DOCKER NOTES

What is a Container?

A lightweight, standalone, executáable software package

Why Lightweight?

- Containers share the host OS kernel
- Faster and lower resource consumption

Docker Commands

$ docker run
$ docker ps
$ docker build
$ docker exec / start

Docker Network

- Bridge
- Host
- Overlay
- Macvlan

Docker Volume

How containers manage peristent storage

What is container ?

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Containers vs Virtual Machine

1. Resource Utilization: Containers share the host operating system kernel, making them lighter and faster than VMs. VMs have a full-fledged OS and hypervisor, making them more resource-intensive.

2. Portability: Containers are designed to be portable and can run on any system with a compatible host operating system. VMs are less portable as they need a compatible hypervisor to run.

3. Security: VMs provide a higher level of security as each VM has its own operating system and can be isolated from the host and other VMs. Containers provide less isolation, as they share the host operating system.

4. Management: Managing containers is typically easier than managing VMs, as containers are designed to be lightweight and fast-moving.

Why are containers light weight ?

Containers are lightweight because they use a technology called containerization, which allows them to share the host operating system's kernel and libraries, while still providing isolation for the application and its dependencies. This results in a smaller footprint compared to traditional virtual machines, as the containers do not need to include a full operating system. Additionally, Docker containers are designed to be minimal, only including what is necessary for the application to run, further reducing their size.

Let's try to understand this with an example:

Below is the screenshot of official ubuntu base image which you can use for your container. It's just ~ 22 MB, isn't it very small ? on a contrary if you look at official ubuntu VM image it will be close to ~ 2.3 GB. So the container base image is almost 100 times less than VM image.

Files and Folders in containers base images :

/bin: contains binary executable files, such as the ls, cp, and ps commands.

/sbin: contains system binary executable files, such as the init and shutdown commands.

/etc: contains configuration files for various system services.

/lib: contains library files that are used by the binary executables.

/usr: contains user-related files and utilities, such as applications, libraries, and documentation.

/var: contains variable data, such as log files, spool files, and temporary files.

/root: is the home directory of the root user.

Files and Folders that containers use from host operating system

The host's file system: Docker containers can access the host file system using bind mounts, which allow the container to read and write files in the host file system.

Networking stack: The host's networking stack is used to provide network connectivity to the container. Docker containers can be connected to the host's network directly or through a virtual network.

System calls: The host's kernel handles system calls from the

container, which is how the container accesses the host's resources, such as CPU, memory, and I/O.

Namespaces: Docker containers use Linux namespaces to create isolated environments for the container's processes. Namespaces provide isolation for resources such as the file system, process ID, and network.

Control groups (cgroups): Docker containers use cgroups to limit and control the amount of resources, such as CPU, memory, and I/O, that a container can access.

What is Containerization?

Containerization is the process of **packaging** software code + dependencies into a **single container**.

What it builds?

A **Container Image**.

This image can be used to **instantiate (run)** multiple containers.

Why it is used?

To **standardize** environments across development, testing, production.

Scale applications easily (microservices architecture).

Reduce system conflicts.

Why Virtual Machines Over Physical Servers?

	Physical Server	Virtual Machine
Provisioning	Manual, slow	Fast, scriptable
Resource Utilization	Wasted if underused	Better utilization
Scaling	Hard	Easy
Isolation	Only via full hardware	Full OS level

VMs allowed companies to:

Run multiple "servers" on **one** physical machine.

Save costs, space, power.

Isolate different apps/teams.

Why Containers Over Virtual Machines?

	Virtual Machine	Container
Size	GBs	MBs
Boot time	Minutes	Seconds
Isolation	Full OS	Process-level
Resource Usage	Heavy	Light

Containers are preferred because:

Much faster to start.

Less RAM, CPU usage.

