Interaction b/n 2 contianers

2 containers can communicate with each other

1. **Docker inspectable objects**

Image, container both are docker inspectable objects

Network is also a docker object—which is inspectable

docker inspect <container-name/id>

there u can see the



From Ubuntu machine we can access nginx container from Ubuntu container,

if u start that nginx container once, untill u kill it will be running ,because it is a server

First u have to go inside Ubuntu

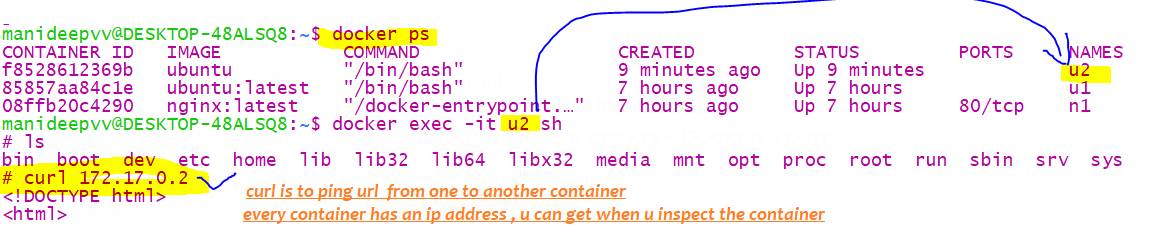
“docker run –it –-name u1 ubuntu”

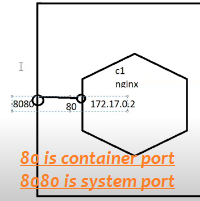
When u run in interactive mode , u will get the console

When u expose ports then only u can access, ,for that u should inspect the image

“docker inspect <image-name>”,

There is a field called “exposed ports”





2)connect using LINK

Docker is not recommending to use LINK due to following reasons

1. Using link ,by default one 1 directional connection can be established, whereas if we use Network object

Its bidirectional, as

All docker objects(images/containers) by default connected to bridge network

Generally one container can ping another container, using ip address & u can get the ip address using

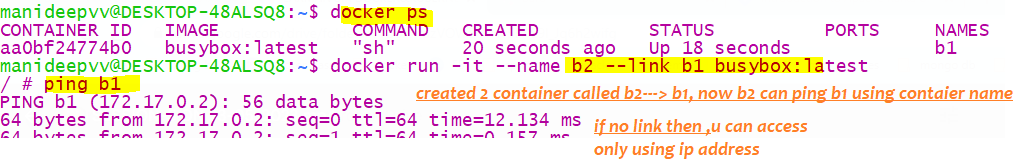
“docker inspect <container-name>”🡪to get the ip address of running container

Whereas if u link , then u can ping them using container name

While creating the container itself, u have to create with link

docker run –-name u1 –-link b1 <image:tag>

|  |  |
| --- | --- |
| Without link , u can communicate wit Ip address “ping <in address of target con>” | Ping 172.17.0.1 |
| With link , u can communicate with container name | Ping <target container name>  No need of container ip address |



If u follow link approach,& if u want to connect from wordpress to mysql

First u have to create mysql container, then while creating word press container u have to link with mysql container

3) Using Network objects

|  |  |
| --- | --- |
| Using LINK | Comm’n b/n containers is uni directional |
| Using Network objects | Communication b/n 2 containers is bi directional |

Docker supports networking between containers using Network object

1. Bridge
2. Host
3. None
4. Overlay
5. Macvlan

By default all objects are connected to bridge network & containers are pingable using their ip addresses

