Prathamesh Arvind Jadhav

Docker Notes

1. What is Docker?

□ Definition:

Docker is an **open-source platform** that allows developers to **build, package, and run applications** in **lightweight, portable containers**.

Each **container** includes everything the application needs — code, runtime, libraries, and dependencies — ensuring it runs **the same way on any environment** (developer's system, test server, or production).

□ Analogy:

Think of a **shipping container** in logistics — it can carry anything (toys, furniture, food) and be loaded on any ship, train, or truck.
Similarly, a **Docker container** can carry any software and run on any OS or cloud environment that supports Docker.

---

2. Why Docker? (Problem it Solves)

□ The Old Problem:

Before Docker, developers used to deploy applications using different environments (local, staging, production). This often caused issues like:

Problem	Description
"Works on my machine" issue	An app runs fine on a developer's computer but fails on another system due to different dependencies.
Heavy Virtual Machines	Running multiple VMs consumed a lot of CPU, RAM, and storage.
Complex Dependency Management	Each app required specific versions of libraries and frameworks.
Slow Setup	Setting up new environments or testing systems took a lot of time.

□ The Docker Solution:

Docker solves these issues by:

- **Packaging** applications and dependencies together (container image).
- Ensuring **consistency** across development, testing, and production.
- Making environments **lightweight and fast**.

- Allowing **rapid deployment** and **scaling** using container orchestration.

 In short: Docker makes apps portable, reliable, and fast to deploy anywhere.

---

3. Difference Between Docker and Virtual Machines

Feature	Docker (Containers)	Virtual Machines (VMs)
Architecture	Shares the host OS kernel	Each VM runs its own full OS
Performance	Lightweight, starts in seconds	Heavy, takes minutes to boot
Resource Usage	Very low (no separate OS)	High (each VM needs full OS)
Isolation	Process-level isolation	Hardware-level isolation
Image Size	Smaller (MBs)	Larger (GBs)
Startup Time	Seconds	Minutes
Portability	Very high	Moderate
Use Case	Microservices, CI/CD, scaling	Running different OS, legacy apps

Diagram (text version):

```
Virtual Machine:
[Hardware]
  ↳ [Host OS]
      ↳ [Hypervisor]
          ↳ [Guest OS + App A]
          ↳ [Guest OS + App B]

Docker:
[Hardware]
  ↳ [Host OS]
      ↳ [Docker Engine]
          ↳ [App A + Dependencies]
          ↳ [App B + Dependencies]
```

 Summary:
Docker containers run on the same OS kernel, making them **faster and more efficient** than virtual machines that require a full OS per instance.

---

4. Key Features and Advantages of Docker

 Key Features:

1. **Containerization Technology** – Packages apps and dependencies into isolated containers.
2. **Portability** – Run anywhere: local machine, cloud, or on-premise server.
3. **Lightweight** – Containers share the host OS kernel, reducing overhead.
4. **Version Control & Reusability** – Docker images can be versioned and reused.
5. **Isolation** – Each container runs independently and securely.
6. **Automation** – Supports CI/CD workflows using tools like Jenkins, GitHub Actions, etc.
7. **Scalability** – Easily scaled using Docker Compose or orchestration tools like Kubernetes.
8. **Docker Hub** – Public registry to store and share container images.

□ **Advantages of Docker:**

Advantage	Description
Consistency	Eliminates "works on my machine" issues.
Speed	Faster builds and deployments.
Efficiency	Uses fewer resources than VMs.
Portability	Run the same container image anywhere.
Security	Isolated runtime environments.
Simplified Testing	Quickly spin up test environments.
Easier Collaboration	Developers can share same images via Docker Hub.

□ **Example: Simple Docker Workflow**

1. **Build an Image**

```
docker build -t myapp .
```

2. **Run a Container**

```
docker run -p 8080:8080 myapp
```

3. **Share Image**

```
docker push myusername/myapp
```

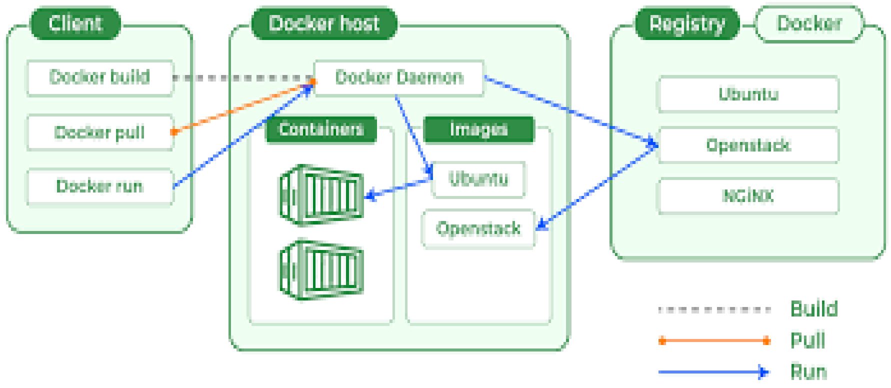
2. Docker Architecture

Docker follows a **client-server architecture**, which includes three main components:

- **Docker Client**
- **Docker Daemon (Server)**

- **Docker Host (Environment where Docker runs)**

All of these work together to **build, run, and manage containers**.



1. Docker Daemon (dockerd)

 Definition:

The **Docker Daemon** is the **core background process** that manages all Docker objects — containers, images, networks, and volumes.

 Responsibilities:

- Builds and runs containers.
- Pulls/pushes images to and from registries.
- Manages container lifecycle (start, stop, delete).
- Handles communication with Docker Client through REST API.

 Command to check:

```
ps aux | grep dockerd
```

 Key Point:

The daemon listens for Docker API requests and performs container-related tasks.

---

2. Docker Client (docker)

 Definition:

The **Docker Client** is the **primary user interface** to interact with Docker.
You use it when you run commands like:

docker build, docker run, docker pull, docker push

 Working:

- The client sends commands to the **Docker Daemon** using REST API (either locally or remotely).
- Example:
- docker run nginx

  → This tells the **daemon** to download the nginx image and start a container from it.

 Multiple Clients:

You can have multiple Docker Clients communicating with **one or more Docker Daemons**.

---

3. Docker Host

 Definition:

The **Docker Host** is the **machine (physical or virtual)** that runs:

- The **Docker Daemon**
- The **containers**

 Components inside Host:

1. **Docker Daemon (dockerd)** – manages containers.
2. **Containers** – running applications.
3. **Images, Volumes, Networks** – used by containers.

 Example:

- Your laptop running Docker Desktop = Docker Host.
- A Linux cloud VM with Docker installed = Docker Host.

---

4. Docker Images, Containers, and Registries

These are **core Docker objects**.

Docker Images

 Definition:

A **Docker image** is a **read-only template** that contains everything needed to run an application — code, libraries, runtime, and configuration files.

 Key Features:

- Immutable (cannot be changed once built)
- Built in layers (using Union File System)
- Created from a Dockerfile

 Example:
```
docker pull python:3.11
```

This downloads an image for Python 3.11.

 View local images:
```
docker images
```

Docker Containers

 Definition:

A **Docker container** is a **running instance of an image**.
It's like a lightweight, isolated environment that executes your application.

 Lifecycle:

Command	Description
docker run	Create + Start a container
docker ps	List running containers
docker stop <id>	Stop container
docker rm <id>	Remove container

 Example:

```
docker run -it ubuntu bash
```

→ Starts an Ubuntu container and opens a terminal inside it.

 Concept:

Image = Template
Container = Running Copy of that Template

Docker Registries

 Definition:

A **Docker Registry** is a **storage location** for Docker images.
It allows you to **push (upload)** and **pull (download)** images.

 Types:

1. **Public Registry** → e.g., Docker Hub
2. **Private Registry** → For enterprise or internal use (e.g., AWS ECR, GitHub Container Registry)

 Example Commands:

```
docker login
docker push prathameshjadhav/myapp:v1
docker pull ubuntu
```

---

5. Docker Engine Overview

 Definition:

Docker Engine is the **core part of Docker platform** — the runtime that runs and manages containers.

It consists of:

1. **Docker Daemon (dockerd)**
2. **Docker API (REST API)**
3. **Docker CLI (Client)**

 Types of Docker Engine:

1. **Docker Engine – Community (CE)** → Free, for developers.
2. **Docker Engine – Enterprise (EE)** → Paid, with advanced features.

 Functionality:

- Builds and runs containers
- Manages networking, storage, and images
- Provides the foundation for orchestration tools like **Docker Swarm** and **Kubernetes**

3. Docker Installation & Setup

(including installation steps on all OS and essential Docker CLI commands)

---

1. Installing Docker on Windows / Linux / Mac

Docker provides **Docker Desktop** for Windows and macOS, and **Docker Engine** packages for Linux systems.

