Docker

(https://projects.100xdevs.com/tracks/docker-2/docker-2-15)

docker run , execute the cmd

Layers in Docker

In Docker, layers are a fundamental part of the image architecture that allows Docker to be efficient, fast, and portable. A Docker image is essentially built up from a series of layers, each representing a set of differences from the previous layer.

How layers are made -

1. Base Layer: The starting point of an image, typically an operating system (OS) like Ubuntu, Alpine, or any other base image specified in a Dockerfile.
2. Instruction Layers: Each command in a Dockerfile creates a new layer in the image. These include instructions like **RUN**, **COPY**, which modify the filesystem by installing packages, copying files from the host to the container, or making other changes. Each of these modifications creates a new layer on top of the base layer.
3. Reusable & Shareable: Layers are cached and reusable across different images, which makes building and sharing images more efficient. If multiple images are built from the same base image or share common instructions, they can reuse the same layers, reducing storage space and speeding up image downloads and builds.
4. Immutable: Once a layer is created, it cannot be changed. If a change is made, Docker creates a new layer that captures the difference. This immutability is key to Docker's reliability and performance, as unchanged layers can be shared across images and containers.

Layers Practically

It is for same Repo: https://github.com/nthapa000/dockerfilePractice

Dockerfile

```
Dockerfile
1  FROM node:20                                  Layer 1
2
3  WORKDIR /usr/src/app                          Layer 2
4
5  COPY . .                                      Layer 3
6
7  RUN npm install                               Layer 4
8  RUN npm run build                             Layer 5
9  RUN npx prisma generate                       Layer 6
10
11
12 EXPOSE 3000
13
14 CMD ["node", "dist/index.js", ]
```

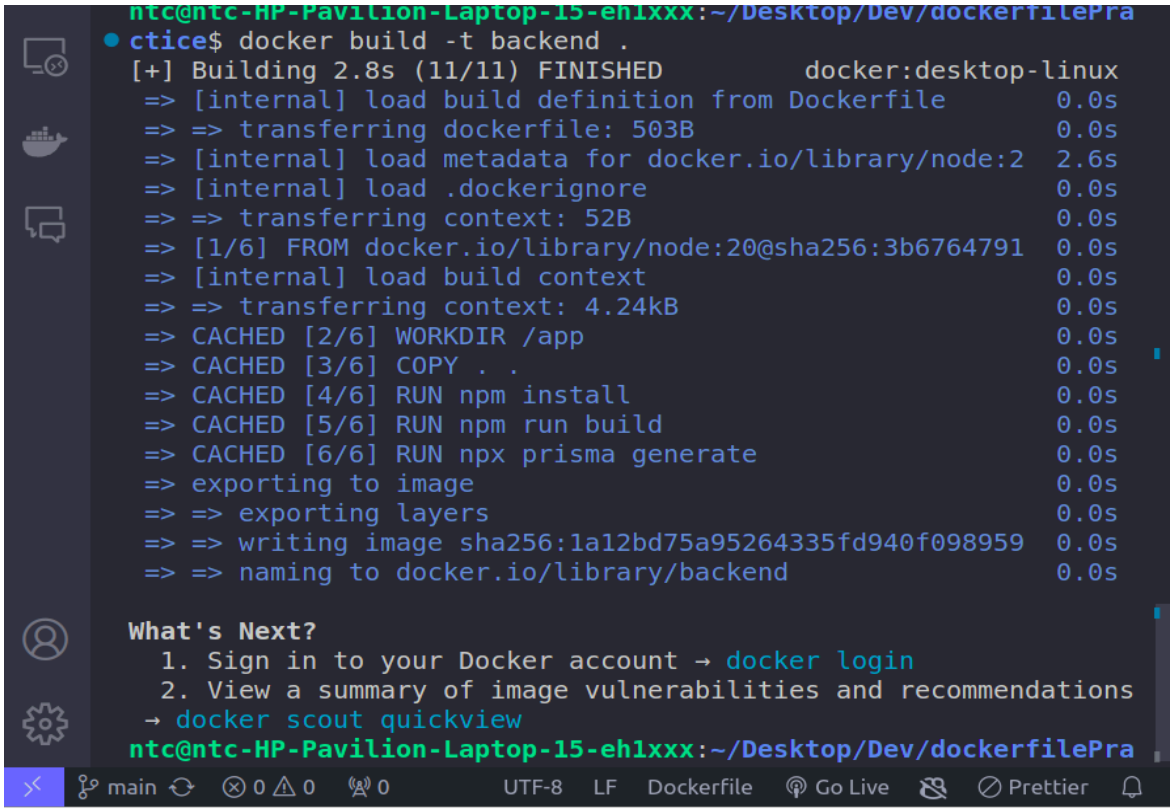
- New layer is created on the top of existing layers whenever FROM, WORKDIR, COPY, RUN etc commands are used.
- If we run build command twice some of the step will be cached , and it will be faster
- Logs

```
● → week-15-live-1 git:(main) ✗ docker build -t hkirat-backend-cool-app .
[+] Building 12.8s (12/12) FINISHED                          docker:desktop-linux
 => [internal] load .dockerignore                                            0.0s
 => => transferring context: 57B                                             0.0s
 => [internal] load build definition from Dockerfile                         0.0s
 => => transferring dockerfile: 189B                                         0.0s
 => [internal] load metadata for docker.io/library/node:20                   3.7s
 => [auth] library/node:pull token for registry-1.docker.io                  0.0s
 => CACHED [1/6] FROM docker.io/library/node:20@sha256:f3299f16246c71ab8b304d6745bb4059f  0.0s
 => [internal] load build context                                            0.0s
 => => transferring context: 7.55kB                                          0.0s
 => [2/6] WORKDIR /usr/src/app                                               0.0s
 => [3/6] COPY . .                                                           0.0s
 => [4/6] RUN npm install                                                    6.0s
 => [5/6] RUN npm run build                                                  1.3s
 => [6/6] RUN npx prisma generate                                            1.1s
 => exporting to image                                                       0.5s
 => => exporting layers                                                      0.4s
 => => writing image sha256:d0ce15534410858645f5ee075ad09dedebbfdafef8353364cf35bf8dafd8  0.0s
 => => naming to docker.io/library/hkirat-backend-cool-app                   0.0s
```

