### **Process Documentation and Key Decisions**

**Title**: **Containerizing a Full-Stack Web Application**

**Introduction**: This document describes the process of containerizing a React-Express-MongoDB application using Docker and Docker Compose, outlining each decision made along the way to ensure efficiency, scalability, and ease of deployment.

#### **Process Steps**

1. **Setting Up Project Structure**
   * Organized the application into separate frontend and backend folders, with MongoDB as the database.
   * Decided to create individual Dockerfiles for frontend and backend to facilitate independent builds and deployment.
2. **Creating Dockerfiles**
   * **Frontend Dockerfile**:
     + Chose a multi-stage build to reduce image size and improve security.
     + Used the node:20-alpine image in the first stage to install dependencies and build the React app.
     + In the second stage, used nginx:alpine to serve the built static files, making the image lighter and more secure.
   * **Backend Dockerfile**:
     + Used node:20-alpine as the base image to keep the backend image small and focused on running the Express server.
     + Set the working directory, installed dependencies, copied the code, and exposed port 5000 to run the server.
3. **Building and Pushing Images to Docker Hub**
   * Used a shell script to build and push images, leveraging the --no-cache option to ensure the latest code is always built and avoid cached layers.
   * Pushed the images to Docker Hub to make them accessible for deployment and shared environments.
4. **Configuring Docker Compose**
   * Used Docker Compose to define and manage the services.
   * Defined depends\_on to ensure each service starts in the correct order.
   * Added a restart policy (restart: always) to automatically restart services in case of failure.
5. **Port Mapping and Environment Variables**
   * Mapped host ports to container ports (e.g., 5001:5000 for the backend) to avoid conflicts with other services on the host.
   * Defined MONGO\_URI in docker-compose.yml for the backend to ensure a dynamic connection to MongoDB.
6. **Testing and Debugging**
   * Tested the setup locally, iterating over Docker Compose configurations to ensure each container operated as expected.
   * Used docker-compose up --build to force rebuilds during testing to capture all recent changes.

#### **Key Decisions**

1. **Multi-Stage Build for Frontend Dockerfile**
   * Using a multi-stage build with Nginx minimized the frontend image size by excluding build tools and development dependencies.
2. **Environment Variable Management**
   * Opted to define environment variables in docker-compose.yml to manage configurations centrally and streamline the Dockerfiles.
3. **Push to Docker Hub for Continuous Deployment**
   * Chose to automate the image push process to Docker Hub for consistent and accessible deployment across different environments.
4. **Use of docker-compose up --build in Script**
   * Including the --build flag in the script ensured that Docker Compose rebuilt images, preventing issues with stale images during deployment.
5. **Restart Policy for Reliability**
   * Adding restart: always to each service improved resilience by automatically restarting containers if they crashed.

#### **Conclusion**

Containerizing the application using Docker and Docker Compose provided a scalable and portable solution, making deployment across different environments consistent and efficient. Each decision aimed to enhance security, optimize performance, and simplify maintenance, preparing the application for production use.