What is container ?
A way of packaging a application with every thing they need insider that package like configuration , dependencies etc .
That package is portable like any other artifact , easily shared and moved around .
This make development and deployment process more efficient .

Where do container live >?
Container repo .
Public repo for docker container . (Docker hub).

How container improved development process > ?
Application development ==>
Before containers : installation process is different in each os environment with multiple steps.
After containers : own isolated environment , packed with complete configuration .
One command to install the app are regardless of with os u are on starting the docker container will be same .
Run same app with 2 different versions .

A**pplication deployment ==>**
Before containers : developers will create a jar file with set of instruction of how to configure and set it up in the server . —> given to the operational team and they will handle setting up the environment this will result in configuration in the same server led to dependency version conflict .
Misunderstanding
After containers : : developers and operation work together to package(dependencies ,jar , configuration) the application in the container .
No environment configuration need in server , except docker runtime .
Only this I need to do is to run a docker command that pull the docker image that u stored some where in the repo and run it.

Other containers : containerD , cri-0 .

What a container made of technically :
Is made up of layers of images. Stack based and the base of it is mostly l**inux based image(**alpine) because of their small size .
On top of base image u will have a**pplication image ,**

Download docker desktop : then go to docker hub search for Postgres

Docker run -e POSTGRES_PASSWORD=mysecretpassword
```
// this will set a environment valuable inside our docker container
//-e stand for environment variable . First one is name of EV and =
second is value of EV .
```

Docker run -e POSTGRES_PASSWORD=mysecretpassword Postgres:13.10
// and provide the version of it
Docker container are made up of layers. // separate images are downloaded.
//advantage of image is if I want to down load the newer version of Postgres the layers which are same btw those two application two versions of postgres only different versions are downloaded.
// as soon as it downloaded it , it starts it also due to scripts .

Docker ps :

It provides details about the active containers in your Docker environment

Docker image vs docker container >
Images
The actual package eg. configuration ,postgres version 15.3,start script,dependencies..etc
This is the artifact that more ROUND. Not running

CONTAINER :
WHEN I PULL THAT IMAGE IN MY LOCAL Machine
Actually run the application in the container .
Containers are virtualised environment where application runs.
Running , container environment is created .

If I want both versions running at the same time in local machine. ??
 docker run -e POSTGRES_PASSWORD=mysecretpassword postgres:14.7

Some of the layers are same . It will download only the layers which are different .

So now I have two postgres with different version running on pc .

How to delete container running :?

```
vishaldwivedi@vishals-MacBook-Air-2 ~ % docker stop modest_neumann
modest_neumann
vishaldwivedi@vishals-MacBook-Air-2 ~ % docker rm modest_neumann
modest_neumann
```

now I can run any no. of application with different version >.with no problem

DOCKER container vs virtual machine ::
Os has two layers OS kernel and os application layer .
Kernel is the core of os.
It directly interact with the hardware and software components,
And the application runs on kernel layer .
Both docker and vm is virtualisation tools what part of os they virtualise . ?
Docker virtualise the os application layer ,service and app installed on top of the layer ,it uses the host kernel as don't have its own.
Virtual machine has application layer and its own kernel which virtualises the complete os.

Meaning : when u download a virtual machine image in out host it does not use your host kernel it boots up its down

Docker image size is much small because they need to implement only one layer.
Eg mb . But vm images will be gb .

Docker containers takes seconds to start but vm take long time as they need boot there own kernel and os application layer on top of it.
Vm is compatible with all os ,compatible with only linux distros . Because u can run linux application layer on windows kernel
Most container are linux bases docker was original build for linux os . But later docker upgrade and introduced docker desktop for windows and Mac with allow u to run linux based container on linux and windows os .
Docker desktop uses a hypervisor layer with a light weight linux distro .

Docker architecture and components
Containers existed already before docker , but docker make container user friendly .
How does docker actually works >?
When u install docker u install docker engine with comes with 3 parts
docker server : pulling images , managing images and containers.
Docker api : interacting with docker server.
Docker cli : client to execute commands , to pull , run , stop , images and containers.

Docker server has :
Container run time : which is responsible to pulling images , managing containers life cycle .
Volumes : persisting data in containers .
Network : configuring network for container communication
Build : build own docker image.(artifact that we can make for our application to run as container.)
Need only a container run time Tool eg: container d , cri-o
Need to build a image ? Tool eg: build ah .

Docker main commands :
Container is a running environment for an image
Eg: file system , environment configuration , application image .
Application image can be postgres , mongo , etc.
Port binded : talk to application running inside of container .
It has virtual file system .

Docker images : list all the existing docker images.
Now I want redis running do that my application connect to it . //docker run redis
Docker ps : // will list the current running containers .

Docker run -d container name
Docker stop idOfContainer
Docker start sameId
Docker ps -a // will show all the container running or stoped container .
Docker run redis:3.3 //pull the images and start the container right away .

Same image with different versions :
Both container are listing to same port no .
Which port the container is listing to the incoming request . Is port no

How these two are dunning on the same port. ?
Container port vs host port >
Multiple container can run on your host machine .
Make binding with the container and host machine port if not containers will be unreachable I cant use the application .
We have 2 redis so I need to bind it to two different ports in our laptop .

Docker run -pn(port of the host):(container port)
Docker run -p 6000:6379 redis

```
vishaldwivedi@vishals-MacBook-Air-2 ~ % docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS                    NAMES
fe66bfa95d2f   redis     "docker-entrypoint.s..."   10 seconds ago   Up 10 seconds   0.0.0.0:6000->6379/tcp   quizzical_goldwasser
```

Binding is done .
Now use 6001:6379 redis:6.2.

```
vishaldwivedi@vishals-MacBook-Air-2 ~ % docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED          STATUS          PORTS                    NAMES
7d0245a29bf8   redis:6.2   "docker-entrypoint.s..."   14 seconds ago   Up 13 seconds   0.0.0.0:6001->6379/tcp   awesome_keller
fe66bfa95d2f   redis      "docker-entrypoint.s..."   3 minutes ago    Up 3 minutes    0.0.0.0:6000->6379/tcp   quizzical_goldwasser
```

Docker run -d redis (detached mode does not block the terminal)
Now I have two different redis versions running both of them bound to different ports on my laptop and containers them self are listing to the req on the same port.

Docker commands for troubling shooting : to see the logs of the containers .

1; Docker ps :
2; Docker logs containerID or name of the container

Giving costume name to the container :
3 ;Docker run -d -p6001:6379 —name redis_older redis:6.2

```
vishaldwivedi@vishals-MacBook-Air-2 devops % docker run -d -p6001:6379 --name redis_older redis:6.2
4f5e8afe0009dd2b90272a1aa0cfd78ee6fe7f620568b5ff86cea50b6dca0874
```

4 Docker exec -it containerIDORcontainderName /bin/bash // now u are inside of the container .

```
root@e98f9532b42c:/data# pwd
/data
root@e98f9532b42c:/data# cd /
root@e98f9532b42c:/# ls
bin  boot  data  dev  etc  home  lib  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
```

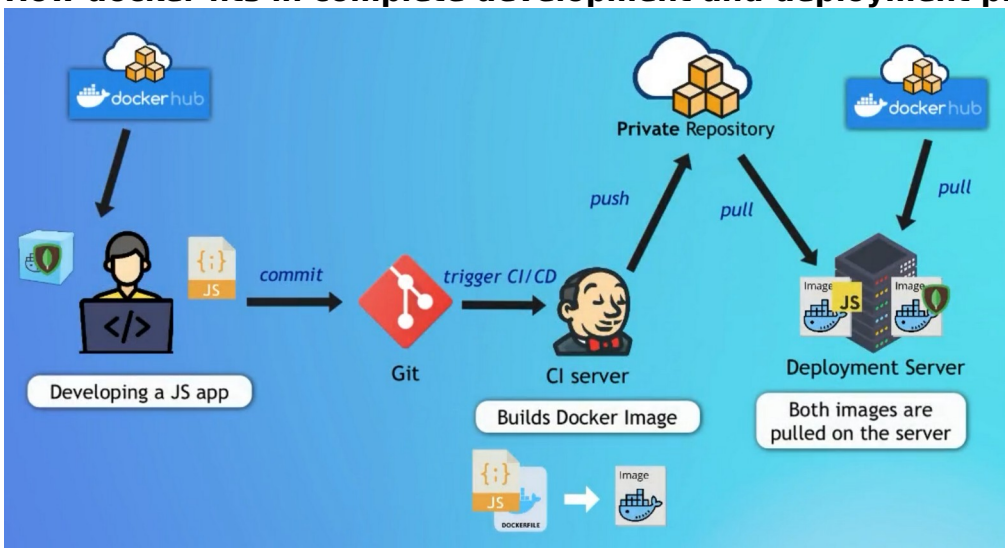
Print the EV : env

Exit to exit : exit from the container

Docker run :: where u create a new container from a image .
Eg. :docker run redis:latest .(it will that the version as attribute) .
Eg: docker run -d(detach) , -p(portioning), —name(set name of the container).

Docker start : if u want to restart the stoped container
Eg : docker start containerID

How docker fits in complete development and deployment process . ?

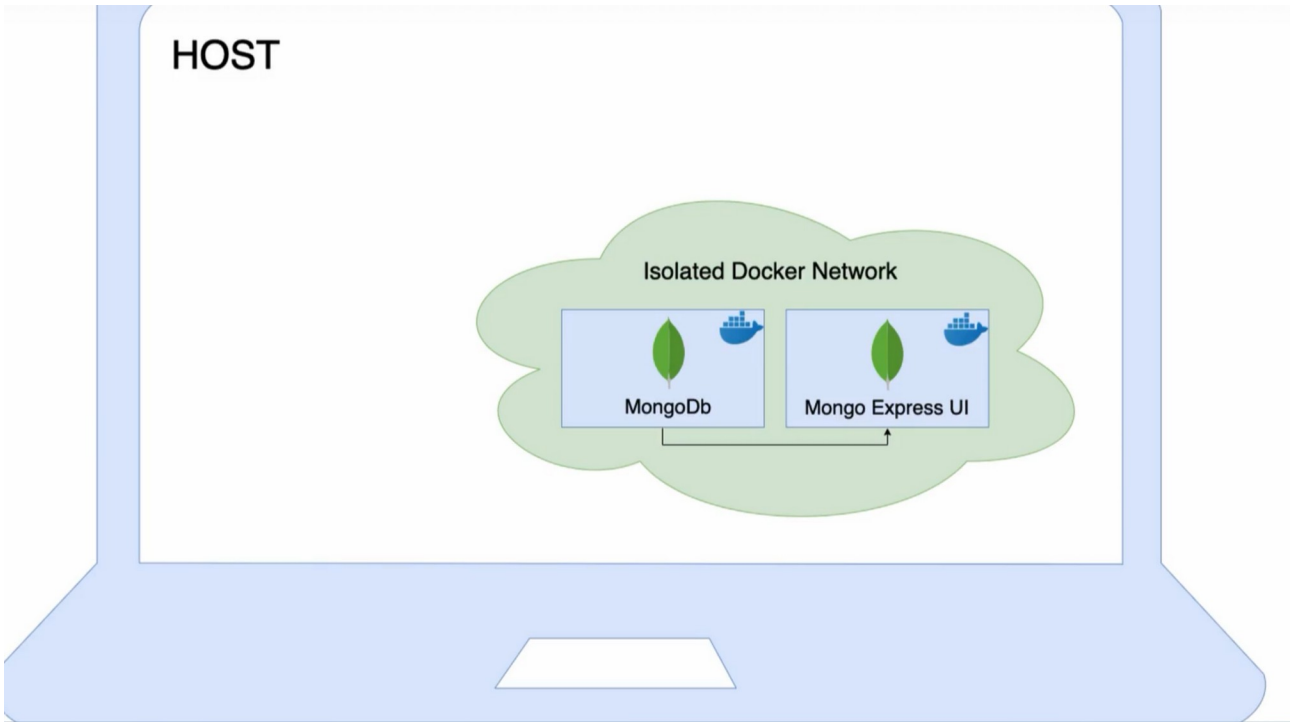


Eg:
I u a developing a js application in out local machine your JS application uses MONGODB DB
Instead of installing it in your laptop u down load a docker container form the docker hub.
U connect your js app with mongo DB
Now u commit your code into git hub which will trigger CI CD eg Jenkins build .
Jenkins build will produce artifice build from your application
First it will build your js application and then build a its artifact(image) .
It will pushed to the private docker repository .(usually in the company u will have the private repo so that other people don’t have the access to your images.)
Now the development server will pull the javascript image from your private repository and pulls the mongoDB image from docker hub the js application is depend on it.

