Docker Reference Steps

What is Docker?

Docker is a **containerization platform** that allows developers to package applications along with their dependencies into **lightweight, portable containers**. These containers ensure that the application **runs consistently across different environments**, solving the "works on my machine" problem.

After Docker (Containerized Deployment)

How Docker Solves These Problems:

✅

Lightweight Containers: Uses fewer resources than VMs.

✅ **No Dependency Issues**: Runs the same everywhere (dev, test, production).

✅ **Faster Deployment**: Containers start in **seconds**, unlike VMs.

✅ **Scalability**: Easily scales applications using tools like Kubernetes.

✅ **CI/CD Integration**: Automates testing & deployment using Jenkins, GitHub Actions, etc.

🟢 Real-Time Example (After Docker)

➡️ E-Commerce Website Deployment (Docker Approach)

1. Developers write the code and create a **Dockerfile** to package the app (e.g., Java Spring Boot + PostgreSQL).
2. The **Docker image** is built and pushed to **Docker Hub** or a private registry.
3. The image is pulled and deployed on **any server** (cloud, on-premise, or Kubernetes).
4. The application **runs identically in every environment**, avoiding "works on my machine" issues.

5. If a new update is required, **a new container is deployed in seconds**, ensuring zero downtime.

Comparison: Before vs After Docker

Feature	Before Docker (Traditional)	After Docker (With Docker)
Setup Time	Hours/Days (Manual installation, VMs)	Minutes (Pre-built Docker images)
Resource Usage	Heavy (VMs require full OS, GBs of RAM)	Lightweight (Uses OS kernel, MBs of RAM)
Portability	Not portable (OS dependencies)	Fully portable (Runs anywhere)
Deployment Speed	Slow (manual setup, VM boot time)	Fast (containers start in seconds)
Scalability	Hard (new VM needed for every instance)	Easy (Kubernetes can auto-scale containers)

Before Docker vs After Docker (Advantages & How It Overcame Issues)

Aspect	**Before Docker** (Traditional Deployment)	**After Docker** (With Docker)
Environment Issues	"Works on my machine" problem due to different OS, dependencies, or configurations.	Runs the same everywhere (development, testing, production).
Dependency Management	Manually install dependencies on every machine.	All dependencies are inside the container, no manual setup needed.
Resource Utilization	Heavy Virtual Machines (VMs) used, consuming more CPU/RAM.	Lightweight containers share the OS kernel, reducing resource usage.
Deployment Speed	Slow, manual deployments with configuration errors.	Fast deployments using Docker images and orchestration tools (Kubernetes, Docker Swarm).

Isolation	Conflicts between different applications on the same server.	Each app runs in an isolated container, avoiding conflicts.
Portability	Difficult to move apps between different servers.	Containers run on any machine with Docker installed.
Scaling	Requires setting up new VMs, slow process.	Easily scale using multiple containers (`docker-compose` , Kubernetes).
CI/CD Integration	Manual deployment process, prone to errors.	Automated builds and deployments using Docker + CI/CD (Jenkins, GitHub Actions).

How Containers Exchange Data and Communicate?

In a Dockerized environment, containers can communicate with each other

How Testing Teams Run the Same Container as Developers?

When a developer creates a Docker container, the same container should be **reproducible** for the testing team without dependency issues. This is done through **Docker images and container orchestration**.

✅ Step-by-Step Process for Developers & Testers

1️⃣ Developer Creates a Docker Image

After developing an application, the developer **creates a Docker image** that can be shared.

Example: Creating a Spring Boot Image

1. **Write a Dockerfile**

```
dockerfile
CopyEdit
FROM openjdk:17
WORKDIR /app
```

```
COPY target/my-app.jar my-app.jar
ENTRYPOINT ["java", "-jar", "my-app.jar"]
```

2. **Build the Docker Image**

   ```
   docker build -t my-app:latest .
   ```

3. **Push to Docker Hub or Private Registry**

   ```
   docker tag my-app:latest my-dockerhub-username/my-app:latest
   docker push my-dockerhub-username/my-app:latest
   ```

2️⃣ Testers Pull & Run the Same Container

Now, the testing team can pull the same image and **run the application in a container**.

