National Textile University

Department of Computer Science

Subject: Operating System

Submitted to: Sir Nasir

Submitted by: Amina

Reg. number: 23-NTU-CS-1136

Lab no.: Lab 12

Semester:5th

Docker Lab 02

Data Persistence, Custom Images, Networking & Docker Compose

Quick Day 1 Recap What

We Learned Yesterday

Core Concepts:

- Docker solves the “It Works on My Machine” problem
- Images = blueprints, Containers = running instances
- Containers are lightweight and fast compared to VMs
- Images contain OS userspace but share host kernel

Essential Comm

```
docker run          # Create and start containers
docker ps           # List containers
docker logs         # View container output
docker exec         # Run commands in containers
```

ands:

Quick Check:

```
# Verify Docker is running
docker --version
docker ps
```

If you see any errors, make sure Docker Desktop is running!

Part 1: Data Persistence with Volumes

Why This Matters:
Before building complex applications, you need to understand how to keep your data safe. Imagine building a blog and losing all posts when you restart!

The Problem: Container Data is Ephemeral

What Happens to Data in Containers?

Let’s see what happens to data when a container stops:

```
# Start a container and create some data
docker run -it --name test-container ubuntu bash

# Inside the container, create a file
echo "Important data!" > /data.txt

cat /data.txt  # Shows: Important data!

exit

docker rm test-container # Remove the container

# Start a new container
docker run -it --name test-container ubuntu bash

cat /data.txt  # Error: No such file!# Data is GONE!
```

```
root@a0924b8ab459:/# cd/home/ubuntu/
bash: cd/home/ubuntu/: No such file or directory
root@a0924b8ab459:/# cd home/ubuntu/
root@a0924b8ab459:/home/ubuntu# ls
Volume
root@a0924b8ab459:/home/ubuntu# cd Volume/
root@a0924b8ab459:/home/ubuntu/Volume# echo "my word" >file.txt
root@a0924b8ab459:/home/ubuntu/Volume# cat file.txt
my word
root@a0924b8ab459:/home/ubuntu/Volume#
```

The problem:

- Container filesystems are temporary
- Data disappears when container is removed
- Can’t share data between containers
- Not suitable for databases, uploads, logs, etc.

The Solution: Docker Volumes

Volumes are Docker-managed storage that persists outside containers.

Think of volumes as:

- External USB drives that you can plug into containers
- Persistent storage that survives container deletion
- Shareable between multiple containers

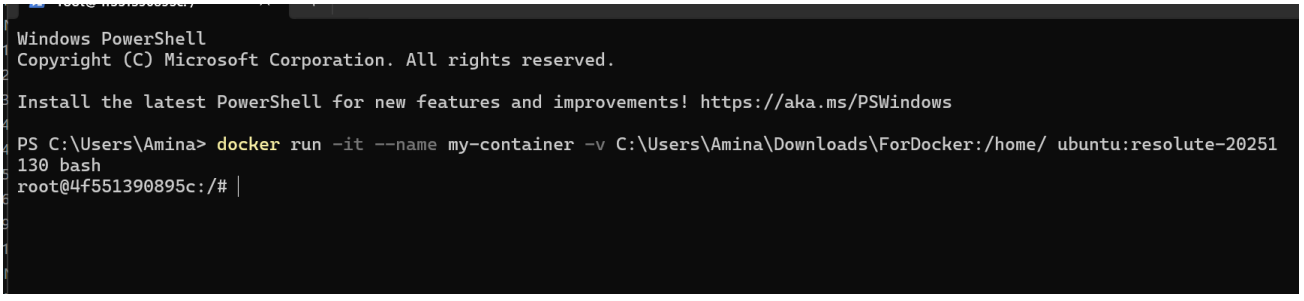
Types of Docker Storage

1. Named Volumes (Recommended for Production)

What: Docker-managed storage in a special location

Where: Docker manages the location (usually `/var/lib/docker/volumes/`) **When:** Databases, persistent app data, production use

```
# Create a named volume
docker volume create my-data

# List volumes
docker volume ls

# Inspect volume details
docker volume inspect my-data

# Use volume in container
docker run -it -v my-data:/home/ubuntu/my-project ubuntu

# Remove volume (only when not in use)
docker volume rm my-data

# Remove all unused volumes
docker volume prune
```



2. Bind Mounts (For Development)

What: Mount a host directory directly into container

Where: You specify exact host path

When: Development (live code changes), config files

```
# Mount with absolute path
docker run -it -v /home/user/myapp:/home/ubuntu ubuntu
```

Volume Management Best Practices

When to Use Which Type?

