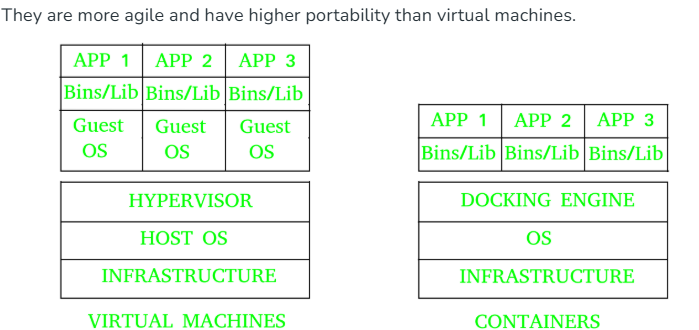
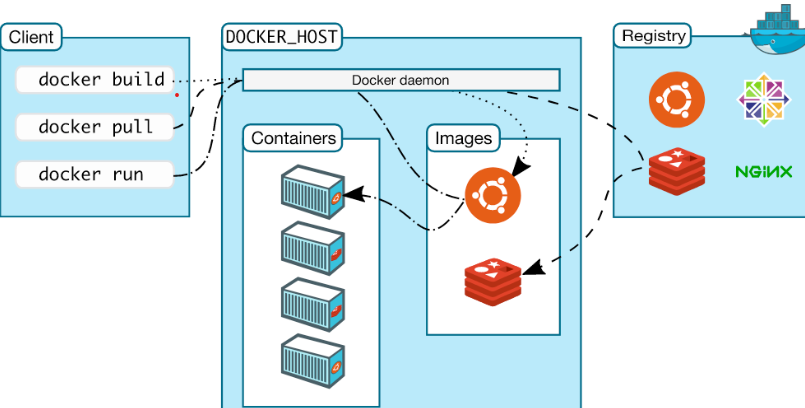
**Virtual Machine vs Container**

****

**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)

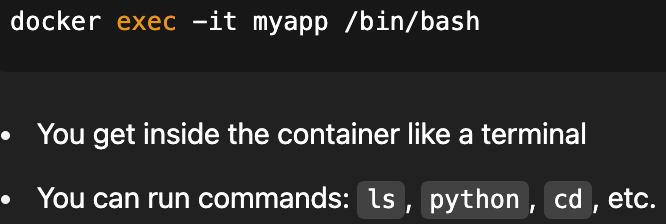
**Architecture**



**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 and start a new container (myapp) from docker image  
docker exec: execute any command inside a running container, without restarting it



docker ps: list all running containers

port mapping: host -> container

docker logs <container-id or container-name>: stream the logs as the container runs - useful for debugging

docker start/stop <container-id or container-name>

docker inspect <container-id or container-name>: shows IP address, status and other networks settings (like bridge/host or any other custom network)

docker network ls : to check all networks in docker  
docker network create apple: to create a custom bridge network  
docker network inspect apple: shows subnet, connected containers and other settings

**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 in .dockerignore

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

CMD ["app.py", “runserver”, 0.0.0.0:80]   #CMD runs when container starts. here it runs app.py inside container. Can be changed at runtime of container. Application port on container i.e. 80 should be mapped with the host (python3 app.py runserver 0.0.0.0:80)

docker build –t username/reponame:latest .

docker push username/reponame:latest

docker run -p 80:80 -d --name UtkarshJava utkarsh/my-java-app:latest

docker exec -it UtkarshJava /bin/bash

**Multi stage docker file (Typescript based NodeJS app)**  
A multi-stage Dockerfile uses multiple FROM statements to separate the build process (with all tools) from the runtime image (which is lightweight and clean). Only the runtime stage which has ENTRYPOINT and CMD is used to run the application, all intermediate stages are discarded. It reduces image size, improves security, and speeds up deployments

