Docker Notes before Interview :

Docker :

- Basically it resolves the problem of replication of multiple Environments.
- Suppose I have a NodeJS Application: Windows11 , NodeJS 16 , MongoDB 5 , Redis 6 etc..
- Now another Developer join my team but he is having : MacOS , NodeJS20 , MongoDB 6 , Redis 7 etc...

How will he run the same NodeJS application on his PC?

This was the major issue solved by Docker. It takes care of multiple environments and dependencies.

Docker is a platform that helps solve the problem of environment inconsistency across different systems. Imagine I have a Node.js app developed on my macOS using Node.js v16, MongoDB v6, and Redis v6. Now, if someone else wants to run this app on their system, but they have different versions—like Node.js v20, MongoDB v7, and Redis v7—it might lead to compatibility issues. Docker fixes this by allowing us to package the entire application along with its dependencies, configurations, and environment into a single container image. This image can then be run anywhere, ensuring consistency regardless of the host machine. It's like a 'build once, run anywhere' solution.

Docker Container vs VM :
Docker containers and virtual machines (VMs) are different, although both are used to isolate environments.
⍰ **Architecture:**

- **VMs** run on a hypervisor and include an entire operating system (guest OS), which makes them heavier.
- **Docker containers** share the host OS kernel and are more lightweight since they only package the app and its dependencies.

⍰ **Startup Time:**

- VMs take minutes to boot up because of the full OS.
- Containers start in seconds because they don't need to boot an OS.

⍰ **Resource Usage:**

- VMs consume more CPU and memory.
- Containers are more efficient and need fewer resources.

⍰ **Isolation:**

- Both offer isolation, but VMs are more isolated due to separate OS layers.
- Containers are isolated at the process level, using namespaces and cgroups.

Docker CLI and Desktop :

- Docker is having Daemon and desktop : Daemon is a real docker which runs the manages the container push or pull containers and desktop is just a UI.

- Now if I want to run ubuntu OS in my Windows OS then I will run: "docker run -it ubuntu" If this image is not present in my PC then it will pull it from docker hub (Just like GitHub for docker).

  Docker has two main interfaces for users—Docker CLI and Docker Desktop. Docker CLI (Command Line Interface) is a tool that lets us interact with the Docker engine, also known as the Docker Daemon, using commands like docker build, docker run, docker push, and so on. The Docker Daemon is the core service that handles all Docker-related operations such as container creation, image management, and networking.
  Docker Desktop, on the other hand, is a graphical user interface (GUI) for managing Docker. It's especially useful for developers who prefer visual tools over the command line.

- Containers are isolated from rest our PC.

- Docker Image : It is a blueprint and read only. Having everything to run an application(recipe)
- Docker Container : Instance of an Image. Actual dish made from that recipe , running version of an image.

- A **Docker Image** is a read-only template that contains the application code, runtime, libraries, environment variables, and configuration files needed to run an application. It's a snapshot of a filesystem and is used to create containers. Images are built using a Docker file and stored in registries like Docker Hub.

- A **Docker Container** is a runtime instance of a Docker image. It includes the image plus a writable layer where the application can write and modify data during execution. Containers are isolated environments that run on the same host OS using Docker Engine, and they can be started, stopped, moved, or deleted.

In summary:

Docker Image is a lightweight, read-only package that contains everything needed to run an application—code, runtime, libraries, and dependencies. You can think of it like a class in object-oriented programming: it's a blueprint. A Docker Container, on the other hand, is like an object or instance of that class. It's a running instance of an image, isolated from the rest of the system. While the image is static and reusable, the container is dynamic and can be started, stopped, or removed. This distinction helps Docker achieve consistency across environments by using the same image to spin up containers anywhere.

---

docker run -it image_name : Run or pulls a particular Image

docker container ls : List running docker containers

docker container -a : Lists all docker containers (running or not running).

docker start container_name : Starts a container

docker stop container_name : Stops a container

docker exec -it container_name bash : Inside a particular container and can do anything we want in an isolated env.

<u>Port Mapping :</u>

Why ?

- Basically if we run a node server inside a container for example it running on PORT 3000 but when I run localhost://3000 it will not show me anything why because container run systems in an isolated environment.

  To resolve this we have port mapping :

  docker run -it -p 3000:3000 image_name : Expose Port 3000 of container to outside our PCs port 3000

- Port mapping in Docker is used to make a container's internal application accessible from outside the container. For example, if my application inside the container is running on port 3000, I won't be able to access it directly from Postman or a browser because containers are isolated environments. To expose this app, I need to map the container's port 3000 to a port on my host machine using the -p flag. So if I run docker run -p 3000:3000 image_name, it maps port 3000 of my PC to port 3000 inside the container. Now, when I go to localhost:3000 in Postman or browser, the request is forwarded to the app running inside the container.

docker run -it -p 6000:3000 image_name : Expose Port 3000 of container to outside our PCs port 6000

Environment Variables : docker run -it -p 6000:3000 -e key=value -e key=value image_name

Docker allows us to pass environment variables to containers using the -e flag. For example, we can run docker run -p 3000:3000 -e NODE_ENV=production image_name. This sets the environment variable NODE_ENV inside the container, which the application can use during runtime.

Now how can we create our own Docker Image for our project :

1⃣ **Create a Node.js application** – For example, a simple server using Express or HTTP module.

