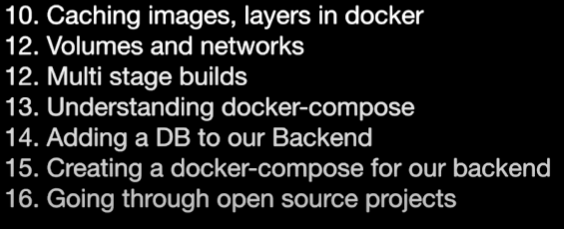
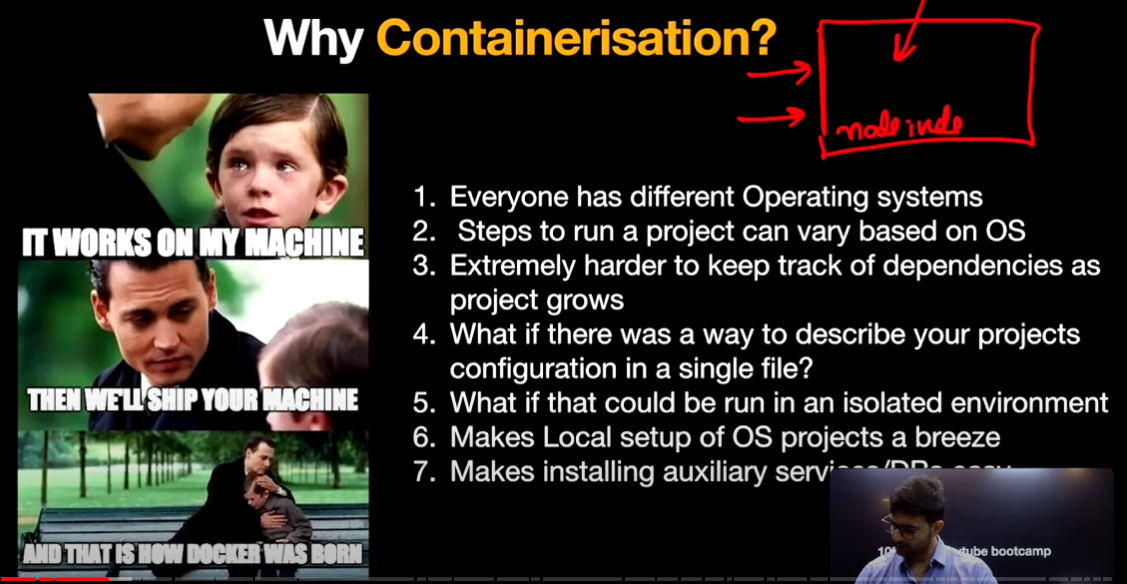
**Open Source Bootcamp - Complete Docker and Devops Roadmap - Part 1**

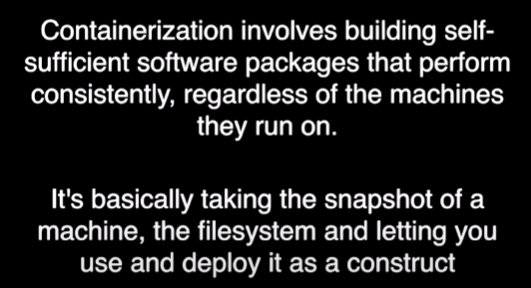
In every open source codebases, on top of the source code there will be two files.

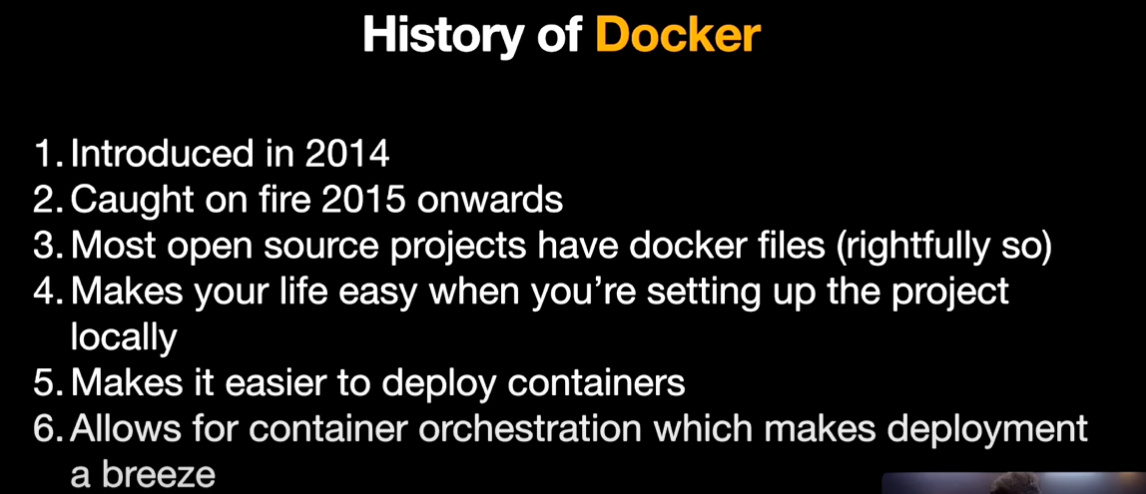
1. Docker file
2. Docker – compose











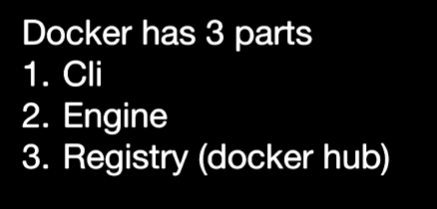
In steps:

1. Docker

2. Infrastructure as code

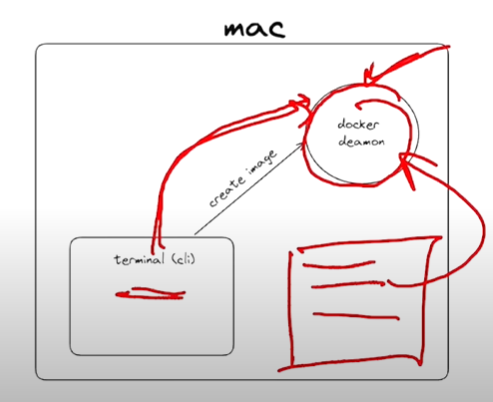
3. Kubernetes

Container Orchestration means maintaining the containers in different different places.