Storage Type	Use Case	Example
Named Volume	Production data, databases	PostgreSQL, MongoDB data
Bind Mount	Development, config files	Live code editing, nginx config

---

Part 2: Building Custom Images with Dockerfile

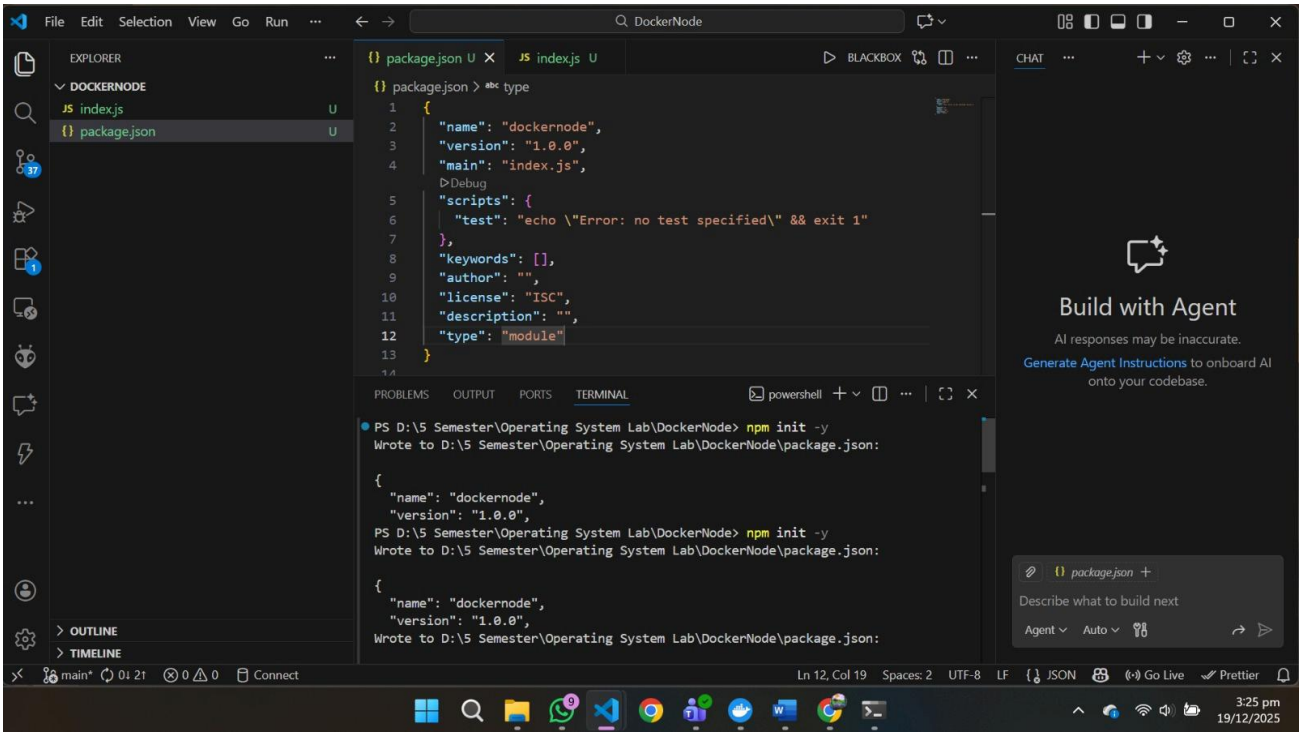
Why This Matters:

Now that you know how to use existing images, let's create your own! This is where Docker becomes truly powerful - you can package your applications with all their dependencies.

Why Do We Need Custom Images?