A. Installing Docker on Windows

 Step-by-Step:

1. **System Requirements**
   - Windows 10 (64-bit, Pro/Enterprise/Education) or Windows 11.
   - Enable **WSL 2** (Windows Subsystem for Linux).
   - At least **4 GB RAM**.
2. **Download Docker Desktop**
   - Visit: https://www.docker.com/products/docker-desktop
3. **Run Installer**
   - Double-click the installer file → follow installation wizard.
4. **Enable WSL 2 Backend**
   - During setup, select **Use WSL 2 instead of Hyper-V** (recommended).
5. **Start Docker Desktop**
   - After installation, open **Docker Desktop** from Start Menu.
   - Wait until you see the **Docker whale icon** running in your system tray.
6. **Verify Installation**
   Open **Command Prompt / PowerShell / VS Code terminal**, run:
7. docker --version
8. docker run hello-world

   Output should confirm Docker is installed and running properly.

 B. Installing Docker on Linux (Ubuntu)

 Step-by-Step:

1. **Uninstall older versions**

```
sudo apt-get remove docker docker-engine docker.io containerd runc
```

2. **Update your system**

```
sudo apt-get update
```

3. **Install dependencies**

```
sudo apt-get install ca-certificates curl gnupg lsb-release
```

4. **Add Docker's official GPG key**

```
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/etc/apt/keyrings/docker.gpg
```

5. **Set up the repository**

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
 https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | \
 sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

6. **Install Docker Engine**

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-
compose-plugin
```

7. **Verify Installation**

```
docker --version
sudo docker run hello-world
```

8. **(Optional) Run Docker Without sudo**

```
sudo usermod -aG docker $USER
newgrp docker
```

□ C. Installing Docker on macOS

□ ***Step-by-Step:***

1. **Download Docker Desktop for Mac**
   - Visit: https://www.docker.com/products/docker-desktop
2. **Install**
   - Open the downloaded .dmg file.
   - Drag **Docker.app** to your **Applications** folder.
3. **Run Docker Desktop**
   - Open Docker from Applications.
   - Wait for the **Docker whale icon** to appear in the menu bar.
4. **Verify Installation**
5. docker --version
6. docker run hello-world

□ Common Verification Command (All OS)

```
docker --version          # Check Docker version
docker info               # Show system-wide Docker information
docker run hello-world    # Test installation
```

If you see:

"Hello from Docker! Your installation appears to be working correctly."

Then your setup is successful □

---

2. Basic Docker CLI Commands

Here are the most commonly used Docker commands to get you started □

□ System Commands

Command	Description
docker --version	Show Docker version

Command	Description
docker info	Display system information about Docker
docker help	List all available commands

Image Commands

Command	Description
docker pull <image>	Download an image from Docker Hub
docker images	List all images on your system
docker rmi <image_id>	Remove an image
docker build -t <name> .	Build an image from a Dockerfile
docker tag <source> <target>	Tag an image with a new name

Example:

```
docker pull nginx
docker images
docker rmi nginx
```

Container Commands

Command	Description
docker run <image>	Create and start a new container
docker run -it ubuntu bash	Run container interactively
docker ps	Show running containers
docker ps -a	Show all containers (running + stopped)

Command	Description
docker stop <container_id>	Stop a running container
docker start <container_id>	Start a stopped container
docker restart <container_id>	Restart a container
docker rm <container_id>	Remove a container
docker exec -it <container_id> bash	Access running container shell

Example:

```
docker run -d -p 8080:80 nginx
docker exec -it <container_id> bash
docker stop <container_id>
```

Logs & Stats

Command	Description
docker logs <container_id>	Show container logs
docker stats	Show real-time container resource usage
docker top <container_id>	Show running processes inside container

---

Clean-Up Commands

Command	Description
docker system prune	Remove unused data
docker image prune	Remove dangling images
docker container prune	Remove stopped containers

Command	Description
docker volume prune	Remove unused volumes

□ Docker Compose (Optional, for multi-container apps)

Command	Description
docker compose up	Start services defined in docker-compose.yml
docker compose down	Stop and remove services
docker compose ps	List running services

4. Docker Images and Containers

This topic is the **core foundation** of working with Docker in real-world development and DevOps.

---

1. What are Docker Images?

□ Definition:

A **Docker image** is a **read-only template** that contains:

- The **application code**
- **Runtime environment** (e.g., Node, Python)
- **Libraries**, **dependencies**, and **system tools**

It acts as a **blueprint** for creating **containers**.

□ Example:

An image for a Node.js app might include:

- Ubuntu base image
- Node.js runtime
- App source code
- Configuration files

□ **Analogy:**

Image = Class (blueprint)
Container = Object (instance of that blueprint)

---

2. What are Docker Containers?

□ **Definition:**

A **Docker container** is a **running instance of a Docker image**.
It provides a **lightweight, isolated environment** to run your application.

Each container:

- Runs independently
- Shares the host OS kernel
- Has its own filesystem, network, and process space

□ **Example:**

If you run:

```
docker run nginx
```

Docker:

1. Pulls the **nginx image** (if not present locally)
2. Creates a **container** from that image
3. Starts a running **Nginx web server** inside that container

---

3. Creating Containers from Images

There are multiple ways to create and run containers.

□ **Basic Syntax**

```
docker run [OPTIONS] IMAGE [COMMAND]
```

□ **Example 1 – Run an Ubuntu Container**

```
docker run -it ubuntu bash
```

 Breakdown:

- -it: interactive terminal
- ubuntu: image name
- bash: command to run inside container

Result: You'll enter into the Ubuntu container's shell.

Example 2 – Run Nginx in Background

```
docker run -d -p 8080:80 nginx
```

 Breakdown:

- -d: detached mode (runs in background)
- -p 8080:80: maps port 8080 of host → 80 inside container
- nginx: image name

Visit http://localhost:8080 to view Nginx running.

Example 3 – Assign Custom Name

```
docker run -d --name myweb nginx
```

Now the container can be managed easily using its name myweb.

Example 4 – Create but Don't Start

```
docker create ubuntu
```

This creates a container but doesn't run it immediately.

Start it later using:

```
docker start <container_id>
```

---

4. Listing, Starting, Stopping, and Removing Containers

List Containers

Command	Description
docker ps	List **running** containers
docker ps -a	List **all** containers (running + stopped)
docker ps -q	List only container IDs

Example Output:

```
CONTAINER ID   IMAGE    COMMAND     STATUS       NAMES
c3f279d17e0a   nginx    "/docker…"  Up 3 minutes  myweb
```

Stop a Running Container

```
docker stop <container_id or name>
```

Example:

```
docker stop myweb
```

Start a Stopped Container

```
docker start <container_id or name>
```

Restart a Container

```
docker restart <container_id or name>
```

Inspect Container Details

```
docker inspect <container_id or name>
```

Shows full configuration details in JSON format.

Remove Containers

Command	Description
docker rm <id>	Remove a stopped container
docker rm -f <id>	Force remove even if running
docker container prune	Remove all stopped containers

Example:

docker rm myweb

□ Useful Shortcuts

Action	Command
Stop all running containers	docker stop $(docker ps -q)
Remove all stopped containers	docker container prune
Remove all containers (force)	docker rm -f $(docker ps -aq)

□ 5. Difference Between Image and Container

Feature	Docker Image	Docker Container
Definition	A read-only template used to create containers	A running instance of an image
State	Static (does not change)	Dynamic (can start, stop, modify)
File System	Read-only	Read + Write layer on top of image
Persistence	Cannot be modified directly	Changes lost when deleted unless saved
Creation Command	Built using docker build	Created using docker run
Storage Location	Stored in local image cache or registry	Runs in memory, stored as a process
Example	nginx:latest	container running nginx web server

□ Analogy:

Image = Blueprint of a house
Container = Actual built house based on that blueprint

You can build **many containers** (houses) from **one image** (blueprint).

---

6. Practical Example: From Image → Container

Step 1: Pull an Image

```
docker pull ubuntu
```

Step 2: Create and Run Container

```
docker run -it ubuntu bash
```

Step 3: List Containers

```
docker ps -a
```

Step 4: Exit and Stop

```
exit          # exits container
docker ps     # verify it stopped
```

Step 5: Remove Container

```
docker rm <container_id>
```

5. Docker Hub

This section will help you clearly understand **what Docker Hub is**, how to **pull and push images**, and how to use **official repositories and tags** effectively.

---

1. What is Docker Hub?

 Definition:

Docker Hub is the **official cloud-based registry** provided by Docker Inc.
It is used to **store, share, and distribute Docker images** publicly or privately.

It acts as a **central repository** similar to:

- **GitHub** → for code
- **Docker Hub** → for container images

□ **Key Features:**

Feature	Description
Image Hosting	Stores container images (public or private)
Image Distribution	Developers can push/pull images from anywhere
Official Images	Verified, secure base images maintained by Docker
Automated Builds	Automatically build images from GitHub or Bitbucket
Version Control (Tags)	Manage different versions of images using tags
Web UI + CLI Access	Accessible from website and Docker CLI

□ **URL:**

□ https://hub.docker.com

You can:

- Search for images
- View official repositories (e.g., nginx, mysql, node, python)
- Log in to manage your own repositories

---

2. Pulling Images from Docker Hub

□ **Definition:**

Pulling an image means **downloading** a Docker image from Docker Hub to your local machine.

□ **Syntax:**

