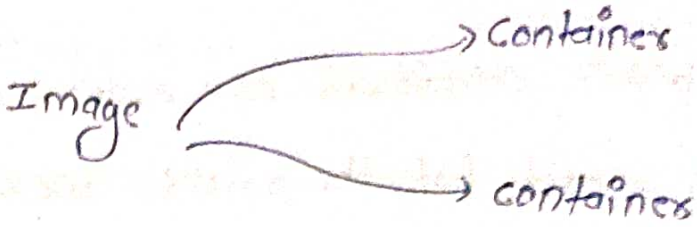
* Docker *



* Containes Identification :

when you create Dockes containes, it is assigned a universally unique identifies (UUID).

※ dockes run -dt -p 80:80 image

- Difficult to remembe UUID because of its length so we also have name of containes

* Docker allow us to supply container names, if we do not specify the name docker randomly generated name from two word joined by underscore

eg: inspiring-poitras → Default [Random]

※ dockes run --name myName -dt -p 80:80 image

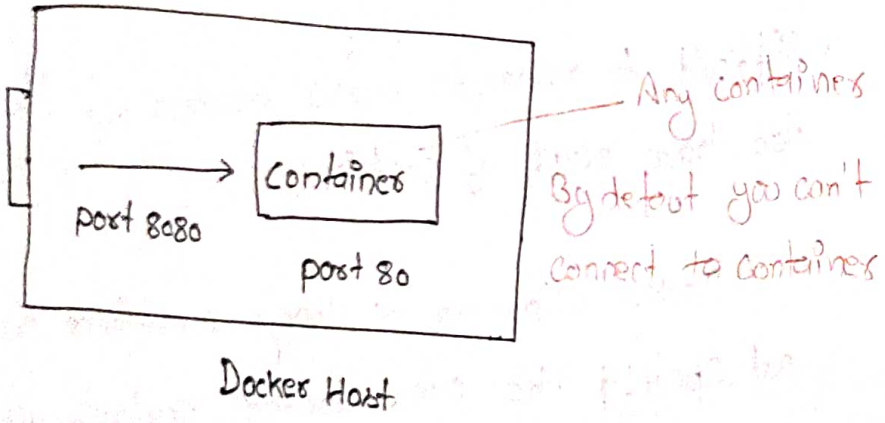
※ docker stop Containes Name

* Port Binding *

** docker run -dt -p 80:80 image

- By Default Docker containers can make connections to the outside world, but the outside world cannot connect to containers.

* If we want to containers to accepts incoming connection from the world, you will have to bind it to host port.



* Here if Someone sent request to Host port 8080 then request would be directed to the port 80 to the container

why Port Binding

—: when you ron container docker will assign private to it [eg: 172.17.0.2] it means from outside n/w you cannot connect. But host cant connect

※ docker inspect container Name

To see info of your containers [eg IPs]

※ docker ps

→ To show all running process status.

* To check. port binding

1) docker ps ↓ if

→ one of the column after you run above command

PORTS

0.0.0.0:80 → 80/tcp

?

Any Request that comes to the port 80 of the Docker host would be sent to the port 80 of dockes container

* Detached Mode * ※ docker run -d -p 8001:80 image

when we start a docker container, we need to decide if we want to run in a default foreground mode or the detached Mode.

※ we can use this if we want to run container but do not want to view and follow all its output.

* Remove Containers *

1) * docker ps ⟶ Running container

2) * docker ps -a ⟶ All the containers.

- Stop one by one
  - ↳ docker container stop Name
- Stop all containers
  - ↳ docker container stop $(docker container ls -aq)

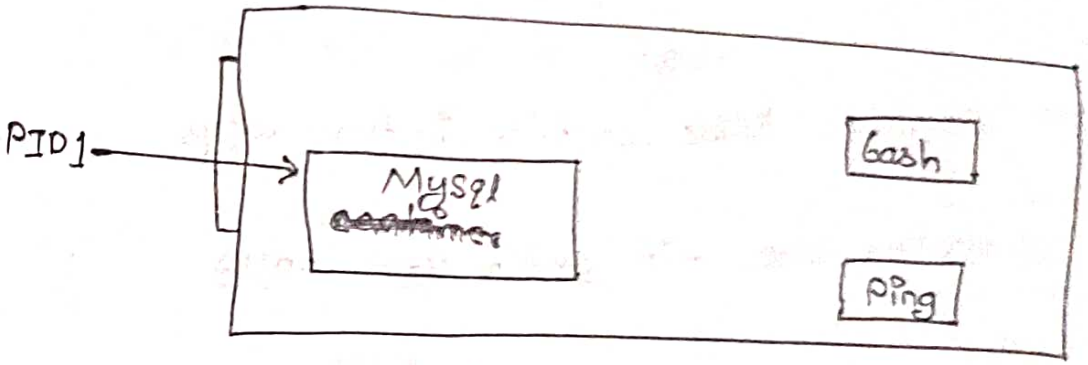
It will give you list of ID's Associated to all containers.

- Remove one by one
  - ↳ docker container rm Name
- Remove all
  - ↳ docker container rm $(docker container ls -aq)

* Dockers Containers exec *

——: dockers containes exec command runs a new command in a running containes

* The command started using dockers exec only runs while the container's primary process (PID 1) is running, and if is not restarted if the containes is restarted.



MySQL Dockes Containes

Here primary process is mysql in PID 1. However within this container there can other programs as well. like bash, ping

we are saying & run dockers container on bash.

* docker containes exec -it docker-containe-Name bash

Reason How we can do it?

Because this containes has bash under /bin as binary.

- Debian based containe ⟶ apt-get update
- Amazon Linux / centos ⟶ yum update

※ ~~netstat~~

※ netstat -ntlp

↳ network statistics

you are binding the STDIN & STDOUT to your host tothe STDIN & STDOUT

* You can run command inside containe in two ways

1) docker exec -it Containe Name -ntlp

2) docker exec -it " bash.

netstat.

