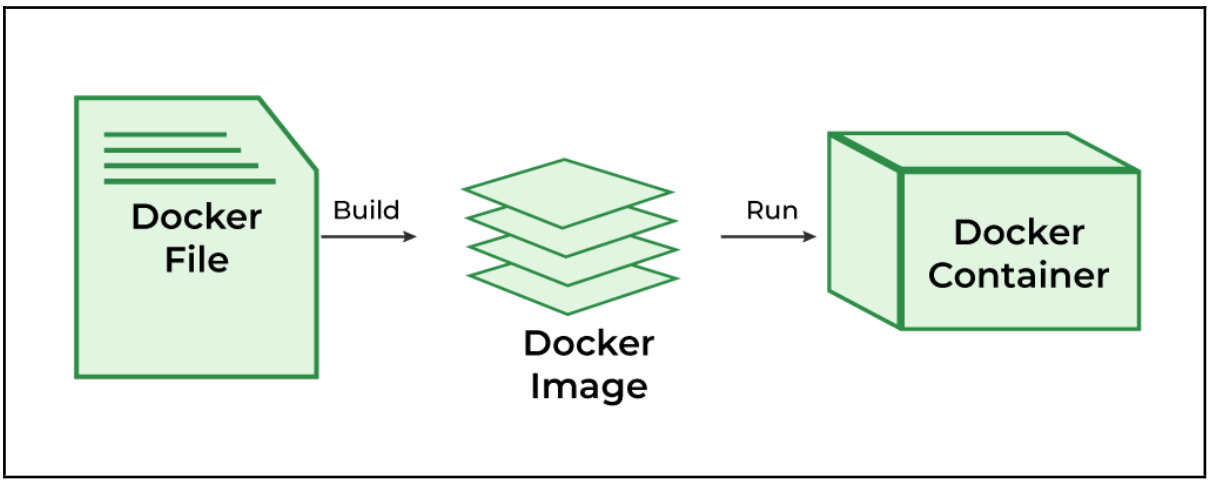
Docker

Docker is a platform for building, sharing and running applications in lightweight, portable environments called containers. These containers bundle an app with everything it needs (code, runtime, dependencies). To run consistently anywhere, solving the ‘it works on my machine’. It ensures portability across different systems and machines, from developer laptops to production servers. It also separates applications from their infrastructures, enabling faster software delivery and efficient DevOps practices.

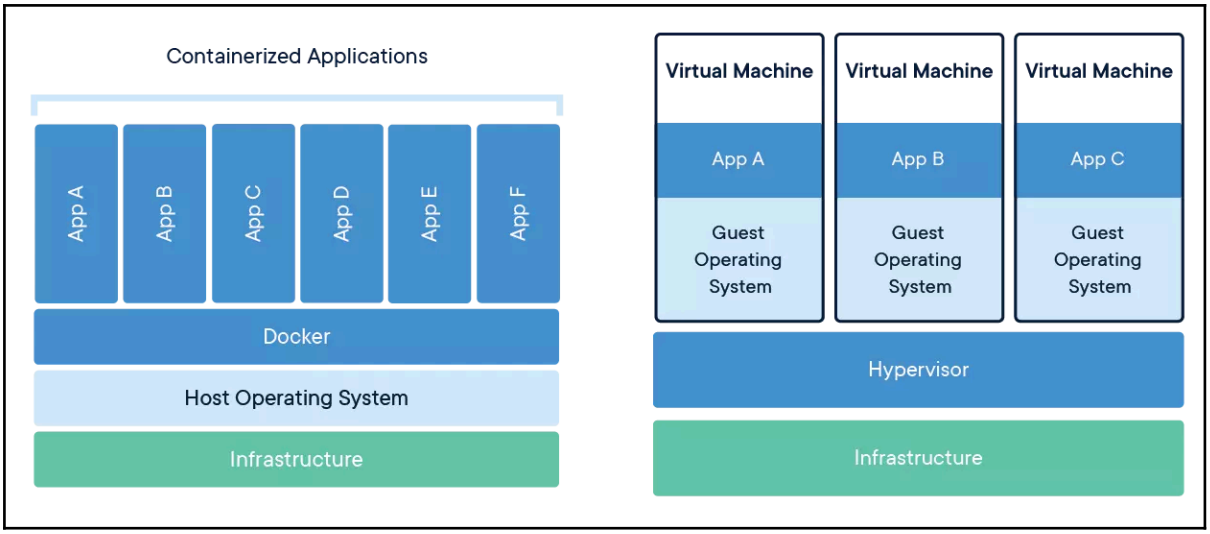


Why Containers ?