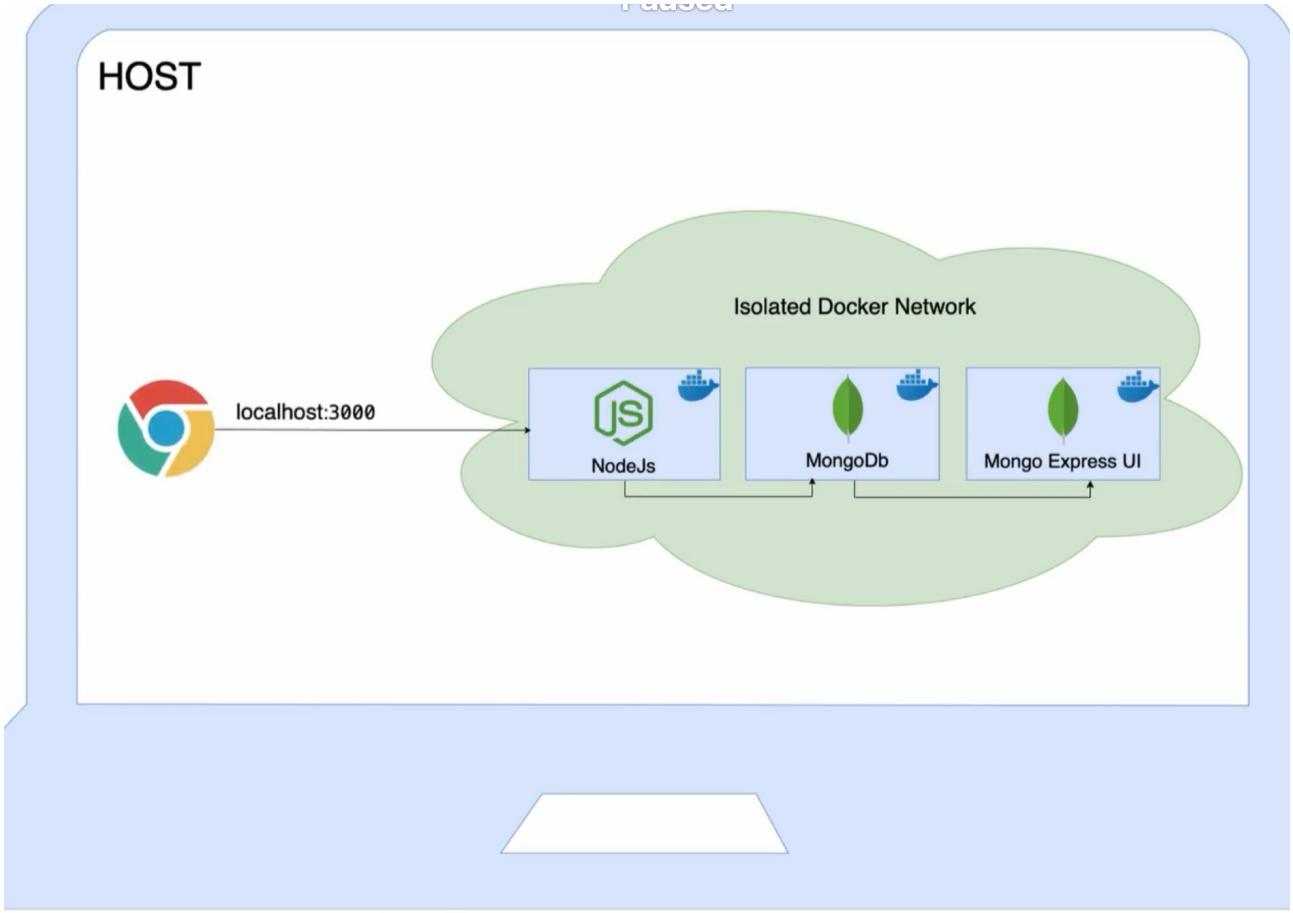
Now these two container will talk with each other .

Demo project :
Go to docker hub : docker pull mongo (this will pull mongo image from docker hub)
: docker pull mongo-express
Now connect the two but first understand docker network concept

Docker network :



These two can talk to each other using just the container name . No local host port no etc. needed. Because they are in the same network .
And the application that run out side the docker like node js runs on node server. Is gonna connect to they using localhost and port no .
So later when we package our application into its own docker image , it will contain the code that we wrote like index.html js. In node js image and the browser which is running out side the docker network will connect via a localhost and port no .



Docker network ls : command will list the default docker network
Now create our own network for mongo db and mongo express.
: docker network create mongo-network
Now run docker mongo container .
// open the port of mongo db .
27017:27017(local host port: container port)

Step 2 .docker run -d \ // to run the mondo container
-p 27017:27017 \ // set up the ports
-e MONGO_INITDB_ROOT_USERNAME=admin \ // password and user name asked at the starting of mongo Db server
-e MONGO_INITDB_ROOT_PASSWORD=password \
--name mongodb \ // over rider the name of the container
—net mongo-network \// this container is gonna run on the mongo network
mongo // start the container .

Docker log idOfTheContainderUWantToSee.

Now we want mongo express to connect to the mongo Db on start up .
Step 3: start mongo-express

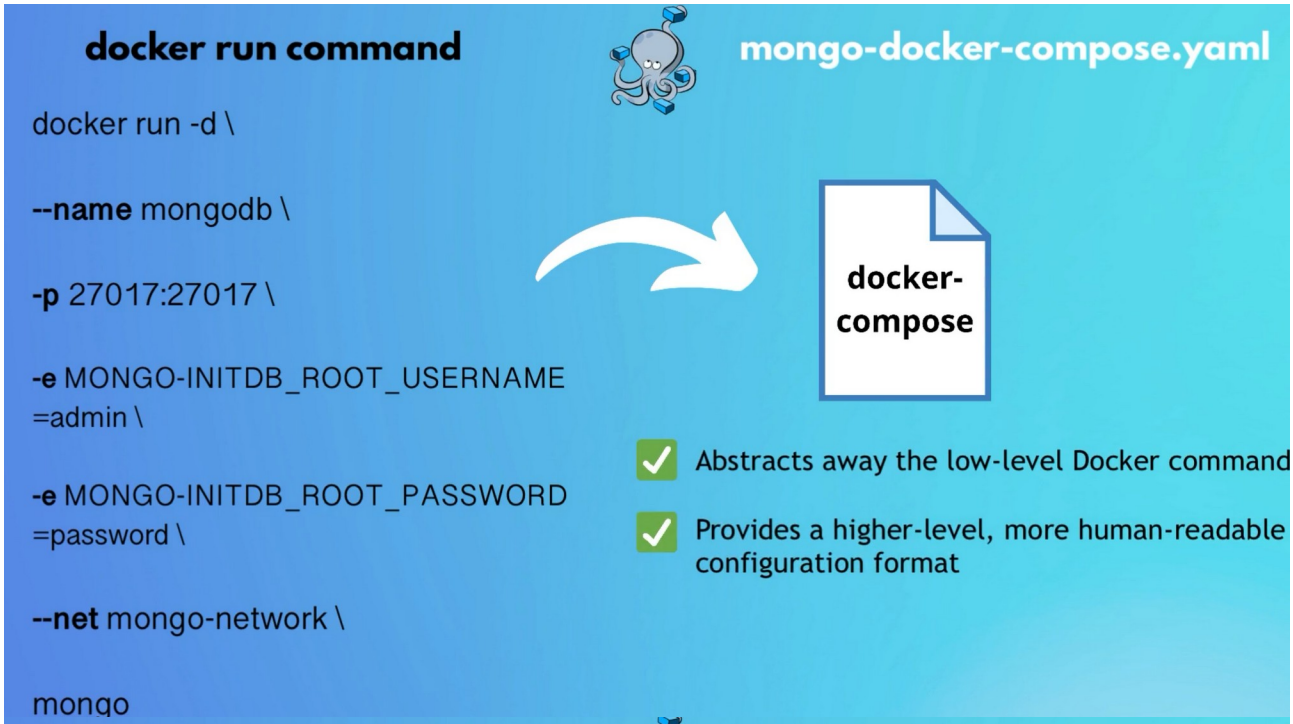
docker run -d -p 8081:8081 -e
ME_CONFIG_MONGODB_ADMINUSERNAME=admin -e
ME_CONFIG_MONGODB_ADMINPASSWORD=password --net mongo-network --

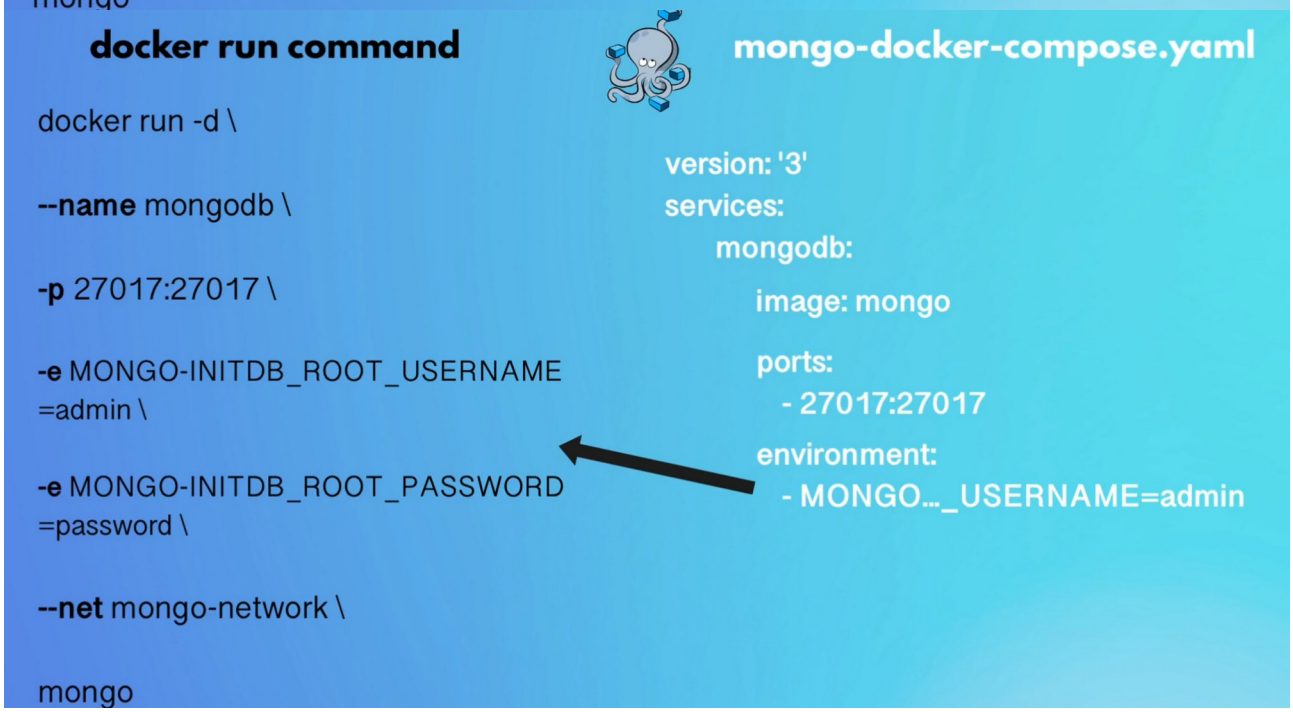
```
name mongo-express -e ME_CONFIG_MONGODB_SERVER=mongodb mongo-express
```

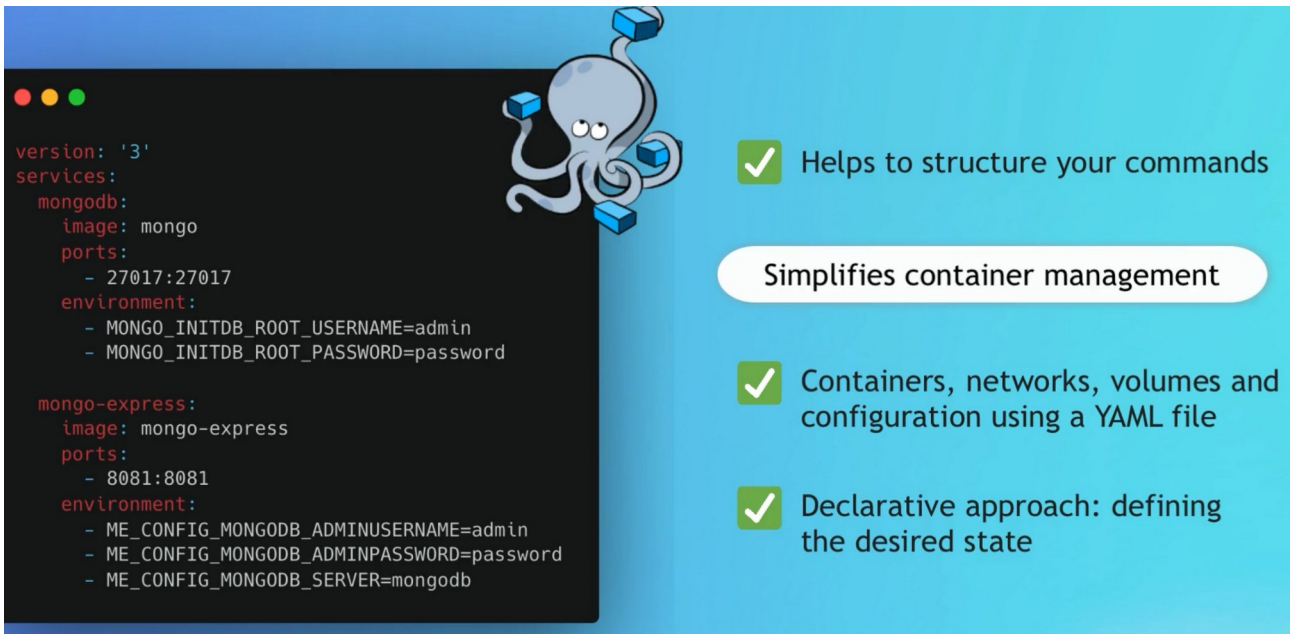
// serach on browser : localhost:8081 mongo db is running there .