# ===== 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” which has only runtime dependencies  
FROM node:18-slim

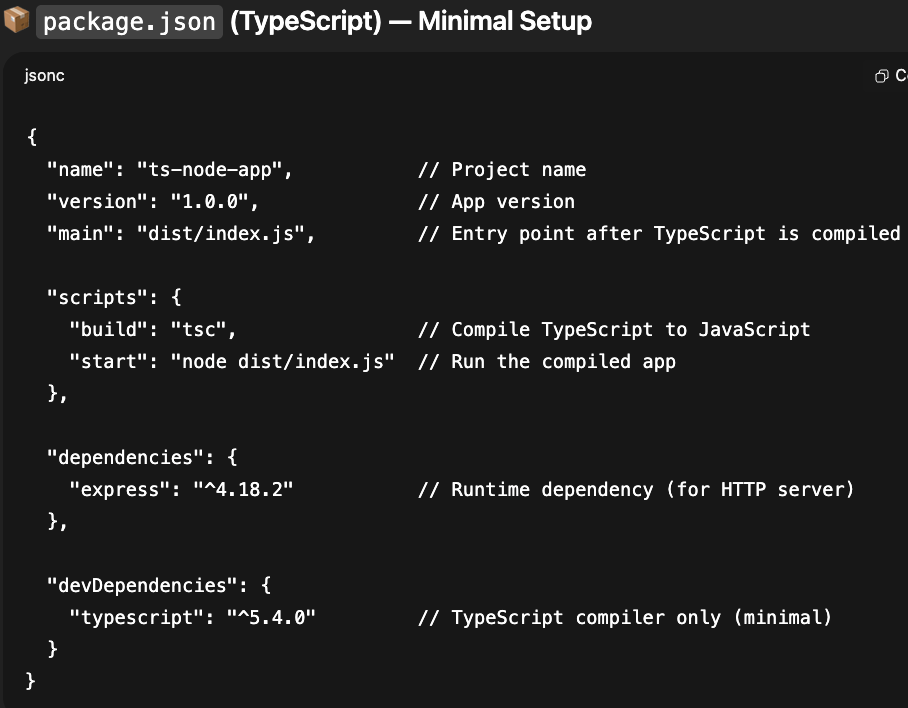
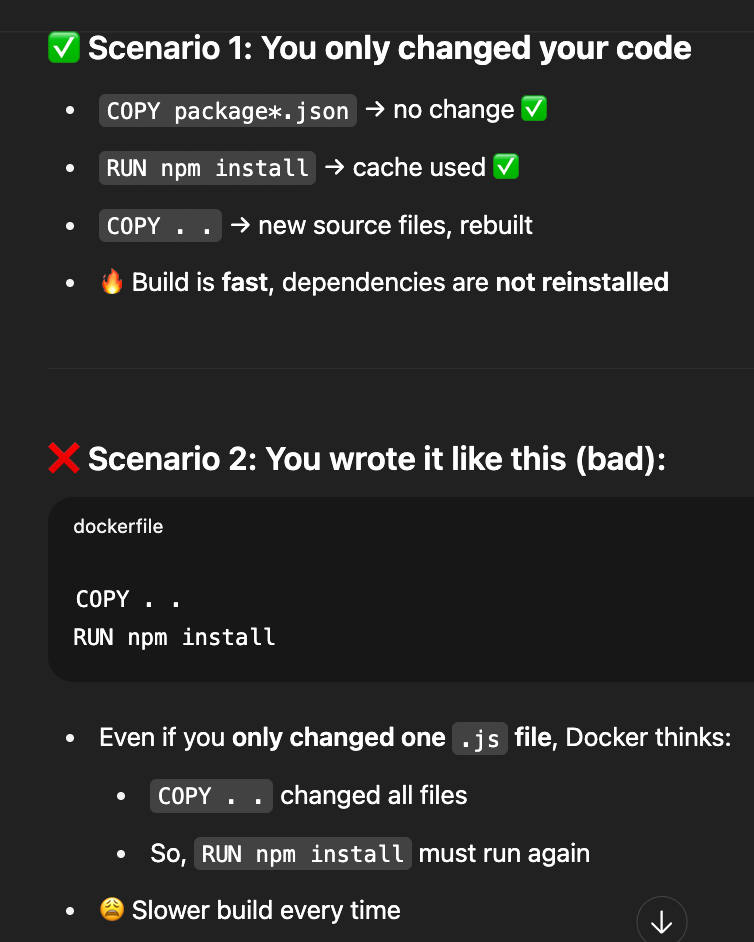
# Set working directory again for the final runtime image  
WORKDIR /app

