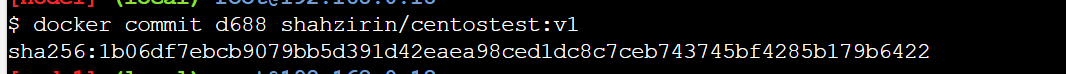


Docker image build –t shahzirin/gsd:first-ctr . ->-t tag where Shahzirin is the docker id, gsd is the repository name and first-ctr is the name of the image/version string. This will create an container image for u. the . to select the current directory

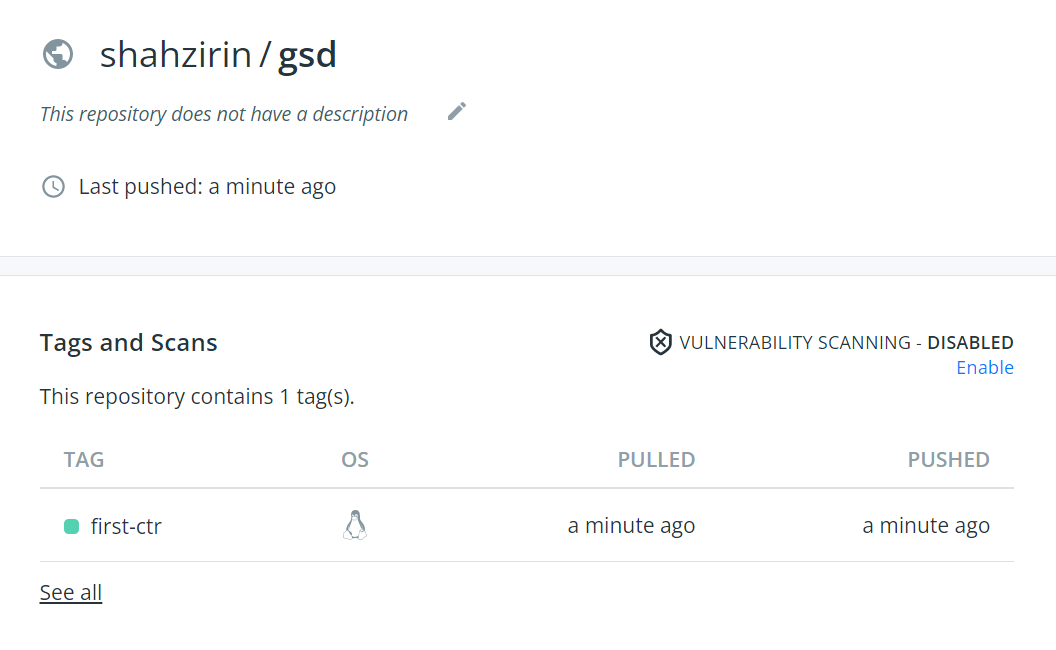
\*Or\*

docker commit containerID/name Shahzirin/gsd:first-ctr -> \*for running container, use this to save the image state