So far, we've used **pre-built images** from Docker Hub:

```
docker run nginx
docker run node:18
docker run postgres
```



These are great, but they're generic. What if you want to:

- Add your own application code?
- Install specific packages?
- Configure custom settings?
- Create a reproducible environment for your project?

Answer: Build your own images with Dockerfile!

Your First Dockerfile

Let's create a simple custom image step by step!

Step 1: Create a Project Directory

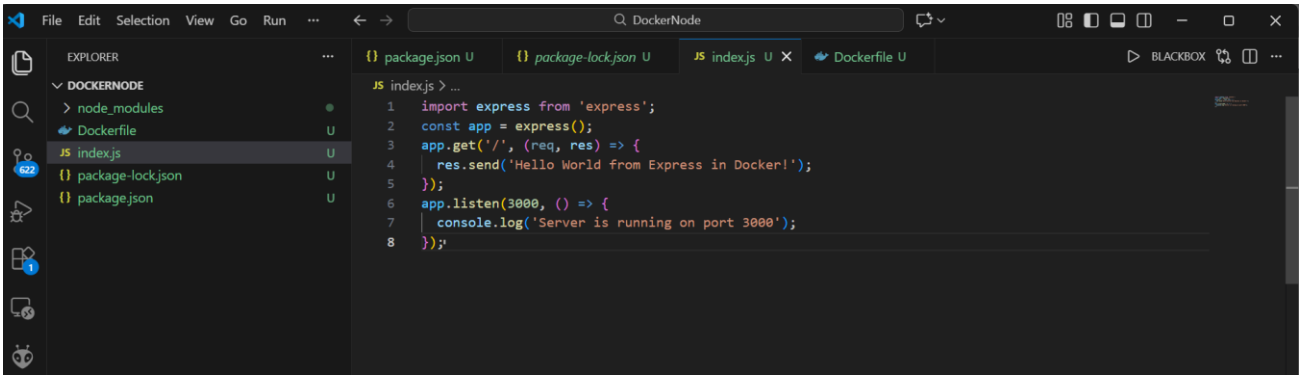
```
cd ~
mkdir my-first-docker-app
cd my-first-docker-app
```