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

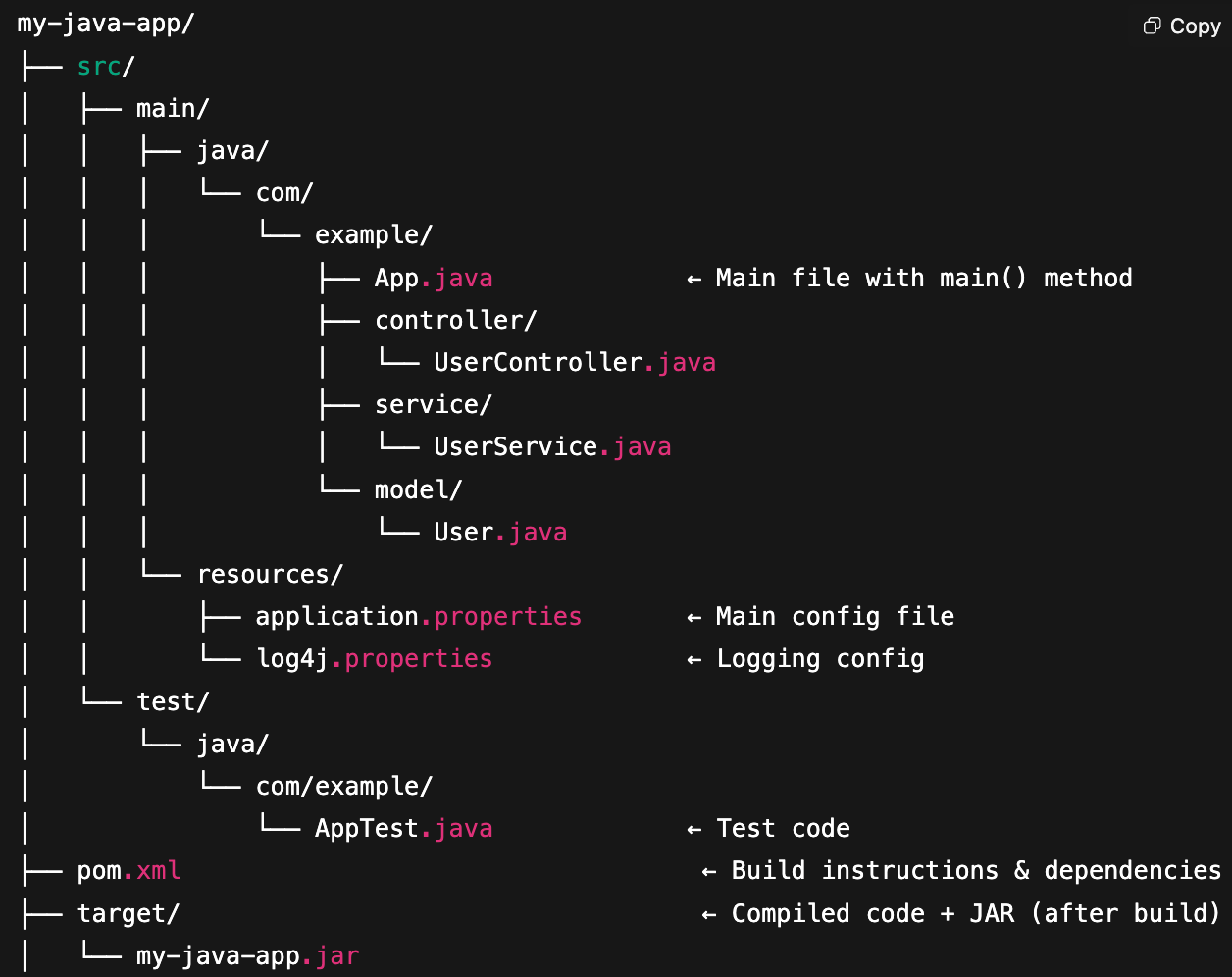
# Copy only the package.json and lock file to install production dependencies  
COPY --from=builder /app/package\*.json ./

