**Trainer: Rajendra K**

**Basics**

Virtual Machine

Container Evolution

Container Runtime Engine

Installation and Configurations

Architecture

Docker basics - create , delete, view, list

**Image Management and Container Registry**

Image basics and commands

Containers

Service

Dockerfile and instructions

Image Layers

Multistage Dockerfiles

Tagging an Image

Image registry and repository

Inspect , remove and prune images

**Networking**

Basics

Architecture

NetworkTypes: None, Host, Bridge, Overlay

Prune networks

Remove networks

Connect/disconnect networks

Ports: Identifying and publishing

Traffic: Ingress & Egress

**Docker-Storage**

Docker inspect

Docker compose: create, update, delete

**Orchestration**

Service and Types

Tasks

Docker Swarm

Nodes: Managers and Workers

Node Management: Draining

Scaling

Volumes: Binding, Creating, mounting

Logs: troubleshooting and Debugging

Node backup and restore

**Docker Security**

**Microservices architecture**

**Course Plan**

Day 1: Basics, Architecture, Microservices basics

Day 2: Image Management and COntainer Registry

Day 3: Networking & Docker-Compose

Day 4 Orchestration

Day 5: Docker Security and Project

Day1:

## **✅ Objective:**

Run a basic Flask web app using Docker.

## **🔧 Step-by-Step Instructions:**

### **1. Create and navigate to a new directory**

mkdir example

cd example

* mkdir example: Creates a new directory named example.
* cd example: Changes into that directory.

### **2. Create the Flask app file**

vi app.py

In the vi editor, press i to insert the following code:

from flask import Flask

import os

app = Flask(\_\_name\_\_)

@app.route('/')

def hello():

return ('\nHello from Container World! \n\n')

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host="0.0.0.0", port=8080, debug=True)

Then save and exit:

* Press ESC, then type :wq and hit Enter.

### **3. Create the Dockerfile**

vi Dockerfile

Insert this content:

FROM ubuntu:22.04

RUN apt update && apt install python3 -y && apt install python3-flask -y

COPY app.py /tmp

EXPOSE 8080

CMD ["python3", "/tmp/app.py"]

Explanation:

* FROM ubuntu:22.04: Use Ubuntu 22.04 as the base image.
* RUN apt update && apt install python3 -y && apt install python3-flask -y: Install Python 3 and Flask.
* COPY app.py /tmp: Copy your app code into the container.
* EXPOSE 8080: Document that the app uses port 8080.
* CMD ["python3", "/tmp/app.py"]: Command to run the Flask app when the container starts.

Save and exit vi the same way.

### **4. Check available Docker images**

docker images

Lists all Docker images currently on your system.

### **5. Build your Docker image**

docker build -t first:1.0 .

* -t first:1.0: Tags the image with the name first and version 1.0.
* .: Tells Docker to use the current directory (with your Dockerfile and app) to build the image.

### **6. Run the container**

docker run -d --name mycontainer -p 8000:8080 first:1.0

* -d: Runs the container in detached (background) mode.
* --name mycontainer: Names the container.
* -p 8000:8080: Maps your host's port 8000 to the container's port 8080.
* first:1.0: Uses the image you just built.

### **7. Test the application**

curl localhost:8000

* This sends a request to your Flask app via port 8000.
* You should see the message: Hello from Container World!

### **8. View container logs**

docker logs mycontainer

* Shows the output (like Flask startup messages) from the container.

### 

### **🔍 Monitoring & Container Info**

t

46 docker stats

* Shows a live stream of resource usage (CPU, memory, network I/O, etc.) for running containers.

47 docker ps

* Lists **only running** containers.

48 docker ps -a

* Lists **all containers**, including those that are stopped or exited.

### **📜 Logs & Lifecycle**

49 docker logs mycontainer

* Displays logs (stdout/stderr) from the container named mycontainer.

50 docker stop mycontainer

* Gracefully stops the container.

51 docker ps

* Again lists only currently running containers (to confirm it stopped).

52 docker ps -a

* Shows all containers, including mycontainer (which should now be in an exited state).

53 docker rm mycontainer

* Removes the container named mycontainer (make sure it’s stopped before removing).

54 docker ps -a

* Confirms that the container is gone.

### **🚀 Running & Accessing Container**

55 docker run -d --name mycontainer -p 8000:8080 first:1.0

* Re-runs your container (same command as earlier).

56 docker exec -it mycontainer bash

* Opens an interactive terminal (bash) **inside** the running container. Super useful for debugging or inspecting.

### **🧹 Image Management**

57 docker images

* Lists all Docker images on your system.

58 docker rmi 78a549e51333

* Removes the image with ID 78a549e51333.  
   Make sure **no container is using** the image when trying to remove it.

## **🌐 Step 1: Create a DockerHub Account**

1. Go to: https://hub.docker.com
2. Click on **"Sign Up"** (top-right).
3. Fill in:  
   * **Username** (this becomes your DockerHub ID, e.g., rajendrait99)
   * **Email address**
   * **Password**
4. Agree to the terms and hit **"Sign Up"**.
5. You may need to verify your email — check your inbox.

## **🐳 Step 2: Tag and Push Docker Image to DockerHub**

Now that your account is ready, here are your commands explained:

### **🔍 Check your existing images**

60 docker images

* Lists all images available on your local system.

### **🏷️ Tag your local image**

61 docker tag first:1.0 rajendrait99/first:1.0

* **docker tag**: Creates a new "alias" for your image.
* first:1.0 = local image name and version.
* rajendrait99/first:1.0 = DockerHub-style name (with your username as prefix).
* Docker needs this format to know where to push the image.

You can confirm it was tagged:

62 docker images

### **🔐 Login to DockerHub**

63 docker login

* Prompts for your DockerHub **username** and **password**.
* On success, stores credentials locally (so you don’t need to log in every time).

### **☁️ Push your image to DockerHub**

64 docker push rajendrait99/first:1.0

* Uploads your image to your DockerHub repository.
* It will be publicly visible by default (unless you change the visibility settings on DockerHub).

## **✅ To Verify:**

* Visit: https://hub.docker.com/repository/docker/rajendrait99/first
* You should see your image listed there.

## **🧹 Step-by-Step Explanation: Docker Cleanup & Pull from DockerHub**

### **🔍 66. List All Containers**

docker ps -a

* Lists all containers (running or stopped).
* Helps confirm container names before stopping/removing.

### **🛑 67–68. Stop a Container**

docker stop mycontainer

* You had a typo in the first command (myconatiner) — corrected in the second one.
* This stops a running container gracefully.

### **🗑️ 69. Remove the Container**

docker rm mycontainer

* Deletes the stopped container. You must stop it first before removing.

### **📷 70. List All Images**

docker images

* Shows all local Docker images (including ones no longer used).

### **🧼 71–72. Remove an Image**

docker rmi 78a549e51333

docker rmi 78a549e51333 -f

* First command tries to remove image with ID 78a549e51333.
* Second command (-f) **forces** the removal (e.g., if the image is still in use or cached).
* You may need to remove containers using the image before it can be deleted.

### **🔍 73. Confirm Image Deletion**

docker images

* Confirms the image was successfully removed from local storage.

### **☁️ 74. Pull from DockerHub**

docker pull rajendrait99/first:1.0

* Downloads the image back from your DockerHub repo.
* Useful for deploying on another system or testing the cloud-stored image.

### **📷 75. Confirm Pull**

docker images

* Ensures the image is now back in your local image list.

