docker

MAIL: Rnraju4u@gmail.com **NUMBER:** +91 9848363431 **YOUTUBE:** SYSGEEKS

DOCKER

- Docker is an open platform tool for developing, shipping, & running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.
- Docker is a bit like a virtual machine. But unlike a virtual machine, rather than creating a whole virtual operating system.
- Docker provides a way to run applications securely isolated in a container, packaged with all its dependencies and libraries.
- It is designed to benefit both developers and system administrators, making it a part of many DevOps tool chains.

VIRTUALIZATION (VT)

- **VT** is a software technology that makes computing environments independent of physical infrastructure.
- It is a process of creating virtual applications, virtual servers, storage & n/w.
- It is the single most effective way to reduce IT expenses while boosting efficiency & agility for all size businesses.

VIRTUALIZATION BENEFITS:

- Reduced capital and operating costs.
- Minimized or eliminated downtime.
- Increased IT productivity, efficiency, agility and responsiveness.
- Faster provisioning of applications and resources.
- Greater business continuity and disaster recovery.
- Simplified data center management.
- Availability of a true Software-Defined Data Center.

VIRTUALIZATION TYPES:

- Server Virtualization
- Network Virtualization
- Desktop Virtualization
- Para-virtualization
- Hardware-level virtualization

HYPERVISORS

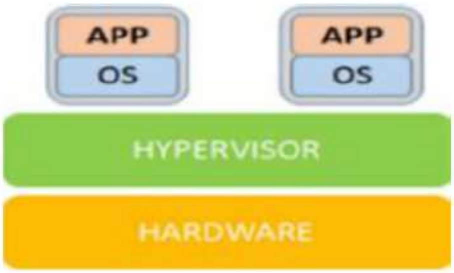
- A hypervisor is a hardware virtualization technique that **allows multiple guest operating systems** to run on a single host system at the same time.
- Guest OS shares hardware of the host computer, have its own processor, memory and other h/w resources.
- A hypervisor is also known as a **Virtual Machine Manager (VMM).**

HYPERVISIOR TYPES:

TYPE1:

- **Type1** is on bare metal. VM resources are scheduled directly to the hardware by the hypervisor.

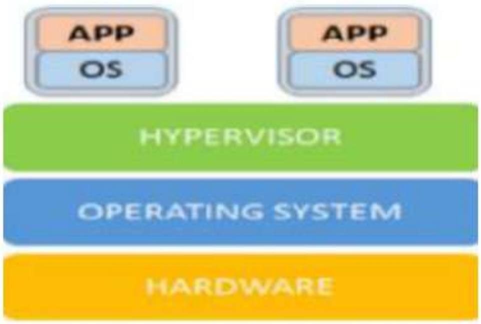
  Eg: VMware ESXI, Citrix XenServer, Microsoft Hyper-V, Linux KVM.



TYPE2:

- **Type2** is hosted. VM resources are scheduled against a host operating system, which is then executed against the hardware.

  Eg: VMware workstation and Oracle virtual box.



VIRTUAL MACHINE (VM)

- **A VM** is a virtual environment that functions as a virtual computer system with its own CPU, memory, network, and storage, created on a physical.
- Most enterprises use a combination of physical and virtual infrastructure to balance the corresponding advantages and disadvantages.

KEY PROPERTIES OF VIRTUAL MACHINE:

PARTITIONING:

- Run multiple operating systems on one physical machine.
- Divide system resources between virtual machines.

ISOLATION:

- Provide fault and security isolation at the hardware level.
- Preserve performance with advanced resource controls.

ENCAPSULATION:

- Save the entire state of a virtual machine to files.
- Move and copy virtual machines as easily as moving and copying files.

HARDWARE INDEPENDENCE:

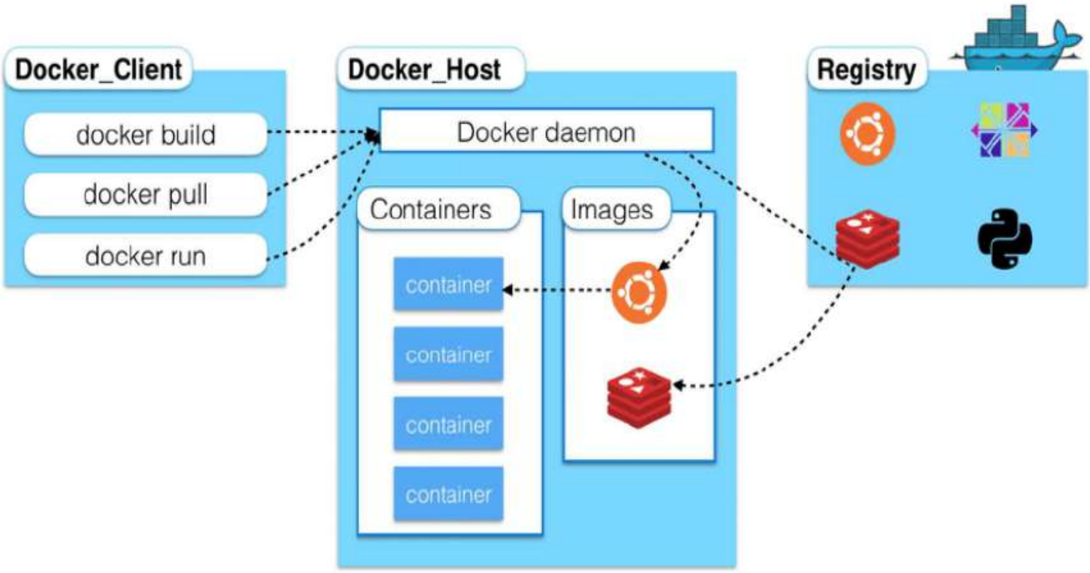
- Provision or migrate any virtual machine to any physical server.

VIRTUALIZATION vs. CLOUD COMPUTING

- Virtualization is software that makes computing environments independent of physical infrastructure.
- Cloud computing is a service that delivers shared computing resources (software and/or data) on demand via the Internet.
- As complementary solutions, organizations can begin by virtualizing their servers and then moving to cloud computing for even greater agility and self-service.

DOCKER ARCHITECTURE

- Docker uses a **client-server** architecture. The Docker ***client*** talks to the **Docker *daemon***, which does the heavy lifting of building, running, and distributing your Docker containers.
- The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote **Docker daemon**.



DOCKER DAEMON:

- The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes.

DOCKER CLIENT:

- The Docker client (docker) is the primary way that many Docker users interact with Docker.

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.

DOCKER OBJECTS

IMAGES:

- An *image* is a read-only template with instructions for creating a Docker container.
- A container is launched by running an image. An **image** is an executable package that includes everything needed to run an application–the code, a runtime, libraries, environment variables, and configuration files.