CLI = Command line interface used for exceuting docker commands in command prompt.

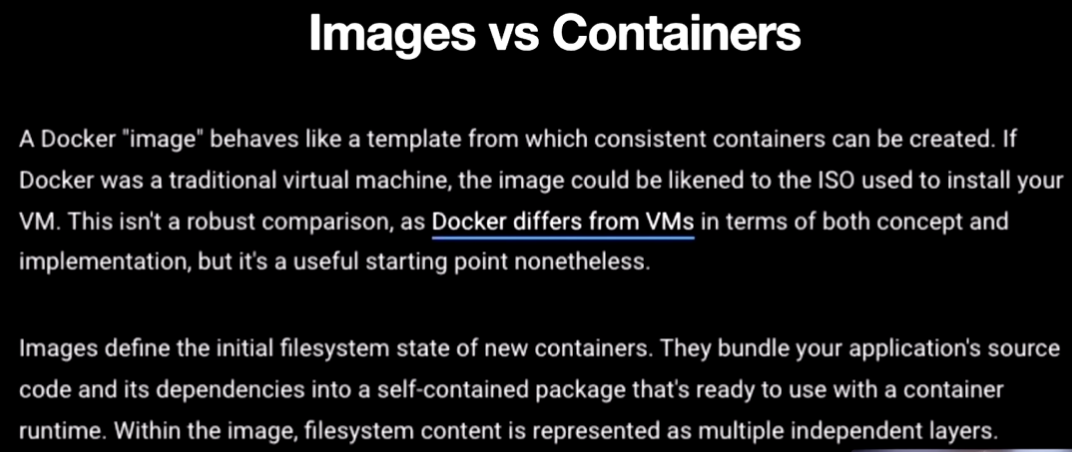
Engine = The commands written in CLI will be communicating with Docker Daemon/ Docker Engine.



Registry or Docker Hub = It is similar to Github

In Github, we will be deploying the code in centralized space.

In Docker Hub, we will be deploying the Image.



Consistent means irrespective of Operating system image will be created.

[Difference between Docker Image and Container - GeeksforGeeks](https://www.geeksforgeeks.org/difference-between-docker-image-and-container/)

Docker Image: It contains all the dependencies of our application.

Ex: In our local machine we have index.js file. So here we have to give details like, all the dependencies required to run this index.js file from any OS inside docker image file.

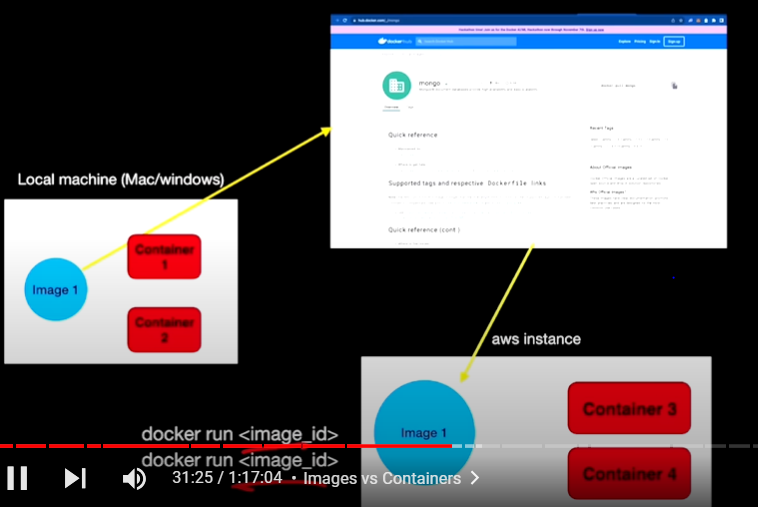
Dependencies like, Node JS version, File system, Expose to which port etc.,.

And then we can send the image to any OS.



Docker run <image\_id> is used to run an image. In simpler words an image when running is called a container.

Also we can push the image to registry/ docker hub.



Step 1: Pushing the image to Docker Hub

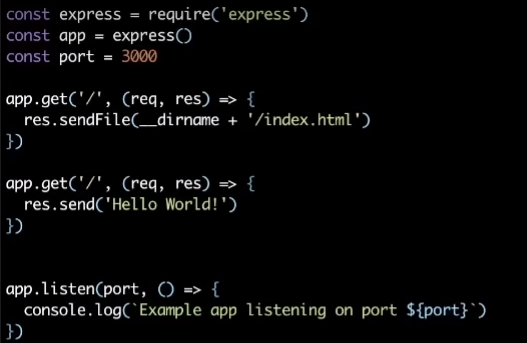
Step 2: anyone can pull the image from hub and run on their locally.

**Let’s Code**:

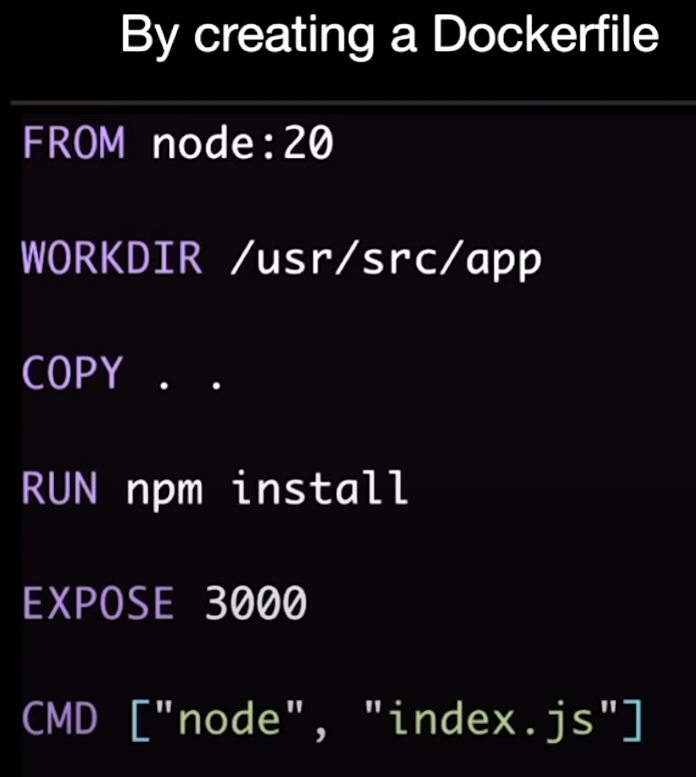
Here we are going to create a Server like HTTP Server.

