**DOCKER**

* **Images** are layers of folders(file system) representings actual situation in the containers.
* **Containers** are ephemeral light weight processes which have its own independent named space. c groups – control everything like cpu, memory
* **Volumes** used to persist the data. Docker volumes are used to store data persistently outside of the container.
* Docker root directory 🡪 /var/lib/docker

Inside this

Buildkit – contains container related things.

Containers- writable layers of container comes here

Volume

Image -here not images store. Metadata of containers comes here

Overlayer 2- images are stored here (default storage driver).

Docker images and the container's read-only layers are stored here.

Docker save -o /tmp/mynewimage.tar <imageid> [another method for store image]

Plugins.

* we can store image in folder other than overlay2 by change the underlying storage plugin. ie, change the storage driver.

(AuFS, btrFS)

* We can change the location by make change in daemon.json file.
* Purpose of running container in detached mode :
* It allows the container to run in the background.
* Detached mode allows you to continue working in the same terminal without interruption.
* To reduce the number of layers in a Docker image:
* **Combine multiple commands** (like RUN, COPY) into single instructions.
* Use **multi-stage builds** to separate build-time and runtime dependencies.
* Use **slim base images** like alpine instead of larger images.
* Clean up temporary files and caches in the same RUN instruction to avoid extra layers.
* Use the **.dockerignore** file to exclude unnecessary files.
* Control group -helps to monitor and control. It I a linux based feature
* Docker will not allow you to remove an image if there are any **running** or **stopped containers** that are using that image. Similarly, if there are any **other images** that depend on the image also will not allow.
* Forcefull removal 🡪 docker rmi --force my\_image
* When we create container it will refer to image and have its own writable layer
* Enable docker - This means ensuring that Docker starts automatically when your system boots up.
* sudo systemctl enable docker
* Disable docker - This means preventing Docker from starting automatically when your system boots
  + sudo systemctl disable docker
* Some commands for docker
* sudo systemctl start docker
* sudo systemctl stop docker
* sudo systemctl restart docker
* systemctl status docker
* What makes u feel they are different virtual machines?

1. File system
2. Processes running inside
3. Network stack (ip, socket, port)
4. User
5. Hostname
6. Inter process communication (in the same machine two process can communicate each other through RPC [remote procedure call]. But between process in different machine is not possible)

Build run commit tag push/pull

Docker file🡪🡪 image 🡪🡪container 🡪🡪image🡪🡪image🡪 🡪docker registry

Stop,

Start,

Restart

Save /load

Image 🡪🡪backup.tar

* Create image from docker file – build
* Create container from image – run
* Create image from container – commit
* Create image from image – tag
* Push – image pushed to docker registry
* Pull – pull back from docker registry
* Save – save image to backup.tar
* Load – load back from backup.tar

**Docker 4 major things**

* images --> docker image
* containers --> docker container
* network --> docker network
* data volumes --> docker volume

**Docker images**

* docker images / docker image ls 🡪 list all the images
* docker build -t image\_name:<tag name > <path> 🡪 build an image from docker file.Tag name is not mandatory. Assume it as latest.
* docker push myusername/myimage:v2 🡪 push the image to registry.
* docker pull ubuntu:latest 🡪 pull image from registry
* docker rmi <image\_name\_or\_id> 🡪 remove image
* docker inspect <image\_name\_or\_id> 🡪 To view detailed information about a Docker image, including its layers, metadata, and configuration.
* docker image prune 🡪 remove all unused image
* Docker image history image\_name 🡪 to see history
* Docker image inspect image \_name | grep -i comment 🡪 see the comment part
* 4 layers of the Image

