it receives a claim, it searches for a fitting persistent volume and binds the two together.

3. **Using**

   Pods then use the claim. The cluster will mount the bound volume into the pod, which then reads and writes to the persistent storage.

4. **Reclaiming**

   Once the pod is finished, the persistent volume claim is deleted and the persistent volume resources are made available to other users.

Configuration

Keeping your web application's configuration data separate from your source code is a sensible best practice for running production web services. It enables you to change configuration settings without redeployment, and also prevents leaking secret keys to the wrong environments or into source control. Kubernetes provides two mechanisms for this—ConfigMaps and Secrets. ConfigMaps and Secrets are virtually identical, except that Secrets are naturally more suited to storing sensitive data like API keys.

ConfigMaps

A ConfigMap is a key-value dictionary whose data is injected into your container's environment when it runs. The first step to using a ConfigMap is to create the ConfigMap resource inside your Kubernetes cluster. The next step is for a pod to consume it by mounting it as a volume or getting its data injected through environment variables.

Creating a ConfigMap

ConfigMaps are very flexible in how they are created and consumed. You can,

- create a ConfigMap directly in YAML — this is our preferred solution
- create a ConfigMap from a directory of files
- create a ConfigMap from a single file
- create a ConfigMap from literal values on the command line

If your application has specific needs, it is well worth searching online for alternative usages of ConfigMaps. In our case, we prefer to keep our configuration settings tracked in version control and consistent with the rest of our Kubernetes resources by defining them directly in YAML.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: example-configmap
data:
  # Configuration values can be set as key-value properties
  database: mongodb
  database_uri: mongodb://localhost:27017

  # Or set as complete file contents (even JSON!)
  keys: |
    image.public.key=771
    rsa.public.key=42
```

We can then create this ConfigMap like any other Kubernetes resource—
`kubectl apply -f configmap.yaml`.

Using a ConfigMap through a volume

Once created, add the ConfigMap as a volume in your pod's specification.
Then, mount that volume into the container's filesystem. Each property
name in the ConfigMap will become a new file in the mounted directory,
and the contents of each file will be the value specified in the ConfigMap.

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-using-configmap
spec:
  # Add the ConfigMap as a volume to the Pod
  volumes:
    # `name` here must match the name
    # specified in the volume mount
    - name: example-configmap-volume
      # Populate the volume with config map data
      configMap:
        # `name` here must match the name
        # specified in the ConfigMap's YAML
        name: example-configmap

  containers:
    - name: container-configmap
      image: nginx:1.7.9
      # Mount the volume that contains the configuration data
      # into your container filesystem
      volumeMounts:
        # `name` here must match the name
        # from the volumes section of this pod
        - name: example-configmap-volume
          mountPath: /etc/config
```

Create the pod, start an interactive shell inside it, and inspect the contents of the ConfigMap that was mounted as a volume. Each entry in the ConfigMap was mounted as a separate file in the **/etc/config** directory, with file contents corresponding to what we specified in the YAML.

```
$ kubectl apply -f pod.yaml
pod "pod-using-configmap" created

$ kubectl exec -it pod-using-configmap sh
# ls /etc/config
database
database_uri
keys
# cat /etc/config/*
mongodb                 <- contents of `database`
http://localhost:27017  <- contents of `database_uri`
image.public.key=771    <- contents of `keys`
rsa.public.key=42        -
```

Using a ConfigMap through environment variables

Another useful way of consuming a ConfigMap you have created in your Kubernetes cluster is by injecting its data as environment variables into your container.

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-env-var
spec:
  containers:
    - name: env-var-configmap
      image: nginx:1.7.9
      envFrom:
        - configMapRef:
            name: example-configmap
```

We can access and view the injected environment variables once the pod is running.

```
$ kubectl apply -f pod-env-var.yaml
pod "pod-env-var" created

$ kubectl exec -it pod-env-var sh
# env
# ...
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
DATABASE=mongodb
DATABASE_URI=http://localhost:27017
image.public.key=771
rsa.public.key=42
NGINX_VERSION=1.7.9-1~wheezy
# ...
```

Secrets

If you understand creating and consuming ConfigMaps, you also understand how to use Secrets. The primary difference between the two is that Secrets are designed to hold sensitive data—things like keys, tokens, and passwords.

Kubernetes will avoid writing Secrets to disk—instead using tmpfs volumes on each node, so they can't be left behind on a node. However, etcd, Kubernetes' configuration key-value store, stores secrets in plaintext. Further, if there are multiple replicas of etcd, data is not necessarily replicated over SSL. Make sure your administrator restricts etcd access before using Secrets.

Kubernetes will only send Secrets to a node when one of the node's pods explicitly requires it, and removes the Secret if that pod is removed. Plus, a Secret is only ever visible inside the Pod that requested it.

Creating a Secret

As we did with a ConfigMap, let's create a YAML file to configure secrets for an API access key and token. When adding Secrets via YAML, they must be encoded in base64. base64 is not an encryption method and does not provide any security for what is encoded—it's simply a way of presenting a string in a different format. Do not commit your base64-encoded secrets to source control or share them publicly.

Encode strings in base64 using the `base64` command in your shell, or with an online tool like base64encode.org.

Let's base64 encode our access key and API token.

