Docker

Contents

Docker

[Imp links:
https://docs.docker.com/engine/install/ubuntu/ → Docker Installation
https://hub.docker.com/ → open-source platform for docker images]

1) Docker:

- Docker is an open-source centralized platform designed to create, deploy, and run applications by using container.
- Docker is a containerisation platform which packages our application and all its dependencies together in the form of container.

2) Docker Images:

used to create containers.

- Docker images are read only templates that contain the instructions and files for creating containers.
- Docker images contain binaries/libraries which are necessary for one software application.
  (blueprints for creating containers)

3) Docker Containers:

isolated environments that run applications.

- 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.
- It's a running instance of an image (run time of docker image) and it's writable.

(i) 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.

Practical:

Install Docker

Ubuntu instance

Docker Installation:

(i) Set up Docker's apt repository

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
/etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

(ii) To install the latest version, run:

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin
docker-compose-plugin
```

(iii) Verify that the Docker Engine installation is successful

```
docker --version
```

Pull Image from Docker hub and create container

(i) list docker images

```
sudo docker images
```

(ii) give permission for docker env. so, no need to use sudo or switch to root user

```
sudo chmod 777 /var/run/docker.sock
   (or)
sudo su              # switches to root user
```

(iii) Pull tomcat image with version to CLI from docker hub

Search tomcat to get tomcat image

Copy command at top left once getting inside image and paste it in CLI

i.e, sudo docker pull tomcat:8.0

(iv) Run tomcat application (Creates container)

sudo docker run -itd -p 8888:8080 --name tomcat_deploy tomcat:8.0

[Note: we can give any port for docker container, just by changing ports we can create multiple containers using single command from docker image

We can remove any container using container ID:

i.e, docker rm -f <container_ID/name>

(v) list docker containers

docker ps --> list running containers

docker ps -a --> list all containers

(vi) Validate in AWS

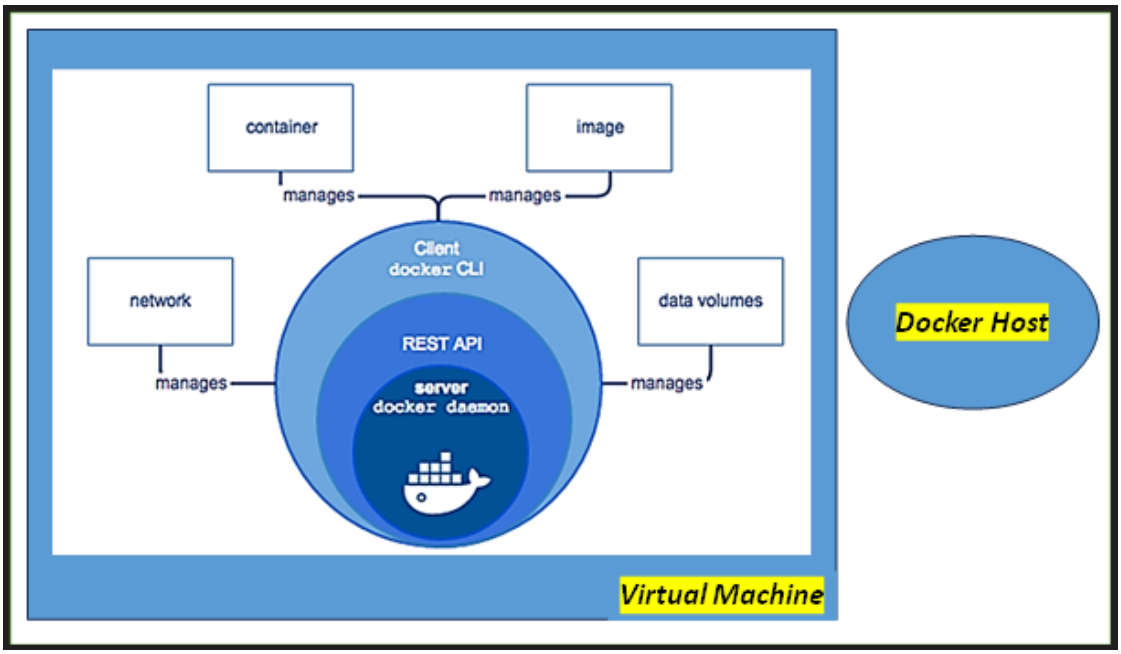
Allow all traffic and copy paste ***instanceIP:port_no*** in browser

[Note: We can create multiple containers using one command once we pull application image]

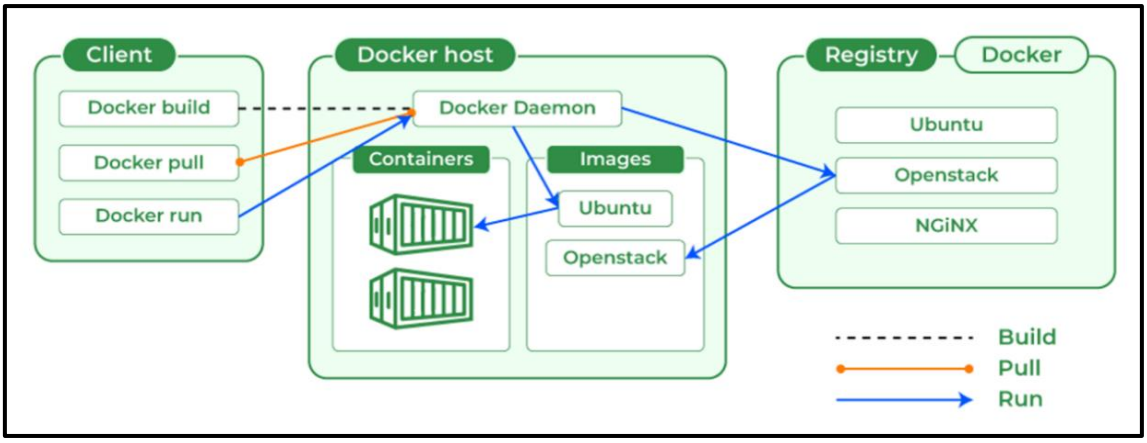
(vii) modify any container: Login to that container

docker exec -it <container_ID> /bin/bash

4) Docker Architecture



- Complete Virtual machine is known as Docker Host



Docker uses a client-server architecture, where the Docker client interacts with the Docker daemon to manage container images, containers, and networks.

(i) Docker Client:

