Multi-Stage Dockerfiles

Overview

In this lab, we will discuss a normal Docker build. You will review and practice `Dockerfile` best practices and learn to create and optimize the size of the Docker images using a builder pattern and multi-stage `Dockerfile`.

Introduction

In the previous lab, we learned about Docker registries, including private and public registries. We created our own private Docker registry to store the Docker images. We also learned how to set up access and store our Docker images in the Docker Hub. In this lab, we will be discussing the concept of multi-stage `Dockerfiles`.

Multi-stage `Dockerfiles` are a feature introduced in Docker version 17.05. This feature is preferable when we want to optimize Docker image size while running Docker images in production environments. To achieve this, a multi-stage `Dockerfile` will create multiple intermediate Docker images during the build process and selectively copy only essential artifacts from one stage to the other.

Before multi-stage Docker builds were introduced, the builder pattern was used to optimize the Docker image size. Unlike multi-stage builds, the builder pattern needs two `Dockerfiles` and a shell script to create efficient Docker images.

In this lab, we will first examine normal Docker builds and the problems associated with them. Next, we will learn how to use the builder pattern to optimize the Docker image size and discuss the problems associated with the builder pattern. Finally, we will learn to use multi-stage `Dockerfiles` to overcome the problems of the builder pattern.

Normal Docker Builds

With Docker, we can use `Dockerfiles` to create custom Docker images. As we discussed in *Lab 2, Getting Started with Dockerfiles*, a `Dockerfile` is a text file that contains instructions on how to create a Docker image. However, it is critical to have minimal-sized Docker images when running them in production environments. This allows developers to speed up their Docker containers' build and deployment times. In this section, we will build a custom Docker image to observe the problems associated with the normal Docker build process.

Consider an example where we build a simple Golang application. We are going to deploy a `hello world` application written in Golang using the following `Dockerfile`:

```
# Start from latest golang parent image
FROM golang:1.15.6
# Set the working directory
WORKDIR /myapp
# Copy source file from current directory to container
COPY helloworld.go .
# Build the application
RUN go build -o helloworld .
# Run the application
ENTRYPOINT ["./helloworld"]
```

This `Dockerfile` starts with the latest Golang image as the parent image. This parent image contains all the build tools required to build our Golang application. Next, we will set the `/myapp` directory as the current working directory and copy the `helloworld.go` source file from the host filesystem to the container filesystem. Then, we will use the `RUN` directive to execute the `go build` command to build the application. Finally, the `ENTRYPOINT` directive is used to run the `helloworld` executable created in the previous step.

The following is the content of the `helloworld.go` file. This is a simple file that will print the text `"Hello World"` when executed:

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

Once the `Dockerfile` is ready, we can build the Docker image using the `docker image build` command. This image will be tagged as `helloworld:v1`:

```
docker image build -t helloworld:v1 .
```

Now, observe the built image with the `docker image ls` command. You will get an output similar to the following:

```
REPOSITORY   TAG   IMAGE ID       CREATED          SIZE
helloworld   v1    23874f841e3e   10 seconds ago   805MB
```

Notice the image size. This build has resulted in a huge Docker image of 805 MB in size. It is not efficient to have these large Docker images in production environments as they will take a lot of time and bandwidth to be pushed and pulled over networks. Small Docker images are much more efficient and can be pushed and pulled quickly and deployed faster.

In addition to the size of the image, these Docker images can be vulnerable to attacks since they contain build tools that can have potential security vulnerabilities.

Note

Potential security vulnerabilities may vary depending on what packages are in the given Docker image. As an example, Java JDK has a number of vulnerabilities. You can have a detailed look at the vulnerabilities related to Java JDK at the following link:

https://www.cvedetails.com/vulnerability-list/vendor_id-93/product_id-19116/Oracle-JDK.html.

To reduce the attack surface, it is recommended to have only the essential artifacts (for example, compiled code) and runtimes when running Docker images in production environments. As an example, with Golang, the Go compiler is required to build the application, but not to run the application.

Ideally, you want a minimal-sized Docker image that only contains the runtime tools and excludes all the build tools that we used to build the application.

We will now build such a Docker image using the normal build process in the following exercise.

Exercise 4.01: Building a Docker Image with the Normal Build Process

Your manager has asked you to dockerize a simple Golang application. You are provided with the Golang source code file, and your task is to compile and run this file. In this exercise, you will build a Docker image using the normal build process. You will then observe the image size of the final Docker image:

1. Create a new directory named `normal-build` for this exercise:

   ```
   $ mkdir normal-build
   ```

2. Navigate to the newly created `normal-build` directory:

   ```
   $ cd normal-build
   ```

3. Within the `normal-build` directory, create a file named `welcome.go`. This file will be copied to the Docker image during the build time:

   ```
   $ touch welcome.go
   ```

4. Now, open the `welcome.go` file using your favorite text editor:

   ```
   $ vim welcome.go
   ```

5. Add the following content to the `welcome.go` file, save it, and exit from the `welcome.go` file:

   ```
   package main
   import "fmt"
   func main() {
       fmt.Println("Welcome to multi-stage Docker builds")
   }
   ```

   This is a simple `hello world` application written in Golang. This will output `"Welcome to multi-stage Docker builds"` on execution.