```
$ echo "OUR_API_ACCESS_KEY" | base64
T1VSX0FQSV9BQ0NFU1NfS0VZCg==

$ echo "SECRET_7t4836378erwdser34" | base64
U0VDUkVUXzd0NDgzNjM3OGVyd2RzZXIzNAo=
```

Now, add these encoded strings to the YAML file for our Secret.

```
kind: Secret
apiVersion: v1
metadata:
  name: api-authentication-secret
type: Opaque
data:
  key: T1VSX0FQSV9BQ0NFU1NfS0VZCg==
  token: U0VDUkVUXzd0NDgzNjM3OGVyd2RzZXIzNAo=
```

Using the Secret is then the same as using a ConfigMap, just with slightly
different field names.

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-using-secret
spec:
  # Add the Secret as a volume to the Pod
  volumes:
    # `name` here must match the name
    # specified in the volume mount
    - name: api-secret-volume
      # Populate the volume with config map data
      secret:
        # `secretName` here must match the name
        # specified in the secret's YAML
        secretName: api-authentication-secret

  containers:
    - name: container-secret
      image: nginx:1.7.9
      volumeMounts:
        # `name` here must match the name
        # from the volumes section of this pod
        - name: api-secret-volume
          mountPath: /etc/secret
```

The Secret will be mounted inside the **/etc/secret** directory in our
container, and we can access those files from our container.

```

```
$ kubectl exec -it pod-using-secret sh
# ls /etc/secret
key   token
cat /etc/secret/key
OUR_API_ACCESS_KEY
cat /etc/secret/token
SECRET_7t4836378erwdser34
```

# Jobs

## Overview

So far, we've used Kubernetes to manage processes that run forever. In particular, in an ideal world our web application would run forever and never die. But sometimes, as an application developer, you want to execute finite tasks. You also want to guarantee that these tasks were completed successfully, and have them automatically re-attempted if they failed.

Kubernetes has Jobs, an object the ensures a finite task completes successfully. On top of Jobs, it also adds CronJobs, which run Jobs at a specific time or periodic schedule.

## Jobs

Jobs are the core mechanism for executing finite tasks in Kubernetes. A Job creates one or more Pods, and guarantees that a specified number of them complete successfully. Common real-world use-cases of Jobs are batch processing, long-running calculations, backup and file sync operations, processing items from a work queue, and file upload to external services (for example, a monitoring or log collection service).

There are three types of jobs: non-parallel, parallel with fixed completion count, and parallel with a work queue. These are characterised by the number of pods they create, what their "completed" state looks like, and the degree of parallelism when they execute.

| Non-parallel | Parallel with fixed completion count | Parallel with work queue |
|---|---|---|
| One pod created. Extra pods only created in case of failure. | Specified number of pods created. | Specified number of pods created. |
| Job complete when Pod is successful. | Specified number of successful terminations. | ≥1 pod successful, all others terminated. |
| `completions = 1`<br>`parallelism = 1` | `completions = n`<br>`parallelism = m` | `completions = 1`<br>`parallelism = n` |

### Success and failure

A job communicates its success or failure state through the process's exit code. If the exit code is non-zero, that means the process had an error, which will cause the container to fail, and hence the Pod and Job will fail. A zero exit code communicates success. Jobs can also fail because of resource restrictions, inability to schedule the pod, a node dying, and other environmental issues. Kubernetes takes care of all of these. When writing a job, all you need to do is ensure that your image returns the correct exit code, and make sure that your task can be restarted in a new pod in case the first one fails.

A pod created by a job must have its `restartPolicy` be `OnFailure` or `Never`. If the `restartPolicy` is `OnFailure`, a failed container will be re-run on the same pod. If the `restartPolicy` is `Never`, a failed container will be re-run on a new pod.

### Using Jobs

Let's create a job that illustrates how Kubernetes ensures your tasks will run successfully. Our job simulates a dice roll, and if we get a six, the job is successful. Any other number means the task has failed.

A job has two interesting fields in its configuration: `completions` and `parallelism`. `completions` specifies how many times the created pod must terminate successfully for the job to be considered a success. `parallelism` specifies how many pods can run simultaneously to complete the job. Other than that, for its template a job just re-uses the pod spec configuration you're already familiar with. Make sure the pod uses exit codes correctly and has a `restartPolicy` of `OnFailure` or `Never`.