# 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/app.jar ./

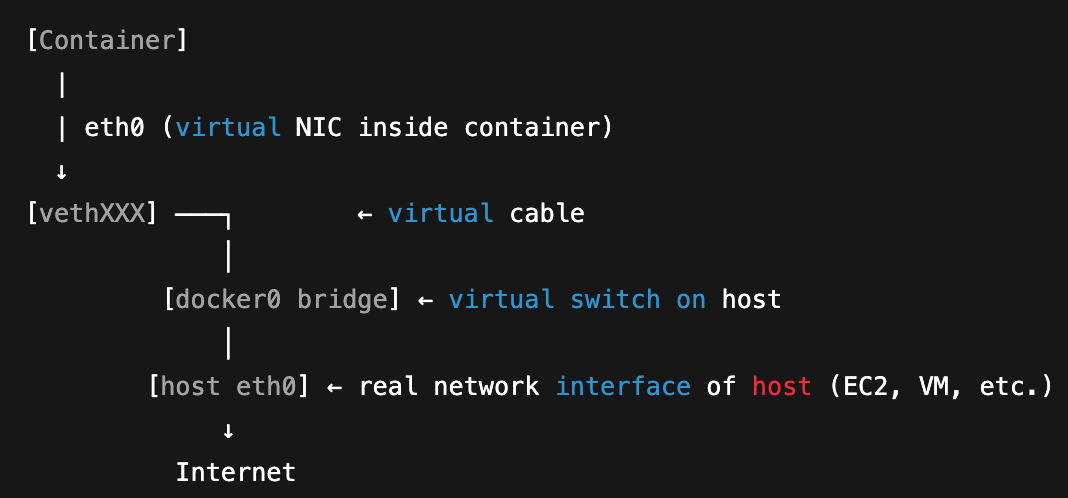
#Declares that the app inside the container listens on port 8080  
EXPOSE 8080

# Run the app  
CMD ["java", "-jar", "app.jar"]

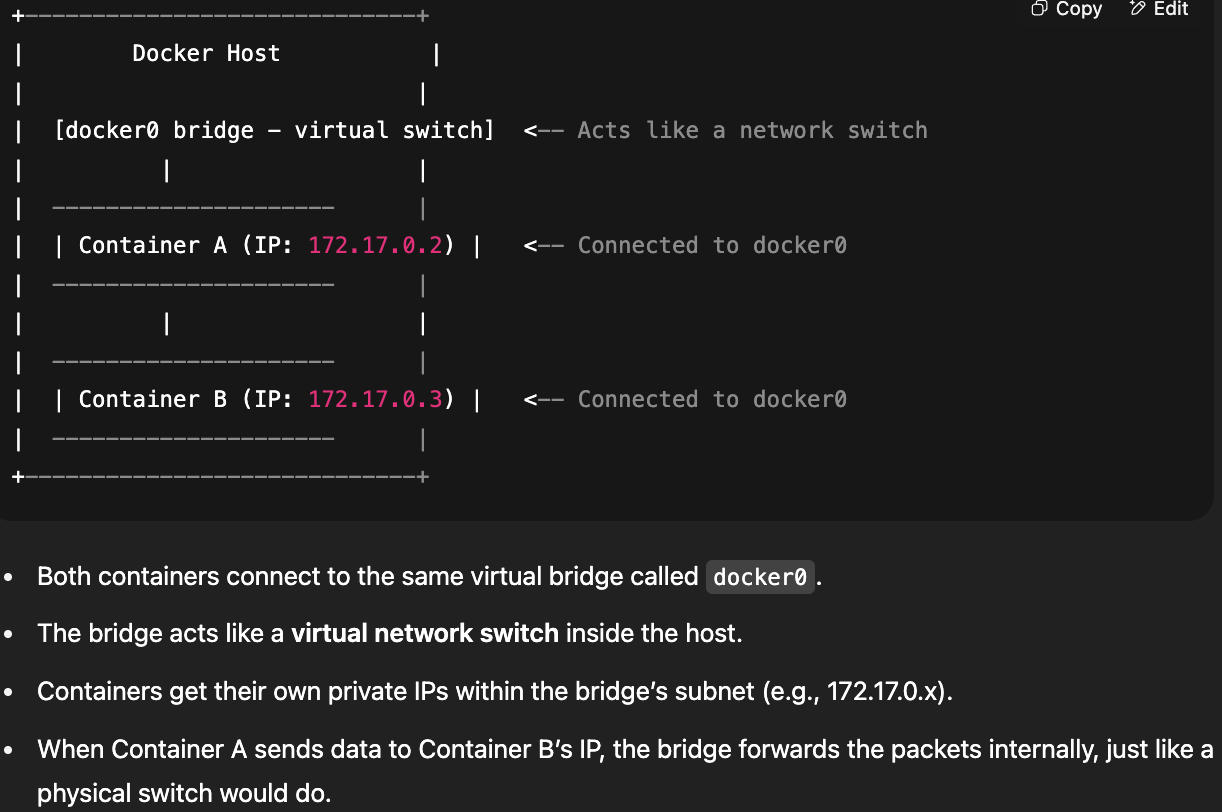
Run this docker container using -> docker run -p 8080:8080 --name springApp1 utkarsh/my-spring-app:latest

