Docker

1. **Basic Dockerfile: Write a Dockerfile for a Node.js Express application that installs dependencies and runs the server on port 3000.**

CODE:

```
// 2.DOCKER\1.Basic_Dockerfile\index.js
const express = require('express');
const app = express();
const PORT = 3000;

app.get('/', (req, res) => {
  res.send('Hello, World!');
});

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

```
# 2.DOCKER\1.Basic_Dockerfile\Dockerfile
# Use the official Node.js image as the base image
FROM node:18-alpine

# Set the working directory inside the container
WORKDIR /app
# Copy package.json and package-lock.json to the working directory
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application files to the working directory
COPY . .

# Expose port 3000
EXPOSE 3000

# Command to start the server
# CMD ["node", "index.js"]
CMD ["npm", "start"]
```

EXPLANATION:

1) Base Image: FROM node:18 - This pulls the official Node.js image (version 18 in this case) from Docker Hub.
2) Working Directory: WORKDIR /usr/src/app - Sets the working directory in the container to /usr/src/app. This is where the app code will reside.
3) Dependency Installation:

- COPY package*.json ./ - Copies the package.json and package-lock.json to the working directory.
- RUN npm install - Runs npm install inside the container to install all the dependencies.

4) Copying App Files: COPY . . - Copies all the remaining application files to the container.
5) Expose Port: EXPOSE 3000 - Exposes port 3000, which is the default port where the Express server runs.
6) Start the Server: CMD ["npm", "start"] - Runs npm start, which starts the Express server. Ensure your package.json file has a start script defined, such as:

```
"scripts": {
    "start": "node index.js"
  },
```

Command for Terminal:

I. Build the Docker image:

docker build -t viveksingh/my-node-app:0.0.1 .

II. Run the container:

docker run -p 3000:3000 viveksingh/my-node-app:0.0.1

This will run your Node.js Express app inside a Docker container, and it will be accessible on http://localhost:3000.

2. **Docker Compose: Using Docker Compose, create a configuration file that sets up a multi-container application with a Node.js server and a MongoDB database.**

CODE:

```
# 2.DOCKER\2.Docker_Compose\Dockerfile
FROM node:18
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

```
# 2.DOCKER\2.Docker_Compose\docker-compose.yml
version: '3'

services:
  # Node.js Application
  app:
    build: .
    container_name: node_app
    restart: always
    ports:
      - "3000:3000"
    depends_on:
      - mongo
    environment:
      MONGO_URL: mongodb://127.0.0.1:27017/myDB
    volumes:
      - .:/usr/src/app
  # MongoDB Database
  mongo:
    image: mongo:latest
    container_name: mongo_db
    restart: always
    ports:
      - "27017:27017"
    volumes:
      - mongo-data:/data/db
# Define named volume for persistent MongoDB data
volumes:
  mongo-data:
```

EXPLANATION:

1) **Version: '3' specifies the version of Docker Compose.**
2) **Services:**

   app: The Node.js application.

   - build: The Dockerfile for the Node.js app is in the current directory (.).

- container_name: Names the container node_app.
- restart: Ensures the container restarts automatically in case of failure.
- ports: Maps port 3000 of the container to port 3000 of the host.
- depends_on: Ensures that the MongoDB service starts before the Node.js app.
- environment: Sets the environment variable MONGO_URL for the app, pointing to the MongoDB container (mongo).
- volumes: Mounts the current directory into /usr/src/app in the container for code sharing.

mongo: The MongoDB service.

- image: Pulls the latest MongoDB image from Docker Hub.
- container_name: Names the MongoDB container mongo_db.
- ports: Maps MongoDB's default port (27017) to the host machine.
- volumes: Persists MongoDB data to a named volume (mongo-data) for durability.

3) Volumes:

mongo-data: A named volume used to store MongoDB data persistently, ensuring that data is not lost when the container restarts.

Command for Terminal:

Run the multi-container application:

```
docker-compose up --build
```

This command will start both the Node.js application and MongoDB, linking them together. The Node.js app can access the MongoDB instance using the MONGO_URL environment variable.

3. **Docker Networking: Modify the Docker Compose configuration to ensure that the Node.js application can communicate with the MongoDB database via Docker networking.**

CODE:

```
# 2.DOCKER\3.Docker_Networking\docker-compose.yml
version: '3'

services:
  # Node.js Application
  app:
    build: .
    container_name: node_app
    restart: always
    ports:
      - "3000:3000"
    depends_on:
      - mongo
    environment:
      MONGO_URL: mongodb://127.0.0.1:27017/myDB  # Mongo service as
hostname
    volumes:
      - .:/usr/src/app
    networks:
      - app-network
  # MongoDB Database
  mongo:
    image: mongo:latest
    container_name: mongo_db
    restart: always
    ports:
      - "27017:27017"
    volumes:
      - mongo-data:/data/db
    networks:
      - app-network
# Define named volume for MongoDB data persistence
volumes:
  mongo-data:
# Create a custom network for both services
networks:
  app-network:
    driver: bridge
```

EXPLANATION:

1) **Networking:**

- Both the app and mongo services are explicitly placed on a custom network called app-network. This ensures the Node.js application and MongoDB can

communicate with each other using Docker's internal DNS, allowing the app to resolve mongo as the MongoDB hostname.