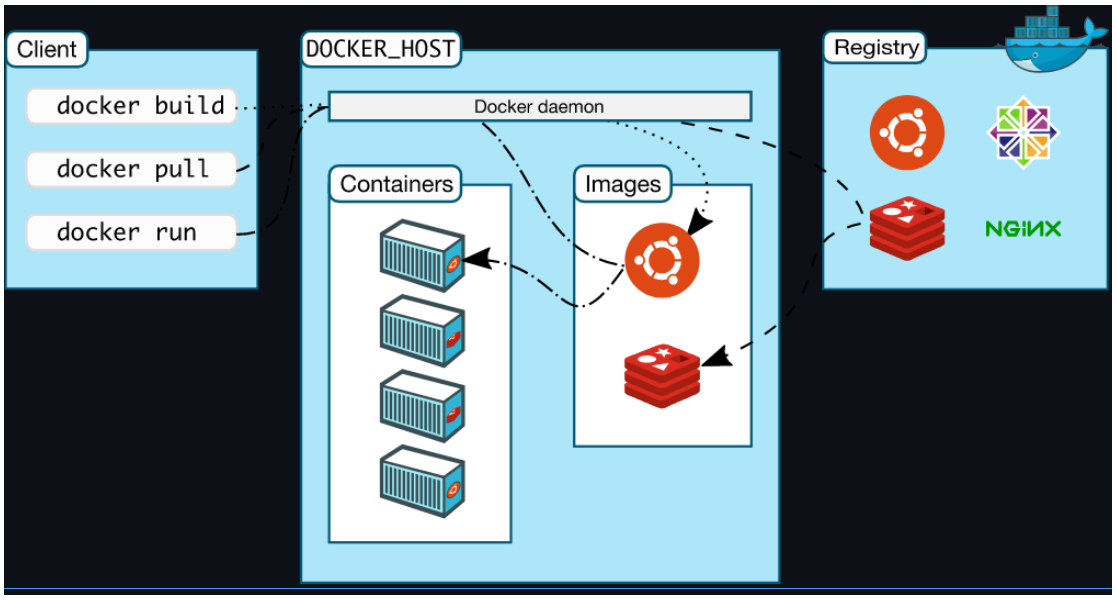
Better for **Microservices**, **Scaling**, **DevOps pipelines**.

What is Docker ?

Docker is a containerization platform that provides easy way to containerize your applications, which means, using Docker you can build container images, run the images to create containers and also push these containers to container regestries such as DockerHub, Quay.io and so on.

In simple words, you can understand as containerization is a concept or technology and Docker Implements Containerization.

Docker Architecture

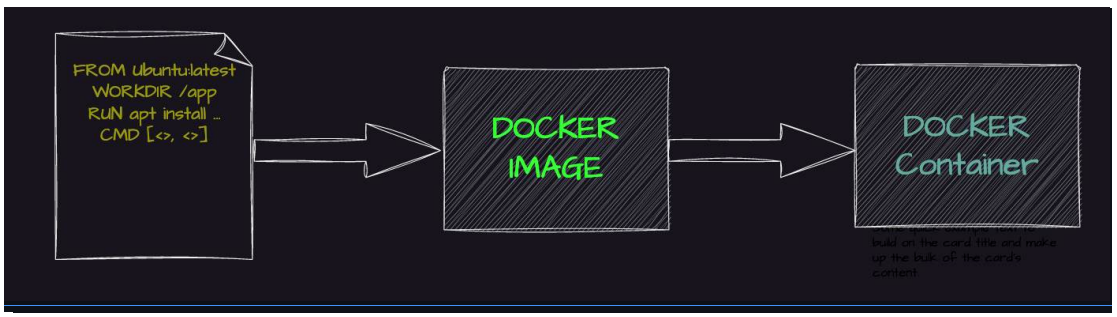


Docker LifeCycle

We can use the above Image as reference to understand the lifecycle of Docker.

There are three important things,

docker build -> builds docker images from Dockerfile
docker run -> runs container from docker images
docker push -> push the container image to public/private regestries to share the docker images.



Docker Components :

Docker daemon

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

Docker client

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

Docker Desktop

Docker Desktop is an easy-to-install application for your Mac, Windows or Linux environment that enables you to build and share containerized applications and microservices. Docker Desktop includes the Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper. For more information, see Docker Desktop.

Docker registries

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry. Docker objects

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief

overview of some of those objects.

Dockerfile

Dockerfile is a file where you provide the steps to build your Docker Image.