```
kind: Job
apiVersion: batch/v1
metadata:
  name: dice-job
spec:
  # At least one pod should exit successfully, and only
  # one pod should run at a time.
  completions: 1
  parallelism: 1

  # The spec for the pod that is created when the job runs.
  template:
    metadata:
      name: dice-pod
    spec:
      # Pods used in jobs *must* use a restart
      # policy of "OnFailure" or "Never"
      # OnFailure = Re-run the container in the same pod
      # Never = Run the container in a new pod
      restartPolicy: Never

      # This container simulates a dice roll and
      # returns a zero exit code (i.e. success)
      # when we get a six.
      containers:
        - name: dice
          image: alpine
          command: ["/bin/sh"]
          # if dice_roll == 6, return 0. else, return 1.
          args: ["-c", "if [ \"$(shuf -i 1-6 -n 1)\" = \"6\" ];
then exit 0; else exit 1; fi"]
```

Now let's create the job and live-tail the output to see how Kubernetes
will automatically recreate pods until we get one whose dice roll was a six
and the pod had status `Completed`.

```
$ kubectl apply -f job.yaml
job.batch "dice-job" created
$ kubectl get pods -w
dice-job-42z4p  Pending
dice-job-42z4p  ContainerCreating
dice-job-42z4p  Error
dice-job-vsd9j  Pending
dice-job-vsd9j  ContainerCreating
dice-job-vsd9j  Error
dice-job-7262x  Pending
dice-job-7262x  ContainerCreating
dice-job-7262x  Error
dice-job-kgczd  Pending
dice-job-kgczd  ContainerCreating
dice-job-kgczd  Completed
```

CronJobs

CronJobs are fundamentally the same as regular jobs, except they can be run at a specific time or on a recurring schedule. They get their name and syntax from the Unix utility cron—if you have any familiarity with Unix's cron, your knowledge will directly translate.

Using Kubernetes CronJobs—over regular Unix cron jobs—solves the most common form of cron job failure, the machine dying. Kubernetes abstracts away the machine your jobs run on, so you don't need to worry about whether the machine is up and running when you need to execute your process—Kubernetes will make sure it gets run.

One caveat for CronJobs is that, in rare cases, more than one Job might get created for a single CronJob task. Ensure your CronJobs can be run repeatedly but produce the same result each time.

cron syntax

Don't worry if you find cron syntax strange or confusing. Just have a basic understanding and then use online tools like cronmaker.com to write your scheduling configuration for you. (Although, if you're taking the Certified Kubernetes Application Developer Exam, it's worth knowing some cron syntax since you can't use online tools in the exam environment.)

Using CronJobs

Let's create a CronJob that'll ping Google every minute and check that their site hasn't gone down. Notice that a CronJob includes two nested configuration templates. First, it reuses the Job spec configuration, and that Job spec configuration reuses the Pod spec configuration.

```
kind: CronJob
apiVersion: batch/v1beta1
metadata:
  name: google-check-cronjob
spec:
  # CronJob configuration -- uses cron syntax
  # to specify job schedule. In this case, run every minute.
  # (I always use cronmaker.com because it's not worth the
  # effort to learn cron syntax.)
  schedule: "*/1 * * * *"

  # How many completed jobs we want to keep around.
  # Let's keep the last ten minutes' jobs.
  successfulJobsHistoryLimit: 10

  # The job to be created every minute. This reuses
  # configuration format for regular jobs.
  jobTemplate:
    metadata:
      name: google-check-job
    spec:
      # The pod to be created to execute the job
      template:
        metadata:
          name: google-check-pod
        spec:
          # OnFailure = Re-run failed container in the same pod
          restartPolicy: OnFailure

          # This container checks that Google is up,
          # just in case no one notices.
          containers:
            - name: google-check-container
              image: alpine
              command: ["/bin/sh"]
              args: ["-c", "ping -w 1 google.com"]
```

Now let's create the CronJob and watch it work. Don't forget to kill the CronJob afterwards!