A container is like a frozen version of a computer. If it runs on a developer’s laptop, it is guaranteed to run exactly the same way on the QA server and the production server because the operating system and libraries are trapped inside the box with the code.

Unlike a virtual machine (VMs) which needs to start a whole new operating system (taking minutes and GBs of RAM), a container shares the host’s ‘brain’ (the kernel).

1. VM: It's like a house that includes full OS (large size), boots like a PC (long start up time) and uses a lot of RAM (heavy, not efficient).
2. Container: It’s like an apartment that includes only code and tools (small size), starts like an app (fast start up time) and is light (shares the host ‘brain’ kernel).



Important Note : The compatibility issue does not occur on the virtual machine (VM); however, it exists in the Docker images. Therefore, if our app depends on kernel-level behaviour, it may work on a VM but fail in docker.

Isolation is also one of the key advantages of using containers. You can run 2 different versions of the same software on one server without them fighting. For example, you can run a container with java 8 for an old app and a container with java 21 for a new app on the same physical machine. They won't even know the other exists.

To make a container, you write a simple text file called a Dockerfile which serves as the 'recipe' of your box.

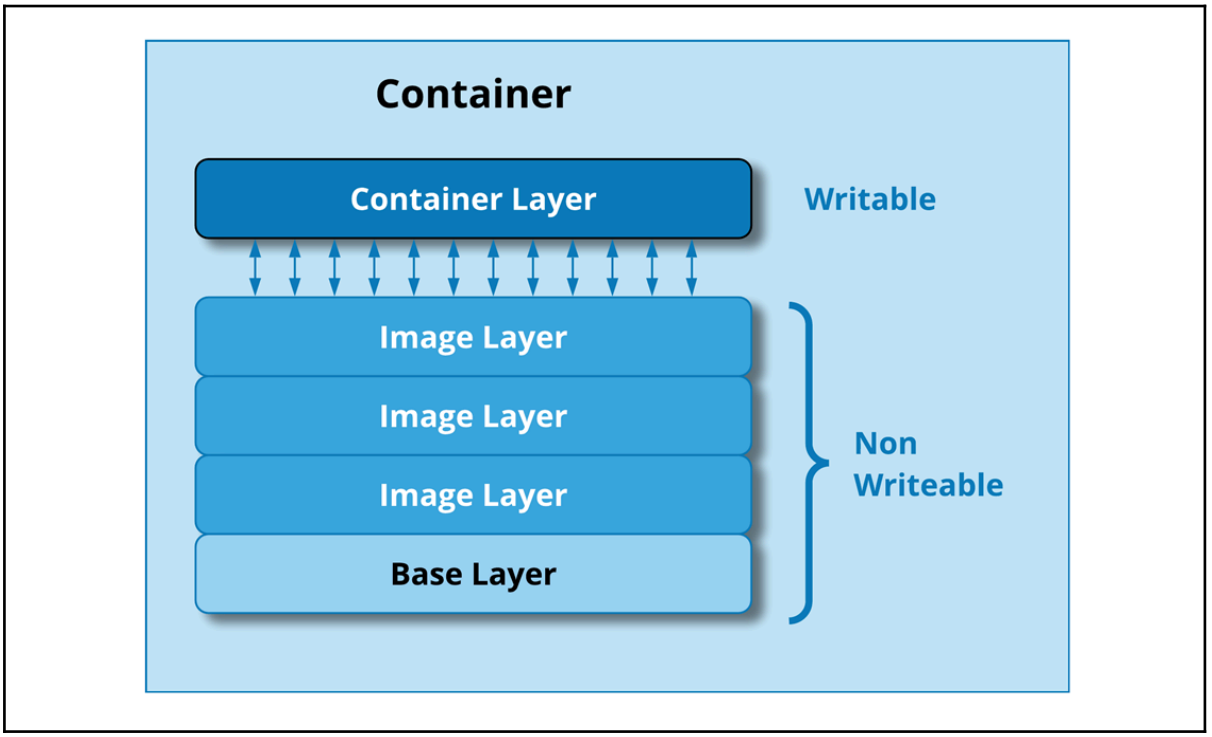
Dockerfile

```
# 1. Start with a specific "kitchen" (Python version)
FROM python:3.10-slim

# 2. Set the workspace
WORKDIR /app

# 3. Bring in the "ingredients" (Code and dependencies)
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .

# 4. Turn it on!
CMD ["python", "app.py"]
```