6. Within the `normal-build` directory, create a file named `Dockerfile`:

   ```
   $ touch Dockerfile
   ```

7. Now, open the `Dockerfile` using your favorite text editor:

   ```
   $ vim Dockerfile
   ```

8. Add the following content to the `Dockerfile` and save the file:

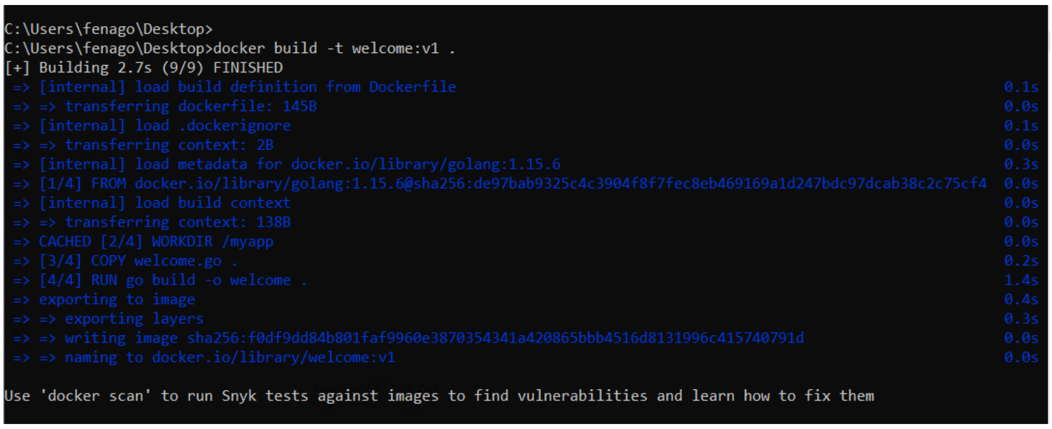
   ```
   FROM golang:1.15.6
   WORKDIR /myapp
   COPY welcome.go .
   RUN go build -o welcome .
   ENTRYPOINT ["./welcome"]
   ```

   The `Dockerfile` starts with the `FROM` directive that specifies the latest Golang image as the parent image. This will set the `/myapp` directory as the current working directory of the Docker image. Then, the `COPY` directive will copy the `welcome.go` source file that you created in *step 3* to the Docker filesystem. Next is the `go build` command, which will build the Golang code that you created. Finally, the welcome code will be executed.

9. Now, build the Docker image:

   ```
   docker build -t welcome:v1 .
   ```

You will see that the image is successfully built and tagged as `welcome:v1`:



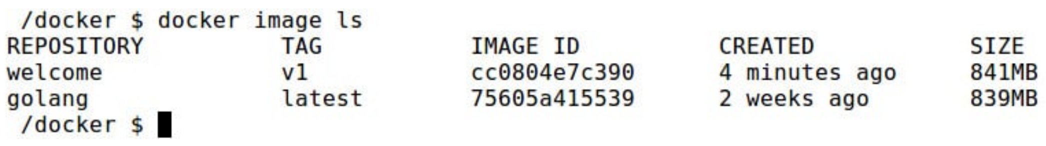
```
C:\Users\fenago\Desktop>
C:\Users\fenago\Desktop>docker build -t welcome:v1 .
[+] Building 2.7s (9/9) FINISHED
 => [internal] load build definition from Dockerfile                                                      0.1s
 => => transferring dockerfile: 145B                                                                      0.0s
 => [internal] load .dockerignore                                                                         0.1s
 => => transferring context: 2B                                                                           0.0s
 => [internal] load metadata for docker.io/library/golang:1.15.6                                          0.3s
 => [1/4] FROM docker.io/library/golang:1.15.6@sha256:de97bab9325c4c3904f8f7fec8eb469169a1d247bdc97dcab38c2c75cf4  0.0s
 => [internal] load build context                                                                         0.0s
 => => transferring context: 138B                                                                         0.0s
 => CACHED [2/4] WORKDIR /myapp                                                                           0.0s
 => [3/4] COPY welcome.go .                                                                               0.2s
 => [4/4] RUN go build -o welcome .                                                                       1.4s
 => exporting to image                                                                                    0.4s
 => => exporting layers                                                                                   0.3s
 => => writing image sha256:f0df9dd84b801faf9960e3870354341a420865bbb4516d8131996c415740791d             0.0s
 => => naming to docker.io/library/welcome:v1                                                             0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
```

Figure 4.1: Building the Docker image

10. Use the `docker image ls` command to list all the Docker images available on your computer:

```
docker image ls
```

The command should return the following output:



```
 /docker $ docker image ls
REPOSITORY          TAG           IMAGE ID         CREATED          SIZE
welcome             v1            cc0804e7c390     4 minutes ago    841MB
golang              latest        75605a415539     2 weeks ago      839MB
 /docker $
```

Figure 4.2: Listing all Docker images

It can be observed in the preceding output that the image size of the `welcome:v1` image is `805MB`.

In this section, we discussed how to use the normal Docker build process to build a Docker image and observed its size. The result was a huge Docker image, over 800 MB in size. The main disadvantage of these large Docker images is that they will take significant time to build, deploy, push, and pull over the networks. So, it is recommended to create minimal-sized Docker images whenever possible. In the next section, we will discuss how we can use the builder pattern to optimize the image size.

What Is the Builder Pattern?

The **builder pattern** is a method used to create optimally sized Docker images. It uses two Docker images and selectively copies essential artifacts from one to the other. The first Docker image is known as the `build image` and is used as the build environment to build the executables from the source code. This Docker image contains compilers, build tools, and development dependencies required during the build process.

The second Docker image is known as the `runtime image` and is used as the runtime environment to run the executables created by the first Docker container. This Docker image contains only the executables, the dependencies, and the runtime tools. A shell script is used to copy the artifacts using the `docker container cp` command.

The entire process of building the image using the builder pattern consists of the following steps:

1. Create the `Build` Docker image.
2. Create a container from the `Build` Docker image.
3. Copy the artifacts from the `Build` Docker image to the local filesystem.
4. Build the `Runtime` Docker image using copied artifacts:

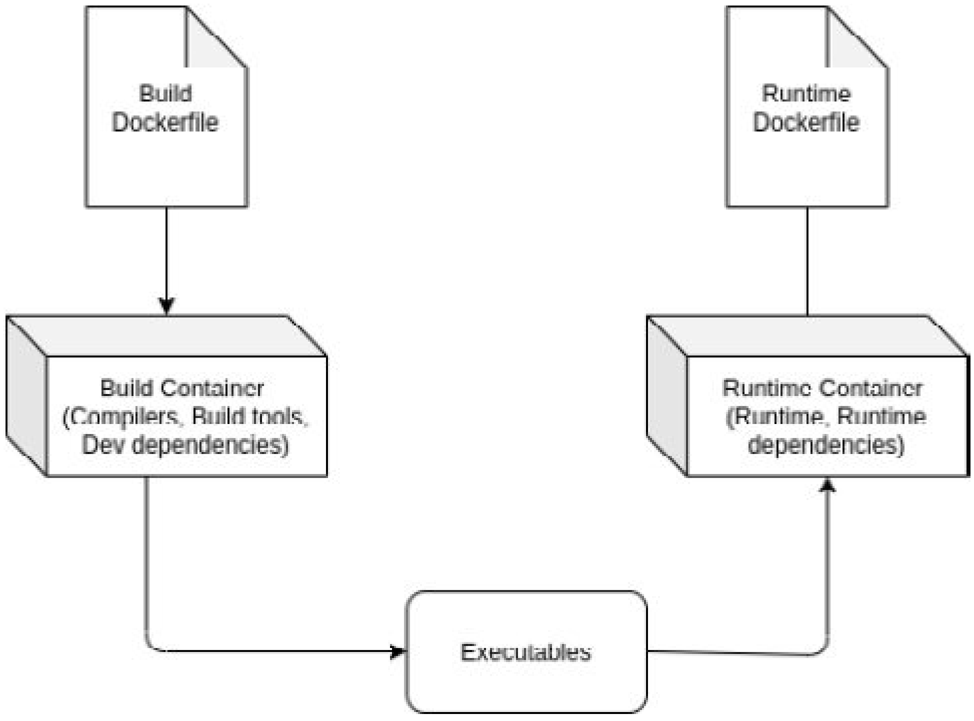

Build Dockerfile
Runtime Dockerfile
Build Container (Compilers, Build tools, Dev dependencies)
Runtime Container (Runtime, Runtime dependencies)
Executables

Figure 4.3: Building images using the builder pattern

As illustrated in the preceding image, the `Build Dockerfile` is used to create the build container that will contain all the tools required to build the source code, including compilers and build tools such as Maven, Gradle, and development dependencies. Once the build container is created, the shell script will copy the executables from the build container to the Docker host. Finally, the `Runtime` container will be created with the executables copied from the `Build` container.

Now, observe how the builder pattern can be used to create minimal Docker images. The following is the first `Dockerfile` used to create the `Build` Docker container. This `Dockerfile` is named Dockerfile.build to distinguish it from the `Runtime Dockerfile`:

```
# Start from latest golang parent image
FROM golang:1.15.6
# Set the working directory
WORKDIR /myapp
# Copy source file from current directory to container
COPY helloworld.go .
# Build the application
RUN go build -o helloworld .
# Run the application
ENTRYPOINT ["./helloworld"]
```

This is the same `Dockerfile` that we observed with the normal Docker builds. This was used to create the `helloworld` executable from the `helloworld.go` source file.

The following is the second `Dockerfile` used to build the `Runtime` Docker container:

```
# Start from latest alpine parent image
FROM alpine:latest
# Set the working directory
WORKDIR /myapp
# Copy helloworld app from current directory to container
COPY helloworld .
# Run the application
ENTRYPOINT ["./helloworld"]
```

As opposed to the first `Dockerfile`, created from the `golang` parent image, this second `Dockerfile` uses the `alpine` image as its parent image because it is a minimal-sized Docker image, at only 5 MB. This image uses Alpine Linux, a lightweight Linux distribution. Next, the `/myapp` directory is configured as the working directory. Finally, the `helloworld` artifact is copied to the Docker image, and the `ENTRYPOINT` directive is used to run the application.

This `helloworld` artifact is the result of the `go build -o helloworld .` command executed in the first `Dockerfile`. We will be using a shell script to copy this artifact from the `build` Docker container to the local filesystem, from where this artifact will be copied to the runtime Docker image.

Consider the following shell script used to copy the build artifacts between Docker containers:

```
#!/bin/sh
# Build the builder Docker image
docker image build -t helloworld-build -f Dockerfile.build .
# Create container from the build Docker image
docker container create --name helloworld-build-container   helloworld-build
# Copy build artifacts from build container to the local filesystem
docker container cp helloworld-build-container:/myapp/helloworld .
# Build the runtime Docker image
docker image build -t helloworld .
# Remove the build Docker container
docker container rm -f helloworld-build-container
# Remove the copied artifact
rm helloworld
```

This shell script will first build the `helloworld-build` Docker image using the `Dockerfile.build` file. The next step is to create a Docker container from the `helloworld-build` image so that we can copy the `helloworld` artifact to the Docker host. Once the container is created, we need to execute the command to copy the `helloworld` artifact from the `helloworld-build-container` to the current directory of the Docker host. Now, we can build the runtime container with the `docker image build` command. Finally, we will execute the necessary cleanup tasks by removing the intermediate artifacts, such as the `helloworld-build-container` container and the `helloworld` executable.

Once we execute the shell script, we should be able to see two Docker images:

```
REPOSITORY         TAG      IMAGE ID       CREATED        SIZE
helloworld         latest   faff247e2b35   3 hours ago    7.6MB
```

```
helloworld-build   latest   f8c10c5bd28d   3 hours ago   805MB
```

Note the size difference between the two Docker images. The `helloworld` Docker image is only 7.6 MB in size, which is a huge reduction from the `helloworld-build` image at 805 MB.

As we can see, the builder pattern can drastically reduce the size of the Docker images by copying only the essential artifacts to the final image. However, the disadvantage with the builder pattern is that we need to maintain two `Dockerfiles` and a shell script.

In the next exercise, we will gain hands-on experience in creating an optimized Docker image using the builder pattern.

