Monolithic:

If all these services are included in one server then it will be called monolithic architecture It is tight coupling i.e. the services highly dependent on each other

Drawback:

If one service is down means we have to shutdown entire application to solve that service. So, user is facing problems because it is tightly coupled

Microservice:

If every service has its own individual servers then it is called microservices Every microservice architecture has its own database for each service
Take same above example. For every service if we keep 1-database, 1-server it is microservice It is loose coupling

Drawback:

It is too cost, because we have to maintain so many servers and database. So, maintenance is high
When compared microservice to monolithic. Microservice is good to use. Because, if one service is not working. So, we can work on it without shutdown the application. That's the reason microservices is good and preferable

Why Docker:

Let us assume that we are developing an application, and every application has Frontend, Backend and database

To overcome the above monolithic architecture we're using "Docker"

So while creating the application we need to install the dependencies to run the code
So, I installed Java11, reactJS and MongoDB to run the code. After sometime, I need another versions of java, react and MongoDB for my application to run the code
So, it's really a hectic situation to maintain multiple versions of same tool in our system

To overcome this problem we will use "Virtualization"

Virtualization:

It is used to create a virtual machines inside on our machine. In that virtual machines we can hosts guest OS in our machine
By using this guest OS we can run multiple application on same machine

Virtualization Architecture

Here, Host OS means our windows machine. Guest OS means virtual machine
Hypervisor is also known as Virtual machine monitor (VMM). It is a component/ software and it is used to create the virtual machines

Drawback:

- It is old method
- If we use multiple guest OS (or) Virtual machines then the system performance is low
- To overcome this virtualization, we are using "Containerization" ie., called Docker

Containerization:

It is used to pack the application along with it's dependencies to run the application. This process is called containerization

Container:

It's the runtime of the application which is created through docker image
Container is nothing but it is a virtual machine, which doesn't have any OS With the help of images container will run
Docker is a tool. It is used to create the containers

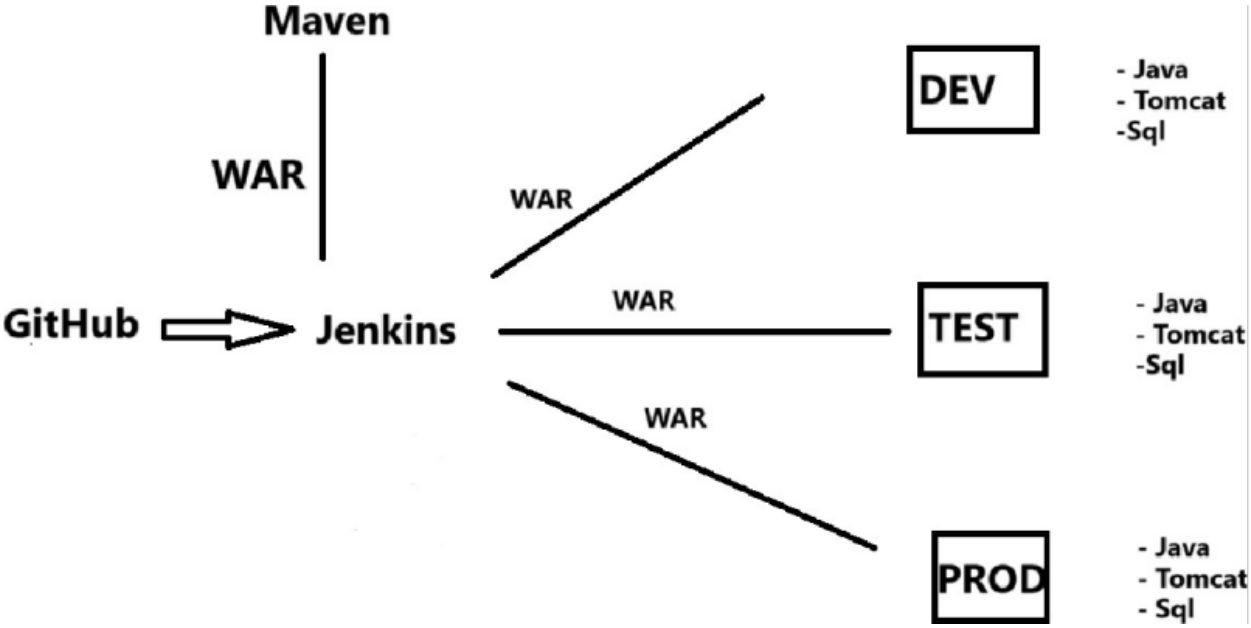
Container Architecture

It is similar to virtualization architecture, instead od hypervisor we are having Docker Engine Through Docker Engine we're creating the containers
Inside the container we're having the application Docker Engine - The software that hosts the container In one container we can keep only image

