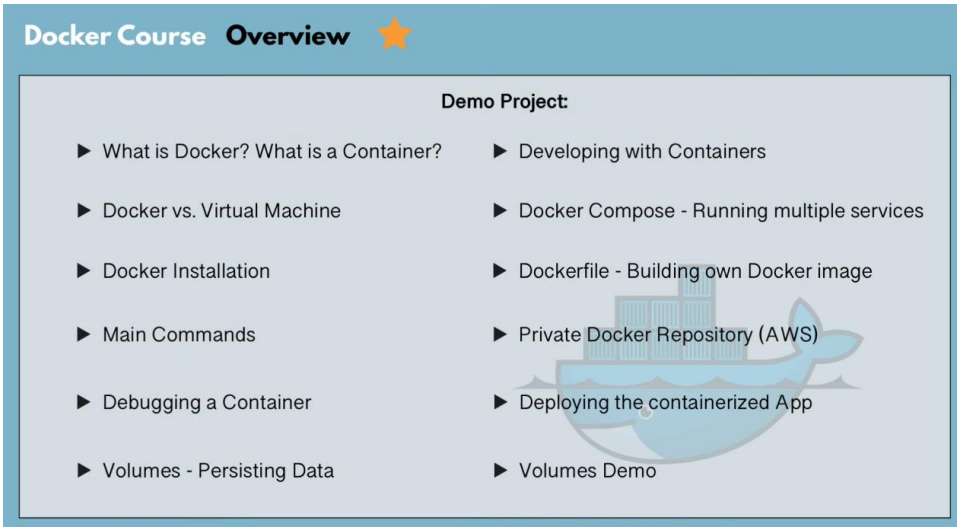
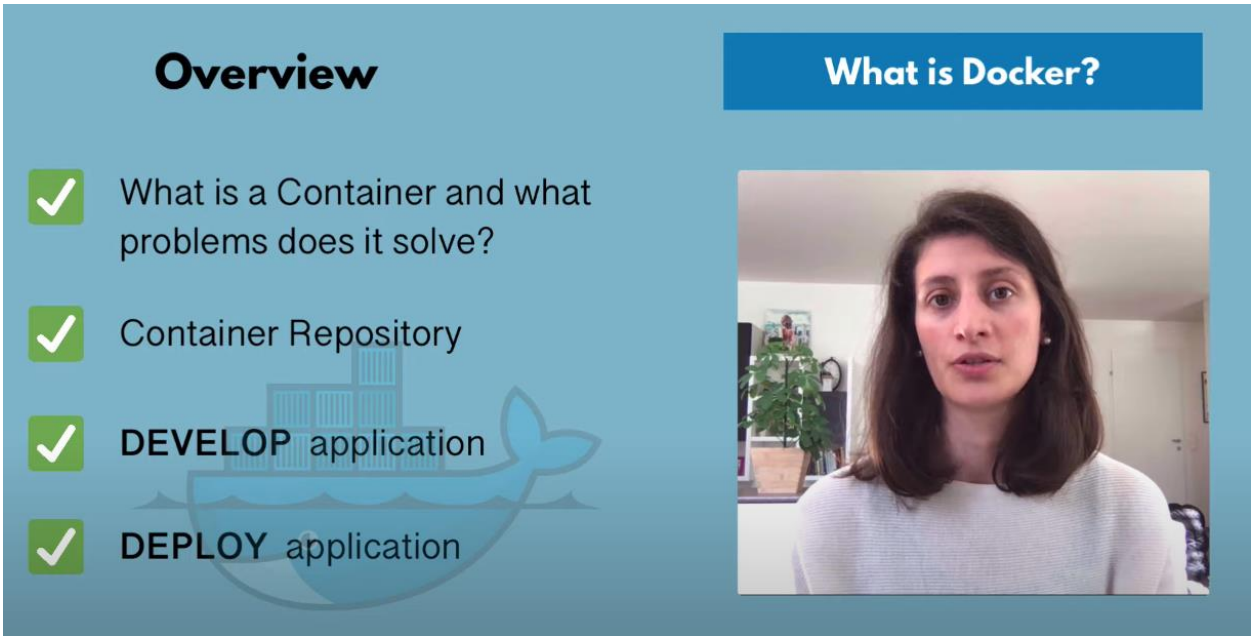
Docker

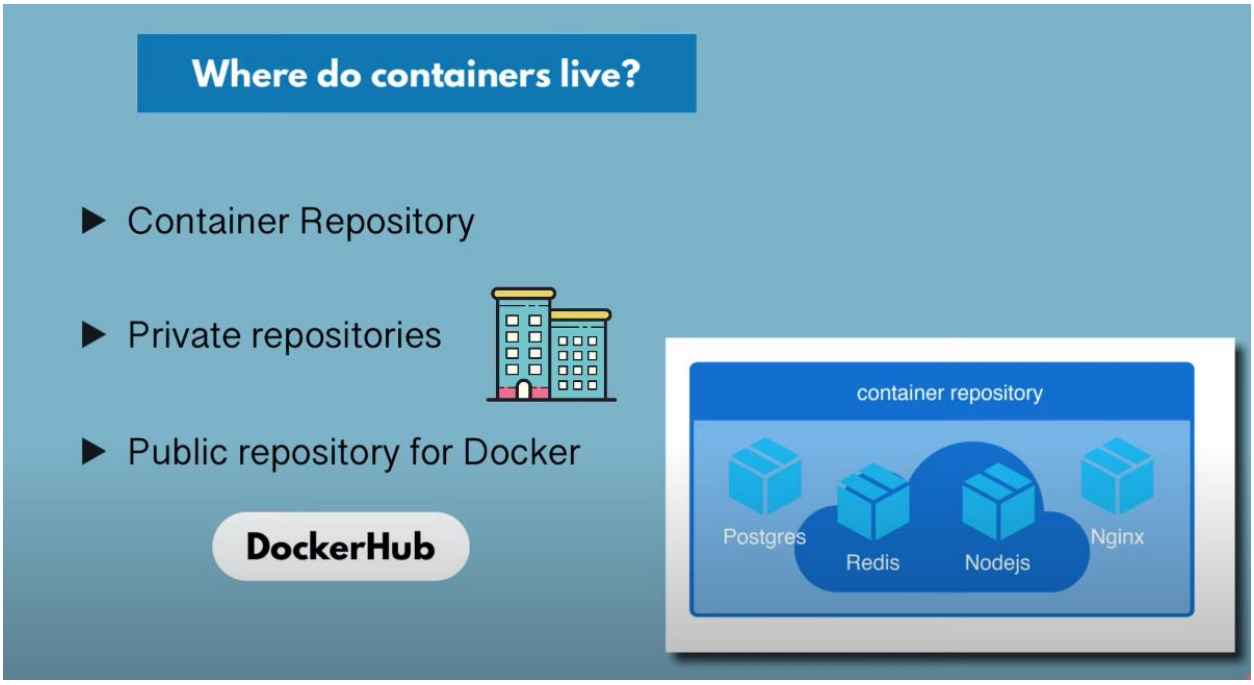


- **What is Docker?**

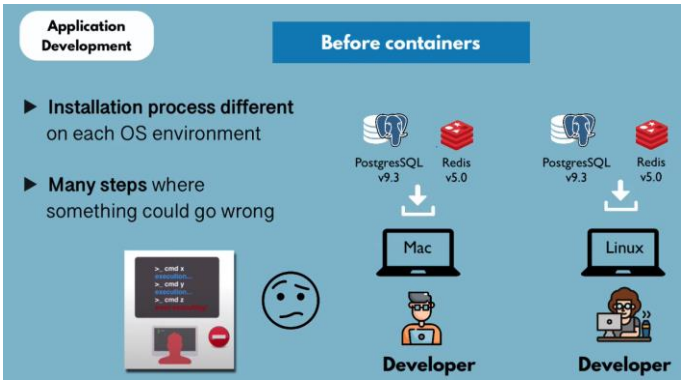


- **What is a Container?**

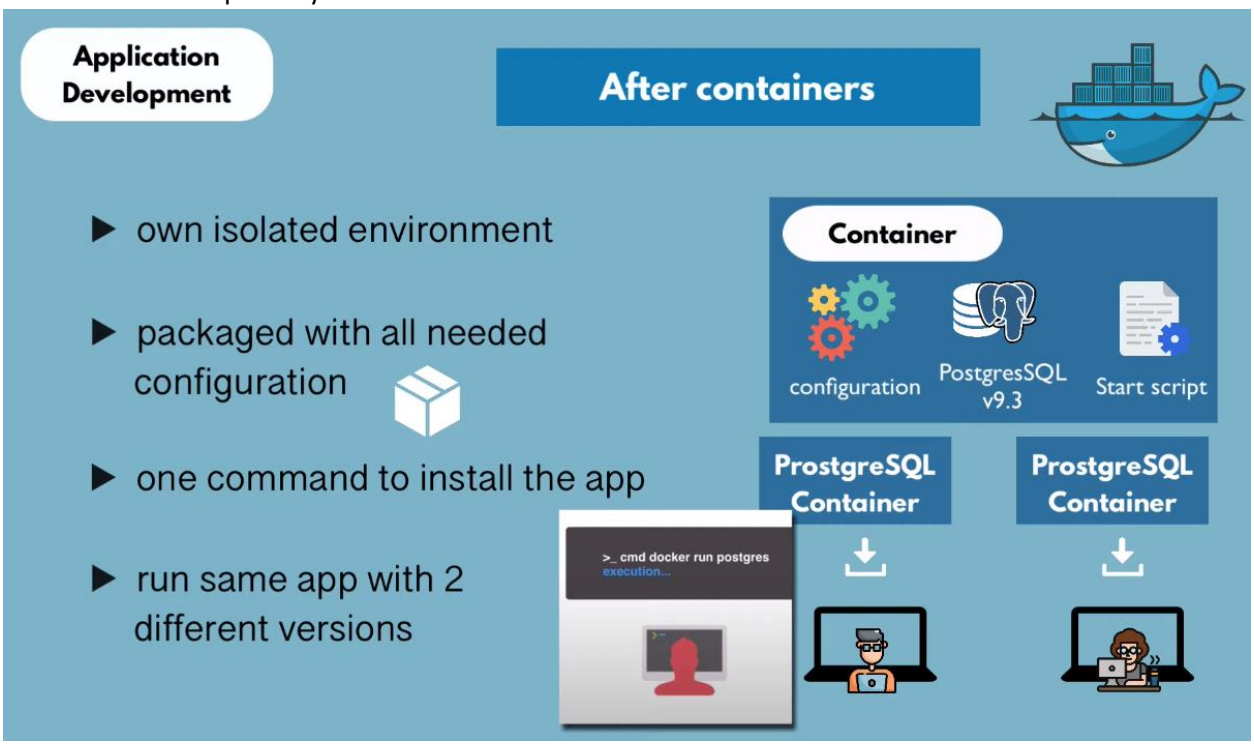
- Since containers are portable, there must be some kind of storage for them so we can store them and move them around.
- **Container Repository** is a special kind of storage for containers.
- Some of the companies have their own private repositories to host and push the containers to them.
- There is also a **public repository** which is called **DockerHub** and it is for docker containers. Where we can find and publish any application container that we want.



- **How containers improved application development?**
- Lets see how do we develop the applications before the containers?
  Usually when we have a team of developers working on developing an application, we need to install most of the services on the operating system directly.
- For example, if we are developing javascript application then every developer need to install binaries of **postgresql** and **redis** and configure and run them in local development environment.
- There will be many steps where something go wrong and this process of setting up an environment could be tedious especially when the application is to complex.

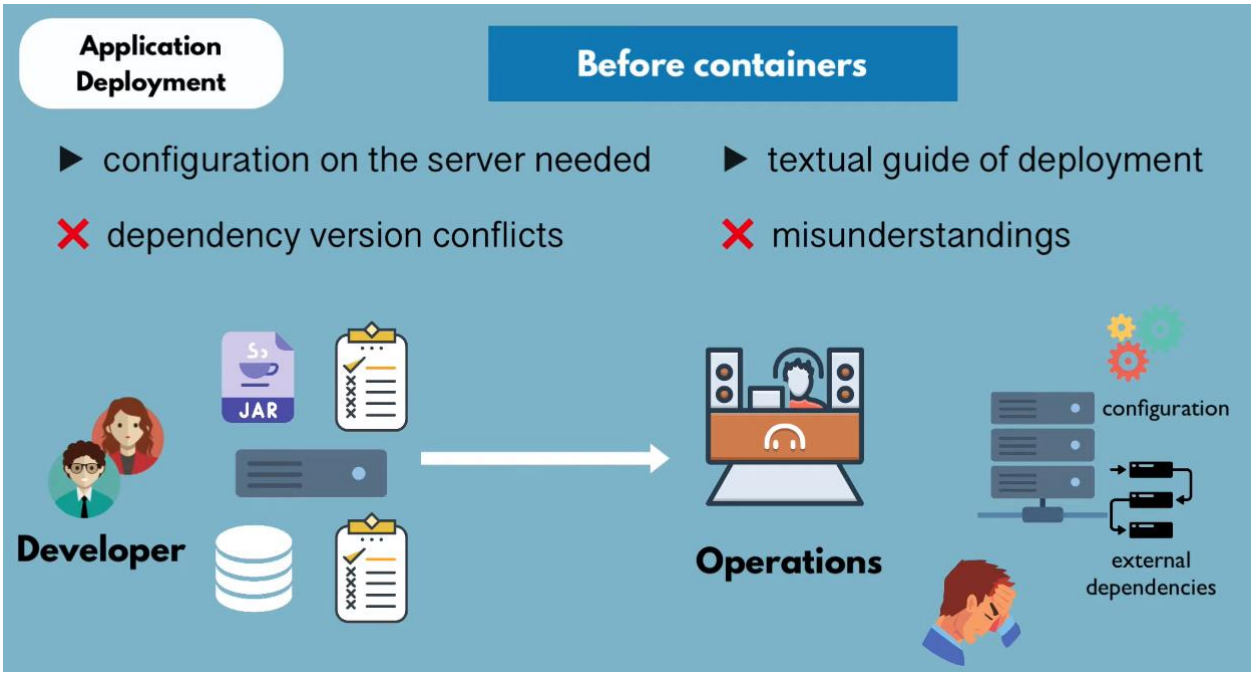


- **Now lets see how the containers actually solved this problem?**
- Before containers , if our application like for example if a website need 10 services like mongodb, sql, and others then we need to install those 10 services in all operating systems.
- Using containers, we do not have to install any of the service directly on our operating system, because containers have their own isolated environment layer with linux based image.
- We will have all the services packaged in the configuration in the start script inside of one container.
- So, instead of downloading the binaries on our machine, we can simply search for the **container** in the container repository and download it on our local machine.

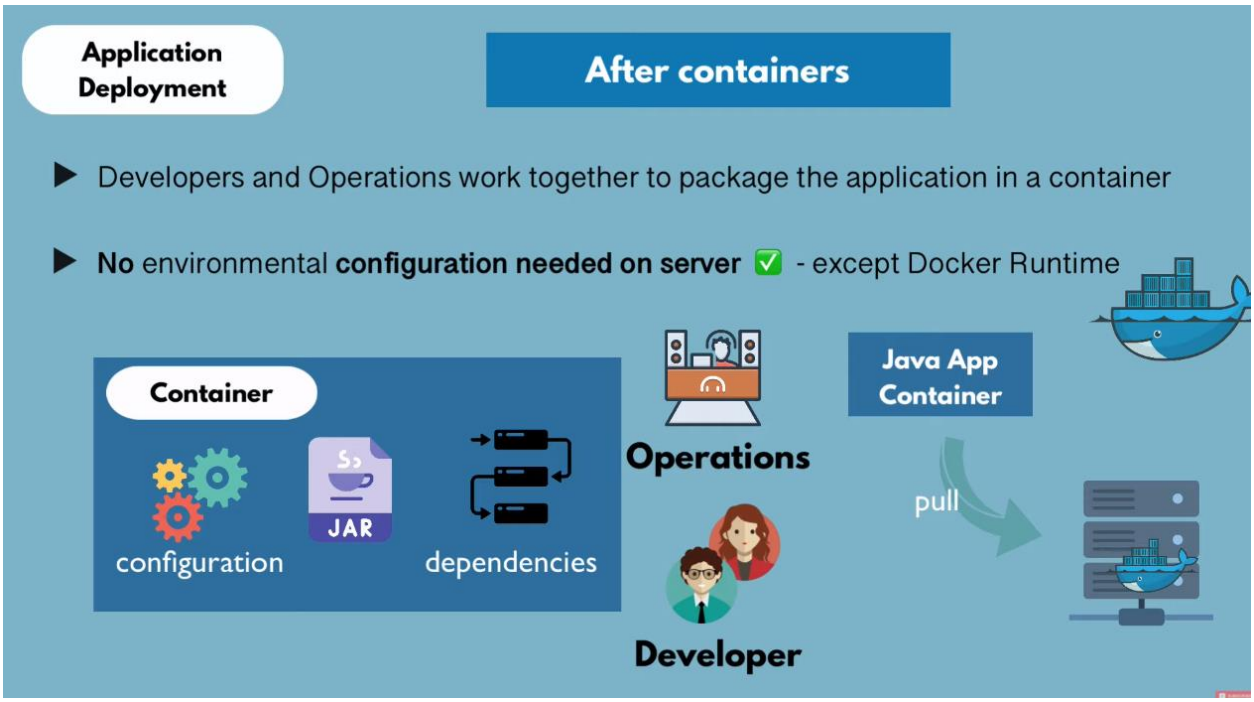


<u>Before Containers</u>

- The development team will produce the artifacts along with the set up instructions to set up in the server and then hand over it to the operations team will handle setting up the environment to deploy those applications.
- The problem with the above approach is that we need to configure everything and install everything on the operating system.

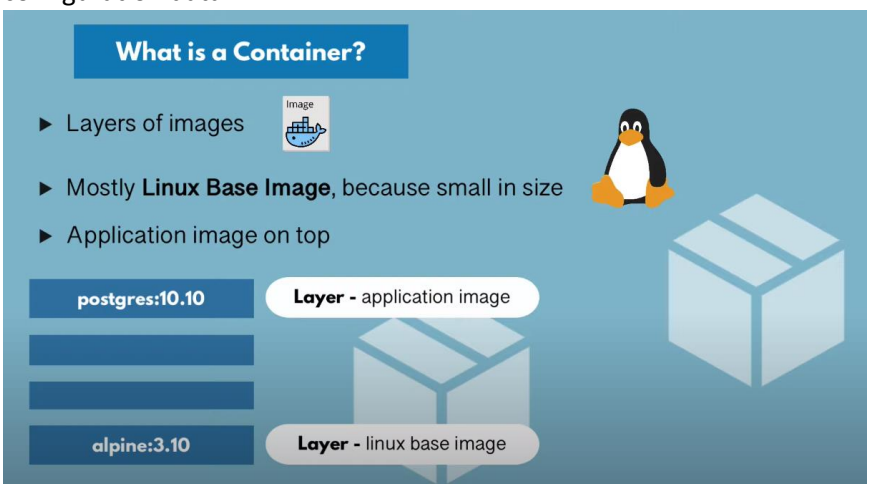


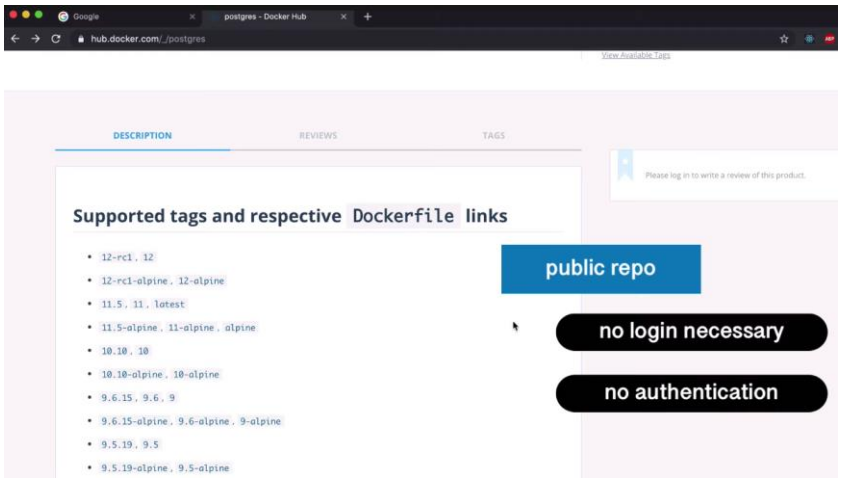
- With containers, this process is actually simplified.
- With containers, we need to just pull the container stored in the repository and run it.
- Also, we need to **install and setup** the docker runtime on the server before we will be able to run the containers but this setting up is only one time effort.



- **What is a container?**

- It is basically a layer of images and we have a stack of images on top of each other.
- At the base of most of the containers, we have linux base image either **alphine with some specific version** or it could be some other linux distribution.
- It is important for the base image to be small in size and that's why most of them are **alphine.** Because that will make sure that the container will stay small in size which is one of the advantage of using the container.
- There will be **application image** on the top and there would be intermediate images that will lead to the **application image** that will run in the container and on top of them we have all configuration data.





- Since **dockerhub** is the public repository we can directly pull the containers with out logging in and we do not need any authentication credentials to access **dockerhub.**
- If we not specified any version, it will just give the latest version of the container image.
- The first line specifies that it is not able to find the image locally , so it knows it need to go to dockerhub and pull it from there.
- We can also oberve **lots of hashes** which says downloading. Actually **docker containers** are made up of layers. It has the linux image layer and the application layer. Below we can see all those layers which are separately downloading from the docker hub on my machine.

```
Last login: Sat Sep 28 10:55:08 on ttys006
Nanas-MBP:~ nanabiz$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
Nanas-MBP:~ nanabiz$ docker run postgres:9.6
Unable to find image 'postgres:9.6' locally
9.6: Pulling from library/postgres
8f91359f1fff: Downloading [==================================================> ] 22.31 MB/22.51 MB
c6115f5efcde: Download complete
28a9c19d8188: Download complete
2da4beb7be31: Download complete
fb9ca792da89: Download complete
cedc20991511: Download complete
b866c2f2559e: Download complete
5d459cf6645c: Download complete
b59ec97820c9: Downloading [=============>                                     ] 13.11 MB/49.33 MB
01e040230c2f: Download complete
d618b32512c9: Download complete
d694ad4e08d5: Download complete
f1fc54212826: Download complete
2debab4418fd: Download complete
```

separate images are downloaded

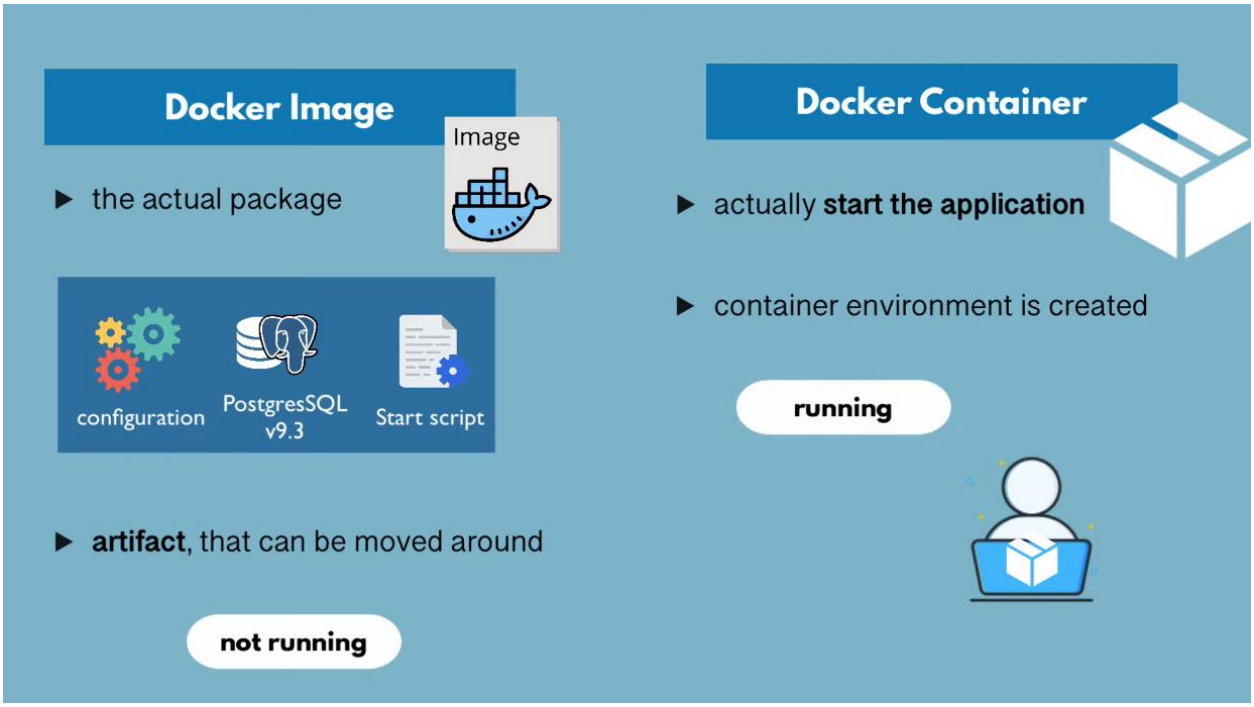
- Advantage of splitting those applications in to layers is actually for example, if the image changes or I need to download the newer version of postgres then the same layers between the older and new version will not get downloaded again.
- **Only different layers are downloaded.**
- Now, it takes 10-15 min to download one image because I do not have **postgres locally.**
- When we want to download the next newer version, then it would take some lesser time because already layers of older version is downloaded.

```
Nanas-MBP:~ nanabiz$ docker run postgres:9.6
```

- The above command will fetches and pulls the container and also executes the start script right away as soon as it downloades it.

```
Last login: Sat Sep 28 10:57:20 on ttys001
Nanas-MBP:~ nanabiz$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
S
fad0f8456ca7        postgres:9.6        "docker-entrypoint..."   45 seconds ago      Up 47 seconds       5432/tcp
eless_haibt
Nanas-MBP:~ nanabiz$
```

- With **docker ps ,** we can list the running containers.
- But, the **postgres:9.6** is listed as Image.
- There are differences between the image and the container.
- Actually the image is the **actual package** which is the combination of **application package + configuration + dependencies.**
- **Container** is when I pull that image on local machine, so that I actually started the container so that the application actually starts that creates the container environment.



- The cmd screen shot above indicates that postgresql is running on my machine.
- If we want to have **postgres newer version** to run on our local machine then we can specify the same command specified above and also here in the image below, we can see that some of the layers of the image are the same and we do not need to fetch them again because they are already present in the machine.

```
Nanas-MBP:~ nanabiz$ docker run postgres:10.10
Unable to find image 'postgres:10.10' locally
10.10: Pulling from library/postgres
8f91359f1fff: Already exists
c6115f5efcde: Already exists
28a9c19d8188: Already exists
2da4beb7be31: Already exists
fb9ca792da89: Already exists
cedc20991511: Already exists
b866c2f2559e: Already exists
5d459cf6645c: Already exists
6de9d066d892: Downloading [==================>
401fcd8e29c4: Download complete
9b130e26214a: Download complete
1c048e77610c: Download complete
431b5e6c27b3: Download complete
4eca80d7c24a: Download complete
```

some layers already exist ✅

- It fetches the layer that are different and it would save a lot of time and more advantageous.
- We can clearly observe that **postgres 9.6** and **postgres 10.0** are running in different tabs.

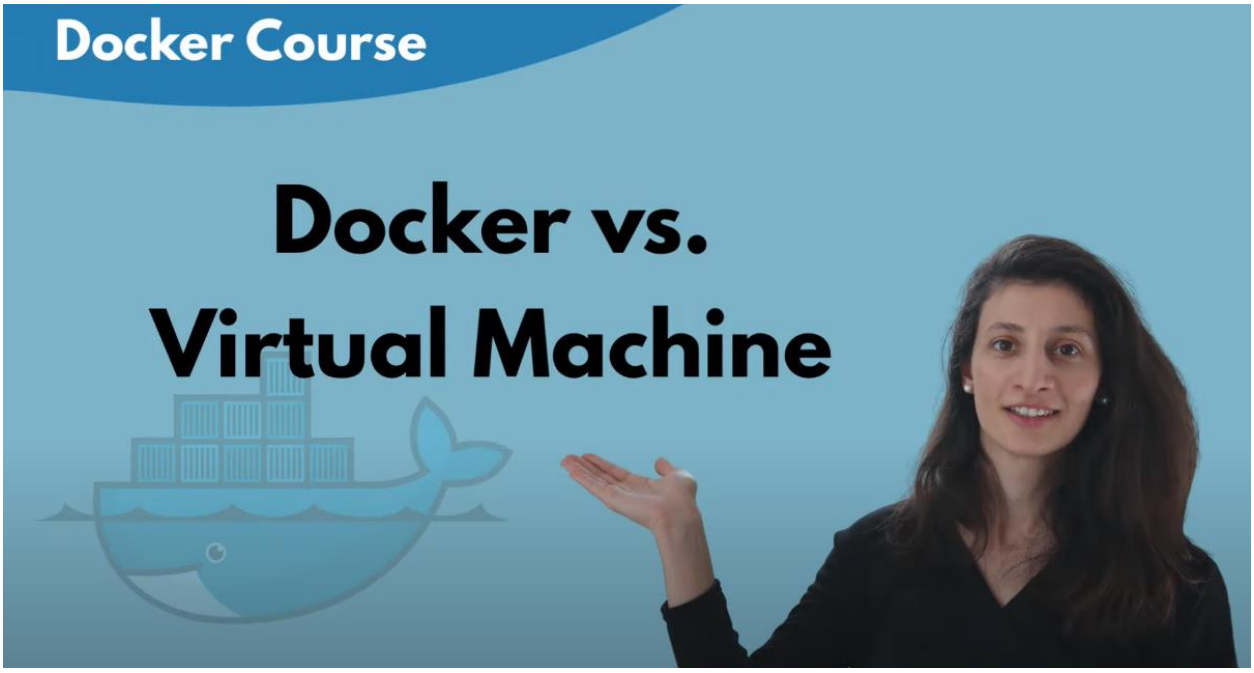
```
~ — docker run postgres:9.6                    ~ — docker run postgres:10.10
        effectively any other container on the same
        system.

        Use "-e POSTGRES_PASSWORD=password" to set
        it in "docker run".
****************************************************
waiting for server to start....2019-09-28 09:12:20.507 UTC [44] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL
"
2019-09-28 09:12:20.516 UTC [45] LOG:  database system was shut down at 2019-09-28 09:12:20 UTC
2019-09-28 09:12:20.519 UTC [44] LOG:  database system is ready to accept connections
 done
server started

/usr/local/bin/docker-entrypoint.sh: ignoring /docker-entrypoint-initdb.d/*

waiting for server to shut down....2019-09-28 09:12:20.602 UTC [44] LOG:  received fast shutdown request
2019-09-28 09:12:20.604 UTC [44] LOG:  aborting any active transactions
2019-09-28 09:12:20.606 UTC [44] LOG:  worker process: logical replication launcher (PID 51) exited with exit code 1
2019-09-28 09:12:20.606 UTC [46] LOG:  shutting down
2019-09-28 09:12:20.616 UTC [44] LOG:  database system is
 done
server stopped

PostgreSQL init process complete; ready for start up.
```

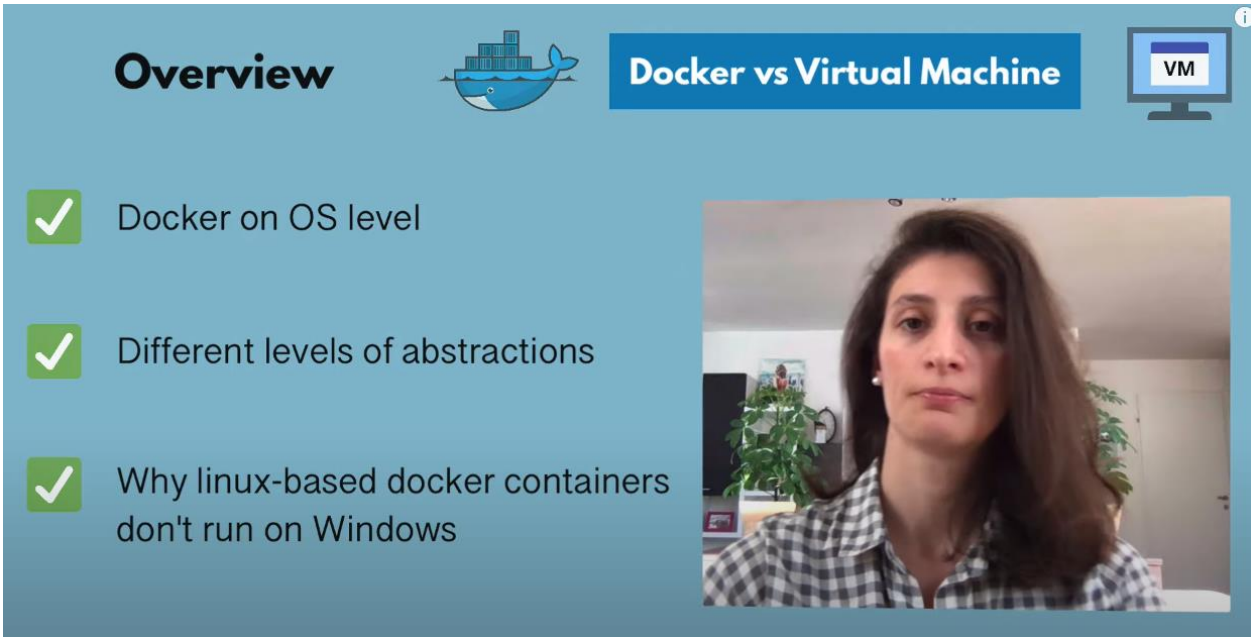
postgres:10:10 running..

- So, now we have two different postgres versions running.

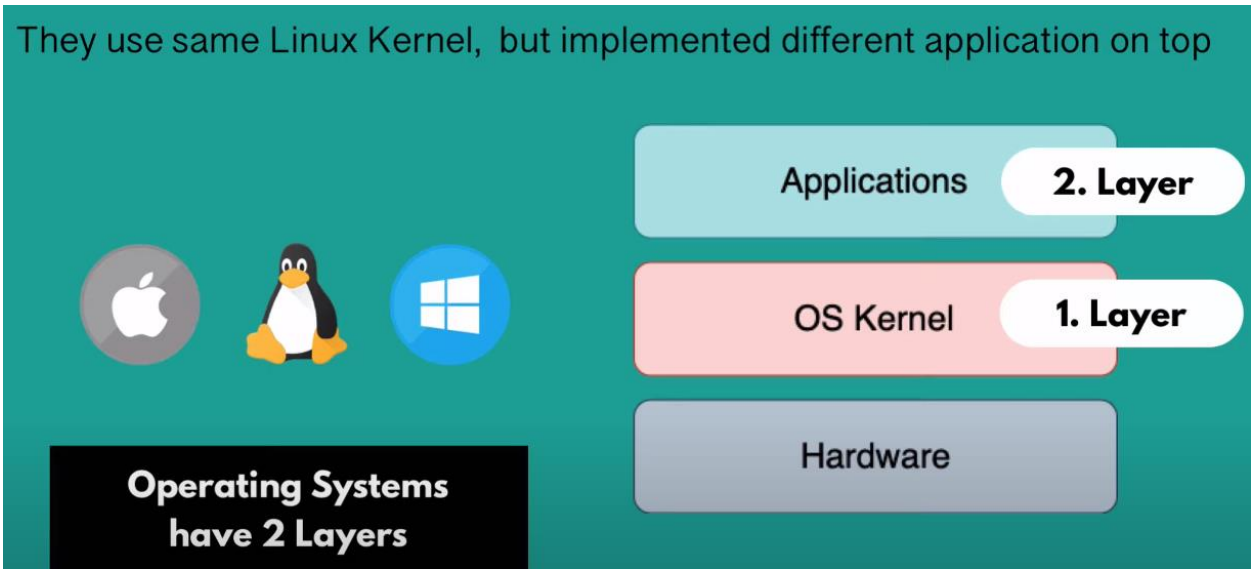
```
Last login: Sat Sep 28 11:08:17 on ttys003
Nanas-MBP:~ nanabiz$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
NAMES
8633ea431a47        postgres:10.10      "docker-entrypoint..."   18 seconds ago      Up 20 seconds       5432/tcp
ecstatic_stallman
fad0f8456ca7        postgres:9.6        "docker-entrypoint..."   5 minutes ago       Up 5 minutes        5432/tcp
priceless_haibt
Nanas-MBP:~ nanabiz$
```

- We can run any number of applications of different versions and there is no problem.

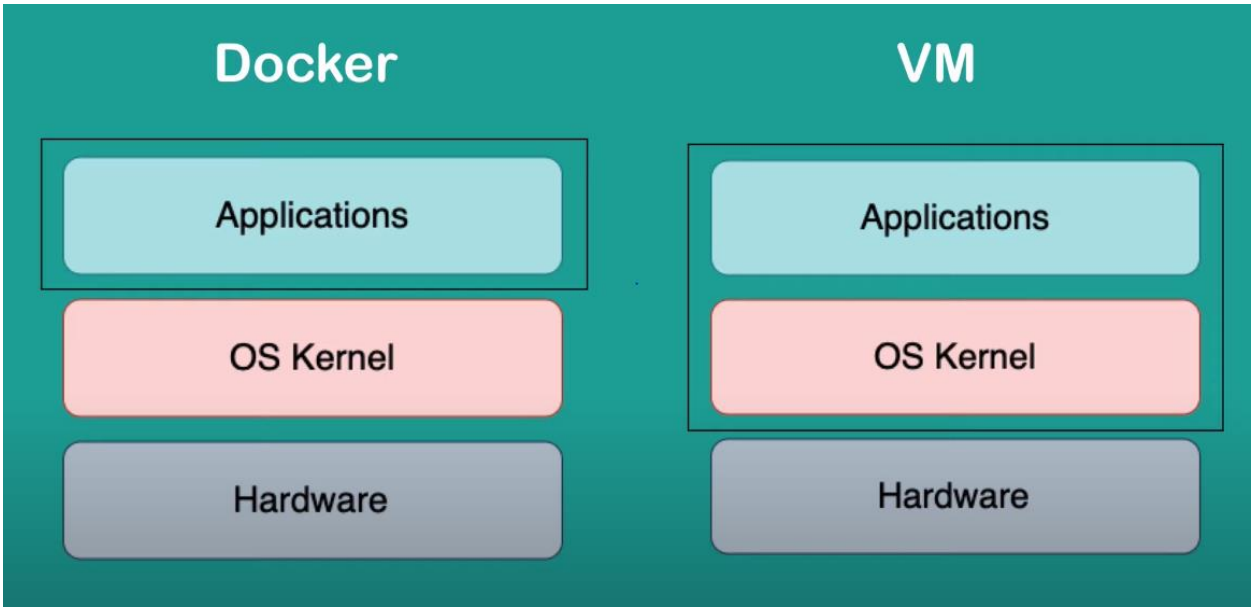




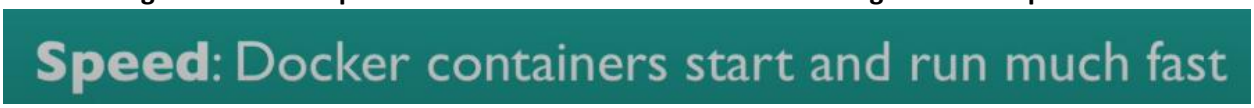
- In order to understand how docker works on the operating system level, lets first look at how operating systems are made of.
- **Kernel** is the part that communicates with the **CPU, Memory etc**
- **Applications** will run on the kernel layer.
- We know that there are different types of distributions of linux operating system.
- So the linux distributions like Debian, mint, fedora and so **all of them use the same linux kernel but different applications are implemented on top of them.**



- We know that both **Docker** and **virtual box** are virtualization tools. But the question is what part of operating system do they virtualize?
- **Docker** virtualizes the **Application layer.**
- So ,when we download the docker image it contains the **application layer** of the operating system and some other applications installed on top of it and it uses the **kernel of the host** because It doesn't have its own kernel.
- The virtual box or the virtual machine on the other hand, has its own application layer and the kernel. So, it virtualizes the complete operating system which means that when we download the virtual machine image on the host, it doesn't use the host kernel rather it boost its own.



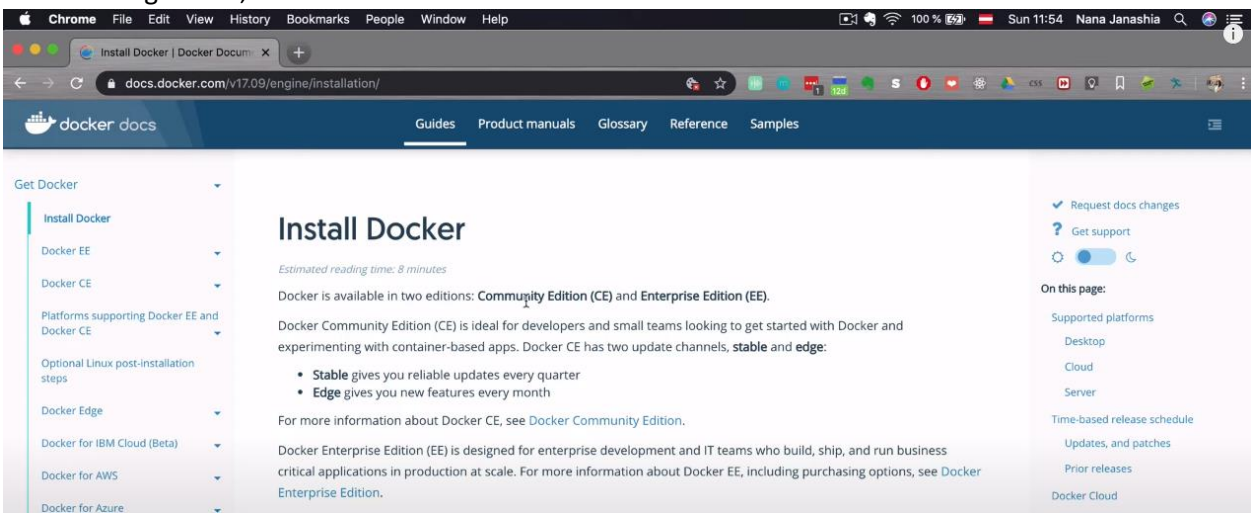
- So , what does the difference between **docker** and **virtual machine mean?**
- **Firstly, the size of docker image are much smaller because they just need to implement one layer.**
- **Docker images can be a couple of MB where as the virtual machine images are a couple of GB.**



- Because everytime, we start the **VM** they need to boot the operating system kernel and the applications on top of it.

- But, we can not do it using docker because let say we have windows operating system with some kernel and applications and we need to run linux based docker image on that windows host. The problem is that docker based linux image might not be compatible with the windows kernel.
- This is actually true for the windows version below 10 and also for old version of mac.
- If suppose we searched for how to install **docker** on different operating system then the first step is to check whether our host can run docker natively or not which basically means that the kernel compatible with the docker images or not.
- In such case, we need to install a **Docker Tool Box** which abstracts the way for the kernel to make it possible for the host to run different docker images.

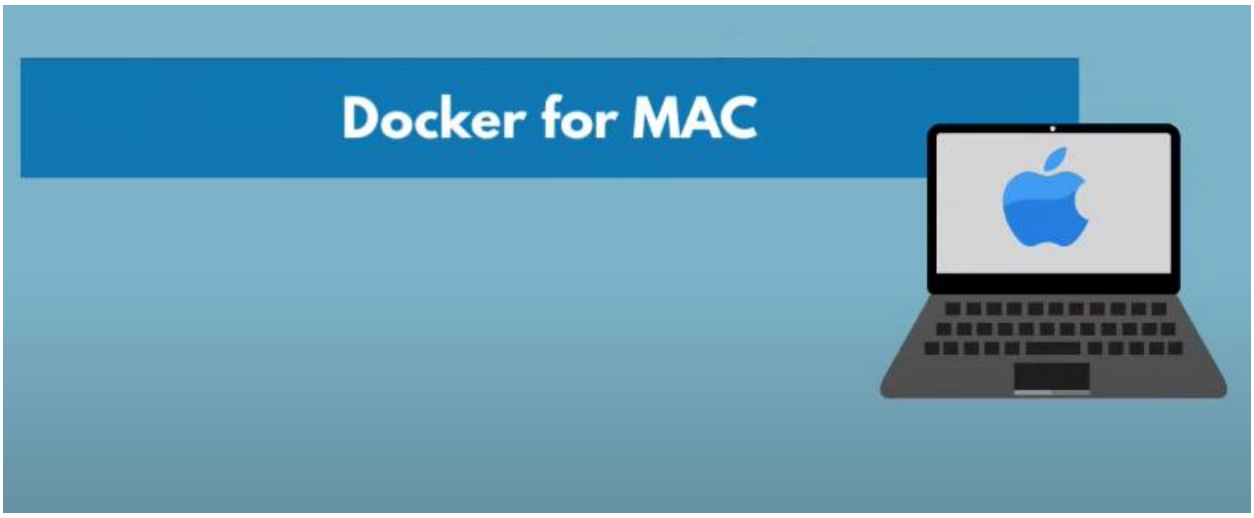
- There are two editions of docker which are community edition and the enterprise edition.
- For us to begin with,



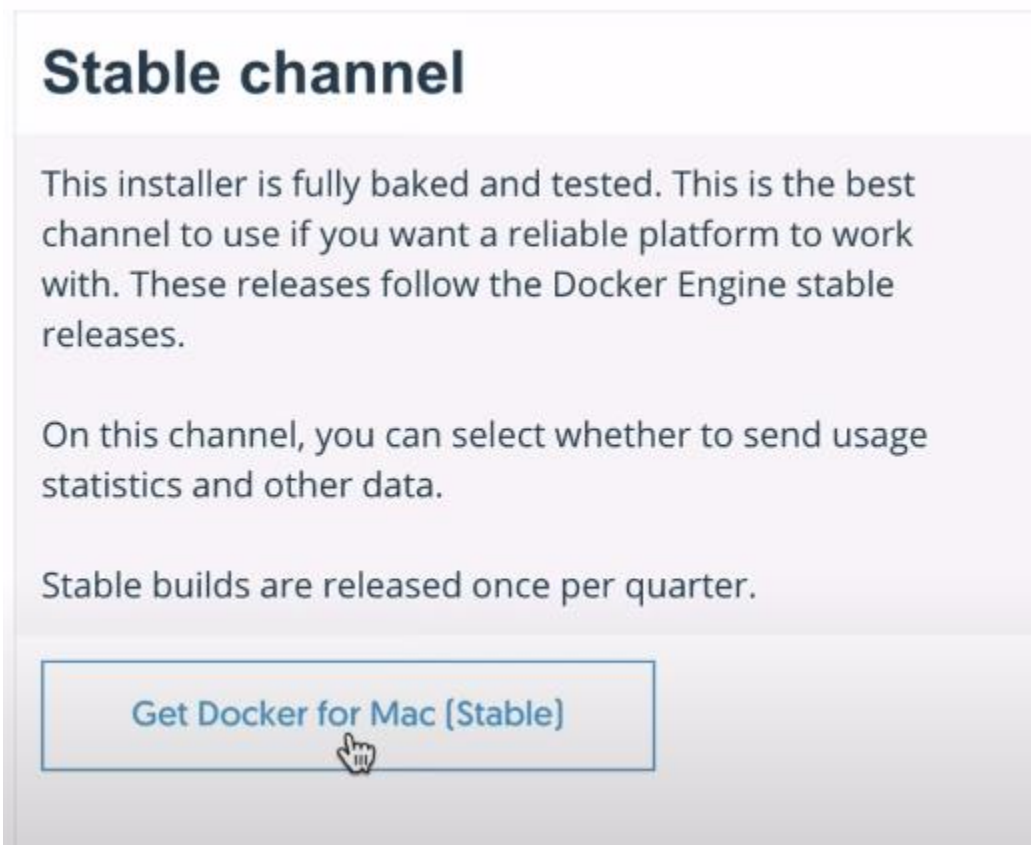
- We can choose the community edition and there is a list of operating systems like windows, mac and linux.

- For mac and windows, there are some criteria of the hardware and the operating system met in order to support running docker.

