CHAPTER 2

First Steps with Kubernetes

You've taken your first step into a larger world.

—Obi-Wan Kenobi, *Star Wars: A New Hope*

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! 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. You can find all of the examples on GitHub.

Running Your First Container

As we saw in Chapter 1, the container is one of the key concepts in cloud native development. The most popular tool for building and running containers is Docker. There are other tools for running containers, but we will cover that in more detail later.

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 29, 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 free package for Mac and Windows. It comes with a complete Kubernetes development environment that you can use to test your applications on your laptop or desktop.

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 23.

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 31).

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

```
docker version
 ...
 Version:           20.10.7
 ...
```

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. You can add your account to the docker group with `sudo usermod -aG docker $USER && newgrp docker` and then you won't need to use `sudo` each time.

What Is Docker?

Docker is actually several different, but related, things: a container image format, a container runtime library that 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 supports Docker containers 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
```

Anytime 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 2.32.0
```

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...
...
```

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

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:

```
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)
        fmt.Println("Running demo app. Press Ctrl+C to exit...")
        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.

2 If you're new to Go, Jon Bodner's *Learning Go* (O'Reilly) is an invaluable guide.

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`:

```
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 their 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 OK: Go does!

The rest of the program takes care of registering the `handler` function as the handler for HTTP requests, printing a message that the app is starting, 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:1.17-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 *multistage 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 containers. By comparison, the official Ruby container image is 850 MB; about 140 times bigger than our Alpine Go image, and that's before you've added your Ruby program! Another great resource to look at for using lean containers is *distroless* images, which only contain runtime dependencies and keep your final container image size small.

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:1.17-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 (*almost*) 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 23:

```
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.

We said that you can use *almost* any port earlier because any port number below `1024` is considered a *priviliged* port, meaning that in order to use those ports, your process must run as a user with special permissions, such as `root`. Normal nonadministrator users cannot use ports below 1024, so, to avoid permission issues, we'll stick with higher port numbers in our example.

Container Registries

In "Running a Container Image" on page 23, 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 on Docker Hub 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 client 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 to a registry, 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 3. 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 31 instead). To enable it, open the Docker Desktop Preferences, select the Kubernetes tab, and check Enable. See the Docker Desktop Kubernetes docs for more info.

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

Linux users will also need to install the `kubectl` tool, following the instructions on the Kubernetes Documentation site.

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
pod/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 pod/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                     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.

Once you are done, you'll want to clean up your demo container:

```
kubectl delete pod demo
pod "demo" deleted
```

We'll cover more of the Kubernetes terminology in the coming chapters, but for now you can think of a *Pod* as a container running in Kubernetes, similar to how you ran a Docker container on your computer.

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 the instructions in the official Minikube "Get Started!" guide.

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 Mac or Windows machine. Minikube is another option and works on Linux.
- The `kubectl` tool is the primary way of interacting with a Kubernetes cluster. It can be used to create resources in Kubernetes, view the status of the cluster and Pods, and apply Kubernetes configuration in the form of YAML manifests.