Locally setup the repository

If base image is same then we can use layer 1 over two docker file

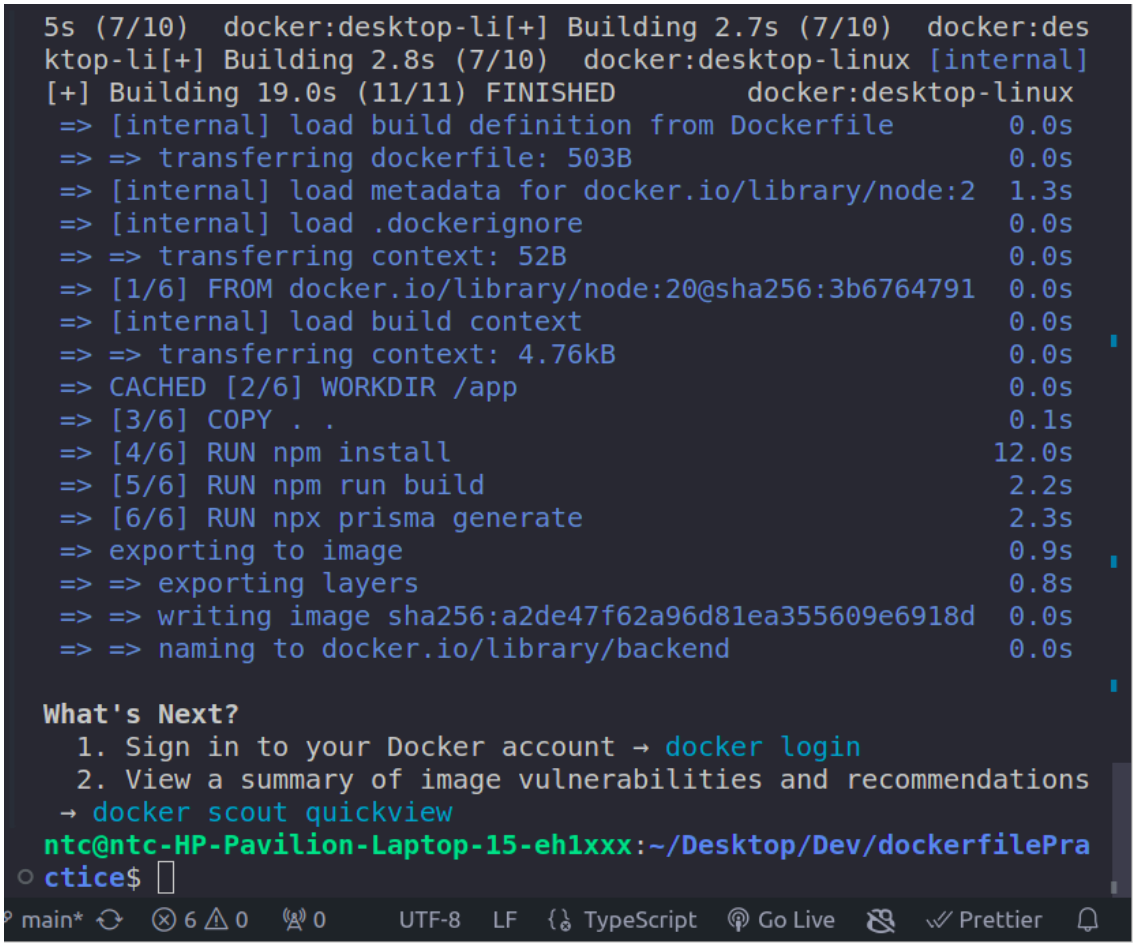
If there is one uncached layer then everything after it will also be uncached



It takes very less time second time onward

Now change something in src , index.ts and see what happens

We have changed the COPY . . layer since the file itself has changed



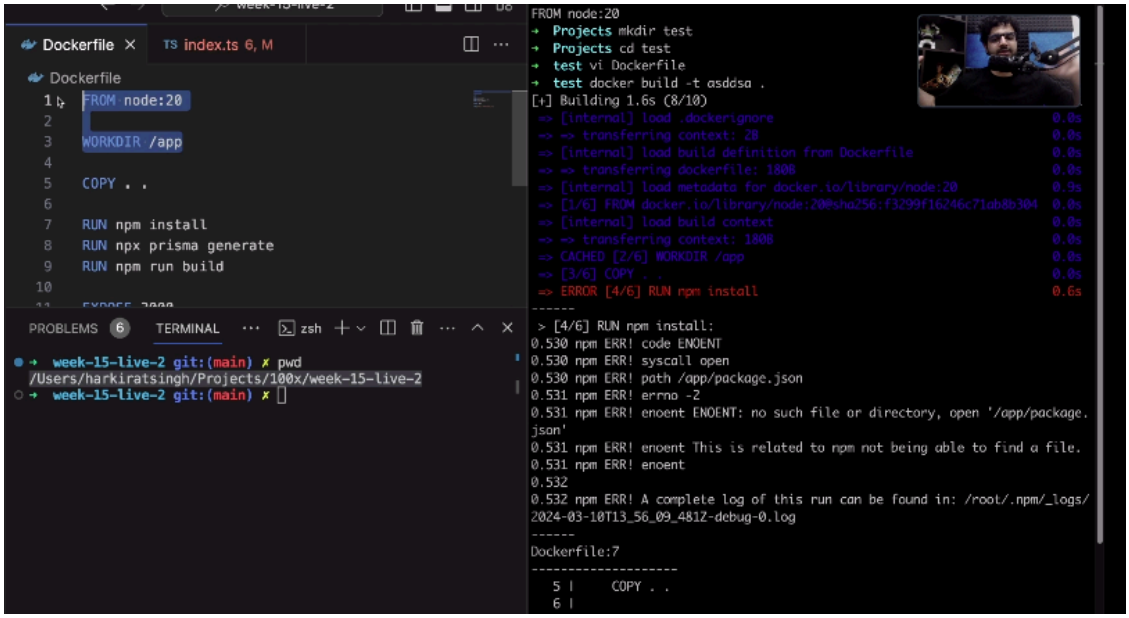
We can see that third layer onward nothing is cached

Observations:

Base image creates the first layer

Each RUN, COPY , WORKDIR command creates a new layer

Layers can get re-used across docker builds (notice CACHED in 1/6)



Here we can see that even if they are not in the same folder yet layer 1 and layer 2 is cached.

