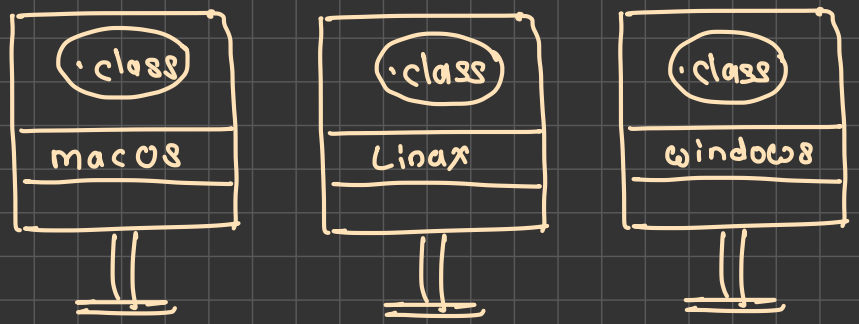
Virtualization

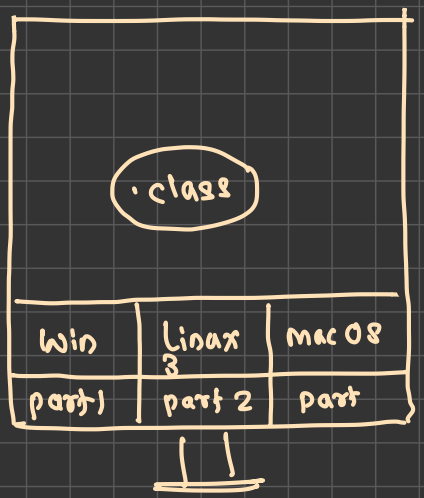
$$$ → capital Expenditure → investment for maching

Operational Expenditure → administrator ....

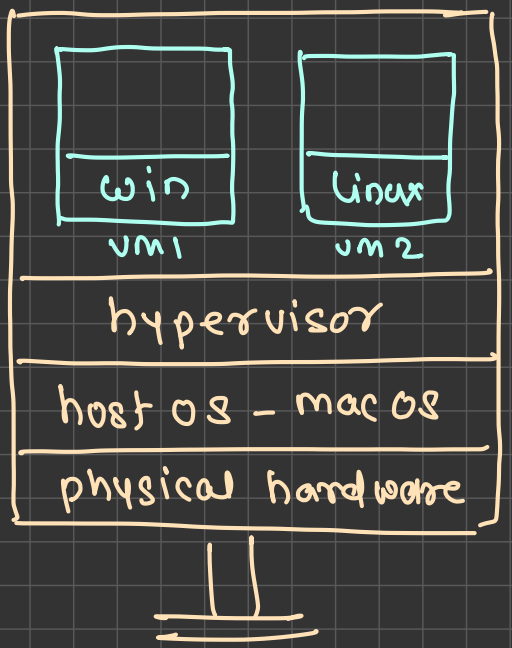


physical machines

multi-booting system



hardware virtualization



Introduction

- Virtualization is the creation of a virtual -- rather than actual -- version of something, such as an operating system (OS), a server, a storage device or network resources
- Virtualization uses software that simulates hardware functionality in order to create a virtual system
- This practice allows IT organizations to operate multiple operating systems, more than one virtual system and various applications on a single server
- Types

  Share

  1. Network virtualization : physical network
  2. Storage virtualization : physical storage
  3. Data virtualization : data
  4. Desktop virtualization : desktop
  5. Application virtualization : application
  6. Hardware virtualization : hardware
  7. OS virtualization → containerization : OS

Hardware Virtualization

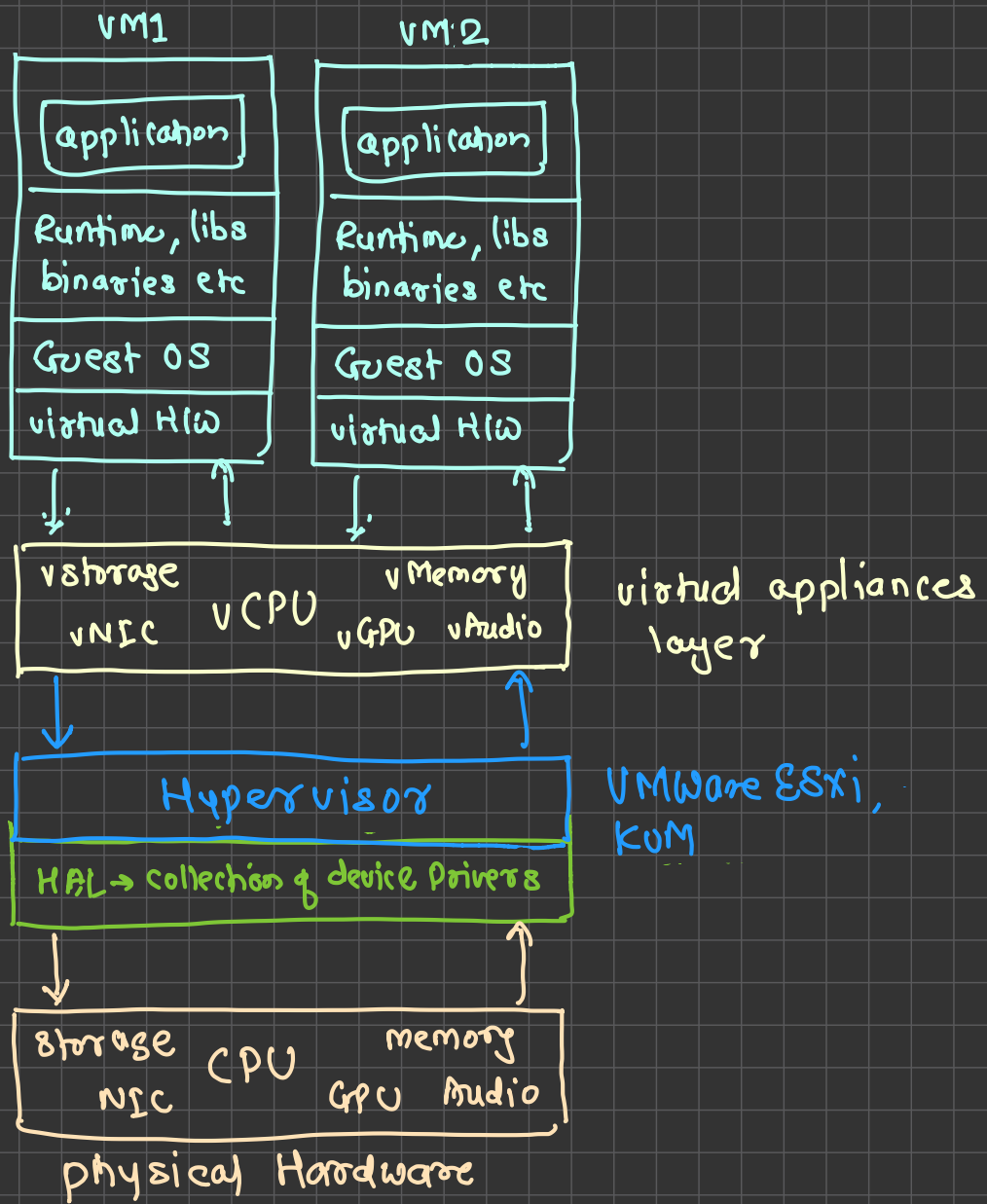
- Hardware virtualization or platform virtualization refers to the creation of a virtual machine that acts like a real computer with an operating system
- The process of masking the hardware resources like
  - CPU
  - Storage
  - Memory

