**Docker Notes – From Basic to Advanced**

**Referred the below video:**

[**https://youtu.be/9bSbNNH4Nqw?si=SLW9\_j42P47m9aRd**](https://youtu.be/9bSbNNH4Nqw?si=SLW9_j42P47m9aRd)

**Basics**

**What is Docker?**

Docker is a platform for developing, shipping, and running applications in lightweight containers.

**Why use Docker?**

* Portable: Same environment across development, testing, and production.
* Lightweight: Containers share the OS kernel.
* Fast: Containers start instantly compared to VMs.

**Key Concepts:**

* **Image**: Blueprint of a container (like a class).
* **Container**: Running instance of an image.
* **Dockerfile**: Script to create Docker images.
* **Docker Hub**: Registry for storing Docker images.
* **Volume**: Persistent data storage.
* **Network**: Communication bridge between containers.

**Common Docker Commands**

| **Command** | **Description** |
| --- | --- |
| docker pull <image> | Download image from Docker Hub |
| docker run -d -p 8080:80 <image> | Run container in detached mode |
| docker ps -a | List all containers |
| docker exec -it <container\_id> bash | Access container shell |
| docker stop <container\_id> | Stop container |
| docker rm <container\_id> | Remove container |
| docker rmi <image> | Remove image |
| docker build -t myimage:v1 . | Build image using Dockerfile |

**Dockerfile Template**

FROM openjdk:17-jdk-slim

WORKDIR /app

COPY target/my-app.jar my-app.jar

EXPOSE 8080

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

**Docker Volumes**

**What is a Docker Volume?**

A **Docker Volume** is a persistent storage mechanism used to store and share data between Docker containers and the host system, or between multiple containers.

Unlike container filesystems which are ephemeral (data is lost once a container is deleted), volumes **retain data even after a container is removed**, making them essential for production systems, databases, logs, and more.

**Why Use Docker Volumes?**

* **Persistence**: Data survives container restarts and deletions.
* **Sharing**: Volumes can be shared across multiple containers.
* **Isolation**: Keeps data separate from the container filesystem.
* **Backup & Restore**: Easily back up and migrate volumes.
* **Performance**: Better I/O performance on Linux vs. bind mounts.

**Where Are Volumes Stored?**

Docker volumes are stored in:

/var/lib/docker/volumes/

Each volume gets its own directory with its data.

**Docker Volume Commands**

# Create a volume

docker volume create myvolume

# List all volumes

docker volume ls

# Inspect volume details

docker volume inspect myvolume

# Remove a volume

docker volume rm myvolume

# Remove all unused volumes

docker volume prune

**Using Volumes in docker run**

docker run -d \

--name mycontainer \

-v myvolume:/app/data \

myimage

This mounts the volume myvolume to the path /app/data inside the container.

**Using Volumes in docker-compose.yml**

services:

db:

image: mysql:8.0

volumes:

- db-data:/var/lib/mysql

volumes:

db-data:

This creates and mounts a named volume db-data for the MySQL container.

**Bind Mount vs Volume**

| **Feature** | **Bind Mount** | **Volume** |
| --- | --- | --- |
| Host Path | Must be specified manually | Managed by Docker |
| Portability | Less portable | Fully portable |
| Backup/Restore | Harder | Easier (standard path) |
| Performance | Slower on Linux | Optimized |
| Use Case | Development (host file editing) | Production (databases, logs) |

**Docker Compose Overview**

**What is Docker Compose?**

A tool for defining and running multi-container Docker applications with a single docker-compose.yml file.

**Common Commands:**

| **Command** | **Description** |
| --- | --- |
| docker-compose up -d | Run all services in background |
| docker-compose down -v | Stop, remove services and delete volumes |
| docker-compose logs | Check logs |
| docker-compose ps | List all containers managed by Compose |

**What is Docker Networking?**

Docker networking allows containers to **communicate** with each other, with the Docker host, and with external networks (like the internet). It is a critical aspect of container orchestration, especially in multi-container apps like microservices.

**Purpose:**

* Container-to-container communication
* Container-to-host communication
* Isolating containers on different networks
* Load balancing and service discovery

**Types of Docker Networks**

Docker provides several types of networks out-of-the-box:

| **Network Type** | **Description** | **Use Case** |
| --- | --- | --- |
| **bridge** *(default)* | Creates a private internal network on the host | Good for standalone container apps |
| **host** | Shares the host’s network stack | Use when container needs max network performance or host-level networking |
| **none** | Disables all networking | Useful for security, testing isolated behavior |
| **overlay** | Enables communication between containers across multiple Docker hosts (Swarm) | Distributed multi-host deployments |
| **macvlan** | Assigns MAC address to container; appears as physical device | For applications that need to appear on the physical network |
| **custom bridge** | User-defined bridge with custom subnet, DNS | More control over inter-container comms |

**Note**: Inspect a network using:

docker network inspect <network-name>

**Docker Container Lifecycle**

Represents the **journey of a container** from creation to deletion.

