Introduction To Docker

What is Docker?

Docker is an open-source containerization tool. It allows developers to package their applications and dependencies into containers, which can be easily deployed across different environments.

It provides a container-based virtualization system that allows developers to package their applications into isolated containers, which can then be deployed on any operating system or cloud platform.

With Docker, developers can quickly and easily create, test, and deploy applications without having to worry about compatibility issues or hardware requirements.

Why Use Docker?

- **Isolation:** Containers isolate applications and their dependencies from the underlying system, averting conflicts and ensuring consistency.
- **Portability:** Docker containers can run on any system that supports Docker, making it easy to move applications between different environments.
- **Scalability:** Docker makes it simple to scale applications horizontally by running multiple containers, improving performance and reliability.

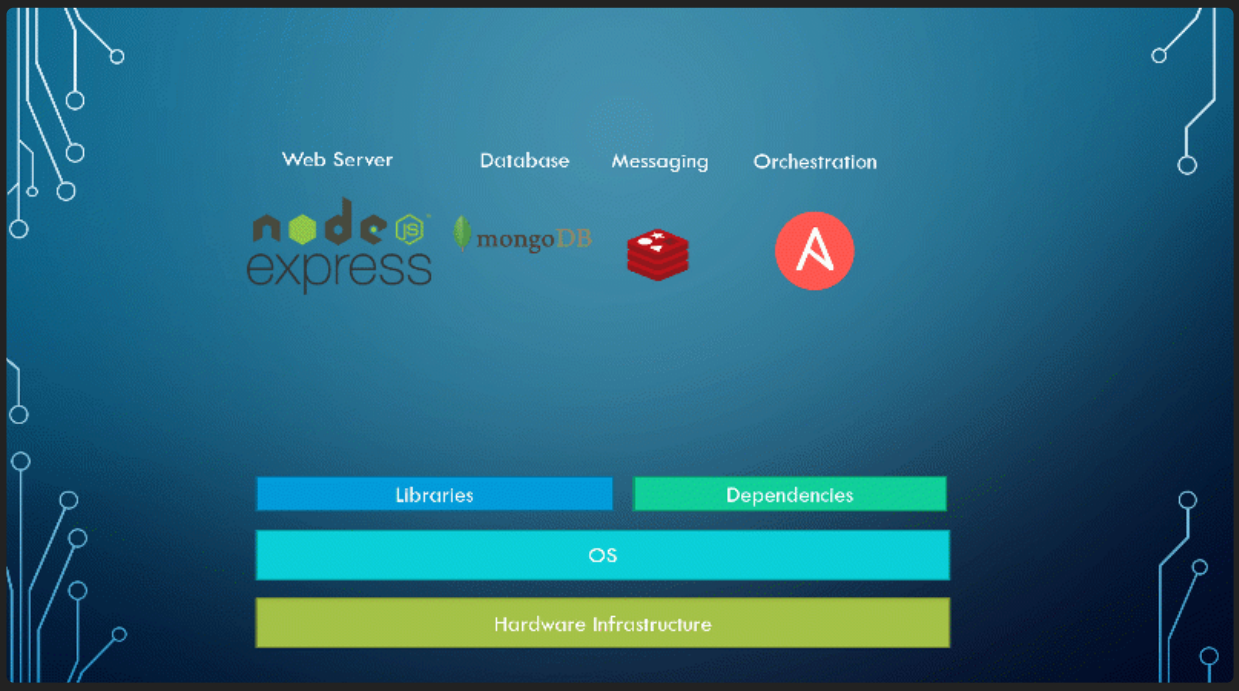
What are containers?

A container is a package containing all the necessary components required to run an application. It makes it super easy to move your app from one environment to another. You don't have to stress about compatibility issues or anything like that. For instance, a developer can quickly move the app from their laptop to a production server without any hassle.

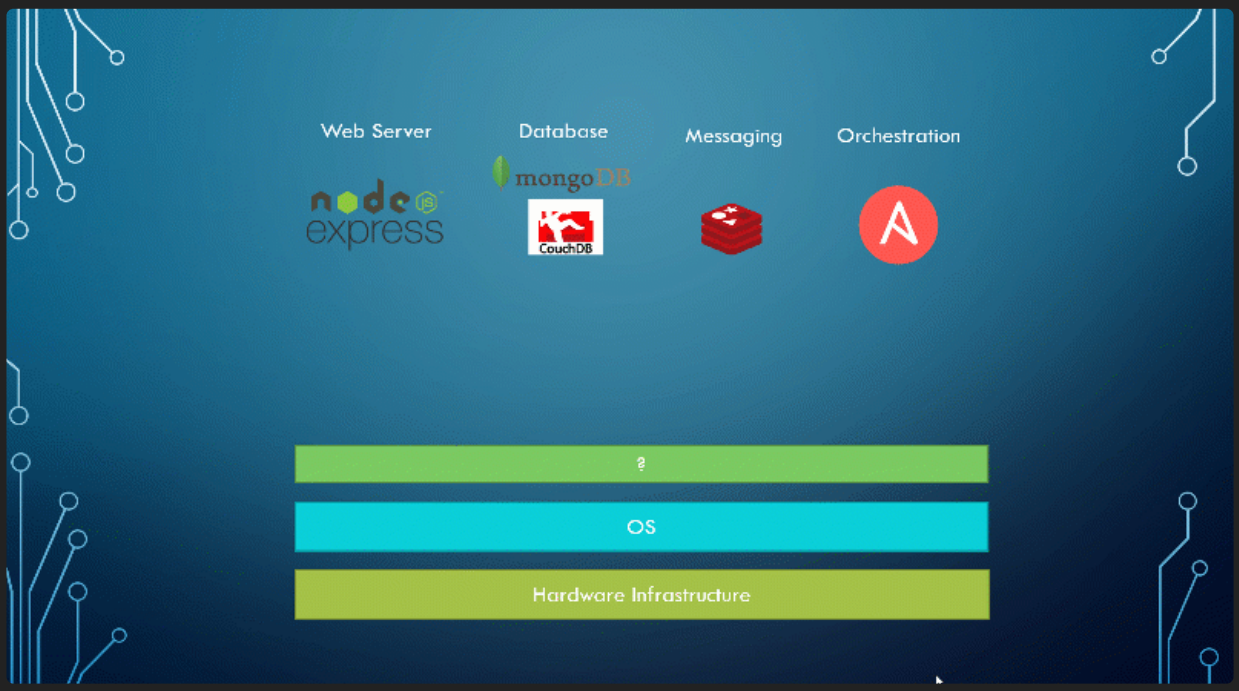
Imagine you're developing a web app with Python. You're using a particular version of Python and a set of libraries to run it on your laptop. When it's time to deploy it on a production server, all you need to do is take the container from your laptop and move it to the server. The container bundles all the dependencies, so you can be sure that the app will run the same way on the server as it did on your laptop. This holds true even if the server has different libraries installed or runs a different version of the operating system.

Why We Need Containerization?

Before containers project look something like this



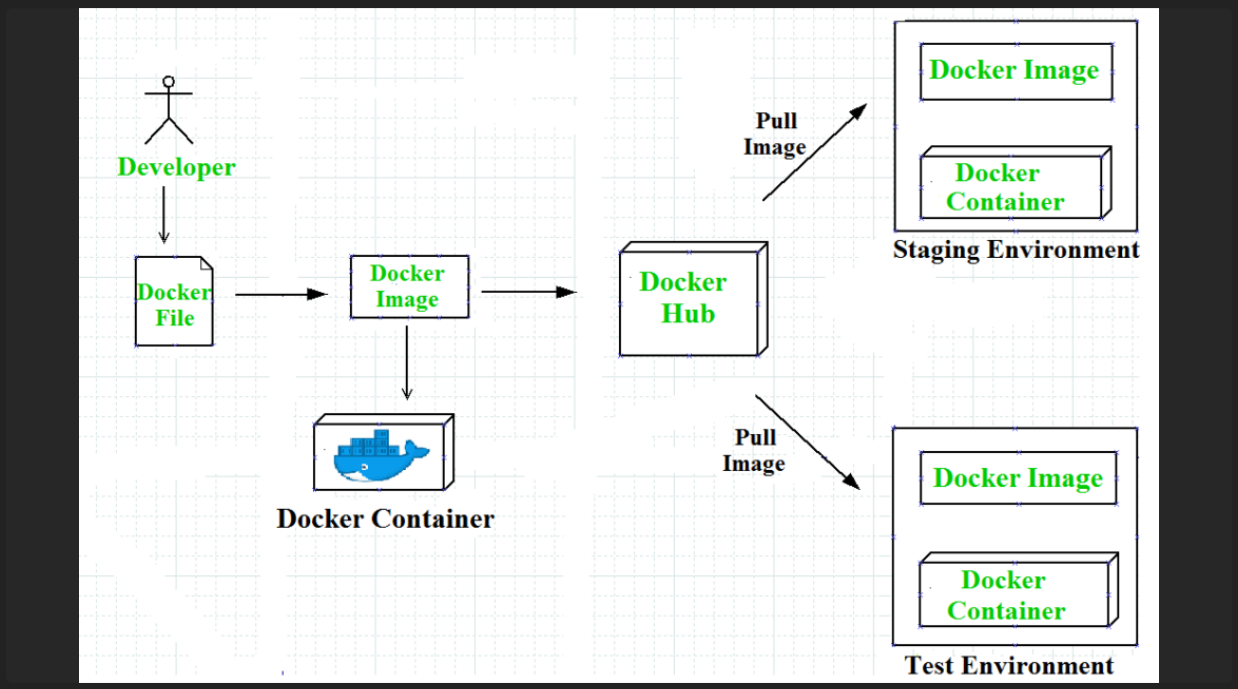
After containerization it looks like



Docker Container: A container is an isolated environment for your code. This means that a container has no knowledge of your operating system or your files.

It runs on the environment provided to you by Docker Desktop. This is why a container usually has everything that your code needs in order to run, down to a base operating system. You can use Docker Desktop to manage and explore your containers.

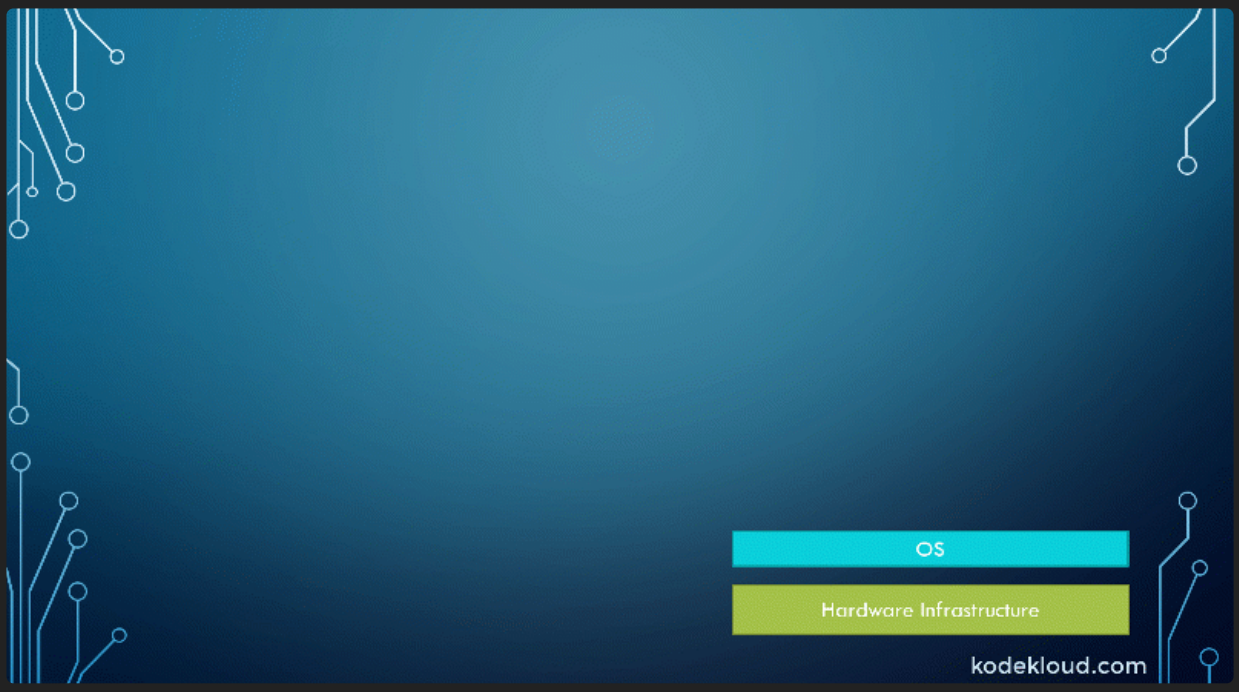
Docker Containers are runtime instances of Docker images. Containers contain the whole kit required for an application, so the application can be run in an isolated way. For eg.- Suppose there is an image of Ubuntu OS with NGINX SERVER when this image is run with the docker run command, then a container will be created and NGINX SERVER will be running on Ubuntu OS.



Virtual Machines vs. Containers

As you can see on the right, in the case of Docker, we have the underlying hardware infrastructure, then the operating system and Docker installed on the OS. Docker can then manage the containers that run with libraries and dependencies alone.

In the case of a virtual machine, we have the OS on the underlying hardware, then the hypervisor like ESX or virtualization of some kind, and then the virtual machines. As you can see, each virtual machine has its own operating system inside it. Then the dependencies, and then the application.



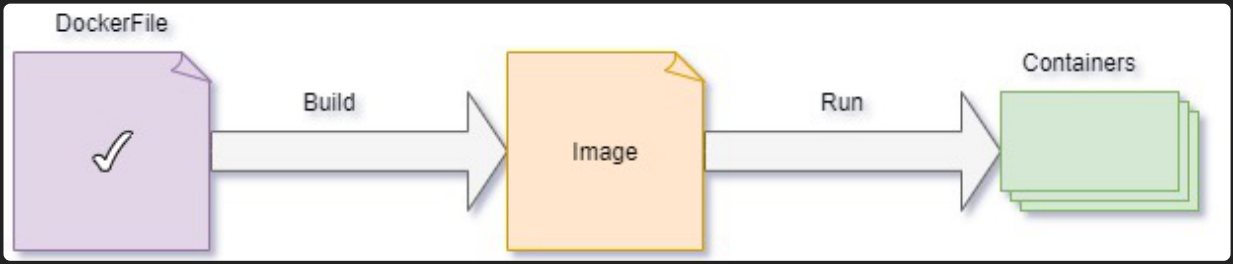
What is Docker Image?

Essentially, a Docker image is a static file that contains everything needed to run an application, including the application code, libraries, dependencies, and the runtime environment. It's like a snapshot of a container that, when executed, creates a Docker container.

A Docker image is composed of multiple layers stacked on top of each other. Each layer represents a specific modification to the file system (inside the container), such as adding a new file or modifying an existing one. Once a layer is created, it becomes immutable, meaning it can't be changed. The layers of a Docker image are stored in the Docker engine's cache, which ensures the efficient creation of Docker images.

Docker Image: A Docker image is a lightweight, standalone, and executable package that contains everything needed to run an application, including the code, runtime environment, libraries, and system tools.

It provides a consistent and reproducible platform for deploying applications regardless of the underlying infrastructure or host operating system. Docker images are created from a base image by defining a set of instructions called a Dockerfile that specify how to build the image step by step.



Difference between Docker Image VS Docker Container

Docker image	Docker container
The Docker image is the Docker container's source code.	The Docker container is the instance of the Docker image.
Dockerfile is a prerequisite to Docker Image.	Docker Image is a pre-requisite to Docker Container.
Docker images can be shared between users with the help of the Docker Registry.	Docker containers can't be shared between the users.
To make changes in the docker image we need to make changes in Dockerfile.	We can directly interact with the container and can make the changes required.

Structure Of Docker Image

The layers of software that make up a Docker image make it easier to configure the dependencies needed to execute the container.

- **Base Image: The basic** image will be the starting point for the majority of Docker files, and it can be made from scratch.
- **Parent Image: The parent i**mage is the image that our image is based on. We can refer to the parent image in the Dockerfile using the **FROM** command, and each declaration after that affects the parent image.
- **Layers: Docker** images have numerous layers. To create a sequence of intermediary images, each layer is created on top of the one before it.

How To Create A Docker Image And Run It As Container?

Step 1: **Create** a Dockerfile.

Step 2: Create a docker image of the application and download all the necessary dependencies needed for the application to run successfully.

```
docker build -t <name>:<tag>
```

Step 3: We have successfully created a Docker file and a respective Docker image for the same.

Step 4: Create a running container with all the needed dependencies and start the application.

```
docker run -p 9000:80 <image-name>:<tag>
```

Docker Commands

Pull an Docker Image From a Docker Hub Registry

```
$ docker pull nginx
```

List Docker Images

```
$ docker images

REPOSITORY    TAG           IMAGE ID       CREATED         SIZE
nginx         latest        0d9c6c5575f5   4 days ago      126MB
ubuntu        18.04         47b199b0cb85   2 weeks ago     64.2MB
```

Run the Container for above pulled image

```
$ docker run <image_name> e.g docker run ngnix
Note: if the above image is not pulled than the above Commands
will directly pull the image and will start the containers.
```

Check the running containers

```
$ docker ps

CONTAINER ID   IMAGE     CREATED          STATUS       NAMES       SIZE
e90b8831a4b8   nginx     11 weeks ago     Up 4 hours   my_nginx   35.58 kB
```

Stop and Start Containers

```
$ docker stop <CONTAINER ID>
  or
$ docker start <CONTAINER ID>
```

Check all container that is running or not running

```
$ docker ps -a
```

Remove an Image from Docker

```
$ docker rmi <image_id>
```

Note : **we need to bind the container port with host port and two container cannot be port to same host**

```
$ docker run -p<host_port>:<container_port>
or
$ docker run-p6000:6379
```

What is Docker Compose?

Docker Compose: Docker Compose is a powerful and efficient tool used to define and manage multi-container Docker applications.

It allows developers to create an application stack, describing the services, networks, and volumes needed for the containers to run seamlessly together. With Docker Compose, the orchestration of complex architectures becomes much simpler as it automates the deployment process.

Docker Compose will execute a YAML-based multi-container application. The YAML file consists of all configurations needed to deploy containers Docker Compose, which is integrated with Docker Swarm, and provides directions for building and deploying containers. With Docker Compose, each container is constructed to run on a single host.

How to Use Docker Compose? With example

In this project, we will create a straightforward Restfull API that will return a list of fruits. We will use a flask for this purpose. And a PHP application will request this service and show it in the browser. Both services will run in their own container.

Step-1 : First, Create a separate directory for our complete project. Use the following command.

```
mkdir dockerComposeProject
```

Step-2: Move inside the directory.

```
cd dockerComposeProject
```

Step-3 : Create API.

we will create a custom image that will use **Python** to serve our **Restful API** defined below. Then the service will be further configured using a **Dockerfile.**

Step-3.1 : Then create a subdirectory for the service we will name it product. and move into the same.

```
mkdir product
cd product
```

Step-3.2 : Build Python api.py

```
from flask import Flask
from flask_restful import Resource, Api

# create a flask object
app = Flask(__name__)
api = Api(app)

# creating a class for Fruits that will hold
# the accessors
class Fruits(Resource):
    def get(self):
      # returns a dictionary with fruits
        return {
            'fruits': ['Mango',
                       'Pomegranate',
                       'Orange',
                       'Litchi']
        }

# adds the resources at the root route
api.add_resource(Fruits, '/')

# if this file is being executed then run the service
if __name__ == '__main__':
```

Step-3.3 : Create a Docker file to define the container in which the above API will run.

```
FROM python:3-onbuild
COPY . /usr/src/app
CMD ["python", "api.py"]
```

Step-3.4 : Create Php HTML Website

```
cd ..
mkdir website
cd website
```

Step-3.5 : index.php

```
<!DOCTYPE html>
<html lang="en">
<head>
   <title>Fruit Service</title>
</head>
<body>
   <h1>Welcome to India's Fruit Shop</h1>
   <ul>
       <?php
           $json = file_get_contents('http://fruit-service');
           $obj = json_decode($json);
           $fruits = $obj->fruits;
           foreach ($fruits as $fruit){
               echo "<li> $fruit </li>";
           }
       ?>
   </ul>
</body>
</html>
```

Step-4 : Now create a compose file where we will define and configure the two services, API and the website And then create the file name as . **docker-compose.yaml**

Create Docker-compose.yaml file

```
version: "3"
services:
 fruit-service:
   build: ./product
   volumes:
     - ./product:/usr/src/app
   ports:
     - 5001:80

 website:
   image: php:apache
   volumes:
     - ./website:/var/www/html
   ports:
     - 5000:80
   depends_on:
     - fruit-service
```

Step-5 : Run the application stack

Now that we have our docker-compose.yml file, we can run it.

To start the application, enter the following command.

```
docker-compose up -d
```

Step-6 : To stop the application, either press CTRL + C or

```
docker-compose stop
```

Example Two: Docker commands:

index.html | docker-commands.md | package.json | server.js

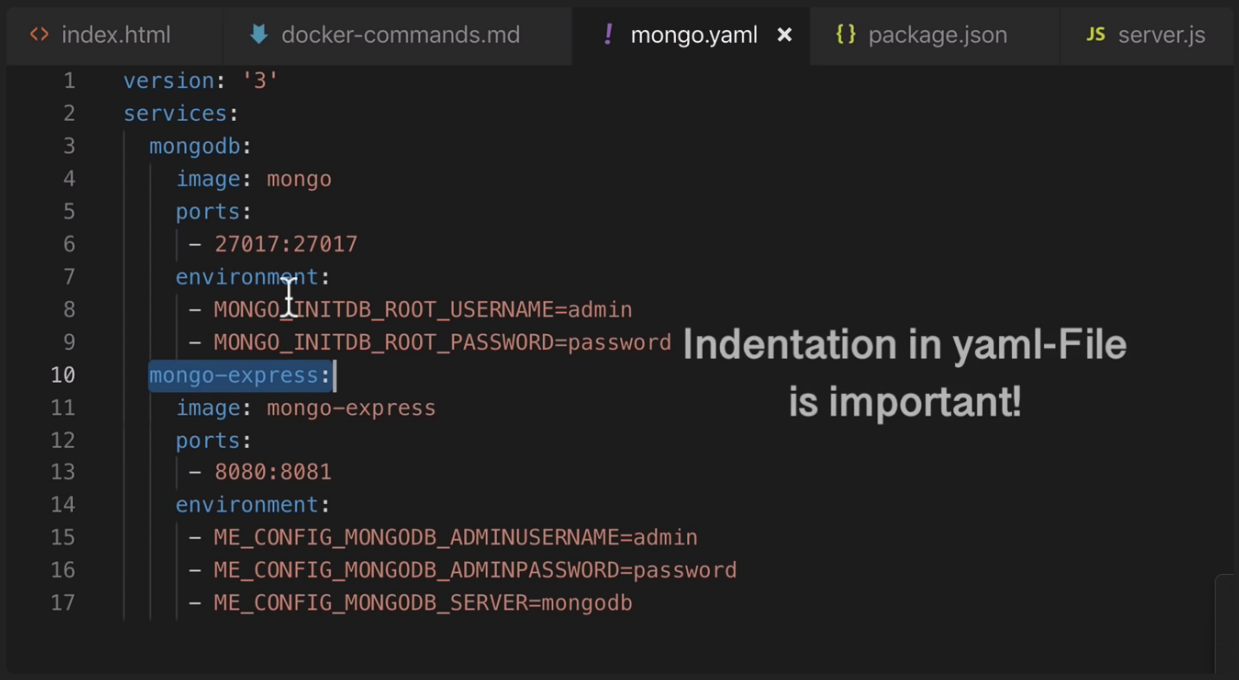
```
# commands

## create docker network
docker network create mongo-network

## start mongodb
docker run -d \
-p 27017:27017 \
-e MONGO_INITDB_ROOT_USERNAME=admin \
-e MONGO_INITDB_ROOT_PASSWORD=password \
--net mongo-network \
--name mongodb \
mongo

## start mongo-express
docker run -d \
-p 8081:8081 \
-e ME_CONFIG_MONGODB_ADMINUSERNAME=admin \
-e ME_CONFIG_MONGODB_ADMINPASSWORD=password \
-e ME_CONFIG_MONGODB_SERVER=mongodb \
--net mongo-network \
--name mongo-express \
mongo-express
```

mongo.yaml

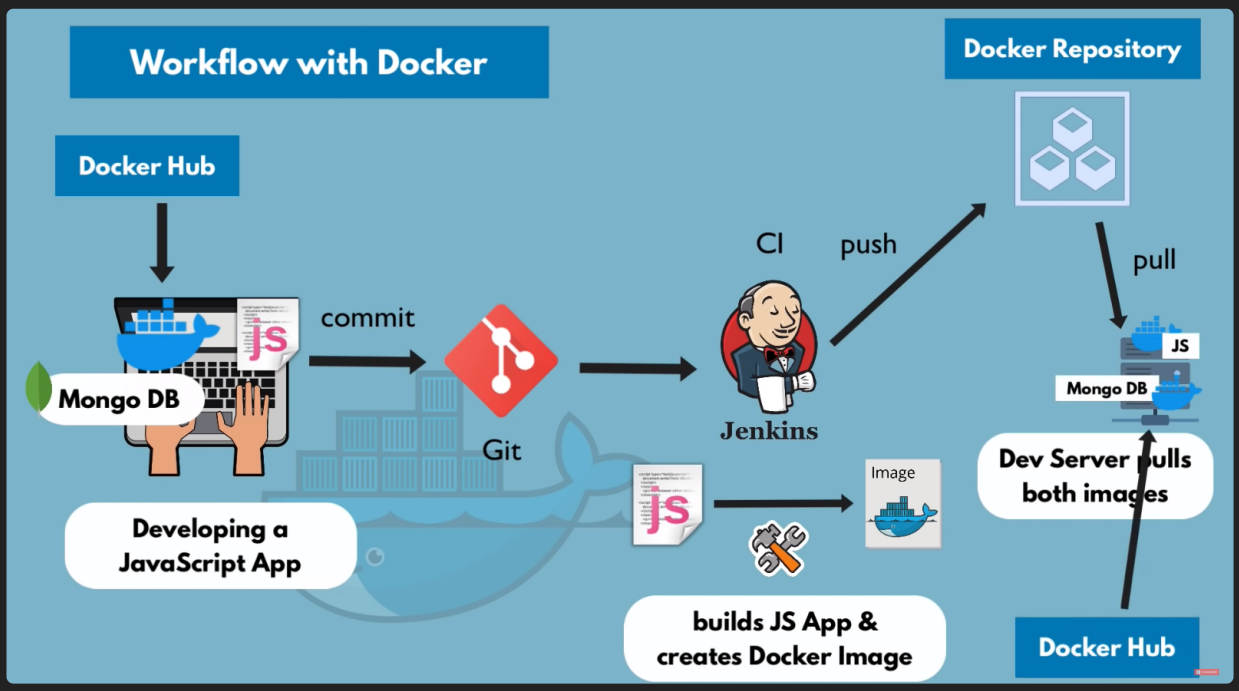



```
# run the above file using command
$ docker-compose -f mongo.yaml up/down
```

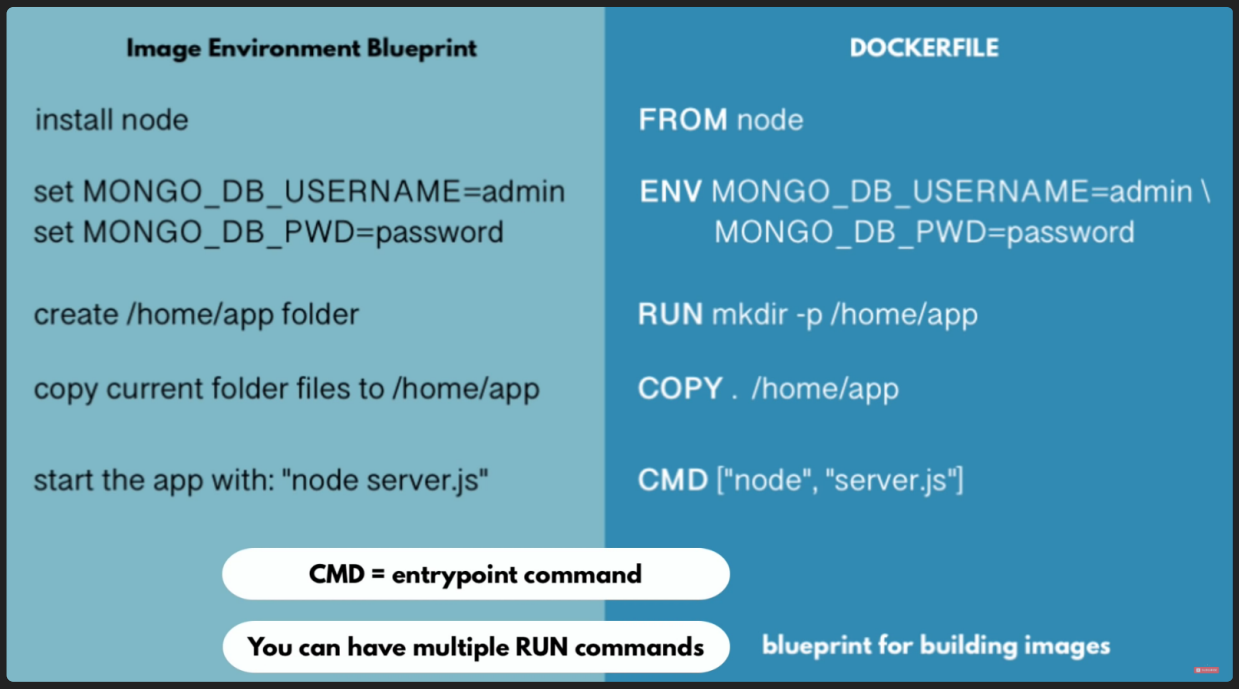
What is Dockerfile?

The Dockerfile uses DSL (Domain Specific Language) and contains instructions for generating a Docker image. Dockerfile will define the processes to quickly produce an image. While creating your application, you should create a Dockerfile in order since the Docker daemon runs all of the instructions from top to bottom.

Dockerfile: Docker can build images automatically by reading the instructions from a `Dockerfile`. A `Dockerfile` is a text document that contains all the commands a user could call on the command line to assemble an image.



Overview Of Creating Dockerfile



Commands

1. FROM - Represents the base image(OS), which is the command that is executed first before any other commands.

```
FROM <ImageName>
```

2. COPY- The copy command is used to copy the file/folders to the image while building the image.

```
COPY <Source> <Destination> e.g COPY ./app /home/app
```

3. ADD - While creating the image, we can download files from distant HTTP/ HTTPS destinations using the ADD command. Example: Try to download Jenkins using ADD command

```
ADD <URL> e.g ADD https://get.jenkins.io/war/2.397/jenkins.war
```

4. RUN - Scripts and commands are run with the RUN instruction. The execution of RUN commands or instructions will take place while you create an image on top of the prior layers (Image).

```
RUN < Command + ARGS> e.g RUN mkdir -p /home/app
```

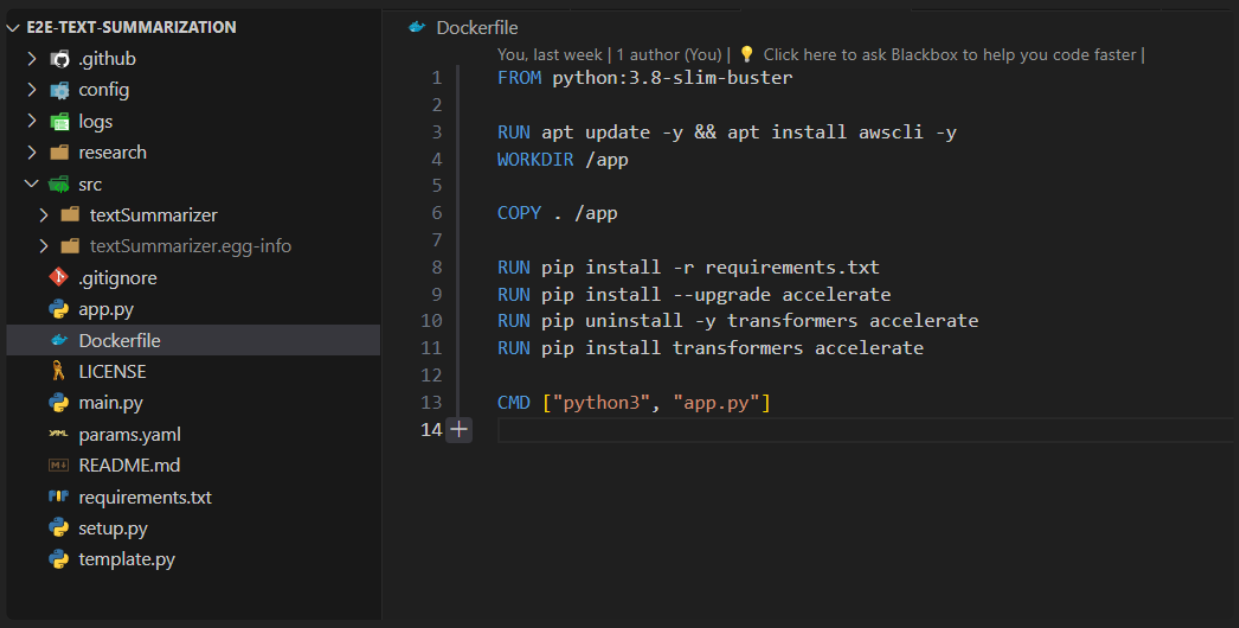
5. CMD - The main purpose of the CMD command is to start the process inside the container and it can be overridden.

```
CMD [command + args] e.g CMD["python", "app.py"]
```

E2E Example

In this Example we will Create the Dockerfile of the one of the project which is having a source code , then we will create the image from the Dockerfile and will run the container for that image . After that we will be exploring the container by running the **/bin/bash** or **/bin/sh** command to enter into container CLI. Finally we will push the image of the source code created from Dockerfile into Docker repository (here in our case in AWS ECR private repository).

Step-1 Write the Dockerfile for your source code



Step-2 Lets Build The Image With above Dockerfile

```
$ docker build -t <image_name>:<tag_name>
e.g docker build -t my-app:1.0
```

```
# lets check the the image
$ docker image
```

```
[my-app]$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
my-app              1.0                 2e0a4d16e074        2 days ago          116 MB
```

Step-3 Lets Run The Image in Container

```
$ docker run <REPOSITORY>:<TAG>
e.g docker run my-app:1.0
```

```
# lets check the container
$ docker ps
```

```
[~]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
51c6912d69f5        my-app:1.0          "docker-entrypoint..."   2 days ago          Up 34 seconds
```

Step-4 Lets Go Inside The Container

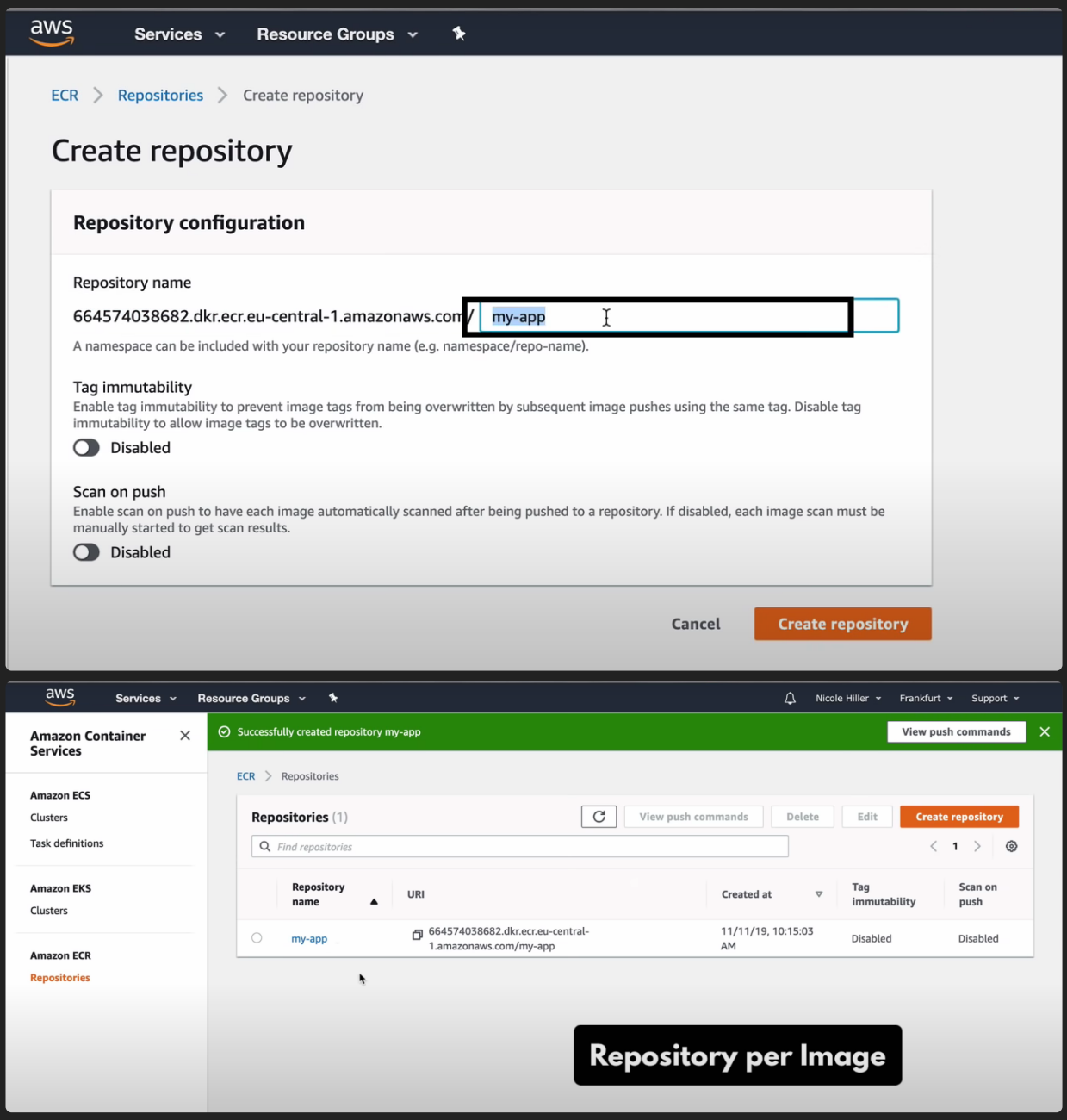
```
$ docker exec -it <CONTAINER ID> /bin/bash
or
$ docker exec -it <CONTAINER ID> /bin/sh
```

```
[~]$ docker exec -it 51c6912d69f5 /bin/sh
/ # ls
bin    dev    etc    home   lib    media  mnt    opt    proc   root   run    sbin   srv    sys    tmp    usr    var
/ # env
no_proxy=*.local, 169.254/16
NODE_VERSION=13.1.0
HOSTNAME=51c6912d69f5
YARN_VERSION=1.19.1
SHLVL=1
HOME=/root
MONGO_DB_USERNAME=admin
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
MONGO_DB_PWD=password
PWD=/
/ # ls /home/app/
Dockerfile         images             mongo.yaml         package-lock.json  resources
examples           index.html         node_modules       package.json       server.js
/ #
```

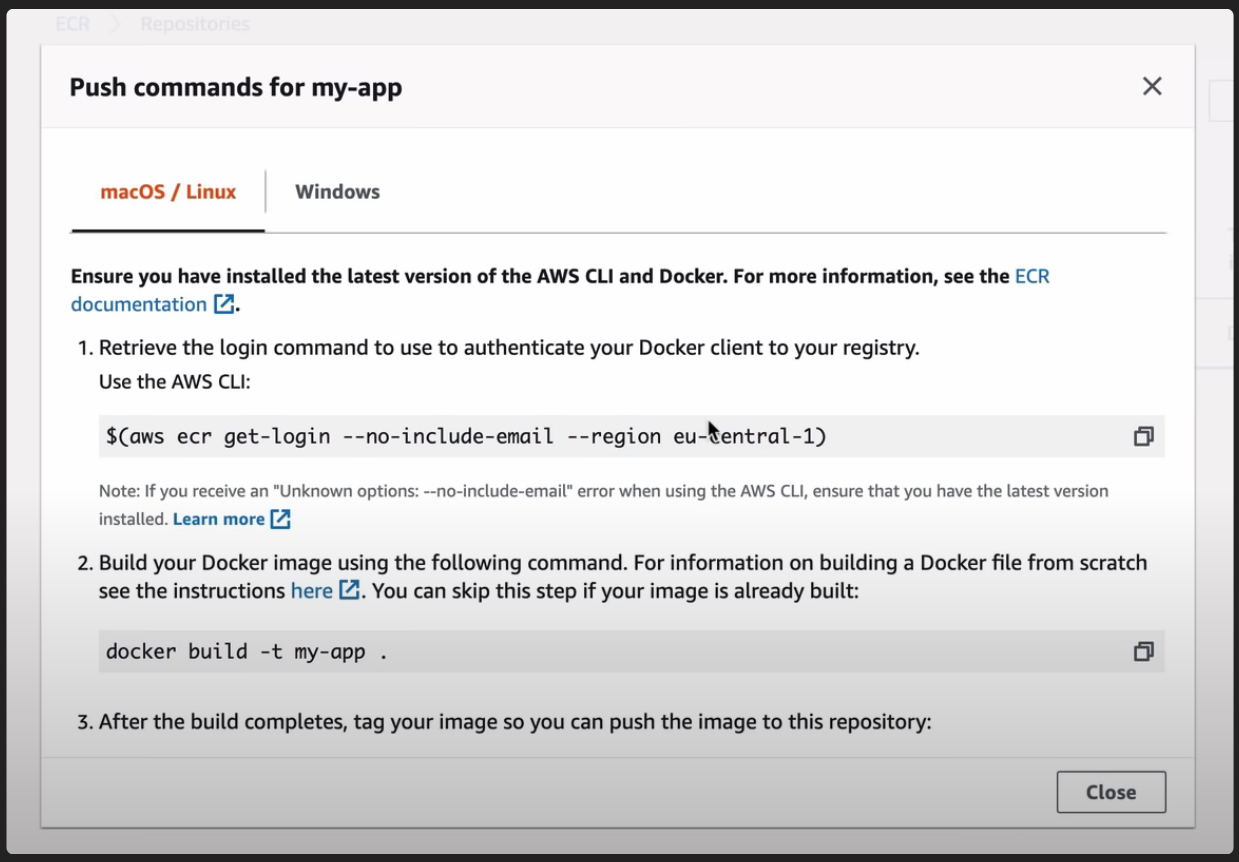
Step-5 Create The Private Repository And Push Container

Step- 5.1 - login into your aws account and search for ECR(elastic container service)

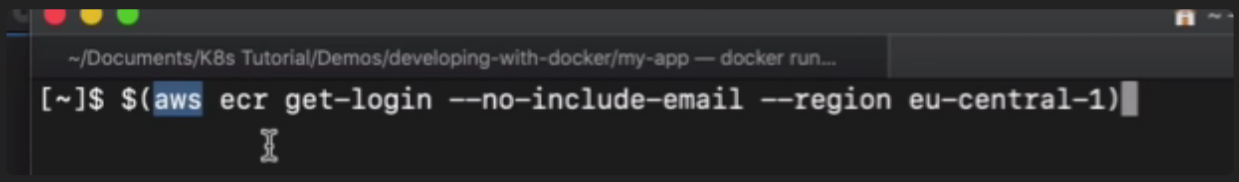
Step -5.2 Enter the name for your repo

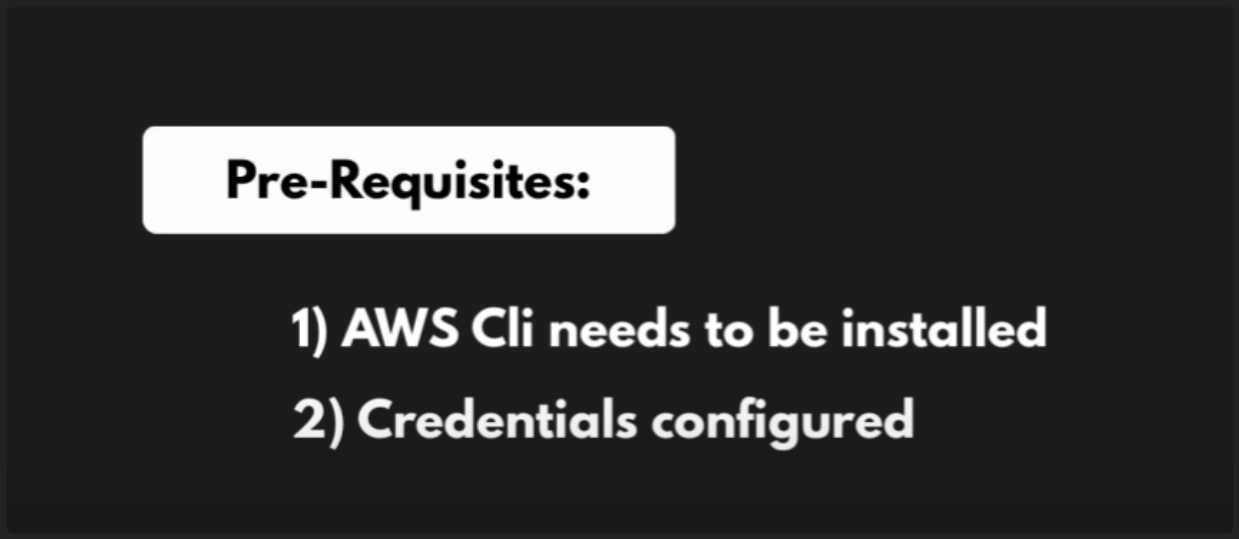


Step -5.3 Lets Push our local images there at ECR



Note : we have to download the AWS CLI for running above command shown below





Note: Naming convention for image Naming

Image Naming in Docker registries

registryDomain/imageName:tag

- In **DockerHub**:
  - docker pull mongo:4.2
  - docker pull docker.io/library/mongo:4.2
- In **AWS ECR**:
  - docker pull 520697001743.dkr.ecr.eu-central-1.amazonaws.com/my-app:1.0

Lets Push it AWS ECR

```
$ docker images
```

```
[~]$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
my-app              1.0                 83fdc778a892        2 days ago          116 MB
```

3. After the build completes, tag your image so you can push the image to this repository:

```
docker tag my-app:latest 664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app:latest
```

```
# give the name for your images
$ docker <USE_ABOVE_COMMAND> e.g shown below
```

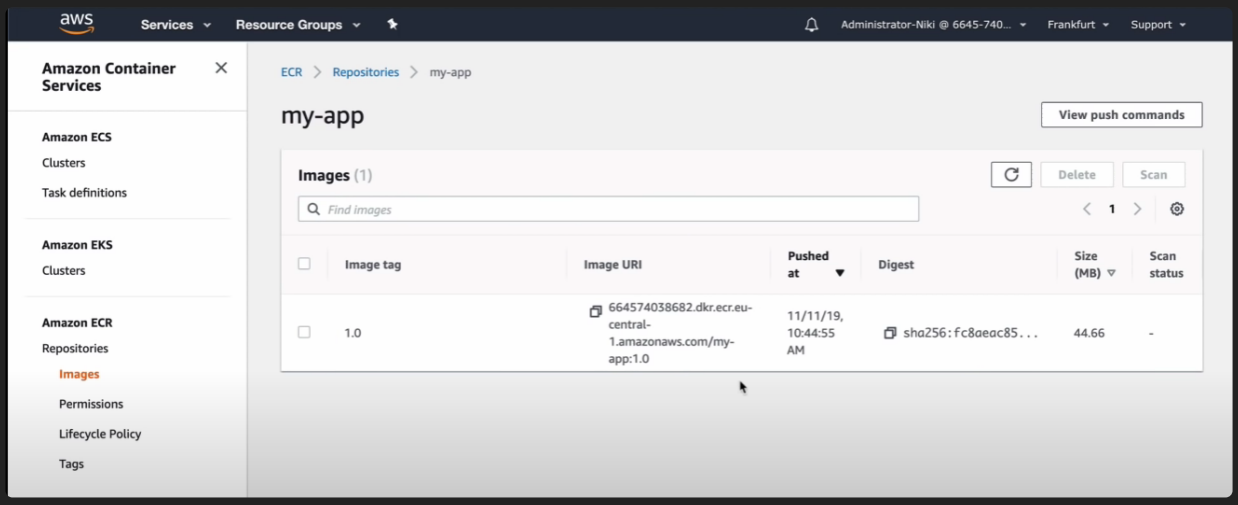
```
[~]$ docker tag my-app:1.0 664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app:1.0
[~]$ docker images
REPOSITORY                                                TAG                 IMAGE ID            CREATED             SIZE
664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app    1.0                 83fdc778a892        2 days ago          116 MB
my-app                                                    1.0                 83fdc778a892        2 days ago          116 MB
```

4. Run the following command to push this image to your newly created AWS repository:

```
docker push 664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app:latest
```

```
# push thwe images in AWS ECR
$ docker <USE_ABOVE_COMMAND> e.g shown below
```

```
[~]$ docker push 664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app:1.0
The push refers to a repository [664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app]
5678c8aa26b3: Pushed
bb2a17dfd6c2: Pushed
099773e542db: Pushed
9efd3ca0eab1: Pushed
a721b64d51de: Pushed
77cae8ab23bf: Pushed
1.0: digest: sha256:fc8aeac852dcb040b727cfb222078a27305c05bb480da50eca086ca7ce398bf9 size: 1576
[~]$
```



Note: What if you change your code? In that case we have to again build the images and then again rename it with next version and finally pushing it to AWS ECR. Below is the screenshot which will give you idea about it.

STEP - 1 Dockerfile Changed As

```
1  FROM node:13-alpine
2
3  ENV MONGO_DB_USERNAME=admin \
4      MONGO_DB_PWD=password
5
6  RUN mkdir -p /home/node-app
7
8  COPY ./app /home/node-app
9
10 CMD ["node", "/home/app/server.js"]
```

STEP - 2 Build Images As

```
[my-app]$ docker build -t my-app:1.1 .
Sending build context to Docker daemon 11.02 MB
Step 1/5 : FROM node:13-alpine
 ---> f20a6d8b6721
Step 2/5 : ENV MONGO_DB_USERNAME admin MONGO_DB_PWD password
 ---> Using cache
 ---> 8c596c87f088
Step 3/5 : RUN mkdir -p /home/node-app
 ---> Running in e66fa502b8f2
 ---> e047bc7d2a2a
Removing intermediate container e66fa502b8f2
Step 4/5 : COPY ./app /home/node-app
 ---> 170aec5715af
Removing intermediate container 9800b8efa0c1
Step 5/5 : CMD node /home/app/server.js
 ---> Running in ce7c77fe2fc8
 ---> 2f48dde6528c
Removing intermediate container ce7c77fe2fc8
Successfully built 2f48dde6528c
```

STEP - 3 Rename it Acc to Naming Conventions of AWS ECR

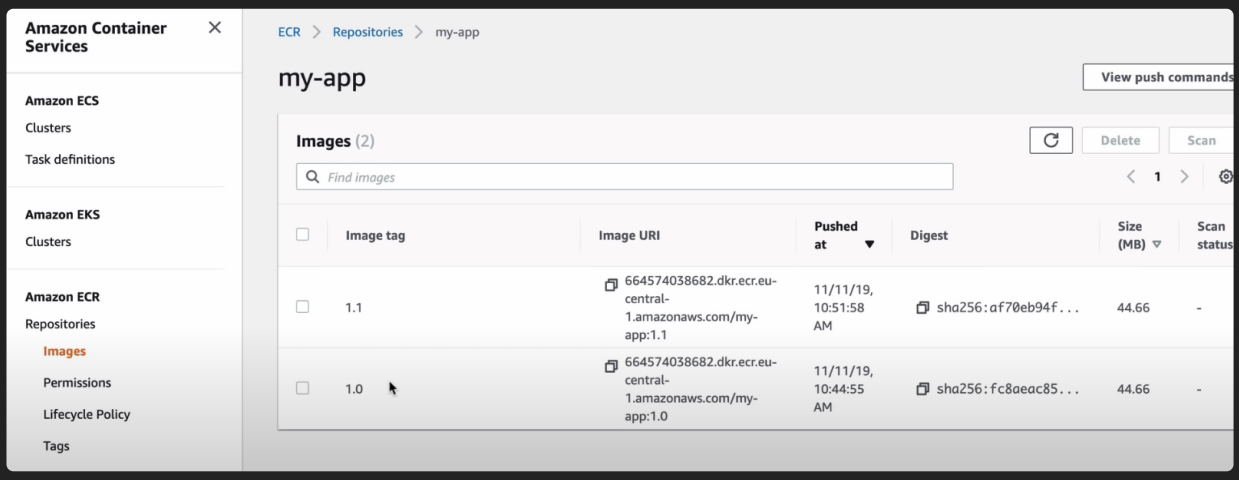
```
[my-app]$ docker images
REPOSITORY                                                  TAG                 IMAGE ID            CREATED             SIZE
my-app                                                      1.1                 2f48dde6528c        2 days ago          116 MB
664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app      1.0                 83fdc778a892        2 days ago          116 MB
```

```
[my-app]$ docker tag my-app:1.1 664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app:1.1
```

```
[my-app]$ docker images
REPOSITORY                                                  TAG                 IMAGE ID            CREATED             SIZE
664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app      1.1                 2f48dde6528c        2 days ago          116 MB
my-app                                                      1.1                 2f48dde6528c        2 days ago          116 MB
664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app      1.0                 83fdc778a892        2 days ago          116 MB
```

STEP - 4 Push it to AWS ECR

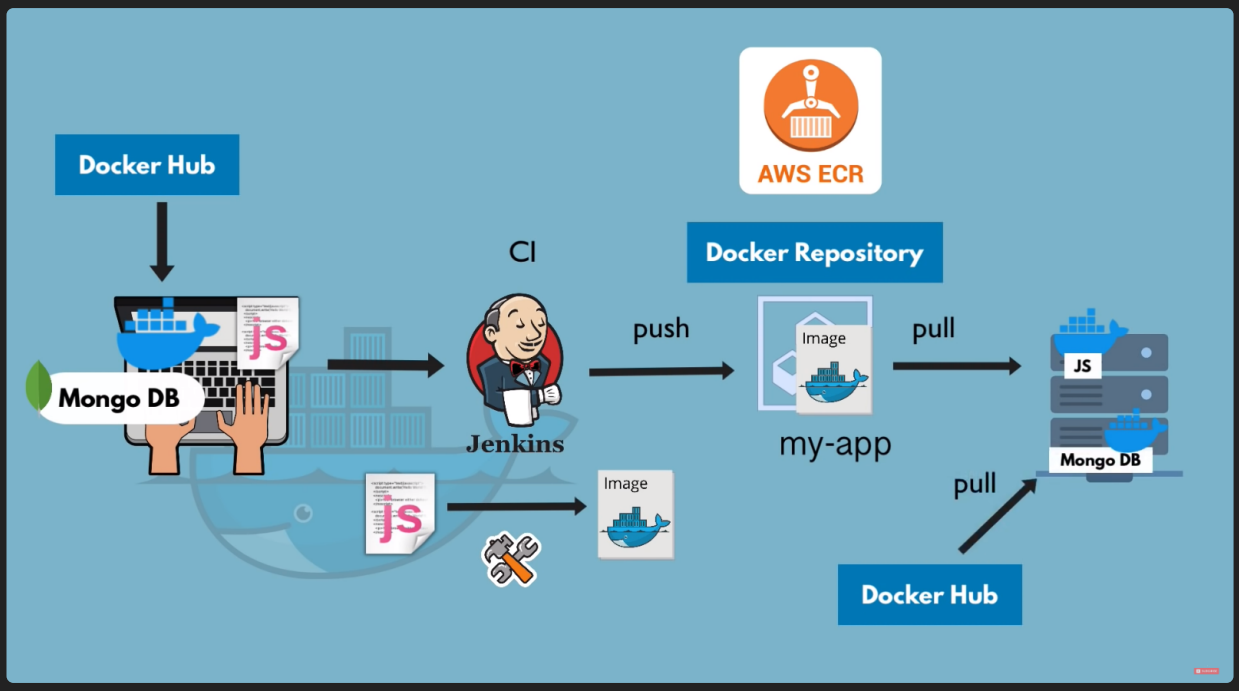
```
[my-app]$ docker push 664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app:1.1
The push refers to a repository [664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app]
eaf1692e831f: Pushed
ca0c58954a65: Pushed
099773e542db: Layer already exists
9efd3ca0eab1: Layer already exists
a721b64d51de: Layer already exists
77cae8ab23bf: Layer already exists
1.1: digest: sha256:af70eb94f7996531a8870b93fd53bb16899e8259591fe7c950971facf516e39d size: 1576
```



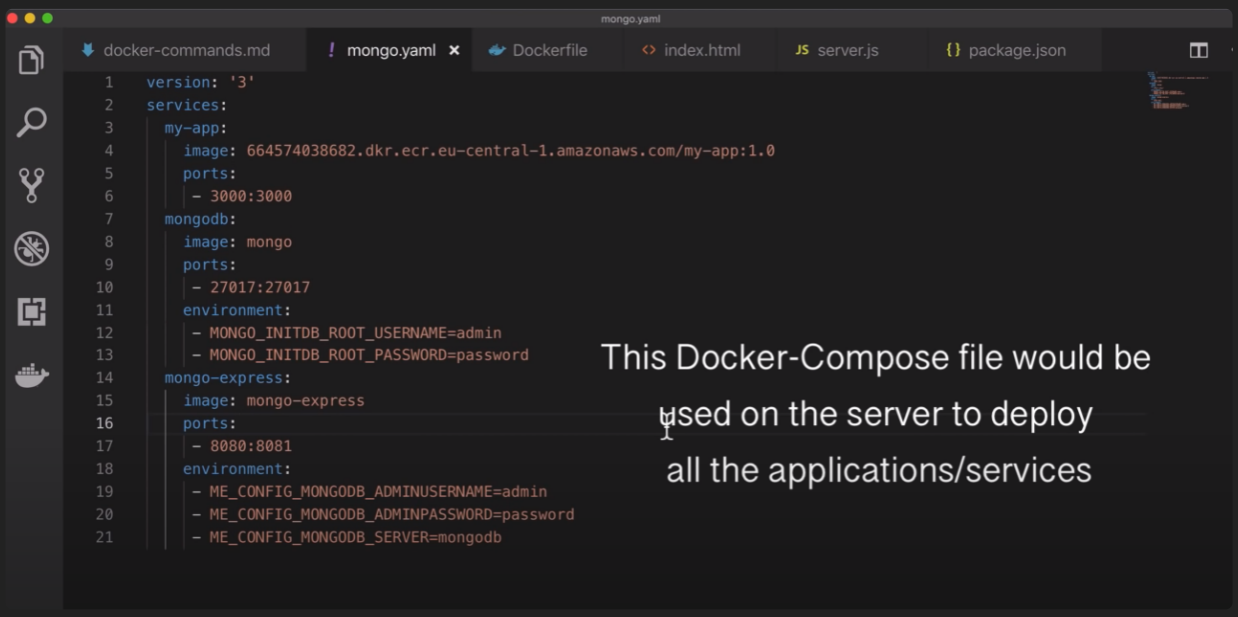
Deployment of Containerized App

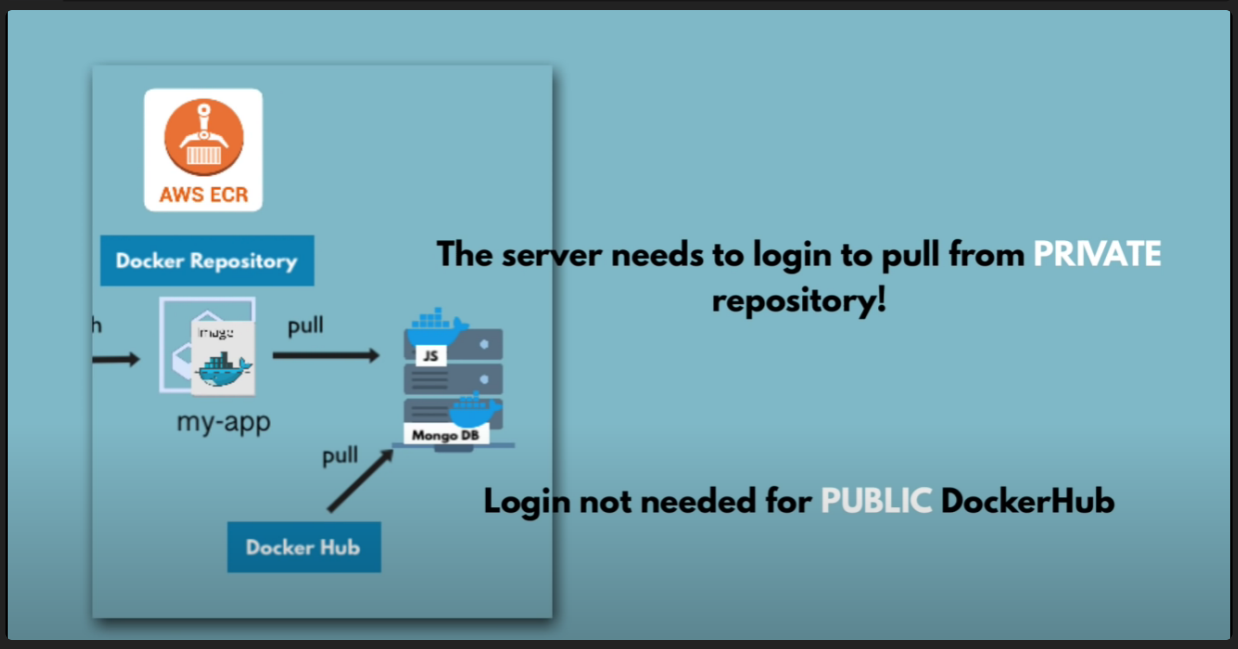
Here now we have our images in AWS CER repo , now we want our app to deployed on server for the end user to use it . For this we will pull the images from different container repo and will run it on Virtual machine like in case of AWS its EC2 machine. Below is the flow diagram for better Understanding.

Note : If you are using images from different container repo that we have to write `docker-compose.yam` l file which will be running on EC2 machine.

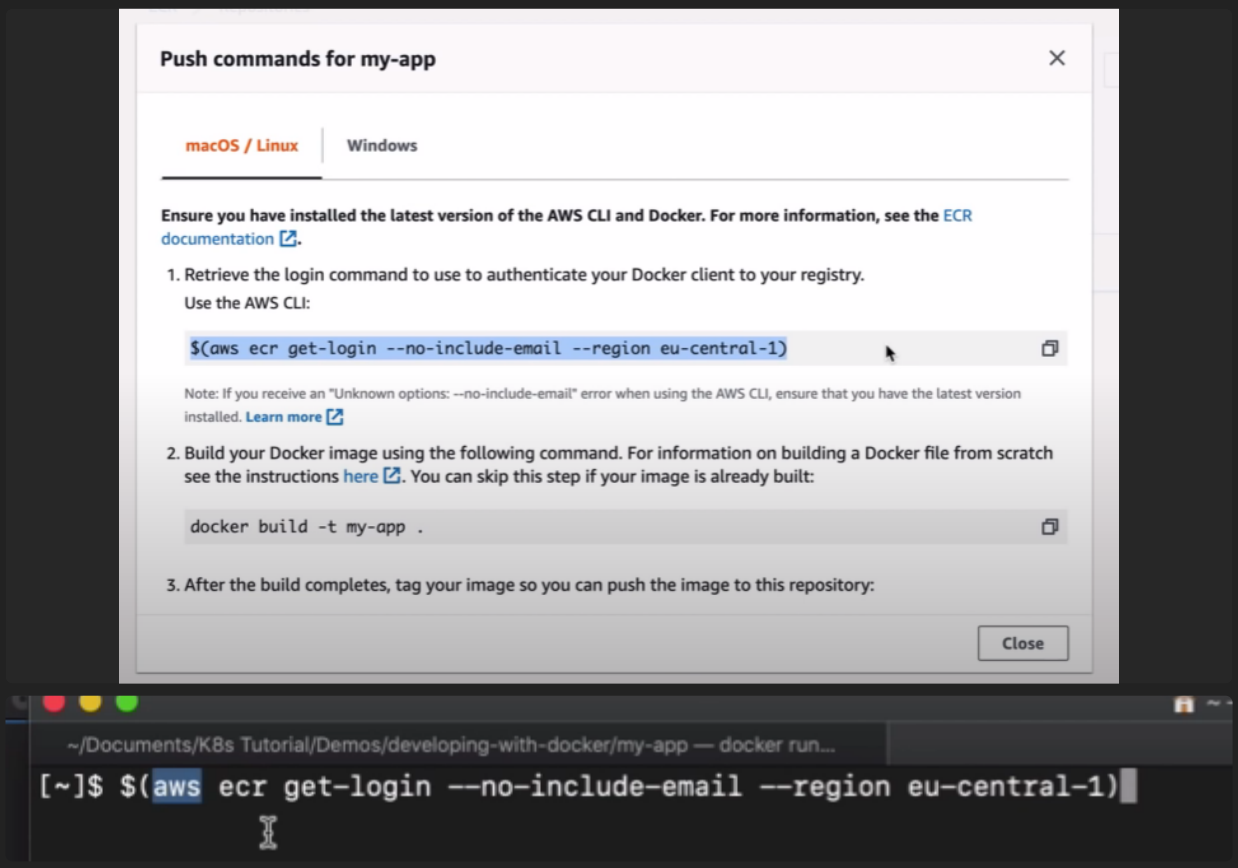


Step - 1 Write the docker-compose.yaml File as

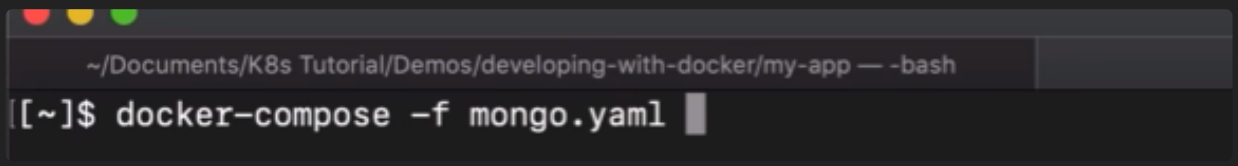




Step - 2 Login To AWS ECR in AWS CLI



Step - 3 Finally Run The .yaml File In AWS EC2 Machine

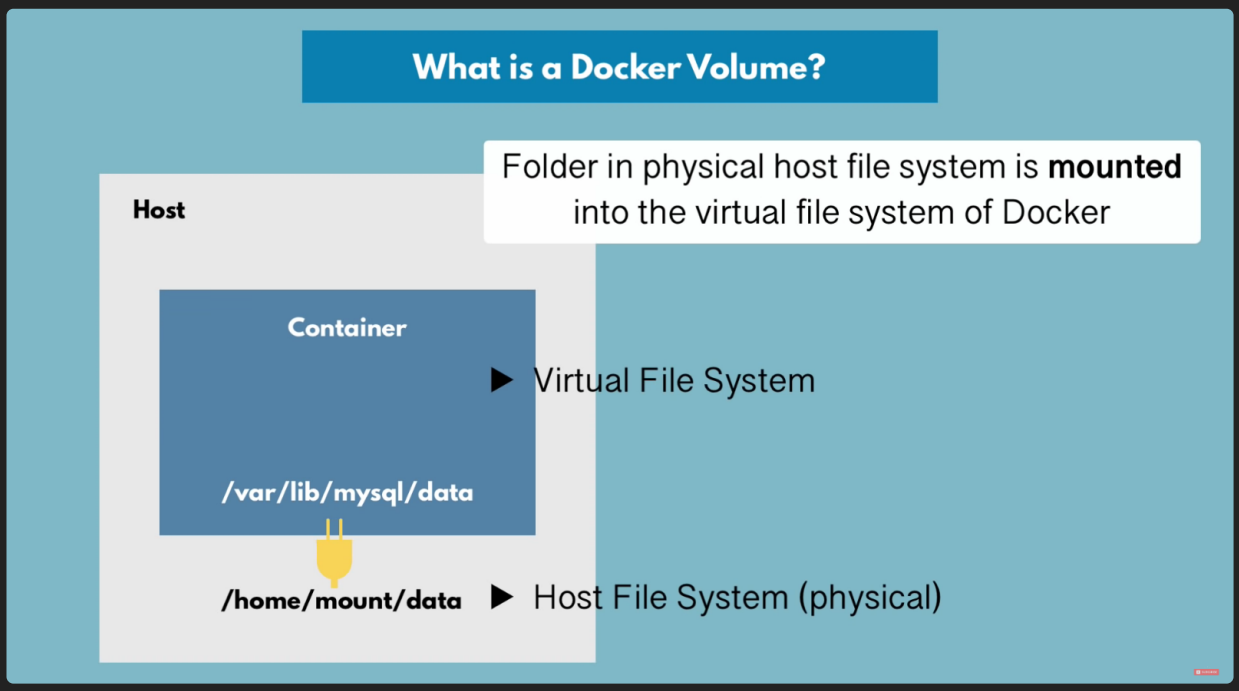


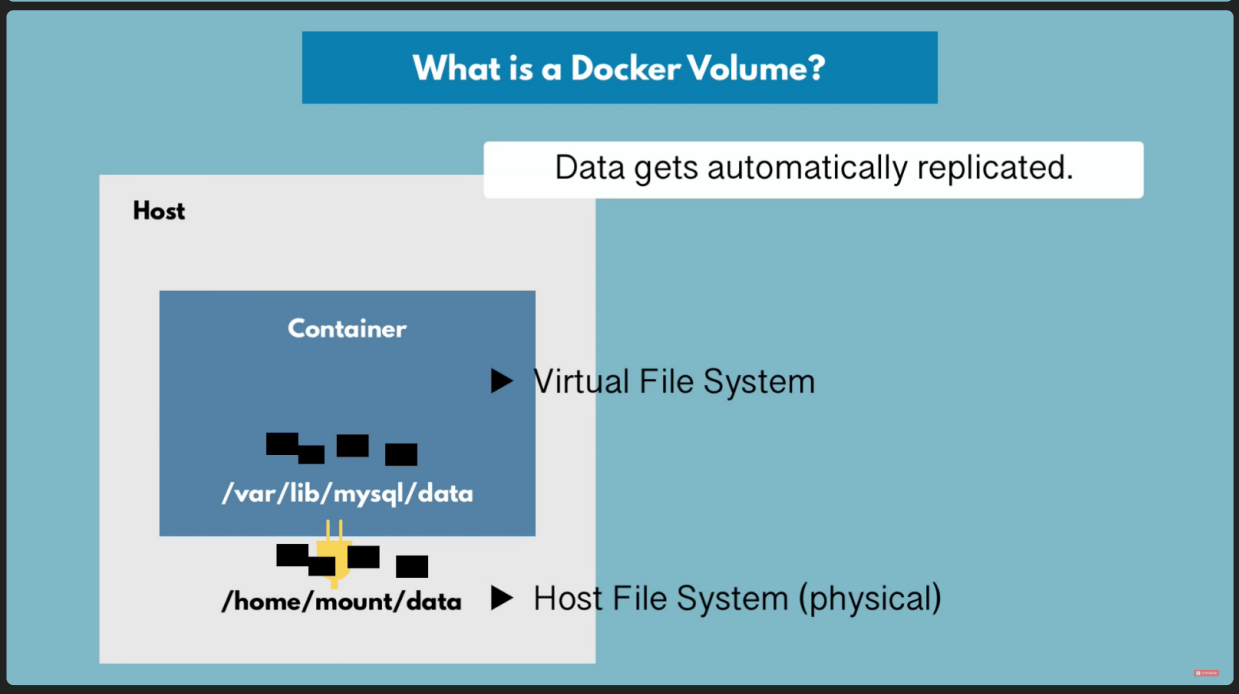
Docker Volumes

Docker Volumes are a popular and effective method for assuring data permanence while working in containers. Docker volumes are file systems that are mounted on Docker containers to preserve the data generated by the container.

Docker containers enable apps to execute in an isolated environment. All modifications made inside the container are lost by default when it ends. Docker volumes and bind mounts can be useful for storing data in between runs. One way to store data outside of containers is with volumes. All volumes are kept in a specific directory on your host and are controlled by Docker.

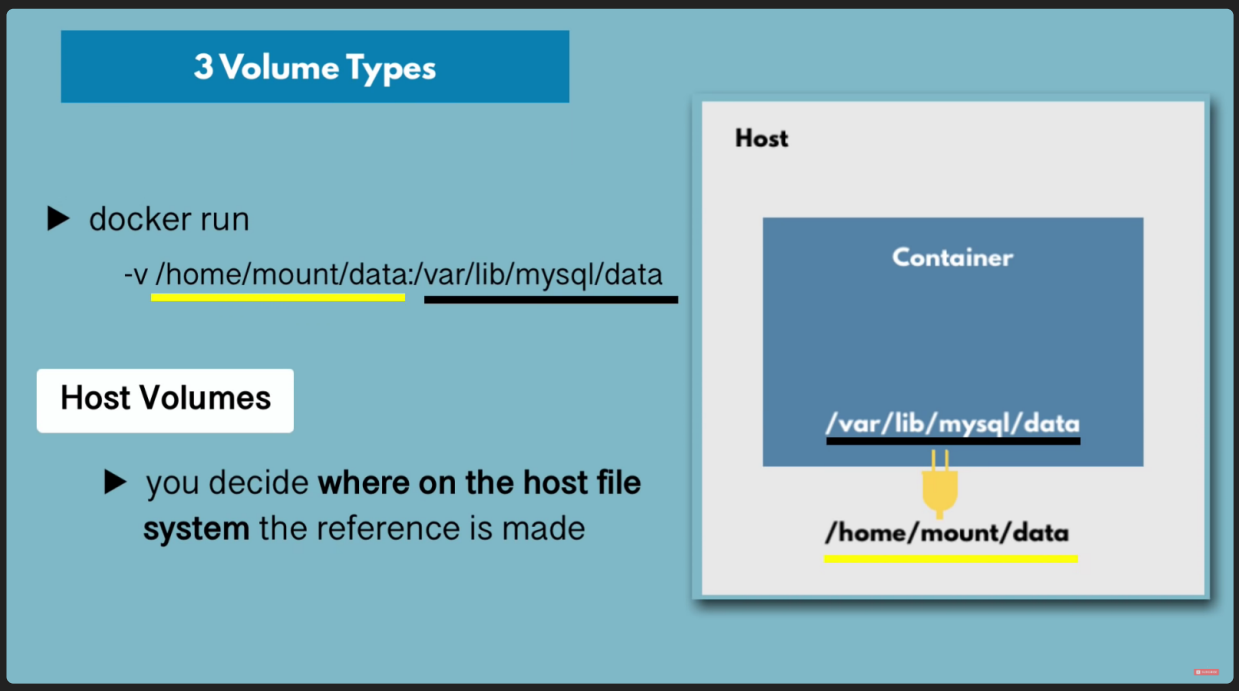
Refer the Below for better understanding



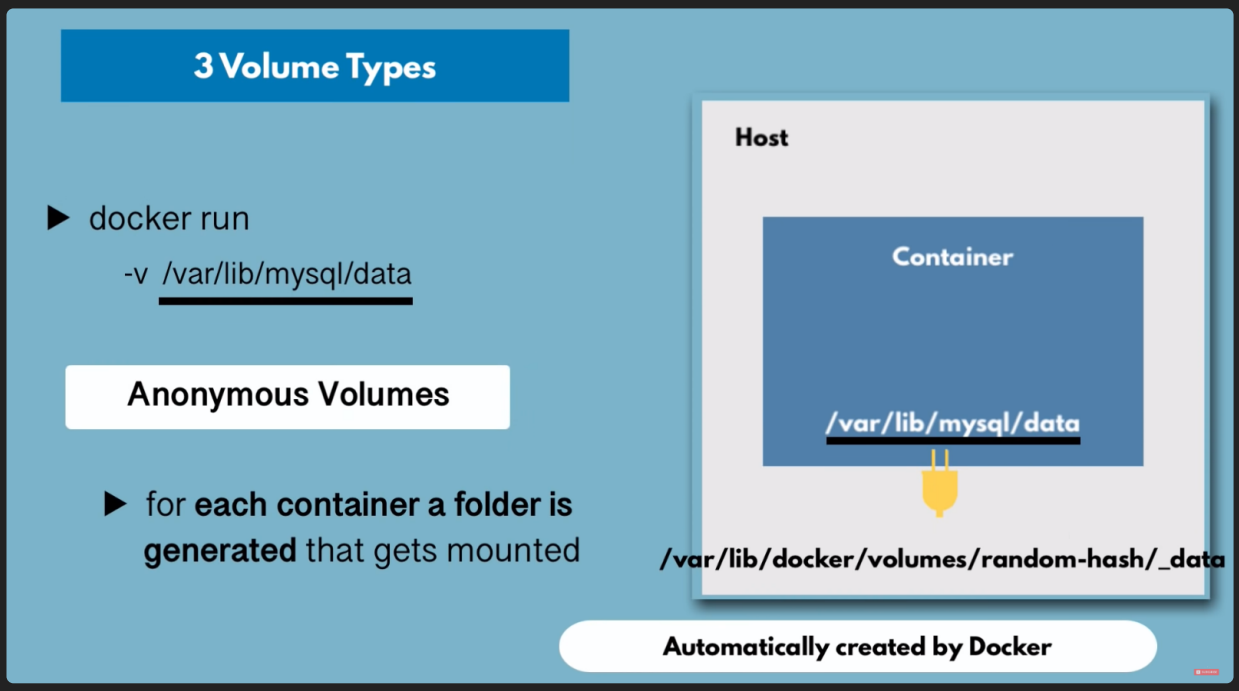


Different Types of Volumes

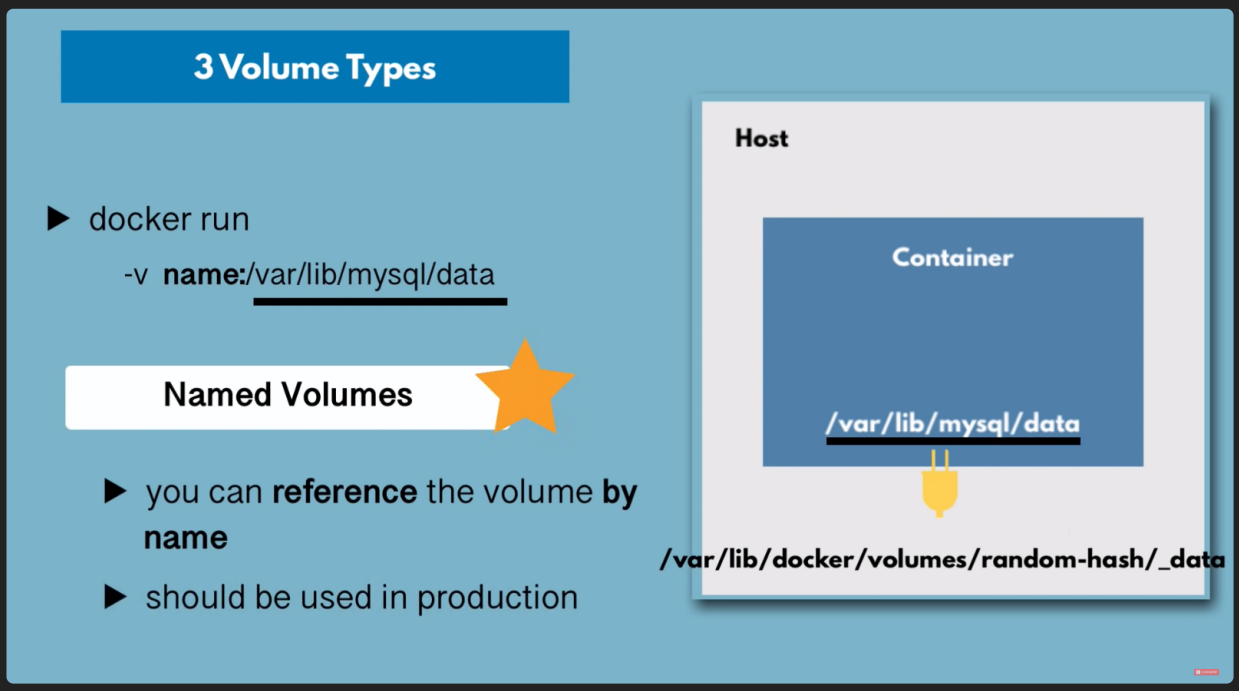
1 - Host Volumes



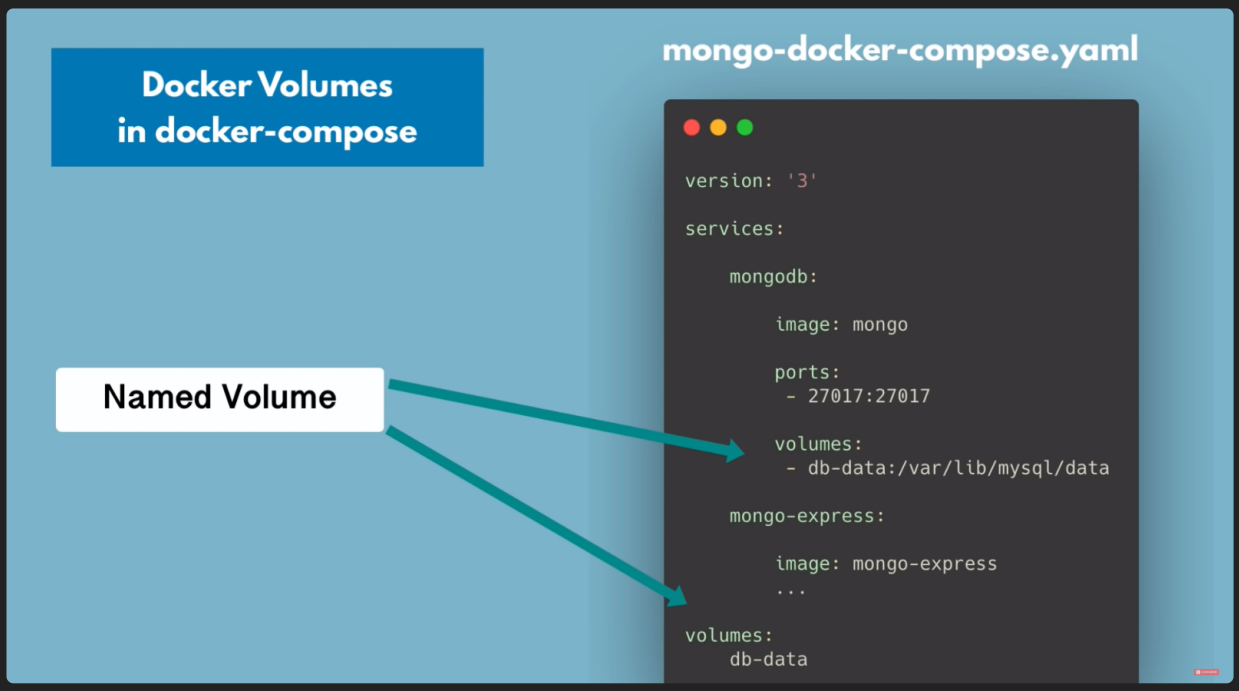
2 - Anonymous Volumes



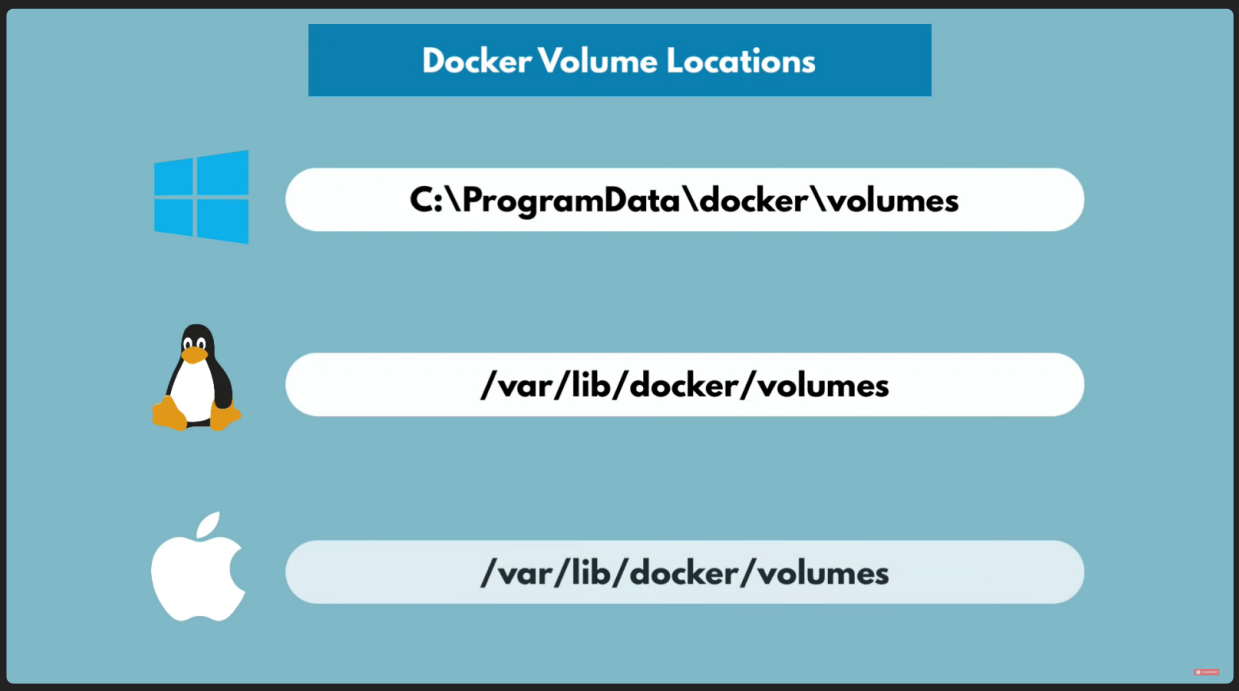
3 - Named Volumes (it should be used in Production among all three)



How we Define volumes in docker-compose.yaml File



Docker volumes Location on Local host



THANK YOU

SATYA SAURABH MISHRA