### **🚀 76. Run the Pulled Image**

docker run -d --name mycontainer -p 8000:8080 rajendrait99/first:1.0

* Runs your DockerHub image just like a local one:  
  + -d: Run detached
  + --name: Names the container
  + -p 8000:8080: Maps host port to container's Flask port
  + rajendrait99/first:1.0: Image from DockerHub

## **✅ Outcome:**

You've:

* Cleaned up local containers and images.
* Pulled your image from DockerHub.
* Verified it runs just like before — but now from the cloud!

## **🏠 Hosting a Local Private Docker Registry**

### **🚀 81. Run a Local Docker Registry**

docker run -d --name private -p 8000:5000 registry:2

* Starts the official **Docker Registry** container (version 2).
* -p 8000:5000 maps your machine's port 8000 to the registry's internal port 5000.
* The registry will now listen at http://localhost:8000.

### **📷 82. Check Images**

docker images

* Lists current images, including registry:2.

### **📦 83. Check Running Containers**

docker ps

* Confirms that the private registry is running.

### **🧭 84. Check Registry Catalog**

curl localhost:8000/v2/\_catalog

* Queries the local registry to list available repositories (images).

Initially, this should return:  
  
  
{"repositories":[]}

* because nothing is uploaded yet.

### **🏷️ 86. Tag an Image for Local Registry**

docker tag rajendrait99/first:1.0 localhost:8000/first:1.0

* Tags your existing image to prepare it for upload to your private registry.
* The format localhost:8000/first:1.0 tells Docker where to push it.

### **📷 87. Check Tagged Image**

docker images

* Shows the new tag pointing to your local registry.

### **☁️ 88. Push to Private Registry**

docker push localhost:8000/first:1.0

* Uploads the image to your local registry.
* You should now see a response showing layer upload progress.

### **📦 89. Recheck Registry Catalog**

curl localhost:8000/v2/\_catalog

Now you should get:  
  
{"repositories":["first"]}

* This confirms your image was successfully pushed to your private registry.

### **📥 90. Pull Image from Private Registry**

docker pull localhost:8000/first:1.0

* Pulls the image back from your local registry.
* Useful to test or deploy it on other hosts (that can access your registry).

## **✅ Result:**

You’ve successfully:

* Hosted a local private Docker registry.
* Tagged and pushed your image to it.
* Verified the registry contains your image.
* Pulled the image back locally.

**Day2**

# **Full Step-by-Step with Layer Management Explained:**

### **1. Move into your project directory**

cd example/

(You should have app.py and the new Dockerfile inside.)

### **2. List the files**

ls

✅ You should see:

app.py Dockerfile

### **3. Build the Docker image**

docker build -t first:1.0 .

➡️ **Docker creates layers like this**:

* Layer 1: FROM ubuntu:22.04 → Base Ubuntu image (large layer ~29MB compressed)
* Layer 2: RUN apt update && apt install python3 -y && apt install python3-flask -y  
   → Installs Python3 and Flask inside the container (this adds **another big layer**)
* Layer 3: COPY app.py /tmp → Adds your application code as a separate small layer.
* Layer 4: EXPOSE 8080 → Metadata layer (no file changes).
* Layer 5: CMD ["python3", "/tmp/app.py"] → Metadata layer (defines the default command).

✅ **Each instruction = separate layer.**

### **4. Check images locally**

docker images

You will see:

* first:1.0 listed, along with its IMAGE ID and SIZE.

Note: **Image size will be bigger** compared to python-slim because you installed Python manually on Ubuntu.

### **5. Rebuild the image (Optional)**

bash

CopyEdit

docker build -t first:1.0 .

✅ Since nothing changed in Dockerfile or app.py,  
 Docker uses **cached layers** and skips redoing installation.

**Benefit:** Speeds up rebuild dramatically!

### **6. Tag the image for Docker Hub**

docker tag first:1.0 rajendrait99/first:1.0

✅ **No layers recreated.** Only another name (tag) is assigned to same image.

### **7. Login to Docker Hub**

docker login

Authenticate your credentials.

### **8. Push your image to Docker Hub**

docker push rajendrait99/first:1.0

✅ **Layer Upload Logic:**

* Docker uploads layers one by one.
* Already existing layers (like ubuntu:22.04) might not need re-upload.

### **9. (Optional) Push Again**

docker push rajendrait99/first:1.0

✅ Docker detects all layers are already on Hub — **nothing new is uploaded**.

### **10. Pull your image (from any other machine)**

docker pull rajendrait99/first:1.0

✅ Docker downloads missing layers only. If Ubuntu 22.04 base image already exists locally, only the additional app layers are pulled.

# **📦 Docker Image Layer View (for your case)**

| **Layer Number** | **Dockerfile Instruction** | **Purpose** |
| --- | --- | --- |
| Layer 1 | FROM ubuntu:22.04 | Ubuntu base system |
| Layer 2 | RUN apt update && apt install python3 python3-flask | Install Python3 and Flask |
| Layer 3 | COPY app.py /tmp | Add your application code |
| Layer 4 | EXPOSE 8080 | Inform container runtime to open port |
| Layer 5 | CMD ["python3", "/tmp/app.py"] | Define default container startup |

# **🧠 What You Are Demonstrating about Layer Management:**

* **Every Dockerfile instruction = a separate layer**.
* **Layer caching**: Only changes trigger new builds.
* **Tagging** doesn't create new layers.
* **Pushing** reuses layers — only new ones are uploaded.
* **Pulling** reuses existing local layers — speeds up setup.
* **Choosing base images (ubuntu:22.04)** affects image size and speed.

**Docker Volume:**

### **Part 1: Demonstrating Ephemeral Data (Without Volumes)**

**In this section, we will create a container, write a file inside it, then remove the container to show that the file is lost.**

* **166 docker run -d --name first -p 8000:8080 first:1.0**
  + **Explanation: This command starts a new Docker container:**
    - **-d: Runs the container in detached mode (in the background).**
    - **--name first: Assigns the name first to this container for easy reference.**
    - **-p 8000:8080: Maps port 8000 on your host machine to port 8080 inside the container. This would typically be used to access an application running on port 8080 within the container from your host's port 8000.**
    - **first:1.0: Specifies the Docker image to use. This assumes you have an image named first with tag 1.0 (e.g., a simple web server or application).**
  + **Expected Output: A long hexadecimal string (container ID) indicating the container has started.**
  + **Purpose: To launch our sample container.**
* **167 docker ps**
  + **Explanation: This command lists all running Docker containers on your system.**
  + **Expected Output: You should see the first container listed, confirming it's running in detached mode.**
  + **Purpose: To verify the container is active.**
* **168 docker exec -it first bash**
  + **Explanation: This command allows you to execute commands inside a running container:**
    - **docker exec: Executes a command in a running container.**
    - **-it: Stands for --interactive and --tty. This allocates a pseudo-TTY and keeps STDIN open, allowing you to interact with the bash shell inside the container.**
    - **first: Specifies the name of the container where the command will be executed.**
    - **bash: The command to execute, which opens a bash shell inside the container.**
  + **Action Inside Container: Once inside the container's bash shell, you would run:**
    - **echo "abc test data" > abc.txt**
      * **Explanation: This command creates a new file named abc.txt in the current directory within the container and writes the text "abc test data" into it.**
    - **ls (Optional, to confirm file creation)**
  + **Expected Output: You will get a command prompt inside the container. After running echo and ls, you'll see abc.txt listed.**
  + **Purpose: To create a new file within the container's writable layer that we will later check for persistence.**
* **169 docker stop first**
  + **Explanation: This command sends a SIGTERM signal to the first container, requesting it to shut down gracefully.**
  + **Expected Output: The container name (first) will be displayed, indicating it's being stopped.**
  + **Purpose: To stop the container before removing it.**
* **170 docker rm first first & 171 docker rm first**
  + **Explanation: (Line 170 seems to be a repeated argument, docker rm first is sufficient). This command removes one or more stopped containers.**
    - **docker rm: Removes a container.**
    - **first: The name of the container to remove.**
  + **Expected Output: The container name (first) will be displayed, indicating it has been removed.**
  + **Observation/Crucial Point: After this step, the container first is completely deleted. Any data that was created directly within the container's writable layer (like abc.txt) is now permanently lost. This demonstrates the ephemeral nature of container data without explicit persistence mechanisms.**
  + **Purpose: To fully delete the container and illustrate data loss.**