* If we stop primary program will automatically came out from bash

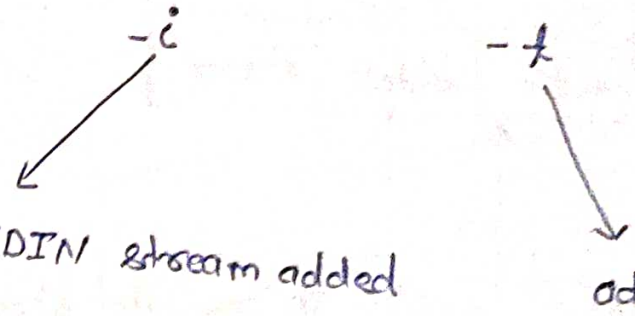
※ docker containe exec -it Name bash

↳ /etc/init.d/name stop.

came out to terminal

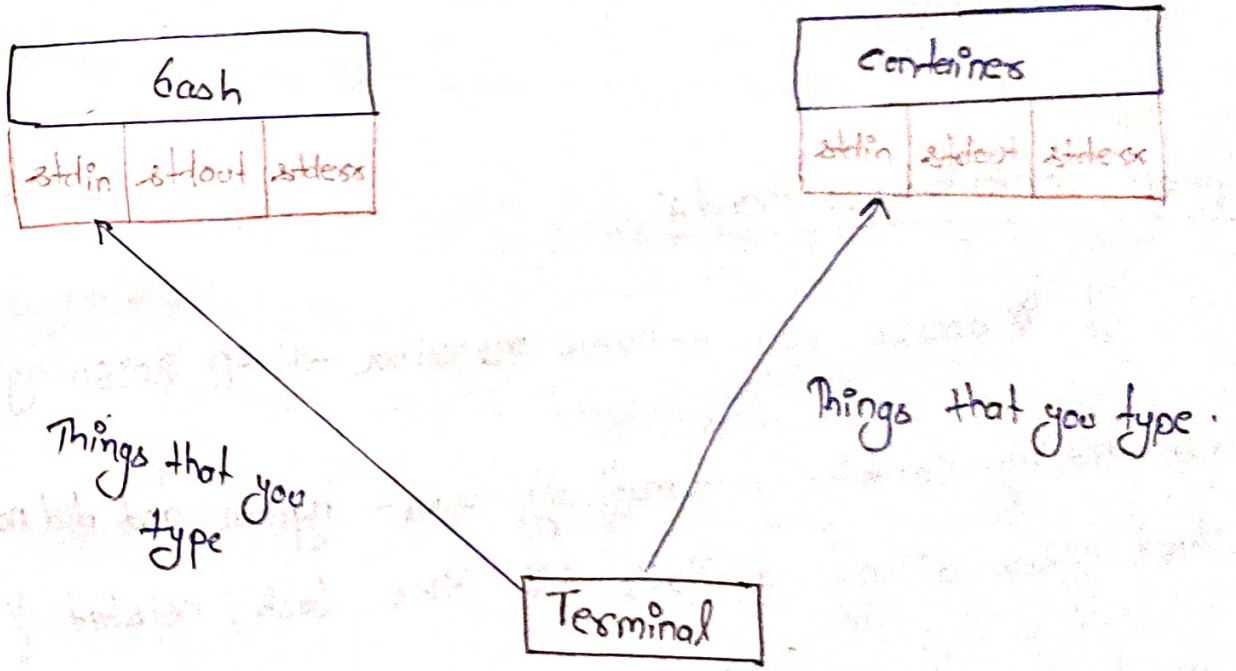
V.I

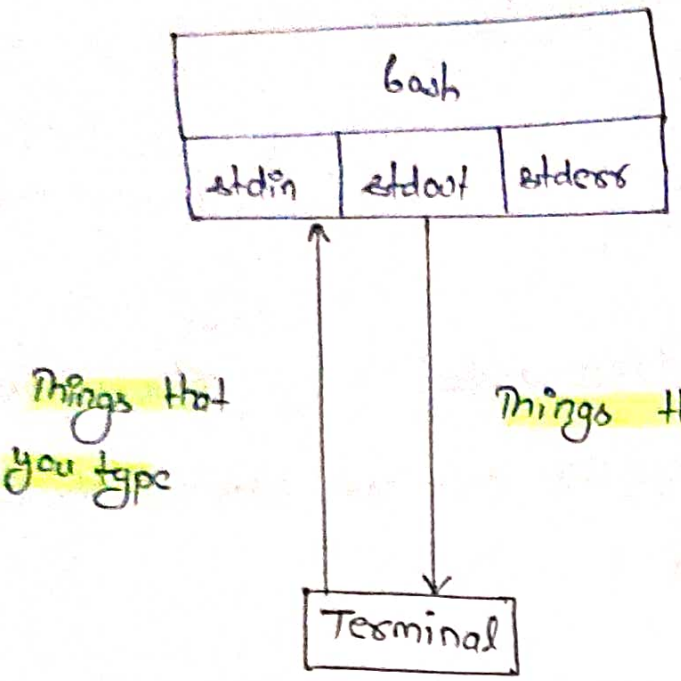
* docker run -it



Basically it makes the container start look like a terminal connection session.

⇒ Every process that we create in Linux env, has three open descriptors: stdin, stdout, stderr.





* -i : -- interactive flag keeps stdin open even if not attached
* --tty : flag allocated a pseudo TTY

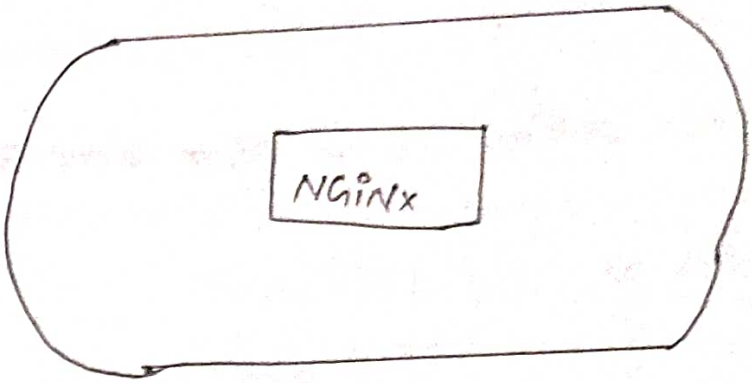
* Default container command *

if * docker run --name myNginx -dt -p 80:80 nginx.

