<u>Task 6: Docker</u>

Docker: Docker is a platform used to develop, ship, and run applications inside lightweight, portable containers.

Docker Containers:

- A container is a lightweight, standalone, executable package of software that includes everything needed to run an application: the code, runtime, libraries, and dependencies.
- Containers are isolated from each other and the host system, which makes them portable and easy to deploy on any environment.
- Unlike traditional virtual machines, containers do not require their own OS. They share the OS kernel of the host system.

Docker Images:

Docker images are the blueprints or templates used to create Docker containers. An image is a **read-only file system** that contains everything needed to run an application, including the application code, libraries, environment variables, dependencies, and configurations. When you run a Docker image, it creates a **container** that can execute the application in the isolated environment defined by the image.

- They are immutable.
- They are tagged.
- It consists of layers.

Layers of Docker Images:

1. Base Image:

   This is the foundation of a Docker image, usually containing a minimal operating system or file system. For example, a base image might be an official Ubuntu, Alpine Linux, or Debian image.
   Example: `FROM ubuntu` or `FROM alpine`
   The base layer contains the essential files and directories needed to run any software but doesn't have any additional application-specific software yet.

2. Layer of dependencies:

   This layer contains the installation of libraries, packages, or other dependencies needed by the application to function. It may include system libraries, utilities, or application-specific libraries.
   This layer is often created when you run commands like `RUN apt-get install` or `RUN pip install` in your `Dockerfile`.
   Example: Installing Node.js libraries or Python packages like `RUN apt-get install -y python3`.

3. Application Layer:

   This layer contains the application code or binaries that are being packaged in the Docker image. It's usually the part that is added or copied into the container by commands like `COPY` or `ADD`.
   Example: `COPY . /app` or `ADD myapp.tar.gz /app`

4. Configuration Layer:

   This layer is responsible for setting up environment variables, configuration files, and any other adjustments to the container environment.

   Example: `ENV NODE_ENV=production`, `RUN echo "myconfig" > /app/config.txt`

5. Runtime Layer:
   The executable layer typically includes the runtime environment and binaries needed to execute the application. This can include things like web servers (e.g., Nginx, Apache), databases, or other runtime tools.
   Example: `RUN npm run build` or `CMD ["python", "app.py"]`

6. Metadata Layer:
   This layer is used to store metadata, such as labels, author information, and the default command to run when a container starts. It helps define the behavior and characteristics of the image.
   Example: `LABEL version="1.0"` or `CMD ["npm", "start"]`

How Layers Work:

- Each of these layers is created by a Dockerfile instruction (e.g., `RUN`, `COPY`, `ADD`, `CMD`, etc.).
- **Immutability**: Once a layer is created, it is immutable and cached. This allows Docker to reuse layers in future builds, improving efficiency by not recreating layers that haven't changed.
- **Layer Caching**: Docker caches layers to speed up build times. If the contents of a layer (such as a `RUN` command) don't change, Docker will reuse the cached version of that layer in subsequent builds.
- **Layer Ordering**: The order in which layers are created is important. Docker builds layers sequentially, so layers at the top of the

Dockerfile (like copying dependencies or installing packages) should change less frequently than application-specific layers.

Example of simple docker image:

```
# Base image (layer 1)
FROM ubuntu:20.04

# Install dependencies (layer 2)
RUN apt-get update && apt-get install -y curl

# Set working directory (layer 3)
WORKDIR /app

# Copy application code (layer 4)
COPY . .

# Install application dependencies (layer 5)
RUN npm install

# Set environment variables (layer 6)
ENV NODE_ENV production

# Set the default command (layer 7)
CMD ["npm", "start"]
```

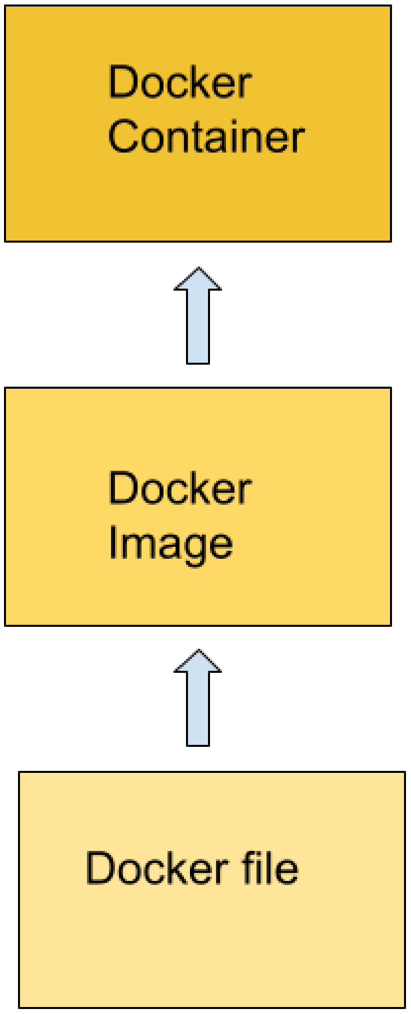
Docker Files:

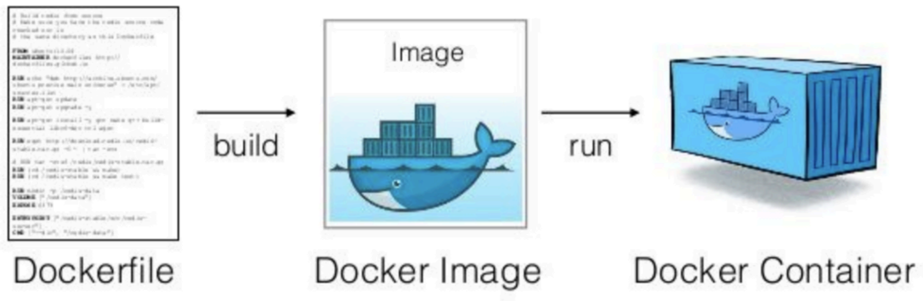
It is a text file that contains a series of instructions that define how to build a Docker image. It provides a blueprint for the Docker image, specifying the steps needed to set up the environment, install dependencies, copy files, and configure the application inside the Docker container.

- **Instructions**: Each line in a Dockerfile contains a specific command or instruction that tells Docker how to build the image.
- **Automates Image Creation**: By using a Dockerfile, you automate the process of creating a Docker image, making it reproducible and consistent.
- **Customizable**: You can specify everything from the base image to the environment variables and even how to run your application in a container.

Common Dockerfile Instructions:

1. FROM:
   Specifies the base image for your Docker image. Every Dockerfile starts with this instruction.
2. COPY:
   Copies files or directories from your local machine (or build context) to the Docker image.
3. ADD:
   Similar to `COPY`, but also has the ability to extract tar files and fetch files from URLs.
4. RUN:
   Executes a command inside the image. It's commonly used to install software dependencies or run setup commands.
5. WORKDIR:
   Sets the working directory inside the container. This directory is where all commands will be executed from.
6. CMD:
   Specifies the default command to run when the container starts. There can only be one `CMD` instruction in a Dockerfile, and it is executed when the container is run.
   Eg: CMD ["npm", "start"]
7. ENTRYPOINT:
   Sets a default executable to run when the container starts. It works similarly to `CMD` but has higher priority.
8. EXPOSE:
   Informs Docker that the container will listen on a specific network port at runtime.
   Eg: EXPOSE 8080
9. ENV:
   Sets environment variables inside the container. These variables can be accessed by applications running inside the container.
   Eg: ENV NODE_ENV=production


Docker Container
Docker Image
Docker file


build
Image
run
Dockerfile
Docker Image
Docker Container