### **Part 2: Demonstrating Data Persistence (With Volumes)**

**Now, we will recreate the container, attach a Docker volume to it, write the same file to the volume-mounted path, and then demonstrate that the file persists even after the container is removed and recreated.**

* **172 docker run -d --name first -p 8000:8080 first:1.0**
  + **Explanation: Re-creates a *new* first container, identical to the initial one (no abc.txt inside it yet).**
  + **Expected Output: A new container ID.**
  + **Purpose: To have a fresh container for our volume demonstration.**
* **173 docker exec -it first bash**
  + **Explanation: Gets an interactive shell into the newly created first container. (We won't create a file here yet, as we need to set up the volume first).**
  + **Expected Output: A command prompt inside the container.**
  + **Type ‘’’exit’’’ to come out of container**
* **174 docker volumes**
  + **Explanation: This is likely a typo. The correct command to list Docker volume commands or get help is docker volume or docker volume --help.**
  + **Expected Output: Docker CLI help text for docker volume commands.**
  + **Purpose: To check the syntax or available options for volume management.**
* **175 docker volume ls**
  + **Explanation: Lists all Docker volumes currently managed by Docker on your host machine.**
  + **Expected Output: A table showing existing volumes (initially, it might be empty or show default anonymous volumes).**
  + **Purpose: To see what volumes exist before creating a new one.**
* **176 docker volume create myvolume**
  + **Explanation: This command creates a new named volume on your Docker host machine. Docker manages where this volume is physically stored.**
    - **myvolume: The name assigned to the new volume.**
  + **Expected Output: The name of the created volume (myvolume) will be displayed.**
  + **Purpose: To prepare a persistent storage location for our container data.**
* **177 docker volume ls**
  + **Explanation: Lists all Docker volumes again to confirm that myvolume has been successfully created and is listed.**
  + **Expected Output: You should see myvolume listed in the output.**
  + **Purpose: To verify the volume creation.**
* **178 ls /var/lib/docker/**
* **179 sudo ls /var/lib/docker/**
* **180 sudo ls /var/lib/docker/volumes**
  + **Explanation: These commands are used to inspect the underlying directory structure where Docker stores its data, including volumes, on a Linux host.**
    - **/var/lib/docker/: This is the default root directory for Docker data on Linux.**
    - **/var/lib/docker/volumes: This specific subdirectory is where named volumes are physically stored.**
    - **sudo: Often required to access these system directories due to permissions.**
  + **Expected Output: You should be able to see a directory corresponding to myvolume (e.g., myvolume/\_data) within /var/lib/docker/volumes, confirming its physical presence on the host.**
  + **Purpose: To visualize where Docker volumes reside on the host filesystem.**
* **181 docker stop first && docker rm first**
  + **Explanation: Stops and removes the first container created in step 172. This is to ensure a clean state before launching a new container with the volume. The volume myvolume itself remains untouched.**
  + **Expected Output: Confirmation that the container is stopped and removed.**
  + **Purpose: Clean up previous container instance.**
* **182 docker run -d --name first -p 8000:8080 -v myvolume:/etc/lala first:1.0**
  + **Explanation: This is the most crucial command for demonstrating persistence with a volume:**
    - **docker run ...: Starts a new container as before.**
    - **-v myvolume:/etc/lala: This is the volume mount option.**
      * **myvolume: The name of the Docker volume on the host that we created earlier.**
      * **:/etc/lala: The absolute path *inside the container* where myvolume will be mounted. Any data written to /etc/lala within this container will actually be written to myvolume on your host.**
  + **Expected Output: A new container ID, indicating the container with the mounted volume is running.**
  + **Purpose: To launch a container with a persistent volume mounted.**
* **183 docker exec -it first bash**
  + **Explanation: Gets an interactive shell into the first container (which now has myvolume mounted).**
  + **Action Inside Container:**
    - **cd /etc/lala**
      * **Explanation: Navigate to the directory inside the container where the volume is mounted.**
    - **echo "abc test data" > abc.txt**
      * **Explanation: Create the file abc.txt within the mounted volume path.**
    - **ls (Optional, to confirm file creation)**
  + **Expected Output: You'll be in the container's bash shell. After cd and echo, you'll see abc.txt listed in /etc/lala.**
  + **Purpose: To write data into the volume-mounted path.**
* **184 docker stop first && docker rm first**
  + **Explanation: Stops and removes the first container that was running with the volume.**
  + **Expected Output: Confirmation that the container is stopped and removed.**
  + **Crucial Point: The file abc.txt is NOT lost here. Because it was written to the myvolume (which resides on the host and is managed by Docker independently of the container's lifecycle), the data persists.**
  + **Purpose: To prove that data written to a volume survives container removal.**
* **185 docker run -d --name first -p 8000:8080 -v myvolume:/etc/lala first:1.0**
  + **Explanation: Re-creates *another* new container. Importantly, it mounts the *same* myvolume to the same path /etc/lala.**
  + **Expected Output: A new container ID.**
  + **Purpose: To launch a fresh container and connect it to the *existing* persistent data.**
* **186 docker exec -it first bash**
  + **Explanation: Gets an interactive shell into this latest container.**
  + **Action Inside Container:**
    - **cd /etc/lala**
    - **ls**
      * **Explanation: List the contents of the /etc/lala directory.**
  + **Expected Output: You will see abc.txt listed!**
  + **Observation/Result: This clearly demonstrates that the abc.txt file persisted across the stopping and removal of the previous container instance. The data stored on myvolume was readily available to the new container.**
  + **Purpose: To confirm data persistence through volume re-attachment.**
* **187 docker volume ls**
  + **Explanation: Lists all Docker volumes again.**
  + **Expected Output: myvolume will still be listed.**
  + **Purpose: To show that the volume itself remains even after containers that use it are removed, ready to be attached to new containers.**

### **Conclusion on Docker Volumes**

**This sequence of commands effectively illustrates:**

1. **Ephemeral Container Data: Without volumes, data written directly to a container's writable layer is lost when the container is removed.**
2. **Data Persistence with Volumes: Docker volumes provide a managed, persistent storage solution. Data written to a mounted volume resides on the host system and is independent of the container's lifecycle, allowing data to persist across container removals and recreations.**

## **💾 Using Docker Volumes with a Local Registry**

### **🔄 93. Check Registry Catalog**

curl localhost:8000/v2/\_catalog

* Confirms the current state of the registry — should still list previously pushed images unless the registry was restarted without persistent storage.

