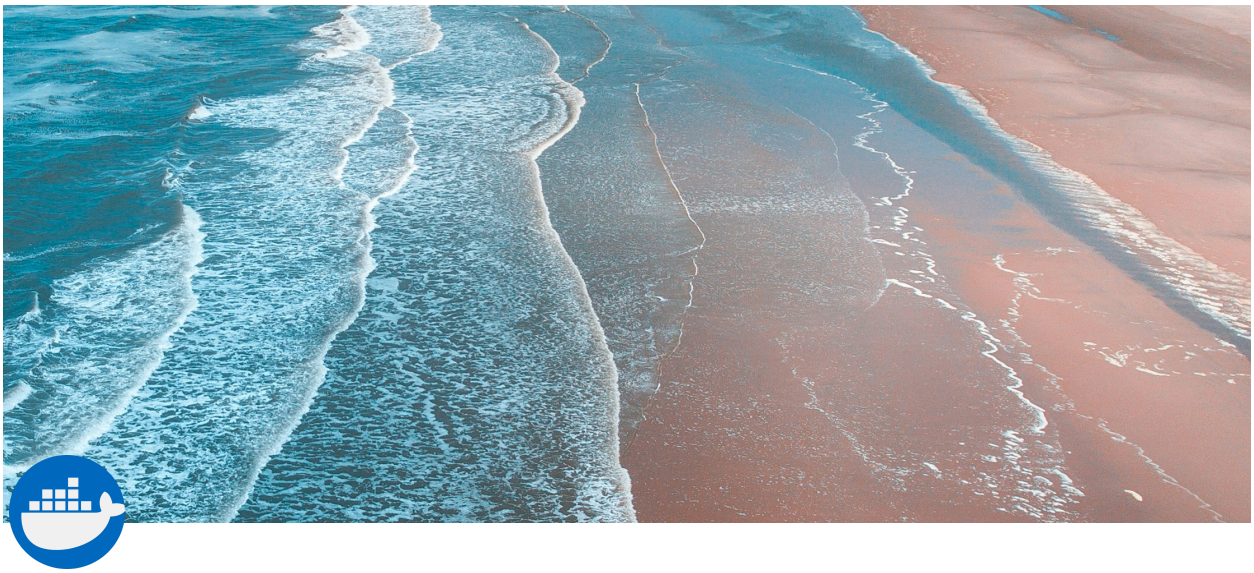

Docker & Containers

Introduction to Containers

Containers are a form of lightweight, isolated environments that allow developers to package applications along with their dependencies. They run on the host operating system using **kernel-level isolation**, such as Linux namespaces and cgroups.

Unlike virtual machines (VMs), containers do not emulate an entire operating system (OS). Instead, they share the host kernel, making them more efficient in terms of performance and resource usage.

What is Docker?

Docker is an open-source platform used to develop, ship, and run applications inside containers. It simplifies the process of creating, deploying, and managing containerized applications by providing a standardized approach to handling application environments across various systems.

Why Use Docker?

Benefits of Docker:

- **Consistency**: Applications behave the same regardless of the environment.
- **Isolation**: Each container runs independently without affecting others.
- **Portability**: Easily move containers between development, testing, and production.
- **Efficiency**: Uses fewer resources than traditional VMs.
- **Scalability**: Quickly scale applications horizontally using orchestration tools.
- **Microservices-friendly**: Ideal for microservices architecture due to modularity.

Docker Architecture

Docker uses a **client-server architecture**:

- **Client**: The command-line interface (`docker` CLI) that users interact with.
- **Docker Daemon (dockerd)**: Background process that manages Docker objects like images, containers, networks, and volumes.
- **REST API**: Interface that allows communication between client and daemon.

Docker Components

Component	Description
Images	Read-only templates used to create containers. Built from a Dockerfile.
Containers	Runnable instances of images. Isolated processes with their own filesystem.
Dockerfile	Text file containing instructions to build an image.
Volumes	Used to persist data outside the lifecycle of a container.
Networks	Enables communication between containers or between containers and the outside world.
Registries	Repositories where images are stored and shared (e.g., Docker Hub).

Getting Started with Docker

Installation

On Linux (Ubuntu):

```
sudo apt update
sudo apt install docker.io
```

To run Docker without `sudo`, add your user to the `docker` group:

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

Log out and back in.

Verify installation:

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

🐋 Docker Command Reference Table

General Commands (Help Commands)

Purpose	Command
Check Docker version	`docker --version`
View system-wide information	`docker info`
Get help (general or specific command)	`docker --help` or `docker <command> --help`

Images: Build, Tag, Push, Pull

Purpose	Command
List all images	`docker images`

Build an image from Dockerfile	`docker build -t <image-name>:<tag> .`
Tag an image	`docker tag <source-image> <target-image>`
Push image to registry (e.g., Docker Hub)	`docker push <image-name>`
Pull image from registry	`docker pull <image-name>`
Remove one or more images	`docker rmi <image-id> [<image-id>...]`
Remove all unused images	`docker image prune -a`