Step 5 . Now connect node server with mongo Db container

These all running the container and setting up there Environment variables is some difficult to setup if there are multiple of them so there is a tool which automate all these .
That is DOCKER COMPOSE .








```
version: '3'
services:
 # my-app:
  # image: ${docker-registry}/my-app:1.0
  # ports:
   # - 3000:3000
 mongodb:
  image: mongo
  ports:
   - 27017:27017. // first is host , for container
  environment:
   - MONGO_INITDB_ROOT_USERNAME=admin
   - MONGO_INITDB_ROOT_PASSWORD=password
  volumes:
   - mongo-data:/data/db


 mongo-express:
  image: mongo-express
  restart: always //to make sure mongoexpress connect to mDB container when we start the project . This will tell mongo express to restart if it fails to connect to mongo DB .
  ports:
   - 8080:8081
  environment:
   - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
   - ME_CONFIG_MONGODB_ADMINPASSWORD=password
   - ME_CONFIG_MONGODB_SERVER=mongodb
```

```
  depends_on:
    - "mongodb"
volumes:
 mongo-data:
   driver: local
```

no network configuration is needed , by default docker compose sets up a default single network for you app

How to create a docker compose file ??
Using docker compose : command to use docker compose yaml
Docker-compose -f(which file I want to execute) mongo.yaml up(start all the containers which are inside mongo.yaml

Key note: when every u recreate a container the data is lost.
Docker volumes : enable persistence data even when u recreate the container .

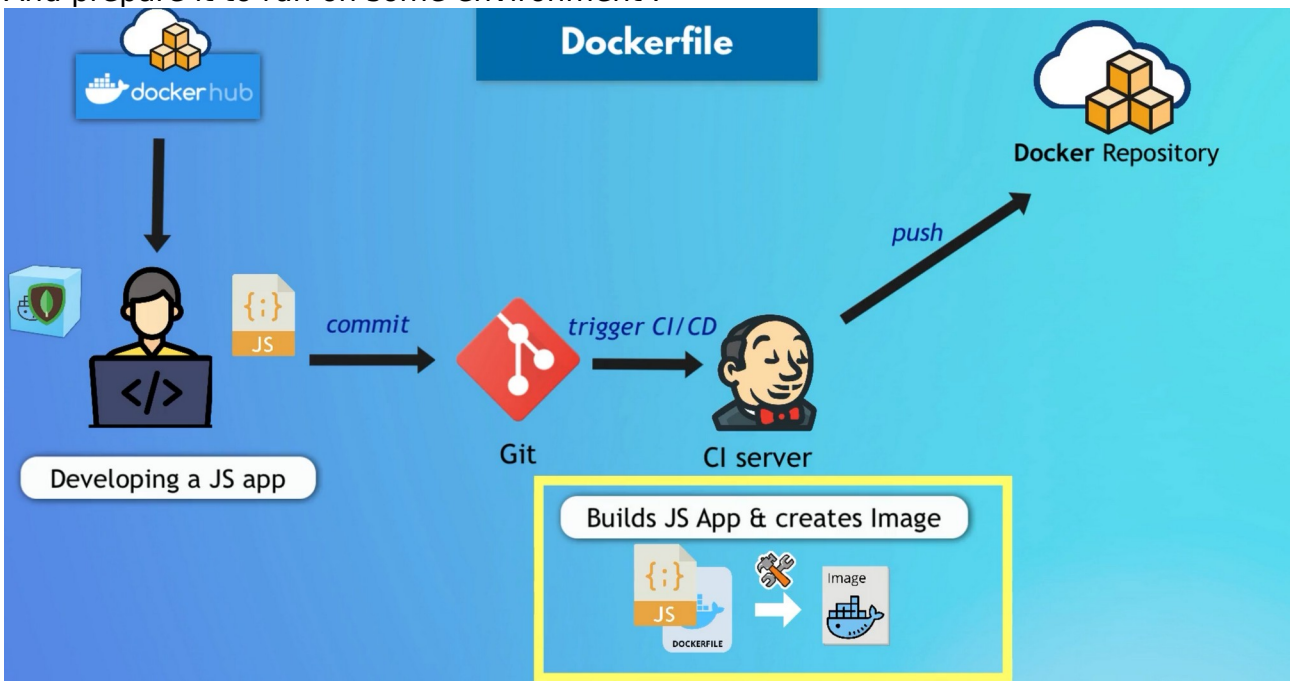
To start our application :
Use : npm install
Node server.js

Docker-compose -f docker-compose.yaml down // this will shut down all the container at once .

Building image with docker file :

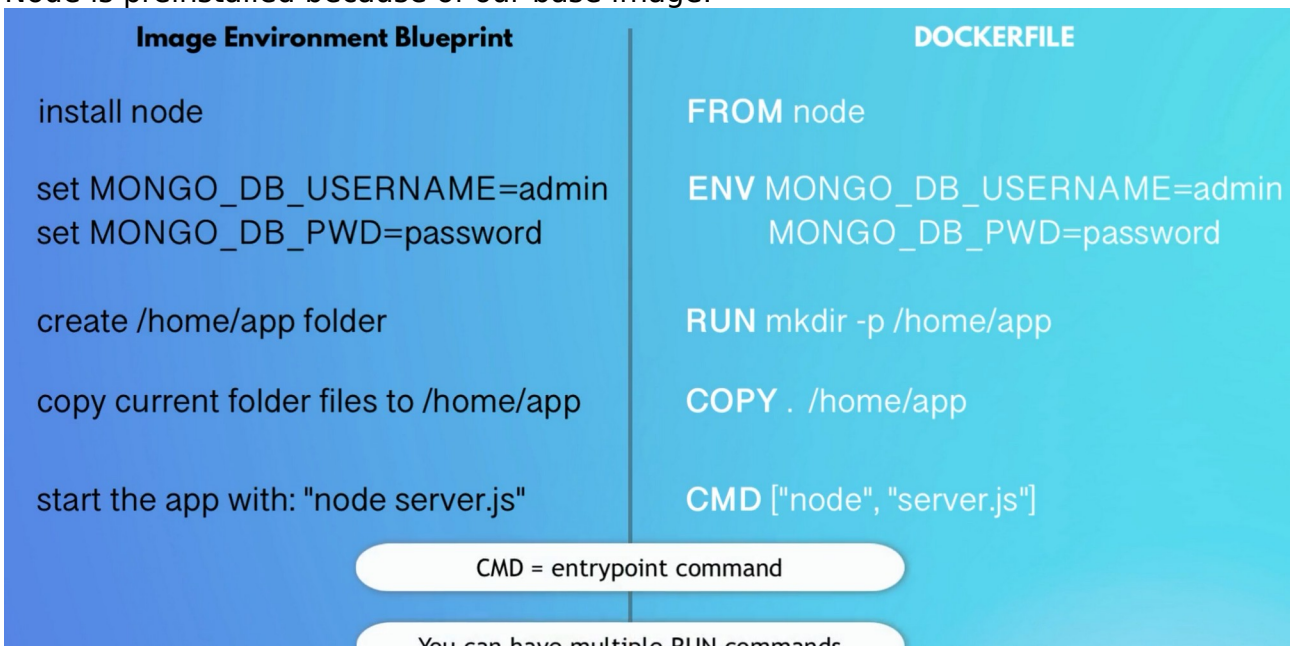
U have a application feature u have tested it how u will deploy it .
To deploy it the application should be packaged on its own docker container ;.

Now we are going to build a docker image from our js - node.js application .
And prepare it to run on some environment .



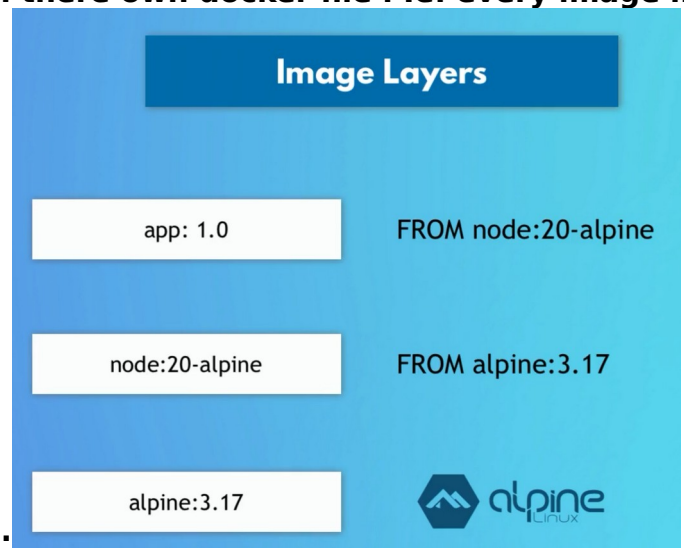
What jenkins do continuous integration : builds js app and create docker file to docker image.
And later we will push docker image to docker repository .
We have to do it manually

What is a dockerFile ??
In order to build a docker image from application we have to copy the content of the application (jar, war ,bundle.js) into docker file
Docker file is a blue print of building images .
First : install node image in docker file so that when we run the image we don't need to install node again in that container .
Can define environment variable in docker file it is a alternative
It is better to define EV externally in docker compose file because if something changes u can over ride it , instead of rebuilding the image.
RUN : using run execute the linux command .
All capital letters are part of docker file syntax.
;RUN mkdir -p/home/app : this directory will be created inside the container . (So when I start a container from this image /home/app directly will be created. Not on the host).
.Commands are applied on the container environment only .
COPY . /home/app : this copy commands executes on the host (. Is source and /home/app is the target , copy files from the host inside the image)
CMD ['node','server.js'] is always part of docker file . Execute a entry point for linux commands
Eg : we start app by : node server.js
Cmd vs run : run can be multiple but cmd only one (as it is the entry point)
Node is preinstalled because of our base image.



First line : **FROM node:13-alpine**

:docker file is a blue print for any docker image ie: every docker image there on docker hub are build on there own docker file . Ie: every image in docker hub has its own dockerfile.



Our image is build on top of image node:20alpine(taken from docker hub)
Node:20-alpine is build on top of image alpine:3.17
Alpine is a light weight linux image that we install node on top of it .
This is how images are build .

The name must be Dockerfile : now how do we build a image out of docker file ..
: docker build -t my-app:1.0(giving our image a name and a tag) (allocation of a docker file build a image using this docker file here if u are in same folder as docker file is just do . Dot).

```
vishaldwivedi@vishals-MacBook-Air-2 developing-with-docker % docker build -t my-app:1.0 .
```

After that u will get a image id .

This is what jenkins does :
It takes the docker file that we create , comment the docker file in the repository with the code , then jenkins will build a docker image based on that docker file . And to share the image with testers or other team of developers the image is pushed in the docker repository from there tester
Can pull the image from there or development server can pull image from docker repo .

// run the container : docker run my-app:1.0

:workdir /home/app : set a default working directory inside the docker container so that when the docker container starts of that image all the following commands that are in the docker file will be executed in the home/app directory .

RUN npm install : to make sure we have upto date version of that dependencies inside the container. While building the image.

When ever u abjust a docker file u have to rebuild the image as old image should not be rewritten .

// rmi is image deletion , rm is container deletion .

// run the container : docker run my-app:1.0
Docker ps : will see our app container is running .

To enter inside the command line terminal of running container : docker exec -it(interactive terminal) containerID /bin/sh
Ls : see which directory we are (root)

```
vishaldwivedi@vishals-MacBook-Air-2 developing-with-docker % docker logs b0b0b75dfd9f
app listening on port 3000!
vishaldwivedi@vishals-MacBook-Air-2 developing-with-docker % docker exec -it b0b0b75dfd9f /bin/sh
/home/app # ls
images          index.html      node_modules    package-lock.json  package.json    server.js
/home/app # cd ..
/home # ls
app   node
/home #  cd ..
/ # ls
bin   dev   etc   home  lib   media  mnt   opt   proc  root  run   sbin  srv   sys   tmp   usr
var
/ #
```

// to see the environment variable we set in dockefile : use command env .
// go to /home/app directory : ls /home/app: this will contains just the file I need for starting a application inside our container ie these:
images index.html **node_modules** package-lock.json package.json server.js

Since we have already done RUN npm install in our docker file it will automatically add node modules and packagaelock.json file every time we build the image. Now every thing we don't need in docker image are out side in the root folder .

Copy ./app /home/app : copy the app contains form the machine to the container home/app.

Using exit command u can exit form the docker container .

Rm: it will remove the container .
Rmi ; it will delete the image.

Project 2 : private docker repository :

Amazon repository manager : here u create a docker repo per image , don't have a repo where u push multiple images , rather than each image has its own repository . Docker repo : collection of related images with same name but different version , u save different versions of same image .

How we can push the image we have locally to amazon elastic container registry :

First u have to always login into your private repo.