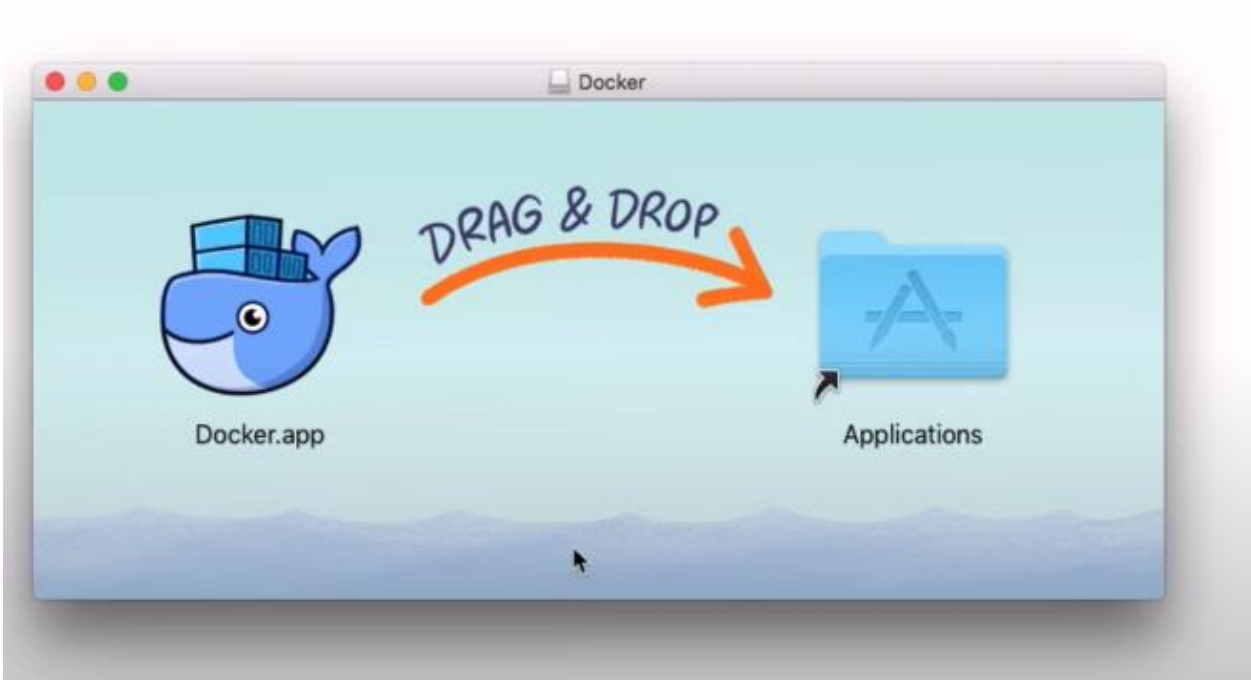
- If our system does not meet the requirements then there is a work around for docker which is **docker tool box** that will actually become a bridge between our system and docker.
- This will enable docker to run on legacy computer.



- We will go with the stable channel of docker, by installing docker we will have the whole package of it which is **Docker Engine** necessary to run the docker containers on our laptop.
- **Docker CLI Client** which will enable us to execute the docker commands.
- **Docker Compose** is just a technology to orchestrate when we have multiple containers.



- Once the docker.dmg file is downloaded then docuble click on it and **drag the whale application** to the **applications folder.**



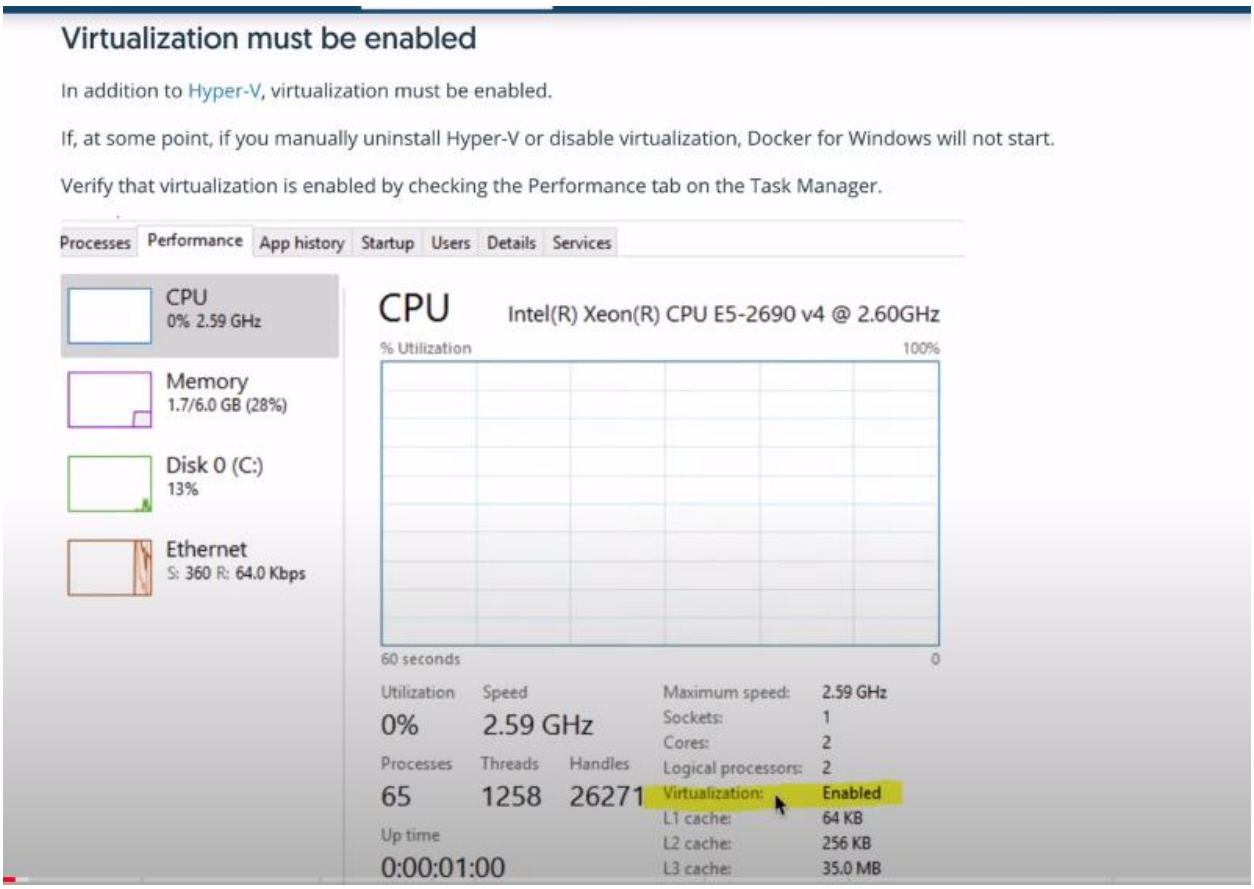
- Now we will see that **docker installed in our applications.**

- We can go ahead and start it and see the whale icon and it states that docker is running.

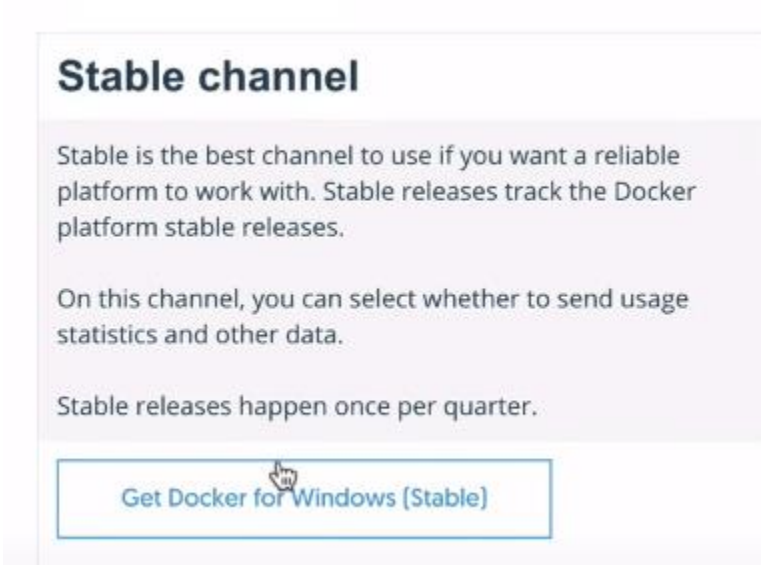
- We can configure some preferences.

- In the above image, we can see the **Quit Docker** option which is useful to quit docker.
- So , suppose we download and install docker and we have more than one accounts in the laptop, we can get errors or conflicts at the same time when running with multiple accounts.
- So, if we switch to another account where we need docker , we quit docker from the old account and start it in the new account. So that we do not get any errors.

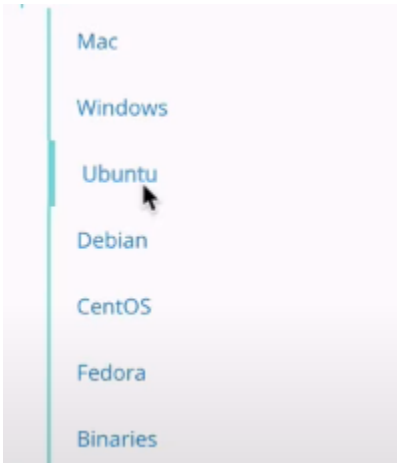
- First we need to go to before you install section and see whether our operating system and computer meets all the criteria to run docker natively.
- First thing to check is whether windows is compatible with docker and the second one is to check **virtualization is enabled or not.**
- If we want to check whether virtualization is enabled or not then go to below image location and view there.



- Then once we check whether the two criteria are met, then we can download the **stable version.**

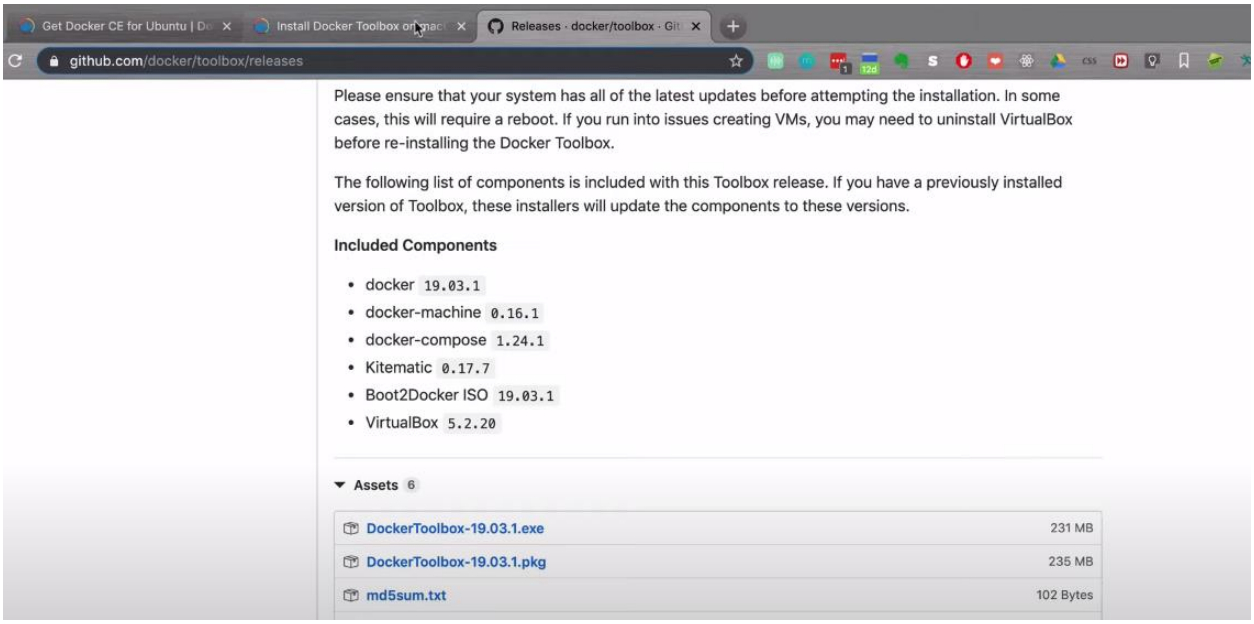


- Now let us see how to install docker for different linux distribution.
- For different linux distribution, the installation steps will differ.

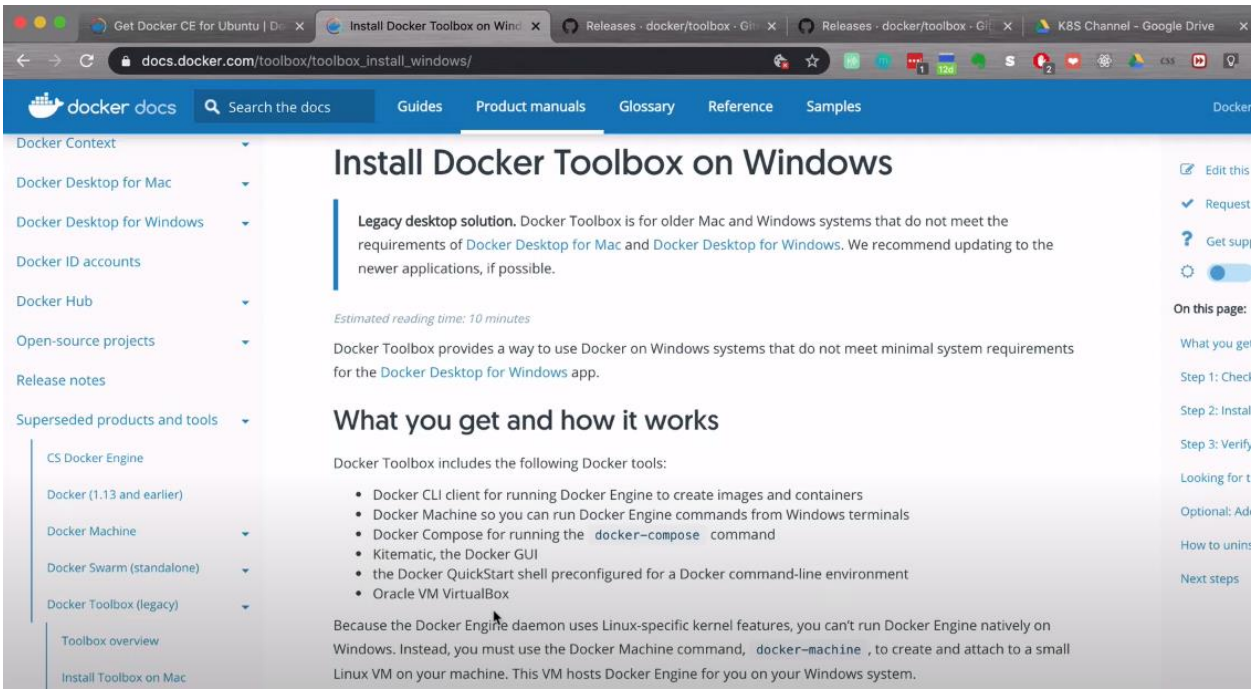


- For environments that do not support docker running natively , there is a work around which is called **Docker Toolbox.**
- It is basically an installer for docker environmental set up on those systems.

- Through the below image, we can see the two files are there and they are the installers for the docker tool box.



- Samwe way for windows too, we will get the package tool box same for native docker installation for windows.



Docker Course

Basic Commands

Overview

- Container vs Image
- Version and Tag
- Docker Commands

docker pull **docker run**

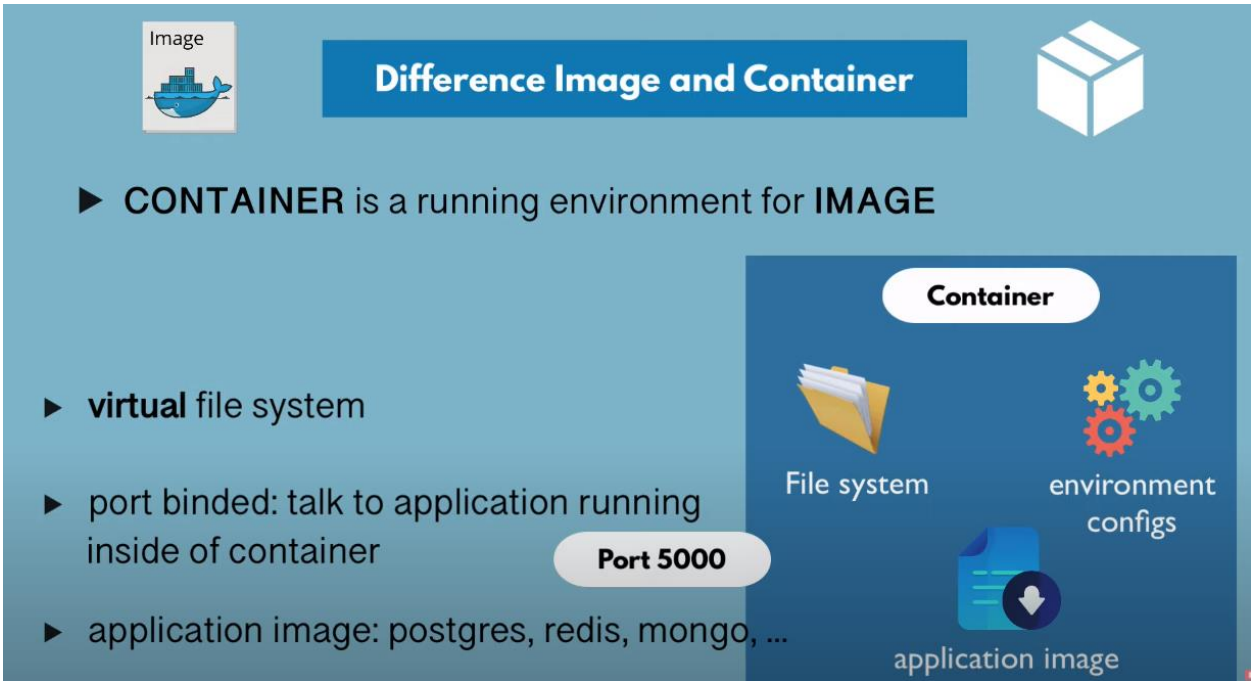
docker start **docker stop**

docker ps

docker exec -it **docker logs**

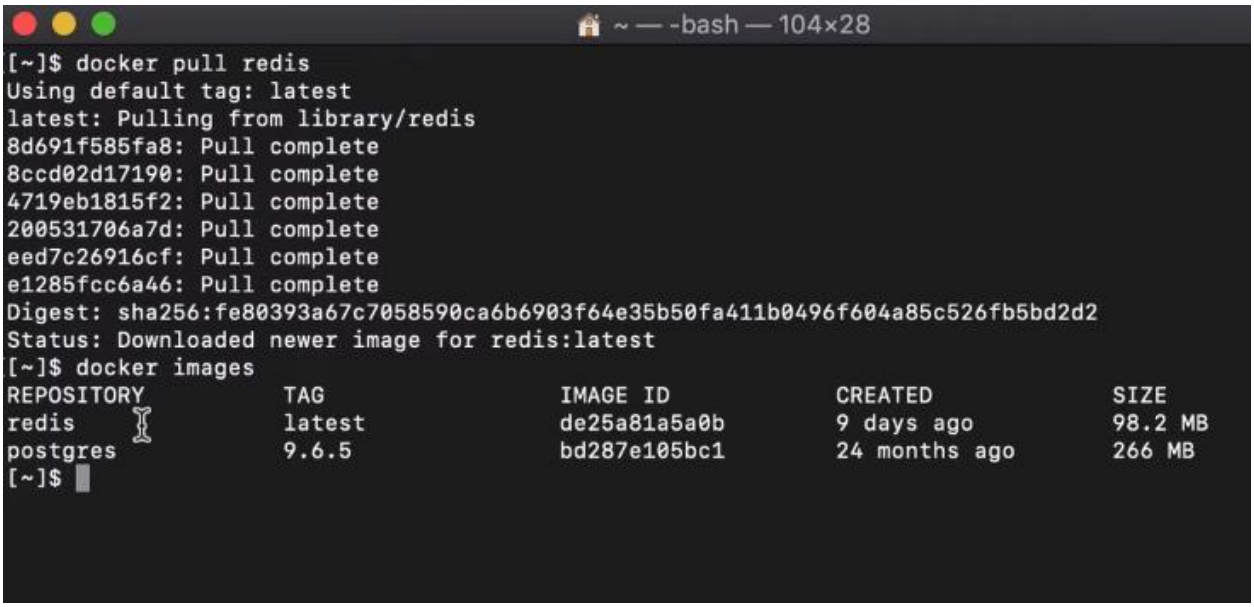
Docker - Basic Commands

- There is a fine difference between **container** and **image.**
- To speak theoretically, **container** is a part of the **container run time.**
- For example, in the image shown below, we have an application image that runs the postgres, redis or mongo needs a **files system** where it can save the log files and store some configuration files and can do some environmental configuration like environmental variables and so on.
- **All these environmental stuff are provided by container.**
- Container also has a **port** binded to it which makes it possible for the container to talk to the application running inside it.
- We need to note that the **file system is virtual in containers** and the containers has their own abstraction of operating system and file system which is of course different from that of the host machine.

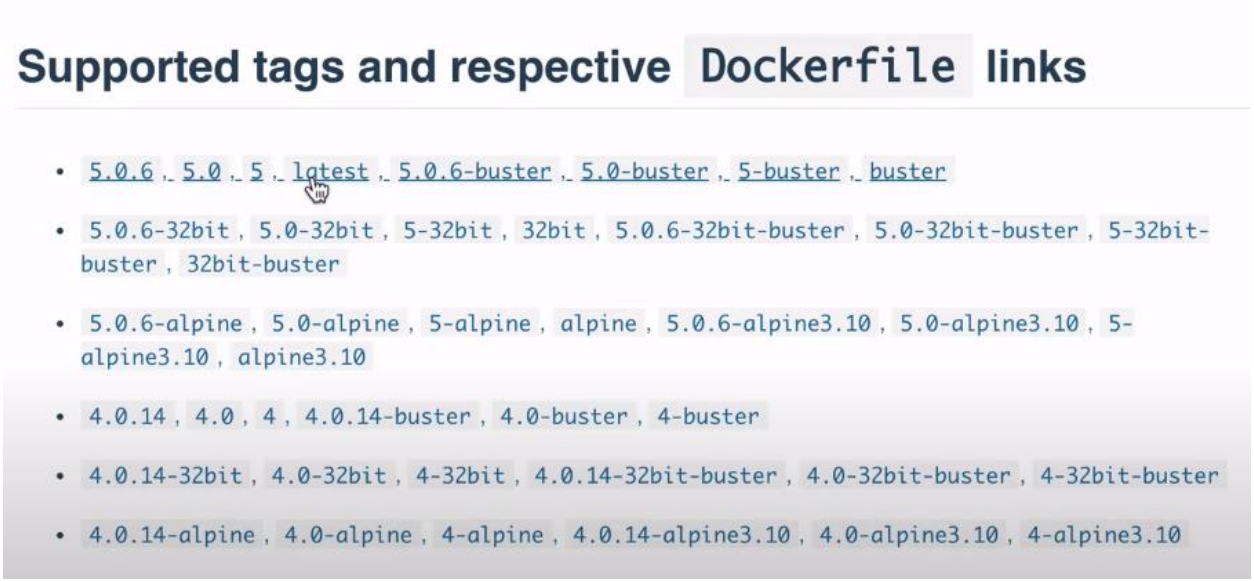


- Go to docker hub and find the redis image. All the artifacts in the dockerhub are images.

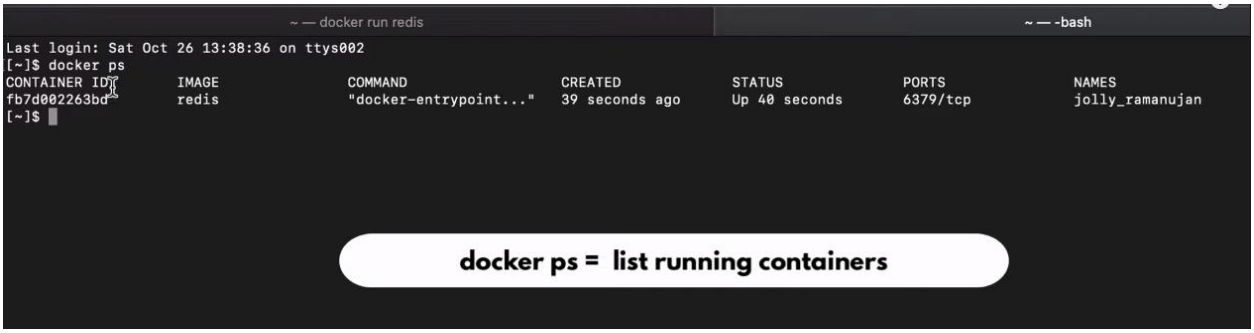
- We can see different layers of images are downloading.
- Once the image is downloaded successfully, we can check all the existing images on our desktop using **docker images command.**



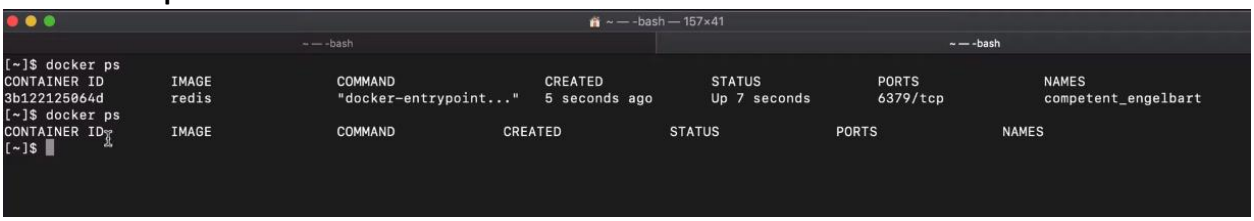
- Another important thing about images is that they have **tags or versions.**
- Each image in the docker hub has a specifi9c version and if we don't specify the version, then **docker will automatically download the latest version.**



- we can also see the size of the image.
- Until this point, we have only seen the **image** and we have no containers running.
- Now, lets say we need **redis to be running** so that our application can connect to it.
- For that, I have to create the container of that redis image that will make it possible to connect to the redis application.
- The command **docker run reddis** will actually start the image in the container.
- We know that container is the running environment of an image.



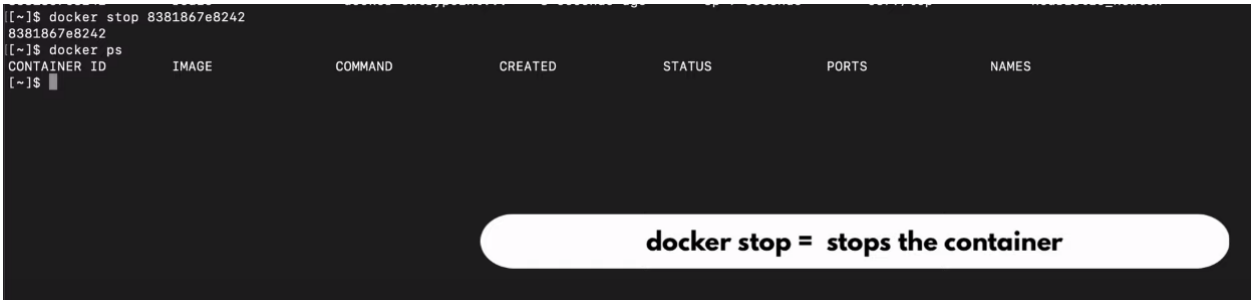
- Now , redis is running and the container id is based on the image of redis.
- The command **docker run reddis** command will start the container in the terminal in an attached mode.
- So, if we terminate this with **ctrl + c** then we see that **redis application** is **stopped** and the **container stops as well.**



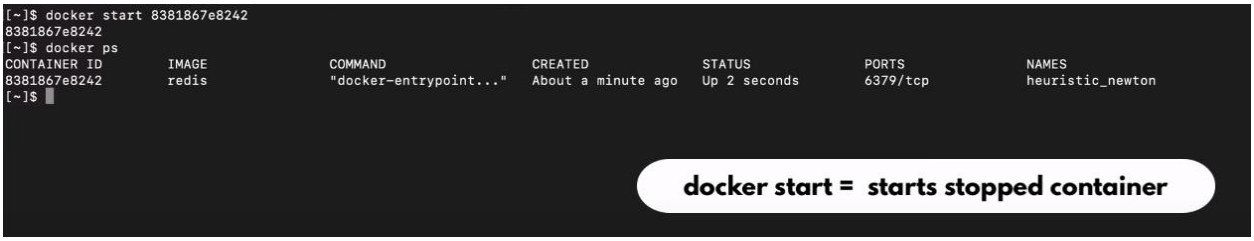
- There is a an option for **docker run command** that makes it to run the container in the detached mode.

- Here , we would just get the **id of the container as the output** and the **container will start running.**

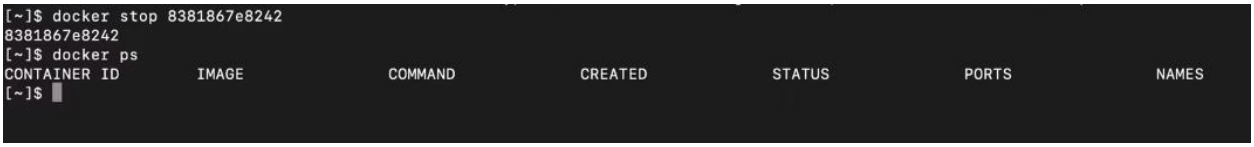
- If we want to **restart the container , because some application crashed inside it** then we will use the command which is shown below.



- Now, the container is stopped and if we want to start it again then use the command **docker start id_of_the_container.**

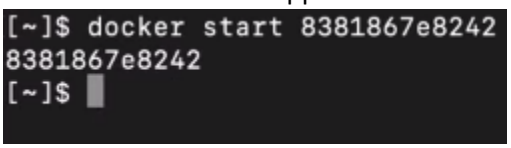


- Suppose, we stopped the docker container and went home and came back and if we want to restart the stopped container then the output will be empty.



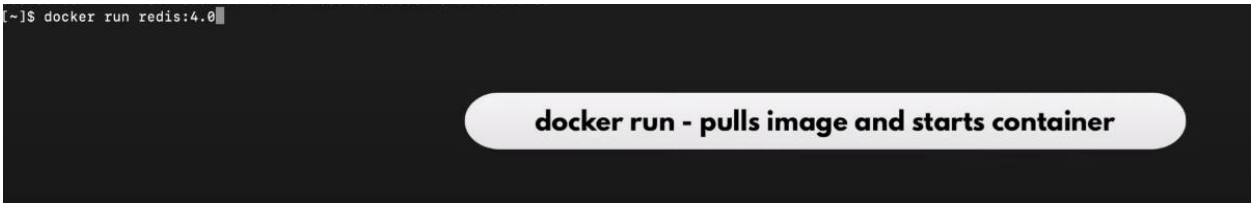
- In this case, if we want to view the history then we can use the command which is **docker ps -a** which lists all the running as well as stopped containers.

- Now we can see the stopped container id in the above image and we can start it again.

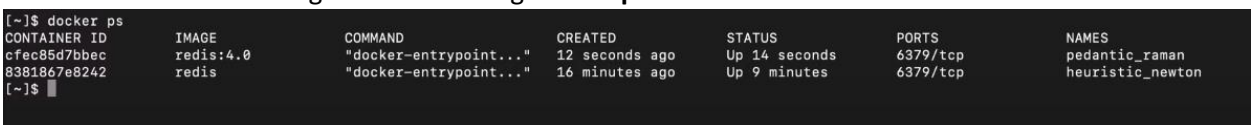


- So ,lets say we have two parts of the application where both use redis but in different version.so we need two redis containers running two images of different versions running on the laptop at different times or same time may be.
- Already we have the latest one, which is **5.06**

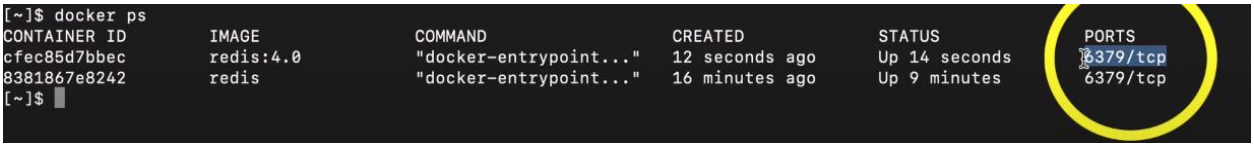
- So,we can use the command **docker run** to pull and start the container right away.



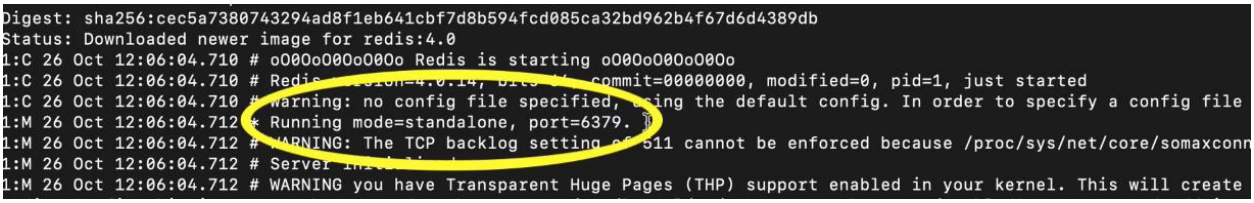
- Now some layers of images will be downloaded and the container would be started immediately.
- Now let us see the running containers using **docker ps** command.



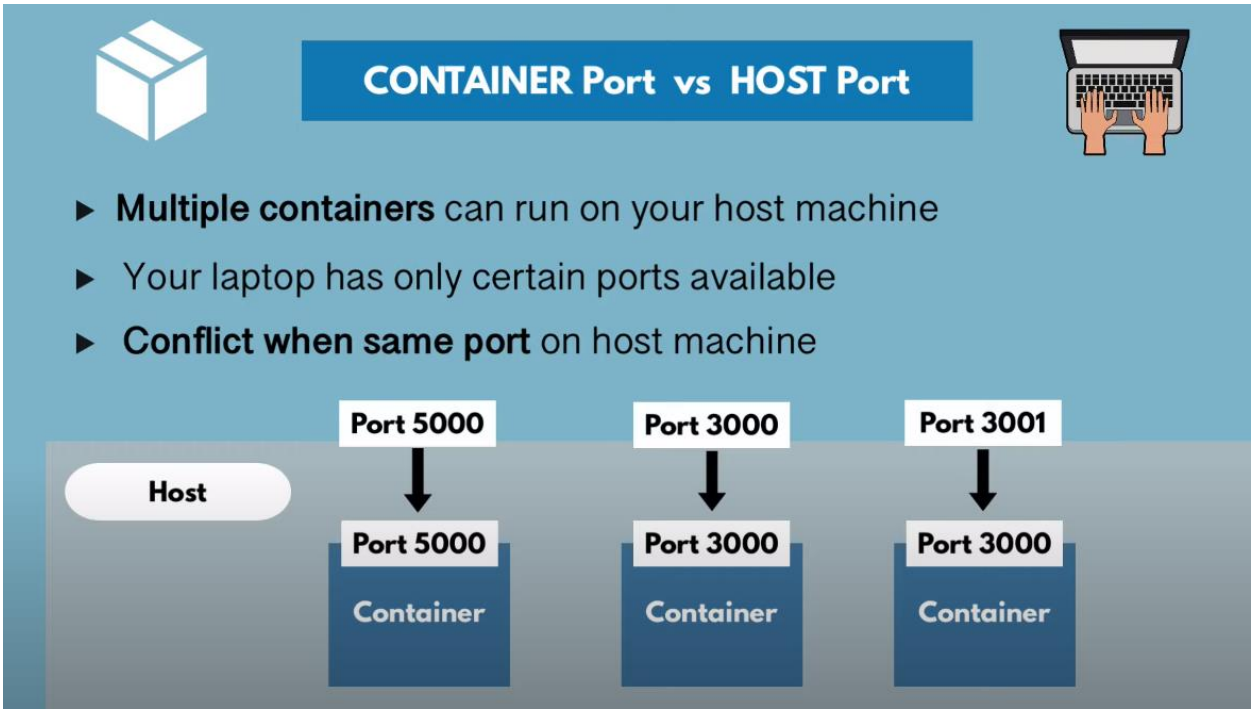
- Here , we have two versions of redis running. Now, how to use the container that we actually started.



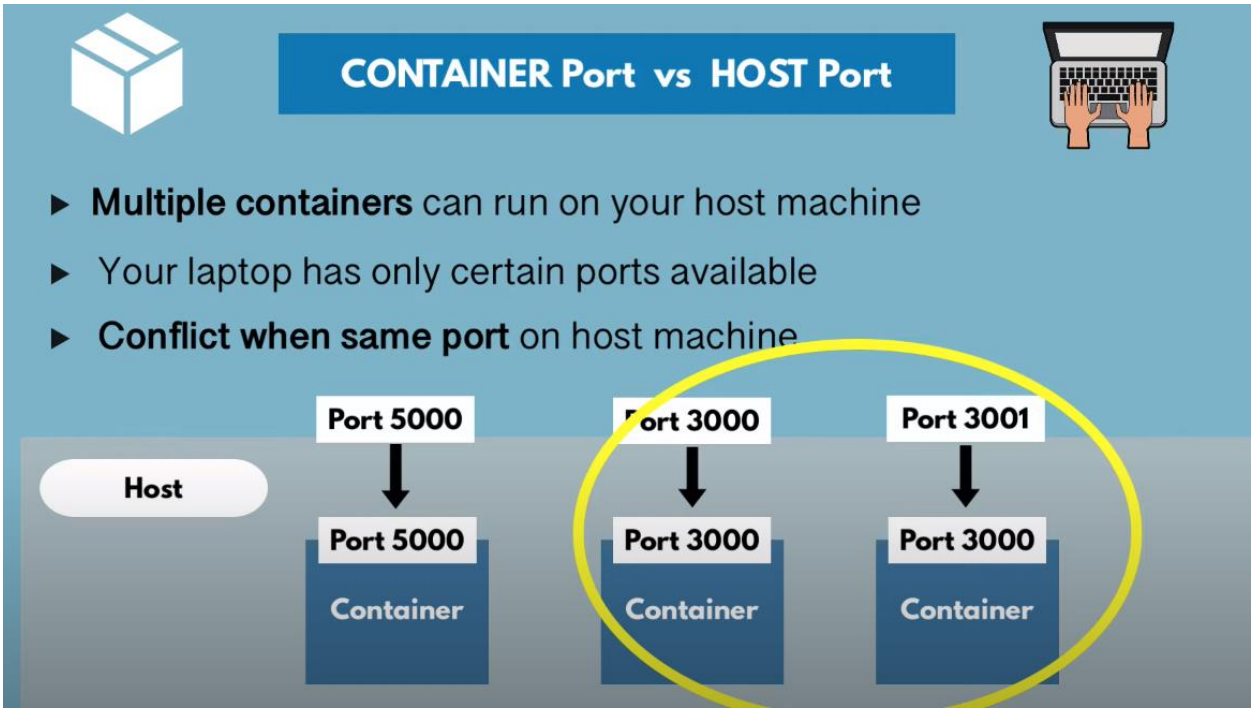
- Also, we can see in which ports the containers will be listening to the incoming requests.
- So, here both containers are using the same ports which was specified in the image.
- In the logs of the container we can see the information about the information mode and the port number which is 6379.



- How do we not have **conflicts while both are running on the same port?**
- We know that containers are like a virtual environment running on our host and we can have multiple containers running on the host. Here host is the laptop/desktop/pc.
- Our laptop will have certain ports which we can open for certain applications.
- We need to create a binding between the port of the laptop and the port of the container.
- We have conflict when we open two 5000 ports on the host.

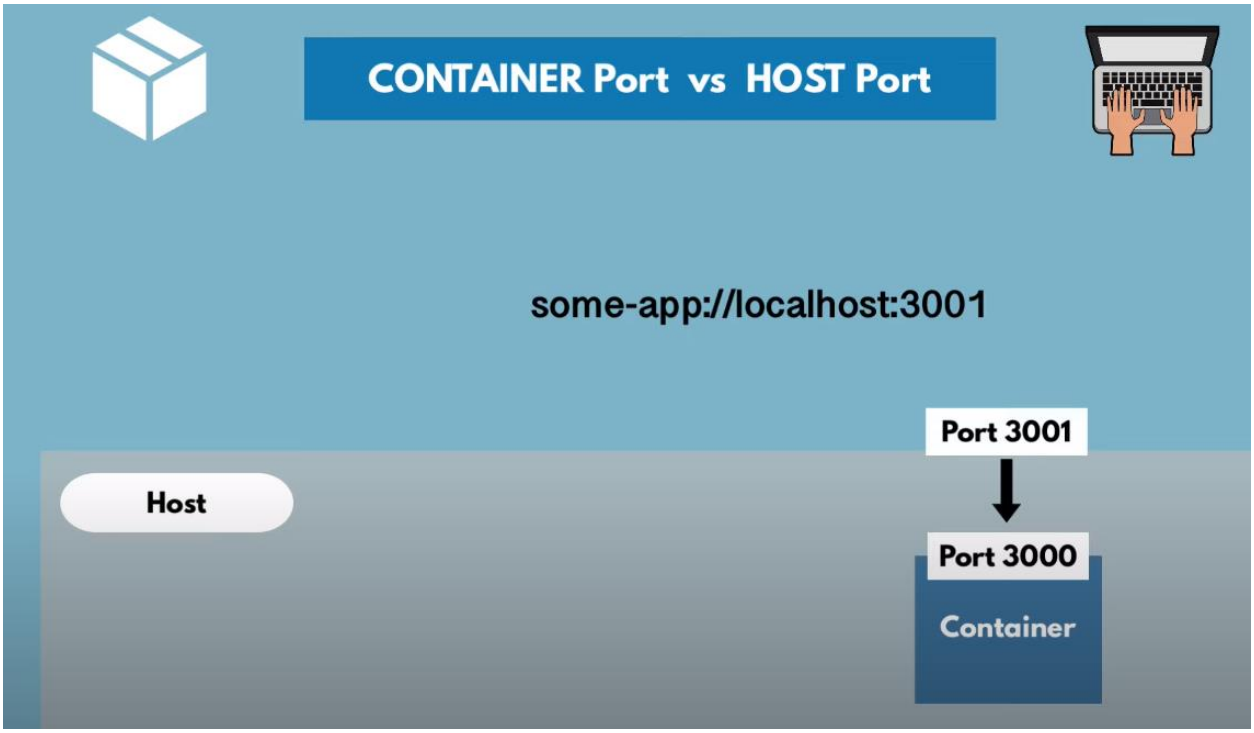


- From the above image, we can understand that the container is listening on port 5000 and we bind our laptops port 5000 to that container.
- We will have conflicts when we open **two 5000 ports** on the host because we get a message that the port is already used or bound and we cannot do that.
- However, we can have **two 3000 ports of containers as long as they are bound to two different ports of the host.**



- Once the **port binding** is done between the host and the container we can connect to the **running container** using the port of the host.

- Here we can tell that the port of the host **3001** can know how to forward the request to the container using port binding.

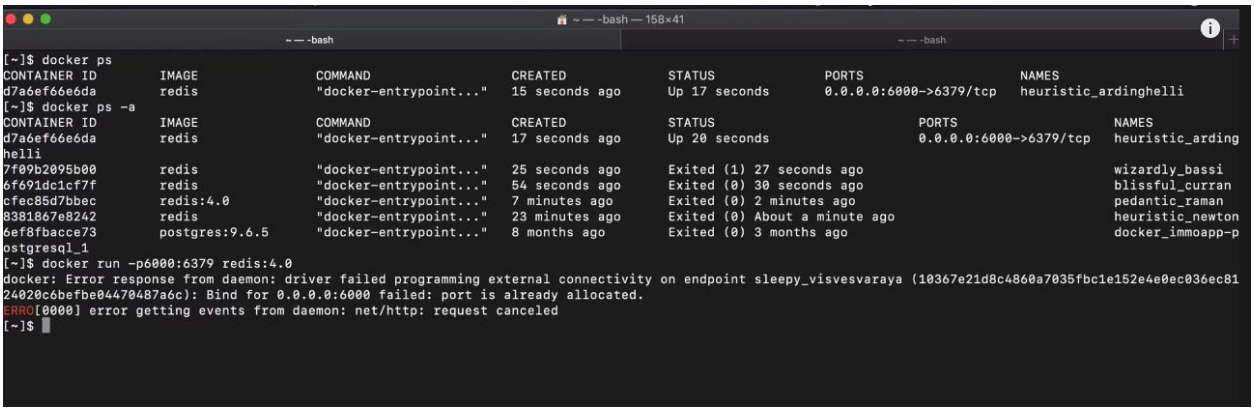


- While running the docker container, we need to specify the port to which it should be mapped or binded.
- The command for it is **docker run -p6000:6379 redis**

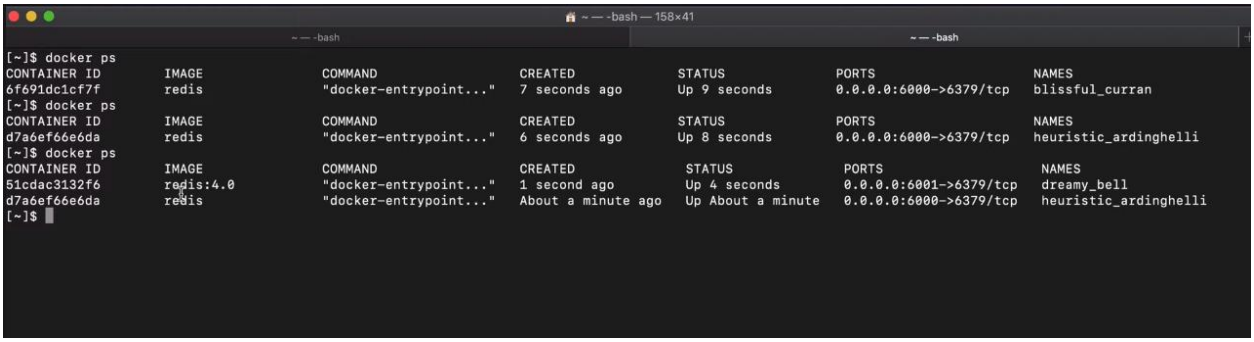
```
[~]$ docker run -p6000:6379
"docker run" requires at least 1 argument(s).
See 'docker run --help'.

Usage:  docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Run a command in a new container
[~]$ docker run -p6000:6379 redis
1:C 26 Oct 2019 12:12:50.142 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
1:C 26 Oct 2019 12:12:50.142 # Redis version=5.0.6, bits=64, commit=00000000, modified=0, pid=1, just started
1:C 26 Oct 2019 12:12:50.142 # Warning: no config file specified, using the default config. In order to specify a config file
conf
1:M 26 Oct 2019 12:12:50.143 * Running mode=standalone, port=6379.
1:M 26 Oct 2019 12:12:50.143 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn
8.
1:M 26 Oct 2019 12:12:50.143 # Server initialized
1:M 26 Oct 2019 12:12:50.143 # WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will create
with Redis. To fix this issue run the command 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it t
 retain the setting after a reboot. Redis must be restarted after THP is disabled.
```

- Now, when we execute the **docker ps** command we can see the following change.