**Docker networking**

When Docker runs a container, it creates a virtual network connection between the container and the host using a veth pair. Its like virtual cable with 2 ends- eth0 (virtual network interface inside container), connected to host via docker0 bridge/switch (virtual switch on host).



1. Bridge Network- The Bridge network allows containers to communicate with each other and the host. Each container gets private IP, hence containers get network isolation.



1. Host Network- The container uses the same network as of host. The container uses the host’s eth0 interface. For example, if the host has eth0: 192.168.1.2, the container use host IP or might use eth0: 192.168.1.3 (hence same cidr block).  
   Eg- docker run -d --name databasetest --network host nginx:latest

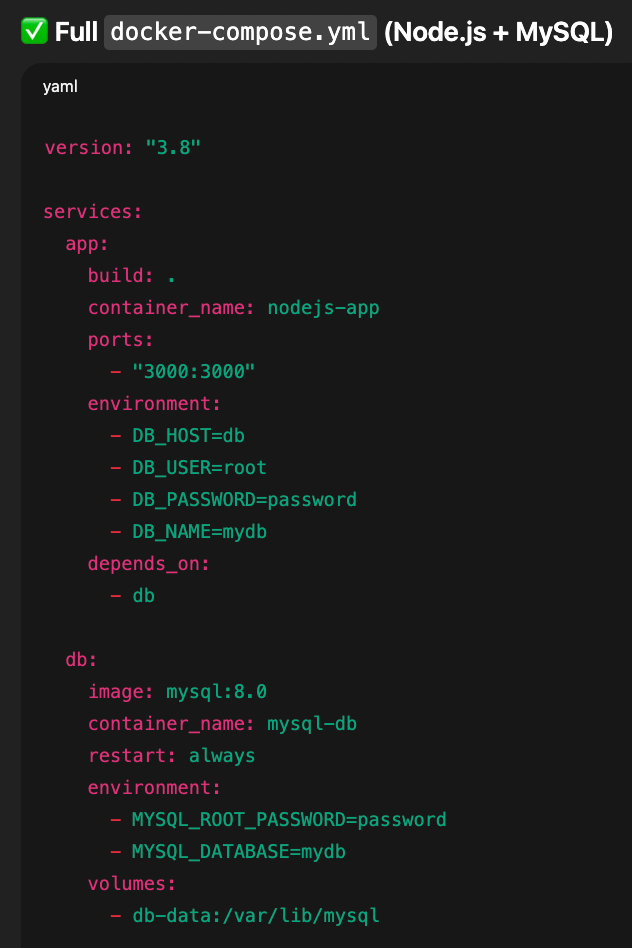
**IMPLEMENTATION:**  
First create 3 containers- frontend, backend, finance. Finance should be isolated, while frontend, backend should communicate with each other. So frontend, backend should use docker0 bridge, while finance should use custom bridge.  
docker run -d --name frontend nginx:latest  
docker run -d --name backend nginx:latest

docker network create secure //to create a custom bridge network  
docker run -d --name finance --network secure nginx:latest

