First Steps with Docker

What is Docker?

- A virtualization software .
- Makes developing and deploying applications much easier .
- It packages an application into something called : Container; that has everything the application needs to run :
  - The application code itself
  - Its libraries and dependencies
  - The runtime and environment configurations
- Portable Artifact, easily shared and distributed

What Problems Docker Solves?

▼ Development Process before containers?

- Each developer needs to install and configure all services directly on their OS on their local machine.
- Installation Process different for each OS environment.

- So, if your application uses 10 services , each developer needs to install these 10 services and still the installation differ from each OS of each developer which may cause a lot of issues .

▼ Development Process with containers?

- With containers, you do not have to install all the 10 services on your Os
- Because with Docker, you have that service package in one isolated environment
  - You have each service with all its dependencies and its configurations inside the container
- Good News: All you need to so as a developer is Start Service as a Docker Container using 1 Docker Command ; which fetches the container package from internet and start it on your computer .
  - The docker command will be the same regardless on the OS you're on , and will be also the same regardless on which service you're installing

⇒ Docker Standardizes process of running any service on any local dev environment.

⇒ In Docker, You can even have different versions of the same application running in your local environment without conflicts

▼ Deployment Process with containers?

- Development Team would produce an application artifact or a package together with a set up instructions on how to install and configure the app package on the service, along with services of the application and its set up constructions
- Development Team would give the Artifact and instructions to the Operations team
  - Operations team would handle installing and configuring apps and its dependencies
- Problem of this approach :
  - Installations and configurations done directly on the server's OS.
  - Dependency version Conflicts
  - Miscommunication Between the Dev and Ops Teams :
    - Since there's only a textual guide
    - A lot of Back and forth Communication until the application is successfully deployed in the server.

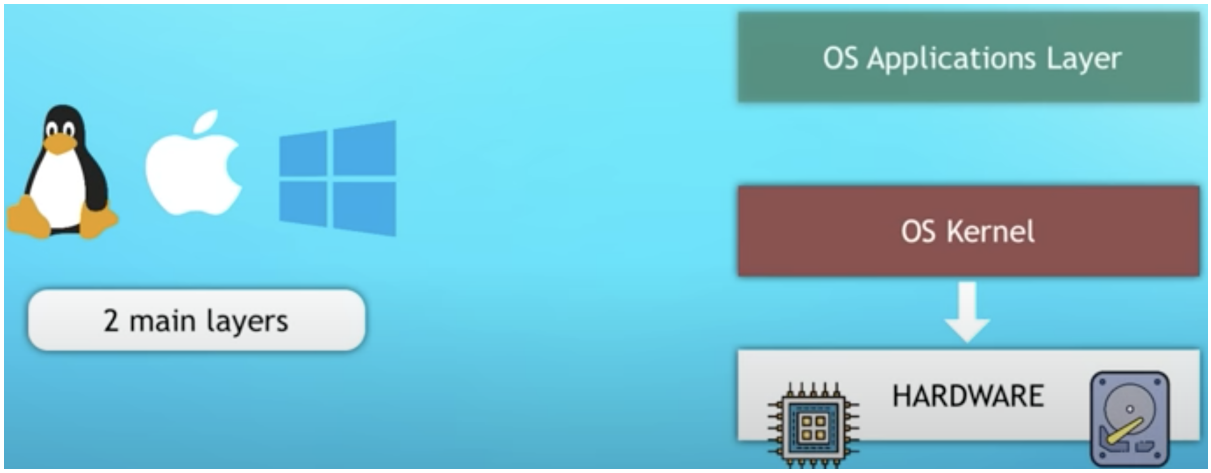
▼ Deployment Process with containers?

- NOW , Developers create an application package that doesn't only include the code itself but also all the dependencies and configurations for the app
- Instead of textual, everything is packaged inside the Docker artifact
- And since it's already encapsulated in one environment
    - The Ops People won't need to do any configurations on the server (except the docker command : Docker runtime)
- Less room for errors
- The only thing the Ops team needs to do is :
    - Install Docker runtime on the server
    - Run Docker command to fetch and run the Docker artifacts

