



What is Docker?

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

A **container** is a lightweight, isolated environment that includes everything your app needs —

- ✓ Code
- ✓ Libraries
- ✓ Dependencies
- ✓ Runtime

So the app runs **exactly the same** on any system — developer laptop, testing server, or production cloud.

Docker Components :

Docker Engine.

- Docker Engine is the **core service** that makes Docker work.
- It's a **client-server application** that builds, runs, and manages containers on your system.

Think of Docker Engine as the "**heart of Docker**" ❤️, it runs in the background and handles everything related to containers.

Docker Image.

- A Docker Image is like a **blueprint or template** for creating containers.

Think of it as a **snapshot** that contains:

- Your application code
- Runtime (like Python, Node.js, etc.)
- Dependencies and libraries
- Configuration and environment variables

Once built, an image is **read-only** — it doesn't change.

Example:

Let's say you have a Python app.

You create a **Dockerfile** like this:

Dockerfile

 Copy code

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

Then you build the image:

bash

 Copy code

```
docker build -t myapp:1.0 .
```

This creates an image named `myapp:1.0`.



Docker Container.

- A container is a running instance of a Docker image.
- You can create multiple containers from the same image.
- Containers are isolated environments each has its own file system, processes, and network — but they all share the host OS kernel, making them **lightweight**.

Example:

To run the image as a container:

bash

 Copy code

```
docker run -d -p 5000:5000 myapp:1.0
```

Now your app runs in an isolated container.

You can start, stop, or delete it anytime.

Dockerfile.

- A Dockerfile is a simple text file that contains a list of instructions on how to build a Docker image.
- You can think of it as a recipe,
Docker reads the Dockerfile line by line and executes each instruction to build an image.

Purpose of a Dockerfile.

- Instead of manually setting up environments, you can automate it.
- For example, if you usually install Python, copy your app, and run it, the Dockerfile can do all that automatically.

Common Docker file Example :

- Let's say you have a Python web app.
- Your Dockerfile might look like this:

```
Dockerfile Copy code

# 1. Start from an existing base image
FROM python:3.10

# 2. Set the working directory inside the container
WORKDIR /app

# 3. Copy your app files into the container
COPY . /app

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

# 5. Command to run your app
CMD ["python", "app.py"]
```

What Each Instruction Means :

| COMMANDS | MEANIING |
|----------|--|
| FROM | Defines the base image (e.g., python:3.10) |
| WORKDIR | Sets the working directory in the container |
| COPY | Copies files from your computer to the container |
| RUN | Executes commands (like installing dependencies) |
| CMD | Specifies the default command to run when the container starts |

💡 How to Build an Image from a Dockerfile

Once your Dockerfile is ready, run:

```
bash
```

 Copy code

```
docker build -t myapp:1.0 .
```

This tells Docker:

- Build an image
- Tag it as `myapp:1.0`
- Use the current directory (`.`) for context

Then check it with:

```
bash
```

 Copy code

```
docker images
```

Docker Hub.

- Docker Hub is a cloud-based repository (or registry) where you can store, share, and download Docker images.
- Think of it like GitHub, but instead of storing code, it stores Docker images.
- Purpose of Docker Hub
- It's the official image registry for Docker.
- Developers and organizations use it to publish their container images publicly or privately.
- Docker automatically pulls images from Docker Hub when you use commands like:

```
bash
```

 Copy code

```
docker run nginx
```

This will download the `nginx` image from Docker Hub (if you don't already have it locally).

In short:

- **Image** = Blueprint
- **Container** = Live Instance of that Blueprint

Analogy

- **Dockerfile** → recipe
- **Image** → ready-made food package
- **Container** → the food being served
- **Docker Hub** → online food delivery store where everyone uploads or downloads ready-made meals.

In short:

Docker Hub is the official online library for Docker images — you can pull public images, push your own, and share with the world.

Basic Docker Commands :

| COMMANDS | DESCRIPTION |
|------------------|---|
| docker --version | Check installed Docker version. |
| docker info | Show system-wide information about Docker. |
| docker help | Show help for Docker commands. |
| docker login | Authenticates you with your Docker Hub account. |

◆ Working with Images

| COMMANDS | DESCRIPTION |
|---|--|
| docker images | List all Docker images. |
| docker pull <image> | Download image from Docker Hub. |
| docker push <image> | Uploads your image to your Docker Hub account. |
| docker build -t <image_name> . | Build image from Dockerfile. |
| docker rmi <image_id> | Remove image. |
| docker tag <image_id> <repo_name>:<tag> | Tag an image with a new name or version. |

◆ Working with Containers

| COMMANDS | DESCRIPTION |
|---|---|
| docker ps | List running containers. |
| docker ps -a | List all containers (including stopped ones). |
| docker run <image> | Create and start a new container. |
| docker run -d <image> | Run container in detached mode (background). |
| docker run -it <image> bash | Run container interactively with a terminal. |
| 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 logs <container_id> | View container logs. |
| docker inspect <container_id> | Get detailed information about a container. |
| docker run -p <host_port>:<container_port><image> | Run container with port mapping. |
| docker exec -it <container_id> bash | Access a running container shell. |
| docker cp <container_id>:<source><destination> | Copy files from container to host. |
| docker cp <source>:<container_id>><destination> | Copy files from host to container. |
| docker stats | Display container resource usage. |

◆ Docker Volumes and Networks

| COMMANDS | DESCRIPTION |
|---------------------------------------|----------------------------|
| docker volume ls | List of volumes. |
| docker volume rm <volume_name> | Remove a volume. |
| docker network ls | List of networks. |
| docker network create <network_name> | Create a custom network. |
| docker network inspect <network_name> | Show details of a network. |

◆ Cleanup Commands

| COMMANDS | DESCRIPTION |
|------------------------|---|
| docker system prune | Remove unused data from (containers, images, networks). |
| docker image prune | Remove unused images. |
| docker container prune | Remove stopped containers. |
| docker volume prune | Remove unused volumes. |

◆ Docker Compose (Multi-container Management)

| COMMANDS | DESCRIPTION |
|---------------------|---|
| docker-compose up | Start all services in docker-compose.yml. |
| docker-compose down | Stop and remove all services. |
| docker-compose ps | List services defined in Compose file. |
| docker-compose logs | View logs of running services. |

Multi-stage build in Docker.

