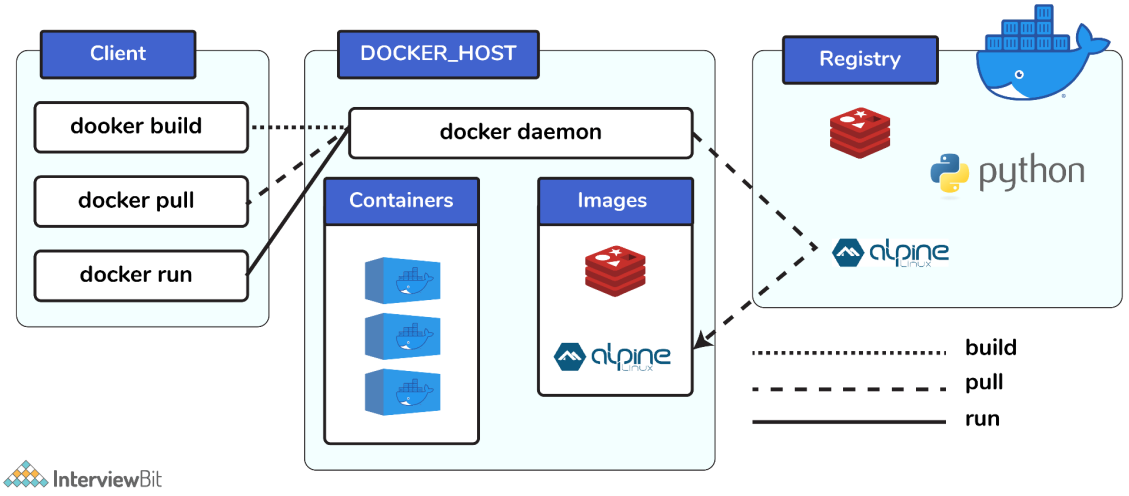
# DOCKER

- **Reference video:** Youtube
- **Why docker ?**
    - To make easy deployment of the applications once they are built up.
    - We need to take care of many things like installing packages, dependencies, environment setup on the machine where we have to deploy our application.
    - By using docker we can do these in a faster and more efficient way by making use of docker containers.
    - Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.
    - It also helps when we want to run our application in the same language but in different versions, at this stage we have to install dependencies based on the requirement of both the versions which may be not possible or may arise some conflicts, but it can be solved using dockers.
    - Docker will give an isolation in the form of containers to the application and their infrastructure (dependencies) needed to run it independently.
    - It also helps in running databases very easily just using one single command.
- **What is docker?**
    - Docker is a containerization platform which helps us in developing, shipping, and running applications in a faster way.
    - Docker is a containerization platform that combines all the dependencies of our applications in a package so that we can ship this package in any environment and run our application seamlessly.
    - Docker uses the OS kernel of the host machine where the docker container is running eg: our server.
    - Docker virtualizes the applications not the OS.
    - If the docker image is not compatible with our local machine OS versions then use docker toolbox to settle up the conflicts of the OS.
- **What do we mean by containerization ?**
    - Containerization is the packaging together of software code with all its necessary components like libraries, frameworks, and other dependencies so that they are isolated in their own "container".
- **Components of Docker Architecture ?**
    - *Docker Client:* This component executes build and run operations to communicate with the Docker Host.
    - *Docker Host:* This component holds the Docker Daemon, Docker images, and Docker containers. The daemon sets up a connection to the Docker Registry.
    - *Docker registry:* It is the place where we store the pre-built docker images which can be directly downloaded as and when needed. EG:

# DOCKER

docker Hub, Azure Container Registry.




- **What is Dockerfile ?**
  - Dockerfile is a special file which contains the set of instructions, dependencies need to be installed, port to be exposed, environment variables setup, and other configurations.
  - It also contains the command to be executed to run the application in “CMD” or “ENTRYPOINT” section.
  - It defines the steps and configurations needed to create a reproducible and standardized environment for running a specific application or service. Example of Dockerfile
  - FROM python:3.10-alpine
  - WORKDIR /deployer
  - COPY . /deployer
  - RUN pip install -r /deployer/requirements.txt
  - ENTRYPOINT ["python3", "/deployer/main.py"]
- **Difference between CMD and ENTRYPOINT**
  - **ENTRYPOINT**
    - The `ENTRYPOINT` instruction allows you to configure a container that will run as an executable.
    - It sets the main command and parameters that will be executed when the container starts.
    - ENTRYPOINT ["echo", "Hello"]
    - When you run a container using this image (`docker run <image>`), it will output `Hello`.
  - **CMD**
    - The `CMD` instruction provides default arguments for the executable defined by `ENTRYPOINT`. It can be overridden by specifying command-line arguments when running the container.