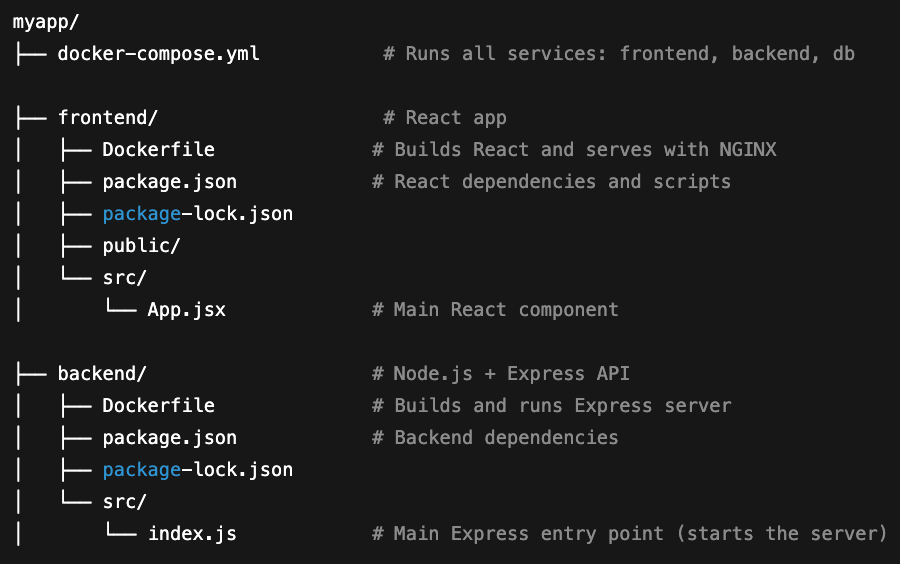
Now check network settings of all 3 containers using “docker network inspect frontend” etc. , you will find frontend, backend has same IPV4 subnet, while finance has different IPV4 subnet.

