Docker is an open-source platform that enables developers to build, deploy, run, update, and manage containers. Containers are standardized, executable components that combine application source code with the operating system (OS) libraries and dependencies required to run that code in any environment.

Docker allows developers to create containers without Docker by working directly with capabilities built into Linux and other operating systems, but Docker makes containerization faster and easier.

It works on my machine

This happens when a developer writes code that runs perfectly on **their local machine**, but fails when:

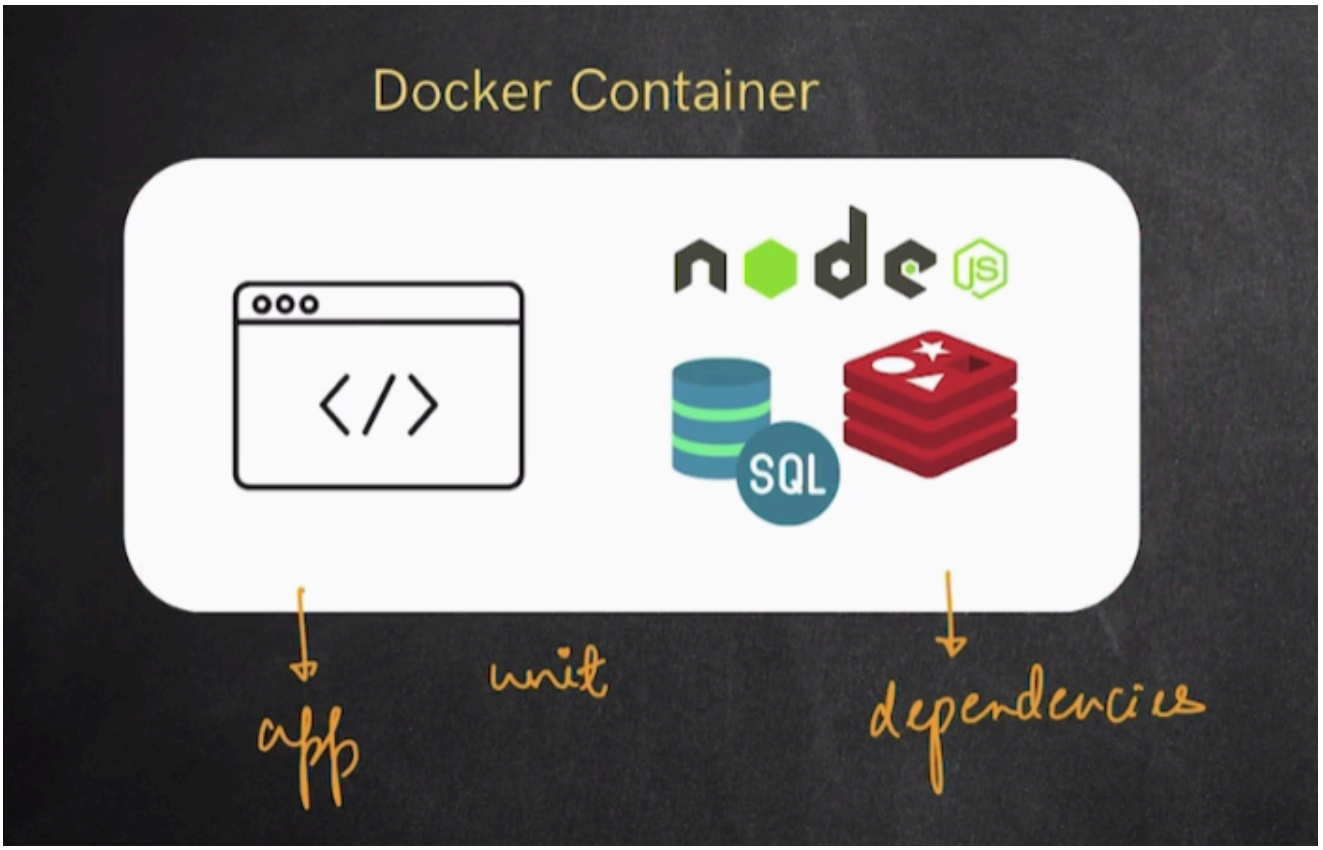
- Another dev pulls the code and tries to run it
- It’s deployed to staging or production
- A tester runs it on their environment

Why does this happen?

Different environments mean different:

- OS versions (Windows vs Linux vs macOS)
- Installed dependencies/libraries
- Language versions (Python 3.10 vs 3.11, Node.js v16 vs v18)
- Environment variables and configs
- Tools and binaries in `PATH`

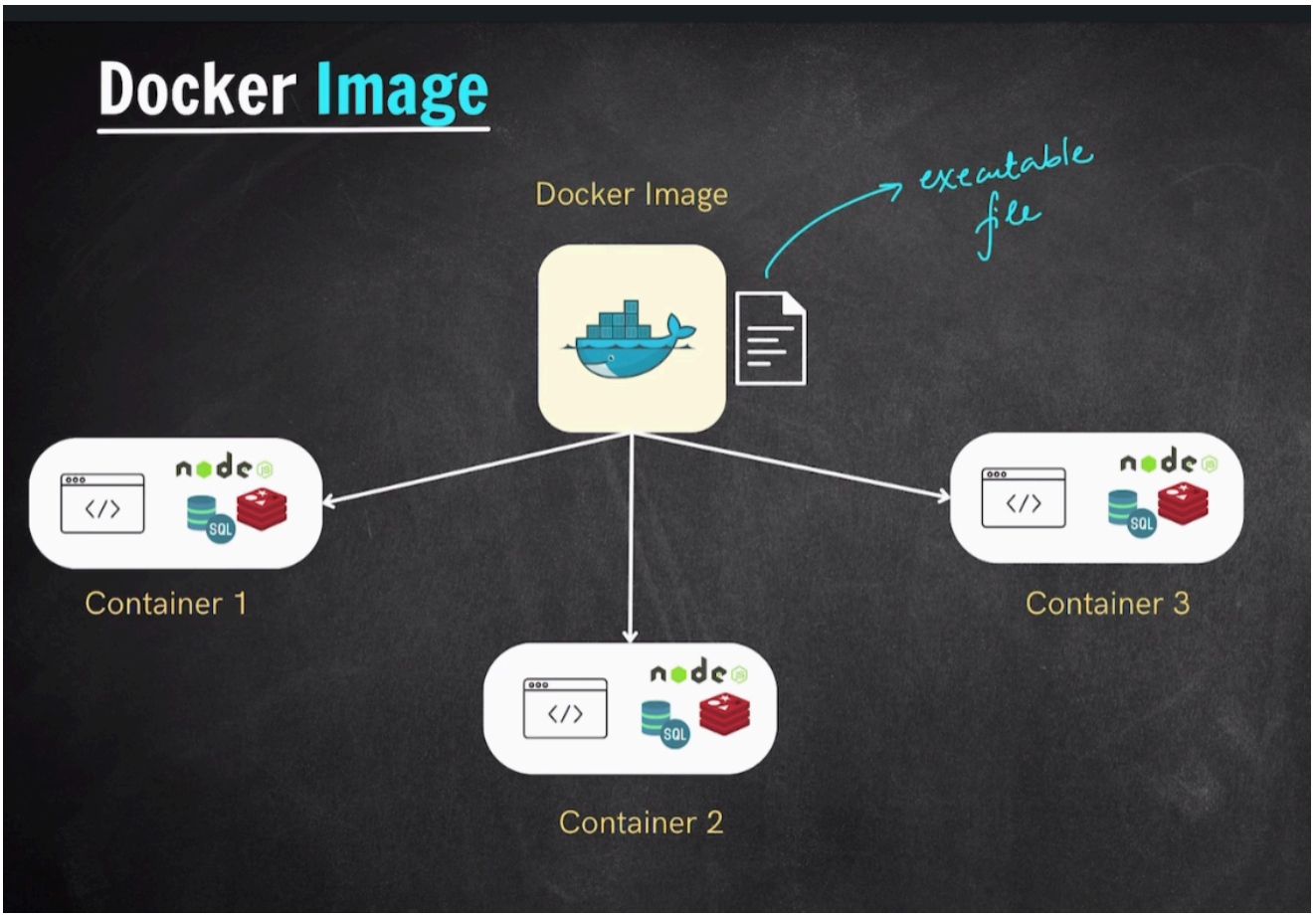
Docker Container:



Basically, it is a container that holds everything your code needs to run. It’s like a lightweight box that contains your application, along with all its dependencies — like the runtime, libraries, and tools — so you don’t have to worry about what's installed on the system where you run it.

Example: Imagine you built a Python web app that needs Python 3.11 and Flask. On your machine, it works perfectly. But when your friend tries to run it, it fails because they have Python 3.8 and no Flask installed. With Docker, you create a **container** that includes your app, Python 3.11, Flask, and all necessary dependencies. Now, your friend can run the container on any system — Windows, macOS, Linux — and it **just works**, no setup needed.
💬 **It’s like sending your app inside a ready-to-use box that works anywhere.**

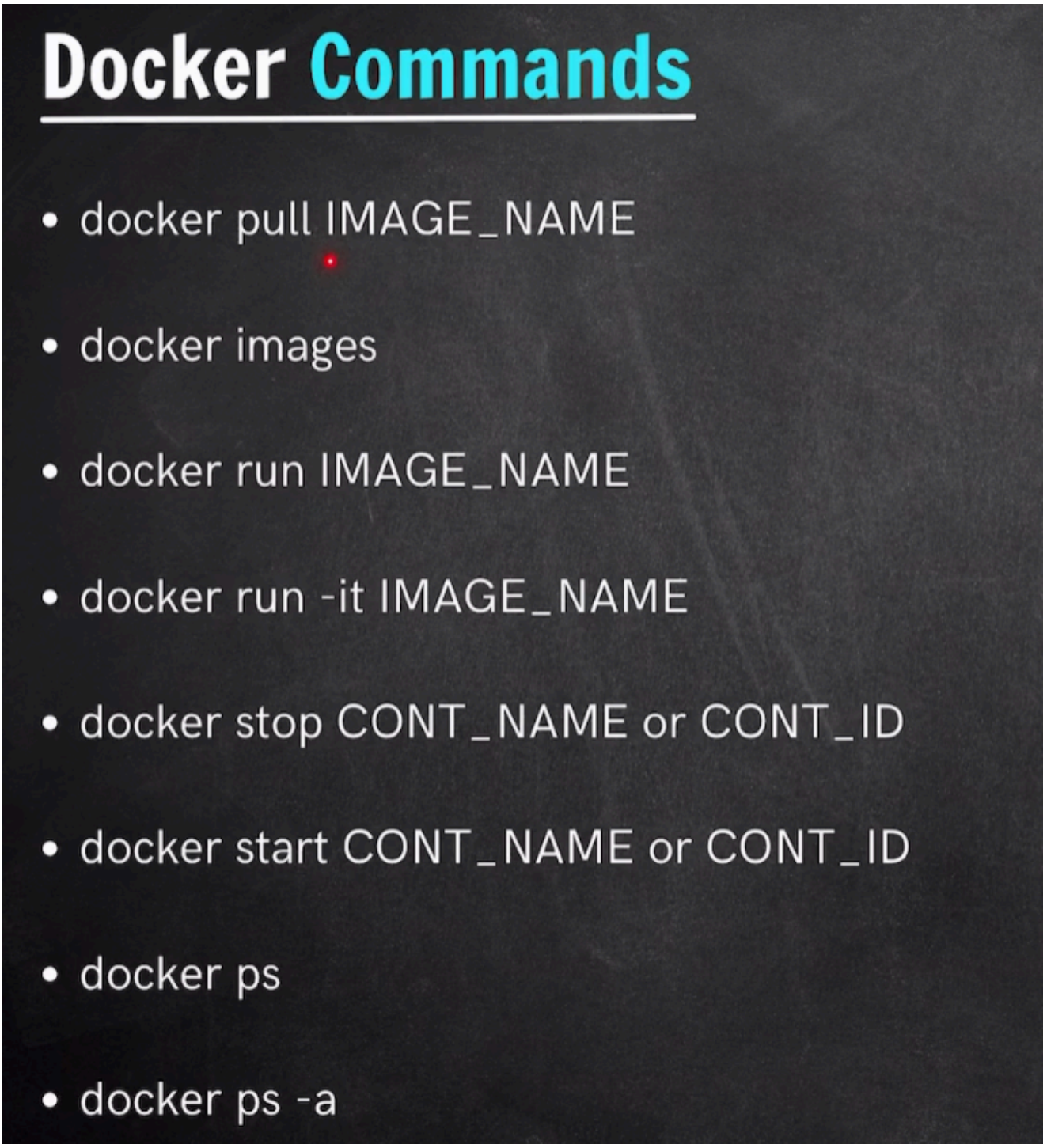
Docker Image:



A Docker image is like a class in OOP — it defines the blueprint of how the container should behave, including the code, dependencies, and environment setup. A Docker container is like an object — it's a running instance created from that image. Just like you can create multiple objects from a class, you can run multiple containers from the same image, each working independently.

A Docker image is made of multiple read-only layers, each created from a line in the Dockerfile. This makes images efficient, fast to build, and easy to share.

Docker Command:



`docker pull hello-world`

- This command **downloads** the official `hello-world` Docker image from Docker Hub (the central Docker image repository).
- It gets the image onto your local machine so you can run it.

`docker images`

- This command **lists all Docker images** that are currently downloaded and stored on your local machine.
- It shows image names, tags, image IDs, creation dates, and sizes.
- Helps you see what images you have available to create containers from.

```
C:\Users\Lenovo>docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
Digest: sha256:ec153840d1e635ac434fab5e377081f17e0e15afab27beb3f726c3265039cfff
Status: Image is up to date for hello-world:latest
docker.io/library/hello-world:latest

C:\Users\Lenovo>docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
hello-world   latest    ec153840d1e6   6 months ago   20.4kB
```

`docker run hello-world`

- This command **creates and starts a container** from the `hello-world` image.
- If you don't have the image locally, Docker will **pull it automatically** from Docker Hub first.
- The `hello-world` container runs a tiny program that prints a welcome message to confirm Docker is working properly.
- After it runs, the container usually **exits immediately** because the program finishes.

`docker ps -a`

- This command **lists all Docker containers** on your system, including:
  - Running containers
  - Stopped/exited containers
- It shows container IDs, names, statuses, and which image they're from.
- Helps you see everything that has run or is running.

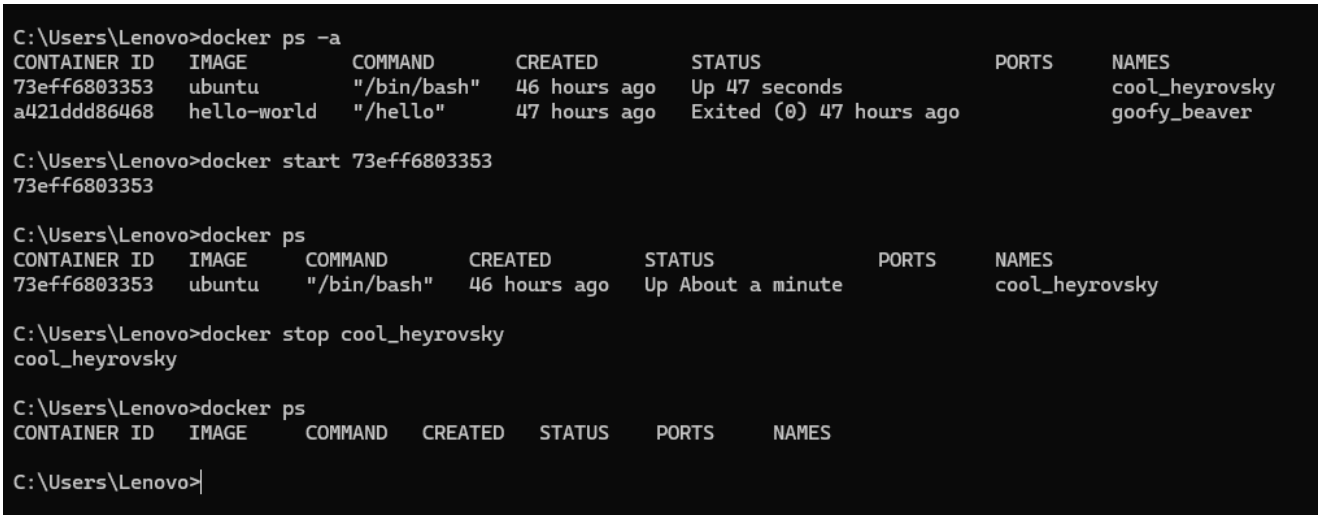
```
C:\Users\Lenovo>docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/


C:\Users\Lenovo>docker ps -a
CONTAINER ID   IMAGE         COMMAND    CREATED          STATUS                      PORTS     NAMES
a421ddd86468   hello-world   "/hello"   22 seconds ago   Exited (0) 21 seconds ago             goofy_beaver
```

`docker run -it Image_Name`

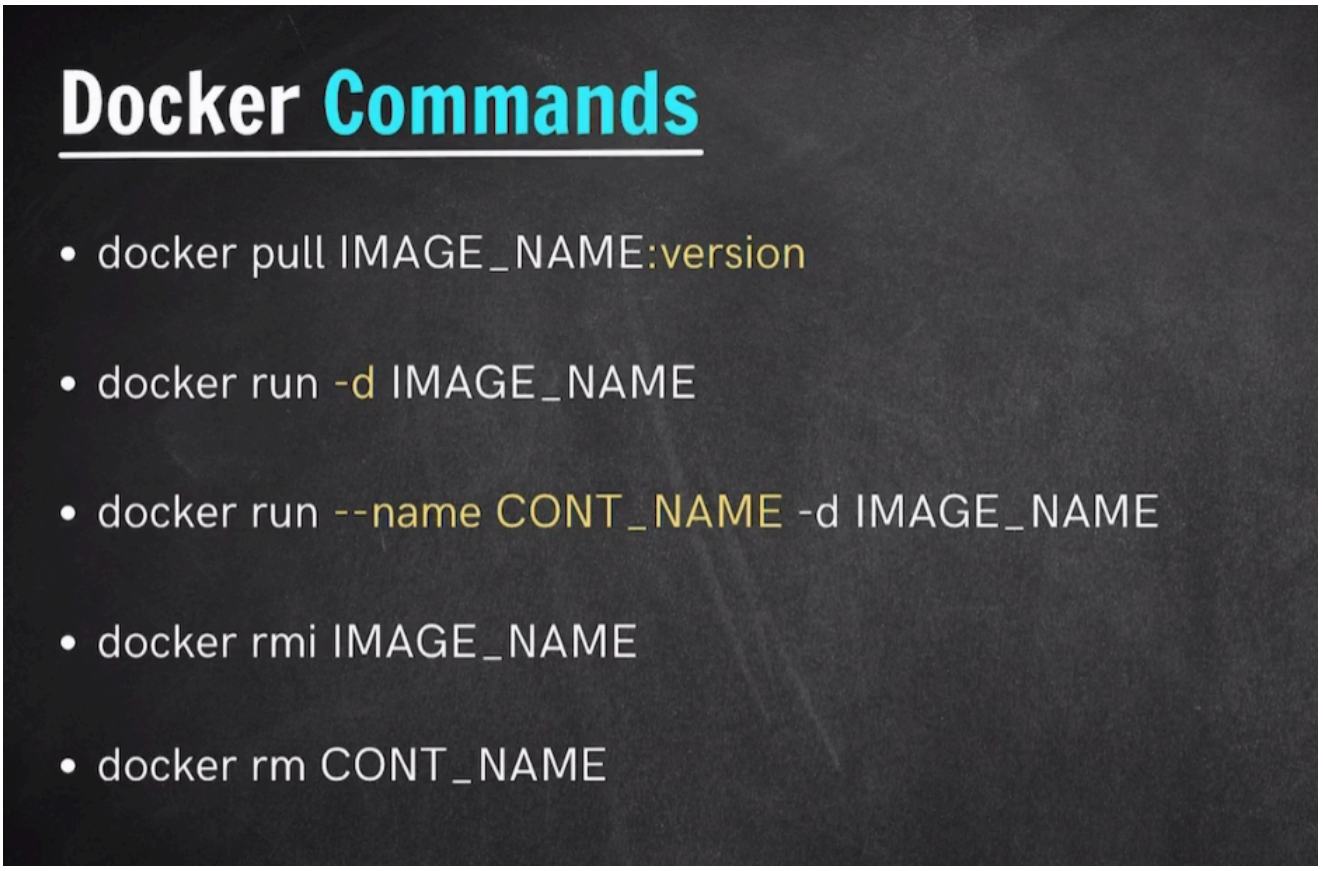
What It Means:

- `docker run` : Runs a new container
- `-it` :
  - `-i` = interactive mode (keeps STDIN open)
  - `-t` = allocates a pseudo-TTY (gives you a terminal)
- `ubuntu` : The Docker **image** you're running (in this case, the official Ubuntu base image)
- You’re now inside a brand-new **Ubuntu container**.
- This container behaves like a fresh Ubuntu Linux system.

`docker start container_Id or Names` & `docker stop container_Id or Names`



- It basically start a previously stopped container or Stops a running container gracefully with it's container_Id or Names.



`docker pull IMAGE_NAME:version`

- **Purpose**: Downloads a Docker image from Docker Hub or any Docker registry.
- `:version` is optional. If omitted, it pulls the `latest` version by default.

```
docker pull ubuntu:22.04
#If we dont add any specific verson it download latest version by default
```

`docker rmi IMAGE_NAME` & `docker rm CONT_NAME`

Both command i used to remove , rmi used to remove Image and rm is used to remove container.

```
docker rmi nginx => image
docker rm myweb => container
```

`docker run -d IMAGE_NAME`

- **Purpose**: Runs a container from the image in **detached mode** (runs in the background).
- `-d` stands for "detached".
- by default all the container we run is in attached mode.

