Docker

P by Praveen Kumar

What is Virtualization?

Virtualization is a technique that allows you to create **multiple virtual environments (Virtual Machines)** on a **single physical system**.

Key points

- Uses a **hypervisor** (VMware, VirtualBox, Hyper-V).
- Each VM has its own **Operating System (OS)**.
- Multiple OSes run on the same hardware.
- Uses virtual CPU, RAM, Disk allocated from physical machine.

Problems with Virtual Machines

Although VMs are useful, they have limitations:

Heavyweight

Each VM contains a full OS → consumes more RAM, CPU, disk.

Slow Startup

Booting a full OS takes time (minutes).

Inefficient Resource Usage

Many duplicated OS kernels running on the same hardware.

App Packaging Issue

Apps inside a VM depend on the OS version and configurations.

Difficult to Move

VM images are large (GBs), hard to share.

What are Containers?

Containers are **lightweight**, **portable** environments that package an application with everything it needs—**without needing a full OS**.

Key points

- Share the **host OS kernel**.
- Start in **milliseconds**.
- Very small in size (MBs instead of GBs).
- Same behavior across machines ("It works on my machine" solved).
- Container Engines: **Docker**, **containerd**, **podman**.

Containers vs Virtual Machines

Feature	Virtual Machines (VMs)	Containers
OS	Each VM has its **own full OS**	Share **host OS kernel**
Size	Heavy (GBs)	Lightweight (MBs)
Startup Time	Slow (minutes)	Fast (seconds/milliseconds)
Resource Usage	High	Low
Portability	Difficult	Highly portable
Performance	Moderate	Almost near native

What is Docker?

Docker is a **containerization platform** that allows developers and DevOps teams to:

- Build applications as **images**
- Run applications as **containers**
- Share containers easily
- Ensure applications run the same everywhere

Docker provides:

- A container engine
- Tools to build, run, ship containers
- Docker Hub for image distribution

Why Docker Became Popular?

Lightweight

Containers are extremely small and fast.

Very Fast Deployment

Boots in seconds → perfect for microservices.

Consistency

“Works on my machine” problem solved.

Easy Packaging

Everything needed (app + dependencies) is packaged in the image.

DevOps Friendly

Perfect for CI/CD pipelines.

Scalability

Suitable for microservices and orchestration (Kubernetes).

Portability

Run the same container on: Windows, Linux, Mac, Cloud, On-premise

Docker Architecture Overview

Docker architecture follows a **client-server model** with these components:

1. **Docker Client**
2. **Docker Daemon (dockerd)**
3. **Docker Host**
4. **Docker Objects** (Images, Containers, Volumes, Networks)
5. **Docker Registry** (Docker Hub/private)

Docker Client

This is the **command-line tool** or UI used to interact with Docker.

You use:

```
docker run
docker build
docker pull
docker stop
```

Client sends the request to the **Docker Daemon** via REST API.

Docker Host

The machine where Docker is installed.

It contains:

- Docker Daemon
- Containers
- Images
- Networks
- Volumes

Examples of Docker Hosts:

- Your laptop
- Cloud machine (AWS EC2)
- Linux server

Docker Daemon (dockerd)

Docker Daemon is the **brain** of Docker.

Responsibilities:

- Build images
- Run containers
- Manage networks
- Manage volumes
- Communicate with Docker Hub
- Listen for API requests from Docker Client

Acts like a background service.

Docker Registry (Docker Hub)

A **registry** stores Docker **images**.

Docker Hub:

- Public registry by Docker Inc.
- Contains official images like **nginx**, **mysql**, **node**, **python**.

Private Registries:

- AWS ECR
- GitHub Container Registry
- Azure ACR
- Google GCR

Docker Images

A **Docker Image** is a **read-only blueprint** used to create containers.

Key characteristics

- Contains:
  - Application code
  - Runtime (Python/Node/Java)
  - Libraries
  - System tools
- Built using a **Dockerfile**
- Images are made up of **layers**
  - Each instruction (FROM, COPY, RUN) creates a new layer
- Images are immutable (cannot be changed once created)

Docker Containers

A **Docker Container** is a **running instance** of a Docker Image.

Key characteristics

- Lightweight and fast
- Includes:
  - App binaries
  - Dependencies
- Uses **copy-on-write** for filesystem changes
- Has its own:
  - Process space
  - Network interface
  - File system

Docker Registry, Repository, Tags

Docker Registry

A registry is a **storage system for Docker images**.

Examples:

- Docker Hub (public)
- AWS ECR
- GitHub Container Registry
- Azure ACR

Docker Repository

A **repository** is a **collection of images for the same application** with different versions.

Basic Docker Information Commands

Command	Description
docker --version	Shows installed Docker version
docker version	Shows Client & Daemon version details
docker info	Displays Docker system information (containers, images, storage, etc.)