        - If a Docker image has both `CMD` and `ENTRYPOINT`, the `CMD` arguments are appended to the `ENTRYPOINT` command when the container is run.
    - **Combining `ENTRYPOINT` and `CMD`**
        - You can use both ENTRYPOINT and CMD in a Dockerfile. The CMD instruction provides default arguments for the ENTRYPOINT command.
        - ENTRYPOINT ["echo"]
        - CMD ["Hello", "world"]
        - When you run a container using this image (`docker run <image>`), it will output `Hello world`.
    - In summary, `ENTRYPOINT` is used to set the main command and parameters, and `CMD` is used to provide default arguments for that command.
- **What is Docker image ?**
    - A Docker image is a lightweight, standalone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and system tools.
    - Image is the actual package that contains all the dependencies, environment, configurations, application code needed to run the image.
- **What is Docker Container ?**
    - Docker container is a running form of images.
    - The upper most image will be our own application running in the container.
    - Container gives the environment to the image to actually run.
- **Difference between normal and alpine images ?**
    - Primary difference is the size of the docker images.
    - In alpine images packer manager used is basic like **apk,** shell used in **ash** instead of **bash** etc.
    - So basically alpine images are less feature giving and flexible for developers in terms of shell n all but very good in terms of fast deployment, security and less image sizes.
    - Ideal for scenarios where a minimal and lightweight image is crucial, such as in microservices, container orchestration, and environments with resource constraints.
- **Can a container restart itself ?**
    - Yes, it is possible only while using certain docker-defined policies while using the docker run command. Following are the available policies:
    - **no:** The default behaviour is to not start containers automatically.
    - **on-failure:** Restart the container if it exited with a non-zero exit code or if the docker daemon restarts.

    - **unless-stopped:** Restart the container unless the container was in stopped state before the Docker daemon was stopped (explained later).
    - **always:** Always restart a stopped container unless the container was stopped explicitly.
    - These policies can be used as:
    - `docker run -dit – restart [restart-policy-value] [container_name]`
- **Life cycle of Docker Container ?**
    - Docker containers go through the following stages:
        - Create a container
        - Run the container
        - Pause the container (optional)
        - Un-pause the container (optional)
        - Start the container
        - Stop the container
        - Restart the container
        - Kill the container
        - Destroy the container
- **How does docker networking work ?**
    - Docker provides several networking options, and the default is the bridge network.
    - **Bridge Network**: When you install Docker, it creates a default bridge network named `bridge`. Containers attached to this network can communicate with each other using container names as hostnames. Docker assigns IP addresses from the `172.17.0.0/16` address range by default, but you can customize this.
    - **Host Network**: You can also use the host network mode, where the container shares the network namespace with the host. In this mode, the container uses the host's network interfaces directly, so it doesn't get its own isolated network stack.
- How is the IP address assigned to the docker container ?
    - **Automatically Assigned IP**: When you create a container, Docker automatically assigns an IP address from the subnet associated with the network to the container.
    - **Container Naming**: Containers can communicate with each other using their names as hostnames. Docker automatically sets up the DNS resolution for containers within the same network.
    - **User-Defined Networks**: If you create a custom bridge network, you can specify the IP address range and manually assign IP addresses to containers.
- How does port mapping works in docker container ?
    - **Publishing Ports with `-p` option:**

        - docker run -p `<host-port>:<container-port>` my-web-app
        - docker run -p 8080:80 my-web-app
        - In this example, the container's port 80 is mapped to the host's port 8080.
    - **Exposing Ports with `-e` option:**
        - You can use the `-e` option to expose a port without publishing it to the host. This allows containers to communicate with each other on the same Docker network.
        - docker run --expose=8080 my-web-app
        - This doesn't make the port accessible from outside the Docker network, but it allows other containers on the same network to communicate with the container on port 8080.
    - **Dynamic Port Allocation with `-P` option:**
        - If you don't specify a particular port, you can use the `-P` option to dynamically allocate a port on the host machine.
        - docker run -P my-web-app
        - Docker will select a random available port on the host and map it to the container's exposed ports.