Why we need to run the container in the detached mode?

when you run a Docker container in **attached mode**, it takes over your **terminal session**, meaning:
You *can't* do other things in that terminal while it's running. so if you Still Want to Run Other Things:

1. Open **another terminal tab/window**.
2. Or use **detach shortcut** (`Ctrl + P` then `Ctrl + Q`) to exit the attached container without stopping it.

Advantages of Detached Mode

- **Keeps your terminal free**
  - You can continue using your terminal for other tasks while the container runs in the background.
- **Ideal for long-running services**
  - Perfect for things like web servers, databases, APIs — services that should always stay up.
- **Good for automation**
  - In scripts, CI/CD pipelines, or deployment tools, you don't want the terminal stuck in a live session.
- **Useful in multi-container apps**
  - If you're running multiple services using `docker-compose` , each runs detached unless you want to debug.

When to use Detached Mode?

Use Case	Detached?
Running a **web server** (like `nginx`)	✅ Yes
Starting a **database** (like `mysql`)	✅ Yes
Deploying with **Docker Compose**	✅ Yes
Debugging, testing, or experimenting	❌ No
Need to **interact with a terminal**	❌ No

`docker run --name CONT_NAME -d IMAGE_NAME`

This command:

- **Runs a container** from a given Docker image.
- **Assigns a custom name** to the container (instead of random one).
- **Runs the container in detached mode**, i.e., in the background.

```
docker run --name my_nginx -d nginx
```

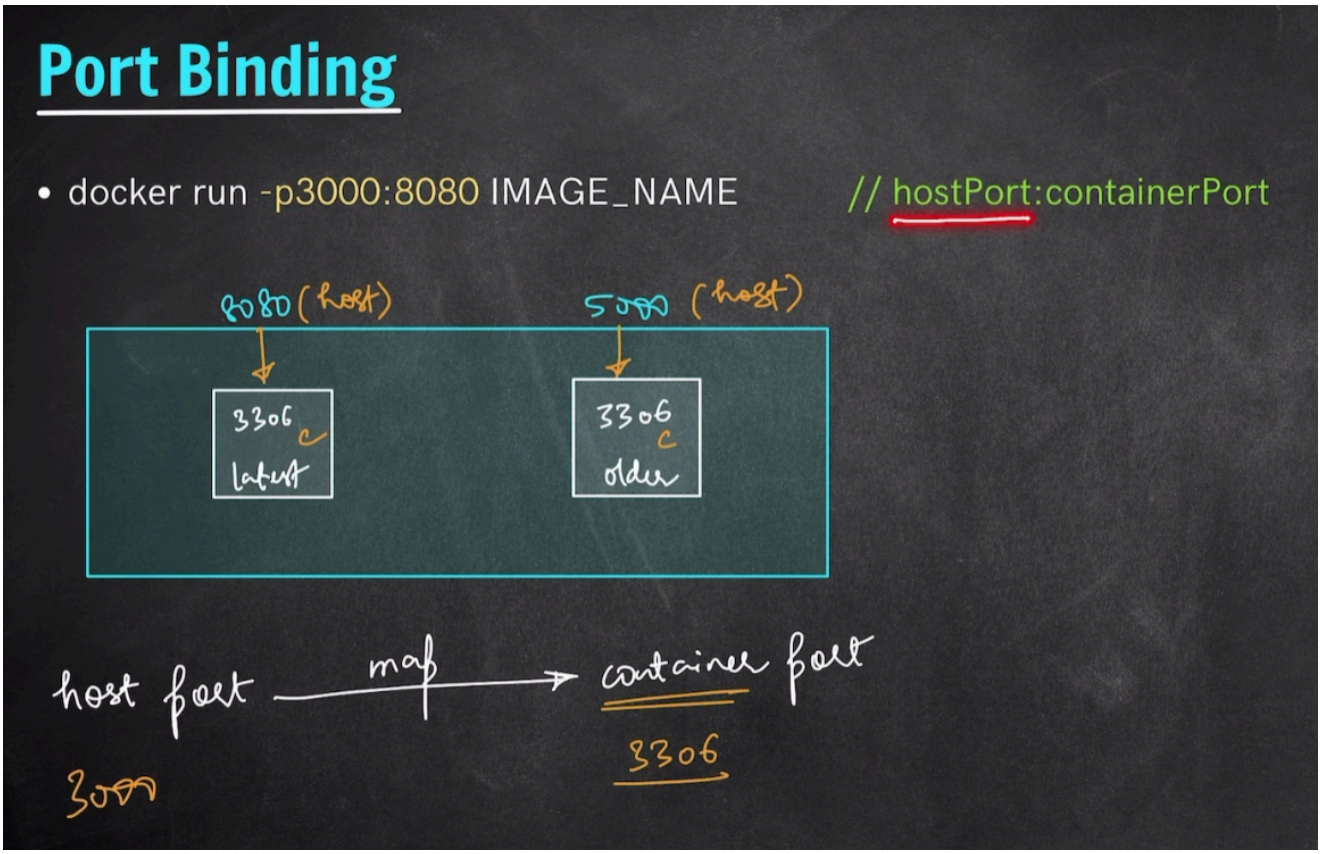
This will:

- Start an **nginx web server** in the background.
- Name the container **my_nginx**.
- Free up your terminal immediately.

#imp Note: If you are thinking something like When you use `docker run`, it always creates a **new container** from the specified image, even if one already exists. This can quickly lead to many unused containers piling up. Instead, after running a container once, you should use `docker start` to reuse an existing stopped container. If you need to run commands inside a running container, you can use `docker exec`. This approach avoids creating duplicates and helps keep your system clean and efficient. So, it's best to use `docker run` only when you truly need a new container.

- `docker run` = **create + start + (optionally attach)**
- `docker start` = **just start an existing container**
- `docker exec` = **run a command in an already running container**

Port Binding



Port binding means connecting a port **inside the container** to a port **on your host machine**, so services running inside the container can be accessed **from outside** (like your

browser or terminal).
By default: A Docker container runs **isolated** — its internal ports aren't exposed to your system or the internet. To access it, you must **bind** a container port to a host port.

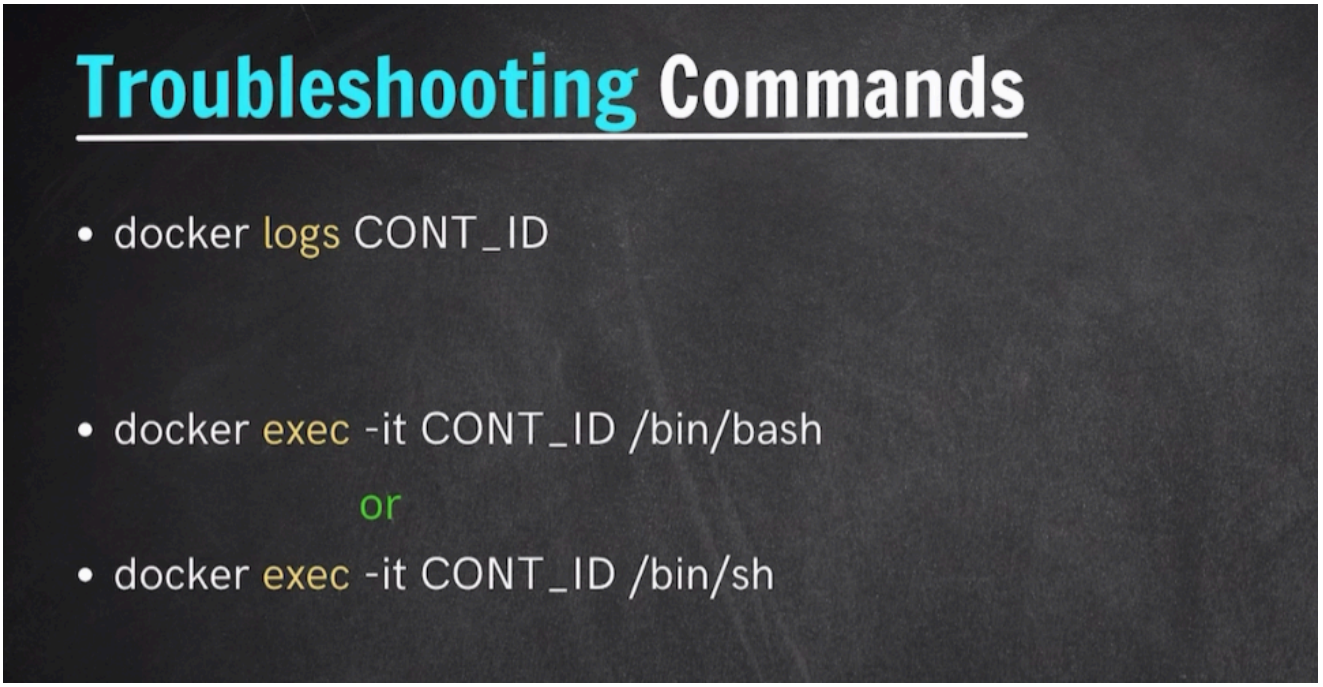
```
docker run -d --name mysql-portbinding -p8080:3306 -e
MYSQL_ROOT_PASSWORD=secret mysql:8.0.43-oraclelinux9
```

Port binding: Maps **port 8080 on your host** to **port 3306 in the container** (MySQL default port).

Troubleshooting Command:

A **log** is a **record of events** or **output messages** that show what the application (or container) is doing. In Docker, **logs tell you**:

- What's happening inside a container
- If there are any errors
- Debugging info
- Startup messages
- Output from your app (like a server starting, database queries, etc.)



`docker logs CONT_ID`

```
docker logs 65615e29539c
```

shows the **standard output (stdout)** and **standard error (stderr)** from inside the container. It helps you understand **what's happening** in the container, especially useful for

debugging.

`docker exec -it CONT_ID /bin/bash` or `docker exe -it CONT_ID /bin/sh`

```
C:\Users\Lenovo>docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED
STATUS             PORTS                                          NAMES
65615e29539c   mysql:8.0.43-oraclelinux9   "docker-entrypoint.s…"   59
minutes ago      Up 59 minutes      0.0.0.0:8080->3306/tcp, [::]:8080-
>3306/tcp   mysql-portbinding
e5588236dfcc   mysql:8.0.43-oraclelinux9   "docker-entrypoint.s…"   About an
hour ago   Up About an hour   3306/tcp, 33060/tcp
mysql-latest

C:\Users\Lenovo>docker exec -it mysql-latest /bin/bash
bash-5.1# env
MYSQL_MAJOR=8.0
HOSTNAME=e5588236dfcc
PWD=/
MYSQL_ROOT_PASSWORD=secret
HOME=/root
MYSQL_VERSION=8.0.43-1.el9
GOSU_VERSION=1.17
TERM=xterm
SHLVL=1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
MYSQL_SHELL_VERSION=8.0.43-1.el9
_=/usr/bin/env
bash-5.1#
```

This command opens an **interactive terminal session inside the container** named `mysql-latest`.

