Experiment 10

Aim: To learn Dockerfile instructions, build an image for a sample web application using DOCKERFILE..

Theory:
What is a Dockerfile?

A Dockerfile is essentially a text file (literally written as "Dockerfile" with no extension) that contains instructions for creating a Docker image. These instructions are written in the following format:
DIRECTIVE argument
Although the DIRECTIVE is case-insensitive, it is recommended to write all directives in uppercase to differentiate them from arguments. A Dockerfile usually consists of multiple lines of instructions that are executed sequentially by the Docker engine during the image-building
process.
Now that you understand the purpose of a Dockerfile, let's get our hands dirty by building one for a sample Python application. Building a Dockerfile for a Sample Python/Flask Application
The application we'll be working with is a simple Flask app with only one home route that returns Hello, World!.
Let's start by setting up the Flask application. Open your favorite code editor and create a new directory for your project. In this directory, create a new file named app.py and add the following Python code:

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello():
    return 'Hello, World!'
```

Now, let's build the Dockerfile.
In the same directory as your app.py, create a new file named Dockerfile (with no file extension). This is where you'll write the instructions for Docker to build your image.
Now, follow the steps below to create the Dockerfile:

1. Specify the base image
The very first instruction you write in a Dockerfile must be the FROM directive, which specifies the base image. The base image is the image from which all other layers in your Docker image will be built. It's the foundation of your Docker image, much like the foundation of a building.
Add the following instruction to your Dockerfile:

```
FROM python:3.11-slim
```

Here, we're telling Docker to use the official Python Docker image, and more specifically, the 3.11-slim version. This slim variant of Python Docker image is a minimal version that excludes some packages to make the image smaller. Note that the base image you specify will be
downloaded from Docker Hub, Docker's official image registry. The reason we're using Python as the base image is that the application containerization is written in Python. The Python base image includes a Python interpreter, which is necessary to run Python code, as well as a number of commonly used Python utilities and libraries. By using the Python base image, we're ensuring that the environment within the Docker container is preconfigured for running Python code.

2. Set the working directory
Once you've chosen the base image, the next step is to determine the working directory using the WORKDIR directive. Insert the following line after the FROM directive:

```
WORKDIR /app
```

Here, we're telling Docker to create a directory named app in the Docker container and use it as the current working directory. All instructions that FROM python:3.11-slim WORKDIR /app follow (like RUN, COPY, and CMD) will be run in this directory inside the container. Think of this as typing the command cd /app in a terminal to change the current working directory to /app. The difference here is that it's being done within the Docker container as part of the build process. A working directory within the container is necessary because it designates a specific location for our application code within the container and determines where commands will be run from. If we don't set a working directory, Docker won't have a clear context for where your application is located, which would make it harder to interact with.

3. Install dependencies Once the working directory is set, the next step is to install the dependencies. Our Python application relies on the Flask web framework, which manages requests, routes URLs, and handles other web-related tasks. To install Flask, add the following instruction in your Dockerfile just under the WORKDIR directive:

```
RUN pip install flask==2.3
```

Here, we're instructing Docker to use pip (a package installer for Python) to install the specific version of Flask we need for our application.

4. Copy application files to the container After setting up the working directory and installing the necessary dependencies, we're now ready to copy the application files into the Docker container. To do this, add the following instruction just below the RUN directive:

```
COPY . /app
```

This line copies everything in the current directory (denoted by ".") on our host machine into the /app directory we previously set as our working directory within the Docker container. It's like using the cp command in the terminal to copy files from one directory to another, but in this context, it's copying files from your local machine to the Docker container. Why do we need to do this? It's simple. Without this step, the Docker container wouldn't have access to our application's code, making it impossible to run our app.

5. Specify the environment variable Once the application files are copied, we need to set up the FLASK_APP environment variable for our Docker container using the ENV directive. Now, you may be wondering why we need this environment variable in the first place. In our app.py file, we create an instance of the Flask application and assign it to the variable app. This application instance is what Flask needs to run, and it's located in the app.py file. When starting our Flask application using the flask run command (which we'll discuss in the next section), Flask must know where to locate the application instance to run. Flask uses the FLASK_APP environment variable to find this instance. Hence, we need to use the ENV directive to set the value of FLASK_APP to app.py. To do this, add the following line under the RUN directive:

```
ENV FLASK_APP=app.py
```

This line ensures Flask knows exactly where to find the application instance to run, which in our case is app.py.

6. Define the default command The last instruction that we need for our application is to specify the default command that will be executed when the Docker container starts:

```
CMD ["flask", "run", "--host=0.0.0.0", "--port=5000"]
```

from the image, we'll build from this Dockerfile. Insert the following instruction below the ENV directive:
Here's what each part of the argument passed to the CMD directive does:

- flask: This is the program that we want to run. In this case, it's the Flask command-line interface.
- run: This command instructs Flask to start a local development server.
- --host=0.0.0.0: This argument tells the Flask server to listen on all public IPs. In the context of Docker, this means the Flask application will be accessible on any IP address that can reach the Docker container.
- --port=5000: This argument specifies the port number that the Flask server will listen on. Port 5000 is the default port for Flask, but it's good practice to explicitly declare it for clarity.

After this, our Dockerfile is ready. It should look like this
:

```
FROM python:3.11-slim
WORKDIR /app
RUN pip install flask==2.3
COPY . /app
ENV FLASK_APP=app.py
CMD ["flask", "run", "--host=0.0.0.0", "--port=5000"]
```

It's worth noting that the directives we used in the Dockerfile for our Python app aren't the only ones available in Docker. But they are the ones you'll often encounter when working with Dockerfiles.

7. Create a .dockerignore file Before we go ahead and build our Docker image, we need to take care of one last thing. Remember the following COPY directive?

