Lab 2: Dockerizing Your Application

I. Learning Objectives

- Write a complete, multi-stage `Dockerfile` to containerize your web application.
- Build a Docker image and test it locally.
- Write a `docker-compose.yml` file to define and run your full multi-service application (app + database).
- Manage your application's lifecycle and data persistence using Docker Compose.

II. Prerequisites

- **Docker Desktop:** Must be installed and running.
- **Group Project:** Your group's repository (e.g., `devops-group-1`) must be cloned to your local machine (from Lab 1).
- **VS Code:** Open your group's project folder in VS Code.
- **New Branch:** All work for this lab MUST be done on a new feature branch.

```
# Make sure you're on main and have the latest changes
$ git checkout main $ git pull origin main

# Create a new branch for this lab's work
$ git checkout -b feature/dockerize-app
```

III. Part A: The `Dockerfile` (Containerizing the App)

Our first goal is to create a "recipe" (a `Dockerfile`) that can build a portable image of our web application.

Step A.1: Create `.dockerignore` (Best Practice)

Before writing the `Dockerfile`, we must create a `.dockerignore` file. This file prevents Docker from copying unnecessary or sensitive files into our image, making it smaller and more secure.

1. In the **root** of your project, create a new file named `.dockerignore`.
2. Add the following content. This tells Docker to ignore these files during the build:

```
# Ignore the local dependencies, we will install them INSIDE the container
node_modules

# Ignore Git files
.git .gitignore

# Ignore Docker files
Dockerfile
.dockerignore
docker-compose.yml

# Ignore local environment files
.env
```

Step A.2: Create the `Dockerfile`

In the **root** of your project, create a new file named `Dockerfile` (no extension).

Step A.3: Define Base, Workdir, and Install Dependencies

We will use a multi-stage build, as discussed in the lecture, to create an optimized final image. Add the first "build" stage to your `Dockerfile`:

```
# --- Stage 1: The "builder" stage ---

# Use a full Node.js image to build our app

FROM node:18-alpine AS builder
```

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


# Copy package.json and package-lock.json (or yarn.lock)
COPY package*.json ./


# Install all dependencies (including devDependencies)
RUN npm install


# Copy the rest of the application source code
COPY . .


# (If your project has a build step, e.g., TypeScript, add it here)
# RUN npm run build
```

> **Why `COPY package*.json` first?** This is Docker's layer caching. As long as `package.json` doesn't change, Docker will re-use the `RUN npm install` layer, making future builds much faster.

Step A.4: Define the Final "Production" Stage

Now, add the second stage. This stage will create the final, lightweight image that we actually run.

```
# --- Stage 2: The "production" stage ---
# Start from a fresh, lightweight base image
FROM node:18-alpine
WORKDIR /app
```

```
# Copy the package.json and install *only* production dependencies
COPY package*.json ./
RUN npm install --production

# Copy the built app code from the "builder" stage
COPY --from=builder /app/src ./src
# (If you had a build step, you would copy the 'dist' folder
instead)
# COPY --from=builder /app/dist ./dist
# Expose the port the app listens on (check your app's code, e.g.,
server.js)
EXPOSE 3000

# The command to run the application (check your package.json
'start' script)
CMD [ "npm", "start" ]
```

Save your `Dockerfile`.

Step A.5: Build and Test Your Image

Let's build the image to make sure the `Dockerfile` works. Give it a name (a **tag**).

```
# Build the image from the Dockerfile in the current directory (.)
# -t tags it as 'my-app:1.0' (replace 'my-app' with your project
name)
$ docker build -t my-app:1.0 .
```

If it builds successfully, test running it. (Note: It will likely crash because it can't find a database, but that's okay! We are just testing if the container **starts).**

```
# Run the container. -p 8080:3000 maps your PC's port 8080 to the
container's 3000
$ docker run -d --name test-app -p 8080:3000 my-app:1.0
```

```
# Check its logs. You might see "Error connecting to database".
This is fine.
$ docker logs test-app


# Stop and remove the test container
$ docker stop test-app
$ docker rm test-app
```

IV. Part B: The `docker-compose.yml` (Multi-Service)

Now we'll define our **full** application (app + database) using Docker Compose.

Step B.1: Create `docker-compose.yml`

In the **root** of your project, create a new file named `docker-compose.yml`.

Step B.2: Define the `app` Service

This service will be built from our `Dockerfile`.