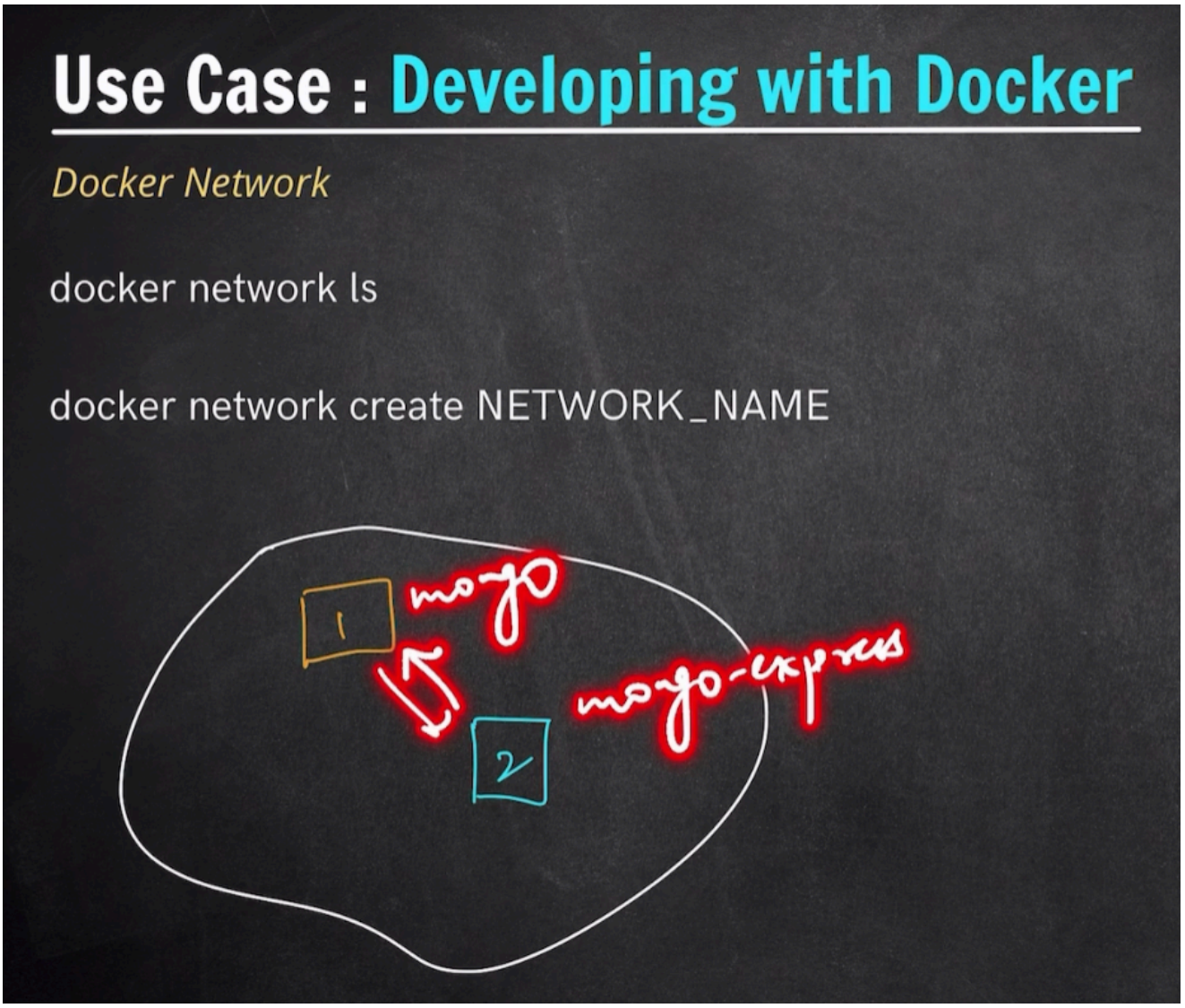
- `docker exec` = Run a command inside a running container
- `-it` = Interactive terminal mode (like opening a terminal window inside the container)
- `mysql-latest` = The name of the running container you're connecting to
- `/bin/bash` = The command you're running (starts the Bash shell)

Now you're inside the container as if you're inside a small Linux machine running MySQL. Also can use sh.

Command	Shell Type	Features	Availability
`/bin/bash`	Full-featured	Advanced	Might not exist in minimal containers
`/bin/sh`	Minimal	Basic	Almost always available

If `bash` doesn't work, try `sh` — it's the most reliable for getting a shell in **any** Docker container.

Docker Network:



A **Docker network** is a way for Docker containers to **communicate with each other** and with the **outside world** — just like computers connected to a real network.

Why is it useful?

- Lets containers **talk to each other by name** (e.g., a web app calling a database)
- Controls **which containers can see each other**
- Enables **custom IP ranges**, **isolation**, and **port mapping**
- Helps **link microservices** in a secure, scalable way

let's explain with example **common use case** for Docker networking: connecting multiple containers for development — in this case, likely a **MongoDB** container and an **Express.js**

app container. Use-case focus is on how Docker networking helps during development, especially when using microservices like databases and APIs.

```
C:\Users\Lenovo>docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
53af2668a0ee   bridge    bridge    local
ab4e9a2fcaf2   host      host      local
9ca6de32e7cf   none      null      local

C:\Users\Lenovo>docker network create mongo-network
0e47fe94f0e0c0a10eea597039659f1e34279212b5167ace594c999969f4ba22

C:\Users\Lenovo>docker network ls
NETWORK ID     NAME            DRIVER    SCOPE
53af2668a0ee   bridge          bridge    local
ab4e9a2fcaf2   host            host      local
0e47fe94f0e0   mongo-network   bridge    local
9ca6de32e7cf   none            null      local
```

now that you've successfully created the custom Docker network `mongo-network`, you **can and should** run both your MongoDB and mongo-express containers in that same network to enable them to communicate with each other.

Why use a custom Docker network like `mongo-network`?

1. **Easier Communication**:
   Containers can talk to each other using **names** (like `mongo`) instead of IP addresses.
2. **More Secure**:
   Only containers in the same network can talk to each other — others can't access them.
3. **Organized Setup**:
   Keeps related containers (like `mongo` and `mongo-express`) **grouped together**.
4. **Automatic Name Resolution**:
   Docker handles the connection between containers for you.

Use Case : Developing with Docker

set up mongo & mongo-express

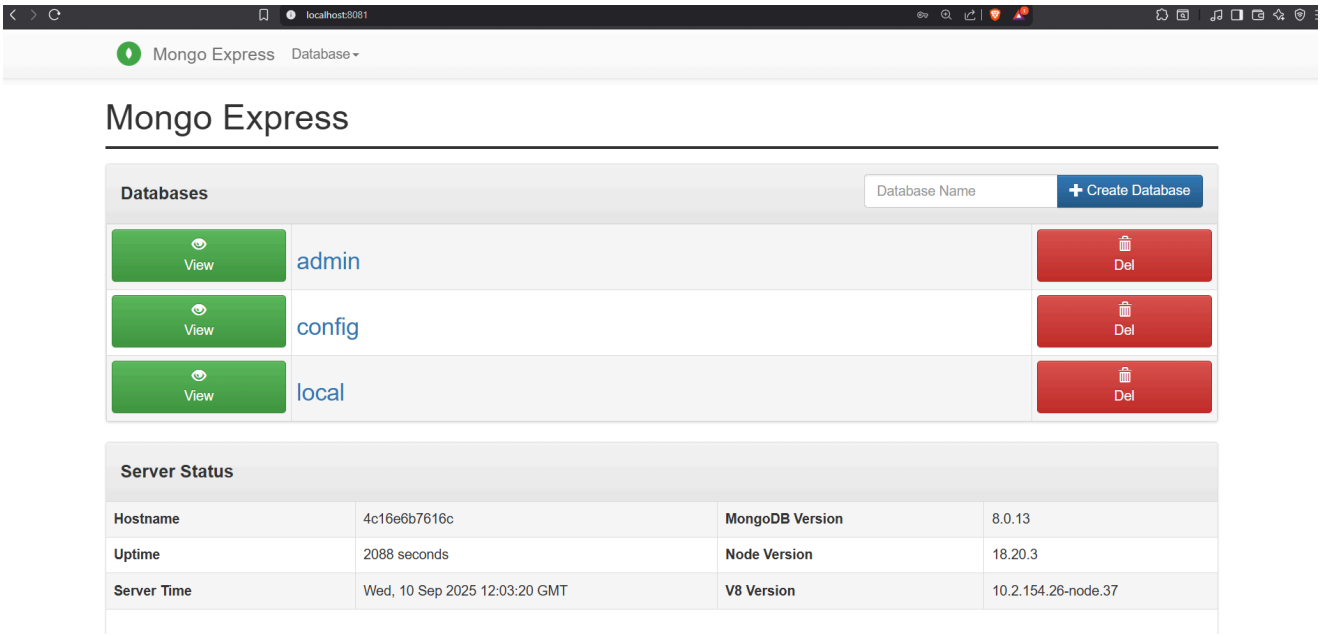
```
C:\Users\Lenovo>docker run -d ^
More? -p27017:27017 ^
More? -e MONGO_INITDB_ROOT_USERNAME=admin_ajay ^
More? -e MONGO_INITDB_ROOT_PASSWORD=normal_password ^
More? --name monog ^
More? --network mongo-network ^
```

```
More? mongo
Unable to find image 'mongo:latest' locally
latest: Pulling from library/mongo
96b1c4e20b5d: Download complete
96b1c4e20b5d: Pull complete
a20c427c2ead: Pull complete
930dfd6340eb: Downloading [==========================================>
]  243.3MB930dfd6340eb: Pull complete
4ba0ed20cdee: Pull complete
76249c7cd503: Pull complete
da3134f8e686: Pull complete
5777c09b20e7: Pull complete
Digest:
sha256:c75092233f998275c7b2c3942bb897994adb709bc5b9b7d043cc642ec521b6c7
Status: Downloaded newer image for mongo:latest
4c16e6b7616c9e21bd144ea045bec00eda4239280867b6567164b8f0c480ff24
```

```
C:\Users\Lenovo>docker run -d ^
More? -p8081:8081 ^
More? --network mongo-network ^
More? --name mongo-express ^
More? -e ME_CONFIG_MONGODB_ADMINUSERNAME=admin_ajay ^
More? -e ME_CONFIG_MONGODB_ADMINPASSWORD=normal_password ^
More? -e
ME_CONFIG_MONGODB_URL="mongodb://admin_ajay:normal_password@mongo:27017" ^
More? mongo-express
0d8382e672a832e37d889c7c7191216f8db8dd58fdc9a19f7213216e999aa662
```