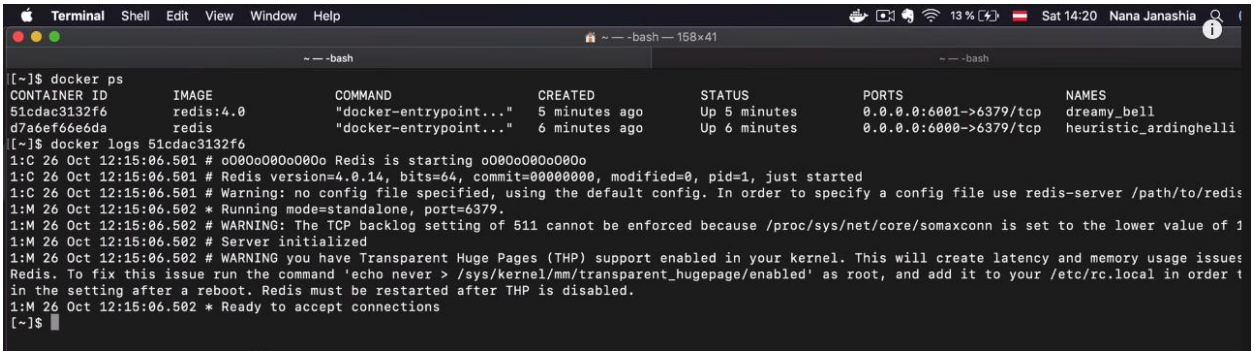
```
~ — -bash — 158×41
~ — docker run -p6000:6379 redis                    ~ — -bash
[~]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
6f691dc1cf7f        redis               "docker-entrypoint..."   7 seconds ago       Up 9 seconds        0.0.0.0:6000->6379/tcp   blissful_curran
[~]$
```



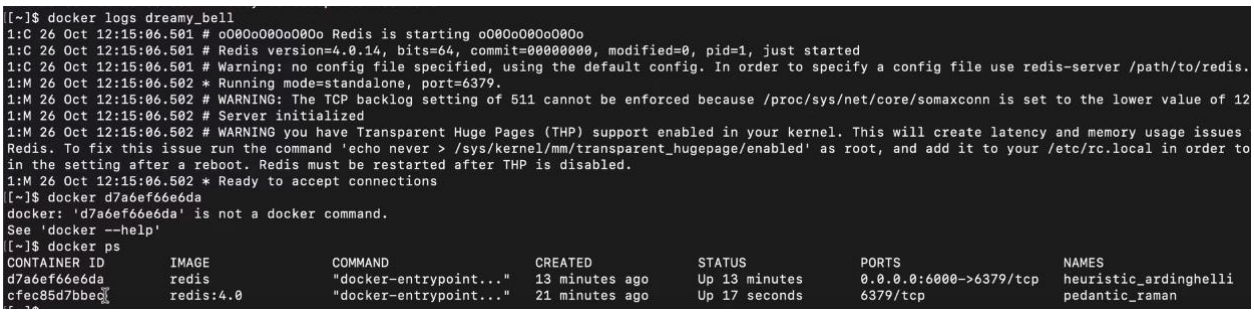
- Now, we get an error message when we try to bind the laptop port 6000 to the 6379 port of the redis:4.0 image because we already binded that port to the port 6379 of redis image container.
- Here, we are **running the containers in detached mode because we want to execute another commands and want to come out of the terminal.**
- We can now observe that we have binded the port **6001** of the host to the **port 6379** of the container. So now if we see **docker ps** then we can observe the port binding.



- So far we have seen docker commands which pulls the image from the repository to the local like **docker pull**
- **docker run** which combines both the docker pull and docker start.
- **docker start** to start the container
- **docker stop** to stop the container and also useful to restart the container if we made some changes and want to update it.
- We can use **docker run -d** which helps me to use the terminal again.
- **docker run -d -p** allows to bind the port to the container.
- **docker ps -a** gives a list of all the containers no matter whether they are running or not.
- **docker images** will list the images and with which we can delete the staled images and containers if they are not in use anymore.
- If there is something wrong in the container, we can view the logs inside the container or we want to get inside the terminal and interact with the container.
- Lets say something happens and our application cannot connect to redis.
- Ideally, we want to see what logs redis container is producing.
- The command is **docker logs id_of_the_container**



- If we don't to do **docker ps** to find out the **container_id** everytime, then we can simply give the name of the container.



- We are getting some random names here, so if we do not want random names and want to give them own names then
- The below command will create a new container with the name redis-older in detached mode with redis:4.0 image.

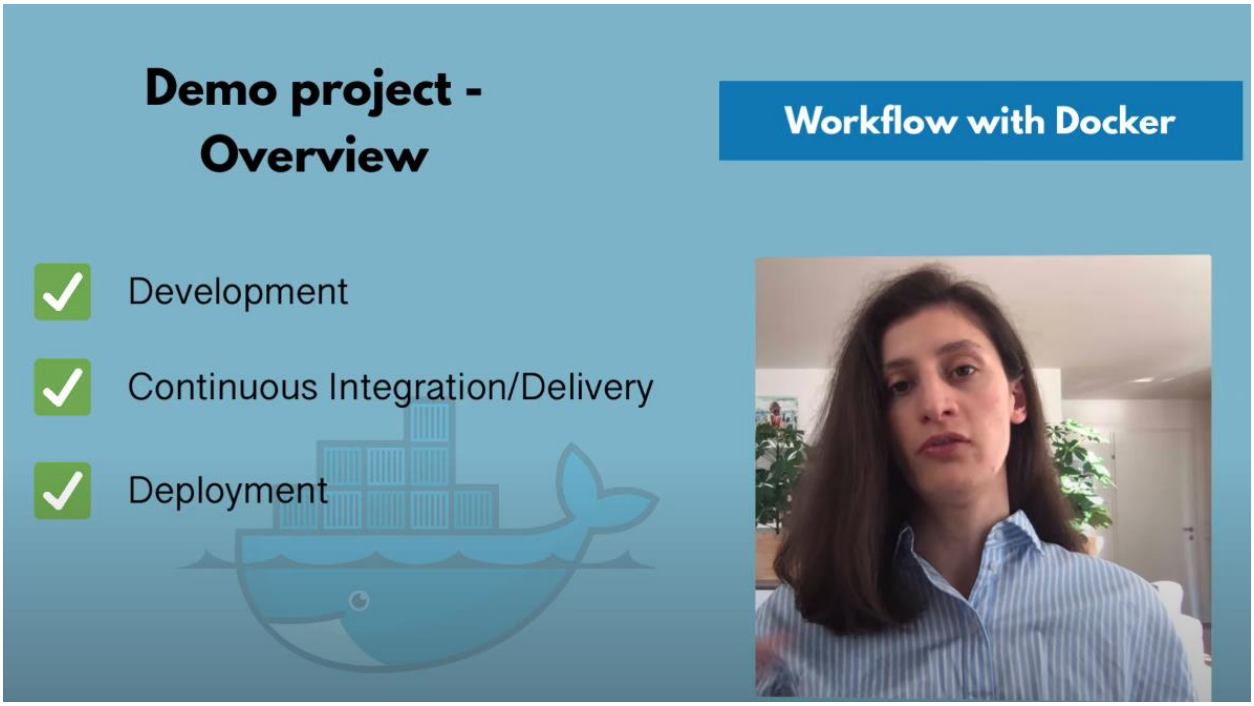
- Lets do this for another container too and here we need to first stop the container and then assign a new name to the newly created container.

- **docker stop d7a6ef66e6da**
- **docker run -d -p6000:6379 –name redis-latest redis**

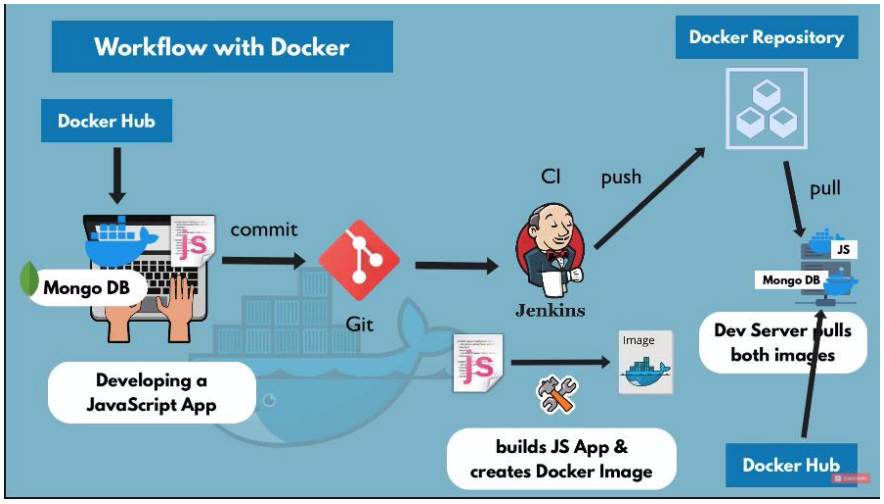
```
[~]$ docker run -d -p6000:6379 --name redis-latest redis
cae903a742020f6532ef9a9ddd49165a44dcddc9cf119d641e46285ab9356f17
[~]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED                  STATUS              PORTS                    NAMES
cae903a74202        redis               "docker-entrypoint..."   Less than a second ago   Up 2 seconds        0.0.0.0:6000->6379/tcp   redis-latest
5b7d84a59a7c        redis:4.0           "docker-entrypoint..."   48 seconds ago           Up 50 seconds       0.0.0.0:6001->6379/tcp   redis-older
[~]$
```

- Now, if the older version has some problems then we can use the command which is **docker logs redis-older**
- Another important command in debugging is **docker exec -it /bin/bash**
- Now we will be logged into the container as the root user and we can interact with the container easily.
- We can also **observe that there is a virtual file system present in the container.**
- Here , we can list the environment variables and also change the configuration if needed.
- Also, we can use the **exit command** to exit from the container.
- Since most of the container images are based on light weight linux distributions, we wouldn't have much of the linux commands or applications installed here.
- Difference between **docker run** and **docker start** is that with docker run we will take the image and start a container.
- With **docker start ,** we are not working with images rather with **containers.**
- Also, when we started the container after it is stopped, then the container will retain the attributes like name(--name), detached mode(-d) which we specified while creating the container using the command **docker run**
- So, **docker run** is to start the new container and **docker start** is to restart the stopped container.

- Once we learnt the basic concepts of docker and how it works, its important to see how actually docker works in practice.
- In software development process , we have this classical steps of development and continuous integration/development and eventually gets deployed on some environment it could be test environment, development environment and production environment.
- Its important to understand that how docker actually integrates in those steps.

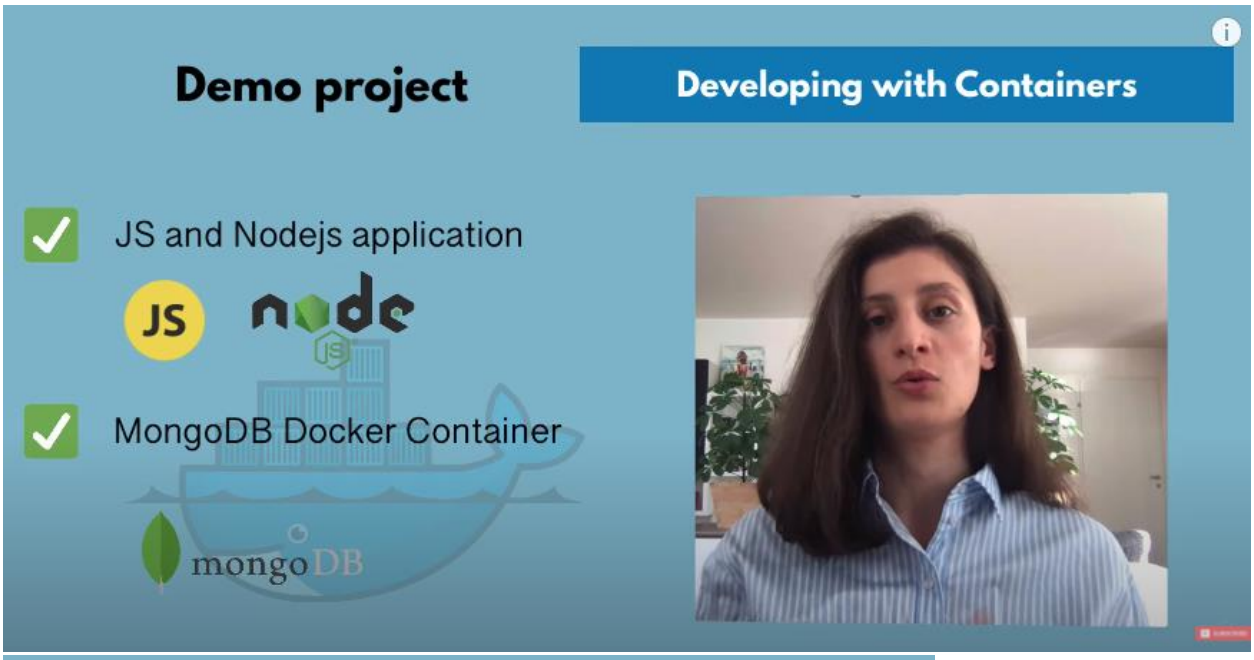


- Lets consider a simple scenario where we are developing a simple javascript application on our local development environment. Our javascript application uses the **mongo db database.**
- Instead of downloading it on our laptop, we will download a docker container from the docker hub.
- So, we now connect our application with the mongo db and we will start developing.
- Lets say that we developed the first version of the application locally and now we want to deploy it in the environment where our team or tester gonna test it.
- Now, we commit our javascript application in git or some other version control system that will trigger a continuous integration like Jenkins build or whatever we configured.
- **Jenkins built** will produce artifacts which is the docker image from our application.
- **So,** first we build the javascript application and then create a docker image out of that artifact.
- **After the docker image is created by the Jenkins built, it gets pushed to the private repository.**
- **In a company, we have private repositories because we do not want other people to have access to our images.**
- The next step could be configured on Jenkins or other tools or scripts that enables docker image to be deployed on a development server.
- So , the development server will pull the image from the private repository which is the javascript application image and also the image of the mongo db image from the dockerhub because it is dependent on it.
- Now, we have two containers .where one is a private container and the other is the publicly available mongodb container on the dev server and we have to configure ofcourse such that they talk and communicate to each other and run as a app.
- Now, if the tester or any other developer logs into the server then they will be able to test the application.

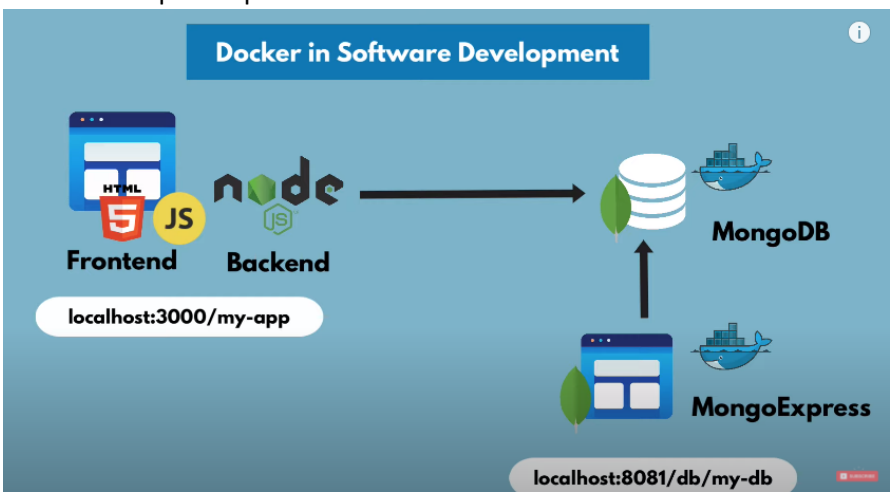


- Now, we are going to look at examples where we can use docker in local development process.

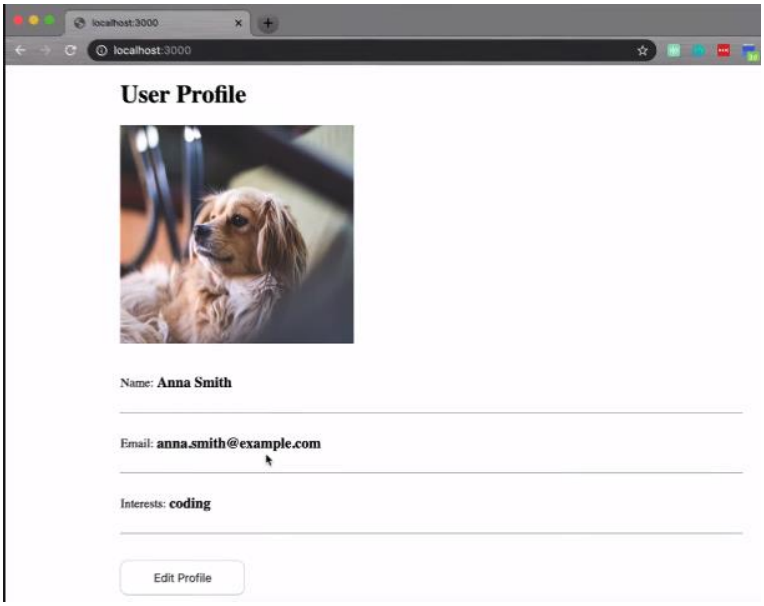
- We will take javascript and node js application to see the local development process and we will then connect it to docker container with a mongodb database in it.



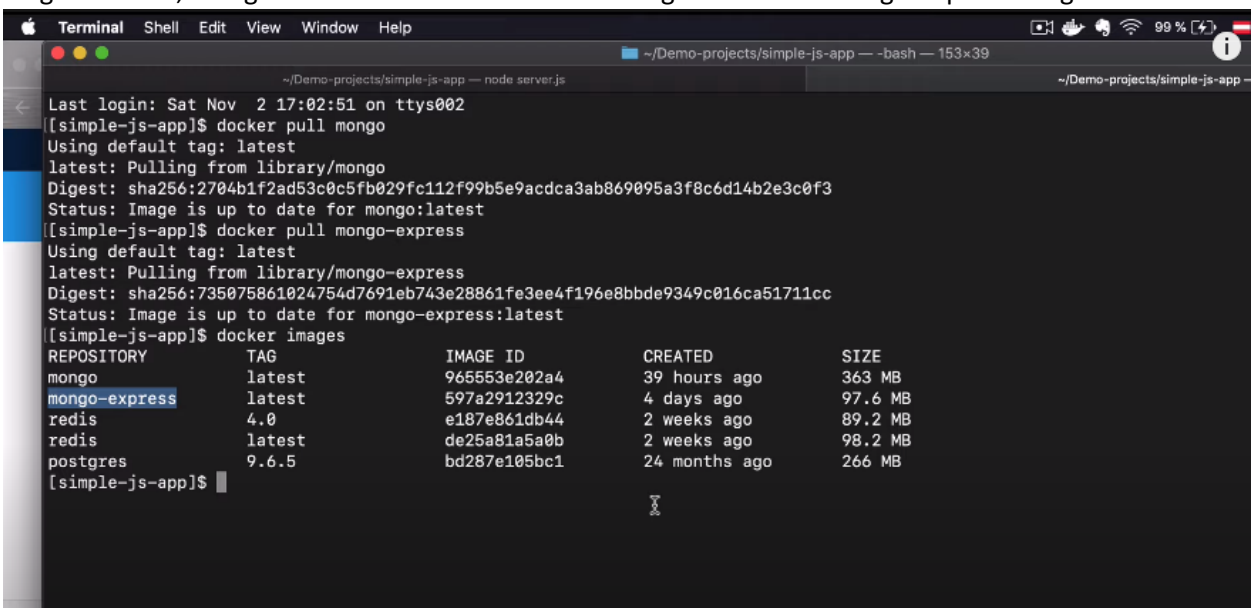
- Here, we gonna see how to use docker containers while developing applications.
- Here, for demo, we will develop a **front end ui** with html, js and backend with **nodejs** and we will connect to **mongodb database.**
- Also, to make working with mongodb much easier , I mean rather than executing the commands in the terminal we gonna deploy a docker container of the mongo ui which is **MongoExpress.** Here, we can see all the updates that our application is making in the database.
- The below development set up must give us an idea of how docker containers are actually used in the development process.



- Now actually, we have one application made of html and javascript along with node js in the backend which means there is a server on which our application is running.

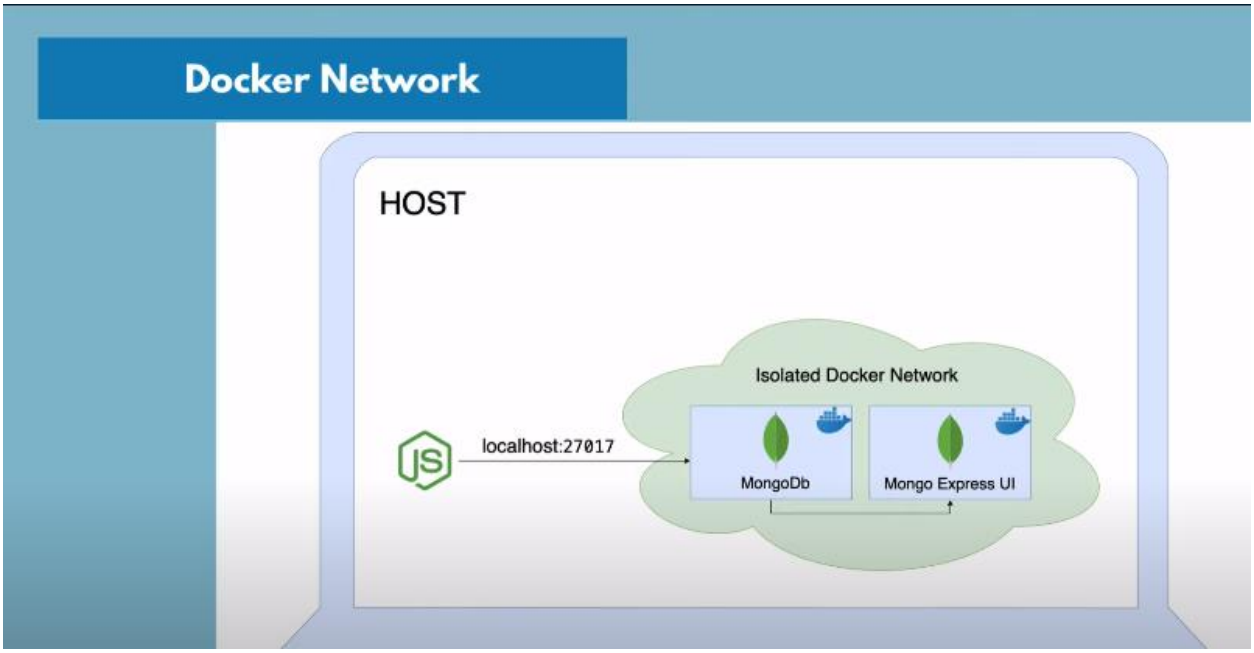


- When we enter the information and refresh the page then the information will be gone and also it wont be updated. So, for this reason we will connect to the mongo db database docker container and also deploy a mongodb ul which is mongo express to view the database insights.
- To get started , first go to **docker hub** and find the mongodb and the mongo-express image.

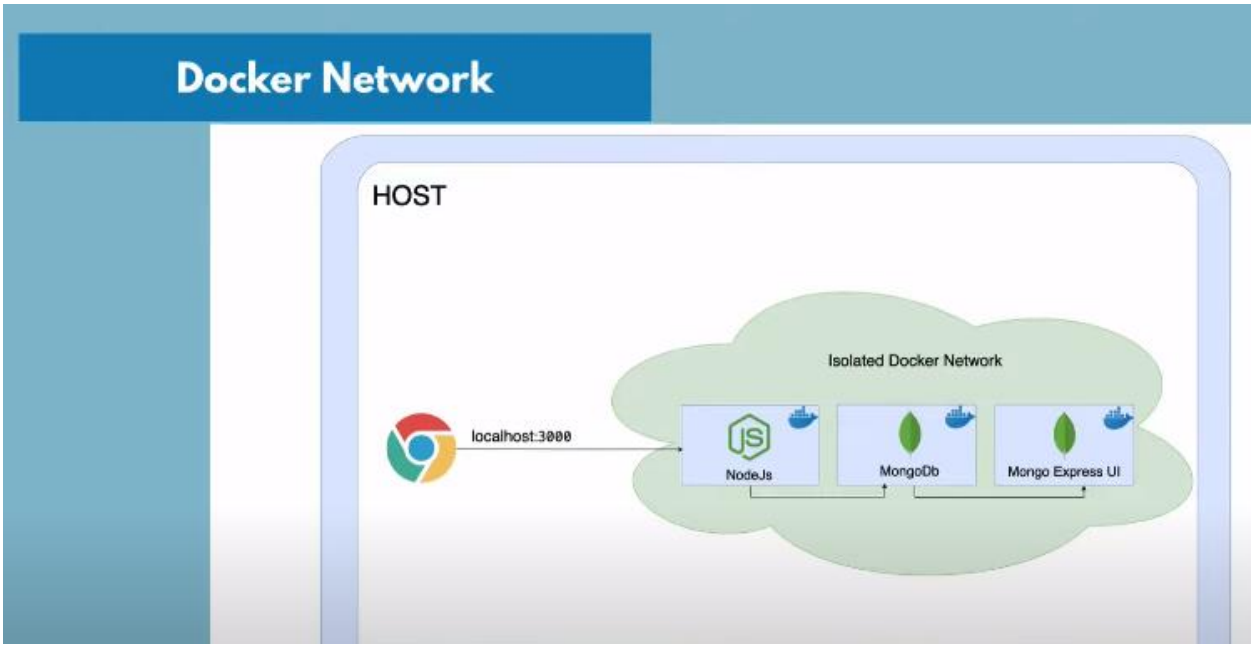


- So now we pulled the mongodb and mongo express images using **docker pull** command.
- Next step is that we should make them to run in order to make mongodb available for our application and mongo express to connect to the mongodb database docker container to view the database insights.
- Lets do the connection between the mongodb and mongo express first.

- In order to do that, like connecting the **mongodb docker container** and the **mongo express ui** we need to understand about the docker network.
- How this works is that **docker creates its isolated docker network where the containers are running in.**
- So, when we deploy two containers like **MongoDB and Mongo Express UI** in the same network, they **can talk to each other using just the container name with out local host or port numbers.**
- This is because the two containers are in the same network.
- The applications which runs outside of docker like **node js** which is gonna run from the node server gonna **connect to the docker network** using the **local host and the port number.**



- **Later when we package our application into its own docker image, we will have mongodb and mongo express container and we will have node.js application which we wrote using index.html and javascript for frontend and its own container and connects to the mongodb container.**
- The browser or the host running outside the docker network is gonna connect to javascript application again using hostname and the port number.



- Docker by default provide some networks. If we say **docker network ls,** we can see autogenerated docker networks by default which can be shown below.

```
[~]$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
ef884144b3e9        bridge              bridge              local
484f006515ef        docker_default      bridge              local
fa524e351841        host                host                local
96afbbc49358        none                null                local
[~]$
```

- Lets now create own docker network for mongodb and mongo express.
- The command is **docker network create mongo-network**

```
[~]$ docker network create mongo-network
70eb8e9f1c1eac66312912d30422e4acf3ade947736b0ca833734b030798a712
[~]$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
ef884144b3e9        bridge              bridge              local
484f006515ef        docker_default      bridge              local
fa524e351841        host                host                local
70eb8e9f1c1e        mongo-network       bridge              local
96afbbc49358        none                null                local
[~]$
```

- In order to make the **mongodb** and the **mongo express** run in the docker network, we need to provide the **network option** in the **docker run command.**

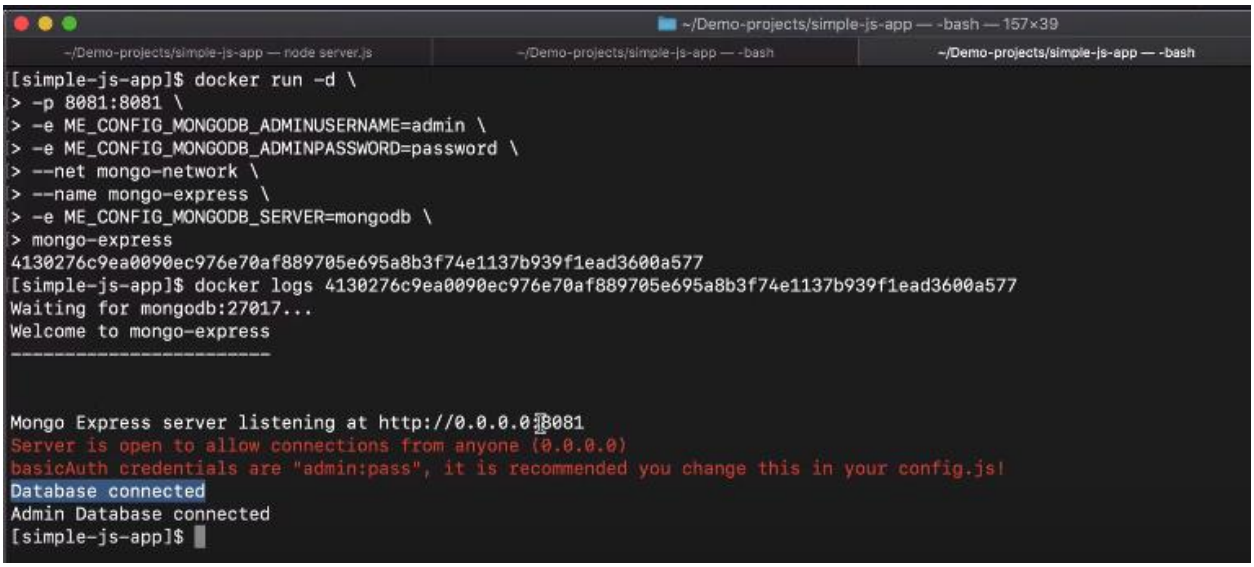
- We all know that **docker run** is the command which helps to start the container from the image.
- **-p 27017:27017** means that we are binding the port of local host **27017** with the port of the container which is **27017** and it is where it is running.
- On the start of mongodb container we can define some environment variables like **MONGO_INITDB_ROOT_USERNAME** and **MONGO_INITDB_ROOT_PASSWORD** which we need to connect express to mongo.
- We are going to provide username and password and we can create the database from the mongo UI later.
- We definitely need container name to connect to the mongo express.
- Also, we can specify the network name.
- Now the command below shows everything we explained.

```
[simple-js-app]$ docker run -d \
> -p 27017:27017 \
> -e MONGO_INITDB_ROOT_USERNAME=admin \
> -e MONGO_INITDB_ROOT_PASSWORD=password \
> --name mongodb \
> --net mongo-network \
> mongo
```

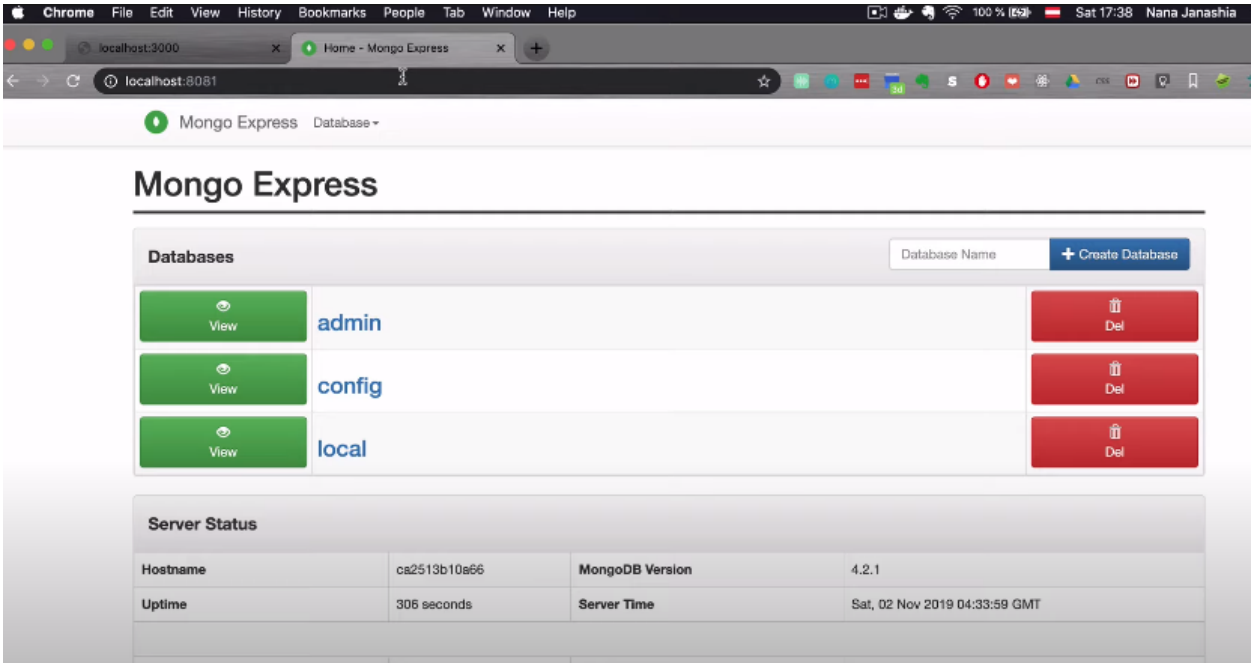
- Here we override the mongo db database image name from **mongo to mongodb** using **–name**
- To see whether everything is successful or not, use the command **docker logs container_id**

```
[simple-js-app]$ docker run -d \
> -p 27017:27017 \
> -e MONGO_INITDB_ROOT_USERNAME=admin \
> -e MONGO_INITDB_ROOT_PASSWORD=password \
> --name mongodb \
> --net mongo-network \
> mongo
ca2513b10a66e620870eecac9c31c74857efe032877f544deecec04fa602985b
[simple-js-app]$ docker logs ca2513b10a66e620870eecac9c31c74857efe032877f544deecec04fa602985b
```

- We want mongo express to connect to mongo db container on start up.
- Express will need some kind of username and password to connect to mongo db container
- Express uses the container name to connect to the mongo db container because they are running in the same network.
- Lets create the **docker run** command for express as well.
- Environmental variables need to be specified with **-e**

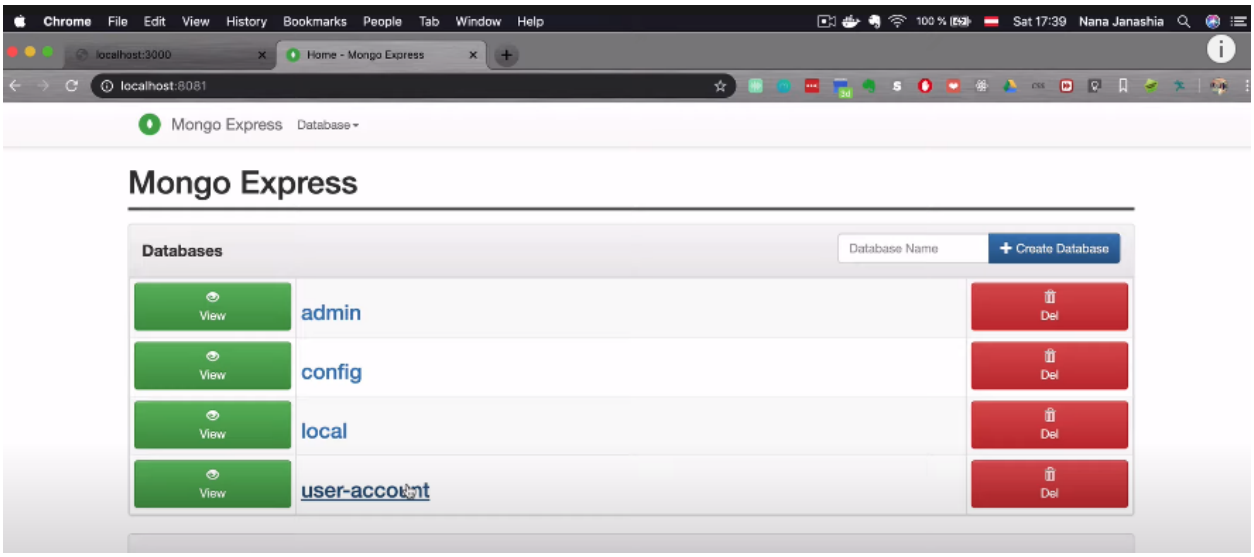


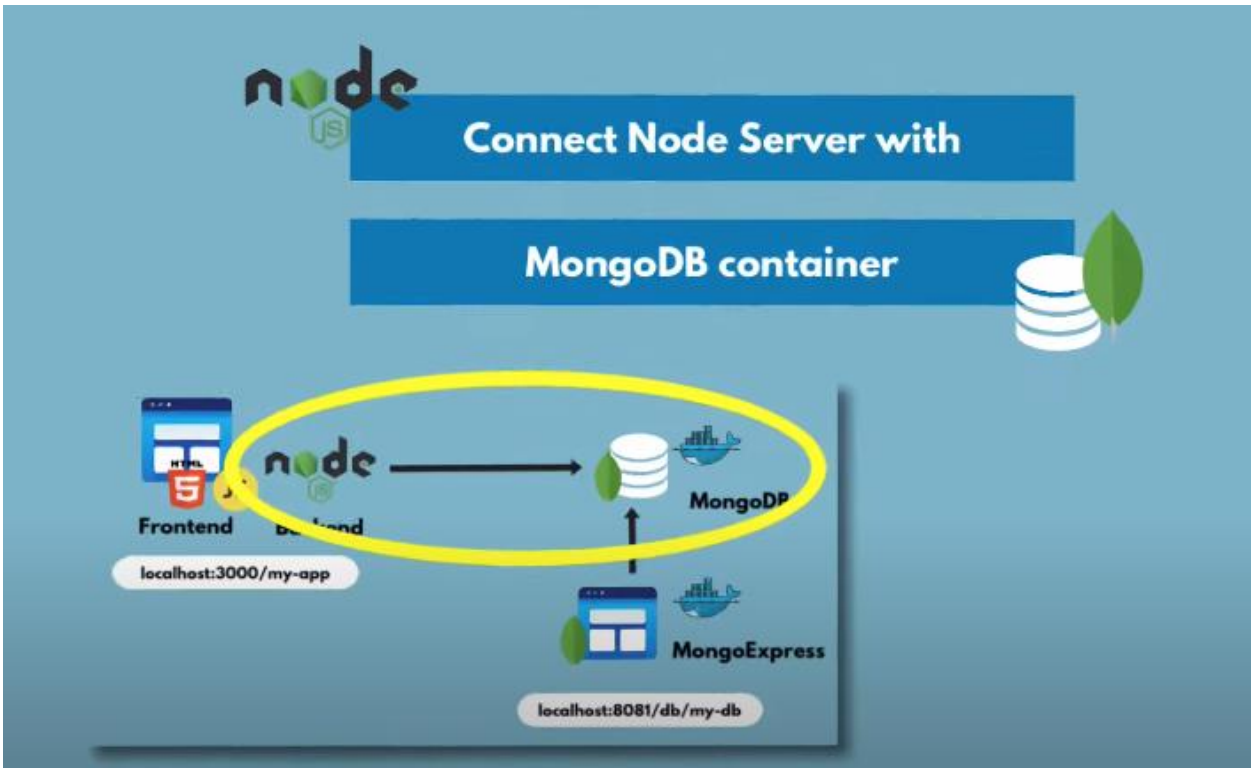
- Actually, here we specified **8081** as the port of the host and the port of the **mongo express container.**
- We need to specify the same network name as that of **mongo db docker container.**
- In the **ME_CONFIG_MONGODB_SERVER** option we specified the name of the mongo db database container which is **mongodb**
- Also, we specified the image as **mongo-express**
- we can see that **Database connected** and mongo express is running on port 8081.
- Lets check at **port 8081** from the local host using the url **localhost:8081**



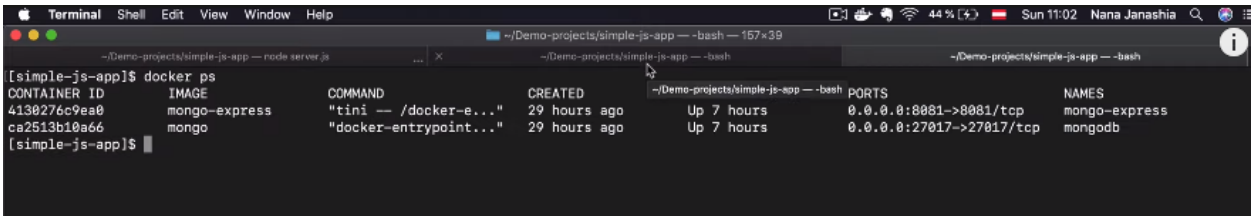
- We can observe that the mongo express is connected.
- In the above image, we are seeing some databases and these are the ones which are created by default in the mongo express.
- Using this **User Interface, we can create our own database.**

- Here , lets create one database namely **user-account** using the mongo-express and we can connect node js application to this database.

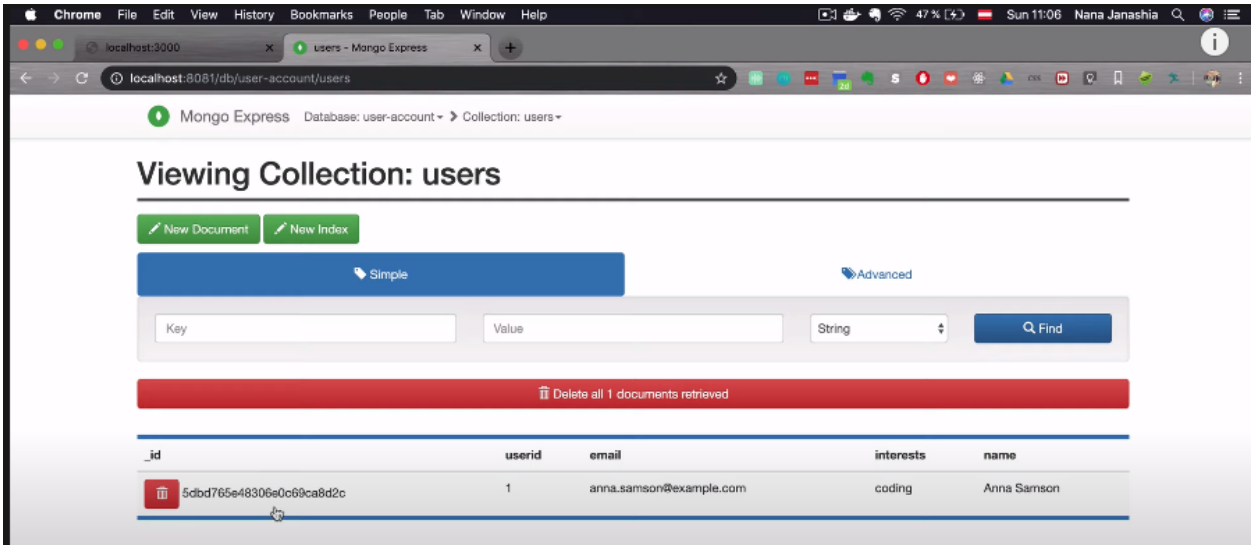




- Now, we have **mongodb container and mongo-express container running and lets check it using the command docker ps**

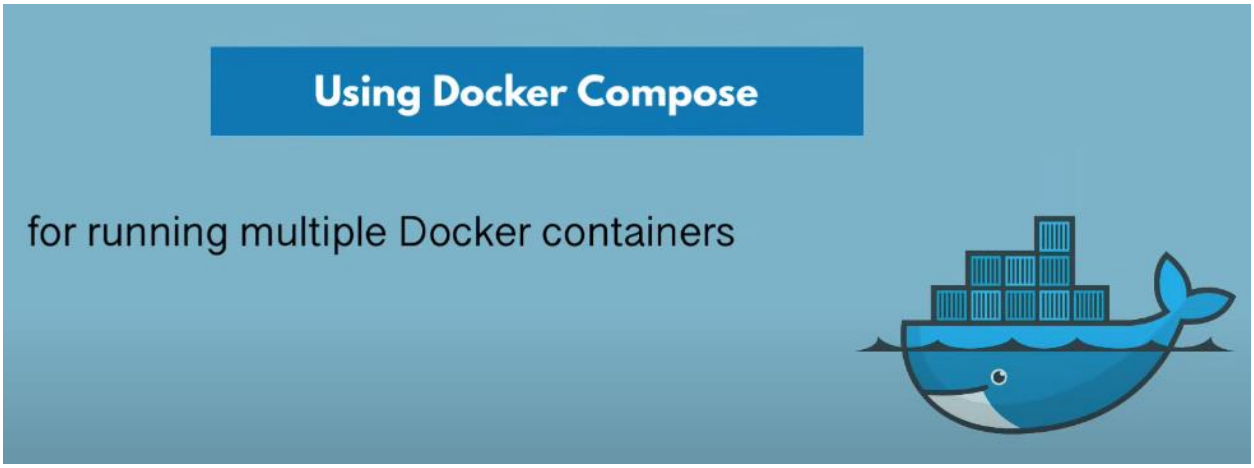


- Now, we need to connect node js to the database.
- The way to do that is to give the protocol of the database and the uri of the database.
- The uri of our database is the local host and the port at which it is accessible at.

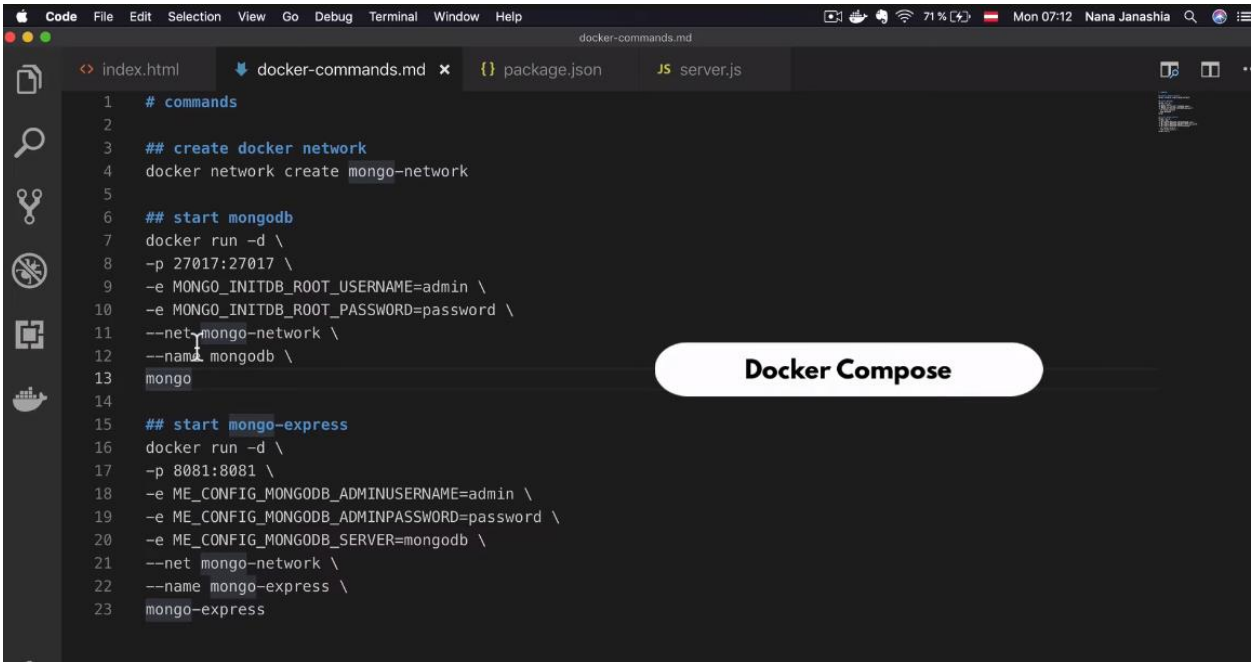


- Users is a collection in the **user-account database.**
- If we make any change in the front end UI then it will get relected in the users collection because in the node js code, we have written such type of logic.
- If we want to see the last part of the activity of the **mongodb container,** we can do it by using **docker logs container_id | tail**

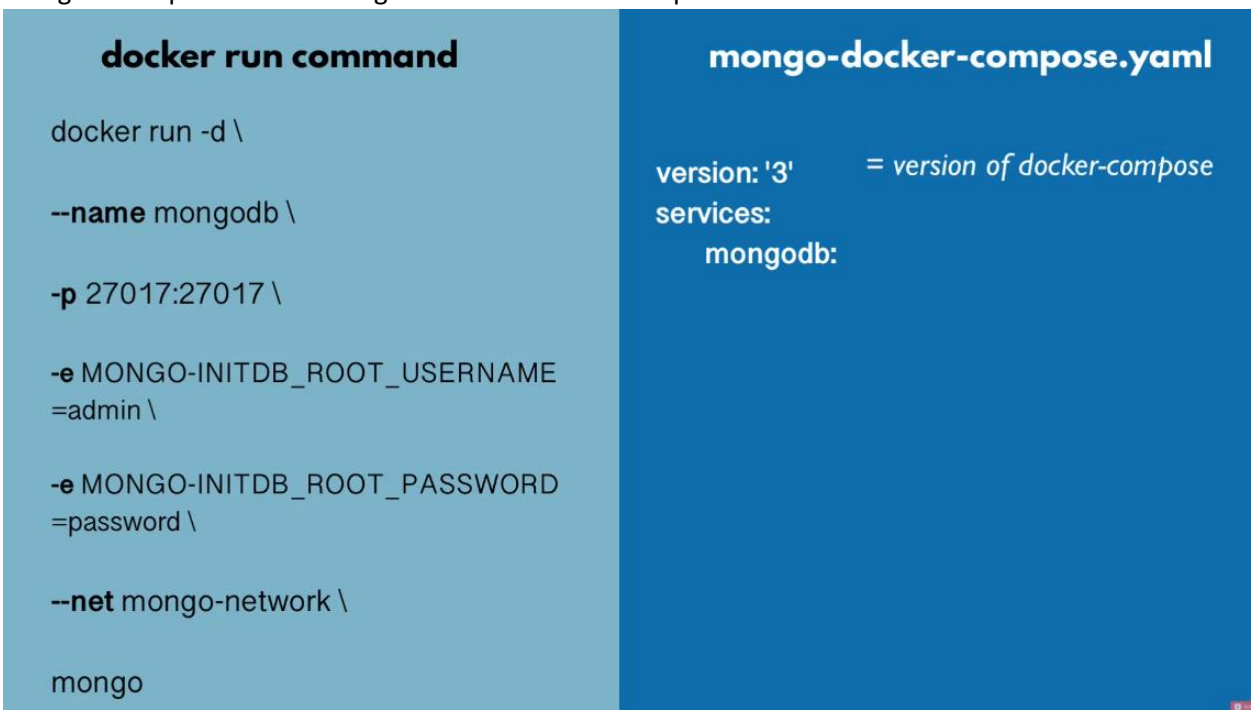
- We can also stream the logs using the command **docker logs container_id -f**
- The above command is especially useful when we want to see the immediate logs when we made an update in the frontend. This will avoid us to type **docker logs** command multiple times.



