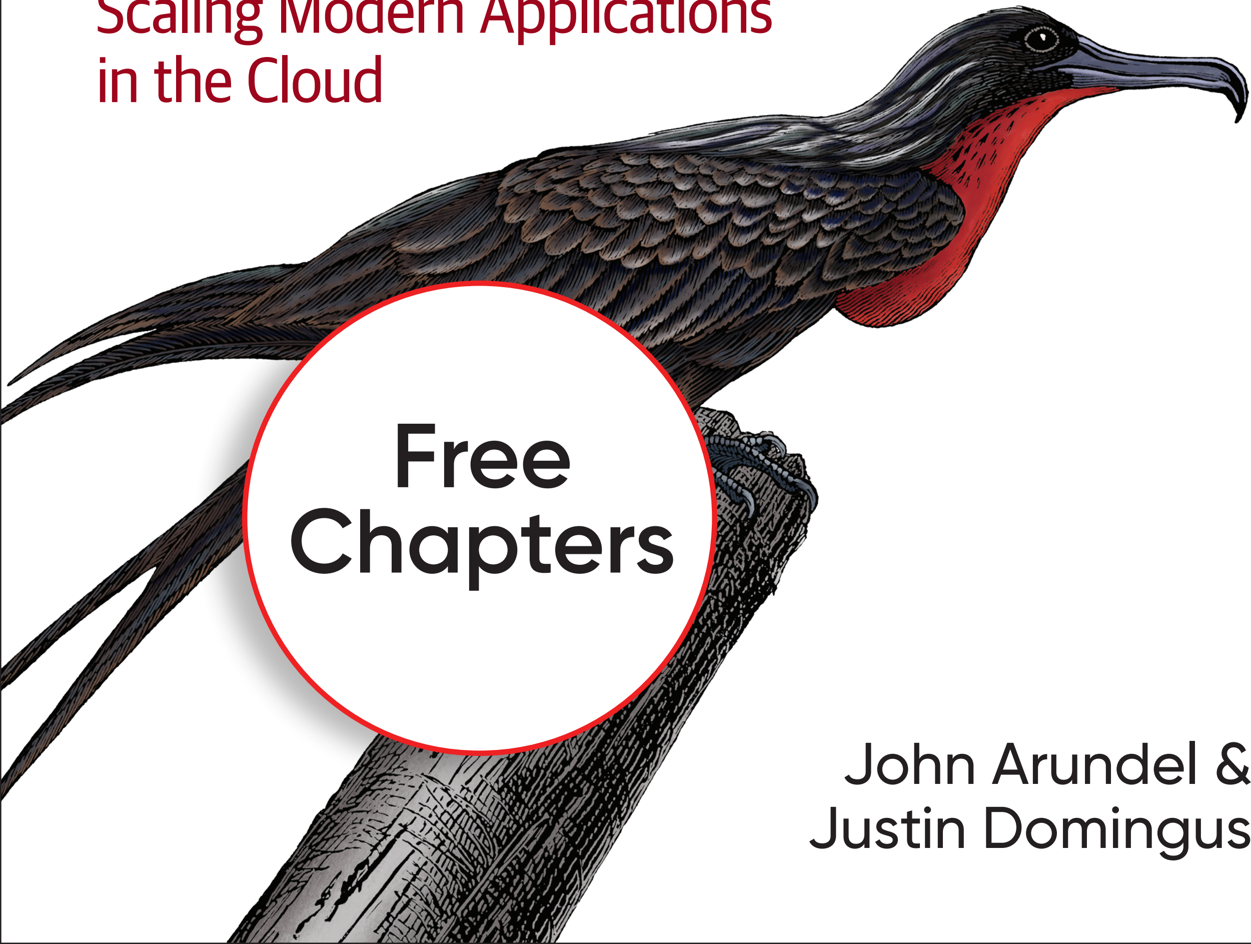
O'REILLY®

Compliments of NGINX

Cloud Native DevOps with Kubernetes

Building, Deploying, and Scaling Modern Applications in the Cloud



Free Chapters

John Arundel & Justin Domingus

Try NGINX Plus and NGINX WAF free for 30 days

Get high-performance application delivery for microservices. NGINX Plus is a software load balancer, web server, and content cache. The NGINX Web Application Firewall (WAF) protects applications against sophisticated Layer 7 attacks.

Cost Savings

Over 80% cost savings compared to hardware application delivery controllers and WAFs, with all the performance and features you expect.

Reduced Complexity

The only all-in-one load balancer, content cache, web server, and web application firewall helps reduce infrastructure sprawl.

Exclusive Features

JWT authentication, high availability, the NGINX Plus API, and other advanced functionality are only available in NGINX Plus.

NGINX WAF

A trial of the NGINX WAF, based on ModSecurity, is included when you download a trial of NGINX Plus.

Download at nginx.com/freetrial

NGINX

Cloud Native DevOps with Kubernetes

Building, Deploying, and Scaling Modern Applications in the Cloud

This excerpt contains Chapters 2, 3, 4, 8, and 9 of the book *Cloud Native DevOps with Kubernetes*. The complete book is available at oreilly.com and through other retailers.

John Arundel and Justin Domingus

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Cloud Native DevOps with Kubernetes
by John Arundel and Justin Domingus

Copyright © 2019 John Arundel and Justin Domingus. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Rachel Roumeliotis
Production Editor: Nan Barber
Proofreader: Amanda Kersey
Interior Designer: David Futato
Illustrator: Rebecca Demarest
Developmental Editors: Virginia Wilson and Nikki McDonald
Copyeditor: Kim Cofer
Indexer: Judith McConville
Cover Designer: Karen Montgomery

February 2019: First Edition

Revision History for the First Edition
2019-01-24: First Release
2019-03-08: Second Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492040767* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Cloud Native DevOps with Kubernetes*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and NGINX. See our statement of editorial independence (*http://www.oreilly.com/about/editorial_independence.html*).

978-1-492-04076-7

[LSI]

Table of Contents

Foreword by NGINX. ix

1. First Steps with Kubernetes. 1

Running Your First Container 1
  Installing Docker Desktop 2
  What Is Docker? 2
  Running a Container Image 2
The Demo Application 3
  Looking at the Source Code 3
  Introducing Go 4
  How the Demo App Works 4
Building a Container 5
  Understanding Dockerfiles 5
  Minimal Container Images 6
  Running docker image build 6
  Naming Your Images 7
  Port Forwarding 7
Container Registries 8
  Authenticating to the Registry 8
  Naming and Pushing Your Image 8
  Running Your Image 9
Hello, Kubernetes 9
  Running the Demo App 10
  If the Container Doesn't Start 10
Minikube 11
Summary 11

2. Getting Kubernetes. 13
Cluster Architecture 13
The Control Plane 14
Node Components 15
High Availability 15
The Costs of Self-Hosting Kubernetes 17
It's More Work Than You Think 17
It's Not Just About the Initial Setup 18
Tools Don't Do All the Work for You 19
Kubernetes Is Hard 19
Administration Overhead 19
Start with Managed Services 20
Managed Kubernetes Services 21
Google Kubernetes Engine (GKE) 21
Cluster Autoscaling 22
Amazon Elastic Container Service for Kubernetes (EKS) 22
Azure Kubernetes Service (AKS) 23
OpenShift 23
IBM Cloud Kubernetes Service 23
Heptio Kubernetes Subscription (HKS) 23
Turnkey Kubernetes Solutions 24
Stackpoint 24
Containership Kubernetes Engine (CKE) 24
Kubernetes Installers 24
kops 25
Kubespray 25
TK8 25
Kubernetes The Hard Way 25
kubeadm 26
Tarmak 26
Rancher Kubernetes Engine (RKE) 26
Puppet Kubernetes Module 26
Kubeformation 26
Buy or Build: Our Recommendations 27
Run Less Software 27
Use Managed Kubernetes if You Can 28
But What About Vendor Lock-in? 28
Use Standard Kubernetes Self-Hosting Tools if You Must 29
When Your Choices Are Limited 29
Bare-Metal and On-Prem 29
Clusterless Container Services 30
Amazon Fargate 30

Azure Container Instances (ACI) 31
Summary 31

3. Working with Kubernetes Objects. 33
Deployments 33
Supervising and Scheduling 34
Restarting Containers 34
Querying Deployments 35
Pods 35
ReplicaSets 36
Maintaining Desired State 37
The Kubernetes Scheduler 38
Resource Manifests in YAML Format 39
Resources Are Data 39
Deployment Manifests 39
Using kubectl apply 40
Service Resources 40
Querying the Cluster with kubectl 43
Taking Resources to the Next Level 44
Helm: A Kubernetes Package Manager 44
Installing Helm 45
Installing a Helm Chart 45
Charts, Repositories, and Releases 46
Listing Helm Releases 47
Summary 47

4. Running Containers. 49
Containers and Pods 49
What Is a Container? 50
What Belongs in a Container? 51
What Belongs in a Pod? 52
Container Manifests 52
Image Identifiers 53
The latest Tag 54
Container Digests 54
Base Image Tags 55
Ports 55
Resource Requests and Limits 55
Image Pull Policy 56
Environment Variables 56
Container Security 57
Running Containers as a Non-Root User 58

Blocking Root Containers 58
Setting a Read-Only Filesystem 59
Disabling Privilege Escalation 59
Capabilities 60
Pod Security Contexts 61
Pod Security Policies 61
Pod Service Accounts 62
Volumes 63
emptyDir Volumes 63
Persistent Volumes 64
Restart Policies 65
Image Pull Secrets 65
Summary 66

5. Managing Pods. 69
Labels 69
What Are Labels? 69
Selectors 70
More Advanced Selectors 71
Other Uses for Labels 71
Labels and Annotations 72
Node Affinities 73
Hard Affinities 74
Soft Affinities 74
Pod Affinities and Anti-Affinities 75
Keeping Pods Together 75
Keeping Pods Apart 76
Soft Anti-Affinities 77
When to Use Pod Affinities 77
Taints and Tolerations 77
Pod Controllers 79
DaemonSets 79
StatefulSets 80
Jobs 82
Cronjobs 83
Horizontal Pod Autoscalers 83
PodPresets 85
Operators and Custom Resource Definitions (CRDs) 86
Ingress Resources 87
Ingress Rules 87
Terminating TLS with Ingress 88
Ingress Controllers 89

Istio 90
Envoy 91
Summary 91

Foreword by NGINX

Digital transformation initiatives in enterprises today are driving innovation and agility as well as delivering superior customer experiences for global consumers. Embracing technologies and software development methodologies such as cloud computing, containers, and DevOps have been pivotal to digital transformation.

Containers offer a host of advantages—they are portable and resource efficient, provide effective isolation, and accelerate the development process. As you build new cloud-native applications using microservices or migrate existing ones to this new environment, building, deploying, and scaling these applications will present its own set of challenges. Service discovery and load balancing, storage, failure recovery, rollbacks, and configuration management all have to be performed at scale for objects that are usually ephemeral, sometimes across a multi-cloud environment. Creating a robust continuous integration and continuous delivery (CI/CD) process in order to realize agility and high feature velocity further adds to this complexity.

Kubernetes is the most pervasive container orchestration platform to address these challenges. It lets you manage complex application deployments quickly in a predictable and reliable manner.

This is a must-have book for both beginning and experienced users of Kubernetes. Authors John Arundel and Justin Domingus provide in-depth guidance in an incremental and logical manner, covering the entire lifecycle of managing a Kubernetes environment, including initial deployment, runtime operation, security, and ongoing maintenance and monitoring. As you'll see throughout this book, NGINX Open Source and our commercial-grade application delivery platform, NGINX Plus, are key components in successful production deployments with Kubernetes.

On the frontend, NGINX Open Source and NGINX Plus act as a stable entry point to your Kubernetes services—a persistent and reliable connection for external clients. NGINX Open Source provides foundational routing and load balancing capabilities to manage traffic across your containers. NGINX Plus, built on top of NGINX Open Source, provides advanced capabilities. It integrates well with service discovery platforms, achieves session persistence for stateful applications, authenticates APIs using JSON Web Token (JWT), and can be reconfigured dynamically when the set of service instances changes.

As you move to a Kubernetes infrastructure, NGINX Plus helps you achieve enterprise-grade traffic management for your containerized applications. We sincerely hope you enjoy this book as it helps you to succeed with your production deployments using Kubernetes.

— Karthik Krishnaswamy
Product Marketer, NGINX, Inc.
December 2018

CHAPTER 1

First Steps with Kubernetes

To do anything truly worth doing, I must not stand back shivering and thinking of the cold and danger, but jump in with gusto and scramble through as well as I can.

—Og Mandino

Enough with the theory; let's start working with Kubernetes and containers. In this chapter, you'll build a simple containerized application and deploy it to a local Kubernetes cluster running on your machine. In the process, you'll meet some very important cloud native technologies and concepts: Docker, Git, Go, container registries, and the `kubectl` tool.

This chapter is interactive! Often, throughout this book, we'll ask you to follow along with the examples by installing things on your own computer, typing commands, and running containers. We find that's a much more effective way to learn than just having things explained in words.

Running Your First Container

As we saw in ???, the container is one of the key concepts in cloud native development. The fundamental tool for building and running containers is Docker. In this section, we'll use the Docker Desktop tool to build a simple demo application, run it locally, and push the image to a container registry.

If you're already very familiar with containers, skip straight to "Hello, Kubernetes" on page 9, where the real fun starts. If you're curious to know what containers are and how they work, and to get a little practical experience with them before you start learning about Kubernetes, read on.

Installing Docker Desktop

Docker Desktop is a complete Kubernetes development environment for Mac or Windows that runs on your laptop (or desktop). It includes a single-node Kubernetes cluster that you can use to test your applications.

Let's install Docker Desktop now and use it to run a simple containerized application. If you already have Docker installed, skip this section and go straight on to "Running a Container Image" on page 2.

Download a version of the Docker Desktop Community Edition suitable for your computer, then follow the instructions for your platform to install Docker and start it up.

Docker Desktop isn't currently available for Linux, so Linux users will need to install Docker Engine instead, and then Minikube (see "Minikube" on page 11).

Once you've done that, you should be able to open a terminal and run the following command:

```
docker version
Client:
 Version:      18.03.1-ce
 ...
```

The exact output will be different depending on your platform, but if Docker is correctly installed and running, you'll see something like the example output shown. On Linux systems, you may need to run `sudo docker version` instead.

What Is Docker?

Docker is actually several different, but related things: a container image format, a *container runtime* library, which manages the life cycle of containers, a command-line tool for packaging and running containers, and an API for container management. The details needn't concern us here, since Kubernetes uses Docker as one of many components, though an important one.

Running a Container Image

What exactly is a container image? The technical details don't really matter for our purposes, but you can think of an image as being like a ZIP file. It's a single binary file that has a unique ID and holds everything needed to run the container.

Whether you're running the container directly with Docker, or on a Kubernetes cluster, all you need to specify is a container image ID or URL, and the system will take care of finding, downloading, unpacking, and starting the container for you.

We've written a little demo application that we'll use throughout the book to illustrate what we're talking about. You can download and run the application using a container image we prepared earlier. Run the following command to try it out:

```
docker container run -p 9999:8888 --name hello cloudnatived/demo:hello
```

Leave this command running, and point your browser to *http://localhost:9999/*.

You should see a friendly message:

```
Hello, 世界
```

Any time you make a request to this URL, our demo application will be ready and waiting to greet you.

Once you've had as much fun as you can stand, stop the container by pressing Ctrl-C in your terminal.

The Demo Application

So how does it work? Let's download the source code for the demo application that runs in this container and have a look.

You'll need Git installed for this part.[1] If you're not sure whether you already have Git, try the following command:

```
git version
git version 2.17.0
```

If you don't already have Git, follow the installation instructions for your platform.

Once you've installed Git, run this command:

```
git clone https://github.com/cloudnativedevops/demo.git
Cloning into demo...
...
```

Looking at the Source Code

This Git repository contains the demo application we'll be using throughout this book. To make it easier to see what's going on at each stage, the repo contains each successive version of the app in a different subdirectory. The first one is named simply *hello*. To look at the source code, run this command:

1 If you're not familiar with Git, read Scott Chacon and Ben Straub's excellent book *Pro Git* (Apress).

```
cd demo/hello
ls
Dockerfile  README.md
go.mod      main.go
```

Open the file *main.go* in your favorite editor (we recommend Visual Studio Code which has excellent support for Go, Docker, and Kubernetes development). You'll see this source code:

```
package main

import (
        "fmt"
        "log"
        "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, 世界")
}

func main() {
        http.HandleFunc("/", handler)
        log.Fatal(http.ListenAndServe(":8888", nil))
}
```

Introducing Go

Our demo application is written in the Go programming language.

Go is a modern programming language (developed at Google since 2009) that prioritizes simplicity, safety, and readability, and is designed for building large-scale concurrent applications, especially network services. It's also a lot of fun to program in.[2]

Kubernetes itself is written in Go, as are Docker, Terraform, and many other popular open source projects. This makes Go a good choice for developing cloud native applications.

How the Demo App Works

As you can see, the demo app is pretty simple, even though it implements an HTTP server (Go comes with a powerful standard library). The core of it is this function, called `handler`:

2 If you're a fairly experienced programmer, but new to Go, Alan Donovan and Brian Kernighan's *The Go Programming Language* (Addison-Wesley) is an invaluable guide.

```
func handler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, 世界")
}
```

As the name suggests, it handles HTTP requests. The request is passed in as an argument to the function (though the function doesn't do anything with it, yet).

An HTTP server also needs a way to send something back to the client. The `http.ResponseWriter` object enables our function to send a message back to the user to display in her browser: in this case, just the string `Hello, 世界`.

The first example program in any language traditionally prints `Hello, world`. But because Go natively supports Unicode (the international standard for text representation), example Go programs often print `Hello, 世界` instead, just to show off. If you don't happen to speak Chinese, that's okay: Go does!

The rest of the program takes care of registering the `handler` function as the handler for HTTP requests, and actually starting the HTTP server to listen and serve on port 8888.

That's the whole app! It doesn't do much yet, but we will add capabilities to it as we go on.

Building a Container

You know that a container image is a single file that contains everything the container needs to run, but how do you build an image in the first place? Well, to do that, you use the `docker image build` command, which takes as input a special text file called a *Dockerfile*. The Dockerfile specifies exactly what needs to go into the container image.

One of the key benefits of containers is the ability to build on existing images to create new images. For example, you could take a container image containing the complete Ubuntu operating system, add a single file to it, and the result will be a new image.

In general, a Dockerfile has instructions for taking a starting image (a so-called *base image*), transforming it in some way, and saving the result as a new image.

Understanding Dockerfiles

Let's see the Dockerfile for our demo application (it's in the *hello* subdirectory of the app repo):

```
FROM golang:1.11-alpine AS build

WORKDIR /src/
COPY main.go go.* /src/
```

```
RUN CGO_ENABLED=0 go build -o /bin/demo

FROM scratch
COPY --from=build /bin/demo /bin/demo
ENTRYPOINT ["/bin/demo"]
```

The exact details of how this works don't matter for now, but it uses a fairly standard build process for Go containers called *multi-stage builds*. The first stage starts from an official `golang` container image, which is just an operating system (in this case Alpine Linux) with the Go language environment installed. It runs the `go build` command to compile the *main.go* file we saw earlier.

The result of this is an executable binary file named *demo*. The second stage takes a completely empty container image (called a *scratch* image, as in *from scratch*) and copies the *demo* binary into it.

Minimal Container Images

Why the second build stage? Well, the Go language environment, and the rest of Alpine Linux, is really only needed in order to *build* the program. To run the program, all it takes is the *demo* binary, so the Dockerfile creates a new scratch container to put it in. The resulting image is very small (about 6 MiB)—and that's the image that can be deployed in production.

Without the second stage, you would have ended up with a container image about 350 MiB in size, 98% of which is unnecessary and will never be executed. The smaller the container image, the faster it can be uploaded and downloaded, and the faster it will be to start up.

Minimal containers also have a reduced *attack surface* for security issues. The fewer programs there are in your container, the fewer potential vulnerabilities.

Because Go is a compiled language that can produce self-contained executables, it's ideal for writing minimal (*scratch*) containers. By comparison, the official Ruby container image is 1.5 GiB; about 250 times bigger than our Go image, and that's before you've added your Ruby program!