Understanding: For example, if we open chrome and type google.com in URL. This request along with URL will go to google servers and returns a HTML page in response to us. This is what we see in chrome browser.

SO here we are creating our own HTTP Server called **express server**



Now instead of running the index.js locally after node js installed. Lets containerize the things by creating a docker file.



Any docker file starts from “FROM” <Base image>

Since our application is node JS base app. Lets use Node JS base image which is node:20

Or if we build everything from scratch use FROM SCRATCH and list all dependencies in the next line.

COPY command will copy all the code into working directory.

Run npm install = will install all external dependencies mentioned in JSON file in our case it is “express’’ Server.

CMD [‘’…”,”…..”] = This step will execute commands when image is started or container is running but not while creating the docker image.

To build an image, run this command

Docker build <current dir> -t <Give name to a image>

After docker image got created.

Converting image to a container means running an image.

Docker run <image\_name>

And then

Docker run -p 3000:3000 test-app

Means Docker run -p 3000 port of windows machine should forward/request to the 3000 port of the container this command is having.

Now run localhost:3000 will work on browser

If Docker run -p 3003:3000 test-app

Now run localhost:3003 will work on browser

Pushing my Image to docker HUB:

Docker build . -t 9553544562/first\_test\_app\_7

Docker push 9553544562/first\_test\_app\_7

**Part -2**

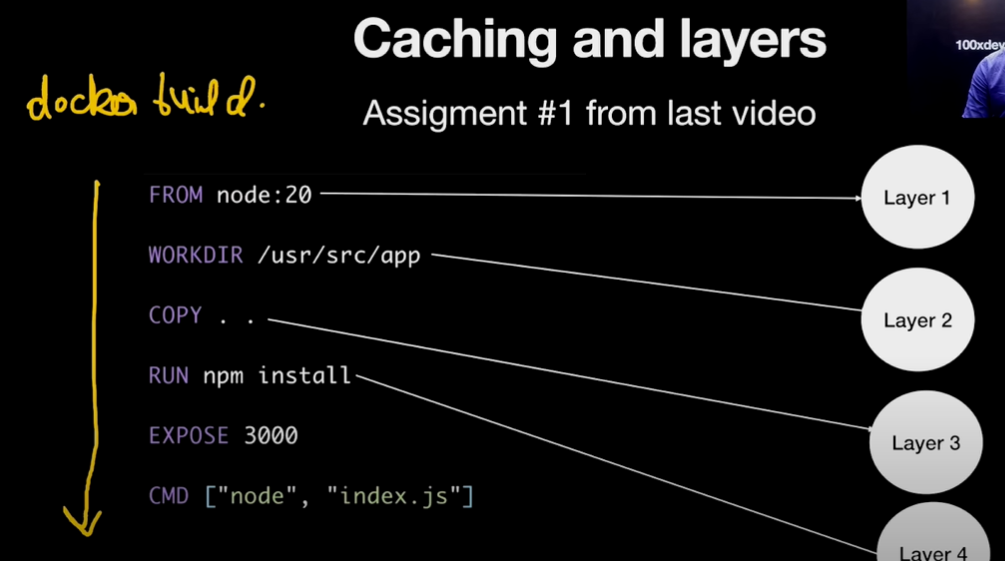
**Assignment -1**

***Caching and Layers:***

Whenever we hit the command,

Docker build

Docker starts to create these layers one by one from top to bottom.



In our case,

Layer 1: Docker will bring the Node JS do from the internet. Probably takes 2 to 3 mins and of size 900 MB

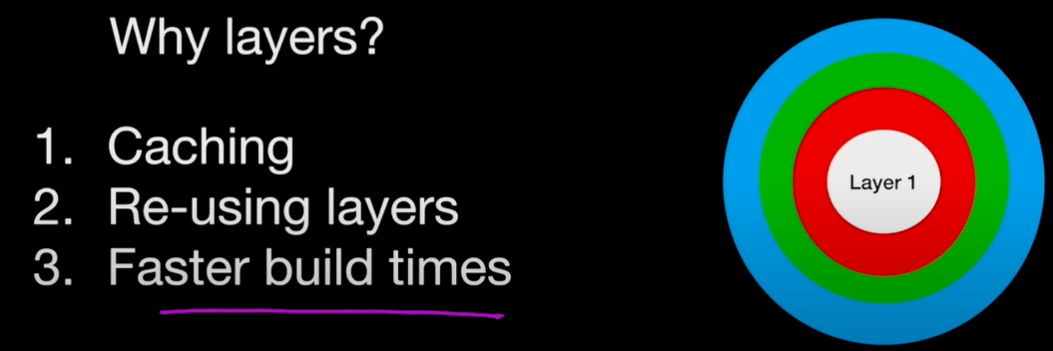
Layer 2: Setting up the working directory. Probably takes 1 to 2seconds and of 0 size.

Layer 3: Copying all the codebase to the current image we are building. Probably takes 3 to 4sec and it increases image size to some 100 KB.

Layer 4: In this step, docker will bring the external dependencies from the npm registry into our image. Probably takes 30 secs and increases the image size to 90 MB.