| Flag value | Description |
|---|---|
| `-p 8080:80` | Map port `8080` on the Docker host to TCP port `80` in the container. |
| `-p 192.168.1.100:8080:80` | Map port `8080` on the Docker host IP `192.168.1.100` to TCP port `80` in the container. |
| `-p 8080:80/udp` | Map port `8080` on the Docker host to UDP port `80` in the container. |
| `-p 8080:80/tcp -p 8080:80/udp` | Map TCP port `8080` on the Docker host to TCP port `80` in the container, and map UDP port `8080` on the Docker host to UDP port `80` in the container. |
| `docker run -p 127.0.0.1:8080:80 nginx` | If you include the localhost IP address (`127.0.0.1`) with the publish flag, only the Docker host can access the published container port. Outside world cannot access it. |

# DOCKER

- **What is docker daemon ?**
  - Docker daemon is a background process that manages Docker containers on a system.
  - **Management of Containers:**
    - The Docker daemon is responsible for creating, running, stopping, and deleting containers.
    - It manages the entire container lifecycle, ensuring that containers are started and stopped as needed.
  - **Communication with Docker CLI:**
    - The Docker daemon listens for Docker commands from the Docker CLI (Command Line Interface) or other Docker clients.
    - When you run a Docker command, such as `docker run` or `docker build`, the Docker CLI communicates with the Docker daemon to execute the requested operation.
- **Difference between VM and container?**
  - VM virtualizes both OS and application, but container virtualizes the app running on it and container will use the OS enviorment of the host_machine itself(host machine can also be one VM running on our laptop or running on the AWS cloud i.e EC2 instance).
- **Advantage of the containers:**
  - Let's say our application uses 2 diff types of mySQL versions. Then I can make one container running MYSQL V1 and another container running MYSQL V2.
- **Difference between container and image**
  - Container is the running environment for image
  - Means container will give the environment, file system needed to store some data, etc to the image(our application or predefined some image downloaded from the docker hub).
  - Container will bind one port to the image running on the container.
- **What do we mean by port mapping of containers?**
  - On 1 host machine/ VM more than 1 docker container can run parallely.
  - 2 containers can run on the same port in the same host machine.
  - Now question arises that how will the API call bifurcated between the 2 containers(basically 2 images running on different containers).
  - Now let's say there are 2 containers running on the same port 3000.
  - Here the concept of binding the host port to the container port comes in.
  - Whenever a container starts running on a VM/host machine, the machine will assign one port to that container(eg: 5000). Now, this 5000 port will be mapped with one of the ports of the container(eg:3000).
  - Now for the other container having the same port(eg: 3000) will be binded to another port number assigned by the machine(eg: 6000).

    - So, whenever the request comes it will come on the port of the host machine/ VM not directly to the container.
    - According to the mapping done between the machine port and container port(basically image is being binded to the host port) the request will be forwarded to that specific container(image-our sample app) by the host machine/ VM.
    - It means the API calls will be made on the port number of the host machine/ VM and not the actual container.
- **What is stored in the docker container?**
    - Docker uses storage drivers to store image layers, and to store data in the writable layer of a container.
    - The container's writable layer does not persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime.
- **General Points**
    - Whenever we are deploying the app the database used by that app will run on different containers.
    - It simply means that let's say we have one java-script app and that app uses mongodb as the data-base for the storage.
    - We will use the image from the docker hub of the mongo db directly. For our app code one separate custom docker image will be formed which will be stored into some private docker repository of the organisation.
    - Now when we want to run both the images, server/host_machine will pull both the images mongo-db from the public docker hub and app-code image from the private docker repository and each image will be deployed on individual separate containers running parallely.
    - There will be some intermediator called jenkins which will make the docker image of our app-code based on the given artifacts of the app-code. (artifacts like the environment info, version info of the language used by the application, etc)

## DOCKER COMMANDS TABLE

| When to use | Command |
|---|---|

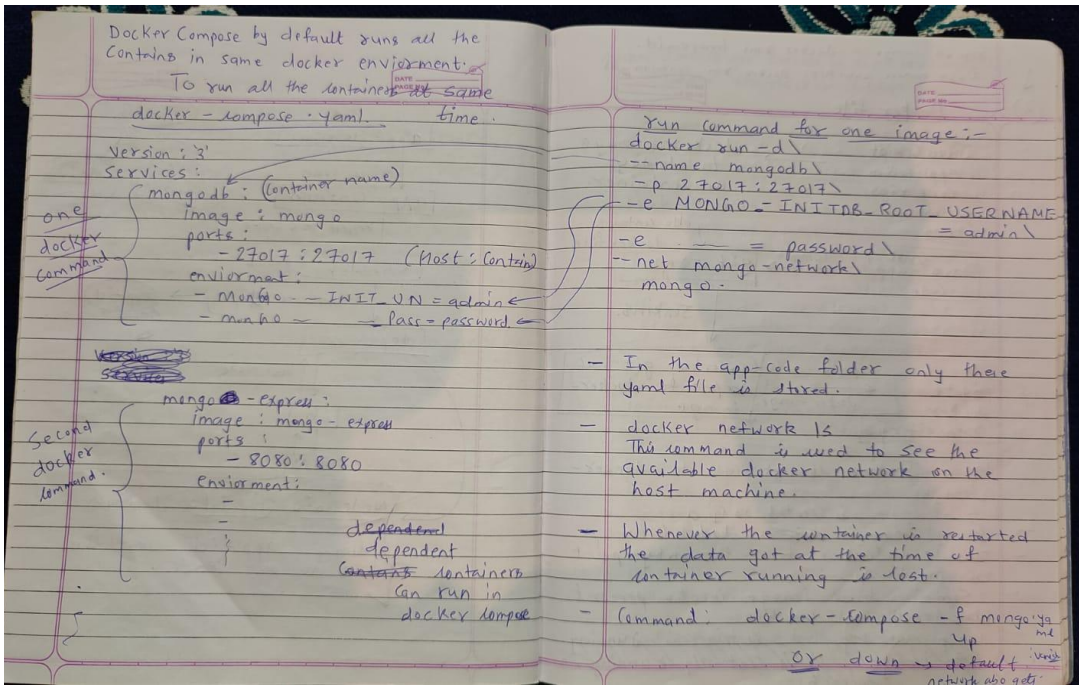
# DOCKER

| | |
|---|---|
| build image | docker build -t <image_name>:<tag> <path_to_Dockerfile> |
| run image | docker run -p <host_port>:<app_port> <image_name>:<tag> |
| list running containers | docker ps |
| list all containers | docker ps -a |
| start container | docker start <container_name/id> |
| stop container | docker stop <container_name/id> |
| remove container | docker rm <container_name/id> |
| remove image | docker rmi <image_name>:<tag> |
| remove all containers | docker rm -f $(docker ps -aq) |
| remove all images | docker rmi -f $(docker images -aq) |
| get container id | docker container ls –quiet –filter name=^<container_name>$ |
| get all info of container | docker inspect <container_name/id> |
| check container status | docker inspect -f '{{.State.Status}}' <container_name/id> |
| live logs of container | docker logs <container_name/id> -f |
| go inside file-system of container | docker exec -it <container_name/id> /bin/sh |
| Expose port | docker run --expose=8080 my-web-app |
| Docker currently used port | docker port <container-id or container-name> |

**Reference:** Docker commands official

# DOCKER

## Additional Images



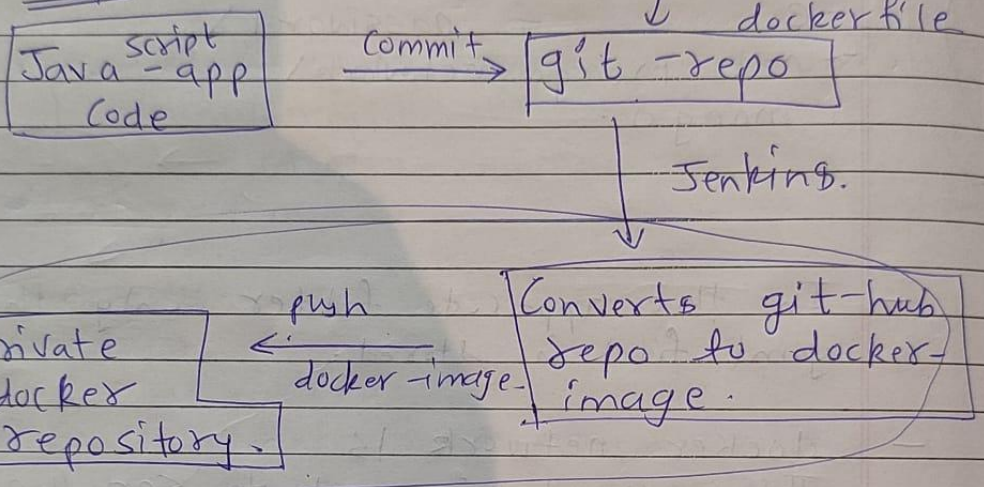
# DOCKER

remove image :- docker rmi image-id.
first remove container: docker rm container-id

(2) Docker file

Java Script application code repo ⟶ docker-image to deploy on container.

Steps :-




How Jenkins convert code to docker-image?

- Every image is created using a docker file.
- Inside docker file the entry-point is mentioned.
- Some base image is also present inside the docker file.
- When we say that container is running on image, it is actually running on docker file.

# DOCKER

docker. Images are stored on the local machine under ~~doc~~ docker folder.

DATE
PAGE No.

~~for exa~~

Running an docker file. image name.

```
docker build -t my-app:1.0 .
```

(. → path to docker file)

- Suppose to run the application of node.js on the container, we want the node enviornment to be present in the ~~doca~~ docker image of the node js application code.
- So firstly the node will be already installed in the image, so when the container runs the image using the start-point command, "node server.js", ~~node~~ on the CLI of container it should not give error that node is not defined. or not installed.
- All the commands written in the docker file will be basically running on the container CMD not affecting the host machine.

# DOCKER

- RUN, ENV, FROM commands will run on the CMD of the container when image is running. but COPY command written in the Dockerfile will be running on the host machine when we ~~fire~~ execute the docker file to build an image.

Questions

- The commands written inside dockerfile, when they are executing.

- Actually what is happening when we build an image of the node using docker file.

Which commands are running when we build image from docker file and which are running when container is running an image.

# DOCKER

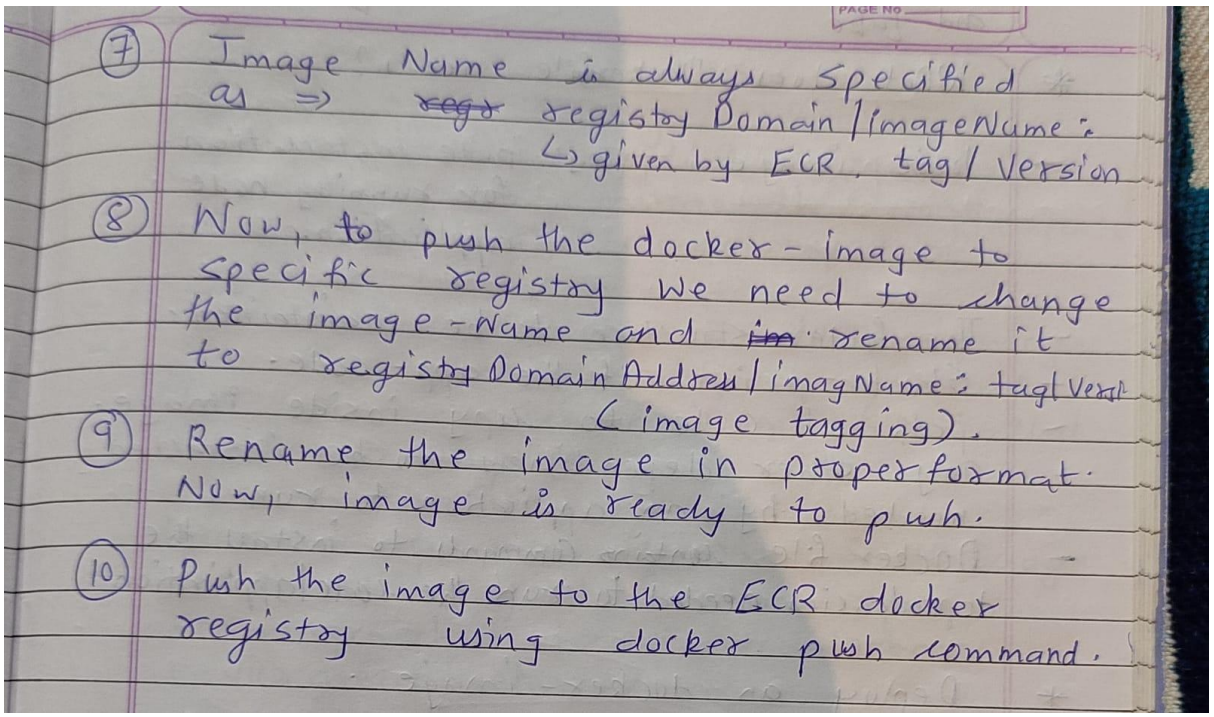
## STEPS TO BUILD and Store docker image :-

Given Information

- Config file.
- Code files.
- Start-point of the code that needs to ~~run~~ execute, which will actually start the application.

1. Make docker file of the app code files using config file info.
2. Keep these docker file inside the main app-code folder.
3. Now, run the docker-file and it will create an docker-image.
4. To store the docker-image in docker registry we Amazon ECR service.
5. In AWS-ECR, We need to make one repo of image-name. Now, these repo will contain different image versions / Tag.
6. Before pushing the image to docker-registry ECR, we need to do login in ECR using the host-machine from where we are going to push the image.

# DOCKER



(7) Image Name is always specified as ⇒ ~~regr~~ registry Domain / imageName : tag / version
↳ given by ECR.

(8) Now, to push the docker-image to specific registry we need to change the image-Name and ~~im~~ rename it to registry Domain Address / imagName : tag / Vers (image tagging).

(9) Rename the image in proper format. Now, image is ready to push.

(10) Push the image to the ECR docker registry using docker push command.

# DOCKER

To run docker first we need to login to the host machine. install docker there and take

* What does Dockerfile needs?

- base image → eg: node installation. for running node application.
- The folder structure to be maintained.
- Start-point of the app-code.
- path of code to copy inside image.
- any other dependicies to be installed needed by an application to run..
- Docker file contains commands to install the dependencies needed to run the code/application.

* Deploy an docker-image.

- Now, we the docker-image ~~Name~~ URL containing the registry domain address, to pull the image and run it on the container.

* Deploy an application using docker-compose (-yaml file).

- Lets g have an application written in java-script, it uses mongo-db and mongo-express to view the data stored in mongo-db.

# DOCKER

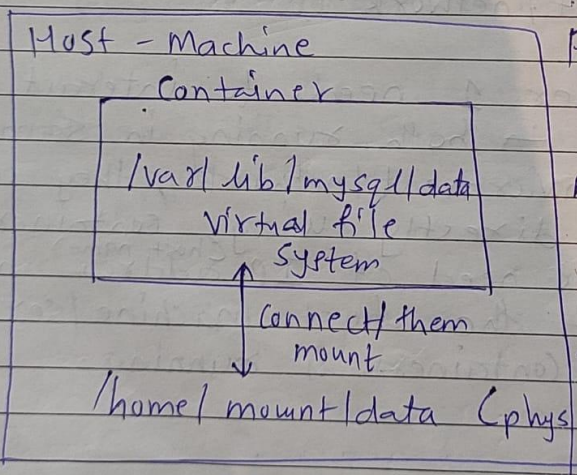
- So, We need 3 images to run,
  1 image for application → interacts,
  1 image for mongo-db ← interacts
  1 image for mongo-express ↗
- To run more than 1 docker-images in same enviorment, at a time we can do using 'yaml' file called docker-compose.
- Docker-compose will run each image mentioned in yaml file on seperate ~~containers~~ but in same docker enviorment/network.
- If container 1 needs to interact with container 2 ~~is~~ both running in same docker-enviorment/network, they can interact directly using container-name (host name) and Port number of the host-machine/server on which container is running.

# DOCKER

* Docker Volumes :-

- Whenever a container is ~~stopped~~ restarted / removed ~~restarted~~ the data gets lost, because the data is stored in the container virtual storage space, once the container is down that space gets released.

- So, lets say I ~~am~~ am running mongo-db on one container, then before stopping the container I need to store / push that data to some permenant / persistent data-storage.




Folder in physical host machine file system is mounted into the virtual file system of docker. (one folder of docker is there in host-machine when it is ~~installed~~).

# DOCKER

- After mounting, the Virtual FS of docker container to the physical FS of host machine, whenever the data is stored in virtual FS it will be replicated on the host-machine mounted folder.

- The folder data that we need to make persist, only that folder of the virtual FS is mounted to the folder of pysical FS on host machine.

3 Methods

① Host Volumes.

```
docker run
  -v /home/mount/data : /var/lib/mysql/data
```

/home/mount/data — host machine physical path

/var/lib/mysql/data — container virtual path.

- you decide where on the host machine file system the reference is made.

② Anonymous Volumes.

# DOCKER

- When the container restarts, these data is first replicated to the virtual FS path where it was mounted.

DATE
PAGE No

docker run -v /var/lib/mysql/data.

- for each container docker will create a folder that gets mounted to physical host machine. FS.

/var/lib/docker/volumes/random hash/_data
(Created ~~they~~ these path in host machine for persistent storage by docker).

#

③ Named Volumes (used in production)

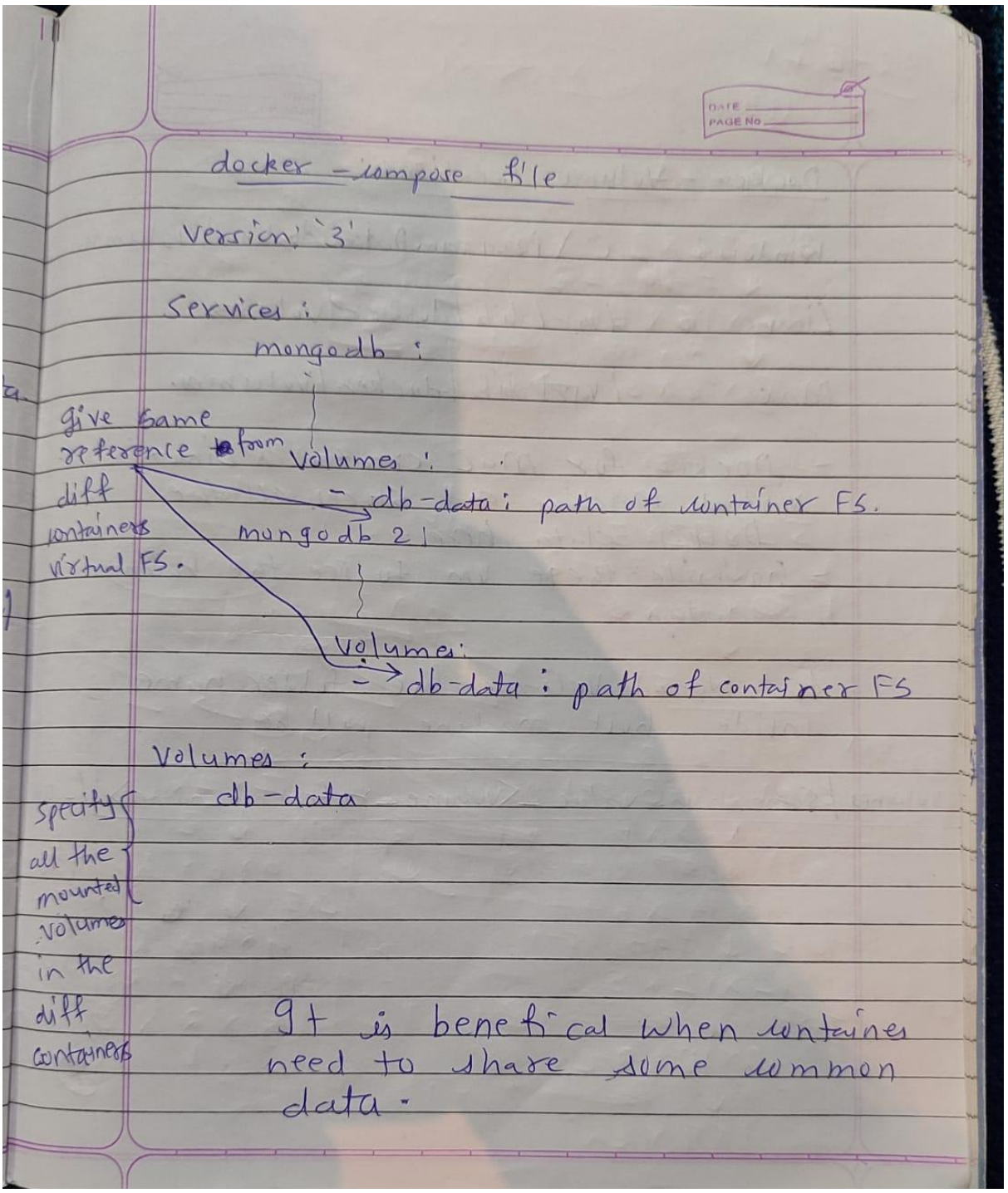
most recommended

docker run -v volume_name:/var/lib/mysql/data

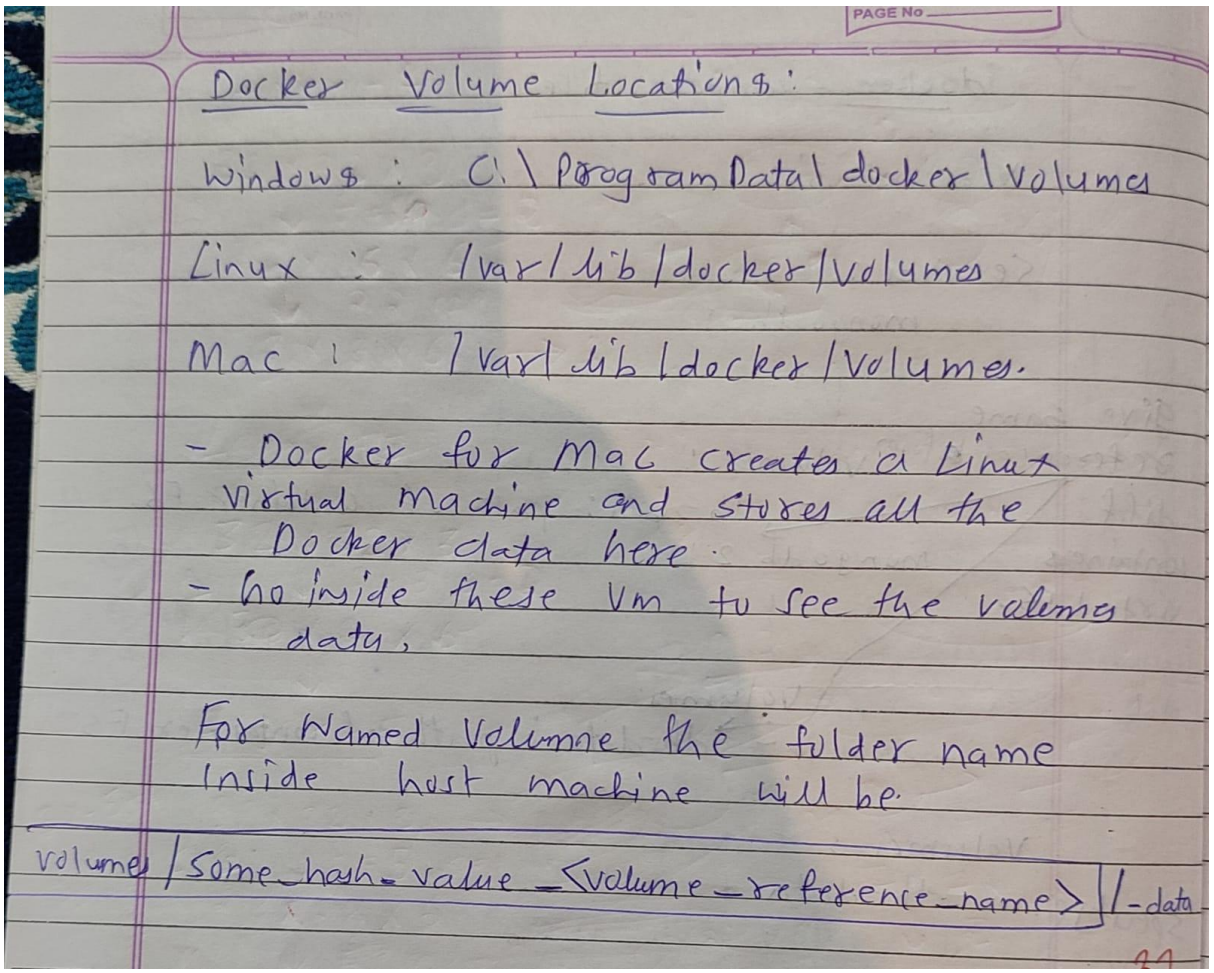
reference → ~~these~~ name directory will be made in physical host machine.

- You can reference the volume by name
- Same reference volume Name can be used by more than one container to store data at same folder on host machine.
- These all volume info in specified in docker-compose file.

# DOCKER


docker - compose file
version: '3'
services:
mongodb:
volumes:
- db-data: path of container FS.
mongodb 2:
volumes:
- db-data: path of container FS
Volumes:
db-data
give same reference from diff containers virtual FS.
specify all the mounted volumes in the diff containers
It is beneficial when containes need to share some common data.

# DOCKER



PAGE No

Docker Volume Locations:

Windows : C:\ProgramData\docker\volumes

Linux : /var/lib/docker/volumes

Mac : /var/lib/docker/volumes.

- Docker for Mac creates a Linux virtual machine and stores all the Docker data here.
- To inside these VM to see the volumes data,

For Named Volume the folder name inside host machine will be.

volumes/some_hash_value_<volume_reference_name>/_data