3

Containerization with Docker

In the previous chapter, we talked about source code management with Git, where we took a crash course on Git and then discussed GitOps and how it shapes modern DevOps practices.

In this chapter, we'll get hands-on and explore **Docker** – the de facto container runtime. By the end of this chapter, you should be able to install and configure Docker, run your first container, and then monitor it. This chapter will also form the basis for the following chapters, as we will use the same setup for the demos later.

In this chapter, we're going to cover the following main topics:

- Installing tools
- Installing Docker
- Introducing Docker storage drivers and volumes
- Running your first container
- Docker logging and logging drivers
- Docker monitoring with Prometheus
- Declarative container management with Docker Compose

Technical requirements

For this chapter, you will need a Linux machine running Ubuntu 18.04 Bionic LTS or later with sudo access. We will be using Ubuntu 22.04 Jammy Jellyfish for the entirety of this book, but feel free to use any OS of your choice. I will post links to alternative installation instructions.

You will also need to clone the following GitHub repository for some of the exercises: `https://github.com/PacktPublishing/Modern-DevOps-Practices-2e`.

We discussed Git extensively in the previous chapter; therefore, you can easily clone the repository using that knowledge. Now, let's move on to installing Docker on your machine.

Installing Docker

We will be installing Docker in an Ubuntu system. For other OSs, please refer to `https://docs.docker.com/engine/install/`.

To install Docker, we need to install supporting tools to allow the `apt` package manager to download Docker through HTTPS. Let's do so using the following commands:

```
$ sudo apt-get update
$ sudo apt-get install -y ca-certificates curl gnupg
```

Download the Docker gpg key and add it to the apt package manager:

```
$ sudo install -m 0755 -d /etc/apt/keyrings
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
$ sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

Then, you need to add the Docker repository to your `apt` configuration so that you can download packages from there:

```
$ echo \
  "deb [arch="$(dpkg --print-architecture)" \
signed-by=/etc/apt/keyrings/docker.gpg] \
https://download.docker.com/linux/ubuntu \
"$(. /etc/os-release && echo "$VERSION_CODENAME")" \
stable" | sudo tee /etc/apt/sources.list.d/docker.list \
> /dev/null
```

Finally, install the Docker engine by using the following commands:

```
$ sudo apt-get update
$ sudo apt-get -y install docker-ce docker-ce-cli \
containerd.io docker-buildx-plugin docker-compose-plugin
```

To verify whether Docker has been installed successfully, run the following:

```
$ sudo docker --version
```

You should expect a similar output to the following:

```
Docker version 24.0.2, build cb74dfc
```

The next thing you should do is allow regular users to use Docker. You want your users to act as something other than root for building and running containers. To do that, run the following command:

```
$ sudo usermod -a -G docker <username>
```

To apply the changes to your profile, log out from your virtual machine and log back in.

Now that Docker has been fully set up on your machine, let's run a `hello-world` container to see this for ourselves:

```
$ docker run hello-world
```

You should see the following output:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
719385e32844: Pull complete
Digest: sha256:fc6cf906cbfa013e80938cdf0bb199fbdbb86d6e3e013783e5a766f50f5dbce0
Status: Downloaded newer image for hello-world:latest
Hello from Docker!
```

You will also receive the following message, which tells you what happened behind the scenes to print the `Hello from Docker!` message on your screen:

```
This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the hello-world image from Docker Hub.(amd64).
 3. The Docker daemon created a new container from that image that runs the executable
that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your
Terminal:
To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash
Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/
For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

All this helpful information is self-explanatory. To explain Docker Hub a bit, it is a public Docker container registry that hosts many Docker images for people like you and me to consume.

As Docker works on a layered architecture, most Docker images are derived from one or more base images hosted on Docker Hub. So, please create a Docker Hub account for yourself to host your containers and share them with the rest of the world.

Most organizations might want to keep their images private, so you have the option of creating private repositories within Docker Hub. You can also host your own internal Docker registry using a SaaS service such as **Google Container Registry** (**GCR**), or installing an artifact repository such as **Sonatype Nexus** or **JFrog Artifactory**. Whatever your choice of tool, the mechanism and how it works always remain the same.

Introducing Docker storage drivers and volumes

Docker containers are ephemeral workloads. Whatever data you store on your container filesystem gets wiped out once the container is gone. The data lives on a disk during the container's life cycle but does not persist beyond it. Pragmatically speaking, most applications in the real world are stateful. They need to store data beyond the container life cycle and want it to persist.

So, how do we go along with that? Docker provides several ways you can store data. By default, all data is stored on the writable container layer, which is ephemeral. The writable container layer interacts with the host filesystem via a storage driver. Because of the abstraction, writing files to the container layer is slower than writing directly to the host filesystem.

To solve that problem and also provide persistent storage, Docker provides volumes, bind mounts, and `tmpfs`. With them, you can interact directly with the host filesystem (and memory in the case of `tmpfs`) and save a ton of **I/O operations per second** (**IOPS**), improving performance. While this section focuses on storage drivers that cater to the container filesystem, it is worth discussing multiple data storage options within Docker to provide a background.

Docker data storage options

Every option has a use case and trade-off. Let's look at each option and where you should use which.

Volumes

Docker volumes store the data directly in the host's filesystem. They do not use the storage driver layer in between, so writing to volumes is faster. They are the best way to persist data. Docker stores volumes in `/var/lib/docker/volumes` and assumes that no one apart from the Docker daemon can modify the data on them.

As a result, volumes provide the following features:

- Provide some isolation with the host filesystems. If you don't want other processes to interact with the data, then a volume should be your choice.
- You can share a volume with multiple containers.
- Volumes can either be named or anonymous. Docker stores anonymous volumes in a directory with a unique random name.
- Volumes enable you to store data remotely or in a cloud provider using volume drivers. This helps a lot if multiple containers share the same volume to provide a multi-instance active-active configuration.
- The data in the volume persists even when the containers are deleted.

Now, let's look at another storage option – bind mounts.

Bind mounts

Bind mounts are very similar to volumes but with a significant difference: they allow you to mount an existing host directory as a filesystem on the container. This lets you share important files with the Docker container, such as `/etc/resolv.conf`.

Bind mounts also allow multiple processes to modify data along with Docker. So, if you are sharing your container data with another application that is not running in Docker, bind mounts are the way to go.

tmpfs mounts

`tmpfs` mounts store data in memory. They do not store any data on disk – neither the container nor the host filesystem. You can use them to store sensitive information and the non-persistent state during the lifetime of your container.

Mounting volumes

If you mount a host directory that already contains files to an empty volume of the container, the container can see the files stored in the host. This is an excellent way to pre-populate files for your container(s) to use. However, if the directory does not exist in the host filesystem, Docker will create the directory automatically. If the volume is non-empty and the host filesystem already contains files, Docker will obscure the mount. This means that while you won't see the original files while the Docker volume is mounted to it, the files are not deleted, and you can recover them by unmounting the Docker volume.

We'll look at Docker storage drivers in the next section.

Docker storage drivers

There are numerous storage driver types. Some of the most popular ones are as follows:

- `overlay2`: This is a production-ready driver and is the preferred storage choice for Docker. It works in most environments.
- `devicemapper`: This was the preferred driver for devices running RHEL and CentOS 7 and below that did not support `overlay2`. You can use this driver if you have write-intensive activities in your containers.
- `btrfs` and `zfs`: These drivers are write-intensive and provide many features, such as allowing snapshots, and can only be used if you are using `btrfs` or `zfs` filesystems within your host.
- `vfs`: This storage driver should be used only if no copy-on-write filesystem is available. It is extremely slow, and you should refrain from using it in production.

Let's concentrate on two of the most popular ones – `overlay2` and `devicemapper`.

overlay2

`overlay2` is the default and recommended storage driver in most operating systems except RHEL 7 and CentOS 7 and older. They use file-based storage and perform best when subject to more reads than writes.

devicemapper

`devicemapper` is block-based storage and performs the best when subject to more writes than reads. Though it is compatible and the default with CentOS 7, RHEL 7, and below, as they don't support `overlay2`, it is currently not recommended in the newer versions of these operating systems that do support `overlay2`.

> **Tip**
> Use `overlay2` where possible, but if you have a specific use case for not using it (such as too many write-intensive containers), `devicemapper` is a better choice.

Configuring a storage driver

For this discussion, we will configure `overlay2` as the storage driver. Although it is configured by default, and you can skip the steps if you are following this book, it is worth a read in case you want to change it to something else.

First, let's list the existing storage driver:

```
$ docker info | grep 'Storage Driver'
Storage Driver: overlay2
```

We can see that the existing storage driver is already `overlay2`. Let's learn how to change it to `devicemapper` if we had to.

Edit the `/etc/docker/daemon.json` file using an editor of your choice. If you're using `vim`, run the following command:

```
$ sudo vim /etc/docker/daemon.json
```

Add the `storage-driver` entry to the `daemon.json` configuration file:

```
{
  "storage-driver": "devicemapper"
}
```

Then, restart the Docker service:

```
$ sudo systemctl restart docker
```

Check the status of the Docker service:

```
$ sudo systemctl status docker
```

Now, rerun `docker info` to see what we get:

```
$ docker info | grep 'Storage Driver'
Storage Driver: devicemapper
WARNING: The devicemapper storage-driver is deprecated, and will be removed in a future
release.
         Refer to the documentation for more information: https://docs.docker.com/go/
storage-driver/
WARNING: devicemapper: usage of loopback devices is strongly discouraged for production
use.
         Use `--storage-opt dm.thinpooldev` to specify a custom block storage device.
```

Here, we can see the `devicemapper` storage driver. We can also see several warnings with it that say that the `devicemapper` storage driver is deprecated and will be removed in a future version.

Therefore, we should stick with the defaults unless we have a particular requirement.

So, let's roll back our changes and set the storage driver to `overlay2` again:

```
$ sudo vim /etc/docker/daemon.json
```

Modify the `storage-driver` entry in the `daemon.json` configuration file to `overlay2`:

```
{
  "storage-driver": "overlay2"
}
```

Then, restart the Docker service and check its status:

```
$ sudo systemctl restart docker
$ sudo systemctl status docker
```

If you rerun `docker info`, you will see the storage driver as `overlay2`, and all the warnings will disappear:

```
$ docker info | grep 'Storage Driver'
Storage Driver: overlay2
```

> **Tip**
>
> Changing the storage driver will wipe out existing containers from the disk, so exercise caution when you do so and take appropriate downtimes if you're doing this in production. You will also need to pull images again since local images will fail to exist.

Now that we have installed Docker on our machine and configured the right storage driver, it's time to run our first container.

Running your first container

You can create Docker containers out of Docker container images. While we will discuss container images and their architecture in the following chapters, an excellent way to visualize them is as a copy of all files, application libraries, and dependencies comprising your application environment, similar to a virtual machine image.

To run a Docker container, we can use the `docker run` command, which has the following structure:

```
$ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

Let's look at the `docker run` command and its variations using working examples.

