

## Experiment 7 : Docker Basics

### Aim :

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

### Theory :

Dockerfiles are the cornerstone of creating Docker images. They contain a set of instructions that automate the process of building a Docker image, specifying everything from the base operating system to the application code, dependencies, and configuration settings.

---

### 1. What is a Dockerfile?

A **Dockerfile** is a plain text file that defines the steps required to build a Docker image. It contains a series of commands (or instructions) that specify how the image should be constructed.

- **Purpose:** Automate the creation of Docker images for reproducibility, scalability, and consistency.
  - **Format:** Written in a simple scripting language, using instructions like **FROM**, **RUN**, **COPY**, **CMD**, etc.
- 

### 2. Basic Structure of a Dockerfile

A typical Dockerfile looks like this:

```
# Use an official Python runtime as a parent image
FROM python:3.9-slim
```

```
# Set the working directory inside the container
WORKDIR /app
```

```
# Copy the current directory contents into the container at /app
COPY . /app

# Install any necessary dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

---

### 3. Common Dockerfile Instructions

#### 1. FROM (Base Image)

- **Purpose:** Specifies the base image for your Docker image.

##### Example:

```
FROM ubuntu:20.04
FROM node:14
FROM python:3.9-slim
```

- **Note:** This is the first instruction and is mandatory in most cases.

#### 2. WORKDIR (Set Working Directory)

- **Purpose:** Defines the directory inside the container where subsequent instructions will be executed.

##### Example:

```
WORKDIR /app
```

### 3. COPY (Copy Files)

- **Purpose:** Copies files or directories from the host system into the container.

#### Example:

```
COPY . /app
```

- **Variants:**
  - **COPY <src> <dest>:** Copies a file or directory from the build context to the container.
  - **ADD** is similar but supports remote URLs and tar file extraction.

### 4. RUN (Execute Commands)

- **Purpose:** Executes commands inside the container during the image build process.

#### Example:

```
RUN apt-get update && apt-get install -y curl  
RUN pip install --no-cache-dir -r requirements.txt
```

- **Tip:** Each **RUN** creates a new layer in the image. Combine commands with **&&** to reduce image size.

### 5. EXPOSE (Expose Ports)

- **Purpose:** Informs Docker that the container will listen on the specified network ports at runtime.

#### Example:

```
EXPOSE 80
```

- **Note:** This does **not** publish the port; it's just a way to document which ports should be exposed.

## 6. **ENV** (Set Environment Variables)

- **Purpose:** Sets environment variables inside the container.

**Example:**

```
ENV APP_ENV=production
```

## 7. **CMD** (Default Command)

- **Purpose:** Specifies the default command to run when the container starts.

**Example:**

```
CMD ["python", "app.py"]
```

- **Key Points:**
  - Only **one** **CMD** instruction is allowed.
  - If you provide a command when running the container (**docker run**), it will override **CMD**.

## 8. **ENTRYPOINT** (Set Entry Point)

- **Purpose:** Defines a command that will always be executed when the container starts.

**Example:**

```
ENTRYPOINT ["python"]  
CMD ["app.py"]
```

- **Difference from **CMD**:** **ENTRYPOINT** is not overridden unless explicitly done with **--entrypoint** in **docker run**.

---

## 4. Building Images from a Dockerfile

To build an image, use the **docker build** command:

```
docker build -t myapp:latest .
```

- **-t myapp:latest**: Tags the image as **myapp** with the **latest** tag.
- **.**: Specifies the build context (the current directory).

#### Build Options:

- **-f <file>**: Specify a custom Dockerfile name.
  - **--no-cache**: Build the image without using the cache.
  - **--build-arg <arg>**: Pass build-time arguments.
- 

## 5. Managing Docker Images

#### List Images:

```
docker images
```

#### Remove an Image:

```
docker rmi myapp:latest
```

#### Run a Container from an Image:

```
docker run -p 8080:80 myapp:latest
```

---

## 6. Multi-Stage Builds (Advanced)

Multi-stage builds help reduce image size by separating the build environment from the runtime environment.