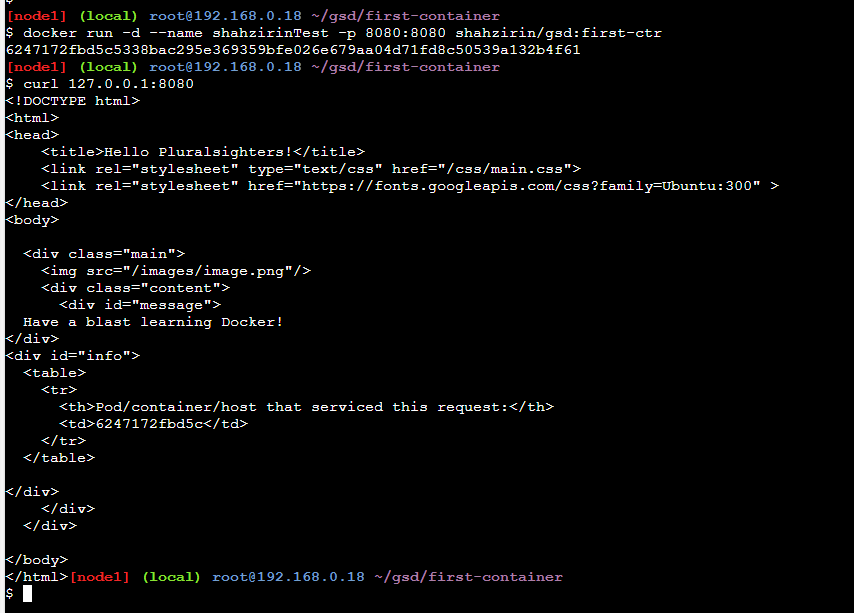


Docker login -> login before pushing image to your docker repository

Docker image push shahziriin/gsd:first-ctr -> push to my repository after loggin in



Run detached name of app shahzirinTest on port 8080 on image Shahzirin/gsd:first-ctr



Docker stop contanerID/name -> stop the container

Docker start contanerID/name -> to start the container again , -d is deatache, -it interactive terminal

Docker rm contanerID/name -> to remove the container

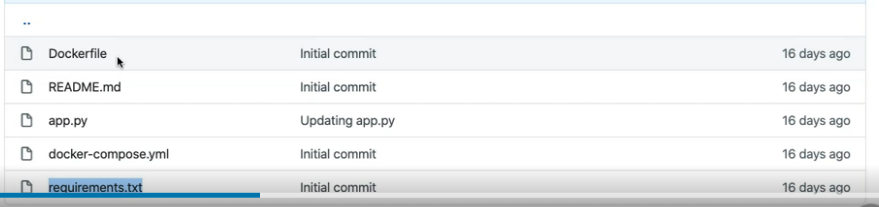
Docker rmi image -> to remove the image

Docker run -it --name test1 alpine sh -> will run interactive image of alipine and access terminal via sh

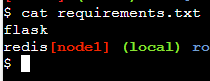
Docker exec -it containerId/name sh -> will go into terminal of the container , I flag for interactive/keep STDIN open even if not attached and -t to allocate TTY

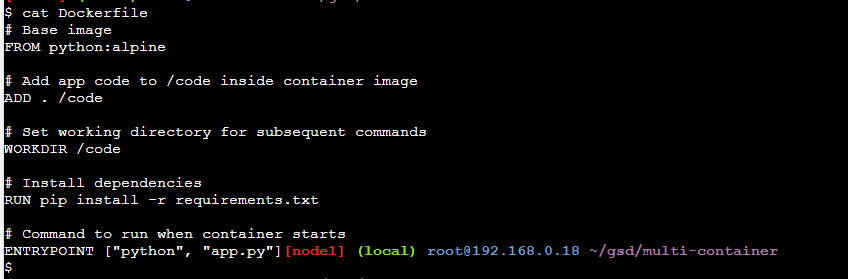
Docker exec web1 pwd -> print current directory of the container

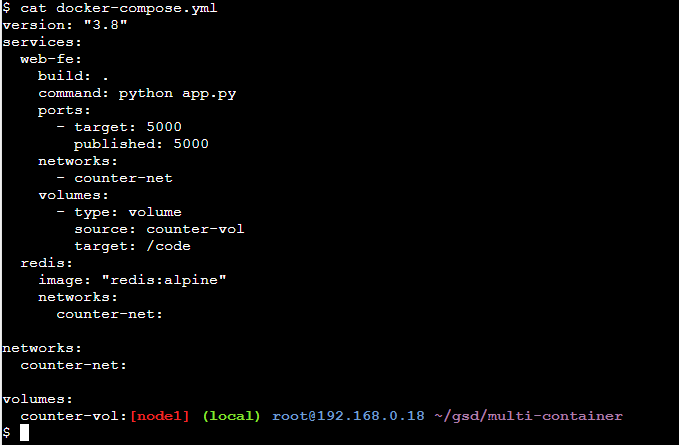
**Docker-compose**



Where the app.py is the app file. Requirements.txt is the dependencies/requirements. And instruction to build app container in Dockerfile