Exercise 4.02: Building a Docker Image with the Builder Pattern

In *Exercise 4.01*, *Building a Docker Image with the Normal Build Process*, you created a Docker image to compile and run the Golang application. Now the application is ready to go live, but the manager is not happy with the size of the Docker image. You have been asked to create a minimal-sized Docker image to run the application. In this exercise, you will optimize the Docker image using the builder pattern:

1. Create a new directory named `builder-pattern` for this exercise:

   ```
   $ mkdir builder-pattern
   ```

2. Navigate to the newly created `builder-pattern` directory:

   ```
   $ cd builder-pattern
   ```

3. Within the `builder-pattern` directory, create a file named `welcome.go`. This file will be copied to the Docker image at build time:

   ```
   $ touch welcome.go
   ```

4. Now, open the `welcome.go` file using your favorite text editor:

   ```
   $ vim welcome.go
   ```

5. Add the following content to the `welcome.go` file, and then save and exit this file:

   ```
   package main
   import "fmt"
   func main() {
       fmt.Println("Welcome to multi-stage Docker builds")
   }
   ```

   This is a simple `hello world` application written in Golang. This will output `"Welcome to multi-stage Docker builds"` once executed.

6. Within the `builder-pattern` directory, create a file named `Dockerfile.build`. This file will contain all the instructions that you are going to use to create the `build` Docker image:

   ```
   $ touch Dockerfile.build
   ```

7. Now, open the `Dockerfile.build` using your favorite text editor:

```
$ vim Dockerfile.build
```

8. Add the following content to the `Dockerfile.build` file and save the file:

   ```
   FROM golang:1.15.6
   WORKDIR /myapp
   COPY welcome.go .
   RUN go build -o welcome .
   ENTRYPOINT ["./welcome"]
   ```

   This has the same content that you created for the `Dockerfile` in *Exercise 4.01*, *Building a Docker Image with the Normal Build Process*.

9. Next, create the `Dockerfile` for the runtime container. Within the `builder-pattern` directory, create a file named `Dockerfile`. This file will contain all the instructions that you are going to use to create the runtime Docker image:

   ```
   $ touch Dockerfile
   ```

10. Now, open the `Dockerfile` using your favorite text editor:

    ```
    $ vim Dockerfile
    ```

11. Add the following content to the `Dockerfile` and save the file:

    ```
    FROM scratch
    WORKDIR /myapp
    COPY welcome .
    ENTRYPOINT ["./welcome"]
    ```

    This `Dockerfile` uses the scratch image, which is the most minimal image in Docker, as the parent. Then, it will configure the `/myapp` directory as the working directory. Next, the welcome executable is copied from the Docker host to the runtime Docker image. Finally, the `ENTRYPOINT` directive is used to execute the welcome executable.

12. Create the shell script to copy the executables between Docker containers. Within the `builder-pattern` directory, create a file named `build.sh`. This file will contain the steps to coordinate the build process between the two Docker containers:

    ```
    $ touch build.sh
    ```

13. Now, open the `build.sh` file using your favorite text editor:

    ```
    $ vim build.sh
    ```

14. Add the following content to the shell script and save the file:

    ```
    #!/bin/sh
    echo "Creating welcome builder image"
    docker image build -t welcome-builder:v1 -f Dockerfile.build .
    docker container create --name welcome-builder-container   welcome-builder:v1
    docker container cp welcome-builder-container:/myapp/welcome .
    docker container rm -f welcome-builder-container
    ```

```
echo "Creating welcome runtime image"
docker image build -t welcome-runtime:v1 .
rm welcome
```

This shell script will first build the `welcome-builder` Docker image and create a container from it. Then it will copy the compiled Golang executable from the container to the local filesystem. Next, the `welcome-builder-container` container is removed as it is an intermediate container. Finally, the `welcome-runtime` image is built.

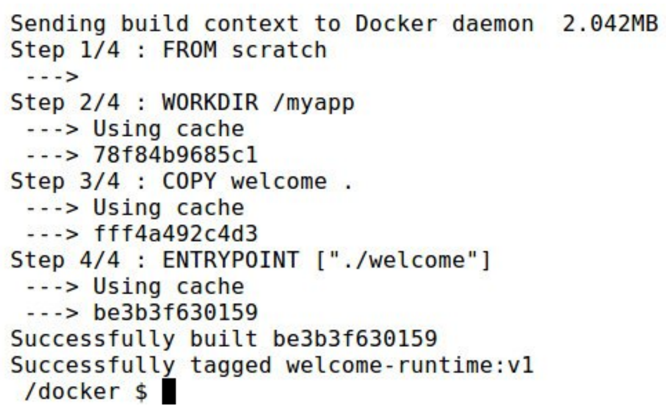
15. Add execution permissions to the `build.sh` shell script:

```
$ chmod +x build.sh
```

16. Now that you have the two `Dockerfiles` and the shell script, build the Docker image by executing the `build.sh` shell script:

```
$ ./build.sh
```

The image will be successfully built and tagged as `welcome-runtime:v1`:



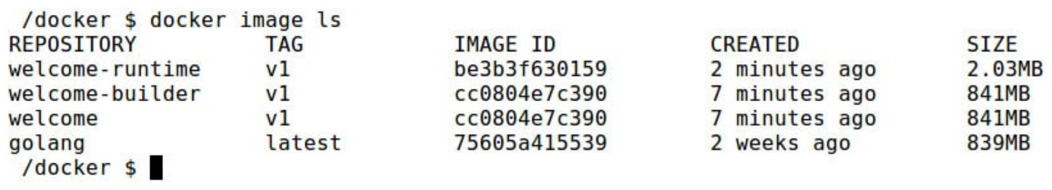
```
Sending build context to Docker daemon  2.042MB
Step 1/4 : FROM scratch
 --->
Step 2/4 : WORKDIR /myapp
 ---> Using cache
 ---> 78f84b9685c1
Step 3/4 : COPY welcome .
 ---> Using cache
 ---> fff4a492c4d3
Step 4/4 : ENTRYPOINT ["./welcome"]
 ---> Using cache
 ---> be3b3f630159
Successfully built be3b3f630159
Successfully tagged welcome-runtime:v1
 /docker $ 
```

Figure 4.4: Building the Docker image

17. Use the `docker image` ls command to list all the Docker images available on your computer:

```
docker image ls
```

You should get the list of all the available Docker images as shown in the following figure:



```
 /docker $ docker image ls
REPOSITORY          TAG             IMAGE ID            CREATED             SIZE
welcome-runtime     v1              be3b3f630159        2 minutes ago       2.03MB
welcome-builder     v1              cc0804e7c390        7 minutes ago       841MB
welcome             v1              cc0804e7c390        7 minutes ago       841MB
golang              latest          75605a415539        2 weeks ago         839MB
 /docker $ 
```

Figure 4.5: Listing all Docker images

As you can see from the preceding output, there are two Docker images available. welcome-builder has all the builds tools and is 805 MB, while welcome-runtime has a significantly lower image size of 2.01 MB. `golang:1.15.6` is the Docker image we used as the parent image of `welcome-builder`.