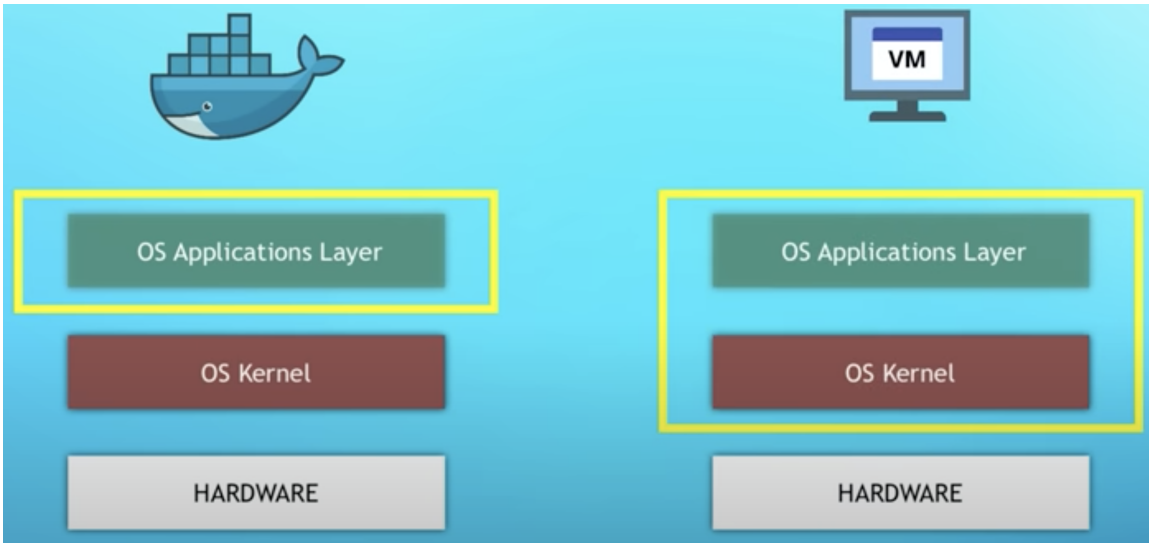
⇒ Deployment Process way easier on the operations side

Virtual Machine VS Docker ?

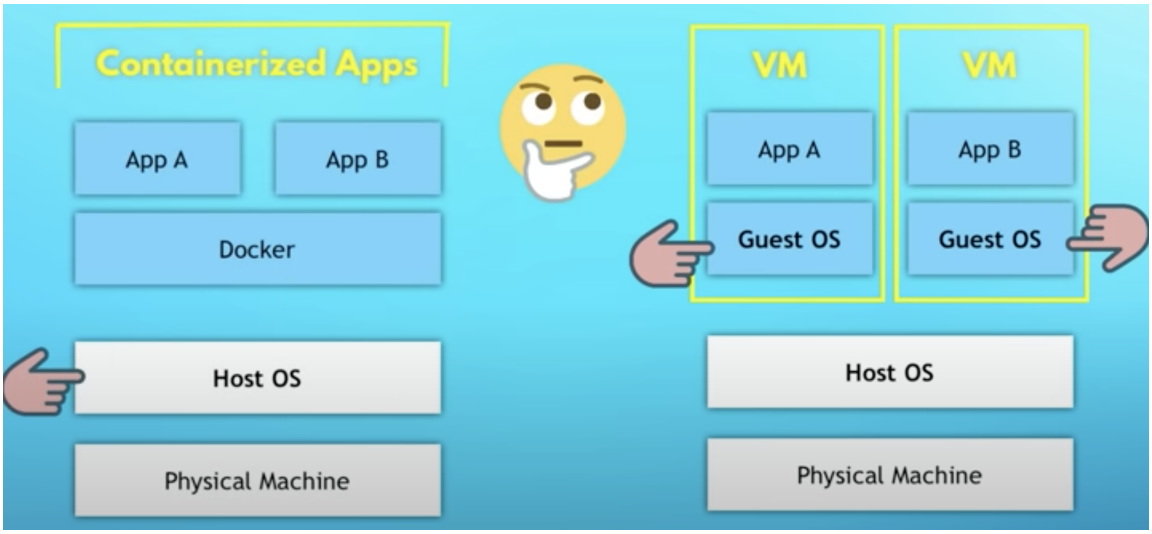
- Docker is a Virtualization tool , just like VM , BUT ?
    - WHY IS DOCKER SO WIDELY USED?
    - WHAT ADVANTAGES DOES IT HAVE OVER VMS?
    - WHAT ARE THE DIFFERENCES?
- Let's See how Docker works on a technical level !
- We said that with Docker, You don't need to install services directly on OS , but in that case , HOW DOES DOCKER RUN ITS CONTAINERS ON A OS?
- Let's First see how an OS is made up:



- A kernel is the part that communicates with the hardware components like CPU , Memory , Storage …
- When you have a physical machine and you install an OS on it ; THE KERNEL IS AT THE CORE OF EVERY OPERATING SYSTEM , and it interacts between hardware & Software components (Allocate Storage etc to the applications running on the OS system). Those Applications are part of the Applications layer , and they run on top of the KERNEL layer
- So the KERNEL is like the middle between the applications that you seem when you interact with your computer, and the underlying hardware of the computer
- Since Both Docker and VM are virtualization tools , What parts of the OS do they virtualize? ⇒ That's When the main difference between Docker and VM actually lyes .
    - DOCKER Virtualizes the Applications layer !
        - This means When you run a docker container, it actually contains the applications layer of the OS and some other applications and services installed on top of that layer (it could be JavaRuntime , python …)
        - It uses the Kernel of the host, cuz it doesn't have its own kernel .
    - The VM on the other hand, heads the Applications layer and its own Kernel , so it virtualizes the complete OS



- Which means when you download a VM on your host , it doesn't use the host kernel , it actually boots up its own .