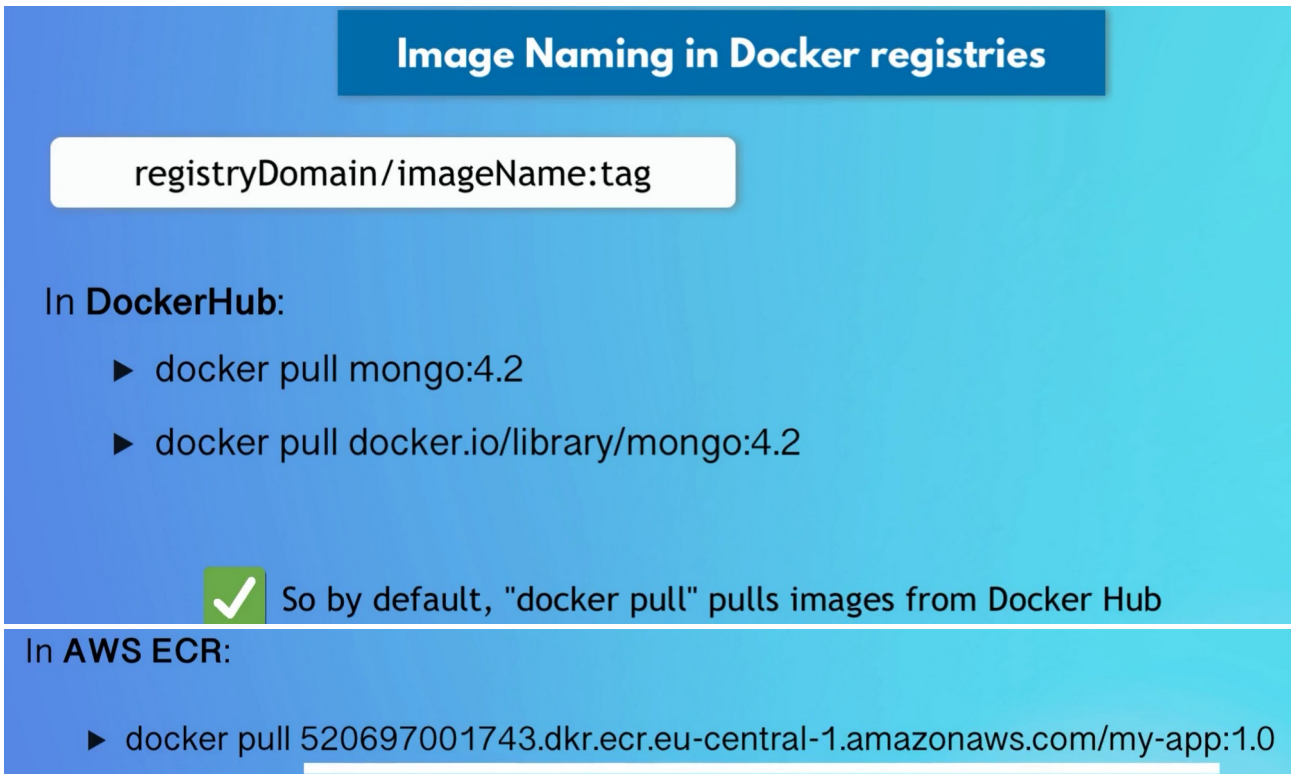
Login to aws : select my-app on aws ecr (view push commands) copy first and execute :
Prerequisite : To do this first aws cli need to be installed and credential configuration also.
https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html
Credential configuration:
https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html

Credentials identify who is calling the API. Access credentials are used to encrypt the request to the AWS servers to confirm your identity and retrieve associated permissions policies. These permissions determine the actions you can perform. For information on setting up your credentials, see Authentication and access credentials.

Other configuration details to tell the AWS CLI how to process requests, such as the default output format and the default AWS Region

Image naming in docker repository :

1.
2. Build your Docker image using the following command. For information on building a Docker file from scratch see the instructions here . You can skip this step if your image is already built:docker tag my-app:latest 339712731626.dkr.ecr.ap-southeast-2.amazonaws.com/my-app:latest



If we do : docker push my-app:1.0 // docker wont know which repo to push

We need to push in aws repo so we use tag :

1. docker tag my-app:latest 339712731626.dkr.ecr.ap-southeast-2.amazonaws.com/my-app:latest**docker tag = rename the image**

Now finally push : docker push nameOfTheImageANDtag

docker push 339712731626.dkr.ecr.ap-southeast-2.amazonaws.com/my-app:latest

Authenticate Docker with ECR Again:

Once the permissions are updated, try authenticating Docker with ECR again: (this is the challenges I face while pushing to ecr).

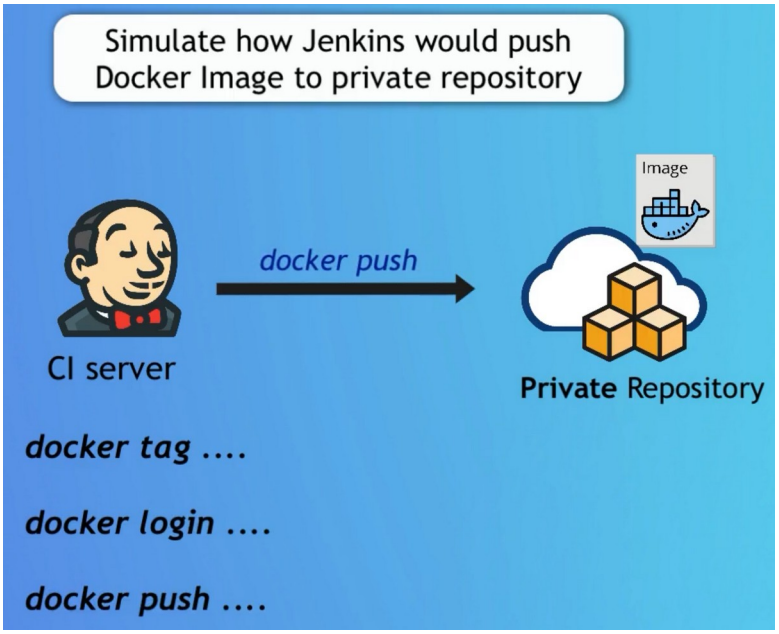
Command : aws ecr get-login-password --region ap-southeast-2 | docker login --username AWS --password-stdin 339712731626.dkr.ecr.ap-southeast-2.amazonaws.com

now we will be able to see the image on our aws ecr :

Image uri : name of the image using the repo name and tag

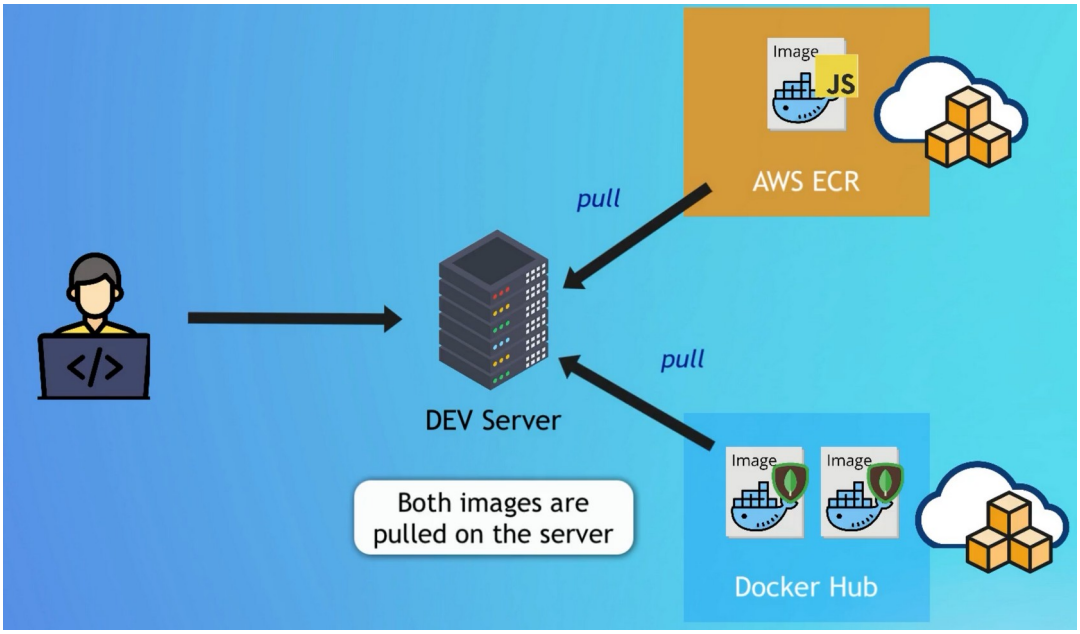
Make some changes to the app , rebuild and push a new version in aws repo.

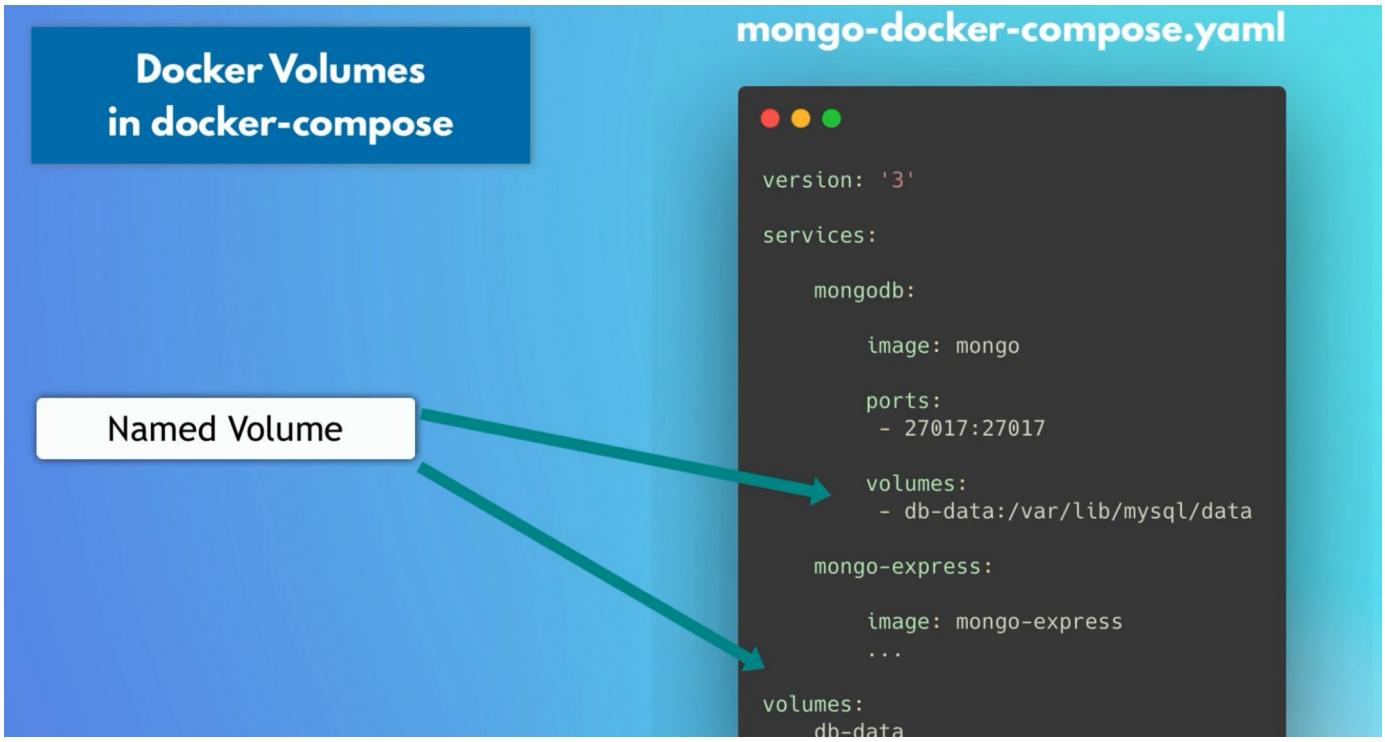
Delete some console logs . // now we have different version of the application .
:building new docker image : docker build -t my-app:1.1 .
:now I have to rename the image
Docker tag my-app:1.1 (image I want to rename) 339712731626.dkr.ecr.ap-southeast-2.amazonaws.com/my-app:1.1(same as the previous one as repo name and the address is the same)
Note : we have one repo for one image but for different versions so it should end up in the same repo.
:docker push (imageName:tag) 339712731626.dkr.ecr.ap-southeast-2.amazonaws.com/my-app:1.1
Note : some of the layers are there only only the needed changed layers are repushed. And now my repo has two versions.
Note : in aws each repo has a capacity of holding one thousand image versions.



We have simulated what jenkins automated do next we will see who to pull form the repo .