U can inspect the bridge network object

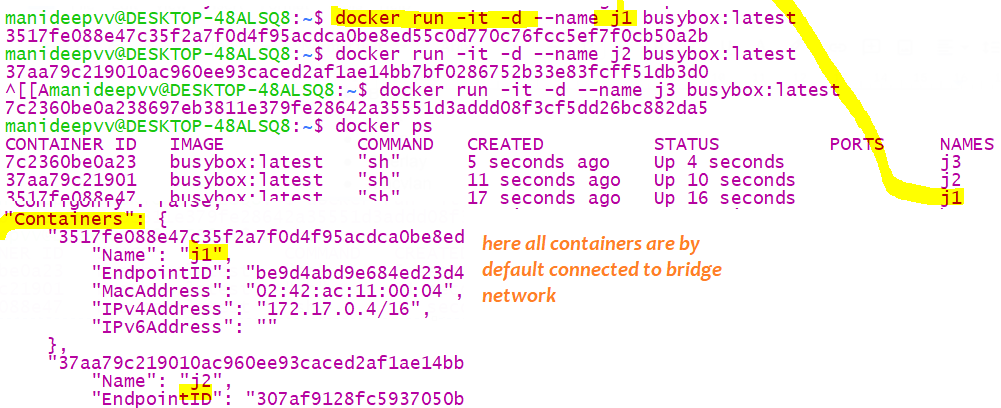
#### Bridge network

Bridge is a private network

Network

object

With this approach, communication is bi directional



As of now, 4 networks are available , by default all containers are connected to bridge network

If we create a custom network object, then they are pingable between by the container name

Creating custom network object

docker network create <your network name>

what is the use of connecting to a custom network?

Here communication is bi directional , whereas using LINK ,communication is only uni directional

Here we didn’t mention the parent network from which it should be created. Because by default its bridge network

CRUD network

|  |  |
| --- | --- |
| Creating a network | docker network create <my-network-name> |
| Creating with network type | docker network create –d <network type> <custom network name>  docker network create –d bridge new\_bridge |
| Connecting to a network | docker network connect <<network name>> <<Container name>>  docker network connect new\_bridge webcontainer |
| Disconnecting from a network | docker network disconnect bridge container1  docker network disconnect <<network name>> <<container name>> |
| List the network | docker network ls |
| Inspect | docker inspect <custom network name>  or  docker network inspect <<network-name>> |
| Deleting a network | docker network rm <network-name> |
| Ex | docker run –-name manicont –d –-net=new\_bridge <image-name>  // -d means detached mode |

BRIDGE

|

|

My-network ,

if 2 containers are connected using bridge, then both containers are pingable **only via ip address**

Whereas , if u create network as a child of bridge network/custom bridge n/w both containers

are pingable using container names

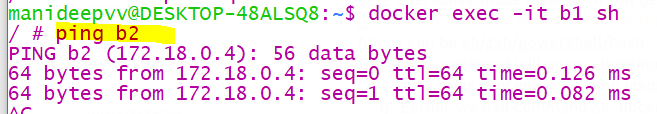
1. Once after creating network-then create your containers under your network

docker run --name b3 -it --network my-net busybox:latest

docker run --name b1 -it --network my-net busybox:latest

// here we created a container and nested under that my custom network

Once all containers are under your custom network, they are pingable using container name as below



Host network

Host is a public network, when ur container is connected to host network, no need to expose port explicitly

Means u are connected directly to computer network

Earlier If u want to expose , we used –p ,

no if u are connected to host network , u do need port mapping

It is a public network

It utilizes the host’s IP address and TCP port space to display the services running inside the container It effectively disables network isolation between the docker host and the docker containers.

Connection to host network

docker run –name <container-name> –network <network name> -it image

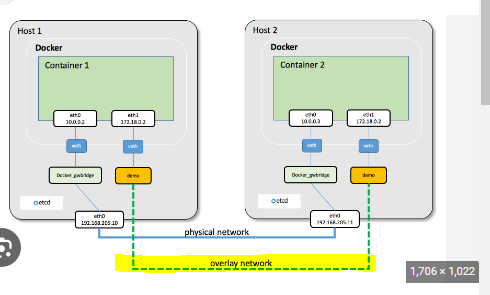
docker run --name n2 -it --network host nginx:latest

overlay network

overlay network is used to connect the containers present in different node machine(each node is a separate machine)

A ***bridge network*** is used when we run a relatively ***small network*** on a ***single host***.

An ***overlay network*** is used when we have a significantly ***larger network*** involving ***multiple host***.