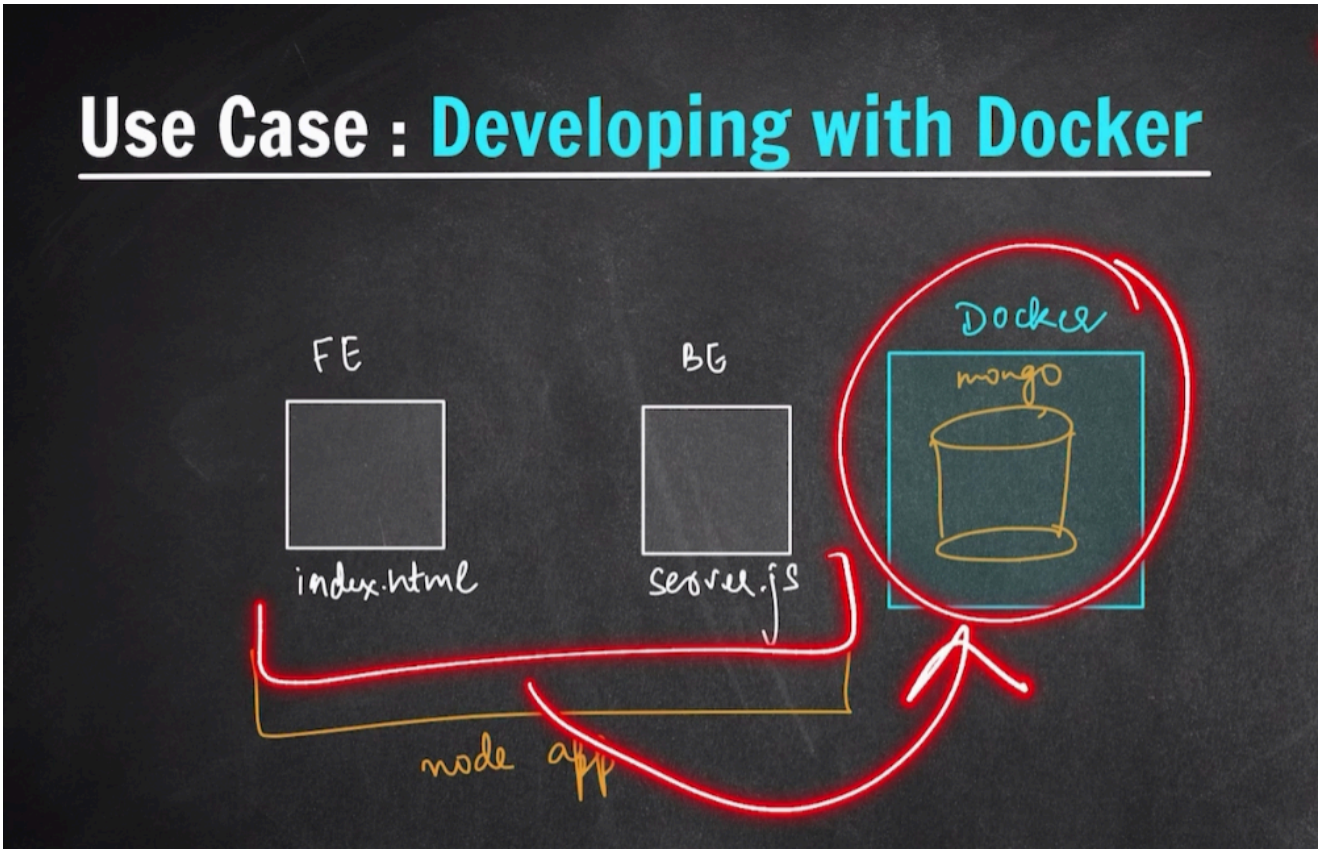
```
C:\Users\Lenovo>docker ps
CONTAINER ID   IMAGE           COMMAND                  CREATED
STATUS          PORTS                                           NAMES
0d8382e672a8   mongo-express   "/sbin/tini -- /dock…"   9 seconds ago    Up
8 seconds    0.0.0.0:8081->8081/tcp, [::]:8081->8081/tcp       mongo-express
4c16e6b7616c   mongo           "docker-entrypoint.s…"   23 minutes ago   Up
23 minutes   0.0.0.0:27017->27017/tcp, [::]:27017->27017/tcp   monog
```

so it is successfully run now check http://localhost:8081/ now we get sign in option add user & pass, then we got in:

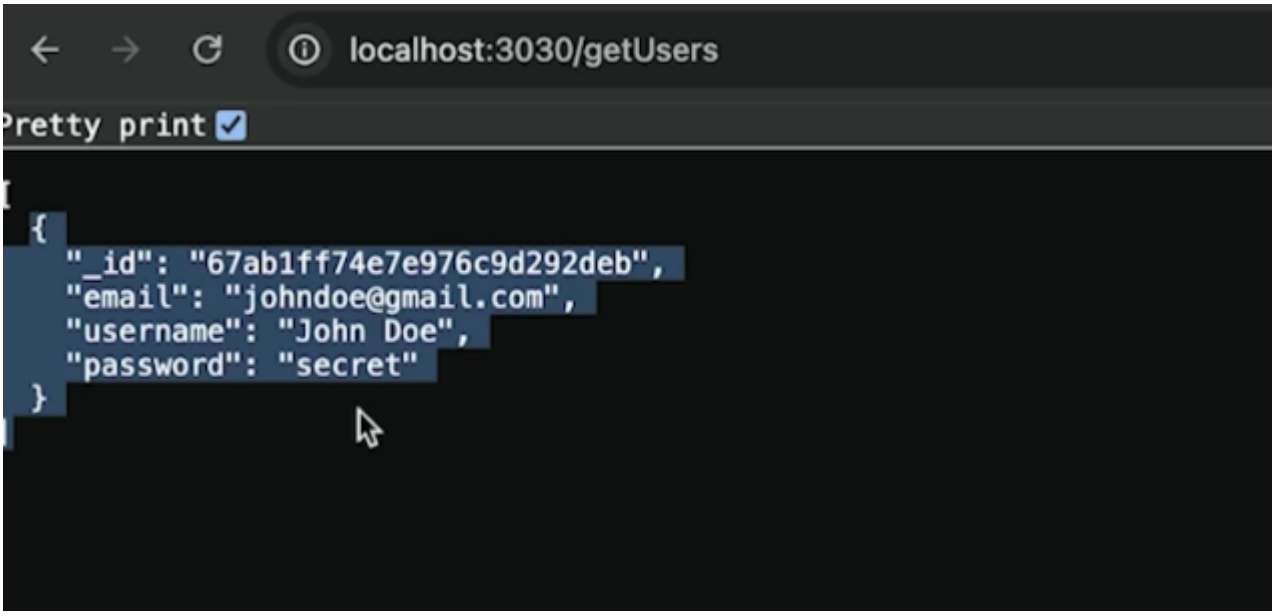


after that create a user database and create a document

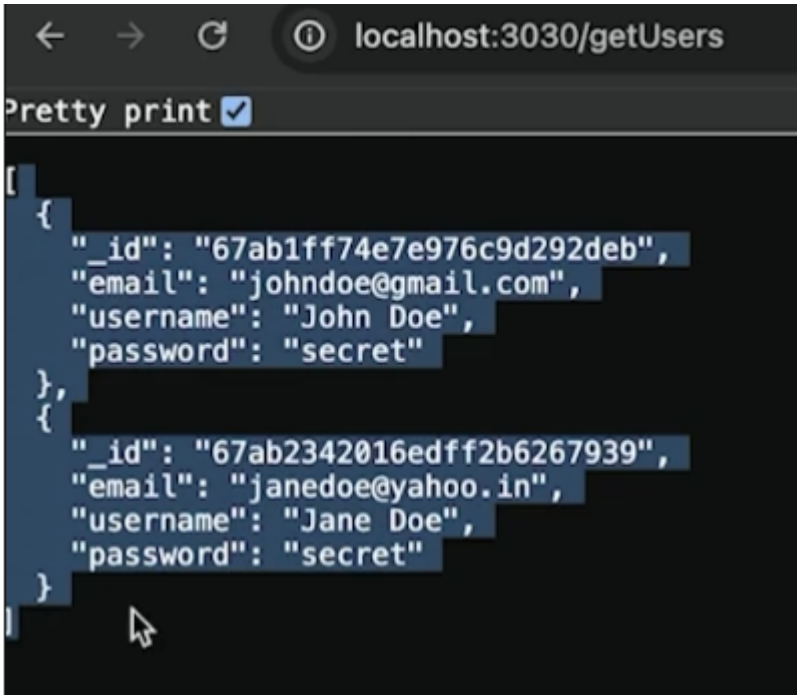
so now we can connect a server by just connecting a mongodb by http://localhost/3030/getUsers



so we get the data:

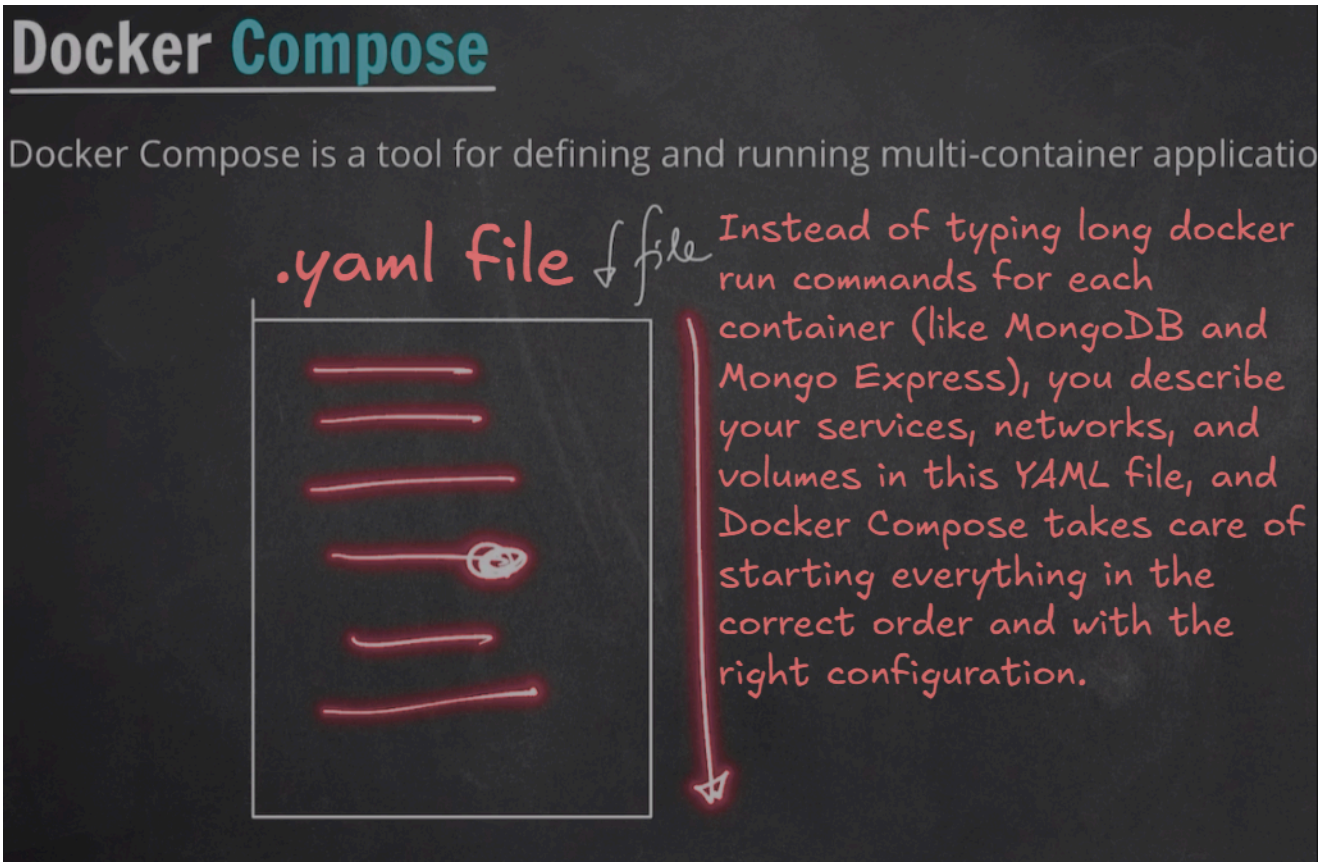


and we can also post the request form and add a new user:



by this way you can set up the db in a real project.