In this exercise, you learned how to use the builder pattern to reduce the size of the Docker image. However, using the builder pattern to optimize the size of the Docker image means that we have to maintain two `Dockerfiles` and one shell script. In the next section, let's observe how we can eliminate them by using a multi-stage `Dockerfile`.

Introduction to Multi-Stage Dockerfiles

Multi-stage Dockerfiles are a feature that allows for a single `Dockerfile` to contain multiple stages that can produce optimized Docker images. As we observed with the builder pattern in the previous section, the stages will usually include a builder state to build the executables from source code, and a runtime stage to run the executables. Multi-stage `Dockerfiles` will use multiple `FROM` directives within the `Dockerfile` for each stage, and each stage will start with a different base image. Only the essential files will be copied selectively from one stage to the other. Before multi-stage `Dockerfiles`, this was achieved with the builder pattern, as we discussed in the previous section.

Multi-stage Docker builds allow us to create minimal-sized Docker images that are similar to the builder pattern but eliminate the problems associated with it. As we have seen in the previous example, the builder pattern needs to maintain two `Dockerfiles` and a shell script. In contrast, multi-stage Docker builds will need only one `Dockerfile` and do not require any shell script to copy the executables between Docker containers. Also, the builder pattern requires that you copy the executables to the Docker host before copying them to the final Docker image. This is not required with the multi-stage Docker builds as we can use the `--from` flag to copy the executables between Docker images without copying them to the Docker host.

Now, let's observe the structure of a multi-stage `Dockerfile`:

```
# Start from latest golang parent image
FROM golang:1.15.6
# Set the working directory
WORKDIR /myapp
# Copy source file from current directory to container
COPY helloworld.go .
# Build the application
RUN go build -o helloworld .
# Start from latest alpine parent image
FROM alpine:latest
# Set the working directory
WORKDIR /myapp
# Copy helloworld app from current directory to container
COPY --from=0 /myapp/helloworld .
# Run the application
ENTRYPOINT ["./helloworld"]
```

The main difference between a normal `Dockerfile` and a multi-stage `Dockerfile` is that a multi-stage `Dockerfile` will use multiple `FROM` directives to build each phase. Each new phase will start with a new parent image and does not contain anything from the previous image other than the selectively copied executables. `COPY --from=0` is used to copy the executable from the first stage to the second stage.

Build the Docker image and tag the image as `multi-stage:v1`:

```
docker image build -t multi-stage:v1 .
```

Now, you can list the available Docker images:

```
REPOSITORY    TAG      IMAGE ID       CREATED         SIZE
multi-stage   latest   75e1f4bcabd0   7 seconds ago   7.6MB
```

You can see that this has resulted in a Docker image of the same size that we observed with the builder pattern.

Note

Multi-stage `Dockerfiles` reduce the number of `Dockerfiles` required and eliminate the shell script without making any difference to the size of the image.

By default, the stages in the multi-stage `Dockerfile` are referred to by an integer number, starting with `0` from the first stage. These stages can be named to increase readability and maintainability by adding `AS <NAME>` to the `FROM` directive. The following is the improved version of the multi-stage `Dockerfile` that you observed in the preceding code block:

```
# Start from latest golang parent image
FROM golang:1.15.6 AS builder
# Set the working directory
WORKDIR /myapp
# Copy source file from current directory to container
COPY helloworld.go .
# Build the application
RUN go build -o helloworld .
# Start from latest alpine parent image
FROM alpine:latest AS runtime
# Set the working directory
WORKDIR /myapp
# Copy helloworld app from current directory to container
COPY --from=builder /myapp/helloworld .
# Run the application
ENTRYPOINT ["./helloworld"]
```

In the preceding example, we named the first stage `builder` and second stage `runtime`, as shown here:

```
FROM golang:1.15.6 AS builder
FROM alpine:latest AS runtime
```

Then, while copying the artifacts in the second stage, you used the name `builder` for the `--from` flag:

```
COPY --from=builder /myapp/helloworld .
```

While building a multi-stage `Dockerfile`, there might be instances where you want to build only up to a specific build stage. Consider that your `Dockerfile` has two stages. The first one is to build the development stage and contains all the build and debug tools, and the second is to build the production image that will contain only the runtime tools. During the code development phase of the project, you might only need to build up to the development stage to test and debug your code whenever necessary. In this scenario, you can use the `--target` flag with the `docker build` command to specify an intermediate stage as the final stage for the resulting image:

```
docker image build --target builder -t multi-stage-dev:v1 .
```

In the preceding example, you used `--target builder` to stop the build at the builder stage.

In the next exercise, you will learn to use a multi-stage `Dockerfile` to create a size-optimized Docker image.

Exercise 4.03: Building a Docker Image with a Multi-Stage Docker Build

In *Exercise 4.02*, *Building a Docker Image with the Builder Pattern*, you used the builder pattern to optimize the size of the Docker image. However, there is an operational burden, as you need to manage two `Dockerfiles` and a shell script during the Docker image build process. In this exercise, you are going to use a multi-stage `Dockerfile` to eliminate this operational burden.