  → Creating virtual form
- For example, a computer that is running Microsoft Windows may host a virtual machine that looks like a computer with the Ubuntu Linux operating system; Ubuntu-based software can be run on the virtual machine
- The process of creating Machines

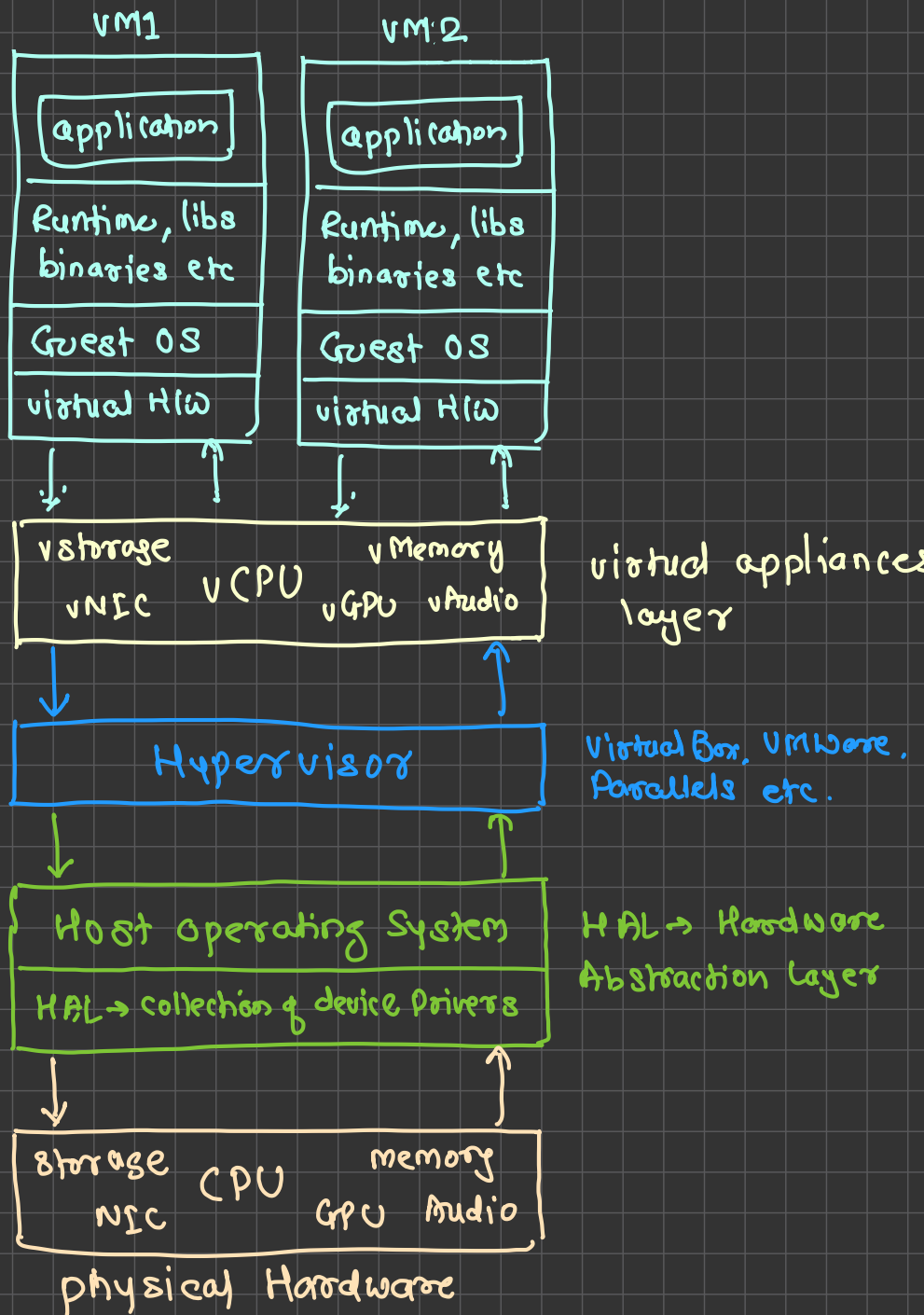
Virtual Machine

- A virtual machine is the emulated equivalent of a computer system that runs on top of another system
- Virtual machines may have access to any number of resources
  - Computing power - through hardware-assisted but limited access to the host machine's CPU
  - Memory - one or more physical or virtual disk devices for storage
  - A virtual or real network interfaces
  - Any devices such as
    - video cards,
    - USB devices,
    - other hardware that are shared with the virtual machine
- If the virtual machine is stored on a virtual disk, this is often referred to as a disk image