How to deploy the application : image form a private repo : deploy multiple container :





Now u have developed you application and u have created your own docker image.
In order to start you application on development server u would need all the container what that up that application environment.

```
version: '3'
services:
 app:
   image: my-app:1.0
   ports:
    - "3000:3000"
 mongodb:
   image: mongo
   ports:
    - 27017:27017
   environment:
    - MONGO_INITDB_ROOT_USERNAME=admin
    - MONGO_INITDB_ROOT_PASSWORD=password
   volumes:
    - mongo-data:/data/db
```

```
mongo-express:
  image: mongo-express
  restart: always
  ports:
   - 8080:8081
  environment:
   - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
   - ME_CONFIG_MONGODB_ADMINPASSWORD=password
   - ME_CONFIG_MONGODB_SERVER=mongodb
  depends_on:
   - "mongodb"
volumes:
 mongo-data:
  driver: local
```

```
app:
  image: my-app:1.0
  ports:
    - "3000:3000"
```

> In order to pull this image from repo (copy uri of image from the aws repo)

```
mongodb:
  image: mongo
  ports:
   - 27017:27017
```

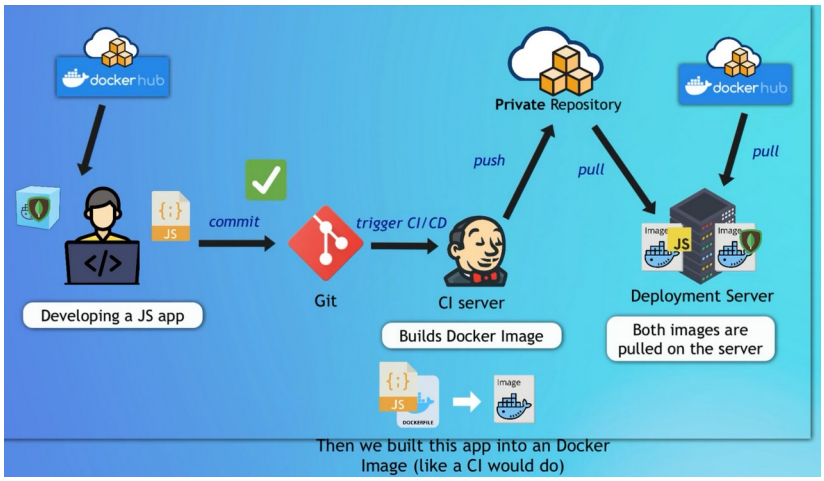
We are pulling these images from the dockerhub so we don’t need to specify the repo domain go mongo or mongoexpress.
But my-app if we are pulling it from private repo then we need to specify the repo domain and tag .

Docker compose in order to pull this image .the environment where u are executing this docker compose file has to be logged in to docker repo
Ie: dev server to pull the image from repo we need to do docker login to private repo . We don’t need docker login for docker hub as it is a public repo .

Our app is listening on port 3000 ie.container

3000:3000 port of the host):(container port)

::: SO THIS WOULD BE THE COMPLETE DOCKER COMPOSE FILE THAT IS USED IN DEV SERVER TO DEPLOY ALL APP INSIDE,.



DOCKER VOLUMES :

Docker volumes are used for data persistence . Eg: if u have databases or other stateful application u need docker volumes for that .

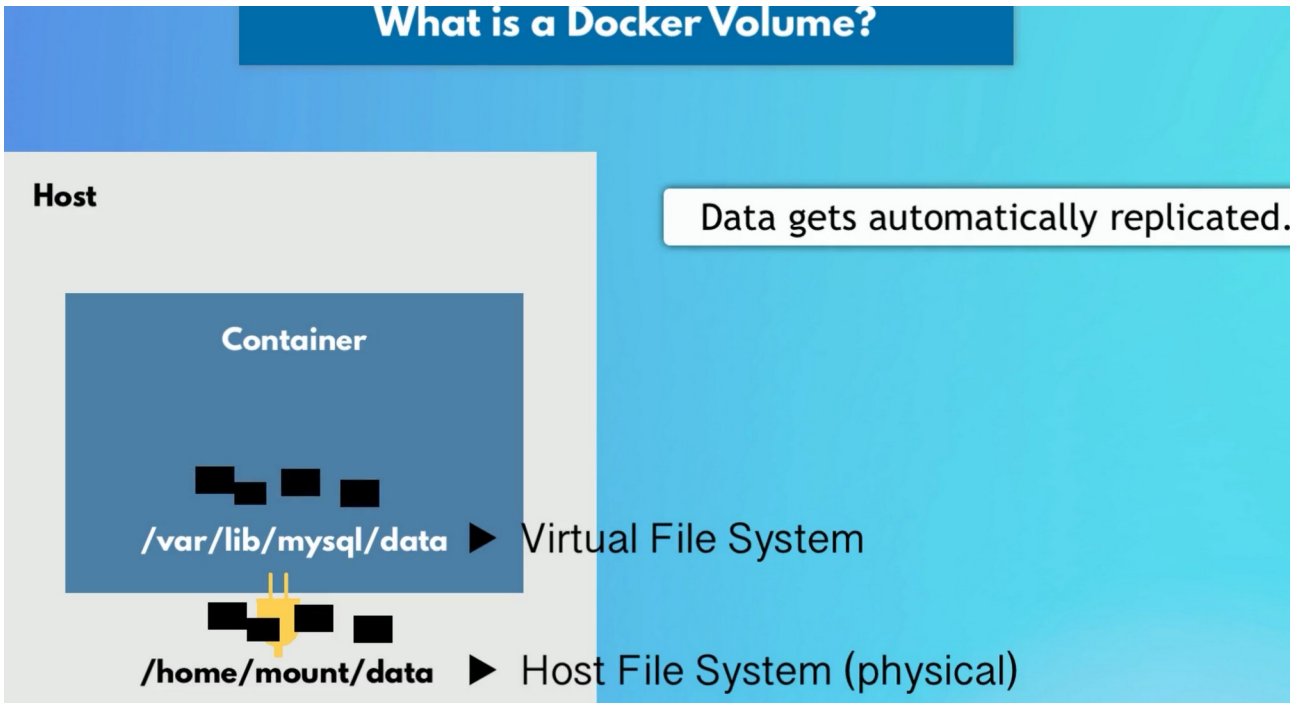
When do we need docker volumes ?

Container runs on the host container has virtual file system where data is stored , here there is no persistence data is gone when recreating the container and starts with fresh state.

On what host there is physical file system how volumes work we plug the physical file system with virtual file system of container .

When container write to its file system ,Data gets automatically replicated in the host file system or if I do change in host file system it automatically

replicated the container file system .



There are different type of docker volumes :
1. Docker run -v hostdirecty:containerdirectory
Called host volume = u decide where in the host file system the reference is made.
2.docker run -v containerDirectory :
Anonymous volumes = Automatically mounted to that specific folder.
3.docker run -v name:/var/lib/sql/data (upgrade of anonymous volume it specify name of the folder in host file system)
Named volumes . Now u can reference the volume just by its name.
So u don’t need to know the exact path.
Named volumes should be used in production .

DOCKER VOLUME IS PRACTICE :

This is a simple mongoldb application we gonna mount the data so that we don’t lost the data from db when restarting the container .

Start the docker-compose so that I can start mongodb and mongoexpress and network
:docker-compose -f docker-comopose.yaml up

Go to the app where server.js is
Then : npm install
Start the application :node server.js

Add named volume in docker-compose.yaml file .

```
volumes:
 mongo-data:// name of the volume reference
   driver: local// additional info for docker to create a storage in local file system
.
```

```
- 27017:27017
   environment:
    - MONGO_INITDB_ROOT_USERNAME=admin
    - MONGO_INITDB_ROOT_PASSWORD=password
   volumes:
    - mongo-data:/data/db
//hostVolumeName:pathinsideofthecontainer(where mongodb explicitly persists
the data
```

```
 mongo-express:
   image: mongo-express
   restart: always
   ports:
    - 8080:8081
```

We can see mongodb stores data in :
Docker exec -t containderID sh // execute or to inside mongodb
#ls /data/db : list all the data mongodb stores .
#exit:

So this is the path inside the container not on our host we need to reference on the volume here :

```
volumes:
    - mongo-data:/data/db //ostvoulename:pathinsideofthecontainer(where
mongodb explicitly persists the data
// we are basically attaching our volume on the host to mongo.
```

eg: for mysql : var/lib/mysql
For postgres:var/lib/postgresql/data

NOW the data is persistence when stores data in the container automatically copy to host and visa versa .

Docker volume location on our local machine :

For Windows : C:\ProgramData\docker\volumes

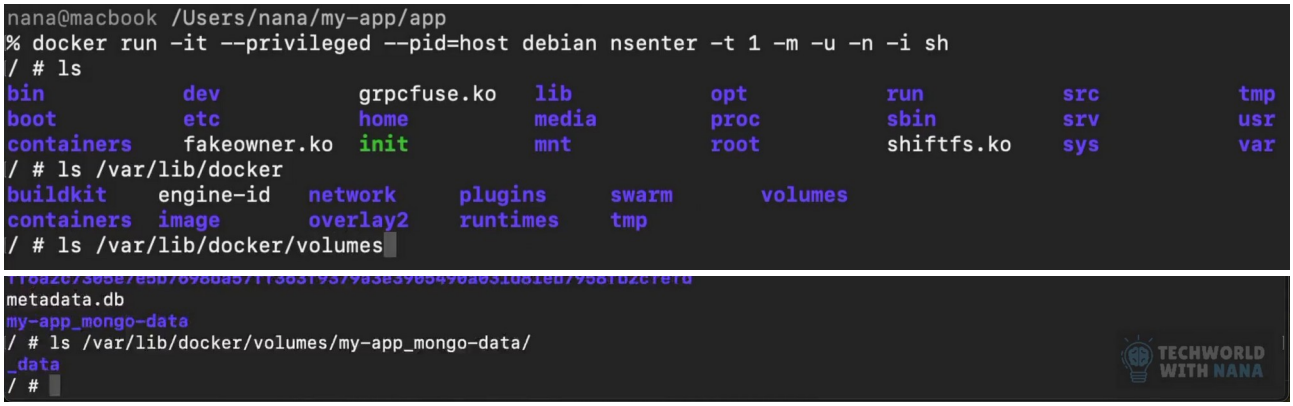
For linux:/var/lib/docker/volumes

For macOS: /var/lib/docker/volumes

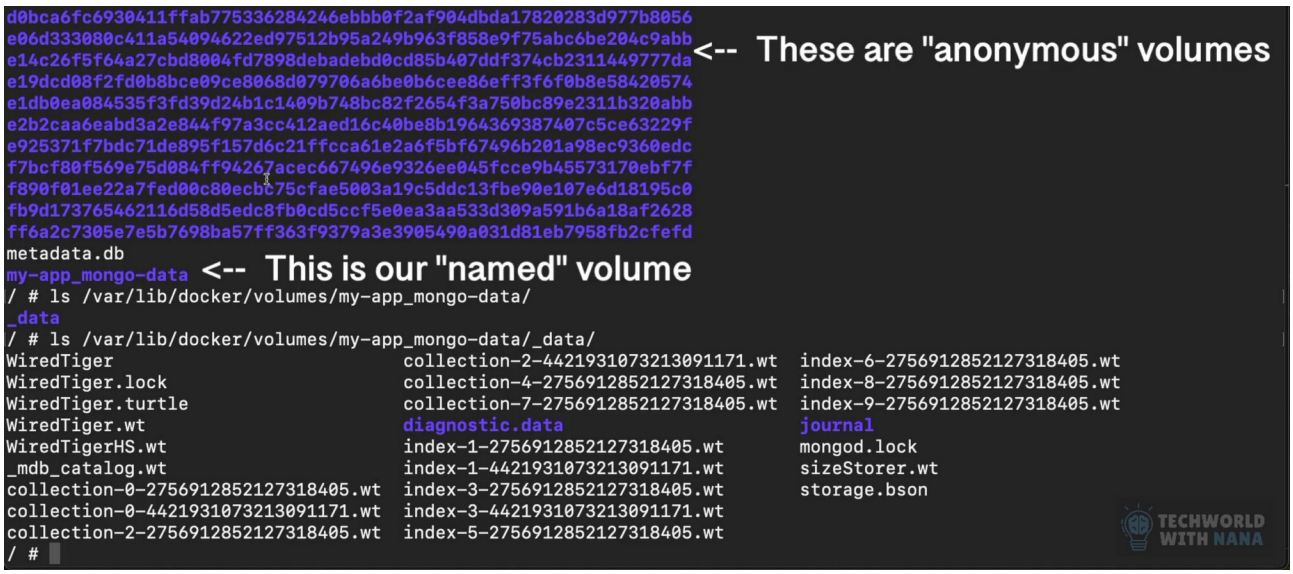
for Mac if u execute the command u will to find the volume these command u will not find volume there :

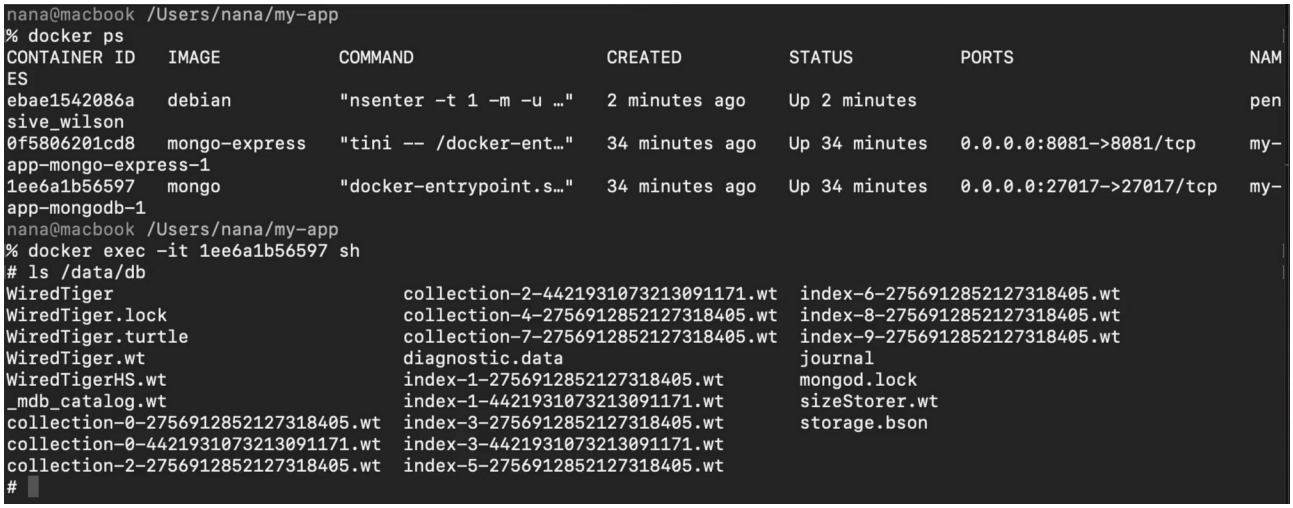
For Mac u need to first execute this command to enter the shell of the docker vm that the container is using :