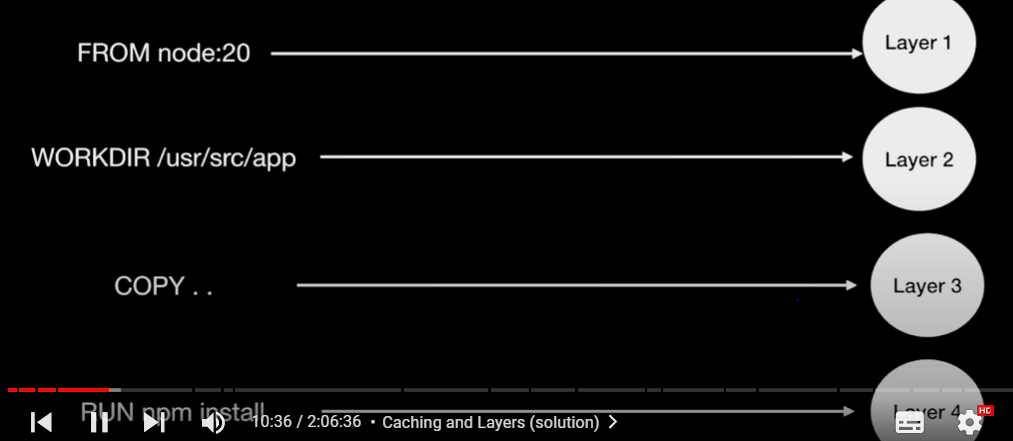
Why docker introduced the Layers and why not in one single step ?

Ans: Layers helps in caching when containers rans second time onwards.



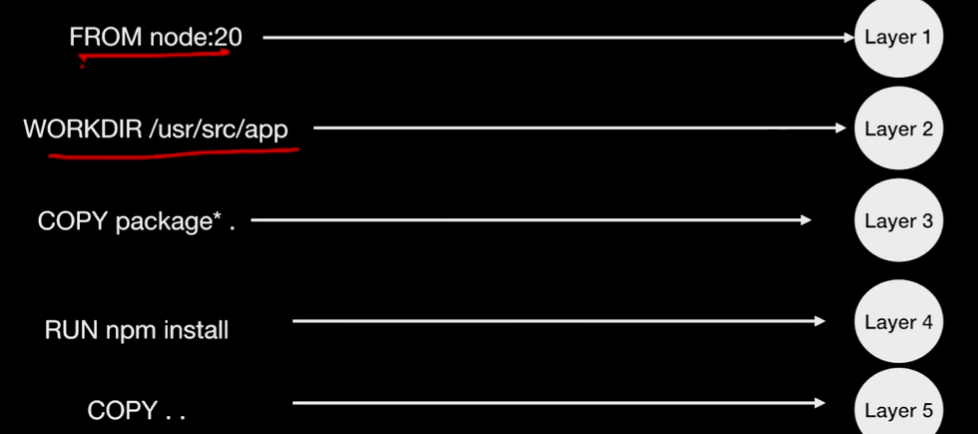
What if we change the layers in docker file ?

Before:



When we change the Index.js file, It means the Layer 3 will get changed. And if we the same docker file again layer 1,2 was already cached so these runs from cache and takes 1 sec each. However, from Layer 3 the steps will run as normal process I mean not from cache. Even though the package.json which contains the external dependencies doesn’t change because of layer 3 (index.js changed now) the layer 4 also won’t run from cache now it gets dependencies from npm registry which in turn unnecessary time waste.

Solution for above problem:



Layer 3: Copying all Packages from code base to IMAGE. So package.json will be copied to IMAGE.

Layer 4: Getting external dependencies from npm registry.

Layer 5: Then copying rest of the files from codebase to IMAGE. Now the index.js is getting copied into image at the end of the docker file step.

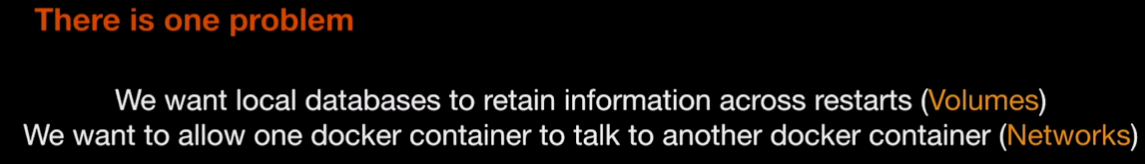
So, if we ran docker file again the “RUN npm install” command will get info from cache which was already cached in first run.

**Volumes and Networks**

Why we need Volumes and Networks ?

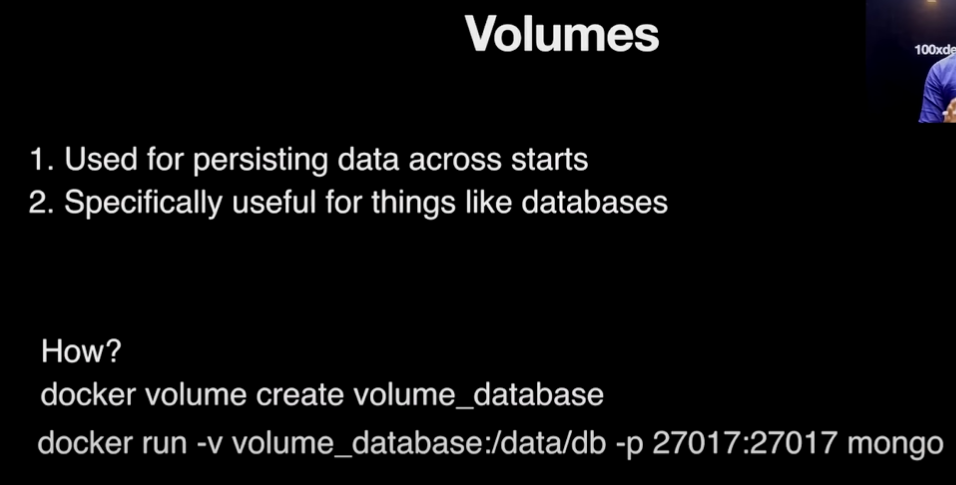
Ans: Because there were two problems.