Docker Images & Containers

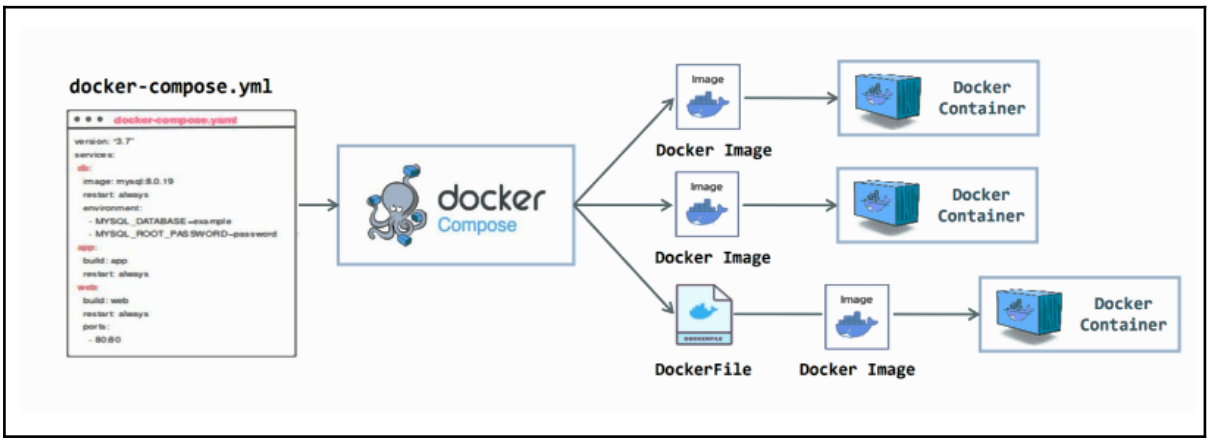
- Docker Image : A docker image is a lightweight, standalone, executable package that includes everything needed to run an application like: code, runtime environment, libraries, dependencies and configuration files. Key characteristics of docker images are :
  I. Read-only template used to create containers.
  II. Built in layers (each instruction in Dockerfile creates a layer).
  III. Immutable - once created, it doesn't change.
  IV. Can be stored in registries (like Docker Hub).
  V. Reusable across multiple containers.

- Docker Container : A container is a running instance of a docker image. It's an isolated, lightweight runtime environment. Key characteristics of docker containers are :
  I. Writable layer added on top of image layers.
  II. Isolated process with its own filesystem, networking, and resources.
  III. Ephemeral - changes are lost when container stops (unless volumes are used).
  IV. Multiple containers can run from the same image.
  V. Portable - runs consistently across different environments.



Docker Compose

Docker compose is a tool for defining and running multi-container docker applications using a single configuration file (docker-compose.yml). It lets us describe multiple services (e.g, app, database, cache) in one file. It defines how containers connect, what images they use, ports, volume and environment variables. It starts and stops all related containers with one command.



In Simpler terms:

- Docker: runs single containers.
- Docker Compose: runs and manages multiple containers as one application.

Why do we need docker compose ?

---

1. It helps to manage multiple containers easily
    a. Without compose: run many docker run commands manually.
    b. With compose:

```
docker compose up
```

2. Consistent environments
    a. Same setup for development , testing, and deployment.
    b. Avoids “works on my machine issues”.
3. Service networking
    a. Containers automatically communicate using service names.
    b. No need to manually configure Docker networks.

4. Centralized configuration
    a. Ports, volumes, environment variables in one file.
    b. Easy to version control.
5. Reproducibilty
    a. Anyone can clone the project and run the entire stack with one command.

When is docker compose typically used?

- Web apps with backend + database.
- ML projects with API + worker + database.
- Local development and testing.
- CI pipelines.