Docker Image Commands

Command	Description
docker images	List all images available locally
docker pull <image>	Download image from Docker Hub/Registry
docker build -t <name> .	Build image using Dockerfile
docker rmi <image>	Remove an image
docker image prune	Remove unused images
docker inspect <image>	Shows detailed metadata of an image
docker tag <src> <target>	Tag an image with a new name/version

Docker Container Commands

Command	Description
`docker run <image>`	Creates + starts a container
`docker run -d <image>`	Run container in detached (background) mode
`docker ps`	List running containers
`docker ps -a`	List all containers (including stopped ones)
`docker start <container>`	Start a stopped container
`docker stop <container>`	Stop a running container
`docker restart <container>`	Restart container

`docker kill <container>`	Force stop the container
`docker rm <container>`	Remove a stopped container
`docker logs <container>`	Show container logs
`docker exec -it <container> bash`	Execute a command inside running container
`docker inspect <container>`	Detailed container information
`docker stats`	Real-time resource usage of containers

Docker Volume Commands

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

Docker Network Commands

Command	Description
`docker network ls`	List available Docker networks
`docker network create <name>`	Create a custom network
`docker network inspect <name>`	Show network details
`docker network rm <name>`	Remove a network
`docker run --network=<name>`	Run container in a specific network

What is a Dockerfile?

A **Dockerfile** is a simple text file containing a **set of instructions** used to build a Docker Image.

It automates the steps:

- install software
- copy source code
- set environment variables
- run commands
- expose ports
- define how the container starts

In simple words:

Dockerfile = Recipe
Image = Cake
Container = A piece of cake you eat (running instance)

Why Dockerfile?

- Reproducible builds
- Portable across environments
- Easy to version control
- Removes manual steps
- Standard way for CI/CD

Basic Structure of a Dockerfile

A Dockerfile is a list of instructions executed **from top to bottom**.

Example:

```
FROM python:3.10
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
EXPOSE 5000
CMD ["python", "app.py"]
```

Dockerfile Instructions

FROM - Every image starts from a base image (like python, ubuntu, node, nginx).

WORKDIR - Sets the working directory inside the container.

COPY - Copies files from your machine → to container.

ADD - Similar to COPY but supports:

- URL downloads
- Automatic extraction of tar files

RUN - Runs a command **during image build**.

CMD - Defines the command that runs when the container starts.

ENTRYPOINT - Similar to CMD but **cannot be overridden easily**.

EXPOSE - Tells Docker which port the container will use.

ENV - Sets environment variables inside the container.

VOLUME - For persistent data storage.

Building an Image Using Dockerfile

Step 1: Create Dockerfile

Step 2: Build the image:

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

Step 3: Run a container:

```
docker run -p 5000:5000 myapp
```

Simple Python App (Flask)

Dockerfile:

```
FROM python:3.10
WORKDIR /app
COPY . .
RUN pip install flask
EXPOSE 5000
CMD ["python", "app.py"]
```

Node.js App

Dockerfile

```
FROM node:18
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

Best Practices

✔️ Use small images (like `alpine`)
✔️ Keep the number of layers small
✔️ Use `.dockerignore`
✔️ Pin version numbers (avoid latest)
✔️ Use multi-stage builds for production
✔️ Combine RUN commands
✔️ Do not install unnecessary packages

What Are Docker Volumes?

A **Docker Volume** is a special storage mechanism used by Docker to **persist data** generated and used by containers.

Because containers are **temporary**, once a container is removed:

- All data stored inside the container is **lost**.

To prevent data loss, Docker provides **Volumes**.

✔️ Summary (easy definition)

> **Volumes store data outside the container's filesystem so the data will not be deleted when the container is removed.**

Why Do We Need Volumes?

Containers are:

- *Ephemeral* (temporary)
- *Stateless* by default

If you store data inside a running container (like in `/var/lib/mysql` for MySQL), deleting or recreating the container deletes all the data.

Example:

- Run MySQL container
- Store 10 rows
- Delete container
- All data gone ❌

To avoid this → **use a volume** ✔️

Benefits of Docker Volumes

Benefit	Explanation
Data Persistence	Data survives even after container removal
Sharing Data	Multiple containers can access the same volume
Better Performance	Faster I/O than bind mounts
Backup & Restore	Easy to back up volume directories
Decoupling Data from Containers	Safe upgrades without losing data

Types of Docker Storage

Docker provides **3 types of storage**:

1️⃣ Volumes (Recommended)

- Managed by Docker
- Stored under `/var/lib/docker/volumes/`
- Best for production
- Works on Linux, Windows, Mac equally

2️⃣ Bind Mounts

- Maps a folder from the host machine → into container
- Used for local development
- Example: Mapping source code from your laptop

```
docker run -v C:/code:/app node
```

3️⃣ tmpfs Mounts

- Data stored in RAM only
- Fastest but temporary
- Good for sensitive data

Types of Docker Volumes

A) Anonymous Volumes

Created automatically when you use `-v /path`.
Not easy to reference again.

Example:

```
docker run -v /app/data nginx
```

B) Named Volumes (most important)

Explicitly created by the user.

Example:

```
docker volume create mydata
docker run -v mydata:/var/lib/mysql mysql
```

Where Are Volumes Stored?

On Linux:

```
/var/lib/docker/volumes/<volume_name>/_data/
```

Docker Volume Commands

1. Create a volume docker volume create myvolume
2. List volumes docker volume ls
3. Inspect a volume docker volume inspect myvolume
4. Remove volume docker volume rm myvolume
5. Remove unused volumes docker volume prune

Named Volume Exam ple

Step 1: Create a volume

```
docker volume create mydata
```

Step 2: Run a container using the volume

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

This mounts the volume **mydata** inside the container at **/app/data**.

Step 3: Copy some data into the container

Step 3: Copy some data into the container

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

Step 4: Stop & remove the container

```
docker rm -f mycontainer
```

Step 5: Run another container with same volume

```
docker run -it --name newcontainer -v mydata:/app/data ubuntu bash
```

Inside this new container:

```
cat /app/data/file.txt
```

You will see your data!
Volumes persist even after container deletion.

Anonymous Volume Example

This creates a volume **without a name**:

Run container

```
docker run -d --name testcontainer -v /app/data nginx
```

Docker automatically creates a volume with a random name (e.g., `73f8d2aca91a34...`)

Dockerfile Example Using Volumes

Create a file named **Dockerfile**:

```
FROM ubuntu:latest
RUN mkdir /appdata
VOLUME ["/appdata"]
CMD ["bash"]
```

Build the image:

```
docker build -t volume-demo .
```

Run the container:

```
docker run -it --name vol1 volume-demo
```

Inside container:

```
docker exec -it <containername> /bin/bash
cd /appdata
echo "Hello Dockerfile Volume" > test.txt
```

Exit container and remove it:

```
docker rm -f vol1
```

Run another container:

```
docker run -it --name vol2 volume-demo
```

✔️ Your file `test.txt` is still present, because Docker automatically creates an anonymous volume for `/appdata`.

What is Docker Compose?

Docker Compose is a tool that allows you to **define and run multi-container Docker applications** using a single configuration file called:

```
docker-compose.yml
```

With Compose, you can:

- Start multiple containers together
- Configure networks & volumes automatically
- Link services together using service names
- Scale services
- Stop and remove everything with one command

Why Docker Compose?

Without Docker Compose, if you want to run a multi-container app (e.g., web + database + cache), you would manually run commands like:

```
docker run -d -p 8080:80 webapp
docker run -d mysql
docker run -d redis
...
```

This becomes **complicated** and **unmanageable** as the number of containers grows.

With Docker Compose:

```
docker compose up
```

Everything starts automatically based on the compose file.

Docker Compose Architecture (Simple Diagram)

```
 docker-compose.yml
        |
        |  (reads configuration)
        ↓
+-----------------------------+
|    Docker Compose CLI       |
+-----------------------------+
        |
        ↓
+-----------------------------+
|      Docker Engine          |
+-----------------------------+
        |
        ↓
Services → Networks → Volumes → Containers
```

docker-compose.yml Structure Overview

A Compose file has the following sections:

```
version: "3.9"

services:
  serviceA:
    image: ...
    ports:
    volumes:
    environment:
    depends_on:
    networks:

networks:
  mynetwork:

volumes:
  myvolume:
```

omplete Example 1 — Basic Web + Database (Very Simple)

File: `docker-compose.yml`

```
version: "3.9"

services:
  web:
    image: nginx
    ports:
      - "8080:80"

  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: root123
```

Run app:

```
docker compose up
```

Run in background:

```
docker compose up -d
```

Stop app:

```
docker compose down
```

Complete Example 2 — Web + MySQL + Volume + Network

This is the **most commonly used structure**.

```
version: "3.9"

services:

  backend:
    build: ./backend
    container_name: backend-app
    ports:
      - "5000:5000"
    depends_on:
      - mysql
    environment:
      DB_HOST: mysql
      DB_USER: root
      DB_PASS: root123
    networks:
      - mynet
```

```yaml
  mysql:
    image: mysql:8
    container_name: mysql-db
    environment:
      MYSQL_ROOT_PASSWORD: root123
      MYSQL_DATABASE: testdb
    volumes:
      - mysql_data:/var/lib/mysql
    networks:
      - mynet

volumes:
  mysql_data:

networks:
  mynet:
```

Useful Docker Compose Commands

Command	Description
`docker compose up`	Start all services
`docker compose up -d`	Start in background
`docker compose down`	Stop + remove containers + network
`docker compose down -v`	Remove volumes also
`docker compose ps`	List running services
`docker compose logs`	View logs of all services
`docker compose logs backend`	Logs for a single service

`docker compose stop`	Stop but do not remove
`docker compose start`	Start previously stopped services
`docker compose build`	Rebuild images
`docker compose exec backend bash`	Enter a container
`docker compose restart`	Restart services

Life Cycle of Docker Compose (Flow)

```
docker-compose.yml
        ↓
docker compose up
        ↓
Creates networks → Creates volumes → Starts containers → Connects them
```