### **🛑🗑️ 94–95. Stop and Remove Registry Container**

docker stop private

docker rm private

* Stops and removes the running registry container.
* At this point, since no volume was attached earlier, the pushed images (like first:1.0) are lost.

### **🔄 96–97. Restart Registry and Check Catalog Again**

docker run -d --name private -p 8000:5000 registry:2

curl localhost:8000/v2/\_catalog

* Starts a new instance of the registry, **but without volume mapping**.

The catalog will return:  
  
{"repositories":[]}

* This proves that the image history was lost due to no persistent storage.

## **🔧 Create and Use a Volume for Persistence**

### **📦 98–100. Create and List Volumes**

docker volume ls

docker volume create myvolume

docker volume ls

* Lists existing volumes.
* Creates a new Docker-managed volume named myvolume.

### **🗂️ 101. Check Volume Path on Host**

ls /var/lib/docker/volumes/

* Shows where Docker stores volumes on the host file system.
* myvolume will have its own subdirectory under this path.

### **🛑🗑️ 102–103. Stop and Remove Registry (Again)**

docker stop private

docker rm private

* Clean stop before re-creating the registry with volume attached.

### **📦 104. Run Registry with Volume Attached**

docker run -d --name private -p 8000:5000 -v myvolume:/var/lib/registry registry:2

* This is key: It maps myvolume to the registry's internal data directory (/var/lib/registry).
* Now the image data will **persist** even if the container is removed.

### **📷 105. List Images**

docker images

* Just a check, to ensure your tagged image localhost:8000/first:1.0 is still available for pushing.

### **☁️ 106. Push Image to Registry**

docker push localhost:8000/first:1.0

* Pushes the image again, now into a **persistent registry**.

### **🔄 107–109. Stop, Remove, and Recreate Registry**

docker stop private

docker rm private

docker run -d --name private -p 8000:5000 -v myvolume:/var/lib/registry registry:2

* The registry is removed and recreated.
* But since it's using myvolume, the data (images) should persist.

### **✅ 110. Check Registry Catalog Again**

curl localhost:8000/v2/\_catalog

You should now get:  
  
{"repositories":["first"]}

* 🎉 Success! Your image is still there, even after stopping/removing the container.

## **🧠 What You Learned**

✅ Pushing images to a **local private Docker registry** ✅ Why persistence matters (no volume = data loss)  
 ✅ How to use Docker **volumes** to retain registry data  
 ✅ How to confirm success with curl and \_catalog

# **📜 Full Step-by-Step with CMD vs ENTRYPOINT**

## **1️⃣ First Phase — Using CMD Only**

**Dockerfile (for test:1.0)**:

FROM ubuntu:20.04

CMD ["echo", "Hello World"]

**What happens:**

* **CMD** sets the **default command** to run.
* But when running the container, you can **override CMD easily**.

**Commands you ran**:

mkdir example2

cd example2

vi Dockerfile # added CMD version

docker build -t test:1.0 .

docker run -d --name test test:1.0

docker logs test

✅ Output: Hello World

👉 **Docker uses CMD and runs echo Hello World.**

Then you overrode CMD:

docker rm test

docker run -d --name test test:1.0 echo "Hello India"

docker logs test

✅ Output: Hello India

👉 **Here, CMD is fully replaced with your custom command** (echo Hello India).  
 ➡️ Docker ignores the CMD when you pass your own command during docker run.

## **2️⃣ Second Phase — Using ENTRYPOINT Only**

**Dockerfile (for test:2.0)**:

Dockerfile

FROM ubuntu:20.04

ENTRYPOINT ["echo", "Hello World"]

**What happens:**

* **ENTRYPOINT** fixes the command to echo Hello World.
* You **cannot replace ENTRYPOINT easily**; only **additional arguments** are appended.

**Commands you ran**:

vi Dockerfile # updated ENTRYPOINT

docker build -t test:2.0 .

docker rm test

docker run -d --name test test:2.0

docker logs test

✅ Output: Hello World

👉 **ENTRYPOINT runs exactly as written.**

Now testing with custom input:

docker rm test

docker run -d --name test test:2.0 "Hello India"

docker logs test

✅ Output: Hello World Hello India

👉 **Docker keeps ENTRYPOINT (echo Hello World) and adds "Hello India"** as an argument.  
 Resulting command run inside container:

echo Hello World Hello India

## **3️⃣ Third Phase — Using ENTRYPOINT + CMD Together**

**Dockerfile (for test:3.0)**:

FROM ubuntu:20.04

ENTRYPOINT ["echo"]

CMD ["Hello World"]

**What happens:**

* **ENTRYPOINT** sets the executable: echo
* **CMD** provides default arguments: "Hello World"
* If you provide new arguments during docker run, they **replace CMD** (but not ENTRYPOINT).

**Commands you ran**:

vi Dockerfile # updated to ENTRYPOINT + CMD

docker build -t test:3.0 .

docker rm test

docker run -d --name test test:3.0

docker logs test

✅ Output: Hello World

👉 **ENTRYPOINT is echo** and **CMD supplies default text Hello World**.

Now testing with override:

docker rm test

docker run -d --name test test:3.0 "Hello India"

docker logs test

✅ Output: Hello India

👉 **CMD gets replaced**, so final command becomes:

echo Hello India

# **📦 Final Behavior Table**

| **Version** | **Dockerfile Setup** | **Run Behavior (no args)** | **Run Behavior (with args)** |
| --- | --- | --- | --- |
| test:1.0 | CMD only (CMD ["echo", "Hello World"]) | echo Hello World | Your custom command fully overrides CMD |
| test:2.0 | ENTRYPOINT only (ENTRYPOINT ["echo", "Hello World"]) | echo Hello World | ENTRYPOINT stays; new args are appended |
| test:3.0 | ENTRYPOINT + CMD (ENTRYPOINT ["echo"] + CMD ["Hello World"]) | echo Hello World | ENTRYPOINT stays; CMD is replaced with new args |

# **🎯 In Short:**

| **Concept** | **CMD** | **ENTRYPOINT** |
| --- | --- | --- |
| Definition | Default command to run | Fixed executable to run |
| Overridable? | Fully overridden during run | Command fixed, only args vary |
| Use case | Provide default commands | Ensure container always runs specific executable |

# **🎯 Quick Analogy**

| **CMD** | **ENTRYPOINT** |
| --- | --- |
| Like a suggestion: "if you don't say anything, run this" | Like a strict instruction: "always run this" |

# **🔥 Very Important Real-World Tip:**

* Combine **ENTRYPOINT + CMD** when you want **fixed app behavior** but allow users to change **input parameters** easily.

Example (common practice):

ENTRYPOINT ["python3"]

CMD ["app.py"]

Here you can run the container normally, or override app.py with another script easily!

**DAY3**

# **📚 Full Explanation — Docker Networking Steps**

## **📍 Step 86: List Docker Networks**

docker network ls

**Meaning**:  
 Lists all existing Docker networks.

You typically see:

* bridge (default network for containers)
* host (shares host network)
* none (no network)

## **📍 Step 87: Check Host IP Interfaces**

ifconfig

**Meaning**:  
 Shows network interfaces on your host system (like eth0, docker0, etc).

When Docker is installed, you will notice a special bridge called:

* docker0

✅ **docker0** acts like a virtual switch — Docker connects containers to this by default.

## **📍 Step 88: List Docker Images**

docker images

✅ Just checking what images are available — your first:1.0 image is ready.