LowerDir - they store the Image

upperDir - they store the Image

WorkDir - only on run time,ref on the Image

MergedDir - only on run time

**Docker Containers**

* docker run -it ubuntu bash 🡪 will create a new container and get you inside it.
* docker run -d -p 8080:80 nginx 🡪 [host:container]
* docker run --name my\_nginx -d nginx 🡪 create a container with customised name
* docker ps 🡪 current running container
* docker ps -a 🡪 show all container
* docker start <container\_id> 🡪 start container
* docker stop <container\_id> 🡪 stop container
* docker restart <container\_id> 🡪 restart container
* docker kill <container\_id> 🡪 kill container
* docker rm <container\_id> 🡪 remove container
* docker container prune 🡪 remove all stopped container
* docker rm -f <container\_id> 🡪 remove force fully
* docker exec -it <container\_id> bash 🡪connect to running container
* docker attach <container\_id> 🡪attach to running process
* docker inspect <container\_id> 🡪 more info about the container
* docker stats 🡪 Docker stats like cpu, memory
* kill -9 <pid>
* docker container prune🡪 remove all stopped container
* docker system prune -a 🡪 remove image,network,volume
* Docker commit -m “ubuntu with ping” container\_id image-name 🡪 create image from container
* Docker pause container\_id 🡪 pause the container
* Docker unpause container\_id 🡪 to unpause
* Docker stop $(docker ps -a -q) 🡪 stop all the containers
* Docker rm $(docker ps -a -q) 🡪 remove all containers
* Docker top cont\_id 🡪 show all the process and how much storage and cpu all it used(for only 1 container
* Docker logs <container\_id> 🡪 see whether there is any error

**Docker Volume**

Application should be stateless

* When u create a container, it allows to mount(attach) one of the folder inside the container to another folder outside (host machine). Whenever the container crashes, data will be the there in outside folder. So you create another container and folder in it and attach that folder to this outside folder will give all data to it.
* Docker supports 3 types of volume

**>** TempFS - stored in RAM. For onetime use. Reduce writable layers

**>** Bind mount permission are maintained by user. Security risk (ie u can accidently mention critical folders)

**>** Volumes

i) Anonymous volume

ii) Named volume - maintained by docker. Stored by docker root directry. lifecycle of this is managed by docker.

**Commands**

* docker volume create <volume name> 🡪 create volume. If no name is specified, Docker will generate a random name for the volume.
* docker volume ls 🡪 list volume
* docker volume inspect <volume name> 🡪 inspect volume
* docker volume rm <volume name 🡪 remove volume
* docker volume prune 🡪 remove all unused volume
* docker run -d -v /home/user/data:/app/data ubuntu --> bind mount (host: container)
* docker run -v my\_named\_volume:/data ubuntu --> named volume (directry inside the container)
* docker run -v /data ubuntu --> anonmous volume
* docker run -v my\_volume:/app/data:ro ubuntu --> change to read only

**Docker network**

Networking used for connecting devices for all communication.

* IP Address: unique routable address given to a device
* >> for a network 🡪 cidr/ subnet
* Ethernet card: NIC, Connect device to network (wired) through ethernet cable
* Ethernet Cable
* Network switch : ARP sent message to broadcast ip, the handshake signal will broadcast to all device in network and cache the data.
* Gateway: Connect between two diff type of network
* Nating /Denating : from address to public ip
* Subnetting
* When u install docker, docker will automatically create a switch docker 0.

Also there is ip address outside and at the time of creation of docker 0 it will also create rules in ip address.

* Also root nic called eth 0.
* Inside the container also have eth 0. Which is connected to docker 0 using veth. Also if some other container have eth 0 and connected to docker 0 through veth. If a container need to sent message to another container, sent through switch.
* If the container sent message to some public ip like google.com, then it will sent to docker 0 and through eth 0 of machine and sent to internet.
* Docker supports 6 diff network. In this 3 are use by default
* Bridge
* Host (no port forfarding required. It directly connect to host network.directly connected through the entry door)
* None
* Networks only present in swap mode 🡪 ipvlan ,maclan,overlay

Commands

------------------

* docker network create my\_network --> create network
* docker network ls --> list network
* docker network inspect my\_network --> inspect network
* docker network rm my\_network --> remove network
* docker network prune --> remove all unused network
* docker run -it --net bridge ubuntu bash