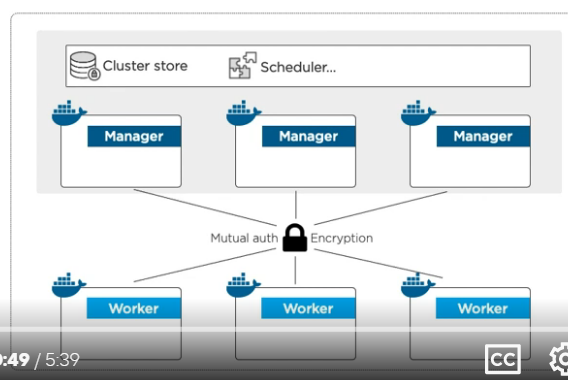
flask and redis required





Your dockerfile consist of following:

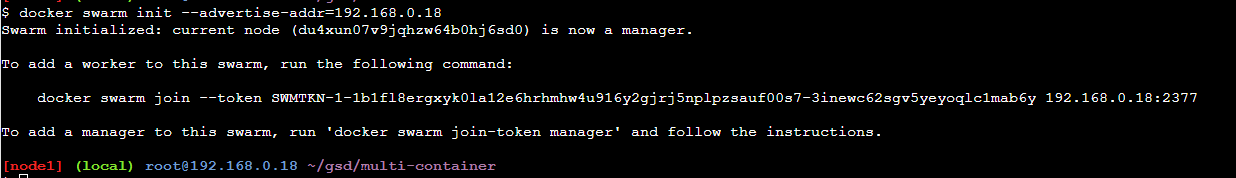
**Docker-swarm**



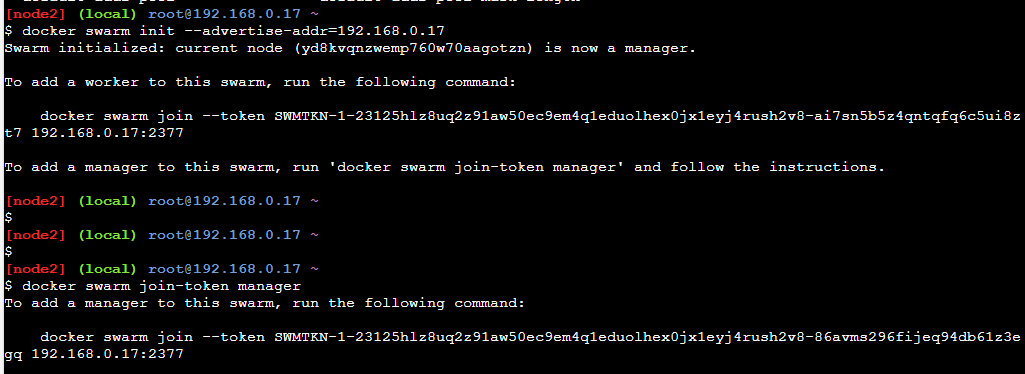
odd number of manager should be used to avoid freeze

docker swarm init --advertisde-addr=192.168.0.20 -> to initialize docker swarm where the host ip is 192.168.0.20

to join as worker, run as the following command with the token provided

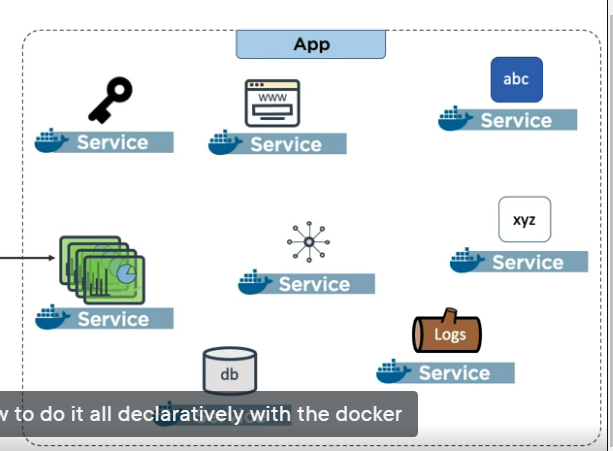


Run docker swarm join-token manager to join as manager -> use the token to run on other nodes to join as manager