Type I : Bare Metal → cloud providers



Type II : Hosted → Developers or tester or users

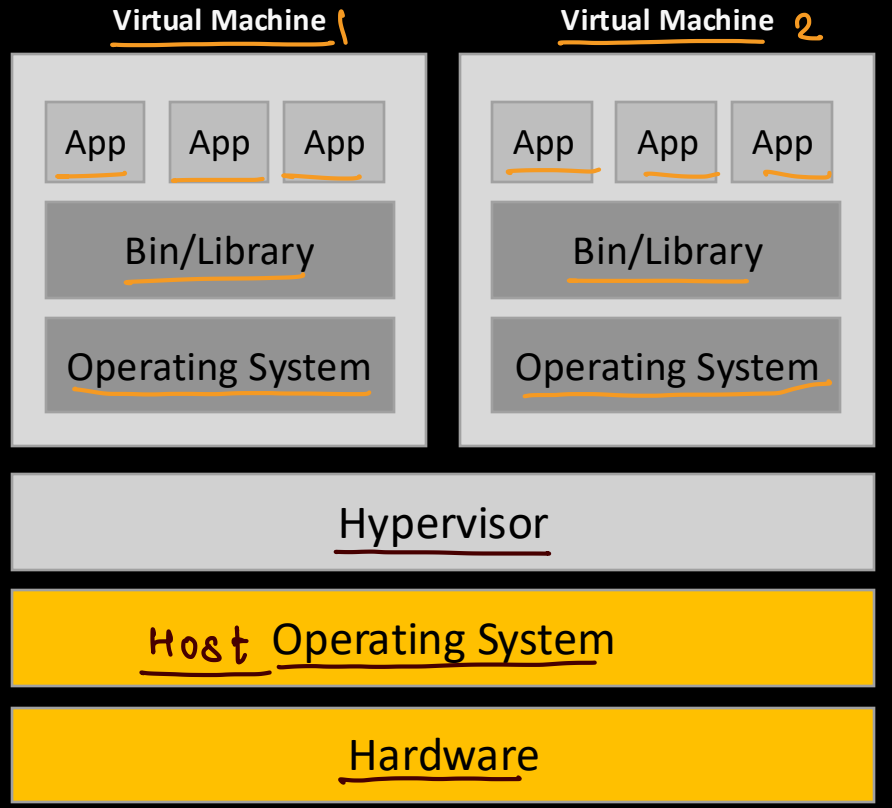


Types of hardware virtualization

- **Type I** : Bare Metal Hypervisor
  - A Type 1 hypervisor runs directly on the host machine's physical hardware, and it's referred to as a bare-metal hypervisor
  - It doesn't have to load an underlying OS first → Here is No host OS
  - With direct access to the underlying hardware and no other software, it is more efficient and provides better performance
  - It is best suited for enterprise computing or data centers
  - E.g. VMware ESXi, Microsoft Hyper-V server and open source KVM
- **Type II** : Hosted Hypervisor
  - A Type 2 hypervisor is typically installed on top of an existing OS, and it's called a hosted hypervisor (→ host OS)
  - It relies on the host machine's pre-existing OS to manage calls to CPU, memory, storage and network resources
  - E.g. VMware Fusion, Oracle VM VirtualBox, Oracle VM Server for x86, Oracle Solaris Zones, Parallels and VMware Workstation

Virtualized Deployment → using virtual machines

- It allows you to run multiple Virtual Machines (VMs) on a single physical server's CPU
- Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application
- Virtualization allows better utilization of resources in a physical server and allows better scalability because
  - an application can be added or updated easily
  - reduces hardware costs
- With virtualization you can present a set of physical resources as a cluster of disposable virtual machines
- Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware



Advantages of virtualization

- **Lower costs**
  - Virtualization reduces the amount of hardware servers necessary within a company and data center
  - This lowers the overall cost of buying and maintaining large amounts of hardware
- **Easier disaster recovery**
  - Disaster recovery is very simple in a virtualized environment
  - Regular snapshots provide up-to-date data, allowing virtual machines to be feasibly backed up and recovered
  - Even in an emergency, a virtual machine can be migrated to a new location within minutes
- **Easier testing**
  - Testing is less complicated in a virtual environment
  - Even if a large mistake is made, the test does not need to stop and go back to the beginning
  - It can simply return to the previous snapshot and proceed with the test
- **Quicker backups**
  - Backups can be taken of both the virtual server and the virtual machine
  - Automatic snapshots are taken throughout the day to guarantee that all data is up-to-date
  - Furthermore, the virtual machines can be easily migrated between each other and efficiently redeployed
- **Improved productivity**
  - Fewer physical resources results in less time spent managing and maintaining the servers
  - Tasks that can take days or weeks in a physical environment can be done in minutes
  - This allows staff members to spend majority of their time on more productive tasks, like raising revenue and fostering business initiatives

