🚀 Docker: Complete Guide From Beginner to Advanced (For DevOps Engineers)

🐳 What is Docker?

Docker is a platform that helps you **build, package, ship, and run applications inside containers** — lightweight, isolated environments.

📌 Think of a *container* as a "mini virtual machine" that contains:
✓ Application code
✓ Runtime
✓ Dependencies
✓ Configurations

It runs *anywhere* — laptop, server, cloud.

⭐ Why Docker? (Top Benefits)

- 🔹 Works on "Build Once, Run Anywhere"
- 🔹 Super fast deployments
- 🔹 Less resource usage than VMs
- 🔹 Better isolation
- 🔹 Easy CI/CD integration
- 🔹 Reproducible environments

🧱 What is a Container?

A **lightweight, standalone, executable** package that includes:
✓ Application code
✓ Runtime
✓ System tools
✓ Libraries
✓ Settings

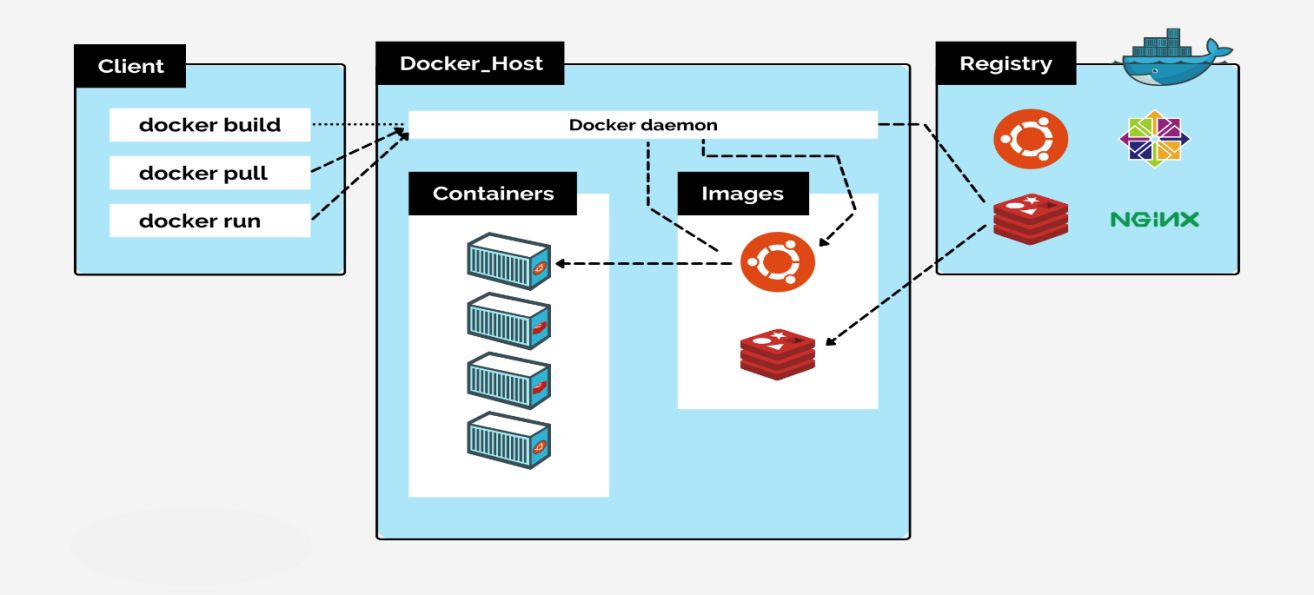
🎯 Why are containers lightweight?

Because they **share the host OS kernel**, unlike Virtual Machines which require a full OS.
➡️ Faster startup
➡️ Lower resource usage
➡️ Smaller images (Ubuntu base image ~22MB vs. VM ~2GB)

📦 What is Containerization?

Containerization = Packaging code + dependencies → **Container Image**

Docker Architecture (Simple Visual)



Docker Architecture

Docker Client → sends commands
Docker Daemon → executes commands
Docker Registry → stores images (e.g., Docker Hub)
Docker Host → runs containers

Docker Components

Docker Daemon → Runs containers
Docker Client → CLI (docker)
Dockerfile → Instructions to build image
Images → Read-only templates
Containers → Running instances
Volumes → Persistent storage
Networks → Communication layer

Docker Lifecycle

1. Write Dockerfile
2. Build Image → docker build
3. Run Container → docker run
4. Push Image → docker push

Image vs Container (Simple Explanation)

Image → A read-only blueprint (recipe)
Container → A running instance of that image (actual meal)

🐳 Docker Installation Commands

Ubuntu

```
sudo apt update
sudo apt install docker.io -y
sudo systemctl enable --now docker
```

Verify Installation

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

🐳 🚀 Docker Commands

🔵 1. Basic Management & Information

📌 docker version

Shows versions of Docker **client** and **server (daemon)**.