Docker Image:

A Docker image is a file used to execute code in a Docker container.
Docker images act as a set of instructions to build a Docker container, like a template. Docker images also act as the starting point when using Docker.
(or)
It is a template that contains applications, bin/libs, configs, etc., packaged together

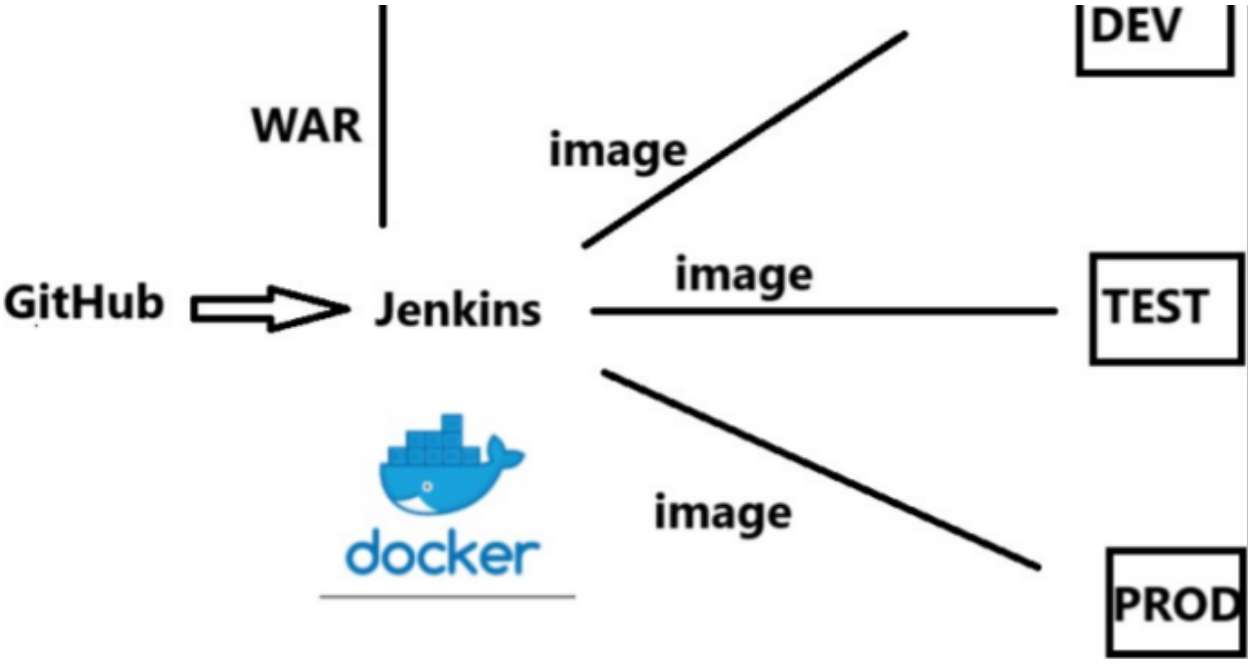
Before Docker:



First, get the code from the GitHub and integrate with Jenkins Integrate maven with Jenkins. So, we get War file
So, that war file we have to deploy in different environments
So, if you want to deploy war file/application we have to install the dependencies
This is the process when docker is not there

After Docker:

First, get the code from the GitHub and integrate with Jenkins Integrate maven with Jenkins. So, we get War file
Here, we're not going to install dependencies on any server. Because, we're following containerization rule