```
$ kubectl apply -f cronjob.yaml
cronjob.batch "google-check-cronjob" created
$ kubectl get pods -w
google-check-cronjob-1530488700-rxbq7  Running
google-check-cronjob-1530488700-rxbq7  Completed
google-check-cronjob-1530488760-mx48x  Pending
google-check-cronjob-1530488760-mx48x  ContainerCreating
google-check-cronjob-1530488760-mx48x  Running
google-check-cronjob-1530488760-mx48x  Completed
google-check-cronjob-1530488820-6h77m  Pending
google-check-cronjob-1530488820-6h77m  ContainerCreating
google-check-cronjob-1530488820-6h77m  Running
google-check-cronjob-1530488820-6h77m  Completed
$ kubectl delete -f cronjob.yaml
cronjob.batch "google-check-cronjob" deleted
```

Resource Quotas

In a previous chapter, we looked at namespaces. Namespaces subdivide a Kubernetes cluster into isolated sections, each of which looks as though it's the full cluster. This helps to avoid naming collisions and isolate resources.

When setting up a Kubernetes cluster, administrators create namespaces for each team that uses the cluster. This helps to avoid naming collisions across teams, and provides some privacy for what a team is running. The other benefit is that it allows the administrator to monitor and restrict the amount of resources consumed by each team. Restricting resource usage is done by setting up a resource quota for each namespace.

A resource quota limits the amount of resources that namespace can use. Resource quotas can limit anything in the namespace, including the total count of each type of object, the total storage used, and the total memory or CPU usage of containers in the namespace.

In YAML, a container defines the minimum amount of CPU or memory it needs through its `resources.requests` property, and its maximum CPU and memory usage through the `resources.limits` property. If the resource quota created for the namespace defines a value for `limit` or `request` for CPU or memory, then every container in that namespace must include these properties in order to be scheduled.

When the Kubernetes scheduler selects a node for a pod to run in, it looks at the sum of CPU and memory requests for each container in that pod to see if it will be able to run. As a result, it is a good practice to set both of these for your pods, even if you expect your pod's resource consumption to be relatively light. This will allow Kubernetes to intelligently schedule pods; if you don't, some pods might never be successfully scheduled because other pods are consuming more than they strictly need.

Creating resource quotas

Resource quotas are defined per namespace, so the first step to creating a resource quota is to set up a namespace. We'll use this namespace for everything in this chapter, so if you're following along, make sure to specify the `metadata.namespace` field and use `--namespace=red-team`.

```
kind: Namespace
apiVersion: v1
metadata:
  name: red-team
spec: {}
```

```
$ kubectl create -f namespace.yaml
namespace "red-team" created
```

Once the namespace is established, create a resource quota in that namespace. We'll limit the total CPU and memory usage of this namespace, as well as the total number of pods that can be created. A resource quota can place restrictions on virtually any aspect of the namespace—count per object type, compute resource, and storage resource limits can be established for a resource.

```
kind: ResourceQuota
apiVersion: v1
metadata:
  name: red-team-resource-quota
  # Important! Define the resource quota for the namespace.
  namespace: red-team
spec:
  # The set of hard limits for each resource
  # This is a map of resource type to request/limit
  hard:
    # Resource constrains can be set up for "counts" of objects.
    # In this case, limit the count of pods to five.
    pods: 5

    # Constrain the sum of `requests` across objects
    # in the namespace.
    # An object must now `request` its minimum
    # resource requirements.
    # If there aren't sufficient available resources
    # in the namespace based on the `request` an object
    # may not run or may be killed.
    "requests.cpu": "2"
    "requests.memory": 1024m

    # Constrain the sum of `limits` across objects in
    # the namespace.
    # A `limit` defines the maximum resources an object can
    # use when it's running -- if it starts to exceed
    # this limit, it might get throttled.
    "limits.cpu": "4"
    "limits.memory": 2048m
```

Now, in the namespace that has a resource quota, let's create a pod that doesn't specify its resource limits.

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-illegal
  # Important! Create this pod in the right namespace.
  namespace: red-team
spec:
  # This pod will fail to be created because it doesn't have
  # the `request` or `limit` property set -- this is required by
  # our resource quota.
  containers:
    - name: nginx-illegal
      image: nginx:1.7.9
```

Let's inspect why this failed to run.

```
$ kubectl create -f pod-illegal.yaml
Error from server (Forbidden): error when creating
"pod-illegal.yaml": pods "pod-illegal" is forbidden:
failed quota: red-team-resource-quota: must specify
requests.cpu,requests.memory,limits.cpu,limits.memory
```

We can see that containers must define their resource limits if the resource quota is defined. Let's create two pods, the first will use the majority of the resource quota.