Docker Compose



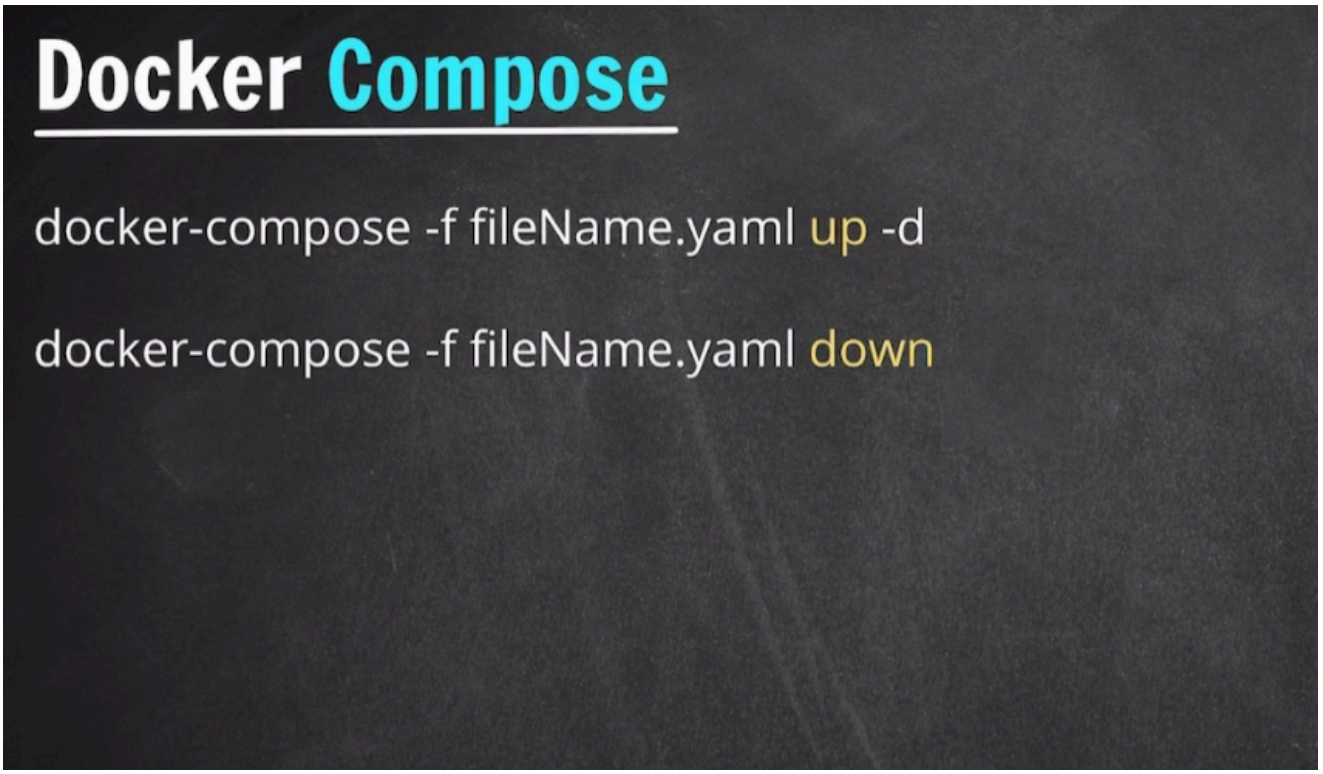
so that we create a yaml file,

sql injection.md | mongo.yaml

C: > Users > Lenovo > Documents > mongo.yaml > {} services > {} mongo-express > {} environment

```yaml
version: "3.8"

services:
  mongo:
    image: mongo
    ports:
      - 27017:27017
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin_ajay
      MONGO_INITDB_ROOT_PASSWORD: normal_password

  mongo-express:
    image: mongo-express
    ports:
      - 8081:8081
    environment:
      ME_CONFIG_MONGODB_ADMINUSERNAME: admin_ajay
      ME_CONFIG_MONGODB_ADMINPASSWORD: normal_password
      ME_CONFIG_MONGODB_URL: mongodb://admin_ajay:normal_password@mongo:27017

```

some command used by docker compose:



Now running the created file :

```
C:\Users\Lenovo\Documents>docker compose -f mongo.yaml up -d
time="2025-09-10T19:42:25+05:45" level=warning
msg="C:\\Users\\Lenovo\\Documents\\mongo.yaml: the attribute `version` is
obsolete, it will be ignored, please remove it to avoid potential confusion"
[+] Running 18/18
 ✓ mongo-express Pulled
25.8s
   ✓ 45b24ec126f9 Pull complete
0.9s
   ✓ 9f7f59574f7d Pull complete
19.4s
   ✓ 619be1103602 Pull complete
3.9s
   ✓ 88f4f8a6bc8d Pull complete
18.3s
   ✓ 0bf3571b6cd7 Pull complete
1.6s
   ✓ d8305ae32c95 Pull complete
2.4s
   ✓ 5189255e31c8 Pull complete
3.0s
   ✓ 7e9a007eb24b Pull complete
18.2s
 ✓ mongo Pulled
60.2s
   ✓ 96b1c4e20b5d Pull complete
0.9s
   ✓ a20c427c2ead Pull complete
```

```
1.3s
   ✓ 4ba0ed20cdee Pull complete
15.7s
   ✓ da3134f8e686 Pull complete
1.6s
   ✓ 930dfd6340eb Pull complete
54.7s
   ✓ 5777c09b20e7 Pull complete
1.6s
   ✓ 76249c7cd503 Pull complete
15.6s
   ✓ a6ccce675b91 Pull complete
1.8s
[+] Running 3/3
 ✓ Network documents_default            Created
0.0s
 ✓ Container documents-mongo-1          Started
0.9s
 ✓ Container documents-mongo-express-1  Started
```

it is successfully run :

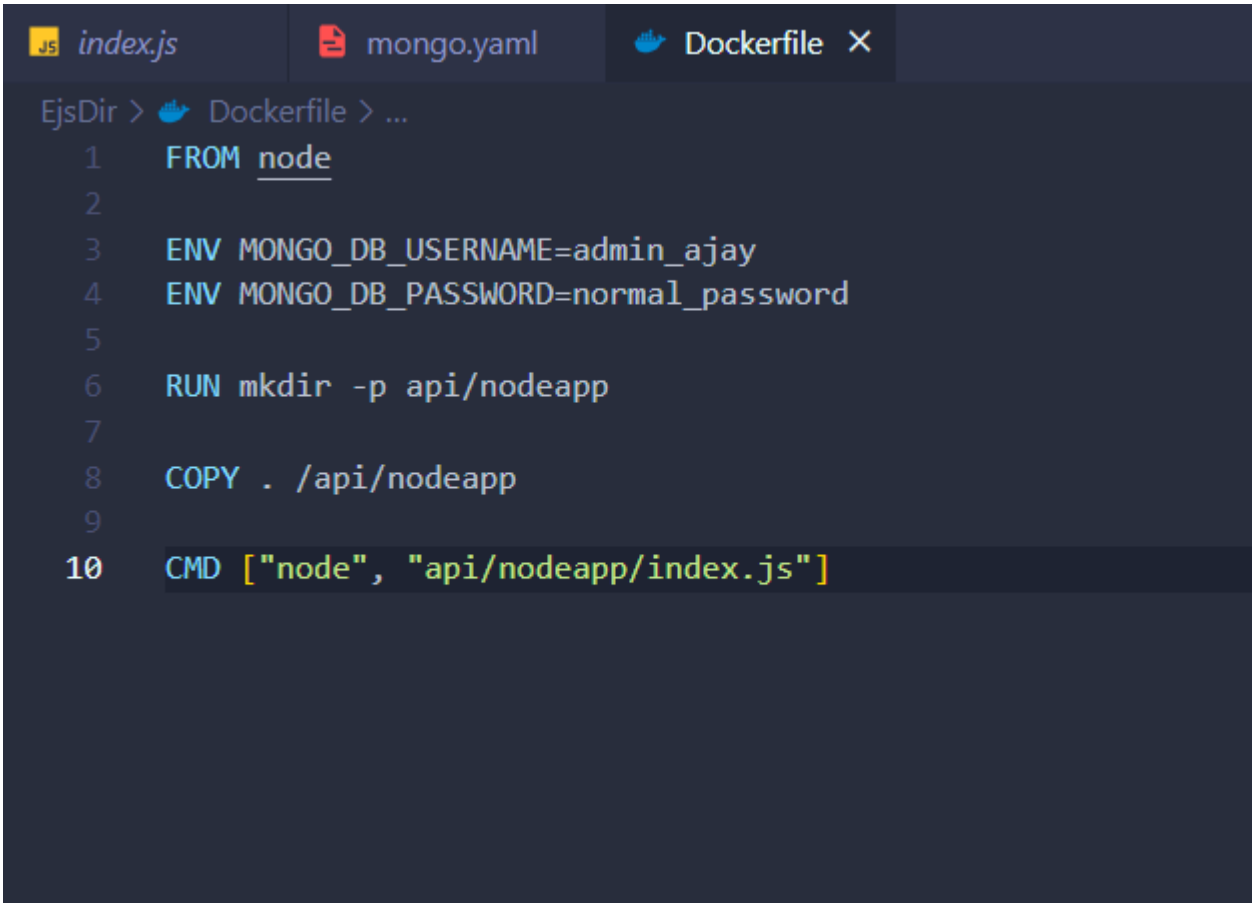
```
C:\Users\Lenovo\Documents>docker ps
CONTAINER ID   IMAGE           COMMAND                  CREATED
STATUS              PORTS                                         NAMES
f5b5d03a170d   mongo-express   "/sbin/tini -- /dock…"   About a minute ago
Up About a minute   0.0.0.0:8081->8081/tcp, [::]:8081->8081/tcp
documents-mongo-express-1
8700b492e0f7   mongo           "docker-entrypoint.s…"   About a minute ago
Up About a minute   0.0.0.0:27017->27017/tcp, [::]:27017->27017/tcp
documents-mongo-1

C:\Users\Lenovo\Documents>docker network ls
NETWORK ID     NAME                DRIVER    SCOPE
68a1398cebac   bridge              bridge    local
a5cd83d40dcb   documents_default   bridge    local
ab4e9a2fcaf2   host                host      local
0e47fe94f0e0   mongo-network       bridge    local
9ca6de32e7cf   none                null      local
```

You no longer need the old manual `docker run` commands—using Docker Compose makes managing your containers **much easier**, including networks, dependencies, and ports.