Running docker image build

We've seen that the Dockerfile contains instructions for the `docker image build` tool to turn our Go source code into an executable container. Let's go ahead and try it. In the *hello* directory, run the following command:

```
docker image build -t myhello .
Sending build context to Docker daemon  4.096kB
Step 1/7 : FROM golang:1.11-alpine AS build
...
```

```
Successfully built eeb7d1c2e2b7
Successfully tagged myhello:latest
```

Congratulations, you just built your first container! You can see from the output that Docker performs each of the actions in the Dockerfile in sequence on the newly formed container, resulting in an image that's ready to use.

Naming Your Images

When you build an image, by default it just gets a hexadecimal ID, which you can use to refer to it later (for example, to run it). These IDs aren't particularly memorable or easy to type, so Docker allows you to give the image a human-readable name, using the `-t` switch to `docker image build`. In the previous example you named the image `myhello`, so you should be able to use that name to run the image now.

Let's see if it works:

```
docker container run -p 9999:8888 myhello
```

You're now running your own copy of the demo application, and you can check it by browsing to the same URL as before (*http://localhost:9999/*).

You should see `Hello, 世界`. When you're done running this image, press Ctrl-C to stop the `docker container run` command.

Exercise

If you're feeling adventurous, modify the *main.go* file in the demo application and change the greeting so that it says "Hello, world" in your favorite language (or change it to say whatever you like). Rebuild the container and run it to check that it works.

Congratulations, you're now a Go programmer! But don't stop there: take the interactive Tour of Go to learn more.

Port Forwarding

Programs running in a container are isolated from other programs running on the same machine, which means they can't have direct access to resources like network ports.

The demo application listens for connections on port 8888, but this is the *container's* own private port 8888, not a port on your computer. In order to connect to the container's port 8888, you need to *forward* a port on your local machine to that port on the container. It could be any port, including 8888, but we'll use 9999 instead, to make it clear which is your port, and which is the container's.

To tell Docker to forward a port, you can use the -p switch, just as you did earlier in "Running a Container Image" on page 2:

```
docker container run -p HOST_PORT:CONTAINER_PORT ...
```

Once the container is running, any requests to HOST_PORT on the local computer will be forwarded automatically to CONTAINER_PORT on the container, which is how you're able to connect to the app with your browser.

Container Registries

In "Running a Container Image" on page 2, you were able to run an image just by giving its name, and Docker downloaded it for you automatically.

You might reasonably wonder where it's downloaded from. While you can use Docker perfectly well by just building and running local images, it's much more useful if you can push and pull images from a *container registry*. The registry allows you to store images and retrieve them using a unique name (like cloudnatived/demo:hello).

The default registry for the docker container run command is Docker Hub, but you can specify a different one, or set up your own.

For now, let's stick with Docker Hub. While you can download and use any public container image from Docker Hub, to push your own images you'll need an account (called a *Docker ID*). Follow the instructions at *https://hub.docker.com/* to create your Docker ID.

Authenticating to the Registry

Once you've got your Docker ID, the next step is to connect your local Docker daemon with Docker Hub, using your ID and password:

```
docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't
have a Docker ID, head over to https://hub.docker.com to create one.
Username: YOUR_DOCKER_ID
Password: YOUR_DOCKER_PASSWORD
Login Succeeded
```

Naming and Pushing Your Image

In order to be able to push a local image to the registry, you need to name it using this format: YOUR_DOCKER_ID/myhello.

To create this name, you don't need to rebuild the image; instead, run this command:

```
docker image tag myhello YOUR_DOCKER_ID/myhello
```

This is so that when you push the image to the registry, Docker knows which account to store it in.

Go ahead and push the image to Docker Hub, using this command:

```
docker image push YOUR_DOCKER_ID/myhello
The push refers to repository [docker.io/YOUR_DOCKER_ID/myhello]
b2c591f16c33: Pushed
latest: digest:
        sha256:7ac57776e2df70d62d7285124fbff039c9152d1bdfb36c75b5933057cefe4fc7
size: 528
```

Running Your Image

Congratulations! Your container image is now available to run anywhere (at least, anywhere with access to the internet), using the command:

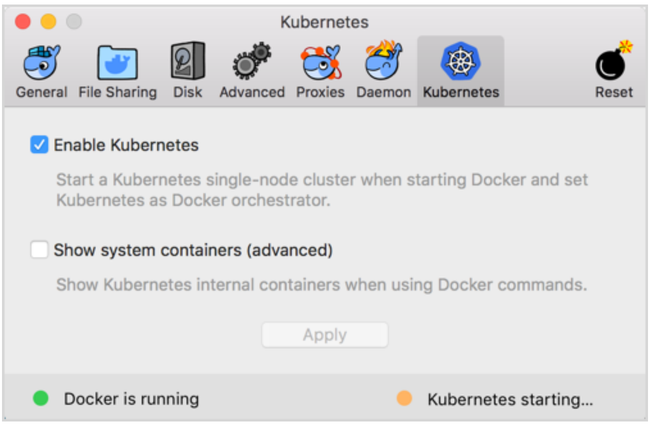
```
docker container run -p 9999:8888 YOUR_DOCKER_ID/myhello
```

Hello, Kubernetes

Now that you've built and pushed your first container image, you can run it using the `docker container run` command, but that's not very exciting. Let's do something a little more adventurous and run it in Kubernetes.

There are lots of ways to get a Kubernetes cluster, and we'll explore some of them in more detail in Chapter 2. If you already have access to a Kubernetes cluster, that's great, and if you like you can use it for the rest of the examples in this chapter.

If not, don't worry. Docker Desktop includes Kubernetes support (Linux users, see "Minikube" on page 11 instead). To enable it, open the Docker Desktop Preferences, select the Kubernetes tab, and check Enable (see Figure 1-1).



[✓] Enable Kubernetes

Start a Kubernetes single-node cluster when starting Docker and set Kubernetes as Docker orchestrator.

[ ] Show system containers (advanced)

Show Kubernetes internal containers when using Docker commands.

● Docker is running ● Kubernetes starting...

Figure 1-1. Enabling Kubernetes support in Docker Desktop

It will take a few minutes to install and start Kubernetes. Once that's done, you're ready to run the demo app!

Running the Demo App

Let's start by running the demo image you built earlier. Open a terminal and run the `kubectl` command with the following arguments:

```
kubectl run demo --image=YOUR_DOCKER_ID/myhello --port=9999 --labels app=demo
deployment.apps "demo" created
```

Don't worry about the details of this command for now: it's basically the Kubernetes equivalent of the `docker container run` command you used earlier in this chapter to run the demo image. If you haven't built your own image yet, you can use ours: `--image=cloudnatived/demo:hello`.

Recall that you needed to forward port 9999 on your local machine to the container's port 8888 in order to connect to it with your web browser. You'll need to do the same thing here, using `kubectl port-forward`:

```
kubectl port-forward deploy/demo 9999:8888
Forwarding from 127.0.0.1:9999 -> 8888
Forwarding from [::1]:9999 -> 8888
```

Leave this command running and open a new terminal to carry on.

Connect to *http://localhost:9999/* with your browser to see the `Hello, 世界` message.

It may take a few seconds for the container to start and for the app to be available. If it isn't ready after half a minute or so, try this command:

```
kubectl get pods --selector app=demo
NAME                     READY     STATUS    RESTARTS   AGE
demo-54df94b7b7-qgtc6    1/1       Running   0          9m
```

When the container is running and you connect to it with your browser, you'll see this message in the terminal:

```
Handling connection for 9999
```

If the Container Doesn't Start

If the `STATUS` is not shown as `Running`, there may be a problem. For example, if the status is `ErrImagePull` or `ImagePullBackoff`, it means Kubernetes wasn't able to find and download the image you specified. You may have made a typo in the image name; check your `kubectl run` command.

If the status is `ContainerCreating`, then all is well; Kubernetes is still downloading and starting the image. Just wait a few seconds and check again.

Minikube

If you don't want to use, or can't use, the Kubernetes support in Docker Desktop, there is an alternative: the well-loved Minikube. Like Docker Desktop, Minikube provides a single-node Kubernetes cluster that runs on your own machine (in fact, in a virtual machine, but that doesn't matter).

To install Minikube, follow these Minikube installation instructions.

Summary

If, like us, you quickly grow impatient with wordy essays about why Kubernetes is so great, we hope you enjoyed getting to grips with some practical tasks in this chapter. If you're an experienced Docker or Kubernetes user already, perhaps you'll forgive the refresher course. We want to make sure that everybody feels quite comfortable with building and running containers in a basic way, and that you have a Kubernetes environment you can play and experiment with, before getting on to more advanced things.

Here's what you should take away from this chapter:

- All the source code examples (and many more) are available in the demo repository that accompanies this book.
- The Docker tool lets you build containers locally, push them to or pull them from a container registry such as Docker Hub, and run container images locally on your machine.
- A container image is completely specified by a Dockerfile: a text file that contains instructions about how to build the container.
- Docker Desktop lets you run a small (single-node) Kubernetes cluster on your machine, which is nonetheless capable of running any containerized application. Minikube is another option.
- The `kubectl` tool is the primary way of interacting with a Kubernetes cluster, and can be used either *imperatively* (to run a public container image, for example, and implicitly creating the necessary Kubernetes resources), or *declaratively*, to apply Kubernetes configuration in the form of YAML manifests.

CHAPTER 2

Getting Kubernetes

Perplexity is the beginning of knowledge.

—Kahlil Gibran

Kubernetes is the operating system of the cloud native world, providing a reliable and scalable platform for running containerized workloads. But how should you run Kubernetes? Should you host it yourself? On cloud instances? On bare-metal servers? Or should you use a managed Kubernetes service? Or a managed platform that's based on Kubernetes, but extends it with workflow tools, dashboards, and web interfaces?

That's a lot of questions for one chapter to answer, but we'll try.

It's worth noting that we won't be particularly concerned here with the technical details of operating Kubernetes itself, such as building, tuning, and troubleshooting clusters. There are many excellent resources to help you with that, of which we particularly recommend Kubernetes cofounder Brendan Burns's book *Managing Kubernetes: Operating Kubernetes Clusters in the Real World* (O'Reilly).

Instead, we'll focus on helping you understand the basic architecture of a cluster, and give you the information you need to decide how to run Kubernetes. We'll outline the pros and cons of managed services, and look at some of the popular vendors.

If you want to run your own Kubernetes cluster, we list some of the best installation tools available to help you set up and manage clusters.

Cluster Architecture

You know that Kubernetes connects multiple servers into a *cluster*, but what is a cluster, and how does it work? The technical details don't matter for the purposes of this book, but you should understand the basic components of Kubernetes and how they fit together, in order to understand what your options are when it comes to building or buying Kubernetes clusters.

The Control Plane

The cluster's brain is called the *control plane*, and it runs all the tasks required for Kubernetes to do its job: scheduling containers, managing Services, serving API requests, and so on (see Figure 2-1).

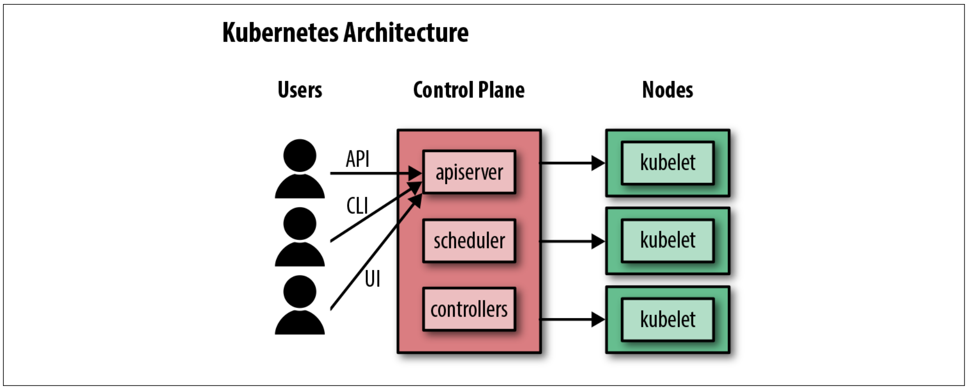


Figure 2-1. How a Kubernetes cluster works

The control plane is actually made up of several components:

`kube-apiserver`
: This is the frontend server for the control plane, handling API requests.

`etcd`
: This is the database where Kubernetes stores all its information: what nodes exist, what resources exist on the cluster, and so on.

`kube-scheduler`
: This decides where to run newly created Pods.

`kube-controller-manager`
: This is responsible for running resource controllers, such as Deployments.

`cloud-controller-manager`
: This interacts with the cloud provider (in cloud-based clusters), managing resources such as load balancers and disk volumes.

The members of the cluster which run the control plane components are called *master nodes*.

Node Components

Cluster members that run user workloads are called *worker nodes* (Figure 2-2).

Each worker node in a Kubernetes cluster runs these components:

`kubelet`
: This is responsible for driving the container runtime to start workloads that are scheduled on the node, and monitoring their status.

`kube-proxy`
: This does the networking magic that routes requests between Pods on different nodes, and between Pods and the internet.

Container runtime
: This actually starts and stops containers and handles their communications. Usually Docker, but Kubernetes supports other container runtimes, such as `rkt` and CRI-O.

Other than running different software components, there's no intrinsic difference between master nodes and worker nodes. Master nodes don't usually run user workloads, though, except in very small clusters (like Docker Desktop or Minikube).

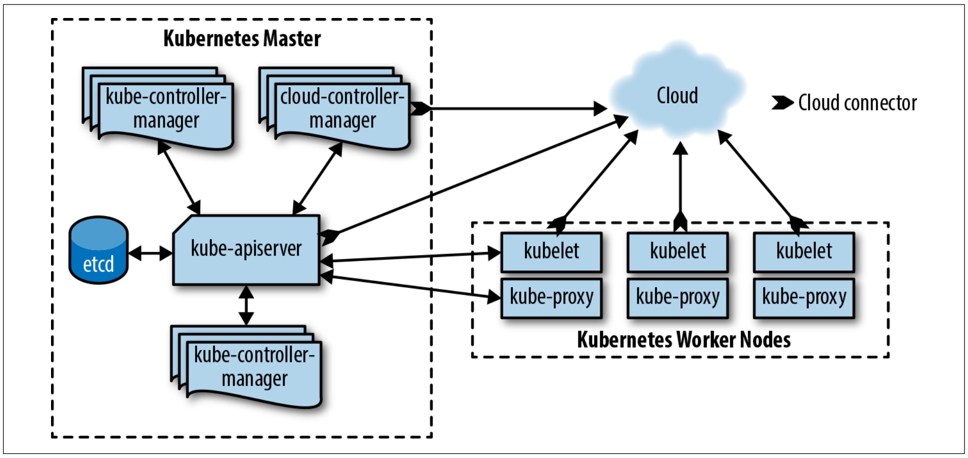


Figure 2-2. How the Kubernetes components fit together

High Availability

A correctly configured Kubernetes control plane has multiple master nodes, making it *highly available*; that is, if any individual master node fails or is shut down, or one of the control plane components on it stops running, the cluster will still work properly. A highly available control plane will also handle the situation where the master nodes are working properly, but some of them cannot communicate with the others, due to a network failure (known as a *network partition*).

The `etcd` database is replicated across multiple nodes, and can survive the failure of individual nodes, so long as a quorum of over half the original number of `etcd` replicas is still available.

If all of this is configured correctly, the control plane can survive a reboot or temporary failure of individual master nodes.

Control plane failure

A damaged control plane doesn't necessarily mean that your applications will go down, although it might well cause strange and errant behavior.

For example, if you were to stop all the master nodes in your cluster, the Pods on the worker nodes would keep on running—at least for a while. But you would be unable to deploy any new containers or change any Kubernetes resources, and controllers such as Deployments would stop working.

Therefore, high availability of the control plane is critical to a properly functioning cluster. You need to have enough master nodes available that the cluster can maintain a *quorum* even if one fails; for production clusters, the workable minimum is three (see ???).

Worker node failure

By contrast, the failure of any worker node doesn't really matter. Kubernetes will detect the failure and reschedule the node's Pods somewhere else, so long as the control plane is still working.

If a large number of nodes fail at once, this might mean that the cluster no longer has enough resources to run all the workloads you need. Fortunately, this doesn't happen often, and even if it does, Kubernetes will keep as many of your Pods running as it can while you replace the missing nodes.

It's worth bearing in mind, though, that the fewer worker nodes you have, the greater the proportion of the cluster's capacity each one represents. You should assume that a single node failure will happen at any time, especially in the cloud, and two simultaneous failures are not unheard of.

A rare, but entirely possible, kind of failure is losing a whole cloud *availability zone*. Cloud vendors like AWS and Google Cloud provide multiple availability zones in each region, each corresponding roughly to a single data center. For this reason, rather than having all your worker nodes in the same zone, it's a good idea to distribute them across two or even three zones.

Trust, but verify

Although high availability should enable your cluster to survive losing one master, or a few worker nodes, it's always wise to *actually test* this. During a scheduled maintenance window, or outside of peak hours, try rebooting a worker and see what happens. (Hopefully, nothing, or nothing that's visible to users of your applications.)

For a more demanding test, reboot one of the master nodes. (Managed services such as Google Kubernetes Engine, which we'll discuss later in the chapter, don't allow you to do this, for obvious reasons.) Still, a production-grade cluster should survive this with no problems whatsoever.

The Costs of Self-Hosting Kubernetes

The most important decision facing anyone who's considering running production workloads in Kubernetes is *buy or build?* Should you run your own clusters, or pay someone else to run them? Let's look at some of the options.

The most basic choice of all is self-hosted Kubernetes. By *self-hosted* we mean that you, personally, or a team in your organization, install and configure Kubernetes, on machines that you own or control, just as you might do with any other software that you use, such as Redis, PostgreSQL, or Nginx.

This is the option that gives you the maximum flexibility and control. You can decide what versions of Kubernetes to run, what options and features are enabled, when and whether to upgrade clusters, and so on. But there are some significant downsides, as we'll see in the next section.

It's More Work Than You Think

The self-hosted option also requires the maximum resources, in terms of people, skills, engineering time, maintenance, and troubleshooting. Just setting up a working Kubernetes cluster is pretty simple, but that's a long way from a cluster that's ready for production. You need to consider at least the following questions:

- Is the control plane highly available? That is, if a master node goes down or becomes unresponsive, does your cluster still work? Can you still deploy or update apps? Will your running applications still be fault-tolerant without the control plane? (See "High Availability" on page 15.)
- Is your pool of worker nodes highly available? That is, if an outage should take down several worker nodes, or even a whole cloud availability zone, will your workloads stop running? Will your cluster keep working? Will it be able to automatically provision new nodes to heal itself, or will it require manual intervention?

- Is your cluster set up *securely*? Do its internal components communicate using TLS encryption and trusted certificates? Do users and applications have minimal rights and permissions for cluster operations? Are container security defaults set properly? Do nodes have unnecessary access to control plane components? Is access to the underlying `etcd` database properly controlled and authenticated?
- Are all services in your cluster secure? If they're accessible from the internet, are they properly authenticated and authorized? Is access to the cluster API strictly limited?
- Is your cluster *conformant*? Does it meet the standards for Kubernetes clusters defined by the Cloud Native Computing Foundation? (See ??? for details.)
- Are your cluster nodes fully *config-managed*, rather than being set up by imperative shell scripts and then left alone? The operating system and kernel on each node needs to be updated, have security patches applied, and so on.
- Is the data in your cluster properly backed up, including any persistent storage? What is your restore process? How often do you test restores?
- Once you have a working cluster, how do you maintain it over time? How do you provision new nodes? Roll out config changes to existing nodes? Roll out Kubernetes updates? Scale in response to demand? Enforce policies?

Distributed systems engineer and writer Cindy Sridharan has estimated that it takes around a million dollars in engineer salary to get Kubernetes up and running in a production configuration from scratch ("And you still might not get there"). That figure should give any technical leader food for thought when considering self-hosted Kubernetes.

It's Not Just About the Initial Setup

Now bear in mind that you need to pay attention to these factors not just when setting up the first cluster for the first time, but for all your clusters for all time. When you make changes or upgrades to your Kubernetes infrastructure, you need to consider the impact on high availability, security, and so on.