→ How did my containe automatically start nginx and did not start other binary available init like bash, netstat ?

Ans : because of default command.

V-Imp
- whenever we run containes, default command executes which typically runs as PID 1
- This command can be define while we are defining the containes image.



Nginx container

* docker images

↳ To show all images

Each image has it's own default container command.

Note: If I want to set any other command for my image we can do that

eg: whenever containe start/restart we want to apache process to automatically restart

Apache image

* Overriding Default container Command

↳ Two ways we can do this.

1) Modify Argument and build new image

2) pass default command at runtime

2) → *docker container run -d nginx sleep 500

* docker ps

↓

COMMAND → column

"sleep 500". ← overriden command.

? container exit after 500 sec.

Note: if PID 1 exit then container also exit

* Restart Policies in Docker

∴ By Defaut docker container will not start when they exit or when docker daemon is restarted.

- Docker Provides restart policies to control whether your container start automatically when they exit, or when docker restarts.

* By using --restart flag with docker run command.

U-Imp

Flag	Description
no (default)	Do not automatically restart
on-failure	Restart if the exits due to an error which manifests as a non-zero exit code
* unless-stopped	Restart unless it is explicitly stopped or Docker itself stop or restarted
always	Always restart the containes if it stops

* Removing Containers

Steps 1) stop the running container

* docker stop Name/Id

2) * docker rm myninx
↳ Name

Best Practice

Note: if you try remove running container then it will give you error However we can we can forcefully remove it

V. Imp

* Desk Usage *

* df -h → for Linux only

↳ This will give info but does not really tell the disk usage specific to docker components

* docker system df

* Automatically Delete container on Exit

1) * docker run -dt --name test busybox ping -c10 google.com

it will ping 10 times and exit but not deleted.

* docker logs test

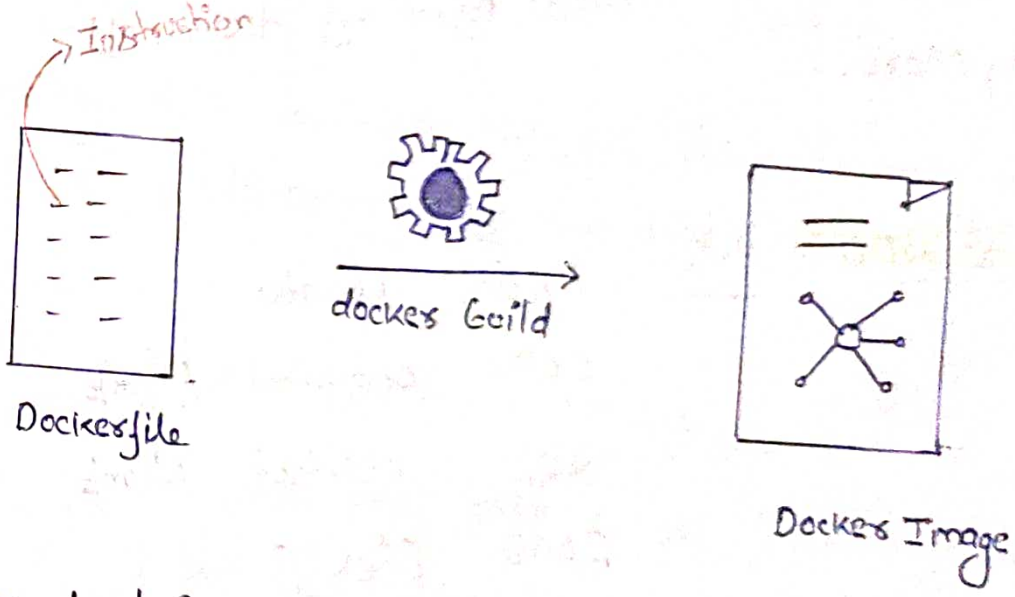
2) * docker run -dt --rm --name test busybox ping -c10 google.com

* Understanding Docker Images *

Note:

* Every Docker container is based on an image
* Docker can build image automatically by reading the instruction from a Dockerfile

Dockerfile: A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.



Format of Dockerfile

```
# comment
INSTRUCTION arguments
```

- A Dockerfile must start with a FROM instruction

FROM instruction specifies the Base Image from which you are building

* Commands

- FROM
- RUN
- CMD
- LABEL
- EXPOSE
- ENV
- ADD
- COPY
- ENTRY POINT
- VOLUME
- USER
- Many More...

eg: We want to create custom nginx image and it should have index.html file which state:

" Welcome to Base Image from Genie Ashwani;"

⇒

```
FROM ubuntu
RUN apt-get update
COPY index.nginx-debian.html /var/www/html
CMD nginx -g 'daemon off;'
```

* COPY Vs ADD

↳ Can use in normal file

eg:

```
FROM ubuntu
COPY copy.txt /temp
ADD add.txt /temp
CMD ["sh"]
```

COPY: COPY takes in a src and destination. It only lets you copy in a local file or directory from your Host

→ Can use for compress files ADD will decompress and copy

ADD: ADD let you do that too, but it also support 2 other sources.

* ⟶ First, you can use a URL instead of a local file/ directory
* ⟶ Secondly, you can extract a tar file from the source directly into the destination

eg: ADD http://example.com/big.tar.xz /usr/src/things/
RUN tar -xJf /usr/src/things/big.tar.xz -C /usr/src/things