```
docker pull <image_name>:<tag>
```

□ **Examples:**

1. **Pull the latest Ubuntu image**
2. docker pull ubuntu

   (If no tag is specified, Docker automatically pulls the latest tag.)

3. **Pull a specific version**
4. docker pull ubuntu:22.04
5. **Pull an Nginx web server image**
6. docker pull nginx
7. **View downloaded images**
8. docker images

□ **How It Works (Internally):**

docker pull nginx:latest
↓
1. Docker client sends request to Docker Hub registry.
2. Hub authenticates & locates "nginx:latest" image.
3. Image layers are downloaded and stored locally.
4. You can now create containers from that image.

---

3. Pushing Images to Docker Hub

□ **Definition:**

Pushing an image means **uploading your local Docker image** to Docker Hub — so it can be shared or used elsewhere.

□ **Prerequisites:**

1. You need a Docker Hub account.
2. Log in from terminal:
3. docker login

   Enter your **Docker ID** and **password**.

Steps to Push an Image

□ ***Step 1: Check existing images***
docker images

Example output:

```
REPOSITORY    TAG       IMAGE ID       SIZE
myapp         latest    1a2b3c4d5e6f   300MB
```

 Step 2: Tag the image

You must tag the image with your **Docker Hub username** before pushing:

```
docker tag myapp:latest prathameshjadhav/myapp:v1
```

Format:

```
docker tag <local_image>:<tag> <username>/<repo_name>:<tag>
```

 Step 3: Push the image

```
docker push prathameshjadhav/myapp:v1
```

 Step 4: Verify on Docker Hub

Visit
 https://hub.docker.com/repositories/prathameshjadhav
You’ll see your uploaded image.

 Step 5: Pull from any machine

Now you (or others) can download it anywhere:

```
docker pull prathameshjadhav/myapp:v1
```

 Example Summary:

```
# Login
docker login

# Tag image
docker tag myapp:latest prathameshjadhav/myapp:v1

# Push image
docker push prathameshjadhav/myapp:v1
```

```
# Verify
docker pull prathameshjadhav/myapp:v1
```

---

4. Official Repositories and Tags

What Are Official Repositories?

Official Repositories are **verified and curated images** maintained by:

- Docker Inc. or
- Trusted third-party organizations (like Ubuntu, MySQL, Nginx, etc.)

They are **secure**, **optimized**, and **frequently updated**.

Example of Official Repositories

Repository	Description	Example Pull Command
nginx	Official Nginx web server	docker pull nginx
mysql	Official MySQL database	docker pull mysql
python	Official Python runtime	docker pull python
ubuntu	Official Ubuntu OS	docker pull ubuntu
node	Official Node.js environment	docker pull node

Note: These repositories are prefixed by **library/** internally (e.g., library/nginx).

What Are Tags?

Tags are used to label **different versions** or **variants** of an image.

Each tag represents a specific **build** of that image.

□ Example of Tags

Repository	Tag	Description
ubuntu:20.04	20.04	Ubuntu 20.04 LTS
ubuntu:22.04	22.04	Ubuntu 22.04 LTS
python:3.11	3.11	Python 3.11 runtime
node:18-alpine	18-alpine	Lightweight Alpine version
mysql:latest	latest	Default MySQL version

If you omit a tag, Docker assumes:

docker pull <image>:latest

□ Viewing Tags on Docker Hub

You can view all available tags on a repository's page, e.g.:

□ https://hub.docker.com/_/nginx/tags

□ 5. Summary Commands

Action	Command	Description
Pull image	docker pull <image>:<tag>	Download from Docker Hub
List images	docker images	Show local images
Tag image	docker tag <local> <user>/<repo>:<tag>	Rename before push
Push image	docker push <user>/<repo>:<tag>	Upload to Docker Hub
Login	docker login	Authenticate to Docker Hub
Logout	docker logout	End session

 Example Workflow Summary

```
# 1. Build an image
docker build -t myapp .

# 2. Tag the image for Docker Hub
docker tag myapp prathameshjadhav/myapp:v1

# 3. Login to Docker Hub
docker login

# 4. Push the image
docker push prathameshjadhav/myapp:v1

# 5. Pull it from anywhere
docker pull prathameshjadhav/myapp:v1
```

6. Dockerfile

---

1. Purpose of Dockerfile

 Definition:

A **Dockerfile** is a **text file** that contains a set of **instructions** to **automate the building of Docker images**.

It acts as a **blueprint** for creating containers — defining everything from the **base image** to the **software** and **commands** needed to run your application.

 Why Use a Dockerfile?

Benefit	Description
Automation	Builds consistent images automatically
Portability	Same image works across environments

Benefit	Description
Version Control	Dockerfile can be versioned with Git
Reusability	Same base Dockerfile can be extended
Simplified Deployment	Build → Push → Run (on any platform)

□ Real-Life Analogy:

Think of a Dockerfile as a **recipe** □.
Each instruction (like FROM, RUN, COPY, etc.) is a **step** to prepare your container image.

□ Example:

A simple Dockerfile for a Node.js app:

```
# Step 1: Base image
FROM node:18-alpine

# Step 2: Set working directory
WORKDIR /app

# Step 3: Copy package files
COPY package*.json ./

# Step 4: Install dependencies
RUN npm install

# Step 5: Copy all source code
COPY . .

# Step 6: Expose port
EXPOSE 8080

# Step 7: Start the application
CMD ["npm", "start"]
```

---

2. Structure of a Dockerfile

Order	Instruction	Purpose
1	FROM	Defines base image
2	LABEL	Adds metadata like author info
3	ENV	Sets environment variables
4	WORKDIR	Sets working directory inside container
5	COPY / ADD	Copies files into image
6	RUN	Executes commands at build time
7	EXPOSE	Declares port to access service
8	CMD / ENTRYPOINT	Defines default command to run

---

3. Common Dockerfile Instructions

Let’s go through each with explanation and examples □

□ (1) FROM

- Defines the **base image** for your Docker image.
- Every Dockerfile **must start** with FROM.

Example:

```
FROM ubuntu:22.04
```

or

```
FROM python:3.11-slim
```

□ You can also use a previous custom image as a base:

```
FROM prathameshjadhav/mybase:v1
```

□ (2) LABEL

- Adds **metadata** (author, version, description).

Example:

```
LABEL maintainer="Prathamesh Jadhav <prathamesh@example.com>"
LABEL version="1.0"
LABEL description="This is a demo Dockerfile"
```

□ **(3) RUN**

- Executes **commands at build time** inside the image.
- Commonly used to install software or dependencies.

Example:

```
RUN apt-get update && apt-get install -y python3 python3-pip
RUN pip install flask
```

□ Each RUN creates a **new image layer** — so combine commands with && to optimize image size.

□ **(4) WORKDIR**

- Sets the **working directory** inside the container (like cd).
- All subsequent commands run from this directory.

Example:

```
WORKDIR /app
```

□ If the folder doesn’t exist, Docker automatically creates it.

□ **(5) COPY**

- Copies files/folders from your **host machine** → into the **container image**.

Example:

```
COPY . /app
```

□ Syntax:

```
COPY <source> <destination>
```

---

(6) ADD

- Works like COPY but with extra features:
  - Can **extract compressed files**.
  - Can **download from URLs**.

Example:
ADD app.tar.gz /app
ADD https://example.com/file.txt /data/

□ Use COPY when possible — ADD is for special cases.

□ **(7) ENV**

- Defines **environment variables** inside the image.

Example:
ENV APP_HOME=/app
ENV PORT=8080
WORKDIR $APP_HOME

□ **(8) EXPOSE**

- Documents the **port number** the container listens on.
- It doesn’t actually publish the port — it’s just metadata.

Example:
EXPOSE 8080

Use docker run -p 8080:8080 to make it accessible outside.

□ **(9) CMD**

- Specifies the **default command** to run when a container starts.
- Can be overridden during docker run.

Example:
CMD ["python3", "app.py"]

□ Only the **last CMD** instruction takes effect.

□ (10) ENTRYPOINT

- Defines the **main executable** for the container.
- Unlike CMD, **it cannot be overridden** easily.
- Often used for fixed commands.

Example:
ENTRYPOINT ["python3", "app.py"]

□ Combine both for flexibility:

ENTRYPOINT ["python3"]
CMD ["app.py"]

This runs:

python3 app.py

□ (11) USER

- Sets the **user** under which container runs (instead of root).

Example:
USER node

□ (12) VOLUME

- Creates a mount point for **persistent storage**.

Example:
VOLUME ["/data"]

□ (13) ARG

- Defines variables available **only at build time** (unlike ENV).

Example:
ARG VERSION=1.0
RUN echo "Building version $VERSION"

Build with:

```
docker build --build-arg VERSION=2.0 .
```

---

4. Building Custom Images

Once your Dockerfile is ready, build the image with:

□ **Syntax:**

```
docker build -t <image_name>:<tag> <path_to_dockerfile>
```

□ **Example:**

If your Dockerfile is in the current directory:

```
docker build -t myapp:v1 .
```

□ **Verify Image:**

```
docker images
```

□ **Run the Container:**

```
docker run -d -p 8080:8080 myapp:v1
```

□ **Check Running Containers:**

```
docker ps
```

---

5. Example: Python Flask App

□ **Dockerfile:**

```
# Step 1: Base image
FROM python:3.11-slim

# Step 2: Set working directory
WORKDIR /app
```

```
# Step 3: Copy code
COPY . .