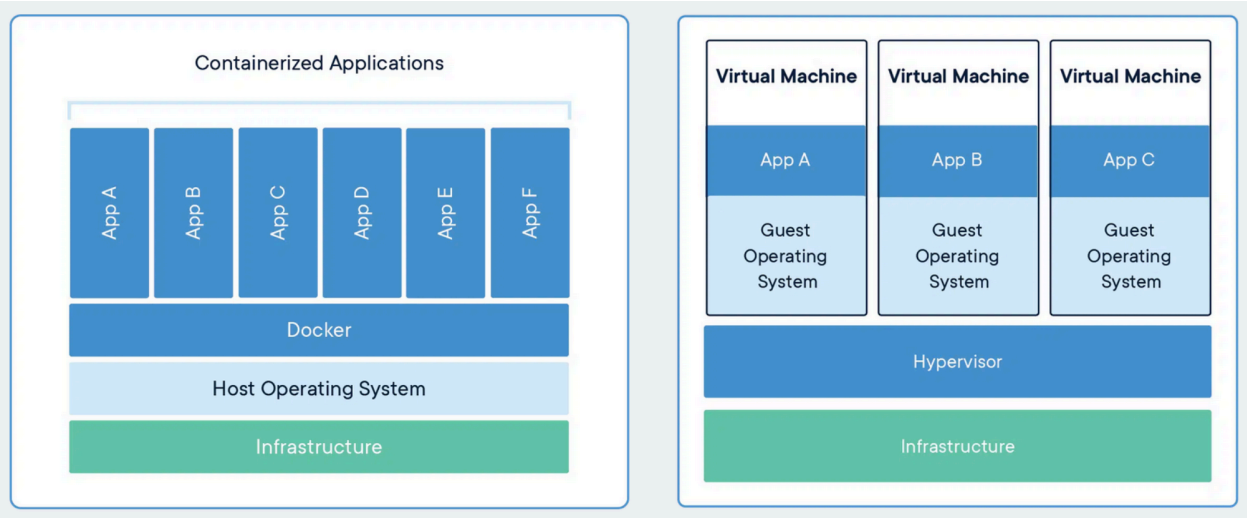
So, here we're creating image i.e. image is the combination of application and dependencies image = war + Java+Tomcat+MySql
Now, in this image application and dependencies present. So, overall this process is called containerization
So, whenever if you want to run your application.
Run that image in the particular environment. No need to install again dependencies. Because these are already present in image
So, after run the image. In that server that applications and dependencies installed
When we run an image, container will gets created. Inside the container we're having application This images are already prebuilted by docker
Container is independent in AMI i.e. If we launch AMI in ubuntu (or) CentOS, any other OS. this container will work
So, overall after docker, In any environment no need to install dependencies. We can just run the images in that particular environment. If container is created means application created

What is a 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 container is a bundle of Application, Application libraries required to run your application and the minimum system dependencies.

Containers vs Virtual Machine



Containers vs Virtual Machine

Both technologies used to isolate applications and their dependencies

key differences:

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.
- Allows them to share the host operating system's kernel and libraries, while still providing isolation for the application and its dependencies.
- 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.

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.

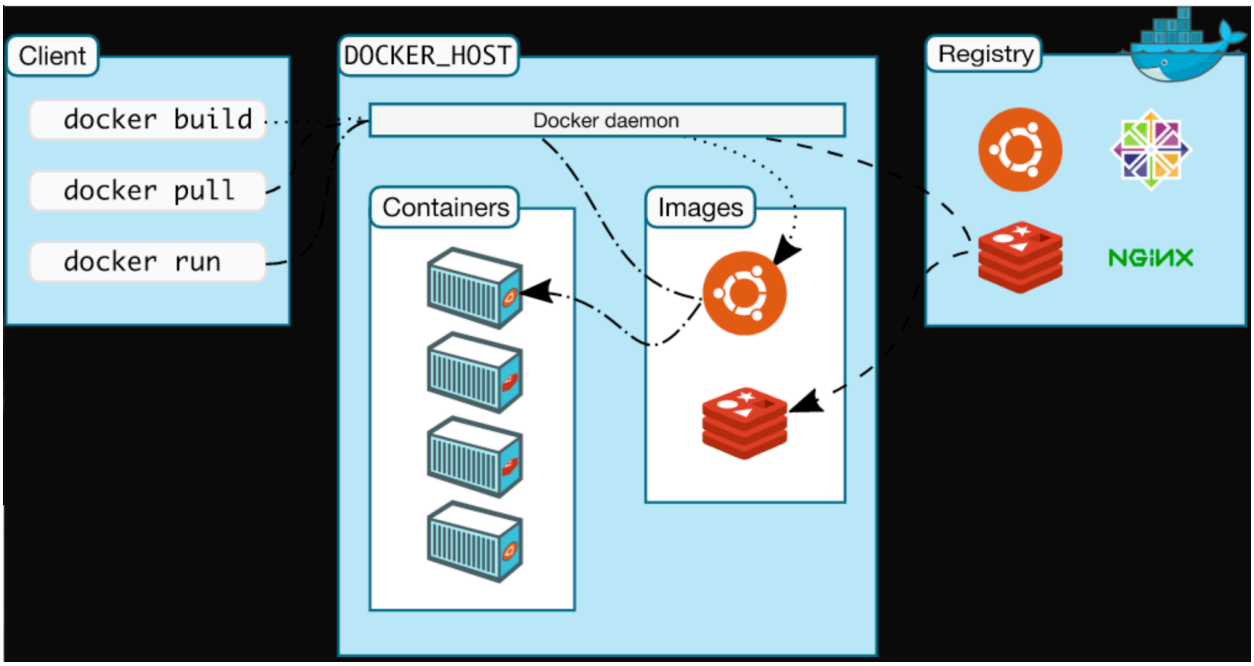
Files and Folders that containers use from host operating system

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.

