**Docker**

An application can have frontend, backend, database, messaging system etc.

All these will have different versions, different dependencies, libraries.

And different dependent OS.

To setup everything and run, it will be very difficult.

Called as ‘matrix from hell`.

In docker, all the above can run on their own container with their own libraries and containers.

Container : Completely isolated env, have their own processors, network interfaces etc, but share same OS kernel

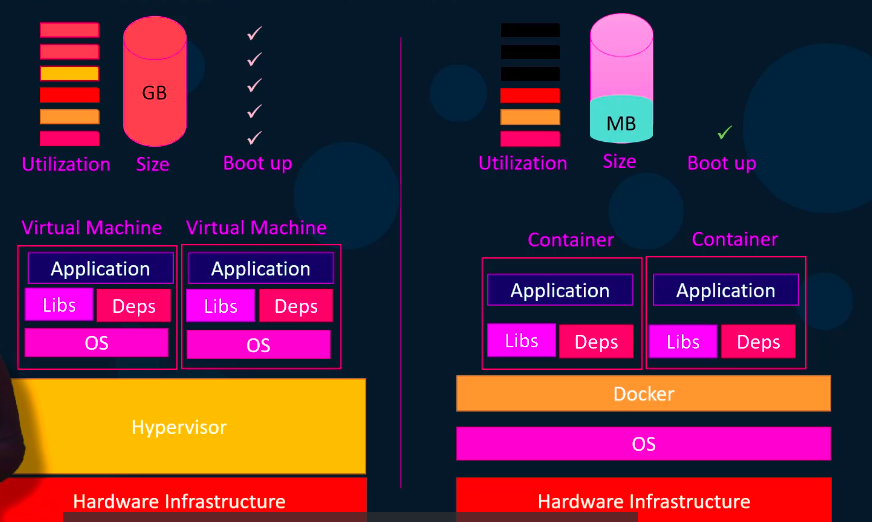
Docker uses LXC container.

OS kernel: interacts with hardware

We can run linux based container on a docker that is hosted on linux kernel

But we cant run windows based container on the same

Docker purpose : package, containerize, ship, run anywhere anytime.



Remove old docker

sudo apt-get remove docker docker-engine docker.io containerd runc

Install using script

curl -fsSL https://test.docker.com -o test-docker.sh

sudo sh test-docker.sh

Docker run appName

This will run an instance of app on docker host

If image is not there, it will go to docker hub and pull

For subsequent execution, it will use same image

Docker ps

Shows docker name, docker given name, status etc

Docker stop containername

Docker rm containerName

Docker rm containerId

Docker images

Docker rmi appname

To delete the docker image

Docker pull appname

Pulls the image

container runs only until the service in it is active

?? If its done executing / crashed, it will stop (to further check later)

Docker exec containername cat /etc/hosts

Shows the content of file inside this container

Docker run –d appname

To run in detached mode (background)

To run again in foreground,

Docker attach containerId

First few chars are enough

Dockerhub will have official and our application images

Docker run centos –it bash

Interactive bash in centos on container

If we try to remove the image without removing the container, then the following error comes

Error response from daemon: conflict: unable to delete 5d0da3dc9764 (must be forced) - image is being used by stopped container d2974ae342da

docker run --name webapp -d nginx:1.14-alpine

Webapp is container name, nginx:1.14-alpine is image

Docker run

Docker run imageName

Docker run imageName:tagName

By default container doesnt wait for standard inputs

Docker run –it imageName

i for interactive

T for terminal

Docker run –p 80:5000 aravisv/araviApp

Maps port 5000 to port 80.