Images

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

Install Docker :

sudo apt update

sudo apt install docker.io -y

Start Docker and Grant Access

docker run hello-world

docker: Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Post "http://%2Fvar%2Frun%2Fdocker.sock/v1.24/containers/create": dial unix /var/run/docker.sock: connect: permission denied.

See 'docker run --help'.

This can mean two things,

Docker deamon is not running.
Your user does not have access to run docker commands.

Start Docker daemon

You use the below command to verify if the docker daemon is actually started and Active

sudo systemctl status docker

If you notice that the docker daemon is not running, you can start the daemon using the below command

sudo systemctl start docker

Grant Access to your user to run docker commands

To grant access to your user to run the docker command, you should add the user to the Docker Linux group. Docker group is create by default when docker is installed.

sudo usermod -aG docker ubuntu

In the above command ubuntu is the name of the user, you can change the username appropriately.

NOTE: : You need to logout and login back for the changes to be reflected.

Docker is Installed, up and running

Use the same command again, to verify that docker is up and running.

docker run hello-world

Docker Commands :

Basic Management & Information:

docker version: Displays the Docker client and server version information.
docker info: Provides detailed system-level information about the Docker environment.
docker ps: Lists running containers.
docker ps -a: Lists all containers, including stopped ones.
docker images: Lists all Docker images on your system.
docker search <image_name>: Searches the Docker Hub registry for a specific image.

Container Operations:

docker run <image_name>: Creates and starts a new container from an image.
docker run -d <image_name>: Runs a container in detached mode (background).
docker start <container_id>: Starts a stopped container.
docker stop <container_id>: Stops a running container.
docker kill <container_id>: Forcefully stops a container.
docker rm <container_id>: Removes a stopped container.
docker exec -it <container_id> <command>: Executes a command within a running container.
docker logs <container_id>: Fetches and displays logs from a container.
docker inspect <container_id>: Displays detailed information about a container.

Image Management:

docker pull <image_name>: Downloads an image from a registry.
docker build -t <image_name> . : Builds a Docker image from a Dockerfile.

docker tag <image_id> <repository>:<tag>: Tags an image to be pushed to a registry.
docker push <repository>:<tag>: Pushes an image to a registry.
docker rmi <image_id>: Removes an image.

Volume Management:

docker volume create <volume_name>: Creates a Docker volume.
docker volume ls: Lists all Docker volumes.
docker volume rm <volume_name>: Removes a volume.
docker volume inspect <volume_name>: Displays information about a volume.

Network Management:

docker network create <network_name>: Creates a new network.
docker network ls: Lists all networks.
docker network rm <network_name>: Removes a network.
docker network connect <network_name> <container_id>: Connects a container to a network.
docker network disconnect <network_name> <container_id>: Disconnects a container from a network.

Other Important Commands:

docker system prune: Removes unused Docker resources.
docker system df: Displays disk usage statistics for Docker.
docker stats <container_id>: Shows real-time resource consumption for a container.
docker cp <source> <destination>: Copies files or directories between the host machine and a container.
docker commit <container_id> <image_name>: Creates a new image from the changes in a container.
docker login: Logs into a Docker registry.
docker logout: Logs out of a Docker registry.
docker save -o <output_file.tar> <image_id>: Saves an image to a file.
docker load -i <input_file.tar>: Loads an image from a file.

DOCKER NETWORKING

What is Docker Networking?

When you run Docker containers, they often need to talk to each other or to the outside world (Internet) or even to your host machine.

Docker networking provides a way for containers to:

- Communicate internally (container-to-container)
- Communicate with external systems
- Maintain isolation and security
- Think of Docker networks like virtual switches or routers connecting containers to each other and/or to the host system.