1. Create a new directory named `multi-stage` for this exercise:

   ```
   mkdir multi-stage
   ```

2. Navigate to the newly created `multi-stage` directory:

   ```
   cd multi-stage
   ```

3. Within the `multi-stage` directory, create a file named `welcome.go`. This file will be copied to the Docker image during the build time:

   ```
   $ touch welcome.go
   ```

4. Now, open the `welcome.go` file using your favorite text editor:

   ```
   $ vim welcome.go
   ```

5. Add the following content to the `welcome.go` file, and then save and exit this file:

   ```
   package main
   import "fmt"
   func main() {
       fmt.Println("Welcome to multi-stage Docker builds")
   }
   ```

   This is a simple `hello world` application written in Golang. This will output `"Welcome to multi-stage Docker builds"` once executed.

   Within the multi-stage directory, create a file named `Dockerfile`. This file will be the multi-stage `Dockerfile`:

   ```
   touch Dockerfile
   ```

6. Now, open the `Dockerfile` using your favorite text editor:

   ```
   vim Dockerfile
   ```

7. Add the following content to the `Dockerfile` and save the file:

   ```
   FROM golang:1.15.6 AS builder
   WORKDIR /myapp
   COPY welcome.go .
   ```

```
RUN go build -o welcome .
FROM scratch
WORKDIR /myapp
COPY --from=builder /myapp/welcome .
ENTRYPOINT ["./welcome"]
```

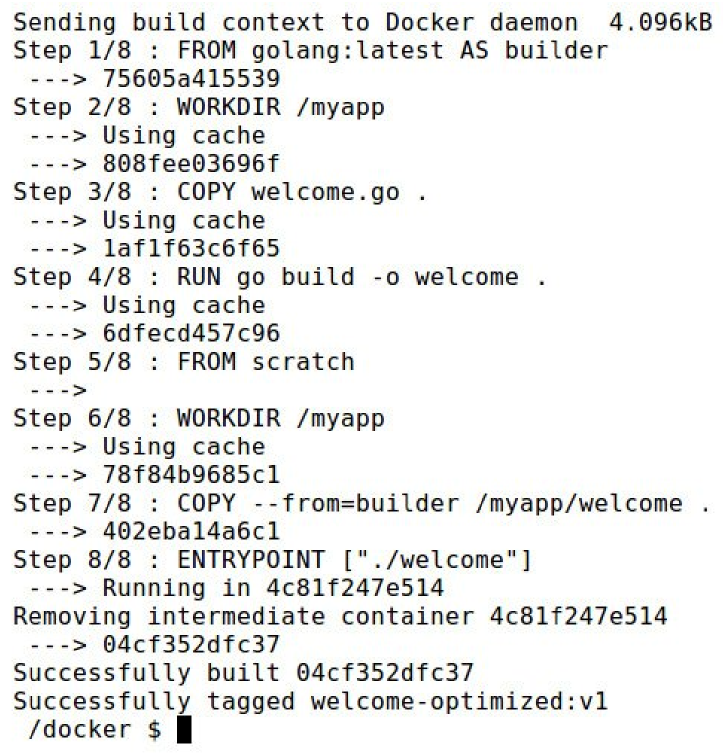
This multi-stage `Dockerfile` uses the latest `golang` image as the parent image and this stage is named `builder`. Next, the `/myapp` directory is specified as the current working directory. Then, the `COPY` directive is used to copy the `welcome.go` source file and the `RUN` directive is used to build the Golang file.

The next stage of the `Dockerfile` uses the `scratch` image as the parent image. This will set the `/myapp` directory as the current working directory of the Docker image. Then, the `COPY` directive is used to copy the `welcome` executable from the builder stage to this stage. Finally, `ENTRYPOINT` is used to run the `welcome` executable.

8. Build the Docker image using the following command:

```
docker build -t welcome-optimized:v1 .
```

The image will be successfully built and tagged as `welcome-optimized:v1`:



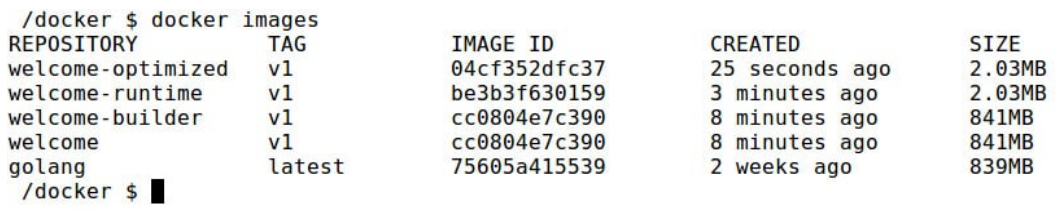
```
Sending build context to Docker daemon  4.096kB
Step 1/8 : FROM golang:latest AS builder
 ---> 75605a415539
Step 2/8 : WORKDIR /myapp
 ---> Using cache
 ---> 808fee03696f
Step 3/8 : COPY welcome.go .
 ---> Using cache
 ---> 1af1f63c6f65
Step 4/8 : RUN go build -o welcome .
 ---> Using cache
 ---> 6dfecd457c96
Step 5/8 : FROM scratch
 --->
Step 6/8 : WORKDIR /myapp
 ---> Using cache
 ---> 78f84b9685c1
Step 7/8 : COPY --from=builder /myapp/welcome .
 ---> 402eba14a6c1
Step 8/8 : ENTRYPOINT ["./welcome"]
 ---> Running in 4c81f247e514
Removing intermediate container 4c81f247e514
 ---> 04cf352dfc37
Successfully built 04cf352dfc37
Successfully tagged welcome-optimized:v1
 /docker $
```

Figure 4.6: Building the Docker image

9. Use the `docker image ls` command to list all the Docker images available on your computer. These images are available on your computer, either when you pull them from Docker Registry or when you build them on your computer:

```
docker images
```

