docker

→ Install using official doc.

Building blocks of docker → container, images and volumes

Image

These are just disk images. official images are provided at dockerhub where we can explore the various disk images.

Install them and run in docker environment.

Containers

systems that are capable of installing and running the disk images.

Use → docker pull image-name
to pull a particular image from web. (or it may already be install).
(we still have not installed this image on any containers.

docker ps -a
↳ lists down the containers.

To install a disk image of a container, we first need to run a container. docker run disk-image name works but not for all images.

postgres docker run --name "c name" -e POSTGRES_PASSWORD=mysecretpassword -d postgres

It will start running the postgres instance of a new container.
If we get out of the terminal, the instance will stop.
If we run again, a new container will be created that will run that image for us.
We can stop this container either from GUI or using docker stop container's name or id
To rerun a container, use GUI or simply docker run container name

* The genius of docker is that it contains layers of os and some configurations only making it much light weight than VMs. It does not even have its own kernel. If needed, it uses APIs to handle such requests. Despite being so lightweight it is capable enough to handle the image installed in its containers.

We can install a disk Image and run multiple containers with instances of the disk image installed (multiple versions). Then we can assign ports to these containers. Then we can connect these systems running on containers to the systems utilizing it.

It is highly useful for connecting with mongo, postgres and some systems that otherwise would have required installation

on our local system. ♡

→ docker container prune removes all the stopped containers.

Next, we pull a mongo image using docker pull mongo.

Then, run a container with the mongo image using

docker run --name some-name -d mongo:tag

Mongo usually runs on 27017

Now, we set up new port for running mongo (or actually any image)

docker run --name some-name -p new port: default port -d mongo

we can avoid tag if we want latest version of the image to be installed

(All the containers must have unique names)

docker logs container-name gives insights of the container. It provides various options to get specific data of the container.

Connecting two applications running on seperate containers

First of all make sure that both applications are containerized. Then we have to set up a network over which the two applications will be connected. We will use docker network commands for this purpose.

Network creation → docker network create -driver bridge some-network

We can check networks using

→ docker network ls

Now that a network is set up, run the containers for the 2 apps by providing correct env variables using -e options and correct network reference using --net some-network

Example → 2 Apps → Mongo

envs
MONGO_INITDB_ROOT_USERNAME
= username
MONGO_INITDB_ROOT_PASSWORD
= password

mongodb-server (exp)
↓ envs
ME_CONFIG_MONGODB_ADMIN
USERNAME =
ME_CONFIG_MONGODB_ADMIN
PASSWORD =

- So, the two commands for running the containers conn'd over our network

1:→
docker run -p 27017:27017 -e MONGO_INITDB_ROOT_USERNAME = username --name mongodb -e MONGO_INITDB_ROOT_PASSWORD=password --net mongo-network -d mongo

2:→
docker run -p 8081:8081 -e ME_CONFIG_MONGODB_ADMINUSERNAME = username -e ME_CONFIG_MONGODB_ADMINPASSWORD = password --name mongo-express --net mongo-network -d mongo-express

This method of connecting containers is not feasible if number of such containers is large.
So, we use a file to maintain connection.
We take help of docker compose command for this purpose.

The file that we are going to use here would be in .yaml format
Data to be put inside the yaml file:-

```
version:
services:
  mongodb:
    image: mongo
    ports:
      - "27017:27017"
    environment:
      - key=value

  mongo-express:
```

```
- image: mongo-express
  ports
    - "8081:8081"
  environment
```

In your terminal, go the loc of the file and use command:

```
docker-compose -f file-name.yaml up
```

We need not create & maintain a separate network here as a new network is automatically setup.

One problem may occur if the containers are required to run in a particular order. We just simply include a characteristics restart: always in the containers that have to be run later.

To maintain data persistence, we define volumes too. In the yaml file, include:

```
volumes:
  mongo-data:
    driver: local
```

and in the mongo:

```
volumes:
  - mongo-data:/data/db
```

How to containerize an application

In order to containerize an app, we build a disk image of the application.

Go to your Project file, create a separate Dockerfile where we'll specify the docker build properties.

FROM base module → here checkout the base module of your app. Let's say app is in nodejs. We'll look for node base module in the dockerhub. Also, first of all create an a/c in dockerhub and use docker login cmd in terminal.

WORKDIR ____

COPY FROM TO

RUN ~~main code~~ installer

EXPOSE PORT

CMD to run the app

then use → docker build -t yourname/imgname:0.0.1.RELEASE .

↳ specifies where the Dockerfile is

-t for tag name specification

version

- Try run the image

  docker container run -d -p 3000:3000 image-name

- docker container stop imagename
- We can push our image to dockerhub so that anyone can pull it and run.
- docker push imagename

For containerizing a Go App

```
FROM golang:1.27 (maybe)
WORKDIR /app
COPY go.mod .
COPY main.go .
RUN go get
RUN go build -o bin
ENTRYPOINT ["/app/bin"]
```

But that's a bad way creating go app image. Rather we should multi-stage builds to utilize go's features