# Step 4: Install dependencies
RUN pip install -r requirements.txt

# Step 5: Expose port
EXPOSE 5000

# Step 6: Run the app
CMD ["python", "app.py"]
```

Commands to Build & Run:

```
docker build -t flask-app:v1 .
docker run -d -p 5000:5000 flask-app:v1
```

Now visit http://localhost:5000

---

6. CMD vs ENTRYPOINT

Feature	CMD	ENTRYPOINT
Purpose	Default command	Fixed command
Overridable	Yes	Not easily
Example	CMD ["npm", "start"]	ENTRYPOINT ["npm", "start"]
Best for	Flexible commands	Always run same binary

---

7. Summary Table

Instruction	Purpose	Example
FROM	Base image	FROM ubuntu:22.04

Instruction	Purpose	Example
LABEL	Metadata	LABEL maintainer="Prathamesh"
RUN	Build-time commands	RUN apt-get install -y nginx
COPY	Copy files	COPY . /app
ADD	Copy + extract	ADD app.tar.gz /app
WORKDIR	Set working directory	WORKDIR /app
ENV	Set environment variables	ENV PORT=8080
EXPOSE	Declare port	EXPOSE 8080
CMD	Default run command	CMD ["npm", "start"]
ENTRYPOINT	Main executable	ENTRYPOINT ["python3"]

7. Image Layers and Caching in Docker

---

1. Understanding Image Layers

□ What Are Image Layers?

Every **Docker image** is made up of **multiple layers** —
each representing an **instruction** in the Dockerfile (FROM, RUN, COPY, etc.).

Each layer is **read-only** and **stacked** on top of the previous one.

□ Think of a Docker image as a **layered cake** □
Each layer adds new ingredients (files, libraries, configurations),
and the final image is a combination of all these layers.

□ How It Works:

When Docker builds an image:

1. It executes **each Dockerfile instruction** step-by-step.
2. After executing each step, it creates a **new immutable layer**.
3. The final image is just a **stack of these layers**.

 Example:

Dockerfile:

```
FROM ubuntu:22.04
RUN apt-get update
RUN apt-get install -y python3
COPY . /app
CMD ["python3", "/app/main.py"]
```

 Layers Created:

Step	Instruction	Description
1	FROM ubuntu:22.04	Base layer (Ubuntu filesystem)
2	RUN apt-get update	Adds update cache files
3	RUN apt-get install -y python3	Adds Python binaries
4	COPY . /app	Adds your app source code
5	CMD ["python3", "/app/main.py"]	Defines runtime instruction

Each layer depends on the previous one — like building blocks .

---

2. Layer Caching in Docker

 What is Build Cache?

When you build an image, Docker **caches** every layer it creates.
If you build the image again and nothing has changed in a step, Docker **reuses** the cached layer instead of rebuilding it.

This makes subsequent builds **much faster**

□ **Example of Caching:**

Suppose you build this Dockerfile once:

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "app.py"]
```

- On first build: all layers are created.
- On second build (if only app.py changed):
  Docker **reuses all cached layers up to RUN pip install**
  and **only rebuilds** the COPY . . and CMD layers.

□ This is why it's efficient to copy dependencies before source code —
it avoids reinstalling dependencies every time you change code.

□ **Command to See Layers:**

```
docker history <image_name>
```

Example:

```
IMAGE         CREATED         CREATED BY                              SIZE
<image_id>    5 minutes ago   CMD ["python" "app.py"]                   0B
<image_id>    5 minutes ago   COPY . .                              50kB
<image_id>    5 minutes ago   RUN pip install -r requirements.txt       120MB
<image_id>    6 minutes ago   FROM python:3.11-slim                    80MB
```

□ **Viewing Layers Physically:**

Each layer is stored in:

```
/var/lib/docker/overlay2/
```

on Linux systems.
They're combined (union filesystem) to make one full container filesystem.

---

3. Why Layers Matter

Benefit	Explanation
Faster Builds	Layers are reused from cache if unchanged
Smaller Updates	Only modified layers need to be rebuilt/pushed
Efficiency	Shared base layers reduce duplication across images
Versioning	You can inspect and debug individual build steps

---

4. Best Practices to Optimize Docker Images

Here are the **pro tips** every DevOps engineer follows

---

1. Order Dockerfile Instructions Smartly

Docker stops caching after a single layer changes.

Good Example:

```
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
```

Here, dependency installation is cached unless requirements change.

Bad Example:

```
COPY . .
RUN pip install -r requirements.txt
```

Here, any code change invalidates the cache and re-installs all dependencies

---

2. Combine RUN Commands

Each RUN instruction creates a new layer.
Combine related commands to reduce total layers.

 Example:

```
RUN apt-get update && \
    apt-get install -y python3 python3-pip && \
    rm -rf /var/lib/apt/lists/*
```

---

3. Use .dockerignore File

Create a .dockerignore file to exclude unnecessary files (logs, node_modules, .git, etc.) from being copied into the image.

 Example .dockerignore:

```
.git
__pycache__
node_modules
*.log
```

This prevents cache invalidation and speeds up builds.

---

4. Use Smaller Base Images

Prefer lightweight images like alpine instead of full OS images.

Language	Full Image	Lightweight Alternative
Python	python:3.11	python:3.11-alpine
Node.js	node:18	node:18-alpine
Java	openjdk:21	eclipse-temurin:21-jre-alpine

 Alpine images are typically **5–10× smaller**!

---

5. Clean Up After Installing Packages

Remove temp files and caches in the same RUN instruction.

Example:

```
RUN apt-get update && apt-get install -y \
    curl \
 && rm -rf /var/lib/apt/lists/*
```

---

6. Multi-Stage Builds (For Production)

Use **multi-stage builds** to copy only the necessary final output (like binaries or build artifacts), not the whole build environment.

□ Example:

```
# Stage 1: Build
FROM node:18 as builder
WORKDIR /app
COPY . .
RUN npm install && npm run build

# Stage 2: Runtime
FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
```

□ This reduces the final image size drastically — only production files remain.

---

7. Tag Images Properly

Use descriptive tags (v1.0, prod, dev) instead of latest everywhere.
It makes cache reuse and deployments more predictable.

8. Docker Volumes

---

1. Why We Need Volumes (Persistent Storage)

□ **Problem:**

By default, **Docker containers are ephemeral** — meaning:

- Any data created **inside a container** is **lost** when:
  - The container stops
  - It's deleted
  - Or rebuilt from a new image

Example:

```
docker run -it ubuntu
# inside container
echo "Hello" > /data.txt
exit
```

When you remove the container and recreate it — /data.txt is gone □

□ **Solution:**

To keep data **persistent** even after container removal, Docker provides **Volumes** and **Bind Mounts**.

These store data **outside the container filesystem**, on the **host system**.

---

2. What is a Volume?

□ **Definition:**

A **Docker Volume** is a **special directory** managed by Docker and stored on the **host machine** (usually under /var/lib/docker/volumes/).

It is designed to:

- **Persist data**
- **Share data** between multiple containers
- **Decouple data** from container lifecycle

□ **Analogy:**

Think of a container as a **temporary office** and a volume as a **filing cabinet** kept outside — even if the office (container) closes, your files (data) remain safe.

---

3. Creating and Managing Volumes

□ **Create a Volume**

```
docker volume create mydata
```

 List All Volumes

```
docker volume ls
```

Example output:

```
DRIVER    VOLUME NAME
local     mydata
```

 Inspect a Volume

```
docker volume inspect mydata
```

Shows details like mount path:

```
"Mountpoint": "/var/lib/docker/volumes/mydata/_data"
```

 Remove a Volume

```
docker volume rm mydata
```

 Prune Unused Volumes

```
docker volume prune
```

---

4. Using Volumes in Containers

 Attach a Volume to a Container

```
docker run -d --name web1 -v mydata:/app/data nginx
```

Explanation:

- -v mydata:/app/data
    - mydata → volume name
    - /app/data → directory inside container where volume is mounted

Now any file written inside /app/data is stored in the volume.

 Verify Persistence

1. Create container and write data:

```
docker exec -it web1 bash
echo "Hello Volume" > /app/data/file.txt
exit
```

2. Remove the container:

```
docker rm -f web1
```

3. Create new container using the same volume:

```
docker run -it --name web2 -v mydata:/app/data ubuntu
cat /app/data/file.txt
```

 You’ll still see:

```
Hello Volume
```

---

5. Bind Mounts vs Volumes

Feature	Bind Mount	Volume
Location	Any path on host (e.g., /home/user/data)	Managed by Docker under /var/lib/docker/volumes/
Created by	User manually	Docker manages automatically
Backup/Restore	Manual	Easy (Docker handles it)
Best for	Sharing code during development	Persisting production data
Performance	Depends on host filesystem	Optimized for Docker
Command Example	-v /host/path:/container/path	-v myvol:/container/path

---

Example: Using Bind Mount

```
docker run -d --name devapp -v /home/prathamesh/app:/usr/src/app node:18
```

Here:

- /home/prathamesh/app → folder on your host
- /usr/src/app → folder inside container

 Any changes on your local files instantly reflect inside container — perfect for development.

