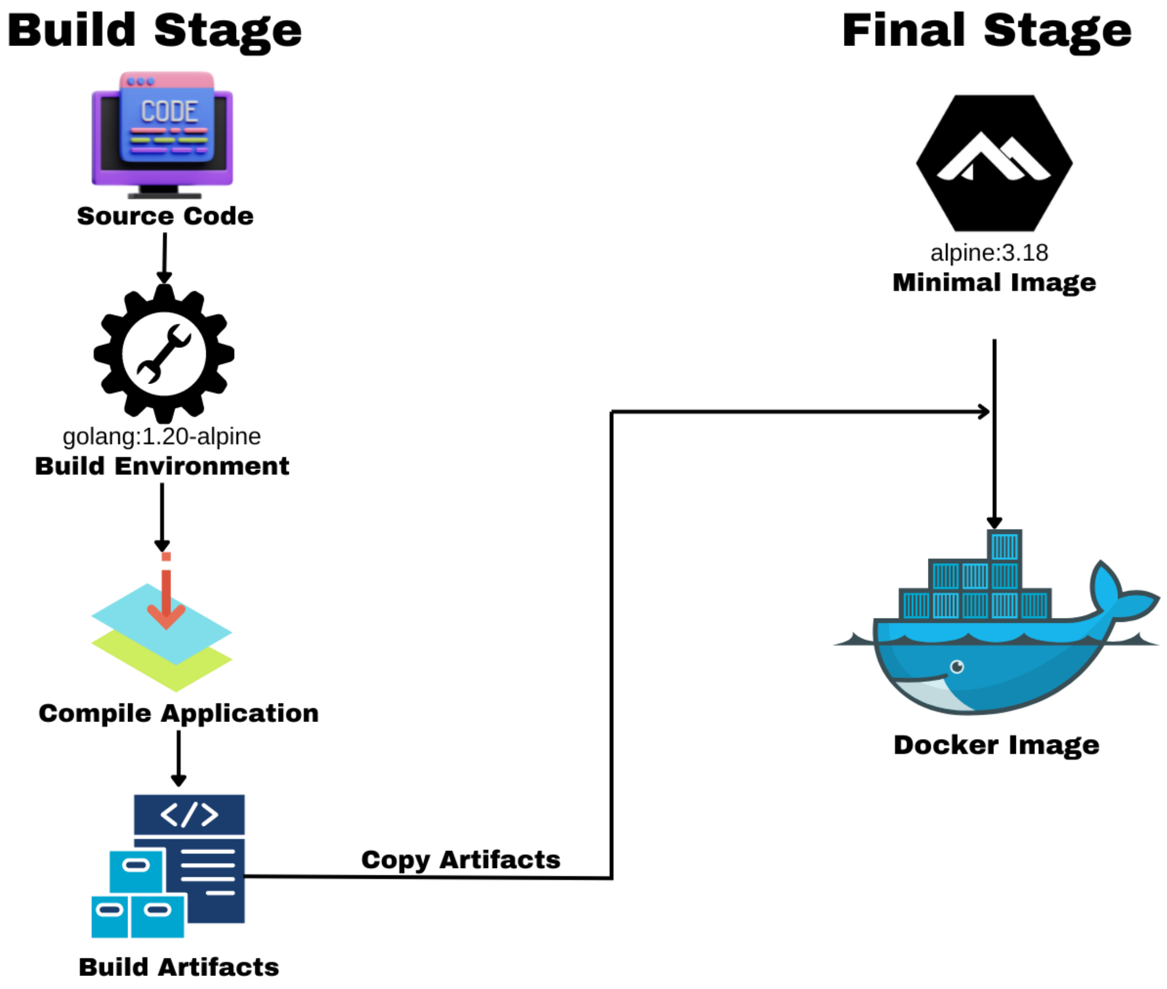
DevOps Shack

BEST SCENARIOS FOR USING MULTI-STAGE DOCKERFILES

Multi-stage Dockerfiles offer a powerful way to optimize your Docker images by separating the build process into distinct stages. This approach helps reduce image size, improve security, and streamline the deployment process. Here are some of the best scenarios where multi-stage Dockerfiles can be particularly beneficial.