Containers: Run, Manage, Inspect

Purpose	Command
Run a new container	`docker run [OPTIONS] <image>`
Run in detached mode	`docker run -d <image>`
Map host port to container port	`docker run -p <host-port>:<container-port> <image>`
Mount a volume	`docker run -v <host-path>:<container-path> <image>`
Assign environment variables	`docker run -e VAR=value <image>`
Name a container	`docker run --name <container-name> <image>`
List running containers	`docker ps`
List all containers (running + stopped)	`docker ps -a`
Stop a running container	`docker stop <container-id>`
Start a stopped container	`docker start <container-id>`
Restart a container	`docker restart <container-id>`
Remove a container	`docker rm <container-id>`
Remove all stopped containers	`docker container prune`
Execute a command inside a running container	`docker exec -it <container-id> <command>`
View logs of a container	`docker logs <container-id>`
Show processes running inside a container	`docker top <container-id>`
Get low-level info about a container/image	`docker inspect <container-or-image>`

Networks: Create, Manage, Inspect

Purpose	Command
List networks	`docker network ls`
Create a custom network	`docker network create <network-name>`
Inspect network details	`docker network inspect <network-name>`
Connect a running container to a network	`docker network connect <network> <container>`
Disconnect a container from a network	`docker network disconnect <network> <container>`
Remove a network	`docker network rm <network-name>`
Remove all unused networks	`docker network prune`

Volumes: Persistent Data Storage

Purpose	Command
List volumes	`docker volume ls`
Create a volume	`docker volume create <volume-name>`
Inspect volume	`docker volume inspect <volume-name>`
Remove a volume	`docker volume rm <volume-name>`
Remove all unused volumes	`docker volume prune`

Docker Compose

Purpose	Command
Start services defined in `docker-compose.yml`	`docker-compose up`
Start in detached mode	`docker-compose up -d`
Stop and remove containers	`docker-compose down`
Rebuild images	`docker-compose build`
View running services status	`docker-compose ps`
View logs for all containers	`docker-compose logs`

Execute command in a running service container	`docker-compose exec <service-name> <command>`

Swarm Mode (Basic Orchestration)

Purpose	Command
Initialize a swarm	`docker swarm init`
Join a worker node to swarm	`docker swarm join --token <token> <manager-ip>:2377`
Deploy a stack from compose file	`docker stack deploy -c docker-compose.yml <stack-name>`
List stacks and services	`docker stack ls`, `docker stack services <stack-name>`
Remove a stack	`docker stack rm <stack-name>`

Cleanup Commands

Purpose	Command
Remove all stopped containers	`docker container prune`
Remove all unused networks	`docker network prune`
Remove all dangling images	`docker image prune`
Remove all unused images and volumes	`docker system prune -a --volumes`

Tips

- Use `f` or `-force` to force removal without confirmation.
- Use `-rm` with `docker run` to automatically remove the container after exit.
- Use `docker history <image>` to see how an image was built.
- Use `docker save` and `docker load` for offline image transfer.
- We need to remove all containers before removing the images
- If a container is dependent on an image, and you try to delete the image, it will fail

Docker Command: `docker run` with a Tagged Image

You typically use `docker run` to start a container from a Docker image, and you reference the image using its **name and tag** (e.g., `nginx:latest`, `python:3.9`, etc.).

Syntax

```
docker run [OPTIONS] IMAGE[:TAG|@DIGEST]
```

Example

```
docker run -d --name my-nginx nginx:latest
```

- `nginx`: Image name
- `latest`: Tag (optional; defaults to `latest` if not specified)

> 💡 If no tag is provided, Docker assumes :latest.

Tagging an Image Before Running It

If you want to **tag** an image first (for example, after building it), use `docker tag` before running it.

Build and Tag an Image

```
docker build -t my-app:v1
```

Run the Tagged Image

```
docker run -d --name app-instance my-app:v1
```

Docker Command: `docker run` with -i and -it

What does `i` mean?

The `-i` flag stands for **interactive**. When you pass this flag to `docker run`, Docker keeps the container's standard input (stdin) open, allowing you to send input to the container manually or programmatically.

Syntax

```
docker run -i <image-name> <command>
```

When to use `i`

Use `-i` when:

- You want to interact with a process inside the container that reads from standard input (e.g., Python REPL, bash, MySQL shell).
- You are piping data into the container.

Common usage patterns

1. Input from a pipe

```
echo "hello" | docker run -i ubuntu cat
```

- `i` is needed here so the container can read the input from `echo`.
- The container runs `cat`, which reads stdin and outputs it.

Without `-i`, the input would be closed, and `cat` would have nothing to read.

2. Interacting with an app that waits for input

For example:

```
docker run -i python
```

This will run the Python interpreter and keep stdin open, but you won't see a shell unless you also include `-t`.

What happens without `i`

If you omit `-i`, stdin is closed by default. This means any application expecting user input (like `bash`, `python`, or `mysql`) will exit immediately or not behave as expected.

Adding `it` : Interactive + TTY

The `-t` flag tells Docker to allocate a **pseudo-terminal (TTY)**, which allows you to use terminal features like input history, arrow keys, and screen clearing.

Syntax with `it`

```
docker run -it <image-name> <command>
```

This is the most common way to run a container interactively.

Example

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

This starts an Ubuntu container and gives you a fully interactive shell.

Summary

Option	Description
`-i`	Keeps standard input open
`-t`	Allocates a pseudo-terminal
`-it`	Best for interactive terminal sessions

Docker Run - Port mapping

In Docker, **port mapping** allows you to expose a container's internal port to the host machine so that you can access applications running inside the container from outside.