- What affects has this difference?
  - **Size** : The size of Docker packages or images are much smaller , because they just have to implement one layer open the OS ⇒ Docker images = couple of MB
  - VM images = couple of GB

  ⇒ Docker saves a lot os Disks Space
  - **SPEED** : You can run and start docker containers much faster than VMs since VM have to boot up a kernel every time it starts , while docker container just reuses the host kernel and starts the application layer in top of it

- **Compatibility** : You can run VMs with all OS , BUT you can't do that with Docker (at least not directly) ⇒ Docker is compatible only with Linux
    - Let's say you have a Windows OS with a windows kernel and its application layer , and you want to run a linux based docker image directly on that windows host . The problem here is **Linux** based Docker images cannot use **Windows Kernel**, it would need a linux kernel to run
    - Docker is originally built for linux OS since most containers are linux based .
    - HAPPY UPDATE : Docker made an update : **Docker Desktop** for Windows and MacOs ; Which made it possible to run Linux containers on Windows and MacOs
        - **Docker Desktop uses a hypervisor Layer with a lightweight linux on top of it**

Install Docker on your local Machine

- For Windows: https://docs.docker.com/desktop/install/windows-install/
- For Mac: https://docs.docker.com/desktop/install/mac-install/

What's included in Docker Desktop?

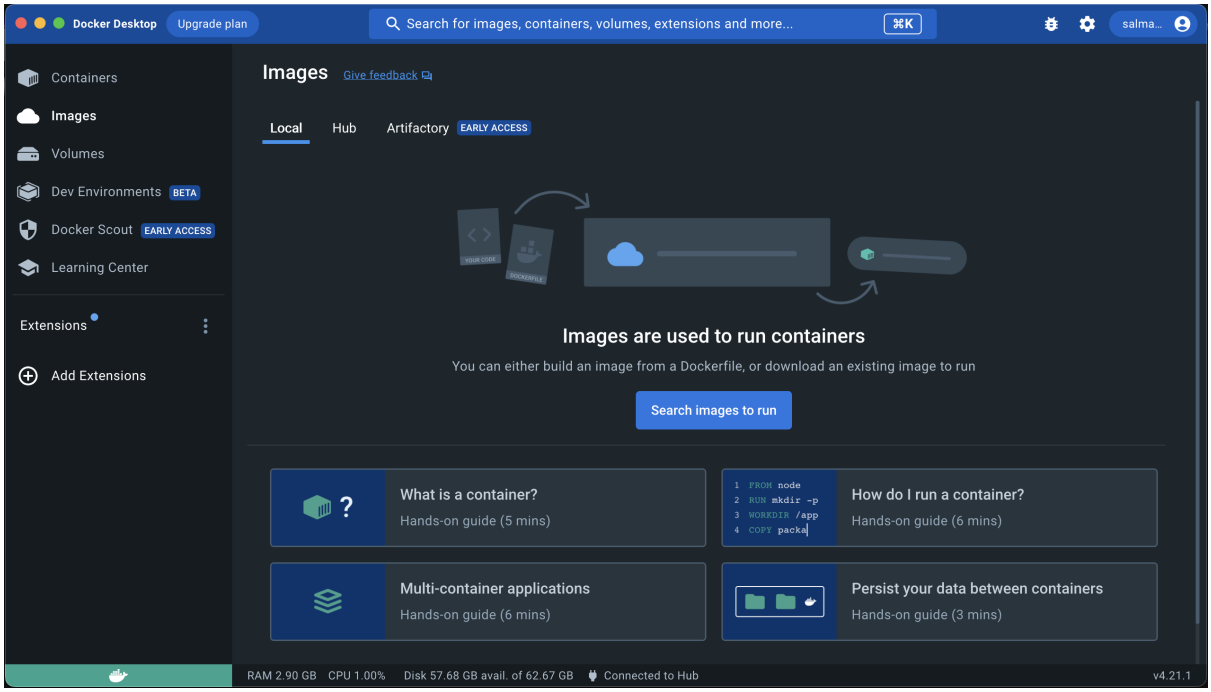
- Docker Engine:
    - It's the docker server itself
    - Main part of docker that makes this virtualisation possible
    - Manages images & Containers
- Docker CLI client:
    - The command line to interact with docker service
    - Execute docker commands to start/stop .. containers.
- GUI Client :
    - If you are not comfortable with CLI , You can use the Graphical User Interface to manage your containers and images .

