

Chapter-6 (Dockerfile)

01 August 2022 02:58

Dockerfile is a script, composed of various commands (instructions) and arguments listed successively to automatically perform an action on a base image in order to create (or form) a new one.

Dockerfile comments:

```
# Dockerfile Starts
FROM openjdk:17-alpine
# Add class file to working directory
ADD HelloWorld.class HelloWorld.class
# Execute class file by firing
CMD ["java", "HelloWorld"]
# Dockerfile Starts
```

Syntax to write instruction and its argument within a Dockerfile is:

INSTRUCTION arguments

- Instruction can be given in lowercase or uppercase letters. But to differentiate from the instructions and arguments we use uppercase letters.
- Docker runs instructions in a Dockerfile in order. A Dockerfile must start (ARG can exist) with a **FROM** instruction. **The FROM instruction specifies the Base Image from which you are building.**
- A Docker image consists of read-only layers each of which represents a Dockerfile instruction. The layers are stacked and each one is a delta of the changes from the previous layer.
- When we run an image and generate a container, we add a new writable layer (the “container layer”) on top of the underlying layers. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer.

Understand Instructions :

- **FROM** : It defines the base image to use to start the build process. This instruction is used to set the base image for subsequent instructions. It is mandatory to set this in a Dockerfile. You can use it any number of times though.

```
FROM openjdk:17-alpine
ADD HelloWorld.class HelloWorld.class
CMD ["java", "HelloWorld"]
```

- **MAINTAINER**: This is a non-executable instruction used to indicate the author of the Dockerfile. It should come nonetheless after FROM

```
FROM openjdk:17-alpine
MAINTAINER CodeHop
ADD HelloWorld.class HelloWorld.class
CMD ["java", "HelloWorld"]
```

- **ADD**: This instruction is used to add files from local as well from remote to current image.

```
FROM openjdk:17-alpine
MAINTAINER CodeHop
ADD HelloWorld.class HelloWorld.class
CMD ["java", "HelloWorld"]
```

■ → Local directory
■ → Container directory

- **RUN**: This instruction allows us to execute a command on top of an existing layer and create a new layer with the result of command execution. This is what runs within the container at build time.
 - ◆ Use case let us consider we are writing into a file using java.
 - ◆ Java writes the file in log directory
 - ◆ But log directory do not exist inside the docker container when created
 - ◆ We can achieve this using RUN command

- Sample Java code :

```
import java.io.*;
class FileWriterLineByLine{
```

- Sample Java code :

```
import java.io.*;
class FileWriterLineByLine{
    public static void main(String[] args) throws Exception {
        File fout = new File("log/out.txt");
        FileOutputStream fos = new FileOutputStream(fout);
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(fos));
        for (int i = 1; i <=10; i++) {
            bw.write("Printing line "+i);
            Thread.sleep(10000);
            bw.newLine();
        }
        bw.close();
    }
}
```

FileWriterLineByLine.java

- Sample Dockerfile :

```
FROM openjdk:17-alpine
MAINTAINER CodeHop
RUN mkdir log
ADD FileWriterLineByLine.class FileWriterLineByLine.class
CMD ["java","FileWriterLineByLine"]
```

- To run a container in detached mode (run a container without waiting it to get up) we use **-d** as input to run command

```
docker run -d -name filewriter filewriter:v1
```

- To run a container in interactive mode (run a container and see the output inside container until container stops) we use **-it** as input to run command

```
docker run -it -name filewriter filewriter:v1
```

- To run a command inside a container while is container is up & running in detached mode we use **exec** (Execute) command

```
docker exec -it container_id sh
```

- We can also use **RUN** command of Dockerfile to install or update a binary which is not present inside a docker container
 - ◆ For case lets say we want to update the Linux distribution before starting out build inside container for the above application

- Sample Dockerfile :

```
FROM openjdk:17-alpine
MAINTAINER CodeHop
RUN apk update && mkdir log
ADD FileWriterLineByLine.class FileWriterLineByLine.class
CMD ["java","FileWriterLineByLine"]
```

- **ENV**: This instruction can be used to set the environment variables in the container. These variables consist of “key-value” pairs which can be accessed within the container by scripts and applications alike.

