Day 1 – Docker Fundamentals

1. What is Docker and why do we need it?

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

◆ What is Docker?

Docker is a tool designed to make it easier to:

-> Create lightweight, portable application environments (called containers).

-> Deploy those containers across different environments consistently.

-> Run applications isolated from the host system.

A Docker container is a standalone unit that contains everything needed to run a piece of software — code, runtime, libraries, and system tools.

◆ Why Do We Need Docker?

Here's why Docker is needed and widely used:

a. Consistency Across Environments

-> "It works on my machine" becomes a thing of the past. Docker containers ensure your application runs the same in dev, test, and production.

b. Isolation

-> Each container runs in its own isolated environment. You can run multiple containers on the same machine without conflict.

c. Lightweight

-> Containers share the host OS kernel, making them more resource-efficient and faster to start than full virtual machines.

d. Portability

-> Build once, run anywhere — on your local machine, cloud, or CI/CD pipelines.

e. Easy Scalability and Microservices

-> Docker works well with microservices architecture. You can split applications into smaller services and scale them independently.

f. DevOps and CI/CD

-> Docker integrates seamlessly into DevOps workflows, enabling automated testing, deployment, and version control.

2. Give a brief description on Containers v/s Virtual Machines.

Ans) Here's a brief comparison between Containers and Virtual Machines (VMs):

◆ Containers

-> Definition: Lightweight, portable units that package an application with its dependencies and share the host OS kernel.

-> Startup Time: Fast (seconds).

-> Resource Usage: Minimal — no separate OS, shares host OS.

-> Isolation Level: Process-level (less than VMs, but sufficient for most apps).

-> Use Case: Ideal for microservices, CI/CD, and rapid development/deployment.

-> Example Tool: Docker

🔹 Virtual Machines (VMs)

-> Definition: Emulate an entire physical machine with its own full OS and virtualized hardware.

-> Startup Time: Slower (minutes).

-> Resource Usage: Heavy — each VM includes a full OS.

-> Isolation Level: Strong — full OS-level isolation.

-> Use Case: Suitable for running multiple different OSes, legacy systems, or high-security applications.

-> Example Tool: VMware, VirtualBox, KVM

3. Give a simple Docker flow used in organizations

Ans) Here is the flow:

🛠️ 1. Develop the Application

-> Write your app code (e.g., in Node.js, Python, Java, etc.).

-> Create a Dockerfile to define how the app should be containerized.

🧱 2. Build the Docker Image

-> Run the build command

-> This packages your app and dependencies into an image.

🧪 3. Test Locally in a Container

-> Run your container locally to test it

🐳 4. Push to a Docker Registry

-> Tag and push the image to a registry (like Docker Hub, AWS ECR, or GCP Artifact Registry):

🔁 5. CI/CD Pipeline Integration

-> Use Jenkins, GitHub Actions, GitLab CI, etc. to: Pull code from Git repo, Build Docker image, run tests, Push image to registry.

🚀 6. Deploy to Production

-> Use tools like Docker Compose, Kubernetes, or Swarm to deploy your containers.

4. Explain Docker Architecture.

Ans) Docker follows a client-server architecture that enables you to build, ship, and run applications in containers. Here's a breakdown of its core components and how they interact:

🧱 1. Docker Components

a. Docker Client (docker CLI)

-> It’s the main way users interact with Docker.

-> When you run a command like docker build, it sends the command to the Docker daemon.

-> It can communicate with local or remote daemons.

b. Docker Daemon (dockerd)

-> A background service that manages Docker objects (containers, images, networks, volumes).

-> Listens for Docker API requests and handles them.

-> It builds, runs, and manages containers.

c. Docker Images

-> Read-only templates used to create containers.

-> Includes the application code, libraries, dependencies, and OS (base image).

-> Built using Dockerfile.

d. Docker Containers

-> Running instances of Docker images.

-> Isolated environments that run your application.

-> Lightweight and share the OS kernel with the host.

e. Docker Registry (e.g., Docker Hub, GitHub Container Registry, Amazon ECR)

-> A storage and distribution system for Docker images.

-> Docker pulls images from the registry (docker pull) and pushes images (docker push).

🔐 2. How It Works – A Typical Flow

-> Developer writes a Dockerfile.

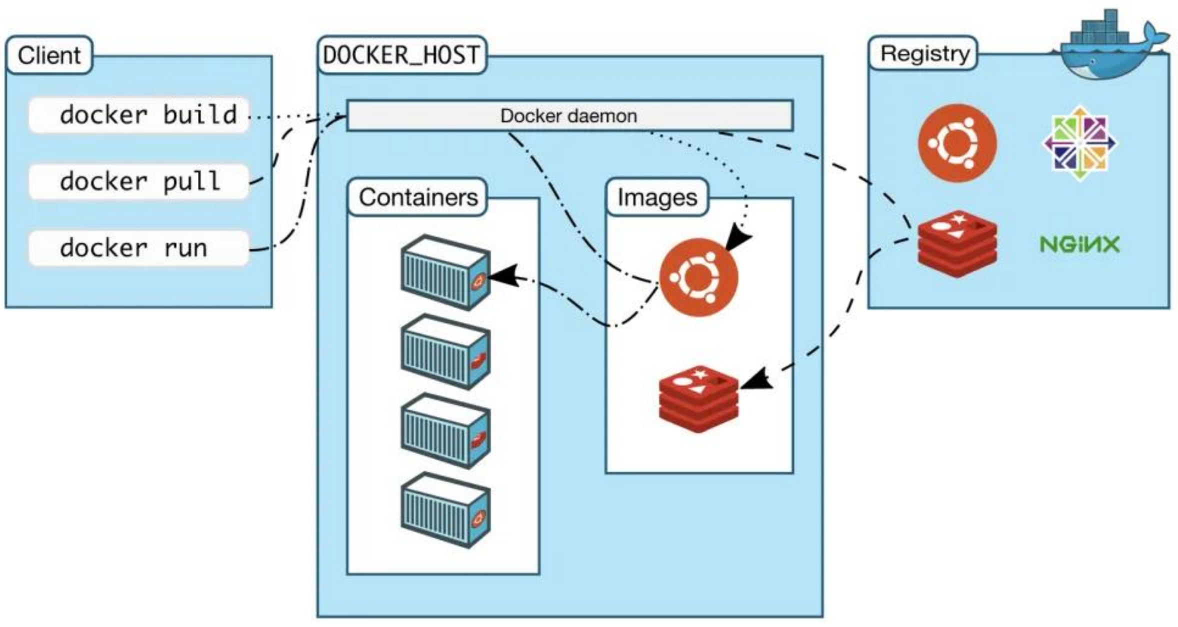
-> Builds the image: docker build -t myapp .

-> Runs the container: docker run -d myapp

-> Docker client sends request to Docker daemon.

-> Docker daemon pulls image from registry (if not available locally).

-> Daemon creates and runs container using container runtime (like containerd).


Client
docker build
docker pull
docker run
DOCKER_HOST
Docker daemon
Containers
Images
Registry
NGINX