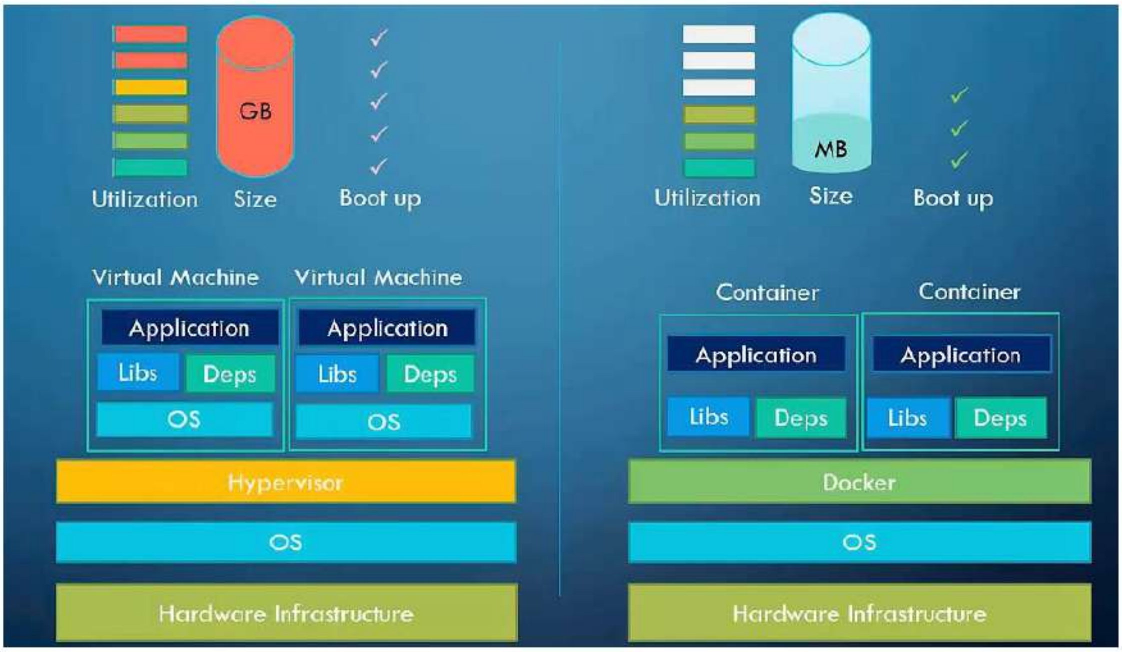
CONTAINERS:

- A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI.
- Docker Containers are:
  - **Flexible** : The most complex applications can be containerized.
  - **Lightweight** : Containers leverage and share the host kernel.
  - **Interchangeable:** You can deploy updates and upgrades on-the-fly.
  - **Portable** : Build locally, deploy to the cloud, and run anywhere.
  - **Scalable** : Increase and automatically distribute container replicas.
  - **Stackable** : You can stack services vertically and on-the-fly.

THE UNDERLYING TECHNOLOGY

- Docker is written in the Go programming language and takes advantage of several features of the Linux kernel to deliver its functionality.
- Docker uses a technology called **namespaces** to provide the **isolated workspace** called the **container.**
- When you run a container, Docker creates a set of namespaces for that container.
- These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

VIRTUAL MACHINES VS CONTAINERS



VIRTUAL MACHINES:

- A virtual machine (VM) is a virtual environment that functions as a virtual computer system with its own CPU, memory, network interface, and storage, created on a physical. In other words, creating a computer within a computer.
- Multiple virtual machines can run simultaneously on the same physical computer.

CONTAINERS:

- A container is a running instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI.

DOCKER INSTALLATION

DOCKER ENGINE OVERVIEW:

- Docker Engine is an open-source containerization technology for building and containerizing your applications.
- Docker Engine acts as a client-server application with:
  - A server with a long-running daemon process dockerd.
  - APIs which specify interfaces that programs can use to talk to and instruct the Docker daemon.
  - A command line interface (CLI) client docker.

SUPPORTED PLATFORMS:

- Docker Desktop for Mac (macOS)
- Docker Desktop for Windows
- Linux distributions:

  Red Hat, Centos, Fedora, Debian, Ubuntu…etc.
- Cloud Platforms:

  AWS, AZURE, GCP, Digital Ocean…. etc.

INSTALL DOCKER ENGINE ON LINUX:

INSTALL ON CENTOS / RHEL 9:

- To install Docker Engine, you need a maintained version:
  - **CentOS 7 or 8**
  - **RHEL 7 or 8**
  - **Fedora 33 or 34**

INSTALL ON UBUNTU:

- To install Docker Engine, you need the **64-bit** version of one of these Ubuntu versions:
  - Ubuntu Hirsute 21.04
  - Ubuntu Groovy 20.10
  - Ubuntu Focal 20.04 (LTS)
  - Ubuntu Bionic 18.04 (LTS)

STEP 1: Setting Up Hostname

#hostname Docker

#vim /etc/hostname

Docker

#bash

STEP2: Security-Enhanced Linux is being disabled or in permissive mode.

#sed -i 's/SELINUX=.*/SELINUX=disabled/g' /etc/selinux/config

#setenforce 0

STEP 3: Update current OS

#yum update -y

STEP 4: Unistall Old Versions

```
yum remove docker \
           docker-client \
           docker-client-latest \
           docker-common \
           docker-latest \
           docker-latest-logrotate \
           docker-logrotate \
           docker-engine \
           podman \
           runc
```

STEP 5: Set up the repository

#yum install -y yum-utils

#yum-config-manager --add-repo
https://download.docker.com/linux/rhel/docker-ce.repo

STEP 6: Install Docker Engine

#yum install docker-ce -y

NOTE: Getting any error, plese change repo lines

#vim /etc/yum.repos.d/docker-ce.repo

[docker-ce-stable]

name=Docker CE Stable - $basearch

baseurl=https://download.docker.com/linux/centos/$releasever/$basearch/sta
ble

enabled=1

gpgcheck=1

gpgkey=https://download.docker.com/linux/centos/gpg

#yum install docker-ce -y

#docker --version

STEP 7: Start and Enable docker service

#systemctl start docker

#systemctl enable docker

#systemctl status docker

INSTALL DOCKER DESKTOP ON WINDOWS

- Windows 10 64-bit: Home or Pro 2004 (build 19041) or higher, or Enterprise or Education 1909 or higher.
- Enable the WSL 2 feature on Windows. (Windows Subsystem for Linux, version 2)
- The following hardware prerequisites are required to successfully run WSL 2 on Windows 10:
  - 64-bit processor with Second Level Address Translation (SLAT)
  - 4GB system RAM
  - BIOS-level hardware virtualization support must be enabled in the BIOS settings.
  - Download and install the Linux kernel update package: https://docs.microsoft.com/en-us/windows/wsl/install-win10#step-4---download-the-linux-kernel-update-package

NOTE: Please follow the bellow link for docker installation. https://docs.docker.com/engine/install/

DOCKER IMAGES & CONTAINERS

DOCKER IMAGES:

#docker search centos

#docker pull centos

#docker images (or) #docker image ls

Image History:

#docker image history a9d583973f65

Image Details:

#docker image inspect ubuntu

Removing dangling images:

A dangling image is an image that is not tagged and is not used by any container.