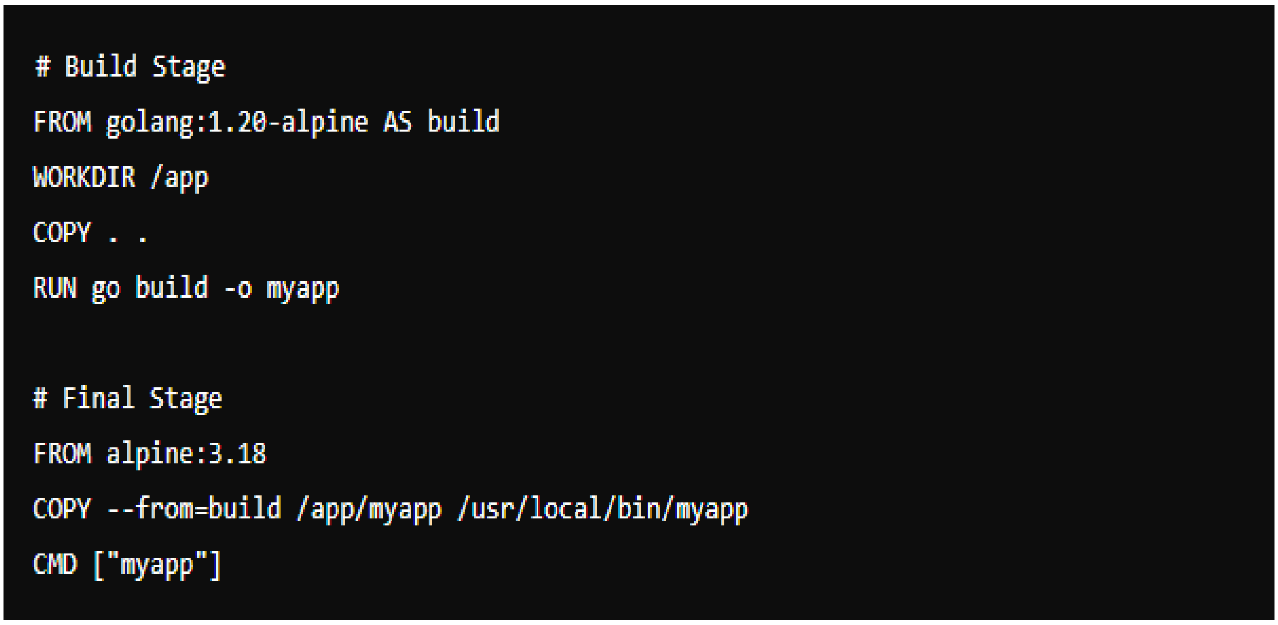
By using multi-stage Dockerfiles, developers can:

- **Reduce Image Size:** Exclude unnecessary build dependencies from the final image, resulting in a smaller, more lightweight container.
- **Improve Security**: Limit the attack surface by removing tools and libraries that are not required in production.
- **Simplify CI/CD Pipelines**: Integrate testing, building, and deployment into a single Dockerfile, making it easier to manage and automate the lifecycle of your application.

SCENARIO 1. Building and Packaging Applications

Scenario: You have a large application that requires compilation or building (e.g., a Go, Java, or Node.js application). The build process requires several development dependencies (e.g., compilers, libraries) that you don't want in the final production image.

Approach: Use the first stage to compile the application, where all development dependencies are installed. In the final stage, only copy the necessary artifacts (e.g., the compiled binaries) from the build stage into a clean, minimal base image. This significantly reduces the size of the final image.