So that other users can access from the ip address of my computer and with the port 80

Whatever data is stored in local will get erased once the container is stopped

Use the below command (-v => bind mount volume)

Docker run –v /opt/datadir:var/lib/mysql mysql

-v -> first param is name of volume unique on host ( laptop absolute path for example ), can omit of anonymous

Second param is where it is mounted in container (container path. /app the paths mentioned in DOCKER file can be put here)

?? ( not very clear )

Docker inspect containerName

Docker logs containerName

Docker run ubuntu:17.10 cat /etc/\*release\*

Docker run –d ubuntu sleep 15 => -d for detached mode

Docker attach containerId => will attach to the terminal

Docker inspect containerName

This will have the networks section, where you can get the **internal** IP address

Use this, and the port of the container, you will be able to access the container on web , if its a web application

To access it **externally**, map the port and run it

Docker run –p 8080:8080 jenkins

Docker images

Every docker file has instruction & arguments

Instructions: FROM, RUN, COPY, ENTRYPOINT etc

FROM – every docker image must be based on OS / another docker image

Dockerfile must start with FROM

RUN – run a particular command on those images

COPY – copies from local system to docker image

ENTRYPOINT – runs on the container image

Example:

---

FROM Ubuntu

RUN apt-get update && apt-get –y install python

RUN pip install flask flask-mysql

COPY . /opt/source-code

ENTRYPOINT FLASK\_APP=/opt/source-code/app.py flask run

---

To build the image =>

Docker build . -f Dockerfile –t imageName

Each line in the dockerfile will act as a layer

We can see the layer size using

docker history imageName

All layer builds are cached

If a step fails, you fix it and run the build again, it starts from that failed layer. Instead of from beginning

Docker login

Docker push imageName

If we want some variable outside the application, based on which the application behaves differently, we can do so using environment variable

Ex: background colour of the application

Inside the code (python) we can access using os.environ.get(“APP\_COLOR”);

And before running the application, we :

EXPORT APP\_COLOR=blue;

Python app.py

While running the container, add environment variables with –e parameter

Docker run –e APP\_COLOR=blue myAppImage

To find the environment variables of an image, do docker inspect imageName

Under config, “Env” environment variables are there

To override the default CMD in dockerfile, we can create another dockerfile

Use the existing imageName

Example:

----

FROM ubuntu

CMD sleep 5

Or

CMD [“sleep”,”5”]

---

Then build again

If we want to pass the seconds only while running the container,

---

FROM ubuntu

ENTRYPOINT [“sleep”]

---

Now whatever you type in command line will get appended to the sleep

Difference:

CMD will be replaced by command line parameters entirely

ENTRYPOINT appends the command line parameters

---

FROM ubuntu

ENTRYPOINT [“sleep”]

CMD [“5”]

---

If no params are passed it will be sleep 5

Otherwise whatever seconds are passed in terminal will be used. “5” will be replaced

If we want to override the entrypoint command while running,

Docker run –entrypoint different\_command params

**Docker compose**

Docker run –d --name=redis redis

...

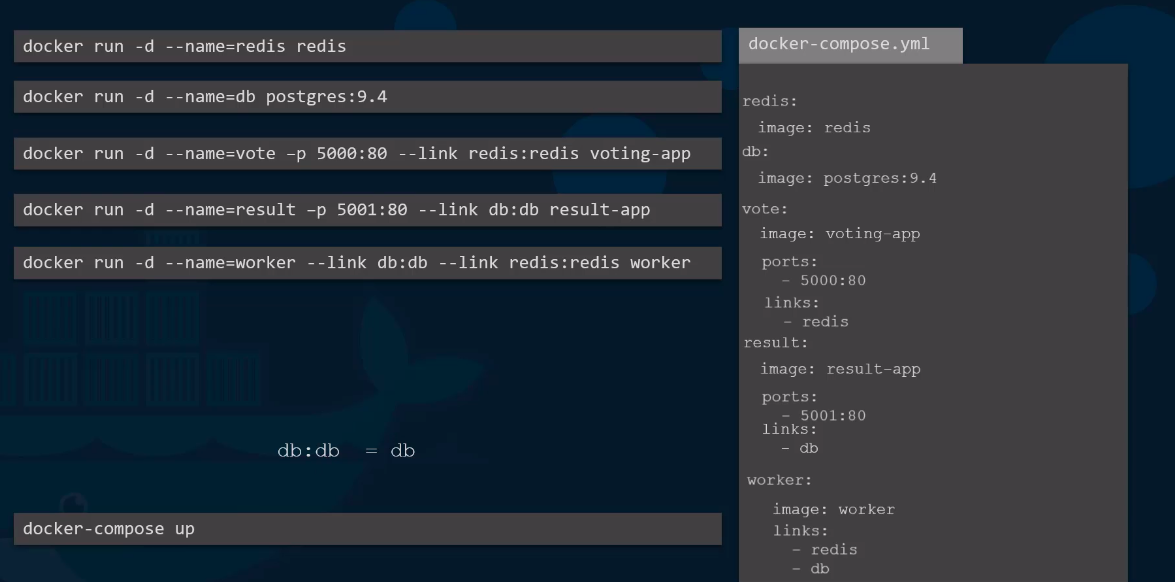
...