- A command-line interface (CLI) or API that communicates with the Docker daemon.
- Allows users to create, start, stop, and manage containers.
- Docker client uses commands and REST APIs to communicate with the Docker Daemon (Server).

(a) *REST API:*

An API is used by applications to interact with the Docker daemon. It can be accessed by an HTTP client.

(ii) Docker Host:

A Docker host is a type of machine that is responsible for running more than one container. It comprises the Docker daemon, Images, Containers, Networks, and Storage.

(a) Docker Daemon:

- Runs as a background process on the host machine.
- Manages container images, containers, and networks.
- Interacts with the Docker client.

(b) Docker Images:

- Read-only templates containing the instructions to build a container.
- Built from a Dockerfile, which specifies the base image, dependencies, and configuration.
- Can be pushed to and pulled from registries like Docker Hub.

(c) Docker Containers:

- Runtime instances of Docker images.
- Can be started, stopped, paused, and restarted.
- Each container has its own isolated environment, including file system, network, and process space.

(d) Docker Network:

- A virtual network created by Docker to connect containers and other network resources.
- Can be configured using various network drivers, such as bridge, overlay, and host.
- Provides isolation and security for containers.

(iii) Docker Registry:

- A central repository for storing Docker images.
- Allows users to share and distribute images.
- Popular registries include Docker Hub, private registries, and cloud-based registries

❖ ***Key Points:***

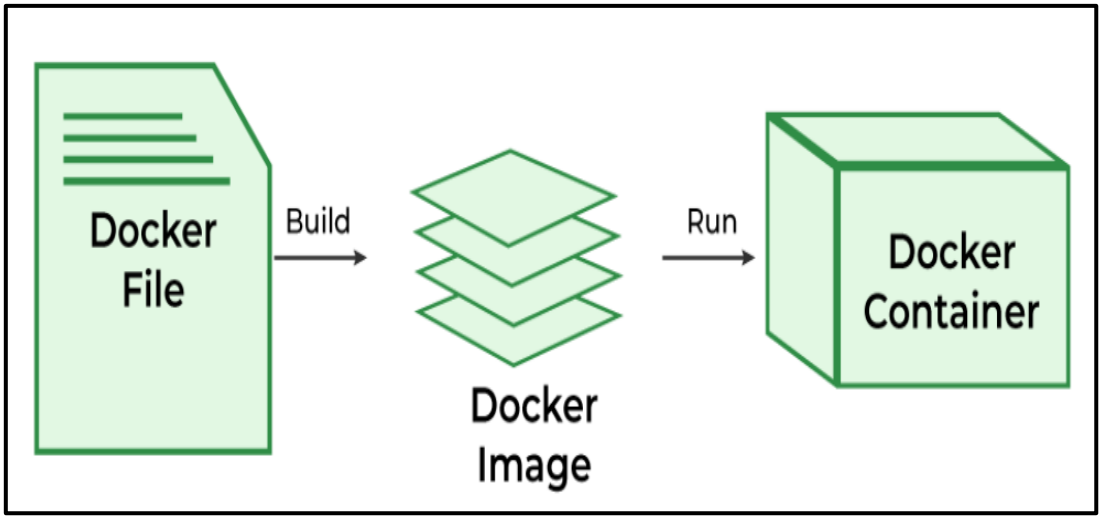
- Docker images are used to create containers.
- Containers are isolated environments that run applications.
- The Docker daemon manages containers and images.
- The Docker client interacts with the daemon.
- Docker networks provide connectivity between containers.
- Registries are used to store and share images

5) Difference between Virtual Machine and Docker Containers

Aspect	Containers	Virtual Machines (VMs)
Resource Utilization	Containers share the host OS kernel, making them **lighter and faster**	VMs have a full-fledged OS and hypervisor, making them **more resource-intensive**
Portability	**Highly portable**, can run on any system with a compatible host OS	**Less portable**, requires a compatible hypervisor to run
Boot Time	**Seconds**	**Minutes**
Security	Provides **less isolation**; shares the host OS	**Provides higher isolation/security**; each VM has its own OS and is isolated from the host and other VMs
Management	**Easier to manage**, designed to be lightweight and fast-moving	More **complex to manage** due to the full OS and hypervisor overhead
Deployment Time	**Faster**, containers are lightweight and easy to deploy	**Slower,** requires full OS installation
Storage	**Minimal storage footprint** (MBs to a few GBs)	**Requires a full OS image** (GBs in size)

6) Dockerfile:

- Text file that contains a set of instructions which are used by docker build for building a custom docker image.
- Dockerfile is a file where you provide the steps to build your Docker Image.



➔ Docker file commands:

(i)	**FROM**	:	specify the base image for building a new image (the first command in a Dockerfile)
(ii)	**RUN**	:	to execute commands
(iii)	**ADD**	:	to connect to internet (download files from distant HTTP/HTTPS destinations)
(iv)	**CMD**	:	to start the process inside the container
(v)	**ENTRYPOINT**	:	to start the process
(vi)	**COPY**	:	copy files/folders from local to container
(vii)	**ENV**	:	Environment variables, we declare inside the docker file
(viii)	**ARG**	:	Run time variables
(ix)	**EXPOSE**	:	Used for open the port
(x)	**USER**	:	Switches between the users
(xi)	**VOLUME**	:	Attach the volumes
(xii)	**WORKDIR**	:	Set the default folder for docker instruction
(xiii)	**LABEL**	:	brief about dockerfile
(xiv)	**MAINTAINER**	:	owner

FROM and RUN command, Create image	
(i) vim dockerfile FROM ubuntu RUN apt update -y RUN apt install git -y Save (ii) sudo chmod 777 /var/run/docker.sock (iii) docker build -t myfirstimage:1.0 -f dockerfile . (iv) docker images	# giving base image as ubuntu using FROM and installing git using RUN # giving permission for image # creating new image # list images
Similarly, creating image with different version	
(i) vim dockerfile1 FROM alphine RUN apk add git Save (ii) docker build -t myfirstimage:2.0 -f dockerfile1 . (iii) docker images	# taking alphine as base image and installing git # image name is same for both but there is difference in versions (tags) so using tags we can create container
ADD and COPY command, create image and container	
(i) vim docerfile2 FROM ubuntu RUN apt update -y ADD https://github.com/Siddeshg672/hello_world_public_war.git /home COPY Dockerfile /home/ubuntu COPY abc /home/ubuntu (ii) create image docker build -t myfirstimage:2.0 -f dockerfile2 . (or) docker build -t myfirstimage:2.0 -f dockerfile2 /home/ubuntu [Note: "." = current path /home/ubuntu --> path of file that needs to be copied]	# adding GitHub repository using ADD # copying files to container (Dockerfiles, abc) Source = local (ubuntu) Destination = container (ubuntu, base image) # we have two approaches to copy files from local to container, ♦ one when file and dockerfile is in same path then we can just create images by giving current path i.e, "."

(iii) list images
docker images

(iv) create container
docker run -itd <img_ID>

(v) list containers
docker ps

(vi) login to container
docker exec -it <container_ID> /bin/bash
cd /home/ubuntu/
ls

(docker build -myfirstimage:2.0 -f dockerfile2 .)

- If we have file and dockerfile in different path then we should give path of file while creating images

(docker build -t myfirstimage:2.0 -f dockerfile2 home/ubuntu/)

can see copied file in container inside ubuntu folder (/home/ubuntu/)

ENV: Environment variable, declared inside the docker file

(i) dockerfile:
vim df1