:docker run -it —-privileged —-pid=host debian nsenter -t 1 -m -u -n -i sh



This my-app_mongo-data: is the name we defined int he docker compose.yaml file mongo-data





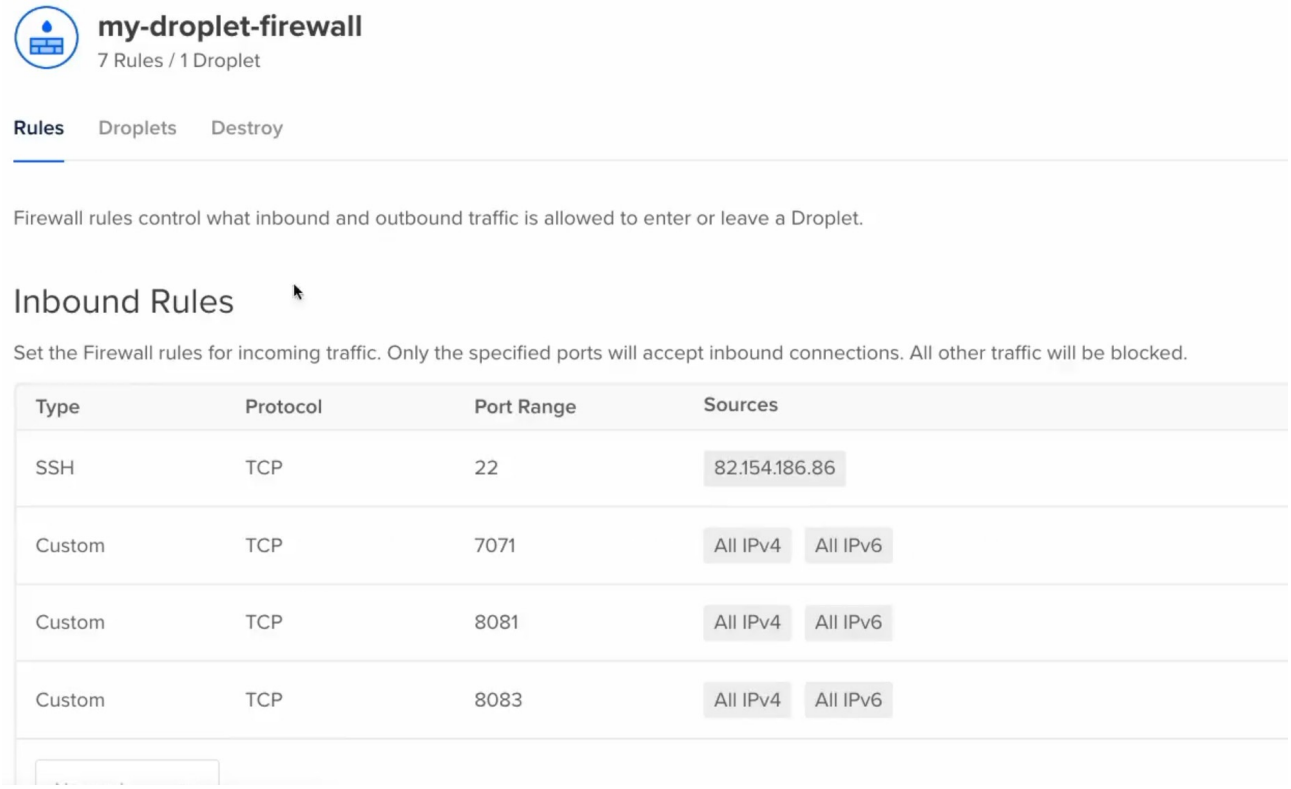
First is the volume data located inside the vm of docker(persistence volume) , second is the container data both data is same .
By doing this we can get to see all our data again during restart of the container .

Run nexus as docker container >

Install nexus in digital ocean drop lit .
**Step 1 ;Install java **
2 .Download nexus package .
3.untar nexus package
4.create nexus user .
5. Give permission to user
6. Run nexus as nexus user.
7. Start of nexus manager.

To use nexus faster we can use nexus as a docker container in digital ocean droplet.
1.create a new droplet.
2.connect using ip address provided

3. Name it as docker-nexus.
4. We need to configure firewall so that the correct ports are open for nexus and docker .
5. Add docker-nexus server to the firewall on port 22



These are the firewall config u need to set up .

7. Now connect to ssh port 22 : ssh root@209.38.237.10

8. And we are on the server.
9. We only need docker to be installed in this server.
10. Apt update. //package manager
11. Snap install docker :// install latest version of docker .
12.go to docker hub look for nexus3 this is the image we need.
13.

Persistent Data

There are two general approaches to handling persistent storage requirements with Docker. See Managing Data in Containers for additional information.

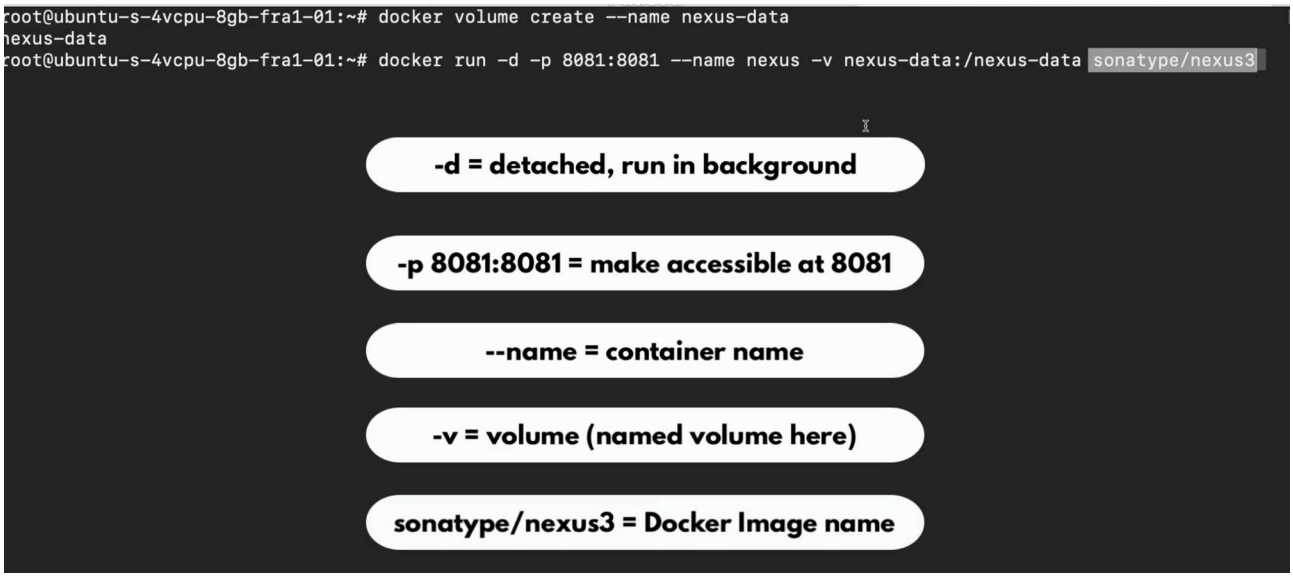
1. *Use a docker volume*. Since docker volumes are persistent, a volume can be created specifically for this purpose. This is the recommended approach.

```
$ docker volume create --name nexus-data
$ docker run -d -p 8081:8081 --name nexus -v nexus-data:/nexus-data sonatype/nexus3
```

2. *Mount a host directory as the volume*. This is not portable, as it relies on the directory existing with correct permissions on the host. However it can be useful in certain situations where this volume needs to be assigned to certain specific underlying storage.

```
$ mkdir /some/dir/nexus-data && chown -R 200 /some/dir/nexus-data
$ docker run -d -p 8081:8081 --name nexus -v /some/dir/nexus-data:/nexus-data sonatype/nexus3
```

If the container died and we need to create a new one then all the nexus data will be gone.



Above command to start nexus container .
8081(it will run on the server port 8081):application is running on this port inside container(8081))
Mount that data name nexus-data(named volume save this is local server as persistent data):/nexus-data(inside the nexus container)

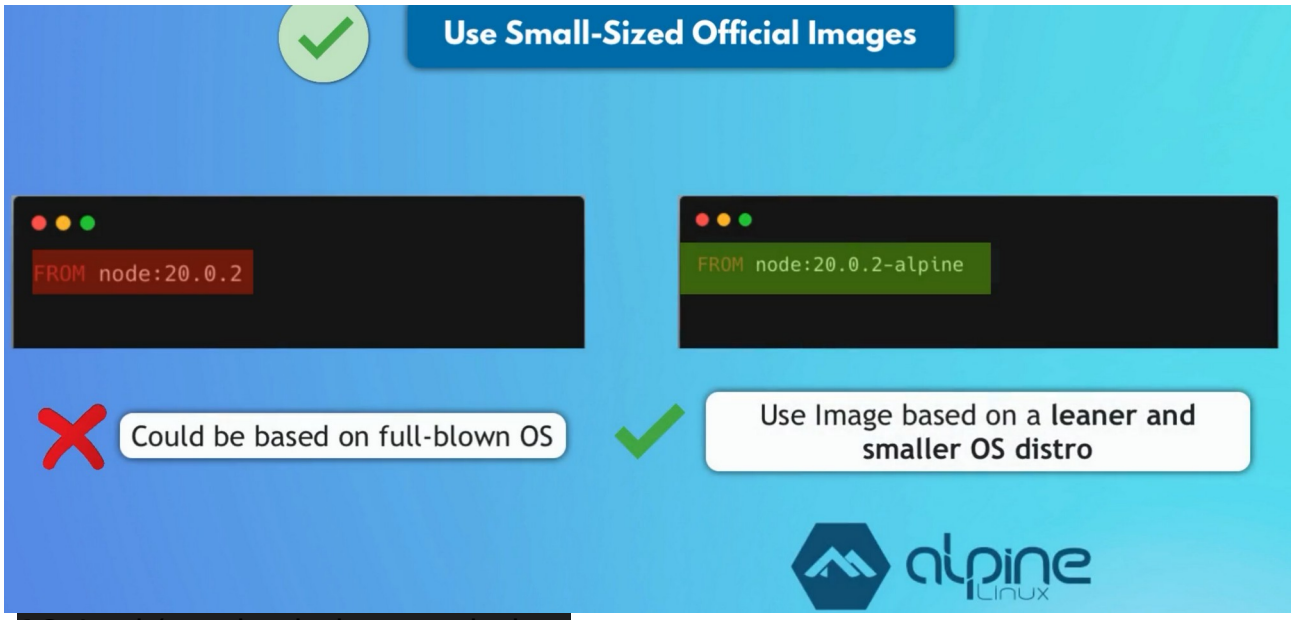
14. Apt intall net-stat : to see which app will running on which port.
15. Netstat -lnpt :
16. Docker ps : container is running port is running .
17. That the droplet ipaddress and the port : a.d.c.d:8081
18.we gonna see the nexus repo ui.
19. We have deploy nexus as a docker image on the server.
20.this is much faster then deploying without doceker.

21.because we only need to install docker runtime .
22. It is a bad practice to run services using root user.
23. Docker ps-> take id of container -> docker exec it xxfasdfa /bin/bash
$ means that it is running as a non root user (insider the container)
whoami(command will that which user r u)->nexus.
// docker volume ls : to see the named_volume in server file system .
// docker inspect namedfvloume: to see the additional info. -> here u well see the mount path the actual location of volume where we see all the data of nexus here.

24. U can look into docker container file system because all this data is in there also.
Docker exec -t adfadf /bin/bash
25. Go to the home directory : and do ls : u will see nexus-data there is same volume data insider the server and inside the container.

Create docker repo on nexus and push our images to nexus docker repo .