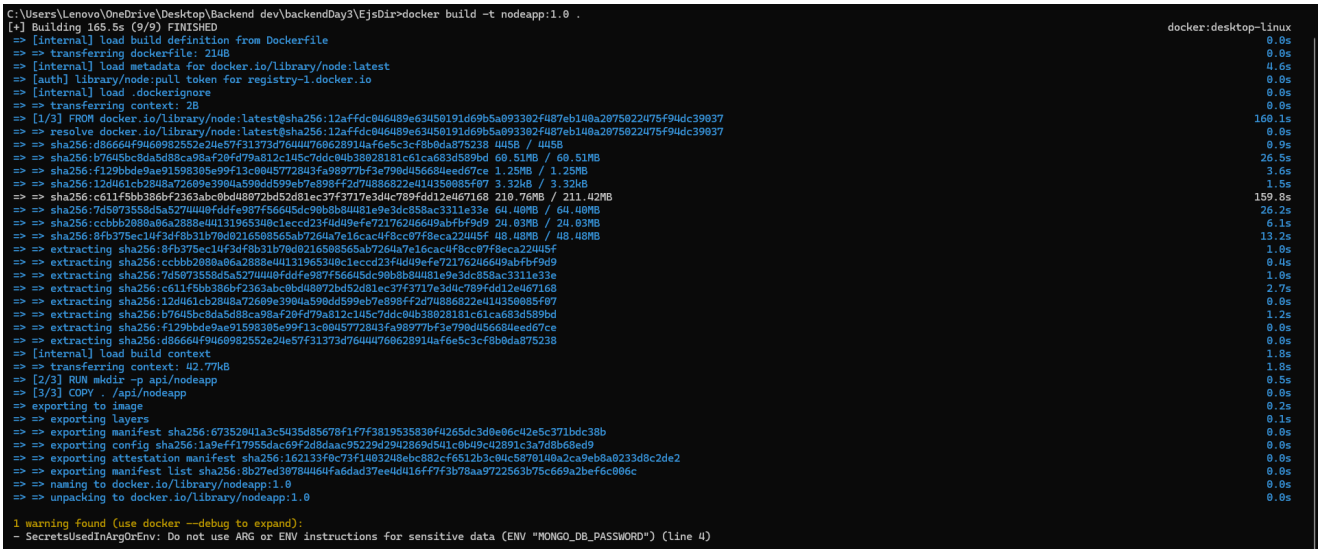
Dockerizing our app

source: [https://docs.docker.com/get-started/docker-concepts/building-images/writing-a-dockerfile/]

at first we need to create a docker file :



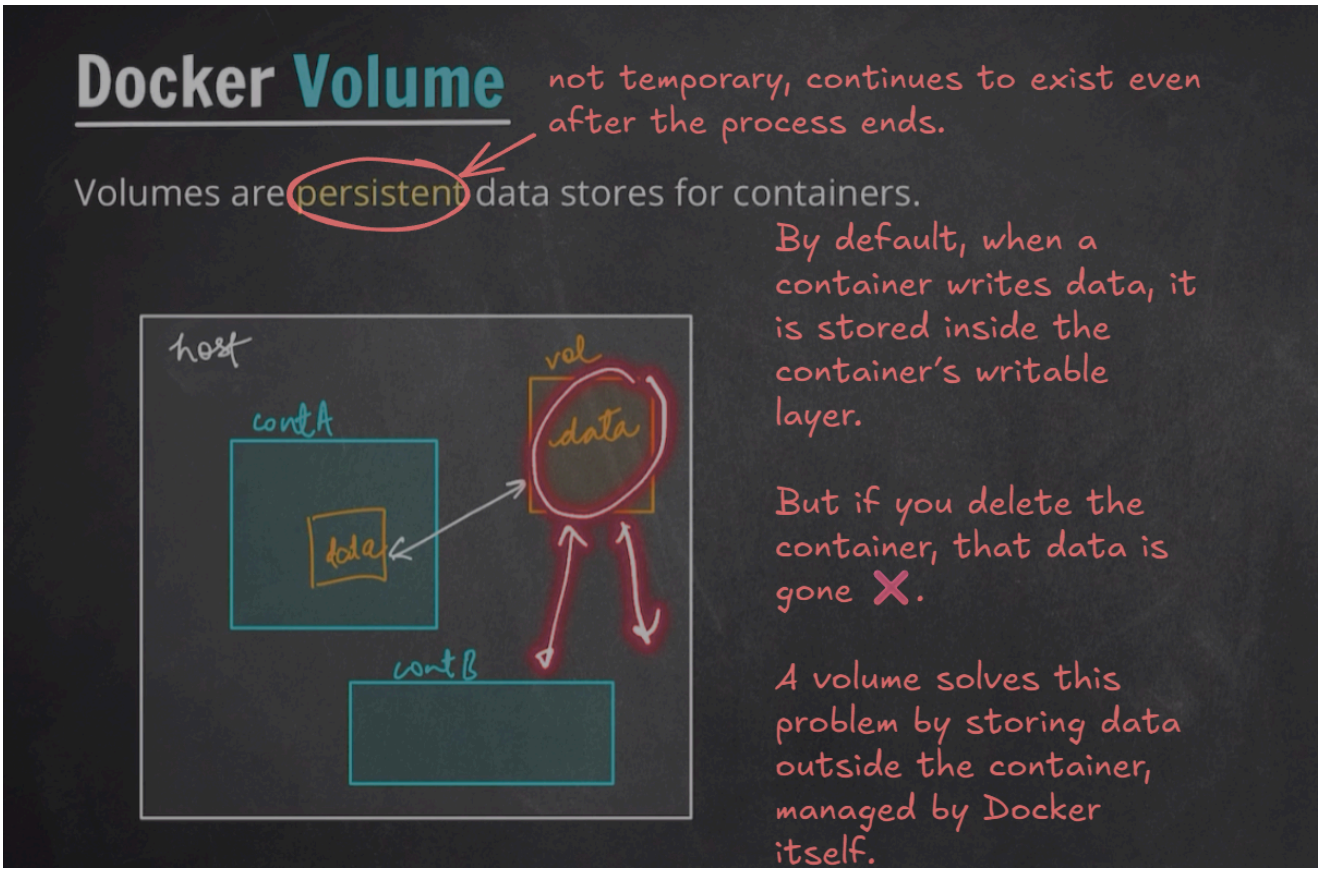
if we don't include node modules in it so we need to add RUN npm install in the Dockerfile then run it:



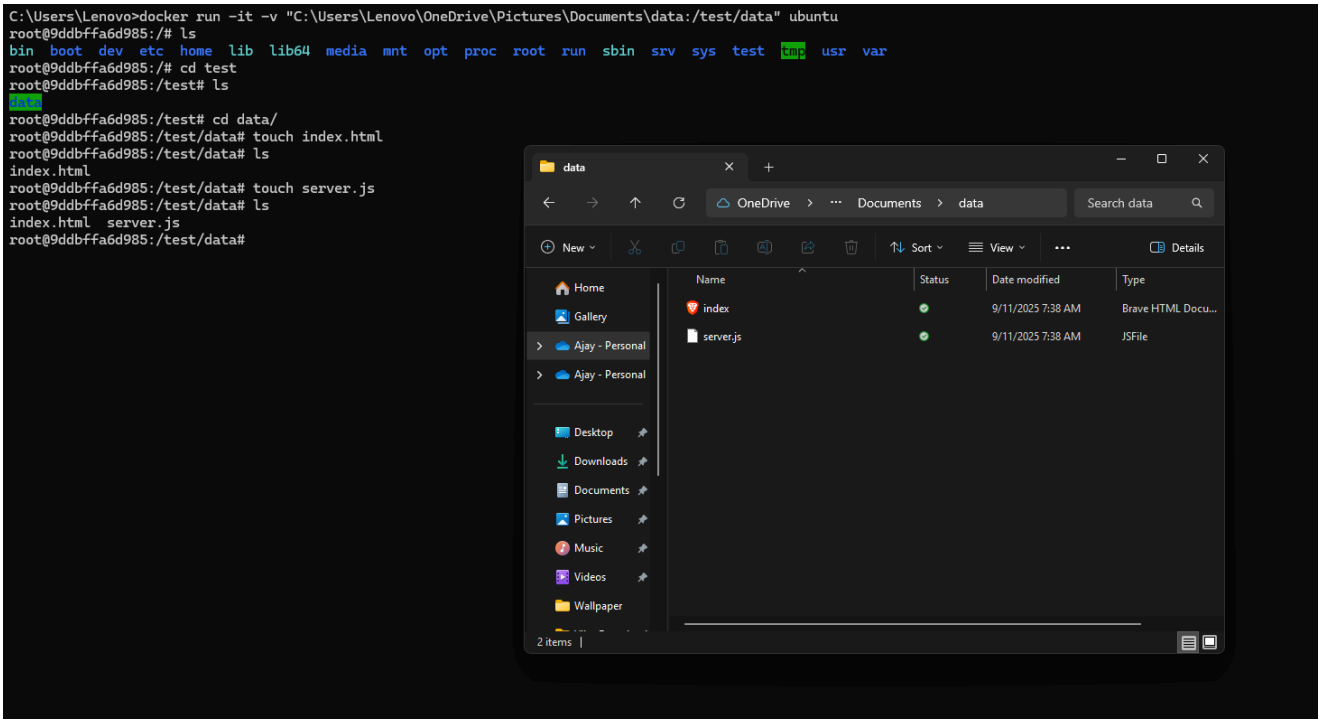
now we create a successfully a docker image of ower own project

---

Docker Volume



now let's see the example of docker volume:




```
docker run -it -v
"C:\Users\Lenovo\OneDrive\Pictures\Documents\data:/test/data" ubuntu
```

- `-it` → interactive mode (so you can use shell inside container).
- `-v "host_path:container_path"` → **bind mount**
  - **Host path**: `C:\Users\Lenovo\OneDrive\Pictures\Documents\data` (your Windows folder).

    - **Container path**: `/test/data` (Ubuntu folder inside container).
- `ubuntu` → container runs Ubuntu image.

So: anything inside `C:\Users\Lenovo\OneDrive\Pictures\Documents\data` ↔ is mirrored into `/test/data` inside the container.