- Driver: The bridge driver is used, which is the default network driver for Docker containers. This allows containers to communicate over a private network.

2) **MONGO_URL Environment Variable:**

The environment variable MONGO_URL is set to mongodb://127.0.0.1:27017/myDB. Here, mongo is the service name for the MongoDB container, acting as the hostname that the Node.js application uses to connect to MongoDB.

Command for Terminal:

I. **Ensure Docker Compose is running:**

docker-compose up --build

II. **Application Access:**

The Node.js app can use the MONGO_URL in its configuration to access MongoDB, and mongo will be resolved to the correct internal IP by Docker's DNS.

With this configuration, the Node.js application can securely and reliably communicate with the MongoDB database using Docker's internal networking features.

4. Containerization: Explain the advantages of using Docker for deploying a MERN Stack application and provide an example of a real-world use case where Docker enhances development workflows.

Advantages of Using Docker for Deploying a MERN Stack Application:

- **Consistency Across Environments:** Docker ensures that the entire MERN (MongoDB, Express, React, Node.js) application runs the same way across development, staging, and production environments. This eliminates the "it works on my machine" problem since the app is containerized with all dependencies included.
- **Simplified Dependency Management:** A Docker container contains everything needed to run an application (OS, libraries, environment variables, etc.), so there's no need to worry about conflicts between different versions of Node.js, MongoDB, or other dependencies in the stack.
- **Scalability and Resource Isolation:** Docker enables you to easily scale individual services (MongoDB, Express, Node.js, React) as separate containers. You can allocate specific resources (CPU, memory) to each service, ensuring better resource isolation and efficient scaling.
- **Efficient CI/CD Integration**: Docker can easily integrate with continuous integration and deployment pipelines (CI/CD). With Docker, you can build, test, and deploy containers automatically, speeding up deployment cycles and ensuring consistent builds.
- **Portability:** Since Docker containers encapsulate everything, you can run them on any platform that supports Docker, be it local machines, cloud servers (AWS, Google Cloud, Azure), or hybrid environments, without worrying about compatibility.
- **Fast and Lightweight:** Containers are more lightweight compared to traditional virtual machines (VMs) because they share the host machine's operating system kernel. This makes spinning up containers faster, reducing boot times and improving developer productivity.
- **Microservices Architecture:** Docker makes it easier to break the MERN stack into separate microservices. Each component (MongoDB, Express, Node.js, React) can be run in its own container, allowing you to scale and manage them independently.

Real-World Use Case: Docker Enhancing Development Workflow

Development Environment: Each team (frontend, backend, database) can work in isolated Docker containers, using different versions of Node.js or MongoDB as needed. Docker Compose can be used to spin up all services together, so developers can work on different parts of the system while still being able to interact with the entire stack in their local environment.

- The front-end team can run React in one container, ensuring their setup is isolated.
- The back-end team can have their Node.js + Express environment in another container.
- MongoDB can run in its own container.

CI/CD Integration: In a CI/CD pipeline, Docker images are built for each service. Every time a new feature is developed or a bug is fixed, the relevant service is containerized and tested in isolation. Automated tests can run against these containers, ensuring that updates don't introduce bugs.

- If the front-end team pushes changes to React, Docker builds a new container for the front-end and runs it alongside the existing services for automated testing.
- Similarly, if the back-end team pushes an update to the Express.js API, the back-end container is rebuilt and tested without affecting other services.

Deployment and Scaling: When deploying to production, Docker containers make scaling much easier. If the back-end API is handling heavy traffic, the Node.js container can be scaled up independently without affecting other services. Kubernetes (or Docker Swarm) can be used to orchestrate and manage scaling automatically.

- For example, the Node.js container handling the Express API could be scaled up to handle more requests during peak hours, while MongoDB containers may need less scaling.

Version Control for DevOps: Docker images provide version control for environments. If something breaks in production, developers can roll back to a previous Docker image quickly and reliably. This can prevent downtime and make disaster recovery more manageable.

CODE:

```yaml
# 2.DOCKER\4.Containerization\docker-compose.yml
version: '3'

services:
  # MongoDB service
  mongo:
    image: mongo:latest
    container_name: mongo_db
    restart: always
    ports:
      - "27017:27017"
    volumes:
      - mongo-data:/data/db
  # Express.js and Node.js service
  backend:
    build: ./backend
    container_name: node_backend
    restart: always
    ports:
      - "5000:5000"
    depends_on:
      - mongo
    environment:
      MONGO_URL: mongodb://127.0.0.1:27017/myDB
  # React front-end service
```

```
  frontend:
    build: ./frontend
    container_name: react_frontend
    restart: always
    ports:
      - "3000:3000"
volumes:
  mongo-data:
```

EXPLANATION:

1. MongoDB, Node.js (Express), and React all run in separate containers, with MongoDB data stored in a persistent volume.
2. This setup works in both development and production, ensuring consistency across environments.

CONCLUSION:

Docker helps in making the development, testing, and deployment of MERN stack applications more efficient by providing isolated environments, simplifying dependency management, enabling CI/CD pipelines, and scaling services independently. This ensures a smooth workflow for both small teams and large-scale applications.