**Custom bridge network**

---------------------------------

* A **custom bridge network** in Docker is useful for scenarios where you need more control over the networking behaviour of your containers.
* It allows you to define your own subnets, gateways, and ensures that containers can communicate securely within a defined network.
* Using custom bridge networks gives you greater flexibility and isolation when managing containerized applications
* docker network create --driver bridge --subnet 172.21.0.0/24 --gateway 172.21.0.1 custom-bridge

**Difference between default and custom bridge ???**

**Default bridge**

* **Configuration**: It uses Docker's default settings, meaning it assigns IP addresses from a default subnet (typically 172.17.0.0/16), with the default gateway 172.17.0.1.

Docker assigns a subnet from its default IPAM (IP Address Management) pool, which often includes subnets like 172.17.0.0/16, 172.18.0.0/16, and sometimes 172.24.0.0/16 depending on the available ranges.

* **Container Communication**: Containers attached to the default bridge network can communicate with each other using their IP addresses,

**Custom bridge**

* **Configuration**: You have full control over the network’s configuration, including:
* **Subnet**: You can define a custom subnet (e.g., 192.168.100.0/24) to isolate containers from other networks or the host.
* **Gateway**: You can specify a custom gateway (e.g., 192.168.100.1).

**Container Communication**: Containers connected to a custom bridge network can automatically communicate with each other using their container names or IP address

>>> create image from docker file

* Every time u do **FROM** it refers to **base layer**.
* FROM – it is a mandatory command in docker file. Even if u use scratch
* FROM ubuntu
* FROM ubuntu:latest 3 different ways to write in docker file
* arg mine = latest (arg is for reduce duplicate)

FROM ubuntu $mine

* docker build -t image\_name . 🡪 to create image from docker file( -t refers tag , . refers
* whenever a image creating it will check for any image present, if present it will points to that. Other wise it will create one and point to that.(eg; if ubuntu present, then new ubuntu \_image will take reference from that)
* inside 2.Runfile 🡪 RUN apt update -y

RUN apt install java

* docker build -t myrun -f 2.Runfile . ( myrun = image\_name)
* commands to execute when u are starting the container

i) CMD : defines default instruction for execute at starting the container which can be over written (if u give any command , what u give in cmd willnot get executed, it will execute only what u mention in command)

ii)ENTRYPOINT - whatever mention in container has been executable. Convert a container into executable

* at build time 🡪 RUN command
* at run time 🡪 cmd and entrypoint command
* if u have cmd and entrypoint in 1 file then entry point will execute and cmd is execute as argument in it. If u give argument in terminal it will execute not the argument in cmd .
* ADD abc.txt /var/vilas and COPY is copy something from the context
* Copy is more recommended instead of add.add can be more vulnerable to security attacks. Also curl is recommended to download files from internet instead of add. Because curl will run like ‘RUN curl’ so u can concatenate this too with other RUN commands. So can **reduce the layers.**
* EXPOSE - for port number
* RUN cp – copy from location inside the image to another location inside the file.

And COPY – will copy from context

* ONBUILD – when somebody use reference as my image and if I created a ONBUILD in it, then when they execute first the things in ONBUILD will build and after that others build.
* HEALTHCHECK – check container health
* WORKDIR – Give relative path reference .we can define any no of WOEKDIR.
* USER – changing default user
* ENV – for environment variables
* ARGS – variables
* MAINTAINER – for defaining the owner
* VOLUME – whenever u create container, this will create an anonymous volume to the container.

What makes u feel they are different virtual machines?