None network

If u are on none network, u cant do port mapping also

If u are connected to none network , that is u are not at all pingable by any another container

docker run --name n5 -it --network **none** nginx:latest



|  |  |
| --- | --- |
| Bridge network | If 2 containers are connected to bridge n/w  They are pingable only via ip address,Not using container name |
| Custom n/w created from child network | Here , if 2 cont are connected to custom network which is a child of bridge network,then   1. They are pingable by ip addresses & 2. They can ping each other using container names |
| Host N/w | For these containers-we don’t need to do port mapping |
| None N/w | If a container is in none network, even 2 containers cant speak with each other, they cant communicate using ip address, & they cant communicate using container name  U cant do port mapping and u cant hit in chrome also |

Deleting a network

docker network rm <network-name>

docker network rm bridg\_sample

Disconnecting from a network

docker network disconnect bridge container1

Docker compose

Refer 53-54 Zips number folder for code

Separate CLI that gets installed along with Docker

Used to start up multiple Docker containers at the same time

Automates some of the long-winded arguments we were passing to 'docker run

And the main advantage is u don’t need to build+run , both it will do as part of “docker-compose up”

In docker-compose.yml file u will write all the below commands

docker-compose.yml

Here are the containers I want created:

redis-server

Make it using the 'redis' image

node-app

Make it using the Dockerfile in the current directory

Man port 8081 to 8081

//Below are the contents of docker-compose.yml file

version: '3'

services:

redis-server: //this says create a container called redis-server

image: 'redis' //using the image named redis

node-app: //this says create a container called node-app

build: . // this says create the above container using this docker file present in the

//current directory, where node needs 2 addnal files index.json..

ports:

- "4001:8081"

If u want to run the image and if u want to create a container from it below is the command

This docker-compose up will look for docker-compose.yml file

Up- means “up” all the containers

docker run <image name> 🡪

docker-compose up

docker compose= docker build+docker run

#### Rebuilding all containers in yml

If u have source code changes-then do rebuild, It builds the source code responsible to create 2 containers

docker-compose up --build

docker compose up –d

// if u want to start all containers at once , use this option with –d flag

Rebuild everything and start all those containers again

Rebuild- so that u can get all those latest changes

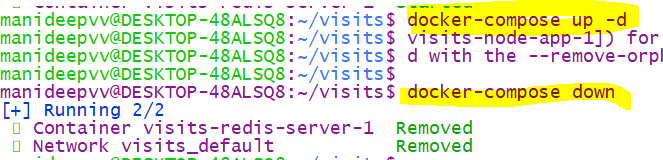
|  |
| --- |
| //build the image using docker file present in current directory  docker build . |
| docker run <image name> |

docker-compose up --build

#### Stopping all containers at once with docker compose

This docker-compose up –d will start 2-3 containers at once, if u want to stop all containers at once

Use docker-compose down



Restart policies

Restart Policies

|  |  |
| --- | --- |
| "no" | Never attempt to restart this. container if it stops or crashes  Mentioning ‘no’ in single quotes is |
| always | If this container stops "for any reason" always attempt to restart it |
| on-failure | Only restart if the container stops with an error code |
| unless-stopped | Always restart unless we (the developers) forcibly stop it |

#### Restarting containers automatically inside ym file

version: '3'

services:

redis-server:

image: 'redis'

node-app:

restart: on-failure

build: .

ports:

- '4001:8081'

In yml file no is interpreted as false,

So , if u want to provide restart policy as no, then provide in single or double quotes as

restart :’no’

to see all the running containers list type

docker-compose ps

we should have different docker files for each env

“dockerfile.dev”

Custom docker file name

So far , we worked without code, I mean we worked only with docker files, now we will start working with

code+docker files

for that refer project num “65-creating”,” 67-starting” these proj will be there as part of my repo

1. Download nodejs software and test installation status using command “node –v” in command prompt
2. Create a sample node js samle project with command

“npx create-react-app frontend”

1. The above command wont work in Ubuntu console, so create a project in windows folder and move

That project to Ubuntu [\\wsl.localhost\Ubuntu\home\manideepvv](file:///\\wsl.localhost\Ubuntu\home\manideepvv), type “\\wsl$”

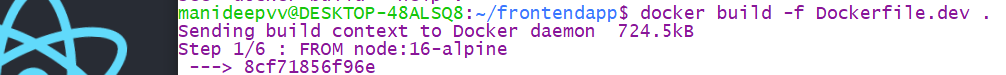
1. In that create Dockerfile and add node related files like “index.js”,”package.json”
2. Since we are creating prod ready proj, instead of creating file named“dockerfile” created “Dockerfile.Dev”

