

Docker Intro

Step 1 - Why Docker?



Docker/containers are important for a few reasons -

1. Kubernetes/Container orchestration
2. Running processes in isolated environments
3. Starting projects/auxiliary services locally

Docker makes open-source project setup very easy.

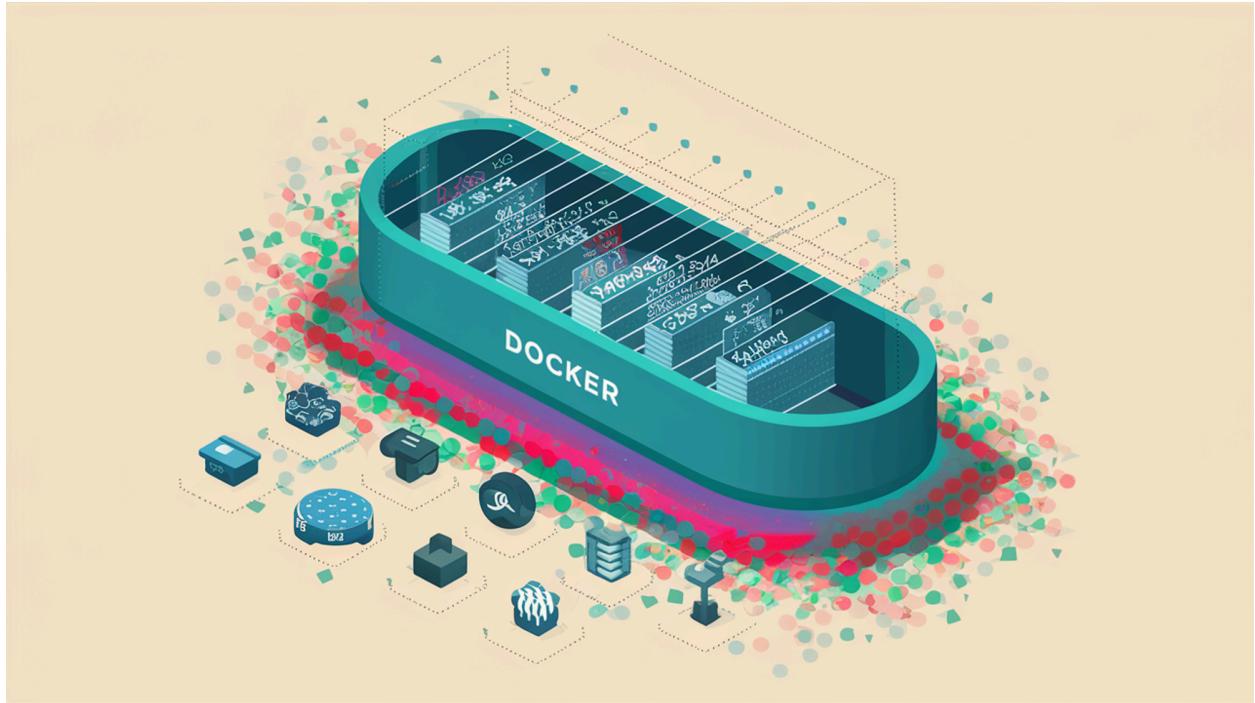
If we have isolated environment we can run many applications and they can't talk to each other

Docker-file make it very easy to start

Auxillary services: postgres etc

What is Containerization

What is Container



Containers are a way to package and distribute software applications in a way that makes them **easy to deploy** and **run consistently across different environments**. They allow you to package an application, along with all its dependencies and libraries, into a single unit that can be **run on any machine with a container runtime**, such as Docker.

Container are mini machine. Container which has some code/filesystem / network

Why Containers

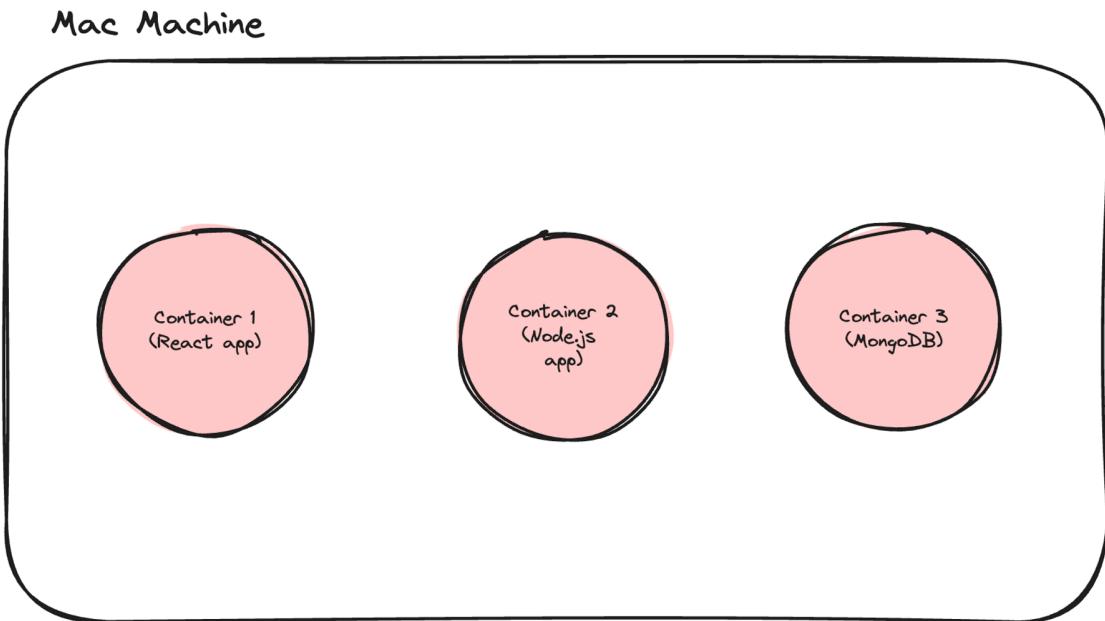
1. Steps to run a project can vary based on the OS