Containerization

OS virtualization

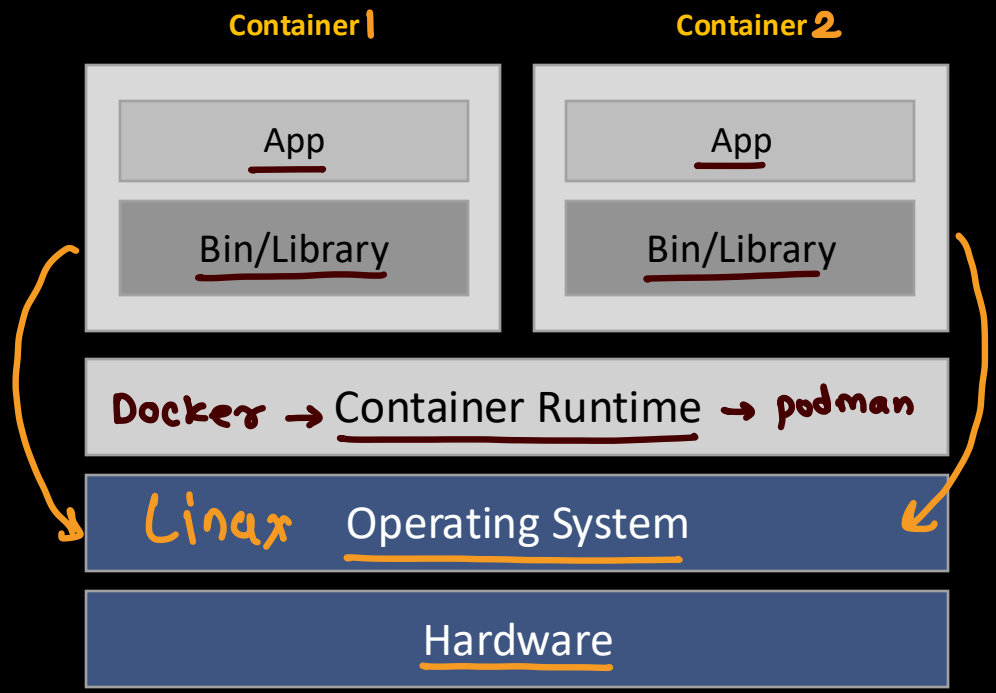
What is Containerization

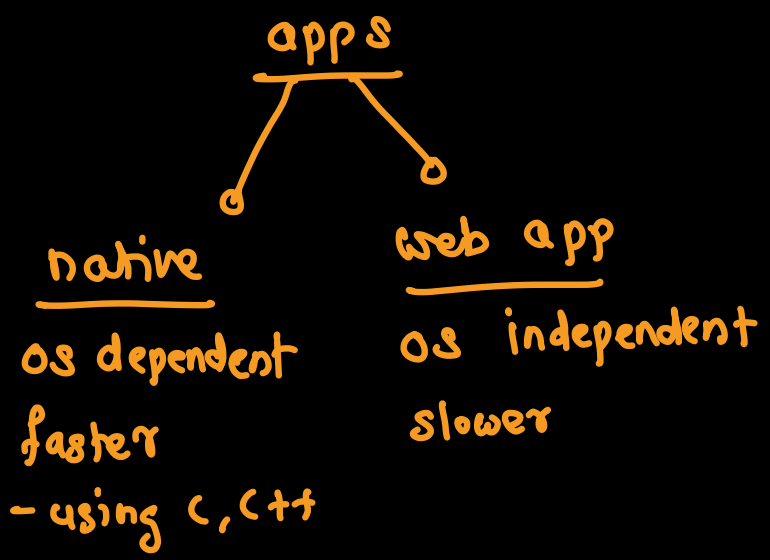
program, libraries, dependencies .... → package → container

- Containerization is the packaging of software code with just the operating system (OS) libraries and dependencies required to run the code to create a single lightweight executable—called a container—that runs consistently on any infrastructure
- More portable and resource-efficient than virtual machines (VMs), containers have become the de facto compute units of modern cloud-native applications
- Containerization allows developers to create and deploy applications faster and more securely
- With traditional methods, code is developed in a specific computing environment which, when transferred to a new location, often results in bugs and errors
- For example, when a developer transfers code from a desktop computer to a VM or from a Linux to a Windows operating system
- Containerization eliminates this problem by bundling the application code together with the related configuration files, libraries, and dependencies required for it to run
- This single package of software or "container" is abstracted away from the host operating system, and hence, it stands alone and becomes portable—able to run across any platform or cloud, free of issues

Container deployment

- Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications
- Therefore, containers are considered lightweight
- Similar to a VM, a container has its own filesystem, CPU, memory, process space, and more
- As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions





Containerization vs Virtualization

Virtual Machine → mutable	Container – immutable
Hardware level virtualization	OS virtualization
Heavyweight (bigger in size)	Lightweight (smaller in size)
Slow provisioning	Real-time and fast provisioning
Limited Performance	Native performance
Fully isolated	Process-level isolation
More secure	Less secure
Each VM has separate OS	Each container can share OS resources
Boots in minutes	Boots in seconds
Pre-configured VMs are difficult to find and manage	Pre-built containers are readily available
Can be easily moved to new OS	Containers are destroyed and recreated
Creating VM takes longer time	Containers can be created in seconds

Benefits

- **Portability**
  - A container creates an executable package of software that is abstracted away from (not tied to or dependent upon) the host operating system, and hence, is portable and able to run uniformly and consistently across any platform or cloud
- **Agility**
  - The open source Docker Engine for running containers started the industry standard for containers with simple developer tools and a universal packaging approach that works on both Linux and Windows operating systems
  - The container ecosystem has shifted to engines managed by the Open Container Initiative (OCI)
  - Software developers can continue using agile or DevOps tools and processes for rapid application development and enhancement
- **Speed**
  - Containers are often referred to as “lightweight,” meaning they share the machine’s operating system (OS) kernel and are not bogged down with this extra overhead
  - Not only does this drive higher server efficiencies, it also reduces server and licensing costs while speeding up start-times as there is no operating system to boot
- **Fault isolation**
  - Each containerized application is isolated and operates independently of others
  - The failure of one container does not affect the continued operation of any other containers
  - Development teams can identify and correct any technical issues within one container without any downtime in other containers
  - Also, the container engine can leverage any OS security isolation techniques—such as SELinux access control—to isolate faults within containers

Benefits

- Efficiency
  - Software running in containerized environments shares the machine's OS kernel, and application layers within a container can be shared across containers
  - Thus, containers are inherently smaller in capacity than a VM and require less start-up time, allowing far more containers to run on the same compute capacity as a single VM. This drives higher server efficiencies, reducing server and licensing costs
- Ease of management
  - Container orchestration platform automates the installation, scaling, and management of containerized workloads and services
  - Container orchestration platforms can ease management tasks such as scaling containerized apps, rolling out new versions of apps, and providing monitoring, logging and debugging, among other functions. Kubernetes, perhaps the most popular container orchestration system available, is an open source technology (originally open-sourced by Google, based on their internal project called Borg) that automates Linux container functions originally
  - Kubernetes works with many container engines, such as Docker, but it also works with any container system that conforms to the Open Container Initiative (OCI) standards for container image formats and runtimes
- Security
  - The isolation of applications as containers inherently prevents the invasion of malicious code from affecting other containers or the host system
  - Additionally, security permissions can be defined to automatically block unwanted components from entering containers or limit communications with unnecessary resources

Docker

What is Docker ?