It's important to note that while a container uses resources from the host operating system, it is still isolated from the host and other containers, so changes to the container do not affect the host or other containers.

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 registries 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 Deamon is brain of Docker. If Docker Deamon is killed, stops working

Docker LifeCycle

There are three important things,

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

Docker daemon

- The dockerd listens for Docker API requests
- 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 primary way that many Docker users interact with Docker.
- When you use commands such as docker run, the client sends these commands to dockerd.
- 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
- You can even run your own private registry.

docker pull or docker run commands
the required images are pulled from your configured registry.

docker push command
your image is pushed to your configured registry.

Docker hub is the default registry 2 Types of registry

Cloud based registry :

when you want to store your images in the cloud like

- docker hub
- GCR - Google Container Registry
- Amazon ECR - Elastic Container Registry

Local registry :

Here, we are storing the images in local like

- Nexus
- JFrog
- DTR - Docker trusted registry

So, here cloud based registry is preferrable. Because in docker hub we just do the account creation and store the images. But in local registry like nexus, we have to take t2.medium and do the setup it is little bit complex.

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.

Instruction	Description
FROM	Create a new build stage from a base image.
RUN	Execute build commands.
MAINTAINER	Specify the author of an image.
COPY	Copy files and directories.
ADD	Add local or remote files and directories.
EXPOSE	Describe which ports your application is listening on.
WORKDIR	Change working directory.
CMD	Specify default commands.
ENTRYPOINTS	Specify default executable.
ENV	Set environment variables.
VOLUME	Create volume mounts.
USER	Set user and group ID.
LABEL	Add metadata to an image.
ONBUILD	Specify instructions for when the image is used in a build.
SHELL	Set the default shell of an image.
ARG	Use build-time variables.
HEALTHCHECK	Check a container's health on startup.
STOPSIGNAL	Specify the system call signal for exiting a container.

Images

- An image is a read-only template with instructions for creating a Docker container.
- Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt.
- This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

INSTALL DOCKER

A very detailed instructions to install Docker are provide in the below link

https://docs.docker.com/get-docker/

For Demo, You can create an Ubuntu EC2 Instance on AWS and run the below commands to install docker.

```
sudo apt update
sudo apt install docker.io -y
```

- Start Docker and Grant Access
- Always ensure the docker daemon is up and running.
- A easy way to verify your Docker installation is by running the below command

```
docker run hello-world
```

If the output says:

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

Solution:

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
```

Docker is Installed, up and running

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

Clone this repository and move to example folder

```
git clone https:URL
cd  examples
```

Login to Docker [Create an account with https://hub.docker.com/]

```
docker login
```

Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one.
Username: xyz
Password:

WARNING! Your password will be stored unencrypted in /home/ubuntu/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/credentials-store
Login Succeeded

Build your first Docker Image

You need to change the username accordingly in the below command

```
docker build -t docker_hub_userid/my-first-docker-image:latest .
```

Verify Docker Image is created

```
docker images
```

Run your First Docker Container

```
docker run -it docker_hub_userid/my-first-docker-image
```

Push the Image to DockerHub and share it with the world

```
docker push docker_hub_userid/my-first-docker-image
```

Docker Networking

- Networking allows containers to communicate with each other and with the host system.
- Containers run isolated from the host system and need a way to communicate with each other and with the host system.
- By default, Docker provides two network drivers for you, the bridge and the overlay drivers.

```
docker network ls
```

```
NETWORK ID          NAME                DRIVER
xxxxxxxxxxxx        none                null
xxxxxxxxxxxx        host                host
xxxxxxxxxxxx        bridge              bridge
```

Bridge Networking

- The default network mode in Docker.
- It creates a private network between the host and containers, allowing containers to communicate with each other and with the host system.

- If you want to secure your containers and isolate them from the default bridge network you can also create your own bridge network.