We can give same container to the MAC user or window user and they have to run a single command

That container has everything installed node/mongo/etc

2. Extremely hard to keep track of dependencies as project grows.

Benefits of using container



- Let us describe our configuration in a single file.
- Can run in isolated environment (hard for to take complete CPU or communicate to mongoDB)
- Makes local setup easy OS easy
- Makes installing auxiliary services easy

Example: docker run -d -p 27017:27017 mongo

Docker isn't the only way to create containers.

History of Docker

<https://www.ycombinator.com/blog/solomon-hykes-docker-dotcloud-interview/>

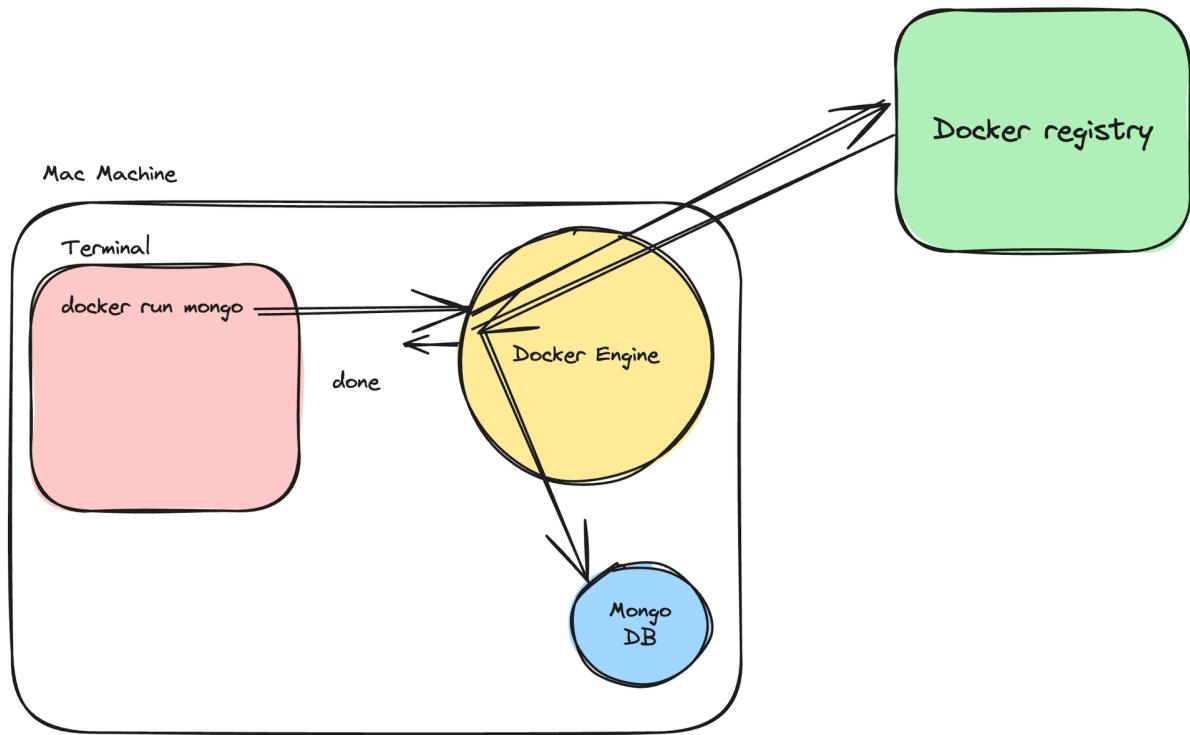
Installing Docker

<https://docs.docker.com/engine/install/>

Dockerfile

docker.yml

Inside docker



Terminal talks to docker engine (starts container download images etc).

Codebase in github

What is mongo image, it is similar to codebase, if we want to setup mongodb locally it will get the image from somewhere

- Docker engine

- Docker cli
- Docker registry

Docker engine

Docker Engine is an open-source containerization technology that allows developers to package applications into container

Containers are standardized executable components combining application source code with the **operating system (OS) libraries** and **dependencies required** to run that code in any environment.

Docker CLI

Command line interface lets us talk to the docker engine and lets you start/stop/list containers

```
docker run -d -p 27017:27017 mongo
```

Docker cli is not the only way to talk to a docker engine. You can hit the docker REST API to do the same things

```
[+]

ntc@ntc-HP-Pavilion-Laptop-15-eh1xxx:~$ docker

Usage: docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Common Commands:
  run      Create and run a new container from an image
  exec    Execute a command in a running container
  ps       List containers
  build   Build an image from a Dockerfile
  pull    Download an image from a registry
  push    Upload an image to a registry
  images  List images
  login   Log in to a registry
  logout  Log out from a registry
  search  Search Docker Hub for images
  version Show the Docker version information
  info    Display system-wide information

Management Commands:
  builder  Manage builds
  buildx*  Docker Buildx
  compose*  Docker Compose
  container  Manage containers
  context   Manage contexts
  debug*   Get a shell into any image or container.
  dev*     Docker Dev Environments
  extension* Manages Docker extensions
  feedback* Provide feedback, right in your terminal!
  image    Manage images
  init*   Creates Docker-related starter files for your project
  manifest  Manage Docker image manifests and manifest lists
  network  Manage networks
  plugin   Manage plugins
```

Docker Registry

The docker registry is how Docker makes money.