- Docker is an open source platform that enables developers to build, deploy, run, update and manage containers—standardized, executable components that combine application source code with the operating system (OS) libraries and dependencies required to run that code in any environment
- Why docker?
  - It is an easy way to create application deployable packages → images → containers
  - Developer can create ready-to-run containerized applications
  - It provides consistent computing environment
  - It works equally well in on-prem as well as cloud environments
  - It is light weight compared to VM

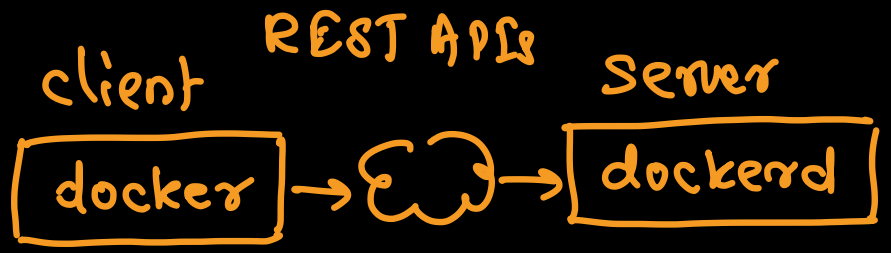
Little history about Docker

- Docker Inc, started by Solomon Hykes, is behind the docker tool
- Docker Inc started as PasS provider called as dotCloud
- In 2013, the dotCloud became Docker Inc
- Docker Inc was using LinuX Containers (LXC) before version 0.9
- After 0.9 (2014), Docker replaced LXC with its own library libcontainer which is developed in Go programming language
- Its not the only solution for containerization
  - "FreeBSD Jails", launched in 2000
  - LXD is next generation system container manager build on top of LXC and REST APIs
  - Google has its own open source container technology lmctfy (Let Me Contain That For You)
  - Rkt is another option for running containers
    - podman

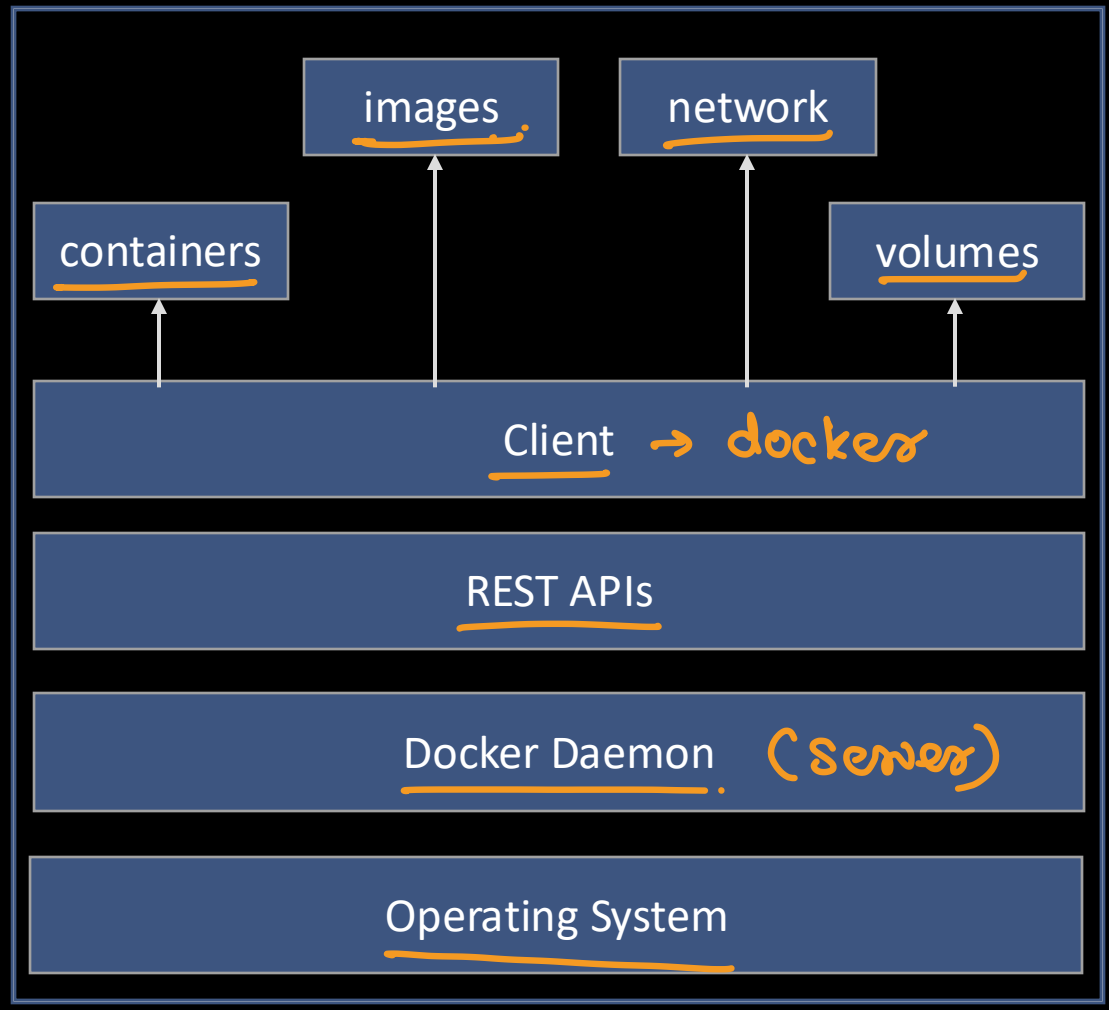
Docker Architecture → client-server architecture