This is done using the `-p` flag with the `docker run` command.

Basic Syntax

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

- `host_port`: The port on your **host machine** (your computer or server).
- `container_port`: The port your **application inside the container** listens on.

Example

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

- This tells Docker to run the `nginx` container.
- Nginx listens on port `80` **inside** the container.
- The `p 8080:80` flag maps it to port `8080` on your **host**, so you can visit `http://localhost:8080` in your browser and reach the Nginx web server.

Multiple Port Mappings

You can specify more than one `-p` flag to expose multiple ports:

```
docker run -p 8080:80 -p 4430:443 nginx
```

Dynamic Host Ports

If you use `-p :<container_port>`, Docker will randomly assign an available host port:

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

Use `docker ps` afterward to see which host port was assigned.

Binding to Specific IP Addresses

You can bind a container port to a specific IP on the host:

```
docker run -p 127.0.0.1:8080:80 nginx
```

Only `localhost` (127.0.0.1) can access this port; it's not exposed publicly.

Summary

Syntax	Meaning
`-p 8080:80`	Map host port 8080 to container port 80
`-p 127.0.0.1:8080:80`	Map to localhost only
`-p :80`	Random host port mapped to container port 80
Multiple `-p` flags	Expose multiple ports

Docker Run - Volume Mapping

The `docker run -v` option (or `--volume`) is used for **volume mapping**, which allows you to **persist data** or **share files** between your host and the Docker container.

Basic Syntax

```
docker run -v <host_path>:<container_path> <image>
```

- `host_path`: The path on your **host machine**.

- `container_path`: The path **inside the container** where the volume is mounted.

Example: Mount a local folder

```
docker run -v /home/user/data:/app/data ubuntu
```

- Files in `/home/user/data` on the host will appear in `/app/data` inside the container.
- Changes made inside the container to `/app/data` will also reflect on the host.

Use Cases

1. **Persisting database data** (e.g., for MySQL or PostgreSQL).
2. **Sharing source code** for live development.
3. **Storing logs or configuration files**.

Anonymous Volumes

If you only specify a container path:

```
docker run -v /data ubuntu
```

Docker creates an **anonymous volume** in a default location on the host and mounts it to `/data` in the container. This is useful when you want data persistence but don't care where it's stored.

Named Volumes

You can use **Docker-managed named volumes**:

```
docker volume create mydata
docker run -v mydata:/data ubuntu
```

- This uses Docker's volume system (checkable with `docker volume ls`).

- Better for portability and Docker Compose.

Read-Only Volumes

You can mount volumes as **read-only**:

```
docker run -v /home/user/config:/app/config:ro ubuntu
```

- The container can read from `/app/config`, but cannot modify it.

Summary

Syntax	Description
`-v host_path:container_path`	Bind mount a local path into the container
`-v name:container_path`	Use a named volume
`-v path:container_path:ro`	Mount as read-only
`-v /path/in/container`	Create an anonymous volume

Dockerfile Basics

A **Dockerfile** defines how to build a Docker image.

Example: Simple Python App Dockerfile

```
# Use official Python runtime as parent image
FROM python:3.9-slim

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install any needed packages specified in requirements.txt
```

```
RUN pip install --no-cache-dir -r requirements.txt

# Make port 80 available to the outside world
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

Building and Managing Images

Build an Image

```
docker build -t my-python-app .
```

List Images

```
docker images
```

Tag an Image

```
docker tag my-python-app username/my-python-app:latest
```

Remove an Image

```
docker rmi my-python-app
```

Running Containers

Start a Container

```
docker run -d -p 4000:80 --name web my-python-app
```

Flags:

- `d`: Detached mode (run in background)
- `p`: Map port 4000 (host) to 80 (container)
- `-name`: Assign a name to the container

View Running Containers

```
docker ps
```

Stop a Container

```
docker stop web
```

Remove a Container

```
docker rm web
```

Networking in Docker

Default Networks

Docker provides default networks like `bridge`, `host`, and `none`.

Create a Custom Network

```
docker network create my-network
```

Attach Container to a Network

```
docker run -d --network my-network --name web my-python-app
```

Inspect Network

```
docker network inspect my-network
```

Volumes and Persistent Data

Types of Volumes

- **Host Volume**: Mount a directory from the host machine.
- **Named Volume**: Managed by Docker.
- **tmpfs**: Temporary filesystem stored in memory.

Mount a Host Directory

```
docker run -d \\
  -v /path/on/host:/path/in/container \\
  --name my-container my-image
```

Named Volume Example

```
docker volume create my-volume
docker run -d -v my-volume:/data --name db mysql
```

Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications using a YAML file.

Example: `docker-compose.yml`

```
version: '3'
services:
  web:
    build: .
    ports:
      - "4000:80"
  redis:
    image: "redis:alpine"
```

Commands

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

Docker Hub and Registries

Push an Image to Docker Hub

