DOCKER | KUBERNETES

DOCKER Part - 1

1. **Problem Statement -** let’s say your device has specific version of nodejs, Postgres, sql, java idk, sql etc and let’s say running in macOS, so now in project, when a new person comes he also needs to set these all config in his system manually install this or that but no one remembers and also some softwares are not supported in macOS, so replicate the environment gets tough in diff systems. So this is the problem docker solves by reducing extra effort, it has concept of containers and in these containers we do all our configurations which tool ,which OS etc.
2. Advantage is let’s say you are in linux, window etc you can run this container anywhere without any hassle.
3. These containers are very lightweight - can be easily build, deployed, destroyed , share etc.
4.
5. **Installation -** download docker desktop from chrome, it will have both Docker CLI and Docker Desktop GUI.
6. Install it, and open it and then you will see in top bar docker will start running and then in app desktop docker will run.
7. Open it, and you will see no container, no image , no volume and that’s good, now we’ll learn how to create and use all them.
8.
9. **Working -** let’s try to run random OS let’s say ubuntu into our macOS from terminal.
10. First check if docker is running or not, check version type: docker or docker -v.
11. Then run ubuntu container using : docker run -it ubuntu
12. Here, -it means interactive and ubuntu here is our image name or for now take it as OS name.
13. Initially on running, it will say no image ubuntu so don’t worry , it will self download it, these images are also light weight.
14. Now in GUI, refresh and see a container is there of ubuntu.
15. In guy, it gives a default name to container on its own.
16. -it means : do not detach from docker so on running we come inside docker.
17. Now after run, we are inside container and do ls, search for node command it will show not found bcos we are isinde container. Containers are isolated they cannot access our machine storage or config unless we set them.
18. Now see it downloaded ubuntu image so how? See it firsts look inside

our container images if image present or not, if not, then it goes to hub.docker.com and download the image from there itself.

19. You can do anything inside container, it's doesn't affect our outside local machine.
20. Okay so IMAGES are like OS and they run on CONTAINERS, same like DarwinOS runs on MacBook Air M4.
21. Two containers are isolated they cannot communicate together, until we want them.
22. In real world, we make our custom image inside it we tell our project config and then our team uses that image and run same code.
23.
24. Use : docker container ls - to see all running docker containers.
25. Use : docker container ls -a - to see all containers running or stopped all.
26. Use : docker start <containername> - use case let's say you want to start a container from CII then use this to start container.
27. Use : docker stop <containername> - to stop a running container.
28. Use : docker exec <containername> ls - to comes inside docker show all things and then return back outside docker, detaches from docker.
29. Use : docket exec -it <containername> bash - -it will not detach container, will keep us inside docker.
30. So we learn 2 commands : docker run -it <image> - it everytime create a new container | docker exec -it <containername> bash - it opens container .
31. We can Install many images from hub.docker like nodes or whatever and then work inside container.
32.
33.
34. **How Big Companies Use -** they give you an image and just tell to run like: docker run <theirimagename>.
35.
36. **Port Mapping -** how to mapping of ports bw container and host.
37. Let's say in container server means your application of nodejs inside container is running on port 8000 and since containers are isolated so you will need to do port mapping first to run the application in your local system, either map it to same port or diff port.
38. So we need to expose the ports using: docker run -it -p 8000:8000 <imagename>
39. Here 8000:8000 -> right side 8000 is port running inside container | left side 8000 is port with which we are mapping inside port, so means inside 8000 map it to 8000 of localhost, we can give any like 1025:8000 and then it will run in our local system also.
40. We can also expose multiple ports, like docker run -it -p 1000:8025 -p 8000:1025 mailbag/mailhog
41. We can also pass env var in command like: docker run -it -e key=value

