Chapter 2. Creating and Running Containers

Kubernetes is a platform for creating, deploying, and managing distributed applications. These applications come in many different shapes and sizes, but ultimately, they are all comprised of one or more *applications* that run on individual machines. These applications accept input, manipulate data, and then return the results. Before we can even consider building a distributed system, we must first consider how to build the *application container images* that make up the pieces of our distributed system.

Applications are typically comprised of a language runtime, libraries, and your source code. In many cases your application relies on external libraries such as `libc` and `libssl`. These external libraries are generally shipped as shared components in the OS that you have installed on a particular machine.

Problems occur when an application developed on a programmer's laptop has a dependency on a shared library that isn't available when the program is rolled out to the production OS. Even when the development and production environments share the exact same version of the OS, problems can occur when developers forget to include dependent asset files inside a package that they deploy to production.

A program can only execute successfully if it can be reliably deployed onto the machine where it should run. Too often the state of the art for deployment involves running imperative scripts, which inevitably have twisty and Byzantine failure cases.

Finally, traditional methods of running multiple applications on a single machine require that all of these programs share the same versions of shared libraries on the system. If the different applications are developed by different teams or organizations, these shared dependencies add needless complexity and coupling between these teams.

In Chapter 1, we argued strongly for the value of immutable images and infrastructure. It turns out that this is exactly the value provided by the container image. As we will see, it easily solves all the problems of dependency management and encapsulation just described.

When working with applications it’s often helpful to package them in a way that makes it easy to share them with others. Docker, the default container runtime engine, makes it easy to package an application and push it to a remote registry where it can later be pulled by others.

In this chapter we are going to work with a simple example application that we built for this book to help show this workflow in action. You can find the application on GitHub.

Container images bundle an application and its dependencies, under a root filesystem, into a single artifact. The most popular container image format is the Docker image format, the primary image format supported by Kubernetes. Docker images also include additional metadata used by a container runtime to start a running application instance based on the contents of the container image.

This chapter covers the following topics:

- How to package an application using the Docker image format
- How to start an application using the Docker container runtime

Container Images

For nearly everyone, their first interaction with any container technology is with a container image. A *container image* is a binary package that encapsulates all of the files necessary to run an application inside of an OS container. Depending on how you first experiment with containers, you will either build a container image from your local filesystem or download a preexisting image from a *container registry*. In either case, once the container image is present on your computer, you can run that image to produce a running application inside an OS container.

The Docker Image Format

The most popular and widespread container image format is the Docker image format, which was developed by the Docker open source project for packaging, distributing, and running containers using the `docker` command. Subsequently work has begun by Docker, Inc., and others to standardize the container image format via the Open Container Image (OCI) project. While the OCI set of standards have recently (as of mid-2017) been released as a 1.0 standard, adoption of these standards is still very early. The Docker image format continues to be the de facto standard, and is made up of a series of filesystem layers. Each layer adds, removes, or modifies files from the preceding layer in the filesystem. This is an example of an *overlay* filesystem. There are a variety of different concrete implementations of such filesystems, including `aufs`, `overlay`, and `overlay2`.

CONTAINER LAYERING

Container images are constructed of a series of filesystem layers, where each layer inherits and modifies the layers that came before it. To help explain this in detail, let’s build some containers. Note that for correctness the ordering of the layers should be bottom up, but for ease of understanding we take the opposite approach:

```
.
└── container A: a base operating system only, such as Debian
    └── container B: build upon #A, by adding Ruby v2.1.10
    └── container C: build upon #A, by adding Golang v1.6
```

At this point we have three containers: A, B, and C. B and C are *forked* from A and share nothing besides the base container’s files. Taking it further, we can build on top of B by adding Rails (version 4.2.6). We may also want to support a legacy application that requires an older version of Rails (e.g., version 3.2.x). We can build a container image to support that application based on B also, planning to someday migrate the app to v4:

```
. (continuing from above)
└── container B: build upon #A, by adding Ruby v2.1.10
    └── container D: build upon #B, by adding Rails v4.2.6
    └── container E: build upon #B, by adding Rails v3.2.x
```

Conceptually, each container image layer builds upon a previous one. Each parent reference is a pointer. While the example here is a simple set of containers, other real-world containers can be part of a larger and extensive directed acyclic graph.

Container images are typically combined with a container configuration file, which provides instructions on how to set up the container environment and execute an application entrypoint. The container configuration often includes information on how to set up networking, namespace isolation, resource constraints (cgroups), and what `syscall` restrictions should be placed on a running container instance. The container root filesystem and configuration file are typically bundled using the Docker image format.