REST APIs

- Docker daemon (dockerd)
  - Continuous running process
  - Manages the containers
- REST APIs
  - Used to communicate with docker daemon
- Client (docker)
  - Provides command line interface
  - Used to perform all the tasks



daemon → background process



libcontainer

- Docker has replaced LXC by libcontainer, which is used to manage the containers
- Libcontainer uses
  - Namespaces
    - Creates isolated workspace which limits what container can see
    - Provides a layer of isolation to the container
    - Each container runs in a separate namespace
    - Processes running in a namespace can interact with other processes or use resources which are the part of the same namespace
    - E.g. process ID, network, IPC, Filesystem
  - Control Groups (cgroups)
    - Used to share the available resources to the conainers
    - It optionally enforces limits and constraints on resource usage
    - It limits how much a container can use
    - E.g. CPU, Disk space, memory

libcontainer

- Union File System (UnionFS)
  - It uses layers
  - It is a lightweight and very fast FS
  - Docker uses different variants of UnionFS
    - Aufs (advanced multi-layered unification filesystem)
    - BtrFS (B-Tree FS)
    - VFS (Virtual FS)
    - Devicemapper

Docker Objects

- **Images**: read only template with instructions for creating docker containers
- **Container**: running instance of a docker image
- **Network**: network interface used to connect the containers to each other or external networks
- **Volumes**: used to persist the data generated by and used by the containers
- **Registry**: private or public collection of docker images
- **Service**: used to deploy application in a docker multi node cluster

- stacks

What is Docker image ?

image → container(s)
class → object

- A **Docker image** is a **read-only template** that contains everything needed to run an application — including the **code**, **runtime**, **libraries**, **environment variables**, and **configuration files**
  - instructions
- Images are the **blueprint** for creating Docker containers
- An image acts like a snapshot of a filesystem and its configuration
- When you run an image, Docker creates a container based on that image
- You can build your own images or pull prebuilt ones from registries (like Docker Hub)
- Docker images are built using a **layered file system**
  - Unionfs
  - Base Laye: The foundation — e.g., ubuntu, alpine, node
  - Intermediate Layers: Contain instructions like installing dependencies or copying code
  - Top Layer: The final application code and entry point

```
# Base image
FROM python:3.11

# Set working directory
WORKDIR /app

# Copy source code
COPY . .

# Install dependencies
RUN pip install -r requirements.txt

# Command to run
CMD ["python", "app.py"]
```

Docker image commands

- **docker build:** Build a new image from a Dockerfile
- **docker image ls:** List all images on the system
- **docker pull:** Download an image from a registry
- **docker push:** Upload an image to a registry
- **docker image rm:** Remove an image
- **docker history:** Show layers of an image
- **docker image inspect:** View image metadata

Advantages

- **Portability:** Runs consistently anywhere Docker is supported
- **Reusability:** Same base image can be used for multiple apps
- **Efficiency:** Layered architecture saves space
- **Versioning:** Tags (:v1, :v2) track image versions
- **Automation:** Integrates with CI/CD pipelines easily

Docker Container

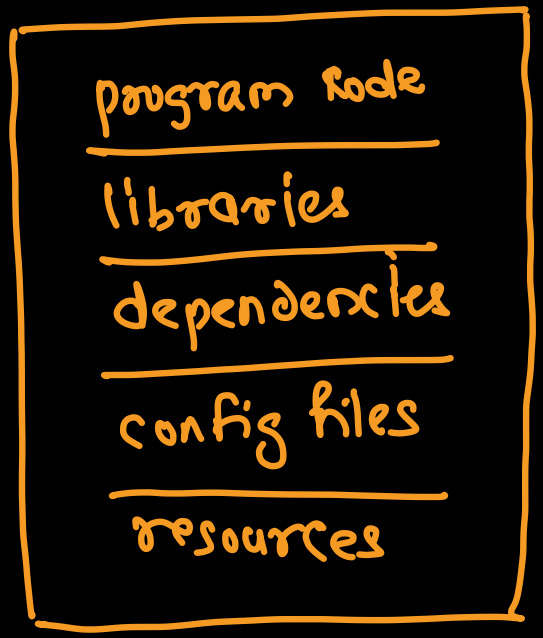
Docker Container

image → run → container

class → object

- It is a running aspect of docker image
- Contains one or more running processes
- It is a self-contained environment → contains everything required to run app
- It wraps up an application into its own isolated box (application running inside a container has no knowledge of any other applications or processes that exist outside the container) → sandbox
- A container can not modify the image from which it is created
- It consists of
  - Your application code
  - Dependencies
  - Networking
  - Volumes
- Containers are stored under /var/lib/docker
- This directory contains images, containers, network volumes etc

Docker Home



container

Basic Operations

- Creating container
- Starting container
- Running container
- Listing running containers
- Listing all containers
- Getting information of a container
- Stopping container
- Deleting container

Commands

Command	Description	Example
`docker run`	Create and start a container	`docker run -d nginx`
`docker ps`	List running containers	`docker ps`
`docker ps -a`	List all containers (including stopped)	`docker ps -a`
`docker stop`	Stop one or more running containers	`docker stop myapp`
`docker start`	Start an existing container	`docker start myapp`
`docker restart`	Restart a container	`docker restart myapp`
`docker rm`	Remove a stopped container	`docker rm myapp`
`docker logs`	View container logs	`docker logs myapp`
`docker exec`	Run a command inside a running container	`docker exec -it myapp bash`
`docker inspect`	View detailed information about a container	`docker inspect myapp`
`docker stats`	Display resource usage (CPU, memory, etc.)	`docker stats`
`docker cp`	Copy files between host and container	`docker cp file.txt myapp:/app/`
`docker rename`	Rename a container	`docker rename old new`
`docker commit`	Create new image from a container’s changes	`docker commit myapp myimage:v2`