Docker node ls -> to view the managers and workers





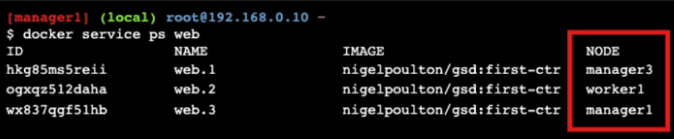
How an app runs with docker

This will run the service with 3 replicas. Will only work if you have docker swarm



Docker service create --name web -p 8080:8080 --replicas 3 shahzirin/flaskapp:v1

This will show all the nodes that are running your app/replicas



Docker service scale web=10 -> this will increase your replicas to 10

If you force to stop/rm your containers, docker-swarm will help create new containers to ensure baseline 10 containers

Replicas can be set in you yaml file

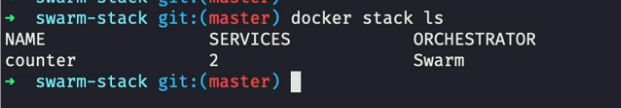
This is a stack as will be running on docker-swarm

\*cannot build for docker swarm. Images must be ready for use. Push whaterver image you build to your repository so other nodes can access

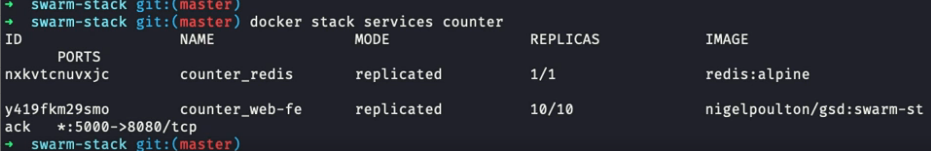


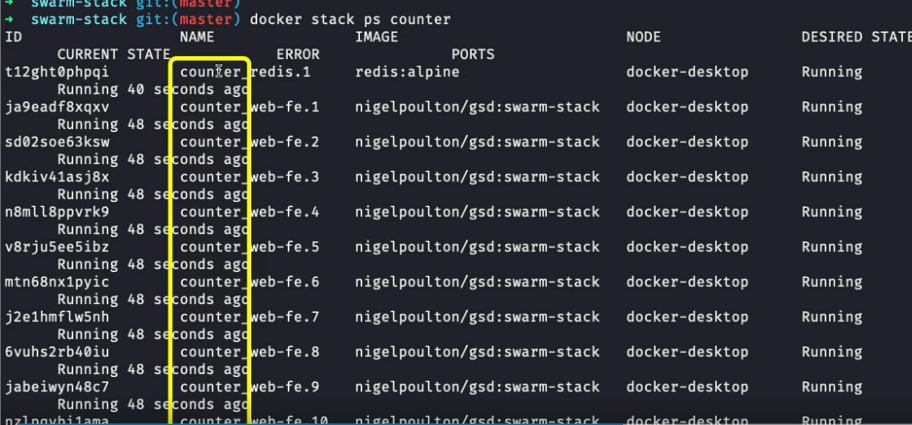
Docker stack deploy -c docker-compose.yml counter -> where counter is the app and –c to tell that we are running a compose file

Docker stack ls -> to view the sercd ..vices



Docker stack services counter -> to view services in your counter app



Docker stack ps counter -> to view the containers running for the app 

**Docker stack rm countr -> to stop the stack**

**Docker service vs docker stack**

The docker service is used when managing individual service on a docker swarm cluster. It is the client command line to access the docker swarm manager.

The docker stack can be used to manage a multi-service application. It also moves many of the options you would enter on the docker service into the .yml file (such as docker-cloud.yml or docker-compose.yml) for easier reuse. It works as a front end "script" on top of the docker swarm manager used by docker swarm cluster, so you can do everything docker stack does with docker service.