Containers fall into two main categories:

- System containers
- Application containers

System containers seek to mimic virtual machines and often run a full boot process. They often include a set of system services typically found in a VM, such as `ssh`, `cron`, and `syslog`.

Application containers differ from system containers in that they commonly run a single application. While running a single application per container might seem like an unnecessary constraint, it provides the perfect level of granularity for composing scalable applications, and is a design philosophy that is leveraged heavily by pods.

Building Application Images with Docker

In general, container orchestration systems like Kubernetes are focused on building and deploying distributed systems made up of application containers. Consequently, we will focus on application containers for the remainder of this chapter.

Dockerfiles

A Dockerfile can be used to automate the creation of a Docker container image. The following example describes the steps required to build the `kuard` (Kubernetes up and running) image, which is both secure and lightweight in terms of size:

```
FROM alpine
MAINTAINER Kelsey Hightower <kelsey.hightower@kuar.io>
COPY bin/kuard /kuard
ENTRYPOINT ["/kuard"]
```

This text can be stored in a text file, typically named *Dockerfile,* and used to create a Docker image.

Run the following command to create the `kuard` Docker image:

```
$ docker build -t kuard-amd64:1 .
```

We have chosen to build on top of Alpine, an extremely minimal Linux distribution. Consequently, the final image should check in at around 6 MB, which is drastically smaller than many publicly available images that tend to be built on top of more complete OS versions such as Debian.

At this point our `kuard` image lives in the local Docker registry where the image was built and is only accessible to a single machine. The true power of Docker comes from the ability to share images across thousands of machines and the broader Docker community.

Image Security

When it comes to security there are no shortcuts. When building images that will ultimately run in a production Kubernetes cluster, be sure to follow best practices for packaging and distributing applications. For example, don’t build containers with passwords baked in — and this includes not just in the final layer, but any layers in the image. One of the counterintuitive problems introduced by container layers is that deleting a file in one layer doesn’t delete that file from preceding layers. It still takes up space and it can be accessed by anyone with the right tools — an enterprising attacker can simply create an image that only consists of the layers that contain the password.

Secrets and images should *never* be mixed. If you do so, you will be hacked, and you will bring shame to your entire company or department. We all want to be on TV someday, but there are better ways to go about that.

Optimizing Image Sizes

There are several gotchas that come when people begin to experiment with container images that lead to overly large images. The first thing to remember is that files that are removed by subsequent layers in the system are actually still present in the images; they’re just inaccessible. Consider the following situation:

```
.
└── layer A: contains a large file named 'BigFile'
    └── layer B: removes 'BigFile'
        └── layer C: builds on B, by adding a static binary
```

You might think that *BigFile* is no longer present in this image. After all, when you run the image, it is no longer accessible. But in fact it is still present in layer A, which means that whenever you push or pull the image, *BigFile* is still transmitted through the network, even if you can no longer access it.

Another pitfall that people fall into revolves around image caching and building. Remember that each layer is an independent delta from the layer below it. Every time you change a layer, it changes every layer that comes after it. Changing the preceding layers means that they need to be rebuilt, repushed, and repulled to deploy your image to development.

To understand this more fully, consider two images:

```
.
└── layer A: contains a base OS
    └── layer B: adds source code server.js
        └── layer C: installs the 'node' package
```

versus:

```
.
└── layer A: contains a base OS
    └── layer B: installs the 'node' package
        └── layer C: adds source code server.js
```

It seems obvious that both of these images will behave identically, and indeed the first time they are pulled they do. However, consider what happens when *server.js* changes. In one case, it is only the change that needs to be pulled or pushed, but in the other case, both *server.js* and the layer providing the `node` package need to be pulled and pushed, since the `node` layer is dependent on the *server.js* layer. In general, you want to order your layers from least likely to change to most likely to change in order to optimize the image size for pushing and pulling.

Storing Images in a Remote Registry

What good is a container image if it’s only available on a single machine?

Kubernetes relies on the fact that images described in a pod manifest are available across every machine in the cluster. One option for getting this image to all machines in the cluster would be to export the `kuard` image and import it on every other machine in the Kubernetes cluster. We can’t think of anything more tedious than managing Docker images this way. The process of manually importing and exporting Docker images has human error written all over it. Just say no!

The standard within the Docker community is to store Docker images in a remote registry. There are tons of options when it comes to Docker registries, and what you choose will be largely based on your needs in terms of security requirements and collaboration features.

Generally speaking the first choice you need to make regarding a registry is whether to use a private or a public registry. Public registries allow anyone to download images stored in the registry, while private registries require authentication to download images. In choosing public versus private, it’s helpful to consider your use case.