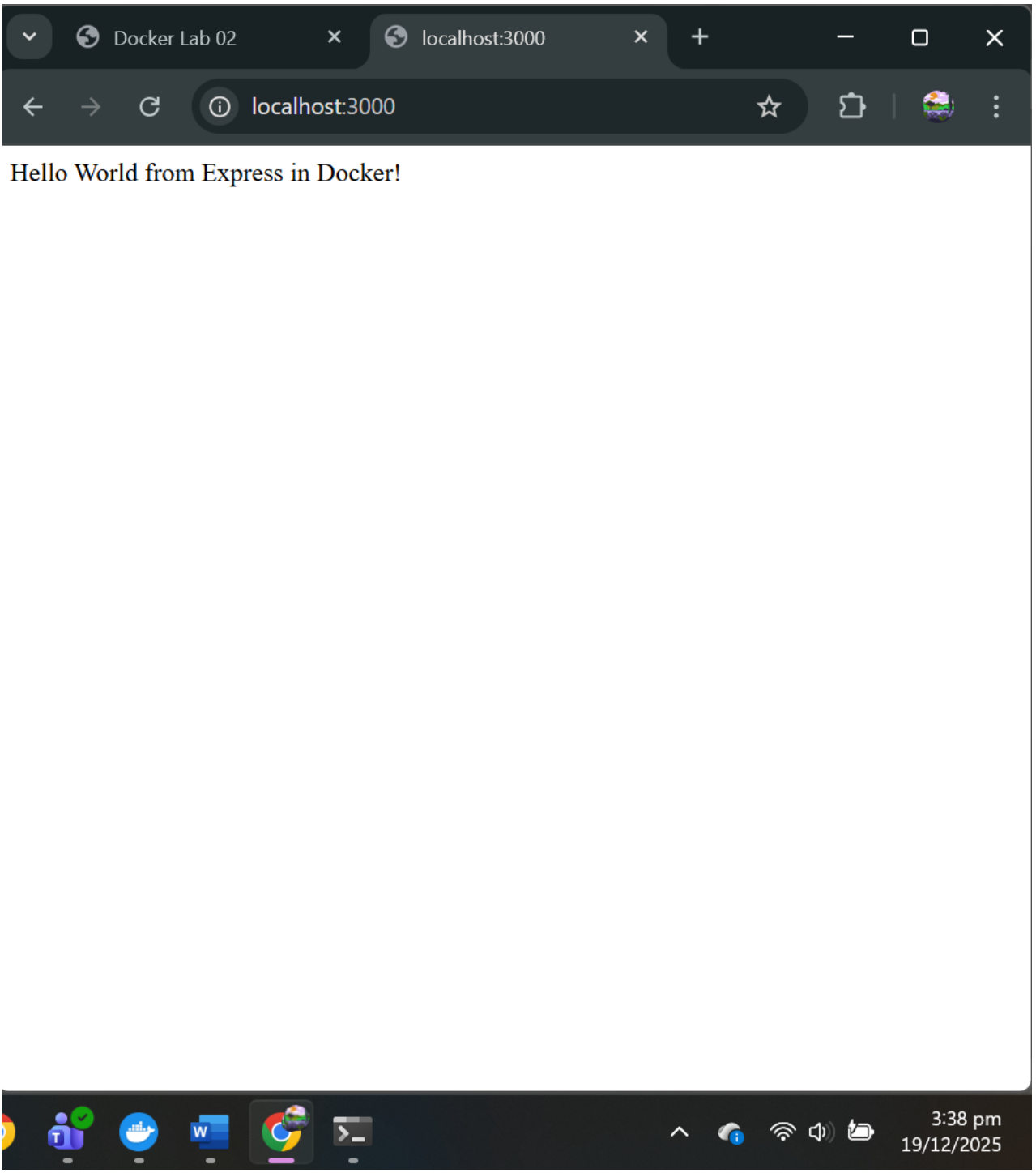
Step 2: Create a Simple Node.js Server

First of all Install `express` in your working working directory using `node` .