```
docker network create -d bridge my_bridge
```

Now, if you list the docker networks, you will see a new network.
docker network ls

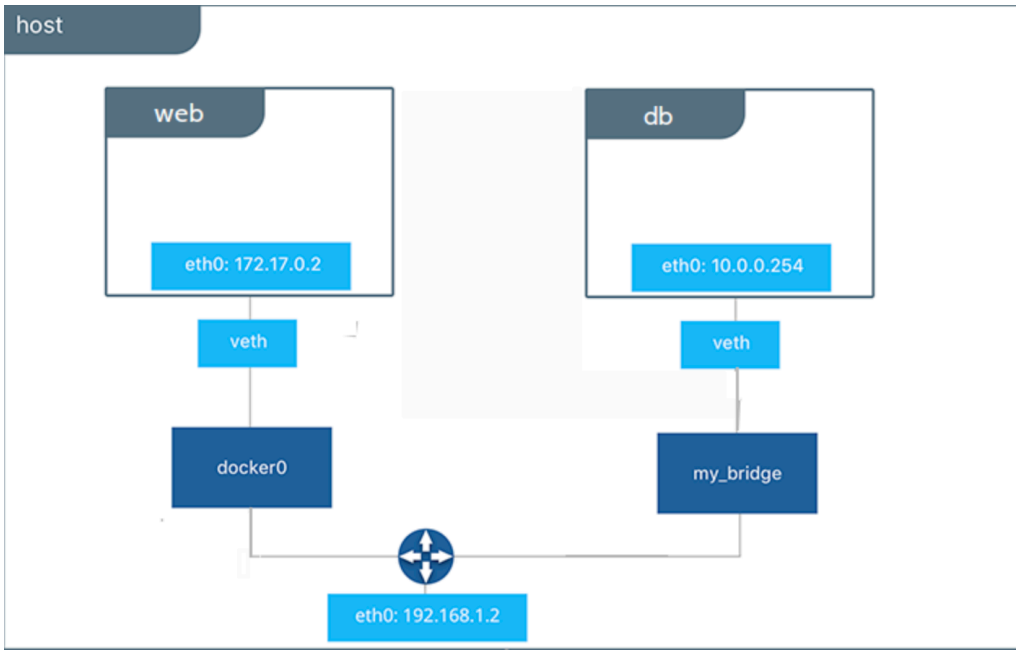
```
NETWORK ID          NAME                DRIVER
xxxxxxxxxxxx        bridge              bridge
xxxxxxxxxxxx        my_bridge           bridge
xxxxxxxxxxxx        none                null
xxxxxxxxxxxx        host                host
```

This new network can be attached to the containers, when you run these containers.

```
docker run -d --net=my_bridge --name db training/postgres
```

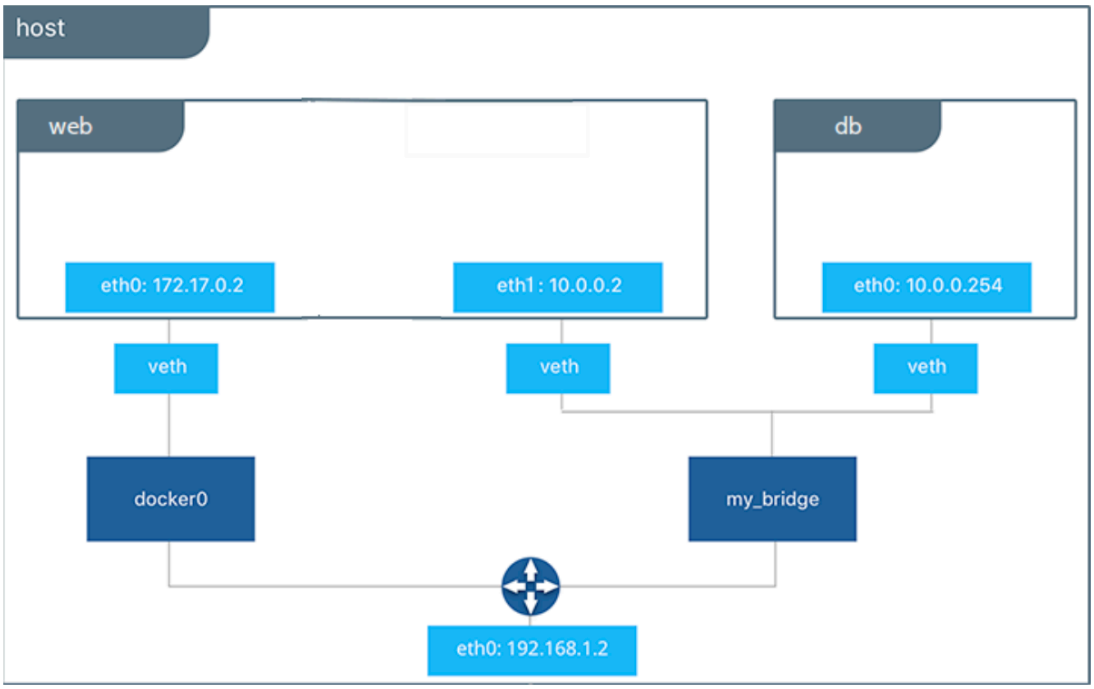
This way, you can run multiple containers on a single host platform where one container is attached to the default network and
the other is attached to the my_bridge network.

These containers are completely isolated with their private networks and cannot talk to each other.



However, you can at any point of time, attach the first container to my_bridge network and enable communication