Steps for Testers

1. **Pull the Image**

   ```
   docker pull my-dockerhub-username/my-app:latest
   ```

2. **Run the Container**

   ```

   docker run -d --name test-container -p 8080:8080 my-dockerhub-username/my-app:latest
   ```

Now, testers can access the application on **http://localhost:8080**.

3️⃣ Using Docker Compose for Consistency

If multiple containers (e.g., app + database) are required, developers create a **docker-compose.yml** file, and testers can use it.

Example: Spring Boot + PostgreSQL for Testing Team

What is Version `3.0` in Docker Compose?

- **Version 3.0** was introduced in **Docker Compose v1.13.0** (early 2017).
- It was mainly designed for **Docker Swarm mode** (for multi-container orchestration).
- It included **basic service definitions**, networking, and volume support.

Differences Between `3.0` and `3.8`

Feature	Version `3.0`	Version `3.8`
Swarm Mode Support	✅ Yes	✅ Yes
depends_on condition	❌ No	✅ Yes
Healthcheck Support	❌ No	✅ Yes
Resource Limits	✅ Yes (limited)	✅ Yes (improved)
Extensions for Swarm	❌ No	✅ Yes

Developer Creates docker-compose.yml

```
version: '3.8'
services:
  app:
    image: my-dockerhub-username/my-app:latest
    ports:
      - "8080:8080"
    depends_on:
      - db

  db:
    image: postgres:15
    environment:
      POSTGRES_USER: test
      POSTGRES_PASSWORD: test123
      POSTGRES_DB: mydb
```

Testers Run the Application with One Command

```
docker-compose up -d
```

This will start **both the app and database** exactly as the developer intended.

4️⃣ Automating with CI/CD (Jenkins, GitHub Actions)

To ensure **continuous integration**, the Docker image can be built & deployed **automatically**.

✅ Example Workflow:

1. Developer **pushes code** to GitHub.
2. **CI/CD pipeline builds Docker image** and pushes it to Docker Hub.
3. Testing team **pulls the latest image** and runs tests.

🔷 Summary: How Testers Run the Same Container

Scenario	Solution
Developer creates the image	`docker build -t my-app .`
Developer shares the image	`docker push my-app`
Testing team runs the same app	`docker pull my-app && docker run -p 8080:8080 my-app`
Multi-container setup	`docker-compose up -d`
Automating Testing	CI/CD (Jenkins, GitHub Actions)

Docker Database Steps

You can check the database inside a **running MySQL container** in Docker by following these steps:

🔷 Step 1: Get the Running MySQL Container Name

Run:

CMD-:   docker ps

Look for your MySQL container (e.g., `mysql` in your `docker-compose.yml`).

🔷 Step 2: Access MySQL Container's Shell

Run:

```
sh

docker exec -it mysql bash
```

This will open a terminal inside the MySQL container.

🔷 Step 3: Log in to MySQL

Once inside the container, log in to MySQL using the root user:

```
sh

mysql -u root -p
```

Enter your **MySQL root password** (e.g., `dockerPassword` from your `docker-compose.yml`).

🔷 Step 4: List Databases

After logging in, see all available databases:

```
sql

SHOW DATABASES;
```

🔷 Step 5: Use Your Database

Select your database (e.g., `docker` from your `docker-compose.yml`):

```
sql

USE docker;
```

🔷 Step 6: List All Tables

Run:

```
sql

SHOW TABLES;
```

🔷 Step 7: See Data Inside a Table

To check how many records are inside a table, use:

```
sql

SELECT COUNT(*) FROM your_table_name;
```

To see all records in a table:

```
sql

SELECT * FROM your_table_name;
```

🔷 Step 8: Exit MySQL

Once done, exit MySQL:

```
sql

exit;
```

Then, exit the container:

```
sh

exit
```

🔷 One-Line Command to Connect to MySQL

If you don’t want to go step by step, run this **one-line command** from your terminal:

```
sh
docker exec -it mysql mysql -u root -p
```

This will directly open MySQL, and you just need to enter the password.

🔷 Example Output

```
sql

mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| docker             |
| information_schema |
| mysql              |
| performance_schema |
+--------------------+

mysql> USE docker;

mysql> SHOW TABLES;
+------------------+
| Tables_in_docker |
+------------------+
| users            |
| orders           |
| products         |
+------------------+

mysql> SELECT COUNT(*) FROM users;
+----------+
| COUNT(*) |
+----------+
|       10 |
+----------+
```