Create new repo -> choose docker hosted on nexus
How to install nexus first:
1 . Create droplet with 48$ server as nexus need higher capacity
2. Attach firewall rules so that it is access in port 22 ssh.
3. Intall java 8 as it need nexus.
4. Move to cd/opt folder.
5.download nexus package in this folder.
6. Wget pastTheLink
7. Untar the package : tar -zxvf
8. Two folder which will be untar from this package sonatype-work and nexus-xyz(contain runtime and app for nexus).
9.ls nexus-xyz
10 ls sonatype-work/nexus3/(contain own config for nexus and data)
11.nexus know the local of sonatype-work directory from its propertyies file .
12. Sonatype-work contains the logs of all ip addresses that are accessing the nexus , logs of nexus application itself, subdirectories depending on your nexus configuration .
And u can use this sonatype-work folder to backup all the nexus data.
13 . Services should not get the root user privileges.
Best practice is to create own user eg nexus .
14. Adduser nexus
15. Change owner of nexus and sonatype-work to nexus:nexus(user and group)



16. Ls -l (to check the permission)
17.now set nexus configuration so that it will run as nexus user .
18.vim nixus-zyx/bin/nexus.rc
19. Run as user = “nexus” save it
20: to run the nexus service we need to switch from root to nexus user . : su - nexus
21:/opt/nexus-3.52.2-01/bin/nexus start
22: service started
: ps aux | grep nexus : to make sure nexus is running
: netstat -lnpt : will show active internet connections .

23.add firewall rule in droplet : open 8081 tcp port
24 : take the ip address of the droplet x.w.x.a:8081 to access nexus ui .

Now creating a docker repo on nexus :

1.Create new repository : choose docker hosted we will be pushing our own docker images to this repository .
2.select blob store -> Mysore. Which we created in nexus part.
3.u will have url of repo used to push the images to this repo.

Create user role for docker repo ->

In order to push our images to this docker repo we need to login as nexus user which has access to docker host repo.
4. Create a new role what has a privilege to access docker host repo.
5. Add privalage nx-repository-view-docker-hosted-add
5. Add http : 8083 port for docker login in docker hosted repo.

6.

```
root@ubuntu-s-4vcpu-8gb-fra1-01:~# netstat -lnpt
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 127.0.0.54:53           0.0.0.0:*               LISTEN      605/systemd-resolv
tcp        0      0 0.0.0.0:8083            0.0.0.0:*               LISTEN      1639/java
tcp        0      0 0.0.0.0:8081            0.0.0.0:*               LISTEN      1639/java
tcp        0      0 127.0.0.1:36523         0.0.0.0:*               LISTEN      1639/java
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN      605/systemd-resolv
tcp6       0      0 :::22                   :::*                    LISTEN      1/init
root@ubuntu-s-4vcpu-8gb-fra1-01:~#
```

This is a droplet where nexus is running we can see both port 8081 and 8083 are running .

7.now to access this port we need to open this port in droplet firewall configuration

8.this port 8083 will allow to talk to docker repo.

9.when we do docker login we get a token for authentication from nexus docker repo for client that token is stored on my local machine :in
my home directory user/vishal %cat ~/.docker/config.json : if u all pull and push the image from the repo than u need to login and token will be take for the config.json

The first time u log in into any dockerrepo , dockehub, aws,or nexus this config.json file will be generated with authenticated token this token is used every time u interact with that repo.

For that : we need to go to realm in nexus :

Select :docker bearer token realm and activate it as inactive by default

10: now we need to config docker because by default docker accepts only client requests when every execute commands like : docker login, docker pull , docker push .

We need to tell docker to use the docker hosted repo now

So go to docker setting in docker desktop on top :

Go to docker engine and paste : “insecure-registries”:[“157.230.22.19:8083”] : now docker engine will allow us to send the client request to that registory with http

11: docker login ipaddressofdockerHostedRegistoryINnexus:portoftheDockerRepo(8083).

12.it will ask for your nexus user name and password

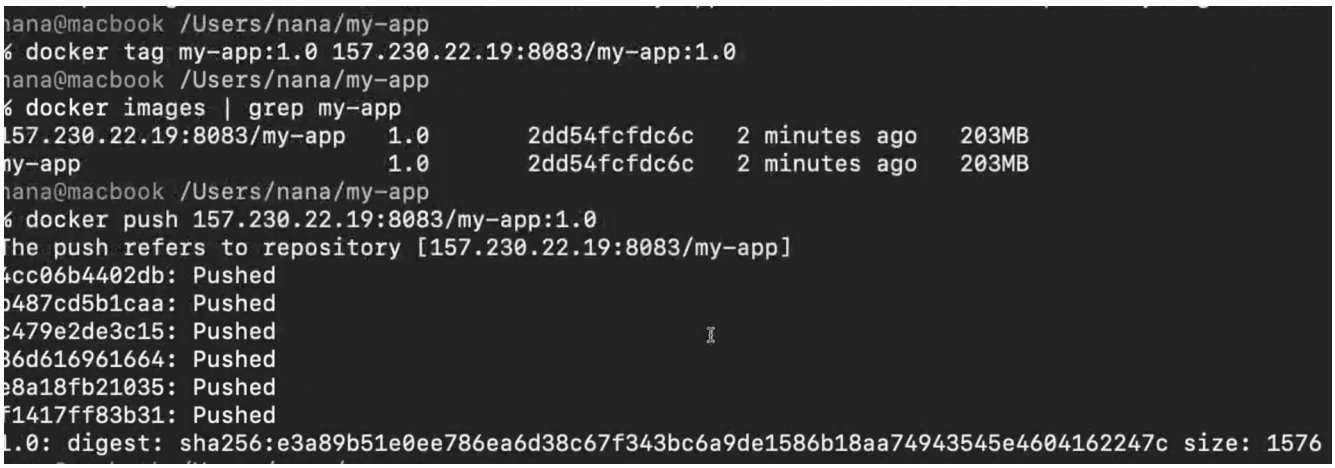
Not all the authentication is saved in config.log file so that if I log in again it will ask for password

13. Now push images to nexus docker repository .

Change to the director what contain docker file

14: dock -t my-app:1.0 .

15.// retag it to include the address the name of the repo itself



when we execute docker push we dont expicity specify the repo it need to pushed to but rather we inplicitly include in the name of the image it self meaning

Docker tag my-app:1.0 157.230.22.19:8083/my-app:1.0

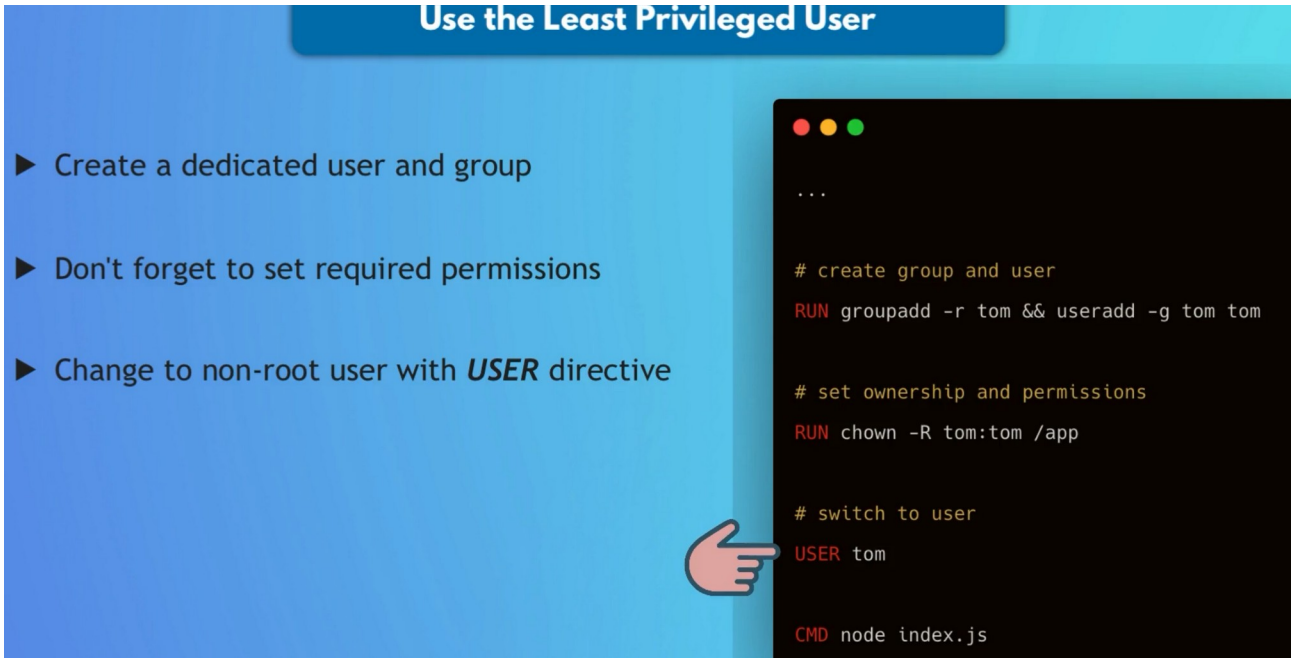
docker tag my-app:1.0 157.230.22.19:8083/my-app:1.0

1. **docker tag**: This is the Docker command used to create a new tag for an existing image.
2. **my-app:1.0**: This is the source image that you want to tag. It consists of:
   my-app: The name of the Docker image.
   1.0: The tag or version of the Docker image.
1. **157.230.22.19:8083/my-app:1.0**: This is the new tag you are creating for the image. It includes:
   157.230.22.19:8083: The Docker registry address where the image will be pushed. This includes:
      157.230.22.19: The IP address of the Docker registry.
      8083: The port number of the Docker registry.
   /my-app:1.0: The name and tag of the image in the registry. This part is the same as the source image, but prefixed with the registry address.

Explanation of Docker Registry
A Docker registry is a place where Docker images are stored and managed. Docker Hub is a public registry, but you can also use private registries such as Nexus Repository, Artifactory, or others.
Docker Push Command
When you push a Docker image to a registry, you include the registry address in the image name, so the **docker push** command knows where to send the image. Here's the command:
sh

Copy code
docker push 157.230.22.19:8083/my-app:1.0
Let's break it down:

1. **docker push**: This is the Docker command used to push an image to a registry.



2. **157.230.22.19:8083/my-app:1.0**: This specifies the image you want to push, including the registry address. It consists of:
   157.230.22.19:8083: The Docker registry address.
   /my-app:1.0: The image name and tag.

16: Now go to nexus docker-hosted repo :-> browse :

U can see the image and its layers pushed in repository

Fetching docker images from nexus:

We can fetch available docker images from the repository from nexus api .

To to my-app directory : curl -u username:password(nexus user and password) -X GET (specify the endpoint)'http://157.230.22.19:8081/service/rest/vi/components?repository=docker-hosted'

```
curl -u username:password -X GET 'http://157.230.22.19:8081/service/rest/v1/components?repository=docker-hosted'
```

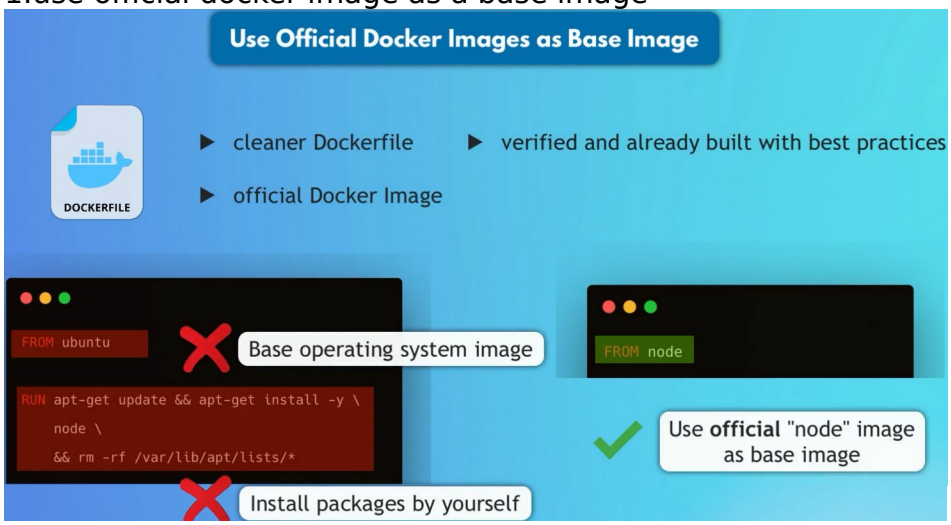
Breaking Down the Command

1. **curl**: This is a command-line tool used to transfer data to or from a server using various protocols (HTTP, HTTPS, FTP, etc.).

2. **-u username:password**: This option is used to provide the username and password for authentication with the Nexus server. Replace **username** and **password** with your actual Nexus username and password.
3. **-X GET**: This specifies the HTTP method to use. **GET** is used here to fetch data from the server.
4. **'http://157.230.22.19:8081/service/rest/v1/components?repository=docker-hosted'**: This is the URL of the Nexus REST API endpoint you are querying. Let's break this down further:
    - **http://157.230.22.19:8081**: The base URL of your Nexus server. Replace **157.230.22.19** with the actual IP address or hostname of your Nexus server, and **8081** with the port number if it's different.
    - **/service/rest/v1/components**: The specific endpoint of the Nexus REST API that retrieves components (in this case, Docker images).
    - **?repository=docker-hosted**: The query parameter specifying the repository from which to fetch the components. **docker-hosted** is the name of the Docker repository in Nexus. Replace **docker-hosted** with the actual name of your Docker repository if different.

Docker best practices :

1.use official docker image as a base image



2. This node image will always use the latest tag(version) why this is bad ?
You might get different docker image version as previous build
Might break stuff
Latest tag is unpredictable
So there fore don’t use the random latest tag but use the specific fixate the version ,the more specific the better . This will make the transparency on which version u are using in your docker file.

