A Comprehensive Guide to Docker

Davoud Azari

Contents

Introduction

What is Docker?

Docker is an **open-source containerization platform** that allows developers to **build, package, deploy, and run applications** in **lightweight, portable, and isolated containers**. Unlike virtual machines (**VMs**), containers share the **host operating system kernel**, making them **efcient, fast, and scalable**.

Wikipedia defines Docker as:

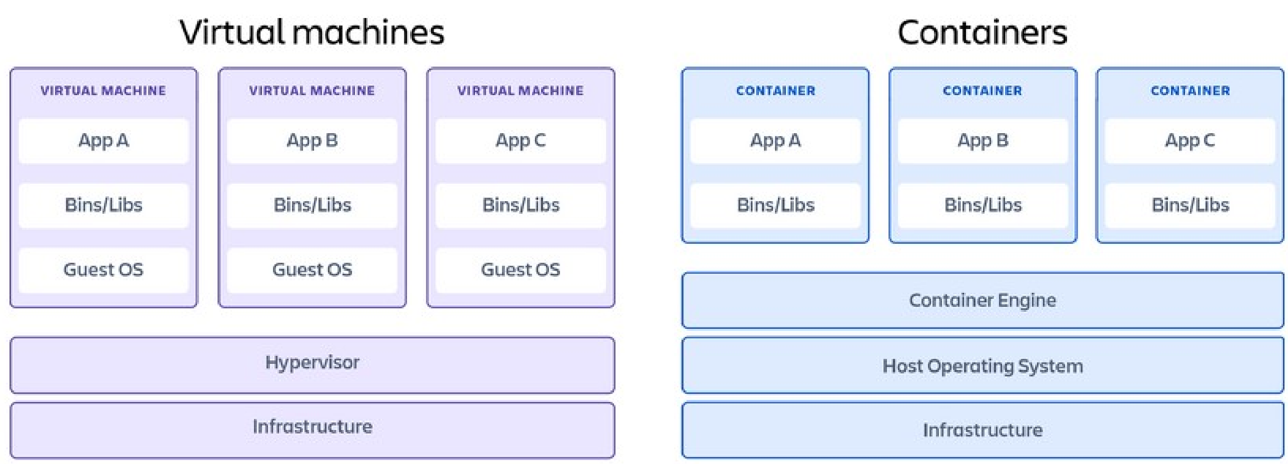
An open-source project that automates the deployment of software applications inside containers by providing an additional layer of abstraction and automation of OS-level virtualization on Linux.

Quite a mouthful, isn't it? Simply put, **Docker is a powerful containerization platform** that enables developers, system administrators, and DevOps engineers to package, deploy, and run applications in lightweight, isolated environments known as **containers**. These containers operate on the host operating system (typically Linux) **without the overhead of traditional virtual machines**, making them far more efficient.

Why Use Docker?

- **Portability:** Run containers anywhere—on any OS, cloud, or local machine.
- **Scalability:** Easily scale applications up or down.
- **Efciency:** Uses fewer system resources than virtual machines.
- **Speed:** Containers start in milliseconds.
- **Security:** Process-level isolation enhances security.

Containers vs. Virtual Machines



Feature	Virtual Machines	Containers
Boot Time	Minutes	Seconds
Size	GBs	MBs
Performance	Lower (VM overhead)	Near-native speed
Isolation	Full OS-level isolation	Process-level isolation
Portability	Limited	High

This image illustrates the key differences between **Virtual Machines (VMs)** and **Containers**.

Structure and Key Differences:

Virtual Machines rely on a **hypervisor** to manage multiple guest operating systems. Each VM includes its own **guest OS**, binaries, and libraries, making them **heavier** and requiring more **resources**.

Containers, on the other hand, share the **host operating system** and run on a **container engine**. They **do not require a full OS** for each instance, making them **lighter, faster, and more efficient**.

Feature Comparison (Table):

Boot Time – VMs take minutes, while containers start in seconds.
Size – VMs are in GBs, while containers are much smaller, typically in MBs.
Performance – VMs have higher overhead, whereas containers offer near-native speed.
Isolation – VMs provide full OS-level isolation, while containers use process-level isolation.
Portability – Containers are highly portable, whereas VMs have limited portability.

Overall, **containers are a more lightweight and scalable alternative** to VMs, making them ideal for **modern DevOps, cloud computing, and microservices architectures**.

Key Features of Docker

Lightweight: Containers share the host OS kernel.