Docker run –d –name=vote –p 5000:80 –link redis:redis voting-app

Link is used to link the application container which requires another container

In the above example, lets say there is a code where it searches for redis service.

Thats running in redis container, hence the above docker command is used to link

Instead of running each docker run command one by one, we can do by  


If the image is not built yet, replace image:imageName in the yml file to build:applicationPath

In next versions, on top oy yml file, add

version:2

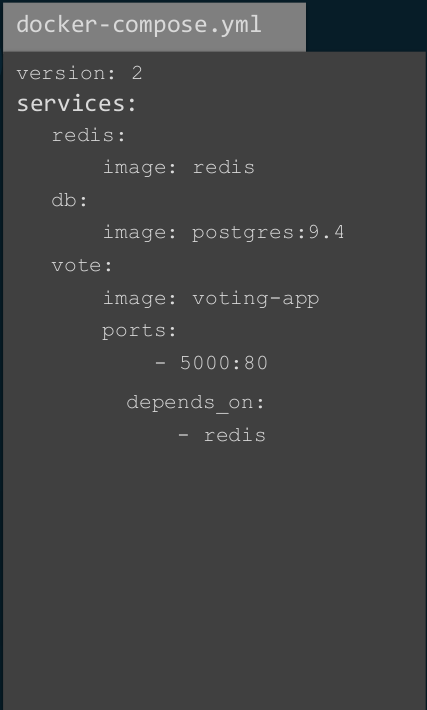
services:

redis:redis

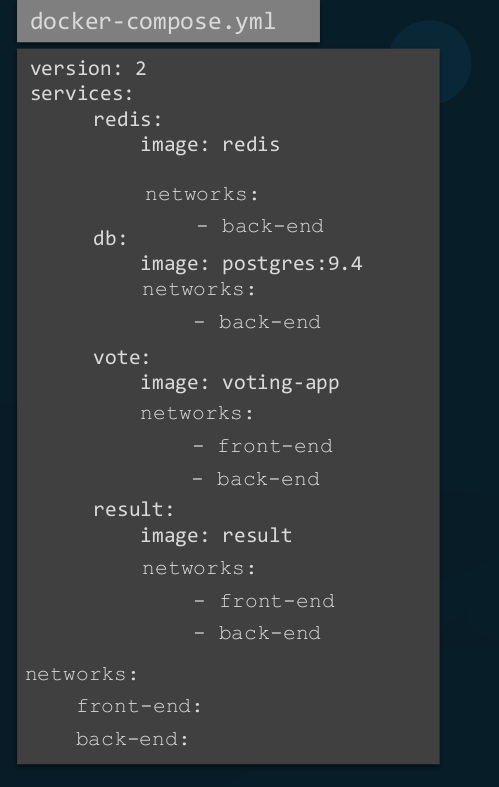
In version 1, docker container attaches all containers to default bridge network & use links

In next versions, automatically creates a dedicated nw. And then communicate each other using service names. No need of links.