We can share layer across different images.

Why Layers??

If you change your dockerfile, layers can get re-used based on where the changes were made

If a layer changes all subsequent layers also change

Case 1: You change your source code

Logs:

```
● → week-15-live-1 git:(main) ✗ docker build -t hkirat-backend-cool-app .
[+] Building 8.7s (11/11) FINISHED                                                                      docker:desktop-linux
 => [internal] load build definition from Dockerfile                                                                    0.0s
 => => transferring dockerfile: 189B                                                                                    0.0s
 => [internal] load .dockerignore                                                                                       0.0s
 => => transferring context: 57B                                                                                        0.0s
 => [internal] load metadata for docker.io/library/node:20                                                              1.1s
 => [1/6] FROM docker.io/library/node:20@sha256:f3299f16246c71ab8b304d6745bb4059fa9283e8d025972e28436a9f9b36ed24        0.0s
 => [internal] load build context                                                                                       0.0s
 => => transferring context: 3.51kB                                                                                     0.0s
 => CACHED [2/6] WORKDIR /usr/src/app                                                                                   0.0s
 => [3/6] COPY . .                                                                                                      0.0s
 => [4/6] RUN npm install                                                                                               4.6s
 => [5/6] RUN npm run build                                                                                             1.4s
 => [6/6] RUN npx prisma generate                                                                                       1.1s
 => exporting to image                                                                                                  0.4s
 => => exporting layers                                                                                                 0.4s
 => => writing image sha256:e0a88f16c2d11e317070c51ec298e61325c0ef4a2214d2c11a2cc063d4b5e338                            0.0s
 => => naming to docker.io/library/hkirat-backend-cool-app                                                              0.0s

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/yimvu9plvmxbypwatyvzd4mdg

What's Next?
  View a summary of image vulnerabilities and recommendations → docker scout quickview
```

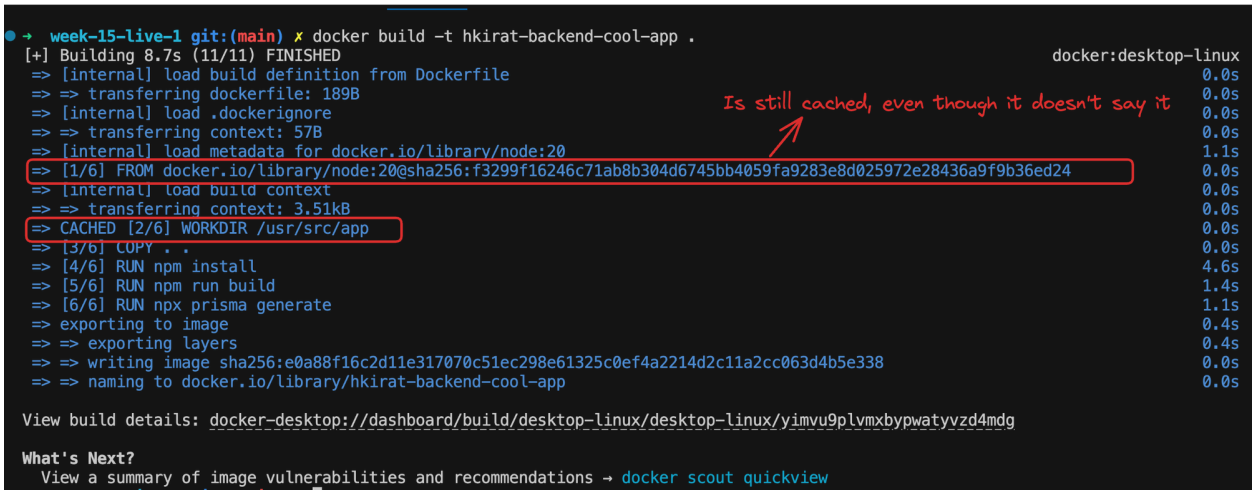
Is still cached, even though it doesn't say it

Case 2: You change the package.json file (added a dependency)

Dockerfile

```
FROM node:20                                  Layer 1 - Cached

WORKDIR /usr/src/app                          Layer 2 - Cached

COPY . .                                      Layer 3 - Changed


RUN npm install                               Layer 4 - Changed
RUN npm run build                             Layer 5 - Changed
RUN npx prisma generate                       Layer 6 - Changed


EXPOSE 3000

CMD ["node", "dist/index.js", ]
```

Logs



How often in a project does dependency change?

How often does the **npm install** layer need to change?

Wouldn't it be nice if we could **cache** the **npm install** step considering dependencies don't change often?

We are re-running npm install even if package.json is not changed

Our goal is to write in a way that across re-build increase the no. of cached layers

Our goal is to decrease the build time

Let's change the docker file to optimize it .