3.based on different operating system distorts which one do you choose ? Does it matter?
Smaller image means lesser storage space in repo as well as server . Transfer the image faster , when pulling and pushing the image server to repo or repo to server.

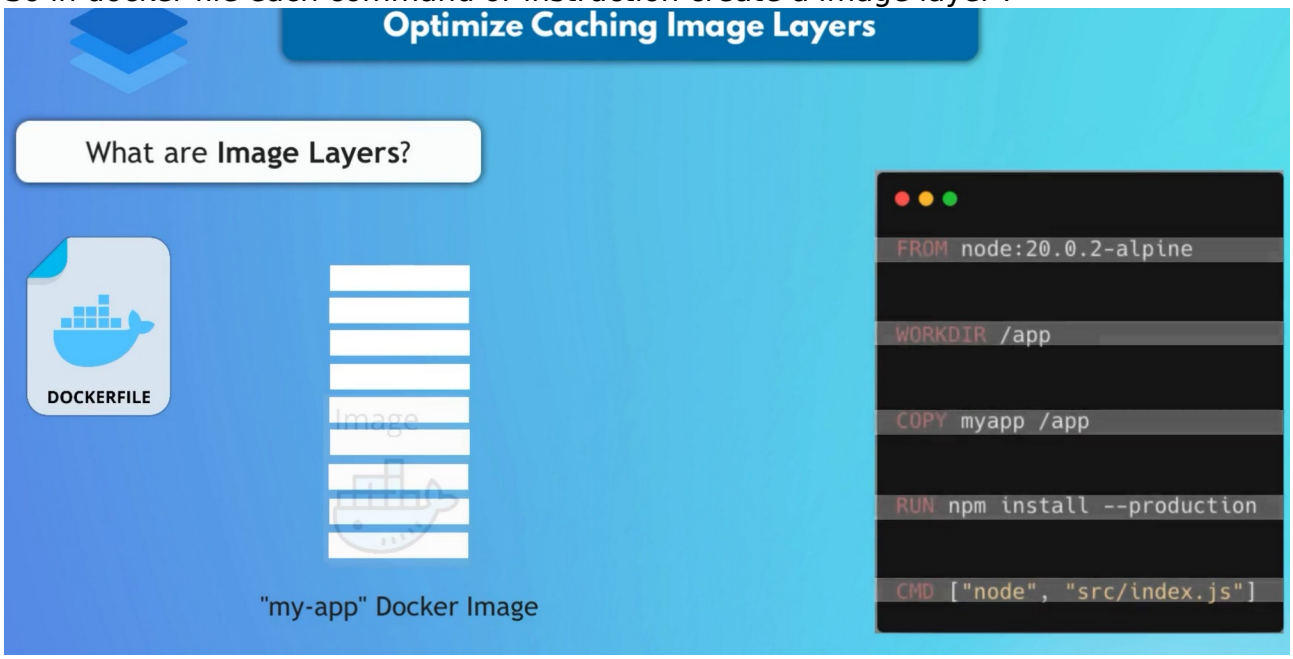
Full blown operating system distro eg: ubuntu, centos, debian etc
Have large image size , more security vernabilities , introduce unnecessary issues from the start.

Learner operation system distro .
Only bundle necessary utilities (eg: busybox instead of GNU core units) , minimise attack surface .

Use alpine instead : light weight linux distro , security -oriented , populate base image for docker container .
So best practice is if your system does not required any specific utilities , choose leaner and smaller images .

4: optimize cache layers while building the image .
What are image layers and what does caching in image layer means >?
Docker image are build on base of docker file :

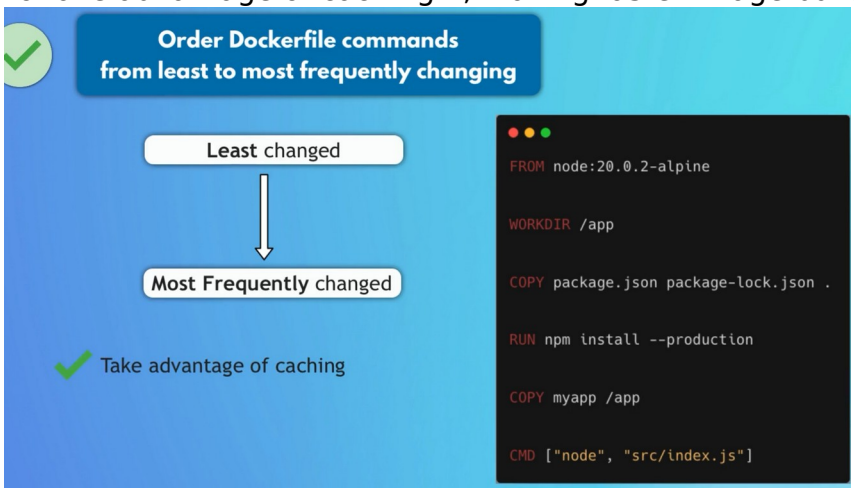
So in docker file each command or instruction create a image layer .



Once building own aplicaiotn : docker build -t my-app:1.0 : u can see the images : :using : docker history my-app:1.0: this will show u the image layers with the corresponding command which created the layer.

Docker caches each layer , saved in local file system , if nothing has change or u have rebuild your image it will reuse form cache . Result in fast image building , if I pull the image version of the same application it will download the only added layer making faster download .

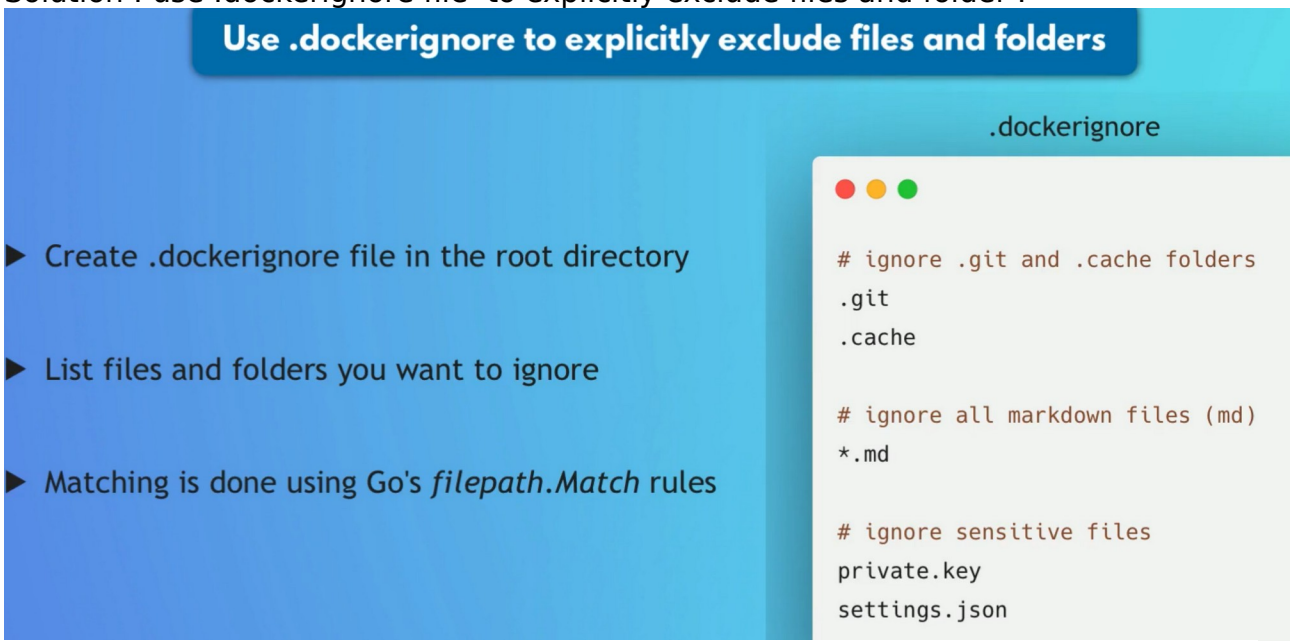
Order docker file commands from least to most frequently changing >
Least changing on top —-> most changing on bottom .
To take advantage of caching. ., making faster image building and fetching .



5 : we don’t need every thing inside our image :
We don’t need the autogenerated folder s eg : large , build or readme files .

How do we exclude these files to be download with my image ?
We want out image size to be smaller , prevert unintended secret exposure .

Solution : use .dockerignore file to explicitly exclude files and folder .



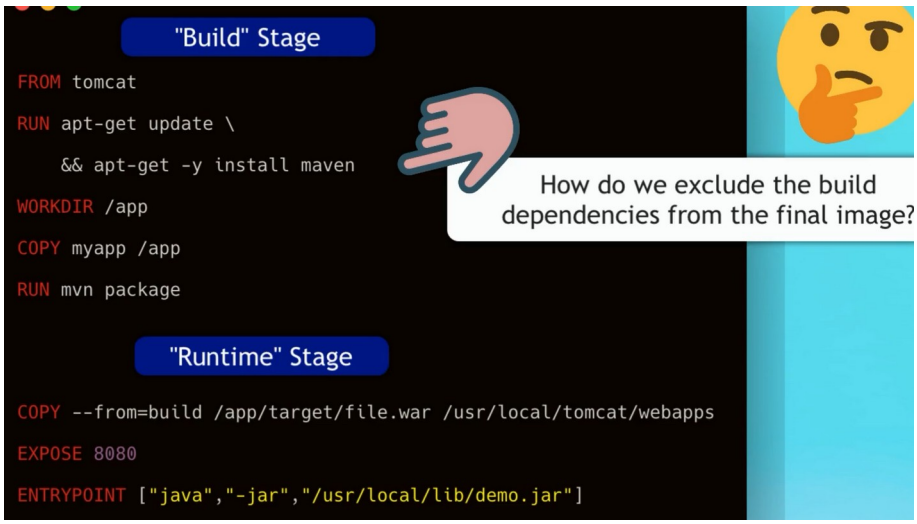
6: content u need to build the image but u don’t need that in your final image to run the app.
Eg : u just need developing tools , build tools , test dependencies .only while building docker image from docker file .

Having all these in our final application will increase image file and increased attack surface .
Eg: pakage.json , pom.xml. or any other dependencies file . How ever when the dependieces are installed we don’t need these files in the image itself to run the application .

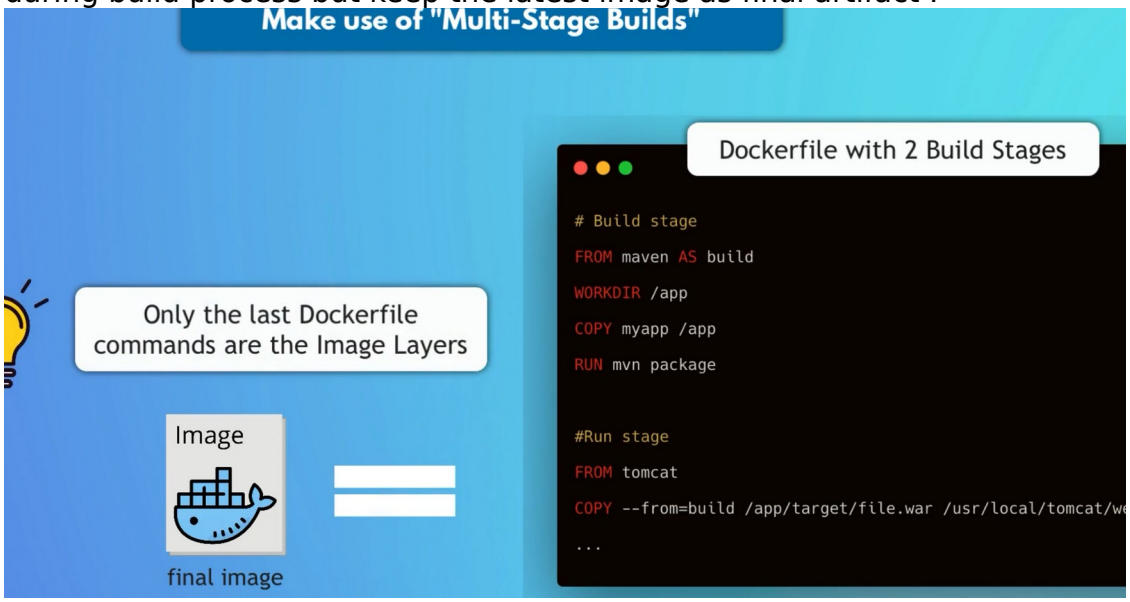
Eg: while build java application we need idk to compile java source code .and jdk is not need to run the java application itself .

Or using maven or gradle build tools to build the java application . These are also not needed in the final image

So who do separate the build state from the runtime state how do we exclude the build dependencies from the final image while still having available while building the image ?



Solution : u can use multi stage builds> it allows u to use multiple temporary image during build process but keep the latest image as final artifact .



7.when we create these image and run it as container which os user will be used to start the application inside .

By default it uses root user > which is a bad security practice .
So create a dedicated user and group in the docker image

Some images already bundles the generic user eg : node js use user name node which u can use to run application inside the container .

8: how to scan the image that u build have any security vulnerabilities or not >??
Solution : scan the image for security vulnerabilities :
Using command : docker scout cves my-app:1.0

U need to be logged in to docker hub in able to scan your images.
On the background docker will scan in the database of known vulnerabilities .