Rapid Deployment: Starts and stops in seconds.
Integrated Ecosystem: Works with Docker Hub, Compose, Swarm, and Kubernetes.

Installing Docker

Installation on Linux (Ubuntu)

Set up Docker's apt repository.

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "${UBUNTU_CODENAME:-$VERSION_CODENAME}") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

Install the Docker packages.

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

Installation on macOS

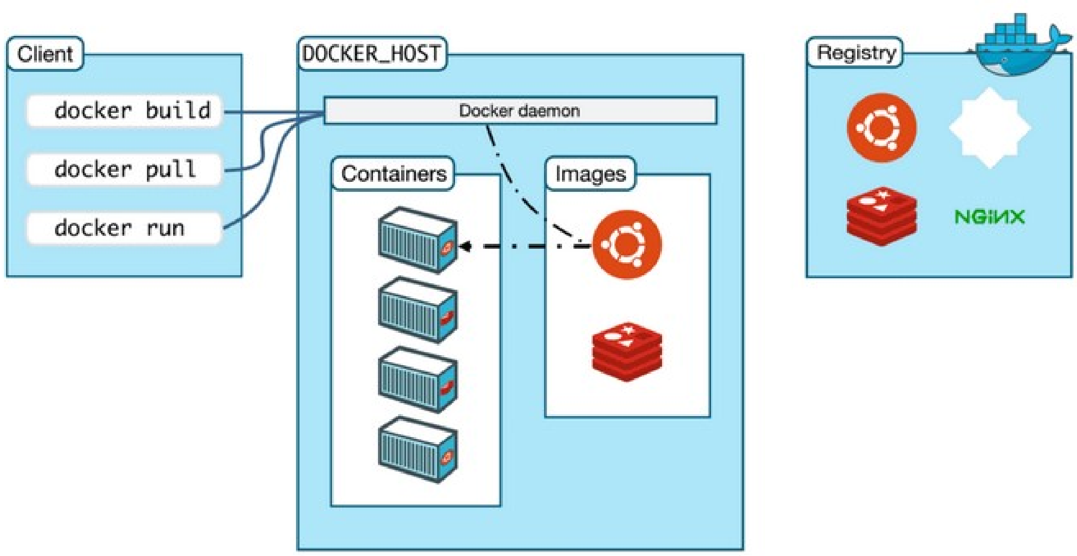
- Download and install **Docker Desktop** from the ofcial Docker website.
- Start Docker Desktop.

Installation on Windows

- Download **Docker Desktop** and install it.
- Ensure **WSL 2 Backend** is enabled.

Verifying Installation

```
docker --version
```



Understanding Docker Architecture

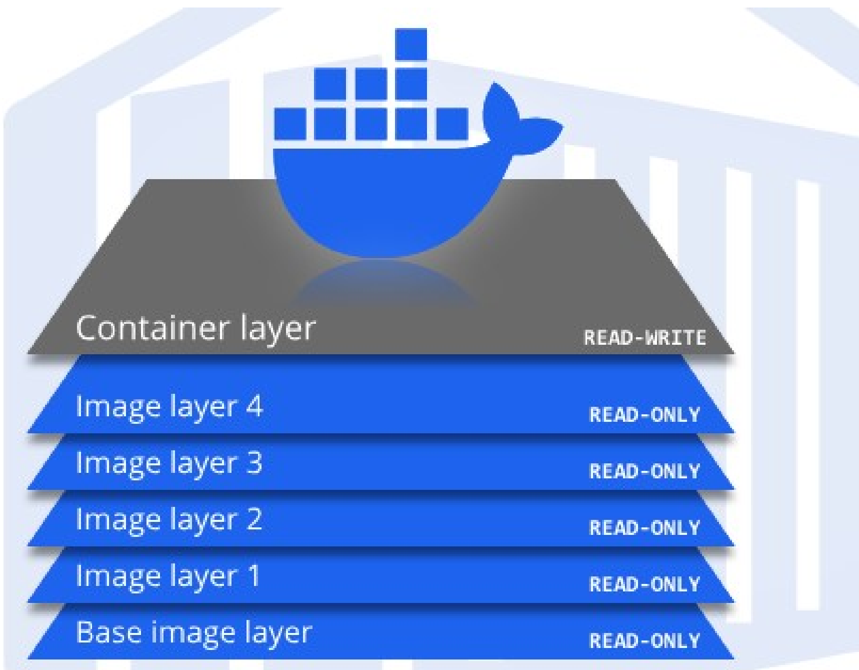
Docker Engine & Docker Daemon

- Docker Engine: Core component responsible for running containers.
- Docker Daemon (dockerd): Runs in the background and manages container lifecycle