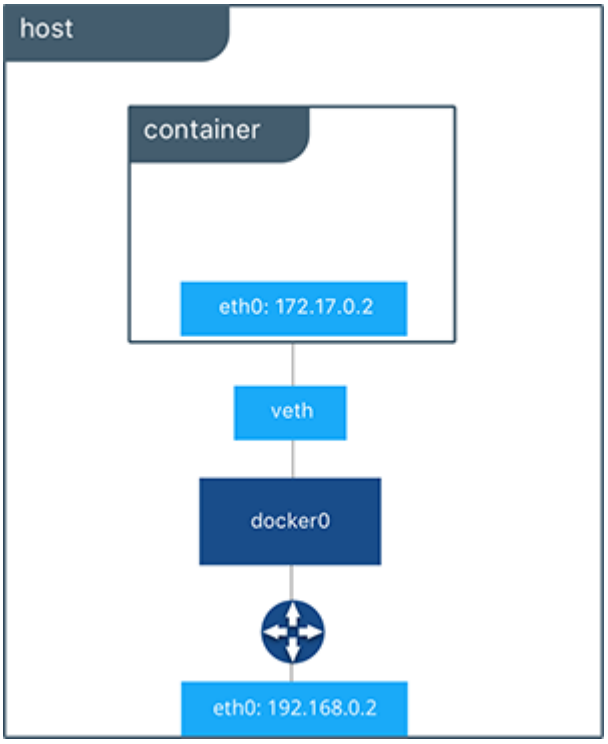
When is Docker Networking Used?

When multiple containers need to talk (e.g. a web app with a database)
When container needs Internet access (e.g. apt-get, API calls)
When exposing services from container to host/clients via ports

1. Bridge Network (Default)

What is it?

A private internal network created by Docker on the host machine. Containers can talk to each other within this network using container names as DNS.



Example:

docker network create my-bridge-net

docker run -d --name db --network my-bridge-net mongo
docker run -d --name app --network my-bridge-net mynodeapp

App container can now talk to DB using db:27017 as hostname.

When?
You want containers to talk internally on one host.
Common in local development or testing.

2. Host Network

What is it?
Container uses host's own network stack. No port mapping required.

Example:

docker run --rm --network host nginx

Nginx will be accessible on port 80 directly from the host.

When?

You want zero overhead.
Performance-sensitive apps (like gaming servers, media).
But no isolation between host & container.

3.Overlay Networking

This mode enables communication between containers across multiple Docker host machines, allowing containers to be connected to a single network even when they are running on different hosts.

4.Macvlan Networking

This mode allows a container to appear on the network as a physical host rather than as a container.

1. Bridge Networking

Advantages:

Isolation: Each container gets its own IP; traffic goes through NAT, isolating containers from the host.
Default mode: Automatically set up by Docker (no extra config).
Custom bridges allow better control (e.g., user-defined DNS, names).

Disadvantages:

Slight overhead: NAT introduces performance overhead.
Limited cross-host communication: Cannot communicate across different hosts.
Manual port mapping: Needed for external access.

Use Case:

Default choice for most standalone containers.
Ideal for local development or when network isolation is important.

2. **Host Networking**

Advantages:

Performance: No network translation layer — direct access to host network.
Low latency: Suitable for performance-sensitive applications.
No port mapping needed: Services are exposed directly on host ports.

Disadvantages:

No isolation: Containers share the host's network namespace.
Port conflicts: Containers can't use the same ports simultaneously.
Security: More risk, as containers access the host's full network stack.

Use Case:

When maximum performance is needed (e.g., media streaming, monitoring agents).
For trusted environments (e.g., internal tools on a secure host).

3. Overlay Networking (used with Docker Swarm or Kubernetes)

Advantages:

Cross-host communication: Containers can talk across multiple Docker hosts.
Built-in encryption: Secure communication between nodes.
Service discovery: Integrated with Swarm's DNS-based discovery.

Disadvantages:

Complex setup: Requires Docker Swarm or orchestrators like Kubernetes.
More overhead: Due to encapsulation and routing layers.
Debugging is harder: Networking issues can be harder to trace.

Use Case:
For distributed applications in multi-host setups.
When using Docker Swarm or Kubernetes for container orchestration.
Scalable microservices applications.

What is etho?
etho is the name of the primary Ethernet interface on a Linux system.
It represents the first physical network interface card (NIC) — like your main network adapter.
Through etho, your system connects to the LAN, gets IP addresses via DHCP/static, and accesses the internet.

What is dockero?
dockero is a virtual bridge automatically created by Docker on the host.
It acts like a software switch.
It connects your containers together and allows them to talk to each other internally.
It typically has an IP like 172.17.0.1.

Docker Netwoking Commands :

1. **docker network ls:** Lists all Docker networks, including their ID, name, driver, and scope (local or global).