## **📍 Step 89: Run First Container (with Port Mapping)**

docker run -d --name first -p 8000:8080 first:1.0

**Meaning**:

* Run a container called first in **detached** mode (-d).
* Map **host port 8000 → container port 8080** (-p 8000:8080).
* Use image first:1.0.

✅ **Port mapping** means when you access localhost:8000 on host, traffic is forwarded to container port 8080.

## **📍 Step 90: Check Interfaces Again**

ifconfig

**Meaning**:  
 After running a container, Docker may dynamically update some virtual interfaces (inside docker0).

## **📍 Step 94: Inspect Bridge Network**

docker network inspect bridge

**Meaning**:  
 View detailed information about the bridge network:

* Subnet (e.g., 172.17.0.0/16)
* Gateway
* Connected containers (your first container is now listed).

✅ **Default behavior**: containers get IP addresses from the bridge subnet.

## **📍 Step 95: Run Second Container**

docker run -d --name second -p 8001:8080 first:1.0

**Meaning**:  
 Start another container:

* second container
* Port mapping: **8001 on host → 8080 in container**

✅ Now both containers (first, second) are attached to the same bridge network, but using different ports.

## **📍 Step 96: Check Interfaces Again**

ifconfig

You’ll notice:

* No new interfaces created (still under docker0).
* Just additional containers added to existing bridge.

## **📍 Step 97: Inspect Bridge Network Again**

docker network inspect bridge

✅ Now both containers (first and second) are visible inside the bridge network.

## **📍 Step 98: List Networks Again**

docker network ls

✅ Still see bridge, host, none.

## **📍 Step 99: Create a Custom Network**

docker network create mynet --subnet=192.168.0.0/16

**Meaning**:

* Create a **new bridge network** called mynet.
* Subnet assigned is 192.168.0.0/16.

✅ Custom networks allow better control over:

* Subnets
* IP ranges
* DNS resolution between containers.

## **📍 Step 100: Inspect Custom Network**

docker network inspect mynet

✅ Check properties of your new mynet network.

Initially, **no containers** connected yet.

## **📍 Step 101: Run Third Container on Custom Network**

docker run -d --name third --network mynet -p 8002:8080 first:1.0

**Meaning**:

* Start a new container called third.
* Connect it to **custom mynet** instead of default bridge.
* Host port 8002 mapped to container port 8080.

✅ The third container gets an IP address from 192.168.0.0/16 range (example: 192.168.0.2).

## **📍 Step 102: Check Interfaces Again**

ifconfig

✅ Still mainly seeing docker0 and possibly a new bridge created for mynet.

## **📍 Steps 103–105: Inspect Networks Again**

docker network inspect mynet

docker network inspect bridge

✅ Compare:

* bridge: contains first and second containers.
* mynet: contains third container.

**Key difference**: custom networks allow better isolation and IP control.

## **📍 Step 106: Exec into Container**

docker exec -it first bash

**Meaning**:  
 Login **inside** the first container interactively using bash.

# **🛠 Installing ping inside the Container**

Once inside the container, to install ping command:

apt update

apt install iputils-ping

✅ This will allow you to **ping** other containers by their IP address or container name (if in the same network).

# **📚 Detailed Explanation of Your Commands (Connecting container to another network)**

## **📍 Step 108: List all Containers**

docker ps -a

**Meaning**:  
 Shows all containers (running + stopped).  
 ✅ You can confirm containers like first, second, and third are created.

## **📍 Step 109: Exec into first Container**

docker exec -it first bash

**Meaning**:  
 Login **inside** first container to check internal settings, networking, etc.

## **📍 Step 110: Exec into third Container**

docker exec -it third bash

**Meaning**:  
 Login inside third container.

At this point:

* third is **only** connected to mynet network (custom network).
* It **cannot** communicate with containers connected to the bridge network (like first, second).

## **📍 Step 111: Connect third Container to bridge Network**

docker network connect bridge third

**Meaning**:

* **Attach** an existing running container (third) to an **additional network** (bridge).
* Now third container is connected to:  
  + mynet
  + **and** bridge

✅ It now has two network interfaces (two IP addresses):

* One from mynet (e.g., 192.168.0.2)
* One from bridge (e.g., 172.17.0.4)

✅ This allows **multi-network connectivity** — container can talk to services on both networks.

## **📍 Step 112: Exec into third Again**

docker exec -it third bash

**Meaning**:  
 Login into third container again.

Now inside third, if you run ip addr or ifconfig, you'll notice:

* Two network interfaces created
* Two IP addresses assigned

✅ You can now **ping** other containers like first which are only in the bridge network!

## **📍 Step 113: Disconnect third Container from bridge Network**

docker network disconnect bridge third

**Meaning**:

* **Detach** third from bridge network.
* Now third is only connected back to its original mynet.

✅ Removing from a network dynamically is useful when you want to **isolate** containers again.

## **📍 Step 114: Exec into third Again**

docker exec -it third bash

**Meaning**:  
 Login again after disconnection.

Inside third, if you check networking (ip addr):

* Only **one network interface** related to mynet will be there.
* No more bridge IP.

✅ third can no longer talk to containers like first (unless reconnected again).

# **🎯 Quick Summary Table**

| **Command** | **Meaning** |
| --- | --- |
| docker network connect bridge third | Attach running container to new network (bridge) |
| docker network disconnect bridge third | Detach running container from a network (bridge) |

# **⚡ Key Docker Networking Concept Here:**

👉 A **container can be attached to multiple networks at the same time**.  
 👉 You can dynamically **connect** or **disconnect** containers **at runtime** without restarting them.  
 👉 Containers get **new IP addresses** per network they join. 👉 After connecting to multiple networks, containers can **communicate** with more services.

# **📸 Diagram to Visualize**

Before:

third --> mynet (192.168.x.x)

After connect bridge:

third --> mynet (192.168.x.x)

bridge (172.17.x.x)

After disconnect bridge:

third --> mynet (192.168.x.x)

# **📚 Full Explanation of Commands for Multi-stage Docker Build Example**

## **📍 Step 116: Clone Git Repository**

git clone https://github.com/rskTech/multi-stage-example.git

**Meaning**:  
 Clones a sample Java Maven project which you can containerize.

## **📍 Step 117: Move to Project Folder**

cd multi-stage-example/

**Meaning**:  
 Switch into the directory containing the project code and Dockerfile.

## **📍 Step 120: Clean System**

docker system prune

**Meaning**:  
 Removes unused containers, images, volumes, networks to **free space**.  
 ✅ Useful before building new images.

## **📍 Step 121: List Existing Images**

docker images

**Meaning**:  
 Lists all locally available Docker images.

# **✨ FIRST: Without Multistage Build**

## **📍 Step 122: Create Dockerfile (Simple)**

FROM openjdk:8-jdk-alpine

RUN mkdir -p /app/source

COPY . /app/source

WORKDIR /app/source

RUN ./mvnw clean package

EXPOSE 8080

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom", "-jar", "/app/source/target/\*.jar"]

### **✅ What Happens Here:**

* **Base Image**: openjdk:8-jdk-alpine
* **Create Directory**: /app/source
* **Copy All Code**: into /app/source
* **Build Java Project**: using ./mvnw clean package
* **Run App**: directly from /app/source/target/\*.jar

⚡ Problem:  
 This image will contain **extra stuff**:

* Maven wrapper (mvnw)
* Source code (/src)
* Unnecessary build files (pom.xml, .git, etc.)

➡ **Image Size becomes large** (Not optimized).