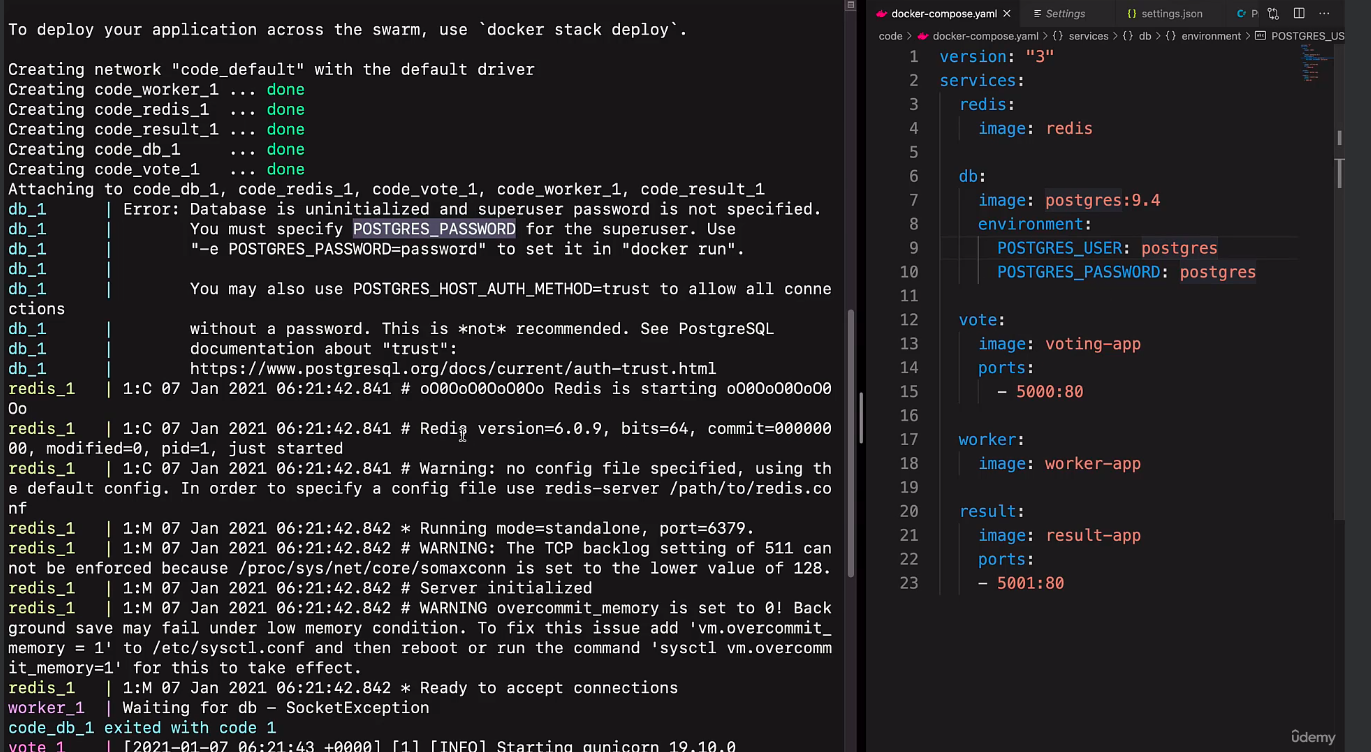
We can also specify the dependency of the services.



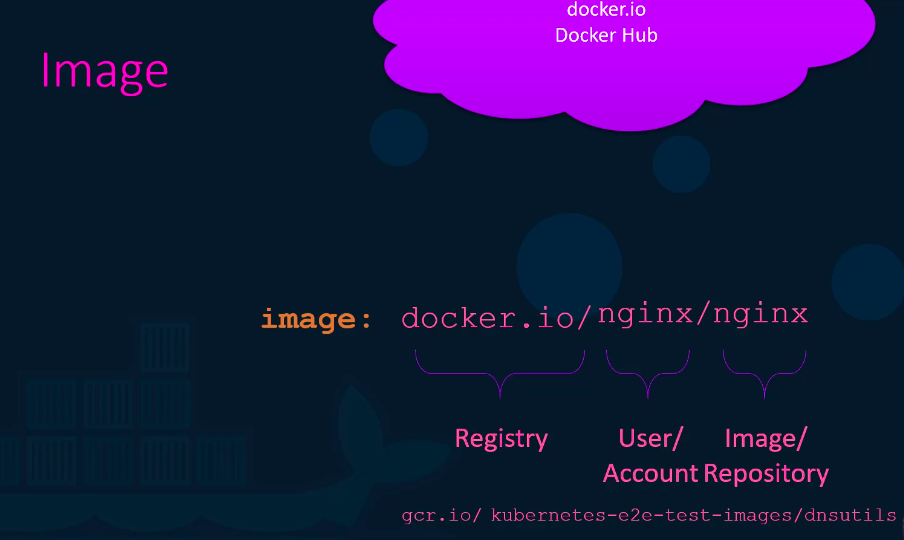
Or we can create our own network and connect containers