You'll need to have monitoring in place to make sure the cluster nodes and all the Kubernetes components are working properly, and an alerting system so that staff can be paged to deal with any problems, day or night.

Kubernetes is still in rapid development, and new features and updates are being released all the time. You'll need to keep your cluster up to date with those, and understand how the changes affect your existing setup. You may need to reprovision your cluster to get the full benefit of the latest Kubernetes functionality.

It's also not enough to read a few books or articles, configure the cluster the right way, and leave it at that. You need to test and verify the configuration on a regular basis—by killing a master node and making sure everything still works, for example.

Automated resilience testing tools such as Netflix's Chaos Monkey can help with this, by randomly killing nodes, Pods, or network connections every so often. Depending on the reliability of your cloud provider, you may find that Chaos Monkey is unnecessary, as regular real-world failures will also test the resilience of your cluster and the services running on it (see ???).

Tools Don't Do All the Work for You

There are tools—lots and lots of tools—to help you set up and configure Kubernetes clusters, and many of them advertise themselves as being more or less point-and-click, zero-effort, instant solutions. The sad fact is that in our opinion, the large majority of these tools solve only the easy problems, and ignore the hard ones.

On the other hand, powerful, flexible, enterprise-grade commercial tools tend to be very expensive, or not even available to the public, since there's more money to be made selling a managed service than there is selling a general-purpose cluster management tool.

Kubernetes Is Hard

Despite the widespread notion that it's simple to set up and manage, the truth is that *Kubernetes is hard*. Considering what it does, it's remarkably simple and well-designed, but it has to deal with very complex situations, and that leads to complex software.

Make no mistake, there is a significant investment of time and energy involved in both learning how to manage your own clusters properly, and actually doing it from day to day, month to month. We don't want to discourage you from using Kubernetes, but we want you to have a clear understanding of what's involved in running Kubernetes yourself. This will help you to make an informed decision about the costs and benefits of self-hosting, as opposed to using managed services.

Administration Overhead

If your organization is large, with resources to spare for a dedicated Kubernetes cluster operations team, this may not be such a big problem. But for small to medium enterprises, or even startups with a handful of engineers, the administration overhead of running your own Kubernetes clusters may be prohibitive.

Given a limited budget and number of staff available for IT operations, what proportion of your resources do you want to spend on administering Kubernetes itself? Would those resources be better used to support your business's workloads instead? Can you operate Kubernetes more cost-effectively with your own staff, or by using a managed service?

Start with Managed Services

You might be a little surprised that, in a Kubernetes book, we recommend that you don't run Kubernetes! At least, don't run it yourself. For the reasons we've outlined in the previous sections, we think that using managed services is likely to be far more cost-effective than self-hosting Kubernetes clusters. Unless you want to do something strange and experimental with Kubernetes that isn't supported by any managed provider, there are basically no good reasons to go the self-hosted route.

In our experience, and that of many of the people we interviewed for this book, a managed service is the best way to run Kubernetes, period.

If you're considering whether Kubernetes is even an option for you, using a managed service is a great way to try it out. You can get a fully working, secure, highly available, production-grade cluster in a few minutes, for a few dollars a day. (Most cloud providers even offer a free tier that lets you run a Kubernetes cluster for weeks or months without incurring any charges.) Even if you decide, after a trial period, that you'd prefer to run your own Kubernetes cluster, the managed services will show you how it should be done.

On the other hand, if you've already experimented with setting up Kubernetes yourself, you'll be delighted with how much easier managed services make the process. You probably didn't build your own house; why build your own cluster, when it's cheaper and quicker to have someone else do it, and the results are better?

In the next section, we'll outline some of the most popular managed Kubernetes services, tell you what we think of them, and recommend our favorite. If you're still not convinced, the second half of the chapter will explore Kubernetes installers you can use to build your own clusters (see "Kubernetes Installers" on page 24).

We should say at this point that neither of the authors is affiliated with any cloud provider or commercial Kubernetes vendor. Nobody's paying us to recommend their product or service. The opinions here are our own, based on personal experience, and the views of hundreds of Kubernetes users we spoke to while writing this book.

Naturally, things move quickly in the Kubernetes world, and the managed services marketplace is especially competitive. Expect the features and services described here to change rapidly. The list presented here is not complete, but we've tried to include the services we feel are the best, the most widely used, or otherwise important.

Managed Kubernetes Services

Managed Kubernetes services relieve you of almost all the administration overhead of setting up and running Kubernetes clusters, particularly the control plane. Effectively, a managed service means you pay for someone else (such as Google) to run the cluster for you.

Google Kubernetes Engine (GKE)

As you'd expect from the originators of Kubernetes, Google offers a fully managed Kubernetes service that is completely integrated with the Google Cloud Platform. Just choose the number of worker nodes, and click a button in the GCP web console to create a cluster, or use the Deployment Manager tool to provision one. Within a few minutes, your cluster will be ready to use.

Google takes care of monitoring and replacing failed nodes, auto-applying security patches, and high availability for the control plane and `etcd`. You can set your nodes to auto-upgrade to the latest version of Kubernetes, during a maintenance window of your choice.

High availability

GKE gives you a production-grade, highly available Kubernetes cluster with none of the setup and maintenance overhead associated with self-hosted infrastructure. Everything is controllable via the Google Cloud API, using Deployment Manager,[1] Terraform, or other tools, or you can use the GCP web console. Naturally, GKE is fully integrated with all the other services in Google Cloud.

For extended high availability, you can create *multizone* clusters, which spread worker nodes across multiple failure zones (roughly equivalent to individual data centers). Your workloads will keep on running, even if a whole failure zone is affected by an outage.

Regional clusters take this idea even further, by distributing multiple master nodes across failure zones, as well as workers.

1 Deployment Manager is Google's command-line tool for managing cloud resources; not to be confused with Kubernetes Deployments.

Cluster Autoscaling

GKE also offers an attractive cluster autoscaling option (see ???). With autoscaling enabled, if there are pending workloads that are waiting for a node to become available, the system will add new nodes automatically to accommodate the demand.

Conversely, if there is spare capacity, the autoscaler will consolidate Pods onto a smaller number of nodes and remove the unused nodes. Since billing for GKE is based on the number of worker nodes, this helps you control costs.

GKE is best-of-breed

Google has been in the Kubernetes business longer than anybody else, and it shows. GKE is, in our opinion, the best managed Kubernetes service available. If you already have infrastructure in Google Cloud, it makes sense to use GKE to run Kubernetes. If you're already established on another cloud, it needn't stop you using GKE if you want to, but you should look first at managed options within your existing cloud provider.

If you haven't made a cloud provider decision yet, GKE is a persuasive argument in favor of choosing Google Cloud.

Amazon Elastic Container Service for Kubernetes (EKS)

Amazon has also been providing managed container cluster services for a long time, but until very recently the only option was Elastic Container Service (ECS), Amazon's proprietary technology.

While perfectly usable, ECS is not as powerful or flexible as Kubernetes, and evidently even Amazon has decided that the future is Kubernetes, with the launch of Elastic Container Service for Kubernetes (EKS). (Yes, *EKS* ought to stand for *Elastic Kubernetes Service*, but it doesn't.)

It's not quite as seamless an experience as Google Kubernetes Engine, so be prepared to do more of the setup work yourself. Also, unlike some competitors, EKS charges you for the master nodes as well as the other cluster infrastructure. This makes it more expensive, for a given cluster size, than either Google or Microsoft's managed Kubernetes service.

If you already have infrastructure in AWS, or run containerized workloads in the older ECS service that you want to move to Kubernetes, then EKS is a sensible choice. As the newest entry into the managed Kubernetes marketplace, though, it has some distance to go to catch up to the Google and Microsoft offerings.

Azure Kubernetes Service (AKS)

Although Microsoft came a little later to the cloud business than Amazon or Google, they're catching up fast. Azure Kubernetes Service (AKS) offers most of the features of its competitors, such as Google's GKE. You can create clusters from the web interface or using the Azure `az` command-line tool.

As with GKE and EKS, you have no access to the master nodes, which are managed internally, and your billing is based on the number of worker nodes in your cluster.

OpenShift

OpenShift is more than just a managed Kubernetes service: it's a full Platform-as-a-Service (PaaS) product, which aims to manage the whole software development life cycle, including continuous integration and build tools, test runner, application deployment, monitoring, and orchestration.

OpenShift can be deployed to bare-metal servers, virtual machines, private clouds, and public clouds, so you can create a single Kubernetes cluster that spans all these environments. This makes it a good choice for very large organizations, or those with very heterogeneous infrastructure.

IBM Cloud Kubernetes Service

Naturally, the venerable IBM is not to be left out in the field of managed Kubernetes services. IBM Cloud Kubernetes Service is pretty simple and straightforward, allowing you to set up a vanilla Kubernetes cluster in IBM Cloud.

You can access and manage your IBM Cloud cluster through the default Kubernetes CLI and the provided command-line tool, or a basic GUI. There are no real killer features that differentiate IBM's offering from the other major cloud providers, but it's a logical option if you're already using IBM Cloud.

Heptio Kubernetes Subscription (HKS)

For large enterprises that want the security and flexibility of running clusters across multiple public clouds, Heptio Kubernetes Subscription (HKS) aims to provide just that.

Heptio has a solid brand in the Kubernetes world: it's run by two of the cofounders of the Kubernetes project, Craig McLuckie and Joe Beda, and has produced many important open source tools, such as Velero (see ???), and Sonobuoy (see ???).

Turnkey Kubernetes Solutions

While managed Kubernetes services are a good fit for most business requirements, there may be some circumstances in which using managed services isn't an option. There is a growing class of *turnkey* offerings, which aim to give you a ready-to-use, production-grade Kubernetes cluster by just clicking a button in a web browser.

Turnkey Kubernetes solutions are attractive both to large enterprises (because they can have a commercial relationship with the vendor) and small companies with scarce engineering and operations resources. Here are a few of the options in the turnkey space.

Stackpoint

Stackpoint is advertised as "The simplest way to deploy a Kubernetes cluster to the public cloud," in as few as three clicks. There are various price points available, starting from $50 a month, and Stackpoint offers unlimited-node clusters, high availability for master nodes and for `etcd`, and support for *federated* clusters, which can span multiple clouds (see ???).

As a compromise between self-hosted Kubernetes and fully managed services, Stackpoint is an attractive option for companies that want to be able to provision and manage Kubernetes from a web portal, but still run worker nodes on their own public cloud infrastructure.

Containership Kubernetes Engine (CKE)

CKE is another web-based interface for provisioning Kubernetes in the public cloud. It lets you get a cluster up and running with sensible defaults, or customize almost every aspect of the cluster for more demanding requirements.

Kubernetes Installers

If managed or turnkey clusters won't work for you, then you'll need to consider some level of Kubernetes self-hosting: that is, setting up and running Kubernetes yourself on your own machines.

It's very unlikely that you'll deploy and run Kubernetes completely from scratch, except for learning and demo purposes. The vast majority of people use one or more of the available Kubernetes installer tools or services to set up and manage their clusters.

kops

kops is a command-line tool for automated provisioning of Kubernetes clusters. It's part of the Kubernetes project, and has been around a long time as an AWS-specific tool, but is now adding beta support for Google Cloud, and support for other providers is planned.

kops supports building high-availability clusters, which makes it suitable for production Kubernetes deployments. It uses declarative configuration, just like Kubernetes resources themselves, and it can not only provision the necessary cloud resources and set up a cluster, but also scale it up and down, resize nodes, perform upgrades, and do other useful admin tasks.

Like everything in the Kubernetes world, kops is under rapid development, but it's a relatively mature and sophisticated tool that is widely used. If you're planning to run self-hosted Kubernetes in AWS, kops is a good choice.

Kubespray

Kubespray (formerly known as Kargo), a project under the Kubernetes umbrella, is a tool for easily deploying production-ready clusters. It offers lots of options, including high availability, and support for multiple platforms.

Kubespray is focused on installing Kubernetes on existing machines, especially on-premise and bare-metal servers. However, it's also suitable for any cloud environment, including private cloud (virtual machines that run on your own servers).

TK8

TK8 is a command-line tool for provisioning Kubernetes clusters that leverages both Terraform (for creating cloud servers) and Kubespray (for installing Kubernetes on them). Written in Go (of course), it supports installation on AWS, OpenStack, and bare-metal servers, with support for Azure and Google Cloud in the pipeline.

TK8 not only builds a Kubernetes cluster, but will also install optional add-ons for you, including Jmeter Cluster for load testing, Prometheus for monitoring, Jaeger, Linkerd or Zipkin for tracing, Ambassador API Gateway with Envoy for ingress and load balancing, Istio for service mesh support, Jenkins-X for CI/CD, and Helm or Kedge for packaging on Kubernetes.

Kubernetes The Hard Way

Kelsey Hightower's *Kubernetes The Hard Way* tutorial is perhaps best considered not as a Kubernetes setup tool or installation guide, but an opinionated walkthrough of the process of building a Kubernetes cluster which illustrates the complexity of the moving parts involved. Nonetheless, it's very instructive, and it's an exercise worth doing for anyone considering running Kubernetes, even as a managed service, just to get a sense of how it all works under the hood.

kubeadm

kubeadm is part of the Kubernetes distribution, and it aims to help you install and maintain a Kubernetes cluster according to best practices. kubeadm does not provision the infrastructure for the cluster itself, so it's suitable for installing Kubernetes on bare-metal servers or cloud instances of any flavor.

Many of the other tools and services we'll mention in this chapter use kubeadm internally to handle cluster admin operations, but there's nothing to stop you using it directly, if you want to.

Tarmak

Tarmak is a Kubernetes cluster life cycle management tool that is focused on making it easy and reliable to modify and upgrade cluster nodes. While many tools deal with this by simply replacing the node, this can take a long time and often involves moving a lot of data around between nodes during the rebuild process. Instead, Tarmak can repair or upgrade the node in place.

Tarmak uses Terraform under the hood to provision the cluster nodes, and Puppet to manage configuration on the nodes themselves. This makes it quicker and safer to roll out changes to node configuration.

Rancher Kubernetes Engine (RKE)

RKE aims to be a simple, fast Kubernetes installer. It doesn't provision the nodes for you, and you have to install Docker on the nodes yourself before you can use RKE to install the cluster. RKE supports high availability of the Kubernetes control plane.

Puppet Kubernetes Module

Puppet is a powerful, mature, and sophisticated general configuration management tool that is very widely used, and has a large open source module ecosystem. The officially supported Kubernetes module installs and configures Kubernetes on existing nodes, including high availability support for both the control plane and `etcd`.

Kubeformation

Kubeformation is an online Kubernetes configurator that lets you choose the options for your cluster using a web interface, and will then generate configuration templates for your particular cloud provider's automation API (for example, Deployment Manager for Google Cloud, or Azure Resource Manager for Azure). Support for other cloud providers is in the pipeline.

Using Kubeformation is perhaps not as simple as some other tools, but because it is a wrapper around existing automation tools such as Deployment Manager, it is very flexible. If you already manage your Google Cloud infrastructure using Deployment Manager, for example, Kubeformation will fit into your existing workflow perfectly.

Buy or Build: Our Recommendations

This has necessarily been a quick tour of some of the options available for managing Kubernetes clusters, because the range of offerings is large and varied, and growing all the time. However, we can make a few recommendations based on commonsense principles. One of these is the philosophy of *run less software*.

Run Less Software

> There are three pillars of the Run Less Software philosophy, all of which will help you manipulate time and defeat your enemies.
>
> 1. Choose standard technology
> 2. Outsource undifferentiated heavy lifting
> 3. Create enduring competitive advantage
>
> —Rich Archbold

While using innovative new technologies is fun and exciting, it doesn't always make sense from a business point of view. Using *boring* software that everybody else is using is generally a good bet. It probably works, it's probably well-supported, and you're not going to be the one taking the risks and dealing with the inevitable bugs.

If you're running containerized workloads and cloud native applications, Kubernetes is the boring choice, in the best possible way. Given that, you should opt for the most mature, stable, and widely used Kubernetes tools and services.

Undifferentiated heavy lifting is a term coined at Amazon to denote all the hard work and effort that goes into things like installing and managing software, maintaining infrastructure, and so on. There's nothing special about this work; it's the same for you as it is for every other company out there. It costs you money, instead of making you money.

The *run less software* philosophy says that you should outsource undifferentiated heavy lifting, because it'll be cheaper in the long run, and it frees up resources you can use to work on your core business.

Use Managed Kubernetes if You Can

With the *run less software* principles in mind, we recommend that you outsource your Kubernetes cluster operations to a managed service. Installing, configuring, maintaining, securing, upgrading, and making your Kubernetes cluster reliable is undifferentiated heavy lifting, so it makes sense for almost all businesses not to do it themselves:

> *Cloud native* is not a cloud provider, it's not Kubernetes, it's not containers, it's not a technology. It's the practice of accelerating your business by not running stuff that doesn't differentiate you.
>
> —Justin Garrison

In the managed Kubernetes space, Google Kubernetes Engine (GKE) is the clear winner. While other cloud providers may catch up in a year or two, Google is still way ahead and will remain so for some time to come.

For companies that need to be independent of a single cloud provider, and want 24-hour-a-day technical support from a trusted brand, Heptio Kubernetes Subscription is worth looking at.

If you want managed high availability for your cluster control plane, but need the flexibility of running your own worker nodes, consider Stackpoint.

But What About Vendor Lock-in?

If you commit to a managed Kubernetes service from a particular vendor, such as Google Cloud, will that lock you in to the vendor and reduce your options in the future? Not necessarily. Kubernetes is a standard platform, so any applications and services you build to run on Google Kubernetes Engine will also work on any other certified Kubernetes provider's system. Just using Kubernetes in the first place is a big step toward escaping vendor lock-in.

Does managed Kubernetes make you more prone to lock-in than running your own Kubernetes cluster? We think it's the other way around. Self-hosting Kubernetes involves a lot of machinery and configuration to maintain, all of which is intimately tied in to a specific cloud provider's API. Provisioning AWS virtual machines to run Kubernetes, for example, requires completely different code than the same operation on Google Cloud. Some Kubernetes setup assistants, like the ones we've mentioned in this chapter, support multiple cloud providers, but many don't.

Part of the point of Kubernetes is to abstract away the technical details of the cloud platform, and present developers with a standard, familiar interface that works the same way whether it happens to be running on Azure or Google Cloud. As long as you design your applications and automation to target Kubernetes itself, rather than the underlying cloud infrastructure, you're as free from vendor lock-in as you can reasonably be.

Use Standard Kubernetes Self-Hosting Tools if You Must

If you have special requirements which mean that managed Kubernetes offerings won't work for you, only then should you consider running Kubernetes yourself.

If that's the case, you should go with the most mature, powerful, and widely used tools available. We recommend kops or Kubespray, depending on your requirements.

If you know that you'll be staying with a single cloud provider long-term, especially if it's AWS, use kops.

On the other hand, if you need your cluster to span multiple clouds or platforms, including bare-metal servers, and you want to keep your options open, you should use Kubespray.

When Your Choices Are Limited

There may be business, rather than technical, reasons, why fully managed Kubernetes services aren't an option for you. If you have an existing business relationship with a hosting company or cloud provider that doesn't offer a managed Kubernetes service, that will necessarily limit your choices.

However, it may be possible for you to use a turnkey solution instead, such as Stackpoint or Containership. These options provide a managed service for your Kubernetes master nodes, but connect them to worker nodes running on your own infrastructure. Since most of the administration overhead of Kubernetes is in setting up and maintaining the master nodes, this is a good compromise.

Bare-Metal and On-Prem

It may come as a surprise to you that being cloud native doesn't actually require being *in the cloud*, in the sense of outsourcing your infrastructure to a public cloud provider such as Azure or AWS.

Many organizations run part or all of their infrastructure on bare-metal hardware, whether colocated in data centers or on-premises. Everything we've said in this book about Kubernetes and containers applies just as well to in-house infrastructure as it does to the cloud.

You can run Kubernetes on your own hardware machines; if your budget is limited, you can even run it on a stack of Raspberry Pis (Figure 2-3). Some businesses run a *private cloud*, consisting of virtual machines hosted by on-prem hardware.