It is similar to github, but it lets you push images rather than sourcecode

Docker's main registry - <https://dockerhub.com/>

Mongo image on docker registry - https://hub.docker.com/_/mongo

If we want to upload in AWS better idea to is get image from AWS registry

Images Vs Container

Docker image

A Docker image is a lightweight, standalone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and config files.

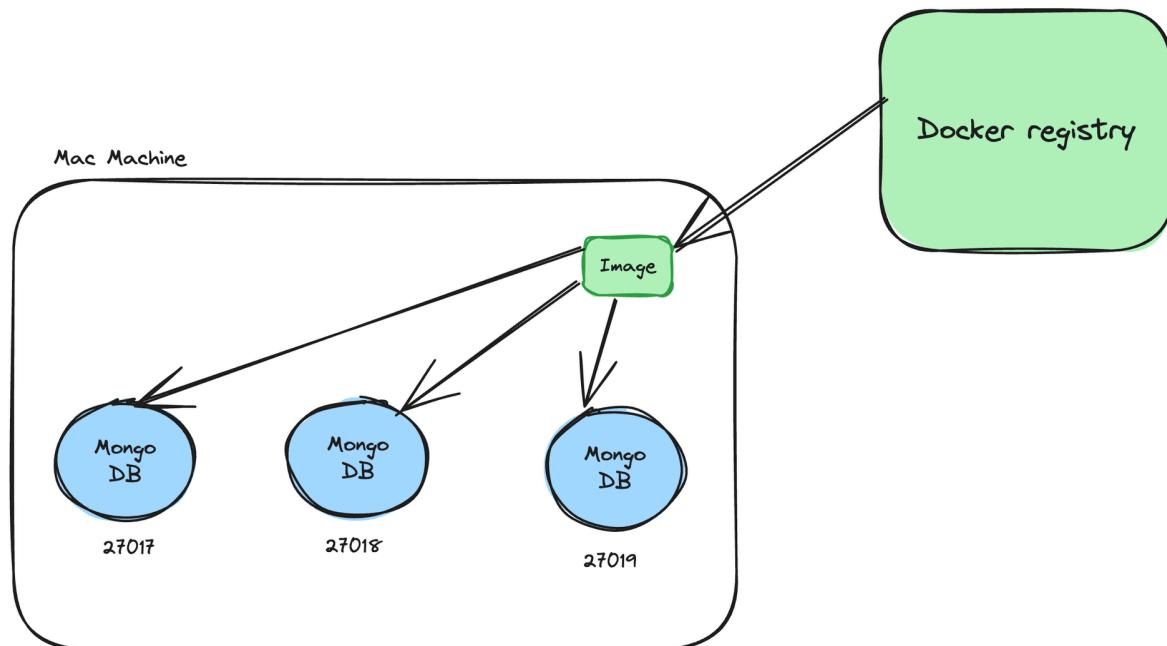
A good mental model for an image is **Your codebase on github**

Has bunch of libraries environmental etc need to run it

Docker container

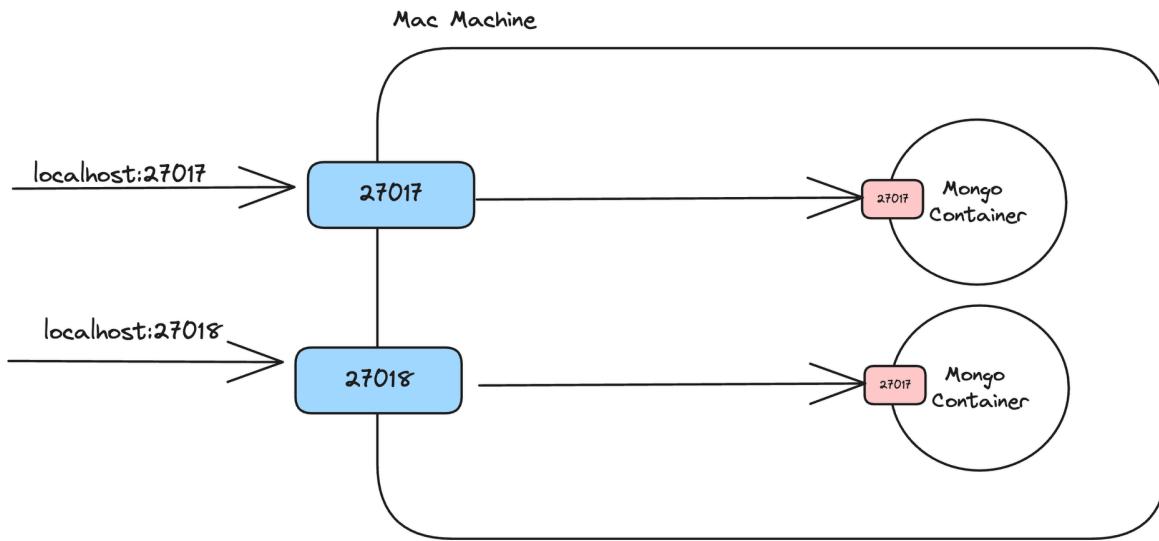
A container is a running instance of an image. It encapsulates the application or service and its dependencies, running in an isolated environment.