## **📍 Step 130: Build Image**

docker build -t app\_without\_multistage:1.0 .

**Meaning**:  
 Builds the docker image with tag app\_without\_multistage:1.0.

## **📍 Step 131: List Images**

docker images

**Meaning**:  
 ✅ You'll notice **image size is BIG** because everything (source code + build tools) is inside!

# **✨ SECOND: With Multi-stage Build (Optimized)**

## **📍 Step 138: Create New Dockerfile (Multi-Stage)**

# ---- First Stage ----

FROM openjdk:8-jdk-alpine as builder

RUN mkdir -p /app/source

COPY . /app/source

WORKDIR /app/source

RUN ./mvnw clean package

# ---- Second Stage ----

FROM openjdk:8-jdk-alpine

COPY --from=builder /app/source/target/\*.jar /app/app.jar

EXPOSE 8080

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom", "-jar", "/app/app.jar"]

### **✅ What Happens Here:**

### **🔥 First Stage (Builder):**

* Uses openjdk:8-jdk-alpine
* Copies code
* Builds using Maven (./mvnw clean package)
* **Output**: .jar file ready inside /app/source/target/

BUT we **don’t keep** this builder stage in final image.

### **🔥 Second Stage (Final):**

* Again start **fresh** with openjdk:8-jdk-alpine
* **ONLY copy** the generated .jar file from builder stage
* Set entrypoint to run the app.

✅ **Result**:

* Very **small** final image
* Only **runtime dependencies** (not source code, not Maven wrapper, not build files)

⚡ **Big Benefit**: Smaller image = Faster deployment + More secure!

## **📍 Step 139: Build Optimized Image**

docker build -t app\_with\_multistage:1.0 .

**Meaning**:  
 Builds the optimized image using multi-stage approach with tag app\_with\_multistage:1.0.

## **📍 Step 143: List Images**

docker images

**Meaning**:  
 ✅ You’ll notice **app\_with\_multistage:1.0** is **much smaller** than **app\_without\_multistage:1.0**.

# **🎯 Summary Table**

| **Without Multistage** | **With Multistage** |
| --- | --- |
| Big image size | Small image size |
| Source code inside image | Only compiled JAR inside |
| Build tools present | No build tools present |
| Less secure | More secure |

**📸 Diagram: Multi-stage Build in Simple Flow**

+------------------------+ +-----------------------+

| Stage 1: Build Stage | | Stage 2: Final Image |

| (openjdk + mvnw + src) | | (openjdk + built .jar) |

| - Compile .jar | COPY ONLY | - Small size |

+------------------------+ ---------> +-----------------------+

# **🔥 Quick 1-Liner to Remember**

**Multi-stage Docker builds** = "Build Big, Deliver Small" ✅

# 🛠 **Docker Compose:**

### Step 39:

git clone https://github.com/rskTech/k8s\_material.git

* Cloning the GitHub repository to get sample files for practice.
* Repository contains Kubernetes and Docker-related examples.

### Step 40:

cd k8s\_material/docker\_compose/

* Navigating into the docker\_compose/ folder where your Docker Compose example files are located.

### Step 42:

vi app.py

* Creating or editing the Python Flask app.

Here’s the full content of app.py you provided:

from flask import Flask

from redis import Redis

import os

import socket

app = Flask(\_\_name\_\_)

redis = Redis(host=os.environ.get('REDIS\_HOST', 'redis'), port=6379)

@app.route('/')

def hello():

redis.incr('hits')

return 'Hello Container World! I have been seen %s times and my hostname is %s.\n' % (redis.get('hits'), socket.gethostname())

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host="0.0.0.0", port=5000, debug=True)

🔹 Explanation:

* Imports Flask (for web server) and Redis client.
* Connects to Redis server (host name picked from env variable REDIS\_HOST, defaults to redis).
* Defines / route:  
  + Increments a counter hits in Redis.
  + Returns a message showing how many times app is accessed and the container’s hostname.
* Runs the app on host 0.0.0.0 (accessible externally) at port 5000.

### Step 43:

vi Dockerfile

* Creating the Dockerfile to build a Docker image for the Flask application.

Here’s your Dockerfile content:

Dockerfile

CopyEdit

FROM python:2.7

COPY . /tmp

RUN pip install -r /tmp/requirements.txt

EXPOSE 5000

CMD ["python", "/tmp/app.py"]

🔹 Explanation:

* Base Image: python:2.7
* Copies current directory into /tmp inside the image.
* Installs Python dependencies listed in requirements.txt.
* Exposes port 5000.
* Sets the container to run app.py when started.

### Step 44:

vi compose.yaml

* Creating the Docker Compose configuration to define how multiple services (web and redis) will run together.

Here’s your compose.yaml:

version: "2"

services:

web:

build: .

ports:

- "8000:5000"

links:

- redis

networks:

- mynet

redis:

image: redis

expose:

- "6379"

networks:

- mynet

networks:

mynet:

🔹 Explanation:

* Services:  
  + web:  
    - Build from local Dockerfile.
    - Maps localhost:8000 → container:5000.
    - Linked to the Redis service for communication.
    - Joins custom network mynet.
  + redis:  
    - Uses official redis image.
    - Exposes Redis port 6379 inside the network.
    - Also joins mynet.
* Network:  
  + mynet allows both services to talk easily without conflict.

### Step 47:

apt update

* Updates package index on your Linux machine to install fresh packages.

### Step 49:

apt install docker-compose

* Installs Docker Compose (required to run multi-container applications).

### Step 51:

docker-compose -f compose.yaml up -d

* Brings up the services defined in compose.yaml in detached mode (background).
* What happens here:  
  + Docker builds the Flask app image.
  + Docker pulls the Redis image (if needed).
  + Creates the custom network mynet.
  + Runs two containers: web (Flask app) and redis.

### Step 52:

docker ps -a

* Lists all containers (running and stopped).
* You should see both containers: one for web and one for redis.

### Step 56:

curl localhost:8000

* Sends a HTTP request to localhost:8000.
* Should respond with something like:

Hello Container World! I have been seen 1 times and my hostname is 7a89d70a8a44.

* The count (seen) will increase each time you refresh or curl again.

### Step 57:

docker-compose -f compose.yaml down

* Stops and removes the running containers, networks, and resources that were created by Docker Compose.

# 📚 In Short

| Step | Action | Details |
| --- | --- | --- |
| Clone repo | Get example code. |  |
| app.py | Flask app integrated with Redis. |  |
| Dockerfile | Build image for app.py. |  |
| compose.yaml | Define web and redis services with networking. |  |
| apt update/install docker-compose | Set up environment. |  |
| docker-compose up | Start web + redis services. |  |
| curl localhost:8000 | Test the app working. |  |
| docker-compose down | Clean up everything. |  |

**Day4:**

# **Docker Swarm:**

### **Step 67**

docker system prune

* **Cleans up** Docker system.
* Removes:  
  + Unused containers
  + Unused networks
  + Unused images
  + Unused build cache
* 🔥 Useful before setting up a clean Swarm cluster to avoid old data causing conflicts.

👉 It will **prompt for confirmation** (y/n).

### **Step 69**

docker info

* **Displays Docker environment information**.
* Key parts:  
  + Docker version
  + How many containers, images, networks, etc.
  + Whether Swarm is enabled (you’ll see it disabled initially).

👉 Good to run **before and after swarm init** to see changes.