docker run hello-world -> to run hello-world image, if not available, will go online to pull the image

docker run busybox echo hi there -> run busybox image, and echo hi there

docker run busybox ls - > list all file, will see all the basic linux folders/files which exist in that container such as bin,dev,etc,root,home etc

docker ps -> to see running containers

docker ps --all -> list all containers including those that ended, you can consider deleting to get space

docker run busybox ping google.com -> will run nonstop till we stop

docker run = docker create + docker start

docker create imageName-> create image

docker start -a containerId-> start image with output(-a flag), u can also restart by running the same containerId after running

docker start containerId-> start image without output, u can also restart by running the same containerId after running

docker logs containerId -> will see logs such as echo statement that had run

docker system prune -> remove all stopped containers and will delete dangling images, cache etc

docker stop containerId -> will send sigterm to containerId, will ask nicely, if after 10 sec doesn stop, will send sigkill

docker kill containerId -> immediate send sigkill, without sending sigterm

\*multi commands

docker run redis - > to start redis image

docker exec -it containerId command -> example below

docker exec -it 535457487 redis-cli -> this will run redis-cli (similar to mongoshell)

\*-i is for input, else immediate will end as cannot input anything to redis-cli

\*-l is to list nicely, not complusory

docker execute -it 535457487 sh -> sh is like shell, immediate go into terminal in the container

docker attach contarinerID -> this will attached to the cli of the container. The difference with exec is that exec creates another command for you so when you exit, the container does not exit the container. Attach is like attaching your cli to the containers cli

\*alternatibvely start immediately sh

docker run -it busybox sh

**Creating dockerfile and building an image from it**

Mkdir redis-image

Cd redis-image

Vim dockerfile

-------------------------------

# use an existing docker image as a base

FROM alpine

#download and install a dependency – execute some command ,

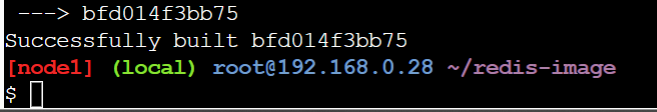
RUN apk add --update redis

#tell image what to do when start as a container – what command should run when container started

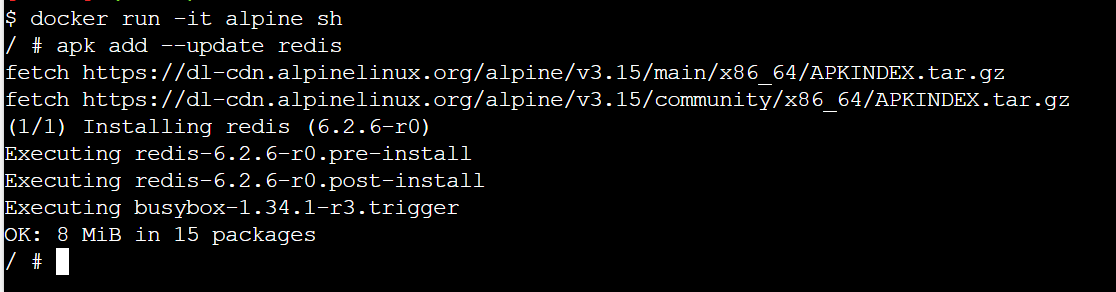
CMD [“redis-server”]

------------------------------------

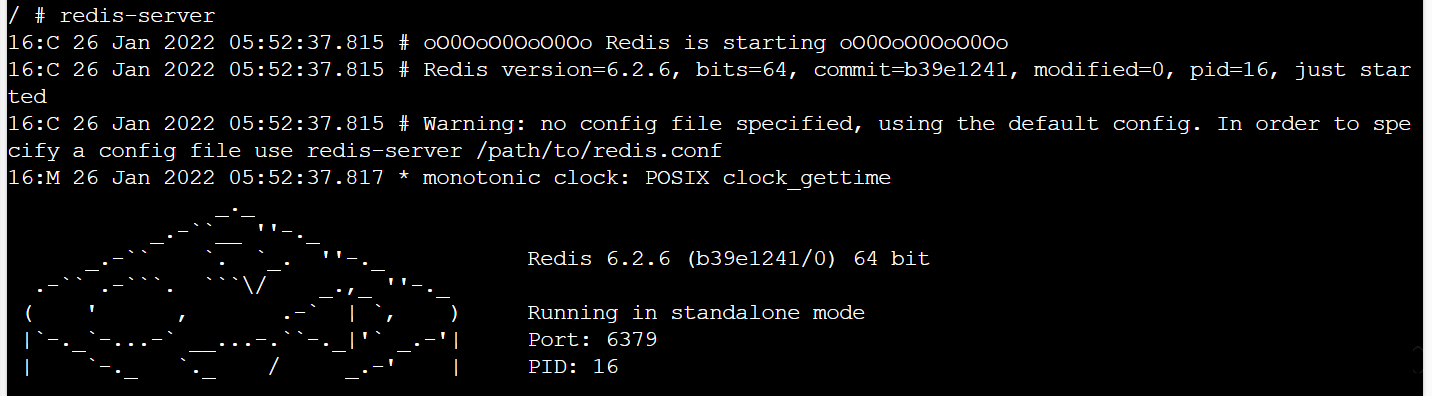
Docker build . -> this will build image and you can capture the image id at the end