A good mental model for container is **node index.js**



docker images : to see the images on the docker

docker ps : to see the containers



running

Port Mapping

docker run -d -p 27018:27017 mongo

First port if of our machine second if of the container

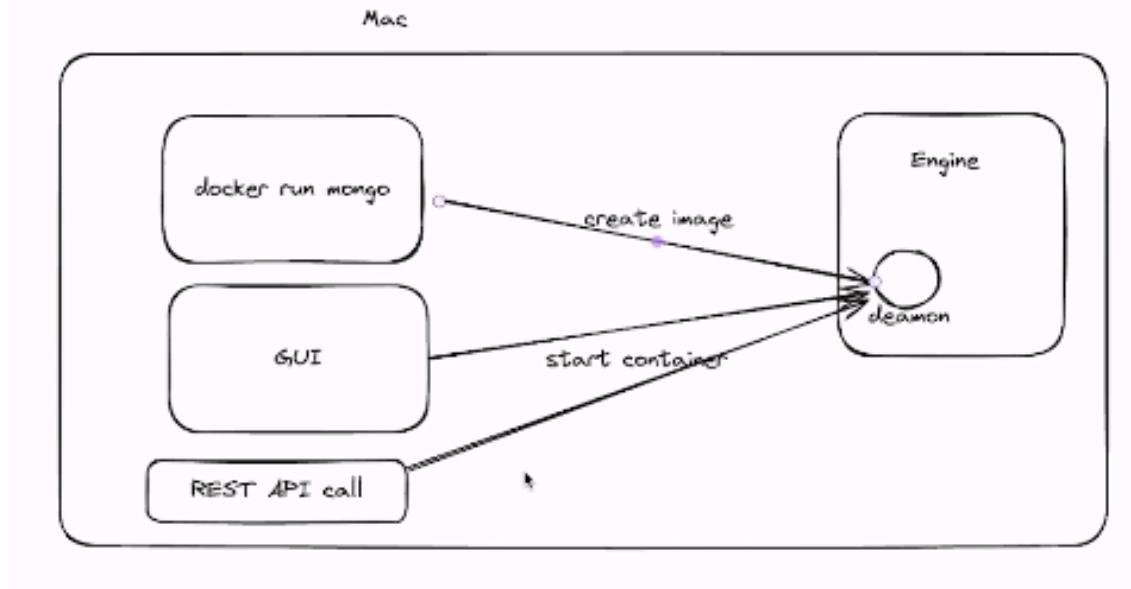
Containers are very isolated, whatever container we are starting. It means even if the mongodb is listening on 27017 port then it is listening on its own set of port its is different from mac 27017 port

Recap:

Docker is something which let us run container .

Docker has a demon , create image / start container . There are various way to contact to deamon.

docker kill



Socket_url then convert it to http url

We either

1. Create an image and push to docker hub
2. Or we start container locally using other people images/our images
3. Deploy to AWS machine using docker/kubernetes

Common docker commands

- **docker images** : Shows you all the images that you have on your machine
- **docker ps** : Shows you all the containers you are running on your machine
- **docker run**

Lets you start a container

-p ⇒ let's you create a port mapping

-d ⇒ Let's you run it in detached mode (terminal becomes free)

- **docker build**

Lets you build an image. We will see this after we understand how to create your own Dockerfile

- **docker push**

Lets you push your image to a registry

- **docker kill container_id**
- **docker exec**
- **docker pull mongo**

Dockerfile

These dockerfile is created once , they are not changed very often , if we want to create opensource project where we want others to contribute use it. If we create a database of our own its good idea to publish to docker registry.

What is Dockerfile

If you want to create an image from your own code, that you can push to dockerhub, you need to create a Dockerfile for your application.

A Dockerfile is a text document that contains all the commands a user could call on the command line to create an image.

How to write a Dockerfile

A dockerfile has 2 parts

Base image

Bunch of commands that you run on the base image (to install dependencies like Node.js)

Let's write our own Dockerfile

Let's try to containerise this backend app -

<https://github.com/100xdevs-cohort-2/week-15-live-1>

```
👉 Dockerfile
1  FROM node:16-alpine → Base Image
2
3  WORKDIR /app → Working directory
4
5  COPY . . → Copy over files
6
7  RUN npm install → Run Commands to build the code
8  RUN npm run build
9
10 EXPOSE 3000 → Expose ports
11
12 CMD ["node", "dist/index.js"] → Final command that runs when running the
13
14
FROM node:20
WORKDIR /app
COPY . .
RUN npm install
RUN npx prisma generate
RUN npm run build
EXPOSE 3000
CMD ["node", "dist/index.js"]
```

Every dockerfile need to have a baseimage at the top, we can even select UBUNTU and then choose the nodejs
Base image is from where we will start, the version we are using is 16-alpine.
Alpine means we get trimmed image.

Common commands

- **WORKDIR** : Sets the working directory for any **RUN**, **CMD**, **ENTRYPOINT**, **COPY** instructions that follow it. (put everything in the app)
- **RUN** : Executes any commands in a new layer on top of the current image and commit the results
- **CMD**: Provides defaults for executing a container. There can only be one CMD instruction in a dockerfile.
- **EXPOSE**: Informs Docker that the container listens on the specified network ports at runtime.
- **ENV** : Sets the environment variable
- **COPY** : Allow files from the Docker host to be added to the Docker image