1. Data is not persistent across container restarts.
2. Container to container communication is not happening.

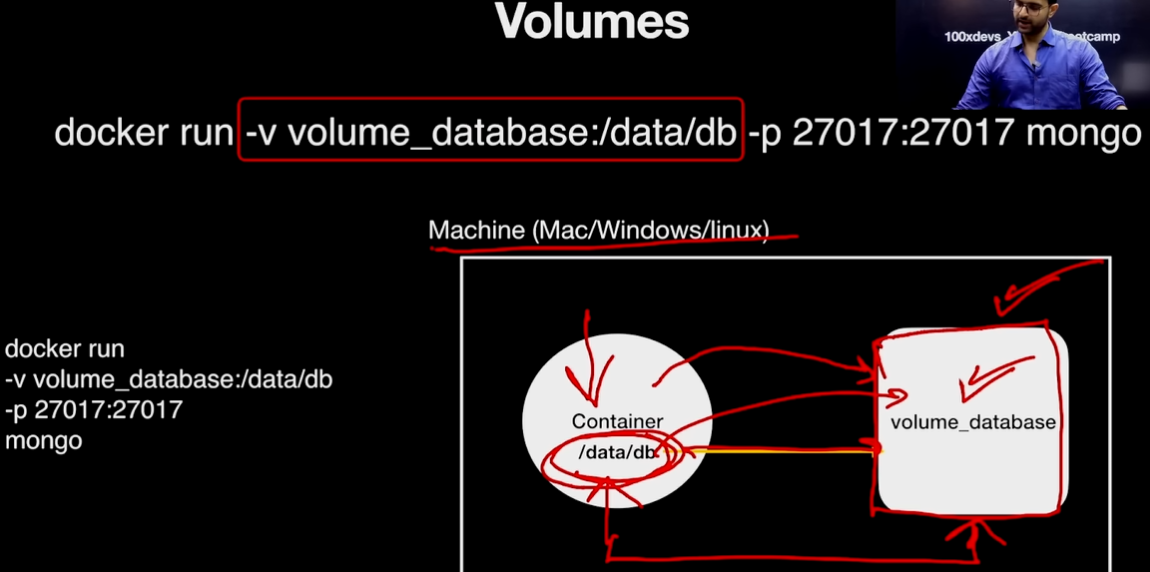
 

Databases need to be persisitent even though docker container shuts down. Answer: ***Volumes***.

If Containers to communicate – we need ***networks***.



If we create Volume before starting our container, and we will dump the data into Volume after container starts. Now even if we bring down the Container data will be saved in Volume until or unless volume also brought down.



**Creating Volume**:

“docker volume create <volume\_database\_name>”

This will create volume inside the Docker Engine.

**Starting a container**:



It will start the mongo container and stores the data in /data/db inside container and this container will get mounted on volume\_database which got already created by step 1 command.

Whenever we use mongo db it will store the data inside /data/db folder. It may be different for different databases.

**Killing the container**: it will kills the container but Volume will be alive.

“docker kill <Container\_id>”

**Remove the Volume**:

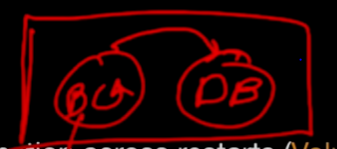
“docker volume rm <Volume\_database\_name>”

**Networks**

Every container has its own network. It means every container has its localhost address.

In our machine if we hit localhost:127.0.0.1 ( It is my machine’s local address.)

In my machine I have a container. If we go inside the container and hits localhost:127.0.0.1 ( It is my Container’s local address.)



The above image indicates, there are two containers running inside the same machine.

Left side container 🡪 Running backend app. ( It has its own local address)

Right side container 🡪 Where database is running. (It has its own local address)

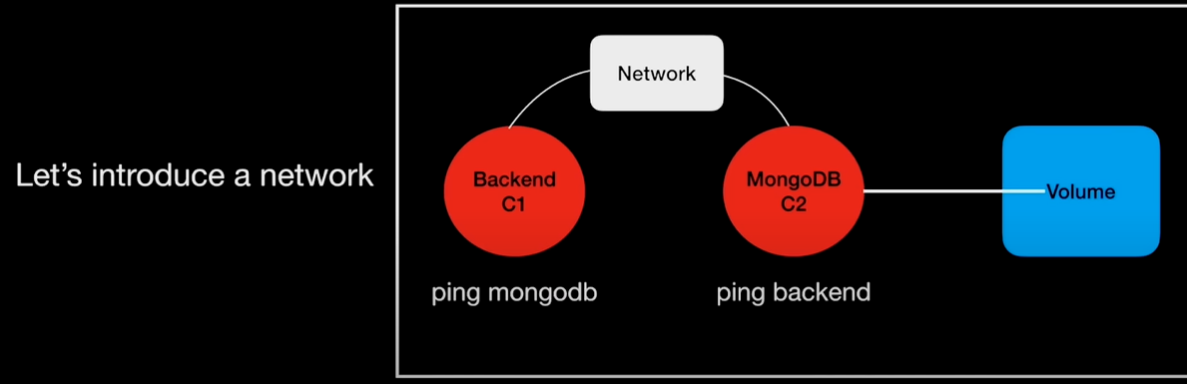
Problem is how to create a network between these two containers?



Container 1: Runs Backend app.

Container 2: Runs MongoDB which is mounted to Volume.

Let’s update the backend to put some data in a Mongo Database



Like Volume creation, we have to create a network and tell container about this network when starts.



In above screenshot,

Step 1: Creating a network.

In the index.js file which resides in Backend container replace the localhost with mongo container name which is “mongodb2”.





Step 2: Starting Backend container with two extra statements.

1) network

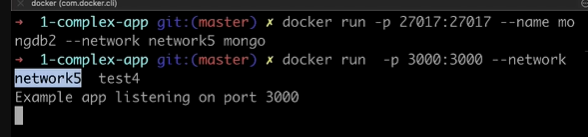
2) backend container name

Backend container wants to connect with mongo container. So we have to keep both the containers in same network.