**Breakdown what happens during build**

This is what happen during the run after image is build 

and this is what happen during the cmd after run



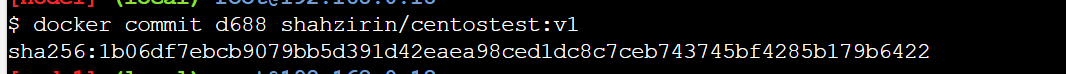
\* every step will snapshot and save a temporary image for next instruction and once all steps are run, final image is saved with an id

When rebuilding images, docker will automatically detect if there is changes and make use of cache for previous commands and rebuild images and update the new steps from the first change found from the top

Docker build –t shahzirin/redis:latest . -> You can tag the images build so you can reference easier rather then using the id. Shahzirin is the docker id, redis is the repo/project name and the last point Is the version

**Using a container to create image \* alternative**

Docker ps - > to get the running container id



U can also aso add –c to add scripts. For example if u want to run a command redis-server

Docker commit –c ‘CMD [“redis-server”]’ d688 shahzirin/centostest:v1

**Things to take note of base image**

Limited programs in the images, default linux programs. You can either find base images with the requirement or you install the required packages with the image. Alpine is a popular image as they install minimum packages . eg node:alpine for a strip down image that has node js package.

**Copying files from from current directory to the image when building**

#base image

FROM node:alpine

# install some dependencies

#this will copy the current directory building the image to the image at /usr/app

COPY ./ ./

RUN npm install

#default command

CMD [“npm”, “start”]

**WORKDIR command -> the folder specified, when any command is executed after, it will be in the path specified**

#base image

FROM node:alpine

#will cd into /usr/app

WORKDIR /usr/app

# install some dependencies

#this will copy the current directory building the image to the image at /usr/app

COPY ./ ./

RUN npm install

#default command

CMD [“npm”, “start”]

\* if you run ‘docker exec-it contatinerId sh’ -> will be in the /usr/app directory as well

**Mapping port in docker**

Docker run –p 5000:8080 shahzirin/simpleweb -> running the image and opening port 5000 from outside to container 8080

**Minizing cache and rebuilds**

Ensure your steps are in order. For example, if you know you will keep making change to the file index.js and rarely on package.json (package file that needs to be defined before installing your npm) , you can instead list to copy package.json file before installing and copying the rest which include index.js after the installation. This will ensure when you make changes to your index.js file, the image wont reinstall the npm as the command to copy the index.js file is after the installation. Example shown below

#base image

FROM node:alpine

# install some dependencies

COPY ./package.json ./

RUN npm install

COPY ./ ./

#default command

CMD [“npm”, “start”]

**Docker compose**

**purpose**

* **getting communication between containers -> using yml file**
* **combine cli commands, automate some of long-winded args**

**Node app + redis database example, assume there is a index.js file that connects to the redis host**

Create 2 services using docker-compose version 3 wth image redis and build another image for node-app from current directory and map the port 4001 to the container 8081 below

-----------------------------------------

Vim docker-compose.yml

Version: ’3’

Services:

Redis-server:

Image: ‘redis’

node-app:

build: .

ports:

- “4001:8081”

-----------------------------------------

Vim dockerfile

FROM node:alpine

WORKDIR ‘/app’

COPY package.json .

RUN npm install

COPY . .