package* copy anyfiles package.json package.lock.json

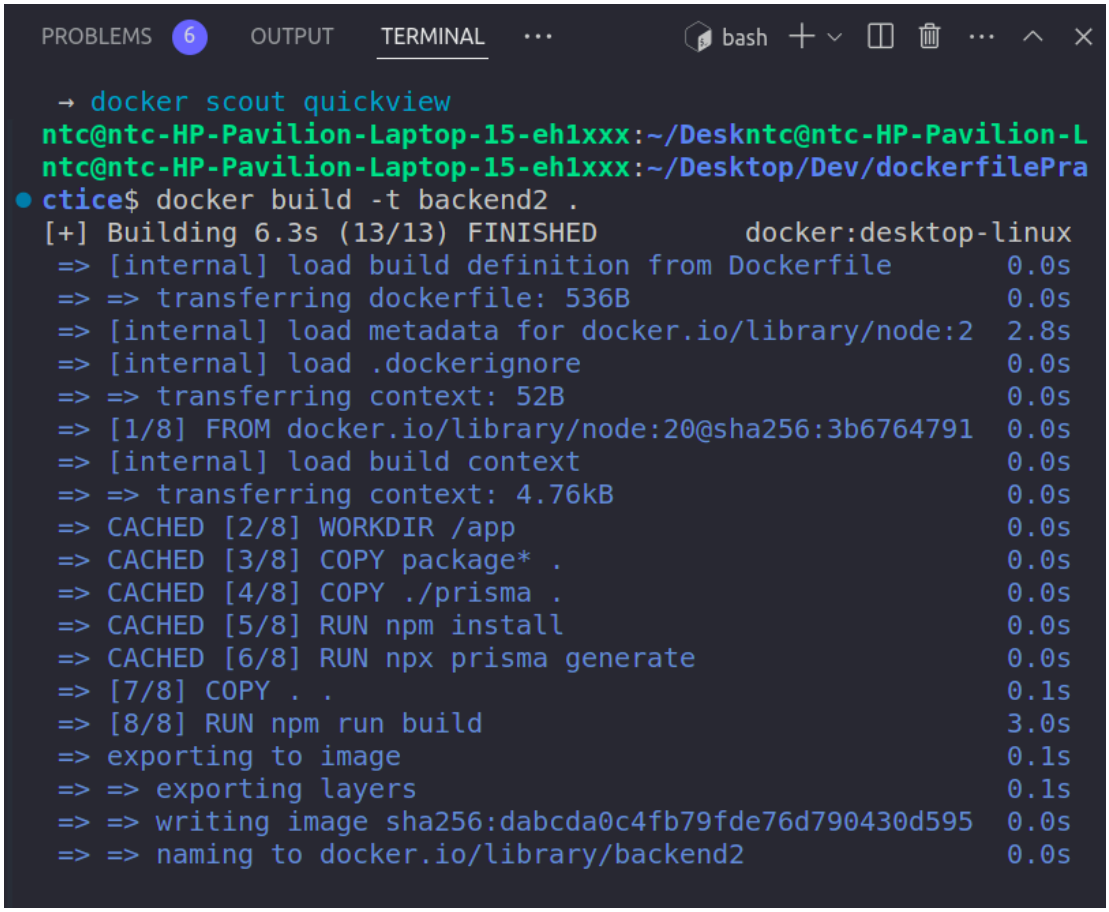
```
FROM node:20

WORKDIR /app

COPY package* .
COPY ./prisma .
# we are currently copying even node_module so add node_module to
.dockerignore

RUN npm install
```

```
RUN npx prisma generate

COPY . .
RUN npm run build
# they don't start the application
EXPOSE 3000
# Runs when we are creating a image
# Runs on thing which bootstrap our application

CMD ["node","dist/index.js"]
# why didn't we use RUN node index.js
# All the code run when we starting the image/container
# actually start our application
```

Let build it

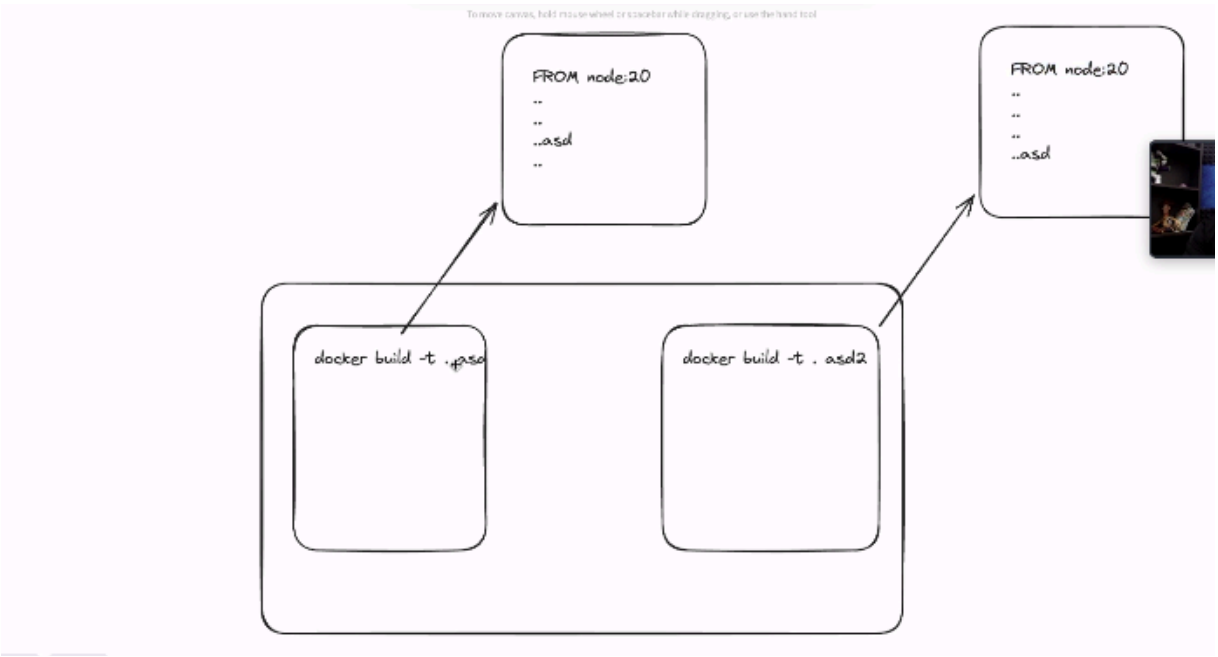
```
ntc@ntc-HP-Pavilion-Laptop-15-eh1xxx:~/Deskntc@ntc-HP-Pavilion-L
ntc@ntc-HP-Pavilion-Laptop-15-eh1xxx:~/Desktop/Dev/dockerfilePra
ctice$ docker build -t backend2 .
[+] Building 22.7s (13/13) FINISHED       docker:desktop-linux
 => [internal] load build definition from Dockerfile       0.0s
 => => transferring dockerfile: 536B                       0.0s
 => [internal] load metadata for docker.io/library/node:2  3.0s
 => [internal] load .dockerignore                          0.0s
 => => transferring context: 52B                           0.0s
 => [1/8] FROM docker.io/library/node:20@sha256:3b6764791  0.0s
 => [internal] load build context                          0.0s
 => => transferring context: 4.75kB                        0.0s
 => CACHED [2/8] WORKDIR /app                              0.0s
 => [3/8] COPY package* .                                  0.1s
 => [4/8] COPY ./prisma .                                  0.1s
 => [5/8] RUN npm install                                 13.5s
 => [6/8] RUN npx prisma generate                          2.4s
 => [7/8] COPY . .                                         0.1s
 => [8/8] RUN npm run build                                2.4s
 => exporting to image                                     0.9s
 => => exporting layers                                    0.9s
 => => writing image sha256:0e4d84e9cca251a26d7b1a73299d4  0.0s
 => => naming to docker.io/library/backend2                0.0s

What's Next?
  1. Sign in to your Docker account → docker login
  2. View a summary of image vulnerabilities and recommendations
main*  ⊗ 6 ⚠ 0   0   UTF-8  LF  Dockerfile  Go Live  Prettier
```

Let change something in src file and see how many layers are getting cached.



