***Docker Introduction***

**What is docker?**



Docker is like the container ship of the software world. 🐳 It's a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow **developers to package up an application with all the parts it needs, such as libraries and other dependencies, and ship it all out as one package.**

So, Docker is an open source

The magic of Docker is that it ensures your application always runs the same, regardless of where it's deployed. This eliminates the classic "but it worked on my machine" problem. With Docker, you can:

* Build your code into containers.
* Share your containers through Docker Hub.
* Run those containers on any machine with Docker installed.

Think of it as having a consistent, portable environment for your apps that runs the same way across development, testing, and production. This brings efficiency, consistency, and scalability to software development and deployment.

**What is a Container?**

A container is like a **portable mini-computer within your computer**. It packages up an application and all its dependencies (like libraries and settings) so it can run consistently across different environments. (Multiple operating systems, windows, Linux distributions, etc)

Imagine you're baking a cake and you put all the ingredients and tools you need into a single box. No matter where you take that box, you'll be able to bake the same cake every time. Similarly, a container ensures that your application runs the same way, whether it's on your laptop, a server, or in the cloud.

It's a way to ensure that everything the application needs is right there with it, making it easier to deploy, manage, and scale.



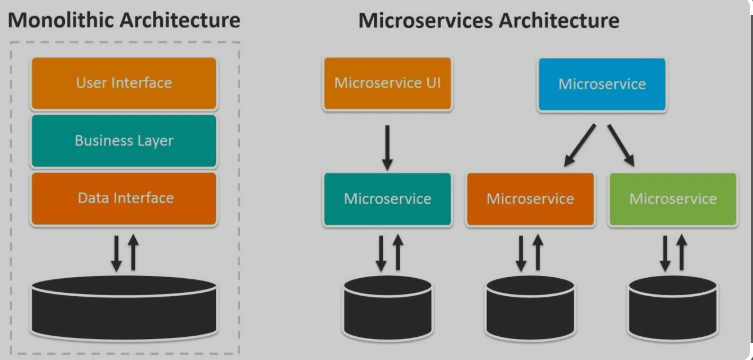
**Why Did Docker Become So Popular?**

Docker gained popularity for several reasons:

* **Portability**: Applications run consistently across different environments (development, testing, production) because they carry their own configurations and dependencies.
* **Scalability**: It enables easy scaling up or down of applications.
* **Efficiency**: Containers are lightweight compared to virtual machines (VMs), requiring less memory and storage space.
* **DevOps Integration**: Fits perfectly into modern DevOps practices, automating and streamlining workflows.

### Microservices vs. Monolithic Architecture

* **Monolithic Architecture**: The application is built as a single, large unit. While it's simpler initially, it becomes more challenging to manage, scale, and maintain as it grows.
* **Microservices Architecture**: The application is broken down into smaller, independent services, each responsible for a specific functionality. This makes it easier to develop, deploy, and scale services independently, but it can introduce complexity in managing inter-service communication.



Quick comparison:



### VMs vs. Containers

* **Virtual Machines (VMs)**: VMs run entire operating systems on virtual hardware. They're isolated, but heavyweight in terms of resource usage.
* **Containers**: Containers share the host OS kernel but run isolated processes. They're lightweight and efficient compared to VMs.