There are five major components in the Docker architecture:

- **a) Docker Daemon** listens to Docker API requests and manages Docker objects such as images, containers, networks and volumes.
- **b) Docker Clients:** With the help of Docker Clients**,** users can interact with Docker. Docker client provides a command-line interface (CLI) that allows users to run, and stop application commands to a Docker daemon.
- **c) Docker Host** provides a complete environment to execute and run applications. It comprises of the Docker daemon, Images, Containers, Networks, and Storage.
- **d) Docker Registry** stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to use images on Docker Hub by default. You can run your own registry on it.
- **e) Docker Images** are read-only templates that you build from a set of instructions written in Dockerfile. Images define both what you want your packaged application and its dependencies to look like what processes to run when it's launched.

Docker Images vs. Containers

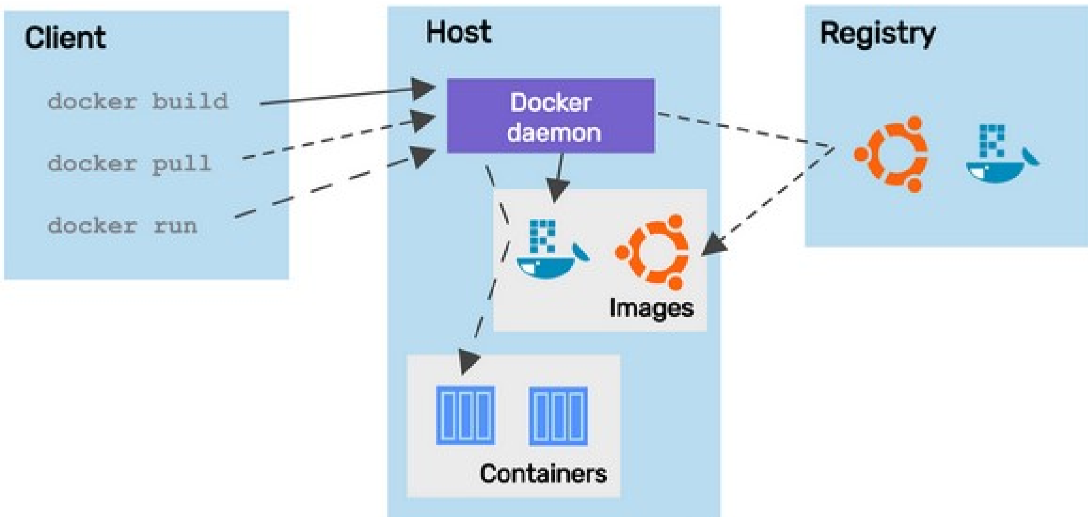


Concept	Docker Image	Docker Container
Defnition	A static blueprint of a container	A running instance of an image
State	Read-only	Read-write
Persistence	Stored in a registry	Temporary, unless persistent storage is used
Lifecycle	Created once, used multiple times	Created from an image, runs, stops, and can be deleted

Docker Registry & Repository

Docker Hub: The default public repository for Docker images.

Private Registry: Used to store and distribute private images securely.



Essential Docker Commands

Container Management

```
docker run <image>              # Create and start a new container
docker ps                       # List running containers
docker ps -a                    # List all containers (including stopped ones)
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
```

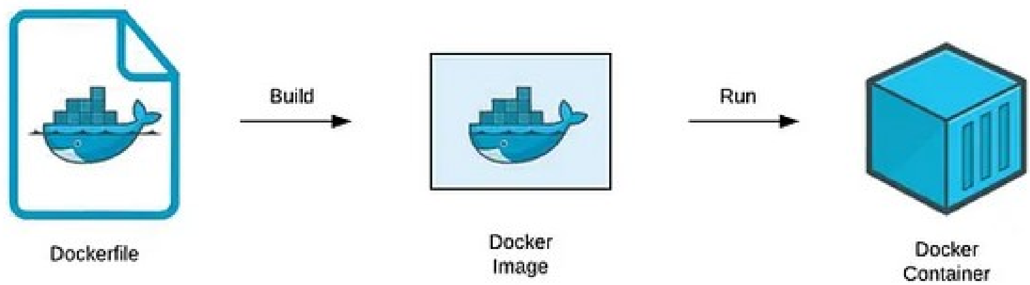
Image Management

```
docker images                   # List all images
docker rmi <image_id>           # Remove an image
docker pull <image>             # Download an image from Docker Hub
docker build -t my_image .      # Build an image from a Dockerfile
```