Npm install is expensive takes lot of time hence we optimized it. npm install is cached until we add a dependencies.

Network and Volumes

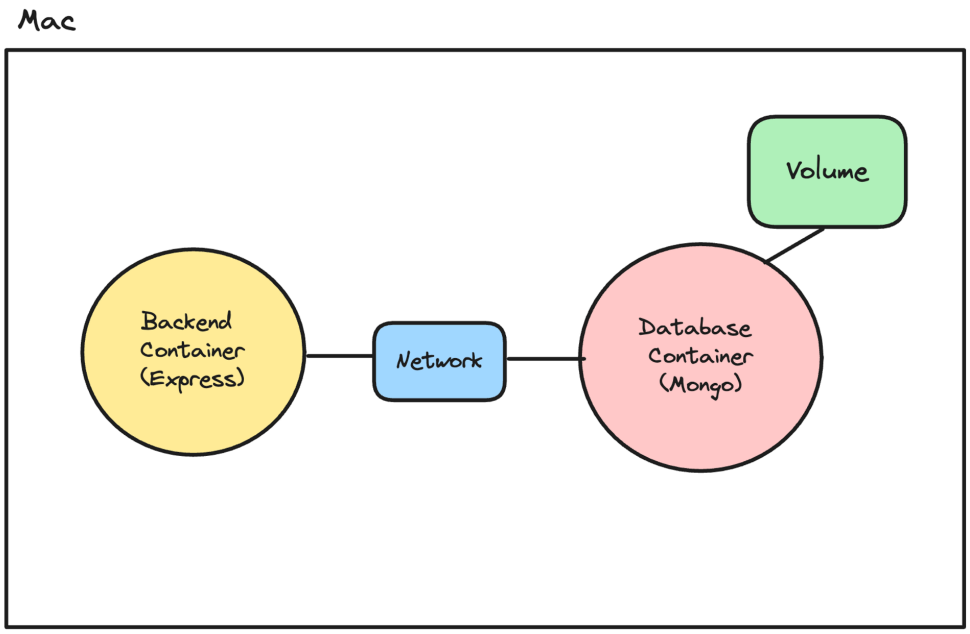


Network and Volumes

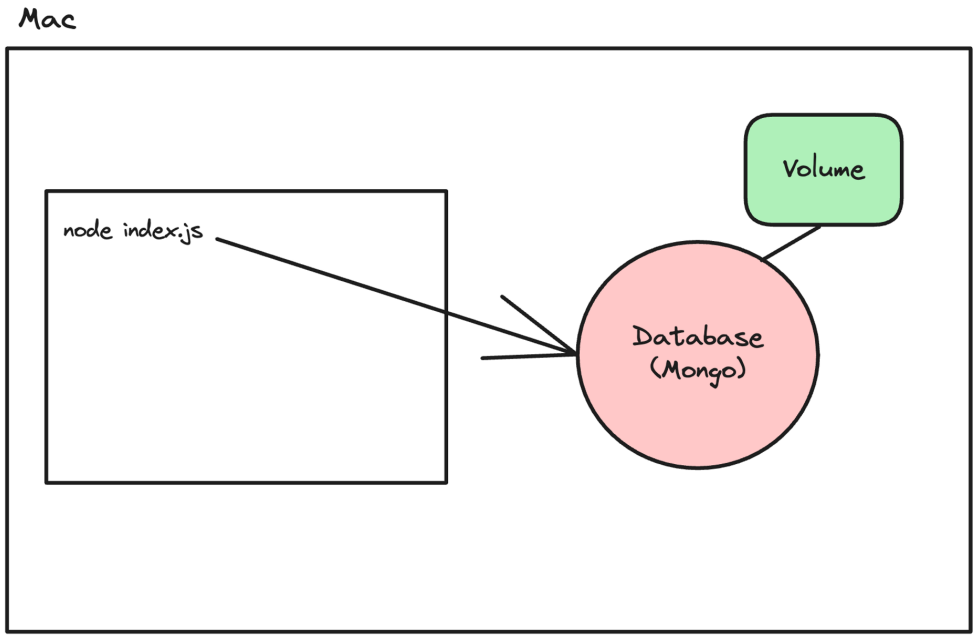
Networks and volumes are concepts that become important when you have multiple containers running in which you

1. Need to persist data across docker restarts (If we are starting mongodb database locally then , it will start mongoDB for us and eventually it will start for us)
2. Need to allow containers to talk to each other

If we have a container where we run backend and in other container we have DB running then we can’t run them locally.



Till now we didnt need the networks because when we started the mongo container it was getting accessed by Node.js process running directly on the machine



Volumes

If you restart a mongo docker container, you will notice that your data goes away.