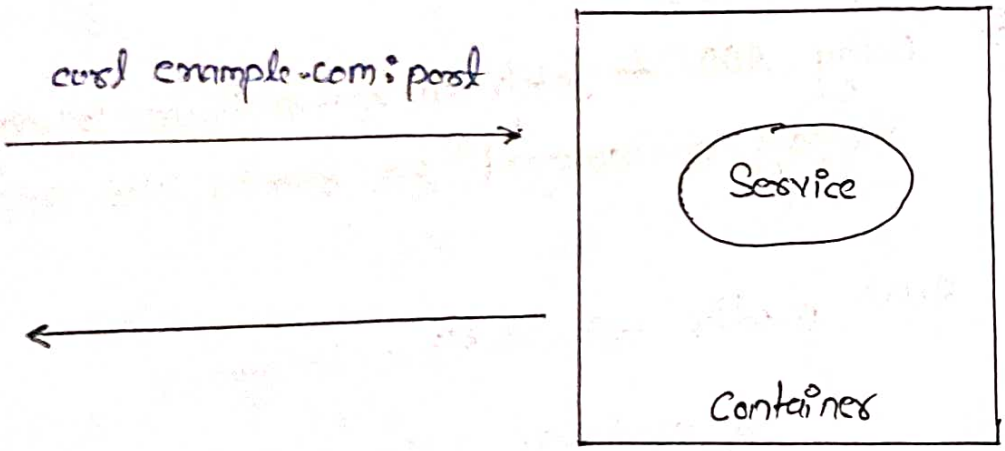
Note: using ADD to fetch packages from remote URLs is strongly discouraged, you should use curl or wget

```
RUN mkdir -p /usr/src/things \
    && curl -SL http://example.com/big.tar.xz \
    | tar -xJC /usr/src/things \
    && make -C /usr/src/things all
```

* EXPOSE -: inform docker that the container listens on the specified network port at runtime

The EXPOSE instruction does not actually publish the port

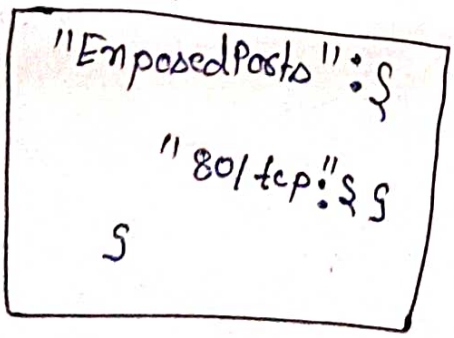
Note: It functions as a type of documentation btw the person who builds the image and the person who run the container, about which ports are intended to be published.



eg:

```
FROM ubuntu
RUN apt-get install service
EXPOSE 9324
CMD ["service"]
```

eg: docker inspect nginx



Note: So once you know the port of your service your publish it on your host machine by using (-p)

※ docker ps

↓

PORT ← column

80/tcp

* Docker HEALTHCHECK : Allow us to tell the platform on how to test that our application is healthy.

Imp when Docker starts a container, it monitors the process that the container runs. if the process (PID 1) ends, the container exits.

That's just a basic check and does not necessarily tell the detail about the application

-- interval
-- timeout
-- start-period
--retries =x

* ENTRYPOINT —: Is use to set the image's main command

↳ don't allow you to override the command

it is imp to understand distinction b/w CMD and ENTRYPOINT

eg: Dockerfile

```
FROM busybox
ENTRYPOINT ["/bin/ping"]
```

* docker build -t baseo1 .

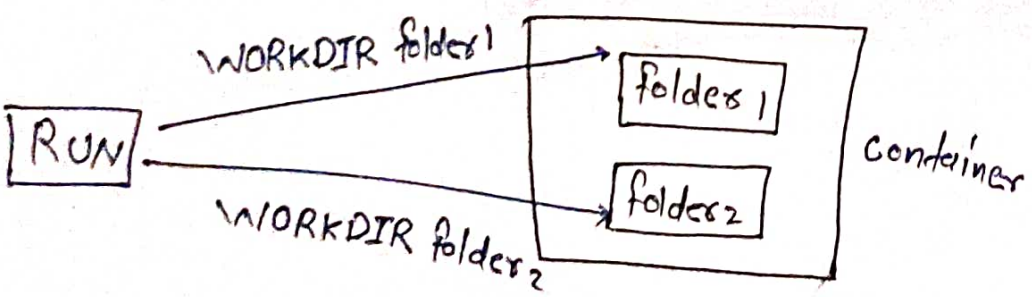
* docker run -dt --name base01 baseo1 -c 20 google.com

appended.

/bin/ping -c20 google.com

we can use it multiple times

* WORKDIR —: Sets the working directory for any RUN, CMD ENTRYPOINT, COPY and ADD instructions that follow in the Dockerfile



eg: Dockerfile

```
FROM busybox
RUN mkdir /root/demo
WORKDIR /root/demo
RUN touch file01.txt
CMD ["/bin/sh"]
```

1) ✻ docker build -t <work-dir> .

2) ✻ docker run -dt --name workdir work-dir sh

3) docker exec -it work-dir sh

~/demo ✻

↳ inside demo.

✻ ENV :- Sets the enviroment variable <key> to the value <value>

eg:

```
ENV NGINX 1.2
RUN curl -SL http://example.com/web- $NGINX.tar.xz
RUN tar -xzvf web -$NGINX-tar.xz
```

Note: we can use the -e, --env, and --env-file flag to set simple env variable in the containes you're running or overwrite variable that are definedinthe Dockerfile

eg ✻ docker run --env VAR1=value --env VAR2=value2 ubuntu env | grep VAR.

* Tagging Docker image

(Two ways

1) docker build (-t) demo:v1.

2) docker tag Id demo:v2

existing image

* Docker Commit :

whenever you make changes inside the container it can be useful to commit a container's file changes or setting into new image

Syntax

docker contair commit Container-ID myimage:v1

Note: By default the container being commited and its processes will be paused while the image is commited.