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-one
  # Important! Create this pod in the right namespace.
  namespace: red-team
spec:
  containers:
    - name: nginx-pod-one
      image: nginx:1.7.9
      resources:
        # This container requires at least 768m memory
        # and 0.5 CPU resource to run
        requests:
          memory: 768m
          cpu: "0.5"
        # This container can use at most 1024m memory
        # and 2 CPU resources when it's running
        limits:
          memory: 1024m
          cpu: "2"
```

Now create the pod, and then inspect our resource quota to see the effects.

```
$ kubectl create -f pod-one.yaml
pod "pod-one" created
$ kubectl describe quota --namespace=red-team
# Important! Notice we specified --namespace=red-team above
Name:            red-team-resource-quota
Namespace:       red-team
Resource         Used   Hard
--------         ----   ----
limits.cpu       2      4
limits.memory    1024m  2048m
pods             1      5
requests.cpu     500m   2
requests.memory  768m   1024m
```

The second pod requests at least 512MB of memory to run. When this request is added to the other requests in the namespace—namely the 768MB required by the first pod—it will exceed the memory requests limit defined by the resource quota.

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-two
  namespace: red-team
spec:
  # This pod is identical to `pod-one` except that it has a
  # slightly lower minimum memory requirement.
  #
  # If created after pod-one, this pod will fail!
  # This is because in their memory `requests`,
  # pod-one requires 768m, pod-two requires 512m,
  # but the resource quota defines the `requests.memory`
  # cap at 1024m.
  containers:
    - name: nginx-pod-two
      image: nginx:1.7.9
      resources:
        # At least 512m of memory is required for this pod.
        requests:
          memory: 512m
          cpu: "0.5"
        limits:
          memory: 1024m
          cpu: "1"
```

Now try to create the second pod.

```
$ kubectl create -f pod-two.yaml
Error from server (Forbidden): error when creating "pod-two.yaml": pods
"pod-two" is forbidden: exceeded quota: red-team-resource-quota,
requested: requests.memory=512m, used: requests.memory=768m, limited:
requests.memory=1024m
```

Debugging resource quotas

There are several kubectl commands that are extremely useful if it looks like your pod was created successfully, but in fact doesn't seem to be running. The first—as we've already seen—is to describe the resource quota for the namespace.

```
$ kubectl describe quota --namespace=red-team
Name:            red-team-resource-quota
Namespace:       red-team
Resource         Used   Hard
--------         ----   ----
limits.cpu       2      4
limits.memory    1024m  2048m
pods             1      5
requests.cpu     500m   2
requests.memory  768m   1024m
```

It is also useful to look at the "events" section of the pod that is being created—it can contain useful information as to why scheduling might have failed.

```
$ kubectl describe my-pod --namespace=red-team
# ... truncated output
Events:
  Type     Reason            Age            From             Message
  ----     ------            ----           ----             -------
  Warning  FailedScheduling  2s (x8 over 1m)  default-scheduler  0/1
nodes are available: 1 Insufficient cpu.
```

One final command that gives an overview of the entire cluster is **kubectl describe nodes**. This lets you inspect all of the nodes running in your cluster, and see what limits might be breached. This is an extremely useful debugging tool because it displays the CPU and memory resources across all namespaces—it might be that the node itself doesn't have space for your pod to be scheduled because of what's happening in other namespaces.

```
$ kubectl describe nodes
# ... truncated output
  red-team                    pod-one
500m (25%)    2 (100%)    768m (0%)  1024m (0%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests  CPU Limits  Memory Requests    Memory Limits
  ------------  ----------  ----------------   -------------
  1315m (65%)   2 (100%)    167772160768m (8%)  178257921024m (8%)
Events:          <none>
```

Debugging the interaction between namespaces, resource quotas, and your pods is an important tool for running your application correctly.

Service Accounts

When you, a human, make a request to Kubernetes' API server using `kubectl`, you are authenticated through a User Account. In contrast, when a process running inside a container needs to make a request to Kubernetes' API Server, a Service Account is used.

Every pod in your Kubernetes cluster has a Service Account it uses. Most of the time, this is set to `default`. If you need to change the Service Account, simply set the pod's `spec.serviceAccountName` field to the name of the Service Account you want to use.

As an application developer, you won't often need to create a new Service Account. If you do, the YAML is straightforward.

```
kind: ServiceAccount
apiVersion: v1
metadata:
  name: my-service-account
```

Using a specific Service Account is similarly straightforward.

```
kind: Pod
apiVersion: v1
metadata:
  name: use-service-account-pod
spec:
  # Set the service account containers in this pod
  # use when they make requests to the API server
  serviceAccountName: my-service-account
  containers:
    - name: container-service-account
      image: nginx:1.7.9
```

To see which service account a pod is using, run `kubectl get pods <pod> -o yaml` and look for the pod's `spec.serviceAccount` field.