---

6. Backing Up and Restoring Data

You can **backup** data from volumes easily using tar archives.

 Backup a Volume

```
docker run --rm \
  -v mydata:/source \
  -v $(pwd):/backup \
  ubuntu tar cvf /backup/mydata_backup.tar /source
```

- /source → mount volume
- /backup → save tar on host

Result: creates mydata_backup.tar on host system.

 Restore a Volume

```
docker run --rm \
  -v mydata:/destination \
  -v $(pwd):/backup \
  ubuntu tar xvf /backup/mydata_backup.tar -C /destination
```

---

7. Sharing Data Between Containers

You can mount the **same volume** into multiple containers.

Example:

```
docker run -d --name web1 -v sharedvol:/data nginx
docker run -d --name web2 -v sharedvol:/data ubuntu tail -f /dev/null
```

Both containers can now **read/write** to /data
→ Useful for logs, uploads, and shared configs.

8. Volume Commands Summary

Command	Description
docker volume create <name>	Create a new volume
docker volume ls	List all volumes
docker volume inspect <name>	Show details of a volume
docker volume rm <name>	Remove a volume
docker volume prune	Delete all unused volumes

9. Best Practices

 Use **named volumes** for data you need to persist.
 Use **bind mounts** for development environments (real-time sync).
 Always use a .dockerignore file to avoid unnecessary host file mounts.
 Regularly **backup important volumes** using tar or external storage.
 Avoid storing sensitive data directly inside containers.

9. Docker Networking

Docker networking allows containers to **communicate with each other**, with the **host machine**, and with **external networks** (like the Internet).
It provides **isolation, flexibility, and control** over how containers connect and exchange data.

Default Networking in Docker

When Docker is installed, it automatically creates a few **default networks**:

Network Name	Type	Description
bridge	Bridge Network	Default network for standalone containers.

Network Name	Type	Description
host	Host Network	Shares the host’s network stack directly.
none	None Network	Disables networking for a container.
overlay	Overlay Network	Used in multi-host or Docker Swarm setups.

---

1. Bridge Network

□ **Description:**

- The **default network** for containers.
- Each container gets its **own IP address** inside an internal bridge (like a virtual switch).
- Containers can **communicate with each other** on the same bridge using their **container names**.

□ **Diagram (conceptually):**

```
Host Machine
 ├── docker0 (bridge network)
 │    ├── container1 (172.17.0.2)
 │    └── container2 (172.17.0.3)
```

□ **Example:**

```
# Run a container using default bridge network
docker run -d --name web1 nginx

# Inspect network
docker network inspect bridge
```

---

2. Host Network

□ **Description:**

- The container **shares the host’s network stack**.
- No virtual network interface is created.
- The container uses **the same IP as the host**.

□ **Use Case:**

- When performance matters (e.g., high-speed networking).
- When container needs to access host network directly.

 Example:

```
docker run -d --network host nginx
```

 Note:

- Works only on **Linux** hosts.
- On Windows/Mac, Docker uses a VM, so --network host behaves differently.

---

3. None Network

 Description:

- The container is **completely isolated** — no network interface except for lo (loopback).
- Useful for **testing** or **security-sensitive** applications.

 Example:

```
docker run -d --network none alpine sleep 1000
```

 Result:

- No external or internal communication — the container runs fully isolated.

---

4. Overlay Network

 Description:

- Used for **multi-host communication** in **Docker Swarm** or distributed applications.
- Allows containers running on different physical/virtual hosts to communicate **securely**.
- Built on top of the **VXLAN protocol**.

 Example (in Swarm mode):

```
docker network create -d overlay my_overlay
docker service create --name web --network my_overlay nginx
```

 Use Case:

- Microservices distributed across multiple Docker hosts.

---

5. Port Mapping (-p option)

When containers run on a **bridge** network, they are isolated from the external world.
You can **map container ports** to host ports using the -p or --publish option.

□ **Syntax:**

```
docker run -d -p <host_port>:<container_port> <image>
```

□ **Example:**

```
docker run -d -p 8080:80 nginx
```

□ **Explanation:**

- 80 → default web server port inside container
- 8080 → port on host system accessible via browser
- Access app via http://localhost:8080

---

6. Linking Containers (Legacy Method)

Before user-defined bridge networks, Docker used **links** to connect containers.
Now, **custom bridge networks** are preferred since they support **automatic DNS-based name resolution**.

□ **Example (old method):**

```
docker run -d --name db mysql
docker run -d --link db:web_db nginx
```

□ Now replaced with:

```
docker network create my_network
docker run -d --name db --network my_network mysql
docker run -d --name web --network my_network nginx
```

□ Now web can access db via container name db.

---

7. Creating and Managing Networks

Commands:

Command	Description
docker network ls	List all networks
docker network create <name>	Create custom network
docker network inspect <name>	View details about a network
docker network connect <network> <container>	Connect container to a network
docker network disconnect <network> <container>	Disconnect container from network
docker network rm <name>	Remove a network

Example:

```
docker network create mynet
docker run -d --name app1 --network mynet nginx
docker run -d --name app2 --network mynet alpine sleep 1000
docker exec -it app2 ping app1
```

Best Practices:

1. Use **user-defined bridge networks** for better control and name resolution.
2. Avoid using --link (deprecated).
3. Assign meaningful network names.
4. For multi-host apps, use **overlay networks** with Docker Swarm or Kubernetes.
5. Use **port mapping** wisely to avoid conflicts.

10. Docker Compose

What is Docker Compose?

Docker Compose is a **tool** that allows you to **define, manage, and run multi-container Docker applications** using a simple YAML configuration file called **docker-compose.yml**.

Instead of running multiple docker run commands manually, you can define **all containers, networks, and volumes** in one file and run everything with a **single command**.

□ Why Use Docker Compose?

Problem	Docker Compose Solution
Managing multiple containers manually	Use one file to configure and run all containers
Complex environment setup	Define app, DB, and cache services in one place
Port conflicts and messy CLI commands	Clean YAML syntax handles everything
Team collaboration	Easily share docker-compose.yml file for same environment setup

□ **In short:** It simplifies multi-container setups, useful for full-stack apps (e.g., Nginx + Backend + Database).

□ docker-compose.yml Structure

The docker-compose.yml file defines how containers should behave.
It includes services, networks, and volumes in a **YAML** format.

□ Basic Structure:

```
version: '3.9'   # Compose file format version

services:        # Define all containers here
  web:           # Container name
    image: nginx
    ports:
      - "8080:80"
    networks:
      - mynet

  db:
    image: mysql:8
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: mydb
    volumes:
      - db_data:/var/lib/mysql
    networks:
      - mynet

volumes:         # Define named volumes
  db_data:

networks:        # Define custom networks
  mynet:
```

□ **Key Sections Explained:**

Section	Description
version	Specifies the Compose file version (recommended: '3' or '3.9').
services	Defines each container/service (like web, db, redis).
image	Docker image to use (from Docker Hub or custom).
build	Path to Dockerfile if you want to build custom image.
ports	Maps host ports to container ports (host:container).
volumes	Persistent storage or file sharing between host and container.
environment	Environment variables (e.g., DB passwords).
depends_on	Defines container dependencies/order of startup.
networks	Custom networks for communication between services.

□ **Example: Multi-Container Web Application**

Here’s an example of a **web + database** app using Compose:

```
version: '3.8'

services:
  app:
    build: .
    ports:
      - "5000:5000"
    depends_on:
      - db
    networks:
      - backend

  db:
    image: mysql:8
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: studentdb
    volumes:
      - db_data:/var/lib/mysql
    networks:
      - backend

volumes:
  db_data:

networks:
  backend:
```

 Explanation:

- **app** → built from Dockerfile, runs backend code
- **db** → MySQL database container
- Both connected via **backend network**
- **db_data** keeps MySQL data persistent even if container is removed

---

Common Docker Compose Commands

Command	Description
docker-compose up	Starts all services (creates containers if not present).
docker-compose up -d	Starts containers in detached mode (background).
docker-compose down	Stops and removes all containers, networks, and volumes created by Compose.

Command	Description
docker-compose build	Builds or rebuilds images defined in the Compose file.
docker-compose ps	Lists all running services.
docker-compose logs	Shows logs of all containers.
docker-compose logs <service>	View logs for a specific service.
docker-compose stop	Stops running containers without removing them.
docker-compose start	Starts existing stopped containers.
docker-compose restart	Restarts all services.
docker-compose exec <service> <command>	Run a command inside a specific service container.

 Examples:

```
docker-compose up -d
docker-compose ps
docker-compose logs app
docker-compose down
```

 How Docker Compose Works (Workflow)

1. **Read YAML file** (docker-compose.yml)
2. **Create networks and volumes**
3. **Build or pull images**
4. **Start containers** based on defined services
5. **Link containers** internally using defined networks

 Advantages of Docker Compose

 Simplified Multi-Container Setup – One file controls all services.
 Consistent Environments – Same setup across dev, test, and production.
 Reusability – Easily share and version control the setup.

 Networking Made Easy – Automatic network creation for service communication.
 Automation – Integrates easily with CI/CD pipelines.

