Session 02: Working with Dockerfile with a multi-stage build , Docker Networking and Volumes

Pre-Lab

1. What is a multi-stage build in Docker?

A **multi-stage build** is a feature of **Docker** that allows using **multiple FROM statements** in a single **Dockerfile**.

Each stage performs a specific task (such as build or runtime), and only the required artifacts are copied to the final image.

Purpose: Build the application in one stage and run it in a smaller, cleaner final image.

2. Why do we use multiple stages in a Dockerfile?

Multiple stages are used to:

- Separate **build-time dependencies** from **runtime dependencies**
- Improve **security** by excluding compilers and tools
- Make images **lighter and faster**
- Follow **best practices** for production containers

3. How does a multi-stage build help reduce image size?

A multi-stage build reduces image size by:

- Discarding intermediate build layers
- Copying **only compiled output** (e.g., binaries, dist/ files) to the final stage
- Excluding unnecessary files such as source code, build tools, and package managers

Result:

Smaller image → Faster pull → Lower storage → Better performance

In-Lab Tasks

1. Create a custom Docker network to enable communication between two on the same host

Communication between containers on same host:

- **check the networks that exists in your docker environment**

  command - docker network ls

```
PS C:\Users\chout> docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
13669d794e73   bridge    bridge    local
8978f2fe82df   host      host      local
f1cc085c9d13   none      null      local
```

- **Create two containers using prebuilt images**

  Commands-

  - **docker run -d --name container1 nginx**
  - **docker run -d --name container2 nginx**

- **list all the containers on your system, including those that are currently running, stopped, or exited**

```
PS C:\Users\chout> docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS     NAMES
6f7b3c362bb4   nginx     "/docker-entrypoint.…"   6 seconds ago    Up 5 seconds    80/tcp    container2
6b2460e5fcd9   nginx     "/docker-entrypoint.…"   25 minutes ago   Up 25 minutes   80/tcp    container1
```

Steps to Create a Network and Ping a Container

1. **Create a Docker Network:** You can create a bridge network using the following command:

   docker network create --driver bridge mynetwork

→Here **mynetwork** is bridge network name

- **check the networks that exists in your docker environment**

  docker network ls

```
PS C:\Users\chout> docker network ls
NETWORK ID     NAME        DRIVER    SCOPE
13669d794e73   bridge      bridge    local
8978f2fe82df   host        host      local
b8a278cfdf23   mynetwork   bridge    local
f1cc085c9d13   none        null      local
PS C:\Users\chout> docker inspect b8a278cfdf23
```

2. Now connect the two containers to network

```
PS C:\Users\chout> docker network connect mynetwork container1
PS C:\Users\chout> docker network connect mynetwork container2
```

3. Check whether containers are connected to network or not

docker inspect <resource_id>

example- docker inspect **mynetwork or use id of network**

```
PS C:\Users\chout> docker network ls
NETWORK ID     NAME        DRIVER    SCOPE
13669d794e73   bridge      bridge    local
8978f2fe82df   host        host      local
b8a278cfdf23   mynetwork   bridge    local
f1cc085c9d13   none        null      local
```

Terminal

```
"ConfigOnly": false,
"Containers": {
    "6b2400e5fcd97ab5ac00dfbd7c6bbb656f74c656454e2284ba258c857dc73697": {
        "Name": "container1",
        "EndpointID": "275e47e983fa3865badb61f08f20c175f2cb38f41a3894e620e902ecbda14c6c",
        "MacAddress": "02:42:ac:12:00:02",
        "IPv4Address": "172.18.0.2/16",
        "IPv6Address": ""
    },
    "6f7b3c362bb4cc0e71b34cb4eb24eb9321536f92420a742376ebdfc905a31bfe": {
        "Name": "container2",
        "EndpointID": "a39e6363e906b03bba51f19d85e7e8c234796a59360412264c5374887950bcaf",
        "MacAddress": "02:42:ac:12:00:03",
```

4. Execute an interactive bash shell inside a running Docker container named container1

docker exec -it container1 bash

5. Install ping Utility:

apt update && apt install -y iputils-ping

6. Ping the Container: You can now ping one container from the other. For example, to ping container2 from container1:

ping container2

Example Output

PING container2 (172.18.0.3): 56 data bytes

64 bytes from 172.18.0.3: seq=0 ttl=64

time=0.123 ms 64 bytes from 172.18.0.3:

seq=1 ttl=64 time=0.124 ms

In-Lab Program2: Create a Dockerfile using multi-stage builds to efficiently build and deploy a Node.js application.

A **Dockerfile** is a set of instructions used to build a Docker image automatically.

A **multi-stage build** is an advanced Dockerfile technique that uses multiple build stages to separate build and runtime environments.