```
docker login
docker push username/my-python-app
```

Pull an Image

```
docker pull nginx
```

Other registries include:

- Google Container Registry (GCR)
- Amazon Elastic Container Registry (ECR)
- Azure Container Registry (ACR)

Best Practices

- Use `.dockerignore` to exclude unnecessary files.
- Minimize layers in Dockerfiles.
- Use multi-stage builds for smaller images.
- Don’t run containers as root unless necessary.
- Keep images small by clearing your cache.
- Use explicit tags instead of `latest`.
- Regularly scan images for vulnerabilities.

Security Considerations

- **Image Vulnerabilities**: Scan images before deployment.
- **User Context**: Run containers as non-root.
- **Capabilities**: Drop unnecessary capabilities.
- **Secret Management**: Use Docker secrets or external vaults.
- **Content Trust**: Enable Docker Content Trust for signed images.
- **Network Policies**: Limit inter-container communication.

Docker vs Virtual Machines

Feature	Docker	Virtual Machine
Boot Time	Seconds	Minutes
Resource Usage	Low	High
Isolation	Process-level	Kernel-level
Guest OS	Same as host	Full OS
Performance	Near-native	Slightly slower
Portability	High	Medium

Orchestration with Docker Swarm and Kubernetes

Docker Swarm

Docker's native orchestration tool.

```
docker swarm init
docker service create --replicas 3 -p 80:80 my-web-app
```

Kubernetes

A more powerful and widely adopted orchestration system.

Kubernetes works with Docker images but abstracts away many Docker-specific details.

Tools like **Helm** and **kubectl** are commonly used.

Use Cases

- **Development Environments**: Consistent dev/test/prod setups.
- **Microservices Architecture**: Deploy each service independently.
- **CI/CD Pipelines**: Easy integration with automation tools.
- **Legacy App Modernization**: Containerize old apps for easier maintenance.
- **Hybrid Cloud Deployments**: Run the same images across cloud providers.

Docker Installation on Ubuntu

```
# 1. Update existing packages
sudo apt update

# 2. Install required packages
sudo apt install apt-transport-https ca-certificates curl software-properties-com
```

```
# 3. Add Docker's official GPG key
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor

# 4. Add Docker repository
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/do
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# 5. Update again
sudo apt update

# 6. Install Docker
sudo apt install docker-ce docker-ce-cli containerd.io -y

# 7. (Optional) Run Docker as non-root user
sudo usermod -aG docker $USER
# Then logout and log back in

# 8. Test Docker
docker --version
docker run hello-world
```

What is a Docker Image?

A **Docker image** is a **lightweight, standalone, executable package** that includes everything needed to run a piece of software: code, runtime, libraries, environment variables, and configuration files.

Think of it like a **template or blueprint** for creating Docker containers. When you run an image, it becomes a container.

Key Features of Docker Images

- **Immutable**: Once created, images cannot be changed (unless rebuilt).

- **Layered architecture**: Built using a union file system, each change creates a new layer.
- **Portable**: Can run on any system where Docker is installed.
- **Versioned**: Supports tagging (e.g., `nginx:1.21`, `nginx:latest`).

How Are Docker Images Created?

Images are typically built from a **Dockerfile**, which is a text document containing a series of instructions that tell Docker how to build the image.

Example Dockerfile:

```
# Use a base image
FROM python:3.9-slim

# Set the working directory
WORKDIR /app

# Copy local files into the image
COPY . .

# Install dependencies
RUN pip install -r requirements.txt

# Define the command to run when the container starts
CMD ["python", "app.py"]
```

To build the image:

```
docker build -t my-python-app .
```

This creates an image named `my-python-app`.

Layered File System Explained

Each instruction in the Dockerfile creates a **layer**:

- Layers are cached for faster builds.
- If one line changes, only that layer and subsequent ones rebuild.

Example layers:

1. Base image (`python:3.9-slim`)
2. Working directory created
3. Files copied
4. Dependencies installed
5. CMD set

Running a Container from an Image

Once the image is built, you can run it as a container:

```
docker run -d -p 8000:8000 my-python-app
```

This starts a container from your image, mapping port 8000.

Managing Docker Images

Common Docker CLI commands for managing images:

Command	Description
`docker images`	List all images on your machine
`docker build`	Build an image from a Dockerfile
`docker pull [image]`	Download an image from a registry
`docker push [image]`	Upload an image to a registry
`docker rmi [image]`	Remove one or more images
`docker inspect [image]`	Show detailed info about an image

Docker Hub & Registries

Docker images are often stored and shared via **registries**, such as:

- **Docker Hub** (default public registry)
- Private registries (like AWS ECR, Azure ACR, Google GCR)

You can pull existing images:

```
docker pull nginx
```

Or push your own:

```
docker tag my-python-app username/my-python-app:1.0
docker push username/my-python-app:1.0
```

Best Practices for Docker Images

1. **Use official base images** when possible.
2. **Minimize layers** by combining `RUN` commands.
3. **Avoid including unnecessary files** (use `.dockerignore`).
4. **Use multi-stage builds** to reduce final image size.
5. **Tag versions clearly** (`v1.2.0`, not just `latest`).
6. **Scan for vulnerabilities** using tools like Trivy or Clair.

Image vs Container

Feature	Docker Image	Docker Container
Type	Template/blueprint	Running instance
Mutable?	No	Yes
Storage	Read-only	Read-write (writable layer)
Usage	To create containers	To run applications

Summary

- Docker images are the foundation of containers.
- They are built from Dockerfiles and used to create reproducible environments.
- They allow developers to package apps with all their dependencies and run them consistently anywhere.
- Efficient layering and caching make building and sharing fast and scalable.

Real-World Dockerfile Examples

1. Python Web App (Flask)

```
# Use a small official Python image
FROM python:3.10-slim

# Set working directory
WORKDIR /app

# Copy only requirements first to leverage Docker layer caching
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Now copy the rest of the app
COPY . .