```
docker network connect my_bridge web
```



Host Networking

This mode allows containers to share the host system's network stack, providing direct access to the host system's network.

To attach a host network to a Docker container, you can use the --network="host" option when running a docker run command. When you use this option, the container has access to the host's network stack, and shares the host's network namespace. This means that the container will use the same IP address and network configuration as the host.

Here's an example of how to run a Docker container with the host network:

```
docker run --network="host" <image_name> <command>
```

when you use the host network, the container is less isolated from the host system, and has access to all of the host's network resources. This can be a security risk, so use the host network with caution.

Additionally, not all Docker image and command combinations are compatible with the host network, so it's important to check the image documentation or run the image with the --network="bridge" option (the default network mode) first to see if there are any compatibility issues.

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.

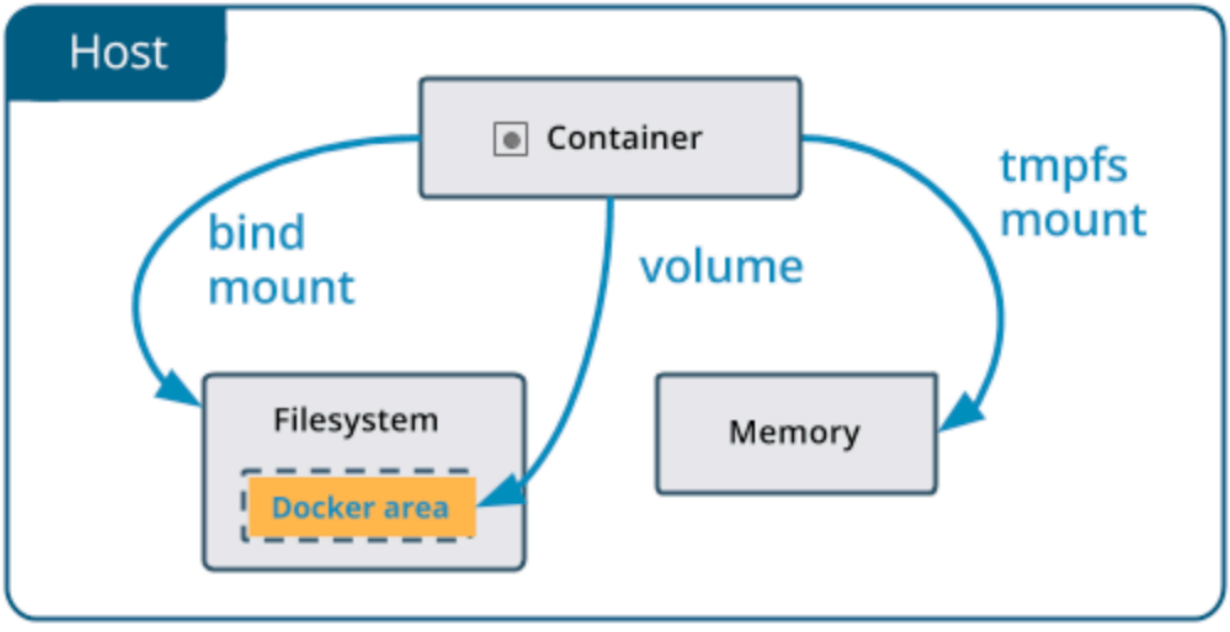
Macvlan Networking : This mode allows a container to appear on the network as a physical host rather than as a container.

Docker Volumes

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
```