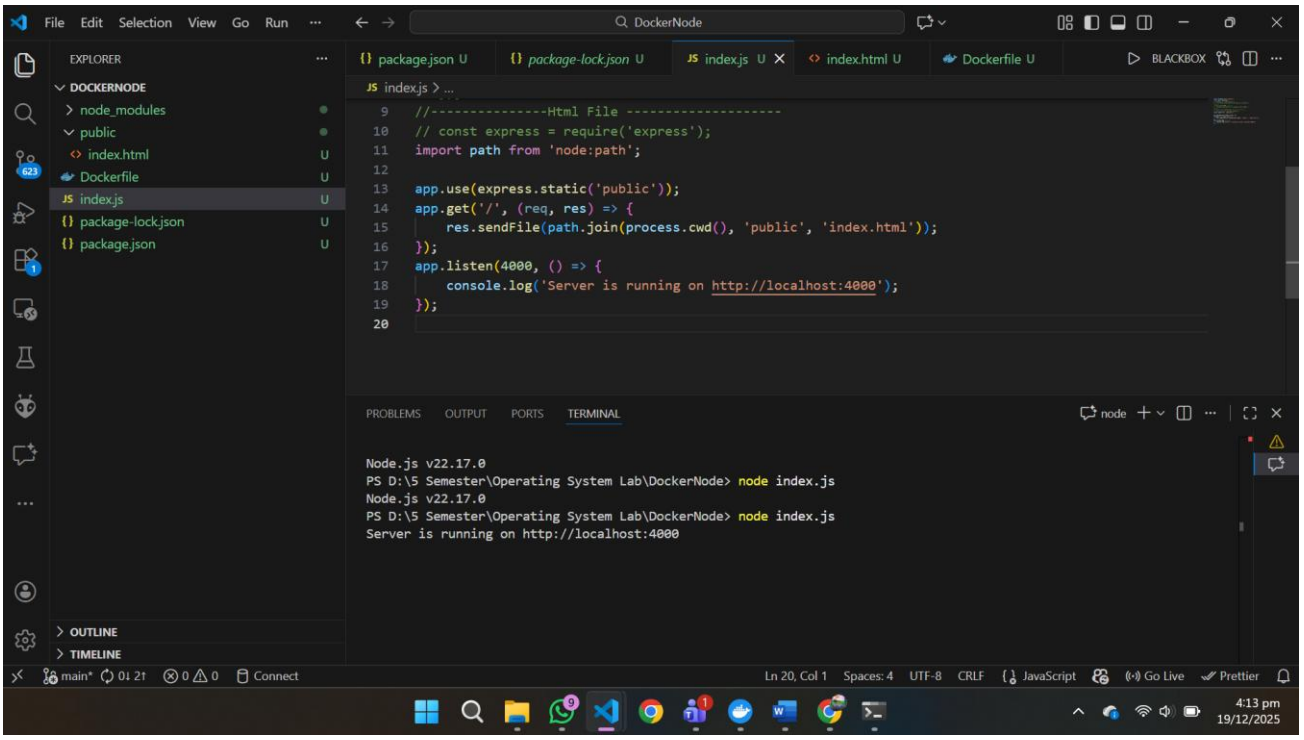
```
import express from 'express'
const app = express()

app.get('/', (request, response) => {
    response.send('Hello from Node')
})
```