    -p 1000:1000 <imagename>
42.
43. **Making a small Node + Express App to learn Dockerisation :**
44. so creating a folder and then in terminal vs code, doing- npm init (initializes node and creates package.json file) then since it's just for test purpose installing express using: npm i express .
45. Creating a file main.js and writing small code of express in it.
46. For **Dockerize** - make a file with exact same name always as "Dockerfile" - it has no extension and this file is like an image file,
47. So we have main.js, node.js , package.json, package-lock.json : now we need to make image of these all files a single image so that other developers can use this image into their containers.
48. That's why we make Dockerfile , in it we define: all configurations, choosing base image (ubuntu etc)
49. Defining all things in Dockerfile like FROM , RUN , COPY (source dest) , ENTRYPOINT("which file should run default").
50. Now our complete configuration is done, converting it into an image from vs CLI using : docker build -t <imagenamecanbeany> .
51. Here -t means -> tag of that image. | Here . Means -> that my file Dockerfile exits in this folder only (.) shows path of our Dockerfile.
52. So it will take time since it's going to pull ubuntu then nodejs and copy files so once our image bets build it wil be in our local system as our custom made image.
53. And not everytime it's going to take time, it's only for once.
54. ONCE OUR IMAGE IS BUILD,
55. Go and check in GUI, we have our own custom image as docker-practice (note: it's in our local, not in browser till now).
56. Now check it, do run and port mapping using: docker run -it -p 8000:8000 docker-practice.
57. In browser, you will see our code response written in express will be shown.
58. Also, as we can manually set environment variable, since in our code for port we have given or condition one is default or one is when we pass manually. So lets do manual from terminal using: docker -it -e PORT=4000 -p 4000:4000 docker-practice
59.
60. **Layer Caching -** in DockerFile code, let's say we have already build one time , when we do another time build, it will complete very fast bcos it caches means It checks if anything got changed in code, let's say if anything gets at line 7 then it will go from like 1–6 very fast as cached but from line 7 onwards since there was a change it will take time and do again, okay so in industry we take care of lines a lot means we don't want nodejs to get install every time so we keep these type of things earlier in line so that in below line if something gets change, then these big things donot need to go again n again.

Publishing Image to Hub(Internet)

1. Make a free account on hub.docker.com
2. After account creation , create a repo in it and give name of image whatever then create, then it will tell by what name first we need to create image, so create an image what that name from vscode cli using: docker build -t ujjwalbansall/docker-practice .
3. When image gets build, then we'll push our image to hub.docker.com to our repo.
4. Now to push, open Mac terminal, in it push using: docker push <imagename> (ujjwalbansall/docker-practice our image name).
5. High change, it can show no access means we haven't login so first do login using username n pass then push it by command.
6. Post login success, it will successfully push and then you can see in hub.docker.com.

Docker Compose

1. Using this we can use, create, setup, destroy multiple containers means in real world, we have separate containers lets say one for reds, one for Postgres and one for node js, so using docker compose we can use n setup all of them multiple containers.
2. To use it : we first know what all things is being required by our code, for now we have node + express and it needs nodejs, Postgres, redis.
3. We make a file in our directory where we generally keep Dockerfile config file for image, we create a file as "docker-compose.yml"
4. What this file do ? Here we give config of multiple containers what all we need.
5. What we keep in this file ? First : version (docker compose version)
6. Second: services , here we define image , ports, environment of services we need to use like Postgres , redis etc .
7. Now to compose, use: docker compose up
8. It composes our docker, post successful run, open docker GUI, there you will see a stack running in container with name same of our project name and opening it you will see it has multiple containers each pointing to each service lets say Postgres, redis etc.
9. use: docker compose down, it removes and stops the complete stack.
10. Now to use in detach mode (means in background it will keep on running) use : docker compose up -d.

Docker Networking

1. When we run a docker and inside it, when we type ping google.com , we can see it can connect with outer world internet, how bcos of networking.