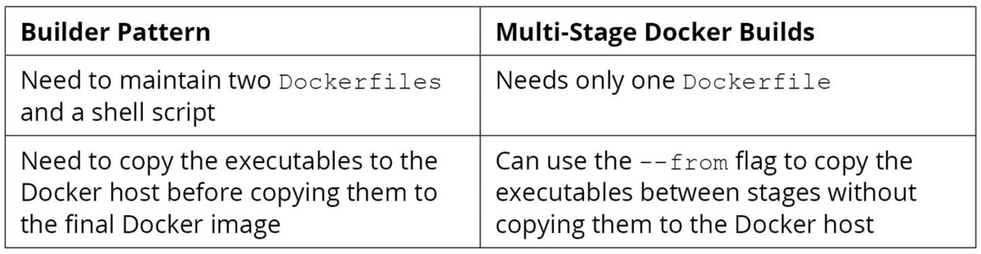
As you can see from the following output, the `welcome-optimized` image has the same size as the `welcome-runtime` image that you built in *Exercise 4.02, Building a Docker Image with the Builder Pattern*:



```
 /docker $ docker images
REPOSITORY          TAG           IMAGE ID           CREATED            SIZE
welcome-optimized   v1            04cf352dfc37       25 seconds ago     2.03MB
welcome-runtime     v1            be3b3f630159       3 minutes ago      2.03MB
welcome-builder     v1            cc0804e7c390       8 minutes ago      841MB
welcome             v1            cc0804e7c390       8 minutes ago      841MB
golang              latest        75605a415539       2 weeks ago        839MB
 /docker $ █
```

Figure 4.7: Listing all Docker images

In this exercise, you learned how to use multi-stage `Dockerfiles` to build optimized Docker images. The following table presents a summary of the key differences between the builder pattern and multi-stage `Docker Builds`:



Builder Pattern	**Multi-Stage Docker Builds**
Need to maintain two `Dockerfiles` and a shell script	Needs only one `Dockerfile`
Need to copy the executables to the Docker host before copying them to the final Docker image	Can use the `--from` flag to copy the executables between stages without copying them to the Docker host

Figure 4.8: Differences between the builder pattern and multi-stage Docker Builds

In the next section, we will review the best practices to follow when writing a `Dockerfile`.

Dockerfile Best Practices

In the previous section, we discussed how we can build an efficient Docker image with multi-stage `Dockerfiles`. In this section, we will cover other recommended best practices for writing `Dockerfiles`. These best practices will ensure reduced build time, reduced image size, increased security, and increased maintainability of the Docker images produced.

Using an Appropriate Parent Image

Using the appropriate base image is one of the key recommendations when building efficient Docker images.

It is always encouraged to use official images from the **Docker Hub** as the parent image when you are building custom Docker images. These official images will ensure that all best practices are followed, documentation is available, and security patches are applied. For example, if you need the **JDK** (**Java Development Kit**) for your application, you can use the `openjdk` official Docker image instead of using the generic `ubuntu` image and installing the JDK on top of the `ubuntu` image:

Inefficient Dockerfile	Efficient Dockerfile
`FROM ubuntu` `RUN apt-get update && \` `apt-get install -y openjdk-8-jdk`	`FROM openjdk`

Figure 4.9: Using appropriate parent images

Secondly, avoid using the `latest` tag for the parent image when building Docker images for production environments. The `latest` tag might get pointed to a newer version of the image as the new versions are released to the Docker Hub, and the newer version might not be backward compatible with your applications, leading to failures in your production environments. Instead, the best practice is to always use a specific versioned tag as the parent image:

Inefficient Dockerfile	Efficient Dockerfile
`FROM openjdk:latest`	`FROM openjdk:8`

Figure 4.10: Avoiding the use of the latest tag of the parent image

Finally, using the minimal version of the parent image is critical to getting a minimal-sized Docker image. Most of the official Docker images in Docker Hub have a minimal-sized image built around the Alpine Linux image. Also, in our example, we can use the **JRE** (**Java Runtime Environment**) to run the application instead of the JDK, which contains the build tools:

Inefficient Dockerfile	Efficient Dockerfile
`FROM openjdk:8`	`FROM openjdk:8-jre-alpine`

Figure 4.11: Using minimal-sized images

The `openjdk:8-jre-alpine` image will be only 84.9 MB in size, whereas `openjdk:8` will be 488 MB in size.

Using a Non-Root User for Better Security

By default, Docker containers run with the root (`id = 0`) user. This allows the user to perform all the necessary administrative activities, such as changing system configurations, installing packages, and binding to privileged ports. However, this is high risk and is considered a bad security practice when running Docker containers in production environments since hackers can gain root access to the Docker host by hacking the applications running inside the Docker container.

Running containers as a non-root user is a recommended best practice to improve the security of the Docker container. This will adhere to the principle of least privilege, which ensures that the application has only the bare minimum privileges to perform its tasks. There are two methods that we can use to run a container as a non-root user: with the `--user` (or `-u`) flag, and with the `USER` directive.

Using the `--user` (or `-u`) flag with the `docker run` command is one method for changing the default user while running a Docker container. Either the username or the user ID can be specified with the `--user` (or `-u`) flag:

```
docker run --user=9999 ubuntu:focal
```

In the preceding command, we have specified the user ID as `9999`. If we are specifying the user as an ID, the corresponding user does not have to be available in the Docker container.

Additionally, we can use the `USER` directive within the `Dockerfile` to define the default user. However, this value can be overridden with the `--user` flag while starting the Docker container:

```
FROM ubuntu:focal
RUN apt-get update
RUN useradd demo-user
USER demo-user
CMD whoami
```

In the preceding example, we have used the `USER` directive to set the default user to `demo-user`. This means that any command after the `USER` directive will be executed as a `demo-user`.

Using dockerignore

The `.dockerignore` file is a special text file within the Docker context that is used to specify a list of files to be excluded from the Docker context while building the Docker image. Once we execute the `docker build` command, the Docker client will package the entire build context as a TAR archive and upload it to the Docker daemon. When we execute the `docker build` command, the first line of the output is `Sending build context to Docker daemon`, which indicates that the Docker client is uploading the build context to the Docker daemon:

```
Sending build context to Docker daemon  18.6MB
Step 1/5 : FROM ubuntu:focal
```

Each time we build the Docker image, the build context will be sent to the Docker daemon. As this will take time and bandwidth during the Docker image build process, it is recommended to exclude all the files that are not needed in the final Docker image. The `.dockerignore` file can be used to achieve this purpose. In addition to saving time and bandwidth, the `.dockerignore` file is used to exclude the confidential files, such as password files and key files from the build context.