For dev environment

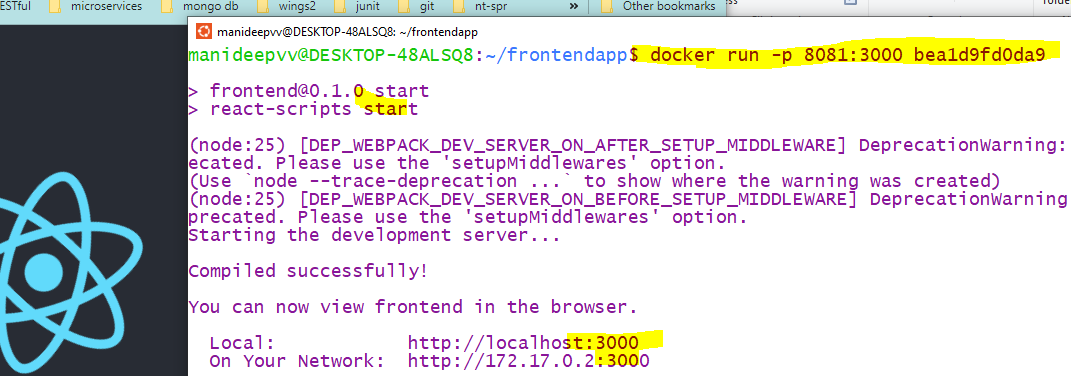
docker build –f <your custom docker file name> .

ex:- docker build –f Dockerfile.dev .

in above –f stands for custom file name and “.” refers to current directory



Run that container and map with system port



### Rebuild when code changes are made

manideepvv@DESKTOP-48ALSQ8:~/frontendapp$ docker build -f Dockerfile.dev .

Step 6/6 : CMD ["npm", "run", "start"]

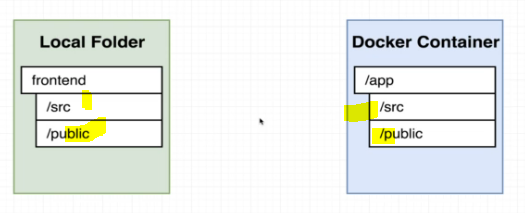
---> Running in 7a325cea22fc

Removing intermediate container 7a325cea22fc

---> 7ec45d18a115

Successfully built **7ec45d18a115**

manideepvv@DESKTOP-48ALSQ8:~/frontendapp$ docker run -p 8081:3000 **7ec45d18a115**

when we rebuild again separate space is created

to that space all those files should get copied

Dynamic code refresh

Refer proj 70,71,72(docker compose yml file proj-best)

In production we will never create a docker volume,bec we don’t want dynamic code refresh

In java we will build and we will get a jar file and we will deploy that jar file in jvm ,

Every time when we make a code change, we have to rebuild and deploy that latest jar in jvm/any server

Way 1:-

Here in docker world also, generally we have to do build every time and run that new continaer,

Way 2:-

Instead of building every time and getting the new image file we can go for hot code replacement

While running that container we cant point those requests to source code

So without rebuilding the image we can get the latest changes every time.

docker run -p 8080:3000 -v /home/node/app/node\_modules -v $(pwd):/home/node/app <image id/jar name>

here –p means port, first port is map with system port and 2nd one is docker port

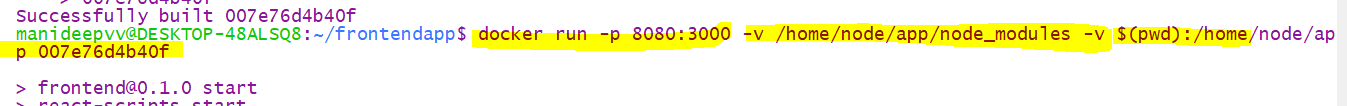
-v means =volume

/home/node/app is the default folder created inside a container

–v /home/node/app/node\_modules says don’t map this folder called node\_modules, this folder will be there in the container, if any request comes to this let it be, don’t map to a folder outside of a container

And when request comes to this folder in the container, don’t map this, because we have deleted node\_modules