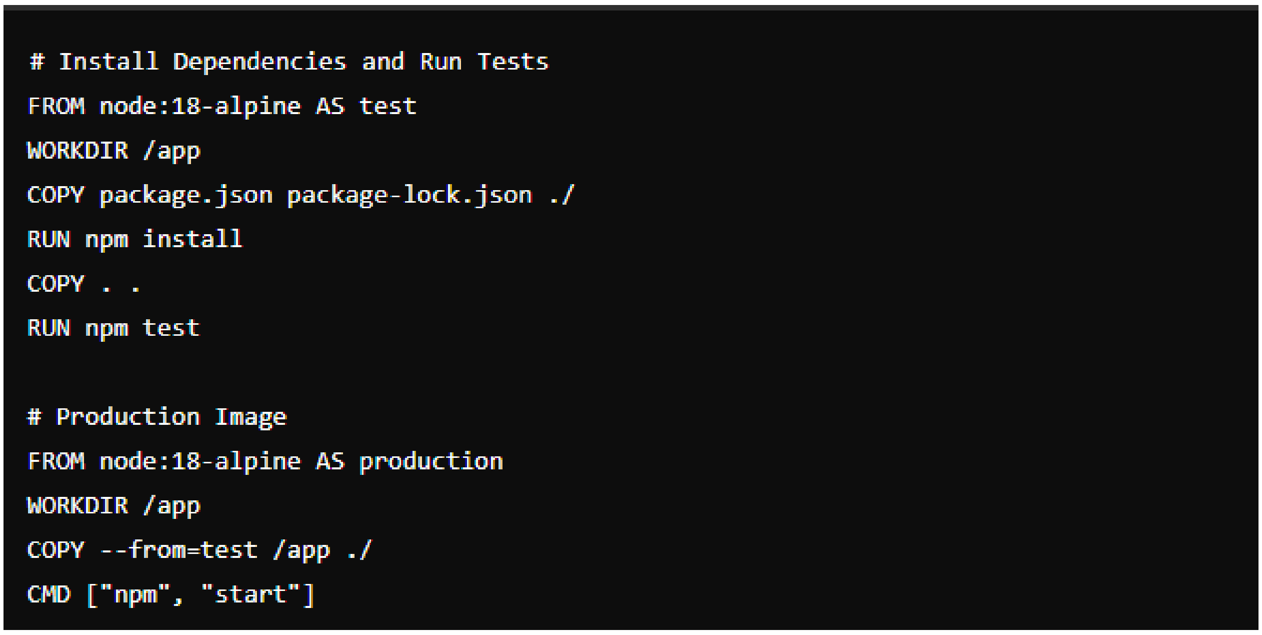
```
# Build Stage
FROM golang:1.20-alpine AS build
WORKDIR /app
COPY . .
RUN go build -o myapp

# Final Stage
FROM alpine:3.18
COPY --from=build /app/myapp /usr/local/bin/myapp
CMD ["myapp"]
```

SCENARIO 2. Testing During the Build Process

Scenario: Ensuring that your application passes all tests before it's included in the final image is crucial for maintaining high code quality and avoiding bugs in production. However, running tests often requires additional tools and dependencies that are not necessary in the production environment

Approach: In this scenario, you can use a multi-stage Dockerfile to separate the testing and production environments. The first stage is dedicated to installing dependencies and running tests. This stage includes all the necessary testing tools and libraries. Only if the tests pass do you proceed to the final stage, where a clean production image is built. This final image excludes all the unnecessary testing dependencies, ensuring a smaller and more secure production-ready image



```
# Install Dependencies and Run Tests
FROM node:18-alpine AS test
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm install
COPY . .
RUN npm test

# Production Image
FROM node:18-alpine AS production
WORKDIR /app
COPY --from=test /app ./
CMD ["npm", "start"]
```

SCENARIO 3. Building and Deploying a Web Application

Scenario: When developing a static website or a frontend application (like React or Angular), the build process typically involves steps like bundling and minification to optimize the code for production. Once built, these static assets need to be served efficiently by a web server such as Nginx.

Approach: In this scenario, the first stage of the multi-stage Dockerfile is used to build the static assets. This stage includes all the tools required for building, such as Node.js and package managers. Once the build process is complete, the final stage is focused solely on serving these assets using a lightweight Nginx server. By copying only the built static files into the Nginx container, you create a streamlined and efficient production image.

```
# Build Stage
FROM node:18 AS build
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm install
COPY . .
RUN npm run build

# Serve Static Files with Nginx
FROM nginx:alpine
COPY --from=build /app/build /usr/share/nginx/html
CMD ["nginx", "-g", "daemon off;"]
```