2. We have 3 network drivers available : bridge(default when we don't specify default is this) | host | none
3. Now to prove, run one container then use: docker network inspect bridge , it will which all containers are running on this network since we haven't done manual network change so by default you will see all containers running in this.
4. So working is what happens: let's say your machine and some containers so dockers does is it makes a bridge connection by default bw your machine and containers and all containers are connected to this bridge network bcos of this container gets IP Adresses and communicate with internet.
5. To check all networks locally in system: docker network ls
6. We can also make our custom networks that is mostly demanded in companies as it allows us to lets 2 containers communicate with each other.
7. To manually change network driver, use: docker run -it —name my_container2 —network=host busy box
8. Diff bw bridge and host? : see. When we explicitly mark as host , then it means it doesn't get any bridge and it's connected to my local machine only. Means the main diff bw these 2 is now see if we have bridge network, then we need to do port mapping (-p 1234:1234) but if we have host then we don't need to do port mapping bcos it's connected to our local only since it's not connected in outerworld bcos of bridge so in host no port mapping required as our host machine and docker container is on same network in host network but not in bridge network.
9. Last one is none network, if we do it, then it means it does not have access to any network so it will not be able to ping to google also but no network access.
10. Make your own CUSTOM Networks : use: docker network create -d bridge <networkname>
11. Here -d is type of driver we want can be bridge, host or none , mostly cases bridge we use so bridge and then network name whatever we want to keep.
12. **Now see, benefits of custom networks? -** we can run two containers with diff name diff OS (image) but both running on same network (—network=customnet)
13. So if in one container inside it lets say running on busy box image, inside it try : ping <othercontainername> now you will see it will be able to communicate with it means two containers can communicate with each other if both are on same network, if we stop the container then it will not be able to ping.

Volume Mounting

1. VERY IMP POINT : Each container has its own memory, so when we destroy a container, its memory also gets destroyed.
2. To precent this, we have DOCKER VOLUMES :
3. So see, : for trial make a folder in your desktop as test-folder , open terminal - cd desktop , ls, cd test-folder , pwd (path of folder)
4. Now copy this path and make a container use: docker run -it -v <pwd folderpath>:/home/test busybox
5. Here -v means volume mapping and after this we give path of folder where we want to store all container data and then we mount it to container inside some folder so let's say inside container inside home we made folder test and mounting it there then image name run.
6. So inside docker after run, do ls and check if our desired folder is there or not test folder open it and make one folder from inside docker using command: mkdir folder, now once you made it you will notice that in our local also the folder got made, so it means whatever we make now inside our container bcos of volume mapping it will also get a copy made in our local so it prevents the memory loss even if container gets destroyed.
7. Now no matter if you delete container, files will be there safe and let's say in future you make another container then just do volume mapping of that and you will find same files inside that container also SIMPLE.
8. We can also create our own volumes and directly mount them too. If want, just for command in google.

Efficient Caching in Layers

1. It's better if we layer our code in good way so that our price and token reduces like keeping lines above which don't need to install again n again.
2. Instead of doing copy filename filename , we can also do: copy . .
3. It copies all files but it will also copy nodemodules so to present it we make a file: .dockerignore (just like .gitignore) and in this we write node modules so it will not copy all those things written inside this.
4. And we can set workdir /app also (path where our files will be get saved).
5. That's how we do efficient way of caching layers.

Docker Multi Stage Builds

1. we can have multiple builds in one Dockerfile, means we have multiple FROM statement, one can be for installing and one for building and one separate for running, so it takes less space.

Docker Image Optimisation

1. When we build our image and write Dockerfile code, as told we make efficient caching in layers and second main thing, in real world, our image should be small in size. There's no point if our image deployed on server is of size 2GB, if it is then think how much it will take to get ready in someone else's system that's why it's very important to keep docker image size small.
2. For this, best practise we do, let's say im doing FROM ubuntu, then it will pull everything from ubuntu which is as per our project not even required, so best way is to use : slim | alpine, it downloads only the required things from image, so reduces size, use like this: FROM node:<alpine version take from google>
3. Second we take care of efficient caching of layers, keep static command lines always at top like : npm i and etc, keep changing lines at bottom like main.js etc.
4. Third we take care of multi stage builds, for example- after build we don't need .ts files, as in build, inside dist folder it automatically converts our .ts to .js so no need. What we do multi stage builds, first FROM as builder we keep all files, then second FROM as runner in it, use: copy --from filepath ourpath. So keep in mind these 3 steps always.

KUBERENETES

1. Made by Google. It’s taken as inspiration from google in-house container orchestration tool : “Borg”.
2. Used for Container Orchestration (means used to manage multiple containers)
3. It takes care of our desired container count and our current container count.
4. It’s architecture involves : Control Plane , Worker Node , Kublet etc.