1. **Create**
   * Command: docker create
   * Container is created but not started.
2. **Start**
   * Command: docker start <container\_id>
   * Container begins execution.
3. **Running**
   * Actively executing processes inside container.
4. **Pause** *(optional)*
   * Command: docker pause <container\_id>
   * Suspends processes without killing.
5. **Unpause** *(optional)*
   * Command: docker unpause <container\_id>
6. **Stop**
   * Command: docker stop <container\_id>
   * Gracefully stops all processes.
7. **Kill**
   * Command: docker kill <container\_id>
   * Immediately kills all processes.
8. **Restart**
   * Command: docker restart <container\_id>
9. **Remove**
   * Command: docker rm <container\_id>
   * Deletes the container.

Checking all container states via: docker ps -a

**Common Dockerfile Lifecycle Stages:**

| **Stage** | **Instruction** | **Purpose** |
| --- | --- | --- |
| **Base Image** | FROM | Sets the base OS/image |
| **Maintainer** | LABEL | Metadata about the image |
| **Run Commands** | RUN | Executes shell commands during build |
| **Working Directory** | WORKDIR | Sets working directory for following instructions |
| **Copy/Add Files** | COPY / ADD | Moves files into image |
| **Install Dependencies** | RUN apt-get install | Adds required packages |
| **Set Environment** | ENV | Sets environment variables |
| **Expose Ports** | EXPOSE | Declares ports to open |
| **Set Entry Command** | CMD / ENTRYPOINT | What to run when container starts |

The result of the Dockerfile lifecycle is a **layered image** which you can run as a container using docker run.

**Differences Between CMD vs ENTRYPOINT**

| **CMD** | **ENTRYPOINT** |
| --- | --- |
| Provides default arguments | Defines executable |
| Can be overridden in docker run | Harder to override unless using --entrypoint |
| Best for configuration | Best for setting the actual command |

Absolutely, Sreyas! Here's the **complete Docker Workflow**, explained step-by-step — from writing code to deploying it using Docker — along with where the lifecycle elements fit in.

**Docker Workflow:**

This workflow shows **how to use Docker to containerize and deploy an application** — including image creation, networking, and lifecycle usage.

**1. Write Your Application Code**

You develop your application in any language (Java, Spring Boot, Python, Node.js, etc.). Alongside it, you prepare:

* Dockerfile
* Optional: .dockerignore
* Optional: docker-compose.yml (for multi-container apps)

**2. Create Dockerfile**

A Dockerfile contains **instructions to build your custom image**.

**Sample for Spring Boot:**

FROM openjdk:17-jdk-slim

WORKDIR /app

COPY target/myapp.jar app.jar

EXPOSE 8080

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

This is where the **Dockerfile Lifecycle** happens — each instruction becomes an image layer.

**3. Build the Docker Image**

You use the Dockerfile to **build the image**:

docker build -t myapp:latest .

This creates an image locally in your Docker host.

**4. Run a Container from the Image**

You now **start a container** from the built image:

docker run -d -p 8080:8080 --name myapp-container myapp:latest

This triggers the **Container Lifecycle**:

* Create ➝ Start ➝ Running

**5. Docker Networking Setup**

By default, containers are attached to the **bridge network**. But you can:

docker network create --driver bridge mycustomnetwork

docker network connect mycustomnetwork myapp-container

Or use docker-compose.yml to define services with isolated or shared networks easily.

**6. Inspect, Log, and Debug**

Check container state:

docker ps

docker logs myapp-container

docker inspect myapp-container

Pause, unpause, stop, or restart containers based on the **lifecycle** stage you’re managing.

**7. Stop & Remove Containers/Images (Cleanup)**

docker stop myapp-container

docker rm myapp-container

docker rmi myapp:latest

This covers:

* Stop
* Remove

**8. Use Docker Compose (Optional for Microservices)**

docker-compose.yml example:

version: '3'

services:

backend:

build: .

ports:

- "8080:8080"

networks:

- app-network

db:

image: mysql:8

environment:

MYSQL\_ROOT\_PASSWORD: root

networks:

- app-network

networks:

app-network:

driver: bridge

Run using:

docker-compose up --build -d

**9. Deploy to Cloud / CI-CD**

You can push the image to:

* **Docker Hub**:
* docker tag myapp username/myapp:latest
* docker push username/myapp:latest
* **AWS**
* Use in **Kubernetes**, **ECS**, or **EC2** using Docker CLI or Compose

**Summary Workflow Diagram**

Write Code

↓

Create Dockerfile

↓

Build Image → docker build

↓

Create & Run Container → docker run

↓

Container Lifecycle (start, pause, stop, kill, remove)

↓

Docker Networking (bridge, host, overlay, etc.)