The other important detail to note is that service accounts are defined per-namespace, while user accounts are not namespaced.

Network Policies

Networking Overview

As an application developer, there are three principles to keep in mind for your mental model of networking in Kubernetes.

1. Containers on the same pod can talk to each other via localhost
2. Pods on the same node can talk to each other via a pod's IP address
3. Pods on *different* nodes can talk to each other via a pod's IP address

Adopting these principles gives us several benefits.

- Communication between containers in a pod can occur via localhost. This lets us use multi-container pod design patterns. Containers running in the same pod will need to coordinate their port usage. But between pods, no port coordination is required!
- Pods can easily communicate with each other even if they aren't on the same node. This is different to regular Docker networking, which requires containers to be on the same machine (further, regular Docker networking doesn't enable localhost communication).
- Each pod in a Kubernetes cluster is assigned an IP address, and containers in a pod can see their pod's own IP address and it is accurate. This means that if a container within a pod dies and restarts, the IP address that other pods (or, more likely, services) use to address this pod is still correct.
- Pods don't need to be concerned about which node a pod they are accessing is running on. So long as the IP address they use is correct, Kubernetes' networking lets the communication occur.

In practice, application developers need to only be concerned coordinating which ports their containers use if there are multiple containers running in the same pod. Beyond that, prefer using services over direct pod access, since it loosely couples pods and lets the service perform automatic load balancing of requests.

Network policies

By default, every pod can communicate directly with every other pod via an IP address. This is one of the major benefits of Kubernetes' networking model. However, you might want to restrict how groups of pods can communicate with each other. Network Policies let you group pods

together using labels, and then define rules around what traffic is allowed between these pods.

Network policies define rules around *ingress*, incoming connections to the group of pods, and *egress*, outgoing connections made by pods in the group. These policies are able to restrict network connections to specific IP ranges and port numbers.

In practice, network policies are set up and used by your cluster administrator, leaving you to focus on your application.

Debugging, monitoring, and logging

Now that we have all the tools to create production deployment environments in Kubernetes, the final piece is to keep track of your running applications.

Debugging

Debugging your Kubernetes object configuration relies heavily on using a few key commands combined with the experience of having seen things break in a similar way in the past. The benefit of Kubernetes—compared to ad-hoc deployment systems—is that it breaks in a predictable way. Once you've seen a certain type of failure and you know how to fix it, you can expect that resolution to work in other situations.

`kubectl get all` is an easy way to get a complete picture of what's running in your namespace. While details are scant, it's useful to get a quick idea of what's running, what's failed, and what's about to start running.

The single most useful command when debugging Kubernetes is `kubectl describe`. It describes the entire state of an object, its desired state, and recent events that have affected the object. `kubectl describe all` gives you a detailed description of everything in your cluster, and `kubectl describe pod <pod>` helps you narrow that down to a single pod.

`kubectl logs` and `kubectl logs --previous` get you the output of a given pod, and the output of a previously-run pod respectively.

Finally, given that a pod is running, `kubectl exec -it <pod> sh` gets you shell access inside that pod. From there, you'll often find yourself `curl`ing localhost, other containers, other pods and other services to check that they're running.

Pods

If the pod isn't running…

- Run `kubectl describe pod <pod>` and `kubectl get pods`. Check the pod's status.
- `CrashLoopBackOff` means that the pod runs a container that immediately exits. This is commonly caused by a misconfiguration or invalid image.

- **`ErrImagePull`** means that the image could not be retrieved from the image repository. Check that the node has network access to the repository. Your pod can be configured with **`imagePullSecrets`** that provide authentication when it needs to access the registry
- Use **`kubectl logs <pod>`** and **`kubectl logs --previous <pod>`** to access the output of the pod that's failing to start
- Check that your pod's volumes and your containers' volumeMounts are correct.
- Check that you are using Secrets and ConfigMaps correctly, and they've been created in the cluster.
- Use **`kubectl get pods`**, then look at the restart count, the ready count, and check the liveness and readiness probes of each pod

If the pod runs but doesn't do what you want…
- Use **`kubectl exec -it <pod> sh`** to get an interactive shell inside the pod. From there, you can use ps aux plus any other Unix utilities you need to debug what's happening.

Services

If a service doesn't seem to work…
- Make sure you understand and are using the correct service type—the default is **`clusterIP`**, where the service is only exposed inside the cluster
- Use **`kubectl exec -it <pod> sh`** to get a shell in a pod inside the cluster, then try to curl one of the pod's directly. **`kubectl get pods -o wide`** gets you the IP address of all the pods in the cluster. Then try to curl the service. This helps you diagnose if the pod is misconfigured or if the service is misconfigured.