- Let us consider a case where we want to make the database connection in our program based upon the environment
- For an instance :
 - ▶ If it is PT environment connect to IP 192.168.0.1
 - ▶ If it is STAGE environment connect to 192.168.0.2

- Sample Java code :

```
class ConnectToDatabaseBasedOnEnvVar{
    public static void main(String[] args) throws Exception {
```

```

class ConnectToDatabaseBasedOnEnvVar{
    public static void main(String[] args) throws Exception {
        String dbHostIp = System.getenv("DB_HOST_IP");
        if(null!=dbHostIp){
            System.out.println("Connecting to DB Host with IP "+dbHostIp);
        }else{
            throw new Exception("No DB Host IP Configured");
        }
    }
}

```

- Sample Dockerfile : [docker build ./ -t connecttodatabasebasedonenvvar:v1 , docker run -it --name connecttodb connecttodatabasebasedonenvvar:v1]

```

FROM openjdk:17-alpine
MAINTAINER CodeHop
ENV DB_HOST_IP='192.168.0.1'
ADD ConnectToDatabaseBasedOnEnvVar.class ConnectToDatabaseBasedOnEnvVar.class
CMD ["java","ConnectToDatabaseBasedOnEnvVar"]

```

- ARG** : These are also known as build-time variables. They are only available from the moment they are 'announced' in the Dockerfile with an ARG instruction up to the moment when the image is built.
 - ▶ If we look at the above implementation of Dockerfile it is tied to IP of PT environment.
 - ▶ Which means we need different Dockerfile for different environment
 - ▶ This is consider as bad practice, our Dockerfile should be constructed in such a way that it is compatible to build image for all the environment.
 - ▶ To resolve or make our Dockerfile we can ask user to pass argument at the build time & then we can utilize that image at the time of container creation.

- Sample Dockerfile would be :

```

FROM openjdk:17-alpine
ARG DB_HOST_IP_ARG
ENV DB_HOST_IP=$DB_HOST_IP_ARG
MAINTAINER CodeHop
ADD ConnectToDatabaseBasedOnEnvVar.class ConnectToDatabaseBasedOnEnvVar.class
CMD ["java","ConnectToDatabaseBasedOnEnvVar"]

```

- ▶ Command to build docker image would be like :

```
docker build . -t connecttodb:v1 --build-arg DB_HOST_IP_ARG='192.168.0.2'
```

- ▶ Command to build docker image would be like :

```
docker run -it --name connecttodb connecttodb:v1
```

- CMD** : The major difference between CMD and RUN is that CMD doesn't execute anything during the build time. It just specifies the intended command for the image. Whereas RUN actually executes the command during build time.

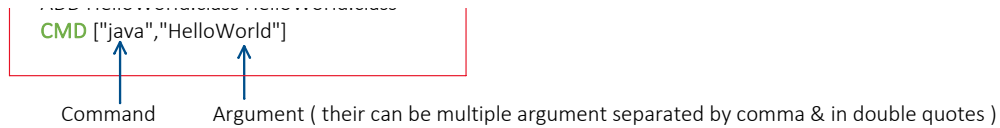
**** Note: there can be only one CMD instruction in a Dockerfile, if you add more, only the last one takes effect.**

- Sample Dockerfile :

```

FROM openjdk:17-alpine
MAINTAINER CodeHop
ADD HelloWorld.class HelloWorld.class
CMD ["java","HelloWorld"]

```



- **COPY** : This instruction is used to copy files and directories from a specified source to a destination (in the file system of the container).
 - ▶ Consider a case we have an application where it reads mail content from text files
 - ▶ And then it sends the mail one by one by reading text content
 - ▶ For this our application is dependent upon multiple files
 - ▶ Files can be stored in a directory
 - ▶ Note when our program will execute it is dependent upon these files which are in a directory
 - ▶ Since there are limitations with the ADD command we need to use the COPY command to achieve it.

- Sample Dockerfile :

```
FROM openjdk:17-alpine
MAINTAINER CodeHop
ADD SendAllMail.class SendAllMail.class
COPY ./content ./content
CMD ["java", "SendAllMail"]
```

- Sample Java code :

```
import java.io.*;
import java.nio.file.Files;
import java.util.stream.*;
class SendAllMail{
    public static void main(String[] args) throws IOException{
        File directoryPath = new File("content/");
        File fileList[] = directoryPath.listFiles();
        for(File file : fileList) {
            StringBuilder contentBuilder = new StringBuilder();
            try (Stream<String> stream = Files.lines(file.toPath())){
                stream.forEach(s -> contentBuilder.append(s).append("\n"));
            }catch (IOException e){
                e.printStackTrace();
            }
            System.out.println("Mail Sent :: "+contentBuilder.toString());
        }
    }
}
```



SendAllMail

- **WORKDIR** : This is used to set the currently active directory for other instructions such as RUN, CMD, COPY, ADD etc.
 - ▶ Note that if a relative path is provided, the next WORKDIR instruction will take it as relative to the path of the previous WORKDIR instruction.