Docker Images VS Docker Containers

- Docker allows to package the application with its environment configuration in his package that you can share and distribute easily (Like a zip or tar file or jar file that you can download to the server whenever you need it) ⇒ This package is called :**Docker Image**
    - An executable application artifact
    - Includes app source code and also complete environment configuration
    - Add environment variables, create directories, files etc
- What's a Container then?
    - Actually, We need to start that application package(image) somewhere right?
    - A Container is a running instance of an image
    - You can run multiple containers from one image

Congrats on installing Docker!

- Now that you have seen what the GUI looks like , let's jump to your terminal from where you can now execute docker commands. Cool Huh?



- Check what images you have available locally :

```
docker images
```

- Check Container:

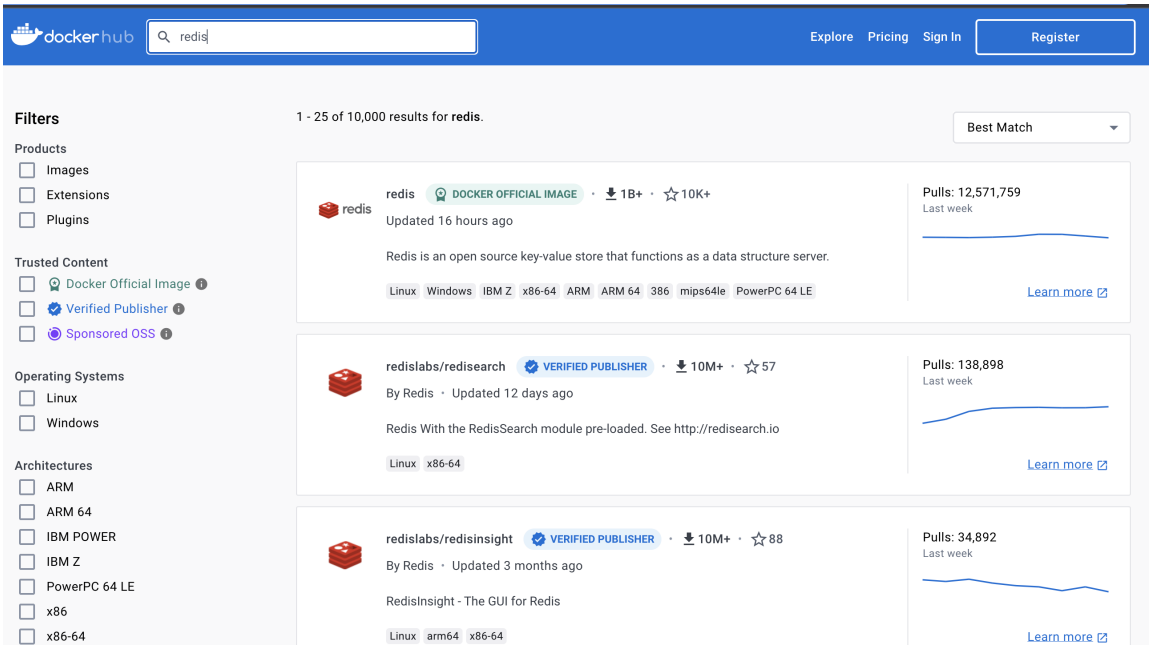
```
docker ps
```

Now it's Clear that we get containers by running images , BUT how do we get these images to get containers from? ⇒ That's Where Docker registries come in !

Docker Registries

- There are ready docker images available online in image storage or registry
- It is a storage and distribution system for docker images
- There are some official images available from applications like Mongo , ..
- Official Images are maintained by the software authors or in collaboration with the docker community
- And Docker itself offers the biggest Docker Registry called " **Docker Hub** "; where you can find and share docker images

  You can search for any service you want inside Docker Hub.



Since Technology changes, and each service develop updated versions , Docker images are versioned as well !