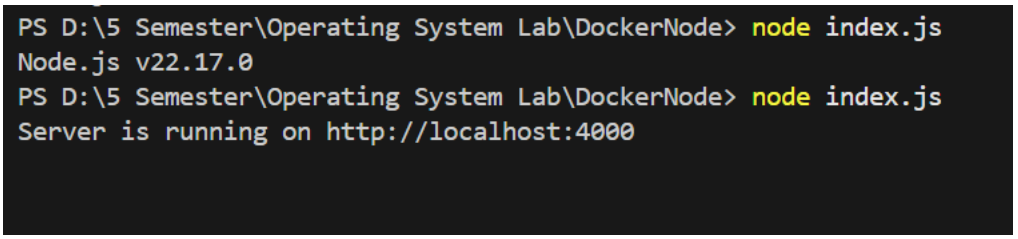
```
app.listen(3000)
```

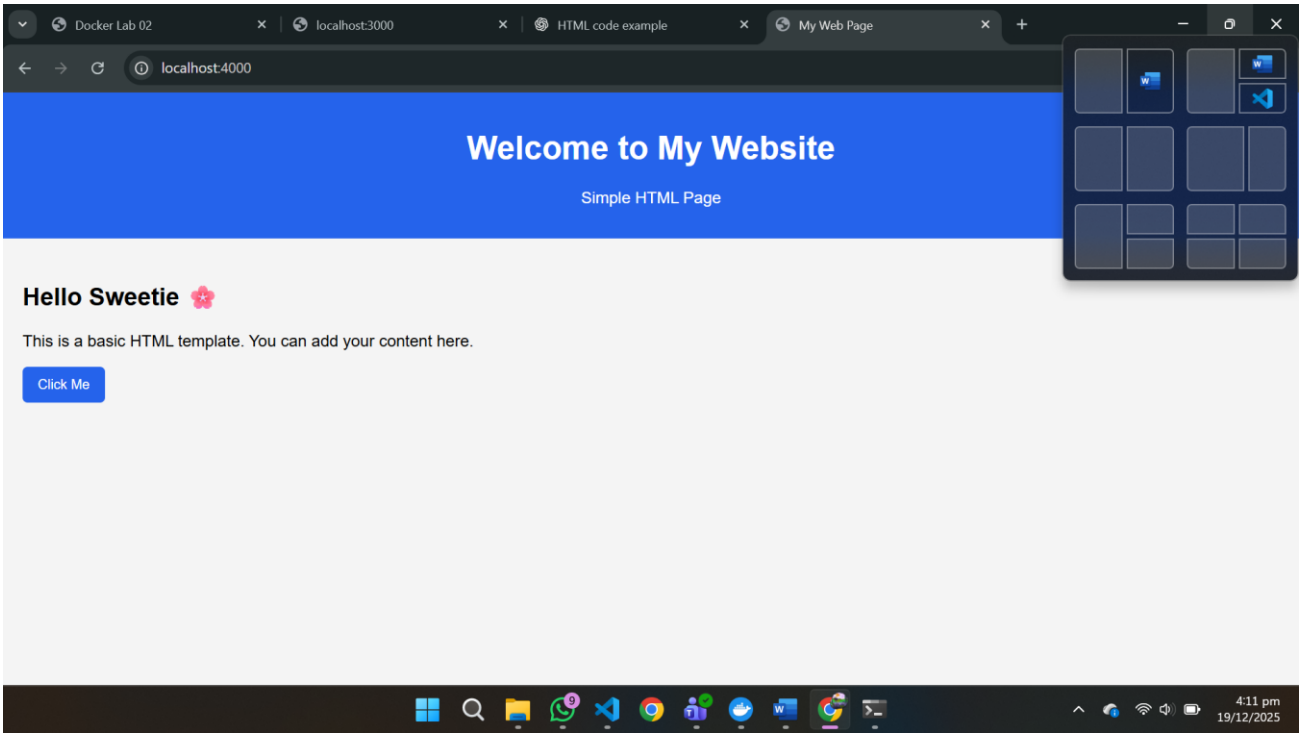



Docker Lab 02
localhost:3000
localhost:3000
Hello World from Express in Docker!
3:38 pm
19/12/2025

Class Task


File Edit Selection View Go Run
DockerNode
EXPLORER
DOCKERNODE
node_modules
public
index.html
Dockerfile
index.js
package-lock.json
package.json
package.json U
package-lock.json U
index.js U
index.html U
Dockerfile U
BLACKBOX
index.js > ...
9 //---------------Html File --------------------
10 // const express = require('express');
11 import path from 'node:path';
12
13 app.use(express.static('public'));
14 app.get('/', (req, res) => {
15 res.sendFile(path.join(process.cwd(), 'public', 'index.html'));
16 });
17 app.listen(4000, () => {
18 console.log('Server is running on http://localhost:4000');
19 });
20
PROBLEMS OUTPUT PORTS TERMINAL
node
Node.js v22.17.0
PS D:\5 Semester\Operating System Lab\DockerNode> node index.js
Node.js v22.17.0
PS D:\5 Semester\Operating System Lab\DockerNode> node index.js
Server is running on http://localhost:4000
OUTLINE
TIMELINE
main* 0↓ 2↑ 0 0 Connect
Ln 20, Col 1 Spaces: 4 UTF-8 CRLF JavaScript Go Live Prettier
4:13 pm
19/12/2025


PS D:\5 Semester\Operating System Lab\DockerNode> node index.js
Node.js v22.17.0
PS D:\5 Semester\Operating System Lab\DockerNode> node index.js
Server is running on http://localhost:4000