Write a Dockerfile – This file contains instructions to build the image, like:

Dockerfile

CopyEdit

```
FROM node:16

WORKDIR /app

COPY . .

RUN npm install

CMD ["node", "index.js"]
```

Build the image – We use the command docker build -t rv-nodejs . which creates a Docker image named rv-nodejs.

Run the container – Use docker run -it -p 8000:8000 rv-nodejs to map ports and run the container interactively.

Access the container (optional) – We can use docker exec -it <container_id> /bin/bash to open a terminal inside the running container.

Set environment variables – We can pass them using the -e flag or an .env file during the docker run command.

Suppose we have to expose it on docker hub then “docker build -t “repo_name””

Build the image with your Docker Hub username as the prefix:
docker build -t username/repo_name .
(For example: docker build -t rv123/my-node-app .)

Login to Docker Hub from your terminal:
docker login
(You'll enter your Docker Hub username and password.)

Push the image to Docker Hub:
docker push username/repo_name
(For example: docker push rv123/my-node-app)

Caching : It will detect changes and run only those changes commands

Each line of code in Dockerfile is a layer.

Docker Compose :

Why ?

For example If I have to run a big application so i have to make different containers like :

C1 : postgres

C2 : Nodejs

C3 : Redis ....

Now we have to execute them again and again for different container this problem was resolved by docker compose.

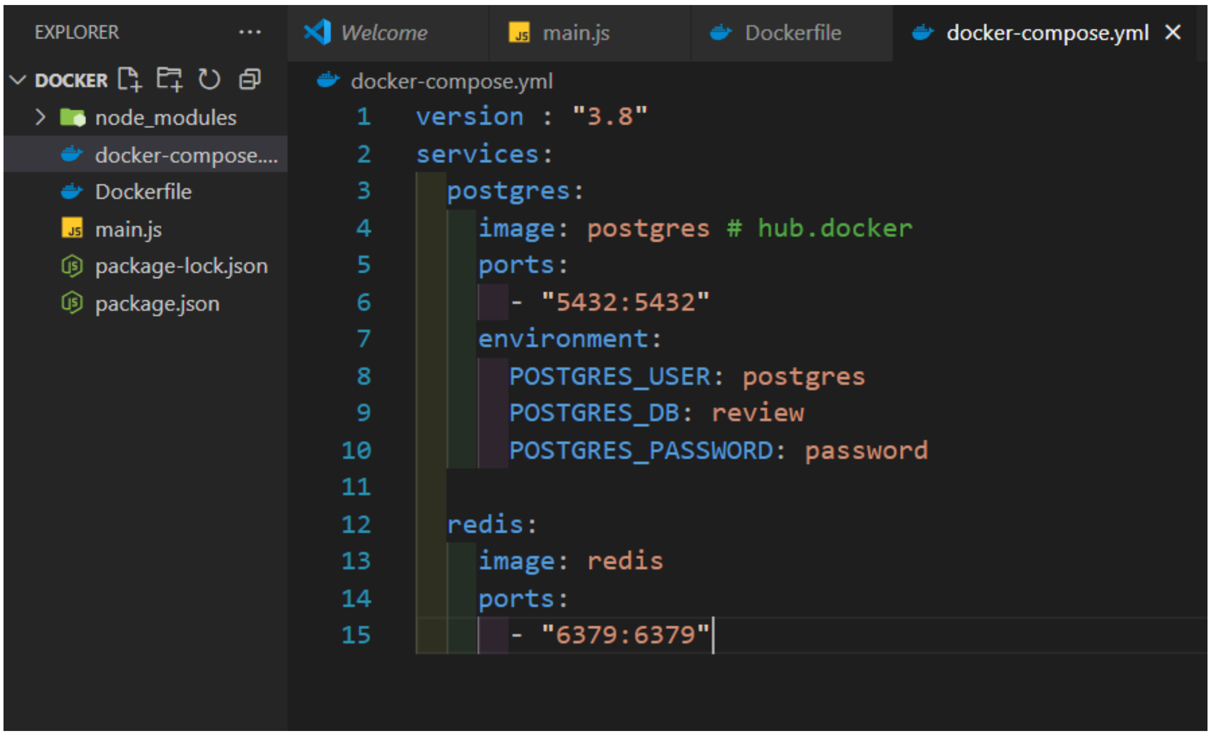
Docker Compose is a tool that helps us manage multi-container applications easily. Suppose I have a large application that requires several services like Node.js for backend, PostgreSQL for the database, and Redis for caching. Running each container manually with long docker run commands becomes repetitive and error-prone. Docker Compose solves this by allowing us to define all our services in a single docker-compose.yml file.

In this file, we can specify details like the image, ports, environment variables, volumes, and dependencies for each service. Then, with a single command—docker-compose up—we can bring up the entire application stack. This simplifies development, testing, and deployment of complex apps.

To resolve this problem :

1. Create a file docker-compose.yml file.

2. Write code there : example :



```yaml
version : "3.8"
services:
  postgres:
    image: postgres # hub.docker
    ports:
      - "5432:5432"
    environment:
      POSTGRES_USER: postgres
      POSTGRES_DB: review
      POSTGRES_PASSWORD: password

  redis:
    image: redis
    ports:
      - "6379:6379"
```

then run : docker compose up

or to stop : docker compose down