The `.dockerignore` file should be created in the root directory of the build context. Before sending the build context to the Docker daemon, the Docker client will look for the `.dockerignore` file in the root of the build context. If the `.dockerignore` file exists, the Docker client will exclude all the files mentioned in the `.dockerignore` file from the build context.

The following is the content of a sample `.dockerignore` file:

```
PASSWORDS.txt
tmp/
*.md
!README.md
```

In the preceding example, we have specifically excluded the `PASSWORDS.txt` file and `tmp` directory from the build context, as well as all files with the `.md` extension except for the `README.md` file.

Minimizing Layers

Each line in the `Dockerfile` will create a new layer that will take up space in the Docker image. So, it is recommended to create as few layers as possible when building the Docker image. To achieve this, combine the `RUN` directives whenever possible.

As an example, consider the following `Dockerfile`, which will update the package repository first and then install the `redis-server` and `nginx` packages:

```
FROM ubuntu:focal
RUN apt-get update
RUN apt-get install -y nginx
RUN apt-get install -y redis-server
```

This `Dockerfile` can be optimized by combining the three `RUN` directives:

```
FROM ubuntu:focal
RUN apt-get update \
  && apt-get install -y nginx redis-server
```

Don't Install Unnecessary Tools

Not installing unnecessary debugging tools (such as `vim`, `curl`, and `telnet`) and removing unnecessary dependencies can help to create efficient Docker images that are small in size. Some package managers such as `apt` will install recommended and suggested packages automatically alongside required packages. We can avoid this by specifying the `no-install-recommends` flag with the `apt-get install` command:

```
FROM ubuntu:focal
RUN apt-get update \
  && apt-get install --no-install-recommends -y nginx
```

In the preceding example, we are installing the `nginx` package with the `no-install-recommends` flag, which will help to reduce the final image size by around 10 MB.

In addition to using the `no-install-recommends` flag, we can also remove the cache of the `apt` package manager to further reduce the final Docker image size. This can be achieved by running `rm -rf /var/lib/apt/lists/*` at the end of the `apt-get install` command:

```
FROM ubuntu:focal
RUN apt-get update \
    && apt-get install --no-install-recommends -y nginx \
    && rm -rf /var/lib/apt/lists/*
```

In this section, we discussed the best practices when writing a `Dockerfile`. Following these best practices will help to reduce build time, reduce the image size, increase security, and increase the maintainability of the Docker image.

Now, let's test our knowledge by deploying a Golang HTTP server with a multi-stage Docker build in the next activity.

Activity 4.01: Deploying a Golang HTTP Server with a Multi-Stage Docker Build

Imagine that you have been tasked with deploying a Golang HTTP server to a Docker container. Your manager has asked you to build a minimal-sized Docker image and observe best practices while building the `Dockerfile`.

This Golang HTTP server will return different responses based on the invoke URL:

Invoke URL	**Message**
`http://127.0.0.1:<port>/`	`Home Page`
`http://127.0.0.1:<port>/contact`	`Contact Us`
`http://127.0.0.1:<port>/login`	`Login Page`

Figure 4.12: Responses based on the invoke URL

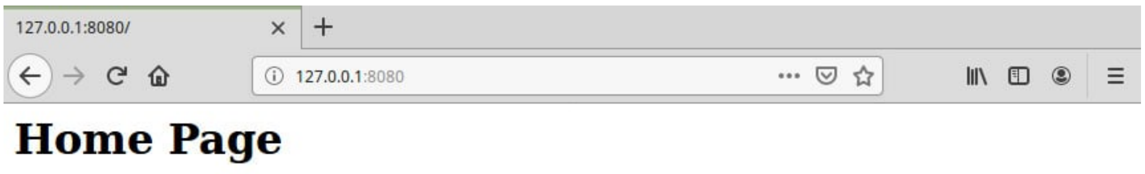
Your task is to dockerize the Golang application given in the following code block using a multi-stage `Dockerfile`:

```
package main
import (
    "net/http"
    "fmt"
    "log"
    "os"
)
func main() {
    http.HandleFunc("/", defaultHandler)
    http.HandleFunc("/contact", contactHandler)
    http.HandleFunc("/login", loginHandler)
    port := os.Getenv("PORT")
    if port == "" {
        port = "8080"
    }
    log.Println("Service started on port " + port)
    err := http.ListenAndServe(":"+port, nil)
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
        return
    }
}
func defaultHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "<h1>Home Page</h1>")
}
func contactHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "<h1>Contact Us</h1>")
}
func loginHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "<h1>Login Page</h1>")
}
```

Execute the following steps to complete this activity:

1. Create a folder to store the activity files.
2. Create a `main.go` file with the code provided in the preceding code block.
3. Create a multi-stage `Dockerfile` with two stages. The first stage will use the `golang` image. This stage will build the Golang application using the `go build` command. The second stage will use an `alpine` image. This stage will copy the executable from the first stage and execute it.
4. Build and run the Docker image.
5. Once completed, stop and remove the Docker container.

You should get the following output when you navigate to the URL `http://127.0.0.1:8080/`:



Home Page

Summary

We started this lab by defining a normal Docker build and creating a simple Golang Docker image using the normal Docker build process. Then we observed the size of the resulting Docker image and discussed how a minimal-sized Docker image can speed up the build and deployment times for Docker containers and enhance security by reducing the attack surface.

We then used the builder pattern to create minimal-sized Docker images, utilizing two `Dockerfiles` and a shell script in this process to create the image. We explored multi-stage Docker builds---a new feature introduced to Docker in version 17.05 that can help to eliminate the operational burden of having to maintain two `Dockerfiles` and a shell script. Finally, we discussed the best practices for writing `Dockerfiles` and how these best practices can ensure reduced build time, reduced image size, and increased security, while increasing the maintainability of the Docker image.

In the next lab, we will cover `docker-compose` and how it can be used to define and run multi-container Docker applications.