Dockerize a React Application with Multi-Stage Build Report

Aim

To create an optimized, production-ready **Docker image** for a React Single Page Application (SPA) using a **multi-stage build process**. This process minimizes the final image size by separating the large build-time dependencies (like Node.js and npm) from the lightweight runtime environment (NGINX).

Objective

To successfully implement a Dockerfile that:

1. Uses a node image to build the React application (npm run build).
2. Discards all build dependencies and copies only the resulting static files (/build) into a fresh, small **NGINX** image.
3. Includes a custom nginx.conf to handle client-side routing (e.g., React Router) correctly by redirecting all unknown paths to index.html.
4. Produces a final Docker image that is efficient, secure, and ready for deployment.

Theory

Multi-Stage Builds

A multi-stage build is a Docker feature that allows you to use multiple FROM statements in a single Dockerfile. Each FROM directive starts a new build stage. You can selectively copy artifacts from one stage to another, leaving behind everything you don't want in the final image.

1. **Builder Stage:** Uses a large image (e.g., node:20-alpine) containing all necessary tools (npm, compilers) to compile the source code. The output is the optimized static bundle (build folder).
2. **Runner Stage (Final Image):** Starts with a minimal base image (e.g., nginx:alpine). It then copies *only* the static bundle from the builder stage, resulting in a significantly smaller and more secure production image.

NGINX Configuration for SPAs

Single Page Applications (SPAs) like React use client-side routing. If a user tries to access a route directly (e.g., /profile), the web server (NGINX) must be configured to serve the main index.html file for *all* non-file requests. This is achieved using the try_files directive in NGINX, which attempts to find a file, then a directory, and finally falls back to /index.html.

Procedure

1. **Project Setup:** Ensure a standard React project structure with a package.json and

source code.
2. **NGINX Configuration:** Create the nginx.conf file to properly handle React Router.
3. **Dockerfile Creation:** Write the multi-stage Dockerfile with the builder stage and the final runner stage.
4. **Build Image:** Execute the Docker build command, tagging the final image.
5. **Run Container:** Run the container, mapping a local port to the container's NGINX port (port 80), and verify the application runs correctly, including deep links.

Code

1. Dockerfile (Multi-Stage Build Definition)

```
# --- STAGE 1: Builder ---
# Uses a Node base image to handle compilation. We use 'alpine' for a
smaller intermediate image.
FROM node:20-alpine AS builder

# Set the working directory inside the container
WORKDIR /app

# Copy package.json and package-lock.json first to cache dependencies
COPY package*.json ./

# Install project dependencies
RUN npm install

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

# Build the React application. This creates the 'build' folder.
RUN npm run build

# --- STAGE 2: Runner (Production Image) ---
# Uses a lightweight NGINX base image. This will be the final, small
image.
FROM nginx:alpine

# Copy the custom NGINX configuration to handle React Router (SPAs)
COPY nginx.conf /etc/nginx/conf.d/default.conf
```

```
# Copy the production build artifacts from the 'builder' stage into the
NGINX serving directory.
# This step is the core of the multi-stage optimization.
COPY --from=builder /app/build /usr/share/nginx/html

# The container exposes port 80 (default NGINX port)
EXPOSE 80

# NGINX starts automatically as defined by the base image CMD
CMD ["nginx", "-g", "daemon off;"]
```

Learning Outcomes

1. **Image Optimization with Multi-Stage Builds:** Students will understand and implement a **multi-stage Docker build**, mastering the technique of separating build dependencies from the runtime environment to create highly optimized and compact production images.
2. **Containerizing SPAs:** The project teaches the professional standard for containerizing Single Page Applications (SPAs) like React, utilizing the lightweight **NGINX** web server to efficiently serve the static assets.
3. **NGINX for Client-Side Routing:** Students will learn how to configure a web server using the crucial `try_files` directive in `nginx.conf` to correctly handle client-side routing (like React Router), a mandatory step for containerized SPAs.