Logging & Debugging

```
docker logs <container_id>                # View logs   of a container
docker inspect <container_id>             # Get detailed information about a container
docker exec -it <container_id> bash       #   Open a terminal in a running container
docker system prune -a                    #   Clean up unused images and containers
```

Building Docker Images (Dockerfile)



A **Dockerfile** is a script that contains **instructions** to **build a Docker image**. It automates the process of defning a container environment, ensuring **consistency across diferrent deployments**.

Understanding Dockerfile Structure

A typical **Dockerfile** follows these steps:

* Define a base image

* Set environment variables and working directory

* Copy necessary files into the image

* Install dependencies

* Expose ports if needed

* Define commands to run at container startup

Instruction	Description	Example
FROM	Defnes the base image	`` `FROM ubuntu:latest` ``
WORKDIR	Sets the working directory inside the container	`` `WORKDIR /app` ``
COPY	Copies fles from the host machine to the container	`` `COPY . /app` ``
ADD	Similar to COPY, but supports extracting tar fles and remote URLs	`` `ADD myfile.tar.gz /data` ``
RUN	Executes commands inside the container **during build time**	`` `RUN apt-get update && apt-get install -y curl` ``
CMD	Specifes the default command **to run the container**	`` `CMD ["python", "app.py"]` ``
ENTRYPOINT	Defnes an executable that always runs when the container starts	`` `ENTRYPOINT ["nginx", "-g", "daemon off;"]` ``
EXPOSE	Specifes the container's listening port	`` `EXPOSE 80` ``
ENV	Sets environment variables	`` `ENV APP_ENV=production` ``
ARG	Defnes build-time variables	`` `ARG VERSION=1.0` ``
LABEL	Adds metadata to the image	`` `LABEL maintainer="John Doe"` ``
VOLUME	Creates a mount point for persistent storage	`` `VOLUME /data` ``
USER	Switches to a specifc user inside the container	`` `USER appuser` ``
HEALTHCHECK	Defnes a command to check if the container is running properly	`HEALTHCHECK CMD curl -f http://localhost
ONBUILD	Executes instructions when a dependent image is built	`` `ONBUILD COPY . /app` ``

Basic Dockerfle Example

```
# 1. Define a base image
FROM python:3.8

# 2. Set the working directory
WORKDIR /app

# 3. Copy application files into the container
COPY . .

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

# 5. Expose an application port
EXPOSE 5000

#   6. Define the default command

CMD ["python", "app.py"
```

Understanding RUN, CMD, and ENTRYPOINT Diferences

Instruction	Purpose	Runs at Build Time or Runtime?
RUN	Executes commands **when building the image**	**Build time**
CMD	Defnes a default command **when running the container**	**Runtime**
ENTRYPOINT	Defnes a mandatory command that always runs	**Runtime**

Example of CMD vs ENTRYPOINT

```
# Using CMD
CMD ["echo", "Hello World"]   # Can be overridden

# Using ENTRYPOINT
ENTRYPOINT ["echo", "Hello World"]   # Cannot be overridden without --entrypoint flag
```

Understanding Dockerfle Stages

There are **two main types** of Dockerfle builds:

Single-Stage Build

Everything is built in **one step**, resulting in **a large fnal image**.

Multi-Stage Build

Reduces the **final image size** by separating the **build environment** from the **runtime environment**.

Multi-Stage Build Example

A **multi-stage build** helps **reduce the fnal image size** by separating the **build** and **runtime** environments.

```
# Stage 1: Build Stage
FROM golang:1.18 AS builder
WORKDIR /app
COPY . .
RUN go build -o my_app

# Stage 2: Runtime Stage
FROM alpine:latest
WORKDIR /root/
COPY --from=builder /app/my_app .
CMD ["./my_app"]
```

Using ARG and ENV in Dockerfle

- **ARG**: Defnes variables available only **at build time**.
- **ENV**: Defnes environment variables **inside the running container**.

```
# Use an ARG to define a build-time variable
ARG VERSION=1.0

# Use ENV to set runtime environment variables
ENV APP_ENV=production
```

Passing ARG value when building an image

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

Best Practices for Writing Dockerfles:

Use a minimal base image (`` `alpine` `` instead of `` `ubuntu` `` for smaller image size).
Leverage multi-stage builds to optimize image size.