What is a Multi-Stage Docker Build?

- A **multi-stage build** in Docker is a way to **optimize your Docker images** by using **multiple FROM statements** in a single Dockerfile.
- Each stage can have its own **base image**, and you can **copy only the necessary artifacts** from one stage to another — keeping the final image **small, secure, and efficient**.

Why It's Needed ?

Normally, if you build an image with all the build tools (compilers, package managers, etc.), those tools stay inside the image — making it large.

Multi-stage builds solve this by separating the **build environment** from the **runtime environment**.

Example:

```
dockerfile
```

```
# Stage 1: Build stage
FROM golang:1.20 AS builder
WORKDIR /app
COPY . .
RUN go build -o myapp

# Stage 2: Final runtime stage
FROM alpine:latest
WORKDIR /app
COPY --from=builder /app/myapp .
CMD ["./myapp"]
```

 Copy code

What Happens Here

- The first stage (builder) compiles the Go app using the **golang** image.
- The second stage uses a **lightweight Alpine image** and copies only the compiled binary (myapp) from the first stage.
- The final image contains **only what's needed to run the app**, not the entire build environment.

Key Benefits

-  Smaller Images : Only required files are included, making deployment faster.
-  More Secure : No unnecessary build tools left behind.
-  Cleaner Dockerfile : All build steps in one file, no need for external scripts.
-  Faster CI/CD : Less data transfer and quicker container startup.

Distroless Container Images

The Minimalist Approach to Docker Security & Performance

While exploring Docker, I came across an important concept that every DevOps learner should know — Distroless Images.

What are Distroless Images?

Distroless container images contain only the application and its runtime dependencies — no shell, package manager, or OS tools.

They are built on the principle of “less is more” — smaller surface area, fewer vulnerabilities.

Why use them?

-  Lightweight: Reduced image size speeds up build and deployment times.
-  More Secure: No unnecessary packages → smaller attack surface.
-  Faster Startup: Containers load only what's required to run the app.
-  Consistent Environment: Ideal for production where stability matters most.

Example:

Instead of using a regular base image like

`FROM python:3.10`

you can use a distroless base image such as

`FROM gcr.io/distroless/python3`

Docker Networking

How Containers Communicate ?

- Networking is one of the most important parts of Docker — it's what allows containers to talk to each other, to the host, and to the outside world.
- Docker networks define how containers communicate — with each other, the host, or external systems.

What is Docker Networking and It's types?

- Docker networking enables isolated containers to communicate securely.
- Each container gets its own IP address, but Docker manages all the heavy lifting of routing and connectivity behind the scenes.

Bridge Network (Default Network)

What it is:

- The **default network** created by Docker when you install it.
- Containers connected to this network can **communicate with each other using container names** as hostnames.
- Externally, they communicate through **port mapping**.

Example:

```
docker run -d --name webapp --network bridge nginx
```

Why we use it:

- Ideal for **local development and testing**.
- Provides **basic isolation** between containers and the host.
- Common for **single-host setups** where containers need to interact.

Host Network

What it is:

- The container **shares the same network namespace** as the host machine.
- It doesn't have its own private IP; instead, it uses the host's IP directly.

Example:

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

Why we use it:

- Reduces **network overhead** and improves **performance**.
- Ideal for **network-intensive applications** like monitoring tools (Prometheus, Grafana) or **web servers** that need host-level access.
- Best for **Linux systems**, as it removes an extra layer of network abstraction.

None Network

What it is:

- Completely **disables networking** for the container.
- The container has **no access** to external networks or other containers.

Example:

```
docker run -d --network none busybox
```

Why we use it:

- Perfect for **highly secure or isolated workloads**.

- Useful when you want to control all communication manually (e.g., testing or security sandboxing).

Overlay Network

What it is:

- Enables communication between containers running on **different Docker hosts**.
- Commonly used in **Docker Swarm** or distributed environments.
- It uses an **underlying network overlay** (VXLAN) to connect containers across hosts.

Example:

```
docker network create -d overlay my-overlay
```

Why we use it:

- Crucial for **multi-host or clustered deployments**.
- Supports **microservices architectures** that span across multiple servers.
- Simplifies **service discovery** and load balancing in Docker Swarm mode.

Macvlan Network

What it is:

- Assigns a **unique MAC address** to each container.
- Each container appears as a **separate physical device** on the same network as the host.

Example:

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

Why we use it:

- Needed when containers must **interact directly with external devices** on the same LAN.
- Useful in **legacy applications** or **enterprise networks** where systems require **static IPs** or **direct access**.
- Gives **fine-grained network control** for advanced setups.

💡 Summary Table

| Network Type | DESCRIPTION | Common Use Case |
|----------------|---|--|
| Bridge | Default network for containers on the same host | Local development and isolated communication |
| Host | Shares host network stack | High-performance apps and monitoring tools |
| None | Disables networking | Secure, isolated workloads |
| Overlay | Connects containers across multiple hosts | Swarm and microservices deployments |
| Macvlan | Gives containers their own MAC address | Direct LAN access, legacy system integration |

- @CodeBit360