Copy everything from this folder over the working directory of the images. Copy source code of projects including node_modules

```
FROM node:20

WORKDIR /app

COPY . .
# we are currently copying even node_module so add node_module to .dockerignore

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

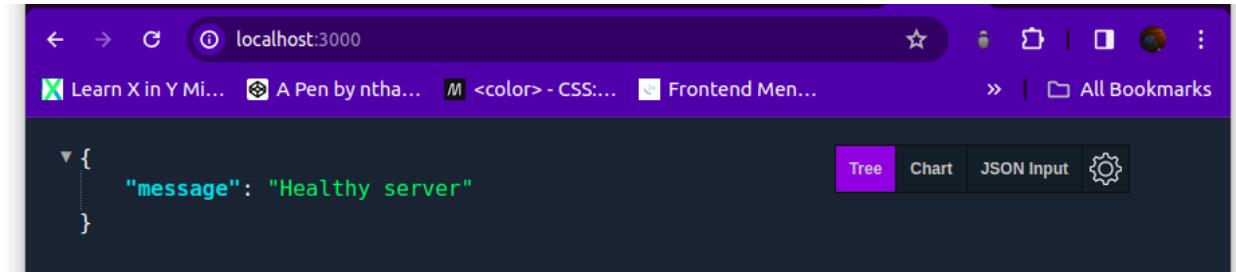
CMD ["node","dist/index.js"]
# why didn't we use RUN node index.js
# All the code run when we starting the image/container
# actually start our application
```

docker build -t backendApp . (this is basically the name of image of docker and . represent from where we want to build image from)

We created our own image and now if we want we can push to docker hub

```
ntc@ntc-HP-Pavilion-Laptop-15-eh1xxx:~/Desktop/Dev/dockerfilePra
● ctice$ docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
backend          latest   262d71f4ddc2  6 hours ago  1.3GB
mongo            latest   fb4debd65238  6 days ago   759MB
postgres         latest   b9390dd1ea18  7 weeks ago  431MB
ntc@ntc-HP-Pavilion-Laptop-15-eh1xxx:~/Desktop/Dev/dockerfilePra
○ ctice$ █
 9 main ⌂ ⑧ 0 △ 0  « 0           UTF-8  LF  Dockerfile  ⚡ Go Live  📁  ⚙️ Prettier  📨
```

docker run -p 3000:3000 backend



Backend running in a machine inside a container we also did put a port

Passing in env variables

docker run -p 3000:3000 -e

DATABASE_URL="postgres://avnadmin:AVNS_EeDiMIdW-dNT4Ox9l1n@pg-353

39ab4-harkirat-d1b9.a.aivencloud.com:25579/defaultdb?sslmode=require"

image_name

The -e argument let's you send in environment variables to your node.js app

Docker is really good at caching

Hence second time it will be very fast

Docker use for database locally don't use for production

How will final container will have the environment variable we cant push env in github.

Anytime you change the env update the DB (migrate)

We may think to write it in Dockerfile. We don't even want it in docker

Hence when we start the docker then we will **pass the env**

More commands

List all contents of a container folder

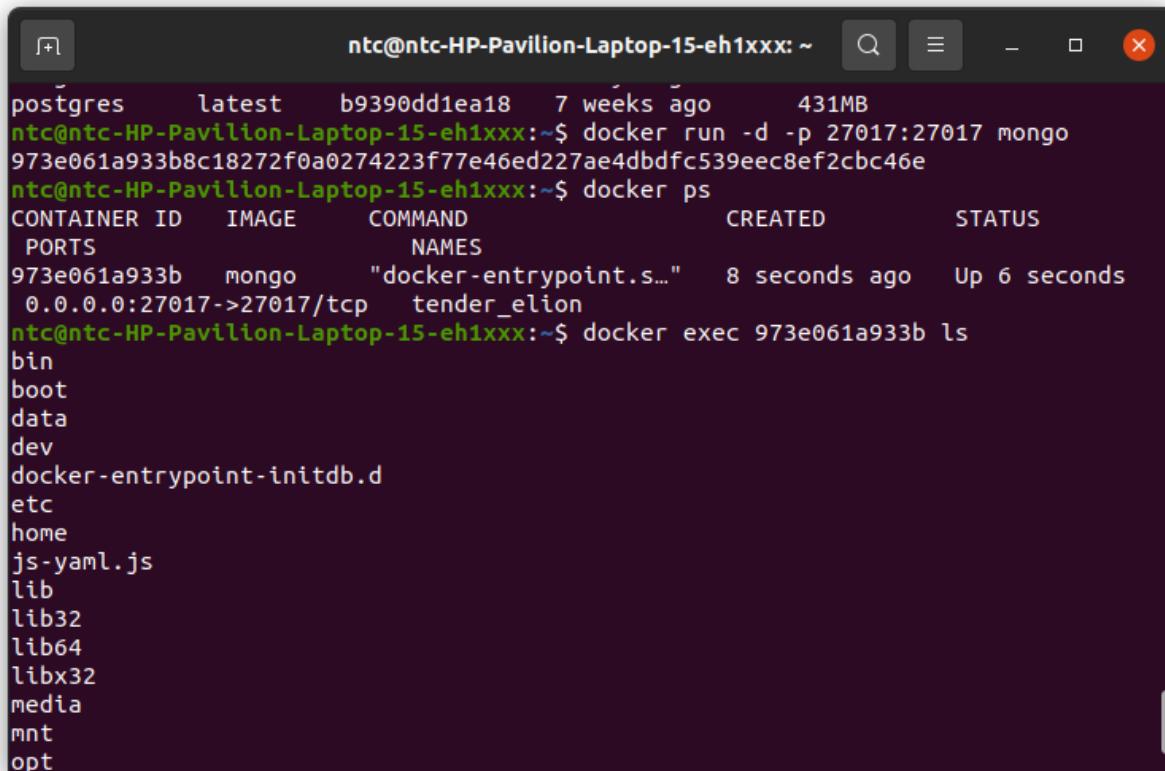
docker exec <container_name_or_id> ls /path/to/directory

Running an interactive shell

docker exec -it <container_name_or_id> /bin/bash

Interactively means that it doesn't exit once we run the command used when we want to run a series of commands.