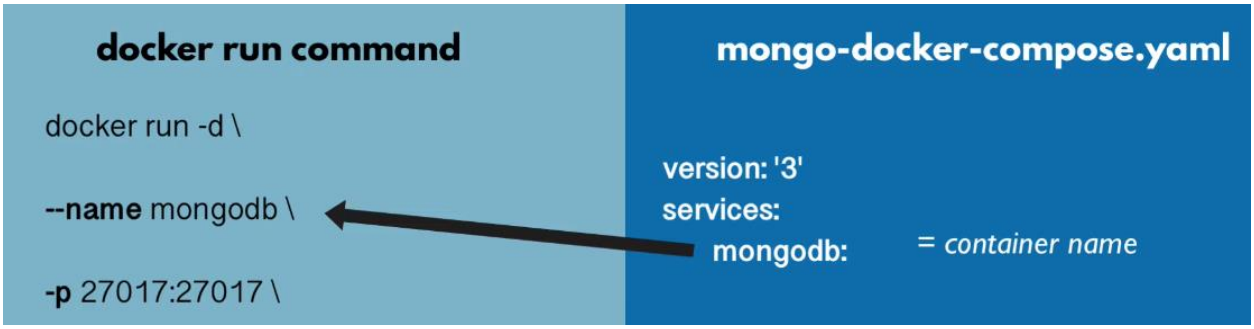
- Actually, we started and created two docker containers which are **mongodb** and **mongo express.**
- Firstly we created a network where the containers can talk to each other using the container name.
- Host port is necessary for that.
- Then we actually ran two docker run commands with all the options and the environmental variables.
- This way of starting the container is always tedious because we do not want to run these commands everytime in the terminal especially when we have a bunch of docker containers, we want to automate it or make it easier.
- There is a tool which makes running multiple docker containers much easier than with docker run commands and that is **Docker Compose.**



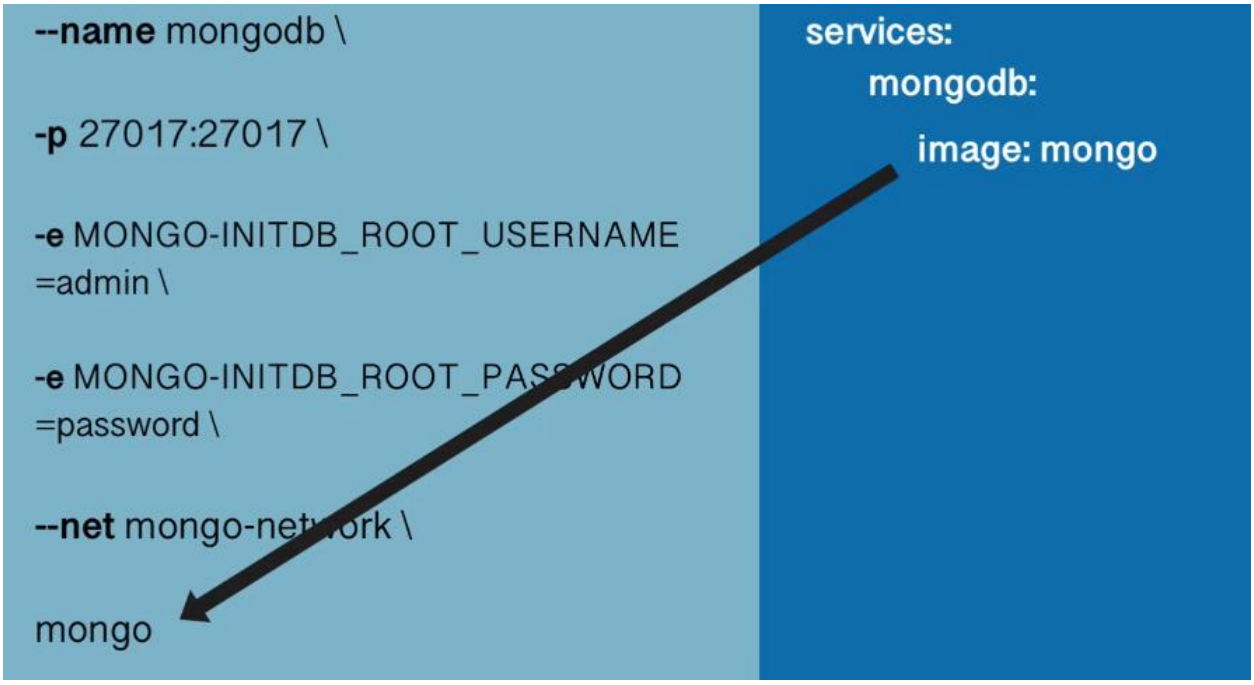
- With **docker compose,** its much easier because if we have 10 docker containers and if they need to communicate with each other then we can write the run commands in a structured way in the docker compose file.
- Lets get a deeper understanding of how the docker-compose structure looks like.



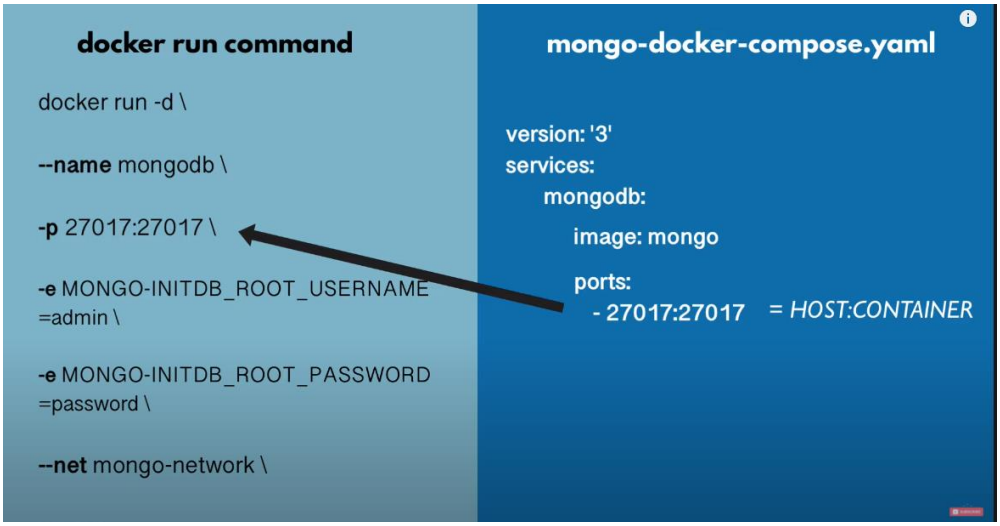
- Also in the services, the first name which is **mongodb** will refer to the name of the container which we had specified.



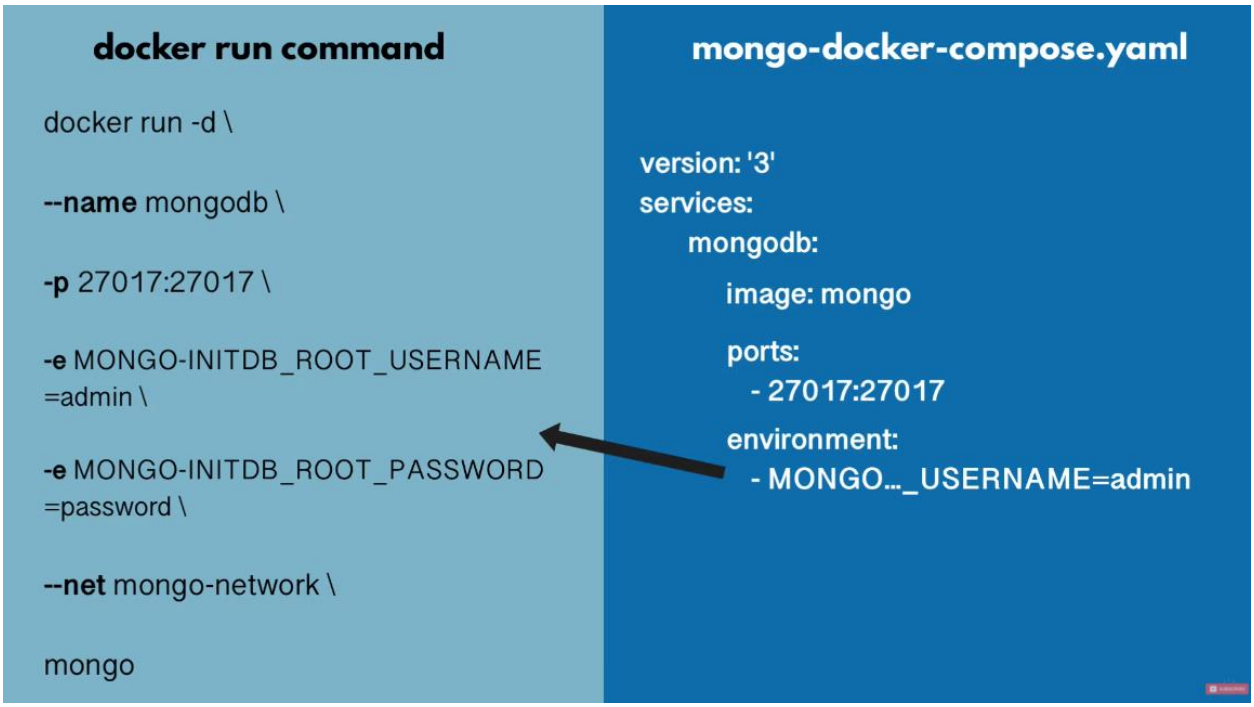
- The next one is the image and ofcourse we need to know from which image the container will be built from.



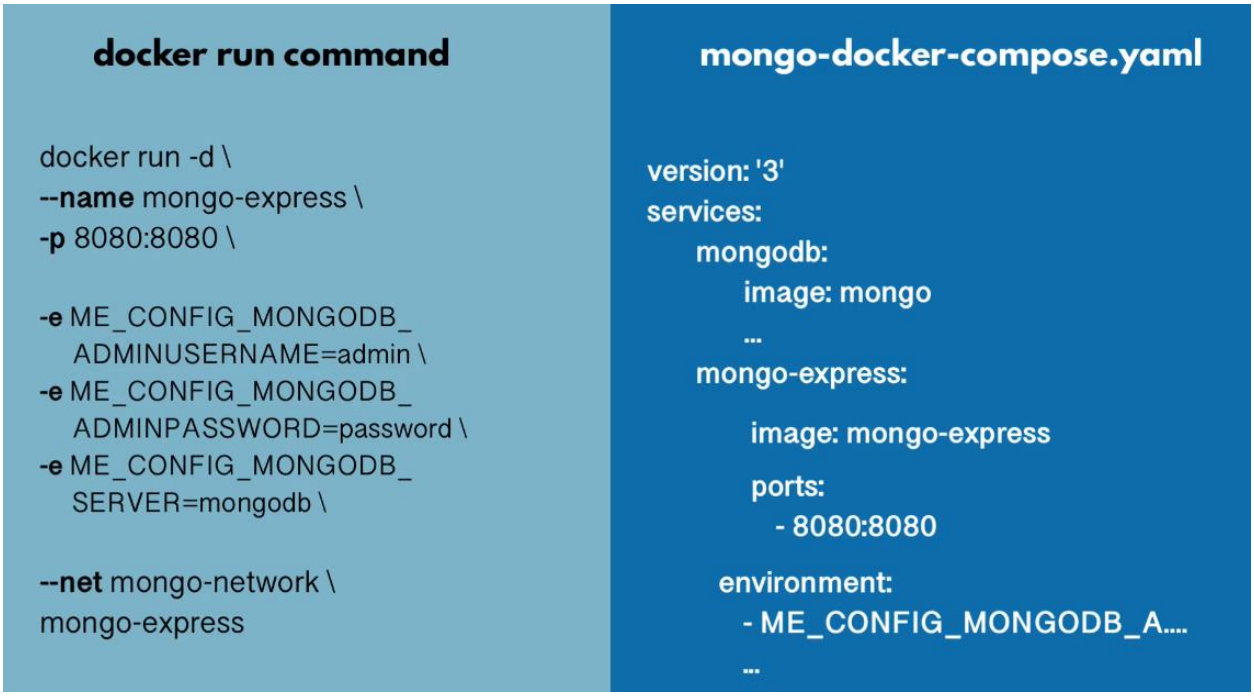
- The next one is the ports , we can specify which ports to open, the first port is the host one and the second one is the container one.



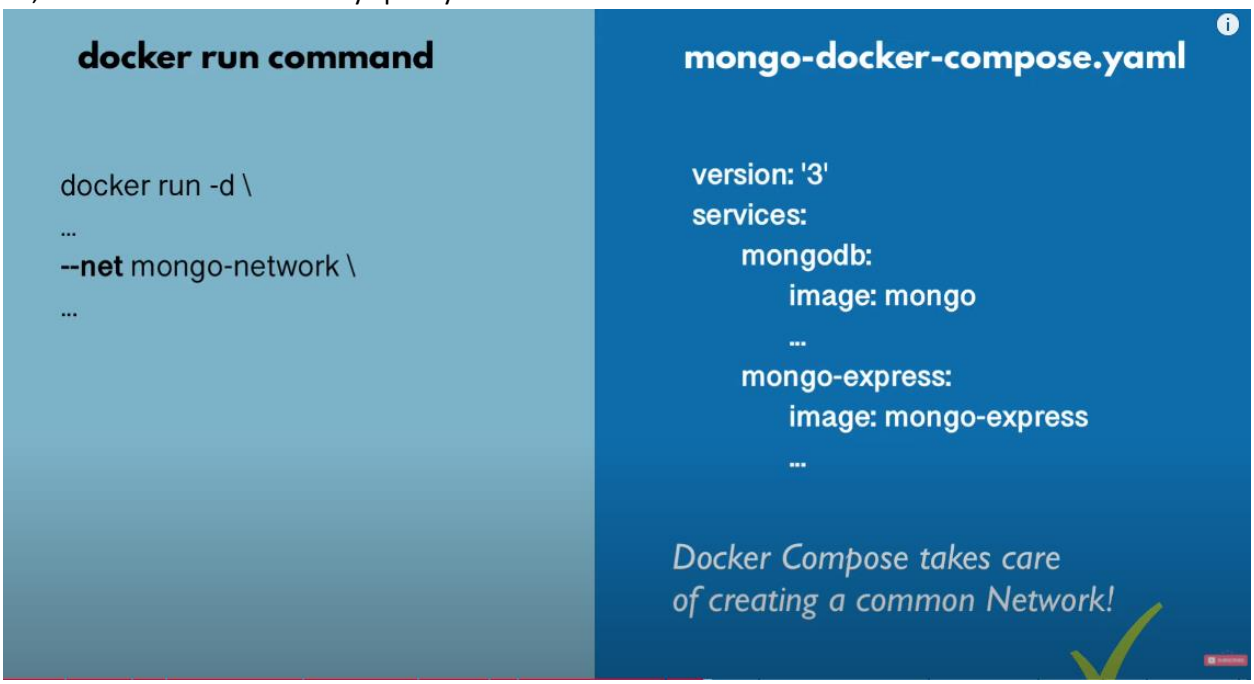
- Ofcourse the environment variables can also be mapped into **docker-compose.**
- This is how specific structure of **docker compose,** for one docker container for example here its mongodb database looks like.

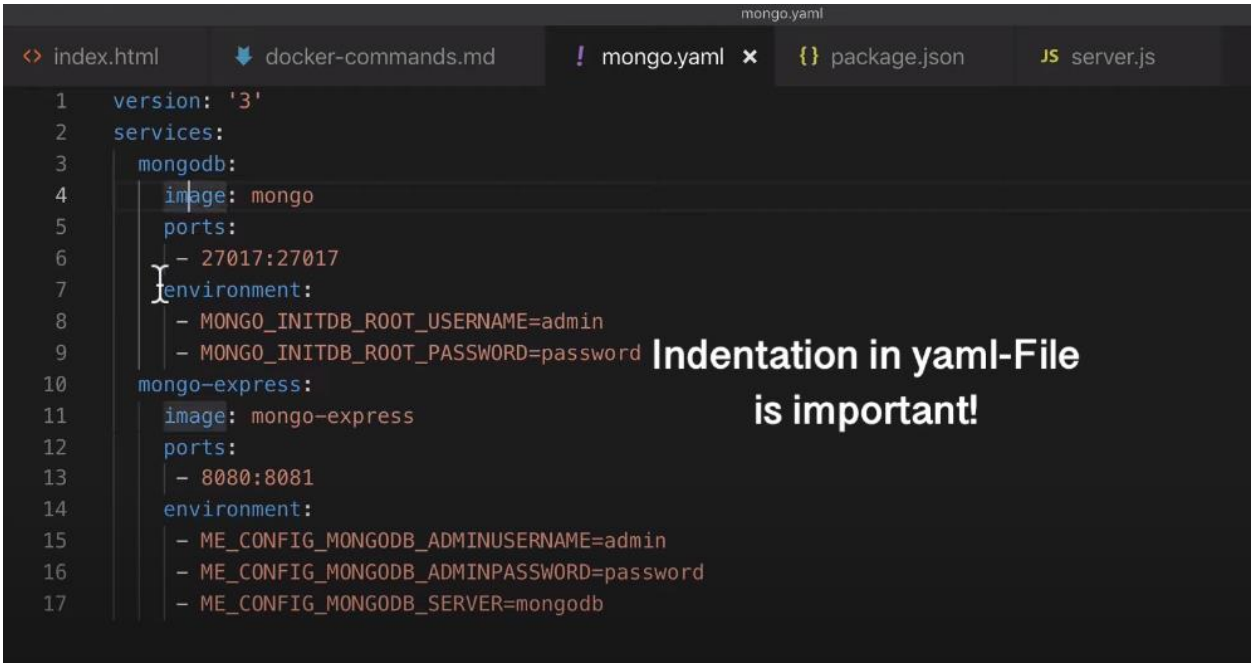


- Now again lets see the **docker-compose file structure for the mongo express.**
- We have **docker run command** for mongo express and lets see how we can map it to docker compose.
- **Services option** will list the containers that we want to create and again in term of names, mongo express will map to the container name.
- **The next one is the image which maps to mongo-express**

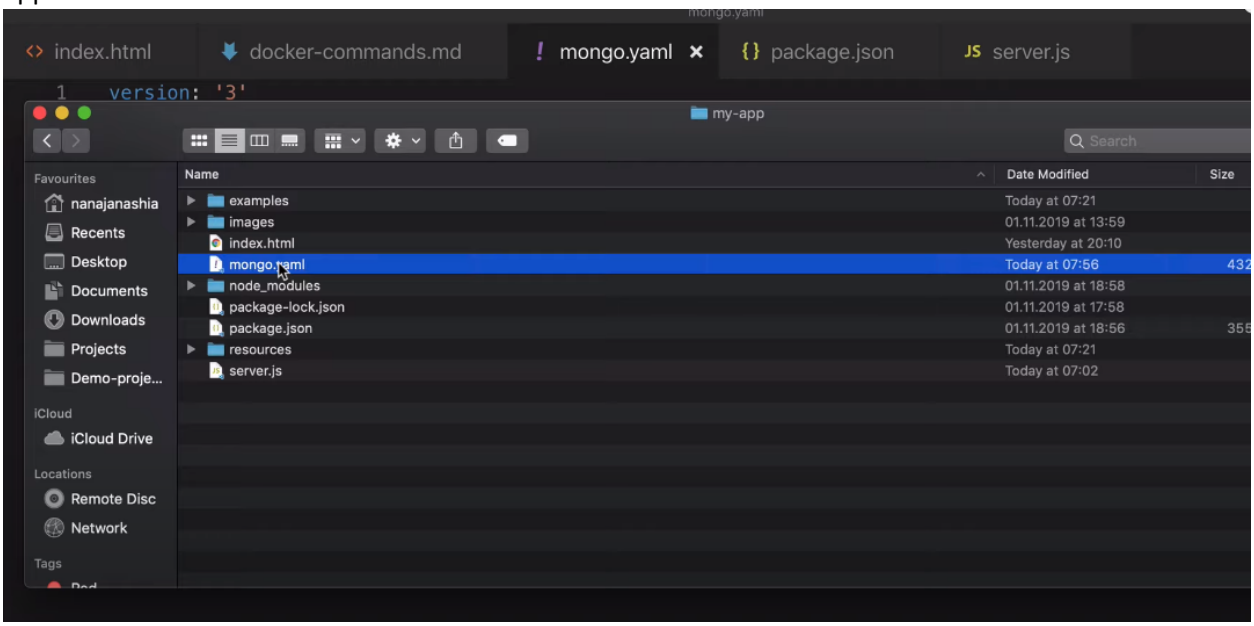


- This is how **docker-compose** will look like and basically **docker-compose is just a structured way to contain very normal common docker commands.**
- It would be **easier** for us to change the variables, edit the ports and add some new options to the **run command.** we can easily notice that network configuration is not there in docker compose so this mongo network which we created, we don't have to do it in a docker compose.
- We have a same concept here ,like containers will talk to each other using the container name. so docker compose will take care of creating a common network for these containers.
- So, we don't need to manually specify in which network these containers will run in.

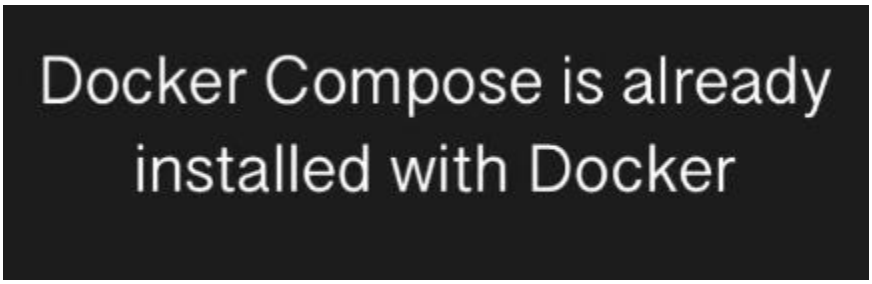




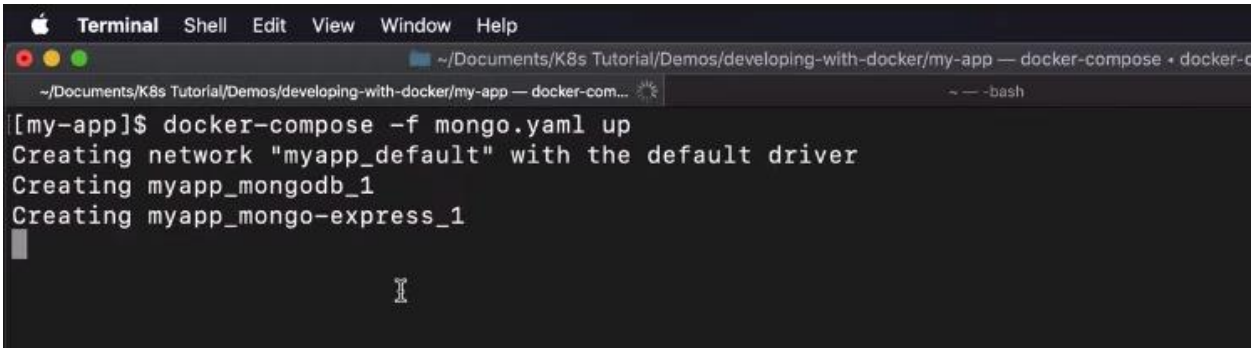
- Indentation in yaml file is important.
- We can see that there are two containers which are **mongodb** and **mongo-express** and each of them contain their own configuration.
- For demonstration purpose, we saved the docker-compose file in the code and it is part of the application code.



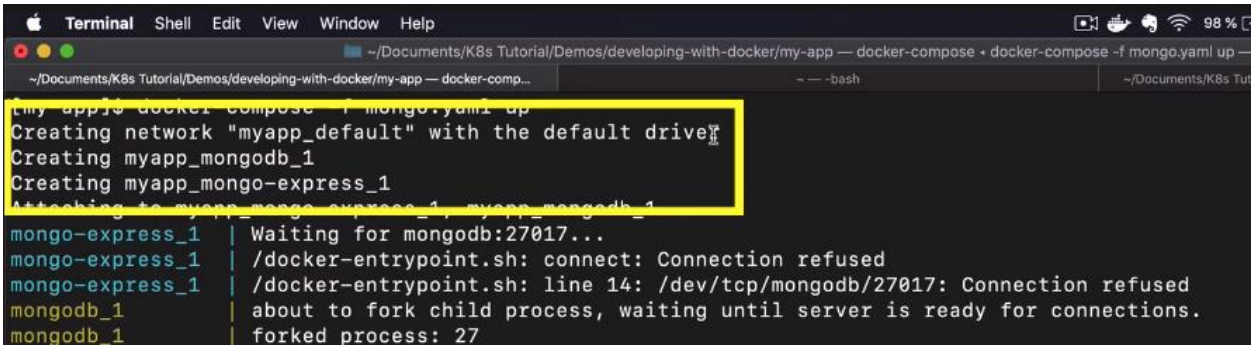
- Now, we have **docker-compose file** and the question is how do we start the containers using them.
- Now, lets go to the command line and start docker containers using the **docker-compose file.**



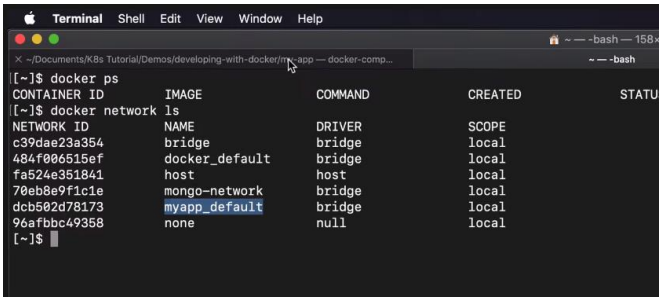
- Now the **docker-compose command** takes the argument as a file and specify the file name which we have saved as **mongo.yaml** and then specify the option **up** which will start the containers mentioned in the **yaml file.**



- Now the two containers starts running.
- Here **docker-compose** will take care of the creation of the network in which the containers should run.



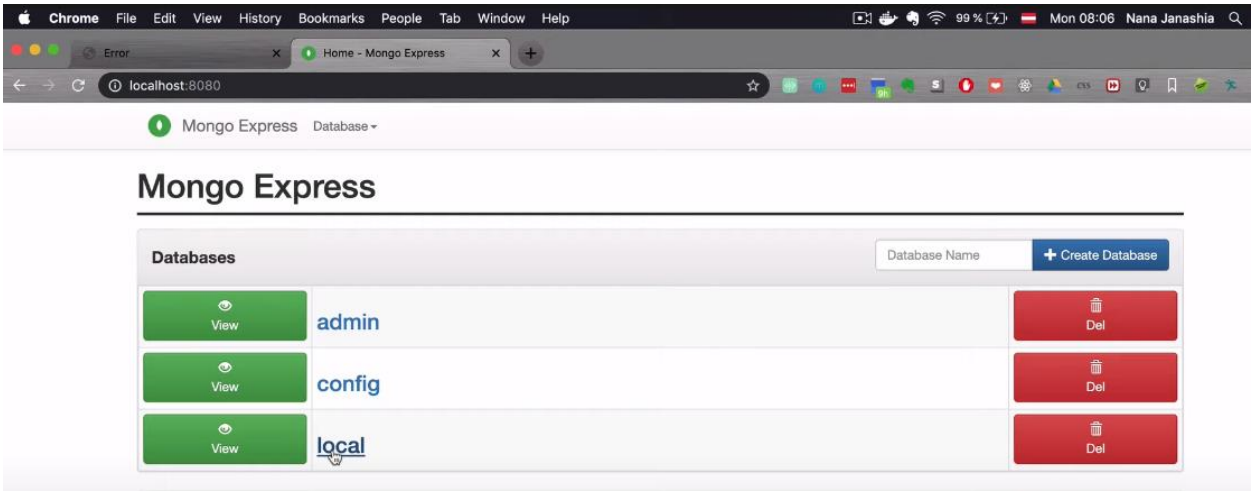
- Now when we execute the command **docker network ls ,** we can see **myapp_default** here.



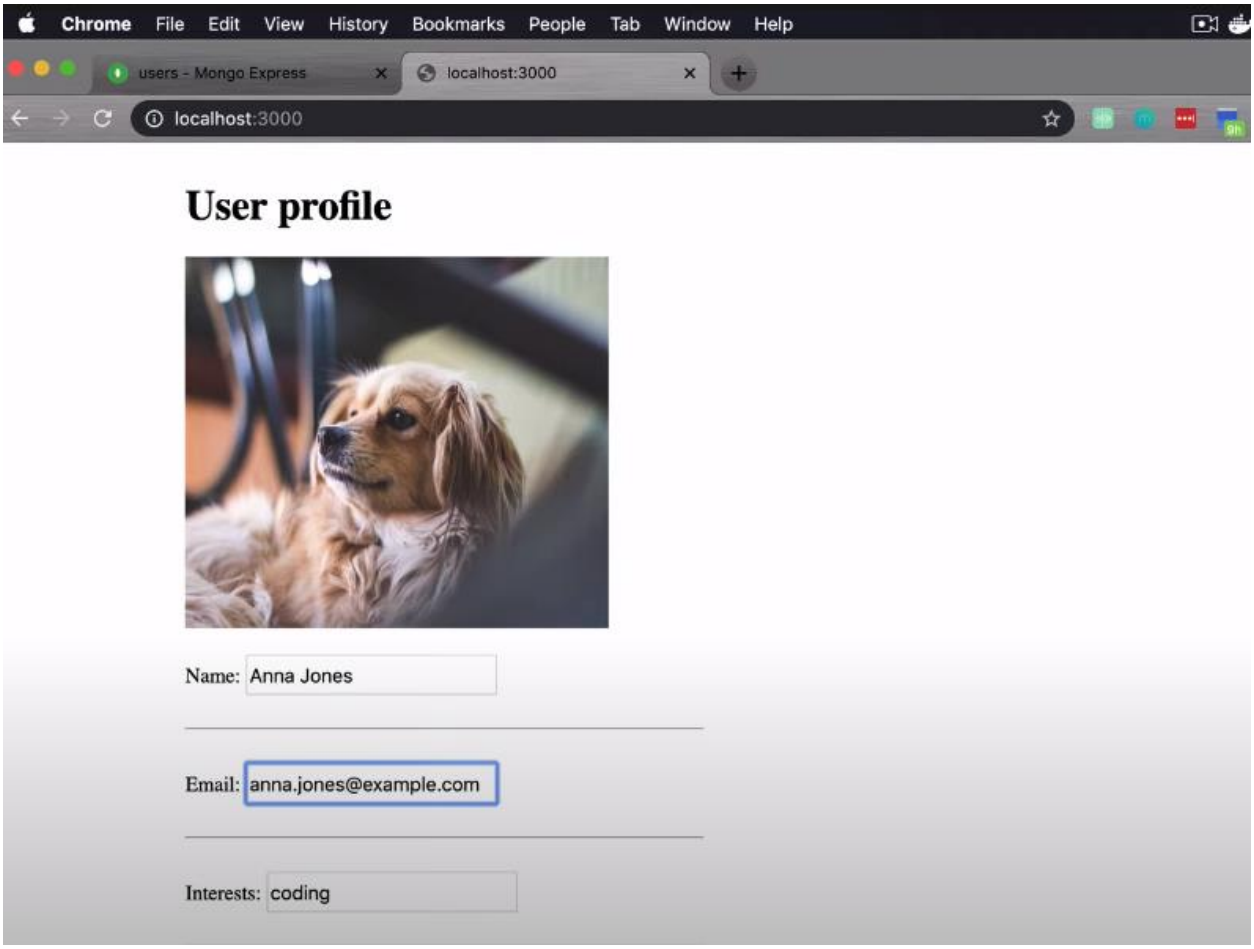
- Another thing is that logs of both the containers are mixed because we are starting them at the same time because mongo-express has to wait for mongo-db to start because it needs to establish a connection.
- When we have two containers where one depends on another one starting , we can actually configure this waiting logic in the **docker-compose file.**
- Now, let us see the docker containers that are running using the command **docker ps**
- We have both of them.

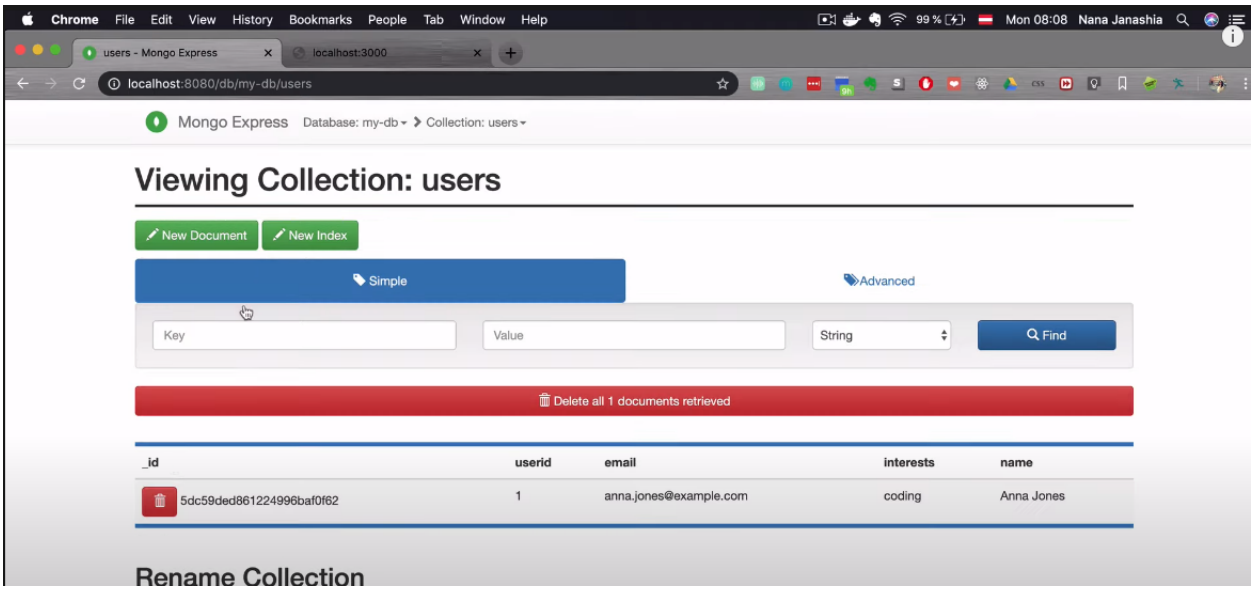
```
[~]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                     NAMES
31ce821760f7        mongo-express       "tini -- /docker-e..."   2 days ago          Up 3 minutes        0.0.0.0:8080->8081/tcp    myapp_mongo-express_1
887a135686a6        mongo               "docker-entrypoint..."   2 days ago          Up 3 minutes        0.0.0.0:27017->27017/tcp  myapp_mongodb_1
[~]$
```

- Here we have container names that **docker-compose** gave the containers.
- One thing we can actually note is that mongo express started on port **8081** inside the container.
- Here, we are opening the port **8080** on our laptop which actually forwards the request to the **port 8081** of the mongo-express.
- Now, we have restarted the containers and lets now check the first one which is **mongo-express.**



- Actually, we have now started the mongo-express in the port **8080** of the local host.
- We have already created the database which is **user-account** and now it is actually gone because we restarted the container and everything we configured in that application container is gone.
- So , the data is lost and there is no data persistence in the containers and it is very inconvenient because we want to have some persistence especially when we are working with the database.
- There is a concept which is called **volume** responsible for data persistence in containers.
- Lets now create the database called **my-db** and inside it lets now create a collection called **users.**
- **Now if we modified anything in the database, we will see the updated entry in the database.**





- Now, what to do when I want to stop those containers?
  The command is **docker-compose -f mongo.yaml down** which will shut down all the containers involved in the docker-compose file and also it will remove the network.

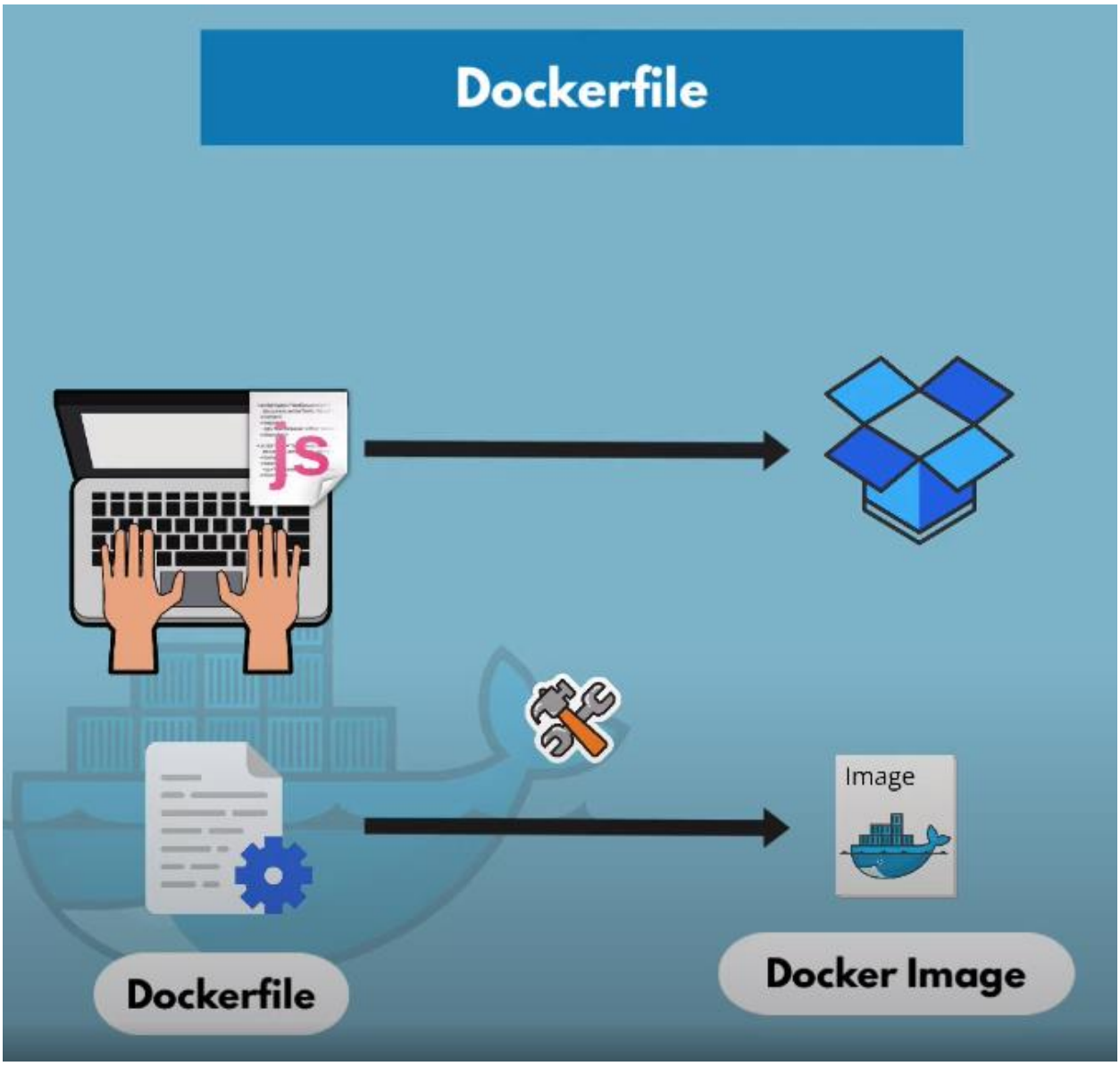
```
[my-app]$ docker-compose -f mongo.yaml down
Stopping myapp_mongodb_1 ... done
Stopping myapp_mongo-express_1 ... done
Removing myapp_mongo-express_1 ... done
Removing myapp_mongodb_1 ... done
Removing network myapp_default
[my-app]$
```

- The next time, we start it , then its gonna actually restart it.

```
[my-app]$ docker-compose -f mongo.yaml up -d
Creating network "myapp_default" with the default driver
Creating myapp_mongodb_1
Creating myapp_mongo-express_1
[my-app]$
```

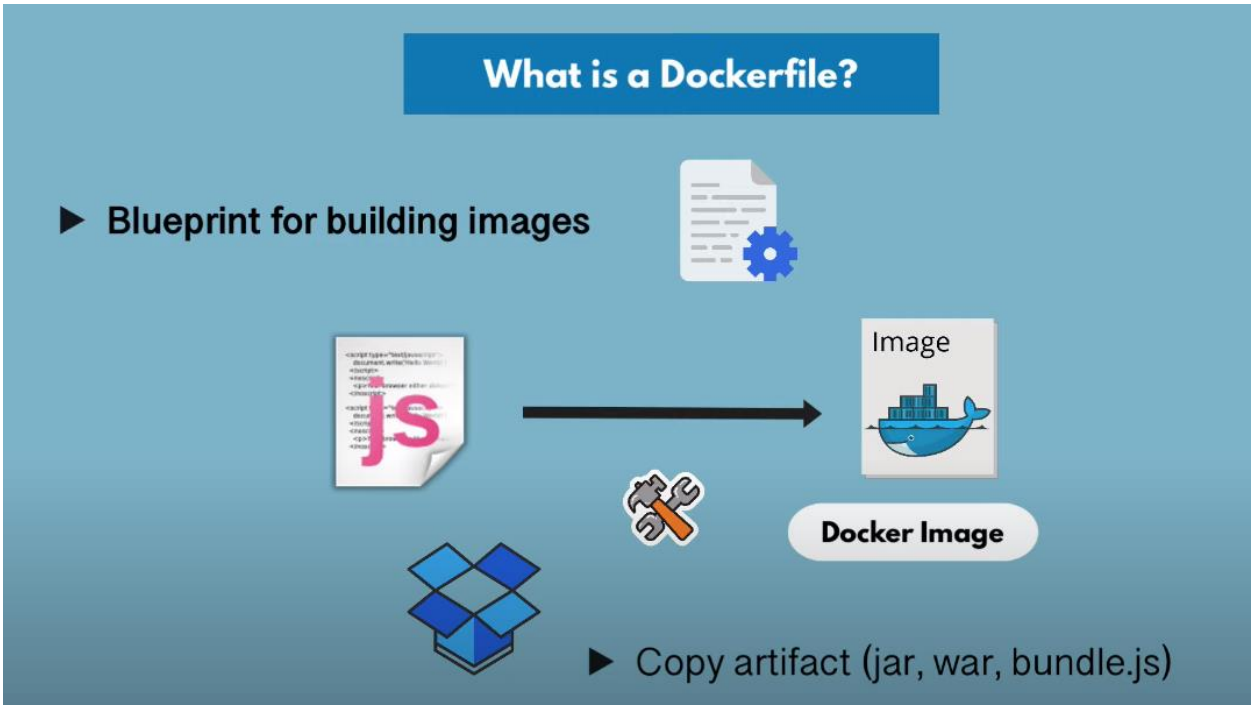
- Now again the two containers are recreated.

- Now lets see how to build our docker image from node js javascript application.
- Lets consider one of the scenario that we developed an application feature and we tested it and ready to deploy it.
- To deploy it, our application must be packaged into its own docker container which means we are gonna build a docker image from thejavascript node js backend application and prepare it to deploy it on some environment.



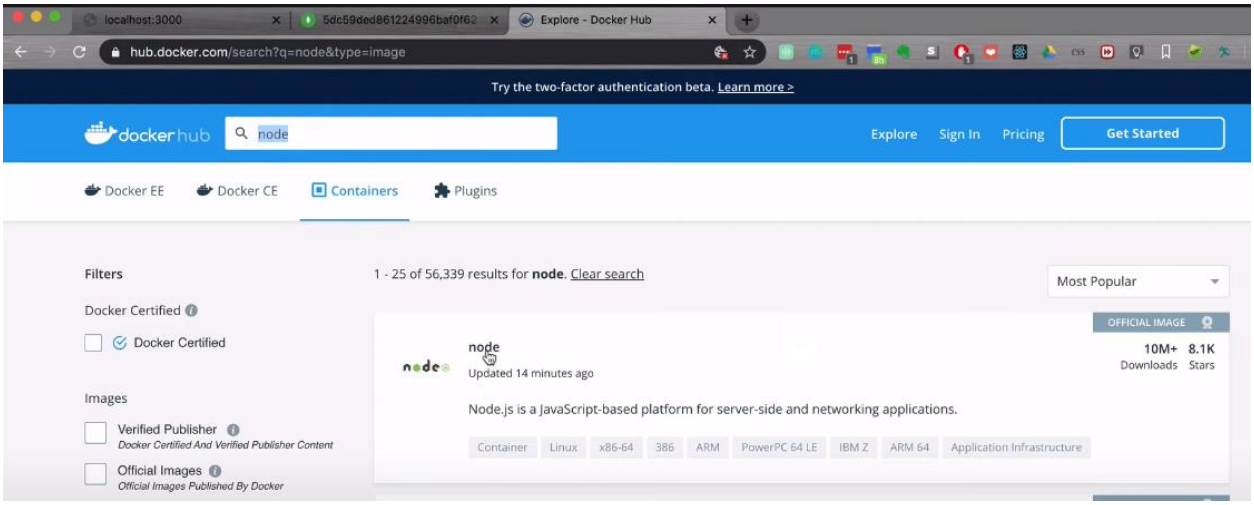
- So, to review it we have developed a javascript application and we have also used mongodb container from the docker hub. Now, its time to commit it to the git.
- We will simulate these steps in the local environment and after commit, we have continuous integration that runs. The question is what does Jenkins do with this application?
- When Jenkins builds the javascript application using **npm build** then it packages it into **Docker image** and then pushes it into docker repository.
- Here we will see how **Jenkins** will build the docker image on the local environment.

- In order to build docker file from the javascript application, we basically need to copy the contents of the application into docker file which means we need to copy artifact.



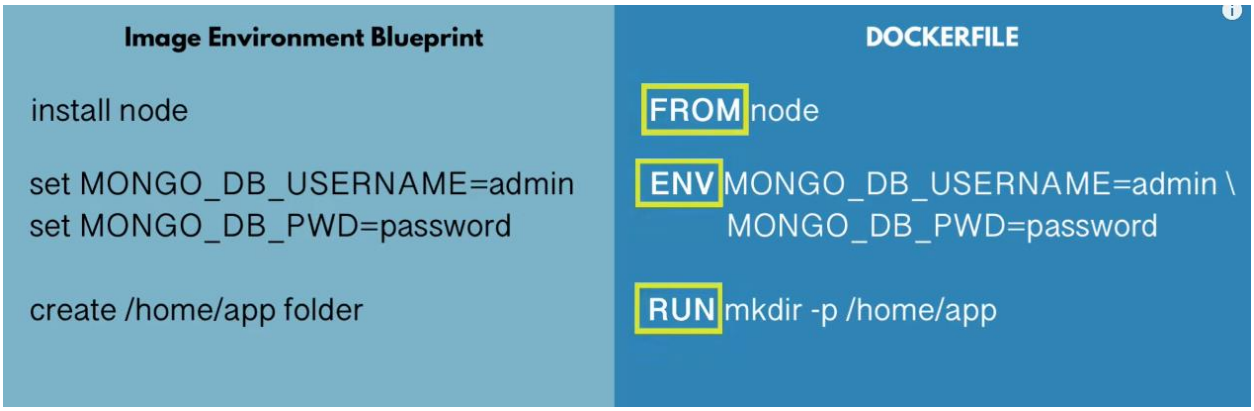
- So **Dockerfile** is the blueprint for building the images.
- The syntax of **DOCKERFILE** is super simple.
- The first line of every docker file is **FROM image** which means whatever image we are building, we need to base it on another image.