Attaching a container

- There are two ways to attach to a container
- Attach
  - Used to attach the container
  - Uses only one input and output stream
  - Task
    - Attach to a running container
- Exec
  - Mainly it is used for running a command inside a container
  - Task
    - Execute a command inside container

Publishing port on container

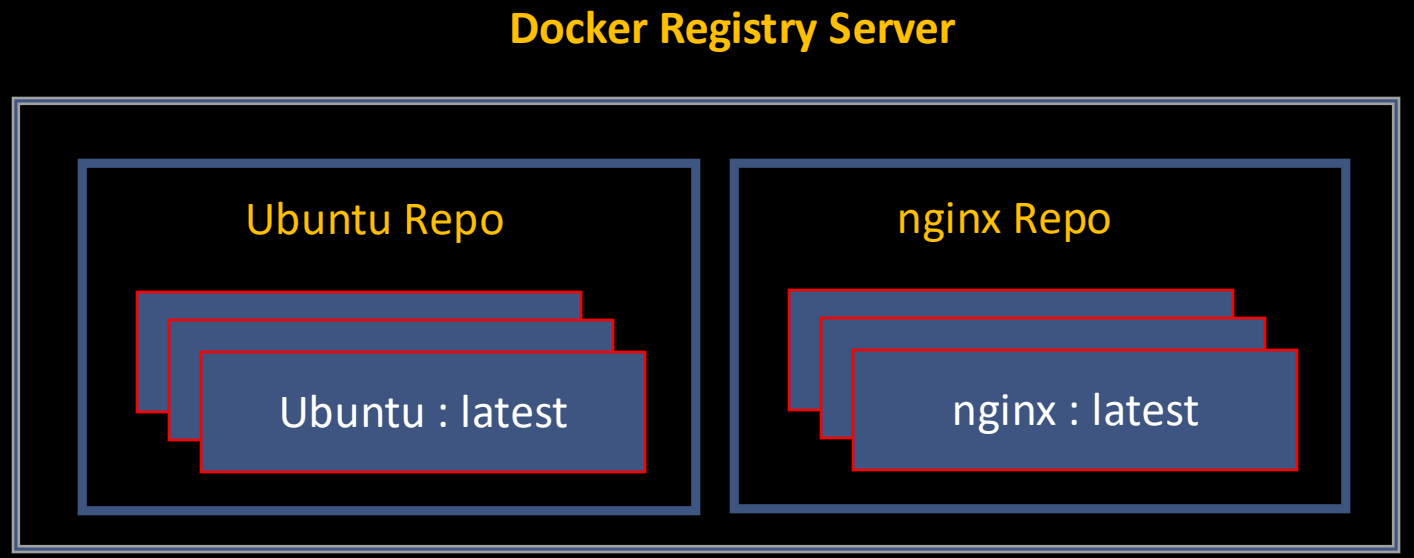
- Publishing a port is required to give an external access to your application
- Port can be published only at the time of creating a container
- You can not update the port configuration on running container
- Task
  - Run a httpd container with port 8080 published, to access apache externally

Docker Images (Advanced)

Docker Image

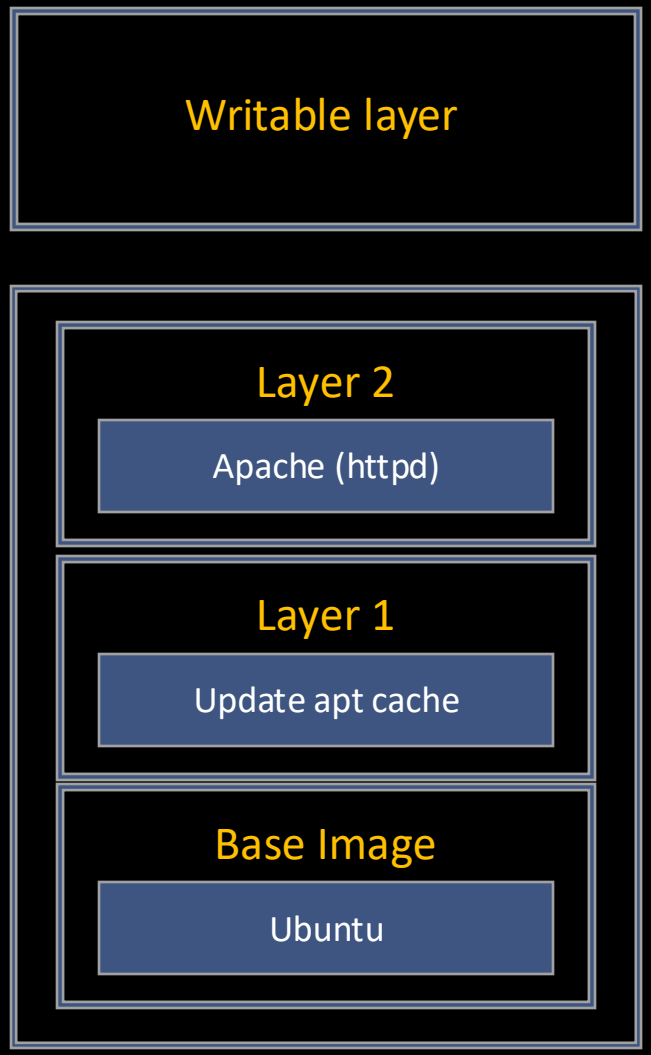
- Read-only instructions to run the containers
- It is made up of different layers
- Repositories hold images
- Docker registry stores repositories
- To create a custom image
  - Commit the running container
  - Use a Dockerfile