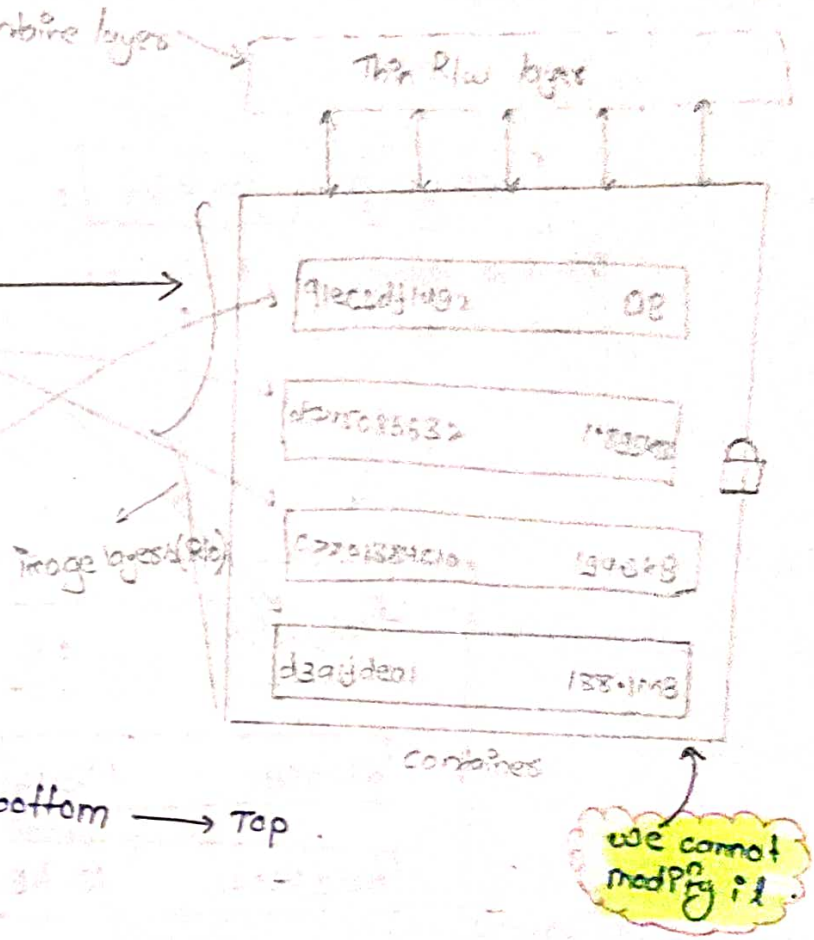
- we can also change instruction like CMD ENV LABLE

V. Imp * Layers of Docker Image *

- Docker Image is built up from a series of layers
- Each layer represents an instruction in the image's Dockerfile

eg:

```
FROM ubuntu
COPY . /app
RUN make /app
CMD python /applapp.py
```



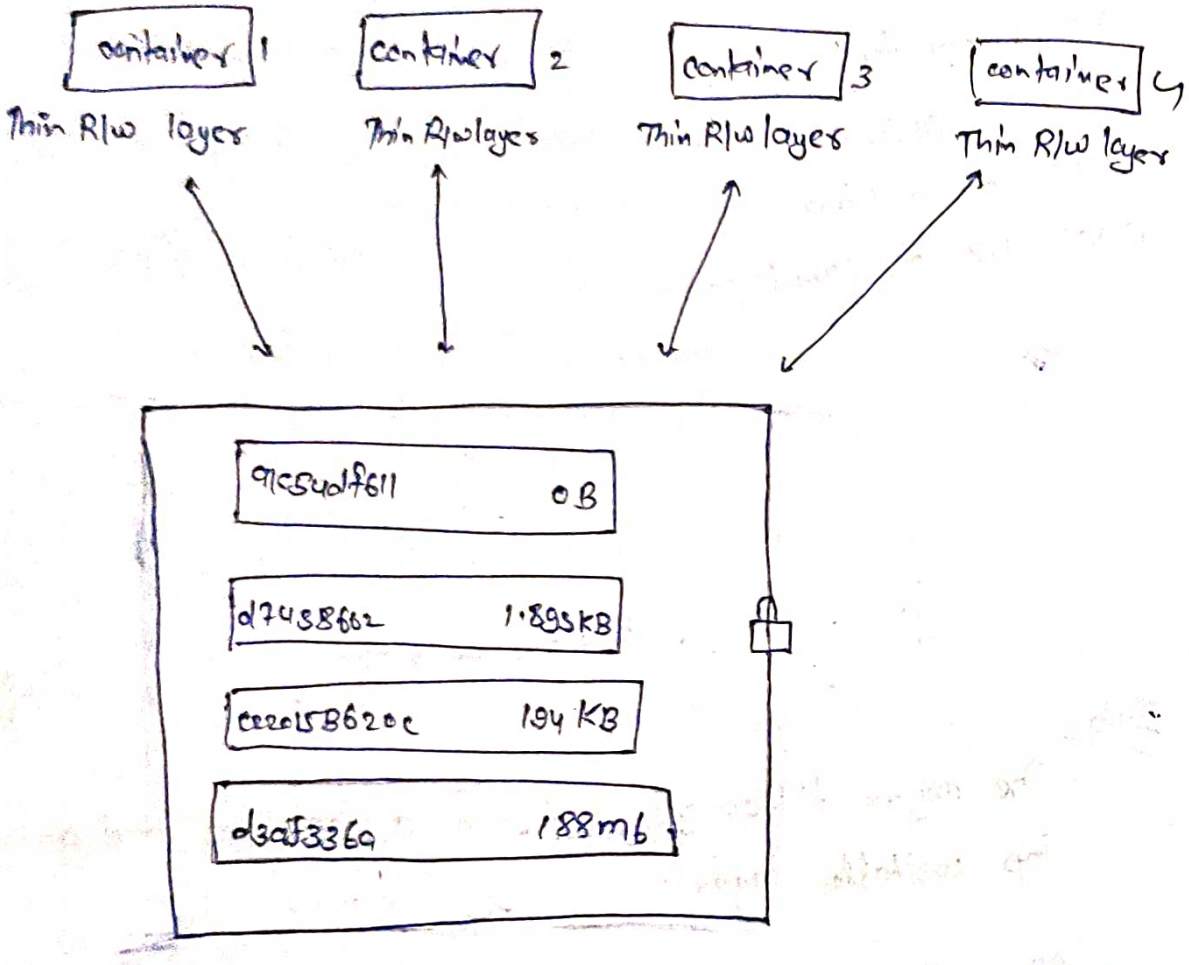
- Bottom → Top

Imp

- The major difference between a container and an image is the top writable layer
- All writes to the container that add new or modify existing data are stored in the writable layer.

Let's understand

- 4 containe using same image because they had own R|w layer any change they made get stored this layer.



eg?:

FROM ubuntu — layer1

RUN dd if=/dev/zero of=/tmp/file.txt bs=1M count=100 — layer 2

RUN dd , , , file2.txt — layer 3

* child command for Docker Images

1) docker image build
2) " " history
3) " " import
4) " " push
5) " " pull
6) " " rm
7) " " inspect → info
8) " " tag → docker image inspect --format='{{.ID}}'
9) " " Pruning
etc.