### **Step 70**

docker swarm init

* **Initializes the current Docker engine as a Swarm Manager**.
* Makes this node the **leader** of the swarm.
* Output shows:  
  + Swarm initialized message
  + A join-token for adding worker nodes.

📌 After this, docker info will show Swarm: active.

### **Step 71**

docker node ls

* **Lists all nodes** part of the swarm.
* Columns:  
  + **ID**: Node ID
  + **HOSTNAME**: Name of the node
  + **STATUS**: Ready/Down
  + **AVAILABILITY**: Active/Drain
  + **MANAGER STATUS**: Leader or Reachable or empty (if Worker)

At this point, **only one node** (your current machine) will be visible as **Leader**.

### **Step 72**

docker swarm join --token <token> <manager-ip>:2377

* **Command for other nodes to join the swarm**.
* Provided by the swarm init output.
* Worker nodes run this command to join **the Manager** node at port **2377** (Swarm communication port).

📌 If you run this on another machine, it joins as a **Worker** node.

**In your case**:

* IP = 172.31.27.27
* Token = SWMTKN-... (token validates the join request).

### **Step 73**

docker info

* **Checking again**.
* Now Docker info will show:  
  + **Swarm: active**
  + **Node Role**: Manager or Worker
  + **Number of Managers** and **Workers**.

👉 Useful to confirm node roles and cluster health.

### **Step 74**

docker node ls

* Lists nodes again after joining more nodes.
* Now you should see **multiple nodes**:  
  + One Manager
  + One or more Workers

📌 **Leader** node is responsible for:

* Scheduling services
* Replicating services
* Managing cluster state.

### **Step 75**

docker service create --name myweb --replicas 5 rajendrait99/first:1.0

* **Creates a Docker Service** (not just a container).
* Service Name: myweb
* **Replicas: 5** → means 5 copies (containers) will be created across available nodes.
* Image: rajendrait99/first:1.0 (your uploaded image on DockerHub).

🎯 **What happens internally:**

* Swarm schedules 5 containers across the cluster.
* It tries to balance them based on node load.

👉 **Important**: If only 1 node is available, it runs all 5 replicas on 1 node.

### **Step 76**

docker service ps myweb

* **Shows the tasks (containers)** created for the myweb service.
* Columns:  
  + **ID**
  + **NAME** (service name and task ID)
  + **IMAGE**
  + **NODE** (where it’s running)
  + **DESIRED STATE** (Running)
  + **CURRENT STATE** (Running or Failed or Shutdown)

📌 Helps verify if all replicas are up and running or any have failed.

**78: Docker swarm:**

docker service rm myweb

Removes (deletes) the service named myweb.

Stops all replicas running under that service.

Cleans up all related tasks and containers across the cluster.

📌 After this, there will be no service named myweb running.

**Step 79**

docker service ls

Lists all services currently running in Swarm.

After removing myweb, this will show empty output or only other active services if any.

**Step 80**

docker service create --name myweb --mode global rajendrait99/first:1.0

Creates a service called myweb.

This time using --mode global, not --replicas.

📌 What Global Mode means:

One container on every node automatically.

No matter how many nodes you have — each node runs exactly 1 replica.

Difference from Replicated Mode:

Replicated Mode → X copies total

Global Mode → 1 copy per node

**Step 81**

docker service ls

Lists services again.

You will see:

Name: myweb

Mode: Global

Replicated: X/X (X = number of nodes)

**Step 82**

docker service ps myweb

Shows all running tasks for service myweb.

Each node should show exactly one task running.

You can see:

Which node is running the task

Current status (Running, Failed, etc.)

**Step 83**

docker service rm myweb

Deletes the myweb service again.

This stops all globally running containers across all nodes.

📌 Clean up again to try new deployment with port publishing.

**Step 84**

docker service create --name myweb --mode global --publish target=8080,published=8000 rajendrait99/first:1.0

Creates service again in Global Mode, but this time exposes ports.

Let’s break this down:

--publish target=8080,published=8000

target=8080: The container inside expects traffic on port 8080.

published=8000: Docker Swarm maps host port 8000 to container port 8080.

So when you hit localhost:8000, traffic goes inside container at 8080.

📌 Important:

Now on every node, port 8000 on the host is available.

Ingress routing ensures correct delivery even in Swarm mode.

**Step 85**

curl localhost:8000

Tests if the service is reachable.

Sends an HTTP request to localhost:8000.

You should get the web response coming from the container (running Flask app inside).

Example output could be something like:

Hello Container World!

**DOcker Volume**

### **Step 42**

docker volume create myvolume

* Creates a **named Docker volume** called myvolume.
* It’s stored at /var/lib/docker/volumes/myvolume on the **host file system**.

🔍 Purpose:

* Persistent storage that survives container restarts.
* Can be **mounted** into containers or services.

### **Step 43**

ls /var/lib/docker/volumes/

* Lists Docker volumes **on the current node**.
* You should see a folder myvolume inside.

📌 Remember:

* Volumes are **node-local** unless you use **a volume plugin** (like NFS, EFS, Portworx).
* In a multi-node Swarm, this volume won’t automatically sync across nodes.

### **Step 45**

docker service create \

--name myweb \

--publish target=8080,published=8000 \

--mount src=myvolume,dst=/etc/lala \

rajendrait99/first:1.0

This is just a **repeat of the same service**, with dst= used instead of dest= — which is valid too.

✅ Both dest= and dst= are acceptable aliases in --mount.

### **Step 46**

docker service ls

* Lists all running services.
* You’ll see your myweb service and info such as:  
  + Mode (replicated/global)
  + Published ports
  + Replicas

### **📌 Important Note on Swarm + Volumes:**

* In Swarm, **volumes are not automatically distributed** across nodes.
* If your service spans multiple nodes and uses a **local named volume**, it might **fail to start on nodes where the volume doesn’t exist**.

✅ Best Practices:

* For production Swarm deployments using volumes, consider:  
  + **Shared storage plugins**: NFS, GlusterFS, Portworx, etc.
  + **Docker Volume Plugins** that support multi-node.
  + Or create the volume manually on all nodes with same name and directory structure.

### **✅ Summary Table**

| **Step** | **Command** | **What It Does** |
| --- | --- | --- |
| 39 | docker swarm init | Enables Swarm mode |
| 40 | docker swarm ls | ❌ Incorrect (should use docker info) |
| 41 | docker node ls | Shows Swarm nodes |
| 42 | docker volume create myvolume | Creates persistent volume |
| 43 | ls /var/lib/docker/volumes/ | Shows volume files |
| 44–45 | docker service create ... --mount | Starts service using volume |
| 46 | docker service ls | Lists services |

**DAY5:**

DOcker stack: https://github.com/rskTech/k8s\_material/tree/master/docker\_compose

### **🔍 Step-by-Step Explanation of Commands**

#### **Step 50**

docker service create --name registry --publish target=5000,published=5000 registry:2

* Creates a **private Docker registry** as a Swarm service.
* Publishes it on port **5000**, which will host your custom images (instead of Docker Hub).
* You'll push your app image here in the next steps.

#### **Step 51**

curl localhost:5000/v2/\_catalog

* Verifies that the local registry is running.
* Lists all pushed images in your **private registry** (initially empty).

#### **Step 52–54**

mkdir stackdemo

cd stackdemo/

* Creates a working directory stackdemo for your new app.