/bin/bash means that we



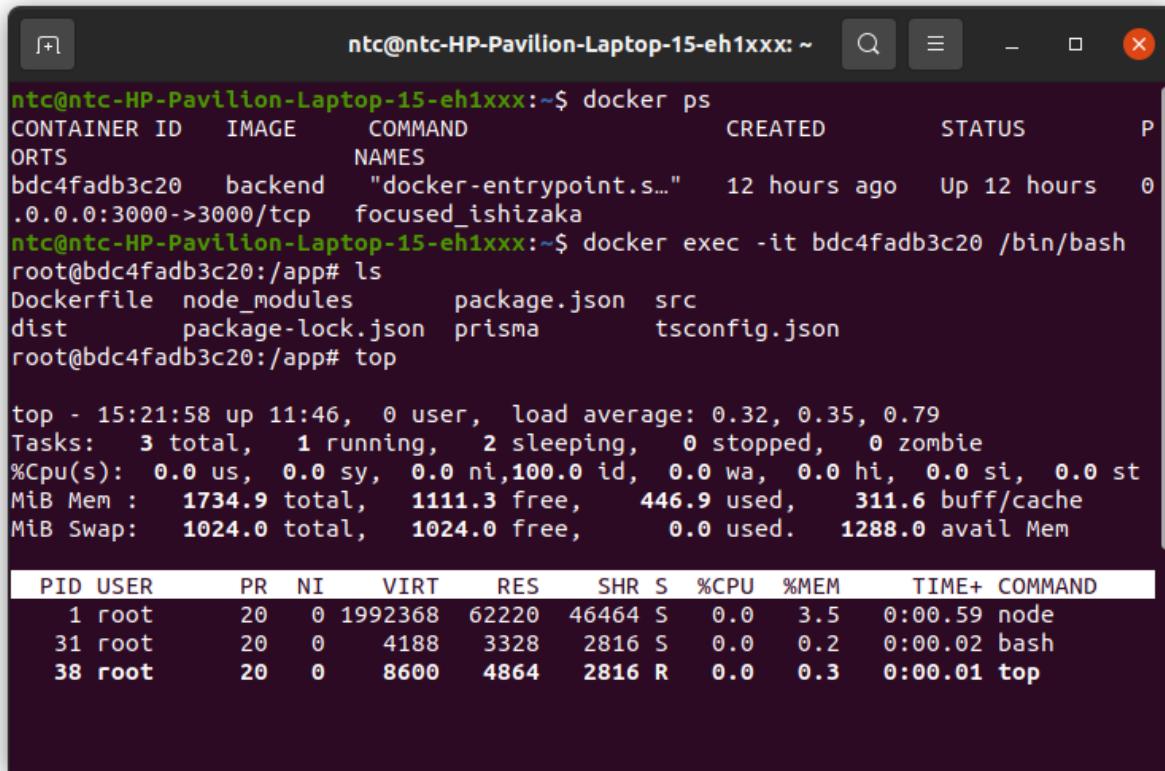
```
ntc@ntc-HP-Pavilion-Laptop-15-eh1xxx: ~
postgres      latest      b9390dd1ea18    7 weeks ago      431MB
ntc@ntc-HP-Pavilion-Laptop-15-eh1xxx:~$ docker run -d -p 27017:27017 mongo
973e061a933b8c18272f0a0274223f77e46ed227ae4dbdfc539eec8ef2cbc46e
ntc@ntc-HP-Pavilion-Laptop-15-eh1xxx:~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
          NAMES
973e061a933b        mongo              "docker-entrypoint.s..."   8 seconds ago       Up 6 seconds
     0.0.0.0:27017->27017/tcp   tender_elion
ntc@ntc-HP-Pavilion-Laptop-15-eh1xxx:~$ docker exec 973e061a933b ls
bin
boot
data
dev
docker-entrypoint-initdb.d
etc
home
js-yaml.js
lib
lib32
lib64
libx32
media
mnt
opt
```

`docker rmi <image_name>`

Will delete the image and free up the space

- **--force** will forcefully do something

Execute a command inside the container to get the shell access.



The screenshot shows a terminal window with the following session:

```
ntc@ntc-HP-Pavilion-Laptop-15-eh1xxx:~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
bdc4fadb3c20        backend            "docker-entrypoint.s..."   12 hours ago       Up 12 hours        0.0.0.0:3000->3000/tcp
ntc@ntc-HP-Pavilion-Laptop-15-eh1xxx:~$ docker exec -it bdc4fadb3c20 /bin/bash
root@bdc4fadb3c20:/app# ls
Dockerfile  node_modules  package.json  src
dist        package-lock.json  prisma      tsconfig.json
root@bdc4fadb3c20:/app# top
top - 15:21:58 up 11:46,  0 user,  load average: 0.32, 0.35, 0.79
Tasks:  3 total,   1 running,   2 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem : 1734.9 total, 1111.3 free,   446.9 used,   311.6 buff/cache
MiB Swap: 1024.0 total, 1024.0 free,     0.0 used. 1288.0 avail Mem

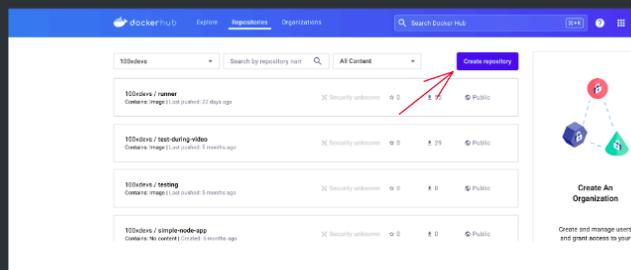
PID  USER      PR  NI    VIRT    RES    SHR S %CPU %MEM TIME+ COMMAND
 1  root      20   0 1992368  62220  46464 S  0.0  3.5  0:00.59 node
 31 root      20   0    4188   3328   2816 S  0.0  0.2  0:00.02 bash
 38 root      20   0    8600   4864   2816 R  0.0  0.3  0:00.01 top
```