This helps reduce image size, improve security, and optimize application deployment.

Your project folder contains:

```
/app
 ├── server.js
 ├── package.json
 ├── package-lock.json (optional, generated automatically)
 └── Dockerfile
```

Dockerfile

```
# Stage 1: Build & dependencies
FROM node:18 AS builder
WORKDIR /app
# Copy only dependency files first (better caching)
COPY package*.json ./
RUN npm install --production
```

```
# Copy application source code
COPY  server.js .
# Stage 2: Runtime (small image)
FROM  node:18-slim
WORKDIR  /app
# Copy only required files from builder
COPY  --from=builder  /app/node_modules  ./node_modules
COPY  --from=builder  /app/server.js .
CMD ["node", "server.js"]
```

server.js file

```
const express =require("express");
const app = express();
// Home route
app.get("/", (req, res) => {
  res.send("Hello! Node.js app is running using Docker Multi-Stage Build
});
// Server port
const PORT = 3000;
// Start server
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

package.json file

package.json file is essential for a Node.js application because it lists the dependencies required to run the application.

```
{
  "name": "docker-multistage-demo",
  "version": "1.0.0",
  "description": "Simple Node.js app for Docker multi-stage build
  demo", "main": "server.js",
  "scripts": {
    "start": "node server.js"
```

```
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

Run Commands

1. **Build the docker image named node-demo**

```
docker build -t node-demo .
```

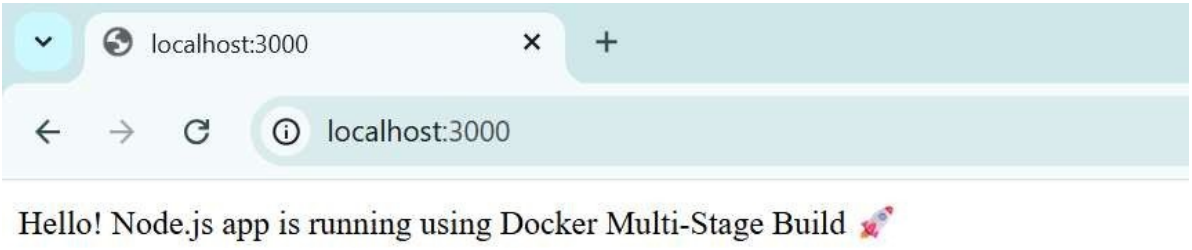
2. **Create the container**

```
docker run -d -p 3000:3000 node-demo
```

Open Browser type in url

http://localhost:3000

output:



Post Lab Tasks

1. Working with Docker Networking and Volumes

Networking connects containers to each other and to the outside world. Volumes store data so it is not lost when containers stop or restart.

Why do we need Docker Volumes?

- Containers are **temporary**
- Data inside containers is **lost on removal**
- Volumes ensure **data persistence**
- Used for databases, logs, uploads

A. Docker Networking

Step 1: List Available Docker Networks

```
docker network ls
```

Step 2: Create a Custom Bridge Network

```
docker network create app_network
```

Step 3: Run Containers on the Same Network

```
docker run -d --name web1 --network app_network nginx
docker run -d --name web2 --network app_network nginx
```

Step 4: Inspect the Network

```
docker network inspect app_network
```

Here, Containers connected to the same bridge network can communicate using **container names**.

B. Docker Volumes

Step 5: Create a Docker Volume

```
docker volume create app_volume
```

Step 6: Run Container with Volume

```
docker run -d --name volume_test -v app_volume:/data busybox sh -c "echo Docker Volume Data > /data/info.txt && sleep 3600"
```

Step 7: Verify Volume Data

```
docker exec volume_test cat /data/info.txt
```

Data stored in Docker volumes persists even after container restarts.

2. Create a Dockerfile using multi-stage builds to efficiently build and deploy a Python application

Step 1: Create Python Application

app.py

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Python App Running Using Multi-Stage Docker Build"

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

requirements.txt

```
flask
```

Step 2: Create Multi-Stage Dockerfile

Dockerfile

```
# -------- Stage 1: Build Stage --------
FROM python:3.11 AS builder
WORKDIR /app
COPY requirements.txt .
RUN pip install --user -r requirements.txt
COPY app.py .

# -------- Stage 2: Runtime Stage --------
FROM python:3.11-slim
WORKDIR /app
COPY --from=builder /root/.local /root/.local
COPY --from=builder /app /app
ENV PATH=/root/.local/bin:$PATH
EXPOSE 5000
CMD ["python", "app.py"]
```

Step 3: Build Docker Image

docker build -t python-multistage .

Step 4: Run Python Container

docker run -d -p 5000:5000 --name python-app python-multistage

Step 5: Verify Output

Open browser:

http://localhost:5000

Expected Output: