

MODULE 5

:DR. KAVITA V. HORADI

BNMIT

INTRODUCTION TO GIT AND DOC

Introduction:

- What is Git? What is Git History? Why Use It? Where to use Git?
- Key Git Concepts: Repository, Clone, Stage, Commit, Branch, Merge, Pull, Push.
- Introduction to CI/CD
- Introduction to Docker
- Key Components of Docker
- Docker file
- Docker Architecture and its working
- Docker Image
- Docker Container

- Docker Hub,
- Docker Commands

WHAT IS GIT?

- **Git** is a **distributed version control system** used to **track changes in source code** during software development. It lets multiple people collaborate, manage code versions, and revert to previous states of a project.
- **Git** is a free, open-source tool for tracking changes in code and collaborating with others. It's the backbone of modern software development.

KEY FEATURES OF GIT:

Feature

Description

Distributed

Every developer has a full copy of the project and its history.

Version Control

Tracks changes to files over time.

Branching & Merging

Create separate lines of development and combine them when needed.

Fast & Lightweight

Designed for speed and efficiency, even on large projects.

Data Integrity

Uses SHA-1 hashes to ensure content isn't corrupted.



WHY USE GIT?

- Supports **team collaboration** on large projects.
- Enables **experimentation** without fear of breaking the main code.
- Makes it easy to **roll back** mistakes.
- Works offline — you don't need a network to use most features.

HOW IT FITS IN THE GIT WORKFLOW:

- Working Directory → Staging Area → Repository (Git history)
- (edit) git add git commit

- **Working Directory:** Where you make changes to files.
- **Staging Area (Index):** Where you prepare the changes.
- **Repository:** Where committed changes are stored permanently.

GIT HISTORY

- **I. Before Git: The Need for Better Version Control**
- In the early 2000s, many projects used centralized version control systems (VCS) like **CVS** and **Subversion (SVN)**.
- These systems had limitations, especially for large, distributed teams:
 - Central servers were a single point of failure.
 - Merging and branching were slow and complicated.
 - Offline work was difficult.

2. LINUX KERNEL AND BITKEEPER

- The **Linux kernel** is a huge open-source project with many contributors worldwide.
- Initially, the kernel used **BitKeeper**, a proprietary distributed VCS, to manage its source code.
- In 2005, the relationship between the Linux community and BitKeeper's developers soured, and BitKeeper became unavailable for free use.

3. CREATION OF GIT (2005)

- **Linus Torvalds** (creator of Linux) started developing Git in April 2005.
- The goals were:
 - **Speed:** Fast performance on large projects.
 - **Distributed:** Every developer has the full repository.
 - **Robustness:** Protect data integrity.
 - **Simple branching and merging.**
- Within weeks, the first version was ready.

4. RAPID ADOPTION

- Git was adopted quickly by the Linux kernel project.
- Its design influenced many other projects and companies.
- Over time, it became the de facto standard for version control.

5. GIT TODAY

- Git is open-source and maintained by a community of developers.
- Platforms like **GitHub**, **GitLab**, and **Bitbucket** made Git accessible to millions.
- It supports thousands of projects worldwide, from small scripts to huge enterprise codebases.

WHY IS GIT'S HISTORY IMPORTANT?

- Understanding Git's history helps appreciate why:
- It's designed for **speed and distributed workflows**.
- It focuses heavily on **data integrity**.
- Branching and merging are lightweight and central features.
- It's **free and open-source**, unlike the tools it replaced.

WHERE TO USE GIT?

- Git is used **anytime you want to track and manage changes in files**, especially source code.

Common scenarios include:

I. Software Development

- **Individual projects:** Keep track of your own coding progress and experiment safely.
- **Team projects:** Collaborate smoothly, merge everyone's changes, and avoid conflicts.
- **Open-source projects:** Manage contributions from developers worldwide.



2.VERSION CONTROL FOR ANY TEXT-BASED FILES

- Documentation (Markdown, LaTeX)
- Configuration files
- Website source code (HTML, CSS, JavaScript)
- Scripts (Python, Bash, etc.)

3. PROJECT MANAGEMENT AND COLLABORATION

- Track changes in documents and notes.
- Manage versions of design files or scripts.
- Coordinate changes across multiple people without losing data.

4. BACKUP & HISTORY TRACKING

- Keep a history of file changes to rollback anytime.
- Create branches to experiment with new features or ideas without affecting the main version.

5. INTEGRATION WITH HOSTING PLATFORMS

- Use Git with platforms like **GitHub**, **GitLab**, **Bitbucket** for remote collaboration, code reviews, CI/CD, issue tracking, and more.

SUMMARY — USE GIT WHEN YOU:

- Want to **track changes** over time.
- Need to **collaborate** with others on files.
- Want to **manage multiple versions or experiments**.
- Need to **recover old versions** of files easily.
- Work on projects involving **code, text, or configuration**.

INTRODUCTION TO CI/CD

- **What is CI/CD?**
- **CI/CD** stands for:
- **CI** – Continuous Integration
- **CD** – Continuous Delivery (or Continuous Deployment)
- It's a modern **DevOps** practice that automates the **building, testing, and releasing** of software.

CI vs CD – What's the Difference?

Term	What It Means	Purpose
Continuous Integration (CI)	Automatically building and testing code every time a developer pushes changes	Catch bugs early, keep codebase stable
Continuous Delivery (CD)	Automatically preparing builds for release to production (manual trigger)	Faster, safer releases
Continuous Deployment	Automatically releasing every code change to production after it passes tests	Full automation, no manual release step

WHAT HAPPENS IN A CI/CD PIPELINE?

- A **pipeline** is a series of steps that run each time code is updated.






Typical stages:

- **Code** – Developer pushes code to version control (e.g., Git)
- **Build** – Code is compiled or built into an artifact
- **Test** – Automated tests are run (unit, integration, etc.)
- **Package** – Build is packaged (e.g., Docker image)
- **Deploy** – Code is deployed to staging or production environments
- **Monitor** – Logs and performance are tracked after release

TOOLS USED IN CI/CD

- CI/CD Platforms
- Version Control
- Containers
- Orchestration
- Testing

BENEFITS OF CI/CD

-  Faster, more reliable deployments
-  Catch bugs early through automated testing
-  Encourages better collaboration and code quality
-  Repeatable, consistent builds
-  Reduces manual errors in deployment

SUMMARY

- **CI/CD** is about automating software delivery
- **CI** = test code early and often
- **CD** = release faster and more safely
- Combined, they lead to **more reliable software and happier teams**

INTRODUCTION TO DOCKER

- **Docker** is a **platform for developing, shipping, and running applications** in lightweight, portable environments called **containers**.
- A **container** is a self-contained unit that packages:
 - Your application code
 - All its dependencies (libraries, tools, etc.)
 - Configuration files.
- This means the container will **run the same way** no matter where it's deployed (your laptop, a server, or the cloud).

WHY USE DOCKER?

Feature



Consistency



Lightweight



Easy to Reproduce



Portability



Speed

Benefit

“It works on my machine” becomes “It works everywhere”

Uses less resources than virtual machines (VMs)

Containers are based on **Dockerfiles** – scripts that automate builds




Run containers anywhere Docker is installed

Start and stop containers very quickly

HOW DOCKER WORKS (SIMPLIFIED)

- **Dockerfile** – A script with instructions to build your app's container.
- **Image** – A snapshot created from the Dockerfile. Like a template.
- **Container** – A running instance of the image.

Think of it like this:

- *Dockerfile* is the recipe 
- *Image* is the cake mold 
- *Container* is the actual cake 

EXAMPLE

A VERY BASIC DOCKERFILE FOR A PYTHON APP:

- `# Use an official Python base image`
- `FROM python:3.10`
- `# Set working directory`
- `WORKDIR /app`
- `# Copy the app code`
- `COPY ..`
- `# Install dependencies`
- `RUN pip install -r requirements.txt`
- `# Run the app`
- `CMD ["python", "app.py"]`

COMMON DOCKER COMMANDS

Build a Docker image

- **docker build -t myapp .**

Run a container from the image

- **docker run -p 5000:5000 myapp**

List running containers

- **docker ps**

Stop a container

- **docker stop <container_id>**

WHEN SHOULD YOU USE DOCKER?

- Use Docker when you want to:
- Ensure consistency between environments
- Package and deploy apps reliably
- Work with microservices or distributed systems
- Simplify CI/CD pipelines

KEY COMPONENTS OF DOCKER

- 1. **Docker Engine**
- 2. **Docker Image**
- 3. **Docker Container**
- 4. **Dockerfile**
- 5. **Docker Compose**
- 6. **Docker Hub / Registry**
- 7. **Volumes**
- 8. **Networks**

Summary Table

Component

Docker Engine

Image

Container

Dockerfile

Docker Compose

Docker Hub

Volumes

Networks

Purpose

Core runtime that manages everything

Blueprint for containers

Lightweight, isolated running app

Script to build an image

Manages multi-container apps

Repository for storing/sharing images

Persistent storage for containers

Isolated communication between containers

DOCKER FILE

- A **Dockerfile** is a **text file** that contains **instructions** to build a Docker image.
It's like a **recipe** that tells Docker how to package your application, its dependencies, and its environment into a container.
- Docker reads the Dockerfile and builds an **image** step by step.

BASIC STRUCTURE OF A DOCKERFILE

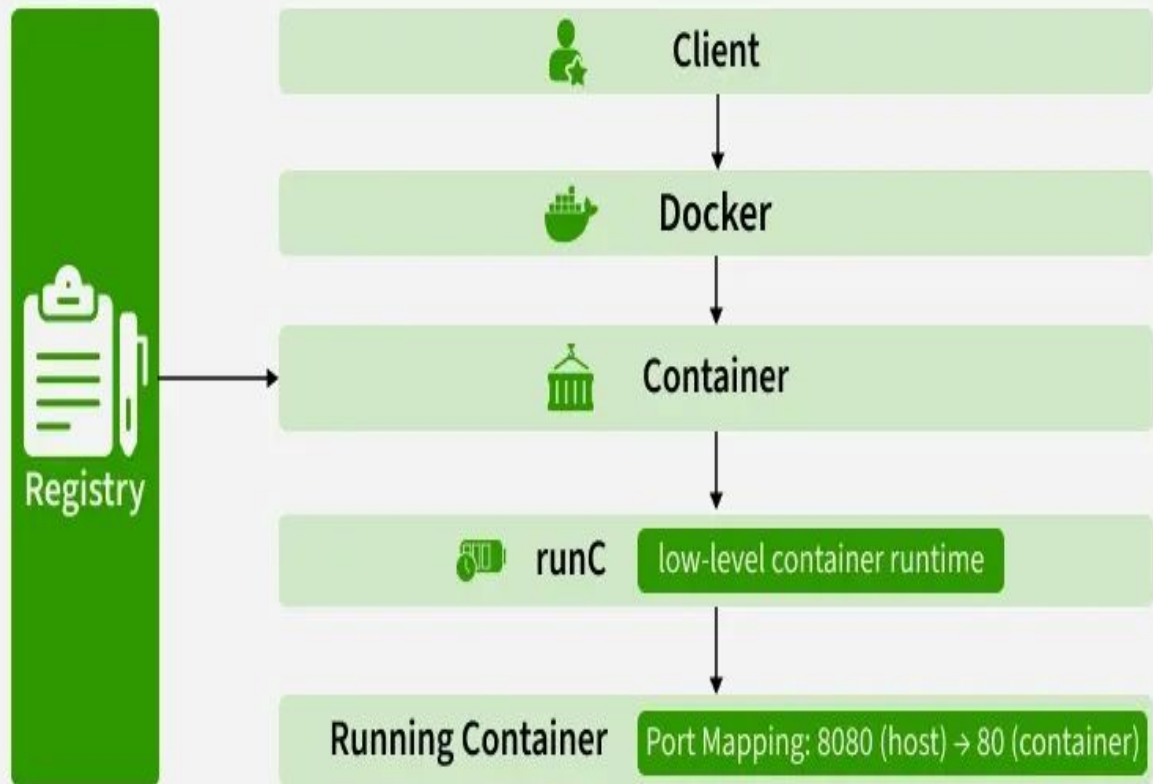
HERE'S A SIMPLE EXAMPLE, FOLLOWED BY AN EXPLANATION:

- # 1. Use a base image
- **FROM** python:3.10
- # 2. Set the working directory in the container
- **WORKDIR** /app
- # 3. Copy files from your host machine into the container
- **COPY** ..
- # 4. Install dependencies
- **RUN** pip install -r requirements.txt
- # 5. Set the command to run the application
- **CMD** ["python", "app.py"]

DOCKER ARCHITECTURE AND ITS WORKING

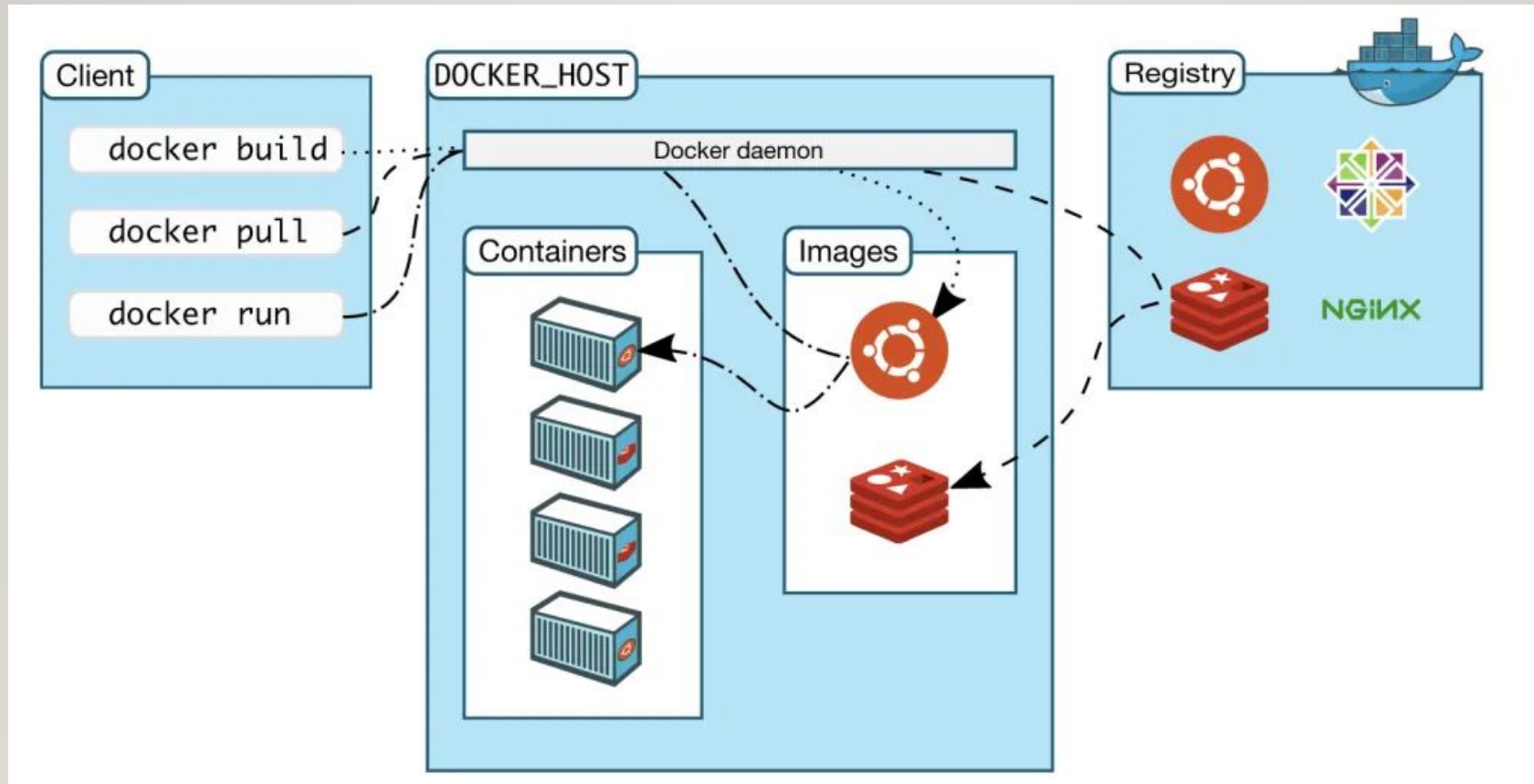
- Docker follows a **client-server architecture**.
- The Docker client communicates with a background process, the Docker Daemon, which does the heavy lifting of building, running, and managing your containers.
- This communication happens over a REST API, typically via UNIX sockets on Linux (e.g., /var/run/docker.sock) or a network interface for remote management.

THE CORE ARCHITECTURAL MODEL



- Docker Client: This is your command center. When you type commands like `docker run` or `docker build`, you're using the Docker Client.
- Docker Host: This is the machine where the magic happens. It runs the Docker Daemon (`dockerd`) and provides the environment to execute and run containers.
- Docker Registry: This is a remote repository for storing and distributing your Docker images.
- This interaction forms a simple yet powerful loop: you use the Client to issue commands to the Daemon on the Host, which can pull images from a Registry to run as containers.

DOCKER ARCHITECTURE OVERVIEW



DOCKER ENGINE:

- The Docker Engine is the heart of the Docker platform.

It comprises two main components:

- Docker Daemon (dockerd): The Docker daemon runs on the host machine and is responsible for managing Docker objects, such as images, containers, networks, and volumes.
- Docker Client: The Docker client is a command-line interface (CLI) tool that allows users to interact with the Docker daemon through commands. Users can build, run, stop, and manage Docker containers using the Docker CLI.

DOCKER IMAGES:

- Docker images are the building blocks of containers.
- They are read-only templates that contain the application code, runtime, system tools, libraries, and other dependencies.
- Docker images are created from Dockerfiles, which are text files containing instructions for building the image layer by layer.

DOCKER CONTAINERS:

- Docker containers are runnable instances of Docker images.
- They encapsulate the application and its dependencies, providing an isolated environment for execution.
- Containers can be created, started, stopped, moved, and deleted using Docker commands.

DOCKER REGISTRY:

- Docker Registry is a centralized repository for storing and sharing Docker images.
- The default public registry is Docker Hub, where users can find a vast collection of images.
- Organizations can also set up private registries to store proprietary images securely.

DOCKER COMPOSE:

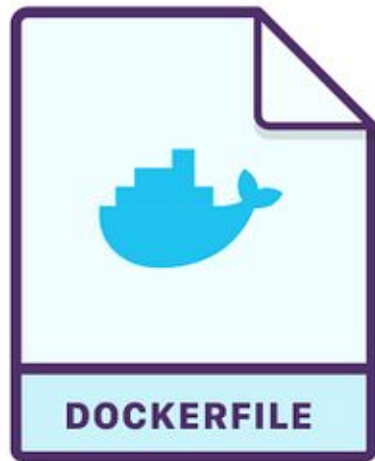
- Docker Compose is a tool for defining and running multi-container Docker applications.
- It uses a YAML file (docker-compose.yml) to specify services, networks, volumes, and other configurations required for the application.
- Docker Compose simplifies the management of complex applications composed of multiple interconnected containers.

DOCKER VOLUMES:

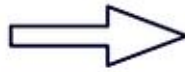
- Docker volumes are used for persisting data generated by and used by Docker containers.
- They provide a way for containers to store and share data independently of the container lifecycle, ensuring data persistence and portability.

DOCKER NETWORKING:

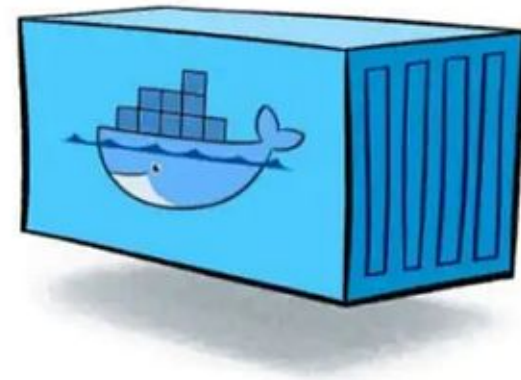
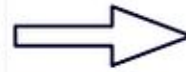
- Docker provides networking capabilities for containers to communicate with each other and with external networks.
- It uses software-defined networks (SDN) to create virtual networks, enabling connectivity and isolation.
- Users can create custom networks, connect containers to networks, and define network policies using Docker commands or Docker Compose.
-



Docker file



Docker Image



Docker Container

CONCLUSION

- In conclusion, Docker's architecture is a sophisticated ecosystem of components that work together seamlessly to enable containerization.
- By understanding the intricacies of Docker architecture, developers can leverage its power to build, ship, and run applications more efficiently and reliably.
- Whether you're a seasoned Docker user or new to containerization, grasping the fundamentals of Docker architecture is essential for mastering modern software development and deployment practices.

THE CORE ARCHITECTURAL MODEL

I. The Docker Daemon (dockerd):

- The Docker Daemon is the persistent background process that acts as the brain of your Docker installation.
- It runs on the **Docker Host**.
- It listens for API requests from the Docker Client.
- It manages all **Docker objects**: images, containers, networks, and volumes.
- It can communicate with other daemons to manage Docker services in a multi-host environment (like a Docker Swarm cluster).

2.THE DOCKER CLIENT:

- The Docker Client is the primary interface through which users interact with Docker.
- This is most commonly the Command Line Interface (CLI).
- It translates user commands like `docker ps` into REST API requests.
- These requests are sent to the Docker Daemon for processing.
- A single client can communicate with multiple daemons.

Common Commands:

- `docker build`: Builds an image from a Dockerfile.
- `docker pull`: Pulls an image from a registry.
- `docker run`: Creates and starts a container from an image.

3.THE DOCKER HOST

- The Docker Host is the physical or virtual machine that provides the complete environment for executing and running containers.

It comprises:

- The Operating System (and its kernel).
- The Docker Daemon.
- Images that have been pulled or built.
- Running Containers.
- Networks and Storage components.

4.THE DOCKER REGISTRY

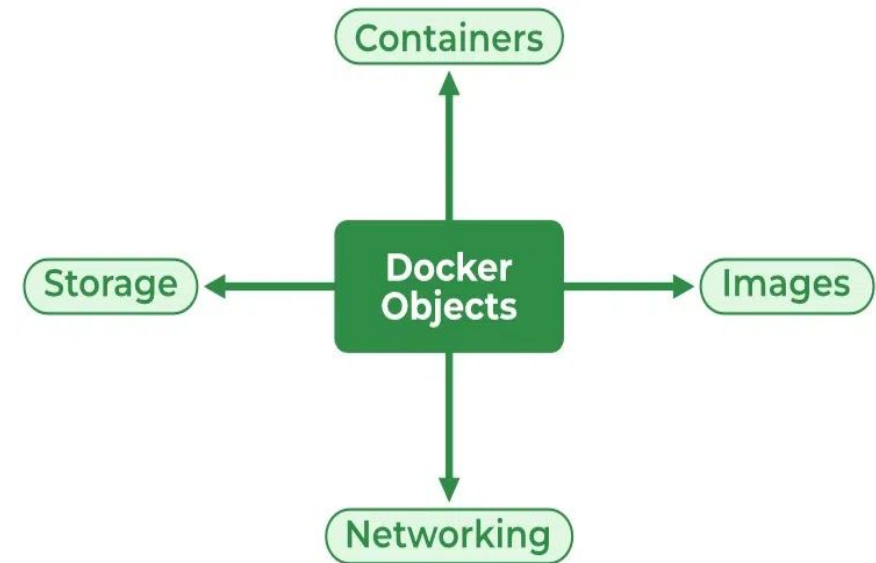
- A Docker Registry is a stateless, scalable storage system for Docker images.
- **Public Registry:** The default public registry is **Docker Hub**, which contains a vast collection of community and official images.
- **Private Registries:** Organizations often use private registries (like Harbor, AWS ECR, or Google Artifact Registry) to store proprietary images for security and control.

IMAGE LIFECYCLE COMMANDS:

- `docker pull <image_name>`: Downloads an image from a configured registry to your local Docker Host.
- `docker push <image_name>`: Uploads a local image to a registry.

DOCKER OBJECTS

- Whenever we are using a docker, we are creating and use images, containers, volumes, networks, and other objects. Now, we are going to discuss docker objects:-



1. Images

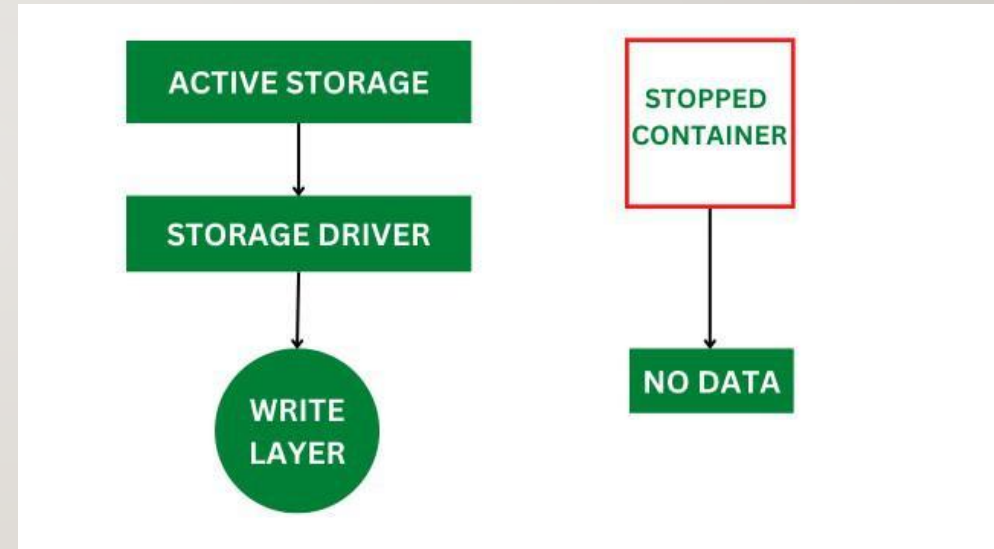
- An image is a read-only, inert template that contains the instructions for creating a Docker container. Think of it as a blueprint or a class in object-oriented programming.
- It's built from a Dockerfile, a simple text file defining the steps to assemble the image.
- Images are built in layers, where each instruction in the Dockerfile corresponds to a layer. This layered architecture makes builds and distribution incredibly efficient.

2. Containers

- A container is a runnable, live instance of an image. If an image is the blueprint, a container is the house built from that blueprint.
- You can create, start, stop, move, or delete containers using the Docker API or CLI.
- Each container is isolated from other containers and the host machine, having its own filesystem, networking, and process space.
- You can run multiple containers from the same image.

3. Storage

- Since a container's writable layer is ephemeral (data is lost when the container is deleted), Docker provides robust solutions for data persistence. Storage driver controls and manages the images and containers on our docker host.



TYPES OF DOCKER STORAGE

- Docker provides multiple storage options to persist, share, and manage data across containers and hosts.
- **Volumes:** The preferred mechanism. Volumes are managed by Docker and stored in a dedicated area on the host filesystem (e.g., `/var/lib/docker/volumes/` on Linux). They are designed to survive the container lifecycle.
- **Bind Mounts:** Allow you to map a file or directory from the host machine directly into a container. This is very useful for development, where you might want to share source code with a container.
- **tmpfs Mounts:** In-memory storage that is temporary and never written to the host filesystem. Useful for sensitive data or high-performance temporary files.

DOCKER NETWORKING

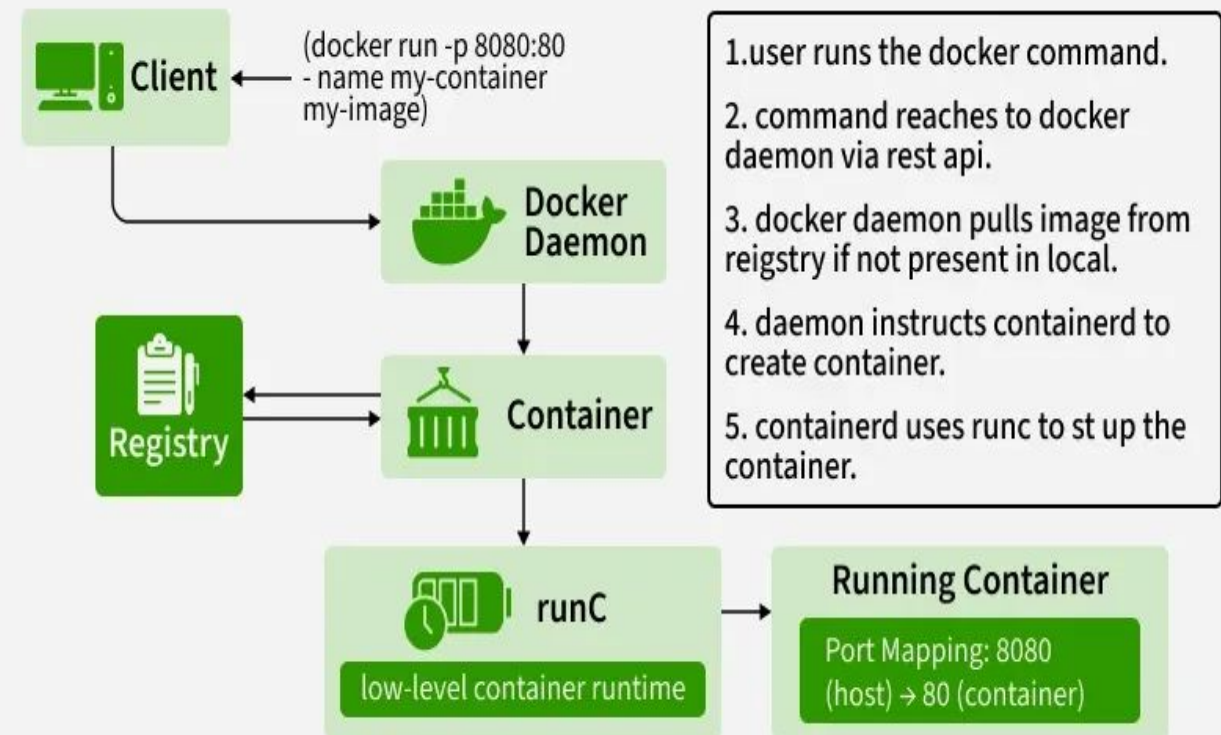
- Docker networking provides complete isolation for docker containers. It means a user can link a docker container to many networks. It requires very less OS instances to run the workload.