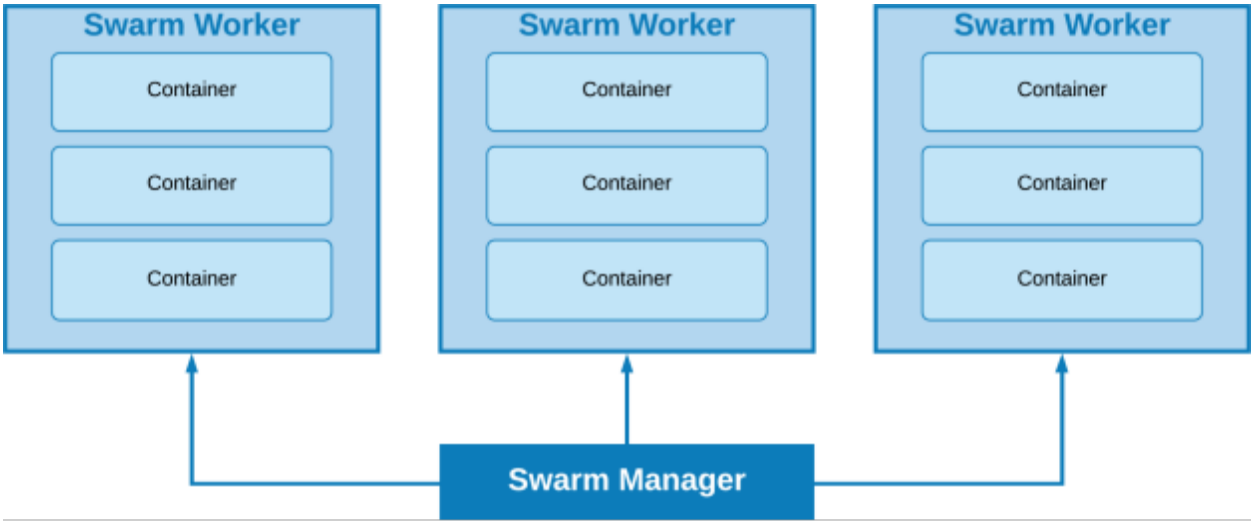
Key Differences between Volumes and Bind Directory on a host as a Mount

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.

DOCKER SWARM

Docker Swarm is an Orchestration service (or) group of service It is similar to master-slave concept
Within the docker that allows us to manage and handle multiple containers at the same time Docker Swarm is a group of servers that runs the docker application
i.e. for running the docker application, in docker swarm we're creating group of servers
We used to manage the multiple containers on multiple servers. This can be implemented by the "Cluster"
The activities of the cluster are controlled by a "Swarm Manager" and machines that have joined the cluster is called "Swarm Worker"
Here, it is the example of master and slave



Docker Engine helps to create Docker Swarm
i.e. if you want to implement Docker Swarm. In that system we have to install docker for 2 servers. i.e. Swarm Manager & Swarm Worker

In the cluster we are having 2 nodes

Worker nodes
Manager nodes

The worker nodes are connected to the manager nodes
So, any scaling i.e. containers increase (or) updates needs to be done means first, it will go to the manager node

From the manager node, all the things will go to the worker node Manager nodes are used to divide the work among the worker nodes
Each worker node will work on an individual service for better performance
i.e. 1 - worker node, 1 - service

COMPONENTS in Docker Swarm

SERVICE
It represents a part of the feature of an application
TASK
A Single part of work (or) Work that we are doing
MANAGER
This manages/distributes the work among the different nodes
WORKER
which works for a specific purpose of the service

DOCKER - COMPOSE

In docker swarm, we created 1 container/service in multiple servers using master & Slave (or) Manager & Worker concept
But In docker compose, we deployed multiple containers in single server It is complete opposite to docker swarm
Here, multiple containers means full application. i.e. frontend, backend, database containers present
So, right now these 3 containers will present in single container. For that, we are using Docker Compose
Here, Manually we can do. But here, we're doing through "compose file"
So, here suppose we have 3 apps. For these, if we have to create container means first, write the docker file. then after build and get the image. After run the image you got a container. These is the manual process for all 3 apps
But, here in Docker compose, we took docker files. For all docker files we write one compose file In this compose file, we are having containers related configuration is present.
i.e. container name, port, volume, networks
So, like this if we write the container related requirements in compose file and if we execute the file means. In compose files, whatever the containers we're having that all will created
Here, Automate happened. i.e. image build & container creation at a time happened

So, overall, In real time if developers write the code means, we are writing the docker files for that. For that docker file, Here we are writing the Compose file and will execute that