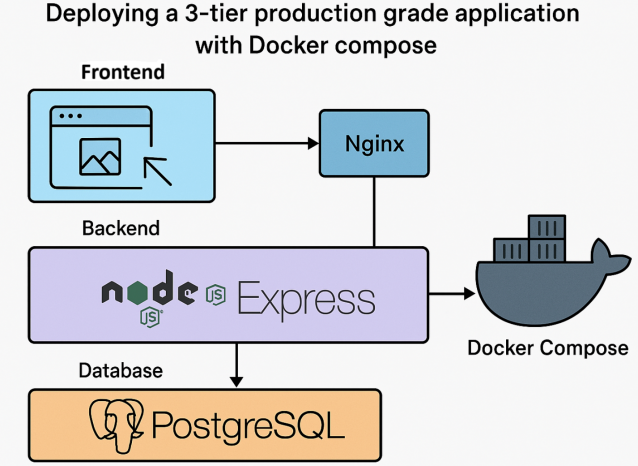
**Docker Compose**

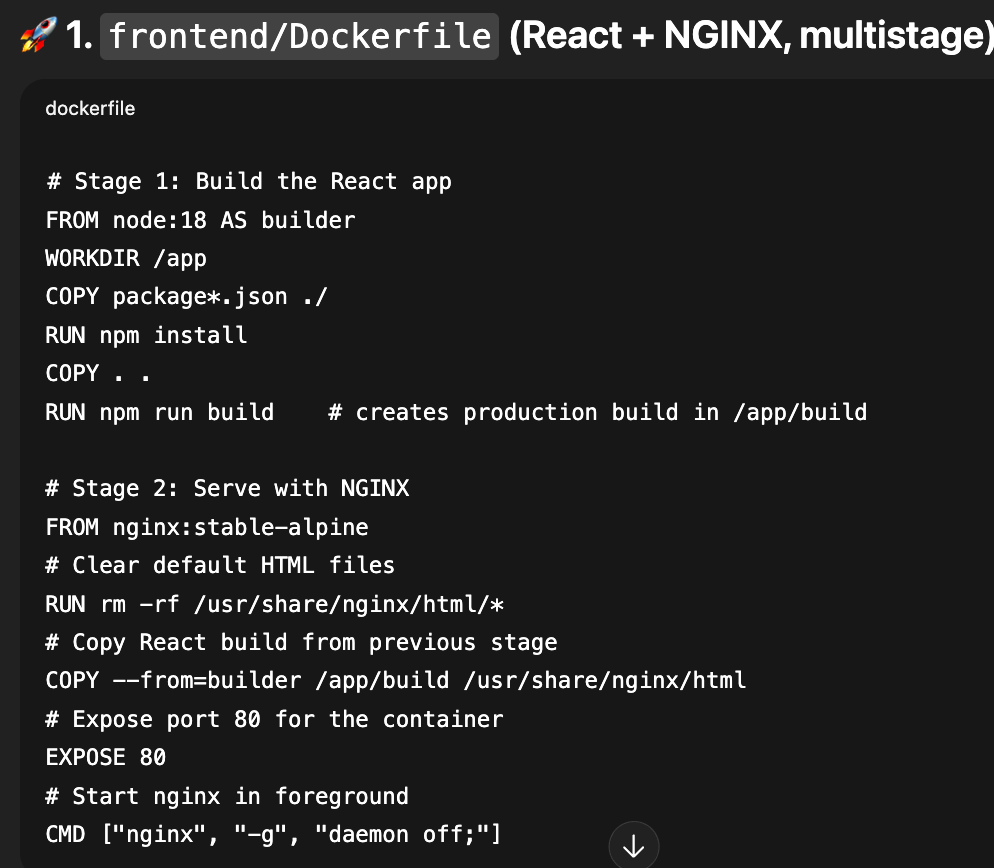
- docker-compose.yml file will first build docker file/image and then defines how to run multi-container Docker applications.  
- It simplifies the management of interconnected services like a web app, a db, and an API, by allowing them to be launched and controlled together.  
- When we run “docker-compose up”, docker creates a default network. Each container joins that network with a hostname which is their service name:



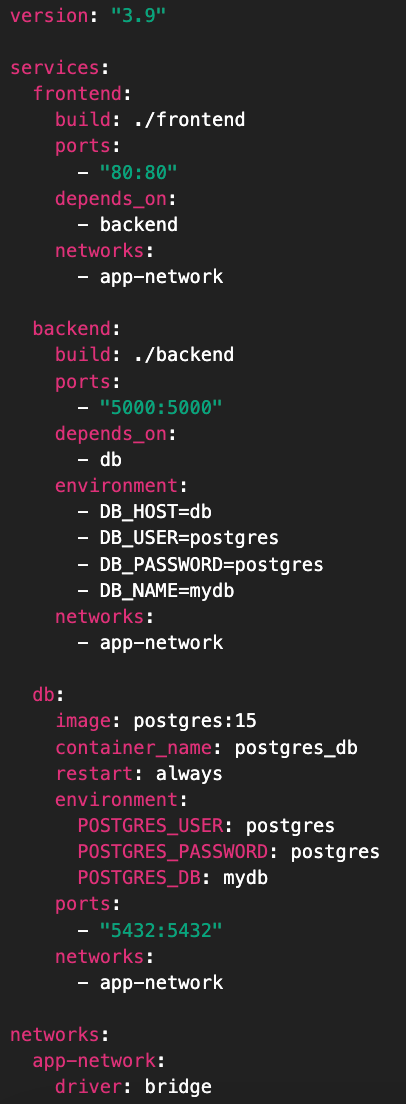
**3 tier application (React + NodeJS + PostgreSQL)**











- The environment variables in db help Docker automatically set up the database when the container is created. So, when the db container starts, it will: Create a database mydb, Create a user postgres with password postgres, Give that user access to the database.  
- These variables are used by your Node.js backend code to connect to the database.