*FROM node:20-alpine as build-stage*

*WORKDIR /react*

*# If the package doesn't change, no need to copy it every time we run the build command  
COPY package\*.json .*

*RUN npm install*

*COPY . .*

*RUN npm run build*

*FROM nginx:stable-alpine as production-stage*

*COPY --from=build-stage /react/dist /usr/share/nginx/html*

*# the container is running at port 3000. It helps DevOps just seeing the docker file*

*# EXPOSE 3000*

*#start nginx and serve the application*

*CMD ["nginx","-g","daemon off;"]*  
  
  
  
Multi-stage builds help reduce the size of Docker images by allowing you to separate the build environment from the runtime environment. This way, you only include the necessary artifacts (such as the built application) in the final image and exclude any build tools and dependencies that are not needed at runtime. Here’s how multi-stage builds achieve this:

Isolate Build Dependencies:

The first stage (build-stage) uses a full development environment with all the necessary build tools and dependencies (in this case, Node.js and npm).

This stage includes everything needed to compile the application, but these build dependencies are not required to run the application.

Copy Only Necessary Artifacts:

After the application is built, the second stage (production-stage) starts from a minimal base image (in this case, an Nginx image).

It copies only the built application files from the first stage into the second stage, excluding all the unnecessary build tools and dependencies.

This results in a significantly smaller final image as it contains only what is required to serve the application.

Example Breakdown

Let's break down the provided Dockerfile to illustrate this:

Build Stage

dockerfile

FROM node:20-alpine as build-stage

WORKDIR /react

COPY package\*.json .

RUN npm install

COPY . .

RUN npm run build

Base Image: Uses a Node.js image (node:20-alpine) to provide a full development environment.

Dependencies: Installs Node.js dependencies required for building the application.

Build Process: Compiles the React application into static files.

Production Stage

dockerfile

FROM nginx:stable-alpine as production-stage

COPY --from=build-stage /react/dist /usr/share/nginx/html

CMD ["nginx","-g","daemon off;"]

Base Image: Uses a minimal Nginx image (nginx:stable-alpine), which is much smaller than the Node.js image.

Copy Artifacts: Copies only the built static files from the build stage into the Nginx image.

Runtime Environment: Contains only the Nginx server and the static files, without any Node.js or build tools.

Benefits

Smaller Image Size:

The final image is smaller because it only includes Nginx and the static files, not the Node.js runtime or the build dependencies.

For example, a Node.js image might be several hundred MBs, while an Nginx image could be much smaller. By excluding Node.js and related dependencies from the final image, you save significant space.

Security:

The final image is more secure because it contains fewer components and less surface area for potential vulnerabilities. Only the minimal runtime environment (Nginx) and the necessary static files are included.

Efficiency:

Builds can be faster and more efficient. Docker caches layers, so if your application code changes but your dependencies do not, Docker can reuse the cached layers for dependencies.

By isolating the build environment, you ensure that changes to the application code do not require re-installation of dependencies, which can save time.

Example Size Reduction

Assuming your node:20-alpine image is around 80 MB and includes all build tools and dependencies, the build stage might end up being quite large (e.g., 200 MB after installing dependencies and building the application).

The nginx:stable-alpine image, on the other hand, might be around 10 MB. By copying only the necessary built files (say 10 MB) into the Nginx image, the final image size could be around 20 MB (10 MB Nginx + 10 MB static files), compared to the much larger build-stage image.

This approach significantly reduces the size of the final Docker image, making it more efficient to distribute and run.