Monitoring

Monitoring for your Kubernetes cluster needs to be set up at the container, pod, service, node, and cluster level.

Heapster is a widely used tool for monitoring performance and resource usage of the cluster. If Heapster is running, the command kubectl top pod outputs resource usage for pods in the cluster. kubectl top node will display resource usage for each node.

Larger monitoring suites are also very useful in Kubernetes—you'll commonly see Prometheus and Google Cloud Monitoring used.

Logging

One defining property of resources in Kubernetes is that they are disposable—their storage and lifecycle is transient. Your application's logging needs to be handled separately to the node, pod, or container it runs on. How this is done depends on which third-party tools you use for managing your logs in Kubernetes. A common pattern is to set up fluentd for cluster-level log collection and aggregation, plus Elasticsearch and Kibana for log querying and analytics. Hosted providers like the Google Cloud Platform integrate logging directly using Stackdriver, which is a huge benefit if you don't want to have to set up logging and monitoring manually.

CKAD exam guide

I was one of the first people to take the Certified Kubernetes Application Developer exam and to receive certification. Here is my personal guide to the exam—it's an overview of what I studied, what surprised me, and what needs the most practice.

Please note that I can't give out specific details because of confidentiality agreements with the CNCF signed before taking the exam. I'll do my best to share my experience in a suitable way.

Also, feel free to email me at matt@matthewpalmer.net with any questions you have. And especially email me if you take the exam and this guide does or does not help! I want to collect some data on whether this guide is useful, and how I can improve it.

If you purchased the upgraded version of this book, you have access to the Github repo with a practice exam, code samples, and complete solutions.

Please contact me at matt@matthewpalmer.net if you haven't received access or if you would like to upgrade from the basic book to the upgraded package to gain access.

Background
The CKAD exam is a two hour practical exam that tests your knowledge, skill, and ability as a developer using Kubernetes. The exam is oriented towards testing this ability. Questions focus on using, debugging, monitoring, and accessing Kubernetes.

Exam style
The test involves interacting with a live Kubernetes cluster via the command line. This occurs in a web-based command-line environment where you need to execute kubectl commands, operate a Unix shell, and edit YAML files.

As you'd expect, the exam follows the syllabus and curriculum closely. There are roughly twenty topics in the curriculum, and roughly that many questions in the exam. The exam covers almost every topic in the syllabus—make sure you're familiar with everything when preparing.

The questions are all practical—you will need to create Kubernetes objects, consume existing objects, debug existing (but failing) objects, and query the cluster.

Assumed skills

First of all, you will need to be comfortable navigating a Unix command line environment. The entire test takes place in a web application that simulates a Unix command line. You will need to be able to navigate a Unix shell, know a command-line text editor, and use Unix programs to complete questions.

One large annoyance for me was that I don't usually edit my YAML files in Vim. This was a problem when I wanted to bulk-indent fifteen lines but couldn't remember how to do it. I had to type space-space fifteen times—a huge time-sink for very little value. Make sure you're fast at navigating Unix and editing YAML in a terminal text editor.

You will need to be able to create Kubernetes objects through both YAML and via kubectl commands!

Content

There were no content-related surprises to my exam—what's in the curriculum is what gets tested. If something seems like something that an administrator would set up on a cluster (for example, a resource quota), it's more likely that you'll need to consume or use that resource, rather than needing to know how to create it yourself. Remember that this is an exam for application developers.

I think there are three priority tiers of content to learn for the exam. The first tier requires deep understanding and experience, and will be tested multiple times. The second tier you need to understand and use, but it is likely to appear fewer times. The third tier are simpler concepts that you are likely to simply use or consume, and they only appear once in the exam.

Prioritise your learning according to these tiers.
1. Debugging Kubernetes, Pods, Deployments, Jobs, Services, editing YAML, Volumes and PVCs, namespaces, labels
2. Liveness and readiness, logging, monitoring, multi-container pods, rolling updates and rollbacks

3. Service Accounts, resource quotas, security contexts

Sample questions

I can't give out any specific questions, but to give you a sense of what the questions are like, here are a couple of sample questions. More sample questions and complete solutions are available in the Github repo for the upgraded package.

Note that these sample questions are easier than what's in the exam, but they give you a hint of the style of the questions.

Sample question 1

Create a deployment of version 1.7.1 of the nginx image. This deployment should have five pods running the nginx image. Create the deployment in the question-four namespace. Save the YAML configuration to create this deployment in a file called deploy.yaml.