TYPES OF DOCKER NETWORK

- **Bridge:** It is the default network driver. We can use this when different containers communicate with the same docker host.
- **Host:** When you don't need any isolation between the container and host then it is used.
- **Overlay:** For communication with each other, it will enable the swarm services.
- **None:** It disables all networking.
- **macvlan:** Assigns a unique MAC address to a container, making it appear as a physical device on your network.

LET'S TRACE A COMMON COMMAND TO SEE HOW ALL THESE COMPONENTS WORK IN HARMONY.

- You run the command: **docker run -d -p 80:80 nginx**



-
- **Client:** The Docker Client sends a REST API request to the Docker Daemon to create and run a container from the nginx image.
 - **Daemon:** The Daemon receives the request. It first checks if the nginx image exists locally on the Host.
 - **Registry (Pull):** If the image is not found locally, the Daemon contacts the configured Registry (Docker Hub by default) and pulls the nginx image.
 - **Runtime (containerd):** The Daemon hands the image and run-configuration over to containerd.
 - **Runtime (runc):** containerd uses runc to create a new container. runc interfaces with the Linux kernel to create isolated **namespaces** and limit resources with **cgroups**.
 - **Execution:** The container is started. Docker maps port 80 of the host to port 80 of the nginx container, as requested by the -p 80:80 flag. The Nginx process runs as PID 1 inside the container's isolated PID namespace.

DOCKER IMAGE

- A **Docker image** is a **lightweight, standalone, and executable package** that includes:
 - The application code
 - Dependencies (libraries, binaries)
 - Runtime environment
 - Configuration files
- In short: **An image is a snapshot of your application's environment** — used to create containers.

Docker Image vs Container

Docker Image

Read-only template

Created with `docker build`

Doesn't change when running

Think: **Blueprint**

Docker Container

Running instance of an image

Created with `docker run`

Can change while running (data, state)

Think: **House built from that blueprint**



WHAT'S INSIDE A DOCKER IMAGE?

- Docker images are made up of **layers**. Each line in a Dockerfile adds a new layer.
- FROM python:3.10 # Layer 1
- WORKDIR /app # Layer 2
- COPY .. # Layer 3
- RUN pip install -r requirements.txt # Layer 4
- CMD ["python", "app.py"] # Layer 5

HOW ARE DOCKER IMAGES CREATED?

- You create a Docker image using a Dockerfile and the docker build command:
- `docker build -t my-python-app .`
- This creates an image named my-python-app.

DOCKER CONTAINER

- **What is a Docker Container?**
- A **Docker container** is a **lightweight, standalone, and executable environment** that runs your application and its dependencies based on a **Docker image**.
-  Think of it this way:
- A **Docker image** is like a **blueprint** 
- A **Docker container** is the **running instance** of that blueprint

WHAT DOES A CONTAINER INCLUDE?

- Application code
- Required libraries
- Configuration files
- Environment variables
- Everything needed to run the app

Key Features of Docker Containers

Feature



Lightweight



Isolated



Portable



Fast startup



Ephemeral

Description

No need to boot an entire OS like a virtual machine

Each container runs independently

Runs the same in dev, test, and production

Containers launch in seconds

Containers can be stopped, removed, and recreated easily

DOCKER COMMANDS

- https://docs.docker.com/get-started/docker_cheatsheet.pdf

GENERAL COMMANDS

- Start the docker daemon `docker -d`
- Get help with Docker.
- Can also use `--help` on all subcommands `docker --help` Display system-wide information
`docker info`

NOTE:

- Refer prescribed text books and reference books