- In our case, we have a javascript application with node.js backend. So, we will be needing node inside of our container so, it can run our node application instead of basing it on a lower level alphine image or some lower level image because then we would have to install node ourselves on it.
- So, here we are taking a ready **node image.**
- Go to dockerhub and search for **node here.**

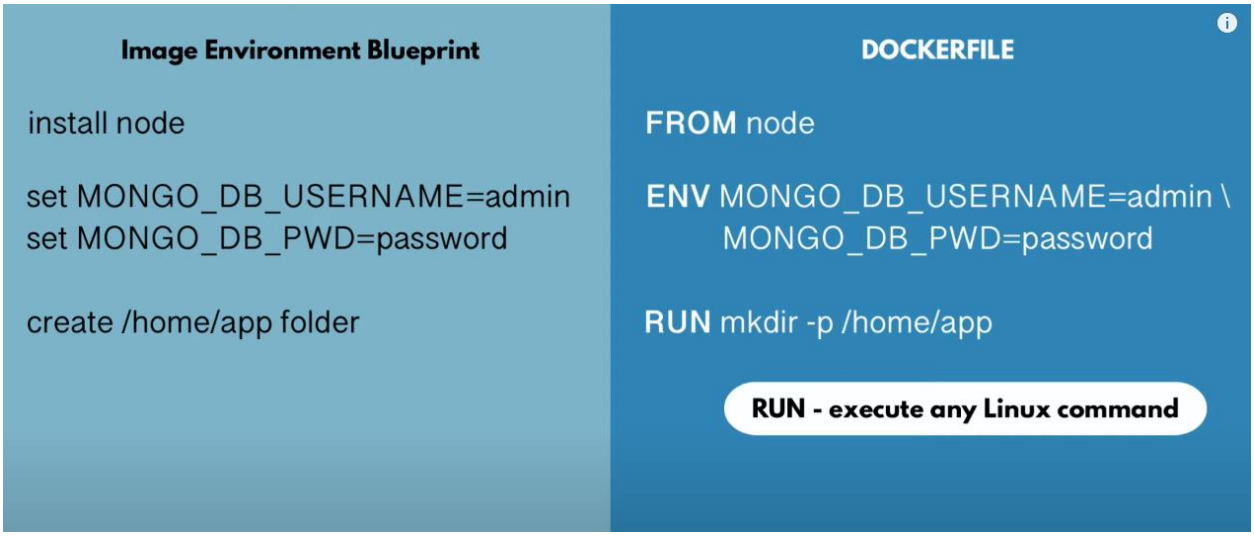


- Here we can find a ready node image from which we can base our image from.
- So, what does that actually means is that basing our own image on a node image is that we are going to have **node installed inside our image.**
- So, when we start the container , we get the terminal of the container and we can actually see node command available because node is installed there.
- This is what **FROM node** actually gives us.
- Now , the next one is that we can configure environmental variables inside our **DOCKERFILE.**
- But, its always better to optionally define the environmental variables.
- As we know that we have already done this using docker run commands or docker compose.
- So, this would be an alternative to defining environmental variables in docker compose.

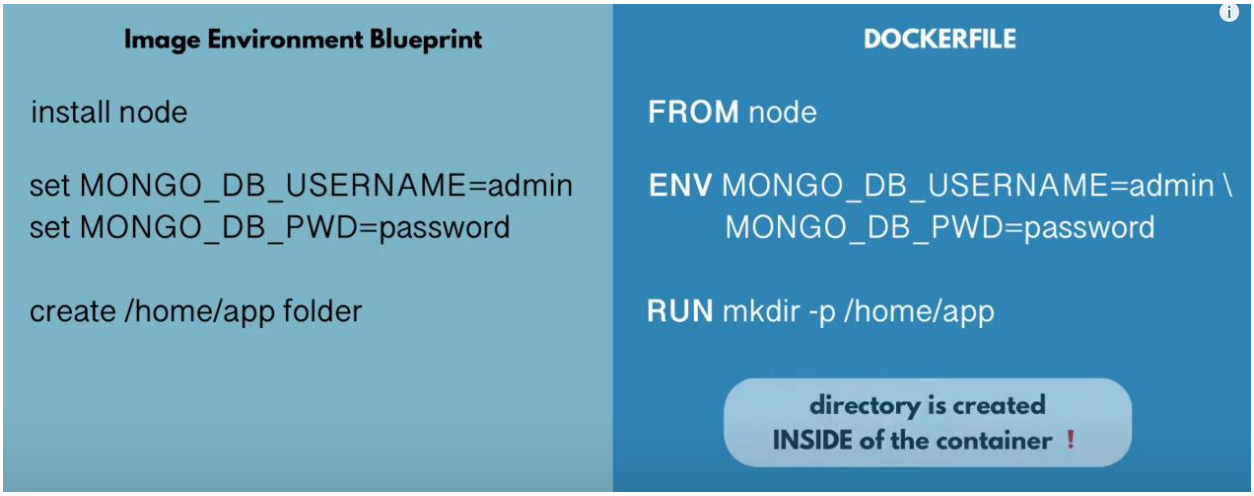
- Its always better to define environmental variables externally in a docker compose file because if something changes, then we can actually override it.
- We can change the docker compose file incorporated instead of rebuilding the image. But, this is an option.
- So this **ENV** command is basically used for setting the environmental variables inside of an image environment.
- So, the next one is **ENV**
- The capital letters which we see in the below image are basically part of the syntax of the docker file.



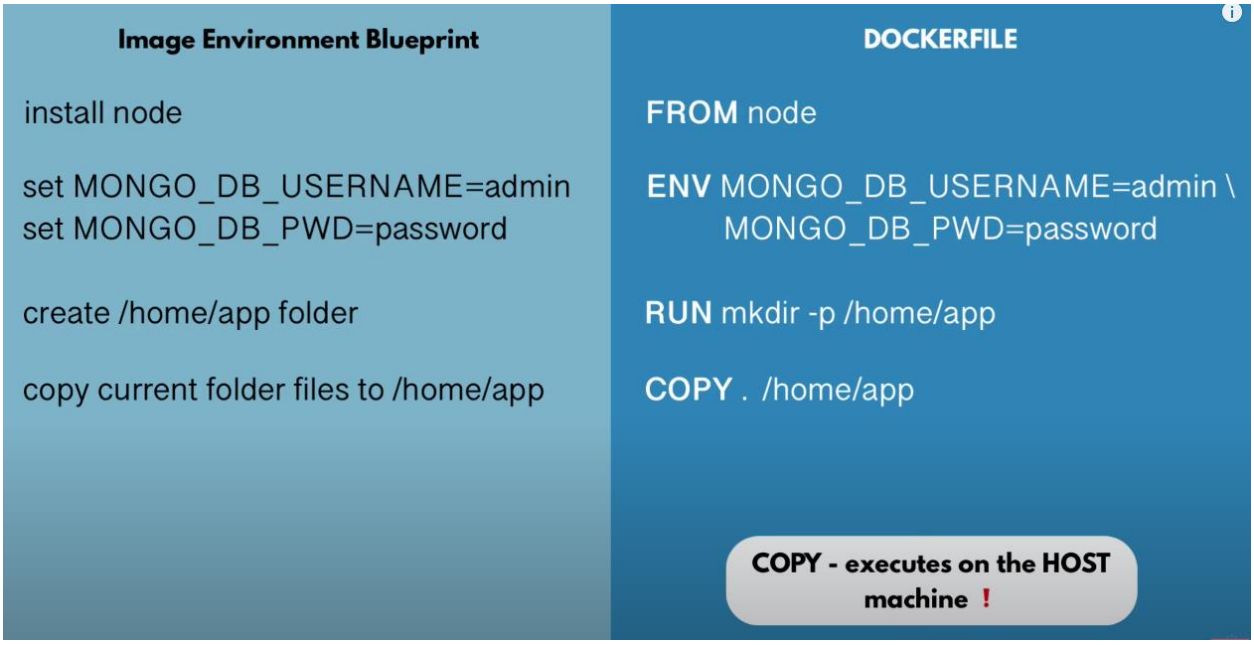
- So , basically using **RUN** command basically we can execute any kind of linux commands, so in the above image we can see that **mkdir makes a directory ie., home/app** directory.
- The very important point to note here is that this directory is going to live inside the container.



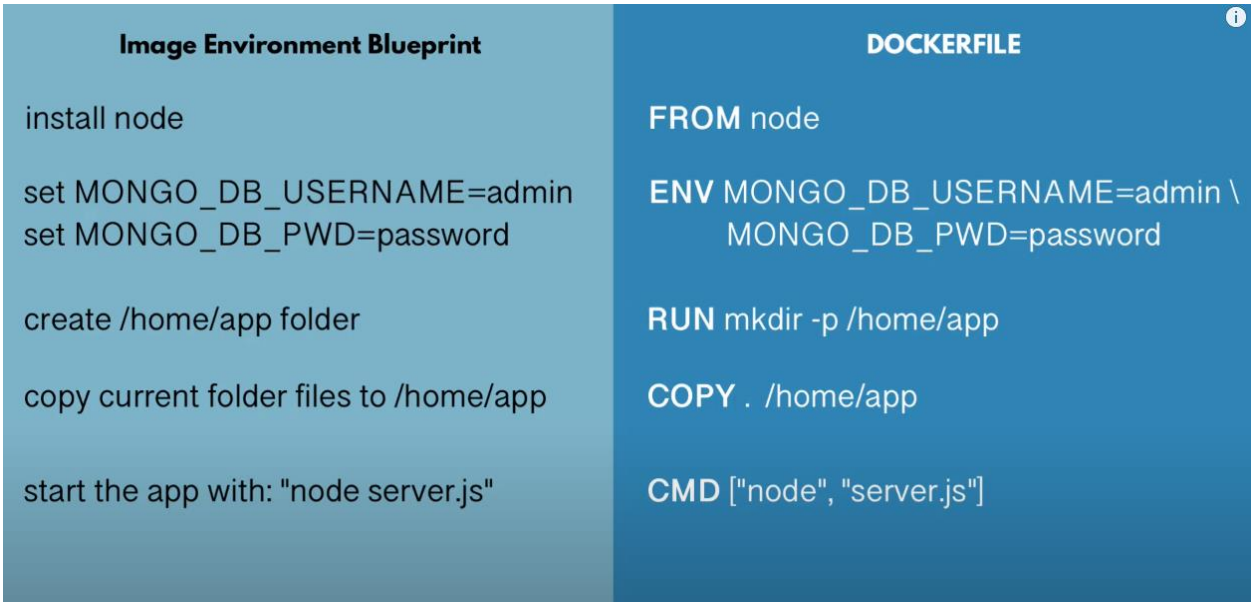
- So, actually when I start a container with this image, then **home/app** directory will be created inside of the directory but not on the laptop ie., not on the host.
- So ,all the commands that we have in **DOCKERFILE** will apply to the container environment.
- None of the commands from the **DOCKERFILE** will be affecting my host environment or laptop environment.
- So , with **RUN** we can execute any linux commands that we want.



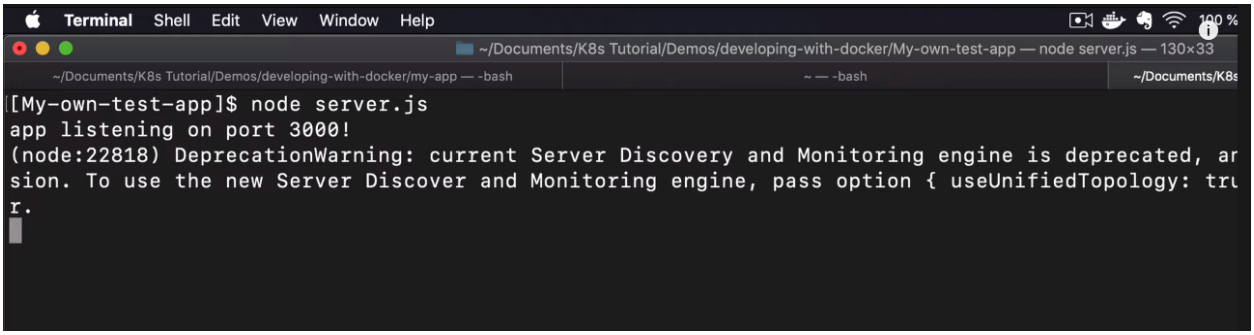
- Now, we have **COPY** command and we could ask that we can execute a copy command in linux copy command using run.
- All the commands gets executed inside of the container.
- The **COPY** command which we see here gets executed on the host.



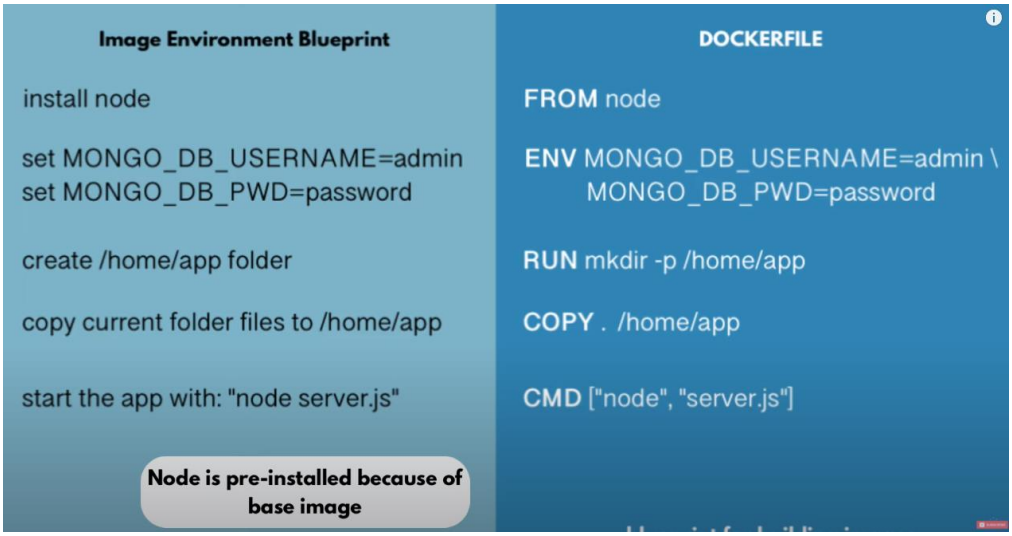
- Here , we can see that the first parameter in the **COPY** command is the host and the second parameter is **/home/app**
- So the first parameter **.** is the source and the second one is the target.
- So, I can copy files that I have on my host inside of that container image.
- Because if I were to execute run cp source destination, then that command would execute inside of the docker container.
- But we have the files that we want to copy on our host.
- Last one which is the **CMD or the command** is actually part of the **DOCKERFILE.**
- **CMD** will execute the entry point linux command.
- the line beside the **CMD** will translate to **node server.js**



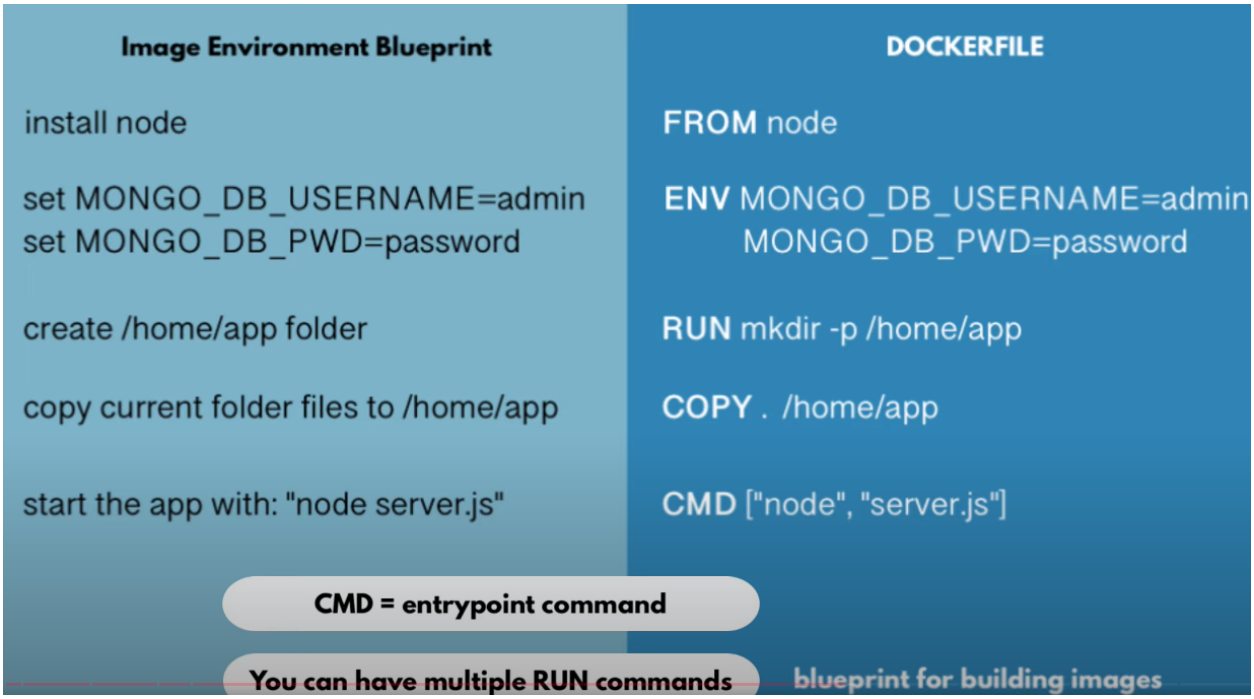
- so the command in the **CMD** will translate to the command as shown below.



- so , here we start the node server with the node js , this is what exactly done by **CMD[“node”,”server.js”]** but inside of the container.
- Once we copied our server.js and other files inside of the container and we can do it because node js is pre installed because of the base image.

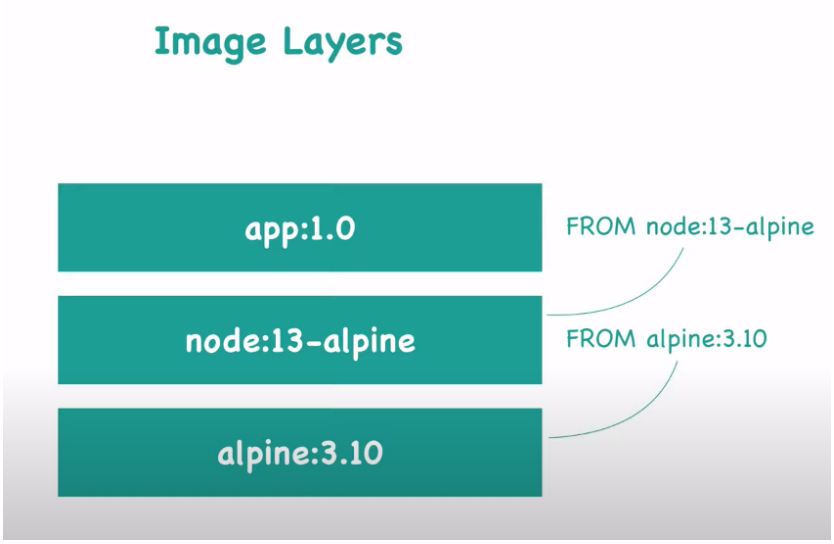


- Difference between **RUN** and **CMD** command is that we can have multiple run commands but we only have one **CMD** command which is the entry point command for the **DOCKERFILE.**

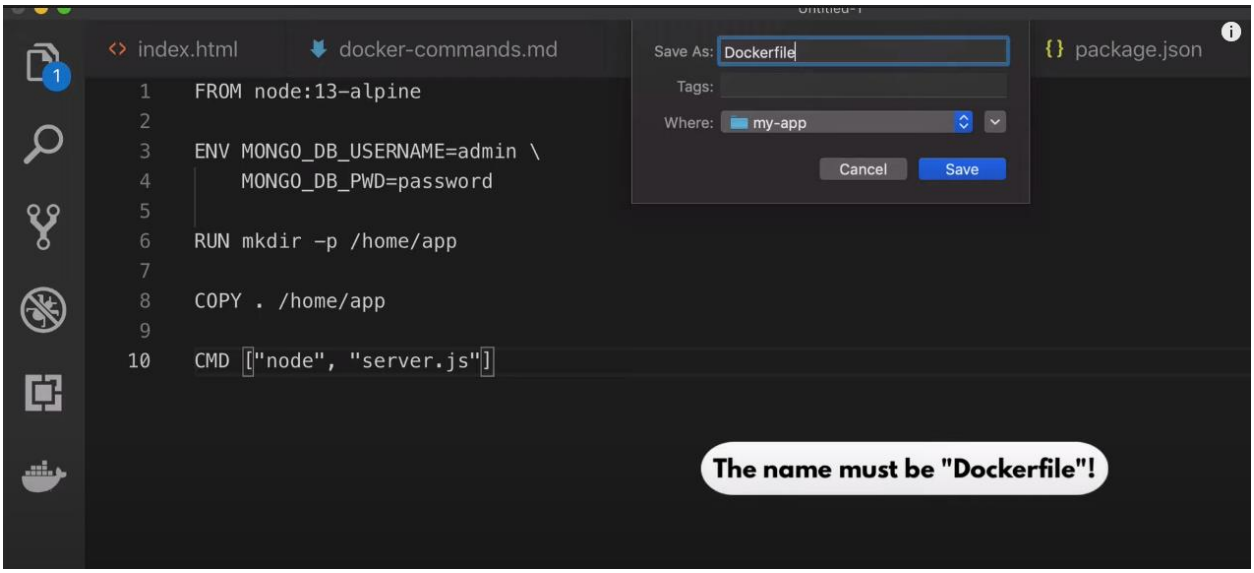


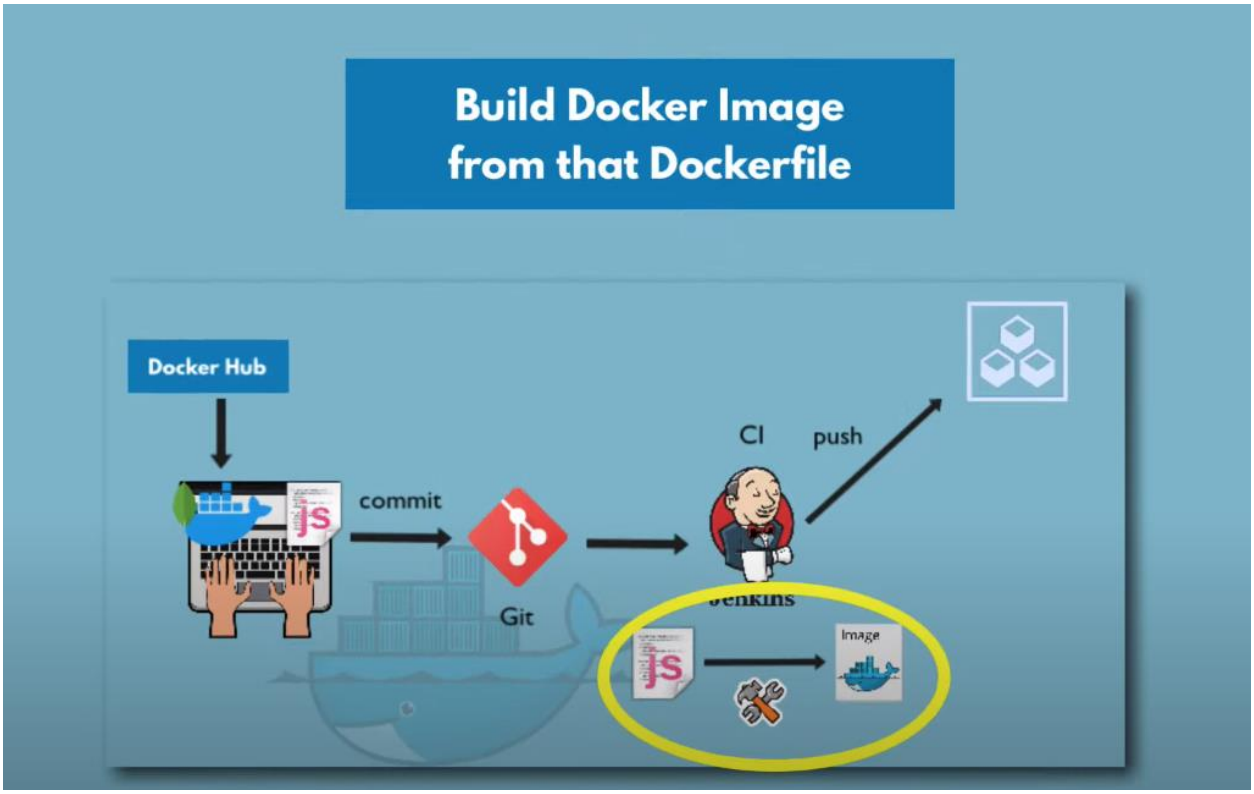
- Now, lets create the **DOCKERFILE.**

- Just like the **docker-compose file, DOCKERFILE** is the part of the application code.
- We know that **DOCKERFILE** is the blueprint for building the images.
- So , the point is that we have docker images in the docker hub. So they must be definitely having the **DOCKERFILE.** And each docker image is based on some base image.
- Lets for example say we are building **app 1.0 image** then they would be on some other two images as shown below.



- There is some naming convention for docker file. It should be saved as **Dockerfile.**

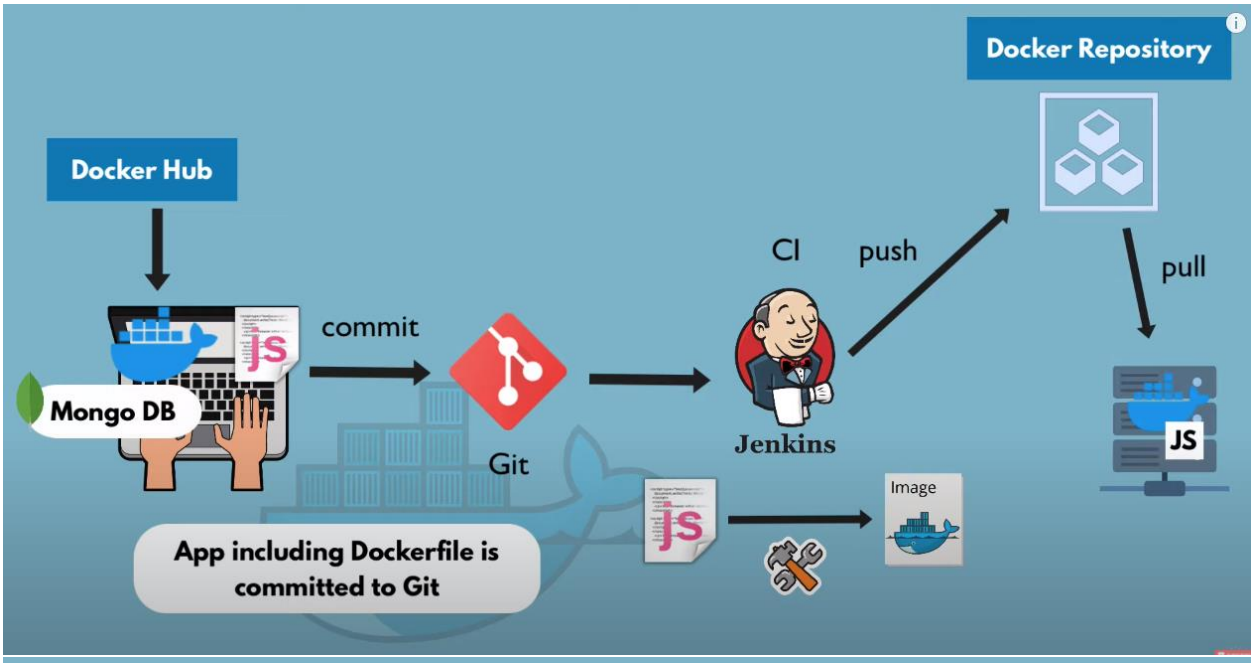


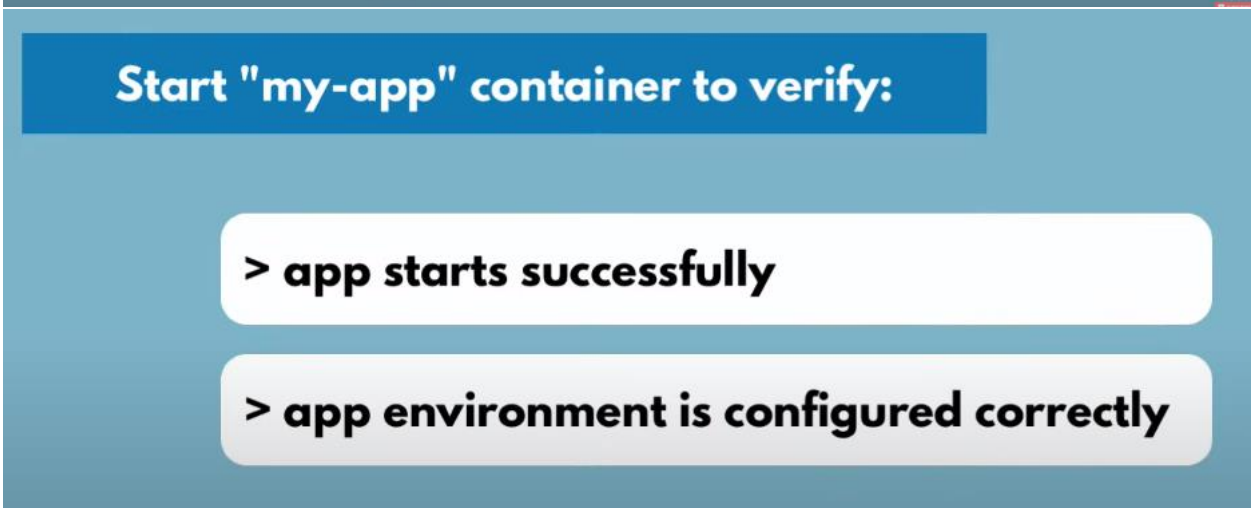


- Now that we have docker file , we need to build docker image from the docker file.
- To build docker image from the docker file we need to specify two options.
- The first one is we need to give a image name and a specific version using **-t** option
- The second option is to specify the location of the docker file.
- Since we are in the same location as of the docker file we can simply tell **.**
- The command will be **docker build -t my-app:1.0 .**

```
~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app — -bash
[my-app]$ docker build -t my-app:1.0 .
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
 ---> Running in b07fe1fc1d7f
 ---> 7d5427f96392
Removing intermediate container b07fe1fc1d7f
Step 3/5 : RUN mkdir -p /home/app
 ---> Running in 109d8ed6845f
 ---> d1676988c5ef
Removing intermediate container 109d8ed6845f
Step 4/5 : COPY . /home/app
 ---> eb60dad54e2c
Removing intermediate container 4ec2462f1801
Step 5/5 : CMD node server.js
 ---> Running in 5103528e9a36
 ---> 2e0a4d16e074
Removing intermediate container 5103528e9a36
Successfully built 2e0a4d16e074
[my-app]$
```

- Now, we can clearly see that the image is built and we can see the id of the image.
- Now execute the command which is **docker images** so that we can see the list of images we have.
- So , what we have done is that we have created Javascript application with the docker containers and then we committed it to the git. Along with the javascript code, docker-compose file we need to push **DOCKERFILE** along with them.
- So what **Jenkins** does is that it pulls the **DOCKERFILE** and then builds the docker image from the **DOCKERFILE.**
- In order to test this image locally by the tester, we need to deploy this image in the development server and to do that we need to actually share the image so it is pushed into the **DOCKER REPOSITORY** like **docker hub.**
- From there, the tester can download it and test it. Or the development server can actually pull it.





- Lets just now run a container and the image name is **my-app:1.0**

- Here just let us give the option which is **docker run my-app:1.0**

```
[my-app]$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
my-app              1.0                 2e0a4d16e074        2 days ago          116 MB
node                13-alpine           f20a6d8b6721        3 days ago          105 MB
mongo               latest              965553e202a4        10 days ago         363 MB
mongo-express       latest              597a2912329c        13 days ago         97.6 MB
redis               4.0                 e187e861db44        3 weeks ago         89.2 MB
redis               latest              de25a81a5a0b        3 weeks ago         98.2 MB
postgres            9.6.5               bd287e105bc1        2 years ago         266 MB
[my-app]$ docker run my-app:1.0
internal/modules/cjs/loader.js:895
    throw err;
    ^

Error: Cannot find module '/server.j'
    at Function.Module._resolveFilename (intern l/modules/cjs/loader.js:892:15)
    at Function.Module._load (internal/modules/ js/loader.js:785:27)
    at Function.Module.runMain (internal/modules/cjs/loader.js:1143:12)
    at internal/main/run_main_module.js:16:11 {
  code: 'MODULE_NOT_FOUND',
  requireStack: []
}
[my-app]$
```

- The problem is that it cannot find the module **server.js file** which is actually logical.

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

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

- Here in the **CMD** we are not telling it to look into the correct directory which is **/home/app**
- Since we are copying all the resources in the **/home/app** directory , then **server.js** is also going to be there as well.
- Whenever we **adjuct the dockerfile => rebuild the image** because the old image cannot be overwritten.

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

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

When you **adjust** the Dockerfile,
you MUST **rebuild the Image**!

- So, now when we try to delete an image , it is throwing an error that the image is being used by the stopped container.

```
[my-app]$ docker rmi 2e0a4d16e074
Error response from daemon: conflict: unable to delete 2e0a4d16e074 (must be forced) - image is being used by stopped container 3c
58e681c8c5
[my-app]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
  NAMES
0f24705dc9f3        mongo               "docker-entrypoint..."   2 days ago          Up 48 minutes       0.0.0.0:27017->27017/tcp
  myapp_mongodb_1
db72c2d560d2        mongo-express       "tini -- /docker-e..."   2 days ago          Up 48 minutes       0.0.0.0:8080->8081/tcp
  myapp_mongo-express_1
[my-app]$
```

- Use the command **docker ps -a | grep myapp**
- From the above command we get the stopped container id and so we will delete that using the command which is **docker rm container_id.**
- To delete the image we use the command which is **docker rmi image_id**

```
[my-app]$ docker ps -a | grep my-app
3c58e681c8c5        my-app:1.0          "docker-entrypoint..."   2 days ago          Exited (1) 2 minutes ago
         distracted_galileo
[my-app]$ docker rm 3c58e681c8c5
3c58e681c8c5
[my-app]$ docker rmi 2e0a4d16e074
Untagged: my-app:1.0
Deleted: sha256:2e0a4d16e074bc8e74fe2b60ecebbcb92cd19039046b421db55d343cf460d93f
Deleted: sha256:eb60dad54e2cca8586784b8d28b6bbbff089e129f4c1c7b0031cb8c8fef78254
Deleted: sha256:ca83e53294d19c676846170282c95bd7ff21bf0f4283bc09dc25d4deacd1cd94
Deleted: sha256:d1676988c5efe6c7deee166c57db7e44836d62e9c2a988d731508fe290355382
Deleted: sha256:15f250494debbb3ce2d69807cc2ae115cbd7f0eddb45c7d1697eb388dce9632a
Deleted: sha256:7d5427f96392d0bbc325bb4bd0d7b10dd192841ecbc5cb7bf1adb4b185c5880b
[my-app]$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
node                13-alpine          f20a6d8b6721        3 days ago          105 MB
mongo               latest              965553e202a4        10 days ago         363 MB
mongo-express       latest              597a2912329c        13 days ago         97.6 MB
redis               4.0                 e187e861db44        3 weeks ago         89.2 MB
redis               latest              de25a81a5a0b        3 weeks ago         98.2 MB
postgres            9.6.5               bd287e105bc1        2 years ago         266 MB
[my-app]$
```

- Now, we can see that the problem is solved as shown in the below screenshot.

```
[my-app]$ docker build -t my-app:1.0 .
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
 ---> Running in 406af7d7f8f0
 ---> 4b096fd04284
Removing intermediate container 406af7d7f8f0
Step 3/5 : RUN mkdir -p /home/app
 ---> Running in b94b69893860
 ---> ffbb5b53805b
Removing intermediate container b94b69893860
Step 4/5 : COPY . /home/app
 ---> c20eda4bccef
Removing intermediate container 23706ce332be
Step 5/5 : CMD node /home/app/server.js
 ---> Running in c31fc7492bbd
 ---> 0a524ee31427
Removing intermediate container c31fc7492bbd
Successfully built 0a524ee31427
[my-app]$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
my-app              1.0                 0a524ee31427        2 days ago          116 MB
node                13-alpine           f20a6d8b6721        3 days ago          105 MB
mongo               latest              965553e202a4        10 days ago         363 MB
mongo-express       latest              597a2912329c        13 days ago         97.6 MB
redis               4.0                 e187e861db44        3 weeks ago         89.2 MB
redis               latest              de25a81a5a0b        3 weeks ago         98.2 MB
postgres            9.6.5               bd287e105bc1        2 years ago         266 MB
[my-app]$ docker run my-app:1.0
app listening on port 3000!
```

```
~ — docker exec -it 51c6912d69f5 /bin/sh — 155×39
~ — docker exec -it 51c6912d69f5 /bin/sh   ~ — -bash   ~/Documents/K8s Tutorial/Demos/devel
[~]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
51c6912d69f5        my-app:1.0          "docker-entrypoint..."   2 days ago          Up 34 seconds
0f24705dc9f3        mongo               "docker-entrypoint..."   2 days ago          Up 50 minutes       0.0.0.0:27017->27017/tc
db72c2d560d2        mongo-express       "tini -- /docker-e..."   2 days ago          Up 50 minutes       0.0.0.0:8080->8081/tcp
[~]$ docker logs 51c6912d69f5
app listening on port 3000!
[~]$ docker exec -it 51c6912d69f5 /bin/bash
rpc error: code = 2 desc = oci runtime error: exec failed: container_linux.go:247: starting container process caused "exec: \"/b
no such file or directory"

[~]$ docker exec -it 51c6912d69f5 /bin/sh
/ # ls
bin    dev    etc    home   lib    media  mnt    opt    proc   root   run    sbin   srv    sys    tmp    usr    var
/ #
```

- We can see that we have entered inside the container.

index.html | docker-commands.md | mongo.yaml | Dockerfile | package.json

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

- Here we can see that we have set some environment variables. So, we should find them in the docker environment too.

```
[~]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
51c6912d69f5        my-app:1.0          "docker-entrypoint..."   2 days ago          Up 34 seconds
0f24705dc9f3        mongo               "docker-entrypoint..."   2 days ago          Up 50 minutes       0.0.0.0:27017->27017/tc
db72c2d560d2        mongo-express       "tini -- /docker-e..."   2 days ago          Up 50 minutes       0.0.0.0:8080->8081/tcp
[~]$ docker logs 51c6912d69f5
app listening on port 3000!
[~]$ docker exec -it 51c6912d69f5 /bin/bash
rpc error: code = 2 desc = oci runtime error: exec failed: container_linux.go:247: starting container process caused "exec: \"/b
no such file or directory"

[~]$ 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=/
/ #
```

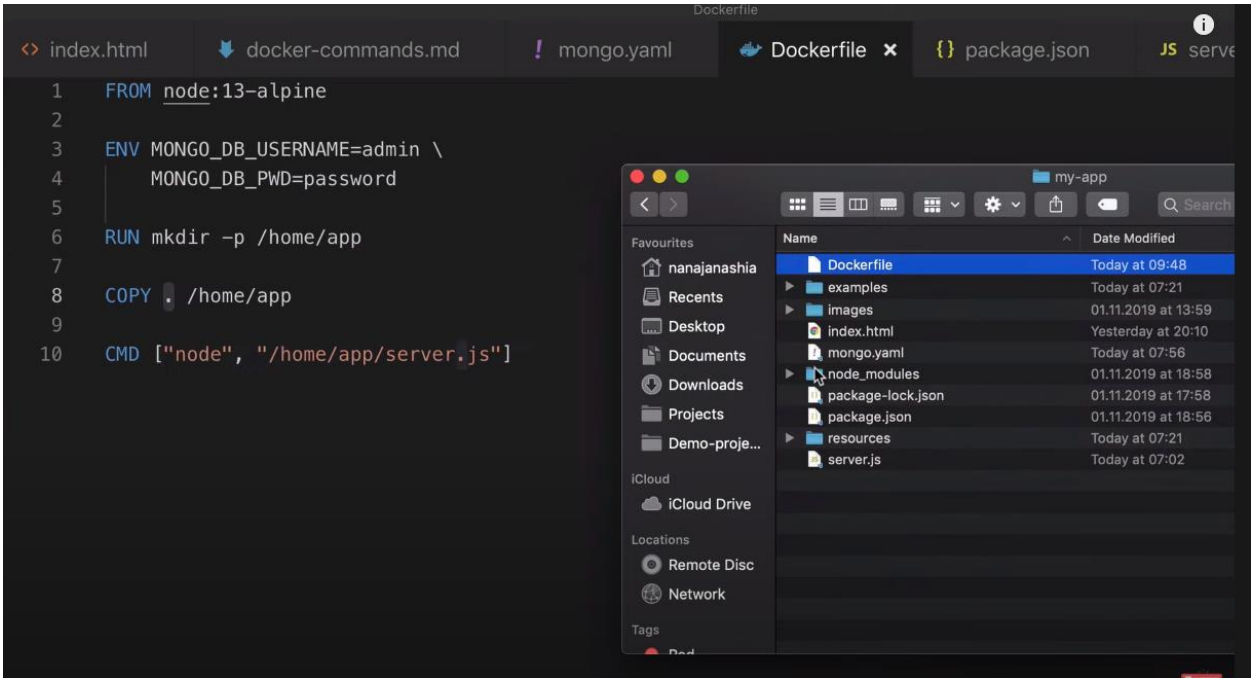
- So if we give the command **env** then we can actually see the MONGO_DB_USERNAME and the MONGO_DB_PwD
- Another thing we can check is the **/home/app directory** because with **RUN /home/app** directory we have actually created it.

Dockerfile

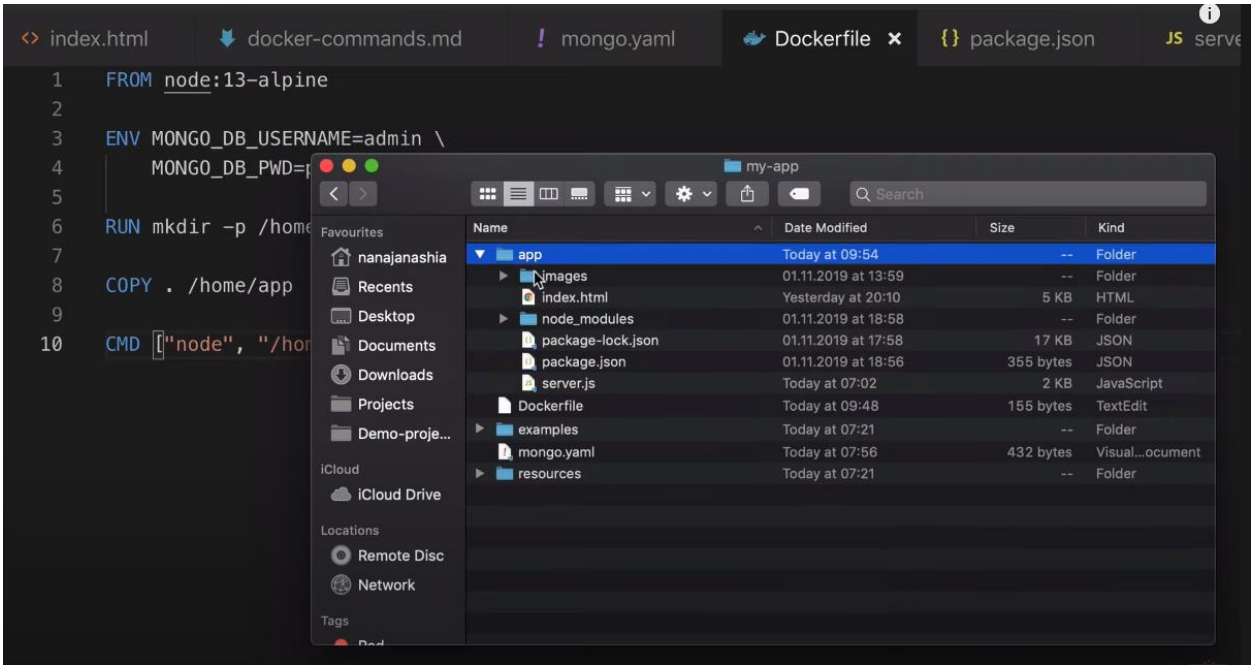
index.html | docker-commands.md | mongo.yaml | Dockerfile | package.json

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

- Use the command **ls /home/app** directory as we can see from the below image the directory is created.

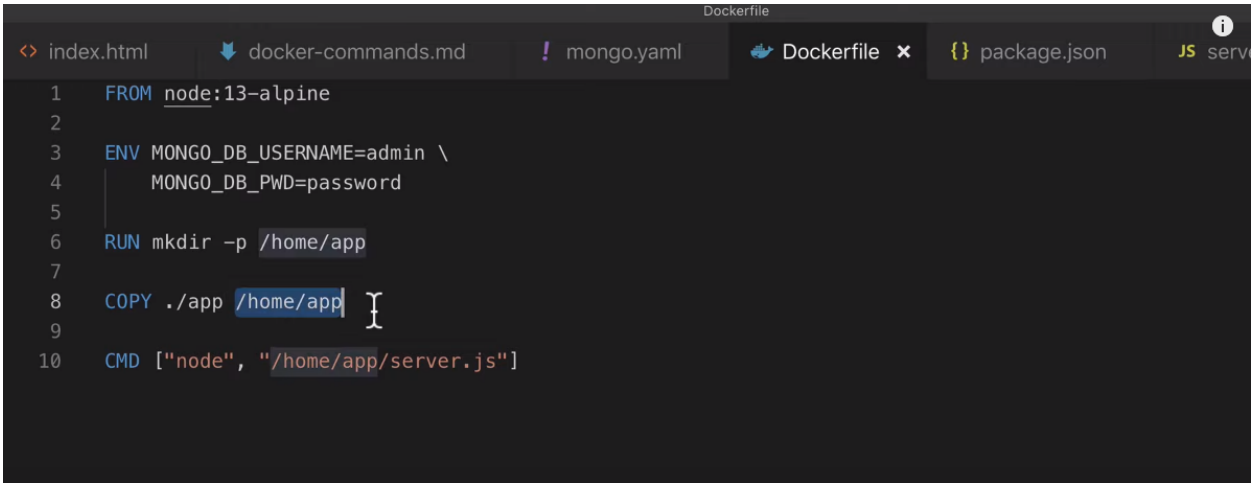


- So ,here basically we have copied everything from the current directory to the container.
- Now, we can do some improvement. We can create one folder called **app** and then we can copy the files which are necessary for starting the application.