Use `` `.dockerignore` `` to avoid unnecessary fles in the image.

Use `` `RUN apt-get update && apt-get install -y` `` **in one line** to reduce layers.

Specify user permissions (`` `USER appuser` ``) for security.

Building and Running a Docker Image

Once the **Dockerfle** is ready, you can build and run the container:

Build the Image

```
docker build -t my_python_app .
```

Run the Container

```
docker run -d -p 5000:5000 my_python_app
```

List Created Images

```
docker images
```

Remove an Image

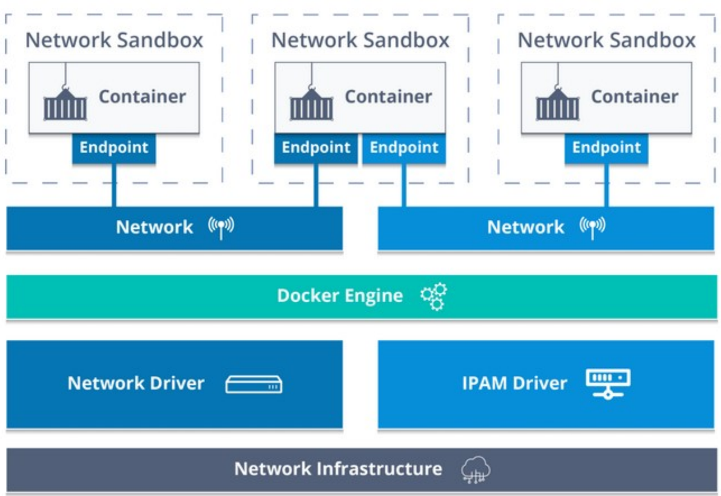
```
docker rmi my_python_app
```

Common Dockerfle Issues and Fixes

Issue	Cause	Solution
Large Image Size	Using a large base image	Use `alpine` or `distroless` images
Layer Caching Issues	Changing layers in the wrong order	Copy dependencies frst, then install packages
Permission Issues	Running as root	Use `USER` to specify a non-root user
Long Build Times	Too many `RUN` commands	Combine multiple commands in a single `RUN`

Summary

- **A Dockerfle** automates the process of creating a **Docker image**.
- **CMD vs ENTRYPOINT:** CMD defnes the default command, while ENTRYPOINT enforces it.
- **Multi-Stage Builds** optimize image size and performance.
- **Best practices** include using minimal base images, `.dockerignore`, and reducing layers.
- **Common issues** include large image sizes, permission problems, and inefcient caching.



Networking in Docker

Docker provides built-in **networking capabilities** that allow **containers to communicate** with each other and the external world. Understanding Docker networking is crucial for **running multi-container applications, exposing services, and ensuring security**.

Types of Docker Networks

Network Type	Description
Bridge	Default network mode for standalone containers. Containers on the same bridge network can communicate with each other but not with external systems.
Host	The container shares the host's network stack (no network isolation). It provides **high performance** but reduces security.
Overlay	Used for **multi-host networking**, typically in **Docker Swarm mode**, allowing containers across multiple nodes to communicate.
Macvlan	Assigns a **MAC address** to a container, allowing it to appear as a physical device on the network. Used for **direct network access**.
None	Completely **disables networking**, making the container isolated from other networks.

Managing Docker Networks

List Available Networks

```
docker network ls
```

Creating a Custom Network

```
docker network create my_custom_network
```

Running a Container on a Custom Network

```
docker run -d --network my_custom_network --name web nginx
```

Connecting an Existing Container to a Network

```
docker network connect my_custom_network my_existing_container
```

Inspecting Network Details

```
docker network inspect my_custom_network
```

Removing a Network

```
docker network rm my_custom_network
```

Using Bridge Network (Default)

By default, Docker assigns **containers** to a **bridge network**. Containers within the same bridge network can **communicate with each other**, but they cannot **access external networks unless explicitly connected**.

```
docker run -d --name container1 --network bridge nginx
docker run -d --name container2 --network bridge alpine sleep 1000
```

Using Host Network (No Isolation)

When a container runs on the **host network**, it directly shares the **host's network interfaces**.

```
docker run --rm --network host -p 8080:8080 nginx
```

Faster networking performance
Less security (since containers share the host's network stack)

Using Overlay Network (Multi-Host Communication)

Overlay networks are used in **Docker Swarm** for multi-node communication.