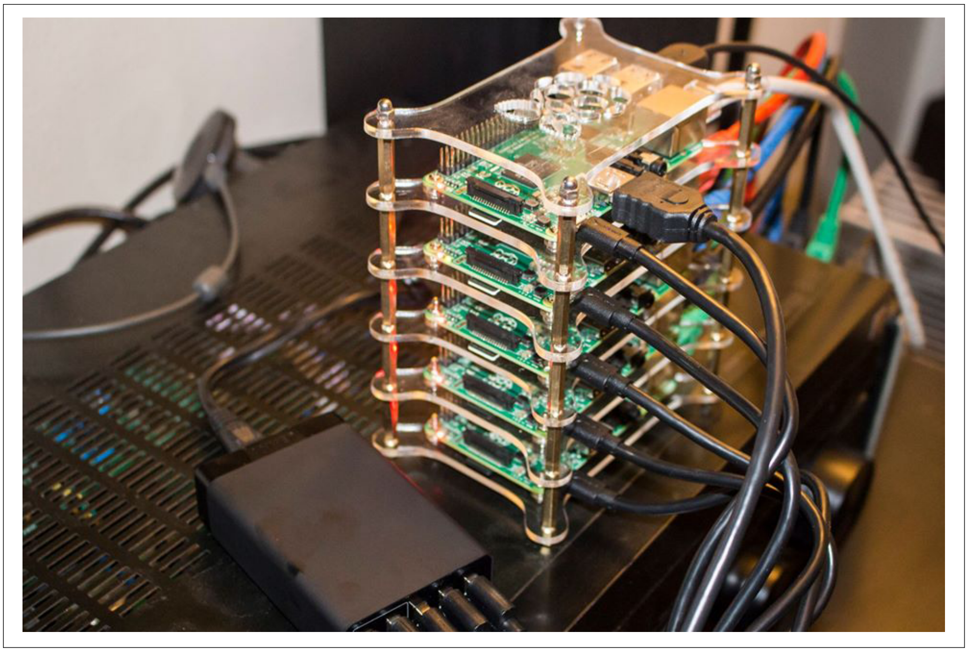


Figure 2-3. Kubernetes on a budget: a Raspberry Pi cluster (photo by David Merrick)

Clusterless Container Services

If you really want to minimize the overhead of running container workloads, there's yet another level above fully managed Kubernetes services. These are so-called *clusterless* services, such as Azure Container Instances or Amazon's Fargate. Although there really is a cluster under the hood, you don't have access to it via tools like `kubectl`. Instead, you specify a container image to run, and a few parameters like the CPU and memory requirements of your application, and the service does the rest.

Amazon Fargate

According to Amazon, "Fargate is like EC2, but instead of a virtual machine, you get a container." Unlike ECS, there's no need to provision cluster nodes yourself and then connect them to a control plane. You just define a task, which is essentially a set of instructions for how to run your container image, and launch it. Pricing is per-second based on the amount of CPU and memory resources that the task consumes.

It's probably fair to say that Fargate makes sense for simple, self-contained, long-running compute tasks or batch jobs (such as data crunching) that don't require much customization or integration with other services. It's also ideal for build containers, which tend to be short-lived, and for any situation where the overhead of managing worker nodes isn't justified.

If you're already using ECS with EC2 worker nodes, switching to Fargate will relieve you of the need to provision and manage those nodes. Fargate is available now in some regions for running ECS tasks, and is scheduled to support EKS by 2019.

Azure Container Instances (ACI)

Microsoft's Azure Container Instances (ACI) service is similar to Fargate, but also offers integration with the Azure Kubernetes Service (AKS). For example, you can configure your AKS cluster to provision temporary extra Pods inside ACI to handle spikes or bursts in demand.

Similarly, you can run batch jobs in ACI in an ad hoc way, without having to keep idle nodes around when there's no work for them to do. Microsoft calls this idea *serverless containers*, but we find that terminology both confusing (*serverless* usually refers to cloud functions, or functions-as-a-service) and inaccurate (there are servers; you just can't access them).

ACI is also integrated with Azure Event Grid, Microsoft's managed event routing service. Using Event Grid, ACI containers can communicate with cloud services, cloud functions, or Kubernetes applications running in AKS.

You can create, run, or pass data to ACI containers using Azure Functions. The advantage of this is that you can run any workload from a cloud function, not just those using the officially supported (*blessed*) languages, such as Python or JavaScript.

If you can containerize your workload, you can run it as a cloud function, with all the associated tooling. For example, Microsoft Flow allows even nonprogrammers to build up workflows graphically, connecting containers, functions, and events.

Summary

Kubernetes is everywhere! Our journey through the extensive landscape of Kubernetes tools, services, and products has been necessarily brief, but we hope you found it useful.

While our coverage of specific products and features is as up to date as we can make it, the world moves pretty fast, and we expect a lot will have changed even by the time you read this.

However, we think the basic point stands: it's not worth managing Kubernetes clusters yourself if a service provider can do it better and cheaper.

In our experience of consulting for companies migrating to Kubernetes, this is often a surprising idea, or at least not one that occurs to a lot of people. We often find that organizations have taken their first steps with self-hosted clusters, using tools like kops, and hadn't really thought about using a managed service such as GKE. It's well worth thinking about.

More things to bear in mind:

- Kubernetes clusters are made up of *master nodes*, which run the *control plane*, and *worker nodes*, which run your workloads.
- Production clusters must be *highly available*, meaning that the failure of a master node won't lose data or affect the operation of the cluster.
- It's a long way from a simple demo cluster to one that's ready for critical production workloads. High availability, security, and node management are just some of the issues involved.
- Managing your own clusters requires a significant investment of time, effort, and expertise. Even then, you can still get it wrong.
- Managed services like Google Kubernetes Engine do all the heavy lifting for you, at much lower cost than self-hosting.
- Turnkey services are a good compromise between self-hosted and fully managed Kubernetes. Turnkey providers like Stackpoint manage the master nodes for you, while you run worker nodes on your own machines.
- If you have to host your own cluster, kops is a mature and widely used tool that can provision and manage production-grade clusters on AWS and Google Cloud.
- You should use managed Kubernetes if you can. This is the best option for most businesses in terms of cost, overhead, and quality.
- If managed services aren't an option, consider using turnkey services as a good compromise.
- Don't self-host your cluster without sound business reasons. If you do self-host, don't underestimate the engineering time involved for the initial setup and ongoing maintenance overhead.

CHAPTER 3

Working with Kubernetes Objects

I can't understand why people are frightened of new ideas. I'm frightened of the old ones.

—John Cage

In Chapter 1, you built and deployed an application to Kubernetes. In this chapter, you'll learn about the fundamental Kubernetes objects involved in that process: Pods, Deployments, and Services. You'll also find out how to use the essential Helm tool to manage application in Kubernetes.

After working through the example in "Running the Demo App" on page 10, you should have a container image running in the Kubernetes cluster, but how does that actually work? Under the hood, the `kubectl run` command creates a Kubernetes resource called a *Deployment*. So what's that? And how does a Deployment actually run your container image?

Deployments

Think back to how you ran the demo app with Docker. The `docker container run` command started the container, and it ran until you killed it with `docker stop`.

But suppose that the container exits for some other reason; maybe the program crashed, or there was a system error, or your machine ran out of disk space, or a cosmic ray hit your CPU at the wrong moment (unlikely, but it does happen). Assuming this is a production application, that means you now have unhappy users, until someone can get to a terminal and type `docker container run` to start the container again.

That's an unsatisfactory arrangement. What you really want is a kind of supervisor program, which continually checks that the container is running, and if it ever stops, starts it again immediately. On traditional servers, you can use a tool like `systemd`, `runit`, or `supervisord` to do this; Docker has something similar, and you won't be surprised to know that Kubernetes has a supervisor feature too: the *Deployment*.

Supervising and Scheduling

For each program that Kubernetes has to supervise, it creates a corresponding Deployment object, which records some information about the program: the name of the container image, the number of replicas you want to run, and whatever else it needs to know to start the container.

Working together with the Deployment resource is a kind of Kubernetes object called a *controller*. Controllers watch the resources they're responsible for, making sure they're present and working. If a given Deployment isn't running enough replicas, for whatever reason, the controller will create some new ones. (If there were too many replicas for some reason, the controller would shut down the excess ones. Either way, the controller makes sure that the real state matches the desired state.)

Actually, a Deployment doesn't manage replicas directly: instead, it automatically creates an associated object called a ReplicaSet, which handles that. We'll talk more about ReplicaSets in a moment in "ReplicaSets" on page 36, but since you generally interact only with Deployments, let's get more familiar with them first.

Restarting Containers

At first sight, the way Deployments behave might be a little surprising. If your container finishes its work and exits, the Deployment will restart it. If it crashes, or if you kill it with a signal, or terminate it with `kubectl`, the Deployment will restart it. (This is how you should think about it conceptually; the reality is a little more complicated, as we'll see.)

Most Kubernetes applications are designed to be long-running and reliable, so this behavior makes sense: containers can exit for all sorts of reasons, and in most cases all a human operator would do is restart them, so that's what Kubernetes does by default.

It's possible to change this policy for an individual container: for example, to never restart it, or to restart it only on failure, not if it exited normally (see "Restart Policies" on page 65). However, the default behavior (restart always) is usually what you want.

A Deployment's job is to watch its associated containers and make sure that the specified number of them is always running. If there are fewer, it will start more. If there are too many, it will terminate some. This is much more powerful and flexible than a traditional supervisor-type program.

Querying Deployments

You can see all the Deployments active in your current namespace (see ???) by running the following command:

```
kubectl get deployments
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
demo      1         1         1            1           21h
```

To get more detailed information on this specific Deployment, run the following command:

```
kubectl describe deployments/demo
Name:                   demo
Namespace:              default
CreationTimestamp:      Tue, 08 May 2018 12:20:50 +0100
...
```

As you can see, there's a lot of information here, most of which isn't important for now. Let's look more closely at the `Pod Template` section, though:

```
Pod Template:
  Labels:  app=demo
  Containers:
   demo:
    Image:        cloudnatived/demo:hello
    Port:         8888/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
```

You know that a Deployment contains the information Kubernetes needs to run the container, and here it is. But what's a Pod Template? Actually, before we answer that, what's a Pod?

Pods

A Pod is the Kubernetes object that represents a group of one or more containers (*pod* is also the name for a group of whales, which fits in with the vaguely seafaring flavor of Kubernetes metaphors).

Why doesn't a Deployment just manage an individual container directly? The answer is that sometimes a set of containers needs to be scheduled together, running on the same node, and communicating locally, perhaps sharing storage.

For example, a blog application might have one container that syncs content with a Git repository, and an Nginx web server container that serves the blog content to users. Since they share data, the two containers need to be scheduled together in a

Pod. In practice, though, many Pods only have one container, as in this case. (See "What Belongs in a Pod?" on page 52 for more about this.)

So a Pod specification (*spec* for short) has a list of `Containers`, and in our example there is only one container, `demo`:

```
demo:
 Image:        cloudnatived/demo:hello
 Port:         8888/TCP
 Host Port:    0/TCP
 Environment:  <none>
 Mounts:       <none>
```

The `Image` spec will be, in your case, **`YOUR_DOCKER_ID`**`/myhello`, and together with the port number, that's all the information the Deployment needs to start the Pod and keep it running.

And that's an important point. The `kubectl run` command didn't actually create the Pod directly. Instead it created a Deployment, and *then* the Deployment started the Pod. The Deployment is a declaration of your desired state: "A Pod should be running with the `myhello` container inside it."

ReplicaSets

We said that Deployments start Pods, but there's a little more to it than that. In fact, Deployments don't manage Pods directly. That's the job of the ReplicaSet object.

A ReplicaSet is responsible for a group of identical Pods, or *replicas*. If there are too few (or too many) Pods, compared to the specification, the ReplicaSet controller will start (or stop) some Pods to rectify the situation.

Deployments, in turn, manage ReplicaSets, and control how the replicas behave when you update them—by rolling out a new version of your application, for example (see ???). When you update the Deployment, a new ReplicaSet is created to manage the new Pods, and when the update is completed, the old ReplicaSet and its Pods are terminated.

In Figure 3-1, each ReplicaSet (V1, V2, V3) represents a different version of the application, with its corresponding Pods.

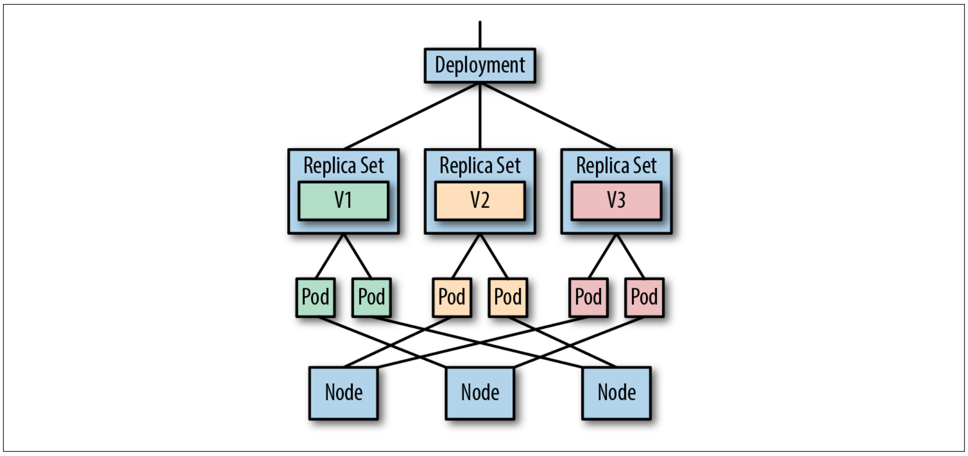


Figure 3-1. Deployments, ReplicaSets, and Pods

Usually, you won't interact with ReplicaSets directly, since Deployments do the work for you—but it's useful to know what they are.

Maintaining Desired State

Kubernetes controllers continually check the desired state specified by each resource against the actual state of the cluster, and make any necessary adjustments to keep them in sync. This process is called the *reconciliation loop*, because it loops forever, trying to reconcile the actual state with the desired state.

For example, when you first create the `demo` Deployment, there is no `demo` Pod running. So Kubernetes will start the required Pod immediately. If it were ever to stop, Kubernetes will start it again, so long as the Deployment still exists.

Let's verify that right now by stopping the Pod manually. First, check that the Pod is indeed running:

```
kubectl get pods --selector app=demo
NAME                    READY     STATUS    RESTARTS   AGE
demo-54df94b7b7-qgtc6   1/1       Running   1          22h
```

Now, run the following command to stop the Pod:

```
kubectl delete pods --selector app=demo
pod "demo-54df94b7b7-qgtc6" deleted
```

List the Pods again:

```
kubectl get pods --selector app=demo
NAME                    READY     STATUS        RESTARTS   AGE
demo-54df94b7b7-hrspp   1/1       Running       0          5s
demo-54df94b7b7-qgtc6   0/1       Terminating   1          22h
```

You can see the original Pod shutting down (its status is `Terminating`), but it's already been replaced by a new Pod, which is only five seconds old. That's the reconciliation loop at work.

You told Kubernetes, by means of the Deployment you created, that the `demo` Pod must *always* be running. It takes you at your word, and even if you delete the Pod yourself, Kubernetes assumes you must have made a mistake, and helpfully starts a new Pod to replace it for you.

Once you've finished experimenting with the Deployment, shut it down and clean up using the following command:

```
kubectl delete all --selector app=demo
pod "demo-54df94b7b7-hrspp" deleted
service "demo" deleted
deployment.apps "demo" deleted
```

The Kubernetes Scheduler

We've said things like *the Deployment will create Pods* and *Kubernetes will start the required Pod*, without really explaining how that happens.

The Kubernetes *scheduler* is the component responsible for this part of the process. When a Deployment (via its associated ReplicaSet) decides that a new replica is needed, it creates a Pod resource in the Kubernetes database. Simultaneously, this Pod is added to a queue, which is like the scheduler's inbox.

The scheduler's job is to watch its queue of unscheduled Pods, grab the next Pod from it, and find a node to run it on. It will use a few different criteria, including the Pod's resource requests, to choose a suitable node, assuming there is one available (we'll talk more about this process in ???).

Once the Pod has been scheduled on a node, the kubelet running on that node picks it up and takes care of actually starting its containers (see "Node Components" on page 15).

When you deleted a Pod in "Maintaining Desired State" on page 37, it was the node's kubelet that spotted this and started a replacement. It *knows* that a `demo` Pod should be running on its node, and if it doesn't find one, it will start one. (What would happen if you shut the node down altogether? Its Pods would become unscheduled and go back into the scheduler's queue, to be reassigned to other nodes.)

Stripe engineer Julia Evans has written a delightfully clear explanation of how scheduling works in Kubernetes.

Resource Manifests in YAML Format

Now that you know how to run an application in Kubernetes, is that it? Are you done? Not quite. Using the `kubectl run` command to create a Deployment is useful, but limited. Suppose you want to change something about the Deployment spec: the image name or version, say. You could delete the existing Deployment (using `kubectl delete`) and create a new one with the right fields. But let's see if we can do better.

Because Kubernetes is inherently a *declarative* system, continuously reconciling actual state with desired state, all you need to do is change the desired state—the Deployment spec—and Kubernetes will do the rest. How do you do that?

Resources Are Data

All Kubernetes resources, such as Deployments or Pods, are represented by records in its internal database. The reconciliation loop watches the database for any changes to those records, and takes the appropriate action. In fact, all the `kubectl run` command does is add a new record in the database corresponding to a Deployment, and Kubernetes does the rest.

But you don't need to use `kubectl run` in order to interact with Kubernetes. You can also create and edit the resource *manifest* (the specification for the desired state of the resource) directly. You can keep the manifest file in a version control system, and instead of running imperative commands to make on-the-fly changes, you can change your manifest files and then tell Kubernetes to read the updated data.

Deployment Manifests

The usual format for Kubernetes manifest files is YAML, although it can also understand the JSON format. So what does the YAML manifest for a Deployment look like?

Have a look at our example for the demo application (*hello-k8s/k8s/deployment.yaml*):

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: demo
  labels:
    app: demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
        - name: demo
          image: cloudnatived/demo:hello
          ports:
          - containerPort: 8888
```

At first glance, this looks complicated, but it's mostly boilerplate. The only interesting parts are the same information that you've already seen in various forms: the container image name and port. When you gave this information to `kubectl run` earlier, it created the equivalent of this YAML manifest behind the scenes and submitted it to Kubernetes.

Using kubectl apply

To use the full power of Kubernetes as a declarative infrastructure as code system, submit YAML manifests to the cluster yourself, using the `kubectl apply` command.

Try it with our example Deployment manifest, *hello-k8s/k8s/deployment.yaml*.[1]

Run the following commands in your copy of the demo repo:

```
cd hello-k8s
kubectl apply -f k8s/deployment.yaml
deployment.apps "demo" created
```

After a few seconds, a `demo` Pod should be running:

```
kubectl get pods --selector app=demo
NAME                        READY     STATUS    RESTARTS   AGE
demo-6d99bf474d-z9zv6       1/1       Running   0          2m
```

We're not quite done, though, because in order to connect to the `demo` Pod with a web browser, we're going to create a Service, which is a Kubernetes resource that lets you connect to your deployed Pods (more on this in a moment).

First, let's explore what a Service is, and why we need one.

Service Resources

Suppose you want to make a network connection to a Pod (such as our example application). How do you do that? You could find out the Pod's IP address and connect directly to that address and the app's port number. But the IP address may change when the Pod is restarted, so you'll have to keep looking it up to make sure it's up to date.

1 *k8s*, pronounced *kates*, is a common abbreviation for *Kubernetes*, following the geeky pattern of abbreviating words as a *numeronym*: their first and last letters, plus the number of letters in between (*k-8-s*). See also *i18n* (internationalization), *a11y* (accessibility), and *o11y* (observability).

Worse, there may be multiple replicas of the Pod, each with different addresses. Every other application that needs to contact the Pod would have to maintain a list of those addresses, which doesn't sound like a great idea.

Fortunately, there's a better way: a Service resource gives you a single, unchanging IP address or DNS name that will be automatically routed to any matching Pod. Later on in "Ingress Resources" on page 87 we will talk about the Ingress resource, which allows for more advanced routing and using TLS certificates.

But for now, let's take a closer look at how a Kubernetes Service works.

You can think of a Service as being like a web proxy or a load balancer, forwarding requests to a set of *backend* Pods (Figure 3-2). However, it isn't restricted to web ports: a Service can forward traffic from any port to any other port, as detailed in the `ports` part of the spec.

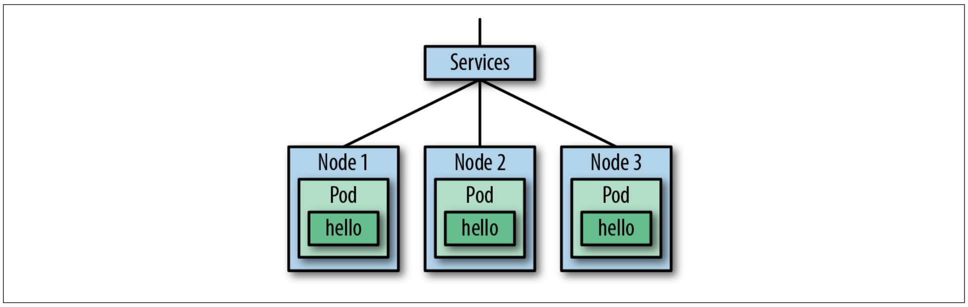


Figure 3-2. A Service provides a persistent endpoint for a group of Pods

Here's the YAML manifest of the Service for our demo app:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: demo
  labels:
    app: demo
spec:
  ports:
  - port: 9999
    protocol: TCP
    targetPort: 8888
  selector:
    app: demo
  type: ClusterIP
```

You can see that it looks somewhat similar to the Deployment resource we showed earlier. However, the `kind` is `Service`, instead of `Deployment`, and the `spec` just includes a list of `ports`, plus a `selector` and a `type`.