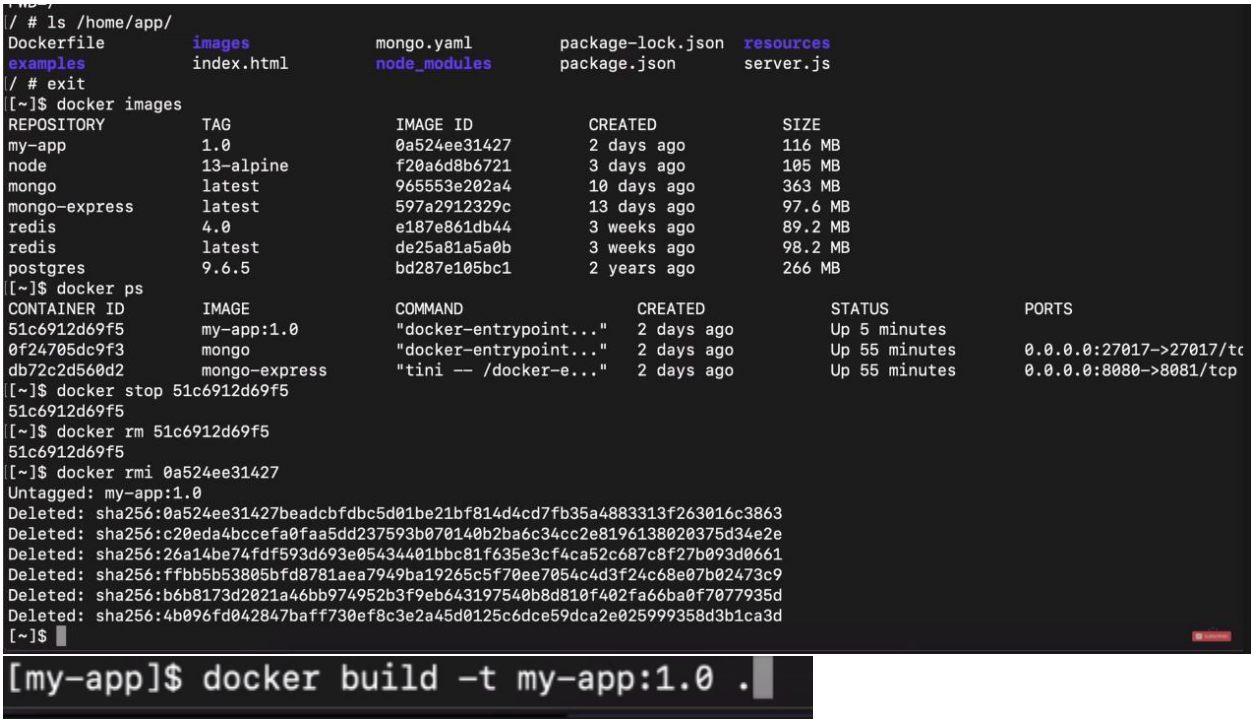


- Now use the **COPY** command to copy the **app folder** to the **/home/app directory**

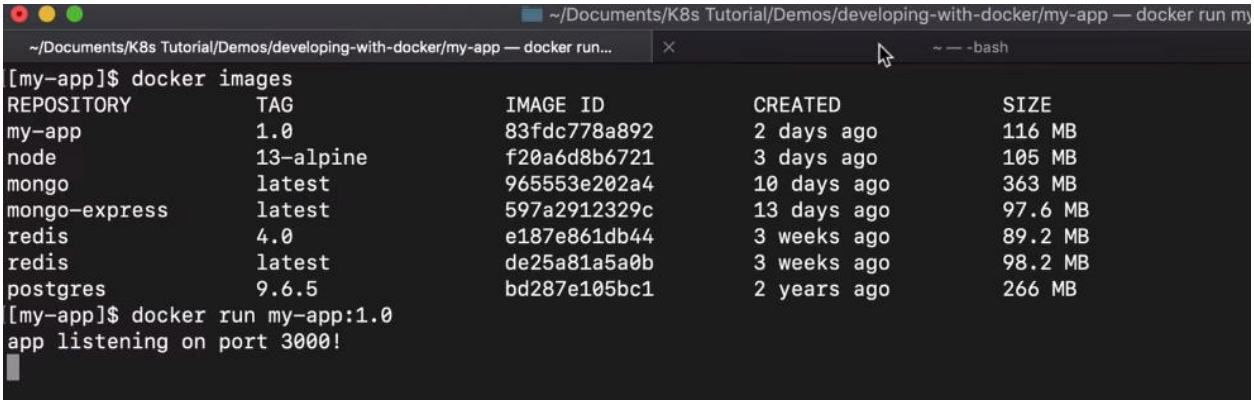
- Since we have modified the **DOCKERFILE,** we need to rebuild the image.

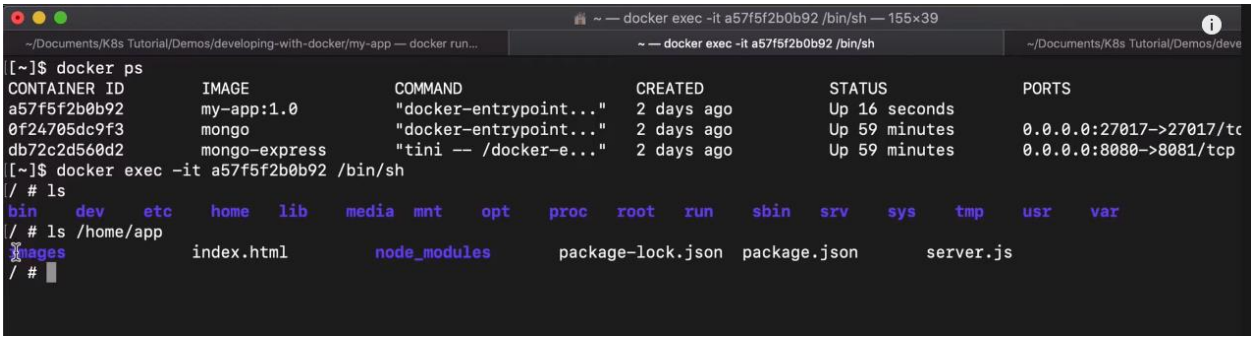


- In order to leave the docker container terminal, we can use **exit command.**



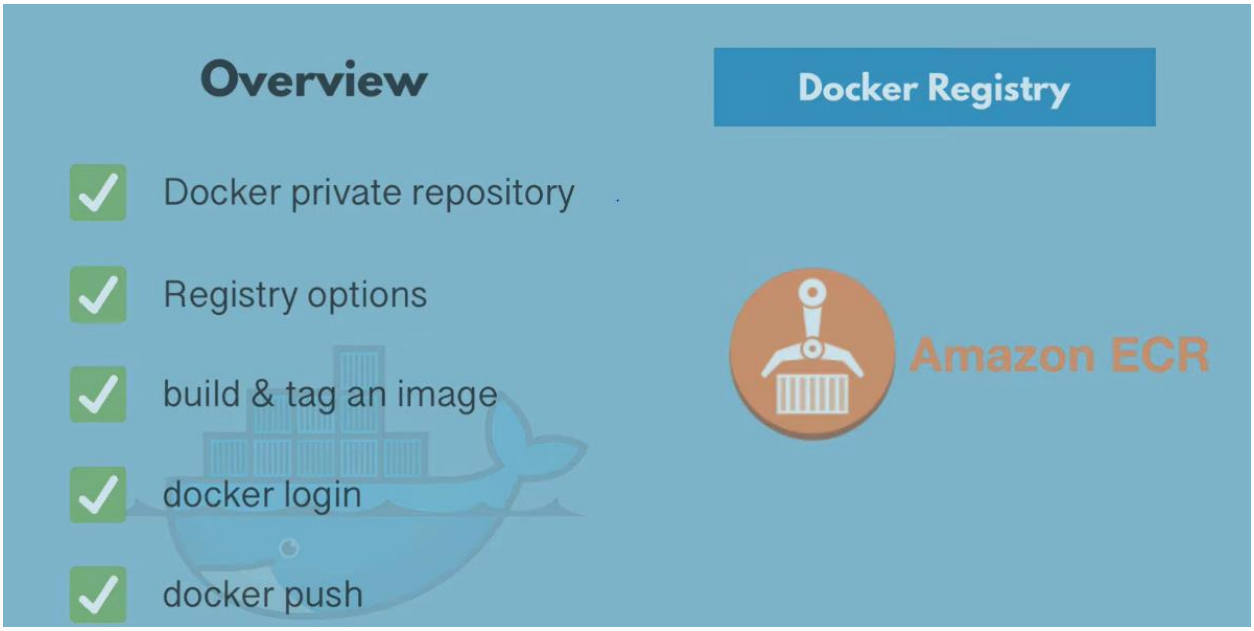
- Now, we have image built and we need to run it.



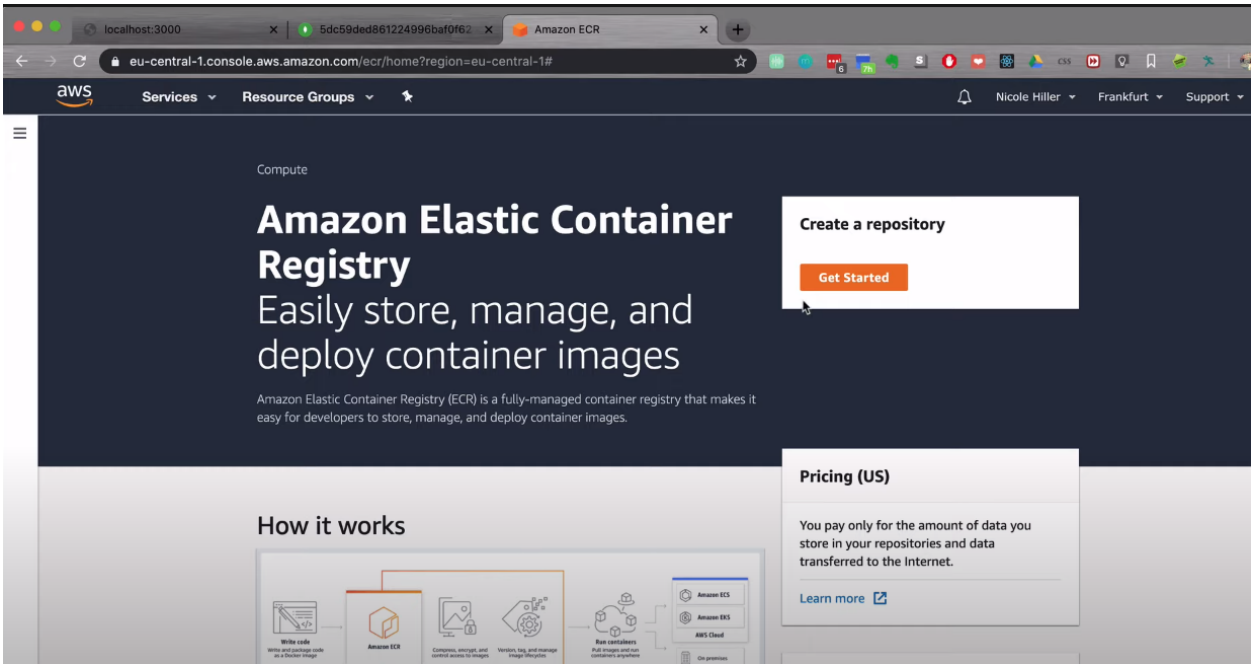


- Since we have few files, we have copied them to the app directory.
- But usually if we have huge applications then we will compress those files and package them into an artifact and then copy that artifact into a docker image container.

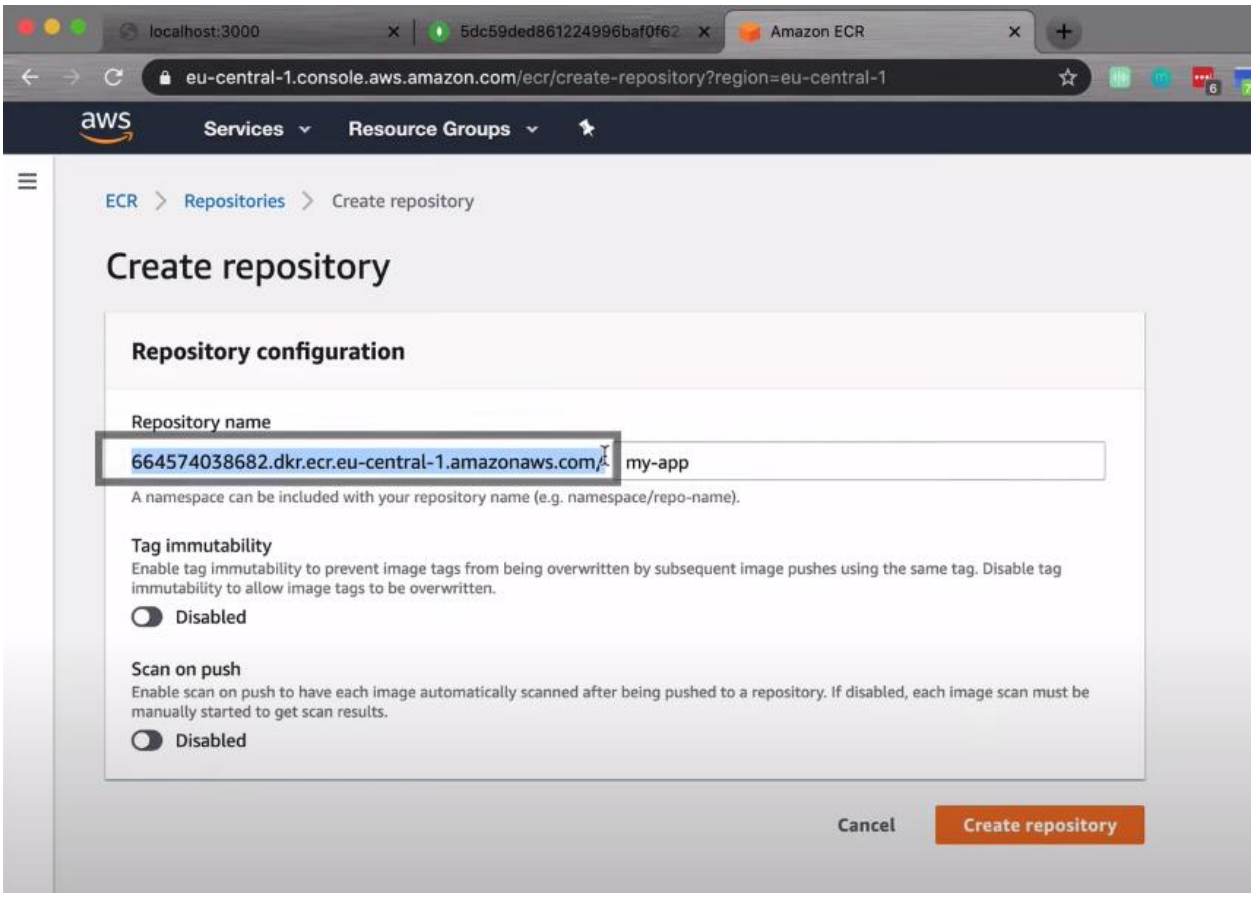
- Here we are going to create private repository for docker images on AWS ECR.
- There are many options for docker registries, among them nexus and digital ocean are the ones.
- We gonna see how to create a registry there.
- We see building and tagging an image, so that we can push that into the repository.
- In order to push the image into the private repository, we first have to login to that repository.



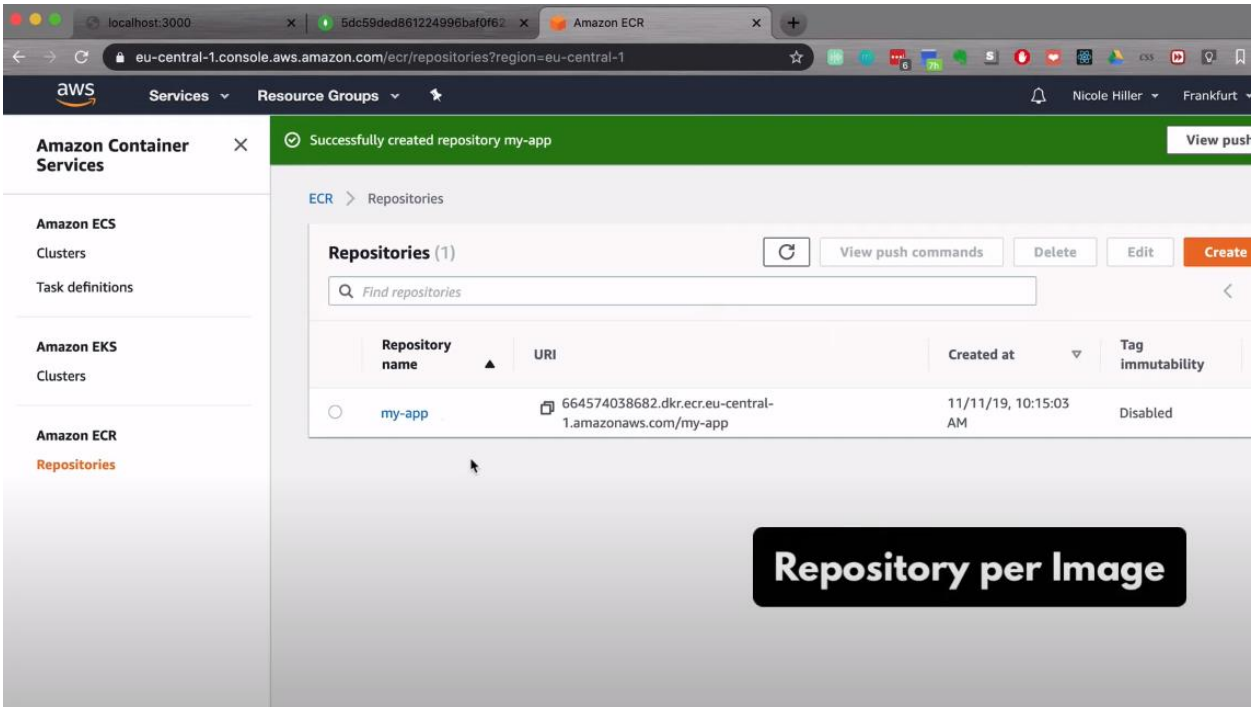
- The first thing is we should create a private repository for docker which is called docker registry.
- We are going to create it in aws.
- The service , we gonna use is **ELASTIC CONTAINER REGISTRY.**



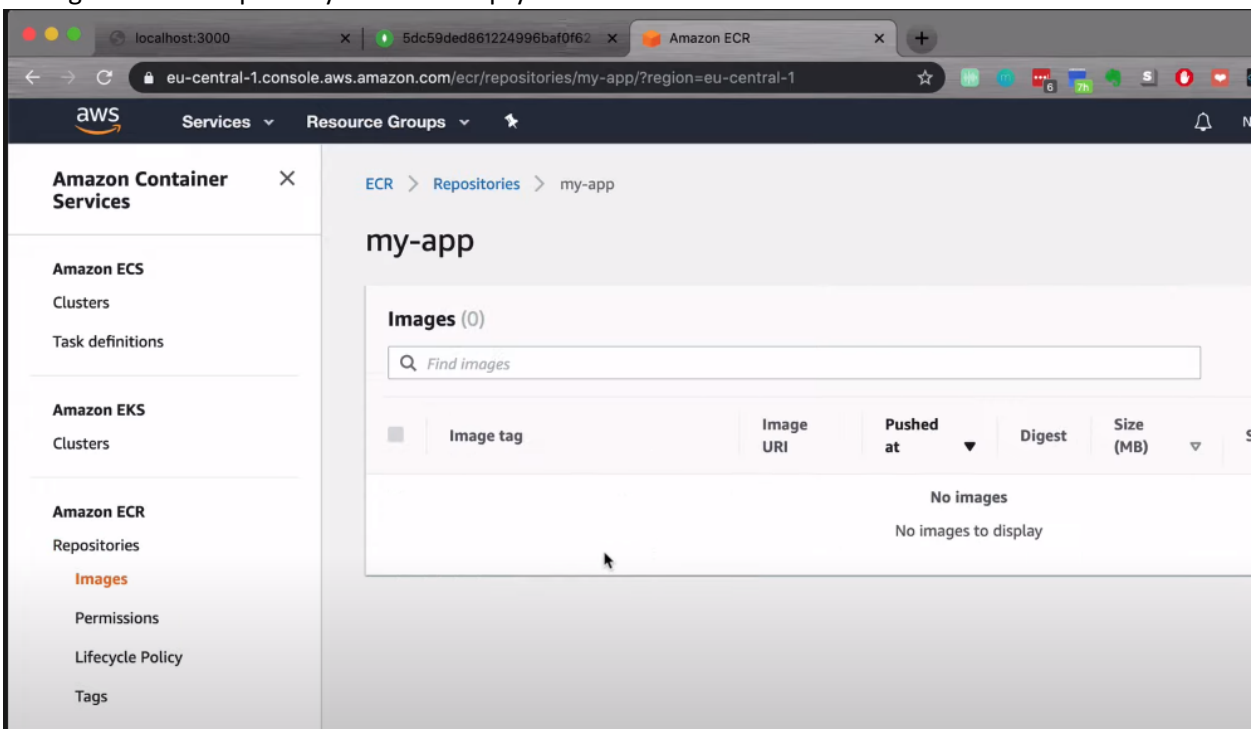
- Now, we need to create a repository here.
- After we click on the get started , we need to give the name for the repository. In this case, let us give the name as our application name.



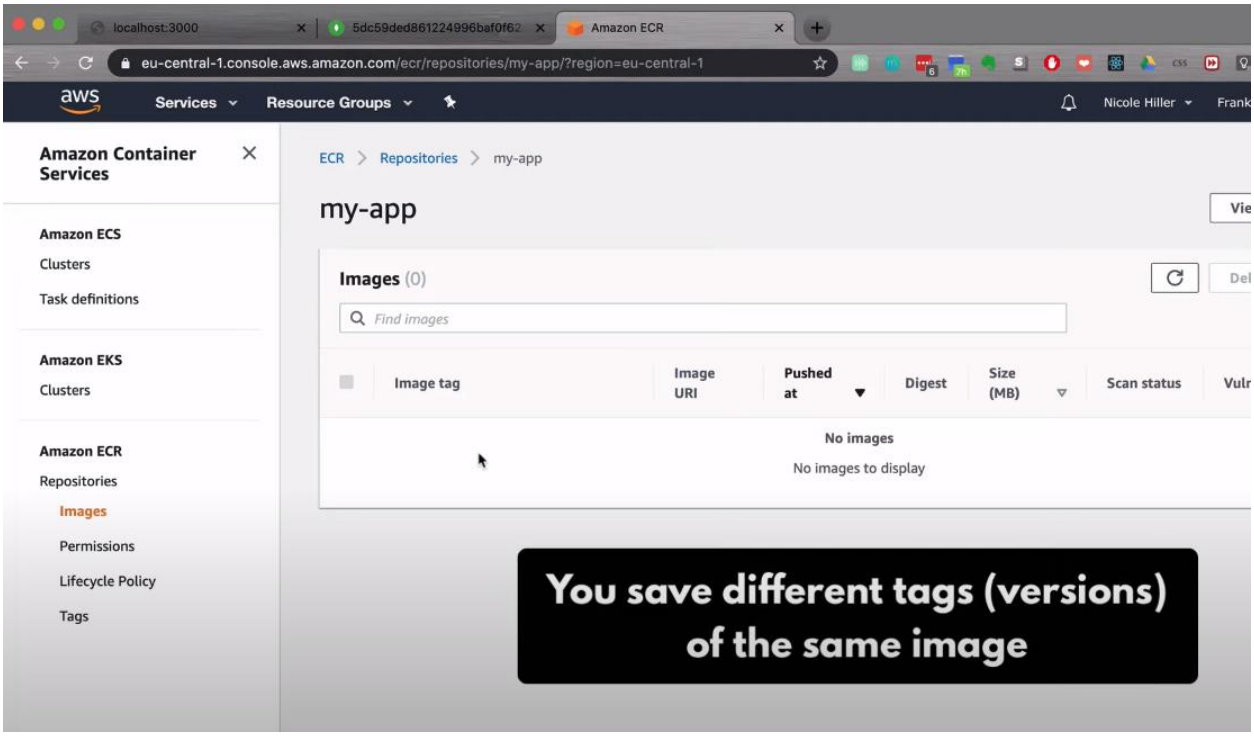
- Here the highlighted thing is the domain of the aws.
- My-app is the name of the repository which is the same as the image name.
- Click on **create repository.**



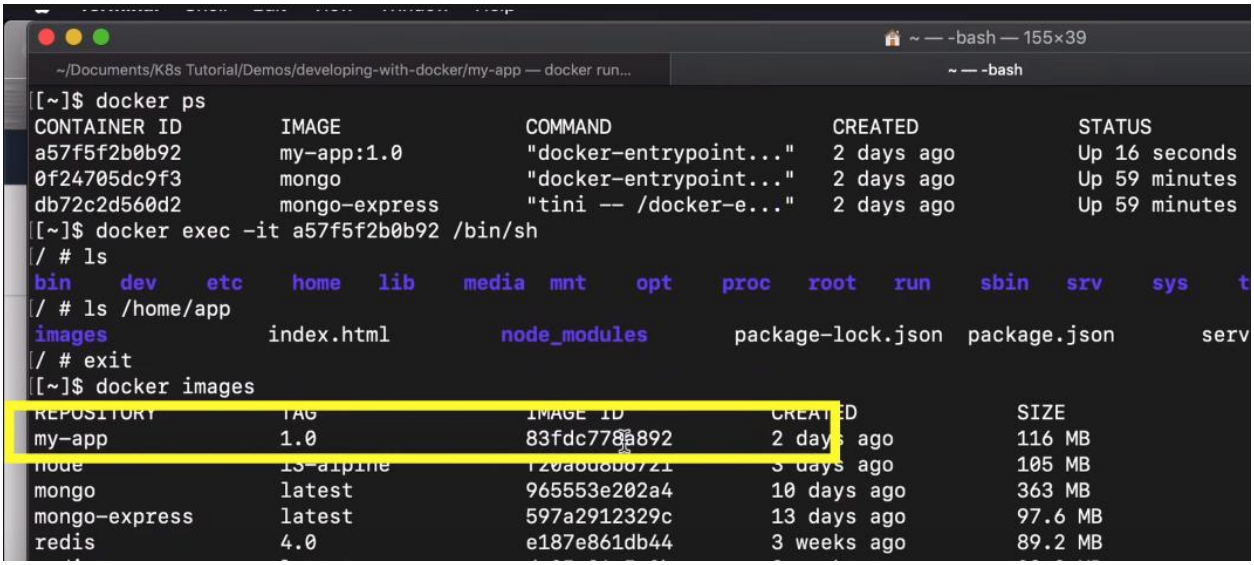
- One thing to note about **Amazon Elastic Container Repository** is that we create **Repository per Image.**
- Here we don't have repository to push multiple images of different applications.
- Rather each image has its own repository.
- If we go into that repository it will be empty for now.

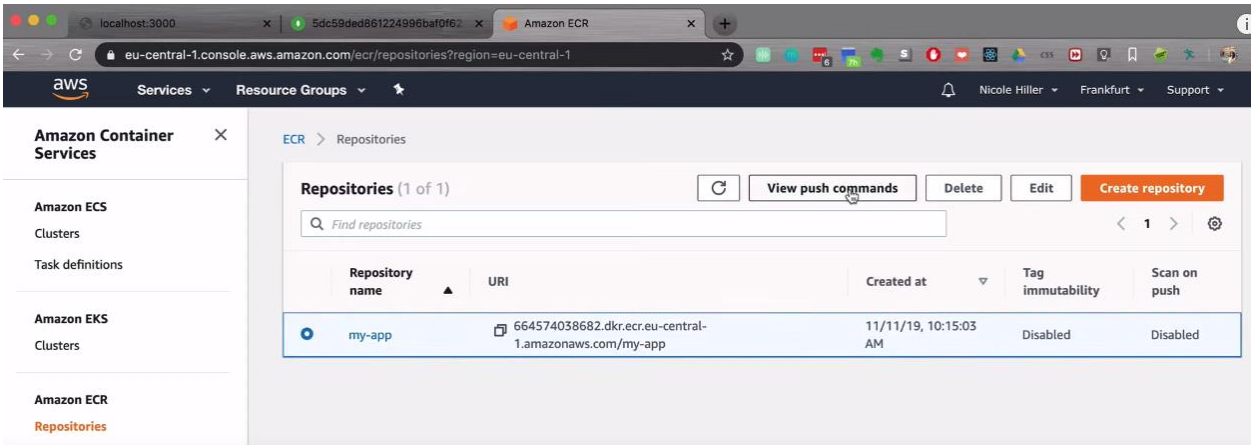


- What we store inside the repository are the different tags or versions of the same image.

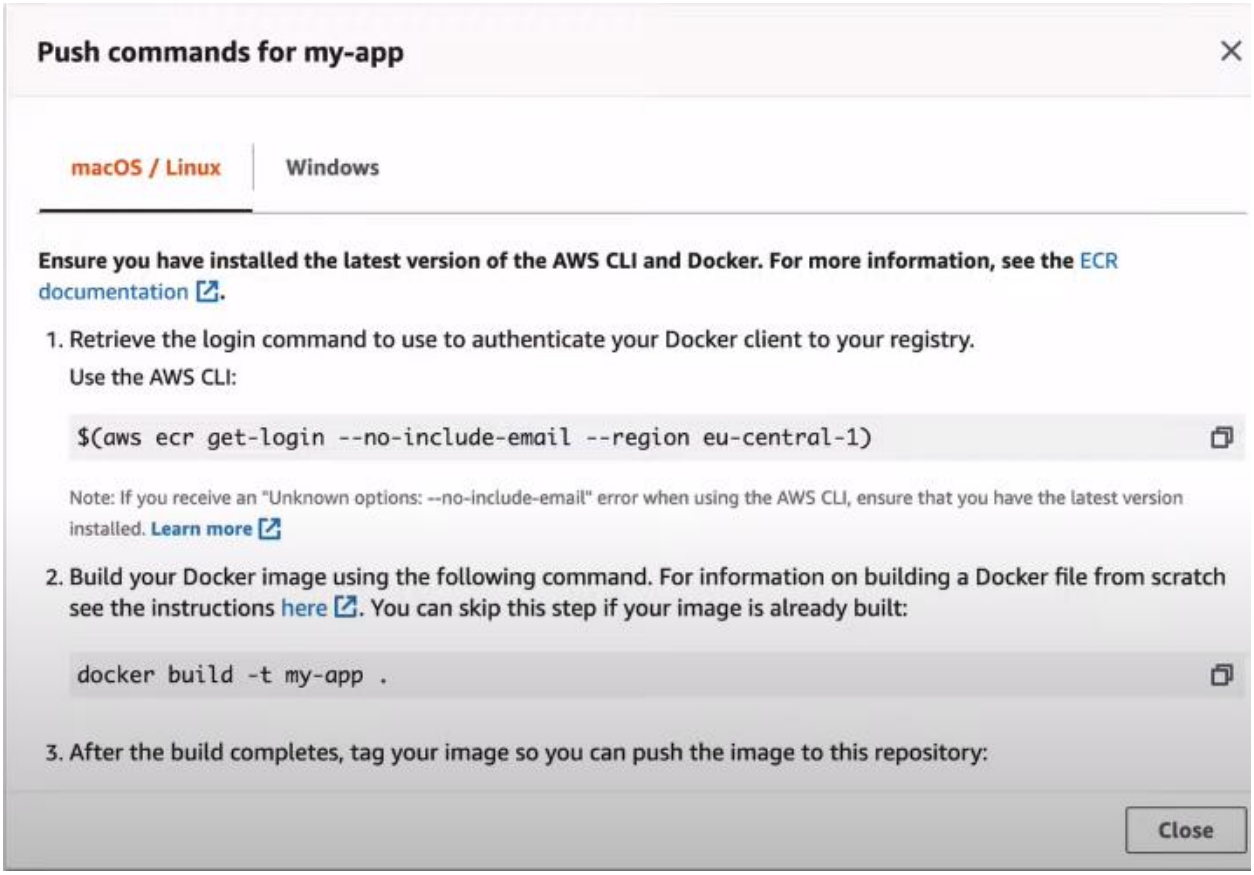


- Lets see how we can push the image we have locally into the docker repository in the AWS.
- Check the images we have in the local machine using **docker images.**
- We want to push the below image into that repository.

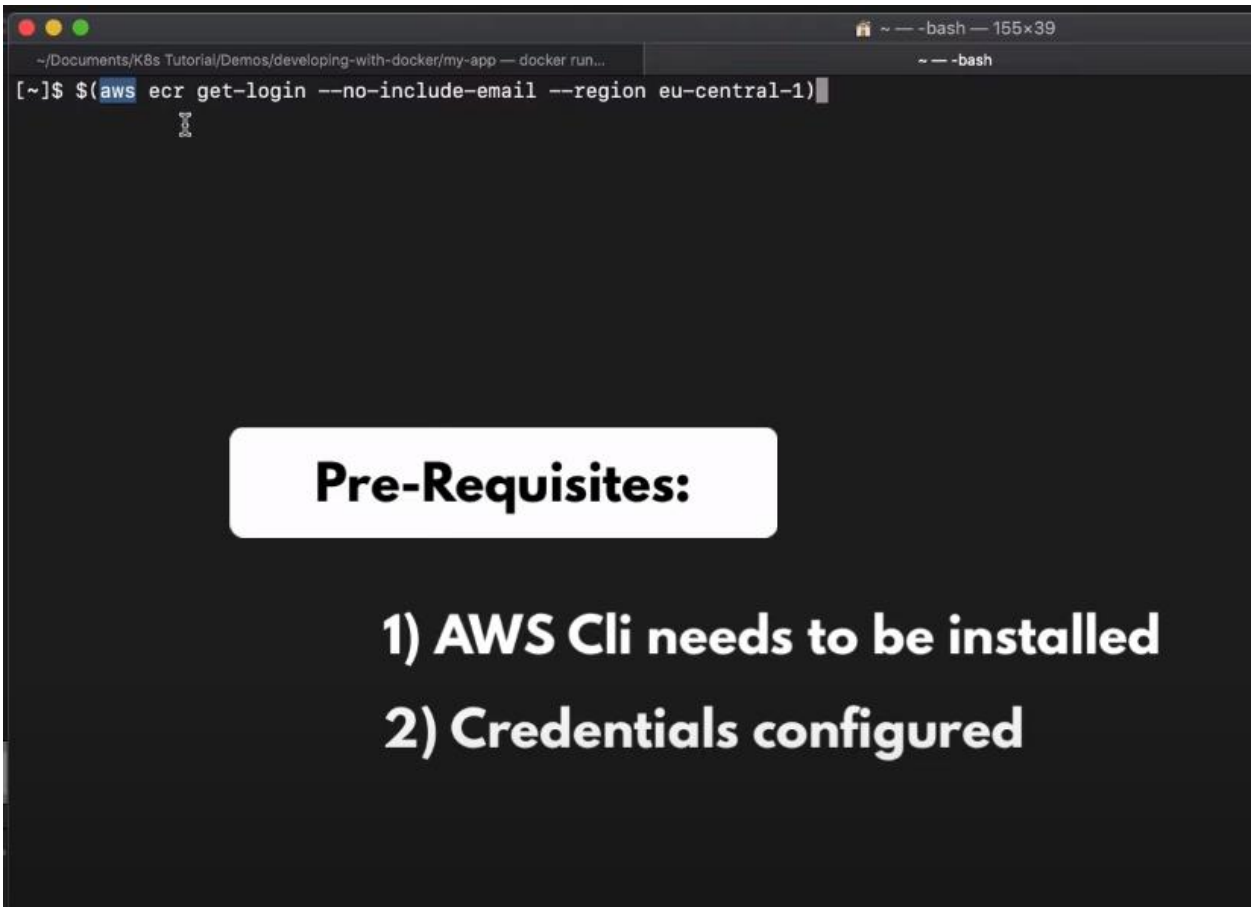




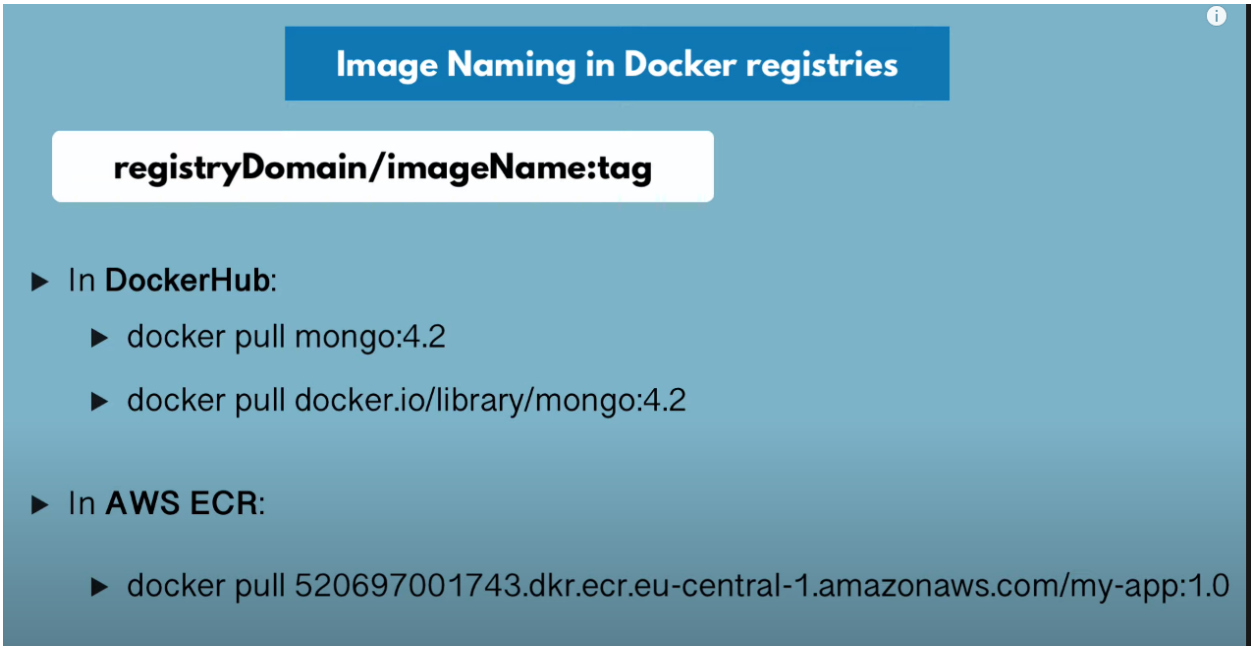
- Here, we can do it by clicking on **view push commands.**
- To push docker image into that private repository, we have to login to that private repository which means if we are pushing the docker image from the local environment or local laptop we have to tell the private repository hey I have access to it and I have the credentials.
- If docker image is pushed from the Jenkins server, we have to give Jenkins repository to login to that repository.



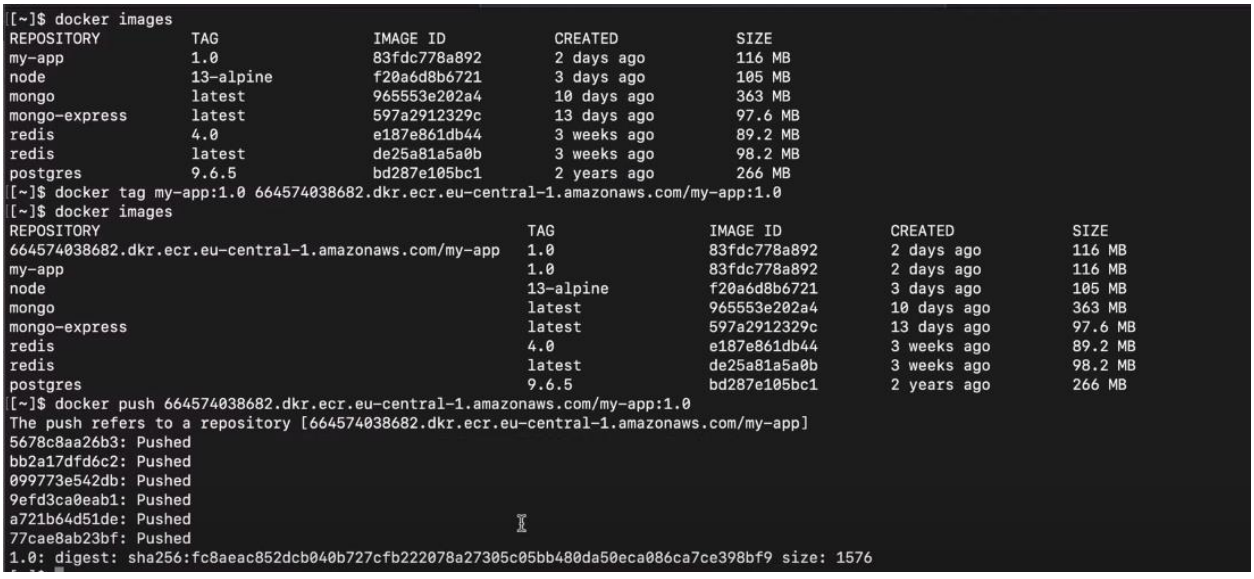
- Here it is the command for **docker login.**
- I am going to execute the login command for docker repository. In the background, it uses docker login to authenticate.
- So, for this we need to have **aws cli installed** and the **credentials need to be configured.**



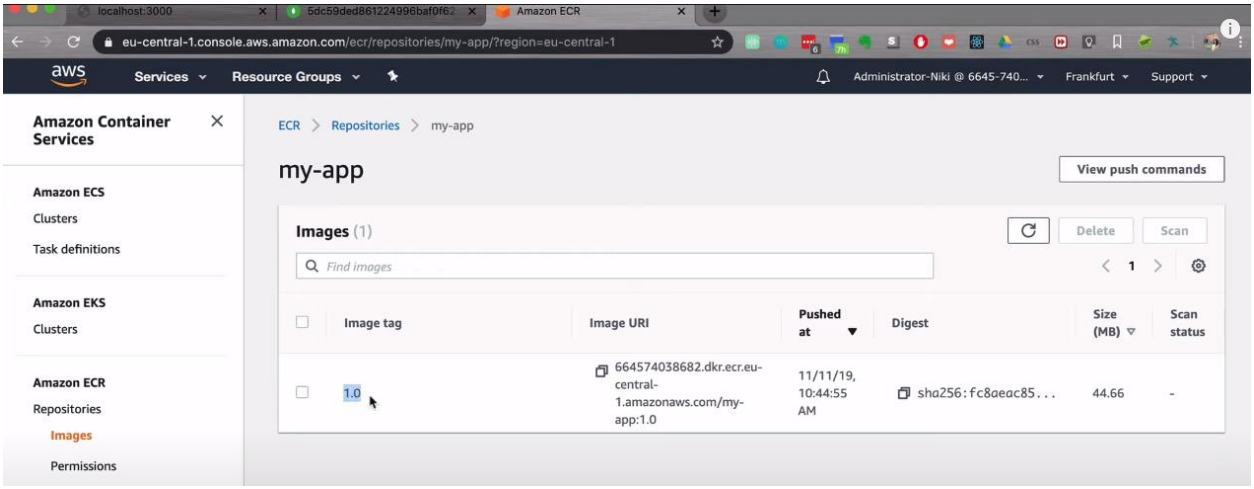
- This is the naming in docker registries which is **registryDomain/imageName:tag**



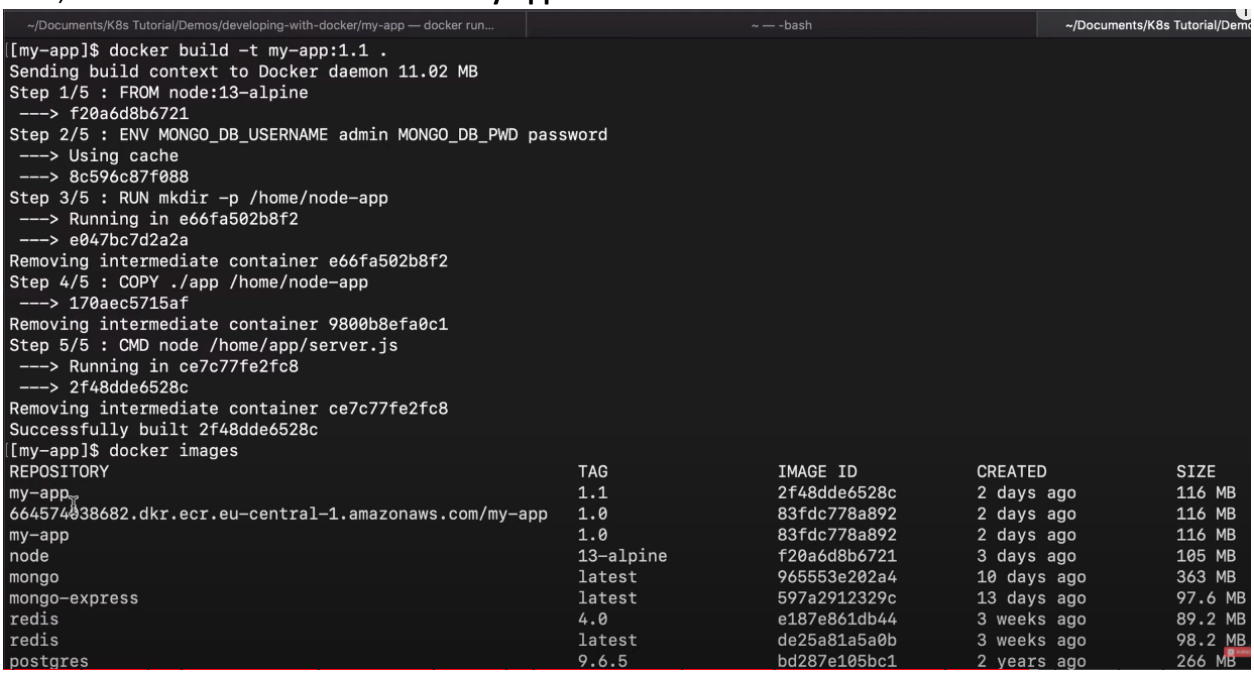
- So, from our local laptop we can not simply tell **docker push my-app:1.0** because docker does not know to which repository it need to push the image.
- By default, it will assume we are trying to push the image to dockerhub using the command **docker push my-app:1.0**
- to let docker know that we want to push this image to **aws repository** with the name **my-app** we have to **first tag the image.**
- We need to first include that information in the name of the image.
- Here we are tagging so as to mention the repository to which the image need to be pushed.



- Here, we are tagging the **my-app** image with the repository name and then a new image with that new name will be created, in this case that is **664574038682.dkr.ecr.eu-central-1.amazon.com/my-app**
- Now, we will push that image using **docker push 664574038682.dkr.ecr.eu-central-1.amazon.com/my-app**
- Now, it begins to push layers of that image into the repository.
- Now, we will be able to see that image in the **aws repository.**

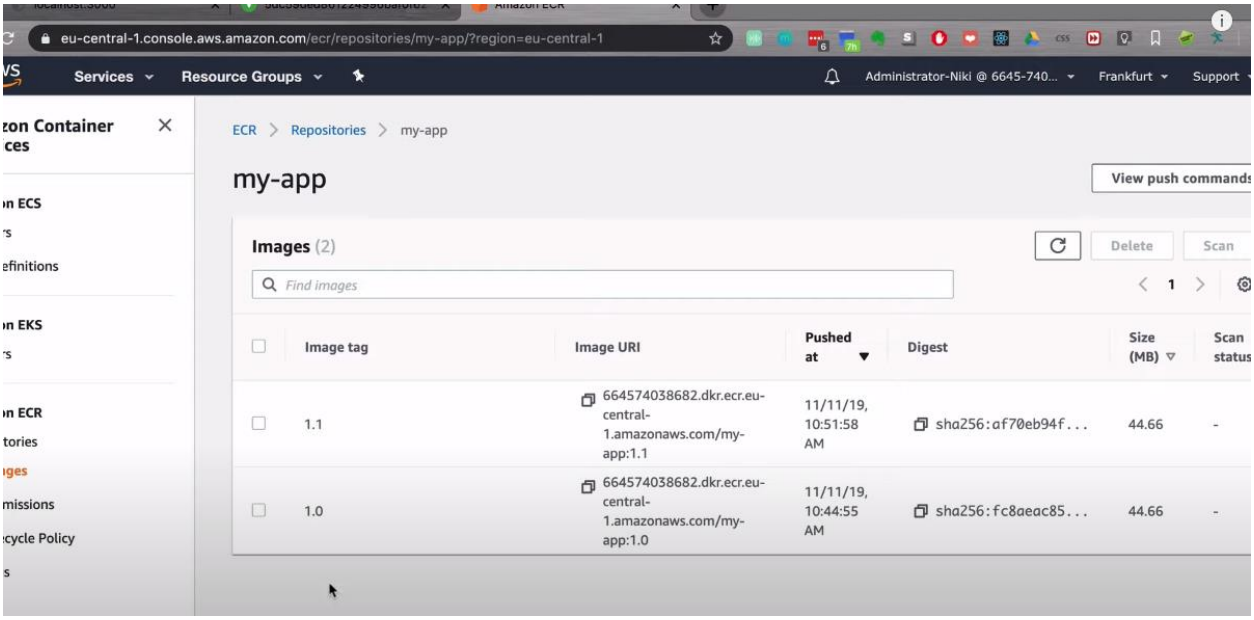


- Now let us make some changes to the App , rebuild it and push a new version to the AWS repo.
- Now, we have second version which is **my-app:1.1**

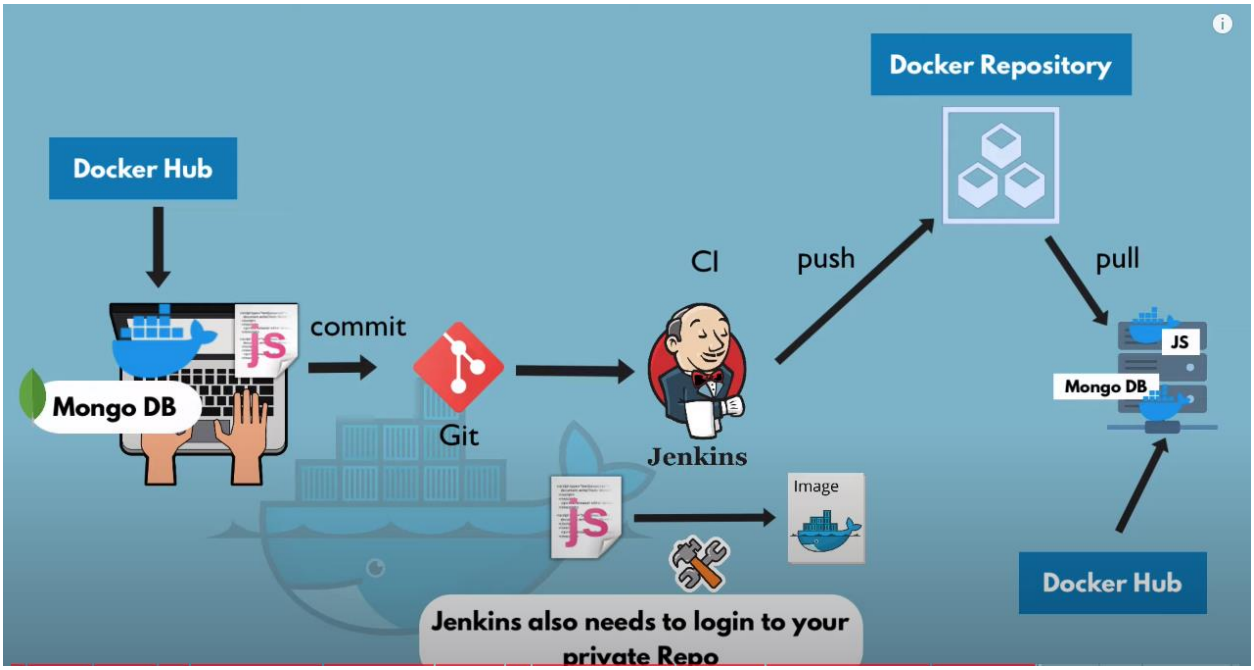


```
[my-app]$ docker tag my-app:1.1 664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app:1.1
[my-app]$ docker images
REPOSITORY                                                  TAG                 IMAGE ID            CREATED
664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app      1.1                 2f48dde6528c        2 days ago
my-app                                                      1.1                 2f48dde6528c        2 days ago
664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app      1.0                 83fdc778a892        2 days ago
my-app                                                      1.0                 83fdc778a892        2 days ago
node                                                        13-alpine           f20a6d8b6721        3 days ago
mongo                                                       latest              965553e202a4        10 days ago
mongo-express                                               latest              597a2912329c        13 days ago
redis                                                       4.0                 e187e861db44        3 weeks ago
redis                                                       latest              de25a81a5a0b        3 weeks ago
postgres                                                    9.6.5               bd287e105bc1        2 years ago
[my-app]$ docker push 664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app:1.1
```

- Now, we have pushed two images with different tags or versions into the docker repository and we can see it in the below, it is pretty useful when we want to test the images with different versions.

