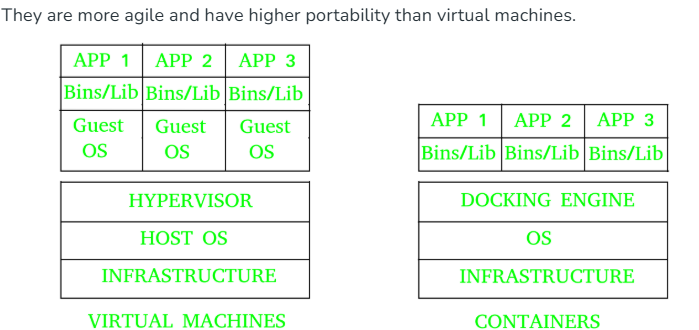


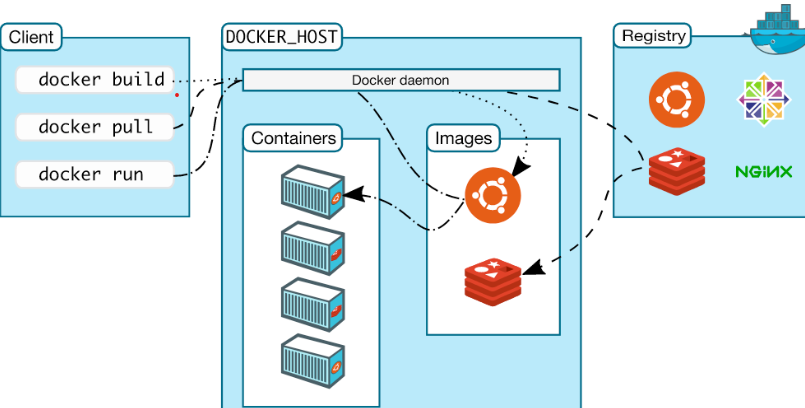
**Virtual Machine vs Container**

****

**How containers are logically isolated?**

1. Each container has its own filesystem & processes so one can’t see other’s files & processes.
2. Each container gets own private IP and hostname. Public IP is common i.e. host IP for external communication.
3. Each container runs its processes in its own ControlGroup which control how much resources a container can use so that other containers are not affected. (ControlGroup is a feature of linux kernel which controls limit of resources that processes can use)

**Architecture**



**Commands**

Docker build: Uses docker file to create an image. Image has: code, libraries, dependencies, runtime

Docker push: sends the built image to container registry (docker hub or acr)

Docker run: creates a container with my app and everything inside it

**Terminologies**

Docker daemon: dockerd listens to dockerAPI requests and manages all docker objects like images, containers, volumes, networks. It is heart of docker.

Docker client: It is docker CLI. Communicates with dockerd what to do using dockerAPI

Docker registry: version control for docker images

Docker image: Its a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization.

**Process:** Install docker. Add Ubuntu user to docker group, since docker runs as admin. So we need to add our user to docker group: sudo usermod –aG docker Ubuntu

**Docker file for python project**

FROM python:3.10  #get python image from docker hub. Here python image already includes minimal Linux OS

WORKDIR /app #set current working directory inside image to /app. All commands copy, run executed inside it

COPY requirements.txt #copy this file from host machine (where docker runs) to /app inside container

RUN pip install requirements.txt #install python packages listed in requirements.txt

COPY . . #copy everything from project folder on host to /app inside image. Keep unwanted files .dockerignore

ENTRYPOINT ["python3"] #Entrypoint can’t be changed at runtime of container

CMD ["app.py", “runserver”, 0.0.0.0:8000]   # Here CMD to run when container starts. here it runs python.py inside container. Can be changed at runtime of container. Application port on container I.e. 8000 should be mapped with the host

docker build –t dockerhub\_username/reponame:latest .

docker push dockerhub\_username/reponame:latest

docker run -p 8000:8000 -it 2435975254

**Multi stage docker file (NodeJS written in.ts)**  
Here we define multiple build stages. These stages are named and referenced within the Dockerfile, enabling seamless communication between them. Only the final stage which has entry-pint and CMD is key and used to run the application, all intermediate stages are discarded.