Pushing to Dockerhub

Step 14 - Pushing to dockerhub

Once you've created your image, you can push it to [dockerhub](#) to share it with the world.

1. Signup to [dockerhub](#)
2. Create a new repository



3. Login to docker cli
1. docker login
2. you might have to create an access token - <https://docs.docker.com/security/for-developers/access-tokens/>
4. Push to the repository

```
docker push your_username/your_reponame:tagname
```

Sign up in dockerhub

nthapa0000/week_15-class

Created less than a minute ago

This repository does not have a description

Docker commands

To push a new tag to this repository:

```
docker push nthapa0000/week_15-class:tagname
```

[Public View](#)

Tags

This repository is empty. Push some images to it to see them appear here.

Automated Builds

Manually pushing images to Hub? Connect your account to GitHub or Bitbucket to automatically build and tag new images whenever your code is updated, so you can focus your time on creating.

Available with Pro, Team and Business subscriptions. [Read more about automated builds](#)

[Upgrade](#)

```
docker push nthapa0000/week_15-class:tagname
```

This is done so that we can't pollute the namespace of the docker and create mongo etc

Go to the terminal

```
docker build -t nthapa0000 .
```

Now push to dockerhub

Before pushing sign in to docker hub

Copy Access Token

When logging in from your Docker CLI client, use this token as a password. [Learn more](#)

ACCESS TOKEN DESCRIPTION

Practice

ACCESS PERMISSIONS

Read, Write, Delete

To use the access token from your Docker CLI client:

1. Run `docker login -u nthapa0000`
2. At the password prompt, enter the personal access token.

```
dckr_pat_KdbVMTADn7E-4AYwdRuyIheXIN8
```

[Copy](#)

`dckr_pat_KdbVMTADn7E-4AYwdRuyIheXIN8`

```
docker push <repository_name>:tagname
```

Tagname can be like V1

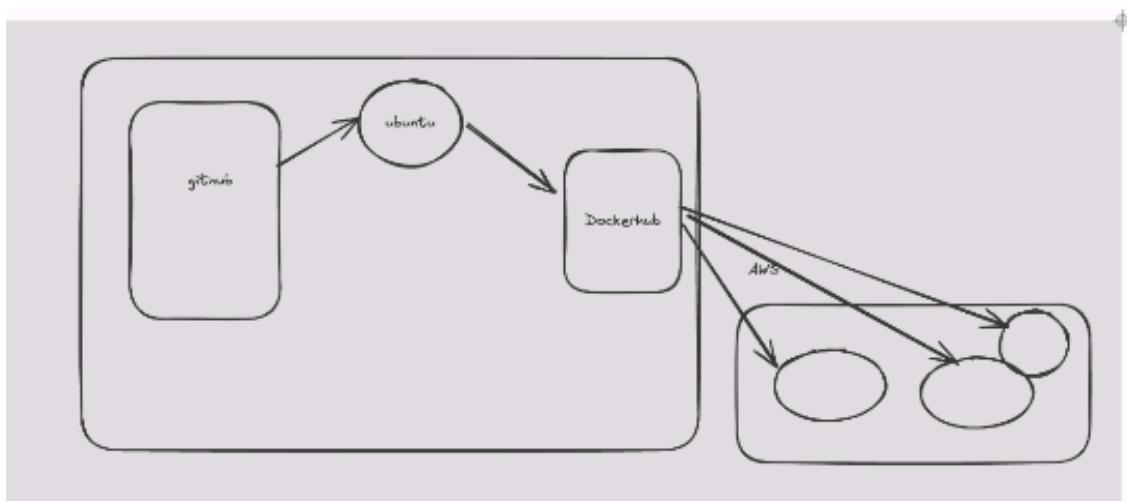
V2 etc anything we want

```

● 2.2$ docker images
REPOSITORY          TAG      IMAGE ID      CREATED
SIZE
nthapa0000          latest   786f6f27d328  9 minutes ago
o  1.16GB
nthapa0000/week_15-class  latest   786f6f27d328  9 minutes ago
o  1.16GB
ntc@ntc-HP-Pavilion-Laptop-15-eh1xxx:~/Desktop/Dev/week-15-live-
○ 2.2$ docker push nthapa0000/week_15-class
Using default tag: latest
The push refers to repository [docker.io/nthapa0000/week_15-clas
s]
ef3fea1538ed: Pushing  11.78kB
259edfcfd2f10: Pushed
ebc796be3914: Pushing  31.45MB/58.18MB
f7a3b9af30c7: Pushed
bd66bfe67719: Pushed
405973eec1c7: Pushing  3.584kB
1748e4c4bd22: Pushing  278kB/7.521MB
636e6fa01167: Waiting
3e81cc85b636: Waiting
893507f6057f: Waiting
2353f7120e0e: Waiting
51a9318e6edf: Waiting
c5bb35826823: Waiting

```

CI/CD every commit we are pushing to docker hub with the commit id



docker-compose

.yml describe data in form of key value pair

Docker Compose is a tool designed to help you define and **run multi-container Docker applications**. With Compose, you use a YAML file to configure your application's services, networks, and volumes. Then, with a single command, you can create and start all the services from your configuration.