#### **Step 55–59**

You create the necessary app files:

* app.py: Your Flask web app.
* Dockerfile: To containerize the app.
* requirements.txt: Python dependencies.
* compose.yaml: Defines multi-container app structure.

version: "3"

services:

web:

build: .

image: localhost:5000/stackdemo\_web:1.0

ports:

- "8000:5000"

deploy:

replicas: 3

links:

- redis

networks:

- mynet

redis:

image: redis

expose:

- "6379"

networks:

- mynet

networks:

mynet:

#### **Step 60–61**

docker-compose # runs docker-compose, error if not installed

sudo apt update

sudo apt install docker-compose

* You install docker-compose if it’s not already present.

#### **Step 62**

docker-compose -f compose.yaml build

* Builds your image using the local Dockerfile and tags it as stackdemo\_web.

#### **Step 63**

docker images

* Verifies the image was built successfully.

#### **Step 64**

docker-compose -f compose.yaml push

* Pushes your built image to the **local registry at localhost:5000**.
* Now it can be pulled and deployed on **any Swarm node**.

#### **Step 65**

curl localhost:5000/v2/\_catalog

* Now should show your pushed image:

{"repositories":["stackdemo\_web"]}

#### **Step 70**

docker stack deploy --compose-file compose.yaml stackdemo

* Deploys your app as a **Swarm stack** named stackdemo.
* Similar to docker-compose up but used in Swarm.
* Reads the same compose.yaml, but uses it in **Swarm mode**.

#### **Step 71**

docker service ls

* Lists services created by your stack. You’ll see one like:

#### **Step 72**

curl localhost:8000

* Accesses the running container via published port 80.
* Should return the Flask response from app.py.

### 

### 

### **📊 Comparison Table: Docker Commands**

| **Feature / Tool** | **docker run** | **docker-compose** | **docker service** | **docker stack** |
| --- | --- | --- | --- | --- |
| **Mode** | Standalone | Local multi-container | Swarm (single service) | Swarm (multi-container via Compose) |
| **Purpose** | Run 1 container | Run multiple containers locally | Run 1 service in Swarm | Run app as a full stack on Swarm |

**Scaling and UPDATE**

#### **188 docker service ls**

* **Purpose: Lists all running Docker services in your Swarm cluster.**
* **Usage: Check current services, replicas, and their status.**

#### **189 docker service scale stackdemo\_web=5**

* **Purpose: Scales the stackdemo\_web service to 5 replicas.**
* **Usage: You use this to increase the number of containers for high availability or load distribution.**

#### **190 docker service ps stackdemo\_web**

* **Purpose: Shows the tasks (containers) running for the stackdemo\_web service.**
* **Usage: Helps you verify where the replicas are running and their current status (running, preparing, etc.).**

#### **191 docker service scale stackdemo\_web=2**

* **Purpose: Scales down the service from 5 to 2 replicas.**
* **Usage: Useful for reducing resource usage during low load.**

#### **192 docker service ps stackdemo\_web**

* **Same as command 190: Shows current running tasks, now likely reduced to 2.**

#### **193 vi app.py**

* **Purpose: Opens the app.py file in the vi editor.**
* **Update Python application code here.**

#### **194 vi compose.yaml**

* **Purpose: Opens your Docker Compose file in the vi editor.**
* **Usage: update image version to 2.0.**

#### **195 docker-compose -f compose.yaml build**

* **Purpose: Builds Docker images as defined in compose.yaml.**
* **Usage: Builds local images using updated app.py or other code changes.**

#### **196 docker-compose -f compose.yaml push**

* **Purpose: Pushes built images to a Docker registry.**
* **Usage: Makes your images available for Docker Swarm to pull, especially if using a local/private registry like localhost:5000.**

#### **198 docker service ps stackdemo\_web**

* **Checks the state of the service before updating (same as earlier).**

#### **199 docker service update --image=localhost:5000/stack:2.0 stackdemo\_web**

* **Purpose: Updates the running service stackdemo\_web to use the new image stack:2.0 from your local registry.**
* **Usage: Rolling update of containers with the new version of the application.**

#### **200 docker service ps stackdemo\_web**

* **Verifies the update: Checks if new tasks have been created and old ones shut down.**

#### **201 curl localhost:8001**

* **Purpose: Sends an HTTP request to localhost:8001 (likely where your app is published).**
* **Usage: Tests the service endpoint to see if the app responds correctly after the update.**

### **✅ Summary:**

**These commands show a complete cycle of:**

* **Scaling services**
* **Updating code**
* **Rebuilding and pushing new images**
* **Rolling out the update using docker service update**
* **Testing the final deployed app using curl**

**Deploying Visualizer:**

**docker service create --name=viz --publish=8080:8080/tcp --constraint=node.role==manager --mount=type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock alexellis2/visualizer-arm:latest**

### **🔐 What is Trivy?**

**Trivy** (short for *Tri*age *Vuln*erabilit*Y*) is an open-source, easy-to-use vulnerability scanner developed by **Aqua Security**. It is widely used to scan:

* **Docker images**
* **Filesystem and configuration files**
* **SBOMs (Software Bill of Materials)**
* **Git repositories and IaC (Terraform, Kubernetes YAML)**

Trivy scans for:

* **OS package vulnerabilities** (e.g., in Alpine, Debian, etc.)
* **Application dependencies vulnerabilities** (e.g., pip, npm, etc.)
* **Secrets** (API keys, tokens, etc.)
* **Misconfigurations** in Dockerfile, Kubernetes, etc.

### **🧪 Commands Explained**

#### **🔧 snap install trivy**

* Installs Trivy via the Snap package manager.
* After this, you can run Trivy as a system-wide CLI tool.

#### **📦 docker images**

* Lists all local Docker images. You typically run this to find which image to scan.

#### **🔍 trivy image rajendrait99/first:1.0**

* Scans the Docker image rajendrait99/first:1.0 for known vulnerabilities.
* It checks OS-level packages and application dependencies (e.g., Python packages if it's a Flask app).
* Outputs a list of detected vulnerabilities, including:  
  + Severity (CRITICAL, HIGH, MEDIUM, etc.)
  + Vulnerable package names and versions
  + Links to CVE details

✅ Output Example:

Total: 12 (CRITICAL: 2, HIGH: 3, MEDIUM: 5, LOW: 2)

...

#### **📘 trivy image -h**

* Shows help/documentation for Trivy's image scanning subcommand.
* Lists available flags like:  
  + --severity
  + --ignore-unfixed
  + -o (output to file)
  + --format json/table/sarif etc.

#### **📄 trivy image rajendrait99/first:1.0 -o out**

* Runs the same image scan, but redirects output to a file named out.
* Useful in CI/CD pipelines or audit logging.

### **✅ Why Trivy?**

| **Feature** | **Details** |
| --- | --- |
| 🛡️ **Security** | Scans containers before deployment |
| 🔄 **CI/CD Ready** | Works with GitHub Actions, GitLab CI, Jenkins, etc. |
| 📦 **Package-aware** | Detects Python, Java, Node.js, Ruby, Go dependencies |
| 🧩 **Extensible** | Supports scanning Kubernetes manifests, Dockerfiles, etc. |
| 🧑‍💻 **Dev-Friendly** | Easy to use with very low learning curve |

### **✅ Example Use Cases**

| **Use Case** | **Command** |
| --- | --- |
| Scan image locally | trivy image nginx:alpine |
| Scan and save report | trivy image nginx -o report.txt |
| Scan with only HIGH and CRITICAL | trivy image --severity HIGH,CRITICAL nginx |
| CI integration | trivy image --exit-code 1 nginx (fails build if vulns found) |