If you zoom in a little, you can see that the Service is forwarding its port 9999 to the Pod's port 8888:

```
...
ports:
- port: 9999
  protocol: TCP
  targetPort: 8888
```

The `selector` is the part that tells the Service how to route requests to particular Pods. Requests will be forwarded to any Pods matching the specified set of labels; in this case, just `app: demo` (see "Labels" on page 69). In our example, there's only one Pod that matches, but if there were multiple Pods, the Service would send each request to a randomly selected one.[2]

In this respect, a Kubernetes Service is a little like a traditional load balancer, and, in fact, both Services and Ingresses can automatically create cloud load balancers (see "Ingress Resources" on page 87).

For now, the main thing to remember is that a Deployment manages a set of Pods for your application, and a Service gives you a single entry point for requests to those Pods.

Go ahead and apply the manifest now, to create the Service:

```
kubectl apply -f k8s/service.yaml
service "demo" created

kubectl port-forward service/demo 9999:8888
Forwarding from 127.0.0.1:9999 -> 8888
Forwarding from [::1]:9999 -> 8888
```

As before, `kubectl port-forward` will connect the `demo` service to a port on your local machine, so that you can connect to *http://localhost:9999/* with your web browser.

Once you're satisfied that everything is working correctly, run the following command to clean up before moving on to the next section:

```
kubectl delete -f k8s/
```

2 This is the default load balancing algorithm; Kubernetes versions 1.10+ support other algorithms too, such as *least connection*. See *https://kubernetes.io/blog/2018/07/09/ipvs-based-in-cluster-load-balancing-deep-dive/*.

You can use `kubectl delete` with a label selector, as we did earlier on, to delete all resources that match the selector (see "Labels" on page 69). Alternatively, you can use `kubectl delete -f`, as here, with a directory of manifests. All the resources described by the manifest files will be deleted.

Exercise

Modify the *k8s/deployment.yaml* file to change the number of replicas to 3. Reapply the manifest using `kubectl apply` and check that you get three `demo` Pods instead of one, using `kubectl get pods`.

Querying the Cluster with kubectl

The `kubectl` tool is the Swiss Army knife of Kubernetes: it applies configuration, creates, modifies, and destroys resources, and can also query the cluster for information about the resources that exist, as well as their status.

We've already seen how to use `kubectl get` to query Pods and Deployments. You can also use it to see what nodes exist in your cluster:

```
kubectl get nodes
NAME                 STATUS    ROLES     AGE       VERSION
docker-for-desktop   Ready     master    1d        v1.10.0
```

If you want to see resources of all types, use `kubectl get all`. (In fact, this doesn't show literally *all* resources, just the most common types, but we won't quibble about that for now.)

To see comprehensive information about an individual Pod (or any other resource), use `kubectl describe`:

```
kubectl describe pod/demo-dev-6c96484c48-69vss
Name:           demo-dev-6c96484c48-69vss
Namespace:      default
Node:           docker-for-desktop/10.0.2.15
Start Time:     Wed, 06 Jun 2018 10:48:50 +0100
...
Containers:
  demo:
    Container ID:   docker://646aaf7c4baf6d...
    Image:          cloudnatived/demo:hello
...
Conditions:
  Type           Status
  Initialized    True
  Ready          True
  PodScheduled   True
```

```
...
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  1d    default-scheduler  Successfully assigned demo-dev...
  Normal  Pulling    1d    kubelet            pulling image "cloudnatived/demo...
...
```

In the example output, you can see that `kubectl` gives you some basic information about the container itself, including its image identifier and status, along with an ordered list of events that have happened to the container. (We'll learn a lot more about the power of `kubectl` in ???.)

Taking Resources to the Next Level

You now know everything you need to know to deploy applications to Kubernetes clusters using declarative YAML manifests. But there's a lot of repetition in these files: for example, you've repeated the name `demo`, the label selector `app: demo`, and the port `8888` several times.

Shouldn't you be able to just specify those values once, and then reference them wherever they occur through the Kubernetes manifests?

For example, it would be great to be able to define variables called something like `container.name` and `container.port`, and then use them wherever they're needed in the YAML files. Then, if you needed to change the name of the app or the port number it listens on, you'd only have to change them in one place, and all the manifests would be updated automatically.

Fortunately, there's a tool for that, and in the final section of this chapter we'll show you a little of what it can do.

Helm: A Kubernetes Package Manager

One popular package manager for Kubernetes is called Helm, and it works just the way we've described in the previous section. You can use the `helm` command-line tool to install and configure applications (your own or anyone else's), and you can create packages called Helm *charts*, which completely specify the resources needed to run the application, its dependencies, and its configurable settings.

Helm is part of the Cloud Native Computing Foundation family of projects (see ???), which reflects its stability and widespread adoption.

It's important to realize that a Helm chart, unlike the binary software packages used by tools like APT or Yum, doesn't actually include the container image itself. Instead, it simply contains metadata about where the image can be found, just as a Kubernetes Deployment does.

When you install the chart, Kubernetes itself will locate and download the binary container image from the place you specified. In fact, a Helm chart is really just a convenient wrapper around Kubernetes YAML manifests.

Installing Helm

Follow the Helm installation instructions for your operating system.

Once you have Helm installed, you will need to create some Kubernetes resources to authorize it for access to your cluster. There's a suitable YAML file in the example repo, in the *hello-helm* directory, named *helm-auth.yaml*. To apply it, run the following commands:

```
cd ../hello-helm
kubectl apply -f helm-auth.yaml
serviceaccount "tiller" created
clusterrolebinding.rbac.authorization.k8s.io "tiller" created
```

Now that the necessary permissions are in place, you can initialize Helm for access to the cluster, using this command:

```
helm init --service-account tiller
$HELM_HOME has been configured at /Users/john/.helm.
```

It may take five minutes or so for Helm to finish initializing itself. To verify that Helm is up and working, run:

```
helm version
Client: &version.Version{SemVer:"v2.9.1",
GitCommit:"20adb27c7c5868466912eebdf6664e7390ebe710", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.9.1",
GitCommit:"20adb27c7c5868466912eebdf6664e7390ebe710", GitTreeState:"clean"}
```

Once this command succeeds, you're ready to start using Helm. If you see an `Error: cannot connect to Tiller` message, wait a few minutes and try again.

Installing a Helm Chart

What would the Helm chart for our demo application look like? In the *hello-helm* directory, you'll see a *k8s* subdirectory, which in the previous example (`hello-k8s`) contained just the Kubernetes manifest files to deploy the application. Now it contains a Helm chart, in the *demo* directory:

```
ls k8s/demo
Chart.yaml             prod-values.yaml staging-values.yaml    templates
values.yaml
```

We'll see what all these files are for in ???, but for now, let's use Helm to install the demo application. First, clean up the resources from any previous deployments:

```
kubectl delete all --selector app=demo
```

Then run the following command:

```
helm install --name demo ./k8s/demo
NAME:   demo
LAST DEPLOYED: Wed Jun  6 10:48:50 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Service
NAME              TYPE       CLUSTER-IP     EXTERNAL-IP  PORT(S)   AGE
demo-service      ClusterIP  10.98.231.112  <none>       80/TCP  0s

==> v1/Deployment
NAME      DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
demo      1        1        1           0          0s

==> v1/Pod(related)
NAME                        READY  STATUS             RESTARTS  AGE
demo-6c96484c48-69vss       0/1    ContainerCreating  0         0s
```

You can see that Helm has created a Deployment resource (which starts a Pod) and a Service, just as in the previous example. The `helm install` does this by creating a Kubernetes object called a Helm *release*.

Charts, Repositories, and Releases

These are the three most important Helm terms you need to know:

- A *chart* is a Helm package, containing all the resource definitions necessary to run an application in Kubernetes.
- A *repository* is a place where charts can be collected and shared.
- A *release* is a particular instance of a chart running in a Kubernetes cluster.

One chart can often be installed many times into the same cluster. For example, you might be running multiple copies of the Nginx web server chart, each serving a different site. Each separate instance of the chart is a distinct release.

Each release has a unique name, which you specify with the `-name` flag to `helm install`. (If you don't specify a name, Helm will choose a random one for you. You probably won't like it.)

Listing Helm Releases

To check what releases you have running at any time, run `helm list`:

```
helm list
NAME    REVISION UPDATED                  STATUS   CHART      NAMESPACE
demo    1        Wed Jun  6 10:48:50 2018 DEPLOYED demo-1.0.1 default
```

To see the exact status of a particular release, run `helm status` followed by the name of the release. You'll see the same information that you did when you first deployed the release: a list of all the Kubernetes resources associated with the release, including Deployments, Pods, and Services.

Later in the book, we'll show you how to build your own Helm charts for your applications (see ???). For now, just know that Helm is a handy way to install applications from public charts.

You can see the full list of public Helm charts on GitHub.

You can also get a list of available charts by running `helm search` with no arguments (or `helm search redis` to search for a Redis chart, for example).

Summary

This isn't a book about Kubernetes internals (sorry, no refunds). Our aim is to show you what Kubernetes can *do*, and bring you quickly to the point where you can run real workloads in production. However, it's useful to know at least some of the main pieces of machinery you'll be working with, such as Pods and Deployments. In this chapter we've briefly introduced some of the most important ones.

As fascinating as the technology is to geeks like us, though, we're also interested in getting stuff done. Therefore, we haven't exhaustively covered every kind of resource Kubernetes provides, because there are a *lot*, and many of them you almost certainly won't need (at least, not yet).

The key points we think you need to know right now:

- The Pod is the fundamental unit of work in Kubernetes, specifying a single container or group of communicating containers that are scheduled together.
- A Deployment is a high-level Kubernetes resource that declaratively manages Pods, deploying, scheduling, updating, and restarting them when necessary.
- A Service is the Kubernetes equivalent of a load balancer or proxy, routing traffic to its matching Pods via a single, well-known, durable IP address or DNS name.
- The Kubernetes scheduler watches for a Pod that isn't yet running on any node, finds a suitable node for it, and instructs the kubelet on that node to run the Pod.

- Resources like Deployments are represented by records in Kubernetes's internal database. Externally, these resources can be represented by text files (known as *manifests*) in YAML format. The manifest is a declaration of the desired state of the resource.
- `kubectl` is the main tool for interacting with Kubernetes, allowing you to apply manifests, query resources, make changes, delete resources, and do many other tasks.
- Helm is a Kubernetes package manager. It simplifies configuring and deploying Kubernetes applications, allowing you to use a single set of values (such as the application name or listen port) and a set of templates to generate Kubernetes YAML files, instead of having to maintain the raw YAML files yourself.

CHAPTER 4

Running Containers

If you have a tough question that you can't answer, start by tackling a simpler question that you can't answer.

—Max Tegmark

In previous chapters, we've focused mostly on the operational aspects of Kubernetes: where to get your clusters, how to maintain them, and how to manage your cluster resources. Let's turn now to the most fundamental Kubernetes object: the *container*. We'll look at how containers work on a technical level, how they relate to Pods, and how to deploy container images to Kubernetes.

In this chapter, we'll also cover the important topic of container security, and how to use the security features in Kubernetes to deploy your applications in a secure way, according to best practices. Finally, we'll look at how to mount disk volumes on Pods, allowing containers to share and persist data.

Containers and Pods

We've already introduced Pods in Chapter 1, and talked about how Deployments use ReplicaSets to maintain a set of replica Pods, but we haven't really looked at Pods themselves in much detail. Pods are the unit of scheduling in Kubernetes. A Pod object represents a container or group of containers, and everything that runs in Kubernetes does so by means of a Pod:

> A Pod represents a collection of application containers and volumes running in the same execution environment. Pods, not containers, are the smallest deployable artifact in a Kubernetes cluster. This means all of the containers in a Pod always land on the same machine.
>
> —Kelsey Hightower et al., *Kubernetes Up & Running*

So far in this book the terms *Pod* and *container* have been used more or less interchangeably: the demo application Pod only has one container in it. In more complex applications, though, it's quite likely that a Pod will include two or more containers. So let's look at how that works, and see when and why you might want to group containers together in Pods.

What Is a Container?

Before asking why you might want to have multiple containers in a Pod, let's take a moment to revisit what a container actually is.

You know from ??? that a container is a standardized package that contains a piece of software together with its dependencies, configuration, data, and so on: everything it needs to run. How does that actually work, though?

In Linux and most other operating systems, everything that runs on a machine does so by means of a *process*. A process represents the binary code and memory state of a running application, such as Chrome, iTunes, or Visual Studio Code. All processes exist in the same global namespace: they can all see and interact with each other, they all share the same pool of resources, such as CPU, memory, and filesystem. (A Linux namespace is a bit like a Kubernetes namespace, though not the same thing technically.)

From the operating system's point of view, a container represents an isolated process (or group of processes) that exists in its own namespace. Processes inside the container can't see processes outside it, and vice versa. A container can't access resources belonging to another container, or processes outside of a container. The container boundary is like a ring fence that stops processes running wild and using up each other's resources.

As far as the process inside the container is concerned, it's running on its own machine, with complete access to all its resources, and there are no other processes running. You can see this if you run a few commands inside a container:

```
kubectl run busybox --image busybox:1.28 --rm -it --restart=Never /bin/sh
If you don't see a command prompt, try pressing enter.
/ # ps ax
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
    8 root      0:00 ps ax

/ # hostname
busybox
```

Normally, the `ps ax` command will list all processes running on the machine, and there are usually a lot of them (a few hundred on a typical Linux server). But there are only two processes shown here: `/bin/sh`, and `ps ax`. The only processes visible inside the container, therefore, are the ones actually running in the container.

Similarly, the `hostname` command, which would normally show the name of the host machine, returns `busybox`: in fact, this is the name of the container. So it looks to the `busybox` container as if it's running on a machine called `busybox`, and it has the whole machine to itself. This is true for each of the containers running on the same machine.

It's a fun exercise to create a container yourself, without the benefit of a container runtime like Docker. Liz Rice's excellent talk on "What is a container, really?" shows how to do this from scratch in a Go program.

What Belongs in a Container?

There's no technical reason you can't run as many processes as you want to inside a container: you could run a complete Linux distribution, with multiple running applications, network services, and so on, all inside the same container. This is why you sometimes hear containers referred to as *lightweight virtual machines*. But this isn't the best way to use containers, because then you don't get the benefits of resource isolation.

If processes don't need to know about each other, then they don't need to run in the same container. A good rule of thumb with a container is that it should *do one thing*. For example, our demo application container listens on a network port, and sends the string `Hello, 世界` to anyone who connects to it. That's a simple, self-contained service: it doesn't rely on any other programs or services, and in turn, nothing relies on it. It's a perfect candidate for having its own container.

A container also has an *entrypoint*: a command that is run when the container starts. That usually results in the creation of a single process to run the command, though some applications often start a few subprocesses to act as helpers or workers. To start multiple separate processes in a container, you'd need to write a wrapper script to act as the entrypoint, which would in turn start the processes you want.

Each container should run just one main process. If you're running a large group of unrelated processes in a container, you're not taking full advantage of the power of containers, and you should think about splitting your application up into multiple, communicating containers.

What Belongs in a Pod?

Now that you know what a container is, you can see why it's useful to group them together in Pods. A Pod represents a group of containers that need to communicate and share data with each other; they need to be scheduled together, they need to be started and stopped together, and they need to run on the same physical machine.

A good example of this is an application that stores data in a local cache, such as Memcached. You'll need to run two processes: your application, and the `memcached` server process that handles storing and retrieving data. Although you could run both processes inside a single container, that's unnecessary: they only need to communicate via a network socket. Better to split them into two separate containers, each of which only needs to worry about building and running its own process.

In fact, you can use a public Memcached container image, available from Docker Hub, which is already set up to work as part of a Pod with another container.

So you create a Pod with two containers: Memcached, and your application. The application can talk to Memcached by making a network connection, and because the two containers are in the same Pod, that connection will always be local: the two containers will always run on the same node.

Similarly, imagine a blog application, which consists of a web server container, such as Nginx, and a Git synchronizer container, which clones a Git repo containing the blog data: HTML files, images, and so on. The blog container writes data to disk, and because containers in a Pod can share a disk volume, the data can also be available to the Nginx container to serve over HTTP:

> In general, the right question to ask yourself when designing Pods is, "Will these containers work correctly if they land on different machines?" If the answer is "no", a Pod is the correct grouping for the containers. If the answer is "yes", multiple Pods is the probably the correct solution.
>
> —Kelsey Hightower et al., *Kubernetes Up & Running*

The containers in a Pod should all be working together to do one job. If you only need one container to do that job, fine: use one container. If you need two or three, that's OK. If you have more than that, you might want to think about whether the containers could actually be split into separate Pods.

Container Manifests

We've outlined what containers are, what should go in a container, and when containers should be grouped together in Pods. So how do we actually run a container in Kubernetes?

When you created your first Deployment, in "Deployment Manifests" on page 39, it contained a `template.spec` section specifying the container to run (only one container, in that example):

```
spec:
  containers:
  - name: demo
    image: cloudnatived/demo:hello
    ports:
    - containerPort: 8888
```

Here's an example of what the `template.spec` section for a Deployment with two containers would look like:

```
spec:
  containers:
  - name: container1
    image: example/container1
  - name: container2
    image: example/container2
```

The only required fields in each container's spec are the `name` and `image`: a container has to have a name, so that other resources can refer to it, and you have to tell Kubernetes what image to run in the container.

Image Identifiers

You've already used some different container image identifiers so far in this book; for example, `cloudnatived/demo:hello`, `alpine`, and `busybox:1.28`.

There are actually four different parts to an image identifier: the *registry hostname*, the *repository namespace*, the *image repository*, and the *tag*. All but the image name are optional. An image identifier using all of those parts looks like this:

```
docker.io/cloudnatived/demo:hello
```

- The registry hostname in this example is `docker.io`; in fact, that's the default for Docker images, so we don't need to specify it. If your image is stored in another registry, though, you'll need to give its hostname. For example, Google Container Registry images are prefixed by `gcr.io`.
- The repository namespace is `cloudnatived`: that's us (hello!). If you don't specify the repository namespace, then the default namespace (called `library`) is used. This is a set of official images, which are approved and maintained by Docker, Inc. Popular official images include OS base images (`alpine`, `ubuntu`, `debian`, `centos`), language environments (`golang`, `python`, `ruby`, `php`, `java`), and widely used software (`mongo`, `mysql`, `nginx`, `redis`).