```
COPY . /app
```

This line instructs Docker to copy everything from our current directory to the app directory inside the container, which includes the Dockerfile itself. But, the Dockerfile isn't required for our app to work—it's just for us to create the Docker image. So, we need to ensure that the Dockerfile doesn't get copied to the app directory in the container. Here's how we do it:
Create a new file called .dockerignore in the same directory as your Dockerfile. This file works much like a .gitignore file if you're familiar with Git. Then, add the word Dockerfile to this file. This tells Docker to ignore the Dockerfile when copying files into the container. Now that we've prepared everything, it's time to build our Docker image, run a container from this image, and test our application to see if everything works as expected.

Building and running the Docker Image:
Open a terminal and navigate to the directory where your Dockerfile is located. Now, run the following command to create an image named sample-flask-app:v1 (you can name the image anything you prefer):

```
$ docker build . -t sample-flask-app:v1
```

In the command above, the dot (.) after the build command indicates that the current directory is the build context. We're using the -t flag to tag the Docker image with the name sample-flask-app and version v1. After running this command, you'll see an output similar to this:

```
$ docker build . -t sample-flask-app:v1
[+] Building 17.7s (10/10) FINISHED docker:default
 => [internal] load build definition from Do  0.0s
 => => transferring dockerfile: 192B          0.0s
 => [internal] load metadata for docker.io/l  3.5s
 => [auth] library/python:pull token for reg  0.0s
 => [internal] load .dockerignore             0.0s
 => => transferring context: 50B              0.0s
 => [1/4] FROM docker.io/library/python:3.11  9.0s
 => => resolve docker.io/library/python:3.11  0.0s
 => => sha256:1103112ebfc46e 3.51MB / 3.51MB  1.9s
 => => sha256:b4b80ef7128d 12.87MB / 12.87MB  2.7s
 => => sha256:a2eb07f336e4f1 1.65kB / 1.65kB  0.0s
 => => sha256:4bcdb5d5bc81ca 1.37kB / 1.37kB  0.0s
 => => sha256:15646a3fa12dde 6.93kB / 6.93kB  0.0s
 => => sha256:8a1e25ce7c4f 29.12MB / 29.12MB  4.8s
 => => sha256:cc7f04ac52f8a3bad5 243B / 243B  2.4s
 => => sha256:87b8bf94a2ace2 3.41MB / 3.41MB  3.3s
 => => extracting sha256:8a1e25ce7c4f75e372e  1.8s
 => => extracting sha256:1103112ebfc46e01c0f  0.2s
 => => extracting sha256:b4b80ef7128dc9bd114  1.0s
 => => extracting sha256:cc7f04ac52f8a3bad5b  0.0s
 => => extracting sha256:87b8bf94a2ace2b005d  0.7s
 => [internal] load build context             0.0s
 => => transferring context: 194B             0.0s
 => [2/4] WORKDIR /app                        0.2s
 => [3/4] RUN pip install flask==2.3          4.7s
 => [4/4] COPY . /app                         0.0s
 => exporting to image                        0.2s
 => => exporting layers                       0.2s
 => => writing image sha256:c6879156c7750c89  0.0s
 => => naming to docker.io/library/sample-fl  0.0s

What's Next?
  View a summary of image vulnerabilities and recommendations → docker scout quickview
```

To make sure the image sample-flask-app:v1 has been successfully created, run the following command to check the list of Docker images on your system:

```
$ docker image ls
```

The resulting output should look something like this:

```
$ docker image ls
REPOSITORY          TAG       IMAGE ID       CREATED          SIZE
sample-flask-app    v1        c6879156c775   10 seconds ago   147MB
mongo               latest    24041ceefc56   6 days ago       755MB
```

In the list, you should see sample-flask-app:v1, which confirms the image is now in our system. Now, run the sample-flask-app:v1 image as a container by executing the following command:

```
$ docker container run -d -p 5000:5000 sample-flask-app:v1
```

The -d flag is short for --detach and runs the container in the background. The -p flag is short for --publish and maps port 5000 of the host to port 5000 of the Docker container. After running this command, you'll see an output like this:

```
$ docker container run -d -p 5000:5000 sample-flask-app:v1
ff37071dd4cef95cc1dc2ce7e145019339cfaec54575659f72aea4e560238f8c
```

The long string you see printed in the terminal is the container ID. To make sure the container is running, list the currently active Docker containers by running the following command:

```
$ docker container ls
```

You should see something like this:

```
$ docker container ls
CONTAINER ID   IMAGE              COMMAND                  CREATED         STATUS         PORTS                    NAMES
c301c50152ac   sample-flask-app:v1   "flask run --host=0…"   14 seconds ago   Up 12 seconds   0.0.0.0:5000->5000/tcp   xenodochial_mendel
```

The container is up and running as expected. Our Flask application is now running inside the container. To test it, open a web browser and go to http://localhost:5000. You should see the message Hello, World! displayed like this:

Hello, World!

Conclusion: We have learnt Dockerfile instructions, built an image for a sample web application using DOCKERFILE.