- Task
  - Create a container
  - Create a directory and a file within it
  - Commit the container to create a new image



Layered File System

- Docker images are made of layered FS
- Docker uses UnionFS for implementing the layered docker images
- Any update on the image adds a new layer
- All changes made to the running container are written inside a writable layer



Dockerfile

- The Dockerfile contains a series of instructions paired with arguments
- Each instruction should be in upper-case and be followed by an argument
- Instructions are processed from top to bottom
- Each instruction adds a new layer to the image and then commits the image
- Upon running, changes made by an instruction make it to the container

Dockerfile instructions

- FROM
- ENV
- RUN
- CMD
- EXPOSE
- WORKDIR
- ADD
- COPY
- LABEL
- MAINTAINER
- ENTRYPOINT

Orchestration

Container Orchestration

- Container orchestration is all about managing the lifecycles of containers, especially in large, dynamic environments
- Software teams use container orchestration to control and automate many tasks
  - Provisioning and deployment of containers
  - Redundancy and availability of containers
  - Scaling up or removing containers to spread application load evenly across host infrastructure
  - Movement of containers from one host to another if there is a shortage of resources in a host, or if a host dies
  - Allocation of resources between containers
  - External exposure of services running in a container with the outside world
  - Load balancing of service discovery between containers
  - Health monitoring of containers and hosts
  - Configuration of an application in relation to the containers running it
- Orchestration Tools
  - Docker Swarm
  - Kubernetes
  - Mesos
  - Marathon

Docker Swarm

- Docker Swarm is a container orchestration engine
- It takes multiple Docker Engines running on different hosts and lets you use them together
- The usage is simple: declare your applications as stacks of services, and let Docker handle the rest
- It is secure by default
- It is built using Swarmkit

What is a swarm?

- A swarm consists of multiple Docker hosts which run in **swarm mode**
- A given Docker host can be a manager, a worker, or perform both roles
- When you create a service, you define its optimal state
- Docker works to maintain that desired state
  - For instance, if a worker node becomes unavailable, Docker schedules that node's tasks on other nodes
- A *task* is a running container which is part of a swarm service and managed by a swarm manager, as opposed to a standalone container
- When Docker is running in swarm mode, you can still run standalone containers on any of the Docker hosts participating in the swarm, as well as swarm services
- A key difference between standalone containers and swarm services is that only swarm managers can manage a swarm, while standalone containers can be started on any daemon

Features

- Cluster management integrated with Docker Engine
- Decentralized design
- Declarative service model
- Scaling
- Desired state reconciliation
- Multi-host networking
- Service discovery
- Load balancing
- Secure by default
- Rolling updates

Nodes

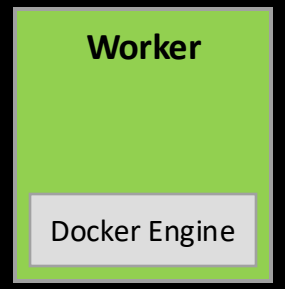
- A **node** is an instance of the Docker engine participating in the swarm
- You can run one or more nodes on a single physical computer or cloud server
- To deploy your application to a swarm, you submit a service definition to a **manager node**
- **Manager Node**
  - The manager node dispatches units of work called tasks to worker nodes
  - Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm
  - Manager nodes elect a single leader to conduct orchestration tasks
- **Worker nodes**
  - Worker nodes receive and execute tasks dispatched from manager nodes
  - An agent runs on each worker node and reports on the tasks assigned to it
  - The worker node notifies the manager node of the current state of its assigned tasks so that the manager can maintain the desired state of each worker

Services and tasks

- **Service**
  - A service is the definition of the tasks to execute on the manager or worker nodes
  - It is the central structure of the swarm system and the primary root of user interaction with the swarm
  - When you create a service, you specify which container image to use and which commands to execute inside running containers
- **Task**
  - A task carries a Docker container and the commands to run inside the container
  - It is the atomic scheduling unit of swarm
  - Manager nodes assign tasks to worker nodes according to the number of replicas set in the service scale
  - Once a task is assigned to a node, it cannot move to another node
  - It can only run on the assigned node or fail

Create a Swarm and add workers to swarm

- In our swarm we are going to use 3 nodes (one manager and two workers)
- Every node must have Docker Engine 1.12 or newer installed
- Following ports are used in node communication
  - TCP port 2377 is used for cluster management communication
  - TCP and UDP port 7946 is used for node communication
  - UDP port 4789 is used for overlay network traffic



docker swarm init --advertise-addr <ip>

docker swarm join --token <token>

Swarm Setup

- Create swarm

  > docker swarm init --advertise-addr <MANAGER-IP>

- Get current status of swarm

  > docker info

- Get the list of nodes

  > docker node ls

Swarm Setup

- Get token (on manager node)

  > docker swarm join-token worker

- Add node (on worker node)

  > docker swarm join --token <token>

Overlay Network

- It is a computer network built on top of another network
- Sits on top of the host-specific networks and allows container, connected to it, to communicate securely
- When you initialize a swarm or join a host to swarm, two networks are created
  - An overlay network called as ingress network
  - A bridge network called as docker_gwbridge
- Ingress network facilitates load balancing among services nodes
- Docker_gwbridge is a bridge network that connect overlay networks to individual docker daemon’s physical network

Service

- Definition of tasks to execute on Manager or Worker nodes
- Declarative Model for Services
- Scaling
- Desired state reconciliation
- Service discovery
- Rolling updates
- Load balancing
- Internal DNS component

Swarm Service

- Deploy a service

  > docker service create --replicas <no> --name <name> -p <ports> <image> <command>

- Get running services

  > docker service ls

- Inspect service

  > docker service inspect <service>

- Get the nodes running service

  > docker service ps <service>

Swarm Service

- Scale service

  > docker service scale <service>=<scale>

- Update service

  > docker service update --image <image> <service>

- Delete service

  > docker service rm <service>