To remove dangling images:

#docker image prune

Remove Image:

#docker rmi imageid or docker images rm imageid

Removing all unused images

#docker image prune -a

MANIPULATING DOCKER IMAGES:

#docker run -i -t <imagename>:<tag> /bin/bash

Options: -i : Gives us an interactive shell into the running container

-t : Will allocate a pseudo-tty

-d : The daemon mode

Create a first Container with name Test1

#docker run -d -it --name Test1 centos

To List Running Containers

#docker ps (or) #docker container ls

To list all containers

#docker ps -a

Docker Exec: Execute a command in a running container.

Execute a command on a container:

#docker exec -d Test1 touch /opt/aws

#docker exec -d Test1 ls /opt/

Execute an interactive bash shell on the container:

#docker exec -it Test1 bash

#exit

To Stop Container (Application shutdown gracefully)

#docker stop cid

#docker start -a cid : -a attach mode

Rename Container:

#docker rename <current_container_name> <new_container_name>

Container stats:

#docker stats <container_name>

#docker stats cid

Monitor Container:

#docker top cid/name

Container Pause:

#docker container pause containerID

Container Unpause:

#docker container unpause cid

Kill one or more Containers:

#docker kill cid [no time for proper shutdown of Application]

Removing Containers:

#docker container rm cid (or) #docker rm cid

Removing all stopped containers:

#docker container prune

Docker Logs: Fetch the logs of a container

Check Docker Logs:

#docker logs cid

Logs to verify Consensually:

#docker logs -f cid

Docker System and stats:

Docker System will manage docker (Host system)

Show docker disk usage:

#docker system df

Display system-wide information

#docker system info :

Get real time events from the server:

#docker system events

Use docker events to get real-time events from the server:

Open two terminals, on first one run:

#docker system events

On second terminal launch or stop any container

#docker stop CID / #docker run -d -it --name Test2 centos

Now to check the events on first terminal

Delete all stopped and unused containers:

#docker system prune

Delete all images and stopped containers:

#docker system prune -a

Docker stats: The docker stats command returns a live data stream for running containers.

#docker stats CID

PORT FORWARDING:

#docker run -d -it -p <host_port>:<container_port> <image>:<tag>

#docker pull nginx

#docker run -d -it --name mynginx -p 8090:80 nginx

#docker ps

To check the running port of the container

Go To Web Browser

http://IP-Address:8090/

To check Container Logs

#docker logs cid

Runtime options with Memory & CPUs:

By default, a container has no resource constraints and can use as much of a given resource as the host's kernel scheduler allows.

Docker provides ways to control how much memory, or CPU a container can use, setting runtime configuration flags of the docker run command.

Container memory limit 512m max

#docker run -d -it –name sample1 -p 800:80 -m 512m nginx

#docker status sample1

cpulimit 50000(50%) total cpu size 100thousend

#docker run -d -it --name sample2 -p 900:80 --cpu-quota=50000 nginx

#docker status sample2

To run a container from the centos image, assigning 1 GB of RAM for the container to use and reserving 1 GB of RAM for swap memory, type:

#docker run -d -it --name sample3 --memory="1g" --memory-swap="2g" centos

DYNAMICALLY UPDATES CONTAINER CONFIGURATION:

Docker Update: Update configuration of one or more containers

Update a container's cpu-shares

#docker update --cpu-shares 512 cid

Update a container with cpu-shares and memory

#docker update --cpu-shares 512 -m 300M cid

If you started a container with this command:

#docker run -dit --name test4 --kernel-memory 50M centos bash

You can update kernel memory while the container is running:

#docker update --kernel-memory 80M test4

MANAGE DOCKER FILE

DOCKER FILE

- Dockerfile is the core file that contains instructions to be performed when an image is built.
- It is a text document that contains all the commands a user could call on the CLI to assemble an image.
- The docker build command builds an image from a Dockerfile and a context.

SYNTAX: Instruction arguments

NOTE:

- In Dockerfile **"#"** is a single line comment.
- The escape character is used both to escape characters in a line, and to escape a newline. This allows a Dockerfile instruction to span multiple lines.
- Note that regardless of whether the escape parser directive is included in a Dockerfile, escaping is not performed in a RUN command, except at the end of a line.

BUILDING IMAGES USING DOCKERFILE:

- The docker build command builds Docker images from a Dockerfile and a "context".
- A build's context is the set of files located in the specified PATH or URL.
- The URL parameter can refer to three kinds of resources: Git repositories, pre-packaged tarball contexts and plain text files.

Build with PATH:

#docker build .

Build with URL:

#docker build github.com/creack/docker-firefox

Build with -:

#docker build - < Dockerfile

Build with Tag and File:

#docker build -f <path_to_Dockerfile> -t <REPOSITORY>:<TAG>

DOCKER FILE INSTRUCTIONS & ARGUMENTS

FROM:

- A docker file must start with a **FROM** instruction.
- The **FROM** instruction initializes a new build stage and sets the Base Image for subsequent instructions.

E.g.: FROM centos:latest

MAINTAINER:

- **MAINTAINER** is used to specify about the author who creates this new docker image for the support.

E.g.: MAINTAINER RAJU rnraju4u@gmail.com

LABEL:

- **LABEL** is used to specify metadata information to an Image, which is a **Key-Value Pair**.

E.g.: LABEL "App_Env"="Production"

RUN:

- It is used to executes any commands on top of the current image and this will create a new layer.
- It has two forms:

SHELL FORM:

E.g.: RUN yum update -y

EXECUTABLE FORM:

E.g.: RUN ["yum","update"]

CMD:

- It is used to set a command to be executed when running a container.
- There must be one CMD in a Dockerfile. If more than one CMD is listed, only the last CMD takes effect.

E.g.: CMD ping google.com

ENTRYPOINT:

- It is used to configure and run a container as an executable.

E.g.: ENTRYPOINT ping google.com

NOTE: If user specifies any arguments (commands) at the end of "docker run" command, the specified commands override the default in CMD instruction, But ENTRYPOINT instructional are not overwritten by the docker run command and ENTRYPOINT instruction will run as it is.

COPY:

- It is used to copy files, directories and Remote url files to the destination (Docker image) within the file system of the docker images.

E.g.: COPY src dest

NOTE: If the "src" argument is compressed file (tar, zip bzip2..etc), then it will copy exactly as a compressed file and will not extract.

ADD:

- It is used to copy files, directories and Remote URL files to the destination within the file system of the docker images.

E.g.: ADD src dest

NOTE: If the **"src"** argument is compressed file (tar, zip bzip2...etc), then it will Extract it automatically inside a destination in the Docker image.