Pruning → * docker image prune

↳ allow us to clean up unused images

by defaut it only clean dangling images

↓

Image without Tag and image not referenced by any container

* docker image prune -a

↳ remove all images not used by containers.

[dangling]

* Flattening Docker Images

Flattening on image to single layer can help reduce the overall size of the image.

↳ we have multiple approach

✓ 1) export containes that running on image you want to flattening

* docker export mycontainer > mynewcontainImage.tar

* cat mynewcontainImage.tar | docker impor - myImage:latest

* Docker Registry *

- there are various type of registry
  - Docker Registry
  - Docker Trusted "
  - Private "
  - Docker Hub

* Push images to DockurHub

1) create account and repo

2) * dockers tag myimage usesName/repo:version

3) * docker push usesName/repo

※ Moving Image Across Hosts

1) The docker save command will save one or more images to a tar archive

※ docker save busybox > busybox.tar

※ docker load < busybox.tar

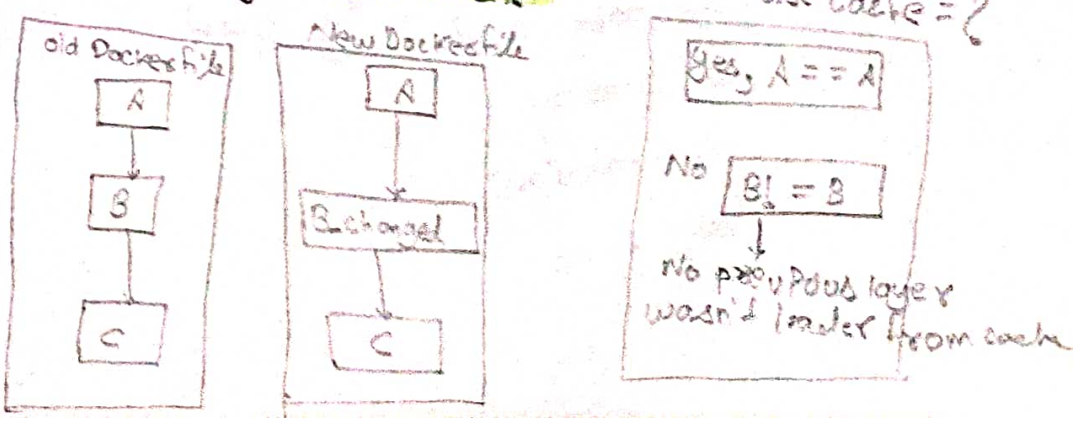
※ Build cache / layer cache —: Docker create containers images using layers.

→ each command that is found in Dockerfile creates a new layer

→ Docker uses a layer cache to optimze the process of building Docker images and make it

Note: If similar instruction is used in some images already then if we build ~~imges~~ images and docker will check does it have instruction already execution in some other image build so docker will utilize that cache.

Imp • If the cache can't be used for a particular layer, all subsequent layers won't be loaded from the cache.



* Docker Networking *

: Sometime we have requirement to connect my container to outside n/w

- Sometime we want containers in isolated condition.

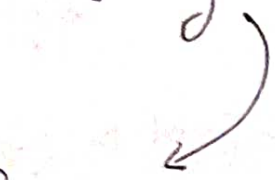
Note:

Docker takes care of the networking aspect so that containers can communicate with other containers and also with the Docker host

→ * ~~ifconfig~~

* ~~[illegible]~~

* ifconfig



if docker install it will give a result of dockers interface which is bridge based network setup.

* Docker networking subsystem is pluggable, using drivers.

There are several drivers available by default, and provide core networking functionality.

1) bridge
2) host — By Default
3) overlay
4) macvlan
5) none.

⇒ To see Docker n/w drivers

* docker network ls

NETWORK ID	NAME	DRIVER	SCOPE
5336f6023809	bridge	bridge	local

* docker inspect bridge

```
"config": [
    {
        "subnet": "172.17.0.0/16",
        "Gateway": "172.17.0.1"
    }
]
```

Network
Host
Subnet Mask

Any container launch with-in the bridge this bridge network will get IP address from Here

User Define Bridge Network

Steps

1) * docker network create --driver bridge mybridge
   * docker network ls
   * docker inspect ~~mybrig~~ mybridge.
   * ifconfig
   * docker container run -dt --name mybridge01 --network mybridge ubuntu
   * brctl show
   * docker network inspect mybridge
   * docker container exec -it mybridge01 bash
   * ping mybridge02

Automatic DNS Resolution in user define Bridge Network.

* Host Network *

→ This drives remove the network isolation b/w the docker host and the docker container

→ if you run a container which binds to port 80 and you use host networking, the container's application will be available on port 80 on the host's IP address.

Steps

* docker container run -dt --name myhost --network host ubuntu.

* docker container exec -it myhost bash

* None Network

→ use to completly disable the networking stack on a container

This mode will not configure any IP for the container

* Publish All Argument for Exposed ports

=> * docker container run -dt --name webserver nginx (not exposed.)

* curl 127.0.0.1:80
↳ Connection refused

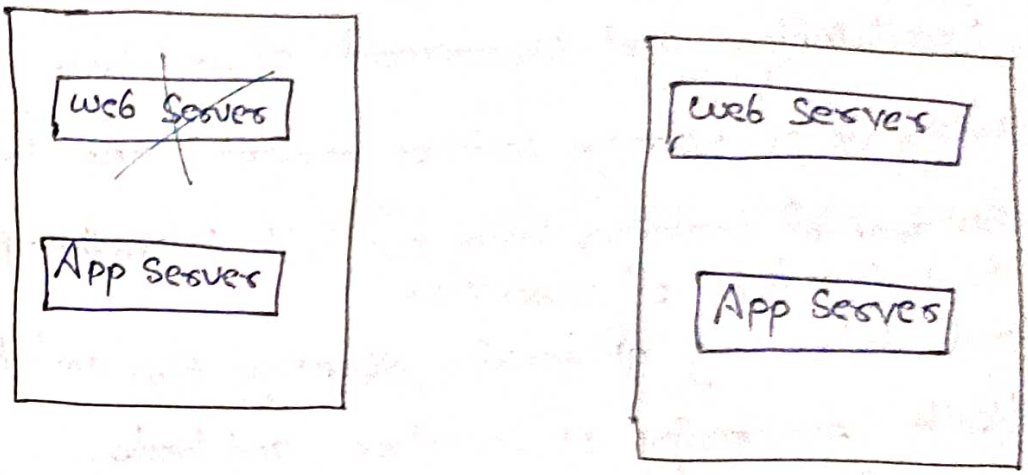
↳ to solve this -P 8080:80

* docker container run -dt (-P) --name demo nginx

Random Port will assign

* Docker Containers Orchestration *

* container orchestration is all about managing the life cycle of containers, especially in large, dynamic environment..



