

- Deploying Docker containers in AWS using Elastic Kubernetes Service (EKS)

A Crash Course on Docker

As you may remember from [Chapter 1](#), Docker images are like self-contained “snapshots” of the operating system (OS), the software, the files, and all other relevant details. Let’s now see Docker in action.

First, if you don’t have Docker installed already, follow the instructions on the [Docker website](#) to install Docker Desktop for your operating system. Once it’s installed, you should have the `docker` command available on your command line. You can use the `docker run` command to run Docker images locally:

```
$ docker run <IMAGE> [COMMAND]
```

where `IMAGE` is the Docker image to run and `COMMAND` is an optional command to execute. For example, here’s how you can run a Bash shell in an Ubuntu 20.04 Docker image (note that the following command includes the `-it` flag so you get an interactive shell where you can type):

```
$ docker run -it ubuntu:20.04 bash
Unable to find image 'ubuntu:20.04' locally
20.04: Pulling from library/ubuntu
Digest:
sha256:669e010b58baf5beb2836b253c1fd5768333f0d1dbc834f7c07a4dc93
f474be
Status: Downloaded newer image for ubuntu:20.04
root@d96ad3779966:/#
```

And voilà, you’re now in Ubuntu! If you’ve never used Docker before, this can seem fairly magical. Try running some commands. For example, you can look at the contents of `/etc/os-release` to verify you really are in Ubuntu:

```
root@d96ad3779966:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="20.04.3 LTS (Focal Fossa)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 20.04.3 LTS"
VERSION_ID="20.04"
VERSION_CODENAME=focal
```

How did this happen? Well, first, Docker searches your local filesystem for the `ubuntu:20.04` image. If you don't have that image downloaded already, Docker downloads it automatically from Docker Hub, which is a *Docker Registry* that contains shared Docker images. The `ubuntu:20.04` image happens to be a public Docker image—an official one maintained by the Docker team—so you're able to download it without any authentication. However, it's also possible to create private Docker images that only certain authenticated users can use.

Once the image is downloaded, Docker runs the image, executing the `bash` command, which starts an interactive Bash prompt, where you can type. Try running the `ls` command to see the list of files:

```
root@d96ad3779966:/# ls -al
total 56
drwxr-xr-x  1 root root 4096 Feb 22 14:22 .
drwxr-xr-x  1 root root 4096 Feb 22 14:22 ..
lrwxrwxrwx  1 root root    7 Jan 13 16:59 bin -> usr/bin
drwxr-xr-x  2 root root 4096 Apr 15 2020 boot
drwxr-xr-x  5 root root  360 Feb 22 14:22 dev
drwxr-xr-x  1 root root 4096 Feb 22 14:22 etc
drwxr-xr-x  2 root root 4096 Apr 15 2020 home
lrwxrwxrwx  1 root root    7 Jan 13 16:59 lib -> usr/lib
drwxr-xr-x  2 root root 4096 Jan 13 16:59 media
(...)
```

You might notice that's not your filesystem. That's because Docker images run in containers that are isolated at the userspace level: when you're in a container, you can only see the filesystem, memory, networking, etc., in that container. Any data in other containers, or on the underlying host operating system, is not accessible to you, and any data in your container is not

visible to those other containers or the underlying host operating system. This is one of the things that makes Docker useful for running applications: the image format is self-contained, so Docker images run the same way no matter where you run them, and no matter what else is running there.

To see this in action, write some text to a *test.txt* file as follows:

```
root@d96ad3779966:/# echo "Hello, World!" > test.txt
```

Next, exit the container by hitting Ctrl-D on Windows and Linux or Cmd-D on macOS, and you should be back in your original command prompt on your underlying host OS. If you try to look for the *test.txt* file you just wrote, you'll see that it doesn't exist: the container's filesystem is totally isolated from your host OS.

Now, try running the same Docker image again:

```
$ docker run -it ubuntu:20.04 bash  
root@3e0081565a5d:/#
```

Notice that this time, since the *ubuntu:20.04* image is already downloaded, the container starts almost instantly. This is another reason Docker is useful for running applications: unlike virtual machines, containers are lightweight, boot up quickly, and incur little CPU or memory overhead.

You may also notice that the second time you fired up the container, the command prompt looked different. That's because you're now in a totally new container; any data you wrote in the previous one is no longer accessible to you. Run *ls -al* and you'll see that the *test.txt* file does not exist. Containers are isolated not only from the host OS but also from each other.

Hit Ctrl-D or Cmd-D again to exit the container, and back on your host OS, run the *docker ps -a* command:

\$ docker ps -a	CONTAINER ID	IMAGE	COMMAND	CREATED
-----------------	--------------	-------	---------	---------

```
STATUS
3e0081565a5d    ubuntu:20.04      "bash"        5 min ago   Exited
(0) 16 sec ago
d96ad3779966    ubuntu:20.04      "bash"        14 min ago  Exited
(0) 5 min ago
```

This will show you all the containers on your system, including the stopped ones (the ones you exited). You can start a stopped container again by using the `docker start <ID>` command, setting `ID` to an ID from the `CONTAINER ID` column of the `docker ps` output. For example, here is how you can start the first container up again (and attach an interactive prompt to it via the `-ia` flags):

```
$ docker start -ia d96ad3779966
root@d96ad3779966:/#
```

You can confirm this is really the first container by outputting the contents of `test.txt`:

```
root@d96ad3779966:/# cat test.txt
Hello, World!
```

Let's now see how a container can be used to run a web app. Hit Ctrl-D or Cmd-D again to exit the container, and back on your host OS, run a new container:

```
$ docker run training/webapp
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

The **training/webapp image** contains a simple Python “Hello, World” web app for testing. When you run the image, it fires up the web app, listening on port 5000 by default. However, if you open a new terminal on your host operating system and try to access the web app, it won’t work:

```
$ curl localhost:5000
curl: (7) Failed to connect to localhost port 5000: Connection refused
```