Create a network:

“docker network create network5”

Now first run mongo container in network5 by including container name also. Next run backend container in same network5.

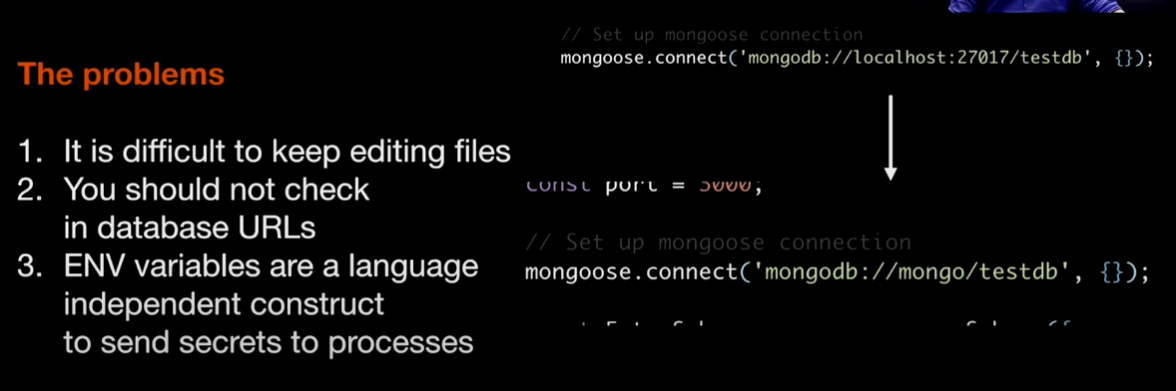


Command 1: Starting mongo container in network5

Command 2: Starting backend container in same network5. ( No need to mention name of the backend here because mongo container don’t want backend container name however backend wants mongo container name since backend container wants to connect with mongo container [ one way connection ])

Backend 🡪 Mongo (This is the connection established now)

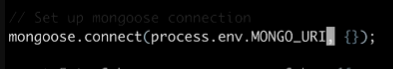
**ENV Variables**



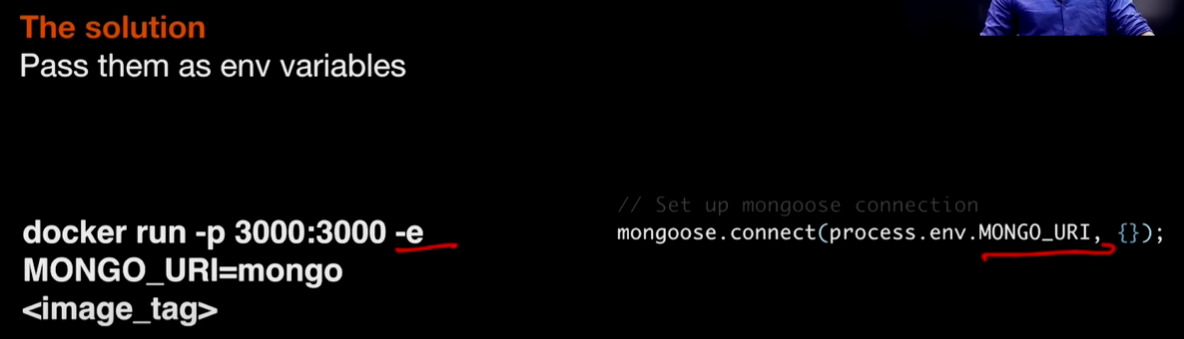
Why we need ENV Variables ?

Ans: what if we have a different database in prod or test environment. It is difficult to keep on editing these files. (here in index.js file) Like we want to run this application in dev or test or prd environment we have to edit this files.

So we need ENV Variables to prevent hardcoding the URLs and secrets in code files.

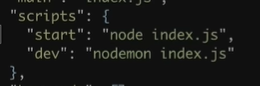


Ex: Process.env.MONGO\_URI is an env variable.



**Multi – Stage Builds**

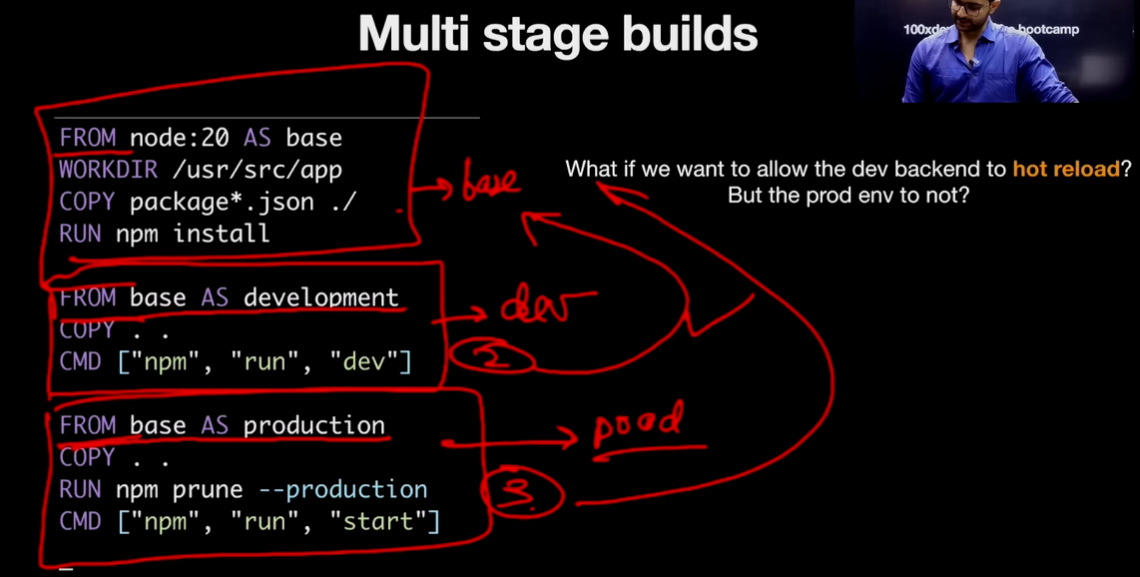
Inside the package.json file we have scripts section



If we use “npm run start” or “node index.js“ both are same, runs the index.js file server will start.

