Docker Containerization for Node.js API

1. Aim

The aim of this project is to successfully **containerize** a Node.js application (specifically, the Account Transfer System) using Docker. This ensures the application is packaged with all its dependencies and can run consistently and reliably on any platform, eliminating "works on my machine" issues.

2. Objectives

Upon completion of this project, the following objectives should be met:

1. **Dockerfile Creation:** Define a functional Dockerfile that specifies all necessary build steps for the Node.js application.
2. **Image Build:** Successfully execute the docker build command to create a portable Docker image.
3. **Application Isolation:** Understand and configure the file system, working directory, and environment inside the container.
4. **Port Mapping:** Correctly map the internal container port (3000) to an external host port for API access.
5. **Runtime Verification:** Run the application within the container using docker run and verify that the /transfer API endpoint is accessible and functional.

3. Theory

A. Docker and Containerization

Docker is a platform used to develop, ship, and run applications inside **containers**. A container is a lightweight, executable package of software that bundles everything needed to run an application: code, runtime, system tools, system libraries, and settings. This isolation ensures the application behaves the same regardless of where it is deployed.

B. Dockerfile

The **Dockerfile** is a text file that contains all the commands a user could call on the command line to assemble an image. Each command in the Dockerfile creates a new layer in the Docker image.

Key Dockerfile instructions used:

- **FROM**: Specifies the base image (e.g., node:20-alpine for a lightweight Node.js environment).
- **WORKDIR**: Sets the working directory inside the container for subsequent commands.
- **COPY**: Copies files from the local host system into the container's filesystem.

- **RUN**: Executes commands needed for the build process (e.g., installing dependencies via npm install).
- **EXPOSE**: Documents the port on which the application listens at runtime (e.g., 3000).
- **CMD**: Provides the command to execute when a container is started from the image.

4. Procedure

The following steps assume the existing Node.js application file is named transfer_server.js and that a package.json file defining the express dependency exists in the root directory.

1. **Create Dockerfile:** Create a file named Dockerfile in the root directory of the project and insert the code provided below.
2. **Ensure Dependencies:** Verify that package.json is present and correctly lists the express dependency.
3. **Build Docker Image:** Execute the build command, tagging the image for easy reference (e.g., banking-api).
   ```
   docker build -t banking-api .
   ```
4. **Run Container:** Run the built image as a container, mapping the internal container port 3000 to the host's port 8080.
   ```
   docker run -d -p 8080:3000 --name banking-container banking-api
   ```
5. **Verify Access:** Test the API from the host machine using the mapped port 8080.
   ```
   curl -X POST http://localhost:8080/transfer -H "Content-Type: application/json" -d
   '{"sourceAccount": "123456", "destinationAccount": "987654", "amount": 10}'
   ```

5. Code

The complete Dockerfile is provided below. This file defines the steps necessary to create a lean and functional container image for the Node.js API.

6. Output

A. Docker Build Output

Successful execution of the build command (docker build -t banking-api .) will show a sequence of steps corresponding to the Dockerfile instructions:

```
[+] Building
...
Step 3/6 : COPY package*.json ./
...
Step 4/6 : RUN npm install
...
```

```
Step 6/6 : CMD ["node", "transfer_server.js"]
Successfully built [Image ID]
Successfully tagged banking-api:latest
```

B. Docker Run and Verification Output

1. **Run Command:** docker run -d -p 8080:3000 --name banking-container banking-api
   - Output: A long container ID (e.g., a1b2c3d4e5f6...), confirming the container is running in detached mode (-d).
2. **API Verification (cURL):**
   - Console Output: The server logs will show the transfer operation successful.
   - HTTP Response (from cURL): The client will receive the success message: {"message":"Transfer successful. Transaction committed.","newBalance":"$990.00"}

7. Learning Outcomes

1. **Immutable Infrastructure:** Understood the core concept of containerization, ensuring the application and its environment are bundled together, leading to reliable, identical execution everywhere.
2. **Docker Image Layering:** Learned how each Dockerfile instruction creates a distinct, cacheable layer, which optimizes build times and minimizes final image size.
3. **Host-Container Networking:** Mastered the technique of port mapping (-p 8080:3000), which is essential for bridging the network stack of the isolated container with the network of the host machine.