Note: if one of the web servers containes went docon then the doeshion is who will monitor this specific web server container and that monitoring script should also have event driven ~~mee~~ mechenisem such a way if this web servers container went down if also restart the container

if due to some reason container not restarting then that script should automatically start new container inthe VM 3

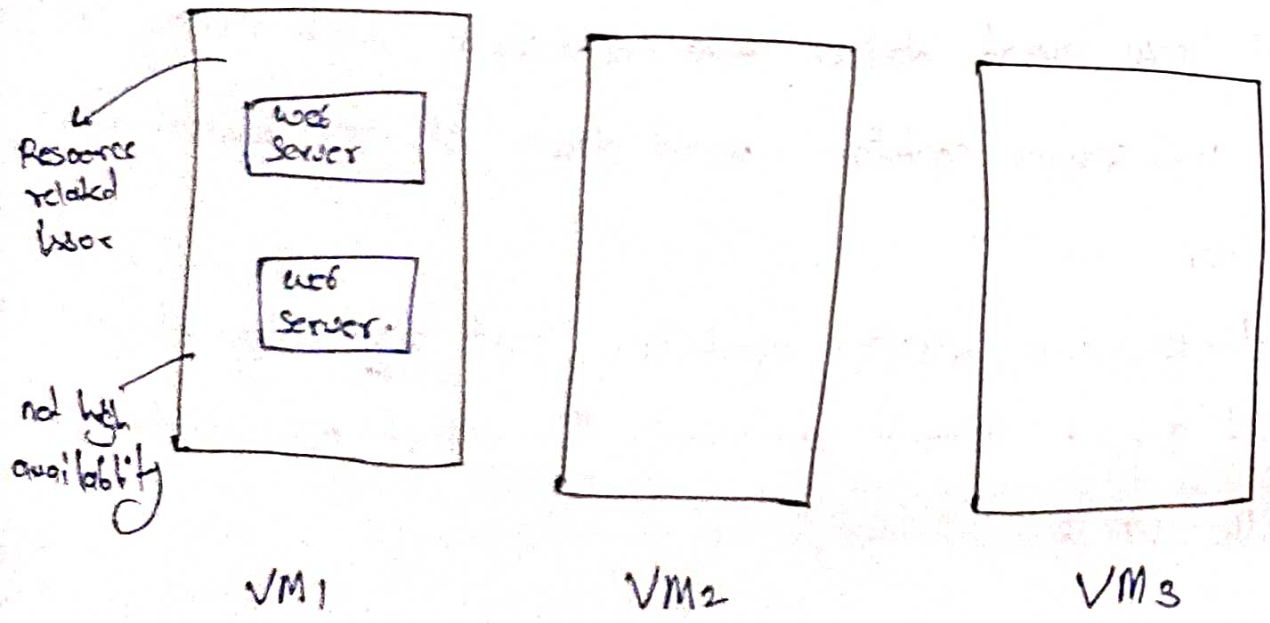
This can be achive with the help of container orchestration.

* Importance of container Orchestration.

→ Container Orchestration can be used to perform lot of tasks, some of them includes:

1) Provisioning and Development of containers.
2) Scaling up or removing containers to spread app load evenly
3) Movement of container from one host to another if there is a shortage of resources
4) Load balancing of service discovery b/w container
5) Health monitoring of container and hosts.

Requirement: Minimum of 2 web-servers should be running all the time.



* Design Pattern *

* Design Pattern are typically split into three categories.

* This is called Gamma Categorization after Erich Gamma, one of GoF authors

1) Creation Patterns: — provide various object creation mechanism which increase flexibility and reuse of existing code

—: Deal with the creation (construction) of objects.

—: Explict (constructor) Vs Implict (DI, reflection)

2) Structural Patterns:

—: concerned with the structure (eg: class members)

—: Stress the importance of good API design.

3) Behavioral Pattens

—: They are all different: no central theme

✱ Builder ✱

* Some objects are simple and can be created in a single constructor call
* other objects require a lot of ceremony to create
* Having an object with 10 constructor argument is not productive
* Instead, opt for piecewise construction.
* Builder provides an API for constructing an object step by step.

