When we build this file, we expect the `index.html` file to be copied to the `/var/www/html` directory within the container filesystem. Let's have a look:

```
$ docker build -t <your_dockerhub_user>/nginx-hello-world .
[+] Building 1.6s (10/10) FINISHED
 => [internal] load build definition from Dockerfile 0.0s
 => => transferring dockerfile: 211B 0.0s
 => [internal] load .dockerignore 0.0s
 => => transferring context: 2B 0.0s
 => [internal] load metadata for docker.io/library/ubuntu:bionic 1.4s
 => [1/5] FROM docker.io/library/ubuntu:bionic@sha256:152dc042… 0.0s
 => [internal] load build context 0.0s
 => => transferring context: 81B 0.0s
 => CACHED [2/5] RUN apt update && apt install -y curl 0.0s
 => CACHED [3/5] RUN apt update && apt install -y nginx 0s
 => [4/5] WORKDIR /var/www/html/ 0.0s
 => [5/5] ADD index.html ./ 0.0s
 => exporting to image 0.0s
 => => exporting layers 0.0s
 => => writing image sha256:cb2e67bd… 0.0s
 => => naming to docker.io/<your_dockerhub_user>/nginx-hello-world
```

This time, the build was much faster! When we executed the Docker build, it used a lot of layers from the cache. That is one of the advantages of layered architecture; you only build the changing part and use the existing one the way it is.

> **Tip**
>
> Always add source code after installing packages and dependencies. The source code changes frequently and the packages more or less remain the same. This will result in faster builds and save a lot of CI/CD time.

Let's rerun the container and see what we get. Note that you need to remove the old container before doing so:

```
$ 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
$ docker rm 092374c45015 -f
092374c45015
```

At this point, we can't see the container anymore. Now, let's rerun the container using the following command:

```
$ docker run -d -p 80:80 <your_dockerhub_user>/nginx-hello-world
cc4fe116a433c505ead816fd64350cb5b25c5f3155bf5eda8cede5a4…
```

```
$ docker ps
CONTAINER ID  IMAGE             COMMAND       CREATED     STATUS   PORTS                NAMES
cc4fe116a433  <your_dockerhub   "nginx -g     52 seconds  Up 50    0.0.0.0:80->80/      eager_
              _user>/nginx-     'daemon of…"  ago         seconds  tcp, :::80->80/tcp   gates
              hello-world
```

Here, we can see that the container is up and running. Let's use `curl localhost` to see what we get:

```
$ curl localhost
Hello World! This is my first docker image!
```

Here, we get a custom message instead of the default NGINX HTML response!

This looks good enough for now, but I will discuss a few more directives to make this image more reliable. First, we haven't explicitly documented what port this container should expose. This works perfectly fine, as we know that NGINX runs on port `80`, but what if someone wants to use your image and doesn't know the port? In that scenario, it is best practice to define the port explicitly. We will use the `EXPOSE` directive for that.

> **Tip**
>
> Always use the `EXPOSE` directive to give more clarity and meaning to your image.

We also need to define the action to the container process if someone sends a `docker stop` command. While most processes take the hint and kill the process, it makes sense to explicitly specify what `STOPSIGNAL` the container should send on a `docker stop` command. We will use the `STOPSIGNAL` directive for that.

Now, while Docker monitors the container process and keeps it running unless it receives a `SIGTERM` or a stop, what would happen if your container process hangs for some reason? While your application is in a hung state, Docker still thinks it is running as your process is still running. Therefore, monitoring the application through an explicit health check would make sense. We will use the `HEALTHCHECK` directive for this.

Let's combine all these aspects and see what we get in the Dockerfile:

```
$ 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 ./
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
STOPSIGNAL SIGTERM
HEALTHCHECK --interval=60s --timeout=10s --start-period=20s --retries=3 CMD curl -f
localhost
```

While `EXPOSE` and `STOPSIGNAL` are self-explanatory, let's look at the `HEALTHCHECK` directive. The `HEALTHCHECK` directive runs a command (hence `CMD`) called `curl -f localhost`. So, this container will report itself as healthy until the result of the `curl` command is a success.

The `HEALTHCHECK` directive also contains the following optional fields:

- `--interval (default: 30s)`: The interval between two subsequent health checks.
- `--timeout (default: 30s)`: The health check probe timeout. If the health check times out, it implies a health check failure.
- `--start-period (default: 0s)`: The time lag between starting the container and the first health check. This allows you to ensure your container is up before a health check.
- `--retries (default: 3)`: The number of times the probe will retry before declaring an unhealthy status.

Now, let's build this container:

```
$ docker build -t <your_dockerhub_user>/nginx-hello-world .
[+] Building 1.3s (10/10) FINISHED
 => [internal] load build definition from Dockerfile 0.0s
 => => transferring dockerfile: 334B 0.0s
 => [internal] load .dockerignore 0.0s
 => => transferring context: 2B 0.0s
 => [internal] load metadata for docker.io/library/ubuntu:bionic 1.2s
 => [1/5] FROM docker.io/library/ubuntu:bionic@sha256:152dc0… 0.0s
 => [internal] load build context 0.0s
 => => transferring context: 31B 0.0s
 => CACHED [2/5] RUN apt update && apt install -y curl 0.0s
 => CACHED [3/5] RUN apt update && apt install -y nginx 0s
 => CACHED [4/5] WORKDIR /var/www/html/ 0.0s
 => CACHED [5/5] ADD index.html ./ 0.0s
 => exporting to image 0.0s
 => => exporting layers 0.0s
 => => writing image sha256:bba3123d… 0.0s
 => => naming to docker.io/<your_dockerhub_user>/nginx-hello-world
```

It's time to run it and see for ourselves:

```
$ docker run -d -p 80:80 <your_dockerhub_user>/nginx-hello-world
94cbf3fdd7ff1765c92c81a4d540df3b4dbe1bd9748c91e2ddf565d8…
```

Now that we have successfully launched the container, let's try `ps` and see what we get:

```
$ docker ps
CONTAINER ID  IMAGE            COMMAND       CREATED     STATUS      PORTS                NAMES
94cbf3fdd7ff  <your_dockerhub  "nginx -g     5 seconds   Up 4        0.0.0.0:80->80/      wonderful_
              _user>/nginx-    'daemon of…"  ago         (health:    tcp, :::80->80/tcp   hodgkin
              hello-world                                starting)
```

As we can see, the container shows `health: starting`, which means the health check hasn't been started yet, and we are waiting for the start time to expire.

Let's wait a while and then try `docker ps` again:

```
$ docker ps
CONTAINER ID  IMAGE             COMMAND       CREATED     STATUS     PORTS                NAMES
94cbf3fdd7ff  <your_dockerhub   "nginx -g     2 minutes   Up 2       0.0.0.0:80->80/      wonderful_
              _user>/nginx-     'daemon of…"  ago         (healthy)  tcp, :::80->80/tcp   hodgkin
              hello-world
```

This time, it reports the container as healthy. So, our container is now more reliable, as anyone monitoring it will know what part of the application is healthy and what part is not.

This health check only reports on the container's health status. It takes no action beyond that. You are responsible for periodically monitoring the containers and writing a script to action unhealthy containers.

One way to manage this would be to create a script that checks for unhealthy containers and restarts them. You can schedule such a script in your crontab. You can also create a long-running `systemd` script that continuously polls the container processes and checks for the health status.

> **Tip**
>
> While using `HEALTHCHECK` is a great option, you should avoid using it to run your containers on Kubernetes or a similar container orchestrator. You should make use of liveness and readiness probes instead. Similarly, you can define health checks on Docker Compose if you are using it, so use that instead of baking the health check into the container image.

Now, let's go ahead and learn how to build and manage Docker images.

Building and managing Docker images

We built some Docker images in the previous section, so by now, you should know how to write Dockerfiles and create Docker images from them. We've also covered a few best practices regarding it, which, in summary, are as follows:

- Always add the layers that do not change frequently first, followed by the layers that may change often. For example, install your packages and dependencies first and copy the source code later. Docker builds the Dockerfile from the part you change until the end, so if you change a line that comes later, Docker takes all the existing layers from the cache. Adding more frequently changing parts later in the build helps reduce the build time and will result in a faster CI/CD experience.
- Combine multiple commands to create as few layers as possible. Avoid multiple consecutive `RUN` directives. Instead, combine them into a single RUN directive using the `&&` clauses. This will help reduce the overall container footprint.

- Only add the required files within your container. Your container does not need the heavyweight package managers and the Go toolkit while running your containers if you have already compiled the code into a binary. We will discuss how to do this in detail in the following sections.

Docker images are traditionally built using a sequence of steps specified in the Dockerfile. But as we already know, Docker is DevOps-compliant and uses config management practices from the beginning. Most people build their code within the Dockerfile. Therefore, we will also need the programming language library in the build context. With a simple sequential Dockerfile, these programming language tools and libraries end up within the container image. These are known as single-stage builds, which we will cover next.

Single-stage builds

Let's containerize a simple Go application that prints `Hello, World!` on the screen. While I am using **Golang** in this application, this concept is applicable universally, irrespective of the programming language.

