DOCKER

SESSION - 42 24-FEB-2025

- Physical Server vs VM vs Container
- Old enterprise vs Monolithic vs Microservices
- Install Docker
- Docker commands

Monolithic vs Microservices

Monolithic - is a more traditional approach to encommerce setup where the entire application is built as a single unit.

→ This can make the development & deployment more straightforward, but it also makes the application slower & more challenging to scale.

→ A monolithic application is more complex to update & maintain than one using microservices. Which is one of the biggest disadvantages of monolithic architecture. Additionally, it is an almost impossible task to scale individual parts of such an application b/c all components are bundled together.

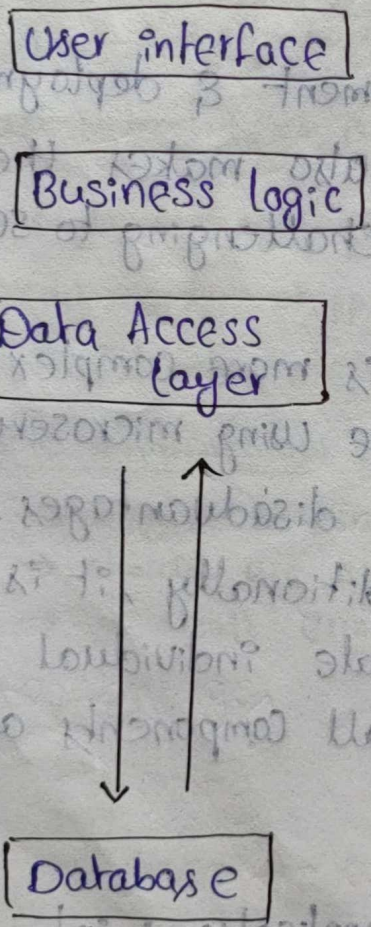
Microservices - is to split up applications into small, self-contained services that can be deployed & updated seperately from the rest of the system. This allows for more speed, flexibility, & scalability as they can be deployed on different servers &

scaled independently.

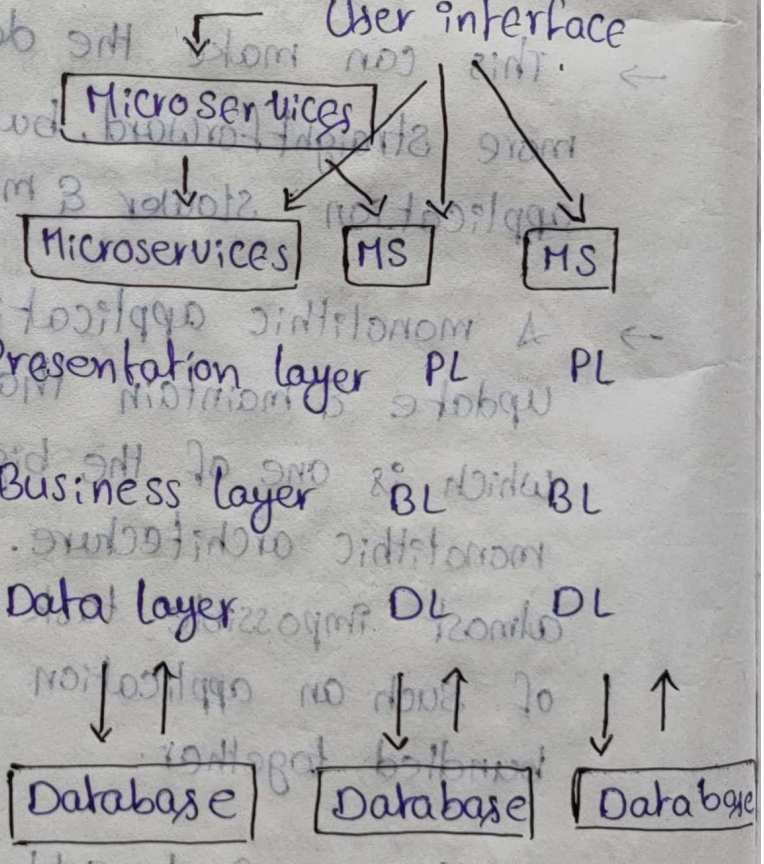
→ Microservice-style applications communicate directly via APIs without going through centralized points. So instead of your app having many different types & size of services, each component will only need lightweight libries provided by their respective vendor frameworks.

Architecture -

Monolithic



Microservices



Monolithic -

- Advantages

→ Robust all-in-one system with many featured included

→ Simplified development & deployment with a single, unified codebase

→ A consistent development approach can improve collaboration & code quality.

- Disadvantages

→ Hard to update & maintain; blocks specific component upgrades.

→ Limited to one technology, making integration with other apps or APIs challenging.

→ Vertical scaling is difficult & expensive, leading to costly development cycles.

→ An entire application can be affected by issue in any part of the code.

→ Challenging to add new features post-launch as all changes must go through the codebase.

→ Tightly linked system makes it difficult to scale & diversify for new market or product lines.

Microservices -

- Advantages.

→ Small, independent components for faster development & high scalability.

→ API-first approach simplifies third party integration, scaling, and independent testing.

→ Each microservices has it's own database, maintaining data integrity & independence

→ Scalable by deploying individual services, supporting rapid business growth.

→ Ability to use best-of-breed technology for each microservice. Easier to manage & scale services.

→ Developers can focus on individual microservices, speeding up development.

→ Focused on single functions leads to higher performance & faster speed.

→ issues with one microservices won't affect the entire app.

- Disadvantages.

→ Debugging is more complex due to the distributed nature of the system.

→ More sophisticated setup needed for configuring & managing multiple services.

Old Enterprise (10 years Back)

frontend + Backend
HTML + CSS + JS + JSP + servlets

→ app size is very high,
→ A small change in any frontend or backend should be released & redeployed

release notes
↓
dev, qa, uat, pre-prod
↓
Client approval
↓
one entire day for deployment
↓
sanity testing (checking all changes are proper or not)

frontend seperate & Backend seperate

→ with the help of API we connect them
✓ No dependency
✓ load on servers are decreased

Angular JS
Monolithic restful API/services
HTTP Methods (get, push) & responds
} frontend

backend team, single component → backend
[user, cart, order, shipping, payment, delivery, catalogue, reviews, recomendations, etc...]

Monolithic applications

→ single backend component
→ should use single programming language
→ a small error can make entire website down

Microservices

User
cart
order
:

→ different components use diff languages

Ex - manabadi/eenadu → emacet results API
Java → NodeJS
client & server can use any programming language
easy deployment
website works if any component goes down

Joint family vs Small family vs Individual

⇒ independent house to host joint family → application size is big → old enterprise
physical serverver → dedicated server
OS → Hardware

⇒ flats → apartments

⇒ single person → PG, shared room

Physical server / Joint family

disadvantages -
- Costly
- Waste of resources → may not use all ram & HD
- time → purchase, installation, configuration
- maintanance → water, electricity, plumbing → OS, network, etc...

advantages -
- Complete privacy → single application

VM / small family

advantages -
- time is less to construct
- less cost
- proper resource utilisation
- less maintanance

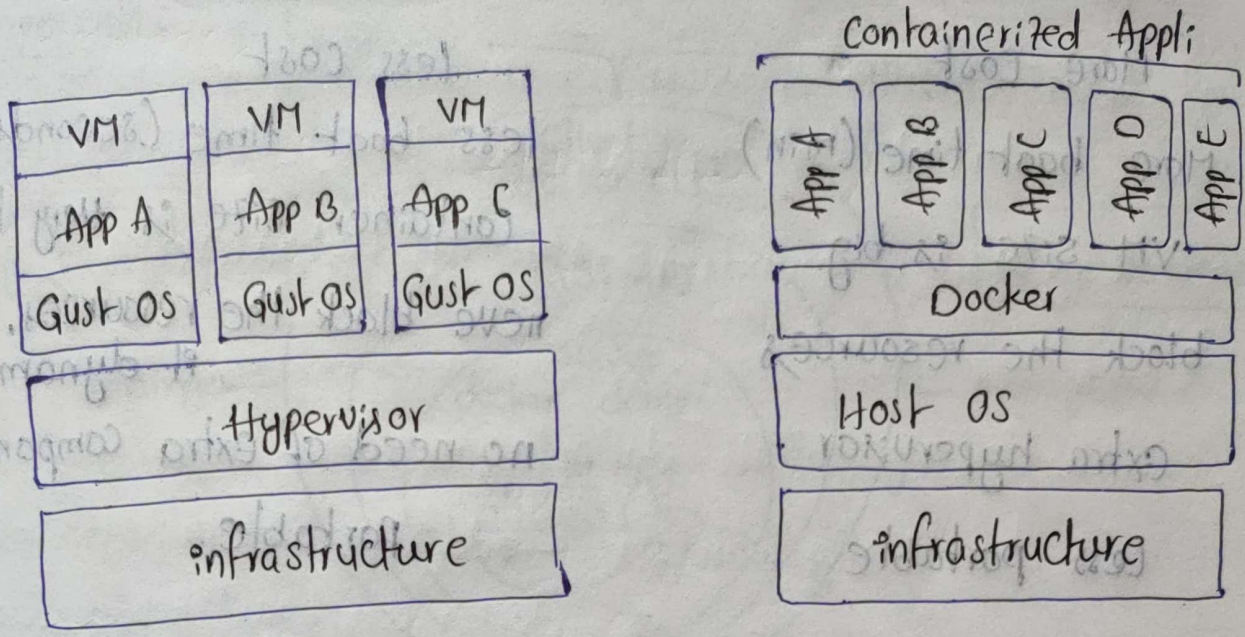
disadvantages -
- less privacy

Container / shared room

advantages -
- less cost
- time is very less
- no maintance / perfect
- perfect resource utilisation

disadvantage -
- very less security

VMs vs Containers



Virtual Machine - VM is a system which acts exactly like a computer. VMs are like physical hardware turning one server into many servers. Which it uses technologies called hypervisor which allows multiple VMs to run on single machine.

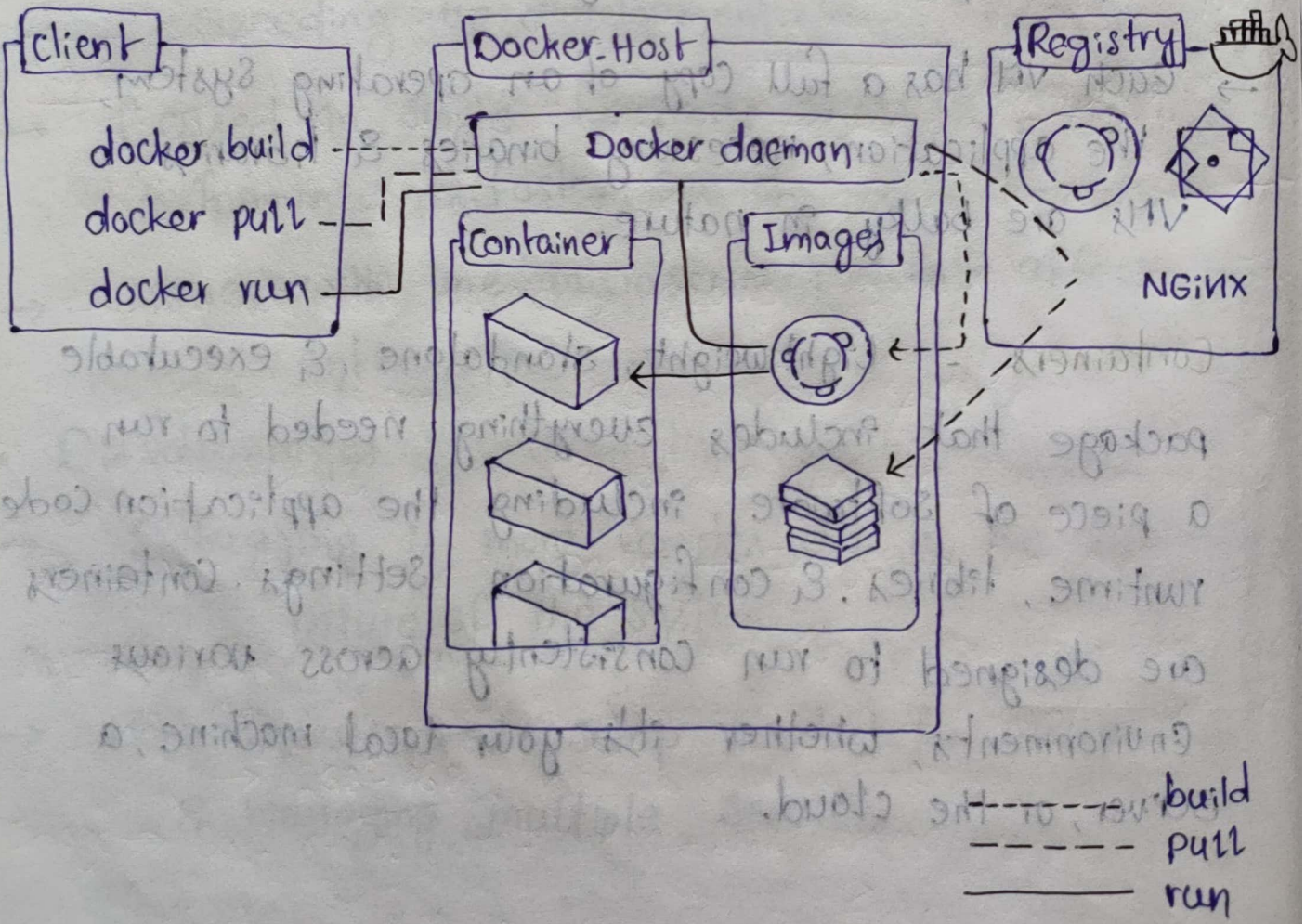
→ Each VM has a full copy of an operating system, the application, necessary binaries & libraries. VMs are bulky in nature.

Containers - Lightweight, standalone, & executable package that includes everything needed to run a piece of software, including the application code, runtime, libries. & configuration settings. Containers are designed to run consistently across various environments, whether it's your local machine, a server, or the cloud.

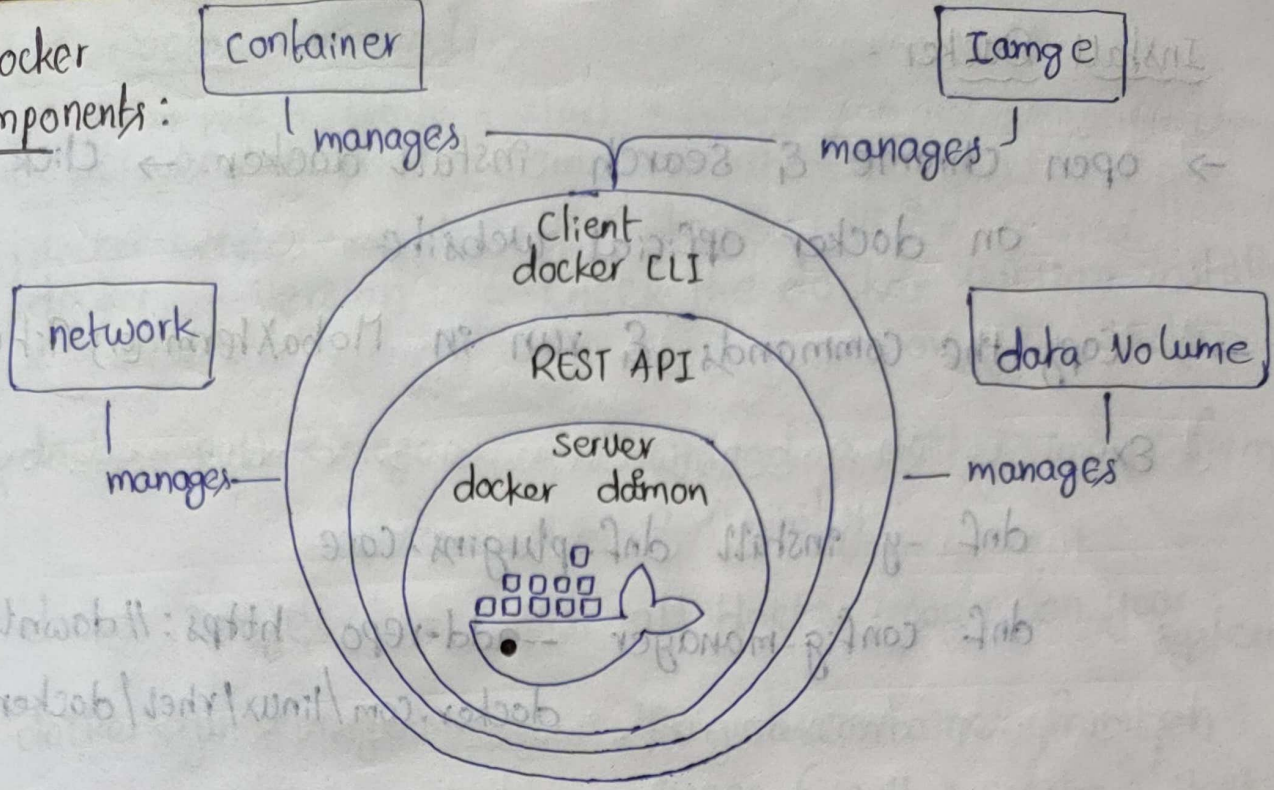
VMs	containers
More cost	less cost
More boot time (min)	less boot time (seconds)
VM size is big	container size is very less
block the resources	neve block the resources, use it dynamically
extra hypervisor	no need of extra components
less portable	portable

Docker - Docker is a platform that simplifies building, shipping, and Running applications by packaging them with all their dependencies into portable, isolated containers.

Architecture -



Docker Components:



Docker daemon - The docker daemon (docked) listens for docker API requests & manages Docker objects such as images, containers, networks, & volumes

→ a daemon can also communicate with other daemon to manage docker services.

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

Docker Registry - A docker registry stores docker images. Docker Hub is a public registry that anyone can use

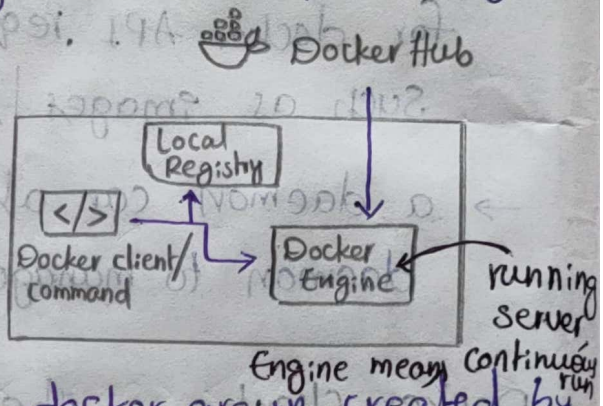
Install Docker -

→ open chrome & search install docker → click on docker official website

→ Copy the commands & run in MobaXterm (or) GitBash

Ex -

```
dnf -y install dnf-plugins-core
dnf config-manager --add-repo https://download.docker.com/linux/rhel/docker-ce.repo
dnf install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
systemctl start docker
systemctl enable docker
systemctl status docker
```



Note -

After installing docker → docker group created by default (they only run the docker command).

→ Now we add secondary group (ec2-user)

```
sudo usermod -aG docker ec2-user
```

exit → logout & login (after we can run commands)

VM → AMI → instance || Docker → Image → Container

AMI ⇒ Base OS + Application runtime + User creation + folder creation + Config files + app code + dependency install + start/restart app

→ 2GB - 4GB (it will remove unwanted packages)

Image ⇒ Bare min OS (10MB - 500MB) + Applic runtime + User creation + folder creation + config files + app code + dependency installation + start/restart app → 150 - 500MB

Imp Docker Commands

✓ docker pull + create + start = docker run -d <image-name>

- **Basic commands -**

✓② docker create <image> - Container will be created

docker --version - Check the docker version installed on your system

✓① docker pull <image> - Download a docker image from a registry

✓ docker images - List all docker images on your system

docker run <image> - Run a container from an image (pull + create + start)

Process Status

✓ docker PS - List all running containers

✓ docker ps -a - List all containers, including stopped ones.

✓ docker images -a -q - List all images id's

- **Container Management -**

✓ docker rm -f `docker ps -a -q` - remove all containers

docker stop <container_id> - stop a running container.

✓③ docker start <container_id> - start a stopped container.

docker restart <container_id> - Restart a running container

✓ docker rm <container_id> - Remove stopped container

✓ docker rm -f <container_id> - forcibly remove a docker container

execute

docker exec -it <container_id> <command> - Run a command inside a running container (e.g., docker exec -it <container_id> bash)

✓ docker logs <container_id> - fetch the logs of container

✓ docker inspect <container_id> - Display detailed information about container

- Image Management -

docker build -t <image_name> - Built a docker image from a dockerfile in current directory

✓ docker rmi <image_id> (or) <image-name> - Remove a Docker image

docker tag <image_id> <repository>/<image_name>:<tag>
tag an image for easier reference
(Eg: docker tag 123abc myrepo/myim:latest)

docker push <repository>/<image_name>:<tag>
push an image to a docker registry

✓ docker rmi `docker images -a -q` - remove all images

- Networking -

docker network ls - list all docker networks

docker network create <network_name> - create a new docker network

docker network connect <network_name> <container_id> :
Connect a container to a network

docker network disconnect <network_name> <container_id>
Disconnect a container from a network.

- Volume Management -

docker volume ls - List all docker volumes

docker volume create <volume_name> - Create a new docker volume

docker volume rm <volume_name> - Remove a docker volume.

docker run -v <volume_name> : /path/in/container <image> - Mount a volume to a container (e.g., docker run -v myvolume:/data nginx)

- Docker Compose Commands -

docker-compose up - Start all services defined in a docker-compose.yml file.

docker-compose down - Stop ~~all~~ and remove all services defined in a docker-compose.yml file

docker-compose ps - List all containers managed by docker compose.

docker-compose build - Build or rebuild services defined in a docker-compose.yml file.

~~docker volume inspect~~

docker run -d -p 80:80 --name my_nginx nginx

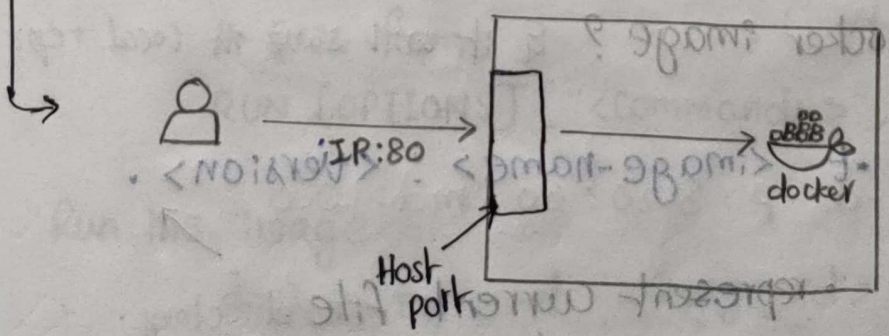
(Host port : container port)

(to run nginx with specified name)

docker exec -it my_nginx bash (login in to nginx with name)

(or) <Container-id>

(interactive terminal)



SESSION-43 25-FEB-2025

- Dockerfile instructions

FROM	ADD
RUN	LABEL
CMD	EXPOSE
COPY	~~ADD~~ ENV

/ How to create custom docker images?

Creating a custom Docker images involves writing a dockerfile & then building the image using Dockerfile.

Dockerfile: used to create our custom images using instructions provided by docker.

NOTE -

RHEL9 == Centos 9 == Almalinux9

Dockerfile reference

- FROM

The FROM instruction initializes a new build stage & sets the base image for subsequent instructions.

→ A valid Dockerfile must start with a FROM instruction & FROM should be the first instruction in Dockerfile, that represent Base OS.

Syntax -

```
FROM <BASE-OS> : <VERSION/tag>
```

how to built docker image? & it will save in local repo

```
docker build -t <image-name> : <version> .
```

. represent current ~~file~~ directory

```
docker images
```

Every server has OS (linux, Red hat, windows) is mandatory, & same every container also have a OS

how to pushing a Docker image to Docker Hub

→ login in to dockerhub

docker login -u <user-name>

→ Before pushing your Docker image, you need to tag it with your Docker Hub repository name.

docker tag <my-custom-image>:<version> <DockerHub Username>/<repo-name>:<tag>.

Example-

docker tag from:1.0.0 joindevops/from:1.0.0

docker push <DockerHubUsername>/<repo-name>:<tag>

- RUN

The RUN instruction allows you to execute commands inside the container during the image build process.

→ it's commonly used to install software packages, setup configurations, and make other modifications to the image.

Syntax-

Shell form:
RUN [OPTIONS] <commands> ...
Exec form:
RUN [OPTIONS] ["<commands>", ...]

Run the image -

docker run -d <image-id> # status Exited

docker run -d <image-id> sleep 10 # it runs 10 sec

- **CMD**

The CMD instruction sets the command to be executed when running a container from an image.

→ Unlike the RUN instruction, which executed commands during the build process to create the image.

Note -

Systemctl will not work in containers.

/etc/systemd/system/*.service

→ Systemd is full os functionality but container is not full OS

Syntax -

```
CMD ["executable", "param1", "param2"]
```

Example -

```
CMD ["nginx", "-g", "daemon off;"]
```

-daemon off = running in the foreground instead of as a background process.

-g = set global directives.

RUN vs CMD

RUN instruction executes at the time of image building

CMD instruction executes at the time of container creation.

✓ Attach port & run this.

Example -

```
docker run -d -p 8080:80 cmd:1.0.0
```

- **COPY**

The copy instruction in a Dockerfile is used to copy files & directories from your local filesystem into the Docker image.

→ Syntax -

```
COPY <source> <destination>
```

- **ADD**

The ADD instruction in a Dockerfile is similar to COPY but with some additional features.

→ it not only copies files & directories from your local filesystem into the Docker image but also can-

1. Automatically extract compressed files (like .tar, .gz, etc.)
2. Copy files from URL. (copying file directly from internet to image)

- **LABEL**

The LABEL instruction is used to add metadata to an image.

→ Metadata can include information like the maintainer's name, the version of the image, description, & other custom data. This info can be useful for documentation & management purpose.

Syntax -

```
LABEL <key> = <value>
```

to add multiple label

LABEL <key> = <value> \
<key> = <value> \
<key> = <value>

✓ docker images -f "LABEL=<key>=<value>"

- EXPOSE

The EXPOSE <instruction informs Docker that the container listens on the specified network ports at runtime.

Syntax :

EXPOSE <port>

- ENV

The ENV instruction in a Dockerfile is used to set environment variables that can be accessed by the running containers.

Syntax -

#same as LABEL
ENV <key> <value>

NOTE

The total number of available ports are both TCP & UDP is 65,535.

→ This number is derived from the 16-bit address space, which allows for $2^{16} = 65,536$ possible port numbers.

→ But port 0 is reserved & not used, leaving 65,536 usable ports.

Ports are categorized into three ranges:

- Well-known ports (0-1023): These ports are reserved for well-known services & portocols such as HTTP (port 80), HTTPS (port 443), & FTP (Port 21)
- Registered ports (1024-49151): These are available for specific applications & services to use.
- Dynamic/Private ports (49152-65535): These are used for private or temporary purpose & are not officially registered.

SESSION-44 26-FEB-2025

- Dockerfile instruction

ENTRYPOINT
USER
WORKDIR
ARG

- ENTRYPOINT

The ENTRYPOINT instruction in a Dockerfile specifies the command that will run when the container starts. it is similar to CMD.

→ but key difference: ENTRYPOIN sets the main command, which can be augmented with addition arguments specified via docker run command.

Syntax -

ENTRYPOIN ["executable", "param1", param2"]

CMD vs ENTRYPOINT

→ CMD can be overridden at runtime by providing arguments to docker run

→ ENTRYPOINT can't be override (if you try to override entrypoint it will not override, but it will append

→ for Best result we can use CMD & ENTRYPOINT together.
- we can mention Command in ENTRYPOINT, default options / inputs can be supplied through CMD. User can always override default options. (ex- ping)

→ Only one CMD & one ENTRPOINT should be used in Dockerfile

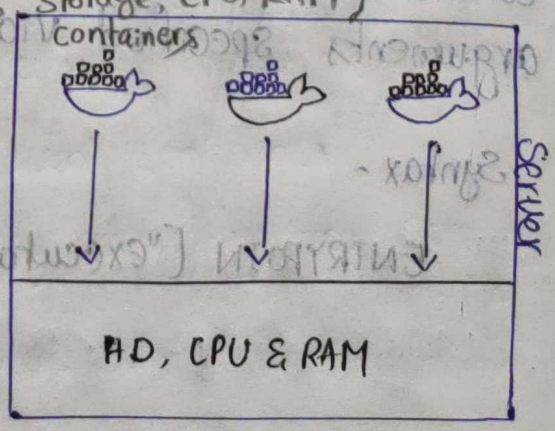
- **USER**

The USER instruction in a Dockerfile is used to specify the user that the container should run as. By ~~default~~.

→ By default, Docker containers run as the root user, which has full administrative privilegs.

→ However, for security reasons, it's often a good practice to run containers as a non-root user.

(all containers use server hardware b/c of they don't have seperate storage, CPU, RAM)



syntax -

```
RUN useradd <user-name>
USER <user-name>
```

Note - Atleast before running last instruction, run your container with non-root user.

- **WORKDIR** (cd is not worked in docker, we use WORKDIR & set the path)

The WORKDIR instruction in a Dockerfile sets the working directory for any subsequent RUN, CMD, ENTRYPOINT, COPY, and ADD instructions. if the directory doesn't exist, it will be created automatically.

Syntax - WORKDIR /path/to/workdir

Note - if not specified, the default working directory is /

- **ARG**

The ARG instruction is used to define variable that can be passed at build-time to the Docker image.

Syntax - ARG key=value

How to access ARG variable within container as a Environment variable.?
set the ARG value to ENV variable inside Dockerfile.
Ex- ARG USERNAME=Cherry
ENV USERNAME=$USERNAME

Example -

```
ARG USERNAME=Cherry
RUN echo "Hello User": ${USERNAME}"
```

Bash.

```
docker build -t arg:1.0.0 -progress=plain --no-cache.
```

↑ it will help to don't trunk.

```
docker build -t arg:1.0.0 -progress=plain --no-cache
             --buid-arg USERNAME=new_user.
```

↑ to override username.

ARG vs ENV

→ ENV var can be access at the image building & in container also

✗ → ARG var are only accessed inside image build, not in container

✓ → ARG can be the first instruction only to provide the version for base image. it can't be useful after FROM instruction

Ex- ARG version
FROM almalinux:${version:-9} #default is 9

SESSION - 45 — 27-FEB-2025

- Dockerfile instruction
  - ONBUILD
- Expense project using docker

Previous class revision -

FROM - Should be the first instruction to represent the base OS.

RUN - Used to configure or install packages.

CMD - executes at the time of container creation, this command should keep container running infinite times.

COPY - Copy from local workspace to image

ADD - Same as copy but 2 extra capabilities, directly download from internet or untar directly.

(information purpose)

LABEL - adds metadata, used for filter. key value pair

EXPOSE - doc purpose, tell the users about ports opend by container

ENV - sets the env variables in the container, we can use at build time also

ENTRYPOINT - can't override, CMD can provide default args, we can always override default args

USER - set the user to run container

WORKDIR - set the working directory for container/image

ARG - ~~built~~ build time variables, in an exceptional case can be first instruction to supply base OS version

How to increase root size in docker ?

docker.sh

increase size

```
growpart /dev/nvme0n1 4    # increase 50 GB
lvextend -l +50%FREE /dev/RootVG/rootVol
lvextend -l +50%FREE /dev/RootVG/varVol
xfs_growfs /               # Docker directory is here that's why
xfs_growfs /var
```

install docker

```
dnf -y install dnf-plugins-core
dnf config-manager --add-repo https://download.
    docker.com/linux/rhel/docker-ce.repo
dnf install docker-ce docker-ce-cli containerd.io
    docker-buildx-plugin docker-compose-plugin -y
systemctl start docker
systemctl enable docker
usermod -aG docker ec2-user
```

lsblk - Check the partition

ONBUILD -

The ONBUILD instruction is used to set up triggers that execute when the image is used as the base for another image. it's particularly useful for creating base images that can include certain instructions to run in child images.

We can set some conditions when user is using our image

nginx → alamlinux:9, install nginx, remove index.html

We will force the users to keep index.html in their workspace for mandatory.

Expense Project Using docker.

expense-docker
- scripts
  - backend.sql
- Dockerfile

Dockerfile.

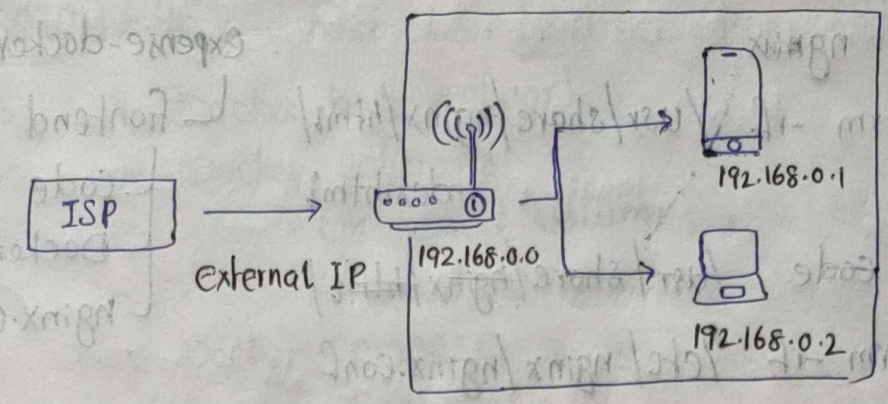
```
#download mysql image
FROM mysql:8.0
# set mysql Root-Password with ENV
ENV MYSQL_ROOT_PASSWORD=ExpenseApp@1
# initializing a fresh instance
COPY scripts/*.sql /docker-entrypoint-initdb.d
```

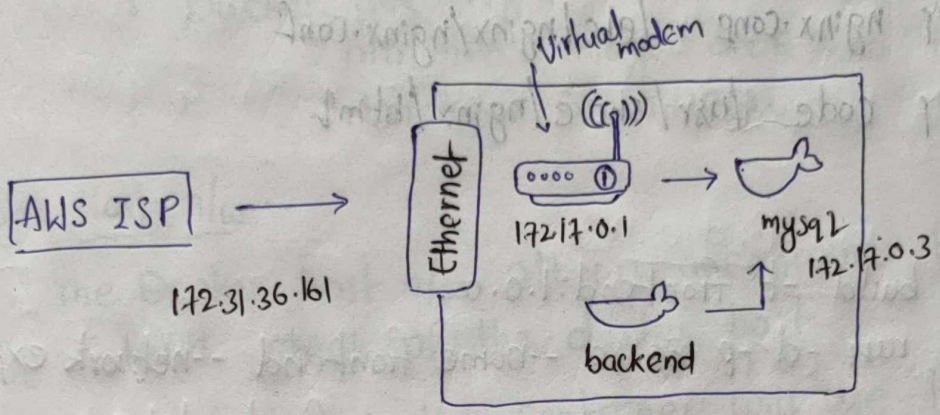
```
docker built -t mysql:1.0.0 .
docker run -d --name mysql mysql:1.0.0
docker ps
docker exec -it mysql bash  #Enter into container
        #mysql -u root -pExpenseApp@1
```

Expense-docker
- backend
  - DbConfig.js
  - Dockerfile
  - index.js
  - package.json
  - Transaction Service.js

Dockerfile

```
FROM node:20
WORKDIR /opt/backend
COPY package.json .
COPY *.js .
RUN # installing dependensis
RUN npm install
ENV DB_HOST="mysql"
CMD ["node","index.js"]
```





Note: if you are using default n/w docker containers can't communicate with each other.

Creating our own network -

```
docker network
docker network create <network-name>  # Ex-expense
docker network ls     # there are two n/w bridge & Host
                        where bridge is default.

docker network disconnect
docker network disconnect bridge mysql
docker network connect expense mysql
docker inspect mysql

#inside the container
apt-get install telnet -y
apt-get update

telnet mysql <port-no>
                3306
```

```
FROM nginx
RUN rm -rf /usr/share/nginx/html/index.html
COPY code /usr/share/nginx/html/
RUN rm -rf /etc/nginx/nginx.conf
COPY nginx.conf /etc/nginx/nginx.conf
COPY code /usr/share/nginx/html
```

expense-docker
- frontend
  - code
  - Dockerfile
  - nginx.conf

```
docker build -t frontend:1.0.0 .
docker run -d -p 80:80 --name frontend --network expense
                                    frontend:1.0.0
```

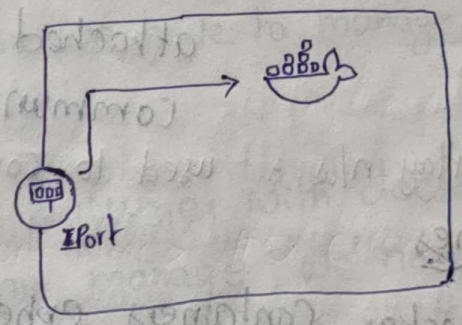
SESSION - 46 28-FEB-2025

- Docker host network
- Docker volumes
  - On named volumes
  - Named volumes
- Docker compose
- Docker Security
  - minimize image size
  - Non root containers

Docker Host n/w -

The Docker host n/w allows a container to share the network stack of the docker host.

→ This means that the container will use the host's ip address & can access the host's network interfaces directly, without any isolation.



Commands -

docker network ls

docker run -d --name mysql --network host mysql:1.0.0

netstat -lntp

netstat command for monitoring n/w connection & listing

-lntp

-l : Show only listening sockets

-n : Show numerical address instead of resolving hostnames

-t : Display TCP connections

-p : Show the process ID (PID) & name of the program

Backend - ENV DB_HOST = "localhost"

frontend - location /api/ {

proxy_pass http://localhost:8080/;

}

Note:

Host n/w in Docker it reduce n/w isolation, expose all host ports, leading to potential security risks & port conflicts

Host n/w vs Bridge n/w

Host n/w - Containers will use same host ip & host n/w

Bridge n/w - When we install docker it will create a virtual bridge n/w, containers will be attached to the bridge n/w, we can communicate b/w the containers

[overlay n/w - used to communicate with multiple-hosts] Used in K8

Volumes (temporary)

docker containers ephemeral, once you remove container by default it removes the data too.

→ you can create one directory in host & map it to container. even we delete container data will not be lost, you can remount it.

Ex- in docker, when a Mysql container is removed, all it's data is lost b/c containers are ephemeral by default. that's why docker volumes exit - to provide persistent storage the survies containers restart & deletions

Ex- docker run -d -P 80:80 nginx

docker ps → container id

docker exec -it 8ed bash

/# cd /usr/share/nginx/html/

echo "<h1> " " " </h1>" > index.html

docker rm -f 8ed

run it again & if will don't show data b/c of it is removed

inspect to docker & copy ("MergedDir" location)

docker inspect <container_id>

cd merged/
cd usr/share/nginx/html/

Creating one directory in host & map it to container. even we delete container data, will not be lost, you can remount it

cd
mkdir nginx-data # <directory-name>

→ [docker run -d -p 80:80 -v <host full path>:<container path>]
volume

docker run -d -p 80:80 -v /home/ec2-user/nginx-data:/usr/share/nginx/html nginx

container id
docker exec -it 147 bash
#

we are saying to container this /usr/share/nginx/html can don't map with your own data, map with what i created directory in host

- un named volumes

  we created directory, so we have to manage it.

- docker volumes

  you have to create volumes with docker commands, so docker can manage, we no need to worry of creation & managing.

  commands -

  docker volume
  docker volume create <volume-name>
  docker volume ls
  docker inspect <volume-name> # where we can see the "Mountpoint", in this path the volume will store

  docker run -d -v <volume-name>:<container-path> -p 80:80 nginx

Where stores the Container data ?
sudo su -
cd /var/lib/docker/
ls -l

Note - Named Docker volumes are preferred over anonymous (un-named) volumes b/c Docker automatically manages them, making storage more persistent & predictable.

Disadvantages of Manually running Containers

1. We need to know the dependency
2. We need to make sure network creation & volume creation.
3. While removing we need to remove in anti dependency order...
4. Manually run docker, run commands.

Docker Compose - is used to manage dependency b/w containers or services or applications, you can start or stop at a time with single command. & also it will automatically create n/w & volumes.

(or)

Docker compose is a tool that helps you define and manage multi-container Docker applications. With Compose, you can use a YAML file to configure your application's services, n/w & volumes, making it easy to set up and manage complex containerized environments.

Commands -

```
                    # we need to create images first & run them
docker compose up -d
docker compose down
```

Whatever that we run commands manually, kept them in a file called shell-scripting

Whatever that we run Ansible ad-hoc commands manually, kept them in a file called playbook.

Whatever that we give in docker run commands, & kept them in a file called docker compose.

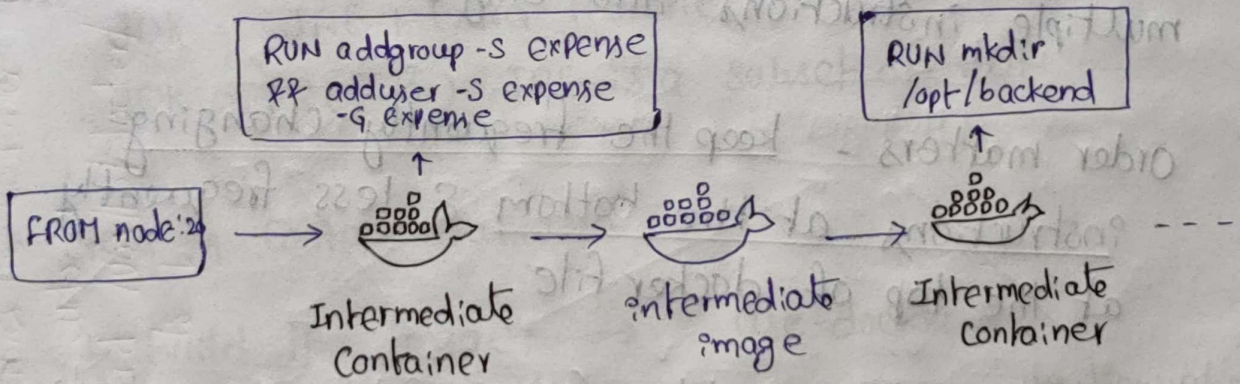
SESSION-47 03-MAR-2025

- Docker layers & optimisation
- Multi stage builds
- Disadvantages of docker

Create Dockerfile
build image
push to docker hub
docker-compose.yaml
image : joindevops/mysql:1.0.0

Image layer -

Ex- FROM node:20.18.3-alpine3.21
RUN addgroup -S expense && adduser -S expense -G expense
RUN mkdir /opt/backend
RUN chown -R expense:expense /opt/backend
WORKDIR /opt/backend



→ image layer is a change or an intermediate image created by each command specified in Dockerfile, such as FROM, RUN, COPY, etc. Each command modifies ~~specified in a~~ the privious image.

Docker maintains the images as layer, each and every instruction is one layer. docker creates.

1. intermediate container from instruction-1
2. docker runs 2nd instruction on top of an IC-1, then docker saves this another layer, IC-1 remov
3. docker saves this container as another image layer. create intermediate container out of it IC-2
4. Now docker runs 3rd instruction in IC-2 container. docker saves this as another layer, IC-2 is removed
5. docker create intermediate container from ~~the~~ this layer as IC-3.

Q. What happend when you run docker built, how the layer works?

RUN <command> &&
<command> &&
<command>

How do you optimise docker layer? ex-

1. less no.of layers faster builds, b/c no.of intermediate containers are less you can club multiple instructions into single instruction
2. order matters - keep the frequently changing instructions at the bottom & less frequently at the top of docker file
3. Use Multi-stage Builds (More useful for Java application)
4. Use Minimal Base image

Multi-stage builds are primarily used to create smaller, more optimized containers images by separating the build environment from the runtime environment. We can have one as builder one as runner, copy the desired output from builder to runner. docker removes builder automatically.

(or)

Multi-stage builders are used to reduce the size & improve the performance of the image & containers. We can have two docker files inside the, we will use one as docker multi-stage builder & copy the output and add it to second dockerfile, finally builder will be remove ~~docker file~~ automatically by docker, so in this way we can reduce the size of image and improve the performance.

→ Multi-stage builder are indeed beneficial for java applications.

JDK → Java development kit (extra libraries)
JRE → Java Runtime Environment (used for run, after developing)

JDK = JRE + Extra libraries
JDK > JRE (or) JRE is a subset of JDK

Steps in Containerization -

1. build the image → use docker
2. run the images as container → use kubernetes

What is underlying docker server is crashed. We need to maintain docker hosts. you need some orchestrator to manage all the docker hosts... docker swarm is docker native orchestrator.

→ kubernetes is the popular container orchestrator tool

Autoscaling of containers

HA - run containers in multiple servers

reliability - orchestrator shifts the container to another host if one host is down

kubernetes n/w & DNS is more stronger than docker swarm

kubernetes integrates with ~~providers~~ Cloud providers

storage is better than in dockerswarm.

Docker Swarm -

Docker swarm is a container orchestration tool that allows you to manage & scale docker containers across multiple hosts, forming a cluster known as a swarm.