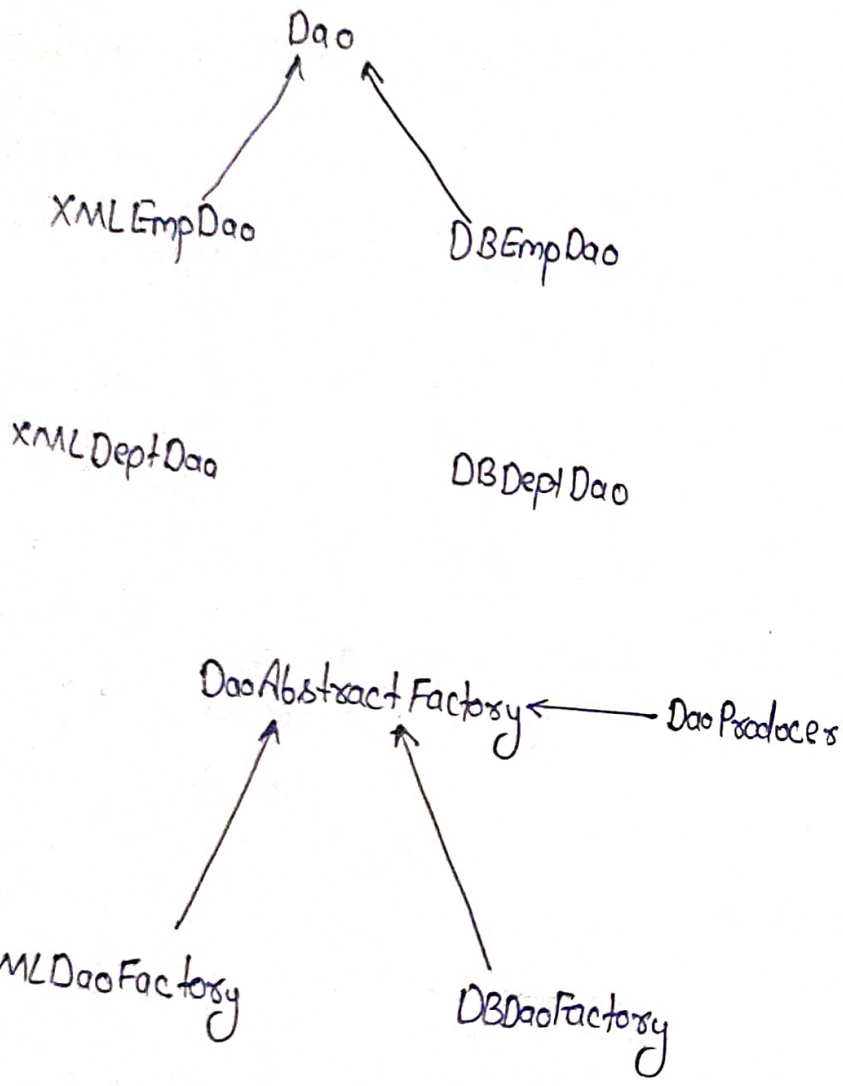
Builder	when piecewise object construction is complicated provide an API for doing it succinctly

Factory

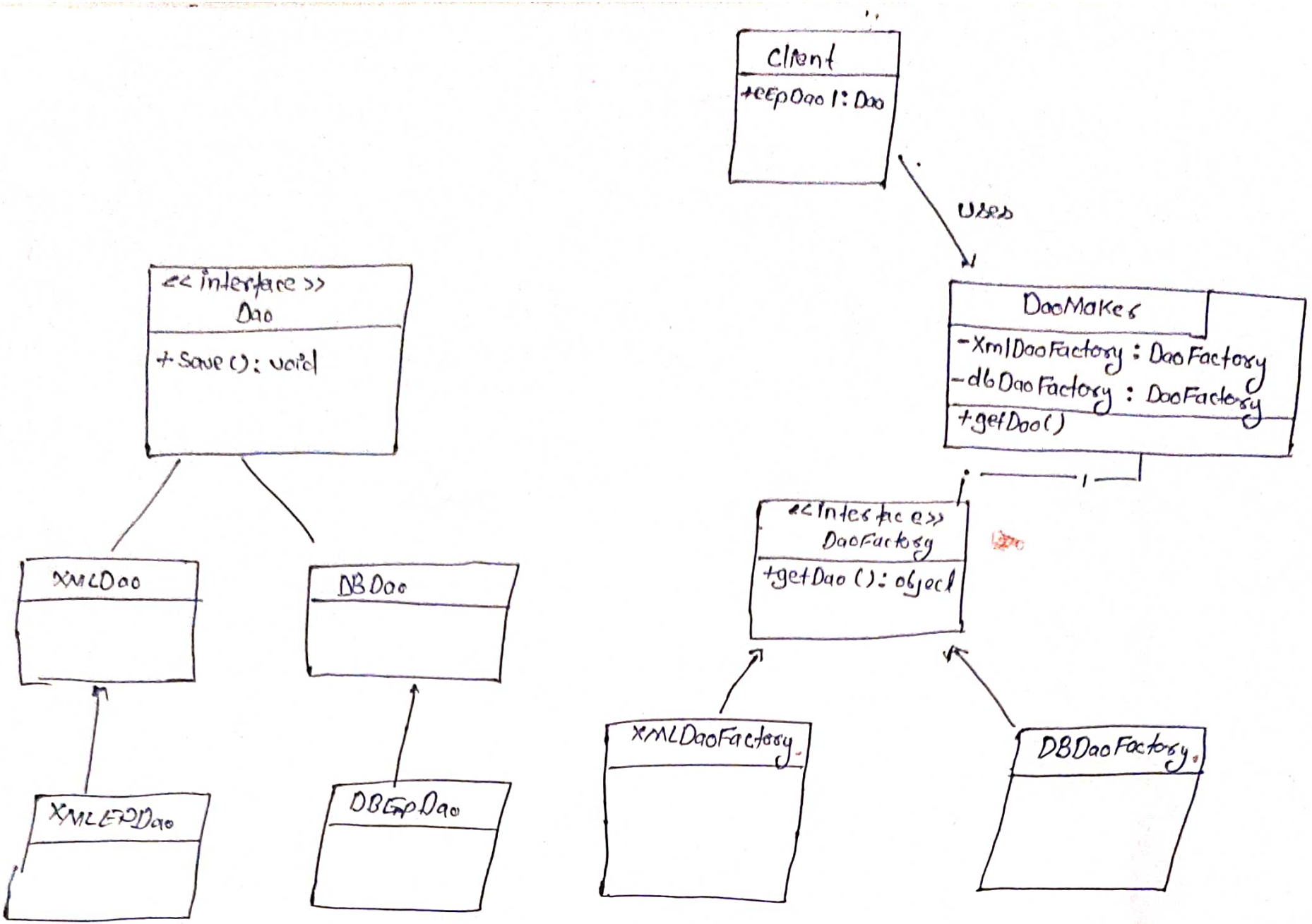
* Abstract Factory *

1) Abstract Factory is a Factory of factories.

Eg: DAO Factory



UML


Client
+EmpDao1: Dao
uses
DaoMaker
-XmlDaoFactory : DaoFactory
-dbDaoFactory : DaoFactory
+getDao()
<<Interface>>
DaoFactory
+getDao(): object
XMLDaoFactory
DBDaoFactory
<<interface>>
Dao
+Save(): void
XMLDao
DBDao
XMLEmpDao
DBEmpDao