# Expose port
EXPOSE 5000

# Command to run the app
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "app:app"]
```

Notes:

- `-no-cache-dir` avoids caching pip packages, reducing image size.
- Splitting `COPY` helps reuse layers when only code changes.

2. Node.js Application

```
# Base image with Node.js
FROM node:18-alpine

# Working directory
WORKDIR /usr/src/app

# Copy package files first
COPY package*.json ./

# Install dependencies
RUN npm ci --only=production

# Copy source code
COPY . .

# Expose port
EXPOSE 3000

# Start command
CMD ["node", "server.js"]
```

Notes:

- `alpine` is a minimal Linux image — great for size.
- `npm ci` ensures repeatable builds using `package-lock.json`.

3. Multi-Stage Build (Go App)

```
# Stage 1: Build the binary
FROM golang:1.21 as builder
WORKDIR /app
```

```
COPY . .
RUN CGO_ENABLED=0 go build -o myapp .

# Stage 2: Final image
FROM gcr.io/distroless/static-debian12
COPY --from=builder /app/myapp /myapp
CMD ["/myapp"]
```

Notes:

- The first stage compiles the Go app.
- The second stage uses a **distroless image**, which contains only your app and no shell or OS tools, ideal for security and size.

Sure! Here's a clear explanation of **Docker and environment variables**, tailored for someone starting in DevOps or containerization:

🐋 Docker and Environment Variables

What Are Environment Variables?

Environment variables are key-value pairs used to configure applications without requiring changes to the code. They're often used to pass sensitive information (like API keys, database URLs, or passwords) or to control app behavior (like enabling debug mode or setting a port number).

Why Use Environment Variables in Docker?

Docker containers are isolated environments. Instead of hardcoding values into your application or Docker files, you can **inject configuration at runtime** using environment variables. This makes your containers:

- **Reusable:** Same image, different settings
- **Secure:** Secrets aren't stored in the code
- **Scalable:** Easily configurable across environments (dev, staging, production)

How to Set Environment Variables in Docker

1. Using the `e` flag with `docker run`

```
docker run -e APP_ENV=production -e DEBUG=false myapp
```

2. Using an `.env` file

Create a file named `.env`:

```
APP_ENV=production
DEBUG=false
```

Then use it with `--env-file`:

```
docker run --env-file .env myapp
```

3. In the Dockerfile

```
ENV APP_ENV=production
ENV DEBUG=false
```

> This sets default values but can be overridden at runtime.

4. In `docker-compose.yml`

```
version: '3.8'
services:
  web:
    image: myapp
    environment:
      - APP_ENV=production
      - DEBUG=false
```

Or load from a file:

```
env_file:
  - .env
```

Best Practices

- **Never store secrets in your Dockerfile** or commit them to version control.
- **Use** `.env` **files locally** and secret managers (like AWS Secrets Manager or Vault) in production.
- **Document required env vars** in your project README or use a template `.env.example`.

Example: Node.js App Using Env Vars

In your Node.js app:

```
console.log(process.env.APP_ENV); // outputs 'production'
```

Run it in Docker:

```
docker run -e APP_ENV=production node-app
```

Summary

- Environment variables let you **configure containers dynamically**.
- They're essential for **12-factor app** principles (especially "config should be in the environment").
- Use them wisely to keep your containers **clean, secure, and portable**.

CMD vs ENTRYPOINT in Docker

Both are used to **specify what runs inside a container,** but they behave differently.

CMD

- **Purpose**: Provides *default arguments or commands* for the container.
- **Can be overridden** by arguments passed to `docker run`.

Example:

```
FROM alpine
CMD ["echo", "Hello from CMD"]
```

Run:

```
docker run my-image
# Output: Hello from CMD

docker run my-image "Overridden CMD"
# Output: Overridden CMD
```

> CMD is like a suggestion.

ENTRYPOINT

- **Purpose**: Sets a *fixed executable* for the container.
- **Always runs;** arguments passed to' `docker run` ' become **arguments to the entrypoint**.

Example:

```
FROM alpine
ENTRYPOINT ["echo"]
```

Run:

```
docker run my-image "Hello ENTRYPOINT"
# Output: Hello ENTRYPOINT
```

> 🧠 ENTRYPOINT is like a mandatory wrapper.

CMD + ENTRYPOINT Combined

You can use them together:

- `ENTRYPOINT` defines **what runs**
- `CMD` provides **default arguments**

Example:

```
FROM alpine
ENTRYPOINT ["ping"]
CMD ["google.com"]
```

```
docker run my-image
# Output: PING google.com

docker run my-image 8.8.8.8
# Output: PING 8.8.8.8
```

CMD vs ENTRYPOINT Cheat Sheet

Feature	CMD	ENTRYPOINT
Sets executable?	✅ (if no ENTRYPOINT used)	✅ Always
Default arguments?	✅	🚫 (unless used with CMD)
Overridden by args?	✅ Yes	🚫 Not unless overridden manually
Typical use case	Default behavior (e.g., `npm start`)	Always run a script or command
Best form	`["executable", "arg1"]` (exec)	`["executable", "arg1"]` (exec)

Tip: Use `ENTRYPOINT` for strict control, and `CMD` for flexibility.

Let me know if you want a live Dockerfile example to experiment with both.

What is Docker Compose?

Docker Compose is a tool for defining and running multi-container Docker applications using a single file: `docker-compose.yml`.

Instead of manually running several `docker run` commands, Compose lets you manage everything from **one config file**.

Why Use Docker Compose?

- **Manage multiple services** (e.g., app + database + cache)
- **Version-controlled configuration**
- **Easy to share and replicate environments**
- **Built-in networking** between containers
- **Reusable and scalable setups**

Basic `docker-compose.yml` Example

Let's say you want to run a simple app with MySQL.

```
version: '3.8'

services:
  db:
    image: mysql:8
    container_name: mysql-db
    environment:
      MYSQL_ROOT_PASSWORD: db_pass123
    ports:
      - "3306:3306"

  web:
    image: nginx:latest
```

```
    ports:
      - "8080:80"
```

What This Does:

- Spins up a **MySQL container** with a root password
- Spins up an **NGINX container** serving on port 8080
- Both are connected to the same **Docker network** automatically

---

Common Docker Compose Commands

Command	Description
`docker-compose up`	Start all services
`docker-compose up -d`	Start in detached (background) mode
`docker-compose down`	Stop and remove all services
`docker-compose ps`	List running services
`docker-compose logs`	View service logs
`docker-compose exec <service> <command>`	Run a command inside a running container

> 💡 Newer Docker versions support docker compose (no hyphen), which is the same but uses the Docker CLI plugin.

Typical Project Structure

```
my-app/
│
├── docker-compose.yml
├── .env
├── app/
│   └── Dockerfile
└── db/
```

Using `.env` with Docker Compose

You can define environment variables in a `.env` file:

```
MYSQL_ROOT_PASSWORD=db_pass123
```

Then in `docker-compose.yml`:

```
environment:
  - MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD}
```

Summary

Feature	Benefit
Simple config	Define whole stacks in one file
Reproducible	Share config with your team
Scalable	Add new services or replicas easily
Built-in networking	Services talk to each other by name

Basic Structure of a Docker Compose File

```
version: "3.8"          # (1) Compose file version

services:               # (2) Define your containers
  app:                  # (3) Name of the service
    image: node:18      # (4) Use an image OR build from a Dockerfile
    build: ./app        # Optional - build from Dockerfile in /app
    ports:              # (5) Port mapping
      - "3000:3000"
    environment:        # (6) Environment variables
      - NODE_ENV=production
    volumes:            # (7) Mount local folders into container
      - ./app:/usr/src/app
```

```
    depends_on:       # (8) Order of startup
      - db

  db:
    image: mysql:8
    environment:
      - MYSQL_ROOT_PASSWORD=mysecret
    ports:
      - "3306:3306"
    volumes:
      - db_data:/var/lib/mysql

volumes:              # (9) Named volumes
  db_data:
```

Explanation of Components

Component	Description
`version`	Compose file version (3.8 is safe and commonly used)
`services`	Top-level key defining all your containers
`image`	Docker image to pull from Docker Hub or a registry
`build`	Directory containing a Dockerfile
`ports`	Map container ports to host ports (`host:container`)
`environment`	Set environment variables (e.g., DB passwords, app settings)
`volumes`	Mount local paths or named volumes into the container
`depends_on`	Specify the startup order (app depends on db, etc.)
`volumes:` (root)	Declare named volumes that persist data beyond container lifecycle

Minimal Docker Compose Example: Node.js + MySQL

```
version: "3.8"
```

```
services:
  app:
    build: ./app
    ports:
      - "3000:3000"
    environment:
      - DB_HOST=db
      - DB_USER=root
      - DB_PASS=mysecret
    depends_on:
      - db

  db:
    image: mysql:8
    environment:
      - MYSQL_ROOT_PASSWORD=mysecret
    volumes:
      - db_data:/var/lib/mysql
    ports:
      - "3306:3306"

volumes:
  db_data:
```

Tips for Writing Compose Files

- Always **indent with 2 spaces** (no tabs!)
- Quote port numbers and env values with special characters
- Use `.env` **files** to store secrets like passwords
- Use `build:` if you have a Dockerfile; otherwise, use `image:`

Typical Project Structure

```
my-project/
├── docker-compose.yml
├── .env
├── app/
│   ├── Dockerfile
│   └── index.js
```

Here are additional practical Docker Compose examples for various tech stacks and use cases, ranging from databases to full-stack applications.

Example 1: Python Flask + PostgreSQL

```
version: "3.8"

services:
  web:
    build: ./flask-app
    ports:
      - "5000:5000"
    environment:
      - DATABASE_URL=postgres://postgres:pass@db:5432/mydb
    depends_on:
      - db

  db:
    image: postgres:14
    environment:
      POSTGRES_DB: mydb
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: pass
    ports:
      - "5432:5432"
```

```
    volumes:
      - pgdata:/var/lib/postgresql/data

volumes:
  pgdata:
```

Your Flask app should read `DATABASE_URL` to connect to PostgreSQL.

Example 2: React Frontend + Express Backend + MongoDB