Image Versioning

- Different versions are identified by tags called Images tags
- On the page of each image , you actually have the list the versions or tags of that image

Supported tags and respective `Dockerfile` links

- `7.2-rc3` , `7.2-rc` , `7.2-rc3-bookworm` , `7.2-rc-bookworm`
- `7.2-rc3-alpine` , `7.2-rc-alpine` , `7.2-rc3-alpine3.18` , `7.2-rc-alpine3.18`
- `7.0.12` , `7.0` , `7` , `latest` , `7.0.12-bookworm` , `7.0-bookworm` , `7-bookworm` , `bookworm`
- `7.0.12-alpine` , `7.0-alpine` , `7-alpine` , `alpine` , `7.0.12-alpine3.18` , `7.0-alpine3.18` , `7-alpine3.18` , `alpine3.18`
- `6.2.13` , `6.2` , `6` , `6.2.13-bookworm` , `6.2-bookworm` , `6-bookworm`
- `6.2.13-alpine` , `6.2-alpine` , `6-alpine` , `6.2.13-alpine3.18` , `6.2-alpine3.18` , `6-alpine3.18`
- `6.0.20` , `6.0` , `6.0.20-bookworm` , `6.0-bookworm`
- `6.0.20-alpine` , `6.0-alpine` , `6.0.20-alpine3.18` , `6.0-alpine3.18`

- Docker tags are used to identify images by names
- You can choose the docker image version that will go with the version of the technology you are using .
- “latest” tag refers to the newest release; Last image that was built.

How Can We Get the Image from Docker Hub?

- First, Locate the image you want to get (We'll be getting for example nginx)
- Pick a specific Image tag (Note that using a specific version is best practice in most cases)
- To get that image :

```
docker pull name:tag
```

- Let it be version 1.25 ngnix version for our case :

```
(base) SalmaDkier@MacBook-Pro-de-mac-2 ~ % docker pull nginx:1.25
1.25: Pulling from library/nginx
648e0aadf75a: Pull complete
262696647b70: Pull complete
e66d0270d23f: Pull complete
55ac49bd649c: Pull complete
cbf42f5a00d2: Pull complete
8015f365966b: Pull complete
4cadff8bc2aa: Pull complete
Digest: sha256:67f9a4f10d147a6e04629340e6493c9703300ca23a2f7f3aa56fe615d75d31ca
Status: Downloaded newer image for nginx:1.25
docker.io/library/nginx:1.25

What's Next?
  View summary of image vulnerabilities and recommendations → docker scout quick
view nginx:1.25
```

- Now your CLI goes to docker hub and download the image you want locally(Docker Hub is by the default when docker will go look for images)
- Now if we execute “docker images” we should see our image now locally

```
(base) SalmaDkier@MacBook-Pro-de-mac-2 ~ % docker images
REPOSITORY   TAG       IMAGE ID       CREATED        SIZE
nginx        1.25      89da1fb6dcb9   45 hours ago   187MB
```

- LET'S RUN OUR IMAGE NOW !

```
docker run name:tag
```

- Now our container have started running its image , we can check it by the logs starting up inside the container:

```
(base) SalmaDkier@MacBook-Pro-de-mac-2 ~ % docker run nginx:1.25
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configurat
ion
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.co
nf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2023/07/29 23:19:02 [notice] 1#1: using the "epoll" event method
2023/07/29 23:19:02 [notice] 1#1: nginx/1.25.1
2023/07/29 23:19:02 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
```

- Now, if we open another terminal window and run :

```
docker ps
```

```
o-de-mac-2 ~ % docker ps
CONTAINER ID   IMAGE        COMMAND                  CREATED         STATUS         PORTS     NAMES
e315286f48e8   nginx:1.25   "/docker-entrypoint.…"   3 minutes ago   Up 3 minutes   80/tcp    sleepy_villani
```

    - Docker generates a random name for the container automatically if you don't specify one .

- Since the logs of the container are blocking the first terminal , you can exit them by “ Cntrl C” , and when you'll run the “docker ps” command, your container will no longer be in the list since it is killed.
- Whereas if you want to run your container without blocking your terminal :

```
docker run -d name:tag
```

```
(base) SalmaDkier@MacBook-Pro-de-mac-2 ~ % docker run -d nginx:1.25
827229480f666af20b9a9e1f02ede8ca7037ea7ddea2effe952e906d9751a164
(base) SalmaDkier@MacBook-Pro-de-mac-2 ~ % docker ps
CONTAINER ID   IMAGE        COMMAND                  CREATED          STATUS          PORTS     NAMES
827229480f66   nginx:1.25   "/docker-entrypoint.…"   8 seconds ago    Up 7 seconds    80/tcp    dazzling_mclaren
```

  Now you can see all your containers without blocking your terminal! And your running container does not block the terminal anymore.

- Sometimes, You may need to see the logs which can be useful for debugging etc.