# ===== STAGE 1: Build =====

# Use full Node.js image with npm, compilers, and build tools preinstalled  
FROM node:18 as builder

# Set the working directory inside the container  
WORKDIR /app

# Copy package.json and package-lock.json to app/ in image.  
COPY package\*.json ./

# Install all dependencies from package.json. Docker caches npm install if package.json file don't change  
RUN npm install

# Copy the entire project code into the container  
COPY . .

# Compile and Build the project because its in Typescript (output will go to the dist/ folder)  
RUN npm run build

# ===== STAGE 2: Run =====

# Use a Node.js light weight “distroless image” with only runtime dependencies  
FROM node:18-slim

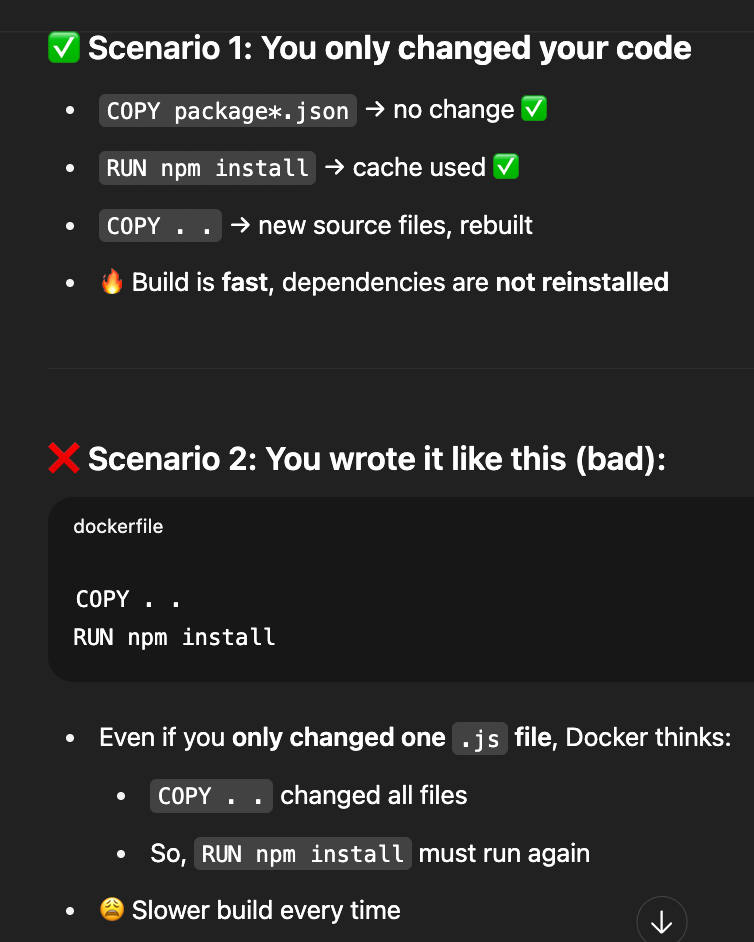
# Set working directory again for the final runtime image  
WORKDIR /app