```
version: "3.8"

services:
  frontend:
    build: ./client
    ports:
      - "3000:3000"
    depends_on:
      - backend

  backend:
    build: ./server
    ports:
      - "5000:5000"
    environment:
      - MONGO_URL=mongodb://mongo:27017/mydb
    depends_on:
      - mongo

  mongo:
    image: mongo
    ports:
      - "27017:27017"
    volumes:
```

```
      - mongodb:/data/db

volumes:
  mongodb:
```

Example 3: WordPress + MySQL

```
version: "3.8"

services:
  wordpress:
    image: wordpress:latest
    ports:
      - "8080:80"
    environment:
      - WORDPRESS_DB_HOST=db
      - WORDPRESS_DB_USER=root
      - WORDPRESS_DB_PASSWORD=wp_pass
      - WORDPRESS_DB_NAME=wordpress
    depends_on:
      - db

  db:
    image: mysql:5.7
    environment:
      - MYSQL_DATABASE=wordpress
      - MYSQL_ROOT_PASSWORD=wp_pass
    volumes:
      - wp_db:/var/lib/mysql

volumes:
  wp_db:
```

Example 4: Django + PostgreSQL + Redis

```
version: "3.8"

services:
  web:
    build: ./backend
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - ./backend:/app
    ports:
      - "8000:8000"
    depends_on:
      - db
      - redis

  db:
    image: postgres
    environment:
      POSTGRES_DB: mydb
      POSTGRES_USER: user
      POSTGRES_PASSWORD: pass

  redis:
    image: redis:alpine

volumes:
  pgdata:
```

Example 5: Simple Redis Testing Setup

```
version: "3.8"

services:
```

```
  redis:
    image: redis:latest
    ports:
      - "6379:6379"
```

Use this to test Redis locally or connect from any client.

Example 6: Nginx as Reverse Proxy + Web App

```
version: "3.8"

services:
  web:
    build: ./web-app
    expose:
      - "3000"

  nginx:
    image: nginx:alpine
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
    ports:
      - "80:80"
    depends_on:
      - web
```

Your `nginx.conf` should proxy requests from port 80 to the `web` service at `http://web:3000`.

Docker Registry

A **Docker Registry** is a storage and distribution system for Docker images. It allows you to **store**, **share**, and **manage** container images that are used to deploy applications across environments.

What Is a Docker Image?

A Docker image is a lightweight, standalone, and executable package that includes everything needed to run a piece of software, code, runtime, libraries, environment variables, and config files.

What Is a Docker Registry?

A **Docker Registry** is where these images live. Think of it like GitHub, but for Docker images.

There are **two main types**:

1. Public Registries

- **Docker Hub** (default): The most popular registry maintained by Docker Inc.
- **GitHub Container Registry** (GHCR)
- **Google Container Registry (GCR)**
- **Amazon Elastic Container Registry (ECR)**

You can pull images directly:

```
docker pull nginx
```

(Which pulls from Docker Hub by default)

2. Private Registries

Organizations often set up **private Docker registries** for:

- Security (control access to sensitive images)
- Speed (faster local network pulls)
- CI/CD integrations

You can run your own registry using the official Docker image:

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

Docker Registry vs. Docker Repository

- **Registry** = The whole storage system (e.g., Docker Hub)
- **Repository** = A specific collection of images (e.g., `nginx`, `myapp/backend`)

Each repository can have multiple **tags**, representing different versions:

```
nginx:latest
nginx:1.25
```

Authentication & Security

- Public registries may allow anonymous pulls but require a login for pushes.
- Use:

```
docker login
docker push myregistry.com/myimage:tag
```

- Private registries should use TLS (HTTPS).
- You can use access tokens, credentials, or cloud IAM roles depending on the registry.

Registry Use in DevOps Pipelines

Registries are critical in **CI/CD pipelines**:

1. **Build** container images
2. **Push** to a registry
3. **Deploy** containers from the registry

Example with GitHub Actions or Jenkins: build and push images on every commit.

Useful Commands

```
# Pull an image
docker pull ubuntu

# Tag an image
docker tag ubuntu localhost:5000/ubuntu

# Push to a registry
docker push localhost:5000/ubuntu

# Login to Docker Hub
docker login
```

Docker Storage

Docker storage refers to the way data is stored and managed within containers. Since containers are ephemeral by nature, any data written inside the container is lost once it stops or is removed. To persist data, Docker provides several storage mechanisms.

Managing these operations, like creating a writable layer, moving files across layers to enable copy and write → is the Storagr Drivers

Storage Driver Types:

- AUFS → Default driver for Ubuntu
- ZFS
- BTRFS
- Device Mapper
- Overlay / Overlay 2

---

1. Container Layer

Each container has its writable layer where changes are stored. However, this data is **non-persistent** and is lost if the container is deleted.

Use Case: Temporary data that doesn't need to be saved between runs.

---

2. Volumes

Volumes are the preferred mechanism for persisting data in Docker. They are managed by Docker and stored outside of the container's writable layer.

- Created and managed using the Docker CLI or Compose
- Stored in `/var/lib/docker/volumes/` on the host
- Can be shared between containers
- Persist even when containers are deleted

Commands:

```
# Create a volume
docker volume create mydata

# Use volume in a container
docker run -v mydata:/app/data myimage
```

Use Case: Databases, persistent application data

---

3. Bind Mounts

Bind mounts map a directory or file from the host machine directly into the container.

- Can access and modify host files directly
- More flexible but less portable than volumes
- Useful for development and debugging

Example:

```
docker run -v /home/user/code:/app/code myimage
```

Use Case: Sharing source code or configuration files during development

---

4. tmpfs Mounts

A **tmpfs** mount stores data in the host's memory. It is fast but does not persist after the container stops.

Example:

```
docker run --tmpfs /app/cache myimage
```

Use Case: Temporary, sensitive data like caches or secrets

Volume vs. Bind Mounts vs. tmpfs

Feature	Volume	Bind Mount	tmpfs
Persistent	Yes	Yes	No
Managed by Docker	Yes	No	No
Performance	High	Depends	Fast (in-memory)
Use Case	Data storage	Dev files, configs	Caches, secrets

Managing Volumes

```
docker volume ls         # List all volumes
docker volume inspect V  # View volume details
docker volume rm V       # Remove volume
```

Summary

Docker provides multiple storage options to handle different use cases:

- Use **volumes** for persistent data across container lifecycles.
- Use **bind mounts** when you need full control over files on the host.
- Use **tmpfs** for non-persistent, in-memory data.

Choosing the right storage strategy ensures better performance, portability, and reliability in your containerized applications.

Both `-v` and `--mount` are used in Docker to **attach storage** (like volumes or bind mounts) to containers, but there are important differences between them.

Difference

Feature	`-v` / `--volume`	`--mount`
Simpler syntax	Yes	No (more verbose)
More control	No	Yes (more explicit options)
Supports all mount types	No (mainly volumes and binds)	Yes (volumes, bind mounts, tmpfs)
Recommended for	Quick CLI usage	Scripts, Docker Compose, production use

1. `v` or `-volume` : Short, Legacy Style

Format:

```
docker run -v <host_path>:<container_path>:<options> IMAGE
```

Example (bind mount):

```
docker run -v /host/data:/app/data myimage
```

Example (named volume):

```
docker run -v myvolume:/app/data myimage
```

Limitations:

- Less readable
- Implicit (Docker guesses if it's a bind mount or volume)
- Limited options for fine-tuning (e.g., can't explicitly set `type=bind`)

2. `-mount` : Modern, Explicit Style (Recommended)

Format:

```
docker run --mount type=<type>,source=<src>,target=<target>,[options] IMAGE
```

Example (bind mount):

```
docker run --mount type=bind,source=/host/data,target=/app/data myimage
```

Example (named volume):

```
docker run --mount type=volume,source=myvolume,target=/app/data myimage
```

Advantages:

- Clear and explicit: you define `type`, `source`, and `target`
- Better error messages
- More flexible (supports `tmpfs`, read-only, volume labels, etc.)

Which Should You Use?

- Use `v` for **quick commands** and simple cases
- Use `-mount` for **clarity**, **control**, and **scripts/production**

Example:

```
# Using -v
docker run -v myvolume:/data myapp

# Using --mount (same thing, clearer)
docker run --mount type=volume,source=myvolume,target=/data myapp
```

Docker Networks

Docker networks allow containers to communicate with each other, the host, and the outside world. Networking is essential for distributed applications, microservices, and managing containerized systems.

Types of Docker Networks

Docker provides several **built-in network drivers**, each suited for different use cases:

---

1. bridge (default)

- The default network for containers launched without a custom network
- Containers can communicate with each other using IP addresses or container names

Example:

```
docker run -d --name app1 nginx
docker run -d --name app2 nginx
```

By default, both containers are on the `bridge` network and can ping each other via IP.

---

2. host

- Shares the host’s networking namespace
- The container has direct access to the host's network stack
- Only available on Linux

Use case: Performance-sensitive apps, or ones that need to bind to host network interfaces

Example:

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

3. none

- Disables all networking for the container
- Used for sandboxing or extremely isolated environments

Example:

```
docker run --network none nginx
```

---

4. overlay

- Used for multi-host networking (Swarm mode)
- Allows containers on different Docker hosts to communicate
- Requires Docker Swarm

Example:

```
docker network create --driver overlay my-overlay
```

---

5. macvlan

- Assigns a MAC address to containers so they appear as physical devices on the network
- Useful for apps that require direct access to the LAN (e.g., DHCP servers)

Advanced Use Case — requires additional configuration.

Docker Network Commands

List Networks:

```
docker network ls
```

Inspect a Network:

```
docker network inspect bridge
```

Create a Network:

```
docker network create --driver bridge my-net
```

Run Container on a Custom Network:

```
docker run -d --name myapp --network my-net nginx
```

Connect/Disconnect a Container:

```
docker network connect my-net myapp
docker network disconnect my-net myapp
```

Summary Table

Driver	Isolated?	Multi-host?	Custom Names?	Notes
bridge	Yes	No	Yes	Default for standalone containers
host	No	No	No	High performance, no isolation
none	Yes	No	No	Completely disables networking
overlay	Yes	Yes	Yes	Used in Docker Swarm
macvlan	No	Yes	Yes	Appears as real network device