**$(pwd) :/app** means



Map a folder (/app) present inside a container to a folder outside a container

If anytime container reaches /app directory, its going to redirect to the pwd

what ever is there in pwd-present working directory map to app folder present inside a running container

docker run -it -p 3000:3000 -v **/home/node/app/**node\_modules -v ~/frontend:**/home/node/app** USERNAME:frontend

Short hand docker compose

In docker file below are the commands

===============

RUN mkdir -p /home/node/app

WORKDIR /home/node/app

Way 3:- using docker compose command

**Alternate to big run command**

The above run command is big, so even for 1 container also we can go with “docker-compose.yml” file

Refer “1.1.comments-dynamic code refresh .YML” in this directory git hub

version: '3'

services:

web: ## this says create a container called web

build: . ## says build using the docker file present in this current directory

ports:

-"8080:3000 ## here the application is running inside docker port 3000

map the system port 8080 to docker container port 3000

so when we hit 8080 in our browser this is mapped to container port 3000

volumes:

- /app/node\_modules

- .:/app # this says when request comes to app folder inside a container that will be

#mapped to current directory

The above will face issues, because build specifies “.” Means current directory

Refer proj 72

Here context specifies

Where should we pull the information like files and folders

version:'3'

services :

web:

build:

context: .

dockerfile:Dockerfile.dev ##because we wrote the commands in "Dockerfile.dev" instead of dockerfile

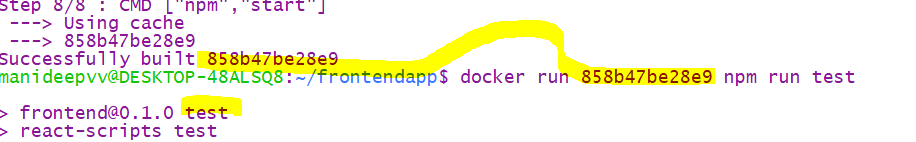
Running tests without docker compose

Here also we will run those tests in a new container

docker run <image-id> <our new command>

remember with image we can create container only, so in above step new container will be created

docker run 858b47be28e9 npm run test



Create a new space /container & run this additional command [npm run test ] in that space

But ,u can create space only when we have image

docker run -it <image name> <command name>

docker run -it 858b47be28e9 npm run test

run command will create a new container and execute this command

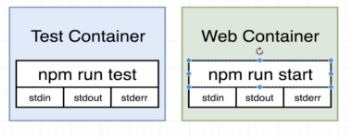
docker run -t manideep:node npm run test

Running tests with docker compose

Refer project 76(it has both for linux, for wsl) ,“ 2.0.seperate container to run the tests.yml”

It says create a separate container and change the start up command

Like below in docker-compose.yml file ,create a separate container called tests

here 2 containers-both test container,web containers are seperate

tests:// says create a new container called tests

stdin\_open: true

build:

context: .

dockerfile: Dockerfile.dev

volumes:

- /home/node/app/node\_modules

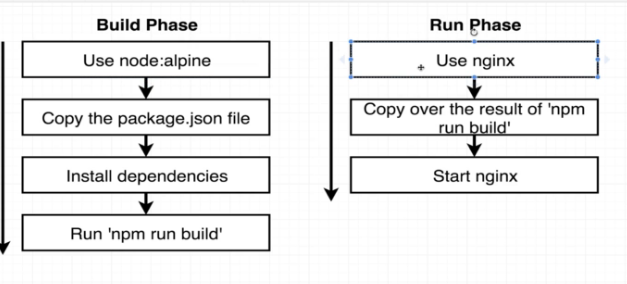
- .:/home/node/app ##when any request comes to this folder ,map to pwd

command: ["npm", "run", "test"]

## Implementing Nginix –Multi step builds

Nginix is a server same like tomcat, it’s a web server whereas tomcat is an application server

Refer prj number 80 wsl version for windows



So far our base image is only-1 , that is node:alpine version

But now we need 2 different softwares- we need nginix also

So we will follow multi step process, now the plan Is lets go with phase-1 ,execute “npm run build” command

And sample docker file and we will use the output of step-1 /phase-1 to the new phase

Refer “3.multi docker steps,,node,nginx.txt”

FROM node:16-alpine as builder #here builder is the phase name

WORKDIR ‘/app’ ## set the current dir to /app,so that here after all commands will be executed in this dir

# once proj is built ,build folder also will be created here

Copy package.json .