```
FROM ubuntu
ENV path=/home
RUN apt update -y
ADD https://github.com/Siddeshg672/hello_world_public_war.git ${path}
COPY abc.txt ${path}
```

(ii) creating image from dockerfile

```
docker build -t myimg:1.0 -f df1 /home/ubuntu
                (or)
docker build -t myimg:1.0 -f df1 .
```

ARG - Run time variables, declared while creating images

(i) dockerfile: vim df2

```
FROM ubuntu
ARG path
RUN apt update -y
ADD https://github.com/Siddeshg672/hello_world_public_war.git ${path}
COPY Dockerfile ${path}
```

(ii) creating image from dockerfile

```
docker build -t myimg:2.0 -f df2 --build-arg path=/home .
```

EXPOSE - Used for open the port (Tomcat installation)

(i) dockerfile: vim df3

```
FROM alpine:latest
LABEL dockerfile to deploy war file
MAINTAINER siddesh
ARG package
RUN apk add $package
RUN git clone https://github.com/Siddeshg672/hello_world_public_war.git -b main

RUN apk add maven
RUN apk add wget
RUN apk add openjdk11
WORKDIR hello_world_public_war
RUN mvn clean install

RUN echo "@testing http://nl.alpinelinux.org/alpine/edge/testing" >>
/etc/apk/repositories \
 && apk add --update \
   gpgme \
   gzip \
 && rm -rf /var/cache/apk/*

ENV CATALINA_HOME /usr/local/tomcat
ENV PATH $CATALINA_HOME/bin:$PATH
RUN mkdir -p "$CATALINA_HOME"
WORKDIR $CATALINA_HOME

# see https://www.apache.org/dist/tomcat/tomcat-8/KEYS

ENV TOMCAT_MAJOR 8
ENV TOMCAT_VERSION 8.5.78
#ENV TOMCAT_TGZ_URL https://www.apache.org/dist/tomcat/tomcat-
$TOMCAT_MAJOR/v$TOMCAT_VERSION/bin/apache-tomcat-
$TOMCAT_VERSION.tar.gz
ENV TOMCAT_TGZ_URL https://archive.apache.org/dist/tomcat/tomcat-
$TOMCAT_MAJOR/v$TOMCAT_VERSION/bin/apache-tomcat-
$TOMCAT_VERSION.tar.gz

RUN wget -O tomcat.tar.gz "$TOMCAT_TGZ_URL" \
&& gunzip < tomcat.tar.gz | tar xvf - \
```

```
&& mv apache-tomcat-*/* ./ \
&& rm -r apache-tomcat-* \
&& rm bin/*.bat \
&& rm tomcat.tar.gz* \
&& (cd /usr/lib/ && ln -s libssl.so.1.0.0 libssl.so.10 && ln -s libcrypto.so.1.0.0
libcrypto.so.10)

ENV LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/usr/lib/
WORKDIR $CATALINA_HOME
EXPOSE 8080
RUN cp /hello_world_public_war/webapp/target/webapp.war
/usr/local/tomcat/webapps/webapp.war
```

(ii) creating image from dockerfile

```
docker build -t myimg:3.0 -f df3 --build-arg package=git .
```

(iii) creating container from image

```
docker run -id --name tomcat -p 8082:8080 <img_ID>
```

(iv) Validate --> check in browser - instanceIP:8082

(or)

(i) once war file is imported/downloaded, copy war file to local from container where war file is downloaded

```
docker cp 0f7d8d384f95:/usr/local/tomcat/webapps/webapp.war /home/ubuntu
```

(ii) create dockerfile: vim df4

```
FROM tomcat:8.0
EXPOSE 8080
COPY webapp.war /usr/local/tomcat/webapps/webapp.war
```

(iii) creating image from dockerfile