The respective files for this example are present in the `ch4/go-hello-world/single-stage` directory within this book's GitHub repository.

Let's look at the Go application file, `app.go`, first:

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, World!")
}
```

The Dockerfile appears as follows:

```
FROM golang:1.20.5
WORKDIR /tmp
COPY app.go .
RUN GOOS=linux go build -a -installsuffix cgo -o app . && chmod +x ./app
CMD ["./app"]
```

This is standard stuff. We take the `golang:1.20.5` base image, declare a `WORKDIR /tmp`, copy `app.go` from the host filesystem to the container, and build the Go application to generate a binary. Finally, we use the `CMD` directive with the generated binary to be executed when we run the container.

Let's build the Dockerfile:

```
$ docker build -t <your_dockerhub_user>/go-hello-world:single_stage .
[+] Building 10.3s (9/9) FINISHED
 => [internal] load build definition from Dockerfile 0.0s
 => => transferring dockerfile: 189B 0.0s
 => [internal] load .dockerignore 0.0s
```

```
 => => transferring context: 2B 0.0s
 => [internal] load metadata for docker.io/library/golang:1.20.5 0.6s
 => [1/4] FROM docker.io/library/golang:1.20.5@sha256:4b1fc02d… 0.0s
 => [internal] load build context 0.0s
 => => transferring context: 27B 0.0s
 => [2/4] WORKDIR /tmp 0.0s
 => [3/4] COPY app.go . 0.0s
 => [4/4] RUN GO111MODULE=off GOOS=linux go build -a -installsuffix cgo -o app .  && chmod
+x ./app 9.3s
 => exporting to image 0.3s
 => => exporting layers 0.3s
 => => writing image sha256:3fd3d261… 0.0s
 => => naming to docker.io/<your_dockerhub_user>/go-hello-world:single_stage
```

Now, let's run the Docker image and see what we get:

```
$ docker run <your_dockerhub_user>/go-hello-world:single_stage
Hello, World!
```

We get the expected response back. Now, let's run the following command to list the image:

```
$ docker images
REPOSITORY              TAG           IMAGE ID      CREATED        SIZE
<your_dockerhub_user>
/go-hello-world         single_stage  3fd3d26111a1  3 minutes ago  803MB
```

This image is huge! It takes 803 MB to print `Hello, World!` on the screen. This is not the most efficient way of building Docker images.

Before we look at the solution, let's understand why the image is so bloated in the first place. We use the Golang base image, which contains the entire Go toolkit and generates a simple binary. We do not need the complete Go toolkit for this application to run; it can efficiently run in an Alpine Linux image.

Docker solves this problem by providing multi-stage builds. You can split your build into stages where you can build your code in one stage and then, in the second stage, export the built code to another context that begins with a different base image that is much lighter and only contains those files and components that we need to run the code. We'll have a look at this in the next section.

Multi-stage builds

Let's modify the Dockerfile according to the multi-stage build process and see what we get.

The respective files for this example are present in the `ch4/go-hello-world/multi-stage` directory within this book's GitHub repository.

The following is the Dockerfile:

```
FROM golang:1.20.5 AS build
WORKDIR /tmp
```

```
COPY app.go .
RUN GO111MODULE=off GOOS=linux go build -a -installsuffix cgo -o app . && chmod +x ./app

FROM alpine:3.18.0
WORKDIR /tmp
COPY --from=build /tmp/app .
CMD ["./app"]
```

The Dockerfile contains two `FROM` directives: `FROM golang:1.20.5 AS build` and `FROM alpine:3.18.0`. The first `FROM` directive also includes an `AS` directive that declares the stage and names it `build`. Anything we do after this `FROM` directive can be accessed using the `build` term until we encounter another `FROM` directive, which would form the second stage. Since the second stage is the one we want to run our image from, we are not using an `AS` directive.

In the first stage, we build our Golang code to generate the binary using the `golang` base image.

In the second stage, we use the Alpine base image and copy the `/tmp/app` file from the build stage into our current stage. This is the only file we need to run in the container. The rest were only required to build and bloat our container during runtime.

Let's build the image and see what we get:

```
$ docker build -t <your_dockerhub_user>/go-hello-world:multi_stage
[+] Building 12.9s (13/13) FINISHED
 => [internal] load build definition from Dockerfile 0.0s
 => => transferring dockerfile: 259B 0.0s
 => [internal] load .dockerignore 0.0s
 => => transferring context: 2B 0.0s
 => [internal] load metadata for docker.io/library/alpine:3.18.0 2.0s
 => [internal] load metadata for docker.io/library/golang:1.20.5 1.3s
 => [build 1/4] FROM docker.io/library/golang:1.20.5@sha256:4b1fc02d… 0.0s
 => [stage-1 1/3] FROM docker.io/library/alpine:3.18.0@sha256:02bb6f42… 0.1s
 => => resolve docker.io/library/alpine:3.18.0@sha256:02bb6f42… 0.0s
 => => sha256:c0669ef3… 528B / 528B 0.0s
 => => sha256:5e2b554c… 1.47kB / 1.47kB 0.0s
 => => sha256:02bb6f42… 1.64kB / 1.64kB 0.0s
 => CACHED [build 2/4] WORKDIR /tmp 0.0s
 => [internal] load build context 0.0s
 => => transferring context: 108B 0.0s
 => [build 3/4] COPY app.go . 0.0s
 => [build 4/4] RUN GO111MODULE=off GOOS=linux go build -a -installsuffix cgo -o app . &&
chmod +x ./app 10.3s
 => [stage-1 2/3] WORKDIR /tmp 0.1s
=> [stage-1 3/3] COPY --from=build /tmp/app . 0.3s
 => exporting to image 0.1s
 => => exporting layers 0.1s
```

```
 => => writing image sha256:e4b793b3… 0.0s
 => => naming to docker.io/<your_dockerhub_user>/go-hello-world:multi_stage
```

Now, let's run the container:

```
$ docker run <your_dockerhub_user>/go-hello-world:multi_stage .
Hello, World!
```

We get the same output, but this time with a minimal footprint. Let's look at the image to confirm this:

```
$ docker images
REPOSITORY              TAG          IMAGE ID      CREATED        SIZE
<your_dockerhub_user>
/go-hello-world         multi_stage  e4b793b39a8e  5 minutes ago  9.17MB
```

This one occupies just 9.17 MB instead of the huge 803 MB. This is a massive improvement! We have reduced the image size by almost 100 times.

That is how we increase efficiency within our container image. Building efficient images is the key to running production-ready containers, and most professional images you find on Docker Hub use multi-stage builds to create efficient images.

> **Tip**
>
> Use multi-stage builds where possible to include minimal content within your image. Consider using an Alpine base image if possible.

In the next section, we will look at managing images within Docker, some best practices, and some of the most frequently used commands.

Managing Docker images

In modern DevOps practices, Docker images are primarily built either on a developer machine or a CI/CD pipeline. The images are stored in a container registry and then deployed to multiple staging environments and production machines. They might run Docker or a container orchestrator, such as Kubernetes, on top of them.

To efficiently use images, we must understand how to tag them.

Primarily, Docker pulls the image once when you do a Docker run. This means that once an image with a particular version is on the machine, Docker will not attempt to pull it on every run unless you explicitly pull it.

To pull the image explicitly, you can use the `docker pull` command:

```
$ docker pull nginx
Using default tag: latest
```

```
latest: Pulling from library/nginx
f03b40093957: Pull complete
eed12bbd6494: Pull complete
fa7eb8c8eee8: Pull complete
7ff3b2b12318: Pull complete
0f67c7de5f2c: Pull complete
831f51541d38: Pull complete
Digest: sha256:af296b18…
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
```

Now, if we attempt to launch a container using this image, it will instantly launch the container without pulling the image:

```
$ docker run nginx
/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
…
2023/06/10 08:09:07 [notice] 1#1: start worker processes
2023/06/10 08:09:07 [notice] 1#1: start worker process 29
2023/06/10 08:09:07 [notice] 1#1: start worker process 30
```

So, using the latest tag on an image is a bad idea, and the best practice is to use semantic versions as your tag. There are two primary reasons for this:

- If you build the latest image every time, orchestrators such as Docker Compose and Kubernetes will assume the image is already on your machine and will not pull your image by default. Using an image pull policy such as `Always` on Kubernetes or a script to pull the image is a waste of network bandwidth. It is also important to note that Docker Hub limits the number of pulls you can make on open source images, so you must limit your pulls to only when necessary.
- Docker tags allow you to roll out or roll back your container deployment quickly. If you always use the latest tag, the new build overrides the old one, so there is no way you can roll back a faulty container to the last known good version. Using versioned images in production is also a good idea to ensure your container's stability. If, for some reason, you lose the local image and decide to rerun your container, you may not get the same version of the software you were already running, as the latest tag changes frequently. So, it's best to use a particular container version in production for stability.

Images comprise multiple layers, and most of the time, there is a relationship between various versions of containers that run on your server. With time, new versions of images roll out in your production environment, so removing the old images by doing some housekeeping is best. This will reclaim some valuable space the container images occupy, resulting in a cleaner filesystem.

To remove a particular image, you can use the `docker rmi` command:

```
$ docker rmi nginx
Error response from daemon: conflict: unable to remove repository reference "nginx" (must
force) - container d5c84356116f is using its referenced image f9c14fe76d50
```

Oh! We get an error, but why? It's because we have a container running and using this image.

> **Tip**
> You cannot remove images currently used by a running container.

First, you will have to stop and remove the container. Then, you can remove the image using the preceding command. If you want to do everything at once, you can force removal by using the `-f` flag, which will stop the container, remove it, and then remove the image. So, unless you know what you are doing, do not use the `-f` flag:

```
$ docker rmi -f nginx
Untagged: nginx:latest
Untagged: nginx@sha256:af296b18…
Deleted: sha256:f9c14fe7…
```

We built our container many times, but what should we do if we need to push it to Docker Hub or other registries? But before we do that, we will have to authenticate it with Docker Hub using the following command:

```
$ docker login
```

Now, you can push the image to Docker Hub using the following command:

```
$ docker push <your_dockerhub_user>/nginx-hello-world:latest
The push refers to repository [docker.io/<your_dockerhub_user>/nginx-hello-world]
2b7de406bdcd: Pushed
5f70bf18a086: Pushed
845348333310: Pushed
96a9e6a097c6: Pushed
548a79621a42: Mounted from library/ubuntu
latest: digest: sha256:11ec56f0… size: 1366
```

This has pushed four layers and mounted the rest from Ubuntu. We used Ubuntu as the base image, which is already available on Docker Hub.

If you have multiple tags for the image and you want to push all of them, then you can use the `-a` or `--all-tags` option in the `push` command. This will push all the tags for that particular image:

```
$ docker push -a <your_dockerhub_user>/go-hello-world
The push refers to repository [docker.io/<your_dockerhub_user>/go-hello-world]
9d61dbd763ce: Pushed
```

```
5f70bf18a086: Mounted from <your_dockerhub_user>/nginx-hello-world
bb01bd7e32b5: Mounted from library/alpine
multi_stage: digest: sha256:9e1067ca… size: 945
445ef31efc24: Pushed
d810ccdfdc04: Pushed
5f70bf18a086: Layer already exists
70ef08c04fa6: Mounted from library/golang
41cf9ea1d6fd: Mounted from library/golang
d4ebbc3dd11f: Mounted from library/golang
b4b4f5c5ff9f: Mounted from library/golang
b0df24a95c80: Mounted from library/golang
974e52a24adf: Mounted from library/golang
single_stage: digest: sha256:08b5e52b… size: 2209
```

When your build fails for some reason and you make changes to your Dockerfile, it's possible that the old images' layers will remain dangling. Therefore, it is best practice to prune the dangling images at regular intervals. You can use `docker images prune` for this:

```
$ docker images prune
REPOSITORY  TAG  IMAGE ID  CREATED  SIZE
```

In the next section, we'll look at another way to improve Docker image efficiency: flattening Docker images.

Flattening Docker images

Docker inherently uses a layered filesystem, and we have already discussed why it is necessary and how it is beneficial in depth. However, in some particular use cases, Docker practitioners have observed that a Docker image with fewer layers performs better. You can reduce layers in an image by flattening it. However, it is still not a best practice, and you need to do this only if you see a performance improvement, as this would result in a filesystem overhead.

To flatten a Docker image, follow these steps:

1. Run a Docker container with the usual image.
2. Do a `docker export` of the running container to a `.tar` file.
3. Do a `docker import` of the `.tar` file into another image.

Let's use the `nginx-hello-world` image to flatten it and export it to another image; that is, `<your_dockerhub_user>/nginx-hello-world:flat`.

Before we move on, let's get the history of the latest image:

```
$ docker history <your_dockerhub_user>/nginx-hello-world:latest
IMAGE          CREATED      CREATED BY                 SIZE            COMMENT
bba3123dde01   2 hours ago  HEALTHCHECK &
                            {["CMD-SHELL"
                            "curl -f localhos…         0B              buildkit.dockerfile.v0
<missing>      2 hours ago  STOPSIGNAL                 0B
                            SIGTERM                    0B              buildkit.dockerfile.v0
<missing>      2 hours ago  CMD ["nginx"
                            "-g" "daemon off;"]        0B              buildkit.dockerfile.v0
<missing>      2 hours ago  EXPOSE map[80/
                            tcp:{}]                    0B              buildkit.dockerfile.v0
<missing>      2 hours ago  ADD index.html ./ #
                            buildkit                   44B             buildkit.dockerfile.v0
<missing>      2 hours ago  WORKDIR /var/www/
                            html/                      0B              buildkit.dockerfile.v0
<missing>      2 hours ago  RUN /bin/sh -c apt
                            update && apt              57.2MB          buildkit.dockerfile.v0
                            install -y…
<missing>      2 hours ago  RUN /bin/sh -c apt
                            update && apt              59.8MB          buildkit.dockerfile.v0
                            install -y…
<missing>      10 days ago  /bin/sh -c #(nop)          0B
                            CMD ["/bin/bash"]
<missing>      10 days ago  /bin/sh -c #(nop) ADD      63.2MB
                            file:3c74e7e08cbf9a876…
<missing>      10 days ago  /bin/sh -c #(nop)  LABEL   0B
                            org.opencontainers.…
<missing>      10 days ago  /bin/sh -c #(nop)  LABEL   0B
                            org.opencontainers.…
<missing>      10 days ago  /bin/sh -c #(nop)  ARG     0B
                            LAUNCHPAD_BUILD_ARCH
<missing>      10 days ago  /bin/sh -c #(nop)          0B
                            ARG RELEASE
```

Now, let's run a Docker image with the latest image:

```
$ docker run -d --name nginx <your_dockerhub_user>/nginx-hello-world:latest
e2d0c4b884556a353817aada13f0c91ecfeb01f5940e91746f168b…
```

Next, let's take an export out of the running container:

```
$ docker export nginx > nginx-hello-world-flat.tar
```

Import `nginx-hello-world-flat.tar` to a new image; that is, `<your_dockerhub_user>/nginx-hello-world:flat`:

```
$ cat nginx-hello-world-flat.tar |  \
docker import - <your_dockerhub_user>/nginx-hello-world:flat
sha256:57bf5a9ada46191ae1aa16bcf837a4a80e8a19d0bcb9fc…
```

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

```
$ docker images
REPOSITORY               TAG     IMAGE ID      CREATED     SIZE
<your_dockerhub_user>/   flat    57bf5a9ada46  34 seconds  177MB
nginx-hello-world                              ago
<your_dockerhub_user>/   latest  bba3123dde01  2 hours
nginx-hello-world                              ago         180MB
```

Here, we can see that the flat image is present and that it occupies less space than the latest image. If we view its history, we should see just a single layer:

```
$ docker history <your_dockerhub_user>/nginx-hello-world:flat
IMAGE         CREATED              CREATED BY  SIZE   COMMENT
57bf5a9ada46  About a minute ago                177MB  Imported from -
```

It has flattened the image. But is it a best practice to flatten Docker images? Well, it depends. Let's understand when and how to flatten Docker images and what you should consider:

- Are several applications using a similar base image? If that is the case, then flattening images will only increase the disk footprint, as you won't be able to take advantage of a layered filesystem.
- Consider alternatives to flattening images using a small base image, such as Alpine.
- Multi-stage builds are helpful for most complied languages and can reduce your image's size considerably.
- You can also slim down images by using as few layers as possible by combining multiple steps into a single `RUN` directive.
- Consider whether the benefits of flattening the image outweigh the disadvantages, whether you'll get considerable performance improvements, and whether performance is critical for your application needs.

These considerations will help you understand your container image footprint and help you manage container images. Remember that although reducing the image's size is ideal, flattening it should be a last resort.

So far, all the images we've used have been derived from a Linux distribution and always used a distro as their base image. You can also run a container without using a Linux distro as the base image to make it more secure. We'll have a look at how in the next section.

Optimizing containers with distroless images

Distroless containers are one of the latest trends in the container world. They are promising because they consider all the aspects of optimizing containers for the Enterprise environment. You should consider three important things while optimizing containers – performance, security, and cost.

Performance

You don't make containers out of thin air. You must download images from your container registry and then run the container out of the image. Each step uses network and disk I/O. The bigger the image, the more resources it consumes and the less performance you get from it. Therefore, a smaller Docker image naturally performs better.

Security

Security is one of the most important aspects of the current IT landscape. Companies usually focus on this aspect and invest a lot of money and time. Since containers are a relatively new technology, they are vulnerable to hacking, so appropriately securing your containers is important. Standard Linux distributions have a lot of stuff that can allow hackers to access more than they could have if you secured your container properly. Therefore, you must ensure you only have what you need within the container.

Cost

A smaller image also results in a lower cost. The lower your container footprint, the more containers you can pack within a machine, so there are fewer machines you would need to run your applications. This means you save a lot of money that would accumulate over time.

As a modern DevOps engineer, you must ensure your images are optimized for all these aspects. Distroless images help take care of all of them. Therefore, let's understand what distroless images are and how to use them.

Distroless images are the most minimal images and only contain your application, dependencies, and the necessary files for your container process to run. Most of the time, you do not need package managers such as `apt` or a shell such as `bash`. Not having a shell has its advantages. For one, it will help you avoid any outside party gaining access to your container while it is running. Your container has a small attack surface and won't have many security vulnerabilities.

Google provides distroless images in their official GCR registry, available on their GitHub page at `https://github.com/GoogleContainerTools/distroless`. Let's get hands-on and see what we can do with them.

The required resources for this exercise are in `ch4/go-hello-world/distroless` in this book's GitHub repository.

Let's start by creating a Dockerfile:

```
FROM golang:1.20.5 AS build
WORKDIR /tmp
COPY app.go .
RUN GO111MODULE=off GOOS=linux go build -a -installsuffix cgo -o app . && chmod +x ./app
FROM gcr.io/distroless/base
WORKDIR /tmp
```

```
COPY --from=build /tmp/app .
CMD ["./app"]
```

This Dockerfile is similar to the multi-stage build Dockerfile for the `go-hello-world` container, but instead of using `alpine`, it uses `gcr.io/distroless/base` as the base image. This image contains a minimalistic Linux glibc-enabled system and lacks a package manager or a shell. You can use it to run binaries compiled in a language such as Go, Rust, or D.

So, let's build this first using the following command:

```
$ docker build -t <your_dockerhub_user>/go-hello-world:distroless .
[+] Building 7.6s (14/14) FINISHED
 => [internal] load build definition from Dockerfile 0.0s
 => => transferring dockerfile: 268B 0.0s
 => [internal] load .dockerignore 0.0s
 => => transferring context: 2B 0.0s
 => [internal] load metadata for gcr.io/distroless/base:latest 3.1s
 => [internal] load metadata for docker.io/library/golang:1.20.5 1.4s
 => [auth] library/golang:pull token for registry-1.docker.io 0.0s
 => [stage-1 1/3] FROM gcr.io/distroless/base@
sha256:73deaaf6a207c1a33850257ba74e0f196bc418636cada9943a03d7abea980d6d 3.2s
 => [build 1/4] FROM docker.io/library/golang:1.20.5@sha256:4b1fc02d 0.0s
 => [internal] load build context 0.0s
 => => transferring context: 108B 0.0s
 => CACHED [build 2/4] WORKDIR /tmp 0.0s
 => CACHED [build 3/4] COPY app.go . 0.0s
 => CACHED [build 4/4] RUN GO111MODULE=off GOOS=linux go build -a -installsuffix cgo -o
app . && chmod +x ./app 0.0s
 => [stage-1 2/3] WORKDIR /tmp 0.9s
 => [stage-1 3/3] COPY --from=build /tmp/app . 0.3s
 => exporting to image 0.1s
 => => exporting layers 0.1s
 => => writing image sha256:51ced401 0.0s
 => => naming to docker.io/<your_dockerhub_user>/go-hello-world:distroless
```

Now, let's run this image and see what we get:

```
$ docker run <your_dockerhub_user>/go-hello-world:distroless
Hello, World!
```

It works! Let's look at the size of the image:

```
$ docker images
REPOSITORY                              TAG         MAGE ID       CREATED         SIZE
<your_dockerhub_user>/go-hello-world  distroless  51ced401d7bf  6 minutes ago  22.3MB
```

It's just 22.3 MB. Yes, it's a bit more than the Alpine image, but it does not contain a shell, so it is more secure from that point of view. Also, there are distroless images available for interpreted programming languages, such as Python and Java, that you can use instead of the bloated image containing the toolkits

Docker images are stored in Docker registries, and we have all been using Docker Hub for a while. In the next section, we'll understand what they are and what our options are for storing our images.

Understanding Docker registries

A **Docker registry** is a stateless, highly scalable server-side application that stores and lets you distribute Docker images. The registry is open source under the permissive **Apache license**. It is a storage and distribution system where all your Docker servers can connect and upload and download images as and when needed. It acts as a distribution site for your images.

A Docker registry contains several Docker repositories. A Docker repository holds several versions of a specific image. For example, all the versions of the `nginx` image are stored within a single repository within Docker Hub called `nginx`.

By default, Docker interacts with its public Docker registry instance, called Docker Hub, which helps you distribute your images to the broader open source community.

Not all images can be public and open source, and many proprietary activities are ongoing. Docker allows you to use a private Docker registry for a scenario you can host within your infrastructure called **Docker Trusted Registry**. Several online options are available, including using a SaaS service, such as GCR, or creating private repositories at Docker Hub.

While the SaaS option is readily available and intuitive, let's consider hosting our private Docker registry.

Hosting your private Docker registry

Docker provides an image that you can run on any server that has Docker installed. Once the container is up and running, you can use that as the Docker registry. Let's have a look:

```
$ docker run -d -p 80:5000 --restart=always --name registry registry:2
Unable to find image 'registry:2' locally
2: Pulling from library/registry
8a49fdb3b6a5: Already exists
58116d8bf569: Pull complete
4cb4a93be51c: Pull complete
cbdeff65a266: Pull complete
6b102b34ed3d: Pull complete
Digest: sha256:20d08472…
Status: Downloaded newer image for registry:2
ae4c4ec9fc7b17733694160b5b3b053bd1a41475dc4282f3eccaa10…
```

Since we know that the registry is running on localhost and listening on port `80`, let's try to push an image to this registry. First, let's tag the image to specify `localhost` as the registry. We will add a registry location at the beginning of the Docker tag so that Docker knows where to push the image. We already know that the structure of a Docker tag is `<registry_url>/<user>/<image_name>:<image_version>`. We will use the `docker tag` command to give another name to an existing image, as shown in the following command:

```
$ docker tag your_dockerhub_user>/nginx-hello-world:latest \
localhost/<your_dockerhub_user>/nginx-hello-world:latest
```

Now, we can go ahead and push the image to the local Docker registry:

```
$ docker push localhost/<your_dockerhub_user>/nginx-hello-world:latest
The push refers to repository [localhost/your_dockerhub_user/nginx-hello-world]
2b7de406bdcd: Pushed
5f70bf18a086: Pushed
845348333310: Pushed
96a9e6a097c6: Pushed
548a79621a42: Pushed
latest: digest: sha256:6ad07e74… size: 1366
```

And that's it! It is as simple as that!

There are other considerations as well since this is too simplistic. You will also have to mount volumes; otherwise, you will lose all the images when you restart the registry container. Also, there is no authentication in place, so anyone accessing this server can push or pull images, but we don't desire this. Also, communication is insecure, and we want to encrypt the images during transit.

First, let's create the local directories that we will mount to the containers:

```
$ sudo mkdir -p /mnt/registry/certs
$ sudo mkdir -p /mnt/registry/auth
$ sudo chmod -R 777 /mnt/registry
```

Now, let's generate an `htpasswd` file for adding authentication to the registry. For this, we will run the `htpasswd` command from within a new Docker registry container to create a file on our local directory:

```
$ docker run --entrypoint htpasswd registry:2.7.0 \
-Bbn user pass > /mnt/registry/auth/htpasswd
```

The next step is to generate some self-signed certificates for enabling TLS on the repository. Add your server name or IP when asked for a **Fully Qualified Domain Name** (**FQDN**). You can leave the other fields blank or add appropriate values for them:

```
$ openssl req -newkey rsa:4096 -nodes -sha256 -keyout \
/mnt/registry/certs/domain.key -x509 -days 365 -out /mnt/registry/certs/domain.crt
```

Before we proceed further, let's remove the existing registry:

```
$ docker rm -f registry
registry
```

Now, we are ready to launch our container with the required configuration:

```
$ docker run -d -p 443:443 --restart=always \
--name registry \
```

```
  -v /mnt/registry/certs:/certs \
  -v /mnt/registry/auth:/auth \
  -v /mnt/registry/registry:/var/lib/registry \
  -e REGISTRY_HTTP_ADDR=0.0.0.0:443 \
  -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
  -e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \
  -e REGISTRY_AUTH=htpasswd \
  -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" \
  -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
  registry:2
02bf92c9c4a6d1d9c9f4b75ba80e82834621b1570f5f7c4a74b215960
```

The container is now up and running. Let's use `https` this time, but before that, let's `docker login` to the registry. Add the username and password you set while creating the `htpasswd` file (in this case, `user` and `pass`):

```
$ docker login https://localhost
Username: user
Password:
WARNING! Your password will be stored unencrypted in /root/
.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/
#credentials-store
Login Succeeded
```

Since the login succeeded, we can go ahead and push our image to the registry:

```
$ docker push localhost/<your_dockerhub_user>/nginx-hello-world
The push refers to repository [localhost/<your_dockerhub_user>/nginx-hello-world]
2b7de406bdcd: Pushed
5f70bf18a086: Pushed
845348333310: Pushed
96a9e6a097c6: Pushed
548a79621a42: Pushed
latest: digest: sha256:6ad07e7425331456a3b8ea118bce36c82af242ec14072d483b5dcaa3bd607e65
size: 1366
```

This time, it works the way we want it to.

Other public registries

Apart from running your registry in a dedicated Docker server, other cloud and on-premises options exist.

Most public cloud providers offer paid online registries and container-hosting solutions that you can easily use while running in the cloud. Some of them are as follows:

- **Amazon Elastic Container Registry** (**ECR**): This is a popular AWS offering you can use if your infrastructure runs on AWS. It is a highly available, highly performant, fully managed solution.

It can host public and private registries, and you only pay for the storage you consume and the amount of data transferred to the internet. The best part is that it integrates with AWS IAM.

- **Google Container Registry** (**GCR**): Backed by **Google Cloud Storage** (**GCS**), GCR is one of the best choices if you run your infrastructure on GCP. It hosts both public and private repositories, and you only pay for the storage on GCS.
- **Azure Container Registry** (**ACR**): This fully managed, geo-replicated container registry only supports a private registry. It is a good option if you are running your infrastructure on Azure. Besides storing container images, it also stores Helm charts and other artifacts that help you manage your containers.
- **Oracle Cloud Infrastructure Registry**: Oracle Cloud Infrastructure Registry is a highly available Oracle-managed container registry. It can host both public and private repositories.
- **CoreOS Quay**: This supports OAuth and LDAP authentication. It offers both (paid) private and (free) public repositories, automatic security scanning, and automated image builds via integration with GitLab, GitHub, and Bitbucket.

If you don't want to go with managed options in the cloud or run on-premises, you can also use distribution management software such as *Sonatype Nexus* or *JFrog Artifactory*. Both tools support Docker registries out of the box. You can create a Docker registry there using fancy UIs, and then use `docker login` to connect to the registry.

Summary

In this chapter, we have covered a lot of ground. At this point, you should understand Docker from a hands-on perspective. We started with Docker images, how to use a Dockerfile to build Docker images, the components and directives of the Dockerfile, and how to create efficient images by following some best practices. We also discussed flattening Docker images and improving container security using distroless images. Finally, we discussed Docker registries, how to run a private Docker registry on a Docker server, and how to use other turnkey solutions, such as Sonatype Nexus and JFrog Artifactory.

Here is a quick summary of some best practices for managing Docker containers effectively and efficiently:

- **Use Official Images**: Whenever possible, start with official Docker images from reputable sources such as Docker Hub. These images are well-maintained, regularly updated, and often come with better security practices.
- **Minimize Containers**: Follow the "one service per container" principle. Each container should have a single responsibility, which helps with maintainability and scaling.
- **Optimize Container Sizes**: Keep containers as lightweight as possible. Use Alpine Linux or other minimal base images and remove unnecessary files and dependencies.
- **Use Environment Variables**: Store configuration and sensitive data in environment variables rather than hardcoding it into the container. This enhances portability and security.

- **Persistent Data**: Store application data outside containers using Docker volumes or bind mounts. This ensures that data persists even if containers are replaced or stopped.
- **Container Naming**: Give containers meaningful and unique names. This helps with easy identification and troubleshooting.
- **Resource Limits**: Set resource limits (CPU and memory) for containers to prevent one misbehaving container from affecting others on the same host.
- **Container Restart Policies**: Define restart policies to determine how containers should behave when they exit or crash. Choose the appropriate policy based on your application's requirements.
- **Docker Compose**: Use Docker Compose to define and manage multi-container applications. It simplifies the deployment and orchestration of complex setups.
- **Network Isolation**: Use Docker networks to isolate containers and control communication between them. This enhances security and manageability.
- **Health Checks**: Implement health checks in your containers to ensure they run as expected. This helps with automated monitoring and recovery.
- **Container Logs**: Redirect container logs to standard output (`stdout`) and standard error (`stderr`) streams. This makes it easier to collect and analyze logs using Docker's logging mechanisms.
- **Security Best Practices**: Keep containers up to date with security patches, avoid running containers as the root, and follow security best practices to avoid vulnerabilities.
- **Version Control Dockerfiles**: Store Dockerfiles in version control systems (e.g., Git) and regularly review and update them.
- **Container Cleanup**: Regularly remove unused containers, images, and volumes to free up disk space. Consider using tools such as Docker's built-in prune commands.
- **Orchestration Tools**: Explore container orchestration tools such as Kubernetes or Docker Swarm for managing larger and more complex container deployments.
- **Documentation**: Maintain clear and up-to-date documentation for your containers and images, including how to run them, their required environment variables, and any other configuration details.
- **Backup and Restore**: Establish backup and restore processes for container data and configuration to recover them quickly in case of failures.
- **Monitoring and Scaling**: Implement monitoring and alerting for your containers to ensure they run smoothly. Use scaling mechanisms to handle the increased load.

By following these best practices, you can ensure that your Docker container environment is well-organized, secure, maintainable, and scalable.

In the next chapter, we will delve into container orchestration using Kubernetes.

Questions

1. Docker images use a layered model. (True/False)
2. You can delete an image from a server if a container using that image is already running. (True/False)
3. How do you remove a running container from a server? (Choose two)

   A. `docker rm <container_id>`

   B. `docker rm -f <container_id>`

   C. `docker stop <container_id> && docker rm <container_id>`

   D. `docker stop -f <container_id>`
4. Which of the following options are container build best practices? (Choose four)

   A. Always add layers that don't frequently change at the beginning of the Dockerfile.

   B. Combine multiple steps into a single directive to reduce layers.

   C. Only use the required files in the container to keep it lightweight and reduce the attack surface.

   D. Use semantic versioning in your Docker tags and avoid the latest version.

   E. Include package managers and a shell within the container, as this helps with troubleshooting a running container.

   F. Only use an `apt update` at the start of your Dockerfile.
5. You should always flatten Docker images to a single layer. (True/False)
6. A distroless container contains a shell. (True/False)
7. What are some of the ways to improve container efficiency? (Choose four)

   A. Try to use a smaller base image if possible, such as Alpine.

   B. Only use multi-stage builds to add the required libraries and dependencies to the container and omit heavyweight toolkits that are not necessary.

   C. Use distroless base images where possible.

   D. Flatten Docker images.

   E. Use single-stage builds to include package managers and a shell, as this will help in troubleshooting in production.
8. It is a best practice to prune Docker images from time to time. (True/False)
9. Health checks should always be baked into your Docker image. (True/False)

Answers

1. True
2. False – you cannot delete an image that is being used by a running container.
3. B, C
4. A, B, C, D
5. False – only flatten Docker images if you'll benefit from better performance.
6. False – distroless containers do not contain a shell.
7. A, B, C, D
8. True
9. False – if you’re using Kubernetes or Docker Compose, use the liveness probes or define health checks with a YAML file instead.

5
Container Orchestration with Kubernetes

In the previous chapter, we covered creating and managing container images, where we discussed container images, Dockerfiles, and their directives and components. We also looked at the best practices for writing a Dockerfile and building and managing efficient images. We then looked at flattening Docker images and investigated in detail distroless images to improve container security. Finally, we created a private Docker registry.

Now, we will deep dive into container orchestration. We will learn how to schedule and run containers using the most popular container orchestrator – Kubernetes.

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

- What is Kubernetes, and why do I need it?
- Kubernetes architecture
- Installing Kubernetes (Minikube and KinD)
- Understanding Kubernetes pods

Technical requirements

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

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

Run the following command to clone the repository into your home directory, and `cd` into the `ch5` directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
  modern-devops
$ cd modern-devops/ch5
```

As the repository contains files with placeholders, you must replace the `<your_dockerhub_user>` string with your actual Docker Hub user. Use the following commands to substitute the placeholders:

```
$ grep -rl '<your_dockerhub_user>' . | xargs sed -i -e \
 's/<your_dockerhub_user>/<your actual docker hub user>/g'
```

What is Kubernetes, and why do I need it?

By now, you should understand what containers are and how to build and run containers using Docker. However, how we ran containers using Docker was not optimal from a production standpoint. Let me give you a few considerations to think about:

- As portable containers can run on any Docker machine just fine, multiple containers also share server resources to optimize resource consumption. Now, think of a microservices application that comprises hundreds of containers. How will you choose what machine to run the containers on? What if you want to dynamically schedule the containers to another machine based on resource consumption?
- Containers provide horizontal scalability as you can create a copy of the container and use a **load balancer** in front of a pool of containers. One way of doing this is to decide upfront and deploy the desired number of containers, but that isn't optimal resource utilization. What if I tell you that you need to horizontally scale your containers dynamically with traffic – in other words, by creating additional container instances to handle the extra load when there is more traffic and reducing them when there is less?
- There are container health check reports on the containers' health. What if the container is unhealthy, and you want to auto-heal it? What would happen if an entire server goes down and you want to schedule all containers running on that server to another?
- As containers mostly run within a server and can see each other, how would I ensure that only the required containers can interact with the other, something we usually do with VMs? We cannot compromise on security.
- Modern cloud platforms allow us to run autoscaling VMs. How can we utilize that from the perspective of containers? For example, if I need just one VM for my containers during the night and five during the day, how can I ensure that the machines are dynamically allocated when we need them?

