Kubernetes

App Development

GOLDEN GUIDE TO
KUBERNETES APPLICATION DEVELOPMENT

By Matthew Palmer

Table of Contents

Containers

Docker and containers

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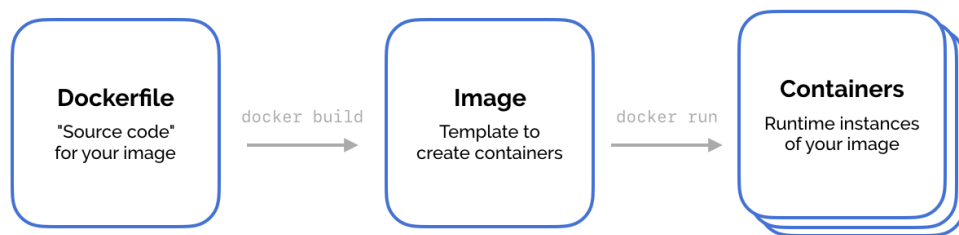
Consider a simple Node.js application that has not been containerized. If you were deploying this on a fresh virtual machine, you'd need to:

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

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

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

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

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

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

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

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

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

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

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

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

# Copy your application source to this directory
COPY . .

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

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

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

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

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

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

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

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

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

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

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

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

Pods

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

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

This is where Kubernetes comes in.

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

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

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

Creating a Kubernetes pod

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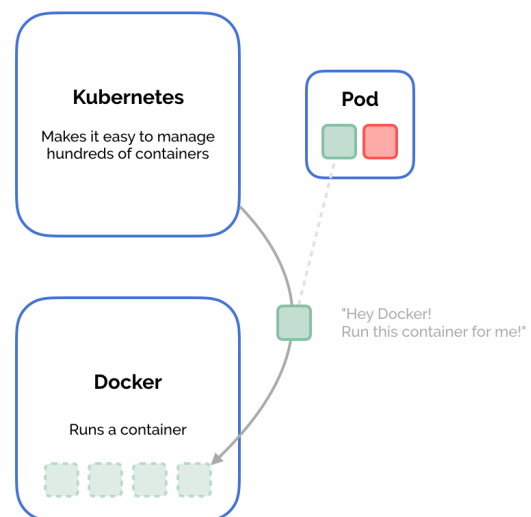
image. This is essentially the same as what we did in the previous chapter. But instead of using `docker run` to create a container running the image, we'll define a pod in Kubernetes that uses the image. Finally, we'll create the pod in Kubernetes, which runs the container for us. Then we'll access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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Common "dummy" images

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

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

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

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