```
docker build -t myimg:7.0 -f df4 .
```

(iv) creating container from image

```
docker run -id --name tom -p 8088:8080 <img_ID>
```

(v) Validate --> check in browser - instanceIP:8088

Note: Do both methods, 1st to get war file and 2nd to make server run

USER - Switches between the users
FROM selenium/standalone-chrome:latest #RUN mkdir test && touch sampletest && ps -ef USER root RUN apt-get update -y RUN apt-get install git -y RUN apt-get install wget -y RUN apt-get install curl -y

7) ***Dangling images***

without repository and tag name

Dangling images in Docker are essentially unused images that have no tags associated with them. They are often created when you rebuild an image with the same tag, causing the old image to lose its tag and become "dangling".

docker images --filter "dangling=true"	# Identifying Dangling Images
docker image prune	# Removing Dangling Images
docker rmi $(docker images -f dangling=true -q)	# Removing all Dangling Images at once

➔ **Preventing Dangling Images**

Tag Images Carefully: Use unique tags for each image version.

Remove Unused Images Regularly: Periodically check for dangling images and remove them to keep your system clean.

8) Most used Docker commands:

Sl No	*Commands*	*Description*
1)	docker attach	Attach local standard input, output, and error streams to a running container
2)	docker build	Build an image from a Dockerfile
3)	docker builder	Manage builds
4)	docker checkpoint	Manage checkpoints
5)	docker commit	Create a new image from a container's changes
6)	docker config	Manage Docker configs

7)	docker container	Manage containers
8)	docker context	Manage contexts
9)	docker cp	Copy files/folders between a container and the local filesystem
10)	docker create	Create a new container
11)	docker diff	Inspect changes to files or directories on a container's filesystem
12)	docker events	Get real time events from the server
13)	docker exec	Run a command in a running container
14)	docker export	Export a container's filesystem as a tar archive
15)	docker history	Show the history of an image
16)	docker image	Manage images
17)	docker images	List images
18)	docker import	Import the contents from a tarball to create a filesystem image
19)	docker info	Display system-wide information
20)	docker inspect	Return low-level information on Docker objects
21)	docker kill	Kill one or more running containers
22)	docker load	Load an image from a tar archive or STDIN
23)	docker login	Log in to a Docker registry
24)	docker logout	Log out from a Docker registry
25)	docker logs	Fetch the logs of a container
26)	docker manifest	Manage Docker image manifests and manifest lists
27)	docker network	Manage networks
28)	docker node	Manage Swarm nodes
29)	docker pause	Pause all processes within one or more containers
30)	docker plugin	Manage plugins
31)	docker port	List port mappings or a specific mapping for the container
32)	docker ps	List containers
33)	docker pull	Pull an image or a repository from a registry
34)	docker push	Push an image or a repository to a registry

35)	docker rename	Rename a container
36)	docker restart	Restart one or more containers
37)	docker rm	Remove one or more containers
38)	docker rmi	Remove one or more images
39)	docker run	Run a command in a new container
40)	docker save	Save one or more images to a tar archive (streamed to STDOUT by default)
41)	docker search	Search the Docker Hub for images
42)	docker secret	Manage Docker secrets
43)	docker service	Manage services
44)	docker stack	Manage Docker stacks
45)	docker start	Start one or more stopped containers
46)	docker stats	Display a live stream of container(s) resource usage statistics
47)	docker stop	Stop one or more running containers
48)	docker swarm	Manage Swarm
49)	docker system	Manage Docker
50)	docker tag	Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
51)	docker top	Display the running processes of a container
52)	docker trust	Manage trust on Docker images
53)	docker unpause	Unpause all processes within one or more containers
54)	docker update	Update configuration of one or more containers
55)	docker version	Show the Docker version information
56)	docker volume	Manage volumes
57)	docker wait	Block until one or more containers stop, then print their exit codes

➔ ***Docker commands examples***

docker build -t myfirstimage:5.0 -f Dockerfile3 .	# Create image using dockerfile
docker images	# List all images
docker run -d -p 8090:8080 <img_name/ID>	# Build container using image

docker ps	# list running containers
docker ps -a	# List all containers
docker exec -it <container_name/ID> /bin/bash (or) docker exec -it <container_name/ID> /bin/sh	# login to container # /bin/bash = default shell
docker stop <container_ID/name>	# Stop container
docker start <container_ID/name>	# Start container
Pull nginx image and create container sudo su cd /home/ubuntu docker run --name some-nginx -id -p 8080:80 nginx	# run = create container # -p = port # -i = keeps STDIN (the input stream) open even if not attached # d = "detached" mode, which means the container runs in the background # nginx = image
docker rmi <image_name/ID>	# Remove images
docker rm <container_name/ID>	# Remove container (removes containers which isn't running)
docker rm <container_name/ID> -f	# Remove container forcefully (stops and terminate)
docker rm -f $(docker ps -a -q)	# removes all containers forcefully
docker inspect <container_name/ID> docker inspect <Image_name/ID>	# gives all information about containers/Images like entry point, working directory, ports, networks, etc.
docker run -id -name nginx -p 8080:80 --memory "200mb" -cpus=".5" nginx	# allocates memory and CPU
docker cp webapp.war nginx:/home	# copying from local to container # nginx = image_name
Configure nginx Login to container cd /home cd /var cd /etc/nginx	

ls cd config.d/ cd /usr/share/nginx/html ls apt update apt install vim vim index.html (change some content in file like "Welcome to DevOps") Check in browser (InstanceIP:8080)	 # Configure html file (i.e, index.html) # Install vim if not available
Creating new image using modified container docker commit <container_ID> ngnix:modified ***Creating container from image*** docker run -id --name nginx33 -p 8095:90 <container_ID> Validate: instanceIP:8095	# creating image from container # nginx:modified = image:image_name

9) *Public and Private repositories (Docker Hub):*

Docker Hub is a cloud-based repository service for Docker images. It provides both public and private repositories.

(i) Public Repositories:

- **Free:** Anyone can access and use these repositories.
- **Large community:** A vast community of developers shares their images here.
- **Popular images:** Many widely used images are available, such as official images from Docker and popular open-source projects.
- **Best for open-source projects and sharing code:** Ideal for projects that want to be accessible to the public.

(ii) Private Repositories:

- **Restricted access:** Only authorized users can access and use these repositories.
- **Secure environment:** Offers a secure environment for storing and sharing sensitive code.
- **Best for proprietary software and internal projects:** Suitable for organizations that want to keep their code private.

Pull image from docker
To pull/push from docker hub/CLI we need to first login to docker hub in CLI, **(i)** docker login -u <dockerHub_accountName> **(ii)** docker pull <dockerHub_accountName/repository_name>:1.0 eg) docker pull siddeshg672/application:push-1.0 docker pull siddeshg672/demoprivaterepo:1.0 [Note: can't pull private repository due to no access, can pull private repo only if we have access/credentials of account as it's protected]
Push image to docker
(i) create tag for image that wanted to be pushed, docker tag <img_ID> <dockerHub_accountName/repository_name>:1.0 eg) docker tag 31dd1992778e yatishv13/test:1.0 **(ii) push image to dockerHub** docker push <dockerHub_accountName/repository_name>:1.0 eg) docker push yatishv13/test:1.0

10) Amazon Elastic Container Registry (ECR):

Fully managed Docker container registry that makes it easy to store, share, and deploy container images.

(i) Install awscli in CLI before login to repo in ECR:

- sudo apt update -y
- sudo apt install awscli
- sudo apt install unzip
- curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
- unzip awscliv2.zip
- sudo ./aws/install
- aws –version

(ii) Create IAM user for access:

Users → Create users → test → AmazonEC2FullAccess → attach policies → ECR (Select all), AdminstatorAccess → next

Go inside user → create access key → CLI → create access key

Give access in CLI,

aws configure

Give Access ID, Secret Access ID, Region

(iii) Create repository → Repository name (test) → create

(iv) Copy paste login command of ECR repo in CLI,

Go inside repository → view push commands → commands available for login to ECR

(a) CLI: login to repo from ECR

eg) aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 560388895748.dkr.ecr.us-east-1.amazonaws.com

(b) Create tag:

```
docker images                                  # get image_ID
docker tag <img_ID> <repo DNS>:1.0             # DNS can be found in repo details
docker push <repo DNS>:1.0                     # repo DNS = image name
eg) docker tag 560388895748.dkr.ecr.us-east-1.amazonaws.com/test_ecr:1.0
    docker push 560388895748.dkr.ecr.us-east-1.amazonaws.com/test_ecr:1.0
```

Delete images	
docker image prune -a	# deletes all images except running image
docker rm <img_ID>	# Delete running images
Delete containers	
docker rm $(docker ps -a -q)	# deletes all containers
docker rm <container_ID>	# Delete container
docker rm -f <container_ID>	# Delete container forecfully

11) Docker context and Docker Ignore

(i) Docker context

Folder where it can contain all the dependencies for docker file to create docker image.

(ii) Docker Ignore

File used to specify files/directories that Docker should exclude from the build context when creating a Docker image.

Create file name as .dockerignore and add files inside it so while building any image then those files will be excluded.

(i) steps inside df2 file

vim df2

COPY df1 /home/ubuntu

COPY abc.txt /home/ubuntu

Save

(ii) Mention files that needs to be ignored while building images

vim .dockerignore

abc.txt # ignores abc.txt while building as it's included in .dockerignore

Save

(iii) Build image using df2 file

docker build -t test -f df2 .

12) Multistage Docker Builds:

Multistage Docker Builds are a powerful feature that allows you to optimize your Docker images by building them in multiple stages, each with its own specific purpose. This approach helps to create smaller, more efficient images, which can lead to improved performance and security.

- To reduce the size of docker image we can use multiple from.
- Multi-stage builds are useful to anyone who has struggled to optimize Dockerfiles while keeping them easy to read and maintain.
- With multi-stage builds, you use multiple FROM statements in your Dockerfile. Each FROM instruction can use a different base, and each of them begins a new stage of the build. You can selectively copy artifacts from one stage to another, leaving behind everything you don't want in the final image.

Installing Tomcat, keeping only necessary steps and removing rest which isn't required for Tomcat	
1. Create dockerfile **vim df1** **FROM bitnami/git:latest as clone** **RUN git clone https://github.com/Siddeshg672/hello_world_public_war.git -b main**	

FROM maven:3.8.6-openjdk-11 as build **RUN mkdir test** **WORKDIR test** **COPY --from=clone hello_world_public_war/ .** **RUN ls -lart** **RUN mvn clean install** **FROM tomcat:8.0-alpine** **COPY --from=build /test/webapp/target/webapp.war /usr/local/tomcat/webapps/webapp.war** **Save**	
2. Create image from Dockerfile docker build -t test:11 . (or) docker build -t myimg:3.0 -f df3 .	# if we don't provide dockerfile name then it will check from context (.) and build from that # name of file should be "dockerfile"
3. Create container from Image docker run -id --name tomcat -p 8090:8080 test:11 Validate: browser InstanceIP:8090 InstanceIP:8090/webapp	
If want to update content in Tomcat (webapp), make changes in GitHub and commit. In CLI, stop old container after which create new image and container for updated content. (No need to change dockerfile) docker stop <container_ID> docker build --no-cache -t test:22_bugfix . docker run -id -p 8099:8080 test:bugfix Validate: browser InstanceIP:8099/webapp	 # Stop container # --no-cache = used to remove cache as previous build cache would have been stored

13) Docker volume:

- A Docker volume is an independent file system entirely managed by Docker and exists as a normal file or directory on the host, where data is persisted.
- Docker volumes allow to store data outside the container, ensuring that it persists even if the container is deleted.

Types of Volumes:

(i) Bind volumes → locally created folders

Volumes that are created by mounting a folder/directory on the host machine (local) to a container.

docker run -d --name cont_name -v /host/path:/container/path <image_ID>

eg) docker run -id --name stmulh -v /root/test:/usr/local/tomcat/webapps/app -p 9116:8080 dcc00d1e67b5

Source vol name: test

Target vol name: app

[Note:

Any changes made in source vol will be reflected in target vol and vice versa.

Even though container is deleted, volume will be present in local path (/home/ubuntu).

We can create folder and bind to that folder or if we give folder name while creating container then folder will be created in mentioned path of host and container]

(ii) Named volumes → created with docker volumes commands

Volumes that are created and managed by Docker.

docker volume create testVolume

eg) docker run -id --name stmulh -v testVolume:/usr/local/tomcat/webapps -p 9116:8080 dcc00d1e67b5

Stored in /var/lib/docker/volumes

/usr/share/nginx

(iii) anonymous volumes → default volume

Volumes that are created automatically when a container is started without specifying a volume.

eg) docker run -id --name stmul -p 8914:8080 -v /usr/local/tomcat/webapps dcc00d1e67b5

[Note: To change anything (index.jsp),

Login to container, docker exec -it <cont_ID> /bin/sh

cd /usr/local/tomcat/webapps/webapp (or) cd webapps/webapp]

docker volume ls	# List Volumes
docker volume inspect <vol_name>	# Inspect Volume
docker volume rm <vol_name>	# Remove Volume
docker system df	# Check total no of images, containers, volumes
docker builder prune	# Remove all build cache

14) Docker networks:

- Key feature that allows Docker containers to communicate with each other and the outside world.
- They provide flexibility in how containers are connected and isolated.

➔ Types of Docker Networks:

(i) Bridge: default network, used when application run in isolation containers that need to be communicated

(ii) Host: directly created in the system network which will remove isolation between the docker host and containers.

(iii) none: disables all the network part

(iv) Overlay: Used for multi-host communication

(v) ipvlan: Users have complete control over both IPv4 and IPv6

(vi) macvlan: this allows to assign the mac address to container

Establishing communication b/w containers using bridge network	
sudo su (root user)	
(i) Create network	
docker network create -d bridge --subnet 172.18.0.0/16 --gateway 172.18.0.1 bridge_test	# Create network # subnet value can be anything
(ii) docker network ls	# lists networks
(iii) Creating 2 container with same network (Assigning network)	
docker run -id --name tomcat -p 8081:8080 -v test:/usr/local/tomcat/webapps --network <network_ID>	# ping <cont_name/ID> # can see establishing communication between both containers as both cont. are in same network having same IP range
(iv) Login to any container	
docker exec -it <container_ID> /bin/sh	
(v) ping tomcat	

Creating containers without network	
(i) Create container without network docker run -id -name tomcat1 -p 8082:8080 -v test:/usr/local/tomcat/webapps <image_ID> **(ii) Login to container** docker exec -it <container_ID> /bin/bash **(iii)** ping tomcat1	# we can see there is no communication between containers if we don't assign network as it will assign default network having different IP
Establishing communication between 2 containers having different network	
sudo su (root user) **(i) <u>Create bridge network</u>** docker network create -d bridge –subnet 172.30.0.0/16 --gateway 172.30.0.1 bridge_test1 **(ii) <u>List network for ID</u>** docker network ls **(iii) <u>Create containers</u>** docker run -id -name tomcat5 -p 8083:8080 -v test:/usr/local/tomcat/webapps --network bridge_test1 <network_ID> **(iv) <u>Login to container</u>** docker exec -it <container_ID> /bin/bash ping tomcat5 exit **(v) <u>connect container having different network into one network</u>** docker network connect bridge_test tomcat5 **(vi)** Login to container ping tomcat5	# didn't connect as containers are in different networks # Connect network and container (bridge_test = network_name tomcat5 = container_name) # can see communication between 2 containers having 2 different networks
docker network ls	# list networks
docker inspect <network_ID>	# Inspect network
docker network disconnect bridge_test tomcat5	# Disconnect network

Host network	
docker run --network="host" <image_name> <container_name/ID>	# when we 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.

15) Docker Compose:

Docker Compose is a tool that allows you to define and manage multi-container Docker applications. Using a YAML file (docker-compose.yml), you can configure your application's services, networks, and volumes, and run everything with a single command. It simplifies orchestrating multiple containers that work together, such as a web server, a database, and other services, within the same project.

- Tool for defining and running multi-container docker applications
- Use yaml files to configure applications services (docker-compose.yml)
- Can start all services with a single command: docker compose up
- Can stop all services with a single command: docker compose down
- Can scale up selected services when required

Step 1 : install docker compose

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose

docker-compose --version
```

Step 2 : Create docker compose file at any location on your system

```
mkdir DockerComposeFile
cd DockerComposeFile
touch docker-compose.yml
vi docker-compose.yml
    version: '3'
     services:
       web:
         image: nginx
         ports:
         -  9090:80
```

database: image: redis Save
Step 3 : Check the validity of file by command docker-compose config
Step 4 : Run docker-compose.yml file by command docker-compose up -d docker ps instanceIP:9090
Steps 5 : Bring down and up application by command docker-compose down docker-compose up
[Note: docker-compose up -d --scale database=4 # 4 databases containers will be created (scale-up)]

16) Best practise to create images and containers

(i) Image Creation:

- Base image: Choose a minimal base image to reduce image size. Consider the operating system (e.g., Alpine Linux, Ubuntu) for application. Use official images from trusted repositories like Docker Hub.
- Layers: Minimize layers and use multi-stage builds.
- Files: Remove unnecessary files.
- Configuration: Use environment variables to configure your application dynamically.
- Security: Follow security best practices.
- Run as non-Root
- Update Base Images Regularly
- Use Trusted Sources
- Scan Images: Regularly scan your images for vulnerabilities using tools like Docker's built-in scanning, Snyk, or Clair.

(ii) Container Creation:

- Tags: Use clear and descriptive tags.
- Containerization: Run each service in its own container.
- Volumes: Use volumes for data persistence.
- Network: Define a clear network topology.
- Restriction of resources: Set appropriate resource limits (CPU, memory) for each container.
- Logging and Monitoring:
  Configure logging for your containers to track and troubleshoot issues.
  Use monitoring tools to collect metrics and monitor container health.

➔ *Additional Tips:*

- Docker Compose: Use Docker Compose for multi-container applications.
- CI/CD: Automate image building, testing, and deployment.
- Optimization: Tailor image and container creation to your application's needs.

17) Real Time Challenges in Docker?

- Docker is Single daemon process. Which can cause single point of failure. If the Docker daemon goes down for some reason all the applications running inside that docker will become down.
  The Above Problem Can be Eliminated by using Podman or Builda.
- Docker Daemon runs as a root user. Which is Security Threat. Any process running as root can have adverse effects. When it comprised to security reasons, It can impact Other applications or Container on Host.
- Resource Constraint if you are running too many containers on a single host you may experience issues with resource Constraints. This can Results in Slow Performance or Applications might Crashes.

18) Steps to Take to Secure our Containers?

- Use Distroless Images or Images with not many Packages as your final image in multi-stage build, so that there is less chance of Security Issues.
- Ensure that networking is Configured Properly. This is one of the most common reasons for security issues. If required configure Custom Bridge network and attach to a container To Isolate the Containers.
- Use Utilities like sync or Use Image Scanning tool like Trivy to scan your Container Images for checking any Vulnerabilities are there or not before Pushing to the Production/Staging Environment.

19)Files and Folders in containers base images

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

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

(iii) ***/etc:*** contains configuration files for various system services.

(iv) ***/lib:*** contains library files that are used by the binary executables.

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

(vi) ***/var:*** contains variable data, such as log files, spool files, and temporary files.

(vii) ***/root:*** is the home directory of the root user.

Summary

1) ***Docker:*** Docker is a containerisation platform which packages our application and all its dependencies together in the form of container.
2) ***Docker file:*** Text file that contains a set of instructions which are used by docker build for building a custom docker image.da
3) ***Docker images:*** Docker images contain binaries/libraries which are necessary for one software application. (blueprints for creating containers)

   Images can be created in 3 types:
   - Using docker file
   - Pulling image directly from docker hub
   - Creating Image using container (docker commit)
4) ***Docker Containers:*** 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.
5) ***Dangling images:*** Dangling images in Docker are essentially unused images that have no tags associated with them. They are often created when you rebuild an image with the same tag, causing the old image to lose its tag and become "dangling".
6) ***Multistage Docker Builds:*** Multistage Docker Builds are a powerful feature that allows you to optimize your Docker images by building them in multiple stages, each with its own specific purpose.
   - This approach helps to create smaller, more efficient images, which can lead to improved performance and security.

7) ***Docker Volumes:*** used to store data outside container ensuring that it persists even of the container is deleted.

- Anonymous vol are default volume but it gets deleted once container is deleted.
- We can either use Bind volume (create folder and assign to container as volume)

(or)

- Named/normal volume (create volume using docker: docker volume create <vol_name> which will be managed by docker.

8) ***Docker Network:*** used to communicate with containers and the outside world.

Bridge network is default network and only network where we can customize by giving required subnet range. It's most secure network and mostly used.

9) ***Public and Private repositories (Docker Hub):***

Public repo: Anyone can access and use these repositories.

Private repo: Only authorized users can access and use these repositories.

10) ***Amazon Elastic Container Registry (ECR):*** Fully managed Docker container registry that makes it easy to store, share, and deploy container images.

11) ***Docker context:*** Folder where it can contain all the dependencies for docker file to create docker image.

12) ***Docker ignore:*** file used to specify files or directories that Docker should exclude from the build context when creating a Docker image.

13) ***Docker Compose:*** Docker Compose is a tool that allows you to define and manage multi-container Docker applications. Using a YAML file (docker-compose.yml), you can configure your application's services, networks, and volumes, and run everything with a single command. It simplifies orchestrating multiple containers that work together, such as a web server, a database, and other services, within the same project.

14) ***Entrypoint:*** Sets default parameters that cannot be overridden while executing Docker containers with CLI parameters.

CMD: Sets default parameters that can be overridden from the Docker command line interface (CLI) while running a Docker container.