If we use “npm run dev” or “nodemon index.js“ both are same, runs the index.js file server will start. However whenever a change happens in index.js server will get restarted even though the server is in running state.

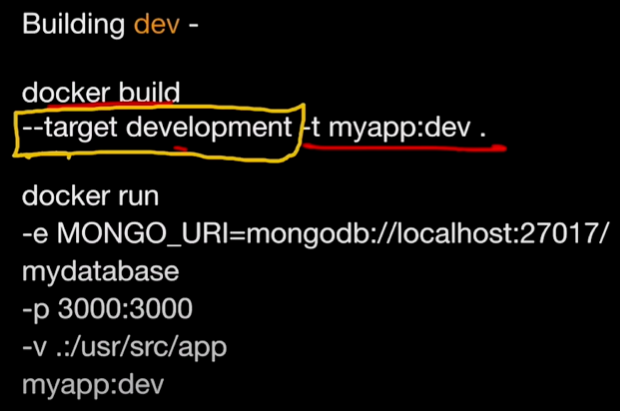
We will use the dev command at development time in local machines. It will helps us in reloading the servers after each every change in files.

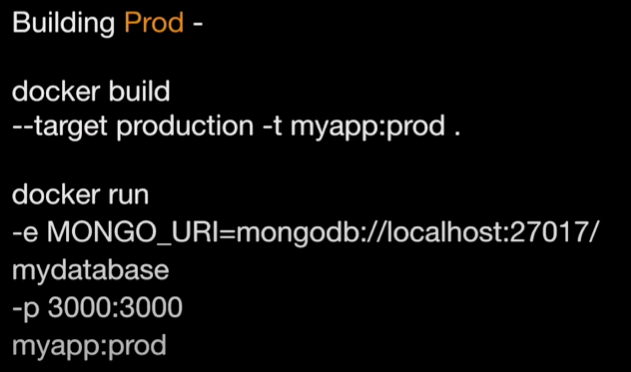


For the first 4 lines docker will create a base image.

For next 3 lines docker will create a second image from base image.

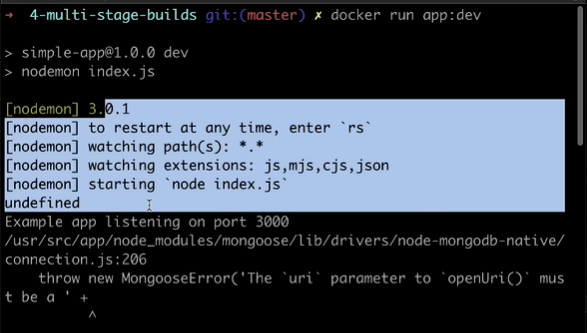
For next 4 lines docker will create a third image from base image.





**Problem Statement 1**:

If I ran container with nodemon process. Server got started. (Command Prompt 1)

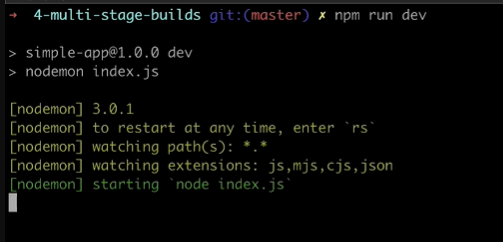


As per the nodemon process whenever a change in index.js happened the server should get restarted right.

But if I edit the index.js file in the same 4-multi-stage-builds (Command prompt 2) the server didn’t get restarted/reloaded (Command prompt 1). **Why** ?

Now kill the docker container. Docker kill <image\_ID>

If we run “npm run dev the server gets started.



Now in command prompt 2, edit the index.js file you will see the reload/restart of the server in Command prompt 1.

Why its reloaded with npm run dev and why not with docker run app:dev command ?

Problem is, I have a index.js file in my mac machine and if I ran the container the copy of index.js will be moved to container. And if make any change in my local mac machine index.js the nodemon won’t restart the server. So how to make a server to be restarted when we ran index.js in local mac machine.

[**hint**: Volumes]



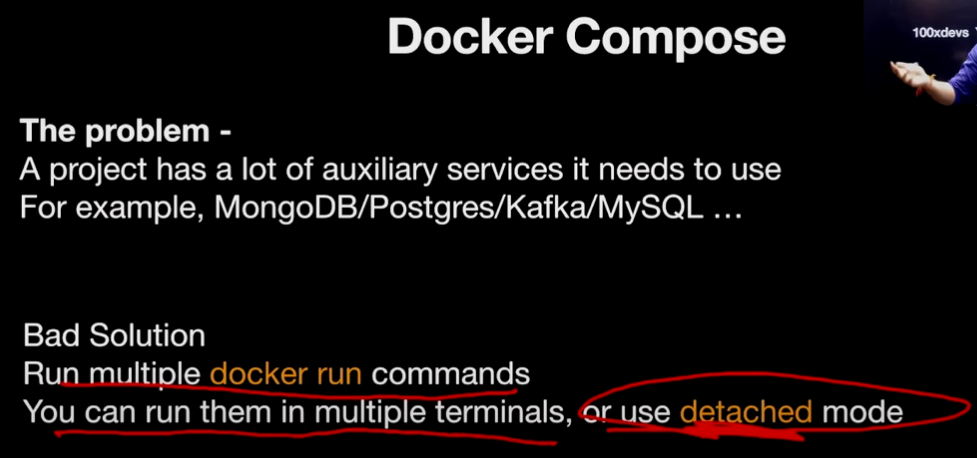
Solution:

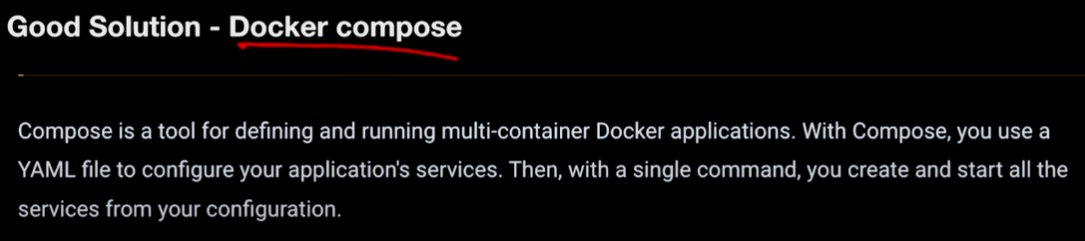


-v .:/usr/src/app It means creating a mount from local mac machine to container.