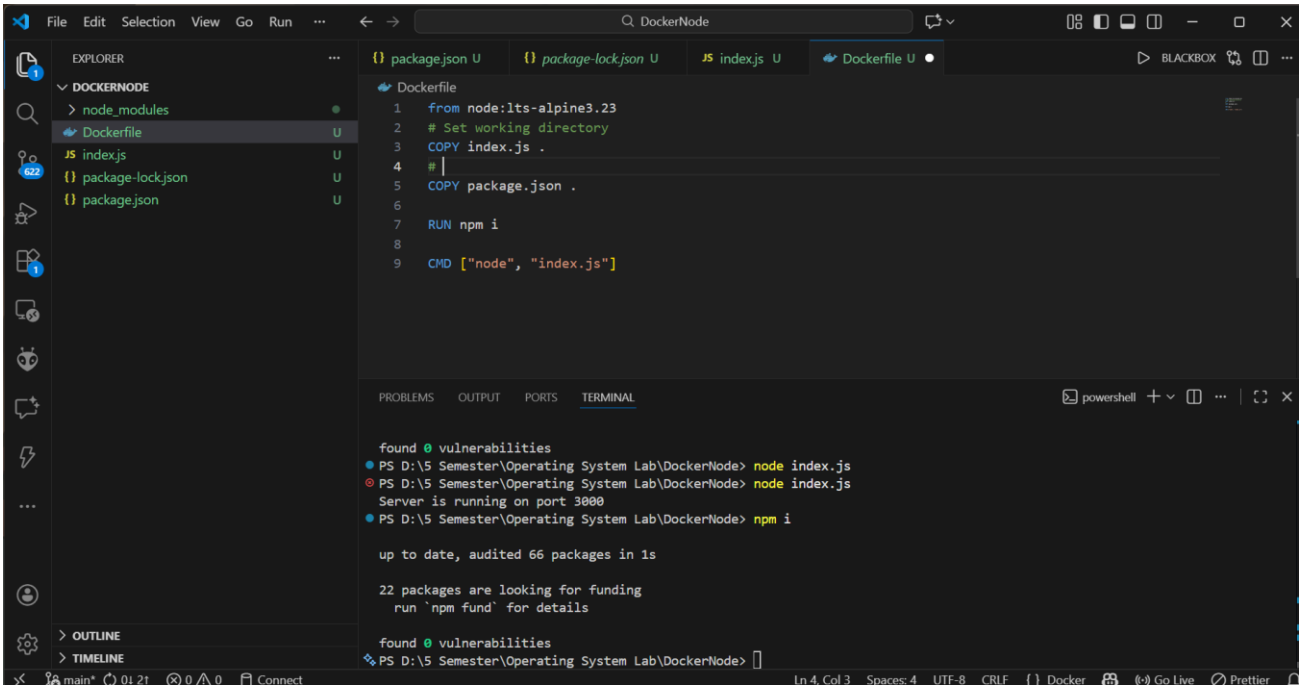

Docker Lab 02
localhost:3000
HTML code example
My Web Page
localhost:4000
Welcome to My Website
Simple HTML Page
Hello Sweetie
This is a basic HTML template. You can add your content here.
Click Me
4:11 pm
19/12/2025

Step 3: Create Your First Dockerfile

Create file named `Dockerfile` . Name must be same.

```
# Use Node.js as the base image
FROM node:lts-alpine3.23

# Set working directory inside container
WORKDIR /app

# Copy our application file into the container
COPY index.js .

# Copy package.json
COPY package.json .

# Install dependencies
npm install

# Define the command to run when container starts
CMD ["node", "index.js"]
```

Vs may Dockerfile Commands:



Run server on Docker

```
PS C:\Users\Amina> docker run -d -p 3000:3000 my-server
5416125fc8ce3d5800ba79f5287a169953abfdd9553318aecca6e22b0041b31c
PS C:\Users\Amina>
```

Understanding Each Instruction

```
FROM node:lts-alpine3.23
```

- **FROM:** The base image to start from
- Every Dockerfile must start with FROM
- `alpine` variant is smallest (based on Alpine Linux)

```
WORKDIR /app
```

- **WORKDIR:** Sets the working directory inside the container
- Creates the directory if it doesn't exist
- All subsequent commands run from this directory

```
COPY index.js .
```

- **COPY:** Copies files from your computer into the image
- `index.js` = file on your computer
- `.` = current directory in container (`/app`)

```
CMD ["node", "app.js"]
```

- **CMD:** The default command to run when container starts
- Only one CMD per Dockerfile (last one wins)

Step 4: Build Your Image

```
docker build -t my-first-app .
```

Understanding the command:

- `docker build` = build an image
- `-t my-first-app` = tag (name) the image
- `.` = look for Dockerfile in current directory

What you'll see:
Docker processes each instruction and creates layers. Watch the output!

Step 5: Run Your Custom Image

```
docker run my-first-app
```

Congratulations! You're now a Docker image builder!