Def:

Docker compose is a tool used to build, run and ship the multiple containers for application It is used to create multiple containers in a single host/server
It used YAML file to manage multi containers as a single service
i.e. In docker compose file we are writing the container configurations, that should be written in YAML format and the compose file extends with ".yml"

The compose file provides a way to document and configure all of the applications service dependencies
like - databases, queues, caches, web service API's, etc.., In one directory, we can write only one docker-compose file

	Docker Commands	
Step	**Command**	**Explanation**
View Docker Version	docker --version	Displays the installed Docker version.
View Docker Info	docker info	Shows system-wide information, including containers, images, and Docker configuration.
Working with Docker Images		
Step	**Command**	**Explanation**
List Images	docker images	Lists all images on the host machine.
Pull Image	docker pull <image_name>	Downloads an image from the Docker registry.
Build Image from Dockerfile	docker build -t <image_name> .	Builds an image from a Dockerfile in the current directory (.); tags it with a name (-t <image_name>).
Remove Image	docker rmi <image_id>	Removes an image using its image ID.
Force Remove Image	docker rmi -f <image_id>	Forcibly removes an image, even if there are dependencies.
Remove All Images	docker rmi $(docker images -q)	Removes all images on the host machine.
Remove Dangling Images	docker image prune	Removes dangling (unreferenced) images.
Remove All Unused Images	docker image prune -a	Removes all unused images (both tagged and untagged).

Working with Docker Containers		
Step	**Command**	**Explanation**
List Running Containers	docker ps	Lists all currently running containers.
List All Containers	docker ps -a	Lists all containers, including stopped ones.
Run a Container	docker run <image_name>	Runs a container from a specified image.
Run in Detached Mode	docker run -d <image_name>	Runs a container in detached mode (in the background).
Port Mapping	docker run -p <host_port>:<container_port> <image_name>	Maps a container port to a host port.
Name a Container	docker run --name <name> <image_name>	Runs a container with a specified name.
View Container Logs	docker logs <container_id>	Shows logs of a specific container.
Start a Stopped Container	docker start <container_id>	Starts a stopped container.
Stop a Running Container	docker stop <container_id>	Stops a specific running container.
Restart a Container	docker restart <container_id>	Restarts a running container.
Stop All Running Containers	docker stop $(docker ps -q)	Stops all currently running containers.
Remove a Stopped Container	docker rm <container_id>	Removes a specific stopped container.
Remove All Stopped Containers	docker rm $(docker ps -a -q)	Removes all stopped containers.
Execute Command in Container	docker exec -it <container_id> <command>	Runs a command in a running container. -it allows for interactive mode.
Open Shell in Container	docker exec -it <container_id> /bin/bash	Opens an interactive shell within a running container.
Copy Files from Container	docker cp <container_id>:/path/to/file /host/destination	Copies files from container to host.
Copy Files to Container	docker cp /host/source <container_id>:/path/to/destination	Copies files from host to container.
Docker Volumes (Data Persistence)		
Step	**Command**	**Explanation**
List Volumes	docker volume ls	Lists all Docker volumes.
Create Volume	docker volume create <volume_name>	Creates a new named volume.
Remove Volume	docker volume rm <volume_name>	Removes a specified volume.
Remove All Unused Volumes	docker volume prune	Removes all unused volumes.
Attach Volume to Container	docker run -v <volume_name>:/container/path <image_name>	Attaches a volume to a container at a specified path.

Docker Networks		
Step	**Command**	**Explanation**
List Networks	docker network ls	Lists all Docker networks.
Create Network	docker network create <network_name>	Creates a new network with a specified name.
Connect Container to Network	docker network connect <network_name> <container_id>	Connects a container to a specified network.
Disconnect Container from Network	docker network disconnect <network_name> <container_id>	Disconnects a container from a network.
Remove Network	docker network rm <network_name>	Removes a specified network.
Remove All Unused Networks	docker network prune	Removes all unused networks.
Docker Cleanup		
Step	**Command**	**Explanation**
Remove Dangling Images	docker image prune	Removes all dangling images.
Remove All Unused Images	docker image prune -a	Removes all unused images, both tagged and untagged.
Remove All Stopped Containers	docker container prune	Removes all stopped containers.
Remove All Unused Volumes	docker volume prune	Removes all unused volumes.
Remove All Unused Networks	docker network prune	Removes all unused networks.
Remove All Unused Resources	docker system prune	Cleans up all unused data, including images, containers, volumes, and networks.