- Sample Dockerfile :

```
FROM openjdk:17-alpine
MAINTAINER CodeHop
WORKDIR /user
WORKDIR home
CMD ["pwd"]
```

- Command for building & running:

- ▶ `docker build . -t workdir:v1`
- ▶ `docker run -it --name workdir workdir:v1`

- **LABEL** : You can assign metadata in the form of key-value pairs to the image using this instruction.
 - ▶ It is important to notice that each LABEL instruction creates a new layer in the image.

- ▶ It is best to use as few LABEL instructions as possible.

- Sample Dockerfile :

```
FROM openjdk:17-alpine
MAINTAINER CodeHop
LABEL version="1.0" description="This is application is used to send mail content in text file"
ADD SendAllMail.class SendAllMail.class
COPY ./content ./content
CMD ["java","SendAllMail"]
```

- Command for building & describing docker object:

- ▶ `docker build . -t sendallmaillabel:v1`

- ▶ `docker inspect sendallmaillabel:v1`

- **VOLUME** : It is a mechanism that is used for persistent data storage generated by Docker containers.
 - ▶ Docker volume can be shared with more than one container.
 - ▶ Docker volumes are managed by Docker itself.
 - ▶ It is better to store data in a volume rather than storing it in a container's writable layer as it does not increase the size of the container.
 - ▶ Containers are ephemeral, which means container lifetime is short, and containers get destroyed, and so the data as well.

- Sample Dockerfile :

```
FROM openjdk:17-alpine
MAINTAINER CodeHop
LABEL version="1.0" description="This is application is used to send mail content in text file"
ADD SendAllMail.class SendAllMail.class
VOLUME ./content
CMD ["java","SendAllMail"]
```

- Command for building, running & describing docker object:

- ▶ `docker build . -t dockervolume:v1`

- ▶ `docker run --name dockervolume dockervolume:v1 sleep 6000`

- ▶ `docker volume ls`

- ▶ `docker volume inspect volume_name`

- When we do docker volume inspect we will get the path of volume where it is stored locally (`cd /var/lib/docker/volumes`)
- With this approach their is limitation that volume will be store at (`cd /var/lib/docker/volumes`) some where on local host and the volume exist until and unless running or exited container exist.
- To overcome this limitation docker provide us a way to mount a custom directory as a docker volume at runtime while running a container where we can perform read and write operation.

- Sample Dockerfile :

```
FROM openjdk:17-alpine
MAINTAINER CodeHop
LABEL version="1.0" description="This is application is used to send mail content in text file"
WORKDIR /var/opt
RUN mkdir app
WORKDIR ./app
ADD SendAllMail.class SendAllMail.class
CMD ["java","SendAllMail"]
```

```
WORKDIR ./app
ADD SendAllMail.class SendAllMail.class
CMD ["java","SendAllMail"]
```

- Command for building, running & describing docker object:

▶ `docker build . -t dockervolume:v1`

▶ `docker run -it -v ./content:/var/opt/app/content --name dockervolume dockervolume:v1`

Local directory path

Container path

- ▶ For windows :

`docker run -it -v ${PWD}/content:/var/opt/app/content --name dockervolume dockervolume:v1`

- **USER** : By default, a Docker Container runs as a Root user. This poses a great security threat if we deploy our applications on a large scale inside Docker Containers.

- You can change or switch to a different user inside a Docker Container using the USER Instruction.
- For this, you first need to create a user and a group inside the Container.

- Sample Dockerfile :

```
FROM openjdk:17-alpine
MAINTAINER CodeHop
LABEL version="1.0" description="This is application is used to send mail content in text file"
RUN apk add shadow && groupadd appgroup \
    && useradd -G appgroup mailuser \
    && mkdir -p /var/opt/app
WORKDIR /var/opt/app/
ADD SendAllMail.class SendAllMail.class
RUN chown -R "mailuser:appgroup" /var/opt/app
USER mailuser
CMD ["java","SendAllMail"]
```

- Command for building, running & describing docker object:

▶ `docker build . -t dockeruser:v1`

▶ `docker run -v ${PWD}/content:/var/opt/app/content/ --name dockeruser -t dockeruser:v1 sleep 6000`

- ▶ To see the output of console we can use **logs** command as :

`docker logs Id_of_container`

- **ENTRYPOINT** : It is similar like CMD command in docker file but the basic difference between the CMD & ENTRYPOINT command is CMD command can be overridden while starting a container where as ENTRYPOINT command can not be overridden while starting a container, lets look at example as below.