Essential Dockerfile Instructions

Let’s learn the most important Dockerfile instructions:

1. FROM - Choose Your Base Image

```
# Official Node.js image
FROM node:18
# Node.js with Alpine (smallest)
FROM node:18-alpine
# Official Python image
FROM python:3.9
# Ubuntu
FROM ubuntu:22.04
```

Best Practice: Use official images when possible (verified, maintained)

2. WORKDIR - Set Working Directory

```
FROM node:18-alpine
WORKDIR /app
# Now all commands run from /app
```

3. COPY - Copy Files Into Image

```
# Copy single file
COPY app.js /app/
# Copy entire directory
COPY ./src /app/src
# Copy multiple files
COPY package.json package-lock.json /app/
```

4. RUN - Execute Commands During Build

```
# Install packages
RUN apk add --no-cache curl
# Install Node.js dependencies
RUN npm install
```

5. ENV - Environment Variables

```
# Set environment variables
ENV NODE_ENV=production
ENV PORT=3000
ENV APP_VERSION=1.0.0
# Available at runtime in your application
```

6. EXPOSE - Document Port Usage

```
# Document that container listens on port 80
EXPOSE 80
# Multiple ports
EXPOSE 80 443
```

Important: EXPOSE is documentation only! Still need `-p` flag when running.

7. CMD - Default Command

```
# Exec form (recommended)
CMD ["node", "app.js"]
# Can be overridden when running container
docker run my-app node other_app.js
```

Understanding Layer Caching

This is important for efficient Docker builds!

The Right Way:

```
# ✅ GOOD: Dependencies cached separately
COPY package*.json ./
RUN npm install
COPY app.js .
```

The Wrong Way:

```
# ❌ BAD: Everything rebuilds when code changes
COPY . .
RUN npm install
```

See caching in action:

```
# Change app.js
echo "// Updated" >> app.js
# Rebuild - watch for "CACHED"
docker build -t express-app .
```

Dependencies stay cached! Only `app.js` is recopied. Much faster!

Practice Exercise: Build Your Own Image

Challenge: Serve static HTML files from our previously build node server!

Part 3: Docker Networking

Why It Matters:

After learning how to build images, the next step is making containers talk to each other. Real applications often have multiple services like web servers, databases, and caches that need to communicate.

Why Containers Need Networks

Example scenario:

- Frontend (React) talks to Backend (Express)
- Backend talks to Database (PostgreSQL) and Cache (Redis)

Problem: By default, containers are isolated and cannot see each other.

Solution: Docker Networks!

Containers Are Isolated by Default

```
docker run -d --name app1 alpine sleep 3600
docker run -d --name app2 alpine sleep 3600
docker exec app1 ping app2
#  >Fails! Containers cannot communicate by default
```

Custom Networks Solve This

```
# Start two Ubuntu containers on a custom network
docker network create my-network

docker run -d --name app1 --network my-network alpine sleep 3600
docker run -d --name app2 --network my-network alpine sleep 3600

# Ping from one to the other
docker exec app1 ping -c 2 app2
```

Cleanup:

```
docker rm -f app1 app2
docker network rm my-network
```

Types of Docker Networks

1. **Bridge Network (Default)**
   - Private internal network for containers
   - Use for most applications
2. **Host Network (Advanced)**
   - Container shares host's network
   - Use for special cases needing maximum performance
   - Only works on Linux

```
docker run -d --network host nginx
```

Key Takeaways

- Containers on the same network can communicate
- Container names work as hostnames (DNS)
- No port mapping is needed for internal communication
- Networks provide isolation: containers only talk if on the same network

Network Management Commands

```
# List networks
docker network ls

# Create network
docker network create my-network

# Inspect network (see connected containers)
docker network inspect my-network

# Connect running container to network
docker network connect my-network my-container

# Disconnect container from network
docker network disconnect my-network my-container

# Remove network
docker network rm my-network

# Remove all unused networks
docker network prune
```

Congratulations!

You've completed Day 2 and learned:

- ☑ **Data Persistence** - Using volumes to keep data safe
- ☑ **Custom Images** - Building your own images with Dockerfile
- ☑ **Docker Networking** - Connecting containers together
- ☑ **Docker Compose** - Orchestrating multi-container applications
- ☑ **Best Practices** - Writing efficient and secure Dockerfiles

Your Docker Journey

What you've accomplished in 2 Labs:

- Lab 1: Running containers, managing images, basic
- operations Lab 2: Building images, connecting services,

orchestration You now have the core skills to:

- Containerize your own applications
- Set up development environments quickly
- Share reproducible environments with your team
- Build multi-container applications