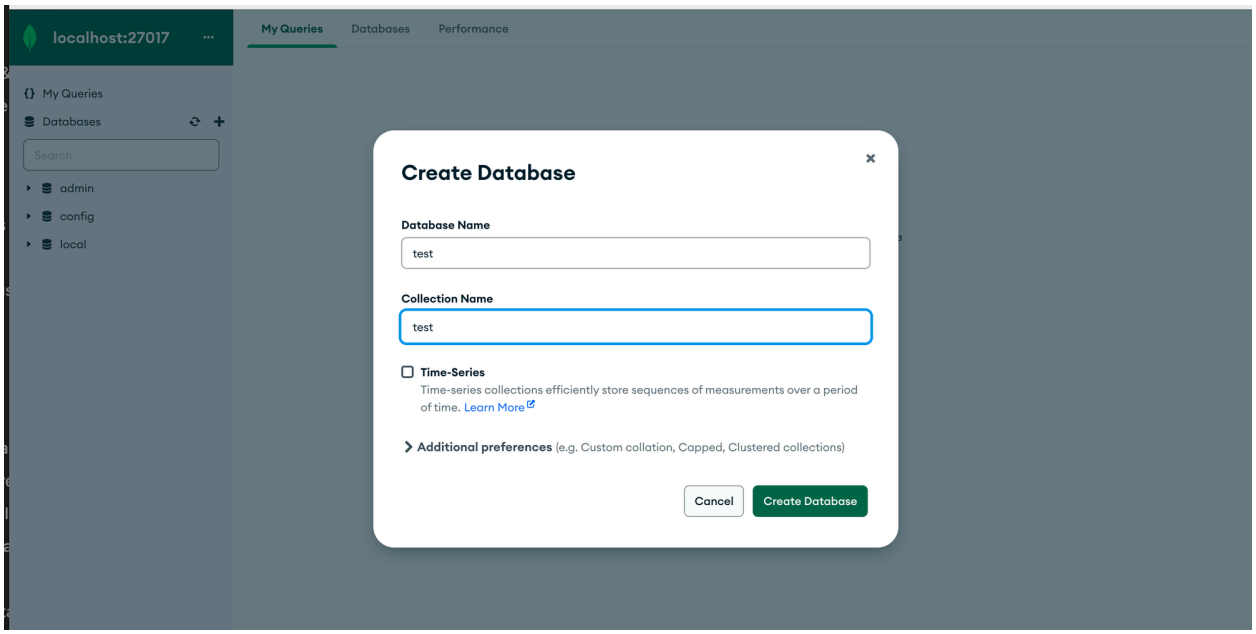
This is because docker containers are transitory (they don’t retain data across restarts)