```
# Stage 1: Build stage
FROM node:14 AS build
WORKDIR /app
COPY package.json ./
RUN npm install
```

```
COPY . .
```

```
# Stage 2: Production stage  
FROM node:14-slim  
WORKDIR /app  
COPY --from=build /app /app  
CMD ["node", "server.js"]
```

- This technique helps keep the final image lean by excluding unnecessary build tools.
- 

## 7. Best Practices for Dockerfiles

1. **Use Minimal Base Images:** e.g., `alpine` for small image sizes.
2. **Leverage Caching:** Order instructions from least to most frequently changing.
3. **Reduce Layers:** Combine `RUN` commands with `&&`.
4. **Avoid Root:** Run applications as non-root users when possible.
5. **Clean Up:** Remove unnecessary files after installation to reduce image size.

Screenshots:

```

1 const express = require("express");
2 const app = express();
3 const PORT = process.env.PORT || 5000;
4 app.get("/", (req, res) => {
5   res.status(200).json({ msg: "Hello, Docker :" });
6 });
7
8 const init = async () => {
9   try {
10     app.listen(PORT, () => {
11       console.log(`Server is Listening on port ${PORT}...`);
12     });
13   } catch (error) {
14     console.log("There was an error : ", error);
15   }
16 };
17 init();

```

```

1 {
2   "name": "docker_demo",
3   "version": "1.0.0",
4   "description": "",
5   "main": "src/server.js",
6   "scripts": {
7     "start": "node src/server.js"
8   },
9   "keywords": [],
10  "author": "taha",
11  "license": "ISC",
12  "dependencies": {
13    "express": "^5.1.0"
14  }
15 }

```

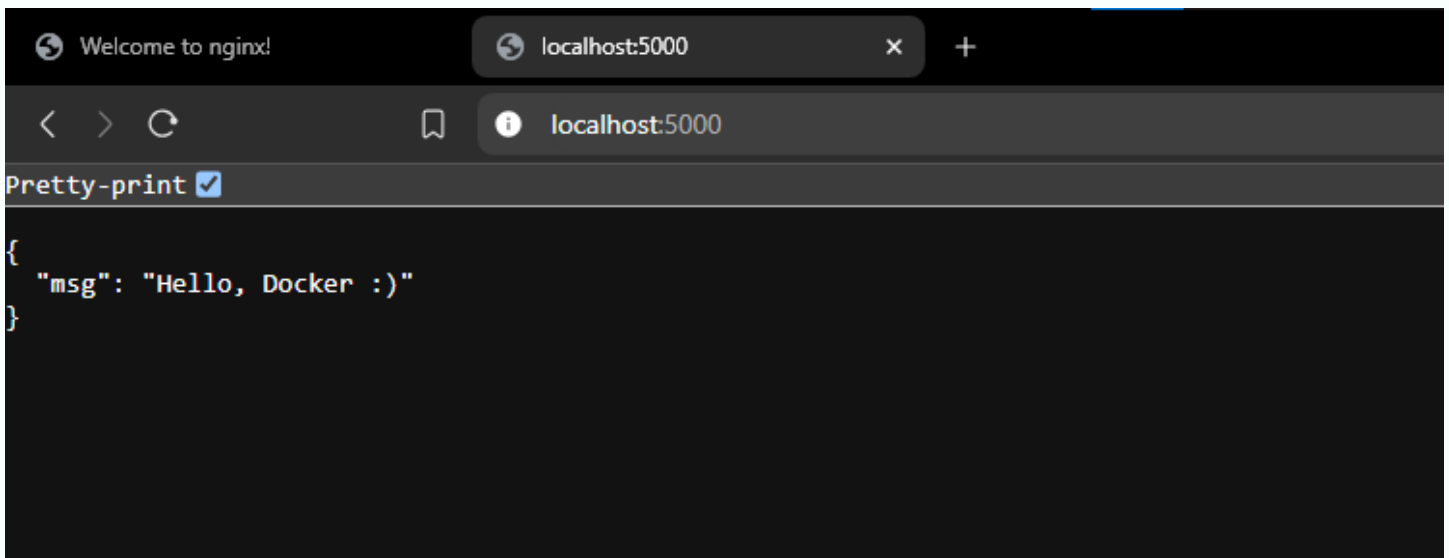
```

10 FROM node:19-alpine
9
8 COPY package.json /app/
7 COPY src /app/
6
5 WORKDIR /app
4
3 RUN npm install
2
1 CMD ["node", "server.js"]
11

```







## Conclusion :

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