What happened?

when we create a file name index.html and server.js inside the ubuntu At the **same time**, Windows File Explorer shows the exact same files (`index.html` and `server.js`) inside `Documents\data` .
This proves the **volume mount is working** — both the container and your Windows system share the same folder.

Why this is powerful?

- **Persistence** → If you stop/remove the container, your files (`index.html` , `server.js`) are still on your Windows machine.
- **Collaboration** → Multiple containers can share the same folder.
- **Real-time sync** → Edit code in Windows (`server.js`), run it immediately inside the container.
- If You remove the **Ubuntu container** Your host `Documents\data` → still there.
- If You remove the **Ubuntu image** Volumes / bind mounts (like your `C:\Users\Lenovo\...data`) → still safe.
- But if you remove the file from the docker (rm index.html server.js) Since `/test/data` is a **bind mount**, the deletion happens in your **Windows folder too**.

Some Commands in docker volume:

1. docker volume ls

is used to **list all Docker volumes** that exist on your system.

```
C:\Users\Lenovo>docker volume ls
DRIVER    VOLUME NAME
local     8fa2ac748c9e40ed64fc0055fff392460055f740638112c30b25bd7743748132
local     8fa7a83e33a1254921b466cfd76acce064fe39ce42c23fef51a9253dbe284938
local     304e5a12c14dad939c0ed52d4ecbfc2e107787044eb89d6e409121a12fe36dd1
local     c625257def7aadf2d92306e2c374507467ffad4908b87fb4e5bcfe59ad1ed6d4
local     da53608f56866126dff5e30c652aeb1c51aecbc6ba1f62f2d152310aba22f7d4
local     e23ee2ace062dcc5a8ca7feeea9bf8422c545cc2a224bb606bdcc7a3e9dfcdb4
```

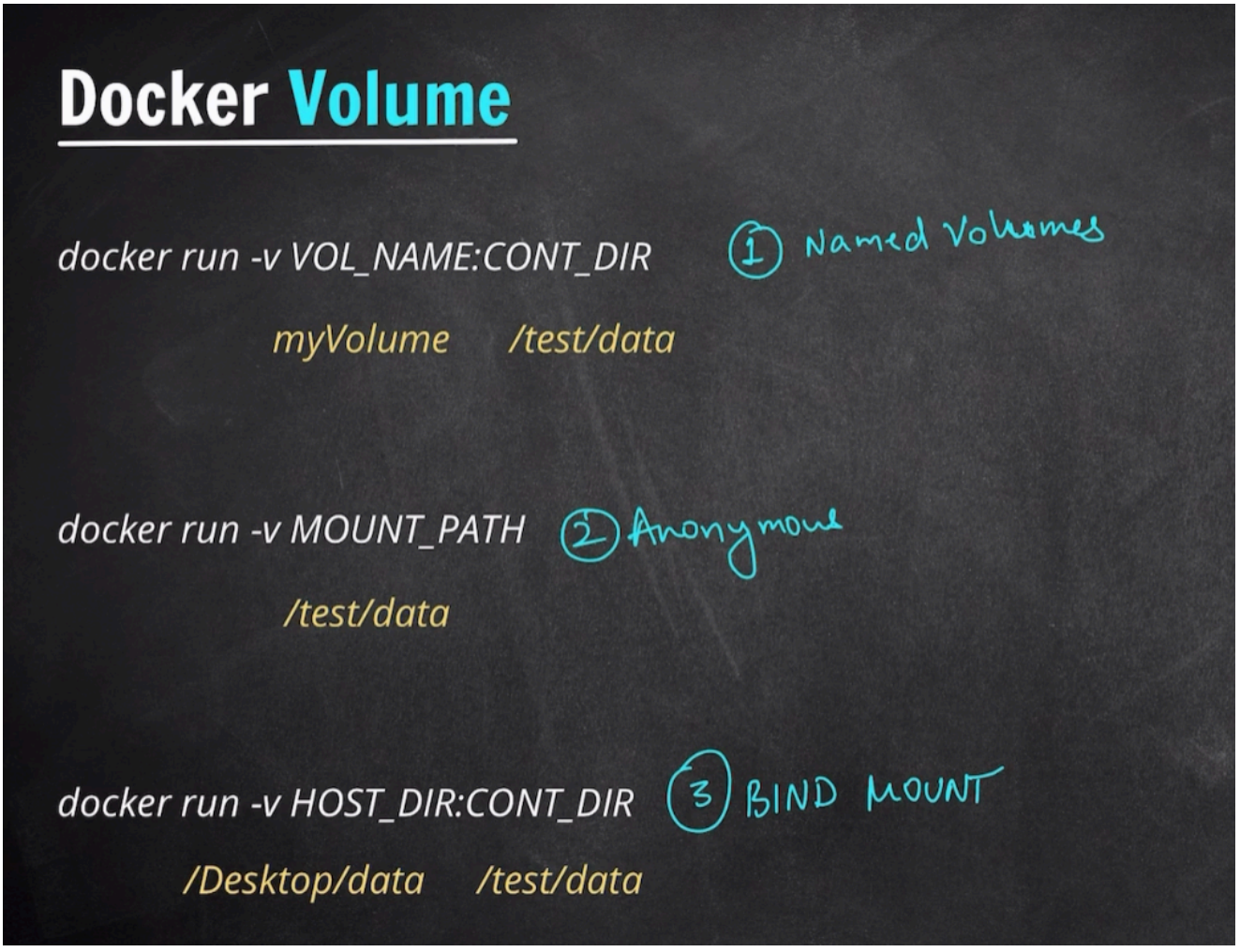
2. docker volume create myvolume

creates a new volume named myvolume

```
C:\Users\Lenovo>docker volume create myvolume
myvolume

C:\Users\Lenovo>docker volume rm myvolume
myvolume //revove the myvolume
```

when we create a volume in which place does it created in windows and linux?

in linux/mac => /var/lib/docker/volumes/
in windows => C:\ProgramData\Docker\volumes\



Type	Defined by user?	Host location	Good for…
Named	✅ Yes	Docker-managed (`/var/lib/docker/volumes/...`)	Databases, persistent app data
Anonymous	❌ No	Same as above	Temporary storage
Bind Mount	✅ Host path	Exactly where you say	Development, host ↔ container sync

So:

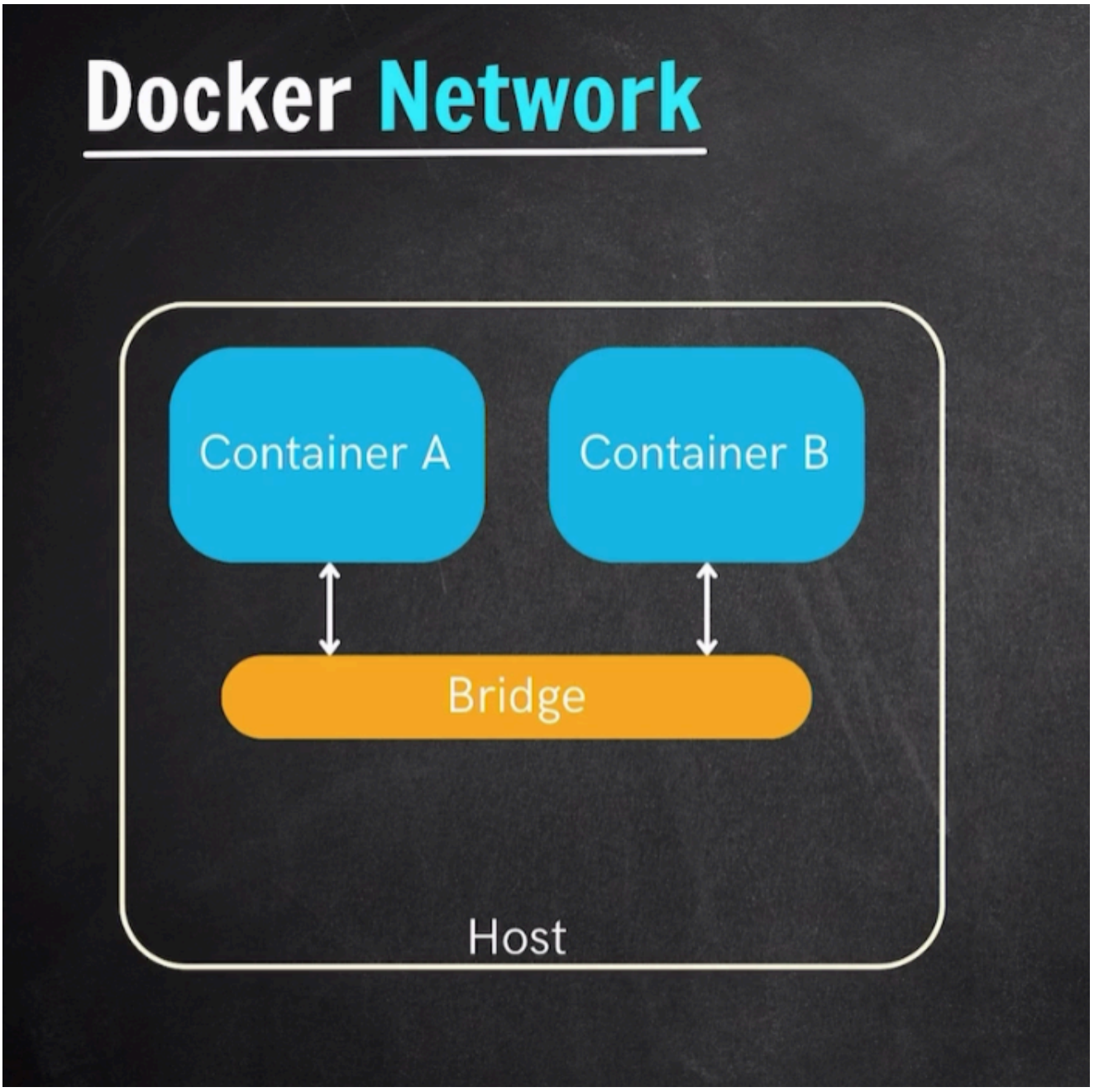
- Use **named volumes** for production persistence.
- Use **anonymous volumes** for quick throwaway containers.
- Use **bind mounts** for development when you want host files inside container.

3. docker volume prune

- Deletes **all unused volumes** from your system.
- "Unused" = volumes **not currently attached to any container**.
- To **clean up disk space** (volumes can get big, especially databases/logs).
- After experimenting with containers that left behind random/anonymous volumes.
- **Be careful**: once removed, the data inside those volumes is gone forever (unless you had a backup).

Docker Network

- A **network** is a virtual LAN (local area network) managed by Docker.
- Containers on the same network can **communicate by name** (not just IP).
- You can **isolate containers** by putting them in different networks.



1. Bridge

The default network driver. If you don't specify a driver, this is the type of network you are creating. Bridge networks are commonly used when your application runs in a container that needs to communicate with other containers on the same host.

2. host

Remove network isolation between the container and the Docker host, and use the host's networking directly.

3. none

Completely isolate a container from the host and other containers. `none` is not available for Swarm services.

Resource[https://docs.docker.com/engine/network/drivers/]