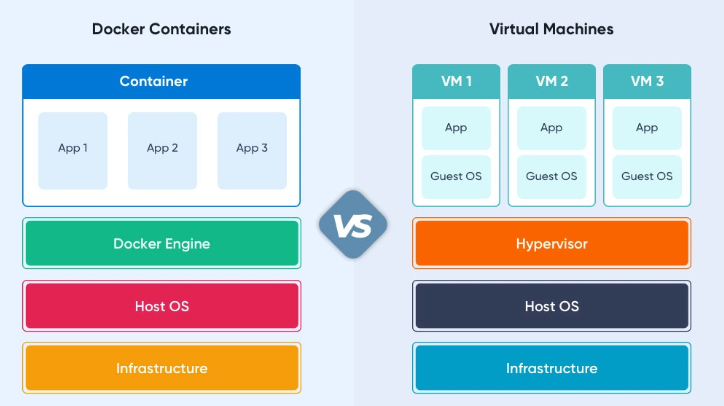
### Advantages of Docker

* **Resource Efficiency**: Containers use less system resources compared to VMs.
* **Faster Start-up Time**: Containers can start almost instantly, unlike VMs that need to boot up an entire OS.
* **Improved Security**: Isolation ensures that each container is secure and doesn't affect others.
* **Ease of Management**: Docker provides tools for managing the lifecycle of containers, making it easier to deploy, update, and maintain applications.

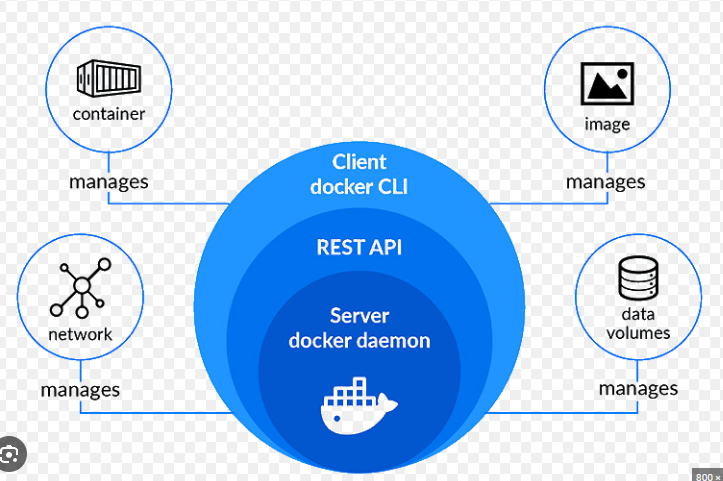
**Hypervisor:** A hypervisor, also known as a virtual machine monitor (VMM), is software that creates and manages virtual machines (VMs). It allows multiple operating systems to run concurrently on a host computer by abstracting and sharing the underlying hardware resources.

### Advantages of Hypervisors

* **Resource Efficiency**: Optimizes the utilization of physical hardware.
* **Isolation**: Each VM runs independently, ensuring that issues in one do not affect others.
* **Scalability**: Easily create, manage, and scale VMs according to needs.
* **Cost Savings**: Reduces the need for additional physical servers, lowering hardware costs.