Sample question 2

A pod called question-five-pod is failing to run in our Kubernetes cluster in the question-five namespace. Query the Kubernetes cluster to determine why the pod isn't starting, and save any relevant events to a file called query-results.txt. Then, fix the pod's configuration so that it starts correctly.

Other advice

Use the practice exam

Purchasers of the upgraded package of this book have access to a Github repo with a practice exam. Make sure you attempt this exam in a realistic environment: use only the command line, kubectl, and kubernetes.io, complete it under a time limit. Afterwards, check your solutions and re-do anything you weren't sure about.

After completing this book, everyone I talk to has a great understanding of Kubernetes concepts. But the most common reason people fail the exam is time constraints. The second is lack of familiarity with the environment. Both of these are easily solved through practice!

Know how to quickly navigate the web and CLI documentation

You have full access to kubernetes.io and kubectl in the exam environment. Make the most of it!

Make sure you can navigate, search, and find examples on kubernetes.io. Most questions require knowledge or configuration that's contained on kubernetes.io—knowing the layout and location of the documentation is a huge advantage.

While navigating the docs is important, the faster (and my preferred) way is to use kubectl's inbuilt documentation to quickly find information you need.

For figuring out which YAML field to use, `kubectl explain <object>.<subobject>` and `kubectl explain <object>.<subobject> --recursive` are my most-used commands. They'll print out types and attributes you need to set on a resource.

Know how to do both YAML and kubectl commands
The exam will require you to interchangeably use YAML or direct kubectl commands—make sure you are comfortable with both.

`kubectl get $resource --export=true -o yaml` will get you the configuration for an existing object in YAML format. This is great for debugging why something might not be working. It's also really useful if it's faster (or you are more comfortable) creating a resource via a kubectl command.

For example, I used something like `kubectl run mydeployment --image=nginx --port=8080 --replicas=5` to create a deployment quickly, and then did `kubectl get deployment mydeployment --export=true -o yaml` to get the YAML for that deployment. This helps me avoid remembering all the details of writing a deployment YAML.

Most things need to happen in a specific namespace
Each question will probably specify which namespace the resource lives in or is to be created in. Namespaces were something I wasn't particularly familiar with when I took the exam, so I kind of had to wing it. Make sure you're comfortable using and understanding namespaces, and can specify them through the command line (the `--namespace` argument) and through YAML (the `metadata.namespace` field).

Be able to move quickly

I found the exam fairly tight time-wise. You'll need to work quickly in a Kubernetes cluster from a terminal environment. Also, be prepared to skip questions if they have a low weighting—sometimes it's not worth spending ten minutes on a two mark question.

The exam environment is not friendly
First of all, the person running your exam will be strict. Even though it's all done remotely, you'll have to remove labels from drink bottles, remove everything from your desk (including, in my case, desk lamps and plants), and show the proctor the entire room through your webcam. Taking the exam in a public setting (like a library, which I almost did) is not really viable unfortunately.

The terminal environment you have to use is pretty terrible. It can be quite laggy and doesn't have the autocompletion functionality you might be used to. Try to find the best internet connection you can to take the exam —a laggy environment is really disruptive.

You have a free retake
Don't worry if you fail the exam—you get a free retake! The next time you'll better understand what's coming in the exam, and you'll be able to manage your time better. If you do fail your first attempt, please email me and let me know which things tripped you up or what would have helped to know. I really want to improve this guide for everyone.

Practice exam

If you've purchased the upgraded book package, you have access to a Github repository containing a practice exam to help you pass your official Certified Kubernetes Application Developer exam.

This is a practice exam designed to help you prepare for the CKAD exam. It's a collection of sample questions around the topics in the CKAD exam curriculum.

I've talked to many people who have taken the exam, and the number one thing they struggle with is time. Following that, navigating the environment and using command-line editors is a pain point.

Most people—after completing the Golden Guide to Kubernetes Application Development book—tell me that they have a great

conceptual understanding of Kubernetes. They just lack practice using kubectl and editing YAML under pressure.

This practice exam is based around the publicly available CKAD exam curriculum. I cannot and won't share specific questions from the real exam, it's simply a set of questions I've put together based on talking to people who are taking the exam. I cannot and will not help you cheat.

This practice exam has thirty questions. You should aim to complete them all in three hours. The real exam only goes for two hours, and you should expect around twenty questions.

You might find this time limit restricting—this is intentional. Most people struggle with time constraints when they take the real exam. Do your best to work through each question and don't cheat—if you aren't moving fast enough, you want to find out now, not in the middle of your exam!

The solutions are not the only way to complete the question, just the way that I would have done it. I'd love for you to post more solutions in the Github repo!

Thirty questions. Three hours. Good luck.