📌 docker info

Displays detailed system-level info:

- Containers
- Images
- Storage driver
- Logging driver
- Memory, CPU usage

📌 docker ps

Lists **running containers**.

📌 docker ps -a

Lists **all containers**, including stopped ones.

📌 docker search <image>

Searches Docker Hub for images.

🟢 2. Container Operations

▶️ docker run <image>

Creates a new container *and runs it*.

▶️ docker run -d <image>

Runs a container in **detached (background)** mode.

▶️ docker start <container>

Starts an **existing stopped** container.

🛑 **docker stop <container>**

Gracefully stops a running container.

🗑️ **docker rm <container>**

Deletes a **stopped** container.

⌨️ **docker exec <container> <command>**

Runs a command inside a **running container**.
Example:

docker exec -it nginx bash

📜 **docker logs <container>**

Displays logs of a container.

🔎 **docker inspect <container>**

Shows detailed JSON metadata:

- IP address
- Mounts
- Environment variables
- Runtime details

🟣 **3. Image Management**

⬇️ **docker pull <image>**

Downloads an image from Docker Hub.

🏗️ **docker build -t .**

Builds an image from a Dockerfile.

🏷️ **docker tag <image_id> :**

Retags an image for pushing:

docker tag f12abc prakash14306/app:v1

⬆️ **docker push :**

Pushes an image to Docker Hub.

❌ **docker rmi <image>**

Removes an image from the local system.

🟤 **4. Volume Management**

📦 **docker volume create <name>**

Creates a persistent volume.

📦 **docker volume ls**

Lists all volumes.

📦 **docker volume rm <name>**

Deletes a volume.

📦 **docker volume inspect <name>**

Shows volume metadata (mount path, scope, driver, etc.)

🔴 **5. Network Management**

🌐 **docker network ls**

Lists existing networks (bridge, host, none).

🌐 **docker network create <name>**

Creates a custom network.

🌐 **docker network rm <name>**

Deletes a network.

🌐 **docker network connect**

Adds a container to a network.

🌐 **docker network disconnect**

Removes a container from a network.

🌐 What is Docker Networking?

When you run Docker containers, they often need communication:

Containers may need to:

- talk to **other containers** (app → database)
- access **internet** (API calls, apt-get update)
- expose ports to **host system** or clients

Docker creates **virtual switches (bridges)** and **virtual network cards** for containers to enable this.

🧐 When is Docker Networking Used?

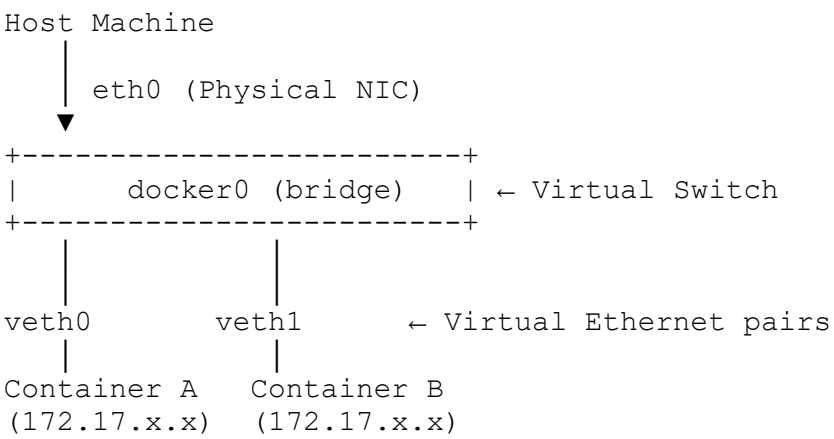
✓ Multi-container apps
Example: **Frontend ↔ Backend ↔ Database**
✓ Containers requiring internet
✓ Exposing container ports to outside users
✓ Internal communication inside same host or across multiple hosts

🔵 1. Bridge Network (Default)

What is Bridge Network?

A private internal network created by Docker.
Containers inside the same bridge can communicate using **container names** as DNS.