### Docker Architecture:

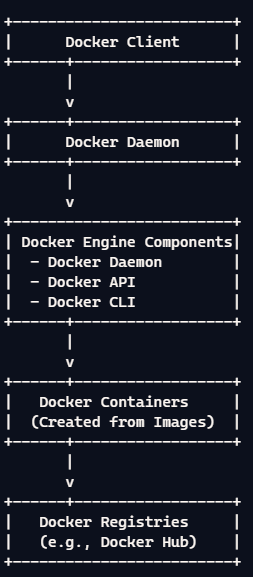


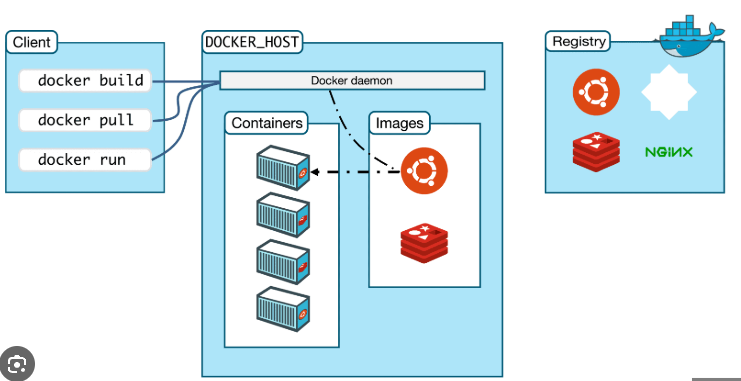
### Docker Architecture Components

1. **Docker Client**:
   * The interface you interact with to manage Docker. Commands you enter are sent from the Docker Client to the Docker Daemon.
2. **Docker Daemon**:
   * The brain of Docker. It runs on the host machine, listens for Docker API requests, and manages Docker objects like images, containers, networks, and volumes.
3. **Docker Engine**:
   * The core of Docker, it includes the Docker Daemon, Docker API, and the CLI (Command Line Interface). Essentially, it manages the interaction between the Client and Daemon.
4. **Docker Images**:
   * Read-only templates that define how a container should be created. Think of it as a snapshot of a specific environment. Images are used to create containers.
5. **Docker Containers**:
   * The runnable instance of an image. It's where your applications run. Containers are lightweight and share the host OS kernel, but remain isolated from each other.
6. **Docker Registries**:
   * Centralized repositories where Docker images are stored and shared. Docker Hub is the most commonly used public registry.
7. **Docker Compose**:
   * A tool for defining and running multi-container Docker applications. With a YAML file, you can specify the services, networks, and volumes for your app.

### Docker Architecture Flow

1. **Client sends a command**: You use the Docker Client to send commands (e.g., to build, run, or stop a container).
2. **Daemon processes the command**: The Docker Daemon receives the command, processes it, and interacts with the underlying OS to execute it.
3. **Image retrieval**: If needed, the Docker Daemon pulls the required image from a Docker Registry.
4. **Container creation**: The Daemon uses the image to create a container and allocate resources for it.
5. **Application execution**: The application runs inside the container, isolated from other processes.



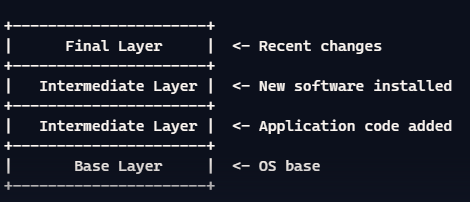


### Docker Images and Layering

Docker images are built in layers. Each layer represents a set of filesystem changes (like adding a file or installing a package) and is built on top of the previous layer. This layering makes images efficient and flexible:

* **Base Layer**: The initial layer, often an OS like Ubuntu.
* **Intermediate Layers**: Each command in the Dockerfile adds a new layer. For example, installing software, copying files, etc.
* **Final Layer**: The final, runnable state of the image.

When you update an image, Docker only rebuilds the layers that have changed, making the process faster and conserving resources. Here's a visual representation:



### Docker Networking

Docker provides several network drivers to manage container communication:

1. **Bridge Network**: The default network driver. It creates a private internal network on a single Docker host, allowing containers to communicate with each other.
2. **Host Network**: Removes network isolation between the Docker host and Docker containers. The container shares the host's network stack.
3. **Overlay Network**: Used for multi-host communication, it enables containers on different Docker hosts to communicate with each other.
4. **None Network**: Disables all networking for a container. It's useful for security isolation.

### Docker Volumes

Docker volumes are used to persist data generated by and used by Docker containers. They are stored outside the container's filesystem, meaning they are not destroyed when the container is removed. There are a few types of volumes:

1. **Volumes**: Managed by Docker and stored in a part of the host filesystem. They're the preferred method for persisting data.
2. **Bind Mounts**: These can be stored anywhere on the host system and mounted into containers. They're tightly coupled with the directory structure of the host machine.
3. **Tmp-fs Mounts**: Temporary storage on the host system that is only available for the life of the container.

Eg: docker run -d -v myvolume:/app/data myimage

Here, myvolume is a Docker-managed volume that mounts to /app/data in the container. This setup allows the data in /app/data to persist even if the container is deleted.

**Docker Commands: Cheat Sheet**



### Basic Commands

* **docker --version**: Show the Docker version installed.
* **docker info**: Display system-wide information.
* **docker help**: Get help on Docker commands.