Real-World Example (Full Stack Setup)

```
version: "3.9"
services:
  frontend:
    image: nginx
    ports:
      - "80:80"
    depends_on:
      - backend

  backend:
    build: ./backend
    ports:
      - "5000:5000"
    depends_on:
      - db

  db:
    image: postgres
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: admin
      POSTGRES_DB: myappdb
    volumes:
      - pgdata:/var/lib/postgresql/data

volumes:
  pgdata:
```

 This setup runs a **frontend (Nginx)**, **backend (custom app)**, and **database (PostgreSQL)** with just:

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

Best Practices

1. Use **named volumes** for data persistence.
2. Always specify **version** and **container names** clearly.
3. Store secrets via **.env files** (not inside YAML).

4. Use **depends_on** to manage startup order.
5. Use **networks** for clear isolation between front-end and back-end layers.

11. Docker Registry & Private Repositories

What is a Docker Registry?

A **Docker Registry** is a **storage and distribution system for Docker images**.
It allows you to **push (upload)** and **pull (download)** Docker images.

There are two types:

1. **Public Registries** – like Docker Hub
2. **Private Registries** – custom or self-hosted registries accessible only to specific users or teams.

Docker Hub (Public Registry)

- The **default registry** used by Docker.
- Anyone can **push**, **pull**, and **share images**.
- Images are categorized as:
  - **Official images** (e.g., nginx, mysql, python)
  - **User images** (e.g., prathameshjadhav/myapp)

Example:

```
# Pulling an official image
docker pull nginx

# Pulling a user image
docker pull prathameshjadhav/myapp
```

Private Docker Registry

A **private registry** is your **own repository** where you can securely store images — ideal for internal company use, CI/CD pipelines, or restricted access projects.

You can:

- Host it **locally** (self-managed)
- Use **cloud-based private registries**:

- Docker Hub (Private Repos)
- GitHub Container Registry
- Amazon ECR
- Google Artifact Registry
- Azure Container Registry

---

1. Creating Your Own Local Docker Registry

Docker provides an official **registry image** to set up a private registry on your own host.

 Step 1: Run the Docker Registry Container

```
docker run -d -p 5000:5000 --name myregistry registry:2
```

 Explanation:

- registry:2 → Official Docker image for the registry.
- The registry will now be available at http://localhost:5000.

 Step 2: Tag an Image for Your Registry

```
docker tag nginx localhost:5000/mynginx
```

 This tags the image with your local registry address.

 Step 3: Push Image to Local Registry

```
docker push localhost:5000/mynginx
```

 Now your image is stored in your local registry, accessible at:

http://localhost:5000/v2/_catalog

 Step 4: Pull Image from Registry

```
docker pull localhost:5000/mynginx
```

 Confirms that your private registry works locally.

---

2. Using Private Repository on Docker Hub

You can also **create private repositories** on hub.docker.com.

□ **Step 1: Login to Docker Hub**

```
docker login
```

Enter your Docker Hub **username** and **password/token**.

□ **Step 2: Tag the Image**

```
docker tag myapp prathameshjadhav/myapp:v1
```

□ **Step 3: Push the Image**

```
docker push prathameshjadhav/myapp:v1
```

□ This uploads your image to your **private Docker Hub repository**.

□ **Step 4: Pull the Image**

```
docker pull prathameshjadhav/myapp:v1
```

If the repo is **private**, only logged-in users with access can pull it.

---

3. Private Registry with Authentication (Advanced)

You can secure your local registry using **basic authentication**.

□ **Step 1: Create a Password File**

```
mkdir auth
docker run --entrypoint htpasswd registry:2 -Bbn admin admin123 > auth/htpasswd
```

 Step 2: Run Secure Registry

```
docker run -d \
  -p 5000:5000 \
  --name secure-registry \
  -v $(pwd)/auth:/auth \
  -e "REGISTRY_AUTH=htpasswd" \
  -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" \
  -e "REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd" \
  registry:2
```

 Now your registry requires authentication (admin / admin123).

 Step 3: Login and Push

```
docker login localhost:5000
docker push localhost:5000/myapp
```

---

4. Popular Private Registry Services

Provider	Description	Access Control
Docker Hub	Default Docker registry (free + paid private repos)	User-based
GitHub Container Registry (GHCR)	Linked to GitHub Actions	Token-based
Amazon ECR	AWS Elastic Container Registry	IAM roles/policies
Google Artifact Registry	GCP-based	Google IAM
Azure Container Registry (ACR)	Azure-based	Role assignments

---

5. Useful Docker Registry Commands

Command	Description
docker login	Authenticate to a registry
docker logout	Logout from a registry
docker tag	Rename image for registry
docker push	Upload image to registry
docker pull	Download image from registry
docker search <term>	Search for images on Docker Hub
docker images	List all local images

□ 6. Best Practices

□ Always use **meaningful tags** like :v1, :latest, :prod.
□ Avoid storing secrets inside images.
□ Regularly clean up old images and tags.
□ Use **HTTPS** for secure communication with registries.
□ Automate image pushes with **CI/CD pipelines**.

12. Docker Swarm (Container Orchestration)

□ What is Docker Swarm?

□ Definition:

Docker Swarm is **Docker's native container orchestration tool** that allows you to **deploy, manage, and scale** containers across multiple **Docker hosts (nodes)** working together as a **cluster**.

A **Swarm cluster** looks like one large virtual system, but under the hood, it's a group of multiple Docker engines communicating and cooperating.

□ Why Orchestration Is Needed:

In large-scale systems, you don’t run just one or two containers — you may run **hundreds or thousands**.
Manually starting, stopping, or managing them becomes impossible.

 Container orchestration automates:

- Deployment and scaling
- Load balancing
- Health checks
- Failover and self-healing
- Rolling updates

Docker Swarm and Kubernetes are two popular orchestration systems (Swarm is simpler and Docker-native).

---

1. Swarm Mode Overview

When Docker runs in **Swarm Mode**, it turns a group of Docker engines into a **single managed cluster**.

To enable Swarm mode:

```
docker swarm init
```

This command:

- Initializes a Swarm cluster on the host
- Converts the host into a **manager node**
- Generates a **join token** for other nodes

Key Components:

Component	Description
Node	Individual Docker host (machine) participating in the Swarm.
Manager Node	Controls and manages the cluster — scheduling, health checks, orchestration.
Worker Node	Executes containers as instructed by the manager.

Component	Description
Service	The definition of tasks to run (like an app or microservice).
Task	A single running container instance of a service.
Stack	A collection of multiple services deployed together (via Compose file).

---

2. Manager and Worker Nodes

□ **Manager Node**

- Responsible for:
  - Cluster management
  - Scheduling tasks
  - Maintaining desired state
  - Managing the **Raft consensus**
- Can also run containers.
- Multiple managers can exist for **high availability** (odd number preferred: 3, 5, etc.)

□ **Initialize a Swarm (Manager Node):**

```
docker swarm init
```

□ Output Example:

```
Swarm initialized: current node (abcd1234) is now a manager.
To add a worker to this swarm, run the following command:
docker swarm join --token SWMTKN-1-xyz <manager-ip>:2377
```

□ **Worker Node**

- Executes the actual containers (tasks).
- Does not make scheduling or management decisions.
- Joins the Swarm using a **join token** provided by the manager.

□ **Join as Worker:**

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

□ **View Nodes in Swarm:**

```
docker node ls
```

□ **Output Example:**

```
ID       HOSTNAME     STATUS    AVAILABILITY   MANAGER STATUS
x12abc    manager1     Ready     Active         Leader
y34def    worker1      Ready     Active
z56ghi    worker2      Ready     Active
```

---

3. Services in Swarm

□ **Definition:**

A **service** defines how containers (tasks) should run in a Swarm — including:

- Image name
- Number of replicas
- Ports
- Networks

When you deploy a service, Swarm automatically distributes and manages the required number of containers (tasks) across available nodes.

□ **Example: Creating a Service**

```
docker service create --name webapp -p 8080:80 nginx
```

□ This command:

- Creates a service named webapp
- Runs an nginx container on port 8080
- Manages it automatically (if it fails, Swarm restarts it)

□ **List Services**

```
docker service ls
```

□ **Inspect a Service**

```
docker service ps webapp
```

---

4. Scaling in Swarm

Swarm allows you to easily **scale services up or down** — meaning, increase or decrease the number of running containers (replicas).

 Scale Command:

```
docker service scale webapp=5
```

 This creates 5 replicas of the webapp service across nodes.

To reduce:

```
docker service scale webapp=2
```

 Swarm automatically balances containers among worker nodes.

---

5. Stacks in Swarm

A **stack** is a collection of multiple **services** that work together, often defined using a **Docker Compose file**.

 Example docker-compose.yml

```
version: "3.8"
services:
  web:
    image: nginx
    ports:
      - "8080:80"
  db:
    image: mysql
    environment:
      MYSQL_ROOT_PASSWORD: root
```

 Deploy a Stack in Swarm:

```
docker stack deploy -c docker-compose.yml mystack
```

 Swarm reads the Compose file and creates all services as part of the stack.