📌 **Visual Diagram**



Example

```
docker network create my-bridge-net
docker run -d --name app --network my-bridge-net nginx
docker run -d --name db --network my-bridge-net mongo
```

Container-to-container communication:
App can reach DB using

```
db:27017
```

✓ **When to Use?**

- Local development
- Need isolated container communication
- Default choice for most setups

🔴 2. Host Network

What is it?

The container uses the **host's network directly**.
No NAT, no virtual network — highest speed.

Example:

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

Now nginx runs directly on host's port 80.

✓ **Advantages**

- Very high performance
- No port mapping required
- Useful for monitoring tools, logging agents, Prometheus exporters

✕ **Disadvantages**

- No network isolation
- Port conflicts

- Lower security (shares host's network namespace)

✓ Use Case

- High-performance apps
- Tools that need direct access to host network

🔍 Bridge Networking — Summary

✓ Advantages

- Default & simple
- Isolated
- DNS-based container discovery
- Custom bridge = more control

✕ Disadvantages

- NAT overhead
- Cannot span multiple hosts
- Requires port mapping for external access

🔍 Host Networking — Summary

✓ Advantages

- Zero overhead
- Best performance
- Direct host access

✕ Disadvantages

- No isolation
- Port conflicts
- Less secure

🧩 Understanding Key Concepts

📌 What is eth0?

`eth0` is the **primary network interface** of a Linux system.
This is how the host connects to the LAN or internet.

📌 What is docker0?

A **virtual Linux bridge** automatically created by Docker.

It works like a virtual switch:

- Assigns IP ranges (default: `172.17.0.1`)
- Connects container network interfaces
- Enables container-to-container communication

🎯 Why Do We Need Docker Volumes? (Problem Statement)

By default:

- A Docker container's **filesystem is temporary**.
- When a container stops → **all inside data is deleted**.
- Example: files inside `/var/lib/mysql` in a MySQL container disappear after container deletion.

This is a major issue for databases, application logs, user uploads, etc.

✅ Solution Provided by Docker

Docker provides **two ways** to persist data beyond container lifetime:

1️⃣ Volumes (Recommended)

Managed by Docker under `/var/lib/docker/volumes/`.

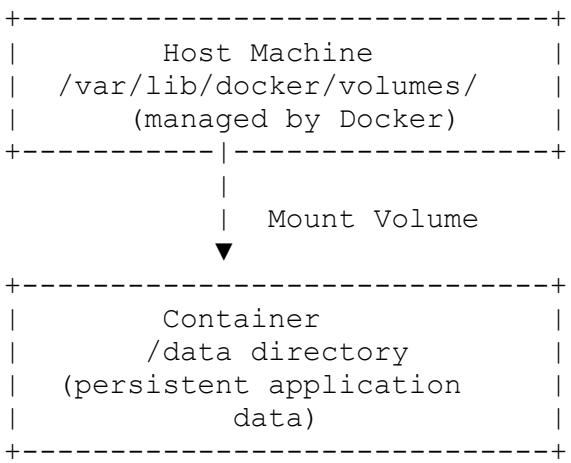
2️⃣ Bind Mounts

Mounting a **host machine's directory** into the container.

📦 1. Docker Volumes (Recommended for Production)

Docker **volumes** store data separately from container's filesystem.
When the container is removed → **data remains**.

📌 Visual Diagram



🛠 Create a Volume

```
docker volume create myvolume
```

🛠 Use the Volume in a Container

Option 1: Using -v

```
docker run -it -v myvolume:/data ubuntu /bin/bash
```

Option 2: Using --mount (newer & clearer)

```
docker run -it --mount source=myvolume,target=/data ubuntu /bin/bash
```

🔍 How it Works?

Anything saved inside `/data` **persists**, even after:

✓ Container stop
✓ Container remove
✓ New container using same volume

💡 Real-life Example (MySQL container)

MySQL stores DB files at `/var/lib/mysql`.

```
docker run -d \
  -v mysql-data:/var/lib/mysql \
  mysql
```

Container can be recreated any number of times but data remains safe in the volume.