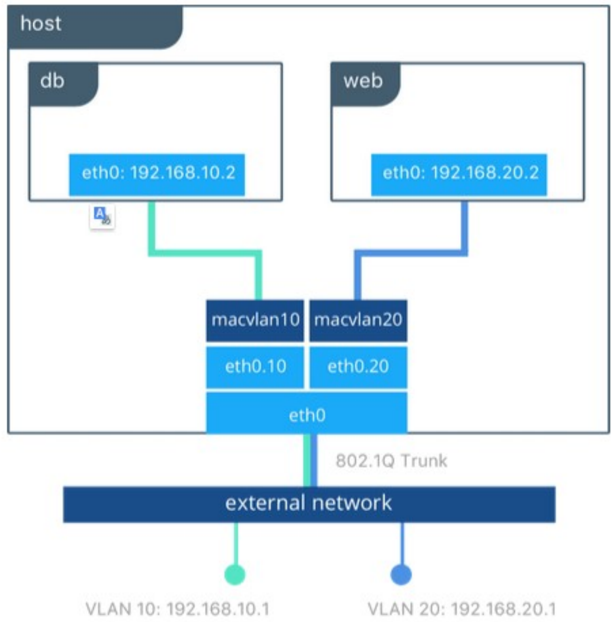
```
docker network create --driver overlay my_overlay_network
```

Used in **multi-node Docker Swarm deployments**
Enables **cross-host container communication**

Macvlan Networking

Macvlan network is used to connect applications directly to the physical network. By using the macvlan network driver to **assign a MAC address** to each container, also allow having full TCP/Ip stack. Then, the **Docker daemon routes traffic** to containers by their MAC addresses. You can isolate your macvlan networks using different physical network interfaces. This is used in legacy applications which require MAC address.

Using Macvlan Network (Direct Access to Physical Network)



Macvlan allows **containers to have their own MAC addresses** and appear as separate devices on the network.

```
docker network create -d macvlan \
  --subnet=192.168.1.0/24 \
  --gateway=192.168.1.1 \
  -o parent=eth0 my_macvlan
```

Used for **connecting directly to a physical network**
Allows **external devices to communicate** with containers

Exposing a Container to External Networks

To allow external systems to access a container:

Expose a Port When Running a Container

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

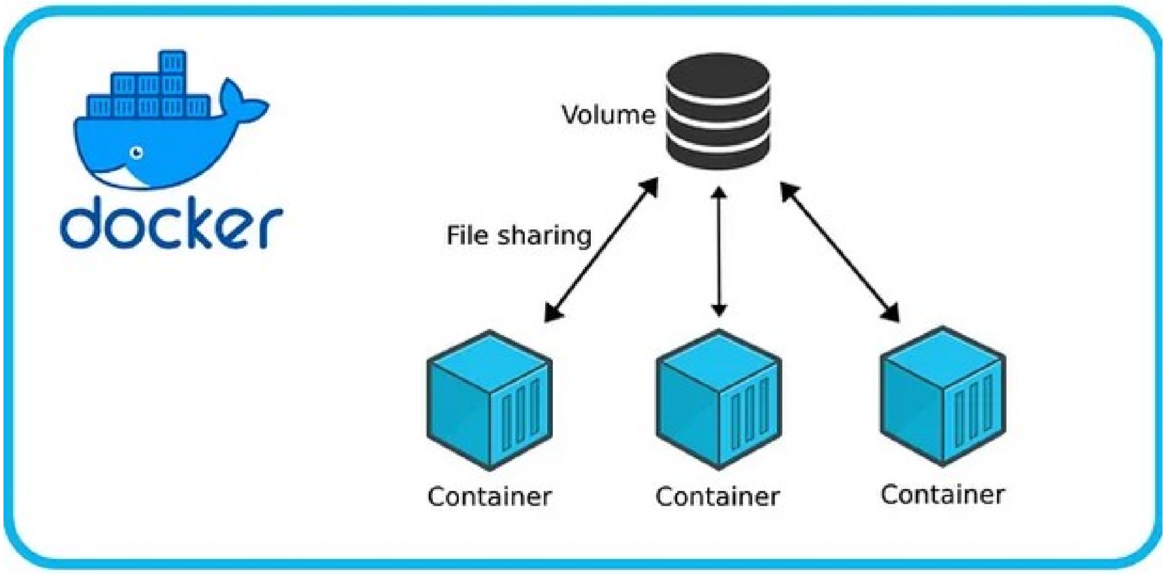
Maps **port 8080 on the host** to **port 80 inside the container**.