In version 3 there is no need to specify links



**Docker registry**



Use private registry for our application

Ex:

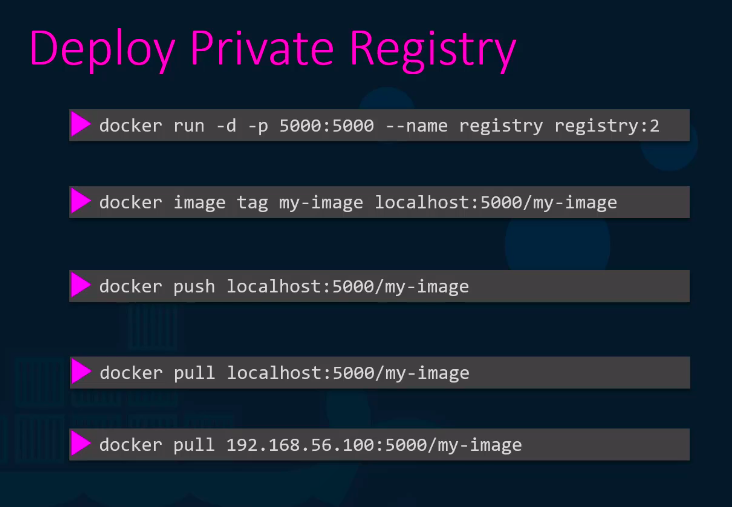
Docker login private-registry.io

(enter username, password)

Docker run private-registry.io/apps/internal-app

Our own private registry –

Registry is an image which helps achieve that



**Docker engine**

Consists of

Docker cli

REST API

Docker deamon

Docker cli can be on different host as well

Ex:

Docker –H=10.23.45.23:4000 run nginx

Uses namspace to isolate workspace.

Each container has its own PID as well as host’s PID

But within the scope of container, the PID is protected.

Container A can have a process with PID4

Container B can have a process with PID4

But in the host their PID will be different.

By default there is no restriction on the host cpu,memory utilization of the container.

We can limit using cgroups (control groups).

Docker run –cpus=0.5 ubuntu

Docker run –memory=100m ubuntu

50 percent of cpu ,100m => 100 megabytes

Docker exec containerId ps –eaf

Lists all the processes running in that container

Docker storage:

/var/lib/docker is the default folder

Container layer is the runtime layer (log or any new files, user inputs will be stored here). Read/Write