- The image repository is `demo`, which identifies a particular container image within the registry and namespace. (See also "Container Digests" on page 54.)
- The tag is `hello`. Tags identify different versions of the same image.

It's up to you what tags to put on a container: some common choices include:

- A semantic version tag, like `v1.3.0`. This usually refers to the version of the application.
- A Git SHA tag, like `5ba6bfd...`. This identifies the specific commit in the source repo that was used to build the container (see ???).
- The environment it represents, such as `staging` or `production`.

You can add as many tags as you want to a given image.

The latest Tag

If you don't specify a tag when pulling an image, the default tag is `latest`. For example, when you run an `alpine` image, with no tag specified, you'll get `alpine:latest`.

The `latest` tag is a default tag that's added to an image when you build or push it without specifying a tag. It doesn't necessarily identify the most recent image; just the most recent image that wasn't explicitly tagged. This makes `latest` rather unhelpful as an identifier.

That's why it's important to always use a specific tag when deploying production containers to Kubernetes. When you're just running a quick one-off container, for troubleshooting or experimentation, like the `alpine` container, it's fine to omit the tag and get the latest image. For real applications, though, you want to make sure that if you deploy the Pod tomorrow, you'll get the exact same container image as when you deployed it today:

> You should avoid using the `latest` tag when deploying containers in production, because this makes it hard to track which version of the image is running and hard to roll back.
>
> —The Kubernetes documentation

Container Digests

As we've seen, the `latest` tag doesn't always mean what you think it will, and even a semantic version or Git SHA tag doesn't uniquely and permanently identify a particular container image. If the maintainer decides to push a different image with the same tag, the next time you deploy, you'll get that updated image. In technical terms, a tag is *nondeterministic*.

Sometimes it's desirable to have deterministic deployments: in other words, to guarantee that a deployment will always reference the exact container image you specified. You can do this using the container's *digest*: a cryptographic hash of the image's contents that immutably identifies that image.

Images can have many tags, but only one digest. This means that if your container manifest specifies the image digest, you can guarantee deterministic deployments. An image identifier with a digest looks like this:

```
cloudnatived/
demo@sha256:aeae1e551a6cbd60bcfd56c3b4ffec732c45b8012b7cb758c6c4a34...
```

Base Image Tags

When you reference a base image in a Dockerfile, if you don't specify a tag, you'll get `latest`, just as you do when deploying a container. Because of the tricky semantics of `latest`, as we've seen, it's a good idea to use a specific base image tag instead, like `alpine:3.8`.

When you make a change to your application and rebuild its container, you don't want to also get unexpected changes as a result of a newer public base image. That may cause problems that are hard to find and debug.

To make your builds as reproducible as possible, use a specific tag or digest.

> We've said that you should avoid using the `latest` tag, but it's fair to note that there's some room for disagreement about this. Even the present authors have different preferences. Always using `latest` base images means that if some change to the base image breaks your build, you'll find out right away. On the other hand, using specific image tags means that you only have to upgrade your base image when *you* want to, not when the upstream maintainers decide to. It's up to you.

Ports

You've already seen the `ports` field used with our demo application: it specifies the network port numbers the application will listen on. This is just informational, and has no significance to Kubernetes, but it's good practice to include it.

Resource Requests and Limits

We've already covered resource requests and limits for containers in detail, in ???, so a brief recap here will suffice.

Each container can supply one or more of the following as part of its spec:

- `resources.requests.cpu`
- `resources.requests.memory`
- `resources.limits.cpu`
- `resources.limits.memory`

Although requests and limits are specified on individual containers, we usually talk in terms of the Pod's resource requests and limits. A Pod's resource request is the sum of the resource requests for all containers in that Pod, and so on.

Image Pull Policy

As you know, before a container can be run on a node, the image has to be *pulled*, or downloaded, from the appropriate container registry. The `imagePullPolicy` field on a container governs how often Kubernetes will do this. It can take one of three values: `Always`, `IfNotPresent`, or `Never`:

- `Always` will pull the image every time the container is started. Assuming that you specify a tag, which you should (see "The latest Tag" on page 54), then this is unnecessary, and wastes time and bandwidth.
- `IfNotPresent`, the default, is correct for most situations. If the image is not already present on the node, it will be downloaded. After that, unless you change the image spec, the saved image will be used every time the container starts, and Kubernetes will not attempt to redownload it.
- `Never` will never update the image at all. With this policy, Kubernetes will never fetch the image from a registry: if it's already present on the node, it will be used, but if it's not, the container will fail to start. You're unlikely to want this.

If you run into strange problems (for example, a Pod not updating when you've pushed a new container image), check your image pull policy.

Environment Variables

Environment variables are a common, if limited, way to pass information to containers at runtime. Common, because all Linux executables have access to environment variables, and even programs that were written long before containers existed can use their environment for configuration. Limited, because environment variables can only be string values: no arrays, no keys and values, no structured data in general. The total size of a process's environment is also limited to 32 KiB, so you can't pass large data files in the environment.

To set an environment variable, list it in the container's `env` field:

```
containers:
- name: demo
  image: cloudnatived/demo:hello
  env:
  - name: GREETING
    value: "Hello from the environment"
```

If the container image itself specifies environment variables (set in the Dockerfile, for example), then the Kubernetes `env` settings will override them. This can be useful for altering the default configuration of a container.

A more flexible way of passing configuration data to containers is to use a Kubernetes ConfigMap or Secret object: see ??? for more about these.

Container Security

You might have noticed in "What Is a Container?" on page 50 that when we looked at the process list in the container with the `ps ax` command, the processes were all running as the `root` user. In Linux and other Unix-derived operating systems, `root` is the superuser, which has privileges to read any data, modify any file, and perform any operation on the system.

While on a full Linux system some processes need to run as root (for example `init`, which manages all other processes), that's not usually the case with a container.

Indeed, running processes as the `root` user when you don't need to is a bad idea. It contravenes the *principle of least privilege*. This says that a program should only be able to access the information and resources that it actually needs to do its job.

Programs have bugs; this is a fact of life apparent to anyone who's written one. Some bugs allow malicious users to hijack the program to do things it's not supposed to, like read secret data, or execute arbitrary code. To mitigate this, it's important to run containers with the minimum possible privileges.

This starts with not allowing them to run as `root`, but instead assigning them an *ordinary* user: one that has no special privileges, such as reading other users' files:

> Just like you wouldn't (or shouldn't) run anything as root on your server, you shouldn't run anything as root in a container on your server. Running binaries that were created elsewhere requires a significant amount of trust, and the same is true for binaries in containers.
>
> —Marc Campbell

It's also possible for attackers to exploit bugs in the container runtime to "escape" from the container, and get the same powers and privileges on the host machine that they did in the container.

Running Containers as a Non-Root User

Here's an example of a container spec that tells Kubernetes to run the container as a specific user:

```
containers:
- name: demo
  image: cloudnatived/demo:hello
  securityContext:
    runAsUser: 1000
```

The value for `runAsUser` is a *UID* (a numerical user identifier). On many Linux systems, UID 1000 is assigned to the first non-root user created on the system, so it's generally safe to choose values of 1000 or above for container UIDs. It doesn't matter whether or not a Unix user with that UID *exists* in the container, or even if there is an operating system in the container; this works just as well with scratch containers.

Docker also allows you to specify a user in the Dockerfile to run the container's process, but you needn't bother to do this. It's easier and more flexible to set the `runAsUser` field in the Kubernetes spec.

If a `runAsUser` UID is specified, it will override any user configured in the container image. If there is no `runAsUser`, but the container specifies a user, Kubernetes will run it as that user. If no user is specified either in the manifest or the image, the container will run as `root` (which, as we've seen, is a bad idea).

For maximum security, you should choose a different UID for each container. That way, if a container should be compromised somehow, or accidentally overwrite data, it only has permission to access its own data, and not that of other containers.

On the other hand, if you want two or more containers to be able to access the same data (via a mounted volume, for example), you should assign them the same UID.

Blocking Root Containers

To help prevent this situation, Kubernetes allows you to block containers from running if they would run as the root user.

The `runAsNonRoot: true` setting will do this:

```
containers:
- name: demo
  image: cloudnatived/demo:hello
  securityContext:
    runAsNonRoot: true
```

When Kubernetes runs this container, it will check to see if the container wants to run as root. If so, it will refuse to start it. This will protect you against forgetting to set a non-root user in your containers, or running third-party containers that are configured to run as root.

If this happens, you'll see the Pod status shown as `CreateContainerConfigError`, and when you `kubectl describe` the Pod, you'll see this error:

```
Error: container has runAsNonRoot and image will run as root
```

Best Practice

Run containers as non-root users, and block root containers from running, using the `runAsNonRoot: true` setting.

Setting a Read-Only Filesystem

Another useful security context setting is `readOnlyRootFilesystem`, which will prevent the container from writing to its own filesystem. It's possible to imagine a container taking advantage of a bug in Docker or Kubernetes, for example, where writing to its filesystem could affect files on the host node. If its filesystem is read-only, that can't happen; the container will get an I/O error:

```
containers:
- name: demo
  image: cloudnatived/demo:hello
  securityContext:
    readOnlyRootFilesystem: true
```

Many containers don't need to write anything to their own filesystem, so this setting won't interfere with them. It's good practice to always set `readOnlyRootFilesystem` unless the container really does need to write to files.

Disabling Privilege Escalation

Normally, Linux binaries run with the same privileges as the user that executes them. There is an exception, though: binaries that use the `setuid` mechanism can temporarily gain the privileges of the user that *owns* the binary (usually `root`).

This is a potential problem in containers, since even if the container is running as a regular user (UID 1000, for example), if it contains a `setuid` binary, that binary can gain root privileges by default.

To prevent this, set the `allowPrivilegeEscalation` field of the container's security policy to `false`:

```
containers:
- name: demo
  image: cloudnatived/demo:hello
  securityContext:
    allowPrivilegeEscalation: false
```

To control this setting across the whole cluster, rather than for an individual container, see "Pod Security Policies" on page 61.

Modern Linux programs don't need `setuid`; they can use a more flexible and fine-grained privilege mechanism called *capabilities* to achieve the same thing.

Capabilities

Traditionally, Unix programs had two levels of privileges: *normal* and *superuser*. Normal programs have no more privileges than the user who runs them, while superuser programs can do anything, bypassing all kernel security checks.

The Linux capabilities mechanism improves on this by defining various specific things that a program can do: load kernel modules, perform direct network I/O operations, access system devices, and so on. Any program that needs a specific privilege can be granted it, but no others.

For example, a web server that listens on port 80 would normally need to run as `root` to do this; port numbers below 1024 are considered privileged *system* ports. Instead, the program can be granted the `NET_BIND_SERVICE` capability, which allows it to bind to any port, but gives it no other special privileges.

The default set of capabilities for Docker containers is fairly generous. This is a pragmatic decision based on a trade-off of security against usability: giving containers *no* capabilities by default would require operators to set capabilities on many containers in order for them to run.

On the other hand, the principle of least privilege says that a container should have no capabilities it doesn't need. Kubernetes security contexts allow you to drop any capabilities from the default set, and add ones as they're needed, like this example shows:

```
containers:
- name: demo
  image: cloudnatived/demo:hello
  securityContext:
    capabilities:
      drop: ["CHOWN", "NET_RAW", "SETPCAP"]
      add: ["NET_ADMIN"]
```

The container will have the `CHOWN`, `NET_RAW`, and `SETPCAP` capabilities removed, and the `NET_ADMIN` capability added.

The Docker documentation lists all the capabilities that are set on containers by default, and that can be added as necessary.

For maximum security, you should drop all capabilities for every container, and only add specific capabilities if they're needed:

```
containers:
- name: demo
  image: cloudnatived/demo:hello
  securityContext:
    capabilities:
      drop: ["all"]
      add: ["NET_BIND_SERVICE"]
```

The capability mechanism puts a hard limit on what processes inside the container can do, even if they're running as root. Once a capability has been dropped at the container level, it can't be regained, even by a malicious process with maximum privileges.

Pod Security Contexts

We've covered security context settings at the level of individual containers, but you can also set some of them at the Pod level:

```
apiVersion: v1
kind: Pod
...
spec:
  securityContext:
    runAsUser: 1000
    runAsNonRoot: false
    allowPrivilegeEscalation: false
```

These settings will apply to all containers in the Pod, unless the container overrides a given setting in its own security context.

Best Practice

Set security contexts on all your Pods and containers. Disable privilege escalation and drop all capabilities. Add only the specific capabilities that a given container needs.

Pod Security Policies

Rather than have to specify all the security settings for each individual container or Pod, you can specify them at the cluster level using a PodSecurityPolicy resource. A PodSecurityPolicy looks like this:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
```

```
metadata:
  name: example
spec:
  privileged: false
  # The rest fills in some required fields.
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
  - *
```

This simple policy blocks any privileged containers (those with the `privileged` flag set in their `securityContext`, which would give them almost all the capabilities of a process running natively on the node).

It's a little more complicated to use PodSecurityPolicies, as you have to create the policies, grant the relevant service accounts access to the policies via RBAC (see ???), and enable the PodSecurityPolicy admission controller in your cluster. For larger infrastructures, though, or where you don't have direct control over the security configuration of individual Pods, PodSecurityPolicies are a good idea.

You can read about how to create and enable PodSecurityPolicies in the Kubernetes documentation.

Pod Service Accounts

Pods run with the permissions of the default service account for the namespace, unless you specify otherwise (see ???). If you need to grant extra permissions for some reason (such as viewing Pods in other namespaces), create a dedicated service account for the app, bind it to the required roles, and configure the Pod to use the new service account.

To do that, set the `serviceAccountName` field in the Pod spec to the name of the service account:

```
apiVersion: v1
kind: Pod
...
spec:
  serviceAccountName: deploy-tool
```

Volumes

As you may recall, each container has its own filesystem, which is accessible only to that container, and is *ephemeral*: any data that is not part of the container image will be lost when the container is restarted.

Often, this is fine; the demo application, for example, is a stateless server which therefore needs no persistent storage. Nor does it need to share files with any other container.

More complex applications, though, may need both the ability to share data with other containers in the same Pod, and to have it persist across restarts. A Kubernetes Volume object can provide both of these.

There are many different types of Volume that you can attach to a Pod. Whatever the underlying storage medium, a Volume mounted on a Pod is accessible to all the containers in the Pod. Containers that need to communicate by sharing files can do so using a Volume of one kind or another. We'll look at some of the more important types in the following sections.

emptyDir Volumes

The simplest Volume type is `emptyDir`. This is a piece of ephemeral storage that starts out empty—hence the name—and stores its data on the node (either in memory, or on the node's disk). It persists only as long as the Pod is running on that node.

An `emptyDir` is useful when you want to provision some extra storage for a container, but it's not critical to have the data persist forever or move with the container if it should be scheduled on another node. Some examples include caching downloaded files or generated content, or using a scratch workspace for data processing jobs.

Similarly, if you just want to share files between containers in a Pod, but don't need to keep the data around for a long time, an `emptyDir` Volume is ideal.

Here's an example of a Pod that creates an `emptyDir` Volume and mounts it on a container:

```
apiVersion: v1
kind: Pod
...
spec:
  volumes:
  - name: cache-volume
    emptyDir: {}
  containers:
  - name: demo
    image: cloudnatived/demo:hello
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
```

First, in the volumes section of the Pod spec, we create an emptyDir Volume named cache-volume:

```
volumes:
- name: cache-volume
  emptyDir: {}
```

Now the cache-volume Volume is available for any container in the Pod to mount and use. To do that, we list it in the volumeMounts section of the demo container:

```
name: demo
image: cloudnatived/demo:hello
volumeMounts:
- mountPath: /cache
  name: cache-volume
```

The container doesn't have to do anything special to use the new storage: anything it writes to the path /cache will be written to the Volume, and will be visible to other containers that mount the same Volume. All containers mounting the Volume can read and write to it.

Be careful writing to shared Volumes. Kubernetes doesn't enforce any locking on disk writes. If two containers try to write to the same file at once, data corruption can result. To avoid this, either implement your own write-lock mechanism, or use a Volume type that supports locking, such as nfs or glusterfs.

Persistent Volumes

While an ephemeral emptyDir Volume is ideal for cache and temporary file-sharing, some applications need to store persistent data; for example, any kind of database. In general, we don't recommend that you run databases in Kubernetes. You're almost always better served by using a cloud service instead: for example, most cloud providers have managed solutions for relational databases such as MySQL and PostgreSQL, as well as key-value (*NoSQL*) stores.

As we saw in ???, Kubernetes is best at managing stateless applications, which means no persistent data. Storing persistent data significantly complicates the Kubernetes configuration for your app, uses extra cloud resources, and it also needs to be backed up.

However, if you need to use persistent volumes with Kubernetes, the PersistentVolume resource is what you're looking for. We won't go into great detail about them here, because the details tend to be specific to your cloud provider; you can read more about PersistentVolumes in the Kubernetes documentation.

The most flexible way to use PersistentVolumes in Kubernetes is to create a PersistentVolumeClaim object. This represents a request for a particular type and size of PersistentVolume; for example, a 10 GiB Volume of high-speed, read-write storage.

The Pod can then add this PersistentVolumeClaim as a Volume, where it will be available for containers to mount and use:

```
volumes:
- name: data-volume
  persistentVolumeClaim:
    claimName: data-pvc
```

You can create a pool of PersistentVolumes in your cluster to be claimed by Pods in this way. Alternatively, you can set up *dynamic provisioning*: when a PersistentVolumeClaim like this is mounted, a suitable chunk of storage will be automatically provisioned and connected to the Pod.

Restart Policies

We saw in ??? that Kubernetes always restarts a Pod when it exits, unless you tell it otherwise. The default restart policy is thus `Always`, but you can change this to `OnFailure` (restart only if the container exited with a nonzero status), or `Never`:

```
apiVersion: v1
kind: Pod
...
spec:
  restartPolicy: OnFailure
```

If you want to run a Pod to completion and then have it exit, rather than being restarted, you can use a Job resource to do this (see "Jobs" on page 82).

Image Pull Secrets

As you know, Kubernetes will download your specified image from the container registry if it isn't already present on the node. However, what if you're using a private registry? How can you give Kubernetes the credentials to authenticate to the registry?

The `imagePullSecrets` field on a Pod allows you to configure this. First, you need to store the registry credentials in a Secret object (see ??? for more about this). Now you can tell Kubernetes to use this Secret when pulling any containers in the Pod. For example, if your Secret is named `registry-creds`:

```
apiVersion: v1
kind: Pod
...
spec:
  imagePullSecrets:
  - name: registry-creds
```

The exact format of the registry credentials data is described in the Kubernetes documentation.

You can also attach `imagePullSecrets` to a service account (see "Pod Service Accounts" on page 62). Any Pods created using this service account will automatically have the attached registry credentials available.

Summary

In order to understand Kubernetes, you first need to understand containers. In this chapter, we've outlined the basic idea of what a container is, how they work together in Pods, and what options are available for you to control how containers run in Kubernetes.

The bare essentials:

- A Linux container, at the kernel level, is an isolated set of processes, with ring-fenced resources. From inside a container, it looks as though the container has a Linux machine to itself.
- Containers are not virtual machines. Each container should run one primary process.
- A Pod usually contains one container that runs a primary application, plus optional *helper* containers that support it.
- Container image specifications can include a registry hostname, a repository namespace, an image repository, and a tag; for example `docker.io/cloudnatived/demo:hello`. Only the image name is required.
- For reproducible deployments, always specify a tag for the container image. Otherwise, you'll get whatever happens to be `latest`.
- Programs in containers should not run as the `root` user. Instead, assign them an ordinary user.
- You can set the `runAsNonRoot: true` field on a container to block any container that wants to run as `root`.
- Other useful security settings on containers include `readOnlyRootFilesystem: true` and `allowPrivilegeEscalation: false`.
- Linux capabilities provide a fine-grained privilege control mechanism, but the default capabilities for containers are too generous. Start by dropping all capabilities for containers, then grant specific capabilities if a container needs them.
- Containers in the same Pod can share data by reading and writing a mounted Volume. The simplest Volume is of type `emptyDir`, which starts out empty and preserves its contents only as long as the Pod is running.

- A PersistentVolume, on the other hand, preserves its contents as long as needed. Pods can dynamically provision new PersistentVolumes using PersistentVolume-Claims.

CHAPTER 5

Managing Pods

There are no big problems, there are just a lot of little problems.

—Henry Ford

In the previous chapter we covered containers in some detail, and explained how containers are composed to form Pods. There are a few other interesting aspects of Pods, which we'll turn to in this chapter, including labels, guiding Pod scheduling using node affinities, barring Pods from running on certain nodes with taints and tolerations, keeping Pods together or apart using Pod affinities, and orchestrating applications using Pod controllers such as DaemonSets and StatefulSets.

We'll also cover some advanced networking options including Ingress resources, Istio, and Envoy.

Labels

You know that Pods (and other Kubernetes resources) can have labels attached to them, and that these play an important role in connecting related resources (for example, sending requests from a Service to the appropriate backends). Let's take a closer look at labels and selectors in this section.

What Are Labels?

Labels are key/value pairs that are attached to objects, such as pods. Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system.

—The Kubernetes documentation

In other words, labels exist to tag resources with information that's meaningful to us, but they don't mean anything to Kubernetes. For example, it's common to label Pods with the application they belong to:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: demo
```

Now, by itself, this label has no effect. It's still useful as documentation: someone can look at this Pod and see what application it's running. But the real power of a label comes when we use it with a *selector*.