Publishing Multiple Ports

```
docker run -d -p 5000:5000 -p 8080:80 nginx
```

Summary

- **Docker networking** enables **container-to-container** and **container-to-host** communication.
- **Bridge networks** are **default** but isolated from external networks.
- **Host networks** remove isolation but expose security risks.
- **Overlay networks** allow **multi-host communication** in **Swarm mode**.
  Macvlan networks make containers behave like **physical network devices**.



Volumes and Persistent Storage in Docker

By default, Docker containers are **ephemeral**, meaning any data stored **inside** the container is lost when the container stops or is removed. To persist data, Docker provides **volumes** and **bind mounts**, which allow **data storage outside the container's flesystem**.

Why Use Persistent Storage?

Data Persistence: Prevents data loss when a container stops or restarts.
Sharing Data: Allows multiple containers to access the same storage.

Performance Optimization: Volumes are **faster** than bind mounts for managing container storage.
Backups & Portability: Easily backup and move data across environments.

Types of Storage in Docker

Storage Type	Description
Volumes	Managed by Docker and stored **outside the container's flesystem** in `` `/var/lib/docker/volumes/` ``. Best for **persistent data storage**.
Bind Mounts	Maps a **host machine directory** to a container. Useful for **sharing fles between host and container** but less portable.
tmpfs Mounts	Stores data **in memory (RAM)** instead of disk. Used for **sensitive data that shouldn't be written to disk**.

Managing Docker Volumes

Creating a Volume

```
docker volume create my_volume
```

Listing Available Volumes

```
docker volume ls
```

Running a Container with a Volume

```
docker run -d -v my_volume:/data nginx
```

The **container writes data** to **`/data`** inside the container.
The data **remains even if the container is removed**.

Inspecting a Volume

```
docker volume inspect my_volume
```

Removing a Volume

```
docker volume rm my_volume
```

Using Bind Mounts

Bind mounts allow **direct access to the host machine's flesystem**.

Running a Container with a Bind Mount

```
docker run -d -v /home/user/app:/usr/src/app nginx
```

The directory **`/home/user/app`** from the **host** is mapped to **`/usr/src/app`** in the **container.**

Any **changes made inside the container refect on the host system**.

Read-Only Bind Mounts (For Security)

```
docker run -d -v /home/user/app:/usr/src/app:ro nginx
```

The `:ro` fag makes the volume **read-only**, preventing the container from modifying fles.

Using tmpfs Mounts (In-Memory Storage)

tmpfs stores **data in RAM** instead of disk, making it faster but **non-persistent**.

Running a Container with a tmpfs Mount

```
docker run -d --tmpfs /data nginx
```

The `/data` directory will be stored **in RAM** instead of the disk.

Data is **lost when the container stops**.

Backing Up and Restoring Volumes

Backup a Docker Volume

```
docker run --rm -v my_volume:/data -v $(pwd):/backup busybox tar -czvf /backup/backup.tar.gz /data
```

Creates a compressed **backup fle** of the volume.

Restore a Volume from a Backup

```
docker run --rm -v my_volume:/data -v $(pwd):/backup busybox tar -xzvf /backup/backup.tar.gz -C /dat
```

Restores data from the **backup archive** into the volume.