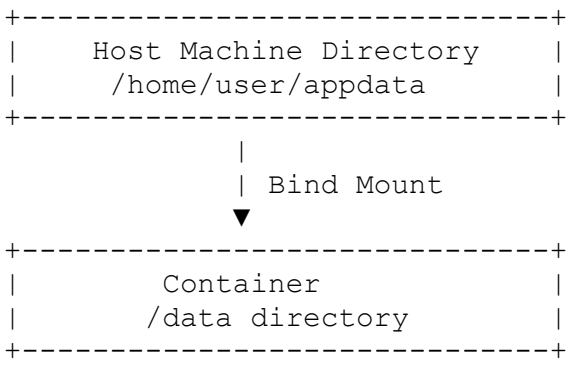
🟡 2. Bind Mount (Host Directory → Container Directory)

Bind mounts allow you to map a **specific folder on your host machine** to a container.

Example:

```
docker run -it -v /home/user/appdata:/data ubuntu
```

📌 Visual Diagram



Everything you change inside the container is also changed directly on the host.

🔥 Volumes vs Bind Mounts — Key Differences

Feature	Volume	Bind Mount
Storage Location	`/var/lib/docker/volumes/`	Any host directory
Managed by Docker	✓ Yes	❌ No
Backup/Restore	Easy	Manual
Portability	High	Low
Recommended for production	✓ Yes	❌ Not always
Good for development	Medium	✓ Yes

🎯 Which One Should I Use?

Use Volumes when:

- Running databases
- Running production workloads
- You want Docker-managed, secure, isolated storage

Use Bind Mounts when:

- You want to edit source code live (dev work)
- You need to access host data directly
- You want full control of paths

📝 Dockerfile

A **Dockerfile** is a blueprint used to build Docker images.
It defines:
✓ Base image
✓ Dependencies
✓ Application code
✓ Commands to run app
✓ Build optimizations

🟦 1. Simple Dockerfile

```
FROM nginx
COPY index.html /usr/share/nginx/html/
```

What Happens Here?

- **FROM nginx** → Uses official Nginx image
- **COPY index.html** → Replaces default page with your custom one

---

🟩 2. Build Docker Image

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

- `-t` → Adds tag
- `.` → Build context (location of Dockerfile)

🟧 3. Run Container

```
docker run -d -p 8080:80 mynginx:v1
```

- Maps **host 8080 → container 80**
- Runs Nginx in detached mode

🔵 4. Intermediate Dockerfile Concepts

✓ Set Working Directory

```
WORKDIR /app
```

✓ Copy Files

```
COPY . /app
```

✓ Install Packages

```
RUN apt-get update && apt-get install -y curl
```

✓ Set Default Command

```
CMD ["node", "app.js"]
```

✓ Expose Ports

```
EXPOSE 3000
```

✓ Environment Variables

```
ENV APP_ENV=production
```

🟥 5. Advanced Dockerfile Concepts

1️⃣ Multi-Stage Build (Reduce Image Size)

```
# Build Stage
FROM node:18 AS builder
WORKDIR /app
COPY package*.json .
RUN npm install
COPY . .
RUN npm run build

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

Benefits:

- Smaller image
- Faster deployments
- Clean, production-ready images

2️⃣ Docker Ignore File

Create `.dockerignore`:

```
node_modules
.git
.env
logs/
```

Reduces build context → faster builds → smaller images.

3️⃣ Layer Optimization

Combine RUN commands:

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

4️⃣ Healthchecks

```
HEALTHCHECK --interval=30s --timeout=5s \
  CMD curl -f http://localhost/ || exit 1
```

5️⃣ EntryPoint vs CMD

ENTRYPOINT → Fixed

CMD → Arguments to entrypoint

Example:

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

🚢 Pushing Docker Image to Docker Hub

Once your image is built, push it into Docker Hub for deployment or sharing.

🟦 1. Login to Docker Hub

```
docker login
```

Enter:

- Docker Hub **username**
- **Access token** (recommended over password)

🟩 2. Tag the Image

```
docker tag myapp:latest prakash14306/myapp:1.0
```

Tag Structure:

```
<username>/<repository>:<version>
```

Examples:

```
prakash14306/myapp:v1
prakash14306/webapp:latest
```

🟧 3. Push Image to Docker Hub

```
docker push prakash14306/myapp:1.0
```

🟥 4. Verify the Upload

Go to:
Docker Hub → Repositories → your-username/myapp

Image should appear with tag **1.0**.

🟦 5. Advanced Image Publishing Tips

✓ Use Semantic Versioning

```
v1, v1.0, v1.0.1
latest
```

✓ Always push both version + latest

```
docker push prakash14306/myapp:1.0
docker push prakash14306/myapp:latest
```

✓ Automate Push Using CI/CD

- Jenkins
- GitHub Actions
- GitLab CI
- Azure DevOps