- How do you manage the networking between multiple containers if they are part of a more comprehensive service mesh?

The answer to all these questions is a container orchestrator, and the most popular and *de facto* standard for that is Kubernetes.

Kubernetes is an open source container orchestrator. A bunch of Google engineers first developed it and then open sourced it to the **Cloud Native Computing Foundation** (**CNCF**). Since then, the buzz around Kubernetes has not subsided, and for an excellent reason – Kubernetes with containers has changed the technology mindset and how we look at infrastructure entirely. Instead of treating servers as dedicated machines to an application or as part of an application, Kubernetes has allowed visualizing servers as an entity with a container runtime installed. When we treat servers as a standard setup, we can run virtually anything in a cluster of servers. So, you don't have to plan for **high availability** (**HA**), **disaster recovery** (**DR**), and other operational aspects for every application on your tech stack. Instead, you can cluster all your servers into a single unit – a Kubernetes cluster – and containerize all your applications. You can then offload all container management functions to Kubernetes. You can run Kubernetes on bare-metal servers, VMs, and as a managed service in the cloud through multiple Kubernetes-as-a-Service offerings.

Kubernetes solves these problems by providing HA, scalability, and zero downtime out of the box. It essentially performs the following functions to provide them:

- **Provides a centralized control plane for interacting with it**: The API server exposes a list of useful APIs that you can interact with to invoke many Kubernetes functions. It also provides a Kubernetes command line called **kubectl** to interact with the API using simple commands. Having a centralized control plane ensures that you can interact with Kubernetes seamlessly.
- **Interacts with the container runtime to schedule containers**: When we send the request to schedule a container to **kube-apiserver**, Kubernetes decides what server to schedule the container based on various factors and then interacts with the server's container runtime through the **kubelet** component.
- **Stores the expected configuration in a key-value data store**: Kubernetes applies the cluster's anticipated configuration and stores that in a key-value data store – **etcd**. That way, Kubernetes continuously ensures that the containers within the cluster remain in the desired state. If there is any deviation from the expected state, Kubernetes will take every action to bring it back to the desired configuration. That way, Kubernetes ensures that your containers are up and running and healthy.
- **Provides a network abstraction layer and service discovery**: Kubernetes uses a network abstraction layer to allow communication between your containers. Therefore, every container is allocated a virtual IP, and Kubernetes ensures a container is reachable from another container running on a different server. It provides the necessary networking by using an **overlay network** between the servers. From the container's perspective, all containers in the cluster behave as if they are running on the same server. Kubernetes also uses a **DNS** to allow communication

between containers through a domain name. That way, containers can interact with each other by using a domain name instead of an IP address to ensure that you don't need to change the configuration if a container is recreated and the IP address changes.

- **Interacts with the cloud provider**: Kubernetes interacts with the cloud provider to commission objects such as **load balancers** and **persistent disks**. So, if you tell Kubernetes that your application needs to persist data and define a **volume**, Kubernetes will automatically request a disk from your cloud provider and mount it to your container wherever it runs. You can also expose your application on an external load balancer by requesting Kubernetes. Kubernetes will interact with your cloud provider to spin up a load balancer and point it to your containers. That way, you can do everything related to containers by merely interacting with your Kubernetes API server.

Kubernetes comprises multiple moving parts that take over each function we've discussed. Now, let's look at the Kubernetes architecture to understand each of them.

Kubernetes architecture

Kubernetes is made of a cluster of nodes. There are two possible roles for nodes in Kubernetes – **control plane** nodes and **worker** nodes. The control plane nodes control the Kubernetes cluster, scheduling the workloads, listening to requests, and other aspects that help run your workloads and make the cluster function. They typically form the brain of the cluster.

On the other hand, the worker nodes are the powerhouses of the Kubernetes cluster and provide raw computing for running your container workloads.

Kubernetes architecture follows the client-server model via an API server. Any interaction, including internal interactions between components, happens via the Kubernetes API server. Therefore, the Kubernetes API server is known as the brain of the Kubernetes control plane.

There are other components of Kubernetes as well, but before we delve into the details, let's look at the following diagram to understand the high-level Kubernetes architecture:

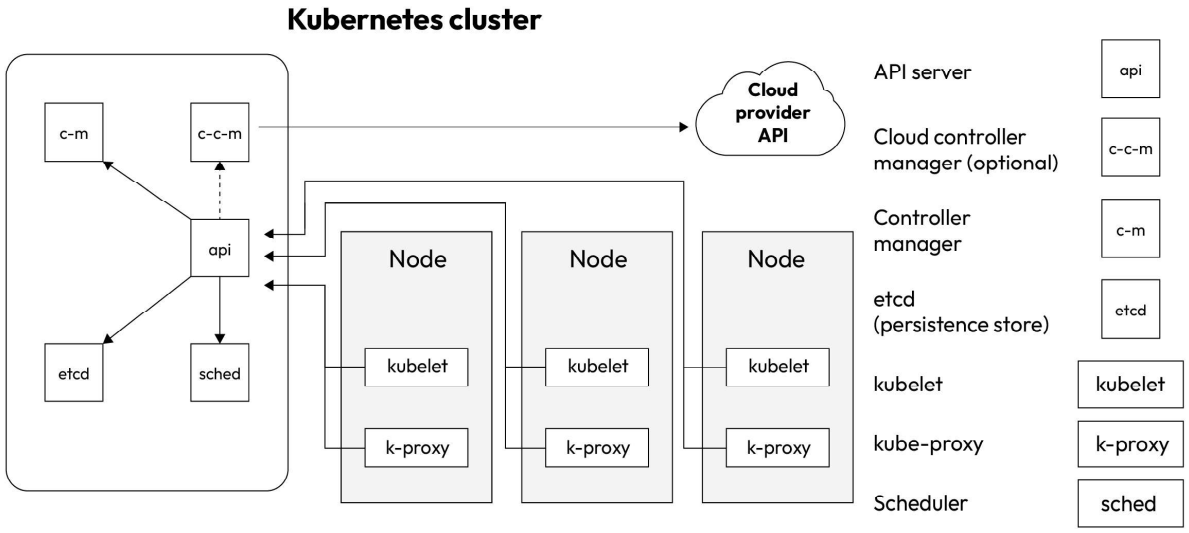

Figure 5.1 – Kubernetes cluster architecture

The control plane comprises the following components:

- **API server**: As discussed previously, the API server exposes a set of APIs for external and internal actors to interact with Kubernetes. All interactions with Kubernetes happen via the API server, as evident from the preceding diagram. If you visualize the Kubernetes cluster as a ship, the API server is the ship's captain.
- **Controller manager**: The controller manager is the ship's executive officer and is tasked with ensuring that the captain's orders are followed in the cluster. From a technical perspective, the controller manager reads the current and desired states and takes all actions necessary to move the current state to the desired state. It contains a set of controllers that interact with the Kubernetes components via the API server as and when needed. Some of these are as follows:
  - **Node controller**: This watches for when the node goes down and responds by interacting with the **Kube scheduler** via the **Kube API server** to schedule the pods to a healthy node.
  - **Replication controller**: This ensures that the correct amount of container replicas defined by replication controller objects in the cluster exist.
  - **Endpoints controller**: These assist in providing endpoints to your containers via services.
  - **Service account and token controllers**: These create default **accounts** and **tokens** for new **namespaces**.
- **Cloud controller manager**: This is an optional controller manager that you would run if you run Kubernetes in a public cloud, such as **AWS**, **Azure**, or **GCP**. The cloud controller manager interacts with the cloud provider APIs to provision resources such as **persistent disks** and **load balancers** that you declare in your Kubernetes configuration.

- **etcd**: **etcd** is the log book of the ship. That is where all the details about the expected configuration exist. From a technical perspective, this is a key-value store where all the desired Kubernetes configuration is stored. The controller manager refers to the information in this database to action changes in the cluster.
- **Scheduler**: The schedulers are the boatswain of the ship. They are tasked with supervising the process of loading and unloading containers on the ship. A Kubernetes scheduler schedules containers in a worker node it finds fit after considering the availability of resources to run it, the HA of your application, and other aspects.
- **kubelet**: kubelets are the seamen of the ship. They carry out the actual loading and unloading of containers from a ship. From a technical perspective, the kubelet interacts with the underlying container runtime to run containers on the scheduler's instruction. While most Kubernetes components can run as a container, the kubelet is the only component that runs as a **systemd** service. They usually run on worker nodes, but if you plan to run the control plane components as containers instead, the kubelet will also run on the control plane nodes.
- **kube-proxy**: **kube-proxy** runs on each worker node and provides the components for your containers to interact with the network components inside and outside your cluster. They are vital components that facilitate Kubernetes networking.

Well, that's a lot of moving parts, but the good news is that tools are available to set that up for you, and provisioning a Kubernetes cluster is very simple. If you are running on a public cloud, it is only a few clicks away, and you can use your cloud's web UI or CLI to provision it very quickly. You can use **kubeadm** for the setup if you have an on-premises installation. The steps are well documented and understood and won't be too much of a hassle.

For development and your CI/CD environments, you can use **Minikube** or **Kubernetes in Docker** (**KinD**). While Minikube can run a single-node Kubernetes cluster on your development machine directly by using your machine as the node, it can also run a multi-node cluster by running Kubernetes nodes as containers. KinD, on the other hand, exclusively runs your nodes as containers on both single-node and multi-node configurations. You need a VM with the requisite resources in both cases, and you'll be good to go.

In the next section, we'll boot a single-node Kubernetes cluster with Minikube.

Installing Kubernetes (Minikube and KinD)

Now, let's move on and install Kubernetes for your development environment. We will begin with Minikube to get you started quickly and then look into KinD. We will then use KinD for the rest of this chapter.

Installing Minikube

We will install Minikube in the same Linux machine we used to install Docker in *Chapter 3*, *Containerization with Docker*. So, if you haven't done that, please go to *Chapter 3*, *Containerization with Docker*, and follow the instructions provided to set up Docker on your machine.

First, we will install **kubectl**. As described previously, kubectl is the command-line utility that interacts with the Kubernetes API server. We will use kubectl multiple times in this book.

To download the latest release of kubectl, run the following command:

```
$ curl -LO "https://storage.googleapis.com/kubernetes-release/release\
/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)\
/bin/linux/amd64/kubectl"
```

You can also download a specific version of kubectl. To do so, use the following command:

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release\
/v<kubectl_version>/bin/linux/amd64/kubectl
```

We will stick with the latest release for this chapter. Now, let's go ahead and make the binary executable and then move it to any directory in your system `PATH`:

```
$ chmod +x ./kubectl
$ sudo mv kubectl /usr/local/bin/
```

Now, let's check whether kubectl has been successfully installed by running the following command:

```
$ kubectl version --client
Client Version: version.Info{Major:"1", Minor:"27", GitVersion:"v1.27.3"}
```

Since kubectl was installed successfully, you must download the `minikube` binary and then move it to your system path using the following commands:

```
$ curl -Lo minikube \
https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
$ chmod +x minikube
$ sudo mv minikube /usr/local/bin/
```

Now, let's install the packages required by Minikube to function correctly by running the following command:

```
$ sudo apt-get install -y conntrack
```

Finally, we can bootstrap a Minikube cluster using the following command:

```
$ minikube start --driver=docker
Done! kubectl is now configured to use "minikube" cluster and "default" namespace by
default
```

As Minikube is now up and running, we will use the kubectl command-line utility to interact with the Kube API server to manage Kubernetes resources. The kubectl commands have a standard structure and are self-explanatory in most cases. They are structured as follows:

```
kubectl <verb> <resource type> <resource name> [--flags]
```

Here, we have the following:

- `verb`: The action to perform – for example, `get`, `apply`, `delete`, `list`, `patch`, `run`, and so on
- `resource type`: The Kubernetes resource to manage, such as `node`, `pod`, `deployment`, `service`, and so on
- `resource name`: The name of the resource to manage

Now, let's use kubectl to get nodes and check whether our cluster is ready to run our containers:

```
$ kubectl get nodes
NAME       STATUS   ROLES           AGE     VERSION
minikube   Ready    control-plane   2m25s   v1.26.3
```

Here, we can see that it is a single-node Kubernetes cluster running version **v1.26.3**. Kubernetes is now up and running!

This setup is excellent for development machines where developers want to deploy and test a single component they are working on.

To stop the Minikube cluster and delete it from the machine, you can use the following command:

```
$ minikube stop
```

Now that we have removed Minikube, let's look at another exciting tool for creating a multi-node Kubernetes cluster.

Installing KinD

KinD allows you to run a multi-node Kubernetes cluster on a single server that runs Docker. We understand that a multi-node Kubernetes cluster requires multiple machines, but how can we run a multi-node Kubernetes cluster on a single server? The answer is simple: KinD uses a Docker container as a Kubernetes node. So, if we need a four-node Kubernetes cluster, KinD will spin up four containers that behave like four Kubernetes nodes. It is as simple as that.

While you need Docker to run KinD, KinD internally uses **containerd** as a container runtime instead of Docker. Containerd implements the container runtime interface; therefore, Kubernetes does not require any specialized components, such as **dockershim**, to interact with it. This means that KinD still works with Kubernetes since Docker isn't supported anymore as a Kubernetes container runtime.

As KinD supports a multi-node Kubernetes cluster, you can use it for your development activities and also in your CI/CD pipelines. In fact, KinD redefines CI/CD pipelines as you don't require a static Kubernetes environment to test your build. KinD is swift to boot up, which means you can integrate the bootstrapping of the KinD cluster, run and test your container builds within the cluster, and then destroy it all within your CI/CD pipeline. This gives development teams immense power and speed.

> **Important**
>
> Never use KinD in production. Docker in Docker implementations are not very secure; therefore, KinD clusters should not exist beyond your dev environments and CI/CD pipelines.

Bootstrapping KinD is just a few commands away. First, we need to download KinD, make it executable, and then move it to the default `PATH` directory using the following commands:

```
$ curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-amd64
$ chmod +x kind
$ sudo mv kind /usr/local/bin/
```

To check whether KinD is installed, we can run the following command:

```
$ kind version
kind v0.20.0 go1.20.4 linux/amd64
```

Now, let's bootstrap a multi-node KinD cluster. First, we need to create a KinD `config` file. The KinD `config` file is a simple YAML file where you can declare what configuration you want for each node. If we need to bootstrap a single control plane and three worker node clusters, we can add the following configuration:

```
$ vim kind-config.yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
- role: worker
- role: worker
```

You can also have an HA configuration with multiple control planes using multiple node items with the control plane role. For now, let's stick with a single control plane, three-worker node configuration.

To bootstrap your KinD cluster with the preceding configuration, run the following command:

```
$ kind create cluster --config kind-config.yaml
```

With that, our KinD cluster is up and running. Now, let's list the nodes to see for certain by using the following command:

```
$ kubectl get nodes
NAME                 STATUS   ROLES           AGE   VERSION
kind-control-plane   Ready    control-plane   72s   v1.27.3
kind-worker          Ready    <none>          47s   v1.27.3
kind-worker2         Ready    <none>          47s   v1.27.3
kind-worker3         Ready    <none>          47s   v1.27.3
```

Here, we can see four nodes in the cluster – one control plane and three workers. Now that the cluster is ready, we'll dive deep into Kubernetes and look at some of the most frequently used Kubernetes resources in the next section.

Understanding Kubernetes pods

Kubernetes pods are the basic building blocks of a Kubernetes application. A pod contains one or more containers, and all containers within a pod are always scheduled in the same host. Usually, there is a single container within a pod, but there are use cases where you need to schedule multiple containers in a single pod.

It takes a while to digest why Kubernetes started with the concept of pods in the first place instead of using containers, but there are reasons for that, and you will appreciate this as you gain more experience with the tool. For now, let's look at a simple example of a pod and how to schedule it in Kubernetes.

Running a pod

We will start by running an NGINX container in a pod using simple imperative commands. We will then look at how we can do this declaratively.

To access the resources for this section, `cd` into the following directory:

```
$ cd ~/modern-devops/ch5/pod/
```

To run a pod with a single NGINX container, execute the following command:

```
$ kubectl run nginx --image=nginx
```

To check whether the pod is running, run the following command:

```
$ kubectl get pod
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          26s
```

And that's it! As we can see, the pod is now running.

To delete the pod, you can run the following command:

```
$ kubectl delete pod nginx
```

The `kubectl run` command was the imperative way of creating pods, but there's another way of interacting with Kubernetes – by using declarative manifests. **Kubernetes manifests** are YAML or JSON files you can use to declare the desired configuration instead of telling Kubernetes everything through a command line. This method is similar to `docker compose`.

> **Tip**
>
> Always use the declarative method to create Kubernetes resources in staging and production environments. They allow you to store and version your Kubernetes configuration in a source code management tool such as Git and enable GitOps. You can use imperative methods during development because commands have a quicker turnaround than YAML files.

Let's look at an example pod manifest, `nginx-pod.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: nginx
  name: nginx
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "200m"
      requests:
        memory: "100Mi"
        cpu: "100m"
  restartPolicy: Always
```

Let's understand the file first. The file contains the following:

- `apiVersion`: This defines the resource version we are trying to define. In this case, as it is a pod and a **generally available** (**GA**) resource, the version we will use is `v1`.
- `kind`: This defines the kind of resource we want to create – a pod.
- `metadata`: The `metadata` section defines the name and labels surrounding this resource. It helps in uniquely identifying the resource and grouping multiple resources using labels.
- `spec`: This is the main section where we define the actual specifications for the resource.

- `spec.containers`: This section defines one or more containers that form the pod.
- `spec.containers.name`: This is the container's name, which is `nginx-container` in this case.
- `spec.containers.image`: This is the container image, which is `nginx` in this case.
- `spec.containers.imagePullPolicy`: This can be `Always`, `IfNotPresent`, or `Never`. If set to `Always`, Kubernetes always pulls the image from the registry. If set to `IfNotPresent`, Kubernetes pulls the image only if the image is not found on the node where the pod is scheduled. If set to `Never`, Kubernetes will never attempt to pull images from the registry and will rely completely on local images.
- `spec.containers.resources`: This defines the resource requests and limits.
- `spec.containers.resources.limit`: This defines the resource limits. This is the maximum amount of resources that the pod can allocate, and if the resource consumption increases beyond it, the pod is evicted.
- `spec.containers.resources.limit.memory`: This defines the memory limit.
- `spec.containers.resources.limit.cpu`: This defines the CPU limit.
- `spec.containers.resources.requests`: This defines the resource requests. This is the minimum amount of resources the pod would request during scheduling and will not be scheduled on a node that cannot allocate it.
- `spec.containers.resources.requests.memory`: This defines the amount of memory to be requested.
- `spec.containers.resources.requests.cpu`: This defines the number of CPU cores to be requested.
- `spec.restartPolicy`: This defines the restart policy of containers – `Always`, `OnFailure`, or `Never`. This is similar to the restart policy on Docker.

There are other settings on the pod manifest, but we will explore these as and when we progress.

Important tips

Set `imagePullPolicy` to `IfNotPresent` unless you have a strong reason for using `Always` or `Never`. This will ensure that your containers boot up quickly and you don't download images unnecessarily.

Always use resource requests and limits while scheduling pods. These ensure that your pod is scheduled in an appropriate node and does not exhaust any existing resources. You can also apply a default resource policy at the cluster level to ensure that your developers don't cause any harm if they miss out on this section for some reason.

Let's apply the manifest using the following command:

```
$ kubectl apply -f nginx-pod.yaml
```

The pod that we created is entirely out of bounds from the host. It runs within the container network, and by default, Kubernetes does not allow any pod to be exposed to the host network unless we explicitly want to expose it.

There are two ways to access the pod – using port forwarding with `kubectl port-forward`, or exposing the pod through a `Service` resource.

Using port forwarding

Before we get into the service side of things, let's consider using the `port-forward` option.

To expose the pod using port forwarding, execute the following command:

```
$ kubectl port-forward nginx 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

The prompt is stuck here. This means it has opened a port forwarding session and is listening on port `8080`. It will automatically forward the request it receives on port `8080` to NGINX port `80`.

Open a duplicate Terminal session and `curl` on the preceding address to see what we get:

```
$ curl 127.0.0.1:8080
...
<title>Welcome to nginx!</title>
...
```

We can see that it is working as we get the default NGINX response.

Now, there are a few things to remember here.

When we use HTTP `port-forward`, we are forwarding requests from the client machine running `kubectl` to the pod, something similar to what's shown in the following diagram:

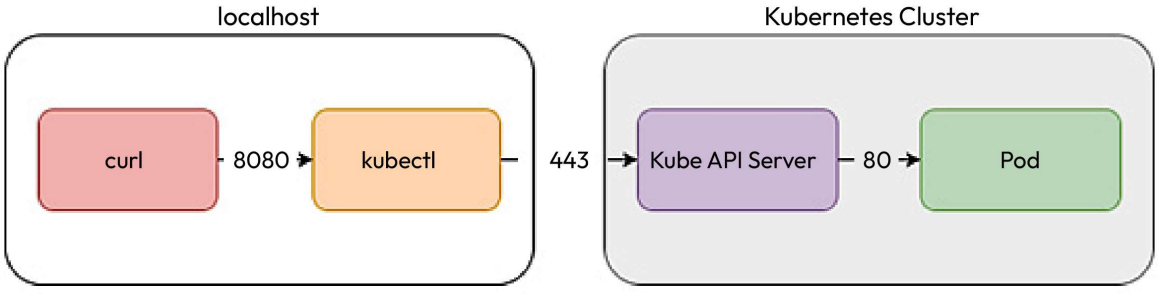

Figure 5.2 – kubectl port-forward

When you run `kubectl port-forward`, the `kubectl` client opens a TCP tunnel via the Kube API server, and the Kube API server then forwards the connection to the correct pod. As the connection between the `kubectl` client and the API server is encrypted, it is a very secure way of accessing your pod, but hold your horses before deciding to use `kubectl port-forward` to expose pods to the outside world.

There are particular use cases for using `kubectl port-forward`:

- For troubleshooting any misbehaving pod.
- For accessing an internal Kubernetes service, such as the Kubernetes dashboard – that is, when you don't want to expose the service to the external world but only allow Kubernetes admins and users to log into the dashboard. It is assumed that only these users will have access to the cluster via `kubectl`.

For anything else, you should use `Service` resources to expose your pod, internally or externally. While we will cover the `Service` resource in the next chapter, let's look at a few operations we can perform with a pod.

Troubleshooting pods

Similar to how we can browse logs from a container using `docker logs`, we can browse logs from a container within a Kubernetes pod using the `kubectl logs` command. If more than one container runs within the pod, we can specify the container's name using the `-c` flag.

To access the container logs, run the following command:

```
$ kubectl logs nginx -c nginx
...
127.0.0.1 - - [18/Jun/2023:14:08:01 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"
```

As the pod is running a single container, we need not specify the `-c` flag, so instead, you can use the following command:

```
$ kubectl logs nginx
```

There might be instances where you may want to get a shell to a running container and troubleshoot what's going on within that. We use `docker exec` for that in the Docker world. Similarly, we can use `kubectl exec` for that within Kubernetes.

Run the following command to open a shell session with the container:

```
$ kubectl exec -it nginx -- /bin/bash
root@nginx:/# cd /etc/nginx/ && ls
conf.d fastcgi_params mime.types modules nginx.conf scgi_params uwsgi_params
root@nginx:/etc/nginx# exit
```

You can even run specific commands without opening a shell session. For example, we can perform the preceding operation with a single line, something like the following:

```
$ kubectl exec nginx -- ls /etc/nginx
conf.d fastcgi_params mime.types modules nginx.conf scgi_params uwsgi_params
```

`kubectl exec` is an important command that helps us troubleshoot containers.

> **Tip**
>
> If you modify files or download packages within the container in `exec` mode, they will persist until the current pod is alive. Once the pod is gone, you will lose all changes. Therefore, it isn't a great way of fixing issues. You should only diagnose problems using `exec`, bake the correct changes in a new image, and then redeploy it.

When we looked at distroless containers in the previous chapter, they did not allow `exec` into the container for security reasons. There are debug images available for distroless that will enable you to open a shell session for troubleshooting purposes if you wish.

> **Tip**
>
> By default, a container runs as the root user if you don't specify the user within the Dockerfile while building the image. You can set a `runAsUser` attribute within your pod's security context if you want to run your pod as a specific user, but this is not ideal. The best practice is to bake the user within the container image.

We've discussed troubleshooting running containers, but what if the containers fail to start for some reason?

Let's look at the following example:

```
$ kubectl run nginx-1 --image=nginx-1
```

Now, let's try to get the pod and see for ourselves:

```
$ kubectl get pod nginx-1
NAME      READY   STATUS             RESTARTS   AGE
nginx-1   0/1     ImagePullBackOff   0          25s
```

Oops! There is some error now, and the status is `ImagePullBackOff`. Well, it seems like there is some issue with the image. While we understand that the issue is with the image, we want to understand the real issue, so for further information on this, we can describe the pod using the following command:

```
$ kubectl describe pod nginx-1
```

Now, this gives us a wealth of information regarding the pod, and if you look at the `events` section, you will find a specific line that tells us what is wrong with the pod:

```
Warning  Failed     60s (x4 over 2m43s)  kubelet            Failed to pull image "nginx-
1": rpc error: code = Unknown desc = failed to pull and unpack image "docker.io/library/
nginx-1:latest": failed to resolve reference "docker.io/library/nginx-1:latest": pull
access denied, repository does not exist or may require authorization: server message:
insufficient_scope: authorization failed
```

So, this one is telling us that either the repository does not exist, or the repository exists but it is private, and hence authorization failed.

> **Tip**
>
> You can use `kubectl describe` for most Kubernetes resources. It should be the first command you use while troubleshooting issues.

Since we know that the image does not exist, let's change the image to a valid one. We must delete the pod and recreate it with the correct image to do that.

To delete the pod, run the following command:

```
$ kubectl delete pod nginx-1
```

To recreate the pod, run the following command:

```
$ kubectl run nginx-1 --image=nginx
```

Now, let's get the pod; it should run as follows:

```
$ kubectl get pod nginx-1
NAME      READY   STATUS    RESTARTS   AGE
nginx-1   1/1     Running   0          42s
```

The pod is now running since we have fixed the image issue.

So far, we've managed to run containers using pods, but pods are very powerful resources that help you manage containers. Kubernetes pods provide probes to ensure your application's reliability. We'll have a look at this in the next section.

Ensuring pod reliability

We talked about health checks in *Chapter 4*, *Creating and Managing Container Images*, and I also mentioned that you should not use them on the Docker level and instead use the ones provided by your container orchestrator. Kubernetes provides three **probes** to monitor your pod's health – the **startup probe**, **liveness probe**, and **readiness probe**.