 List Stacks

docker stack ls

 View Stack Services

docker stack services mystack

 Remove Stack

docker stack rm mystack

---

6. Load Balancing and Service Discovery

- Swarm provides **built-in load balancing** — automatically routes traffic across container replicas.
- Each service gets an internal **DNS name**, so containers can communicate easily (e.g., db or web).

---

7. High Availability

- Multiple manager nodes ensure redundancy.
- Uses the **Raft consensus algorithm** to maintain a consistent cluster state.
- If one manager fails, another takes over as **leader**.

---

8. Common Swarm Commands

Command	Description
docker swarm init	Initialize a new Swarm
docker swarm join	Add a worker to Swarm
docker node ls	List all nodes
docker service create	Create a new service
docker service ls	List services

Command	Description
docker service scale	Scale a service
docker service rm	Remove a service
docker stack deploy	Deploy services via Compose
docker stack ls	List stacks
docker stack rm	Remove a stack

---

9. Example: Simple Swarm Workflow

```
# 1. Initialize swarm
docker swarm init

# 2. Deploy a service
docker service create --name web -p 8080:80 nginx

# 3. Check running services
docker service ls

# 4. Scale service
docker service scale web=3

# 5. View tasks (containers)
docker service ps web

# 6. Remove service
docker service rm web
```

---

10. Benefits of Docker Swarm

- Easy setup compared to Kubernetes
- Integrated with Docker CLI
- Supports service discovery and load balancing
- High availability and self-healing
- Scales up/down with single command
- Built-in rolling updates

---

11. Limitations of Swarm

 Smaller ecosystem than Kubernetes
 Limited advanced scheduling policies
 Less community support
 Not as feature-rich for large-scale production

13. Docker Networking in Depth

What is Docker Networking?

Docker Networking allows **containers to communicate** with:

- Each other (within same host or across hosts)
- The host machine
- The external world (internet)

Each container has its **own IP address**, isolated network namespace, and connects to a **Docker-managed virtual network**.

---

1. Default Docker Networks

When you install Docker, three networks are automatically created:

Network	**Driver**	**Description**
bridge	bridge	Default network for standalone containers on a single host
host	host	Removes network isolation; container shares host's network stack
none	null	Disables networking entirely
overlay	overlay	Used in Swarm mode for multi-host networking

View All Networks

docker network ls

 Output Example:

```
NETWORK ID     NAME      DRIVER    SCOPE
b1d3f2a6b1f1   bridge    bridge    local
8f2c3e4c9b6a   host      host      local
12d3a4f5e7b8   none      null      local
```

---

2. Bridge Network (Single Host)

Default network for containers on **one Docker host**.
Each container gets an IP from the bridge subnet and can talk to others on the same network.

 Create Custom Bridge Network

```
docker network create --driver bridge mynetwork
```

 Run Containers in It

```
docker run -d --name web1 --network mynetwork nginx
docker run -d --name web2 --network mynetwork nginx
```

Now both containers (web1 and web2) can communicate using **container names** (Docker DNS).

```
docker exec -it web1 ping web2
```

 Output:

```
PING web2 (172.18.0.3): 56 data bytes
64 bytes from 172.18.0.3: icmp_seq=1 ttl=64 time=0.1 ms
```

---

3. Overlay Networks (Multi-Host Communication)

 Definition:

An **Overlay network** connects multiple Docker daemons (on different hosts) together, enabling **container-to-container communication across hosts — securely using VXLAN encapsulation**.

Overlay networks are mainly used in **Swarm Mode**.

 How It Works:

- Each Docker host is part of a **Swarm cluster**.
- The **overlay network** sits **on top of host networks**.

- Docker encrypts container communication automatically.
- Containers on different machines behave as if they're on the same LAN.

Create an Overlay Network

```
docker network create -d overlay my_overlay_net
```

Use It in a Service

```
docker service create --name web --network my_overlay_net -p 8080:80 nginx
```

List Networks

```
docker network ls
```

Output:

```
NETWORK ID     NAME            DRIVER    SCOPE
u3n4t5h6j7k8   ingress         overlay   swarm
v4b5c6d7e8f9   my_overlay_net  overlay   swarm
```

Example: Connecting Containers Across Two Hosts

Host	Role	Container
Host1	Manager	web1
Host2	Worker	web2

Steps:

```
# On Host1 (Manager)
docker swarm init

# On Host2 (Worker)
docker swarm join --token <worker_token> <manager_ip>:2377

# On Manager, create overlay network
docker network create -d overlay global_net

# Deploy containers on the same overlay network
```

```
docker service create --name web1 --network global_net nginx
docker service create --name web2 --network global_net nginx
```

Now web1 (on Host1) and web2 (on Host2) can communicate **securely via overlay network** using DNS names.

---

4. Service Discovery & DNS in Docker

□ **Definition:**

Service discovery is how Docker automatically resolves **container names to IP addresses** using **built-in DNS**.

Every user-defined network includes an **embedded DNS server**.
This allows containers to talk to each other by **name**, not IP.

□ **Example:**

```
docker network create myapp_net

docker run -d --name db --network myapp_net mysql
docker run -d --name backend --network myapp_net my-backend
```

Inside the backend container, the hostname db automatically resolves to the IP of the mysql container.

```
docker exec -it backend ping db
```

□ Output:

```
PING db (172.18.0.3): 56 data bytes
64 bytes from 172.18.0.3: icmp_seq=1 ttl=64 time=0.09 ms
```

□ No need for manual IP configuration — **Docker handles it.**

---

5. Connecting Containers Across Networks

You can attach or detach containers to networks dynamically.

□ **Attach Container to a Network**

docker network connect mynetwork web1

 Detach from Network

docker network disconnect mynetwork web1

---

6. Inspecting Networks

You can view detailed info about a network — its subnet, containers, and settings.

 Inspect Command

docker network inspect mynetwork

 Output Example:

```
[
  {
    "Name": "mynetwork",
    "Driver": "bridge",
    "Containers": {
      "a1b2c3d4e5f6": {
        "Name": "web1",
        "IPv4Address": "172.18.0.2/16"
      }
    }
  }
]
```

---

7. Ingress Network

When you deploy a service with -p in Swarm mode, Docker automatically uses the **ingress network** to:

- Load balance requests across service replicas.
- Expose the service on **all Swarm nodes** (regardless of where container runs).

 Example:

docker service create --name webapp -p 8080:80 nginx

→ Accessible from any node at http://<any-node-ip>:8080

---

8. Network Drivers Summary

Driver	Scope	Description
bridge	Local	Default network for containers on one host
host	Local	Container shares host's network stack
none	Local	No network access
overlay	Swarm	Connects containers across multiple hosts
macvlan	Local	Assigns containers direct access to physical network

---

9. Example: Connecting Containers Across Hosts (Step-by-Step)

Step 1: Initialize Swarm on Manager Node

```
docker swarm init
```

Step 2: Create Overlay Network

```
docker network create -d overlay multi_host_net
```

Step 3: Deploy Services

```
docker service create --name app1 --network multi_host_net nginx
docker service create --name app2 --network multi_host_net httpd
```

Now, app1 and app2 can communicate across **different physical hosts** via multi_host_net.

---

10. Key Docker Networking Commands

Command	Description
docker network ls	List all networks
docker network create <name>	Create a network

Command	Description
docker network rm <name>	Remove a network
docker network inspect <name>	Show detailed info
docker network connect	Attach container to network
docker network disconnect	Detach container from network

---

11. Best Practices

 Prefer **user-defined networks** over default bridge for better DNS and isolation.
 Use **overlay networks** in Swarm for multi-host apps.
 Avoid using **host network** unless necessary (security risk).
 Assign **meaningful names** to networks and services.
 Regularly prune unused networks:

```
docker network prune
```

14. Docker Security

Introduction

Docker provides **container-level isolation** — each container runs in its own environment.
However, security risks arise when:

- Containers share host resources.
- Images contain vulnerabilities.
- Secrets (passwords, API keys) are exposed.

Docker includes several **built-in mechanisms** and best practices to secure containers, images, and data.

---

1. Managing User Permissions

□ Default Behavior

By default, Docker requires **root privileges** to run because it interacts directly with:

- Kernel namespaces
- cgroups
- File systems and network interfaces

This can be risky because:

If an attacker compromises a container running as root, they might gain access to the **host system**.

□ Best Practices for User Permissions

1. Add your user to the Docker group (non-root access)

On Linux:

```
sudo usermod -aG docker $USER
```

Then log out and back in.

Now you can run Docker commands without sudo.

---

2. Run containers as non-root users

You can define a specific user inside the **Dockerfile**:

```
FROM ubuntu:latest
RUN useradd -ms /bin/bash appuser
USER appuser
CMD ["bash"]
```

□ Prevents containers from running as root internally.

---

3. Use user namespaces

User namespaces map **container users to non-root host users**, increasing security.

Enable it in /etc/docker/daemon.json:

```
{
  "userns-remap": "default"
}
```

Restart Docker:

```
sudo systemctl restart docker
```

---

 4. Restrict Docker socket access

- The Docker daemon socket (/var/run/docker.sock) gives **root-level access** to the system.
- Avoid mounting it directly inside containers (very common mistake in CI/CD).

 Bad example:

```
-v /var/run/docker.sock:/var/run/docker.sock
```

 Use **Docker context** or remote APIs instead.