WORKDIR:

- It is used to set the working directory.

  E.g.: WORKDIR /tmp

EXPOSE:

- This instruction informs Docker that the container listens on the specified network ports at runtime.
- By default, EXPOSE assumes TCP. You can also specify UDP.
- To publish the port when running the container, use the **-p** flag on docker run

  E.g.: EXPOSE 80/tcp

USER:

- The USER instruction lets you specify the username to be used when a command is run.
- It is used to set the user's name, group name, UID, GID for running subsequent commands. Else root user will be used.

  E.g.: USER webadmin

ENV:

- It is used to set environmental variables with **Key Value** set.

  E.g.: ENV username admin **(or) ENV** username=admin

ONBUILD:

- It lets you stash a set of commands that will be used when the image is used again as a base image for a container.

  E.g.: ONBUILD ADD

VOLUME:

- The VOLUME instruction is used to specify a mount point for a volume within the container.
- The volume will be created when the container is built, and it can be accessed and modified by processes running inside the container.

  E.g.: VOLUME /my_data

ARG:

- It is also used to set environment variables with Key-Value, but this variable will set only during the image build not on the container.

  E.g.: ARG tmp_ver 2.0

Understand how ARG and FROM interact:

FROM instructions support variables that are declared by any ARG instructions that occur before the first FROM.

ARG CODE_VERSION=latest

FROM base:${CODE_VERSION}

E.g.:

```
ARG VERSION=latest
FROM centos:$VERSION
ARG VERSION
RUN echo $VERSION > image_version
```

PROJECT 1: SIMPLE DOCKERFILE IMAGE BUILD

FROM centos:7

MAINTAINER RAJU rnraju4u@gmail.com

RUN yum update -y

LABEL "Env"="Prod" \

“Project”=”Airtel” \

"Version"="8.4"

COPY *.txt /opt/

PROJECT 2: BUILDING PYTHON FLASK

FROM centos:latest

MAINTAINER Raju "rnraju4u@gmail.com"

RUN yum update -y

RUN yum install -y python3 python3-pip wget

RUN pip3 install Flask

ADD hello.py /home/hello.py

WORKDIR /home

BUILDING DOCKER IMAGE FROM A FILE:

#docker build -t python3:latest .

#docker ps

#docker run -d -p 5000:5000 python:centos python3 hello.py

#docker logs cid

Hello World

PROJECT 3: BUILDING APACHE HTTP SERVER

#Download an Image file

FROM centos:laest

#Installing Apache httpd on a image

RUN yum install httpd -y

#Copy index.html file into Document root location

COPY index.html /var/www/html/

#Running Port with 80

EXPOSE 80

#To make service to start

CMD ["/usr/sbin/httpd","-D","FOREGROUND"]

Create an index.html file in the locattion

#vim index.html

WELCOME TO SYSGEEKS…!

BUILDING DOCKER IMAGE FROM A FILE:

#docker build -t webserver:httpd-2.4 .

#docker run -d -it –name webserver -p 8000:80 webserver:httpd-2.4

#docker ps

Go To Web Browser, type:

http://10.10.10.10:8000

IMAGE FROM CONTAINER CHANGES

DOCKER COMMIT: Create a new image from a container's changes

- It can be useful to commit a container's file changes or settings into a new image. This allows you to debug a container by running an interactive shell, or to export a working dataset to another server.
- The commit operation will not include any data contained in volumes mounted inside the container.
- By default, the container being committed and its processes will be paused while the image is committed.

Download and Create a New Container:

#docker pull centos

#docker images

#docker run -d -it –name Mycentos centos

#docker ps

Connect and Changes on a Container:

#docker exec -it Mycentos bash

Changing Centos Repo mirrors in a container

#cd /etc/yum.repos.d/

#sed -i 's/mirrorlist/#mirrorlist/g' /etc/yum.repos.d/CentOS-*

#sed -i 's|#baseurl=http://mirror.centos.org|baseurl=http://vault.centos.org|g' /etc/yum.repos.d/CentOS-*

#yum update -y

#exit

Create an image from Container changes:

#docker commit CID My-Centos:raju

#docker image ls

DOCKER HUB

- Docker Hub is a service provided by Docker for **finding and sharing container images** with your team.
- Link your images to the **GitHub/Bitbucket** repositories that can be built automatically based on web hooks.

PUBLIC REPOSITORIES:

- Users get access to free public repositories for storing and sharing images.
- Anyone can use **docker pull command** to download an image and run or build further images from it.

PRIVATE REPOSITORIES:

- Private repositories are just that private.
- Users can choose a **subscription plan** for private repositories.

DOCKER HUB Vs DOCKER SUBSCRIPTION

DOCKER HUB:

- Shareable image, but it can be private.
- No hassle of self-hosting.
- Free (except for a certain number of private images).

DOCKER SUBSCRIPTION:

- Integrated into your authentication services (that is, **AD/LDAP**).
- Deployed on your own infrastructure (or **cloud**).
- Commercial support.

NOTE: By default, repositories are pushed as public.

DOCKER HUB ENTERPRISE

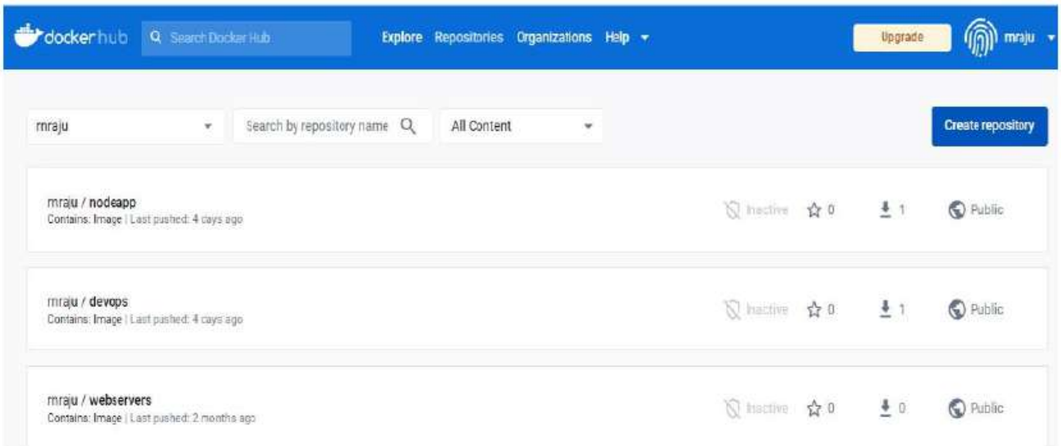
- It offers you is access to the software, access to **updates/patches/security fixes,** and support relating to issues with the software.
- The open-source Docker repository image doesn't offer these services at this level;

MANAGING DOCKER HUB:

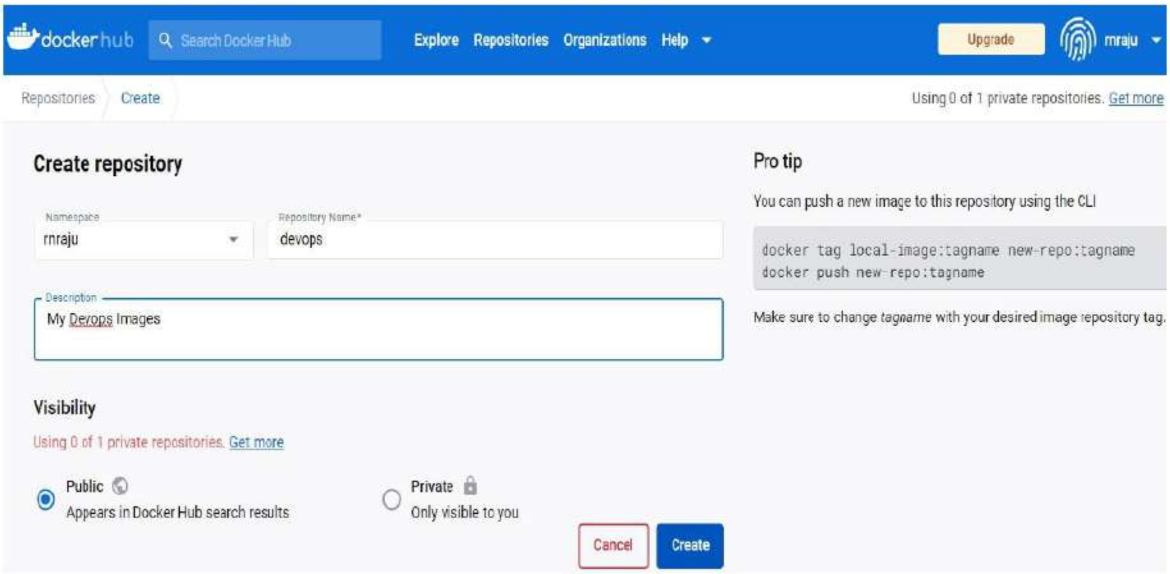
STEP 1: Sign up and Login docker hub through web interface:

https://hub.docker.com/

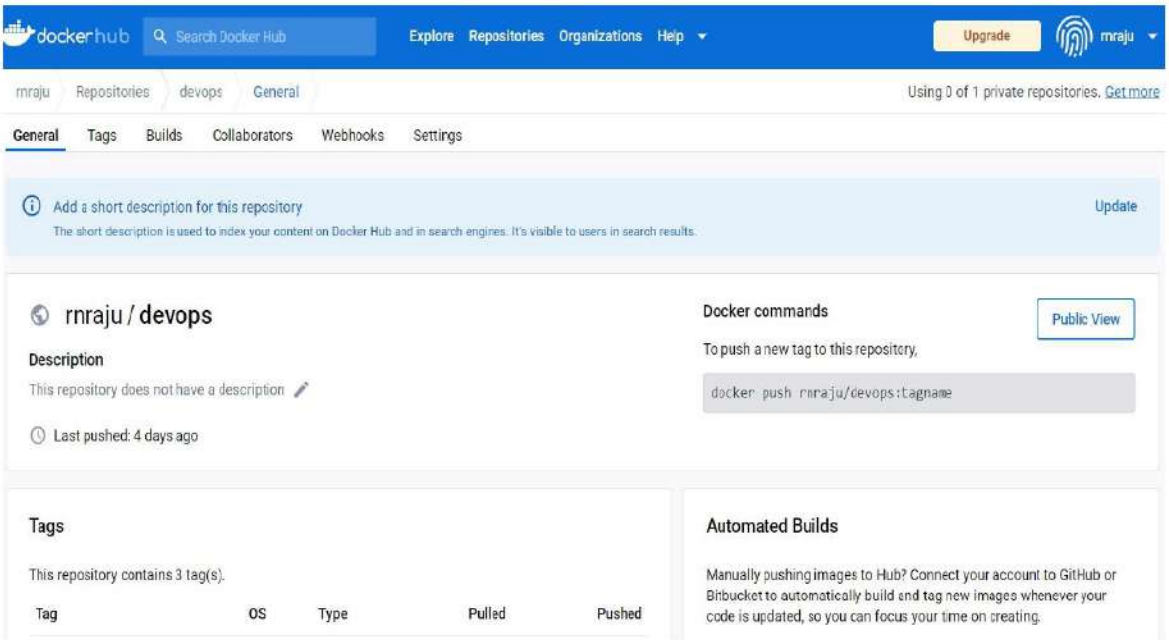
STEP 2: Select Create Repository:



Create a Repository:



STEP 3: Verify Created Repository



DOCKER HUB FROM THE CLI:

#docker login

Login : rnraju

Pass : xxxxx

BUILDING & PUSHING AN IMAGES TO DOCKER HUB:

PROJECT 1: BUILDING WEB SERVER IMAGE

STEP1: Create a Dockerfile

#vim Dockerfile

#Installing Apache Webserver

#From Ubuntu Image

FROM ubuntu:latest

#Update image

RUN apt-get update -y

#Installing Apache2 server

RUN apt-get install apache2 -y

#Copy index.html file into Document root location

COPY index.html /var/www/html/

#Running Port with 80

EXPOSE 80

#To make service to start

CMD ["apache2ctl", "-D", "FOREGROUND"]

STEP2: Building an Image:

#docker build -t rnraju/webserver:2.4 .

#docker images

STEP3: Make a container for Testing

#docker run -it -d -p 6000:80 --name Apache-webserver rnraju/webserver:2.4

#docker ps

STEP3: Go to Browser

http://IP-Address:6000

STEP4: Push the image to Docker hub

#docker login

#docker push rnraju/webserver:2.4

STEP5: Go and Verify in docker Hub

PROJECT 2: BUILDING DOCKER IMAGE FOR NODEJS:

STEP 1: Create a Dockerfile

#vim Dockerfile

#Specify a base image

FROM node:alpine

#Copy Local files to container

WORKDIR /usr/app

COPY ./ /usr/app

#Install Some Dependencies

RUN npm install

#Default Command

CMD ["npm", "start"]

STEP 2: Create a “package.json” file

```
#vim package.json
{
"dependencies": {
"express": "*"
},
"scripts": {
"start": "node index.js"
}
}
```

STEP 3: Create a “index.js” file

```
#vim index.js
const express = require('express');
const app = express();
app.get('/', (req, res) => {
res.send('WELCOME TO NODEJS...');
});
app.listen(8080, () => {
 console.log('Listening on port 8080');
});
```

STEP 4: Build an image:

```
#docker build -t rnraju/nodejs  .
#docker images
```