In its simplest form, you can use `docker run` by simply typing the following:

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:e7c70bb24b462baa86c102610182e3efcb12a04854e8c582
838d92970a09f323
Status: Downloaded newer image for hello-world:latest
Hello from Docker!
...
```

As you may recall, we used this command when we installed Docker. Here, I have purposefully omitted `tag`, `options`, `command`, and `arguments`. We will cover it with multiple examples to show its actual use cases.

As we didn't supply `tag`, Docker automatically assumed the `tag` as `latest`, so if you look at the command output, you will see that Docker is pulling the `hello-world:latest` image from Docker Hub.

Now, let's look at an example with a specific version tag.

Running containers from versioned images

We can run `nginx:1.18.0` using the following command:

```
$ docker run nginx:1.18.0
Unable to find image 'nginx:1.18.0' locally
1.18.0: Pulling from library/nginx
852e50cd189d: Pull complete
48b8657f2521: Pull complete
b4f4d57f1a55: Pull complete
d8fbe49a7d55: Pull complete
04e4a40fabc9: Pull complete
Digest: sha256:2104430ec73de095df553d0c7c2593813e01716a48d66f
85a3dc439e050919b3
Status: Downloaded newer image for nginx:1.18.0
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform
```

```
configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-
listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-
envsubst-on-templates.sh
/docker-entrypoint.sh: Configuration complete; ready for
start up
```

Note that the prompt will be stuck after this. There is a reason for this: `nginx` is a long-running process, also known as a daemon. Since NGINX is a web server that needs to listen to HTTP requests continuously, it should never stop. In the case of the `hello-world` application, its only job was to print the message and exit. NGINX has a different purpose altogether.

Now, no one would keep a Bash session open for a web server to run, so there has to be some way to run it in the background. You can run containers in the detached mode for that. We'll have a look at this in the next section.

Running Docker containers in the background

To run a Docker container in the background as a daemon, you can use `docker run` in detached mode using the `-d` flag:

```
$ docker run -d nginx:1.18.0
beb5dfd529c9f001539c555a18e7b76ad5d73b95dc48e8a35aecd7471ea938fc
```

As you can see, it just prints a random ID and provides control back to the shell.

Troubleshooting containers

To see what's going on within the container, you can use the `docker logs` command. But before using that, we need to know the container's ID or name to see the container's logs.

To get a list of containers running within the host, run the following command

```
$ docker ps
CONTAINER ID  IMAGE         COMMAND        CREATED        STATUS        PORTS   NAMES
beb5dfd529c9  nginx:1.18.0  "/docker-      2 minutes ago  Up 2 minutes  80/tcp  fervent_
                            entrypoint.…"                                       shockley
```

The preceding command lists the NGINX container that we just started. Unless you specify a particular name for your container, Docker allocates a random name to it. In this case, it has called it `fervent_shockley`. It also assigns every container a unique container ID, such as `beb5dfd529c9`.

You can use the container ID or the container name to interact with the container to list the logs. Let's use the container ID this time:

```
$ docker logs beb5dfd529c9
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform
configuration
...
/docker-entrypoint.sh: Configuration complete; ready for start up
```

As you can see, it prints a similar log output as it did when we ran it in the foreground.

Practically speaking, you will use `docker logs` 90% of the time unless you need to debug something with BusyBox. BusyBox is a lightweight shell container that can help you troubleshoot and debug issues with your container – mostly network issues.

Let's make BusyBox echo `Hello World!` for us:

```
$ docker run busybox echo 'Hello World!'
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
325d69979d33: Pull complete
Digest: sha256:560af6915bfc8d7630e50e212e08242d37b63bd5c1ccf9bd4acccf116e262d5b
Status: Downloaded newer image for busybox:latest
Hello World!
```

As we can see, Docker pulls the latest `busybox` image from Docker Hub and runs the `echo 'Hello World'` command.

You can also use BusyBox in interactive mode by using the `-it` flag, which will help you run a series of commands on the BusyBox shell. It is also a good idea to add an `--rm` flag to it to tell Docker to clean up the containers once we have exited from the shell, something like this:

```
$ docker run -it --rm busybox /bin/sh
/ # echo 'Hello world!'
Hello world!
/ # wget http://example.com
Connecting to example.com (93.184.216.34:80)
saving to 'index.html'
index.html           100% |********************************
****|  1256  0:00:00 ETA
'index.html' saved
/ # exit
```

Upon listing all the containers, we do not see the `busybox` container in there:

```
$ docker ps -a
CONTAINER ID  IMAGE   COMMAND        CREATED     STATUS    PORTS   NAMES
beb5dfd529c9  nginx:  "/docker-      17 minutes  Up 17     80/tcp  fervent_
              1.18.0  entrypoint.…"  ago         minutes           shockley
```

There are various other flags that you can use with your containers, each serving a specific purpose. Let's look at a few common ones.

Putting it all together

The best setting for a highly available NGINX container should be something like the following:

```
$ docker run -d --name nginx --restart unless-stopped \
-p 80:80 --memory 1000M --memory-reservation 250M nginx:1.18.0
```

Let's take a look at this in more detail:

- `-d`: Run as a daemon in detached mode.
- `--name nginx`: Give the name `nginx`.
- `--restart unless-stopped`: Always automatically restart on failures unless explicitly stopped manually, and also start automatically on Docker daemon startup. Other options include `no`, `on_failure`, and `always`.
- `-p 80:80`: Forward traffic from host port `80` to container port `80`. This allows you to expose your container to your host network.
- `--memory 1000M`: Limit the container memory consumption to `1000M`. If the memory exceeds this limit, the container stops and acts according to the `--restart` flag.
- `--memory-reservation 250M`: Allocate a soft limit of `250M` memory to the container if the server runs out of memory.

We will look into other flags in the subsequent sections as we get more hands-on.

> **Tip**
> Consider using `unless-stopped` instead of `always` as it allows you to stop the container manually if you want to do some maintenance.

Now, let's list the containers and see what we get:

```
$ docker ps -a
CONTAINER ID  IMAGE   COMMAND          CREATED     STATUS   PORTS       NAMES
06fc749371b7  nginx   "/docker-        17 seconds  Up 16    0.0.0.0:    nginx
                      entrypoint.…"    ago         seconds  80->80/tcp
beb5dfd529c9  nginx:  "/docker-        22 minutes  Up 22    80/tcp      fervent_shockley
              1.18.0  entrypoint.…"    ago         minutes
```

If you look carefully, you'll see a container called `nginx` and a port forward from `0.0.0.0:80 -> 80`.

Now, let's `curl` on `localhost:80` on the host to see what we get:

```
$ curl localhost:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
</html>
```

We get the NGINX welcome message. This means NGINX is running successfully, and we can access it from the machine. If you have exposed your machine's port `80` to the external world, you can also access this using your browser as follows:

Figure 3.1 – The NGINX welcome page

You also might want to restart or remove your container occasionally. We'll look at ways to do that in the next section.

Restarting and removing containers

To restart your containers, you must stop them first and then start them.

To stop your container, run the following:

```
$ docker stop nginx
```

To start your container, run the following:

```
$ docker start nginx
```

If you want to get rid of your container completely, you need to stop your container first and then remove it, using the following command:

```
$ docker stop nginx && docker rm nginx
```

Alternatively, you can use the following command to do it in one go:

```
$ docker rm -f nginx
```

Now, let's look at how we can monitor our containers with tools such as `journald` and Splunk.

Docker logging and logging drivers

Docker not only changed how applications are deployed but also the workflow for log management. Instead of writing logs to files, containers write logs to the console (`stdout/stderr`). Docker then uses a logging driver to export container logs to the specified destinations.

Container log management

Log management is an essential function within Docker, as with any application. However, due to the transient nature of Docker workloads, it becomes more critical as we lose the filesystem and potentially logs when the container is deleted or faces any issue. So, we should use log drivers to export the logs into a particular place and store and persist it. If you have a log analytics solution, the best place for your logs to be is within it. Docker supports multiple log targets via logging drivers. Let's have a look.

Logging drivers

At the time of writing, the following logging drivers are available:

- `none`: No logs are available for the container, and therefore they are not stored anywhere.
- `local`: Logs are stored locally in a custom format, which minimizes overhead.
- `json-file`: The log files are stored in JSON format. This is the default Docker logging driver.
- `syslog`: This driver uses `syslog` for storing the Docker logs as well. This option makes sense when you use `syslog` as your default logging mechanism.
- `journald`: Uses `journald` to store Docker logs. You can use the `journald` command line to browse the container and the Docker daemon logs.
- `gelf`: Sends logs to a **Graylog Extended Log Format** (**GELF**) endpoint such as Graylog or Logstash.
- `fluentd`: Sends logs to Fluentd.
- `awslogs`: Sends logs to AWS CloudWatch.
- `splunk`: Sends logs to Splunk using the HTTP Event Collector.
- `etwlogs`: Sends logs to **Event Tracing for Windows** (**ETW**) events. You can only use it on Windows platforms.
- `gcplogs`: Sends logs to Google Cloud Logging.
- `logentries`: Sends logs to Rapid7 Logentries.

While all these are viable options, we will look at `journald` and Splunk. While `journald` is a native operating system service monitoring option, Splunk is one of the most famous log analytics and monitoring tools. Now, let's understand how to configure a logging driver.

Configuring logging drivers

Let's start by finding the current logging driver:

```
$ docker info | grep "Logging Driver"
Logging Driver: json-file
```

Currently, the default logging driver is set to `json-file`. If we want to use `journald` or Splunk as the default logging driver, we must configure the default logging driver in the `daemon.json` file.

Edit the `/etc/docker/daemon.json` file using an editor of your choice. If you're using `vim`, run the following command:

```
$ sudo vim /etc/docker/daemon.json
```

Add the `log-driver` entry to the `daemon.json` configuration file:

```
{
  "log-driver": "journald"
}
```

Then, restart the Docker service:

```
$ sudo systemctl restart docker
```

Check the status of the Docker service:

```
$ sudo systemctl status docker
```

Now, rerun `docker info` to see what we get:

```
$ docker info | grep "Logging Driver"
Logging Driver: journald
```

Now that `journald` is the default logging driver, let's launch a new NGINX container and visualize the logs:

```
$ docker run --name nginx-journald -d nginx
66d50cc11178b0dcdb66b114ccf4aa2186b510eb1fdb1e19d563566d2e96140c
```

Now, let's look at the `journald` logs to see what we get:

```
$ sudo journalctl CONTAINER_NAME=nginx-journald
...
```

```
Jun 01 06:11:13 99374c32101c fb8294aece02[10826]: 10-listen-on-ipv6-by-default.sh: info:
Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
...
Jun 01 06:11:13 99374c32101c fb8294aece02[10826]: 2023/06/01 06:11:13 [notice] 1#1: start
worker process 30
...
```

We can see the logs in the journal.

Similarly, we can configure the Splunk logging driver to send data to Splunk for analytics and visualization. Let's have a look.

Edit the `/etc/docker/daemon.json` file using an editor of your choice. If you're using `vim`, run the following command:

```
$ vim /etc/docker/daemon.json
```

Add the log-driver entry to the `daemon.json` configuration file:

```
{
  "log-driver": "splunk",
  "log-opts": {
    "splunk-token": "<Splunk HTTP Event Collector token>",
    "splunk-url": "<Splunk HTTP(S) url>"
  }
}
```

Then, restart the Docker service:

```
$ sudo systemctl restart docker
```

Check the status of the Docker service:

```
$ sudo systemctl status docker
```

Now, rerun `docker info` to see what we get:

```
$ docker info | grep "Logging Driver"
Logging Driver: splunk
```

Since Splunk is now the default logging driver, let's launch a new NGINX container and visualize the logs:

```
$ docker run --name nginx-splunk -d nginx
dedde062feba33f64efd89ef9102c7c93afa854473cda3033745d35d9065c9e5
```

Log in to your Splunk instance; you will see the Docker logs streaming. You can then analyze the logs and create visualizations out of them.

You can also have different logging drivers for different containers, and you can do so by overriding the defaults by passing the `log-driver` and `log-opts` flags from the command line. As our current

configuration is Splunk, and we want to export data to a JSON file, we can specify `log-driver` as `json-file` while running the container. Let's have a look:

```
$ docker run --name nginx-json-file --log-driver json-file -d nginx
379eb8d0162d98614d53ae1c81ea1ad154745f9edbd2f64cffc2279772198bb2
```

To visualize JSON logs, we need to look into the JSON log directory – that is, `/var/lib/docker/containers/<container_id>/<container_id>-json.log`.

For the `nginx-json-file` container, we can do the following:

```
$ cat /var/lib/docker/containers\
/379eb8d0162d98614d53ae1c81ea1ad154745f9edbd2f64cffc2279772198bb2\
/379eb8d0162d98614d53ae1c81ea1ad154745f9edbd2f64cffc2279772198bb2-json.log
{"log":"/docker-entrypoint.sh: /docker-entrypoint.d/ is not
empty, will attempt to perform configuration\n","stream":"
stdout","time":"2022-06-01T06:27:05.922950436Z"}
...
{"log":"/docker-entrypoint.sh: Configuration complete; ready
for start up\n","stream":"stdout","time":"2023-06-01T06:27:
05.937629749Z"}
```

We can see that the logs are now streaming to the JSON file instead of Splunk. That is how we override the default log driver.

> **Tip**
>
> In most cases, it is best to stick with one default logging driver so that you have one place to analyze and visualize your logs.

Now, let's understand some of the challenges and best practices associated with Docker logging.

Typical challenges and best practices to address these challenges with Docker logging

Docker allows you to run multiple applications in a single machine or a cluster of machines. Most organizations run a mix of virtual machines and containers, and they have their logging and monitoring stack configured to support virtual machines.

Most teams struggle to make Docker logging behave the way virtual machine logging works. So, most teams will send logs to the host filesystem, and the log analytics solution then consumes the data from there. This is not ideal, and you should avoid making this mistake. It might work if your container is static, but it becomes an issue if you have a cluster of servers, each running Docker, and you can schedule your container in any virtual machine you like.

So, treating a container as an application running on a virtual machine is a mistake from a logging point of view. Instead, you should visualize the container as an entity – just like a virtual machine. It would be best if you never associated containers with a virtual machine.

One solution is to use the logging driver to forward the logs to a log analytics solution directly. But then, the logging becomes heavily dependent on the availability of the log analytics solution. So, it might not be the best thing to do. People faced issues when their services running on Docker went down because the log analytics solution was unavailable or there were network issues.

Well, the best way to approach this problem is to use JSON files to store the logs temporarily in your virtual machine and use another container to push the logs to your chosen log analytics solution the old-fashioned way. That way, you decouple from the dependency on an external service to run your application.

You can use the logging driver to export logs directly to your log analytics solution within the log forwarder container. There are many logging drivers available that support many log targets. Always mark the logs in such a way that the containers appear as their own entities. This will disassociate containers from virtual machines, and you can then make the best use of a distributed container-based architecture.

So far, we've looked at the logging aspects of containers, but one of the essential elements of a DevOps engineer's role is monitoring. We'll have a look at this in the next section.

Docker monitoring with Prometheus

Monitoring Docker nodes and containers is an essential part of managing Docker. There are various tools available for monitoring Docker. While you can use traditional tools such as Nagios, Prometheus is gaining ground in cloud-native monitoring because of its simplicity and pluggable architecture.

Prometheus is a free, open source monitoring tool that provides a dimensional data model, efficient and straightforward querying using the **Prometheus query language** (**PromQL**), efficient time series databases, and modern alerting capabilities.

It has several exporters available for exporting data from various sources and supports both virtual machines and containers. Before we delve into the details, let's look at some of the challenges with container monitoring.

Challenges with container monitoring

From a conceptual point of view, there is no difference between container monitoring and the traditional method. You still need metrics, logs, health checks, and service discovery. These aren't things that are unknown or haven't been explored before. The problem with containers is the abstraction that they bring with them; let's look at some of the issues:

- Containers behave like mini virtual machines; however, in reality, they are processes running on a server. However, they still have everything to monitor that we would in a virtual machine.

A container process will have many metrics, very similar to virtual machines, to be treated as separate entities altogether. When dealing with containers, most people make this mistake when they map containers to a particular virtual machine.

- Containers are temporary, and most people don't realize that. When you have a container and it is recreated, it has a new IP. This can confuse traditional monitoring systems.
- Containers running on clusters can move from one node (server) to another. This adds another layer of complexity as your monitoring tool needs to know where your containers are to scrape metrics from them. This should not matter with the more modern, container-optimized tools.

Prometheus helps us address these challenges as it is built from a distributed application's point of view. To understand this, we'll look at a hands-on example. However, before that, let's install Prometheus on a separate Ubuntu 22.04 Linux machine.

Installing Prometheus

Installing Prometheus consists of several steps, and for simplicity, I've created a Bash script for installing and setting up Prometheus on an Ubuntu machine.

Use the following commands on a separate machine where you want to set up Prometheus:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd modern-devops/ch3/prometheus/
$ sudo bash prometheus_setup.sh
```

To check whether Prometheus is installed and running, check the status of the Prometheus service using the following command:

```
$ sudo systemctl status prometheus
prometheus.service - Prometheus
   Loaded: loaded (/etc/systemd/system/prometheus.service; enabled; vendor preset:
enabled)
   Active: active (running) since Tue 2023-06-01 09:26:57 UTC; 1min 22s ago
```

As the service is `Active`, we can conclude that Prometheus has been installed and is running successfully. The next step is configuring the Docker server to enable Prometheus to collect logs from it.

Configuring cAdvisor and the node exporter to expose metrics

Now, we'll launch a cAdvisor container on the machine running Docker to expose the metrics of the Docker containers. cAdvisor is a metrics collector that scrapes metrics from containers. To launch the container, use the following command:

```
$ docker run -d --restart always --name cadvisor -p 8080:8080 \
-v "/:/rootfs:ro" -v "/var/run:/var/run:rw" -v "/sys:/sys:ro" \
-v "/var/lib/docker/:/var/lib/docker:ro" google/cadvisor:latest
```

Now that cAdvisor is running, we need to configure the node exporter to export node metrics on the Docker machine. To do so, run the following commands:

```
$ cd ~/modern-devops/ch3/prometheus/
$ sudo bash node_exporter_setup.sh
```

Now that the node exporter is running, let's configure Prometheus to connect to cAdvisor and the node exporter and scrape metrics from there.