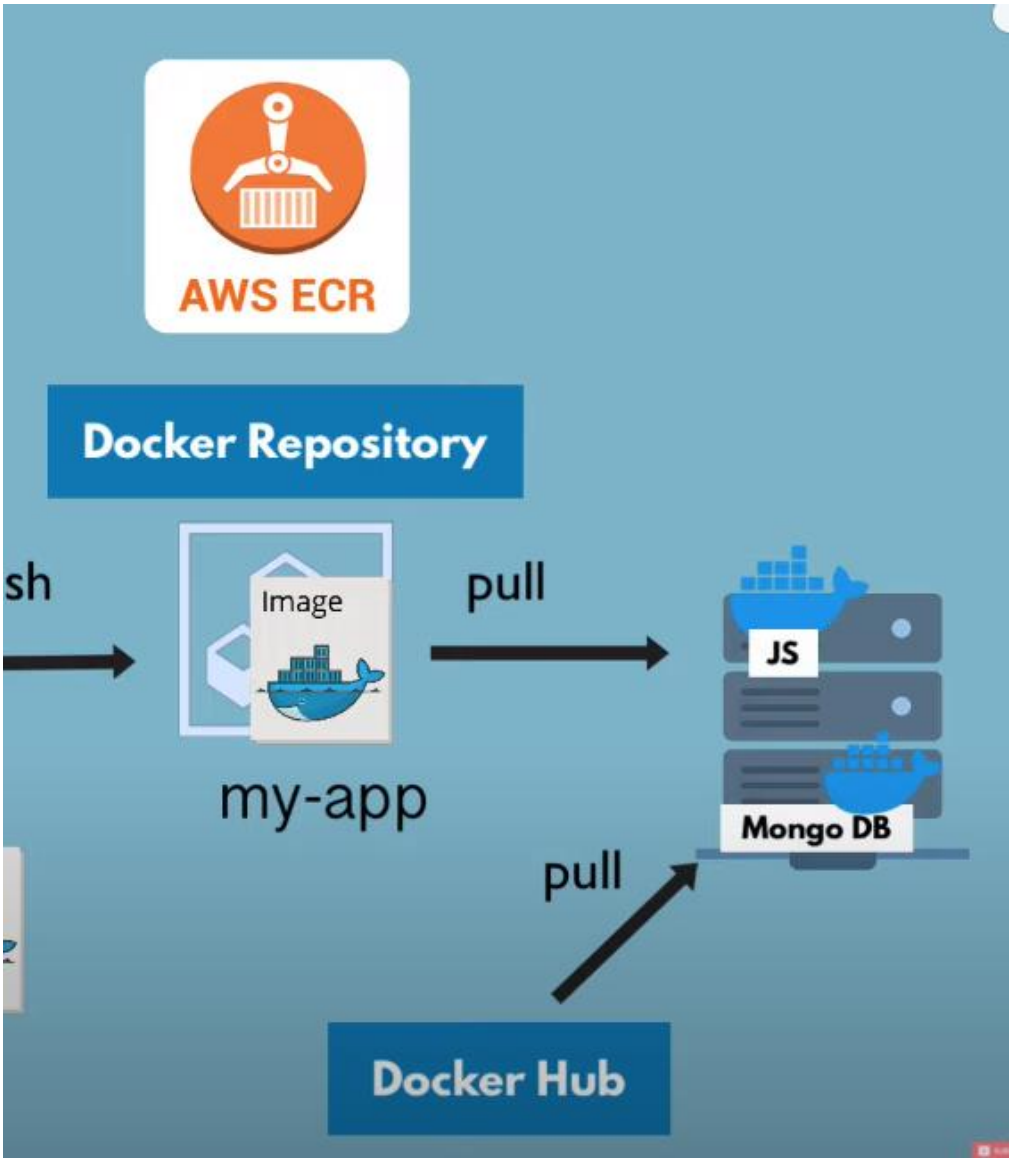


- In AWS, each repository has a capacity of holding 1000 images.
- For example, in **my-app** we can have 1000 versions of the same image which is **my-app.**



- Now, we will see how to use the image lying in the docker repository , we will see how to do it in local environment but it will be the same if we use the development server or any other environment.

- Here, we will see how to deploy the application that we built into the docker image.
- After we package our application and save it in a private repository, we need to deploy it on development server.
- We are going to use **docker-compose** to deploy that application.
- Now we will log into the development server and then we want to run the docker image of the application we pushed into the aws private repository and the docker images of both mongodb and mongo express.



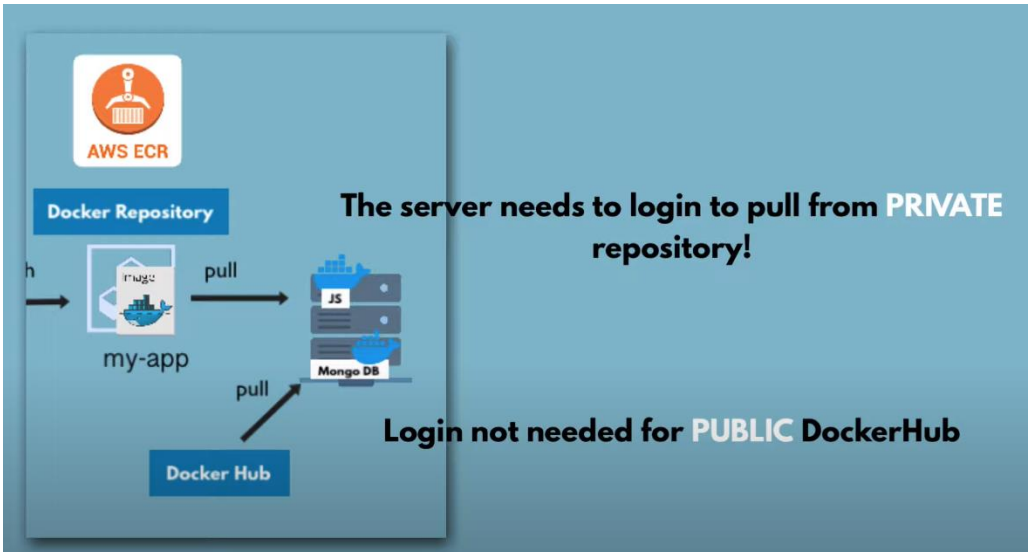
- To start up the application on the development server, we need all that containers that make up that application container.
- Here , we have mentioned the image of the **my-app** pulled from the docker private repository.

mongo.yaml

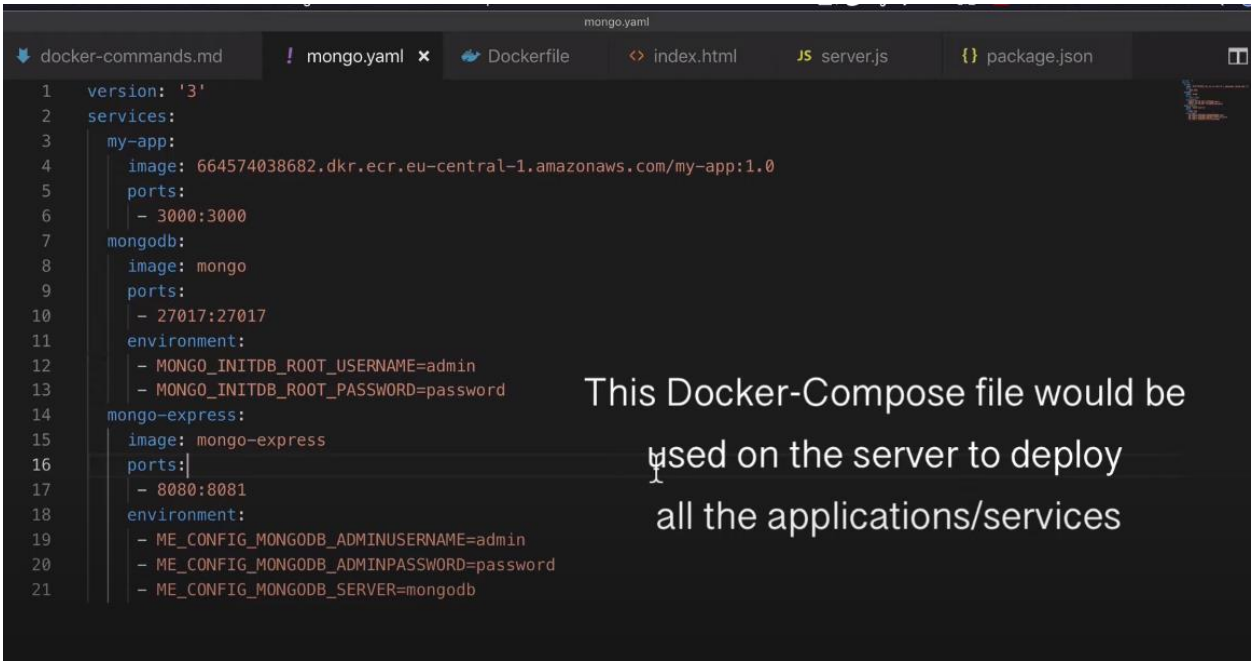
docker-commands.md | mongo.yaml | Dockerfile | index.html | server.j

```yaml
version: '3'
services:
  my-app:
    image: 664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app:1.0
  mongodb:
    image: mongo
    ports:
    - 27017:27017
    environment:
    - MONGO_INITDB_ROOT_USERNAME=admin
    - MONGO_INITDB_ROOT_PASSWORD=password
  mongo-express:
    image: mongo-express
    ports:
    - 8080:8081
    environment:
    - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
    - ME_CONFIG_MONGODB_ADMINPASSWORD=password
    - ME_CONFIG_MONGODB_SERVER=mongodb
```

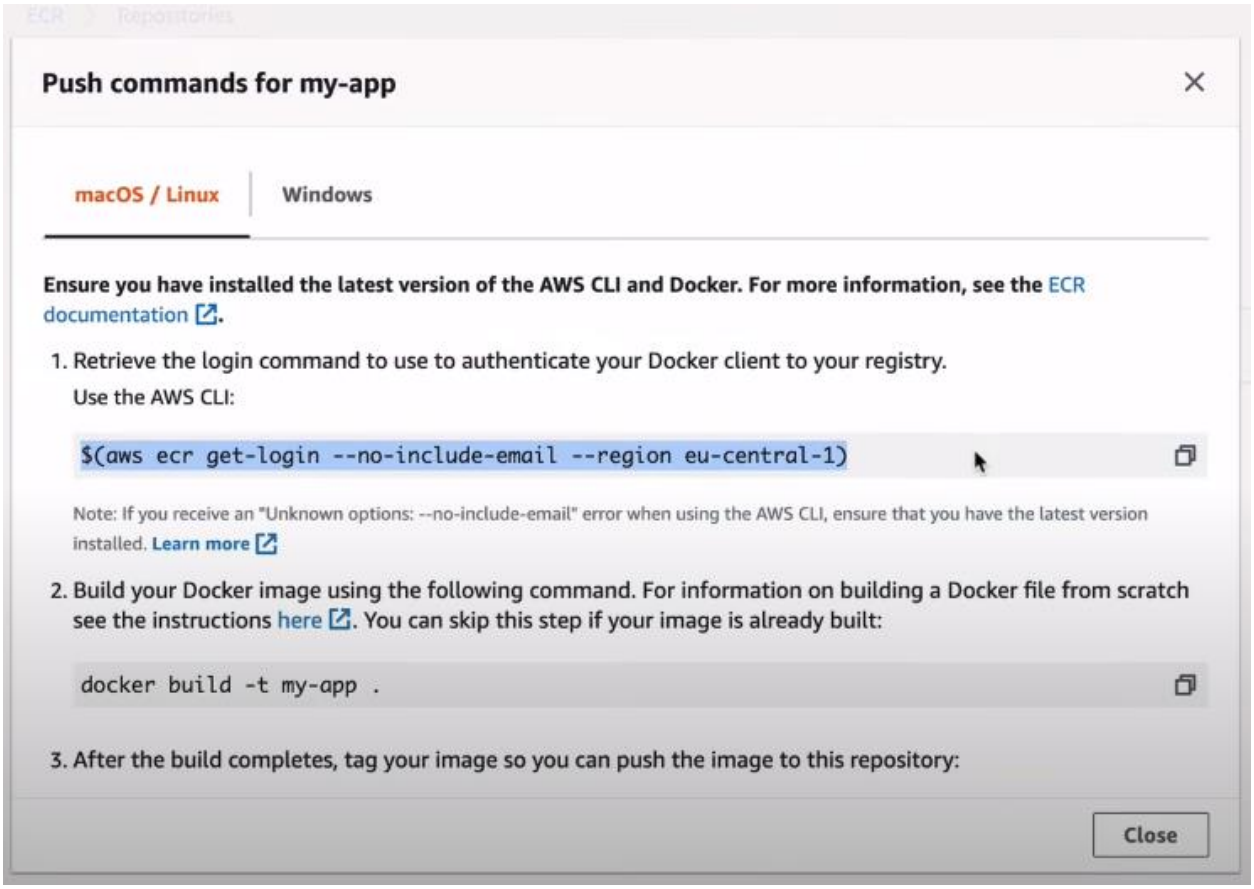
- For the docker-compose to pull the **664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app:1.0** image, then the environment where we are executing the docker-compose file has to be logged into a docker repository.
- Here as the development server has to pull the image from the repository, what we need to do on the development server is actually do a login before we execute the docker-compose.
- Ofcourse we don't a docker login for the backup, those mongo images will be pulled freely from the public dockerhub.



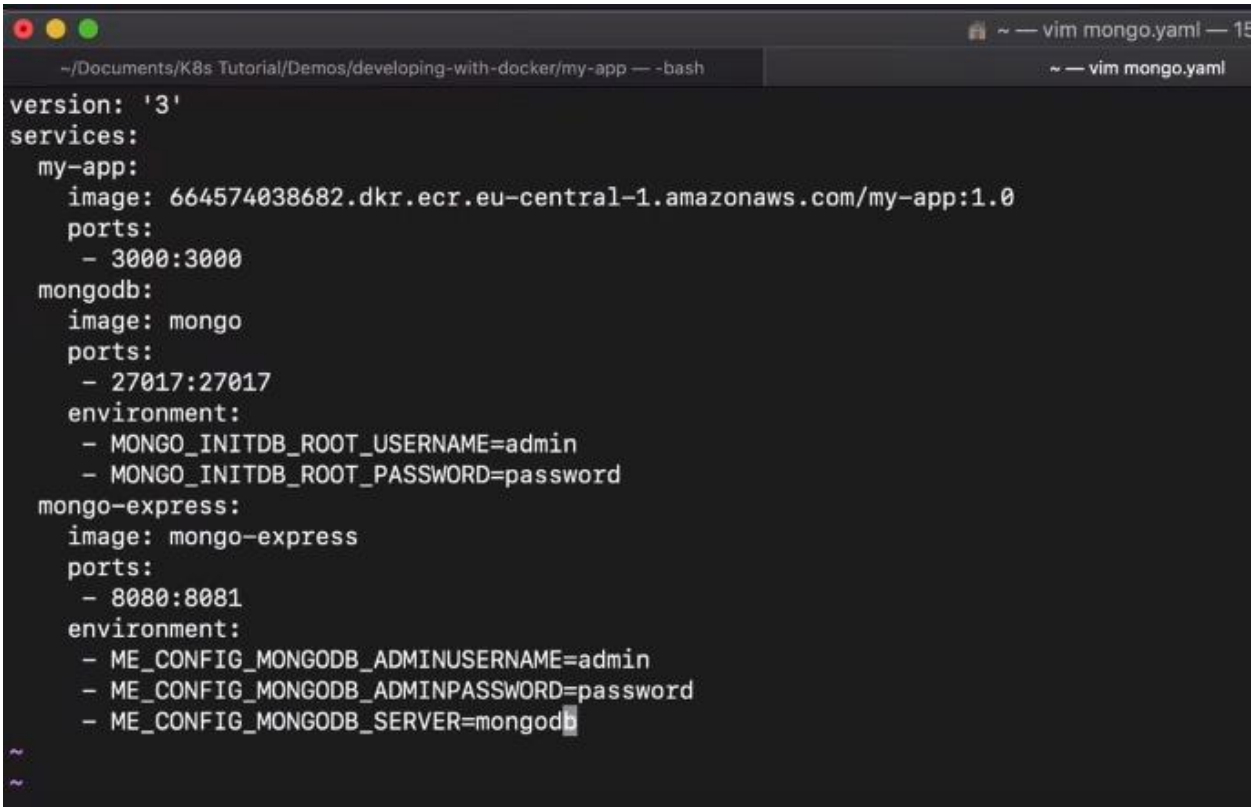
- The next thing we need to configure are the ports , **3000:3000** which means mapping port 3000 of the host to the port 3000 of the container.



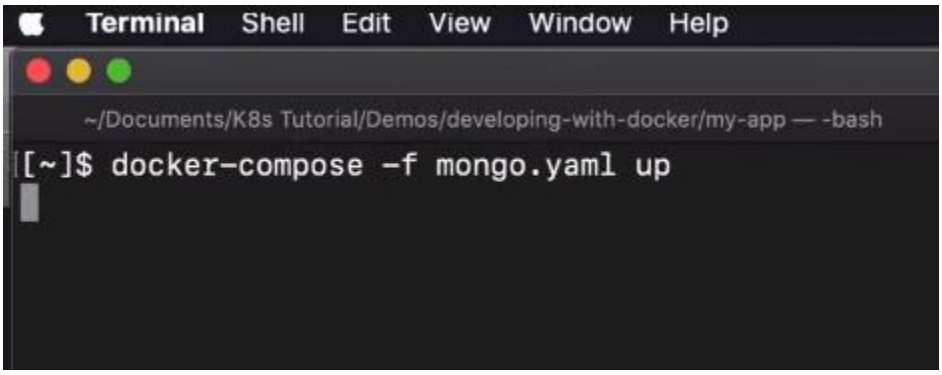
- Again if we are trying to simulate the development server, the first step would be to us the command **docker login**
- In this case , we have below command for aws.



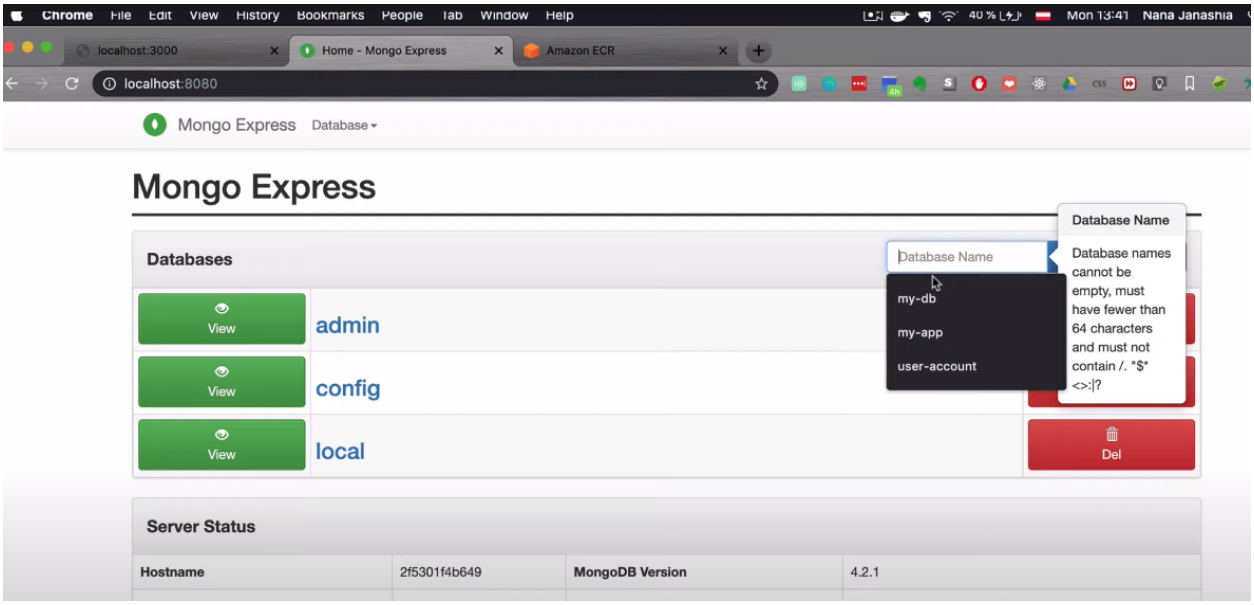
- This is the command for logging into aws repository.
- The next step would be to have docker-compose file to be available on the development server.
- For this , we will first create a mongo.yaml file.
- We will copy the content from the docker-compose file of our application and paste below the vim command.



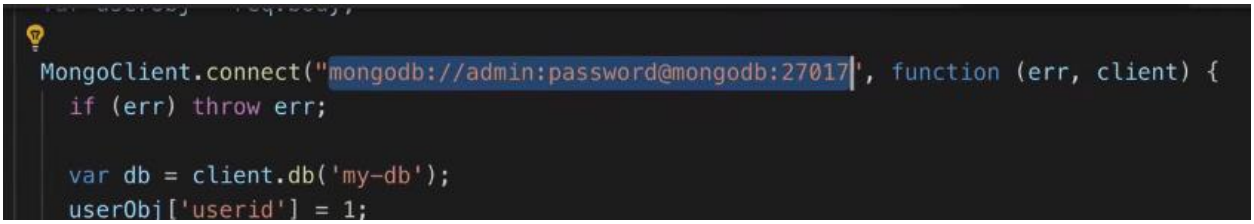
- Now we can start all the three containers at the same time using the command which **docker-compose -f mongo.yaml up**



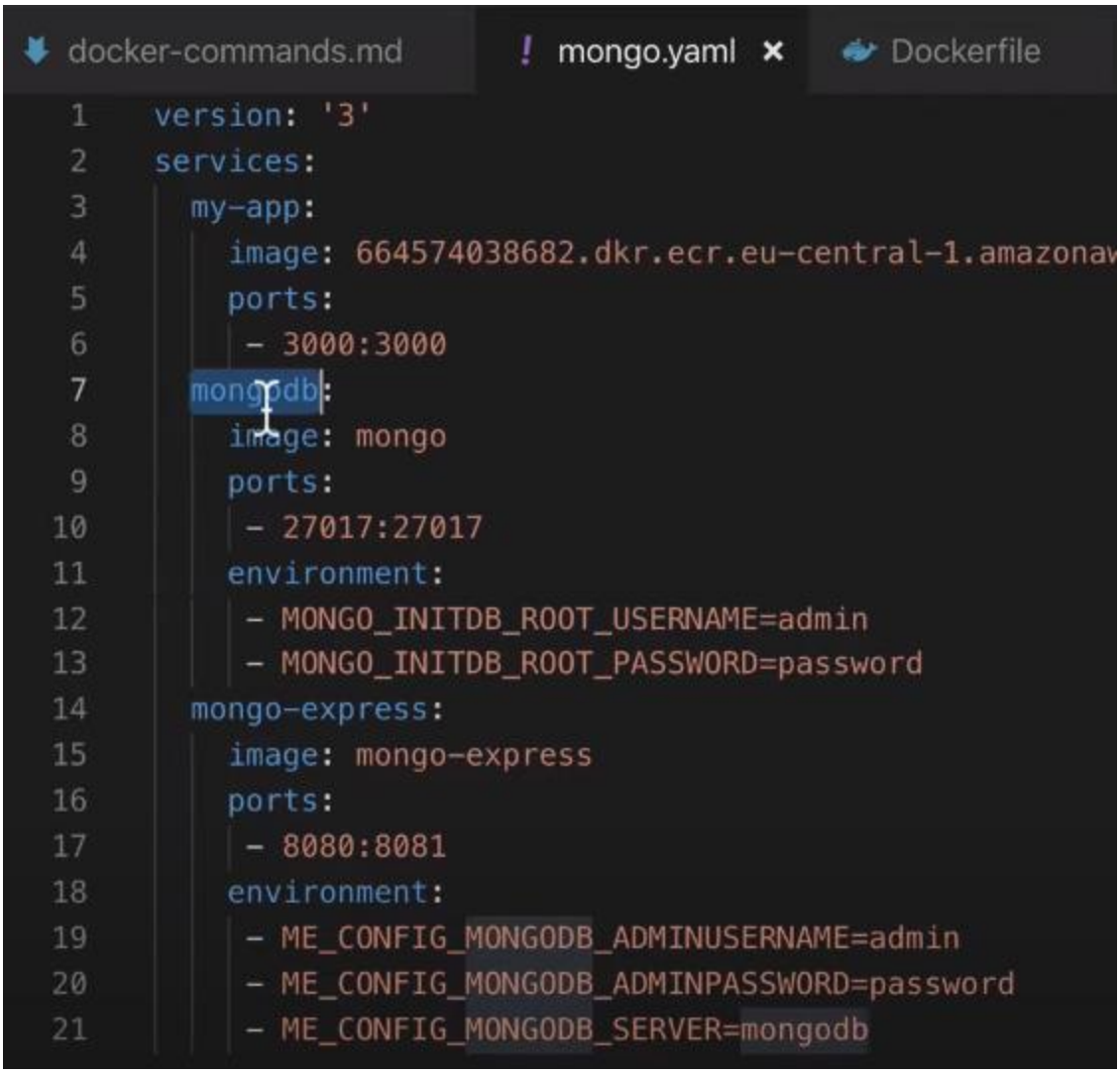
- Here we can see that the database is lost everytime when we restarted the container.



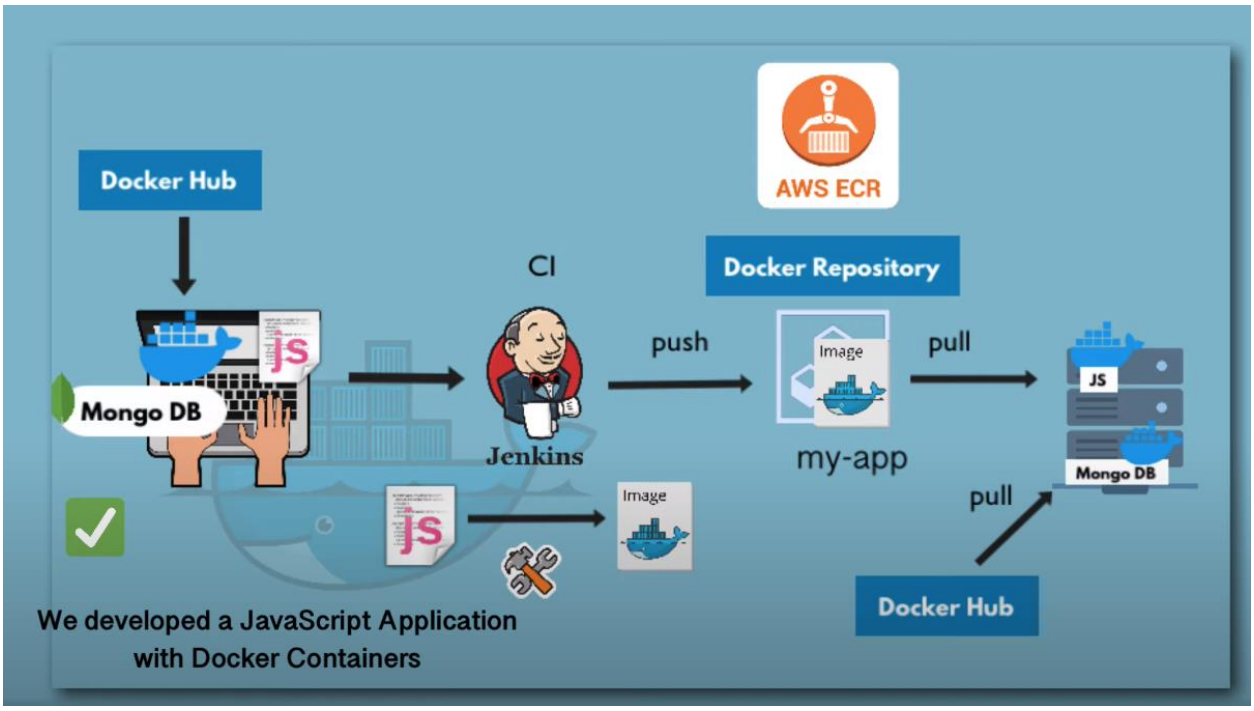
- This is how we can learn how to preserve the data of the database, when the container restarts using docker volumes.
- There is actually one thing to change in the code when trying to connect node.js with the mongo db database.

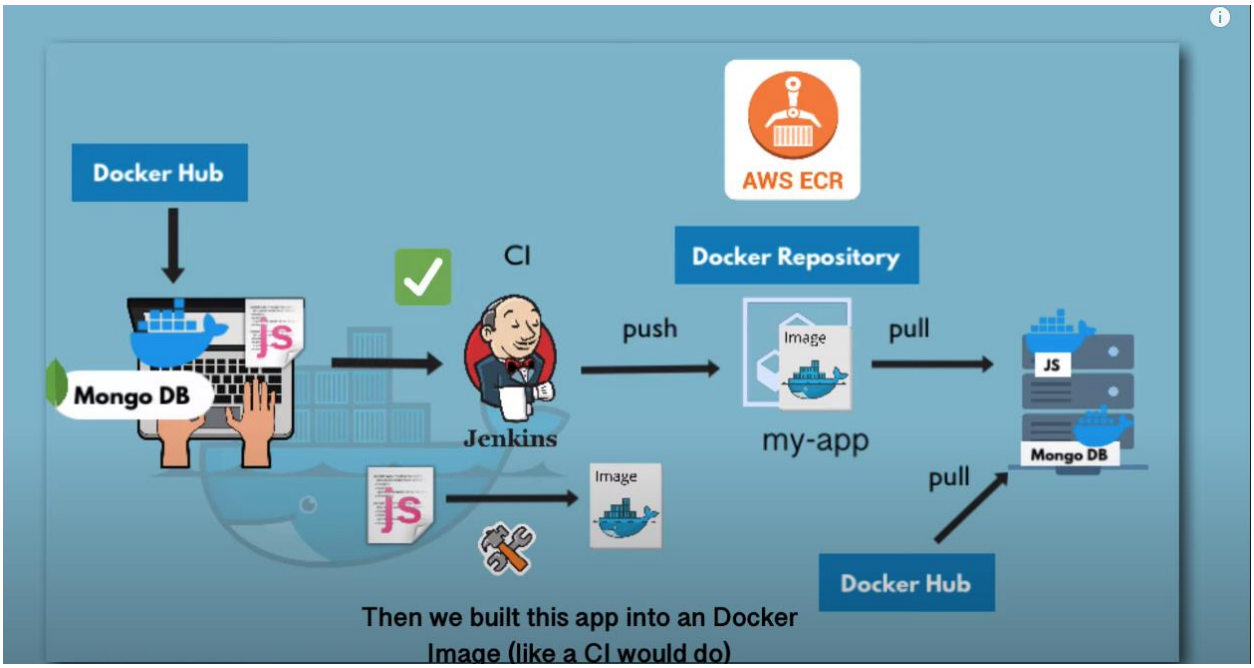


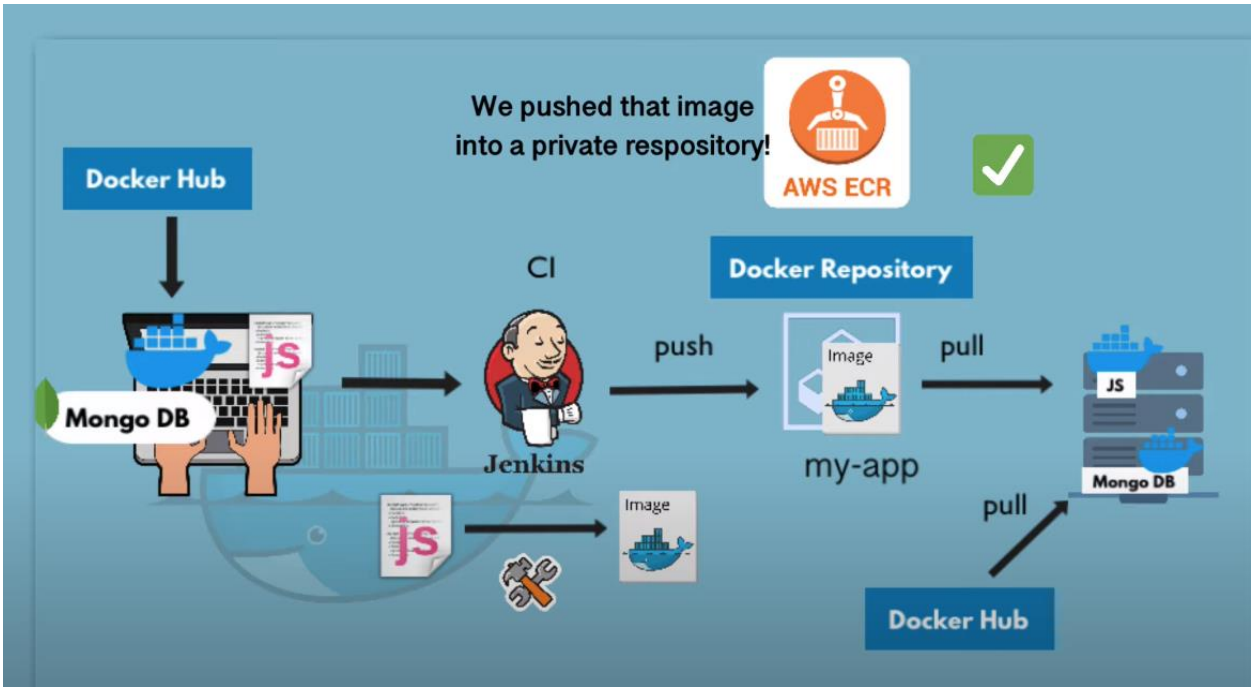
- These are the handlers with which we can connect node js to the mongo db.
- Instead of **mongodb:27017 ,** it was local host before.
- Because this mongodb refers to the service or the container we specify in the below image.

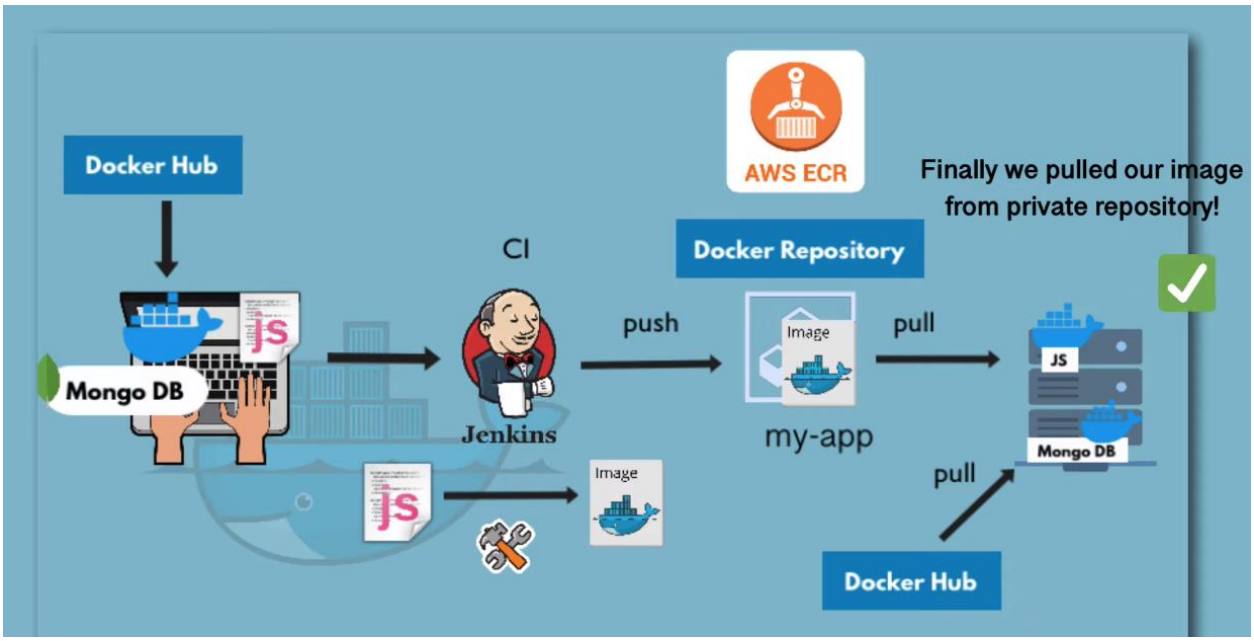


- This actually leads back to docker network and how it will take care of it.
- So here , nodejs application which is **my-app** will be able to connect to mongodb database using the service name which is **mongodb** and no need to mention the entire url in the code because everything is configured in the docker-compose file.

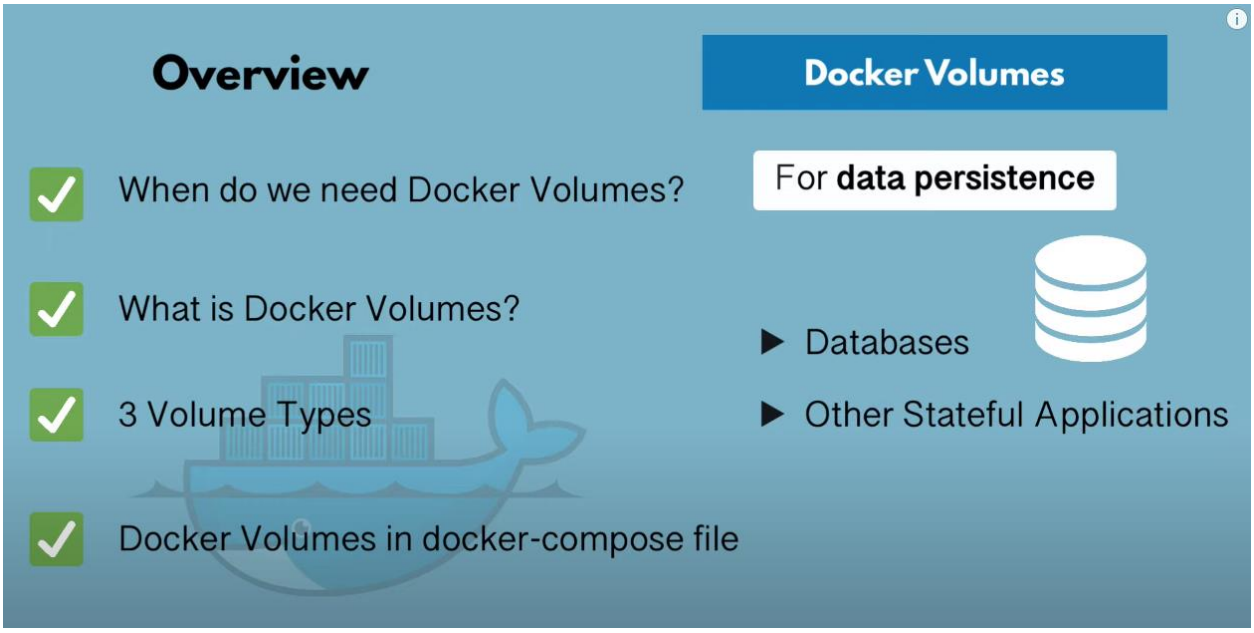

Docker Hub
AWS ECR
CI
Docker Repository
Mongo DB
Jenkins
push
Image
pull
my-app
JS
Mongo DB
Image
pull
Docker Hub
We developed a JavaScript Application
with Docker Containers


Docker Hub
AWS ECR
CI
Docker Repository
Mongo DB
Jenkins
push
Image
pull
my-app
JS
Mongo DB
Image
pull
Docker Hub
Then we built this app into an Docker
Image (like a CI would do)

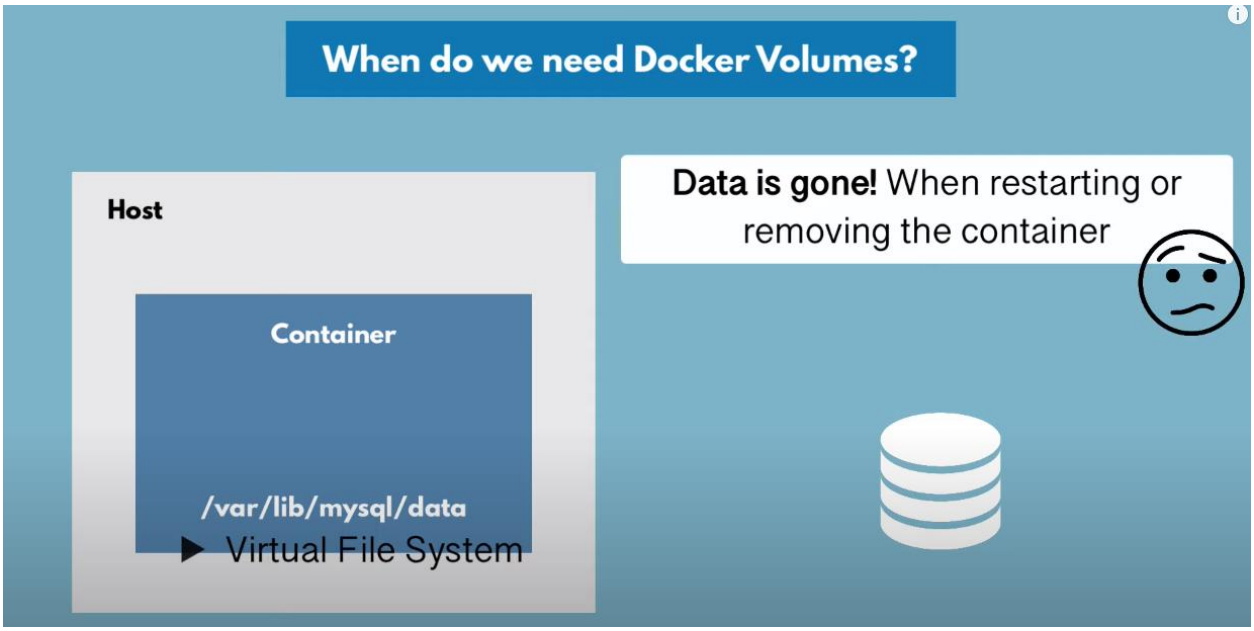

We pushed that image into a private respository!
AWS ECR
Docker Hub
CI
Docker Repository
Mongo DB
Jenkins
push
Image
pull
my-app
JS
Mongo DB
Image
pull
Docker Hub


AWS ECR
Finally we pulled our image from private repository!
Docker Hub
CI
Docker Repository
Mongo DB
Jenkins
push
Image
pull
my-app
JS
Mongo DB
Image
pull
Docker Hub

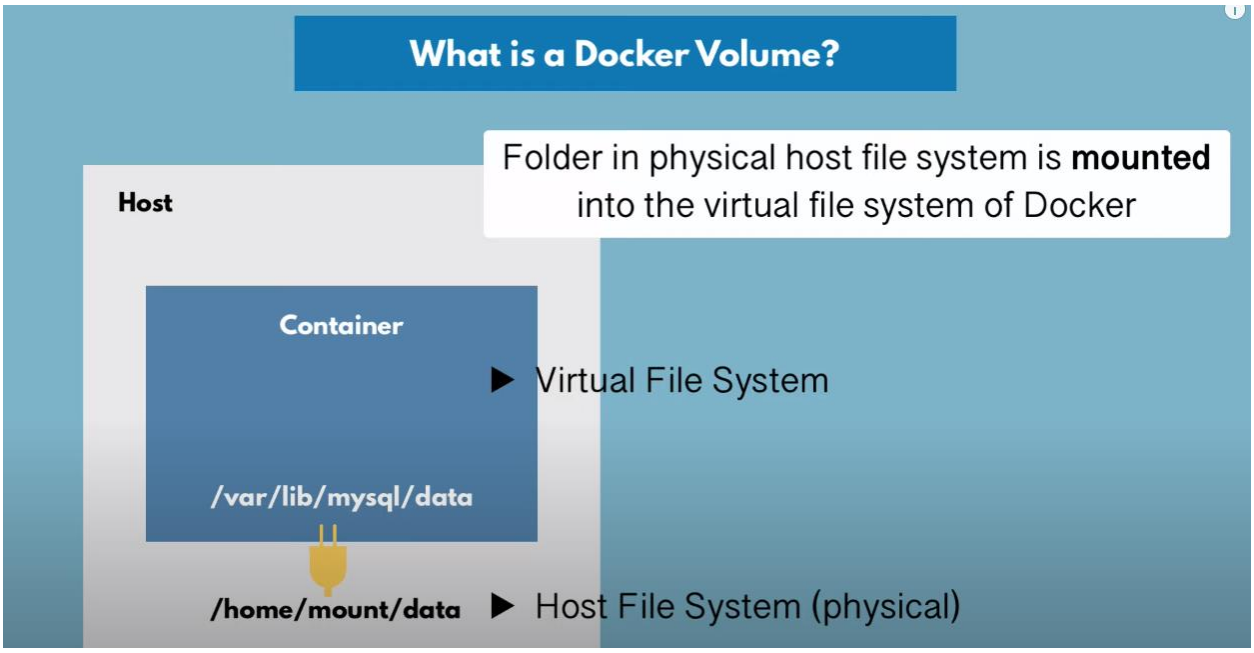
PERSITING DATA WITH VOLUMES



- As we know container runs on the host and we know it has a virtual file system where the data is usually stored.
- Think for example we are using the database container, but here there is no persistence of data, if I were to remove the container or stop it and restart the container then the data is gone and the container starts from the fresh state which is obviously not practical and because I want to save the changes that my application makes in the database.