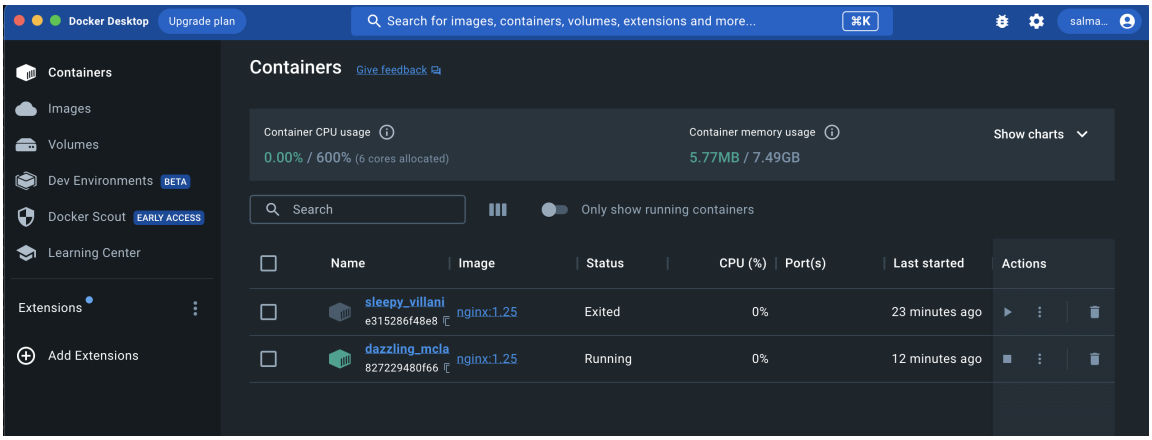
```
docker logs {containerID}
```

```
CONTAINER ID   IMAGE        COMMAND                  CREATED          STATUS          PORTS     NAMES
827229480f66   nginx:1.25   "/docker-entrypoint.…"   8 seconds ago    Up 7 seconds    80/tcp    dazzling_mclaren
(base) SalmaDkier@MacBook-Pro-de-mac-2 ~ % docker logs 827229480f66
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2023/07/29 23:30:30 [notice] 1#1: using the "epoll" event method
2023/07/29 23:30:30 [notice] 1#1: nginx/1.25.1
2023/07/29 23:30:30 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
2023/07/29 23:30:30 [notice] 1#1: OS: Linux 5.15.49-linuxkit-pr
2023/07/29 23:30:30 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2023/07/29 23:30:30 [notice] 1#1: start worker processes
2023/07/29 23:30:30 [notice] 1#1: start worker process 29
```

- Little TIP Spoiler :
    - docker pull …
    - docker run …

⇒ You can save your time by only executing “docker run …” command without pulling first , because Docker pulls image automatically if it doesn't find it locally.

- Btw, You can always check your existing containers at the GUI of Docker Desktop:



Now , The important question is How do we access this container?

- Well , We can' right now!!
  - The container is running inside the close docker network, so we can't access it from our local computer browser for example .
  - Because, Application inside containers runs in an isolated Docker network .
  - This allows us to run the same app running on the same port multiple times.
- So, We need first to expose the container to our local network (Don't worry ! It's super easy) ⇒ IT'S PORT BINDING

Port Binding

Bind the container's port to the host port to make the service available to the outside work.

Too much? Let's clarify it !

- The Container is running on a specific port right?
  - Each application has some standard port on which it's running(like nginx application always runs on port 80: and that's the port which container is running on by default)
  - You can check the port of the application you are running here : “ docker ps “

Now the application(nginx) is running on port 80 inside the container

- Now if I try to access the container from my local browser by : localhost:80 , nothing would be available!

- What to do ? We can tell docker , You know what ? Bind that container port (80) to our localhost on any port that I'll tell you (we can choose 8080 for example) so that I can access the container and whatever is running inside the container as if it was running on my localhost port 8080⇒ We do this with an additional fleg while creating the container.
- So we will stop our running container first

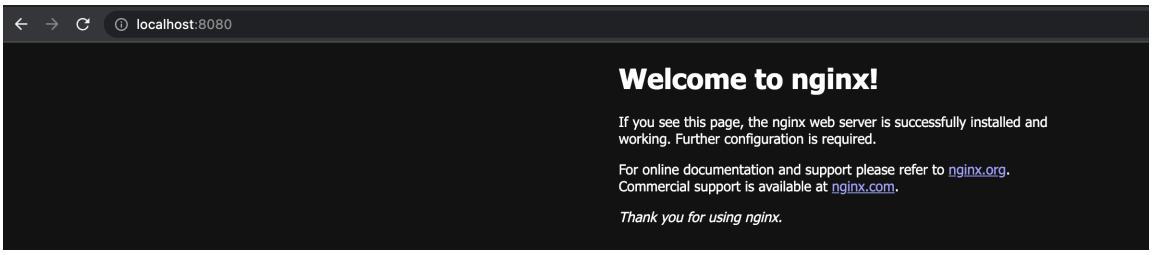
```
docker stop {containerID}
```

- And we're going to create our new container

```
 -p{HOST_PORT}:{CONTAINER_PORT}
```

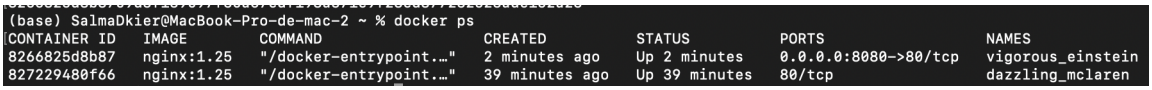
We used also '-d' to have the container in the background to not block our terminal

- TA-DA!