### Image Commands

* **docker images**: List all images.
* **docker pull [image]**: Pull an image from a registry.
* **docker build [options] [path]**: Build an image from a Dockerfile.
* **docker rmi [image]**: Remove an image.
* **docker tag [source\_image] [target\_image]**: Tag an image with a new name.

### Container Commands

* **docker ps**: List running containers.
* **docker ps -a**: List all containers, including stopped ones.
* **docker run [options] [image] [command]**: Run a new container.
* **docker stop [container]**: Stop a running container.
* **docker start [container]**: Start a stopped container.
* **docker restart [container]**: Restart a container.
* **docker rm [container]**: Remove a container.
* **docker exec [options] [container] [command]**: Run a command in a running container.
* **docker logs [container]**: Fetch the logs of a container.
* **docker inspect [container/image]**: Inspect details of a container or image.
* **docker attach [container]**: Attach local standard input, output, and error streams to a running container.

### Network Commands

* **docker network ls**: List all networks.
* **docker network create [name]**: Create a new network.
* **docker network rm [name]**: Remove a network.
* **docker network inspect [name]**: Display detailed information about a network.
* **docker network connect [network] [container]**: Connect a container to a network.
* **docker network disconnect [network] [container]**: Disconnect a container from a network.

### Volume Commands

* **docker volume ls**: List all volumes.
* **docker volume create [name]**: Create a new volume.
* **docker volume rm [name]**: Remove a volume.
* **docker volume inspect [name]**: Display detailed information about a volume.

### Other Useful Commands

* **docker login**: Log in to a Docker registry.
* **docker logout**: Log out from a Docker registry.
* **docker push [image]**: Push an image to a registry.
* **docker save [image]**: Save an image to a tar archive.
* **docker load [archive]**: Load an image from a tar archive.
* **docker top [container]**: Display the running processes of a container.
* **docker stats [container]**: Display a live stream of container(s) resource usage statistics.
* **docker stats [container id]: Shows container stat’s**

### Docker Compose Commands

* **docker-compose up**: Build, create, start, and attach to containers for a service.
* **docker-compose down**: Stop and remove containers, networks, images, and volumes.
* **docker-compose ps**: List containers.
* **docker-compose start**: Start existing containers for a service.
* **docker-compose stop**: Stop running containers without removing them.
* **docker-compose restart**: Restart running containers.
* **docker-compose build**: Build or rebuild services.

Command to remove all containers and remove all images in one command:

* docker ps -aq | xargs docker stop | xargs docker rm

Docker prune: Remove un-used Docker resources, such as containers, networks, images, and volumes,

* docker system prune
* docker container prune
* docker image prune
* docker network prune

**Docker Compose:**

Docker Compose is a tool that helps you define and manage multi-container Docker applications.

Docker Compose is a YAML-based configuration file called docker-compose.yml that defines how to set up and run multiple Docker containers as a single service.

Instead of manually running docker run commands for each container, you can define everything in one file and run it with a single command.

### Key Features

1. **Multi-Container Deployment**: Run multiple interconnected containers simultaneously.
2. **Service Definitions**: Define all necessary containers in a single file.
3. **Network Configuration**: Easily manage networking between containers.
4. **Volume Management**: Handle persistent storage for your containers.
5. **Environment Variables**: Set environment-specific configurations.

### Benefits

* **Simplicity**: One command to start and stop your entire application stack.
* **Portability**: Easy to share and deploy the same configuration across different environments.
* **Consistency**: Ensures all components start in a defined order and are properly interconnected.

### Commands

* **Starting Services**: Run all the services defined in docker-compose.yml with docker-compose up.
* **Stopping Services**: Stop all services with docker-compose down.
* **Scaling Services**: Scale the number of containers for a service with docker-compose up --scale web=3.

### Common Use Cases

* **Microservices**: Manage microservices architecture with multiple interdependent containers.
* **Development**: Quickly spin up development environments.
* **Testing**: Easily replicate production environments for testing purposes.