If we delete index.js inside the container the file will get deleted in container and as well as in local mac machine also.

**Docker Compose**





All the docker commands composed in a single file. Here we mentioned all our desired resources.

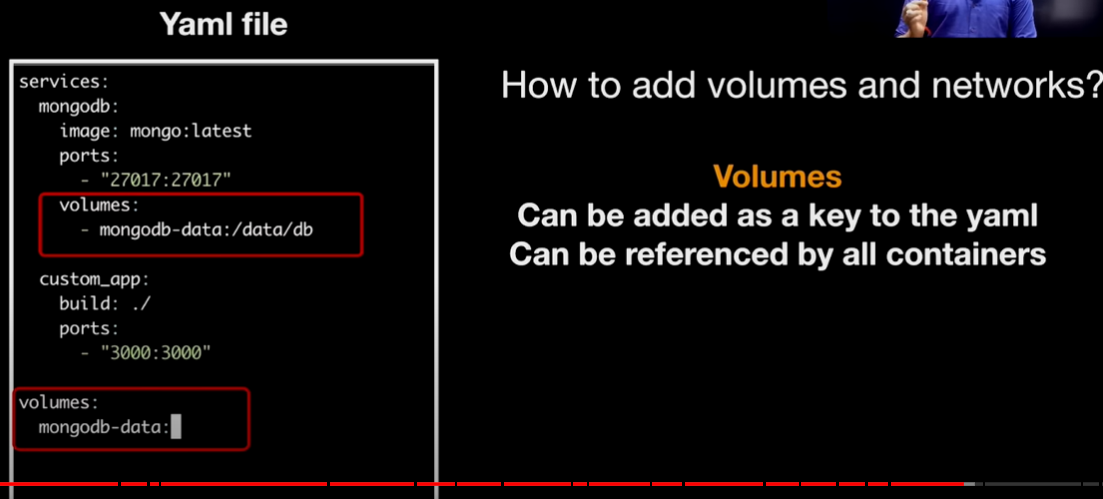




How to add Volumes and networks ?

Already containers are connected via a network by docker compose.

Then how to add volumes.?

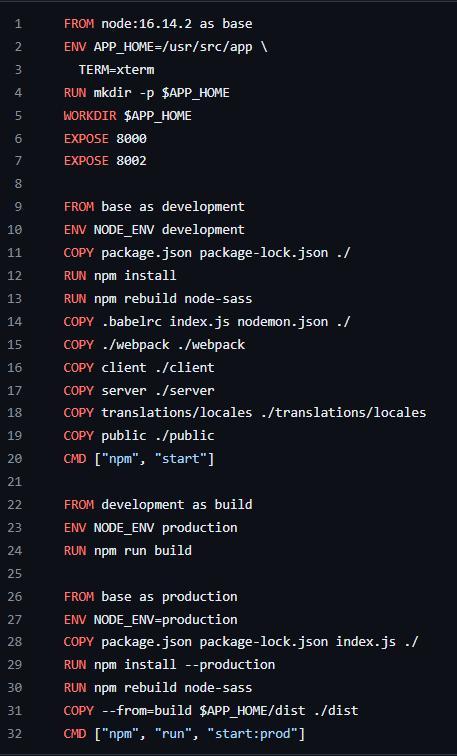
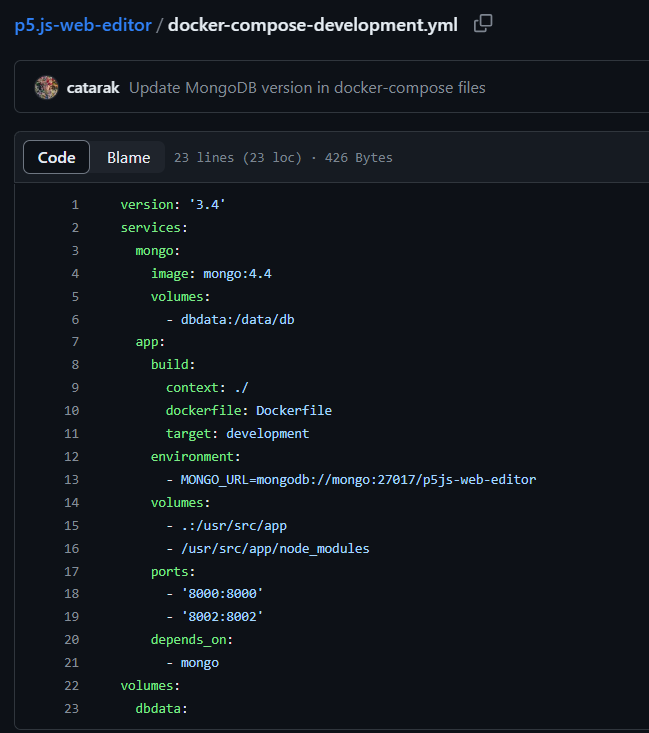


Open source project:

This is an open source project which comes to GSOC very often.

[GitHub - processing/p5.js-web-editor: p5.js Web Editor, officially launched!](https://github.com/processing/p5.js-web-editor)

Let’s explore the docker part in this project.

Open source project setup video link: <https://www.youtube.com/watch?v=Bp1g3guEO0I>

This entire course videos:

Part 1: <https://youtu.be/fSmLiOMp2qI?list=PLVKLWop9wWA9h7qWQb6xByfXsRq18U-68>

Part 2: <https://youtu.be/KuCwrySinqI?list=PLVKLWop9wWA9h7qWQb6xByfXsRq18U-68>