- Sample Java Code :

```
import java.util.Arrays;
class CommandLineArgument{
    public static void main(String[] args){
        if(args != null && args.length>0){
            System.out.println("Command line argument found with value : "+Arrays.toString(args)+" env");
        }else{
            System.out.println("No command line argument found executing with default dev env");
        }
    }
}
```

- Sample Dockerfile :

```
FROM openjdk:17-alpine
MAINTAINER CodeHop
ADD CommandLineArgument.class CommandLineArgument.class
CMD ["java","CommandLineArgument"]
```

- Build command

▶ `docker build . -t commandlinearg:v1`

- ◆ Now lets consider we want to pass command line argument to above image at run time

- ◇ Command can be :

▶ `docker run -it --name commandlinearg commandlinearg:v1 dev`

This **dev** will replace
 CMD ["java","CommandLineArgument"]
 TO CMD ["dev"]

- ◇ This make the entire program in valid , however if we pass full command like :

▶ `docker run -it --name commandlinearg commandlinearg:v1 java CommandLineArgument dev`

- ◇ But the above way is has its own drawback (if image is build by third party, an additional instruction is required to run container from image).

- Solution to above is to use ENTRYPOINT command in docker file.

- Sample Dockerfile:

```
FROM openjdk:17-alpine
MAINTAINER CodeHop
ADD CommandLineArgument.class CommandLineArgument.class
ENTRYPOINT ["java","CommandLineArgument"]
```

- Build command and run command

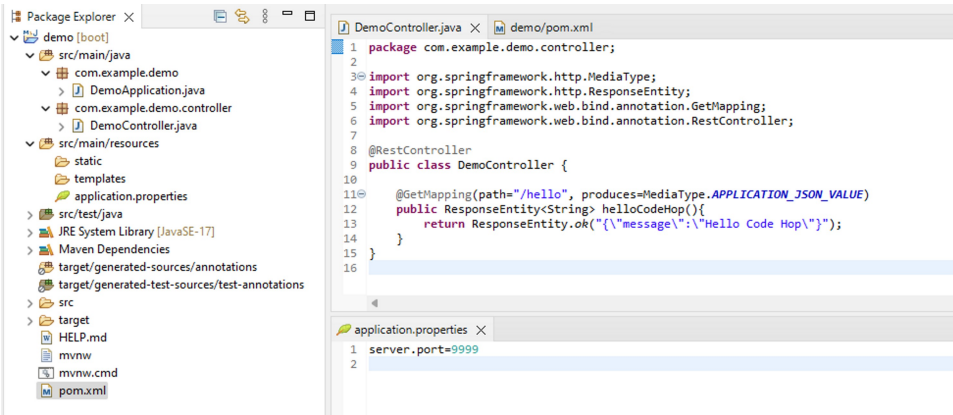
▶ `docker build . -t commandlinearg:v2`

▶ `docker run -it --name commandlineargv2 commandlinearg:v2 prod`

This prod will be appended to ENTRYPOINT Command
 ENTRYPOINT ["java","CommandLineArgument"]
 Will act like → ENTRYPOINT ["java","CommandLineArgument", "prod"]

- **EXPOSE** : While container are isolated with network we can not communicate with container service outside of container. To overcome this problem docker provide us EXPOSE instruction which allow running container containing our service to listen to the specified port.

- For use case let us a sample Springboot application :



- ◆ Springboot application is running on port **9999**
- ◆ It has end-point call hello with get-Mapping
- ◆ Triggering which give us output as : {"message":"Hello Code Hop"}

- Sample Dockerfile:

```
FROM openjdk:17-alpine
MAINTAINER CodeHop
RUN mkdir -p /var/opt/app
WORKDIR /var/opt/app/
ADD demo.jar demo.jar
EXPOSE 9999
ENTRYPOINT ["java", "-jar", "demo.jar"]
```

- Command to build and run :

- ◆ `docker build . -t demoserver:v1`
- ◆ `docker run -d -p9999:8081 --name demoserver demoserver:v1`
 - ◆ **-p** : Runtime instruction for port mapping with docker internal to localhost
 - ◆ **9999** : Docker container exposed port
 - ◆ **8081** : localhost port to which docker container port is mapped