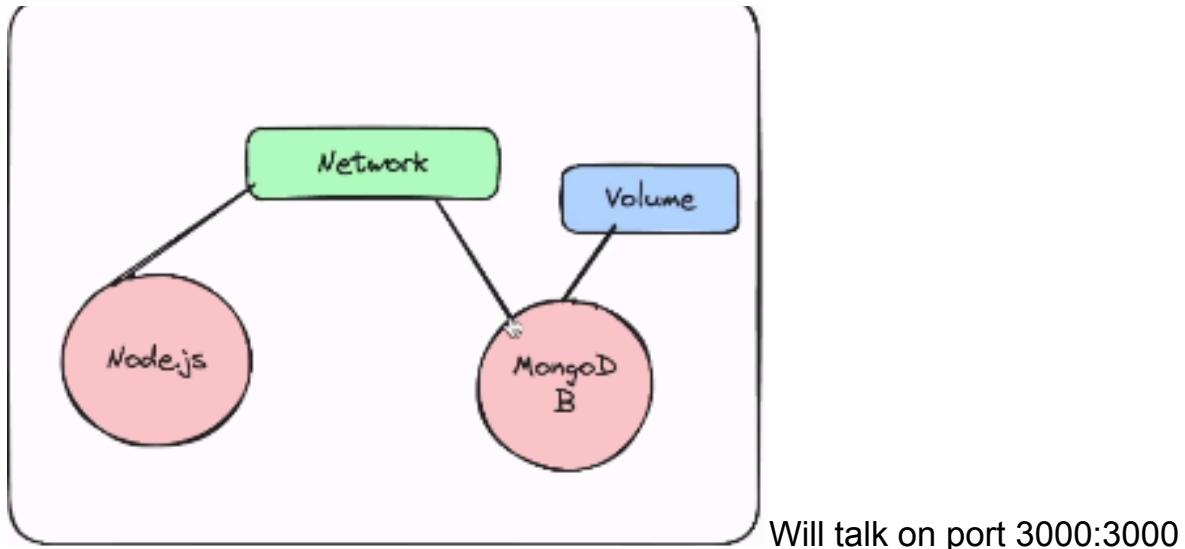
The diagram illustrates the structure of a Docker Compose configuration file (docker-compose.yaml) for a multi-container application. The file defines two services: 'mongodb' and 'backend22'. The 'mongodb' service is defined on lines 1 through 9, featuring a red box around the 'version' key. The 'backend22' service is defined on lines 11 through 20, featuring a red box around the 'depends_on' key. Both services have red boxes around their 'volumes' sections. A red bracket on the right side of the file groups the 'volumes' sections under the heading 'Volumes to create'. Annotations with arrows point from the red boxes to these labels: 'Version of docker-compose' points to the 'version' box; 'Mongo service' points to the 'mongodb' service section; 'Backend service' points to the 'backend22' service section; and 'Volumes to create' points to the 'volumes' boxes.

```
docker-compose.yaml
1 version: '3.8'                                Version of docker-compose
2 services:
3   mongodb:
4     image: mongo
5     container_name: mongodb
6     ports:
7       - "27017:27017"
8     volumes:
9       - mongodb_data:/data/db
10
11  backend22:
12    image: backend22
13    container_name: backend_app
14    depends_on:
15      - mongodb
16    ports:
17      - "3000:3000"
18    environment:
19      MONGO_URL: "mongodb://mongodb:27017"
20
21 volumes:
22   mongodb_data:                                Volumes to create
```

Talk to each other using name.

We have define volume

What problem does it solves??



Will talk on port 3000:3000

Till now we have used this approach to run multi container applications.

In future we may have small golang microprocess and multiple container talking to each other and we may need multiper network

What if we can describe all this in a single file

Before docker-compose

- Create an network

```
docker network create my_custom_network
```

- Create a volume

```
docker volume create volume_database
```

- Start mongo container

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

- Start backend container

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

Container can talk to each other without any port , port are generally for the host machine

After docker-compose

1. Install docker-compose <https://docs.docker.com/compose/install/>
2. Create a yaml file describing all your container and volumes (by default all containers in a docker-compoese run on the same network)

```
version: '3.8'
services:
  mongodb:
    image: mongo
    container_name: mongodb
    ports:
      - "27017:27017"
    volumes:
      - mongodb_data:/data/db

  backend22:
    image: backend
    container_name: backend_app
    depends_on:
      - mongodb
    ports:
      - "3000:3000"
    environment:
      MONGO_URL: "mongodb://mongodb:27017"

volumes:
  mongodb_data:
```

This much command is

```
mongodb:  
  image: mongo  
  container_name: mongodb  
  ports:  
    - "27017:27017"  
  volumes:  
    - mongodb_data:/data/db
```

Same as :

```
docker run -name mongodb -p 27017:27017 -v mongodb_data:/data/db  
mongo
```

Docker compose understand this commands and runs multiple docker run commands

```
backend22:  
  image: backend  
  container_name: backend_app  
  depends_on:  
    - mongodb  
  ports:  
    - "3000:3000"  
  environment:  
    MONGO_URL: "mongodb://mongodb:27017"
```

Ports are array

Whenever we have multiple container in docker-compose it starts them in same network , volume we have to define in same level as services

docker-compose up