Public registries are great for sharing images with the world, because they allow for easy, unauthenticated use of the container images. You can easily distribute your software as a container image and have confidence that users everywhere will have the exact same experience.

In contrast, a private repository is best for storing your applications that are private to your service and that you don’t want the world to use.

Regardless, to push an image, you need to authenticate to the registry. You can generally do this with the `docker login` command, though there are some differences for certain registries. In the examples here we are pushing to the Google Cloud Platform registry, called the Google Container Registry (GCR). For new users hosting publicly readable images, the Docker Hub is a great place to start.

Once you are logged in, you can tag the `kuard` image by prepending the target

Docker registry:

```
$ docker tag kuard-amd64:1 gcr.io/kuar-demo/kuard-amd64:1
```

Then you can push the `kuard` image:

```
$ docker push gcr.io/kuar-demo/kuard-amd64:1
```

Now that the `kuard` image is available on a remote registry, it’s time to deploy it using Docker. Because we pushed it to the public Docker registry, it will be available everywhere without authentication.

The Docker Container Runtime

Kubernetes provides an API for describing an application deployment, but relies on a container runtime to set up an application container using the container-specific APIs native to the target OS. On a Linux system that means configuring cgroups and namespaces.

The default container runtime used by Kubernetes is Docker. Docker provides an API for creating application containers on Linux and Windows systems.

Running Containers with Docker

The Docker CLI tool can be used to deploy containers. To deploy a container from the `gcr.io/kuar-demo/kuard-amd64:1` image, run the following command:

```
$ docker run -d --name kuard \
  --publish 8080:8080 \
  gcr.io/kuar-demo/kuard-amd64:1
```

This command starts the `kuard` database and maps ports 8080 on your local machine to 8080 in the container. This is because each container gets its own IP address, so listening on *localhost* inside the container doesn’t cause you to listen on your machine. Without the port forwarding, connections will be inaccessible to your machine.

Exploring the kuard Application

`kuard` exposes a simple web interface, which can be loaded by pointing your browser at *http://localhost:8080* or via the command line:

```
$ curl http://localhost:8080
```

`kuard` also exposes a number of interesting functions that we will explore later on in this book.

Limiting Resource Usage

Docker provides the ability to limit the amount of resources used by applications by exposing the underlying cgroup technology provided by the Linux kernel.

Limiting memory resources

One of the key benefits to running applications within a container is the ability to restrict resource utilization. This allows multiple applications to coexist on the same hardware and ensures fair usage.

To limit `kuard` to 200 MB of memory and 1 GB of swap space, use the `--memory` and `--memory-swap` flags with the `docker run` command.

Stop and remove the current `kuard` container:

```
$ docker stop kuard
$ docker rm kuard
```

Then start another `kuard` container using the appropriate flags to limit memory usage:

```
$ docker run -d --name kuard \
  --publish 8080:8080 \
  --memory 200m \
  --memory-swap 1G \
  gcr.io/kuar-demo/kuard-amd64:1
```

Limiting CPU resources

Another critical resource on a machine is the CPU. Restrict CPU utilization using the `--cpu-shares` flag with the `docker run` command:

```
$ docker run -d --name kuard \
  --publish 8080:8080 \
  --memory 200m \
  --memory-swap 1G \
  --cpu-shares 1024 \
  gcr.io/kuar-demo/kuard-amd64:1
```

Cleanup

Once you are done building an image, you can delete it with the `docker rmi` command:

```
docker rmi <tag-name>
```

or

```
docker rmi <image-id>
```

Images can either be deleted via their tag name (e.g., `gcr.io/kuar-demo/kuard-amd64:1`) or via their image ID. As with all ID values in the `docker` tool, the image ID can be shortened as long as it remains unique. Generally only three or four characters of the ID are necessary.

It's important to note that unless you explicitly delete an image it will live on your system forever, *even* if you build a new image with an identical name. Building this new image simply moves the tag to the new image; it doesn't delete or replace the old image.

Consequently, as you iterate while you are creating a new image, you will often create many, many different images that end up taking up unnecessary space on your computer.

To see the images currently on your machine, you can use the `docker images` command. You can then delete tags you are no longer using.

A slightly more sophisticated approach is to set up a `cron` job to run an image garbage collector. For example, the `docker-gc` tool is a commonly used image garbage collector that can easily run as a recurring `cron` job, once per day or once per hour, depending on how many images you are creating.

Summary

Application containers provide a clean abstraction for applications, and when packaged in the Docker image format, applications become easy to build, deploy, and distribute. Containers also provide isolation between applications running on the same machine, which helps avoid dependency conflicts. The ability to mount external directories means we can run not only stateless applications in a container, but also applications like `influxdb` that generate lots of data.