Selectors

A selector is an expression that matches a label (or set of labels). It's a way of specifying a group of resources by their labels. For example, a Service resource has a selector that identifies the Pods it will send requests to. Remember our demo Service from "Service Resources" on page 40?

```
apiVersion: v1
kind: Service
...
spec:
  ...
  selector:
    app: demo
```

This is a very simple selector that matches any resource that has the `app` label with a value of `demo`. If a resource doesn't have the `app` label at all, it won't match this selector. If it has the `app` label, but its value is not `demo`, it won't match the selector either. Only suitable resources (in this case, Pods) with the label `app: demo` will match, and all such resources will be selected by this Service.

Labels aren't just used for connecting Services and Pods; you can use them directly when querying the cluster with `kubectl get`, using the `--selector` flag:

```
kubectl get pods --all-namespaces --selector app=demo
NAMESPACE   NAME                      READY     STATUS    RESTARTS   AGE
demo        demo-5cb7d6bfdd-9dckm     1/1       Running   0          20s
```

You may recall from ??? that `--selector` can be abbreviated to just `-l` (for ***l**abels*).

If you want to see what labels are defined on your Pods, use the `--show-labels` flag to `kubectl get`:

```
kubectl get pods --show-labels
NAME                       ... LABELS
demo-5cb7d6bfdd-9dckm      ... app=demo,environment=development
```

More Advanced Selectors

Most of the time, a simple selector like `app: demo` (known as an *equality selector*) will be all you need. You can combine different labels to make more specific selectors:

```
kubectl get pods -l app=demo,environment=production
```

This will return only Pods that have *both* `app: demo` and `environment: production` labels. The YAML equivalent of this (in a Service, for example) would be:

```
selector:
  app: demo
  environment: production
```

Equality selectors like this are the only kind available with a Service, but for interactive queries with `kubectl`, or more sophisticated resources such as Deployments, there are other options.

One is selecting for label *inequality*:

```
kubectl get pods -l app!=demo
```

This will return all Pods that have an `app` label with a different value to `demo`, or that don't have an `app` label at all.

You can also ask for label values that are in a *set*:

```
kubectl get pods -l environment in (staging, production)
```

The YAML equivalent would be:

```
selector:
  matchExpressions:
  - {key: environment, operator: In, values: [staging, production]}
```

You can also ask for label values not in a given set:

```
kubectl get pods -l environment notin (production)
```

The YAML equivalent of this would be:

```
selector:
  matchExpressions:
  - {key: environment, operator: NotIn, values: [production]}
```

You can see another example of using `matchExpressions` in ???.

Other Uses for Labels

We've seen how to link Pods to Services using an `app` label (actually, you can use any label, but `app` is common). But what other uses are there for labels?

In our Helm chart for the demo application (see ???) we set a `environment` label, which can be, for example, `staging` or `production`. If you're running staging and production Pods in the same cluster (see ???) you might want to use a label like this to distinguish between the two environments. For example, your Service selector for production might be:

```
selector:
  app: demo
  environment: production
```

Without the extra `environment` selector, the Service would match any and all Pods with `app: demo`, including the staging ones, which you probably don't want.

Depending on your applications, you might want to use labels to slice and dice your resources in a number of different ways. Here are some examples:

```
metadata:
  labels:
    app: demo
    tier: frontend
    environment: production
    environment: test
    version: v1.12.0
    role: primary
```

This allows you to query the cluster along these various different dimensions to see what's going on.

You could also use labels as a way of doing canary deployments (see ???). If you want to roll out a new version of the application to just a small percentage of Pods, you could use labels like `track: stable` and `track: canary` for two separate Deployments.

If your Service's selector matches only the `app` label, it will send traffic to all Pods matching that selector, including both `stable` and `canary`. You can alter the number of replicas for both Deployments to gradually increase the proportion of `canary` Pods. Once all running Pods are on the canary track, you can relabel them as `stable` and begin the process again with the next version.

Labels and Annotations

You might be wondering what the difference is between labels and annotations. They're both sets of key-value pairs that provide metadata about resources.

The difference is that *labels identify resources*. They're used to select groups of related resources, like in a Service's selector. Annotations, on the other hand, are for non-identifying information, to be used by tools or services outside Kubernetes. For example, in ??? there's an example of using annotations to control Helm workflows.

Because labels are often used in internal queries that are performance-critical to Kubernetes, there are some fairly tight restrictions on valid labels. For example, label names are limited to 63 characters, though they may have an optional 253-character prefix in the form of a DNS subdomain, separated from the label by a slash character. Labels can only begin with an alphanumeric character (a letter or a digit), and can only contain alphanumeric characters plus dashes, underscores, and dots. Label values are similarly restricted.

In practice, we doubt you'll run out of characters for your labels, since most labels in common use are just a single word (for example, `app`).

Node Affinities

We mentioned node affinities briefly in ???, in relation to preemptible nodes. In that section, you learned how to use node affinities to preferentially schedule Pods on certain nodes (or not). Let's take a more detailed look at node affinities now.

In most cases, you don't need node affinities. Kubernetes is pretty smart about scheduling Pods onto the right nodes. If all your nodes are equally suitable to run a given Pod, then don't worry about it.

There are exceptions, however (like preemptible nodes in the previous example). If a Pod is expensive to restart, you probably want to avoid scheduling it on a preemptible node wherever possible; preemptible nodes can disappear from the cluster without warning. You can express this kind of preference using node affinities.

There are two types of affinity: hard and soft. Because software engineers aren't always the best at naming things, in Kubernetes these are called:

- `requiredDuringSchedulingIgnoredDuringExecution` (hard)
- `preferredDuringSchedulingIgnoredDuringExecution` (soft)

It may help you to remember that `required` means a hard affinity (the rule *must* be satisfied to schedule this Pod) and `preferred` means a soft affinity (it would be *nice* if the rule were satisfied, but it's not critical).

The long names of the hard and soft affinity types make the point that these rules apply *during scheduling*, but not *during execution*. That is, once the Pod has been scheduled to a particular node satisfying the affinity, it will stay there. If things change while the Pod is running, so that the rule is no longer satisfied, Kubernetes won't move the Pod. (This feature may be added in the future.)

Hard Affinities

An affinity is expressed by describing the kind of nodes that you want the Pod to run on. There might be several rules about how you want Kubernetes to select nodes for the Pod. Each one is expressed using the `nodeSelectorTerms` field. Here's an example:

```
apiVersion: v1
kind: Pod
...
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: "failure-domain.beta.kubernetes.io/zone"
            operator: In
            values: ["us-central1-a"]
```

Only nodes that are in the `us-central1-a` zone will match this rule, so the overall effect is to ensure that this Pod is only scheduled in that zone.

Soft Affinities

Soft affinities are expressed in much the same way, except that each rule is assigned a numerical *weight* from 1 to 100 that determines the effect it has on the result. Here's an example:

```
preferredDuringSchedulingIgnoredDuringExecution:
- weight: 10
  preference:
    matchExpressions:
    - key: "failure-domain.beta.kubernetes.io/zone"
      operator: In
      values: ["us-central1-a"]
- weight: 100
  preference:
    matchExpressions:
    - key: "failure-domain.beta.kubernetes.io/zone"
      operator: In
      values: ["us-central1-b"]
```

Because this is a `preferred...` rule, it's a soft affinity: Kubernetes can schedule the Pod on any node, but it will give priority to nodes that match these rules.

You can see that the two rules have different `weight` values. The first rule has weight 10, but the second has weight 100. If there are nodes that match both rules, Kubernetes will give 10 times the priority to nodes that match the second rule (being in availability zone `us-central1-b`).

Weights are a useful way of expressing the relative importance of your preferences.

Pod Affinities and Anti-Affinities

We've seen how you can use node affinities to nudge the scheduler toward or away from running a Pod on certain kinds of nodes. But is it possible to influence scheduling decisions based on what other Pods are already running on a node?

Sometimes there are pairs of Pods that work better when they're together on the same node; for example, a web server and a content cache, such as Redis. It would be useful if you could add information to the Pod spec that tells the scheduler it would prefer to be colocated with a Pod matching a particular set of labels.

Conversely, sometimes you want Pods to avoid each other. In ??? we saw the kind of problems that can arise if Pod replicas end up together on the same node, instead of distributed across the cluster. Can you tell the scheduler to avoid scheduling a Pod where another replica of that Pod is already running?

That's exactly what you can do with Pod affinities. Like node affinities, Pod affinities are expressed as a set of rules: either hard requirements, or soft preferences with a set of weights.

Keeping Pods Together

Let's take the first case first: scheduling Pods together. Suppose you have one Pod, labeled `app: server`, which is your web server, and another, labeled `app: cache`, which is your content cache. They can still work together even if they're on separate nodes, but it's better if they're on the same node, because they can communicate without having to go over the network. How do you ask the scheduler to colocate them?

Here's an example of the required Pod affinity, expressed as part of the `server` Pod spec. The effect would be just the same if you added it to the `cache` spec, or to both Pods:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: server
  labels:
    app: server
...
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        labelSelector:
        - matchExpressions:
          - key: app
            operator: In
```

```
              values: ["cache"]
            topologyKey: kubernetes.io/hostname
```

The overall effect of this affinity is to ensure that the `server` Pod is scheduled, if possible, on a node that is also running a Pod labeled `cache`. If there is no such node, or if there is no matching node that has sufficient spare resources to run the Pod, it will not be able to run.

This probably isn't the behavior you want in a real-life situation. If the two Pods absolutely must be colocated, put their containers in the same Pod. If it's just preferable for them to be colocated, use a soft Pod affinity (`preferredDuringSchedulingIgnoredDuringExecution`).

Keeping Pods Apart

Now let's take the anti-affinity case: keeping certain Pods apart. Instead of `podAffinity`, we use `podAntiAffinity`:

```
apiVersion: v1
kind: Pod
metadata:
  name: server
  labels:
    app: server
...
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        labelSelector:
        - matchExpressions:
          - key: app
            operator: In
            values: ["server"]
        topologyKey: kubernetes.io/hostname
```

It's very similar to the previous example, except that it's a `podAntiAffinity`, so it expresses the opposite sense, and the match expression is different. This time, the expression is: "The `app` label must have the value `server`."

The effect of this affinity is to ensure that the Pod will *not* be scheduled on any node matching this rule. In other words, no Pod labeled `app: server` can be scheduled on a node that already has an `app: server` Pod running. This will enforce an even distribution of `server` Pods across the cluster, at the possible expense of the desired number of replicas.

Soft Anti-Affinities

However, we usually care more about having enough replicas available than distributing them as fairly as possible. A hard rule is not really what we want here. Let's modify it slightly to make it a soft anti-affinity:

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 1
      podAffinityTerm:
        labelSelector:
        - matchExpressions:
          - key: app
            operator: In
            values: ["server"]
        topologyKey: kubernetes.io/hostname
```

Notice that now the rule is `preferred...`, not `required...`, making it a soft anti-affinity. If the rule can be satisfied, it will be, but if not, Kubernetes will schedule the Pod anyway.

Because it's a preference, we specify a `weight` value, just as we did for soft node affinities. If there were multiple affinity rules to consider, Kubernetes would prioritize them according to the weight you assign each rule.

When to Use Pod Affinities

Just as with node affinities, you should treat Pod affinities as a fine-tuning enhancement for special cases. The scheduler is already good at placing Pods to get the best performance and availability from the cluster. Pod affinities restrict the scheduler's freedom, trading off one application against another. When you use one, it should be because you've observed a problem in production, and a Pod affinity is the only way to fix it.

Taints and Tolerations

In "Node Affinities" on page 73, you learned about a property of Pods that can steer them toward (or away from) a set of nodes. Conversely, *taints* allow a node to repel a set of Pods, based on certain properties of the node.

For example, you could use taints to create dedicated nodes: nodes that are reserved only for specific kinds of Pods. Kubernetes also creates taints for you if certain problems exist on the node, such as low memory, or a lack of network connectivity.

To add a taint to a particular node, use the `kubectl taint` command:

```
kubectl taint nodes docker-for-desktop dedicated=true:NoSchedule
```

This adds a taint called `dedicated=true` to the `docker-for-desktop` node, with the effect `NoSchedule`: no Pod can now be scheduled there unless it has a matching *toleration*.

To see the taints configured on a particular node, use `kubectl describe node...`.

To remove a taint from a node, repeat the `kubectl taint` command but with a trailing minus sign after the name of the taint:

```
kubectl taint nodes docker-for-desktop dedicated:NoSchedule-
```

Tolerations are properties of Pods that describe the taints that they're compatible with. For example, to make a Pod tolerate the `dedicated=true` taint, add this to the Pod's spec:

```
apiVersion: v1
kind: Pod
...
spec:
  tolerations:
  - key: "dedicated"
    operator: "Equal"
    value: "true"
    effect: "NoSchedule"
```

This is effectively saying "This Pod is allowed to run on nodes that have the `dedicated=true` taint with the effect `NoSchedule`." Because the toleration *matches* the taint, the Pod can be scheduled. Any Pod without this toleration will not be allowed to run on the tainted node.

When a Pod can't run at all because of tainted nodes, it will stay in `Pending` status, and you'll see a message like this in the Pod description:

```
Warning  FailedScheduling  4s (x10 over 2m)  default-scheduler  0/1 nodes are
available: 1 node(s) had taints that the pod didn't tolerate.
```

Other uses for taints and tolerations include marking nodes with specialized hardware (such as GPUs), and allowing certain Pods to tolerate certain kinds of node problem.

For example, if a node falls off the network, Kubernetes automatically adds the taint `node.kubernetes.io/unreachable`. Normally, this would result in its `kubelet` evicting all Pods from the node. However, you might want to keep certain Pods running, in the hope that the network will come back in a reasonable time. To do this, you could add a toleration to those Pods that matches the `unreachable` taint.

You can read more about taints and tolerations in the Kubernetes documentation.

Pod Controllers

We've talked a lot about Pods in this chapter, and that makes sense: all Kubernetes applications run in a Pod. You might wonder, though, why we need other kinds of objects at all. Isn't it enough just to create a Pod for an application and run it?

That's effectively what you get by running a container directly with `docker container run`, as we did in "Running a Container Image" on page 2. It works, but it's very limited:

- If the container exits for some reason, you have to manually restart it.
- There's only one replica of your container and no way to load-balance traffic across multiple replicas if you ran them manually.
- If you want highly available replicas, you have to decide which nodes to run them on, and take care of keeping the cluster balanced.
- When you update the container, you have to take care of stopping each running image in turn, pulling the new image and restarting it.

That's the kind of work that Kubernetes is designed to take off your hands using *controllers*. In "ReplicaSets" on page 36, we introduced the ReplicaSet controller, which manages a group of replicas of a particular Pod. It works continuously to make sure there are always the specified number of replicas, starting new ones if there aren't enough, and killing off replicas if there are too many.

You're also now familiar with Deployments, which as we saw in "Deployments" on page 33, manage ReplicaSets to control the rollout of application updates. When you update a Deployment, for example with a new container spec, it creates a new ReplicaSet to start up the new Pods, and eventually closes down the ReplicaSet that was managing the old Pods.

For most simple applications, a Deployment is all you need. But there are a few other useful kinds of Pod controllers, and we'll look briefly at a few of them in this section.

DaemonSets

Suppose you want to send logs from all your applications to a centralized log server, like an Elasticsearch-Logstash-Kibana (ELK) stack, or a SaaS monitoring product such as Datadog (see ???). There are a few ways to do that.

You could have each application include code to connect to the logging service, authenticate, write logs, and so on, but this results in a lot of duplicated code, which is inefficient.

Alternatively, you could run an extra container in every Pod that acts as a logging agent (this is called a *sidecar* pattern). This means each application doesn't have to have built-in knowledge of how to talk to the logging service, but it does means you potentially have several copies of the logging agent running on each node.

Since all it does is manage a connection to the logging service and pass on log messages to it, you really only need one copy of the logging agent on each node. This is such a common requirement that Kubernetes provides a special controller object for it: the *DaemonSet*.

The term *daemon* traditionally refers to long-running background processes on a server that handle things like logging, so by analogy, Kubernetes DaemonSets run a *daemon* container on each node in the cluster.

The manifest for a DaemonSet, as you might expect, looks very much like that for a Deployment:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  ...
spec:
  ...
  template:
    ...
    spec:
      containers:
      - name: fluentd-elasticsearch
        ...
```

Use a DaemonSet when you need to run one copy of a Pod on each of the nodes in your cluster. If you're running an application where maintaining a given number of replicas is more important than exactly which node the Pods run on, use a Deployment instead.

StatefulSets

Like a Deployment or DaemonSet, a StatefulSet is a kind of Pod controller. What a StatefulSet adds is the ability to start and stop Pods in a specific sequence.

With a Deployment, for example, all your Pods are started and stopped in a random order. This is fine for stateless services, where every replica is identical and does the same job.

Sometimes, though, you need to start Pods in a specific numbered sequence, and be able to identify them by their number. For example, distributed applications such as

Redis, MongoDB, or Cassandra create their own clusters, and need to be able to identify the cluster leader by a predictable name.

A StatefulSet is ideal for this. For example, if you create a StatefulSet named `redis`, the first Pod started will be named `redis-0`, and Kubernetes will wait until that Pod is ready before starting the next one, `redis-1`.

Depending on the application, you can use this property to cluster the Pods in a reliable way. For example, each Pod can run a startup script that checks if it is running on `redis-0`. If it is, it will be the cluster leader. If not, it will attempt to join the cluster by contacting `redis-0`.

Each replica in a StatefulSet must be running and ready before Kubernetes starts the next one, and similarly when the StatefulSet is terminated, the replicas will be shut down in reverse order, waiting for each Pod to finish before moving on to the next.

Apart from these special properties, a StatefulSet looks very similar to a normal Deployment:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  selector:
    matchLabels:
      app: redis
  serviceName: "redis"
  replicas: 3
  template:
    ...
```

To be able to address each of the Pods by a predictable DNS name, such as `redis-1`, you also need to create a Service with a `clusterIP` type of `None` (known as a *headless service*).

With a nonheadless Service, you get a single DNS entry (such as `redis`) that load-balances across all the backend Pods. With a headless service, you still get that single service DNS name, but you also get individual DNS entries for each numbered Pod, like `redis-0`, `redis-1`, `redis-2`, and so on.

Pods that need to join the Redis cluster can contact `redis-0` specifically, but applications that simply need a load-balanced Redis service can use the `redis` DNS name to talk to a randomly selected Redis Pod.

StatefulSets can also manage disk storage for their Pods, using a VolumeClaimTemplate object that automatically creates a PersistentVolumeClaim (see "Persistent Volumes" on page 64).

Jobs

Another useful type of Pod controller in Kubernetes is the Job. Whereas a Deployment runs a specified number of Pods and restarts them continually, a Job only runs a Pod for a specified number of times. After that, it is considered completed.