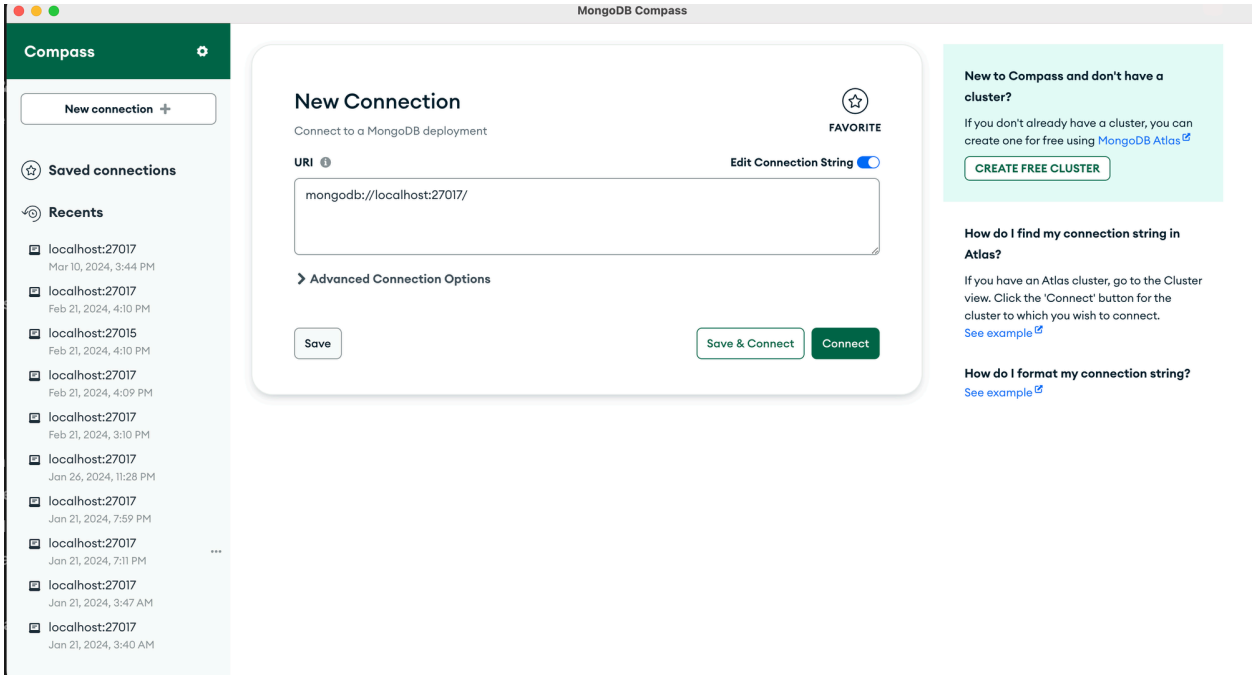
Without volumes

1. Start mongo container locally

docker run -p 27017:27017 -d mongo

2. Open it in MongoDB Compass and add some data to it.





3. Kill the container

docker kill <container_id>

4. Restart the container

docker run -p 27017:27017 -d mongo

We find that database is compass won't persist

With Volumes

1. Create a **volume**

docker volume create volume_database

docker volume ls —- to know the volume created

2. Mount the folder in **mongo** which actually stores the data to this volume

docker run -v volume_database:/data/db -p 27017:27017 mongo

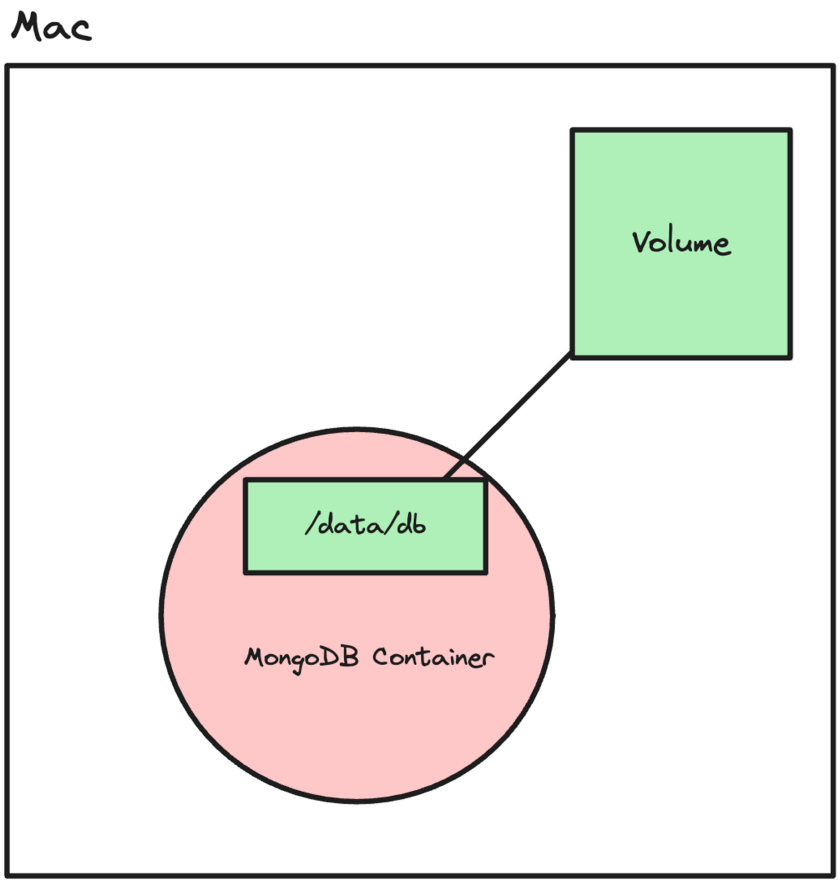
For postgres the place where the data is stored may be different, and maybe there is more than one volume needed.

3. Open it in MongoDB compass and add some data to it
4. Kill the container

docker kill <container_id>

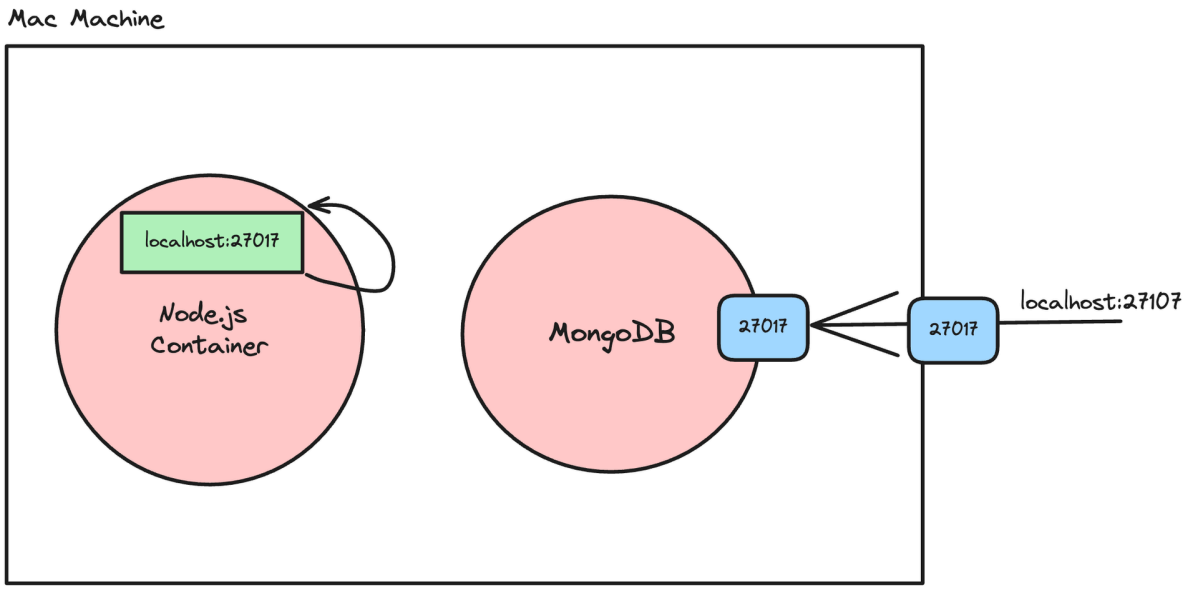
5. Restart the container

docker run -v volume_database:/data/db -p 27017:27017 mongo



Network

In docker a network is a powerful feature that allows containers to communicate with each other and with the outside world.
Docker containers can’t talk to each other by default
Localhost on a docker container means it’s own network and not the network of the host machine



How to make container talk to each other?
Attach them to the same network

1. Clone the repo :-- https://github.com/100xdevs-cohort-2/week-15-live-2.2

2. Build the image
docker build -t image_tag .

3. Create a network
docker network create my_custom_network

4. Start the backend process with the network attached to it

docker run -d -p 3000:3000 --name backend --network my_custom_network image_tag

Start mongoDB first before the backedn

5. Start mongo on the same network

   docker run -d -v volume_database:/data/db --name mongo --network my_custom_network -p 27017:27017 mongo

Name is very important Now we have connected our network with the MongoDB container and we will change the url of the database , change local host with the name

```
const mongoUrl: string = 'mongodb://thapa_database:27017/myDatabase';
```

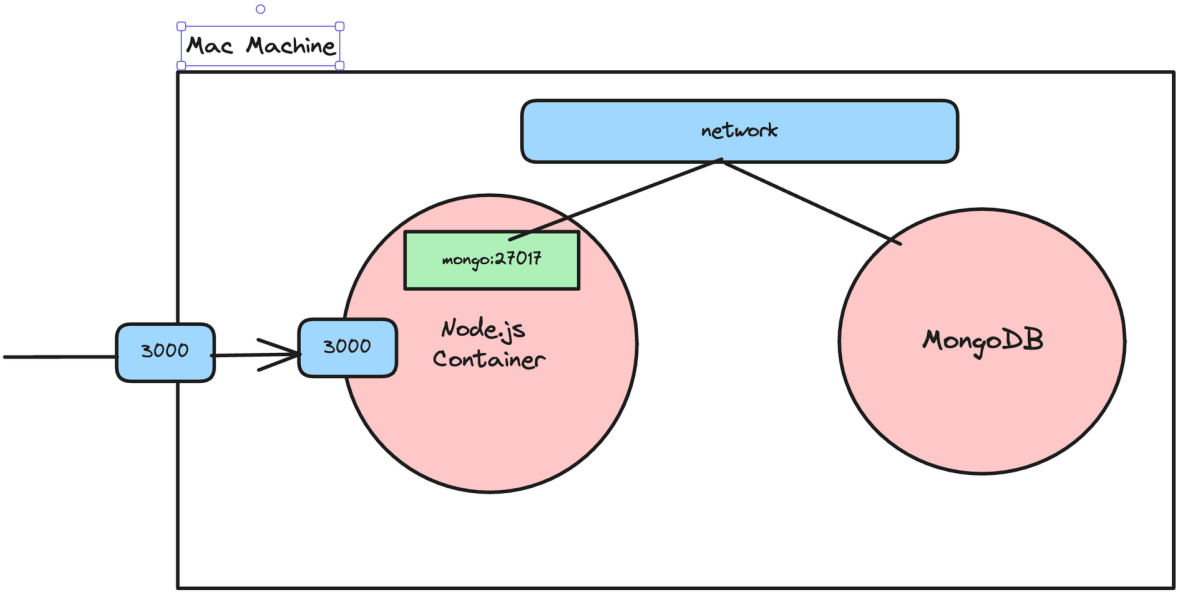
This name is similar to the domain name but in the context of the container **Rebuild** the image and then run the above command, but make sure to kill the nodejs container