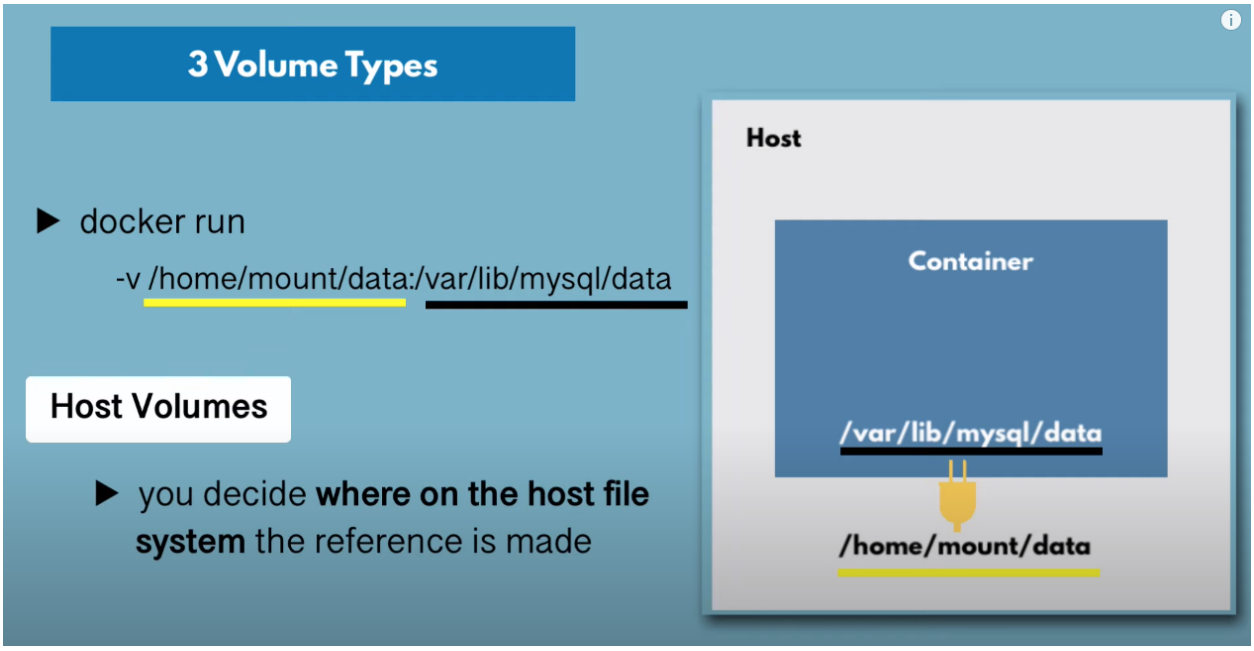


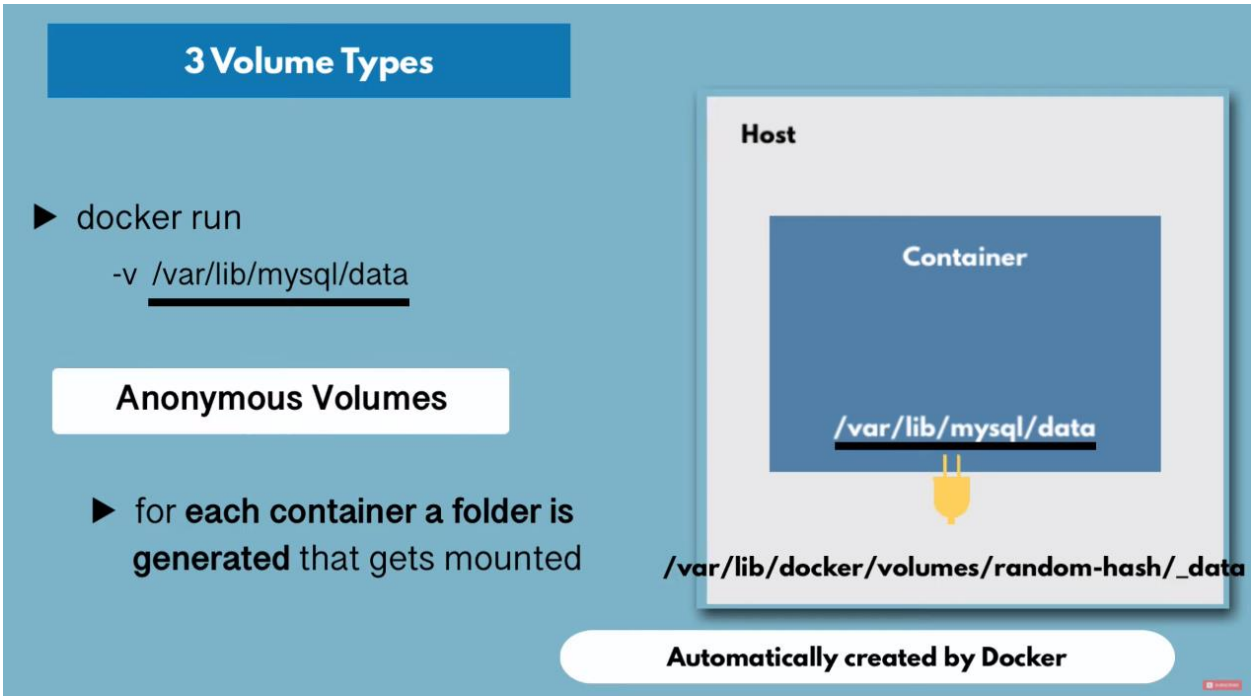
- This is where we use docker volumes.
- On a host , we have physical file system.
- Docker volumes will plugin the folder or directory from the physical file system of the host to the file system of the container.



- When a container writes to its file system then it will get automatically replicated on the host file system directory and vice versa.
- So if we change something in the host file system it appears in the container as well.
- So when a container restarts and even if it starts from a fresh state, it gets data from the host file system because it is plugged to it.
- That's how data gets populated on the container, everytime we restart it.

- Now there are different types of docker volumes which means that there are different ways of creating them.



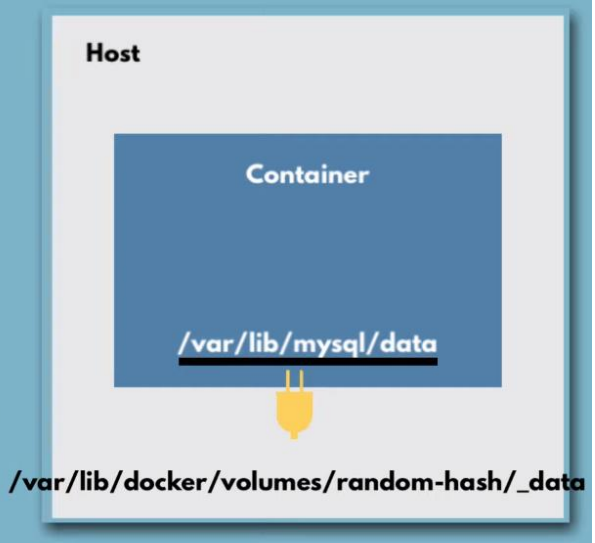


3 Volume Types

- docker run
  -v **name:**/var/lib/mysql/data

Named Volumes

- you can **reference** the volume **by name**
- should be used in production



Docker Volumes in docker-compose

Named Volume

mongo-docker-compose.yaml

```
version: '3'

services:

    mongodb:

        image: mongo

        ports:
         - 27017:27017

        volumes:
         - db-data:/var/lib/mysql/data

    mongo-express:

        image: mongo-express
        ...

volumes:
    db-data
```

techworld-js-docker-demo-app [~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js-docker-demo-app]

techworld-js-docker-demo-app 〉 docker-compose.yaml

Project

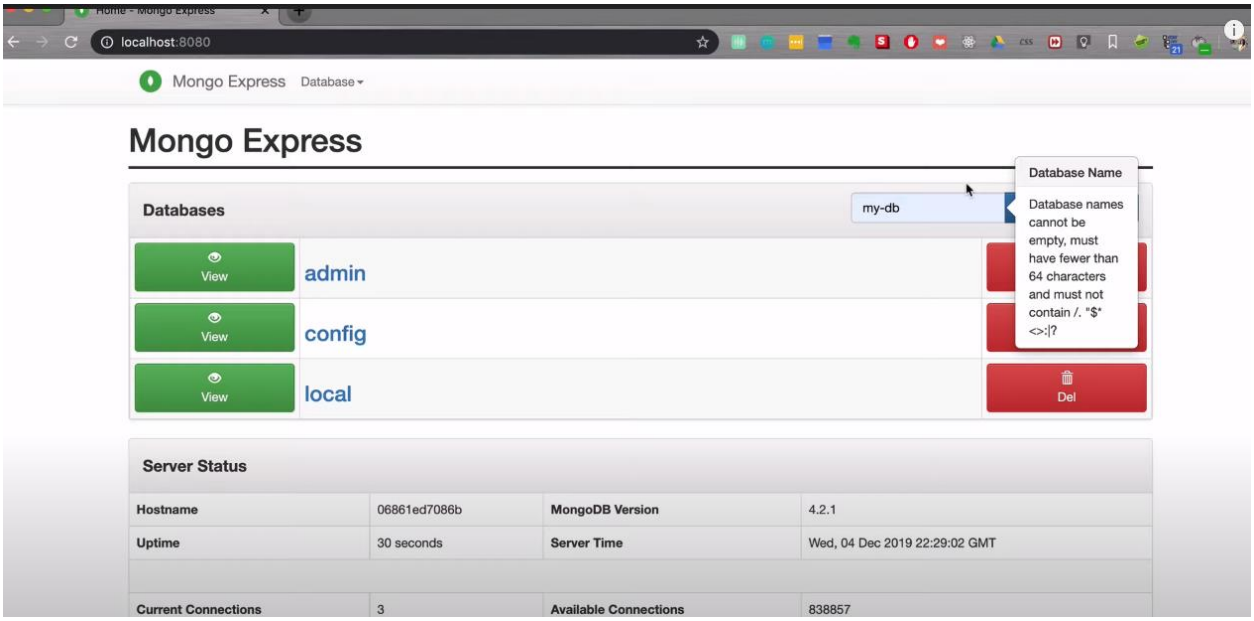
techworld-js-docker-demo-a
- .idea
- app
  - images
    - profile-1.jpg
    - profile-2.jpg
  - node_modules library ro
  - index.html
  - package.json
  - package-lock.json
  - server.js
- .gitignore
- docker-compose.yaml
- Dockerfile
- README.md
- External Libraries
- Scratches and Consoles

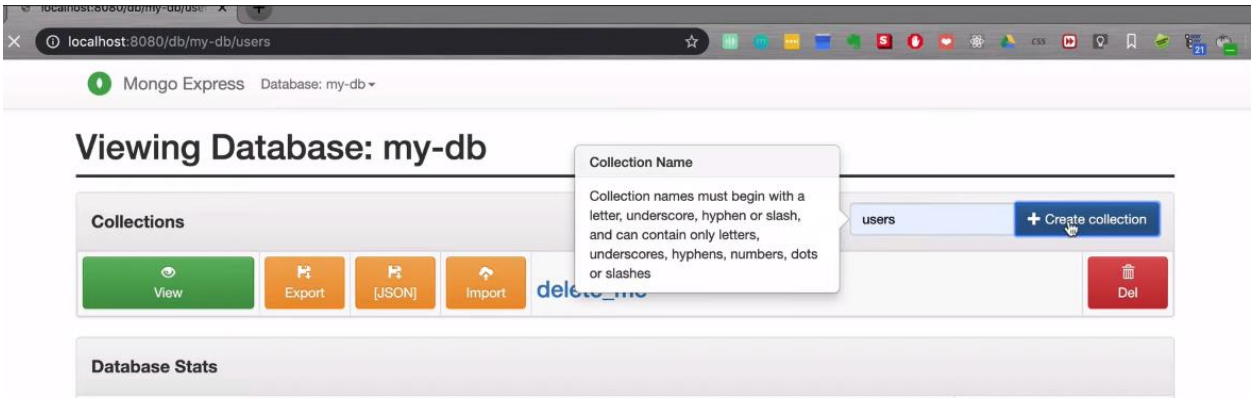
.gitignore | index.html | package.json | server.js | docker-compose.yaml | README.md

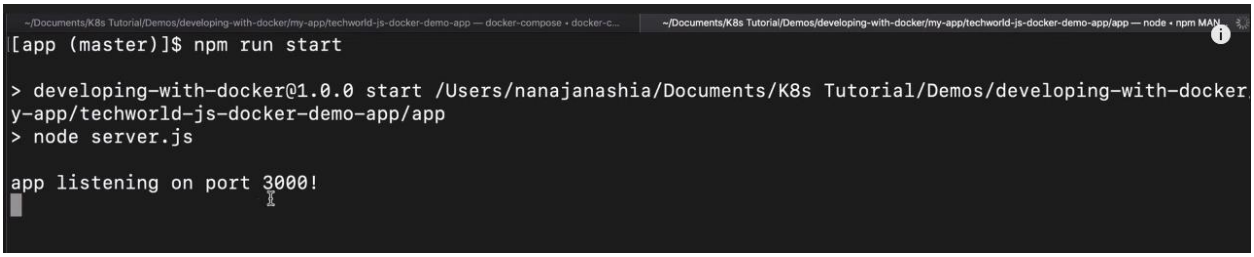
```yaml
version: '3'
services:
  # my-app:
    # image: ${docker-registry}/my-app:1.0
    # ports:
     # - 3000:3000
  mongodb:
    image: mongo
    ports:
     - 27017:27017
    environment:
     - MONGO_INITDB_ROOT_USERNAME=admin
     - MONGO_INITDB_ROOT_PASSWORD=password
  mongo-express:
    image: mongo-express
    ports:
     - 8080:8081
    environment:
     - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
     - ME_CONFIG_MONGODB_ADMINPASSWORD=password
     - ME_CONFIG_MONGODB_SERVER=mongodb
```

~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js-docker-demo-app — docker-compose • docker-c... | ~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js-docker-demo-app — -bash

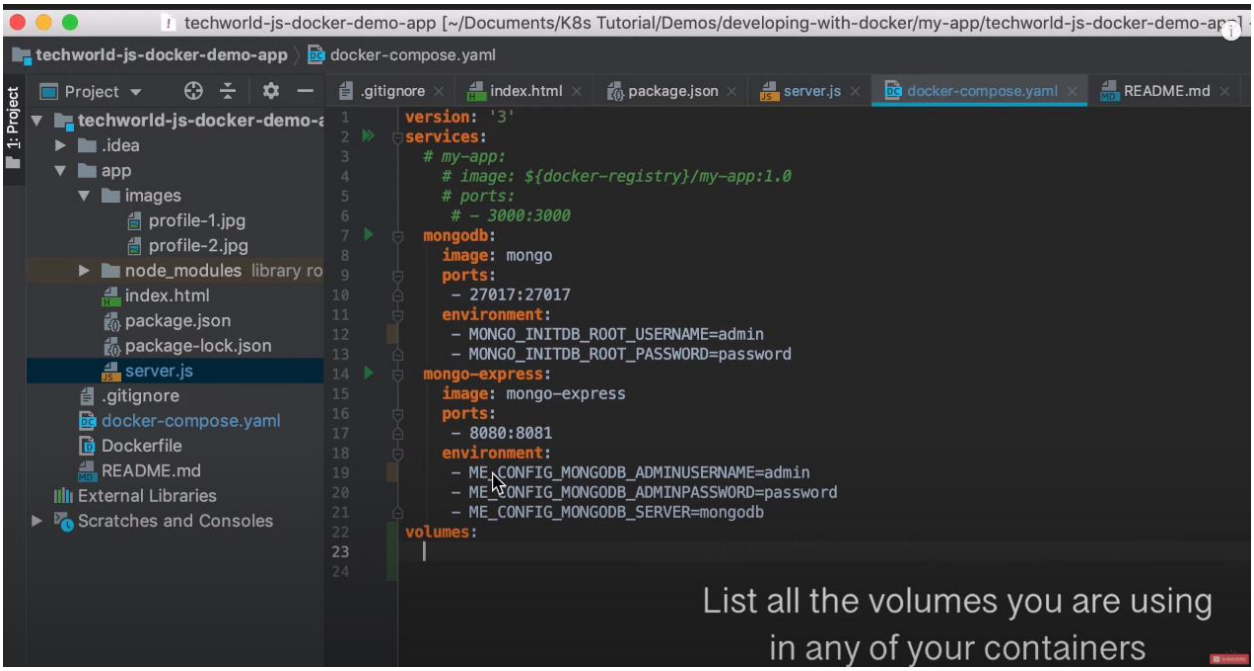
```
[techworld-js-docker-demo-app (master)]$ docker-compose -f docker-compose.yaml up
```

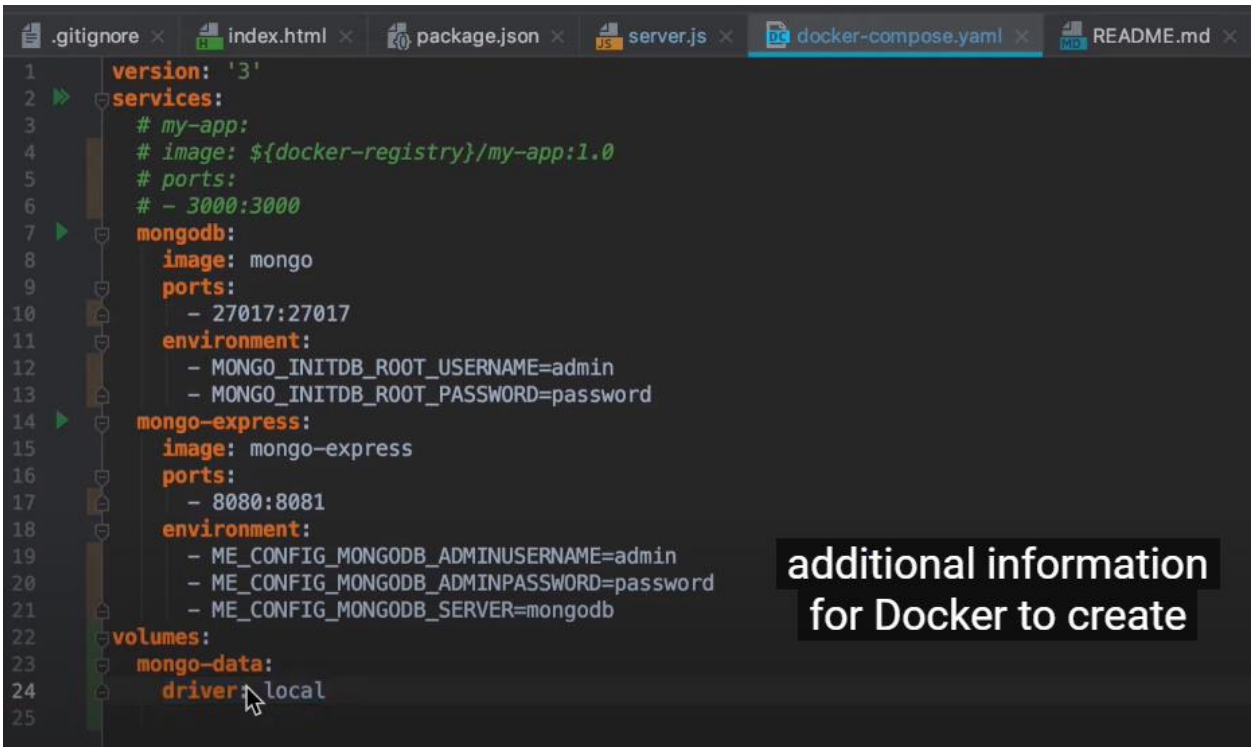

localhost:8080
Mongo Express
Database
Mongo Express
Databases
my-db
Database Name
Database names cannot be empty, must have fewer than 64 characters and must not contain /. *$* <>:|?
View
admin
config
local
Del
Server Status
Hostname
06861ed7086b
MongoDB Version
4.2.1
Uptime
30 seconds
Server Time
Wed, 04 Dec 2019 22:29:02 GMT
Current Connections
3
Available Connections
838857

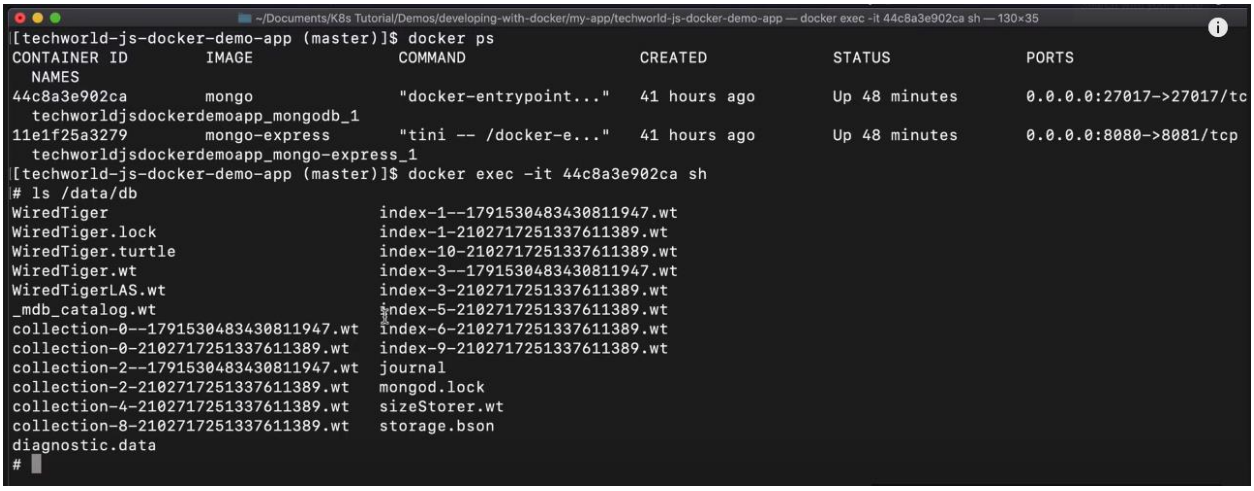

localhost:8080/db/my-db/users
Mongo Express
Database: my-db
Viewing Database: my-db
Collection Name
Collection names must begin with a letter, underscore, hyphen or slash, and can contain only letters, underscores, hyphens, numbers, dots or slashes
Collections
users
+ Create collection
View
Export
[JSON]
Import
Del
Database Stats

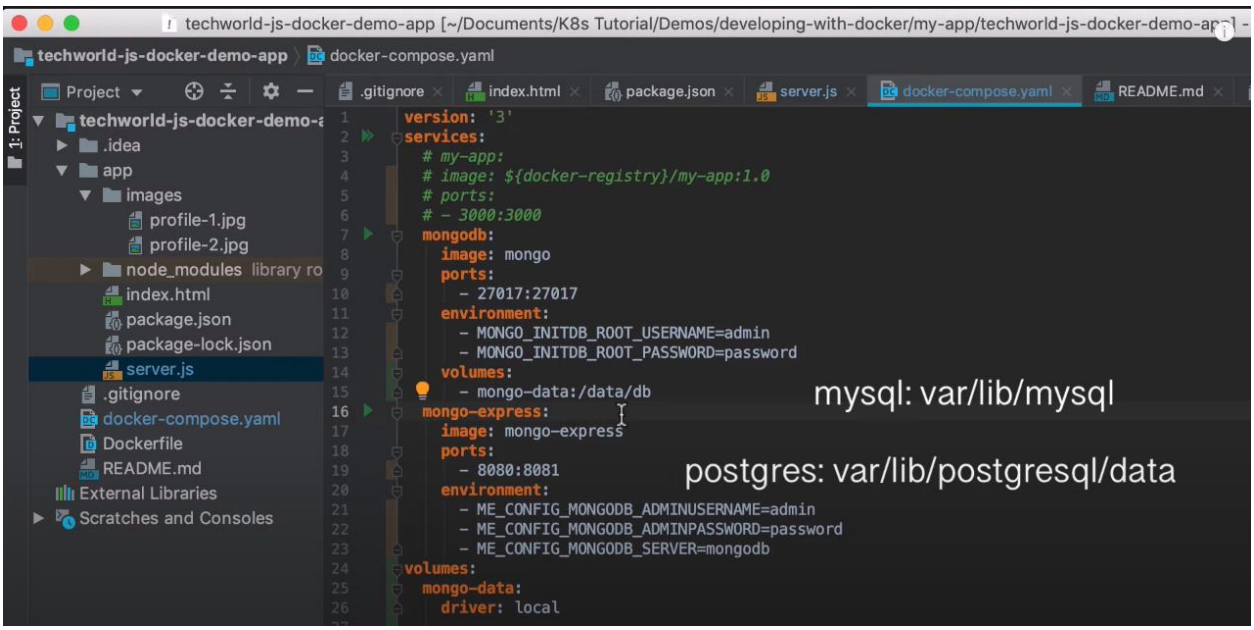

[app (master)]$ npm run start
> developing-with-docker@1.0.0 start /Users/nanajanashia/Documents/K8s Tutorial/Demos/developing-with-docker
y-app/techworld-js-docker-demo-app/app
> node server.js
app listening on port 3000!

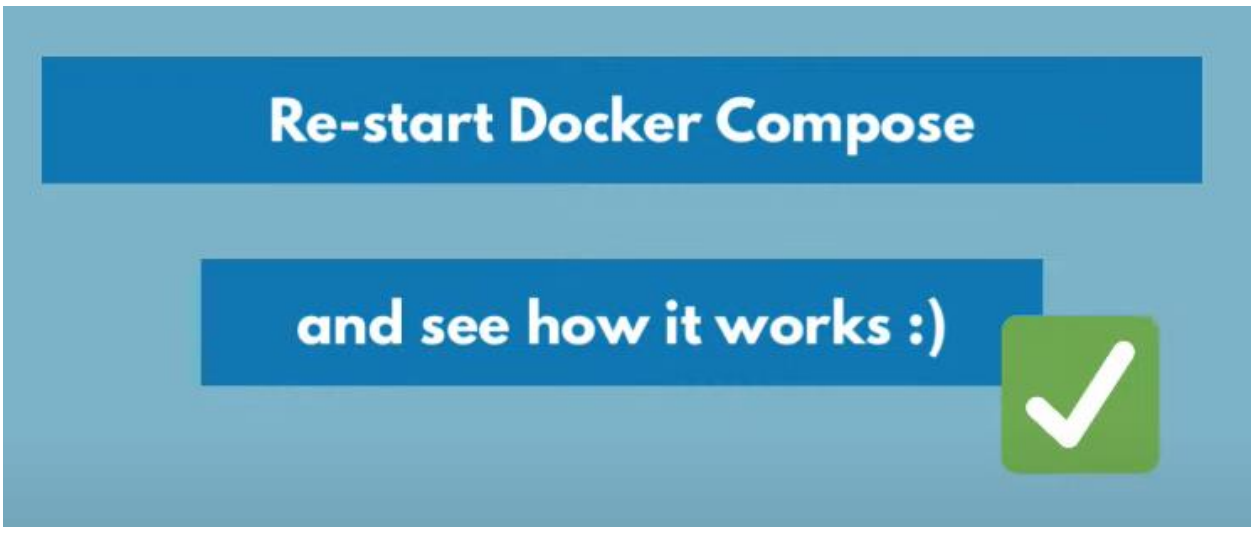

Define Named Volume in
Docker Compose File


techworld-js-docker-demo-app [~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js-docker-demo-app]
techworld-js-docker-demo-app
docker-compose.yaml
Project
techworld-js-docker-demo-a
.idea
app
images
profile-1.jpg
profile-2.jpg
node_modules library ro
index.html
package.json
package-lock.json
server.js
.gitignore
docker-compose.yaml
Dockerfile
README.md
External Libraries
Scratches and Consoles
.gitignore
index.html
package.json
server.js
docker-compose.yaml
README.md
version: '3'
services:
my-app:
image: ${docker-registry}/my-app:1.0
ports:
- 3000:3000
mongodb:
image: mongo
ports:
- 27017:27017
environment:
- MONGO_INITDB_ROOT_USERNAME=admin
- MONGO_INITDB_ROOT_PASSWORD=password
mongo-express:
image: mongo-express
ports:
- 8080:8081
environment:
- ME_CONFIG_MONGODB_ADMINUSERNAME=admin
- ME_CONFIG_MONGODB_ADMINPASSWORD=password
- ME_CONFIG_MONGODB_SERVER=mongodb
volumes:
List all the volumes you are using
in any of your containers


.gitignore
index.html
package.json
server.js
docker-compose.yaml
README.md
version: '3'
services:
my-app:
image: ${docker-registry}/my-app:1.0
ports:
- 3000:3000
mongodb:
image: mongo
ports:
- 27017:27017
environment:
- MONGO_INITDB_ROOT_USERNAME=admin
- MONGO_INITDB_ROOT_PASSWORD=password
mongo-express:
image: mongo-express
ports:
- 8080:8081
environment:
- ME_CONFIG_MONGODB_ADMINUSERNAME=admin
- ME_CONFIG_MONGODB_ADMINPASSWORD=password
- ME_CONFIG_MONGODB_SERVER=mongodb
volumes:
mongo-data:
driver: local
additional information
for Docker to create


~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js-docker-demo-app — docker exec -it 44c8a3e902ca sh — 130×35
[techworld-js-docker-demo-app (master)]$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
NAMES
44c8a3e902ca mongo "docker-entrypoint..." 41 hours ago Up 48 minutes 0.0.0.0:27017->27017/tc
techworldjsdockerdemoapp_mongodb_1
11e1f25a3279 mongo-express "tini -- /docker-e..." 41 hours ago Up 48 minutes 0.0.0.0:8080->8081/tcp
techworldjsdockerdemoapp_mongo-express_1
[techworld-js-docker-demo-app (master)]$ docker exec -it 44c8a3e902ca sh
ls /data/db
WiredTiger index-1--1791530483430811947.wt
WiredTiger.lock index-1-2102717251337611389.wt
WiredTiger.turtle index-10-2102717251337611389.wt
WiredTiger.wt index-3--1791530483430811947.wt
WiredTigerLAS.wt index-3-2102717251337611389.wt
_mdb_catalog.wt index-5-2102717251337611389.wt
collection-0--1791530483430811947.wt index-6-2102717251337611389.wt
collection-0-2102717251337611389.wt index-9-2102717251337611389.wt
collection-2--1791530483430811947.wt journal
collection-2-2102717251337611389.wt mongod.lock
collection-4-2102717251337611389.wt sizeStorer.wt
collection-8-2102717251337611389.wt storage.bson
diagnostic.data
#


techworld-js-docker-demo-app [~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js-docker-demo-app]
techworld-js-docker-demo-app
docker-compose.yaml
Project
techworld-js-docker-demo-a
.idea
app
images
profile-1.jpg
profile-2.jpg
node_modules library ro
index.html
package.json
package-lock.json
server.js
.gitignore
docker-compose.yaml
Dockerfile
README.md
External Libraries
Scratches and Consoles
.gitignore
index.html
package.json
server.js
docker-compose.yaml
README.md
version: '3'
services:
my-app:
image: ${docker-registry}/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
ports:
- 8080:8081
environment:
- ME_CONFIG_MONGODB_ADMINUSERNAME=admin
- ME_CONFIG_MONGODB_ADMINPASSWORD=password
- ME_CONFIG_MONGODB_SERVER=mongodb
volumes:
mongo-data:
driver: local
mysql: var/lib/mysql
postgres: var/lib/postgresql/data


Re-start Docker Compose
and see how it works :)

```
~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js-docker-demo-app — docker-compose ◂ docker-compose -f doc
~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js-docker-demo-app — -bash ... ~/Documents/K8s Tutorial/Demos/developing-with-docker/my-a
Last login: Fri Dec  6 18:33:58 on ttys007
[techworld-js-docker-demo-app (master)]$ docker-compose -f docker-compose.yaml down
Stopping techworldjsdockerdemoapp_mongodb_1 ... done
Stopping techworldjsdockerdemoapp_mongo-express_1 ... done
Removing techworldjsdockerdemoapp_mongodb_1 ... done
Removing techworldjsdockerdemoapp_mongo-express_1 ... done
Removing network techworldjsdockerdemoapp_default
[techworld-js-docker-demo-app (master)]$ docker-compose -f docker-compose.yaml up
Creating network "techworldjsdockerdemoapp_default" with the default driver
```

```
[techworld-js-docker-demo-app (master)]$ docker-compose -f docker-compose.yaml down
Removing techworldjsdockerdemoapp_mongodb_1 ... done
Removing techworldjsdockerdemoapp_mongo-express_1 ... done
Removing network techworldjsdockerdemoapp_default
[techworld-js-docker-demo-app (master)]$ docker-compose -f docker-compose.yaml up
Creating network "techworldjsdockerdemoapp_default" with the default driver
Creating techworldjsdockerdemoapp_mongodb_1
Creating techworldjsdockerdemoapp_mongo-express_1
```

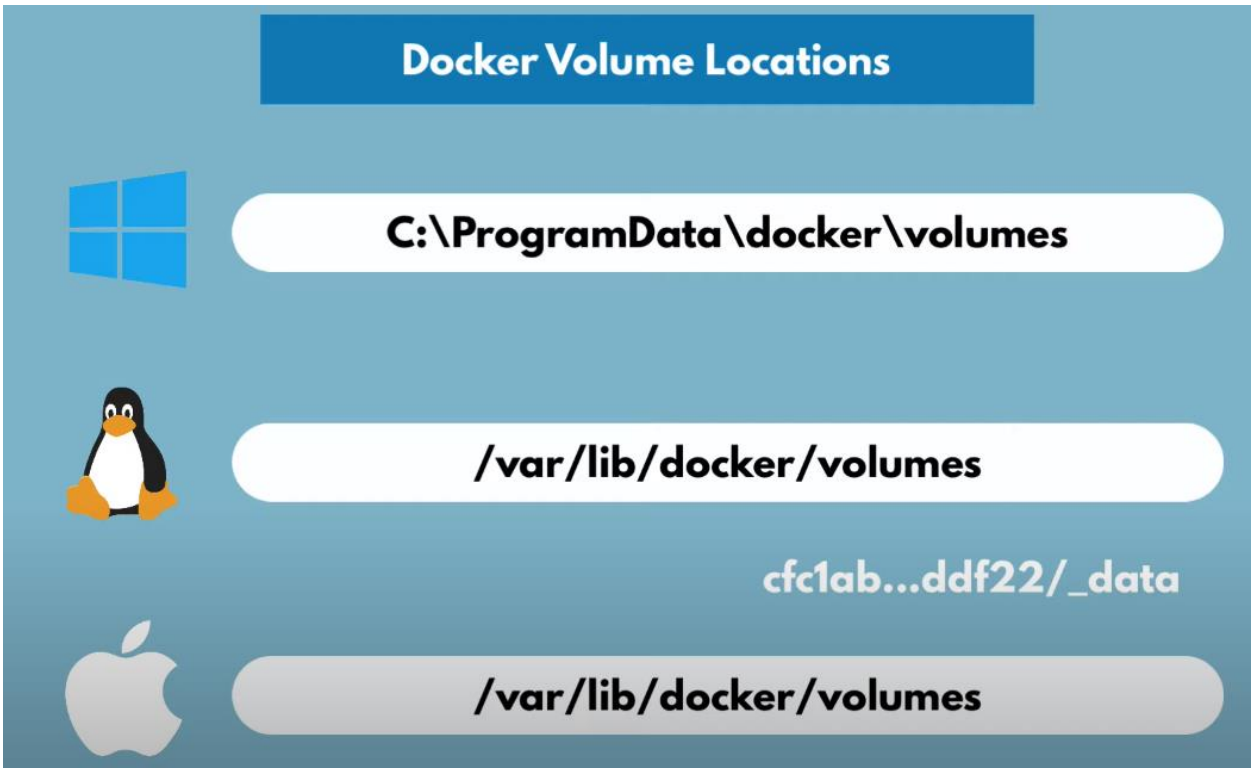
See where the Docker Volumes are located

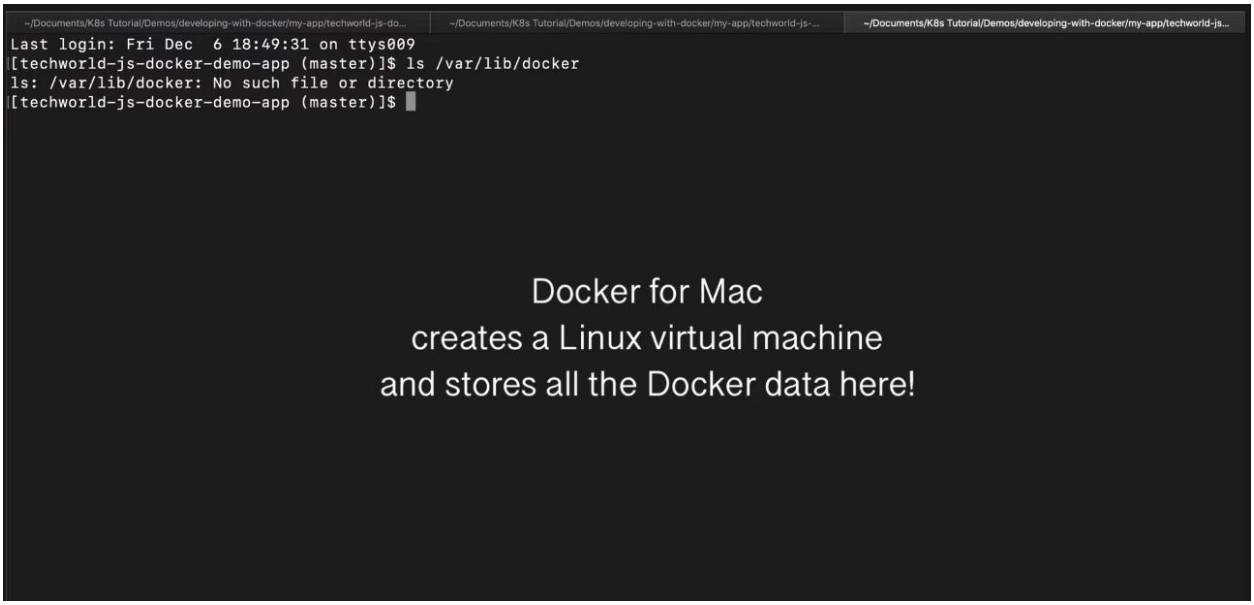
Docker Volume Locations

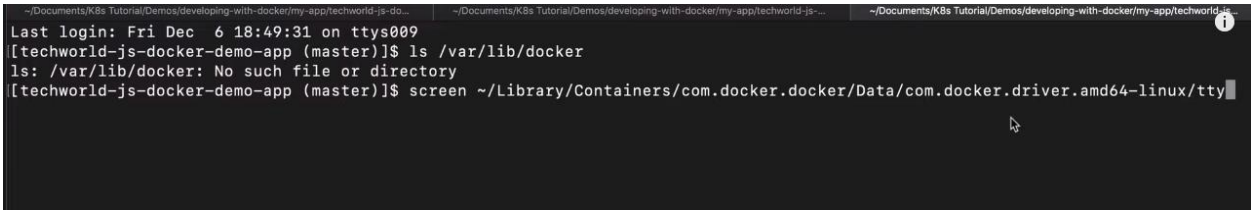
C:\ProgramData\docker\volumes

/var/lib/docker/volumes

/var/lib/docker/volumes

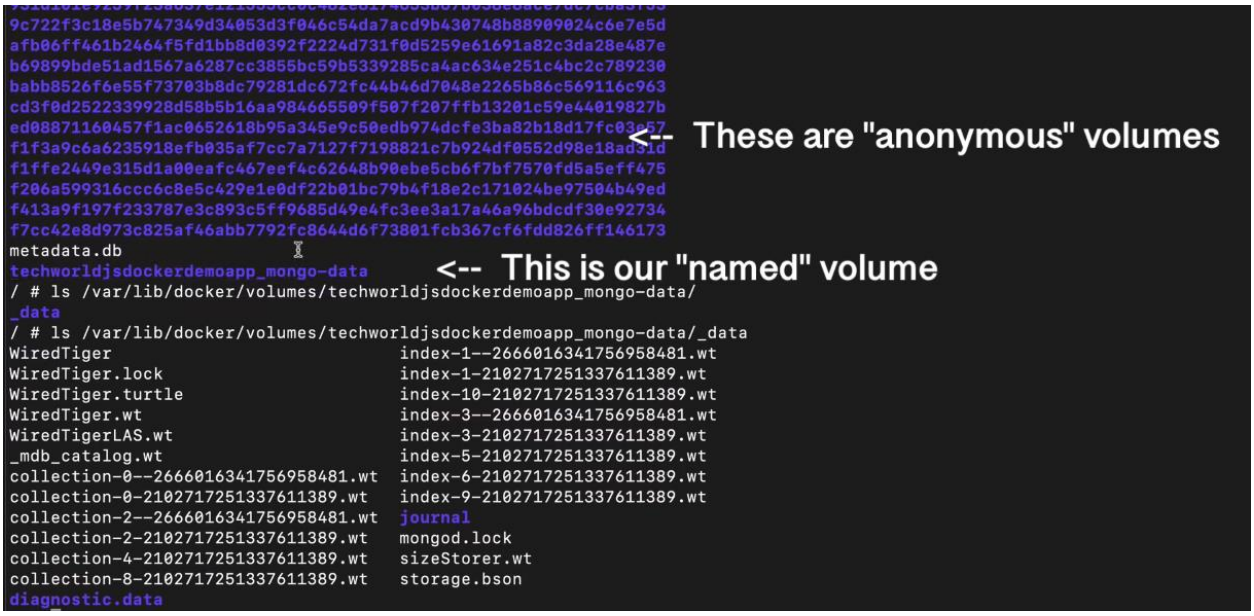

Docker Volume Locations
C:\ProgramData\docker\volumes
/var/lib/docker/volumes
cfc1ab...ddf22/_data
/var/lib/docker/volumes

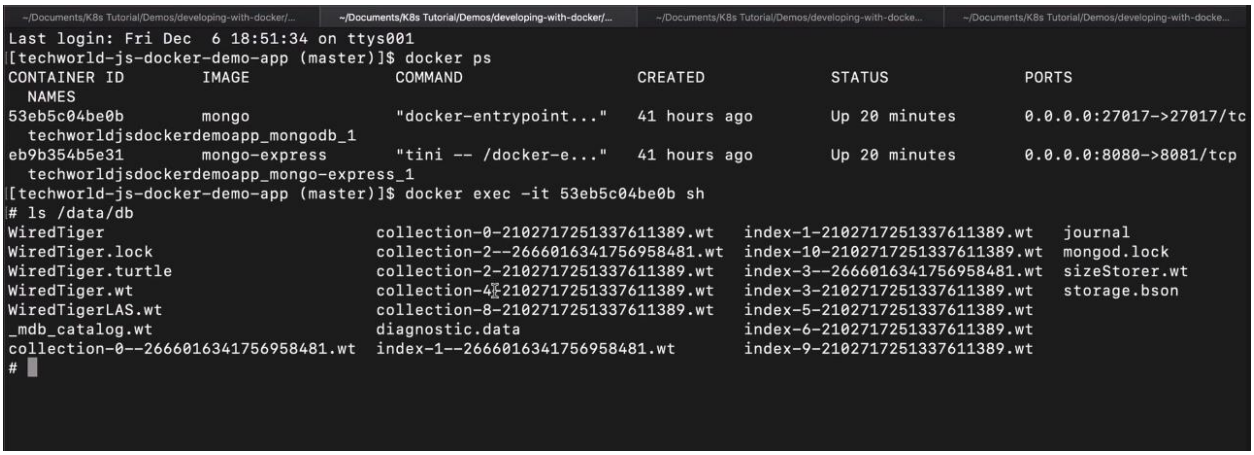

~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js-do...
~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js-...
~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js...
Last login: Fri Dec 6 18:49:31 on ttys009
[techworld-js-docker-demo-app (master)]$ ls /var/lib/docker
ls: /var/lib/docker: No such file or directory
[techworld-js-docker-demo-app (master)]$
Docker for Mac
creates a Linux virtual machine
and stores all the Docker data here!

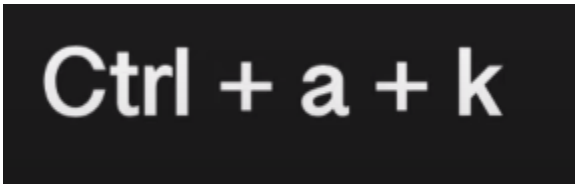

~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js-do...
~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js-...
~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js...
Last login: Fri Dec 6 18:49:31 on ttys009
[techworld-js-docker-demo-app (master)]$ ls /var/lib/docker
ls: /var/lib/docker: No such file or directory
[techworld-js-docker-demo-app (master)]$ screen ~/Library/Containers/com.docker.docker/Data/com.docker.driver.amd64-linux/tty


~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js-docker-demo-app — screen ~/Library/Containers/com.docker.docker/
~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js-do...
~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js-...
/ # ls
Database etc mnt sbin
Users home port srv
Volumes host_docker_app private sys
bin init proc tmp
containers lib root usr
dev media run var
/ # ls /var/lib/docker
containers network plugins tmp volumes
image overlay2 swarm trust
/ #


~/Documents/K8s Tutorial/Demos/developing-with-docker/my-app/techworld-js-do...
~/Documents/K8s Tutorial/Demos/developing-with-docker/
/ # ls /var/lib/docker/volumes/
1cb23af214bd09396b0b8e18e5b8772830994c19376a5c4ade739bc32b708520
26b1f55fb000c53934e92102d668381496c46ef611f0f95bdc1618d539e7106d
28465e583edfcd53205cbaae139f033daf2d2c7ac18ca56d9dd4aa66a653f83b
2a2e949c7a05612c288a6f993025850ff1f7111541dde308a73b1ebaaeea7c34
2b2074afdf5f250ac0d2f573f7cb1606c429738c4494036b9ccff211f811eb4d
30e8cf45b4d7f717ba196212908162 2d17f7e220bbabdf65687492f6cc11277d
3c33388ab8c990159ef458b6994dd91a78f39630adf8d9183023a98c023e5c51
45d3293e60f97b27642303449cb72847e82ed63cbbabdf576f3369e1e8bb8e2d
51b73c5dcb54a12c1724847cbbe38e23906f907cdc927c4586dda436b1556b7a
5edb61a83824436d600fc639a3027e019e4151f769d73b17572dc1bdd710c172
657b3fe2a6bac038bc97e069f3f979069063700f5aedea12b57c577b08f9911f
6bab4d6bd4363e0d6af87208d0bb3c655152695dc479e52ec2f0d7ad3359c170
6f738e49f180a55da829c348f645a169c2014ccaf4504976758fea0ee91cd50e
73b59771e3f02508cca84afc4f5d8108ab1487a2de9f8f6bd90af47669039d53
7a8e8a84f9f44a29e617dd3620790f0ca8f28606e3b54915dd4c6cdee4780f2b
7d4761d64e8f0b2ce78b6848e4ed9017cf15acefce42172fcdede245ed9f6f10
83e8a1dded35b85b00f0301c93e4282976a2bc8c128e3d242a21acbf34cb043e
931d101e9259f23a637e121555cc0c482e8174653b67b038e8ace7dc7cba3f53
9c722f3c18e5b747349d34053d3f046c54da7acd9b430748b88909024c6e7e5d
afb06ff461b2464f5fd1bb8d0392f2224d731f0d5259e61691a82c3da28e487e
b69899bde51ad1567a6287cc3855bc59b5339285ca4ac634e251c4bc2c789230
babb8526f6e55f73703b8dc79281dc672fc44b46d7048e2265b86c569116c963
cd3f0d2522339928d58b5b16aa984665509f507f207ffb13201c59e44019827b
ed08871160457f1ac0652618b95a345e9c50edb974dcfe3ba82b18d17fc03e57
f1f3a9c6a6235918efb035af7cc7a7127f7198821c7b924df0552d98e18ad31d
f1ffe2449e315d1a00eafc467eef4c62648b90ebe5cb6f7bf7570fd5a5eff475
f206a599316ccc6c8e5c429e1e0df22b01bc79b4f18e2c171024be97504b49ed
f413a9f197f233787e3c893c5ff9685d49e4fc3ee3a17a46a96bdcdf30e92734
f7cc42e8d973c825af46abb7792fc8644d6f73801fcb367cf6fdd826ff146173
metadata.db
techworldjsdockerdemoapp_mongo-data







Now type y and the session will be completed.

```
~/Documents/K8s Tutorial/Demos/developing-with-docker/...  ~/Documents/K8s Tutorial/Demos/developing-with-docker/...  ~/Documents/K8s Tutorial/Demos/developing-with-docke...  ~/Documents/K8s Tutorial/Demos/developing-with-docke...
Last login: Fri Dec  6 18:49:31 on ttys009
[techworld-js-docker-demo-app (master)]$ ls /var/lib/docker
ls: /var/lib/docker: No such file or directory
[techworld-js-docker-demo-app (master)]$ screen ~/Library/Containers/com.docker.docker/Data/com.docker.driver.amd64-linux/tty
[screen is terminating]
[techworld-js-docker-demo-app (master)]$ screen ls
[screen is terminating]
[techworld-js-docker-demo-app (master)]$
```