STEP 5: Running a container from the image file

#docker run -d -it –name mynodejs -p 4000:8080 rnraju/nodejs

#docker images

STEP 6: Enter the container with shell prompt

#docker run -it rnraju/nodejs

#cd /usr/app

#ls

STEP 7: Go and connect to the browser:

http://IP-Address:4000

STEP 8: Push the image to Docker hub

#docker login

#docker push rnraju/nodejs

STEP 9: Go and Verify in Docker Hub Repository

MANAGE DOCKER NETWORK

DOCKER NETWORKING

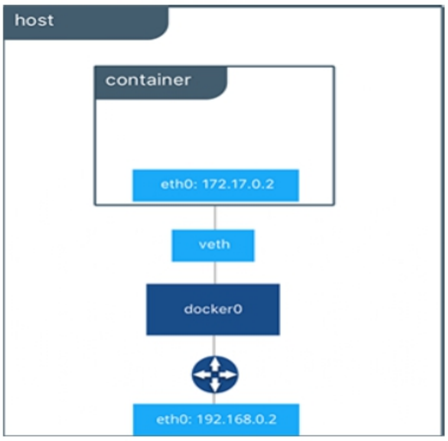
- Docker includes support for networking containers through the use of **network drivers.**
- It is used to establish communication between Containers and outside world via the Docker host machine.
- Docker supports different types of network drivers, each fit for certain use cases.

NETWORK DRIVERS:

- Docker's networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality:
  - Bridge
  - Host
  - Overlay
  - Ipvlan
  - Macvlan
  - None
  - Network plugins

BRIDGE NETWORK:

- A **Default Bridge Network (Bridge)** is created automatically, when you start docker.
- A bridge network is a Link Layer device which forwards traffic between network segments. A bridge can be a hardware device or a software device running within a host machine's kernel.



NOTE: The relationship between a host and containers is 1:N

To Check default bridge driver:

#ifconfig

#docker network ls

#docker network inspect bridge

Create a New Container:

#docker run -d -it --name Sample1 centos

#docker exec -it Sample1 ip a

#docker network inspect bridge

Create Another New Container

#docker run -d -it --name Sample2 centos

#docker exec -it Sample2 ip a

Testing Network connection:

#docker exec -it Sample2 bash

#ping 172.17.0.2

NOTE: It will communicate each one because both are running on same bridge

To remove container after exit use --rm flag:

#docker run --rm -it --name Sample3 centos bash

To Change Container Hostname:

#docker run -it --name Sample4 --hostname Sample4.example.com centos bash

#hostname

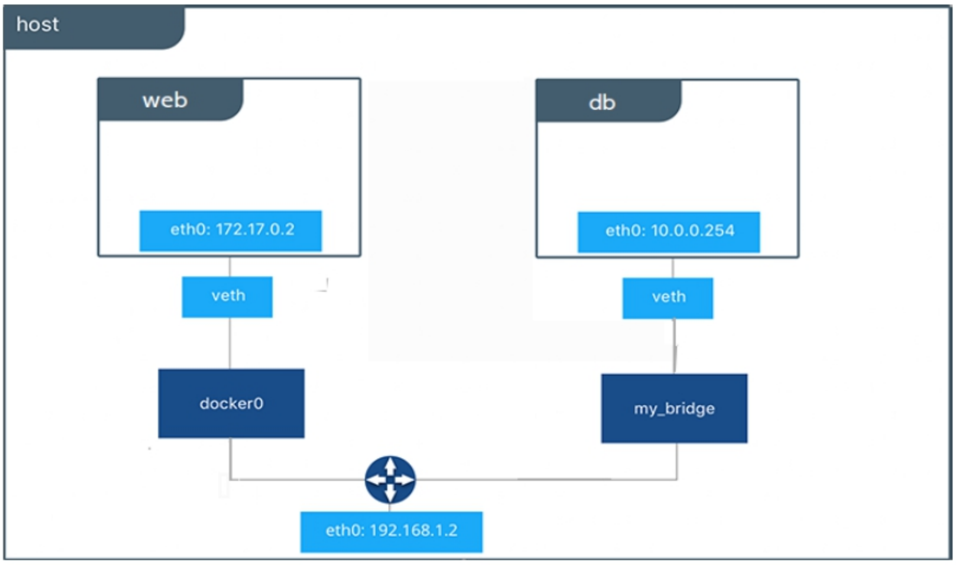
#vim /etc/hosts

172.x.0.x   Sample4.example.com

#exit

USER-DEFINED BRIDGE NETWORK

- These networks are superior to the **default bridge network**.
- It is usually used when your applications run in standalone containers that need to communicate.
- These are best when you need multiple containers to communicate on the same Docker host.



DIFFERENCE BETWEEN USER-DEFINED BRIDGES AND THE DEFAULT BRIDGE:

- User-defined bridges provide automatic DNS resolution between containers.
- User-defined bridges provide better isolation.
- Containers can be attached and detached from user-defined networks on the fly.
- Each user-defined network creates a configurable bridge.
- Linked containers on the default bridge network share environment variables.

Manage a user-defined bridge:

#docker network ls

#docker network create my-bridge

#docker network ls

Create a container on my-bridge:

#docker run -d -it --name Test1 --network my-bridge centos

#docker ps

#docker exec -it Test1 ip a

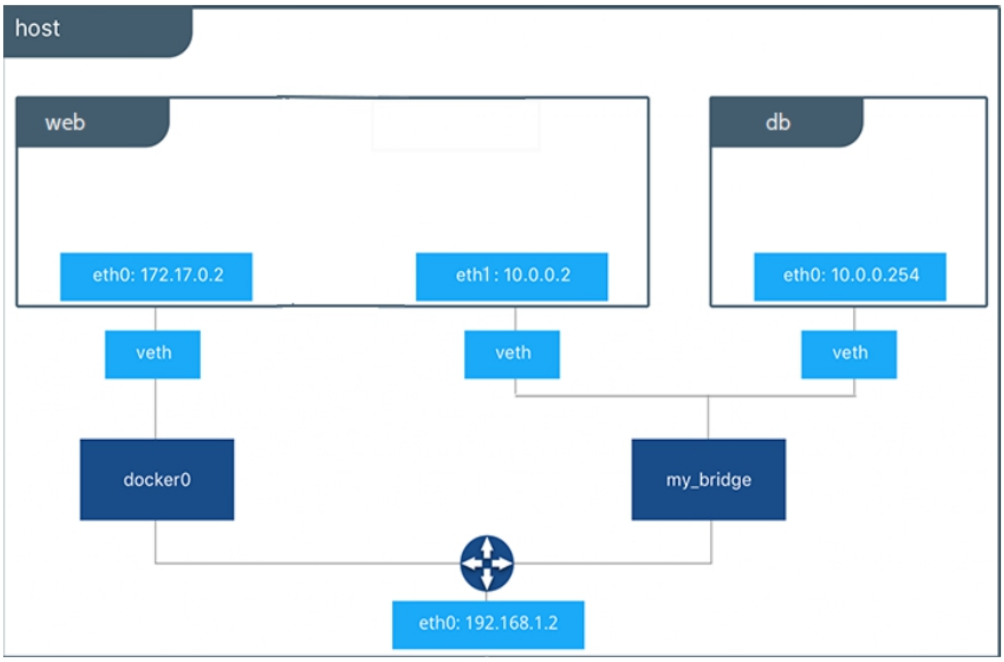
Connect a container to a user-defined bridge:

#docker run -it -d --name web-nginx --network my-bridge -p 8080:80 nginx

Disconnect a container from a user-defined bridge:

#docker network disconnect my-net my-nginx

COMMUNICATING DEFAULT & CUSTOM NETWORK CONTAINERS:



Create a new container on default bridge:

#docker run -d -it –name web centos

#docker ps

#docker exec -it web ip a

Creating custom subnet:

#docker network create my-bridge --subnet 10.0.0.0/16 --gateway 10.0.0.1

#docker network inspect my-bridge

Create a Container on My-Bridge:

#docker run -d -it --name db --net my-bridge centos

#docker exec -it db ip a

#docker exec -it db bash

#ping 172.17.0.2

Connect a Default network from My-Bridge network:

docker network connect bridge db

NOTE: Now Network Test is working

Disconnect a container:

#docker network disconnect bridge db

Removing User-Defined Bridge network:

Before Removing, if containers are currently connected to the network, disconnect them first.

#docker network ls

#docker network rm my-bridge

HOST NETWORK:

- If you use the **host network** mode for a container, that container's network stack is not isolated from the Docker host (the container shares the host's networking namespace), and the container does not get its own IP-address allocated.
- These are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.

  NOTE: The host networking driver only works on Linux hosts.

Create a new container:

#docker run -d -it--name Web1 host --network host nginx

#docker ps

#docker exec -it web1 ip addr show

Create another container from Centos image:

#docker run -it --name Web2 –network host centos bash

#ifconfig

NOTE: Host and Container networks are same

OVERLAY NETWORK:

- It connects multiple Docker daemons together and enable **swarm services** to communicate with each other.
- It provides to facilitate communication between a **swarm service** and a **standalone container.**
- These are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.

IPVLAN NETWORK:

- IPvlan networks give users total control over both IPv4 and IPv6 addressing.

MACVLAN NETWORK:

- It allows you to assign a **MAC address** to a container, making it appear as a physical device on your n/w.
- **macvlan driver** is sometimes the best choice when dealing with **legacy applications.**
- These are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.

NONE NETWORK:

- None network disables the complete networking stack on a container.
- Usually used in **conjunction** with a custom network driver.

  NOTE: Not available for **swarm services**.

Create a New Container (Disabling network for a container):

docker run --rm -dit --network none --name myalpine alpine sh

Check the container's network stack, by executing some common networking commands within the container.

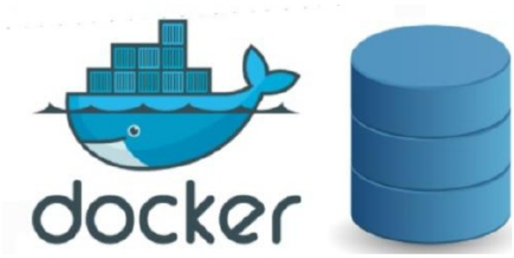
docker exec myalpine ip link show

NOTE: No ethernet was created

Only one instance of "host" and "null" networks are allowed.

#exit

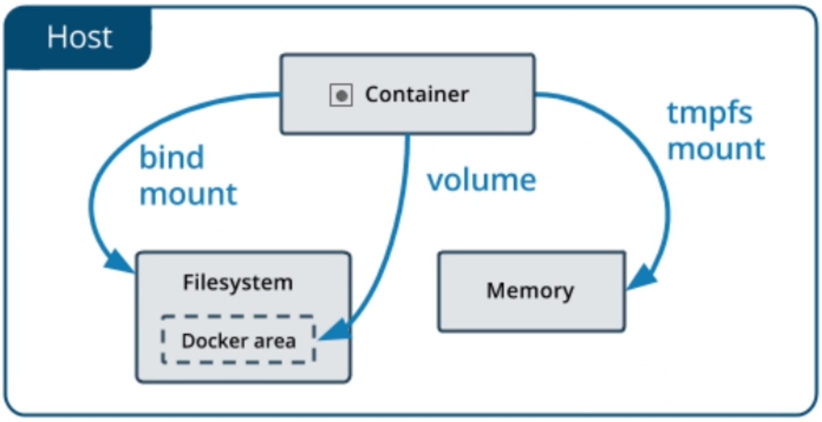
MANAGE DOCKER STORAGE



MANAGING DATA IN DOCKER

- By default, all files created inside a container on a **writable container layer.**
- That means:
  - The data doesn't persist when that container no longer exists.
  - You can't easily move the data somewhere else.
- Docker has two options for containers to store files in the **host machine persistently**.
  - **volumes**
  - **bind mounts**
- Running Docker on **Linux** use a **tmpfs mount** and on **Windows** use a **named pipe.**

MOUNTING TYPES:



- **volumes** are often a better choice than persisting data in a container's writable layer, because a volume doesn't increase the size of the containers using it, and the volume's contents exist outside the lifecycle of a given container.
- If your container generates non-persistent state data, consider using a **tmpfs** mount to avoid storing the data anywhere permanently, and to increase the container's performance by avoiding writing into the container's writable layer.

TMPFS MOUNT

- When you create a container with a tmpfs mount, the container can create files outside the container's writable layer. As opposed to volumes and bind mounts, a tmpfs mount is temporary, and only persisted in the host memory.

LIMITATIONS OF TMPFS MOUNTS:

- Unlike volumes and bind mounts, you can't share tmpfs mounts between containers.
- This functionality is only available if you're running Docker on Linux.

Create a New Container with tmpfs mount:

#docker run -d -it --name app1 --tmpfs /data centos

#docker ps

Connect a Container:

#docker exec -it app1 bash

#cd /data

#touch abc

#exit

Now Stop & Start a Container:

#docker stop app1

#docker start app1

Verify the data in a /data mount point:

#docker exec -it app1 ls /data   [No files in a mount point /data]

VOLUMES

- **Volumes** are the best way to **persist data** in Docker.
- These are stored in a **host filesystem** which is managed by Docker **(/var/lib/docker/volumes/** on Linux).
- Volumes support volume drivers, which allows you to store your data on **remote hosts / cloud providers.**

VOLUMES HAVE SEVERAL ADVANTAGES OVER BIND MOUNTS:

- Volumes are easier to back up or migrate than bind mounts.
- You can manage volumes using Docker CLI commands or the Docker API.
- Volumes work on both Linux and Windows containers.
- Volumes can be more safely shared among multiple containers.
- Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
- New volumes can have their content pre-populated by a container.
- Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts.