6. Check the logs to ensure the db connection is successful

   docker logs <container_id>

We wont be able to see the logs if we run in detached mode then we should run the above command to see the logs

Try to visit an endpoint and ensure you are able to talk to the database
If you want, you can remove the port mapping for mongo since you don't necessarily need it exposed on your machine

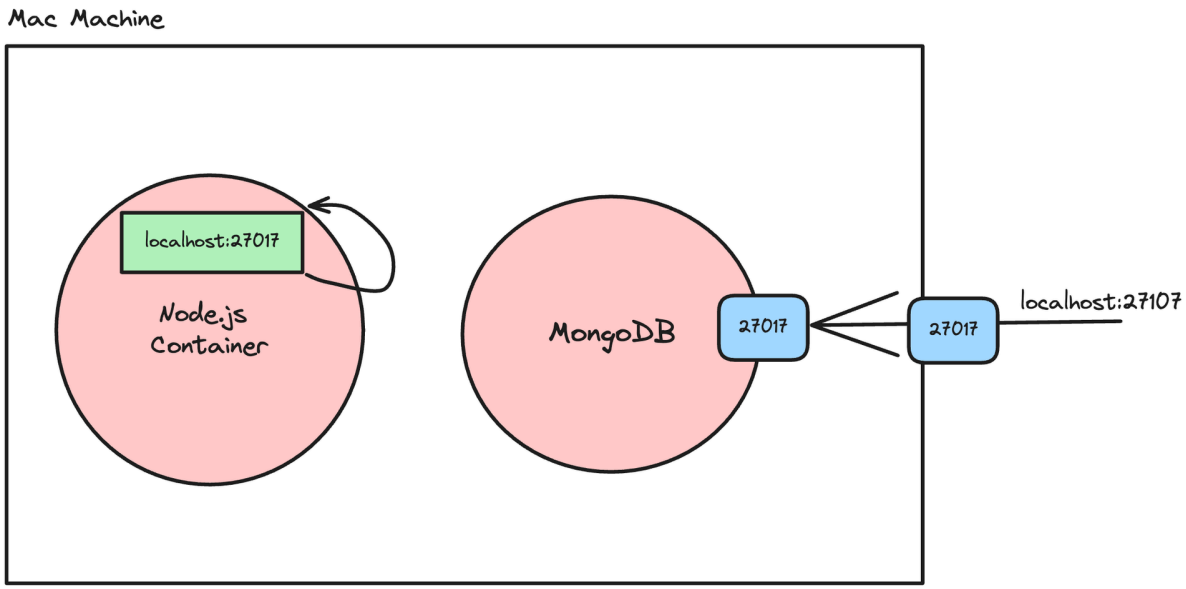


What should be the url

```
import mongoose, { Schema, model } from 'mongoose';

const mongoUrl: string = 'mongodb://localhost:27017/myDatabase';
```

docker run -p 27017:27017 mongo

Will start the pink mongoDB container at port 27017



```
ntc@ntc-HP-Pavilion-Laptop-15-eh1xxx:~/Desktop/Dev/week-15-live-
2.2$ docker run mongo_app
Server running at http://localhost:3000
MongoDB connection error: MongooseServerSelectionError: connect
ECONNREFUSED ::1:27017, connect ECONNREFUSED 127.0.0.1:27017
    at _handleConnectionErrors (/app/node_modules/mongoose/lib/c
onnection.js:875:11)
    at NativeConnection.openUri (/app/node_modules/mongoose/lib/
connection.js:826:11) {
  reason: TopologyDescription {
    type: 'Unknown',
    servers: Map(1) { 'localhost:27017' => [ServerDescription] }
,
    stale: false,
    compatible: true,
    heartbeatFrequencyMS: 10000,
    localThresholdMS: 15,
    setName: null,
    maxElectionId: null,
    maxSetVersion: null,
    commonWireVersion: 0,
    logicalSessionTimeoutMinutes: null
  },
  code: undefined
}
```

Both the container are unable to talk to each other.

Types of network

- Bridge: The default network driver for containers. When you run a container without specifying a network, it's attached to a bridge network. It provides a private internal network on the host machine, and containers on the same bridge network can communicate with each other.

- Host: Removes network isolation between the container and the Docker host, and uses the host's networking directly. This is useful for services that need to handle lots of traffic or need to expose many ports.

Bind mounts

Till now we have bound container to volume.

But we haven't bound it to the host machine , here we will find a folder in container to the bind a folder in machine and any change will reflect on both

How can we start application by docker and have hot reloading(very useful in local development) in it

npx create-next-app@latest

We see the basic NextJs boiler plate code

Dockerfile

```
FROM node:21-alpine
WORKDIR /nextapp
COPY packages* .
RUN npm install
COPY . .
CMD ["npm","run","dev"]
```

docker build -t nextapp .

docker run -p 3000:3000 nextapp

To run the nextjs application

It wont hot reload anymore ,

But we can see the hot relaod only if we are inside the docker container

For seeing hot reload from changing code on vs code only we will use **bind mount**

We can mount multiple folder also let see for src

docker run -p -v ./app:/nextapp/app 3000:3000 nextapp