```
version: '3.8'
services:
    app:
      build: . # Tells Compose to build the Dockerfile in this
directory
      ports: - "8080:3000" # Map host port 8080 to container
port 3000
```

Step B.3: Define the `db` Service

Our app needs a database. Let's add a PostgreSQL service. We'll use a pre-built image from Docker Hub.

```
# ... (version and services: are above) ...
app:
  # ... (app config) ...
db:
  image: postgres:14-alpine
  environment:
    - POSTGRES_USER=myuser
    - POSTGRES_PASSWORD=mypass
    - POSTGRES_DB=mydatabase
  ports:
    - "5432:5432" # (Optional) Lets you connect to the DB from
your PC
```

Step B.4: Connect the Services

How does the `app` find the `db`? Docker Compose creates a private network.

The hostname for the `db` service is simply `db`.

We pass this to our app using environment variables.

```
services:
  app:
    build: .
    ports: - "8080:3000"
    environment:
        - DB_HOST=db
        - DB_USER=myuser
        - DB_PASSWORD=mypass
        - DB_NAME=mydatabase
    depends_on:
        - db # Tells 'app' to wait for 'db' to start first

  db:
    # ... (db config) ...
```

Important: Your application code (e.g., `server.js` or `db.js`) MUST be configured to read these environment variables (`process.env.DB_HOST`, `process.env.DB_USER`, etc.) to connect to the database.

Step B.5: Add Data Persistence (Volumes)

If we stop the `db` container, all our data is lost. We must use a **volume** to store the data on the host machine.

```
# ... (at the end of the db service) ...
  db:
    image: postgres:14-alpine
    environment:
   # ... (env vars) ...
    volumes:
        - db-data:/var/lib/postgresql/data # Persist data
here
# Add this at the VERY end of the file (top-level)
volumes:
    db-data:
```

Save your `docker-compose.yml` file. It is now complete.

V. Part C: Run, Verify, and Commit

Step C.1: Run the Full Application

With your `docker-compose.yml` file, you can now run everything with one command.

```
# Build the 'app' image and start all services
(-d = detached/background)
$ docker-compose up -d --build
```

Step C.2: Check Logs

Watch the logs to see both containers start up. You should see your app log "Connected to database".

```
# Show logs from all services and "follow" them in real-time
$ docker-compose logs -f
```

(Press `Ctrl+C` to stop following the logs).

Step C.3: Verify in Browser

Open your browser and go to `http://localhost:8080`. Your app should load. **Test it!** Try to register a user, log in, or add data. This will confirm your app is successfully communicating with the `db` container.

Step C.4: Verify Persistence

Let's test our volume. Stop and remove the containers:

```
$ docker-compose down
```

Now, start them again (no `--build` needed this time):

```
$ docker-compose up -d
```

Go back to `http://localhost:8080` and try to log in with the user you created. Your data should still be there! This proves your `db-data` volume is working.

Step C.5: Commit and Push

Your application is now fully containerized! It's time to commit your work.

```
# 1. Check status. You should see 3 new files:
# Dockerfile, .dockerignore, docker-compose.yml
$ git status

# 2. Add all new files
$ git add Dockerfile .dockerignore docker-compose.yml

# 3. Commit your work
$ git commit -m "feat: Add Docker and Docker Compose
files"

# 4. Push your branch
$ git push origin feature/dockerize-app
```

Step C.6: Open Pull Request

Go to your group's GitHub repository. Open a Pull Request for your `feature/dockerize-app` branch. Add your teammates as reviewers.

Your team leader will merge this PR after it's approved. Your project is now ready for the next stage: CI/CD.