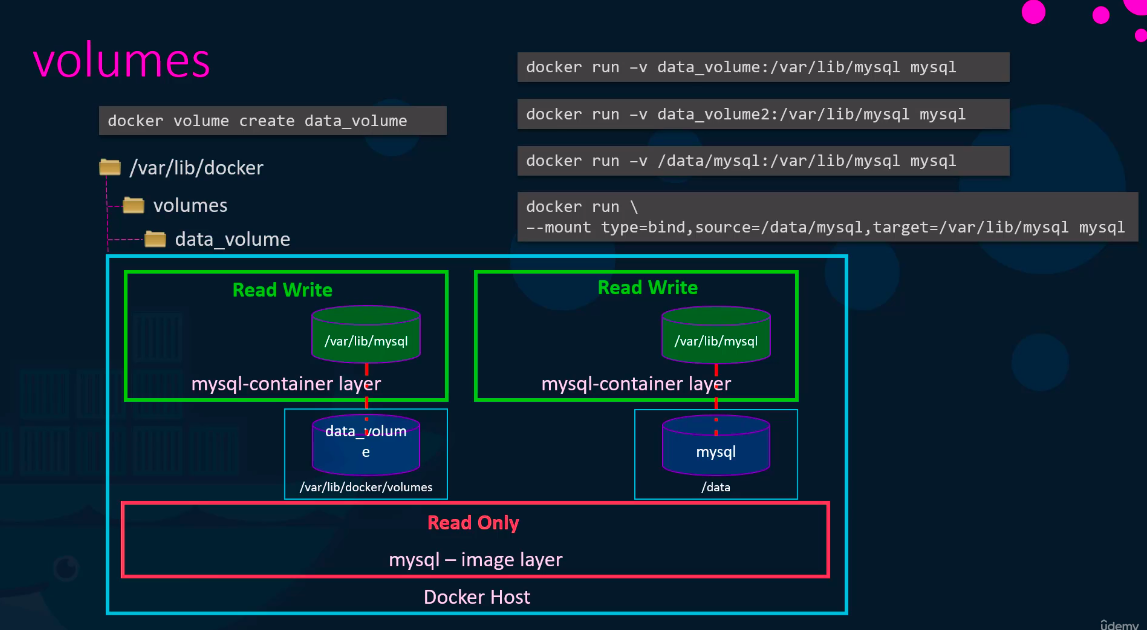
Deleted when image dies

Image layers. Read Only

Volume mounting and bind mounting

Volume mount stores the data inside /var/lib/docker/volumes

Bind mount stores data anywhere in the host



Docker history imageId

=> shows how the docker image is built

When doing docker build, if the action is already performed (image already available / ran a script to install some package etc) as part of another image build, it reuses that result from the cache.

This helps docker build the next image faster.

But if the cache is not there at particular level, it will create new one and build the next layers again (even if the next layers have the cache).

Layers are stored in docker/aufs/diff/

Du –fs \*

=> gives the file size of each layer

Docker system df –v   
=> displays information regarding the amount of disk space used by the Docker daemon

Docker network:

Docker network ls

3 kinds of networks: host, none, default

Docker network create –driver –subnet 182.18.0.0/16 custom-isolated-network-name

**Container orchestration**

To maintain the health of containers using tools, scripts

Automatically start new container if the container or host itself crashes

Can scale up the number of individual containers or scale down based on the usage

Advance networking bw diff hosts and their containers

Docker service create –replicas=100 imageName (to be run in the swarm manager)

Solutions: docker swarm, kubernetes, mesos

Docker swarm:

One swarm manager and multiple worker hosts (nodes)

Docker swarm init (to be run in swarm manager)

Docker swarm join –token <token> (to be run in worker)

Kubernetes:

Kubectl run –replicas=1000 imageName

Kubectl scale –replicas=2000 imageName

Rolling update, rollback, autoscalres, different plugins supported, authentication

Kubernetes cluster consists of nodes

Master watches, manages workers

Kubectl run imageName

Kubectl cluster-info

Kubectl get nodes

Article link

<https://www.freecodecamp.org/news/how-to-dockerize-a-react-application/>

WORKDIR /app

Does this create a folder in our workspace itself

Or in docker environment?

Out of scope notes

For a computer connected to a network with an IP address, a port is a communication endpoint. Ports are designated by numbers, and below 1024 each port is associated by default with a specific protocol.

For example, the default port for the HTTP protocol is 80 and the default port for the HTTPS protocol is 443, so a HTTP server waits for requests on those ports. Each Internet protocol is associated with a default port: SMTP (25), POP (110), IMAP (143), IRC (194), and so on.

We can change the port while launching our react app using -> in package.json file

"scripts": {

"test": "jest",

"build:tsc": "tsc -p tsconfig.json && tsc -p tsconfig-cjs.json",

"build": "lerna bootstrap && rimraf ./dist && webpack",

"start": "webpack-cli serve --https --mode=development --env development --open --hot --port 4200",

"lint": "eslint './src/\*\*/\*.{ts,tsx}'",

"lint:fix": "eslint './src/\*\*/\*.{ts,tsx}' --fix"

},

Or

Create a .env file at your project root and specify port number there. Like:

PORT = 3005

Yaml – yet another markup language

Used for config files, Ansible playbooks, kubernetes deployments