# Copy the built output/compiled-code in dist folder from the builder stage  
COPY --from=builder /app/dist ./dist

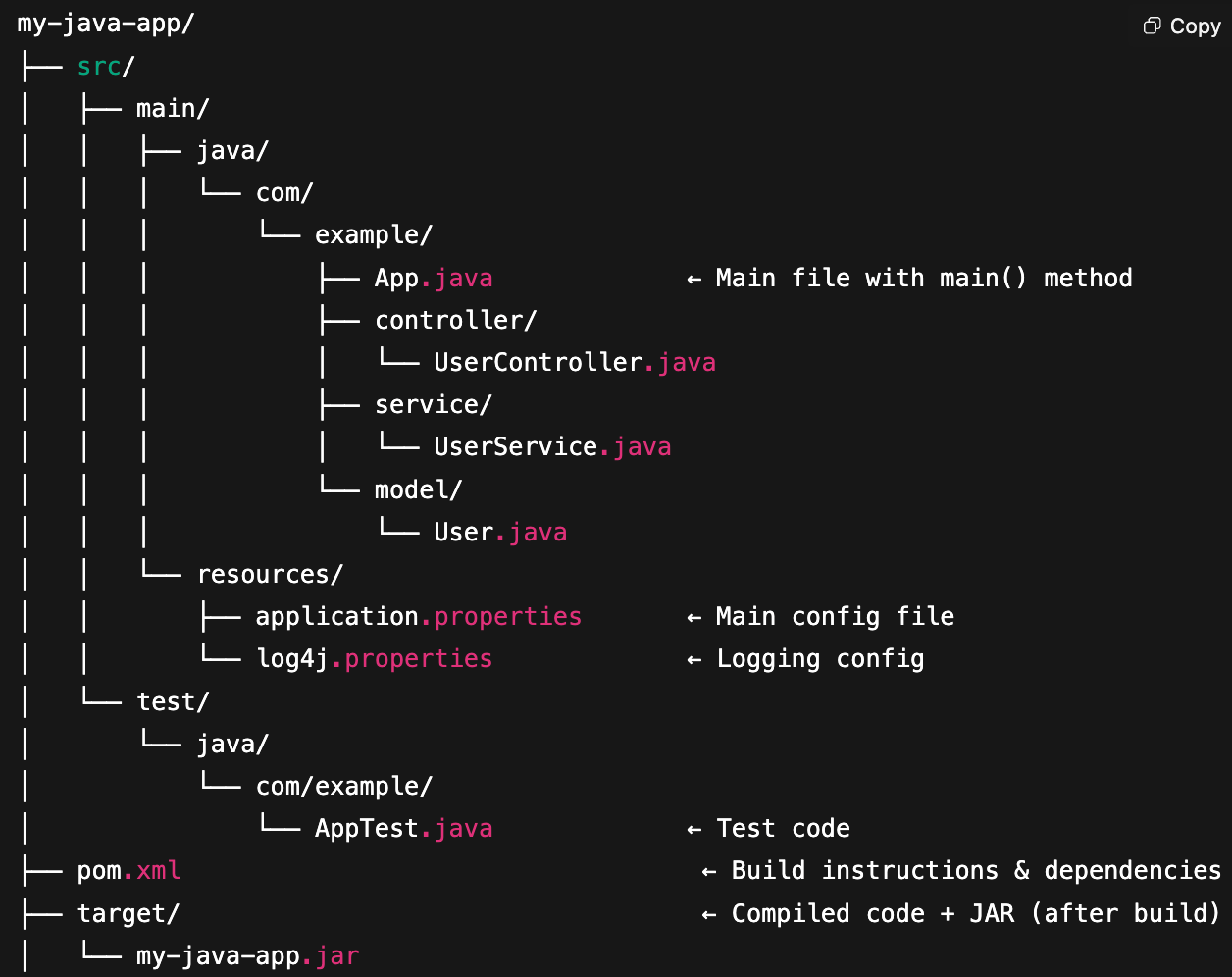
# Copy only the package.json and lock file to install production dependencies  
COPY --from=builder /app/package\*.json ./

# Install only production dependencies (skips dev tools)  
RUN npm install --only=production

# Define the default command to run when the container starts  
CMD ["node", "dist/index.js"] # Start the app using the built entry point

**Multi stage docker file (Java springboot)**



# ------------ STAGE 1: BUILD THE JAVA APP ------------

FROM maven:3.9-openjdk-17 AS builder

# Set working directory inside the container

WORKDIR /app

# Copy only pom.xml to leverage Docker cache for dependencies

COPY pom.xml .

# Download dependencies separately for caching

RUN mvn dependency:go-offline

# Copy source code

COPY src ./src

# Build the project and create the .jar file

RUN mvn clean package

# ------------ STAGE 2: RUN THE JAVA APP ------------

FROM openjdk:17-jdk-slim

# Set working directory for runtime

WORKDIR /app

# Copy only the built JAR from the builder stage

COPY --from=builder /app/target/\*.jar app.jar

# Expose app port (e.g., if Spring Boot uses 8080)

EXPOSE 8080

# Run the app

CMD ["java", "-jar", "app.jar"]