2. docker network inspect <network_name | network_id>: Displays detailed information about a specific network, including its configuration, connected containers, and IPAM settings.

3. docker network create [OPTIONS] <network_name>: Creates a new network. Options can include specifying the driver, IPAM settings, and other configuration details.

4. docker network connect [OPTIONS] <network_name> <container_name | container_id>: Connects a container to a network.

5. docker network disconnect [OPTIONS] <network_name> <container_name | container_id>: Disconnects a container from a network.

6. docker network rm <network_name | network_id>: Removes one or more networks.

7. docker network prune: Removes all unused networks. This command is useful for cleaning up your environment.

DOCKER VOLUME

Problem Statement:

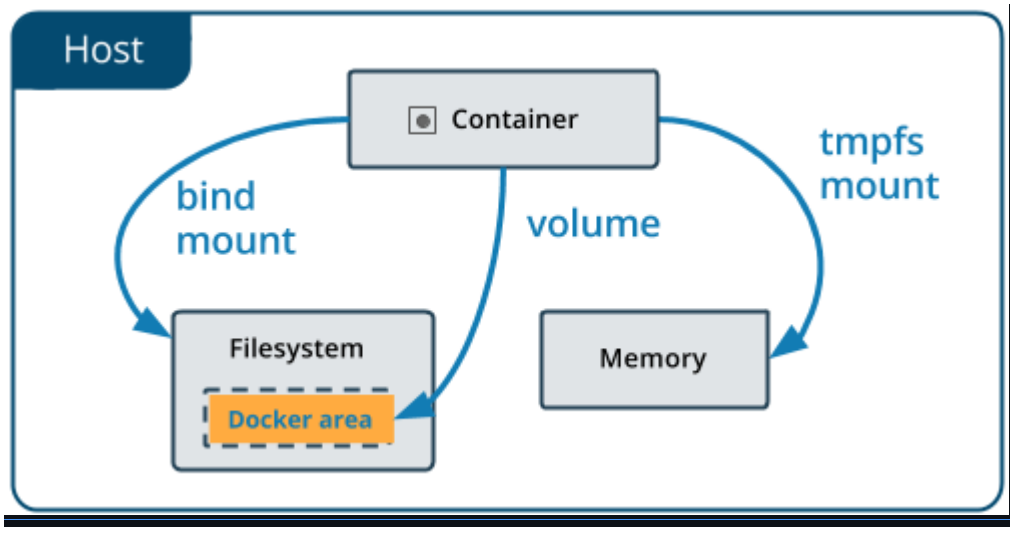
It is a very common requirement to persist the data in a Docker container beyond the lifetime of the container. However, the file system
of a Docker container is deleted/removed when the container dies.

Solution

There are 2 different ways how docker solves this problem.
1. Volumes
2. Bind Directory on a host as a Mount

Volumes

Volumes aims to solve the same problem by providing a way to store data on the host file system, separate from the container's file system, so that the data can persist even if the container is deleted and recreated.



Volumes can be created and managed using the docker volume command. You can create a new volume using the following

command:

docker volume create <volume_name>

Once a volume is created, you can mount it to a container using the -v or --mount option when running a docker run command.

For example:

docker run -it -v <volume_name>:/data <image_name> /bin/bash

This command will mount the volume <volume_name> to the /data directory in the container. Any data written to the /data directory inside the container will be persisted in the volume on the host file system.

Bind Directory on a host as a Mount

Bind mounts also aims to solve the same problem but in a complete different way.

Using this way, user can mount a directory from the host file system into a container. Bind mounts have the same behavior as volumes, but are specified using a host path instead of a volume name.

For example,

docker run -it -v <host_path>:<container_path> <image_name> /bin/bash

Key Differences between Volumes and Bind Directory on a host as a Mount

Volumes are managed, created, mounted and deleted using the Docker API. However, Volumes are more flexible than bind mounts, as they can be managed and backed up separately from the host file system, and can be moved between containers and hosts.

In a nutshell, Bind Directory on a host as a Mount are appropriate for simple use cases where you need to mount a directory from the host file system into
a container, while volumes are better suited for more complex use cases where you need more control over the data being persisted in the container.