Configuring Prometheus to scrape metrics

We will now configure Prometheus on the Prometheus machine so that it can scrape the metrics from cAdvisor. To do so, modify the `/etc/prometheus/prometheus.yml` file so that it includes the following within the server running Prometheus:

```
$ sudo vim /etc/prometheus/prometheus.yml
  ...
  - job_name: 'node_exporter'
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:9100', '<Docker_IP>:9100']
  - job_name: 'Docker Containers'
    static_configs:
      - targets: ['<Docker_IP>:8080']
```

After changing this configuration, we need to restart the Prometheus service. Use the following command to do so:

```
$ sudo systemctl restart prometheus
```

Now, let's launch a sample web application that we will monitor using Prometheus.

Launching a sample container application

Now, let's run an NGINX container called `web` that runs on port `8081` on the Docker machine. To do so, use the following command:

```
$ docker run -d --name web -p 8081:80 nginx
f9b613d6bdf3d6aee0cb3a08cb55c99a7c4821341b058d8757579b52cabbb0f5
```

Now that we've set up the Docker container, let's go ahead and open the Prometheus UI by visiting `https://<PROMETHEUS_SERVER_EXTERNAL_IP>:9090` and then running the following query by typing it in the textbox:

```
container_memory_usage_bytes{name=~"web"}
```

It should show something like the following:

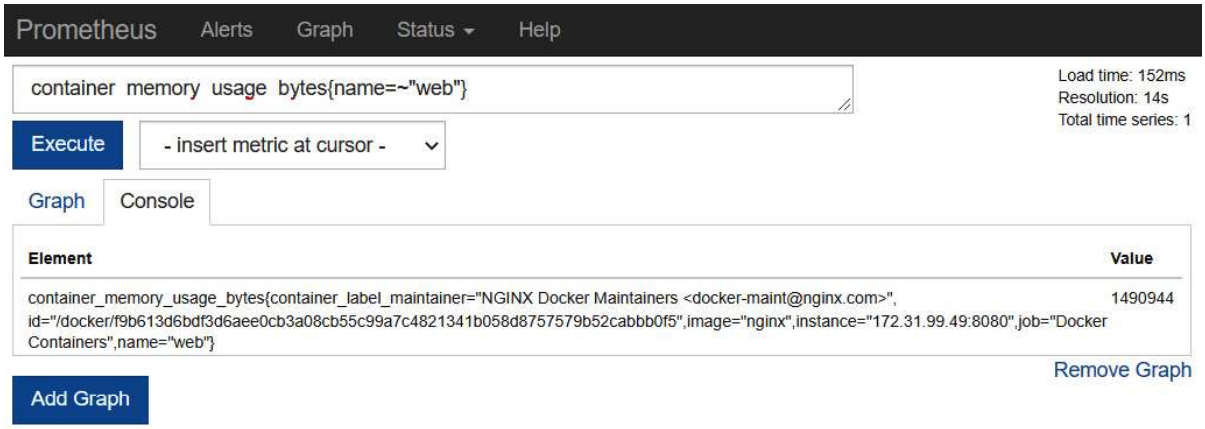

Figure 3.2 – Prometheus – container_memory_usage_bytes

We can also view the time series of this metric by clicking on the **Graph** tab. However, before doing so, let's load our NGINX service using the Apache Bench tool. Apache Bench is a load-testing tool that helps us fire HTTP requests to the NGINX endpoint using the command line.

On your Docker server, run the following command to start a load test:

```
$ ab -n 100000 http://localhost:8081/
```

It will hit the endpoint with 100,000 requests, which means it provides a fair amount of load to do a memory spike. Now, if you open the **Graph** tab, you should see something like the following:

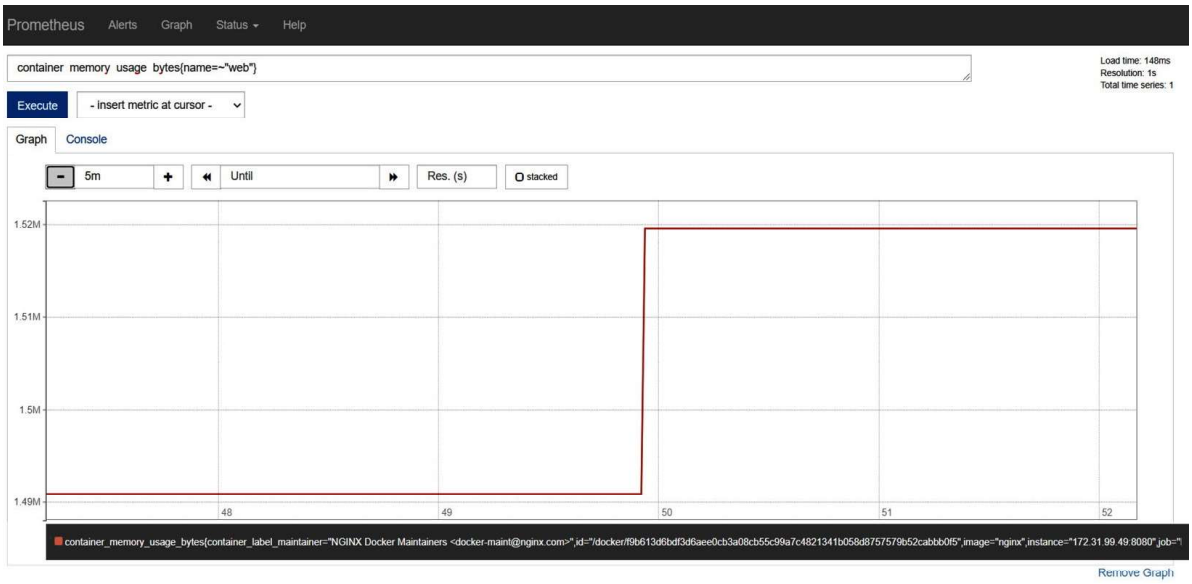

Figure 3.3 – Prometheus – container_memory_usage_bytes – Graph

To visualize node metrics, we can use the following PromQL statement to get the `node_cpu` value of the Docker host:

```
node_cpu{instance="<Docker_IP>:9100",job="node_exporter"}
```

As shown in the following screenshot, it will provide us with the `node_cpu` metrics for multiple modes:

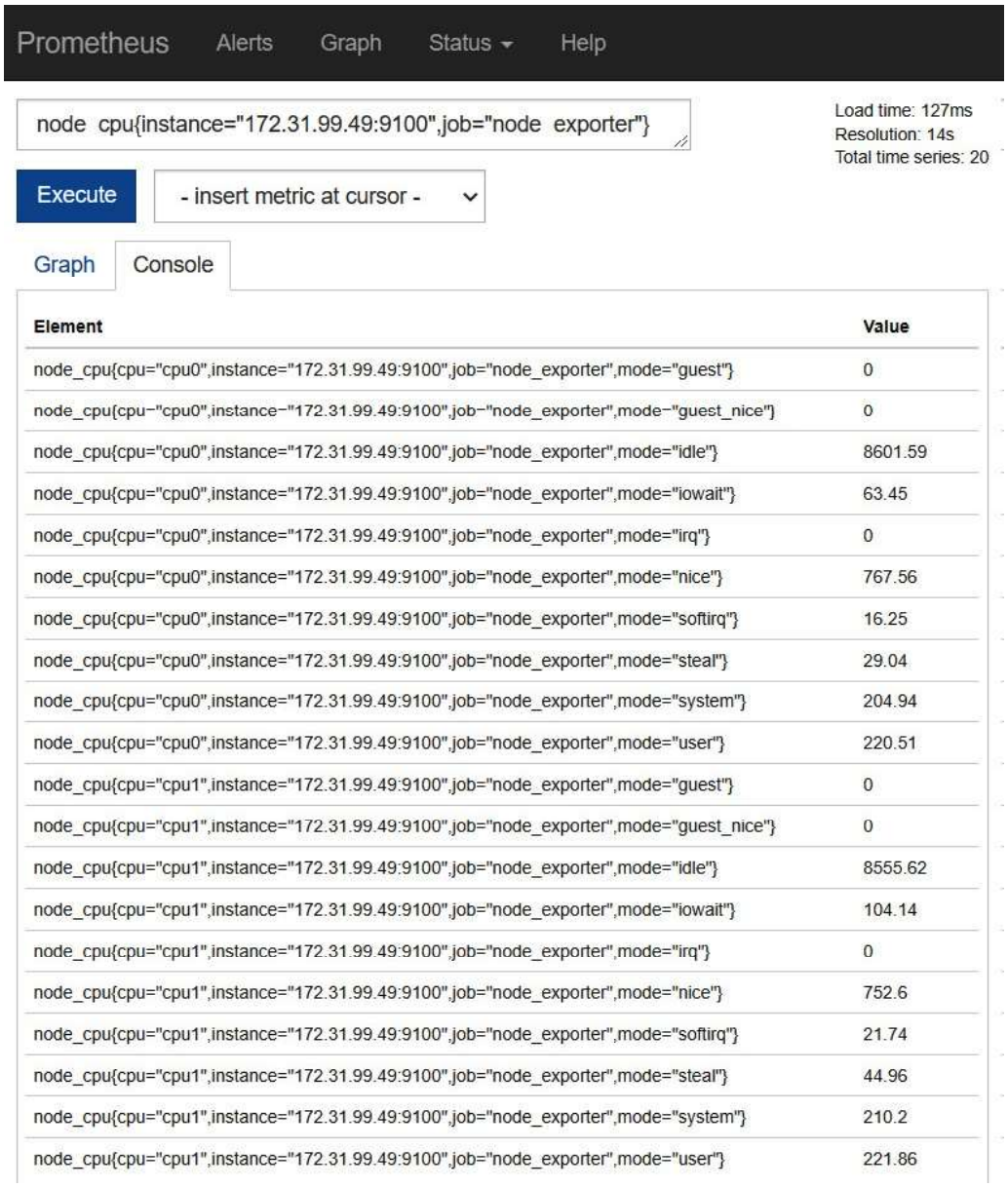

Figure 3.4 – Prometheus – node_cpu

There are a variety of other metrics that Prometheus gives you to visualize. Let's understand some of the metrics you can monitor.