---

2. Image Vulnerability Scanning

 Why?

Images may include outdated libraries or packages with known CVEs (Common Vulnerabilities and Exposures).
Scanning ensures you only deploy **secure, trusted images**.

 Built-in Docker Scan

Docker integrates with **Snyk** for image scanning.

 Scan an Image

```
docker scan nginx:latest
```

 Output Example:

```
Testing nginx:latest...
```

✗ High severity vulnerability found in openssl
Description: OpenSSL Denial of Service
Fix: upgrade to 1.1.1n or later

□ Scan All Local Images

```
docker scan --file Dockerfile .
```

□ Third-Party Scanning Tools

Tool	Description
Trivy	Fast, open-source vulnerability scanner for containers
Clair	Static analysis of vulnerabilities in images
Anchore	Deep policy-based scanning and compliance checks
Snyk	Integrated security platform for developers

□ ***Example with Trivy:***

```
trivy image myapp:latest
```

□ Reports vulnerabilities in OS packages and dependencies.

□ Best Practices for Image Security

□ Use **official base images** (e.g., python:3.10-slim, ubuntu:22.04).
□ Keep images updated with RUN apt-get update && apt-get upgrade -y.
□ Avoid unnecessary packages and reduce image size.
□ Always scan before pushing to registry.

---

3. Secrets Management

□ What Are Secrets?

"Secrets" refer to **sensitive data** like:

- API keys
- Passwords
- Certificates
- SSH keys

Storing them in plain text inside images or environment variables is unsafe □

□ Docker Secrets (Swarm Mode)

Docker provides a secure way to store and use secrets — encrypted at rest and in transit.

Step 1: Create a Secret

```
echo "MySQL@123" | docker secret create db_password -
```

Step 2: Deploy a Service Using the Secret

```
docker service create --name dbservice \
  --secret db_password \
  mysql:latest
```

Step 3: Access Secret Inside Container

Secrets are mounted inside the container at:

```
/run/secrets/<secret_name>
```

□ Example:

```
docker exec -it <container_id> cat /run/secrets/db_password
```

Output:

```
MySQL@123
```

□ Step 4: List and Remove Secrets

```
docker secret ls
docker secret rm db_password
```

 Using Secrets in Non-Swarm Containers (Alternative Methods)

1. **Environment Variables (less secure):**
2. docker run -e DB_PASSWORD=MySQL@123 mysql:latest

    Visible via docker inspect.

3. **Bind Mount (file-based secrets):**
4. docker run -v /secure/secrets/db_password:/run/secrets/db_password mysql:latest

    Safer than env vars, but ensure proper file permissions.

---

4. Additional Docker Security Features

Feature	Purpose
Seccomp profiles	Restrict system calls available to containers
AppArmor / SELinux	Enforce kernel-level access control
Read-only file systems	Prevent unwanted writes inside containers
Resource limits (cgroups)	Prevent resource abuse (CPU, memory)
Capabilities	Limit container privileges (drop unnecessary ones)

 Example: Run Container with Dropped Privileges

```
docker run --cap-drop=ALL --cap-add=NET_ADMIN alpine
```

 Drops all Linux capabilities except network admin rights.

---

5. Docker Bench for Security (Auditing Tool)

Docker Bench automatically checks security best practices.

 Run it:

```
docker run -it --net host --pid host --cap-add audit_control \
  -v /var/lib/docker:/var/lib/docker \
  -v /etc:/etc \
  docker/docker-bench-security
```

It generates a detailed security audit report for your Docker environment.

---

6. Docker Security Best Practices Summary

Area	Best Practice
User Permissions	Run as non-root user, enable user namespaces
Image Security	Use official images, scan with Trivy/Snyk
Secrets	Use Docker Secrets (Swarm) or encrypted files
Daemon Access	Limit /var/run/docker.sock mounts
Kernel Security	Use Seccomp, AppArmor, and cgroups
Regular Audits	Run Docker Bench Security

15. Docker Logs & Monitoring

Docker logs and monitoring help you understand:

- What's happening **inside containers**
- How much **CPU, memory, network, I/O** apps are using
- Detect **errors**, **performance issues**, or **failures**

---

1. Viewing Container Logs

Every Docker container writes logs to a logging driver.
By default, Docker uses the **json-file** driver (stores logs on host).

□ Basic Log Commands

□ 1. View Logs of a Running Container

```
docker logs <container_name_or_id>
```

□ 2. Follow Logs (Real-Time Tail)

```
docker logs -f <container>
```

Same as tail -f.

□ 3. Show Last N lines

```
docker logs --tail 100 <container>
```

□ 4. Show Timestamps

```
docker logs -t <container>
```

□ Example

```
docker logs -ft --tail 50 webapp
```

Shows last 50 log lines with timestamps and continuous updates.

□ Where Logs Are Stored (Default)

For Linux:

```
/var/lib/docker/containers/<container-id>/<container-id>-json.log
```

---

5. Changing Default Logging Driver

Docker supports multiple logging drivers:

Driver	Purpose
json-file	Default
local	Efficient local storage
syslog	Send logs to syslog daemon
journald	Integrates with systemd
awslogs	AWS CloudWatch
gelf	Graylog
fluentd	Fluentd logging system

 Example: Run Container with syslog

```
docker run --log-driver=syslog nginx
```

---

2. Docker Monitoring Overview

Monitoring shows:

- CPU usage
- Memory consumption
- Network traffic
- Block I/O
- Container performance
- Node-level metrics

Docker does **not** provide advanced monitoring out of the box →
so we use tools like:

 cAdvisor
 Prometheus
 Grafana

---

3. Monitoring with cAdvisor (Container Advisor)

What is cAdvisor?

- A Google open-source tool
- Monitors **container-level** metrics:
  - CPU
  - Memory
  - Network
  - Filesystem
- Provides a **web UI dashboard**
- Exposes metrics to **Prometheus**

Install cAdvisor

```
docker run -d \
  --name=cadvisor \
  --volume=/:/rootfs:ro \
  --volume=/var/run:/var/run:ro \
  --volume=/sys:/sys:ro \
  --volume=/var/lib/docker/:/var/lib/docker:ro \
  -p 8080:8080 \
  gcr.io/cadvisor/cadvisor:v0.47.0
```

Access cAdvisor UI

Open browser:

http://localhost:8080

You can view:

- Real-time CPU & memory graphs
- Container lists
- Per-container usage metrics

---

4. Prometheus (Time-Series Monitoring)

What is Prometheus?

A metrics-based monitoring tool used for:

- Collecting container metrics
- Storing time-series data
- Alerting
- Querying metrics using **PromQL**

Prometheus scrapes metrics from exporters (like cAdvisor).

□ Sample Prometheus Docker Compose

```
version: '3'

services:
  prometheus:
    image: prom/prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"

  cadvisor:
    image: gcr.io/cadvisor/cadvisor:latest
    ports:
      - "8080:8080"
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:ro
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
```

□ Prometheus Config (prometheus.yml)

```
scrape_configs:
  - job_name: 'cadvisor'
    static_configs:
      - targets: ['cadvisor:8080']
```

□ Access Prometheus

http://localhost:9090

 Sample queries:

- CPU usage:
- container_cpu_usage_seconds_total
- Memory usage:
- container_memory_usage_bytes

---

5. Grafana (Visualization Dashboard)

 What is Grafana?

A tool for creating dashboards from Prometheus data.

Provides:
 Beautiful graphs
 Dashboards
 Alerts
 Real-time container metrics

 Run Grafana

```
docker run -d -p 3000:3000 --name=grafana grafana/grafana
```

Default login:

- **Username:** admin
- **Password:** admin

Then add **Prometheus** as a data source:

 Data source config:

- URL: http://prometheus:9090

 Import Docker / cAdvisor Dashboard

Grafana has official dashboards:

- Dashboard ID: **193** → Docker Monitoring
- Dashboard ID: **10619** → cAdvisor + Prometheus Full Monitoring

---

6. Complete Monitoring Stack Setup (One Command)

Here's a ready-to-run **docker-compose.yaml** that deploys:
- Prometheus
- cAdvisor
- Grafana

```
version: '3'

services:

  cadvisor:
    image: gcr.io/cadvisor/cadvisor:latest
    ports:
      - "8080:8080"
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:ro
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro

  prometheus:
    image: prom/prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"

  grafana:
    image: grafana/grafana
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
```

Start all services:

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

---

7. Docker Stats Command (Basic Monitoring)

Quickly check CPU, RAM, and I/O usage:

```
docker stats
```

Output:

```
CONTAINER ID   NAME     CPU %   MEM USAGE / LIMIT    NET I/O
abc123        web1     1.23%   100MiB / 1GiB         10kB/5kB
```

---

8. Best Practices for Docker Logging & Monitoring

- Redirect logs to **centralized systems** (ELK, Splunk, Fluentd).
- Use **health checks** to restart unhealthy containers.
- Install **cAdvisor + Prometheus** for metrics.
- Use **Grafana** for visualization.
- Limit log file size:

```
docker run --log-opt max-size=10m --log-opt max-file=3 nginx
```

- Use **external log drivers** for production.