1. File system
2. Processes running inside
3. Network stack (ip, socket, port)
4. User
5. Hostname
6. Inter process communication (in the same machine two process can communicate each other through RPC [remote procedure call]. But between process in different machine is not possible)

**DOCKER COMPOSE**

* one command will take care of build image and run container. also for multiple containers to create
* If image is already there, it will download from that.otherwise build
* automatically create a network
* if the file name is docker-compose.yaml don’t need to give. Otherwise give filename also

>>docker compose -f dc1.yaml up -d

* dc1.yaml

version: "3.8"

services: --> name of the service is web

 web:

   build: . --> location of dockerfile

   ports: -->port mapping

     - "8080:8080"

## this will create a image named docker-k8s-web and container named docker-k8s-web-1

## this will create a network named docker-k8s\_default

* dc2.yaml

services:

 mynginx: -->service name

   image: "nginx" --> directly give the image name instead of location of dockerfile

   container\_name: mynginx

   ports:

     - 80:80

## this will create a image named nginx and container mynginx

Dockerfile, docker-compose.yml

* FROM base as dev 🡪 they represent **stages within the same build process**, each with its own context and environment.
* COPY . . 🡪 copy code from current folder to working directory mentioned in container
* ./vote/usr/local/app 🡪 bind mount volume
* npm VS npm ci install 🡪 both are for installing dependencies for node.js . npm ci more reliable bcz it will delete all node\_module already have before installation but npm will update the node\_module.
* There are 2 networks created 🡪 front-tier and back-tier
* Dockerfile will seen on file mentioned on context
* ARG – arguments are like variables
* COPY package.json\* ./ 🡪 copy all oackage.json format files to WORKDIR
* mcr.microsoft.com/dotnet/sdk 🡪 Docker image hosted on Microsoft's Container Registry (MCR) that contains the .NET SDK (Software Development Kit). This SDK is used to develop and build .NET applications.
* Profile [“seed’] 🡪 whatever has the profile willnot execute in compose. Conditionally execute can be possible through this through the command 🡪

docker compose - -profile seed up -d

* ab 🡪 bench marking tool

**DOCKER SWARM**

* Docker Swarm is Docker’s native cluster management and orchestration tool that allows you to manage a group of Docker hosts as a single virtual system.
* docker swarm init 🡪 initialize a Docker Swarm on a Docker host.
* docker node ls 🡪 It shows the status and information of all the nodes (both **manager** and **worker** nodes) that are part of the swarm
* docker info 🡪 provides detailed information about the current state of your Docker installation. It shows useful system and Docker-specific details such as the version, storage driver, container stats, and whether Docker is running in **Swarm mode**, among other things.
* Docker service create -p 80:80 –replicas 2 –name webserver nginx 🡪 Two task will create. And each task will create container on each machine.if a container crash the task will take care again creating the container , it may be in node1 or node2
* Docker service ls 🡪 list all the service
* Docker service ps webserver 🡪 (werserver is service name) show in which machine the task running.
* Curl node2-vm , curl node1-vm, curl localhost
* Internally we doing this docker will automatically create a loadbalancer.

Normally in docker we create a container and give -p then it listen to port 80 of machine through its port 80. And try to create another container and connect through same port it willnot allow.If u create a software which is using port 80 even it will not work. But in docker swarm, there is a load balancer created and it will help to create multiple container and connect through same port. Also no port forwarding is not happening.

* Curl run based on ip rules.
* At the time of docker swarm init , default create a overlay network(ingress) and a bridge network.overlay responsible for deliver to machine and bridge for deliver to containers in it.

**Rolling update**

* one a container up and running and have good health, then the other one is deleted
* A **Rolling Update** in Docker is a deployment strategy where new versions of a service or application are gradually rolled out to replace old ones, minimizing downtime and ensuring that the system remains available throughout the update process.

**Stack**

docker stack deploy --compose-file docker-stack.yml myvote

* docker stack deploy - used to deploy a stack to a Docker Swarm cluster
* stack - group of services that work together as part of an application.
* myvote - name of the stack
* here it will create overlay network and connect to that.

**Notes**

Kubernetes 🡪 docker-shim🡪docker 🡪 containerd 🡪 containers

* Docker container is an instance of containerd
* Docker containerd shim – runc v2

>> when an image created it will refer to image and writable layer. By again starting the container it will refer to same writable layer,not new writable layer will create.

* Docker cp filename.txt container\_name:path 🡪 host to container
* Docker cp container\_name:filename . 🡪 container to host