For example, a batch processing task or queue worker Pod usually starts up, does its work, and then exits. This is an ideal candidate to be managed by a Job.

There are two fields that control Job execution: `completions` and `parallelism`. The first, `completions`, determines the number of times the specified Pod needs to run successfully before the Job is considered complete. The default value is 1, meaning the Pod will run once.

The `parallelism` field specifies how many Pods should run at once. Again, the default value is 1, meaning that only one Pod will run at a time.

For example, suppose you want to run a queue worker Job whose purpose is to consume work items from a queue. You could set `parallelism` to 10, and leave `completions` at the default value of 1. This will start 10 Pods, each of which will keep consuming work from the queue until there is no more work to do, and then exit, at which point the Job will be completed:

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: queue-worker
spec:
  completions: 1
  parallelism: 10
  template:
    metadata:
      name: queue-worker
    spec:
      containers:
        ...
```

Alternatively, if you want to run something like a batch processing job, you could leave both `completions` and `parallelism` at 1. This will start one copy of the Pod, and wait for it to complete successfully. If it crashes, fails, or exits in any non-successful way, the Job will restart it, just like a Deployment does. Only successful exits count toward the required number of `completions`.

How do you start a Job? You could do it manually, by applying a Job manifest using `kubectl` or Helm. Alternatively, a Job might be triggered by automation; your continuous deployment pipeline, for example (see ???).

Probably the most common way to run a Job, though, is to start it periodically, at a given time of day or at a given interval. Kubernetes has a special type of Job just for this: the Cronjob.

Cronjobs

In Unix environments, scheduled jobs are run by the `cron` daemon (whose name comes from the Greek word χρόνος, meaning "time"). Accordingly, they're known as *cron jobs*, and the Kubernetes Cronjob object does exactly the same thing.

A Cronjob looks like this:

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: demo-cron
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      ...
```

The two important fields to look at in the CronJob manifest are `spec.schedule` and `spec.jobTemplate`. The `schedule` field specifies when the job will run, using the same format as the Unix `cron` utility.

The `jobTemplate` specifies the template for the Job that is to be run, and is exactly the same as a normal Job manifest (see "Jobs" on page 82).

Horizontal Pod Autoscalers

Remember that a Deployment controller maintains a specified number of Pod replicas. If one replica fails, another will be started to replace it, and if there are too many Pods for some reason, the Deployment will stop excess Pods in order to achieve the target number of replicas.

The desired replica count is set in the Deployment manifest, and we've seen that you can adjust this to increase the number of Pods if there is heavy traffic, or reduce it to scale down the Deployment if there are idle Pods.

But what if Kubernetes could adjust the number of replicas for you automatically, responding to demand? This is exactly what the Horizontal Pod Autoscaler does. (*Horizontal* scaling refers to adjusting the number of replicas of a service, in contrast to *vertical* scaling, which makes individual replicas bigger or smaller.)

A Horizontal Pod Autoscaler (HPA) watches a specified Deployment, constantly monitoring a given metric to see if it needs to scale the number of replicas up or down.

One of the most common autoscaling metrics is CPU utilization. Remember from ??? that Pods can request a certain amount of CPU resources; for example, 500 millicpus. As the Pod runs, its CPU usage will fluctuate, meaning that at any given moment the Pod is actually using some percentage of its original CPU request.

You can autoscale the Deployment based on this value: for example, you could create an HPA that targets 80% CPU utilization for the Pods. If the mean CPU usage over all the Pods in the Deployment is only 70% of their requested amount, the HPA will scale down by decreasing the target number of replicas. If the Pods aren't working very hard, we don't need so many of them.

On the other hand, if the average CPU utilization is 90%, this exceeds the target of 80%, so we need to add more replicas until the average CPU usage comes down. The HPA will modify the Deployment to increase the target number of replicas.

Each time the HPA determines that it needs to do a scaling operation, it adjusts the replicas by a different amount, based on the ratio of the actual metric value to the target. If the Deployment is very close to the target CPU utilization, the HPA will only add or remove a small number of replicas; but if it's way out of scale, the HPA will adjust it by a larger number.

Here's an example of an HPA based on CPU utilization:

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: demo-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: demo
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 80
```

The interesting fields here are:

- `spec.scaleTargetRef` specifies the Deployment to scale
- `spec.minReplicas` and `spec.maxReplicas` specify the limits of scaling
- `spec.metrics` determines the metrics that will be used for scaling

Although CPU utilization is the most common scaling metric, you can use any metrics available to Kubernetes, including both the built-in *system metrics* like CPU and memory usage, and app-specific *service metrics*, which you define and export from your application (see ???). For example, you could scale based on the application error rate.

You can read more about autoscalers and custom metrics in the Kubernetes documentation.

PodPresets

PodPresets are an experimental alpha feature in Kubernetes that allow you to inject information into Pods when they're created. For example, you could create a PodPreset that mounts a volume on all Pods matching a given set of labels.

A PodPreset is a type of object called an *admission controller*. Admission controllers watch for Pods being created and take some action when Pods matching their selector are about to be created. For example, some admission controllers might block creation of the Pod if it violates a policy, while others, like PodPreset, inject extra configuration into the Pod.

Here's an example PodPreset that adds a `cache` volume to all Pods matching the `tier: frontend` selector:

```
apiVersion: settings.k8s.io/v1alpha1
kind: PodPreset
metadata:
  name: add-cache
spec:
  selector:
    matchLabels:
      role: frontend
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

The settings defined by a PodPreset are merged with those of each Pod. If a Pod is modified by a PodPreset, you'll see an annotation like this:

```
podpreset.admission.kubernetes.io/podpreset-add-cache: "<resource version>"
```

What happens if a Pod's own settings conflict with those defined in a PodPreset, or if multiple PodPresets specify conflicting settings? In that case, Kubernetes will refuse to modify the Pod, and you'll see an event in the Pod's description with the message `Conflict on pod preset`.

Because of this, PodPresets can't be used to override a Pod's own configuration, only to fill in settings which the Pod itself doesn't specify. A Pod can opt out of being modified by PodPresets altogether, by setting the annotation:

```
podpreset.admission.kubernetes.io/exclude: "true"
```

Because PodPresets are still experimental, they may not be available on managed Kubernetes clusters, and you may have to take extra steps to enable them in your self-hosted clusters, such as supplying command-line arguments to the API server. For details, see the Kubernetes documentation.

Operators and Custom Resource Definitions (CRDs)

We saw in "StatefulSets" on page 80 that, while the standard Kubernetes objects such as Deployment and Service are fine for simple, stateless applications, they have their limitations. Some applications require multiple, collaborating Pods which have to be initialized in a particular order (for example, replicated databases or clustered services).

For applications which need more complicated management than StatefulSets can provide, Kubernetes allows you to create your own new types of object. These are called *Custom Resource Definitions* (CRDs). For example, the Velero backup tool creates custom Kubernetes objects such as Configs and Backups (see ???).

Kubernetes is designed to be extensible, and you're free to define and create any type of object you want to, using the CRD mechanism. Some CRDs just exist to store data, like the Velero BackupStorageLocation object. But you can go further, and create objects that act as Pod controllers, just like a Deployment or StatefulSet.

For example, if you wanted to create a controller object that sets up replicated, high-availability MySQL database clusters in Kubernetes, how would you go about it?

The first step would be to create a CRD for your custom controller object. In order to make it do anything, you then need to write a program that communicates with the Kubernetes API. This is easy to do, as we saw in ???. Such a program is called an *operator* (perhaps because it automates the kinds of actions that a human operator might perform).

You don't need any custom objects in order to write an operator; DevOps engineer Michael Treacher has written a nice example operator that watches for namespaces being created, and automatically adds a RoleBinding to any new namespace (see ??? for more about RoleBindings).

In general, though, operators use one or more custom objects created via CRDs, whose behavior is then implemented by a program talking to the Kubernetes API.

Ingress Resources

You can think of an Ingress as a load balancer that sits in front of a Service (see Figure 5-1). The Ingress receives requests from clients and sends them on to the Service. The Service then sends them to the right Pods, based on the label selector (see "Service Resources" on page 40).

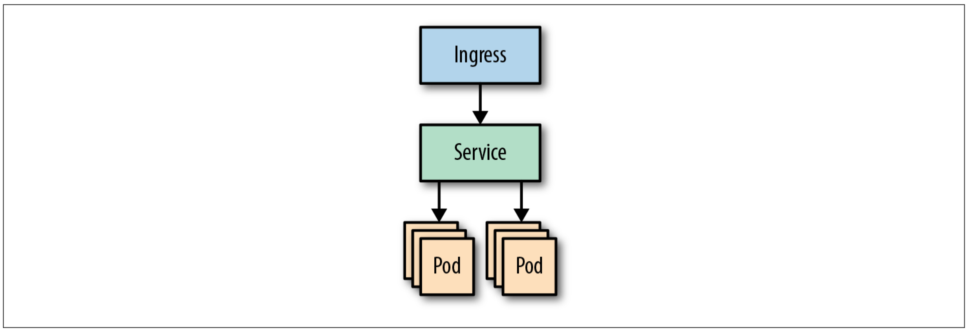


Figure 5-1. The Ingress resource

Here is a very simple example Ingress resource:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: demo-ingress
spec:
  backend:
    serviceName: demo-service
    servicePort: 80
```

This Ingress forwards traffic to a Service named `demo-service` on port 80. (In fact, requests go directly from the Ingress to a suitable Pod, but it's helpful to think of them conceptually as going via the Service.)

By itself, this example doesn't seem very useful. Ingresses can do much more, however.

Ingress Rules

While Services are useful for routing *internal* traffic in your cluster (for example, from one microservice to another), an Ingress is useful for routing *external* traffic into your cluster and to the appropriate microservice.

An Ingress can forward traffic to different services, depending on certain rules that you specify. One common use for this is to route requests to different places, depending on the request URL (known as a *fanout*):

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: fanout-ingress
spec:
  rules:
  - http:
      paths:
      - path: /hello
        backend:
          serviceName: hello
          servicePort: 80
      - path: /goodbye
        backend:
          serviceName: goodbye
          servicePort: 80
```

There are lots of uses for this. Highly available load balancers can be expensive, so with a fanout Ingress, you can have one load balancer (and associated Ingress) route traffic to a large number of services.

You're not just limited to routing based on URLs; you can also use the HTTP `Host` header (equivalent to the practice known as *name-based virtual hosting*). Requests for websites with different domains (such as `example.com`) will be routed to the appropriate backend Service based on the domain.

Terminating TLS with Ingress

In addition, Ingress can handle secure connections using TLS (the protocol formerly known as SSL). If you have lots of different services and applications on the same domain, they can all share a TLS certificate, and a single Ingress resource can manage those connections (known as *TLS termination*):

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: demo-ingress
spec:
  tls:
  - secretName: demo-tls-secret
  backend:
    serviceName: demo-service
    servicePort: 80
```

Here we've added a new `tls` section, which instructs the Ingress to use a TLS certificate to secure traffic with clients. The certificate itself is stored as a Kubernetes Secret resource (see ???).

Using existing TLS certificates

If you have an existing TLS certificate, or you're going to purchase one from a certificate authority, you can use that with your Ingress. Create a Secret that looks like this:

```yaml
apiVersion: v1
kind: Secret
type: kubernetes.io/tls
metadata:
  name: demo-tls-secret
data:
  tls.crt: LS0tLS1CRUdJTiBDRV...LS0tCg==
  tls.key: LS0tLS1CRUdJTiBSU0...LS0tCg==
```

Put the contents of the certificate in the `tls.crt` field, and the key in `tls.key`. As usual with Kubernetes Secrets, you should base64-encode the certificate and key data before adding them to the manifest (see ???).

Automating LetsEncrypt certificates with Cert-Manager

If you want to automatically request and renew TLS certificates using the popular LetsEncrypt authority (or another ACME certificate provider), you can use `cert-manager`.

If you run `cert-manager` in your cluster, it will automatically detect TLS Ingresses that have no certificate, and request one from the specified provider (for example, LetsEncrypt). `cert-manager` is a more modern and capable successor to the popular `kube-lego` tool.

Exactly how TLS connections are handled depends on something called your *Ingress controller*.

Ingress Controllers

An Ingress controller is responsible for managing Ingress resources in a cluster. Depending on where you are running your clusters, the controller you use may vary.

Usually, customizing the behavior of your Ingress is done by adding specific annotations that are recognized by the Ingress controller.

Clusters running on Google's GKE have the option to use Google's Compute Load Balancer for Ingress. AWS has a similar product called an Application Load Balancer. These managed services provide a public IP address where the Ingress will listen for requests.

These are good places to start if you need to use Ingress and are running Kubernetes on Google Cloud or AWS. You can read through the documentation of each product in their respective repositories:

- Google Ingress documentation
- AWS Ingress documentation

You also have the option to install and run your own Ingress controller inside your cluster, or even run multiple controllers if you like. Some popular options include:

nginx-ingress
: NGINX has long been a popular load balancer tool, even before Kubernetes came into the scene. This controller brings much of the functionality and features offered by NGINX to Kubernetes. There are other ingress controllers based on NGINX, but this is the official one.

Contour
: Here is yet another useful Kubernetes tool maintained by the folks at Heptio, whom you will see mentioned several times throughout this book. Contour actually uses another tool under the hood called Envoy to proxy requests between clients and Pods.

Traefik
: This is a lightweight proxy tool that can automatically manage TLS certificates for your Ingress.

Each of these controllers has different features, and comes with its own setup and installation instructions, as well as its own ways of handling things like routes and certificates. Read about the different options and try them out in your own cluster, with your own applications, to get a feel for how they work.

Istio

Istio is an example of what's often referred to as a *service mesh*, and becomes very useful when teams have multiple applications and services that communicate with each other. It handles routing and encrypting network traffic between services, and adds important features like metrics, logs, and load balancing.

Istio is an optional add-on component to many hosted Kubernetes clusters, including Google Kubernetes Engine (check the documentation for your provider to see how to enable Istio).

If you want to install Istio in a self-hosted cluster, use the official Istio Helm chart.

If your applications rely heavily on communicating with each other, Istio may be worth researching. Istio deserves a book of its own, and will probably get one, but in the meantime a great place to start is the introductory documentation.

Envoy

Most managed Kubernetes services, such as Google Kubernetes Engine, provide some kind of cloud load balancer integration. For example, when you create a Service of type `LoadBalancer` on GKE, or an Ingress, a Google Cloud Load Balancer is automatically created and connected to your service.

While these standard cloud load balancers scale well, they are very simple, and don't give you much to configure. For example, the default load balancing algorithm is usually `random` (see "Service Resources" on page 40). This sends each connection to a different backend at random.

However, `random` isn't always what you want. For example, if requests to your service can be long-running and CPU-intensive, some of your backend nodes may get overloaded while others remain idle.

A smarter algorithm would route requests to the backend that is least busy. This is sometimes known as `leastconn` or `LEAST_REQUEST`.

For more sophisticated load balancing like this, you can use a product called Envoy. This is not part of Kubernetes itself, but it's commonly used with Kubernetes applications.

Envoy is a high-performance C++ distributed proxy designed for single services and applications, but can also be used as part of a service mesh architecture (see "Istio" on page 90).

Developer Mark Vincze has written a great blog post detailing how to set up and configure Envoy in Kubernetes.

Summary

Ultimately, everything in Kubernetes is about running Pods. We've gone into some detail about them, therefore, and we apologize if it's too much. You don't need to understand or remember everything we've covered in this chapter, at least for now. Later on, you may run into problems that the more advanced topics in this chapter can help you solve.

The basic ideas to remember:

- Labels are key-value pairs that identify resources, and can be used with selectors to match a specified group of resources.
- Node affinities attract or repel Pods to or from nodes with specified attributes. For example, you can specify that a Pod can only run on a node in a specified availability zone.

- While hard node affinities can block a Pod from running, soft node affinities are more like suggestions to the scheduler. You can combine multiple soft affinities with different weights.
- Pod affinities express a preference for Pods to be scheduled on the same node as other Pods. For example, Pods that benefit from running on the same node can express that using a Pod affinity for each other.
- Pod anti-affinities repel other Pods instead of attracting. For example, an anti-affinity to replicas of the same Pod can help spread your replicas evenly across the cluster.
- Taints are a way of tagging nodes with specific information; usually, about node problems or failures. By default, Pods won't be scheduled on tainted nodes.
- Tolerations allow a Pod to be scheduled on nodes with a specific taint. You can use this mechanism to run certain Pods only on dedicated nodes.
- DaemonSets allow you to schedule one copy of a Pod on every node (for example, a logging agent).
- StatefulSets start and stop Pod replicas in a specific numbered sequence, allowing you to address each by a predictable DNS name. This is ideal for clustered applications, such as databases.
- Jobs run a Pod once (or a specified number of times) before completing. Similarly, Cronjobs run a Pod periodically at specified times.
- Horizontal Pod Autoscalers watch a set of Pods, trying to optimize a given metric (such as CPU utilization). They increase or decrease the desired number of replicas to achieve the specified goal.
- PodPresets can inject bits of common configuration into all selected Pods at creation time. For example, you could use a PodPreset to mount a particular Volume on all matching Pods.
- Custom Resource Definitions (CRDs) allow you to create your own custom Kubernetes objects, to store any data you wish. Operators are Kubernetes client programs that can implement orchestration behavior for your specific application (for example, MySQL).
- Ingress resources route requests to different services, depending on a set of rules, for example, matching parts of the request URL. They can also terminate TLS connections for your application.
- Istio is a tool that provides advanced networking features for microservice applications and can be installed, like any Kubernetes application, using Helm.
- Envoy provides more sophisticated load balancing features than standard cloud load balancers, as well as a service mesh facility.

About the Authors

John Arundel is a consultant with 30 years of experience in the computer industry. He is the author of several technical books, and works with many companies around the world consulting on cloud native infrastructure and Kubernetes. Off the clock, he is a keen surfer, a competitive rifle and pistol shooter, and a decidedly uncompetitive piano player. He lives in a fairytale cottage in Cornwall, England.

Justin Domingus is an operations engineer working in DevOps environments with Kubernetes and cloud technologies. He enjoys the outdoors, learning new things, coffee, crabbing, and computers. He lives in Seattle, Washington, with a wonderful cat and an even more wonderful wife and best friend, Adrienne.

Colophon

The animal on the cover of *Cloud Native DevOps with Kubernetes* is the Ascension frigatebird (*Fregata aquila*), a seabird found only on Ascension Island and nearby Boatswain Bird Island in the South Atlantic Ocean, roughly between Angola and Brazil. The eponymous bird's home island is named after the day it was discovered, Ascension Day on the Christian calendar.

With a wingspan of over 6.5 feet (2 meters) but weighing less than three pounds (1.25 kilograms), the Ascension frigatebird glides effortlessly over the ocean, catching fish that swim near the surface, especially flying fish. It sometimes feeds on squid, baby turtles, and prey stolen from other birds. Its feathers are black and glossy, with hints of green and purple. The male is distinguished by a bright red gular sac, which it inflates when seeking a mate. The female has slightly duller feathers, with patches of brown and sometimes white on the underside. Like other frigatebirds, it has a hooked bill, forked tail, and sharply pointed wings.

The Ascension frigatebird breeds on the rocky outcroppings of its island habitat. Instead of building a nest, it digs a hollow in the ground and protects it with feathers, pebbles, and bones. The female lays a single egg, and both parents care for the chick for six or seven months before it finally learns to fly. Since breeding success is low and habitat is limited, the species is usually classified as Vulnerable.

Ascension Island was first settled by the British for military purposes in the early 19th century. Today the island is home to tracking stations for NASA and the European Space Agency, a relay station for the BBC World Service, and one of the world's four GPS antennas. During most of the 19th and 20th centuries, frigatebirds were limited to breeding on small, rocky Boatswain Bird Island off Ascension's coast, since feral cats were killing their chicks. In 2002, the Royal Society for the Protection of Birds launched a campaign to eliminate the cats from the island, and a few years later, frigatebirds began to nest on Ascension Island once again.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover illustration is by Karen Montgomery, based on a black-and-white engraving from Cassel's Natural History. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.