CMD [“npm”,”start”]

Docker-compose up -> to run the containers using docker-compose

Docker-compose up --build -> to rebuild before running the containers

Docker-compose up –d -> to run in background

Docker-compose down -> stop all the containers

**Docker restart policies**

no -> never attempt to restart, default

always -> if container stops whatever reason, restart

on-failure -> only restart if container stop with error code

unless-stopped -> always restart unless we forcibly stop

Version: ’3’

Services:

Redis-server:

Image: ‘redis’

node-app:

restart: ‘always’

build: .

ports:

- “4001:8081”

Docker-compose ps -> to see the status of the status but must be in the same directory of the docker-compose.yml file

**Running docker for development and production -> workflow understanding, different dockerfile used**

Vim dockerfile.dev

FROM node:alpine

WORKDIR ‘/app’

COPY package.json .

RUN npm install

COPY . .

CMD [“npm”,”run”,”start”]

Docker build –f dockerfile.dev -> to call specific dockerfile

**Mapping volume -> this will not require to keep copying the file over, instead referencing**

Vim dockerfile.dev

FROM node:alpine

WORKDIR ‘/app’

COPY package.json .

RUN npm install

COPY . .

CMD [“npm”,”run”,”start”]

Docker run –v localDirecotry:containerDirectory image ->

Docker run –v /app/node\_modeles –v /home/app:/app image -> placeholder,bookmark the folder in the container so it won’t be touched. This is usefule when u want to mark a child directory and not touch while manipulating the parent directory

**Mapping volume via docker compose**

Version: ’3’

Services:

web:

build:

context: .

dockerfile : Dockerfile.dev

ports:

- “4001:8081”

Volumes:

-/app/node\_modules

-.:/app

Build is to specify the folder to build the image , . To specify the current folder and Dockerfile.dev is the file that was created previously \* if not specified, when building dokcer will look for dockerfile in the folder specified in context

#the first one is to bookmark /app/node\_modeules and the second one is the map the current directory to /app

**Add test service on 2nd container**

Version: ’3’

Services:

web:

build:

context: .

dockerfile : Dockerfile.dev

ports:

- “4001:8081”

Volumes:

-/app/node\_modules

-.:/app

tests:

build:

context: .

dockerfile : Dockerfile.dev

Volumes:

-/app/node\_modules

-.:/app

Command: [“npm”,”run”,”test”]

This will be good as instead of running another container from the 1 runner image, docker-compose will run 2 containers straight away. The first container to run the app and 2nd container to test the app

\*downside is both stdout will show so you cannot run any commands on the specific application.

run docker ps to get the container id. Then run docker exec –it containerID sh to access the container shell but u still cannot give stdin for the apps such as the test app

**multi steps build phases**

to copy over stuff from phases that is created. All config from the 1ststep will be removed, only the folder will be kept for use in the 2nd step

vim dockerfile

#as builder is to tag the image as builder, using version 16 as there is a bug of the latest

FROM node:16-alpine as builder

WORKDIR ‘/app’

COPY package.json .

RUN npm install

COPY . .

RUN npm run build

FROM nginx

#COPY FRom the builder phase

COPY --from=builder /app/build /usr/share/nginx/html

**Exposing ports through dockerfile, it is only reference for developers but for deployment such as aws beanstalk will automatically detect and expose the port**

vim dockerfile

#as builder is to tag the image as builder, using version 16 as there is a bug of the latest

FROM node:16-alpine as builder

WORKDIR ‘/app’

COPY package.json .

RUN npm install

COPY . .

RUN npm run build

FROM nginx

EXPOSE 80

COPY --from=builder /app/build /usr/share/nginx/html

**Builidng a multi-container application – fibenaci calculator**

Nginx ->-> react server,express server

React server -> express server

Express server ->-> redis,postgres

Redis <-> worker

Use docker-compose

**Environment variables**

2 ways to Set variable in container at runtime

variableName=value

or

variableName

#the 2nd option will take the value from the machine running the docker-compose file

In docker compose, we put it inside environment

Vim docker-compse.yml

version: '3'

services:

postgre:

image: 'postgres:latest'

environment:

- POSTGRES\_PASSWORD=postgres\_password3

redis:

image: 'redis:latest'

api:

build:

dockerfile: Dockerfile.dev

context: ./server

volumes:

- /app/node\_modules #dont overwrtie anything from here

- ./server:/app #all folder except the node\_modules will use /server folder

environment:

- REDIS\_HOST=redis

- REDIS\_PORT=6379

- PGUSER=postgres

- PGHOST=postgres

- PGDATABASE=postgres

- PGPASSWORD=postgres\_password

- PGPORT=5432

client:

stdin\_open: true

build:

dockerfile: Dockerfile.dev

context: ./client

volumes:

- /app/node\_modules

- ./client:/app

worker:

build:

dockerfile: Dockerfile.dev

context: ./worker

volumes:

- /app/node\_modeules

- ./worker:/app

environment:

- REDIS\_HOST=redis

- REDIS\_PORT=6379

**Creating nginx config for routing**

**Setup following**

* Listen at port 80
* Any / path will go to client upstream
* Any /api path will go to server upstream
* Upstream client:3000
* Upstream server:5000

\* rewrite /api/(.\*) /$1 break;-> this will remove the /api before routing to the api server . $1 will return the url with /api removed. Break is to stop after removing the /api first time

\* sockjs-node is for websocket. For faster/better communication. Not compulsory

Mkdir nginx

Cd nginx

Vim default.conf

Upstream client {

Server client: 3000;

}

Upstream api {

Server api: 5000;

}

server {

listen 80;

location / {

proxy\_pass <http://client>;

}

location /sockjs-node {

proxy\_pass <http://client>;

proxy\_http\_version 1.1;

proxy\_set\_header Upgrade $http\_upgrade;

proxy\_set\_header Connection “Upgrade”;

}

location /api {

rewrite /api/(.\*) /$1 break;

proxy\_pass http://api;

}

}

**Now with the nginx, we can create the dockerfile for the nginx**

Vim Dockerfile.dev

FROM nginx

COPY ./default.conf /etc/nginx/conf.d/default.conf

Add the nginx image to docker-compose, restart : always is to always restart if crash. Ports

is to enable port 3050 externally to the container port 80 and depends\_on is to wait until the other services stated has started

version: '3'

services:

postgre:

image: 'postgres:latest'

environment:

- POSTGRES\_PASSWORD=postgres\_password3

redis:

image: 'redis:latest'

nginx:

depends\_on:

- api

- client

restart: always

build:

dockerfile: Dockerfile.dev

context: ./nginx

ports:

- '3050:80'

server:

build:

dockerfile: Dockerfile.dev

context: ./server

volumes:

- /app/node\_modules #dont overwrtie anything from here

- ./server:/app #all folder except the node\_modules will use /server folder

environment:

- REDIS\_HOST=redis

- REDIS\_PORT=6379

- PGUSER=postgres

- PGHOST=postgres

- PGDATABASE=postgres

- PGPASSWORD=postgres\_password

- PGPORT=5432

client:

stdin\_open: true

build:

dockerfile: Dockerfile.dev

context: ./client

volumes:

- /app/node\_modules

- ./client:/app

worker:

build:

dockerfile: Dockerfile.dev

context: ./worker

volumes:

- /app/node\_modeules

- ./worker:/app

environment:

- REDIS\_HOST=redis

- REDIS\_PORT=6379

**\*adding another nginx config on the client server,**

server {

listen 3000;

location / {

root /usr/share/nginx/html;

index index.html index.htm;

}

}

vim dockerfile

#as builder is to tag the image as builder, using version 16 as there is a bug of the latest

FROM node:16-alpine as builder

WORKDIR ‘/app’

COPY ./package.json .

RUN npm install

COPY . .

RUN npm run build

FROM nginx

EXPOSE 3000

#copy the default.conf you created

COPY ./nginx/default.conf /etc/nginx/conf.d/default.conf

#copy the folder /app/build from the builder

COPY --from=builder /app/build /usr/share/nginx/html

**Below command to login docker using variable environment without requiring input for jenkins**

Echo “$DOCKER\_PASSWORD” | docker login –I “$DOCKER\_ID” --password-stdin

\*assuming that both env variables have been set