**What is Docker?**

Docker is a **tool that packages your application and all its dependencies** (like Java, libraries, database drivers, etc.) into a **container** so it can run consistently on any environment — whether it’s your laptop, a test server, or the cloud.

Think of a **container** as a **lightweight, portable mini-computer** that runs just your application and what it needs — nothing more.

**Benefits:**

* **Environment Consistency**: "It works on my machine" problems go away.
* **Faster Onboarding**: New developers can run the project instantly with docker-compose up.
* **Easy Deployment**: You can deploy the same container from development to production.
* **Isolation**: No interference between apps or dependencies.
* **Supports Microservices**: Each service can run in its own container.

**Key Docker Concepts**

**1. Image**

An **image** is like a template or blueprint. It contains:

* Your application (e.g., a .jar file for Spring Boot)
* The base OS and runtime (e.g., Linux + Java)
* Any commands to run your app

Think: A ready-to-use packaged app.

**2. Container**

A **container** is a **running instance** of an image.

* You can run, stop, restart, and delete containers.
* It behaves like a lightweight computer that runs your app.

**3. Dockerfile**

A Dockerfile is a **text file** with a list of instructions to **build an image**.

For a Java app, it looks like this:

FROM openjdk:17-jdk-alpine # Use a lightweight Java 17 image

COPY target/myapp.jar app.jar # Copy your built app to the container

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

**4. Docker Compose**

**Docker Compose** is a tool that lets you define and run multi-container Docker applications using a single YAML file. Instead of starting each container manually, you can define all your services (like a Spring Boot app and a PostgreSQL database) in one file and bring them up together with a single command: docker-compose up. It also supports networking, volumes (for data persistence), and environment variables—making it ideal for local development and testing.

**5. Docker Volumes (Data Persistence)**

**Problem**: When a container is removed, all the data inside it is lost.  
**Solution**: Docker **volumes** allow us to store data **outside the container**, so the data stays safe even if the container is stopped or deleted.

**6. Docker Networks (Communication Between Containers)**

Docker containers can talk to each other through **networks**.

In docker-compose, a **default network** is created automatically so that your app container can reach the DB using the DB service name.

**Docker in CI/CD (Continuous Integration/Deployment)**

Docker makes **automation easier**. In CI/CD pipelines (like GitHub Actions, Jenkins, GitLab CI):

* Build and test your app in a clean Docker environment
* Package your app into a Docker image
* Push it to **Docker Hub** or **Amazon ECR**
* Deploy it to a **production server or Kubernetes cluster**

**Example Steps in CI/CD:**

1. mvn clean package to build the app.
2. docker build -t my-app .
3. docker push my-app (to Docker Hub or other registry)
4. Server pulls and runs the image with docker run

**Why Docker is Used in Real Projects**

* No more “It works on my machine” issues.
* Easy deployment to cloud or production.
* Works well with microservices (each service in its own container).
* Great for automation in CI/CD pipelines.

**How to Use Docker with a Spring Boot App**

**Step 1: Build your Spring Boot app JAR file**

mvn clean package

**Step 2: Create a Dockerfile in your project root**

FROM openjdk:17-jdk-alpine # Use a base Java image

COPY target/myapp.jar app.jar # Copy the jar file into the container

ENTRYPOINT ["java", "-jar", "app.jar"] # Command to run the app

**Step 3: Build the Docker Image**

docker build -t my-spring-app .

**Step 4: Run the Docker Container**

docker run -p 8080:8080 my-spring-app

This maps your computer’s port 8080 to the container’s port 8080.

**Common Docker Commands**

docker build -t name . Build an image from Dockerfile

docker run -p 8080:8080 name Run a container

docker ps Show running containers

docker stop <id> Stop a container

docker images Show all images

docker rm <id> Remove a container

docker rmi <id> Remove an image

**CMD vs ENTRYPOINT (in Dockerfile)**

Both define the **starting point** of your application in a container, but they behave differently.

| **Feature** | **CMD** | **ENTRYPOINT** |
| --- | --- | --- |
| Purpose | Default command (can be overridden) | Always runs this command |
| Flexibility | Can be changed when running docker run | Harder to override |

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

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

Usually, **ENTRYPOINT** is preferred for apps like Spring Boot.

**Docker vs Virtual Machine**

Feature Docker Virtual Machine

OS Kernel Uses host OS kernel Emulates a full OS

Size Lightweight (MBs) Heavy (GBs)

Speed Starts in seconds Takes minutes to boot

Isolation Process-level OS-level

Portability High Medium

Performance Better (less overhead) Slower (more resource-hungry)

Use Case App deployment, CI/CD, microservices Full OS testing, legacy systems

**Kubernetes (Bonus - Only Basics Needed)**

Docker runs **a single container**, but in real-world systems with many containers, we use a **container orchestration tool** like **Kubernetes (K8s)**.

**Why Kubernetes?**

* Automatically manages multiple containers
* Handles scaling (up/down)
* Performs health checks, restarts containers
* Exposes services to the outside world

You don’t need to master Kubernetes now, but know that:

Docker = runs one or few containers  
Kubernetes = manages 100s or 1000s of containers at scale