Metrics to monitor

Monitoring metrics is a complex subject, and it would depend mostly on your use case. However, the following are some guidelines on what metrics you want to monitor.

Host metrics

You need to monitor your host metrics as your containers run on them. Some of the metrics that you can watch are as follows:

- **Host CPU**: It's good to know whether your host has sufficient CPU to run your containers. If not, it might terminate some of your containers to account for that. So, to ensure reliability, you need to keep this in check.
- **Host memory**: Like the host CPU, you need to watch the host memory to detect issues such as memory leaks and runaway memory.
- **Host disk space**: As Docker containers use the host filesystem to store transient and persistent files, you need to monitor it.

Docker container metrics

Docker container metrics are the next thing to consider:

- **Container CPU**: This metric will provide the amount of CPU used by the Docker container. You should monitor it to understand the usability pattern and decide where to place your container effectively.
- **Throttled CPU time**: This metric allows us to understand the total time when the CPU was throttled for a container. This lets us know whether a particular container needs more CPU time than others, and you can adjust the CPU share constraint accordingly.
- **Container memory fail counters**: This metric provides the number of times the container requested more than the allocated memory. It will help you understand what containers require more than the allocated memory, and you can plan to run those containers accordingly.
- **Container memory usage**: This metric will provide the amount of memory used by the Docker container. You can set memory limits according to the usage.
- **Container swap**: This metric will tell you what containers were using swap instead of RAM. It helps us identify memory-hungry containers.
- **Container disk I/O**: This is an important metric and will help us understand containers' disk profiles. Spikes can indicate a disk bottleneck or suggest that you might want to revisit your storage driver configuration.

- **Container network metrics**: This metric will tell us how much network bandwidth the containers use and help us understand traffic patterns. You can use these to detect an unexpected network spike or a denial-of-service attack.

> **Important tip**
>
> Profiling your application during the performance testing phase in the non-production environment will give you a rough idea of how the system will behave in production. The actual fine-tuning of your application begins when you deploy them to production. Therefore, monitoring is critical, and fine-tuning is a continuous process.

So far, we have been running commands to do most of our work. That is the imperative way of doing this. But what if I told you that instead of typing commands, you could declare what you want, and something could run all the required commands on your behalf? That is known as the declarative method of managing an application. Docker Compose is one of the popular tools to achieve this. We'll have a look at this in the next section.

Declarative container management with Docker Compose

Docker Compose helps you manage multiple containers in a declarative way. You can create a YAML file and specify what you want to build, what containers you want to run, and how the containers interact with each other. You can define mounts, networks, port mapping, and many different configurations in the YAML file.

After that, you can simply run `docker compose up` to run your entire containerized application.

Declarative management is quickly gaining ground because of its power and simplicity. Now, sysadmins don't need to remember what commands they had run or write lengthy scripts or playbooks to manage containers. Instead, they can simply declare what they want in a YAML file, and `docker compose` or other tools can help them achieve that state. We installed Docker Compose when we installed Docker, so let's see it in action with a sample application.

Deploying a sample application with Docker Compose

We have a Python Flask application that listens on port `5000`, which we will eventually map to host port `80`. The application will connect to the Redis database as a backend service on its default port, `6379`, and fetch the page's last visit time. We will not expose that port to the host system. This means the database is entirely out of bounds for any external party with access to the application.

The following diagram depicts the application architecture:

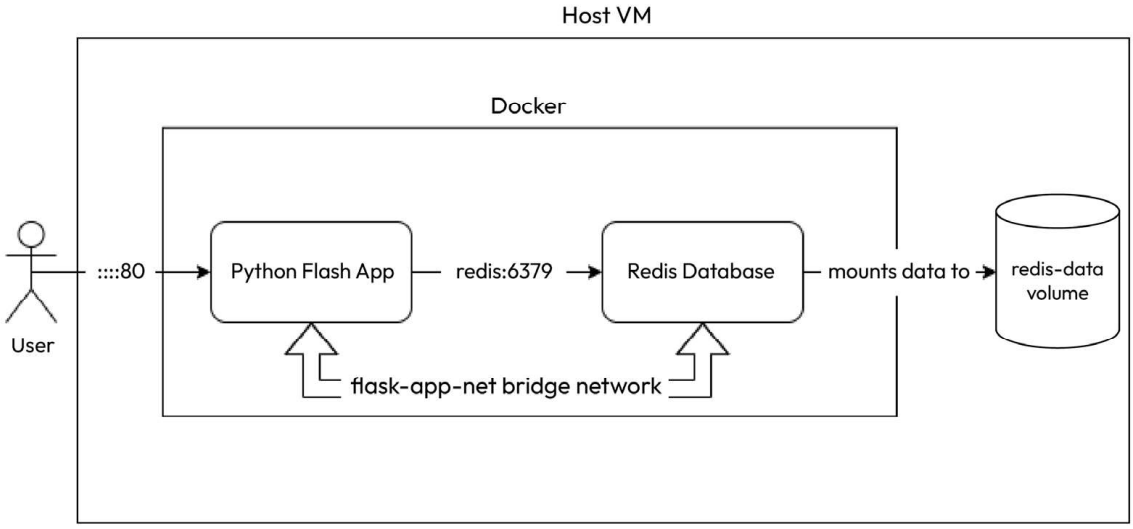

Figure 3.5 – Sample application

The necessary files are available in this book's GitHub repository. Run the following command to locate the files:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd modern-devops/ch3/docker-compose
$ ls -l
total 16
-rw-r--r-- 1 root root 681 Nov 25 06:11 app.py
-rw-r--r-- 1 root root 389 Nov 25 06:45 docker-compose.yaml
-rw-r--r-- 1 root root 238 Nov 25 05:27 Dockerfile
-rw-r--r-- 1 root root  12 Nov 25 05:26 requirements.txt
```

The `app.py` file looks as follows:

```
import time
import redis
from flask import Flask
from datetime import datetime
app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)
def get_last_visited():
    try:
        last_visited = cache.getset('last_visited',str(datetime.now().strftime("%Y-%m-%d,
%H:%M:%S")))
        if last_visited is None:
            return cache.getset('last_visited',str(datetime.now().strftime("%Y-%m-%d,
%H:%M:%S")))
        return last_visited
    except redis.exceptions.ConnectionError as e:
```

```
        raise e
@app.route('/')
def index():
    last_visited = str(get_last_visited().decode('utf-8'))
    return 'Hi there! This page was last visited on {}.\n'.format(last_visited)
```

The `requirements.txt` file looks as follows:

```
flask
redis
```

I've already built the application for you, and the image is available on Docker Hub. The next chapter will cover how to build a Docker image in detail. For now, let's have a look at the `docker-compose` file.

Creating the docker-compose file

The next step in the process is to create a `docker-compose` file. A `docker-compose` file is a YAML file that contains a list of services, networks, volumes, and other associated configurations. Let's look at the following example `docker-compose.yaml` file to understand it better:

```
version: "2.4"
services:
  flask:
    image: "bharamicrosystems/python-flask-redis:latest"
    ports:
      - "80:5000"
    networks:
      - flask-app-net
  redis:
    image: "redis:alpine"
    networks:
      - flask-app-net
    command: ["redis-server", "--appendonly", "yes"]
    volumes:
      - redis-data:/data
networks:
  flask-app-net:
    driver: bridge
volumes:
  redis-data:
```

The YAML file describes two services – `flask` and `redis`.

The `flask` service uses the `python-flask-redis:latest` image – the image we built with the preceding code. It also maps host port `80` to container port `5000`, exposing this application to your host machine on port `80`, and you can access it via `http://localhost`.

The `redis` service uses the official `redis:alpine` image and does not expose any port, as we don't want this service outside the container network's confines. However, it declares a persistent volume,

`redis-data`, that comprises the `/data` directory. We can mount this volume on the host filesystem for persistence beyond the container life cycle.

There is also a `flask-app-net` network that uses the bridge driver, and both services share the same network. This means the services can call each other by using their service names. If you look at the `app.py` code, you will see that we established a Redis service connection using the `redis` hostname.

To apply the configuration, simply run `docker-compose up -d`:

```
$ docker compose up -d
[+] Running 17/17
flask 9 layers []      0B/0B      Pulled 10.3s
redis 6 layers []      0B/0B      Pulled 9.1s
[+] Building 0.0s (0/0)
[+] Running 4/4
Network docker-compose_flask-app-net  Created 0.1s
Volume "docker-compose_redis-data"    Created 0.0s
Container docker-compose-flask-1      Started 3.8s
Container docker-compose-redis-1      Stated
```

Now, let's list the Docker containers to see how we fare:

```
$ docker ps
CONTAINER ID  IMAGE              COMMAND         CREATED    STATUS   PORTS             NAMES
9151e72f5d66  redis:             "docker-        3 minutes  Up 3     6379/tcp          docker-compose
              alphone            entrypoint.s…"  ago        minutes                    -redis-1
9332c2aaf2c4  bharamicrosystems  "flask run"     3 minutes  Up 3     0.0.0.0:80->      docker-compose
              /python-flask-                     ago        minutes  5000/tcp,         -flask-1
              redis:latest                                           :::80->5000/tcp
```

We can see that two containers are running for both services. We can also see host port `80` forwarding connections to container port `5000` on the `flask` service.

The `redis` service is internal and therefore, there is no port mapping.

Let's run `curl localhost` and see what we get:

```
$ curl localhost
Hi there! This page was last visited on 2023-06-01, 06:54:27.
```

Here, we get the last visited page from the Redis cache according to the sample Flask application code.

Let's run this a few times and see whether the time changes:

```
$ curl localhost
Hi there! This page was last visited on 2023-06-01, 06:54:28.
$ curl localhost
Hi there! This page was last visited on 2023-06-01, 06:54:51.
```

```
$ curl localhost
Hi there! This page was last visited on 2023-06-01, 06:54:52.
```

We can see that the last visited time changes every time we `curl`. Since the volume is persistent, we should get similar last visited times even after a container restarts.

First, let's `curl` and get the last visited time and also the current date:

```
$ curl localhost && date
Hi there! This page was last visited on 2023-06-01, 06:54:53.
Thu Jun  1 06:55:50 UTC 2023
```

Now, the next time we `curl`, we should get a date-time similar to `2023-06-01, 06:55:50`. But before that, let's restart the container and see whether the data persists:

```
$ docker compose restart redis
[+] Restarting 1/1
Container docker-compose-redis-1  Started
```

Now that Redis has been restarted, let's run `curl` again:

```
$ curl localhost
Hi there! This page was last visited on 2023-06-01, 06:55:50.
```

As we can see, we get the correct last visited time, even after restarting the `redis` service. This means that data persistence works correctly, and the volume is adequately mounted.

You can do many other configurations on `docker compose` that you can readily get from the official documentation. However, you should now have a general idea about using `docker compose` and its benefits. Now, let's look at some of the best practices associated with Docker Compose.

Docker Compose best practices

Docker Compose provides a declarative way of managing Docker container configuration. This enables GitOps for your Docker workloads. While Docker Compose is primarily used in development environments, you can use it in production very effectively, especially when Docker runs in production and does not use another container orchestrator such as Kubernetes.

Always use docker-compose.yml files alongside code

The YAML file defines how to run your containers. So, it becomes a valuable tool for declaratively building and deploying your containers from a single space. You can add all dependencies to your application and run related applications in a single network.

Separate multiple environment YAMLs using overrides

Docker Compose YAML files allow us to both build and deploy Docker images. Docker has enabled the *build once, run anywhere* concept. This means we build once in the development environment and then use the created image in subsequent environments. So, the question arises of how we can achieve that. Docker Compose allows us to apply multiple YAML files in a sequence where the next configuration overrides the last. That way, we can have separate override files for various environments and manage multiple environments using a set of these files.

For example, say we have the following base `docker-compose.yaml` file:

```
version: "2.4"
services:
  flask:
    image: "bharamicrosystems/python-flask-redis:latest"
    ports:
      - "80:5000"
    networks:
      - flask-app-net
  redis:
    image: "redis:alpine"
    networks:
      - flask-app-net
    command: ["redis-server", "--appendonly", "yes"]
    volumes:
      - redis-data:/data
networks:
  flask-app-net:
    driver: bridge
volumes:
  redis-data:
```

We only have to build the Flask application container image in the development environment so that we can create an override file for the development environment – that is, `docker-compose.override.yaml`:

```
web:
  build: .
  environment:
    DEBUG: 'true'
redis:
  ports:
    - 6379:6379
```

Here, we added a `build` parameter within the web service. This means the Python Flask application will be rebuilt and then deployed. We also set the `DEBUG` environment variable within the `web` service and exposed the `redis` port to the host filesystem. This makes sense in the development environment, as we might want to debug Redis from the development machine directly. Still, we would not want something of that sort in the production environment. Therefore, the default `docker-compose.yaml` file will work in the production environment, as we saw in the previous section.

Use an .env file to store sensitive variables

You might not want to store sensitive content such as passwords and secrets in version control. Instead, you can use an `.env` file that contains a list of variable names and values and keep it in a secret management system such as HashiCorp Vault.

Be mindful of dependencies in production

When you change a particular container and want to redeploy it, `docker-compose` also redeploys any dependencies. Now, this might not be something that you wish to do, so you can override this behavior by using the following command:

```
$ docker-compose up --no-deps -d <container_service_name>
```

Treat docker-compose files as code

Always version control your `docker-compose` files and keep them alongside the code. This will allow you to track their versions and use gating and Git features such as pull requests.

Summary

This chapter was designed to cater to both beginners and experienced individuals. We started by covering the foundational concepts of Docker and gradually delved into more advanced topics and real-world use cases. This chapter began with installing Docker, running our first Docker container, understanding various modes of running a container, and understanding Docker volumes and storage drivers. We also learned how to select the right storage driver, volume options, and some best practices. All these skills will help you easily set up a production-ready Docker server. We also discussed the logging agent and how to quickly ship Docker logs to multiple destinations, such as **journald**, **Splunk**, and JSON files, to help you monitor your containers. We looked at managing Docker containers declaratively using **Docker Compose** and deployed a complete composite container application. Please try out all the commands mentioned in this chapter for a more hands-on experience – practice is vital to achieving something worthwhile and learning something new.

As a next step, in the following chapter, we will look at Docker images, creating and managing them, and some best practices.

Questions

Answer the following questions to test your knowledge of this chapter:

1. You should use `overlay2` for CentOS and RHEL 7 and below. (True/False)
2. Which of the following statements is true? (Choose four)

   A. Volumes increase IOPS.

   B. Volumes decrease IOPS.

C. `tmpfs` mounts use system memory.

D. You can use bind mounts to mount host files to containers.

E. You can use volume mounts for a multi-instance active-active configuration.

3. Changing the storage driver removes existing containers from the host. (True/False)
4. `devicemapper` is a better option than `overlay2` for write-intensive containers. (True/Fals)
5. Which of the following logging drivers are supported by Docker? (Choose four)

   A. `journald`

   B. Splunk

   C. JSON files

   D. Syslog

   E. Logstash

6. Docker Compose is an imperative approach to managing containers. (True/False)
7. Which of the following `docker run` configurations are correct? (Choose three)

   A. `docker run nginx`

   B. `docker run --name nginx nginx:1.17.3`

   C. `docker run -d --name nginx nginx`

   D. `docker run -d --name nginx nginx --restart never`

Answers

The following are the answers to this chapter's questions:

1. False – You should use `devicemapper` for CentOS and RHEL 7 and below as they do not support `overlay2`.
2. B, C, D, E.
3. True.
4. True.
5. A, B, C, D.
6. False – Docker Compose is a declarative approach to container management.
7. A, B, C.

4

Creating and Managing Container Images

In the previous chapter, we covered containerization with Docker, where we installed Docker and ran our first container. We covered some core fundamentals, including Docker volumes, mounts, storage drivers, and logging drivers. We also covered Docker Compose as a declarative method of managing containers.

Now, we will discuss the core building blocks of containers: container images. Container images also fulfill a core principle of modern DevOps practices: config as code. Therefore, understanding container images, how they work, and how to build an image effectively is very important for a modern DevOps engineer.

In this chapter, we're going to cover the following main topics:

- Docker architecture
- Understanding Docker images
- Understanding Dockerfiles, components, and directives
- Building and managing Docker images
- Flattening Docker images
- Optimizing containers with distroless images
- Understanding Docker registries

Technical requirements

For this chapter, we assume that you have Docker installed on a Linux machine running Ubuntu 18.04 Bionic LTS or later with sudo access. You can read *Chapter 3*, *Containerization with Docker*, for more details on how to do that.

You will also need to clone a GitHub repository for some of the exercises in this chapter, which you can find at `https://github.com/PacktPublishing/Modern-DevOps-Practices-2e`. Also, you need a Docker Hub account for most of the activities. To create one, go to `https://hub.docker.com/`.

Docker architecture

Imagine you're a passionate chef dedicated to creating mouthwatering dishes that satisfy hungry customers. In your kitchen, which is a magical place called Docker, you have special powers to plan, make, and showcase your culinary creations. Let's break down the key parts:

Ingredients (Application Code and Dependencies): Imagine your kitchen has shelves filled with ingredients such as flour, eggs, and spices. These ingredients come together in a specific way to make a dish. Similarly, your application code and dependencies work together to build your application.

Recipe (Image): Each recipe is like a plan for a particular dish. Imagine having a recipe for chocolate cake or pasta carbonara. These recipes are like the building blocks for your creations. In the same way, a Docker image is a plan for making your Docker container.

Recipe Cards (Dockerfile): Your cooking journey involves using special recipe cards called Dockerfiles. These cards show you the important steps and ingredients (commands) to follow. For example, a Dockerfile for a chocolate cake might have steps such as "Mix the flour and sugar" or "Add eggs and cocoa powder." These Dockerfiles guide your helpers (Docker) in making the dish (container).

Cooked Dish (Container): When someone wants a dish, you use the recipe (image) to make it. Then, you have a fresh, hot dish ready to serve. These dishes are separate, but they can be made again and again (thanks to the recipe), just like a container.

Kitchen Staff (Docker Engine): In your bustling kitchen, your helpers (Docker Engine) play a big role. They do the hard work, from getting ingredients to following the recipe and serving the dish. You give them instructions (Docker commands), and they make it happen. They even clean up after making each dish.

Special Set Menu (Docker Compose): Sometimes, you want to serve a special meal with multiple dishes that go well together. Think of a meal with an appetizer, a main course, and a dessert. Using Docker Compose is like creating a special menu for that occasion. It lists recipes (images) for each part of the meal and how they should be served. You can even customize it to create a whole meal experience with just one command.

Storage Area (Volumes): In your kitchen, you need a place to keep ingredients and dishes. Think of Docker volumes as special storage areas where you can keep important things, such as data and files, that multiple dishes (containers) can use.

Communication Channels (Networks): Your kitchen is a busy place with lots of talking and interacting. In Docker, networks are like special communication paths that help different parts of your kitchen (containers) talk to each other.

So, Docker is like your magical kitchen where you make dishes (containers) using plans (Dockerfiles) and ingredients (images) with the assistance of your kitchen helpers (Docker Engine). You can even serve entire meals (Docker Compose) and use special storage areas (volumes) and communication paths (networks) to make your dishes even more amazing. Just like a chef gets better with practice, exploring Docker will help you become a master of DevOps in no time! Now, let's dive deeper into Docker architecture to understand its nuances!

As we already know, Docker uses the *build once, run anywhere* concept. Docker packages applications into images. Docker images form the blueprint of containers, so a container is an instance of an image.

A container image packages applications and their dependencies, so they are a single immutable unit you can run on any machine that runs Docker. You can also visualize them as a snapshot of the container.

We can build and store Docker images in a Docker registry, such as **Docker Hub**, and then download and use those images in the system where we want to deploy them. Images comprise several layers, which helps break images into multiple parts. The layers tend to be reusable stages that other images can build upon. This also means we don't have to transmit the entire image over a network when changing images. We only transmit the delta, which saves a lot of network I/O. We will talk about the layered filesystem in detail later in this chapter.

The following diagram shows the components Docker uses to orchestrate the following activities:

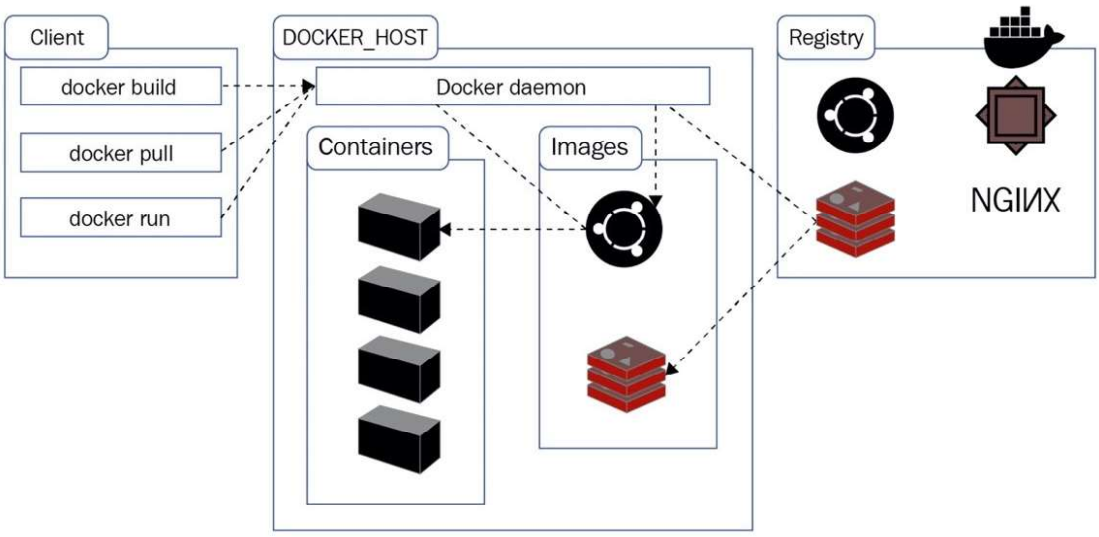

Figure 4.1 – Docker architecture

The components are:

- **Docker daemon**: This process runs on the servers that we want to run our containers on. They deploy and run containers on the Docker server.
- **Docker registries**: These store and distribute Docker images.
- **Docker client**: This is the command-line utility that we've been using to issue `docker` commands to the Docker daemon.

Now that we understand Docker architecture's key components and how Docker images play an essential role, let's understand Docker images and their components, directives, and registries in detail.

Understanding Docker images

Docker images form the blueprint of Docker containers. Just like you need a blueprint for a shipping container to determine its size and what goods it will contain, a Docker image specifies what packages, source code, dependencies, and libraries it needs to use. It also determines what it needs to do for the source code to run effectively.

Technically, it consists of a series of steps you would perform on a base OS image to get your application up and running. This may include installing packages and dependencies, copying the source code to the correct folder, building your code to generate a binary, and so on.

You can store Docker images in a container registry, a centralized location from where your Docker machines can pull images to create containers.

Docker images use a layered filesystem. Instead of a huge monolithic block on the filesystem that comprises the template to run containers, we have many layers, one on top of the other. But what does this mean? What problem does this solve? Let's have a look in the next section.

The layered filesystem

Layers in Docker are intermediate Docker images. The idea is that every Dockerfile statement we execute on top of a layer changes something within the layer and builds a new one. The subsequent statement modifies the current one to generate the next one. The final layer executes the Docker `CMD` or `ENTRYPOINT` command, and the resulting image comprises several layers arranged one on top of the other. Let's understand this by looking at a simple example.

If we pull the *Flask application* we built in the previous chapter, we will see the following:

```
$ docker pull bharamicrosystems/python-flask-redis
Using default tag: latest
latest: Pulling from bharamicrosystems/python-flask-redis
188c0c94c7c5: Pull complete
a2f4f20ac898: Pull complete
f8a5b284ee96: Pull complete
```

```
28e9c106bfa8: Pull complete
8fe1e74827bf: Pull complete
95618753462e: Pull complete
03392bfaa2ba: Pull complete
4de3b61e85ea: Pull complete
266ad40b3bdb: Pull complete
Digest: sha256:bb40a44422b8a7fea483a775fe985d4e05f7e5c59b0806a2
4f6cca50edadb824
Status: Downloaded newer image for bharamicrosystems/python-flask-redis:latest
docker.io/bharamicrosystems/python-flask-redis:latest
```

As you can see, many `Pull complete` statements are beside random IDs. These are called **layers**. The current layer contains just the differences between the previous and current filesystem. A container image comprises several layers.

Containers contain an additional writable filesystem on top of the image layers. This is the layer where your containers modify the filesystem to provide the expected functionality.

There are several advantages of using layers instead of merely copying the entire filesystem of the container. Since image layers are read-only, multiple containers created from an image share the same layered filesystem, decreasing the overall disk and network footprint. Layers also allow you to share filesystems between images. For example, if two images come from a single base image, both images share the same base layer.

The following diagram shows a Python application that runs on an Ubuntu OS. At a high level, you will see a base layer (Ubuntu OS) and Python installed on top of it. On top of Python, we've installed the Python app. All these components form the image. When we create a container out of the image and run it, we get the writable filesystem on top as the final layer:

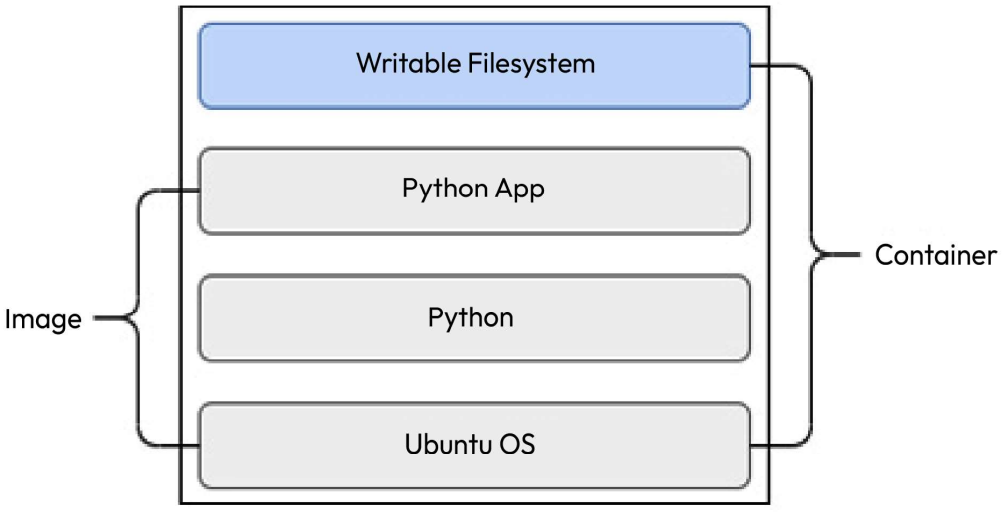

Figure 4.2 – Container layers

So, you can create multiple Python app images from the same base image and customize them according to your needs.

The writable filesystem is unique for every container you spin from container images, even if you create containers from the same image.

Image history

To understand images and their layers, you can always inspect the image history.

Let's inspect the history of the last Docker image by running the following command:

```
$ docker history bharamicrosystems/python-flask-redis
IMAGE          CREATED      CREATED BY                                      SIZE    COMMENT
6d33489ce4d9   2 years ago  /bin/sh -c #(nop)  CMD ["flask" "run"]          0B
<missing>      2 years ago  /bin/sh -c #(nop) COPY dir:61bb30c35fb351598…   1.2kB
<missing>      2 years ago  /bin/sh -c #(nop)  EXPOSE 5000                  0B
<missing>      2 years ago  /bin/sh -c pip install -r requirements.txt      11.2MB
<missing>      2 years ago  /bin/sh -c #(nop) COPY file:4346cf08412270cb…   12B
<missing>      2 years ago  /bin/sh -c apk add --no-cache gcc musl-dev l…   143MB
<missing>      2 years ago  /bin/sh -c #(nop)  ENV FLASK_RUN_HOST=0.0.0.0   0B
<missing>      2 years ago  /bin/sh -c #(nop)  ENV FLASK_APP=app.py         0B
<missing>      2 years ago  /bin/sh -c #(nop)  CMD ["python3"]              0B
<missing>      2 years ago  /bin/sh -c set -ex;   wget -O get-pip.py "$P…   7.24MB
<missing>      2 years ago  /bin/sh -c #(nop)  ENV PYTHON_GET_PIP_SHA256…   0B
<missing>      2 years ago  /bin/sh -c #(nop)  ENV PYTHON_GET_PIP_URL=ht…   0B
<missing>      2 years ago  /bin/sh -c #(nop)  ENV PYTHON_PIP_VERSION=20…   0B
<missing>      2 years ago  /bin/sh -c cd /usr/local/bin  && ln -s idle3…   32B
<missing>      2 years ago  /bin/sh -c set -ex  && apk add --no-cache --…   28.3MB
<missing>      2 years ago  /bin/sh -c #(nop)  ENV PYTHON_VERSION=3.7.9     0B
<missing>      2 years ago  /bin/sh -c #(nop)  ENV GPG_KEY=0D96DF4D4110E…   0B
<missing>      2 years ago  /bin/sh -c set -eux;  apk add --no-cache   c…   512kB
<missing>      2 years ago  /bin/sh -c #(nop)  ENV LANG=C.UTF-8             0B
<missing>      2 years ago  /bin/sh -c #(nop)  ENV PATH=/usr/local/bin:/…   0B
<missing>      2 years ago  /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>      2 years ago  /bin/sh -c #(nop) ADD file:f17f65714f703db90…   5.57MB
```

As you can see, there are several layers, and every layer has associated commands. You can also see when the layers were created and the size of the disk space occupied by each. Some layers do not occupy any disk space, as they haven't added anything new to the filesystem, such as `CMD` and `EXPOSE` directives. These perform some functions, but they do not write anything to the filesystem. While commands such as `apk add` write to the filesystem, you can see them taking up disk space.

Every layer modifies the old layer in some way, so every layer is just a delta of the filesystem configuration.

In the next section, we will deep dive into Dockerfiles and find out how we can build Docker images and see what the layered architecture looks like.

Understanding Dockerfiles, components, and directives

A Dockerfile is a simple file that constitutes a series of steps to build a Docker image. Each step is known as a **directive**. There are different kinds of directives. Let's look at a simple example to understand how this works.

We will create a simple NGINX container by building the image from scratch rather than using the one available on Docker Hub. NGINX is very popular web server software that you can use for a variety of applications; for example, it can serve as a load balancer or a reverse proxy.

Start by creating a Dockerfile:

```
$ vim Dockerfile
FROM ubuntu:bionic
RUN apt update && apt install -y curl
RUN apt update && apt install -y nginx
CMD ["nginx", "-g", "daemon off;"]
```

Let's look at each line and directive one by one to understand how this Dockerfile works:

- The `FROM` directive specifies what the base image for this container should be. This means we are using another image as the base and will be building layers on top of it. We use the `ubuntu:bionic` package as the base image for this build since we want to run NGINX on Ubuntu.
- The `RUN` directives specify the commands we need to run on a particular layer. You can run more than one command by separating them with `&&`. We want to run multiple commands in a single line if we're going to club dependent commands in a single layer. Every layer should meet a particular objective. In the preceding example, the first `RUN` directive is used to install `curl`, while the next `RUN` directive is used to install `nginx`.
- You might be wondering why we have `apt update` before every installation. This is required, as Docker builds images using layers. So, one layer should not have implicit dependencies on the previous one. In this example, if we omit `apt update` while installing `nginx`, and if we want to update the `nginx` version without changing anything in the directive containing `apt update` (that is, the line that installs `curl`), when we run the build, `apt update` will not run again, so your `nginx` installation might fail.
- The `CMD` directive specifies a list of commands that we need to run when the built image runs as a container. This is the default command that will be executed, and its output will end up in the container logs. Your container can contain one or more `CMD` directives. For a long-running process such as NGINX, the last `CMD` should contain something that will not pass control back to the shell and continue to run for the container's lifetime. In this case, we run `nginx -g daemon off;`, which is a standard way of running NGINX in the foreground.

Some directives can easily be confused with each other, such as `ENTRYPOINT` and `CMD` or `CMD` and `RUN`. These also test how solid your Docker fundamentals are, so let's look at both.

Can we use ENTRYPOINT instead of CMD?

Instead of `CMD`, you can use `ENTRYPOINT`. While they serve a similar purpose, they are two very different directives. Every Docker container has a default `ENTRYPOINT` – `/bin/sh -c`. Anything you add to `CMD` is appended post-`ENTRYPOINT` and executed; for example, `CMD ["nginx", "-g", "daemon off;"]` will be generated as `/bin/sh -c nginx -g daemon off;`. If you use a custom `ENTRYPOINT` instead, the commands you use while launching the container will be appended after it. So, if you define `ENTRYPOINT ["nginx", "-g"]` and use `docker run nginx daemon off;`, you will get a similar result.

To get a similar result without adding any `CMD` arguments while launching the container, you can also use `ENTRYPOINT ["nginx", "-g", "daemon off;"]`.

> **Tip**
>
> Use `ENTRYPOINT` unless there is a need for a specific `CMD` requirement. Using `ENTRYPOINT` ensures that users cannot change the default behavior of your container, so it's a more secure alternative.

Now, let's look at `RUN` versus `CMD`.

Are RUN and CMD the same?

No, `RUN` and `CMD` are different and serve different purposes. While `RUN` is used to build the container and only modifies the filesystem while building it, `CMD` commands are only executed on the writable container layer after the container is running.

While there can be several `RUN` statements in a Dockerfile, each modifying the existing layer and generating the next, if a Dockerfile contains more than one `CMD` command, all but the last one are ignored.

The `RUN` directives are used to execute statements within the container filesystem to build and customize the container image, thus modifying the image layers. The idea of using a `CMD` command is to provide the default command(s) with the container image that will be executed at runtime. This only changes the writeable container filesystem. You can also override the commands by passing a custom command in the `docker run` statement.

Now, let's go ahead and build our first container image.

Building our first container

Building a container image is very simple. It is actually a one-line command: `docker build -t <image-name>:version <build_context>`. While we will discuss building container images in detail in the *Building and managing container images* section, let's first build the Dockerfile:

```
$ docker build -t <your_dockerhub_user>/nginx-hello-world .
[+] Building 50.0s (7/7) FINISHED
 => [internal] load .dockerignore 0.0s
 => => transferring context: 2B 0.0s
 => [internal] load build definition from Dockerfile 0.1s
 => => transferring dockerfile: 171B 0.0s
 => [internal] load metadata for docker.io/library/ubuntu:bionic 2.4s
 => [1/3] FROM docker.io/library/ubuntu:bionic@sha256:152dc042… 2.8s
 => => resolve docker.io/library/ubuntu:bionic@sha256:152dc04… 0.0s
 => => sha256:152dc042… 1.33kB / 1.33kB 0.0s
 => => sha256:dca176c9… 424B / 424B 0.0s
 => => sha256:f9a80a55… 2.30kB / 2.30kB 0.0s
 => => sha256:7c457f21… 25.69MB / 25.69MB 1.0s
 => => extracting sha256:7c457f21… 1.6s
 => [2/3] RUN apt update && apt install -y curl 22.4s
 => [3/3] RUN apt update && apt install -y nginx 21.6s
 => exporting to image 0.6s
 => => exporting layers 0.6s
 => => writing image sha256:9d34cdda… 0.0s
 => => naming to docker.io/<your_dockerhub_user>/nginx-hello-world
```

You might have noticed that the name of the container had a prefix in front of it. That is your Docker Hub account name. The name of the image has a structure of `<registry-url>/<account-name>/<container-image-name>:<version>`.

Here, we have the following:

- `registry-url`: The URL to the Docker registry – defaults to `docker.io`
- `account-name`: The user or account that owns the image
- `container-image-name`: The container image's name
- `version`: The image version

Now, let's create a container out of the image using the following command:

```
$ docker run -d -p 80:80 <your_dockerhub_user>/nginx-hello-world
092374c4501560e96a13444ce47cb978b961cf8701af311884bfe…
$ docker ps
CONTAINER ID  IMAGE            COMMAND       CREATED     STATUS   PORTS                NAMES
092374c45015  <your_dockerhub  "nginx -g     28 seconds  Up 27    0.0.0.0:80->80/      loving_
              _user>/nginx-    'daemon of…"  ago         seconds  tcp, :::80->80/tcp   noether
              hello-world
```

Here, we can see that the container is up and running.

If we run `curl localhost`, we get the default `nginx` `html` response:

```
$ curl localhost
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
</body>
</html>
```

That's great! We have built our first image using a Dockerfile.

What if we wanted to customize the image according to our requirements? Practically speaking, no one would want an NGINX container just responding with the default `Welcome to nginx!` message, so let's create an index page and use that instead:

```
$ vim index.html
Hello World! This is my first docker image!
```

This one outputs a custom message instead of the default NGINX HTML page.

We all know that the default NGINX directory containing the `index.html` file is `/var/www/html`. If we can copy the `index.html` file into this directory, it should sort out our problem.

So, modify the Dockerfile so that it includes the following:

```
$ vim Dockerfile
FROM ubuntu:bionic
RUN apt update && apt install -y curl
RUN apt update && apt install -y nginx
WORKDIR /var/www/html/
ADD index.html ./
CMD ["nginx", "-g", "daemon off;"]
```

Here, we've added two directives to the file: `WORKDIR` and `ADD`. Let's understand what each one does:

- `WORKDIR`: This defines the current working directory, which is `/var/www/html` in this case. The last `WORKDIR` in the Dockerfile also specifies the working directory when the container is executed. So, if you `exec` into a running container, you will land in the last defined `WORKDIR`. `WORKDIR` can be absolute as well as relative to the current working directory.
- `ADD`: This adds a local file to the container filesystem – the working directory, in this case. You can also use a `COPY` directive here instead of `ADD`, though `ADD` offers some more features, such as downloading files from a URL and using an archive such as a TAR or ZIP package.