Cleaning Up Unused Volumes

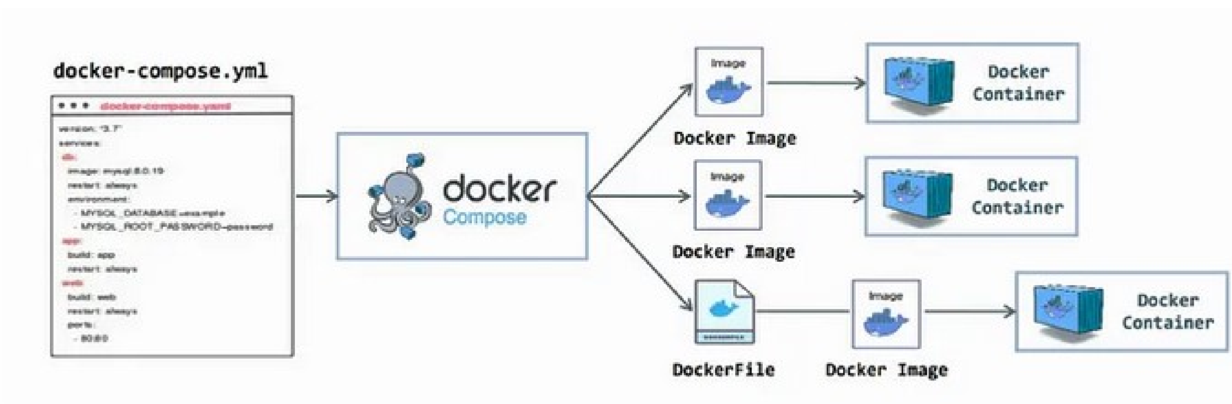
To remove **all unused Docker volumes**, run:

```
docker volume prune
```

Helps **free up disk space** by deleting unused volumes.

Summary

- **Docker Volumes** are **persistent**, managed by Docker, and stored outside the container's flesystem.
- **Bind Mounts** provide **direct host-to-container access**, but are **less portable**.
- **tmpfs Mounts** store data in **RAM**, making them **fast but non-persistent**.
  Backup and Restore techniques ensure **data protection and portability**



Docker Compose

When working with **complex applications** that require multiple **containers**, manually managing them with `` `docker run` `` can become **time-consuming and error-prone**. **Docker Compose** simplifes multi-container applications by allowing you to defne services, networks, and volumes in a **single confguration fle** (`` `docker-compose.yml` ``).

Why Use Docker Compose?

Docker Compose is a tool that makes it easy to run multiple containers at once. It allows you to define all the containers, networks, and volumes for your application in a single file. This file is called a "docker-compose.yml" file.

- Simplified Multi-Container Management – Define and manage all services in a single file (docker-compose.yml).
- Portability – Easily replicate and deploy environments across different machines.
- Scalability – Scale services up or down with a single command (docker-compose up --scale).
- Automated Networking & Storage – Automatically sets up container networks and persistent storage, ensuring seamless communication and data persistence.

Installing Docker Compose

Docker Compose is **included in Docker Desktop** (Windows/macOS) but must be installed manually on Linux.

Installing on Linux

```
sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)
sudo chmod +x /usr/local/bin/docker-compose
docker-compose --version
```

Verify Installation:

```
docker-compose --version
```

Writing a `docker-compose.yml` File

A **Compose fle** defnes **services, networks, and volumes** in **YAML format**.

Example: Running a Web App with Nginx and PostgreSQL

```yaml
version: '3'
services:
  web:
    image: nginx
    ports:
      - "8080:80"
    volumes:
      - web_data:/usr/share/nginx/html
    networks:
      - my_network

  db:
    image: postgres
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass
    volumes:
      - db_data:/var/lib/postgresql/data
    networks:
      - my_network

volumes:
  web_data:
  db_data:

networks:
  my_network:
```

Restart All Services

```
docker-compose restart
```

Scale a Service

```
docker-compose up --scale web=3
```

Runs **3 instances** of the `web` service.

Rebuild Services

```
docker-compose up --build
```

Adding a Load Balancer with Docker Compose

To distribute traffic **across multiple containers**, you can use **Nginx as a load balancer**.

Example: Load Balancing Across Multiple Web Containers

```yaml
version: '3'
services:
  nginx:
    image: nginx
    ports:
      - "8080:80"
    depends_on:
      - web1
      - web2
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
    networks:
      - my_network

  web1:
    image: my_web_app
    networks:
      - my_network

  web2:
    image: my_web_app
    networks:
      - my_network

networks:
  my_network:
```

Nginx acts as a **reverse proxy**, distributing traffic **between `web1` and `web2`**.

Best Practices for Docker Compose

- Use **environment variables** instead of hardcoded values.
- Store sensitive credentials in **Docker secrets or `.env` files**.

- Use `depends_on` to control service startup order.
- Use **named volumes** for **persistent data storage**.
- Avoid **exposing unnecessary ports** for better security.

Summary

- **Docker Compose** simplifies multi-container **application management**.
- Services, networks, and volumes are defined in `docker-compose.yml`.
- Use `docker-compose up -d` to launch all services.
- Supports **scaling, logging, and service dependencies**.
  Load balancing can be implemented with **Nginx and multiple web services**.