MANAGING VOLUMES:

Create a new container:

#docker run -d -it –name -v /my-data Test1 centos

#docker exec -it Test1 bash

#cd /my-data

#touch file1 file2

By default docker volumes are: /var/lib/docker/volumes/

#cd /var/lib/docker/volumes/

#ls CID/_dada/

CREATE AND MANAGE VOLUMES:

Creates a new volume that containers can consume and store data in. If a name is not specified, Docker generates a random name.

Syntax: #docker volume create [options] [VOLUME]

#docker volume create my-vol

#docker volume ls

#cd /var/lib/docker/volumes

#ls

To inspect a Volume:

#docker volume inspect my-vol

Start a Container with a Volume:

#docker run -it --name Test2 **-v** my-vol:/my-data centos bash

#cd /my-data

#touch aws azure gcp

#ls

(or)

#docker run -it --name Test2 --**mount** source=my-vol,target=/my-data centos sh

##cd /my-data

#touch aws azure gcp

#ls

Creating volume from an existing directory with data:

#docker run -it -v volume:/var centos bash

#cd /var

#ls

#cd /var/lib/docker/volumes/volume/_data/

#ls

DATA VOLUME CONTAINERS:

Data volume containers come in handy(easy) when you have data that you want to share between containers.

Create a container with one volume:

#docker run -it -v /data --name datavolume centos bash

#cd /data

#touch abc

#exit

Now, we need to connect some containers to this /data directory in the container.

#docker run -it --**volumes-from** datavolume ubuntu bash

#cd /data

#ls

DOCKER VOLUME BACKUPS:

- while your containers are immutable, the data inside your volumes is mutable.
- It changes, while the items inside your Docker containers do not. For this reason, you need to make sure that you are backing up your volumes.
- Volumes are stored on the system at **/var/lib/docker/volumes/**

BIND MOUNT

- **Bind mounts** may be stored anywhere on the **host system.**
- When you use a bind mount, a file or directory on the **host machine** is mounted into a container.
- The file or directory is referenced by its absolute path on the host machine.

NOTE: Bind mounts have **limited functionality** compared to **volumes.**

Start a Container with Bind mount:

#docker run -it -v /cloud:/aws centos bash

Here /cloud (on the Docker host) to the /aws directory inside the now running Docker container.

#cd /aws

#touch aws azure gcp

On the Host Machine verify the path:

#cd /cloud

#ls

USE A READ-ONLY VOLUME:

It mounts the directory as a read-only volume, by adding **ro.**

mount it in the read-only mode:

#docker run -it -v /data:/app:ro centos bash

#cd /app

#touch abc [

NOTE: Read-only file system error

Mount into a non-empty directory on the container:

- If you bind-mount a directory into a non-empty directory on the container, the directory's existing contents are obscured by the bind mount. This can be beneficial, such as when you want to test a new version of your application without building a new image.

#docker run -d -it --name broken-container -v /tmp:/usr nginx

The container is created but does not start. Remove it:

#docker container rm broken-container

REMOVING ONE OR MORE VOLUMES:

You cannot remove a volume that is in use by a container.

#docker volume ls

#docker volume rm my-vol

To Remove volume forcefully:

#docker volume rm -f my-vol

Remove all unused local volumes

#docker volume prune

#docker volume ls

NAMED PIPES:

- A **npipe** mount can be used for communication between the **Docker host** and a **container.**
- Common use case is to run a third-party tool inside of a container and connect to the Docker Engine API.

VOLUMES VS BIND MOUNT

- Compared to Bind Mounts, Volumes are more flexible and have more features, making them the most recommended option.
- In your container, Bind Mount provides you access to local file/directory storage on your local machine.

Volumes	Bind Mount
Easy backups and recoveries	There is a bit of complexity involved in backup and recovery. You don't have to worry about it if you know what folders to backup
To mount it, you only need the volume name. Paths are not required.	It is necessary to provide a path to the host machine when mounting with bind mounts.
Containers can have volumes created while they are being created.	The mount folder will be created when the host machine doesn't contain the folder.
There are APIs and CLIs for interacting with Docker volumes.	Using CLI commands, you cannot access bind mounts. The host machine still allows you to work with them instantly.
The volumes are stored in `/var/lib/docker/` volumes.	A bind mount can reside anywhere on a host computer.

docker Compose

MAIL: Rnraju4u@gmail.com **NUMBER:** +91 9848363431 **YOUTUBE:** SYSGEEKS

DOCKER COMPOSE

- Docker Compose is a tool for defining and running multiple containers as a single service.
- with a single command, you create and start all the services from your configuration.
- Compose works in all environments: production, staging, development, testing, as well as CI workflows. Commands for application:
  - Start, stop, and rebuild services
  - View the status of running services
  - Stream the log output of running services
  - Run a one-off command on a service
- The key features of Compose that make it effective are:
  - Have multiple isolated environments on a single host
  - Preserves volume data when containers are created
  - Only recreate containers that have changed
  - Supports variables and moving a composition between environments

SERVICE:

- A **service** can be run by one or multiple containers.
- Examples of services might include an HTTP server, a database, or any other type of executable program that you wish to run in a distributed environment.

DOCKER COMPOSE FILE STRUCTURE:

```yaml
services:
  foo:
    image: foo
  bar:
    image: bar
    profiles:
      - test
  baz:
    image: baz
    depends_on:
      - bar
    profiles:
      - test
  zot:
    image: zot
    depends_on:
      - bar
    profiles:
      - debug
```

USING COMPOSE IS BASICALLY A THREE-STEP PROCESS:

STEP1: Define your app's environment with a **Dockerfile** so it can be reproduced anywhere.

STEP2: Define the services that make up your app in **docker-compose.yml** so they can be run together in an isolated environment.

STEP3: Run **docker compose up** and the **Docker compose command** starts and runs your entire app.

INSTALL DOCKER COMPOSE:

STEP 1: To download and install Compose standalone, run:

#curl -L "https://github.com/docker/compose/releases/download/1.28.6/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compos

STEP 2: Apply executable permissions to the standalone binary in the target path for the installation.

#chmod +x /usr/local/bin/docker-compos

STEP 3: Create a Soft link for binary:

#ln -s /usr/local/bin/docker-compos /usr/bin/docker-compos

STEP 4: Test and execute compose commands using docker-compose.

#docker-compos version

SAMPLE APPLICATION WITH COMPOSE:

Create docker compose file at any location on your system.

#mkdir /dockercompos

#vim docker-compose.yml

```
version: '3'   ###https://docs.docker.com/compose/compose-file/     [versions]
services:
  web:
    image: nginx
    ports:
      - 9090:80
  database:
    image: redis
```

Check the validity of file:

#docker-compos config

Run the compose file:

#docker-compos up -d

#docker-compos ps or #docker ps

Bring down application

#docker-compos down

SCALE A SERVICES:

#docker-compos up -d --scale database=4

#docker-compos ps

#docker-compos down