↓

Log & Monitor → docker logs, inspect

↓

Cleanup or Deploy to Cloud/Registry

**3-Tier Project Docker Deployment on AWS EC2**

**EC2 Setup**

1. **Create EC2 Instance**
   * OS: Ubuntu
   * Type: t2.medium
   * Storage: 20 GB
   * Inbound Rules: SSH, HTTP, HTTPS, **All TCP - IPv4 Anywhere**
2. **SSH Connection & Project Transfer**
   * Use **WinSCP** to copy frontend and backend folders.
   * Move them into a proper folder:
   * mkdir project
   * sudo mv <project-folder-name> project/
3. **Rename backend JAR**
   * Rename file with meaningful name:
   * mv oldname.jar Quantumsoft.jar

**Docker Installation & Setup**

sudo apt-get update

sudo apt-get install docker.io -y

sudo systemctl status docker # Check Docker service

sudo usermod -aG docker $USER

newgrp docker # Refresh group

docker --version # Check Docker version

**Install Docker Compose**

sudo apt-get install docker-compose -y

sudo apt-get install docker-compose-plugin

# Manual Installation (Latest)

sudo curl -SL https://github.com/docker/compose/releases/download/v2.32.0/docker-compose-linux-x86\_64 -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose

docker-compose --version

**Frontend Setup (Angular)**

**Configurations**

1. Update Config.js with EC2 public IP
2. cd config
3. sudo nano Config.js
4. Create default.conf for NGINX:

server {

  listen 80;

  server\_name ***PUBLICIPADDRESS***;

   # Serve Angular application

   root /usr/share/nginx/html;

   index index.html;

  location / {

      try\_files $uri $uri/ /index.html;

   }

   # Proxy API requests to Spring Boot service

  location /api/ {

      proxy\_pass http://backend:8080;

      proxy\_set\_header Host $host;

      proxy\_set\_header X-Real-IP $remote\_addr;

      proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

      proxy\_set\_header X-Forwarded-Proto $scheme;

   }

   # Error handling for Angular routes

  error\_page 404 /index.html;

   location = /index.html {

      allow all;

   }

}

1. Copy dist files:
2. cp -rf ./\* dist
3. Create Dockerfile for frontend:
4. Build and Push Docker image:
5. docker build -t spquantum/frontend:v1 .
6. docker push spquantum/frontend:v1 .

**Backend Setup**

1. Create Dockerfile for backend:
2. FROM openjdk:17-jdk-slim
3. WORKDIR /app
4. COPY target/Quantumsoft-0.0.1-SNAPSHOT.jar Quantumsoft.jar
5. EXPOSE 8080
6. CMD ["java", "-jar", "Quantumsoft.jar"]
7. Push Docker image:
8. docker build -t spquantum/backend:v1 .
9. docker push spquantum/backend:v1
10. Update application.properties:
11. spring.datasource.url=jdbc:mysql://mysqldb:3306/quantumsoft
12. spring.datasource.username=root
13. spring.datasource.password=root

**Docker Compose File (docker-compose.yml)**

version: '3.4'

services:

backend:

build:

context: ./backend

environment:

SPRING\_DATASOURCE\_URL: jdbc:mysql://mysql-service:3306/quantumsoft

SPRING\_DATASOURCE\_USERNAME: root

SPRING\_DATASOURCE\_PASSWORD: root

ports:

- "8080:8080"

depends\_on:

mysql-service:

condition: service\_healthy

networks:

- mynetwork

frontend:

build:

context: ./frontend

ports:

- "5502:80"

depends\_on:

- backend

networks:

- mynetwork

mysql-service:

image: mysql:8.0

environment:

MYSQL\_ROOT\_PASSWORD: "root"

MYSQL\_DATABASE: "quantumsoft"

MYSQL\_USER: "root"

MYSQL\_PASSWORD: "root"

ports:

- "3306:3306"

volumes:

- mysql-data:/var/lib/mysql

networks:

- mynetwork

healthcheck:

test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]

interval: 10s

retries: 5

start\_period: 30s

networks:

mynetwork:

volumes:

mysql-data:

**Run the Application**

docker-compose up -d # Start containers

docker ps # Check running containers

docker exec -it <container\_id> bash # Enter container

**MySQL Access Inside Container**

mysql -u root -p

SHOW DATABASES;

USE quantumsoft;

SHOW TABLES;

SELECT \* FROM <table\_name>;