# Experiment No. 2 – Docker Installation and Deployment

## Title:

Docker Installation, Containerization & Deployment

## Aim:

To install Docker, understand and use containerization commands, create Docker containers with various operating system images, and deploy containerized applications using Docker and Docker Hub.

## Introduction to Docker:

Docker is a tool designed to make it easier to create, deploy, and run applications using containers. Containers allow a developer to package up an application with all the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, Docker ensures that the application will run on any other Linux machine regardless of any customized settings that machine might have.

## Procedure with Command Explanations and Impacts:

### Step 1: Install Docker

- Command:

  **sudo apt update**

  Explanation: Updates the list of available packages and their versions.
  Impact: Ensures you're installing the latest version of Docker.

- Command: docker.io is the package that installs the Docker engine

  **sudo apt install docker.io**

  Explanation: Installs the Docker engine from Ubuntu's repository.
  Impact: Enables container creation and image management.

- Command: systemctl commands start and enable Docker to run in the background as a service

  **sudo systemctl start docker**

  Explanation: Starts the Docker daemon (service).
  Impact: Activates Docker to run commands and containers.

- Command:

  **sudo systemctl enable docker**

  Explanation: Enables Docker to start on boot.
  Impact: Ensures Docker is always ready without manual intervention.

### Step 2: Verify Docker Installation

- Command:

  **docker --version**

  Explanation: Displays installed Docker version.
  Impact: Confirms Docker is correctly installed.

- Command:

  **docker info**

  Explanation: Provides system-wide Docker configuration i.e., Displays detailed configuration info such as total containers, images, and OS/architecture
  Impact: Helps verify resource limits, runtime versions, and settings.

### Step 3: Pull and Run Docker Images

- Command:

  **docker pull ubuntu**

  Explanation: Downloads the official Ubuntu operating system image from Docker Hub.
  Impact: Provides a base OS image for container creation.

- Command:

  **docker run -it ubuntu**

  - Explanation: Starts a container in interactive mode with terminal access. `-it` allows interactive terminal access.

    Impact: Allows users to interact with the Ubuntu OS like a virtual machine.

### Step 4: Manage Docker Containers

- Command:

  **docker ps**

  Explanation: Lists running containers that are currently active
  Impact: Used to identify active containers.

- Command:

  **docker ps -a**

  Explanation: Lists all containers, including stopped ones.
  Impact: Helps in reviewing past container executions.

- Command:

  **docker stop <container_id>**

  Explanation: Stops a running container.
  Impact: Useful for halting services or conserving system resources.

- Command:

**docker rm <container_id>**

Explanation: Removes a stopped container.
Impact: Cleans up unused resources and storage.

### Step 5: Manage Docker Images

- Command:

  **docker images**

  Explanation: Lists all downloaded Docker images.
  Impact: Helps identify reusable base images.

- Command:

  **docker rmi <image_name>**

  Explanation: Removes an image from the local system.
  Impact: Frees up disk space and removes obsolete builds.

### Step 6: Create a Flask Application and Dockerize

- Command:
  **app.py**

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def home():
    return "Hello from Docker!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

  Explanation: Defines a simple web application using Flask.
  Impact: Serves as the application code for containerization.

- Command:
  **requirements.txt**

```
flask
```

  Explanation: Lists dependencies needed by the app.
  Impact: Used to automatically install packages inside Docker.

- Command:

  **Dockerfile**

```
# Use official Python base image
FROM python:3.8-slim

# Set working directory in container
WORKDIR /app
```

```
# Copy project files
COPY . .

# Install dependencies
RUN pip install -r requirements.txt

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

Explanation: Contains instructions to build a Docker image for the app.
Impact: Automates the setup of an environment to run the app.

### Step 7: Build and Run Docker Container

- Command:

  **docker build -t flask-app**

  - `flask-app` is the name of the image.
  - refers to the current directory where the Dockerfile is located

  Explanation: Builds a Docker image with the name 'flask-app' from current directory.
  Impact: Packages the app for containerized deployment.

- Command:

  **docker run -d -p 5000:5000 flask-app**

  - `-d` runs the container in detached mode (in the background).
  - `-p 5000:5000` maps the container port to your host system.

  Explanation: Runs the container in detached mode, exposing it on port 5000.
  Impact: Allows access to the app via browser at http://localhost:5000

### Step 8: Push Image to Docker Hub

- Command:

  **docker login**

  Explanation: Authenticates to Docker Hub.
  Impact: Allows pushing/pulling private or public images.

- Command:

  **docker tag flask-app your_dockerhub_username/flask-app**

  Explanation: Tags the image with your Docker Hub username.
  Impact: Prepares the image for upload.

- Command:

  **docker push your_dockerhub_username/flask-app**

Explanation: Uploads the image to your Docker Hub repository.
Impact: Enables sharing and pulling from anywhere.

## Conclusion:

This experiment demonstrated how to install Docker, manage containers and images, and deploy a simple application. Understanding each Docker command and its effect enables students to efficiently develop and manage containerized applications. This is a crucial skill in modern DevOps practices, allowing seamless CI/CD implementation and software delivery.