Magical Huh?

- PS: Only one service can run on a specific port on the host .
- You can always check your used ports by :



And check PORTS list!

- For choosing host port , It is standard to use the same port on your host as container is using.
- Not much complicated Right?

Start and Stop Containers

- docker run command
  - Creates a new container each time
  - Doesn't re-use previous containers
  - Means that since we have used now “docker run “ couple of times already, we should actually have multiple containers on our laptop. However, as in the last captured photo, “docker ps” only gives us the running container not all the containers we've started and stopped .BUT those containers actually still exists

```
-a or --all
```

    - Though , You can stop your running container to be shown as “Exited” in the status

```
docker stop {Container_ID}
```

  - In the same way, you can restart a past container without having to create a new one .

```
docker start {CONTAINER_ID}
```

  - You can also refer to the container by its name docker has randomly given to it .
  - You can still give meaningful names to your containers

```
--name
```

```
(base) SalmaDkier@MacBook-Pro-de-mac-2 ~ % docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
(base) SalmaDkier@MacBook-Pro-de-mac-2 ~ % docker run --name web-app -d -p 8080:80 nginx:1.25
d4dfbd8d428d547ea86f9e8beb555bab7cb3c563da417907d34321ab23ad3b7b
(base) SalmaDkier@MacBook-Pro-de-mac-2 ~ % docker ps
CONTAINER ID   IMAGE        COMMAND                  CREATED         STATUS         PORTS                  NAMES
d4dfbd8d428d   nginx:1.25   "/docker-entrypoint.…"   6 seconds ago   Up 5 seconds   0.0.0.0:8080->80/tcp   web-app
```

- Now , we can refer to our container by its name :

```
docker logs web-app
```

Private Docker Registries

Now we've learnt about Docker Hub which is the largest public image registry

- Anyone can search and download Docker images

There's also Private Docker Registries related to a specific big company :

- You need to authenticate before accessing the registry
- All big cloud provider offer private registries : Amazon ECR, Google Container Registry, etc
- Even Docker Hub has its private docker registry

Registry VS Repository

- Docker Registry:

  AWS ECR is:

  - A service providing storage
  - Can be hosted by a third party, like AWS or by yourself
  - And inside that registry you can have multiple repositories for all your different application images
- Docker Repository:
  - Each application gets its own repository
  - And in that repository, we can store different image with the same name but different versions or tags of the same application

- Docker Hub is a registry:
    - On Docker Hub you can host private or public repositories for your applications

+Companies creates custom images for their application , How does that actually work? How can you create your own image for your application?

+The use case for that is when you are done with development of the application, which is ready and have some features and you want to release it to the users , So you want to run it on a deployment server .

+And to make the deployment process easier, you 'll have to deploy your application as a Docker container along with the database and other services they will also run as docker containers .

+So how con we take our created deployed application code and package it into a docker image ?

⇒ For that We need to create a definition of how to build an image from our application, and that definition is written in a file called **Docker File**

Docker File

- Is a text document that contains commands to assemble an image .
- Docker can them build an image by reading those instructions.

In this part , we're gonna take a super simple node.js application, and we'll write a Docker file for this application to build a docker image from it and start it as a Docker container.

- Don't panic , It's very easy to do ! I will let you hear my repository to check yourself the easy and clean code!
- Structure of Docker File:
    - Dockerfiles start from a parent image or “ base image”
    - You choose the base image , depending on which tools you need to have available
        - It's a docker image that your image is based on.
            - So for node application , you 'll have node base image , if you have java application you will use an image that have java runtime installed
        - We define that based image by : “FROM “ ⇒ like we're telling docker to build the image based on the base image

- The base image is just like other images , you can pile and build on top of the images in docker , and they also have image versions.
- Every image consists of multiple image layers

```
FROM node:19-alpine
```

```
RUN npm install
```

- Till now we have node and npm installed , and we're executing npm install to install dependencies. However, We need the application code inside the container as well.

- For our case, we need “server.js” and “package.json” inside because that's what npm will need to actually read the dependencies

```
COPY <src> on our machine <dest> in the container
```

```
# Copy package.json, wildcard used so both package.json AND package-lock.json are copied
# slash '/' at the end of app is important, so it created an app directory, otherwise you'll get an error
COPY package*.json /usr/app/

# Copy app files from src directory
COPY src /usr/app/
```

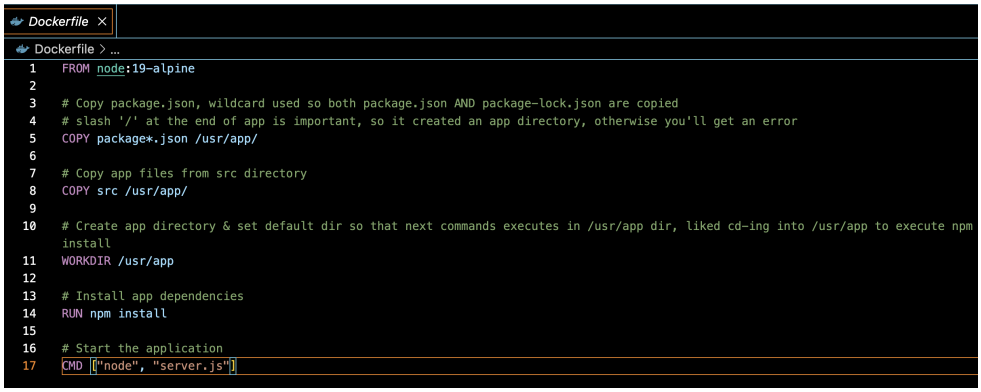
- Now after copying our files to a directory , we need to change the container to this directory /usr/app/ we created inside it .

```
WORKDIR /usr/app
```

- We need now to finally run the application; means the node command should be executed to start the application itself

```
CMD ["node", "server.js"]
```

  - The instruction that is to be executed when a docker container starts.
  - There can only be one “CMD” instruction in a dockerfile



That's IT ! That's the complete Dockerfile that will create the docker image for our node.js application which we can then start as a container

Build Image

- Now We need to actually build the Docker image from the dockerfile.

  ⇒ We can execute a docker command to build a docker image :

```
docker build{path}
```

We can build it via a name that we will associate by:

```
-t or --tag
```

Then in your EDE terminal , you tell docker to build an image named”node-app” with a tag of “1.0” and from the docker file which is in the current directory we're in (we referred to it by “.”)

- We used 3 arguments : -name of docker image / - its version(tag) / the directory of the docker file
- Now, Docker built really a docker image from our docker file ! Congrats on that !

- Remember:
  - A Docker image consists of layers
    - Each construction in the docker file creates one layer

They're all mentioned here .

- Now if we run “docker images “ , we will see our new built image in addition to the nginx image we downloaded from Docker Hub

```
(base) SalmaDkier@MacBook-Pro-de-mac-2 Get started with Docker % docker images
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
node-app     1.0       73aa870257b4   7 minutes ago   183MB
nginx        1.25      89da1fb6dcb9   2 days ago      187MB
```

- Now we can finally start this image and work with it like we worked with previous images downloaded from Docker Hub !
- So let's go a head and run a container from our node-app image we've just created to be sure the application inside is actually working
  - Remember?

```
(base) SalmaDkier@MacBook-Pro-de-mac-2 Get started with Docker % docker run -d -p 3000:3000 node-app:1.0
dd4f351f1133214e089672cb9552ae2d96c3558f2198d8e1618851d80d25db4e
```

  - As we did earlier :

- -d for the container to run in detached mode without blocking the terminal.
- -p for the port binding so that we can access our application for port 3000”host port” (3000 “container post”was the default port we set for our application (check server.js)
- Then finally name of the image with its tag
    - Now , we should be able to see our application running inside localhost:3000
        - TA-DA!

Let's make more sure:

```
(base) SalmaDkier@MacBook-Pro-de-mac-2 Get started with Docker % docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS         PORTS                    NAMES
dd4f351f1133   node-app:1.0   "docker-entrypoint.s…"   5 minutes ago   Up 5 minutes   0.0.0.0:3000->3000/tcp   festive_hopper
d4dfbd8d428d   nginx:1.25     "/docker-entrypoint.…"   3 hours ago     Up 3 hours     0.0.0.0:8080->80/tcp     web-app
```

EFFECTIVELY, our image is running on port 3000!

We can even listen to our logs :

```
(base) SalmaDkier@MacBook-Pro-de-mac-2 Get started with Docker % docker logs dd4f351f1133
app listening on port 3000
```

Dockerize your own application !

That's how easy it is to take your application and package it into a docker image using DockerFile and then run it as a container

Congratulations on Making it till the end!