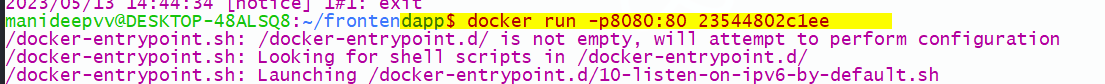
RUN npm install

COPY . .

RUN npm run build

FROM nginx

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



Nginx always runs on port 80

Once the image is created, ran that using image name

docker run <image name>

## Dockerizing a java application

Sample Docker compose file-create container and install maven,jdk

FROM openjdk:8-jdk-alpine

RUN apk update && apk add bash

WORKDIR /app

COPY . .

##To install maven

RUN wget https://mirrors.estointernet.in/apache/maven/maven-3/3.6.3/binaries/apache-maven-3.6.3-bin.tar.gz

RUN tar -xvf apache-maven-3.6.3-bin.tar.gz

RUN mv apache-maven-3.6.3 /opt/

ENV M2\_HOME='/opt/apache-maven-3.6.3'

ENV PATH="$M2\_HOME/bin:$PATH"

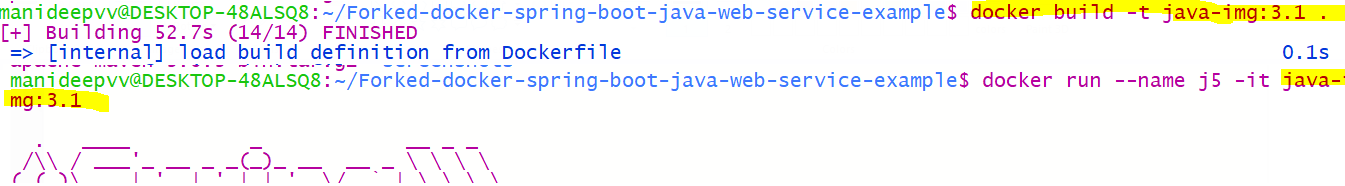
RUN export PATH

RUN mvn clean install

EXPOSE 8080

# Run jar file when the container launches

CMD ["java", "-jar", "target/docker-java-app-example.jar"]



The main problem with this approach is ,into main container maven software is also installed and the image is getting bulk

So to avoid it , create a multi stage docker file, where we know multiple intermediate containers will be created and from those multiple intermediate images will be created, so among those intermediate containers, in 1 of them install maven

And generate jar file, when u create a image from that container, in that final image done include maven software

So that image will be light weight

Multi stage docker file

The main pri

When u use this command, maven will be installed and path will be set automatically

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

Docker swarm

A ***swarm*** is a cluster of ***Docker engines, or nodes***, where you deploy services.

The ***cluster management*** and ***orchestration*** features embedded in the Docker Engine are built using ***SwarmKit***.

Docker engines participating in a cluster are running in ***swarm mode***. Swarm mode can be enabled by either initializing a swarm or joining an existing swarm.

**Docker Swarm** is a **Cluster Management and orchestration tool** which is available inbuilt in Docker engine.

The swarm nodes exchange overlay network information using a ***gossip protocol***.

By default the nodes ***encrypt and authenticate*** information they exchange via gossip using the ***AES algorithm in GCM mode***.

Manager nodes in the swarm rotate the key used to encrypt gossip data every 12 hours.

* Swarm does not require any additional installation. This comes as an in-built feature in Docker itself.

To initialize docker swarm service just type “docker swarm init”

* With the decentralised service discovery, Swarm v2 supports clusters with multi thousand nodes.
* Swarm mode works out-of-the-box. You need no changes to the existing container system to adapt to this tool

original swarm

* 1. Swarm manager nodes allocates a unique DNS name for the services. You will be able to find containers in the swarm via the DNS server embedded in the swarm.
  2. Swarm Kit has an internal load balancer which distributes the service containers within nodes. You can also include an external load balancer as well.
  3. Docker Node is a Docker Engine instance that is included in the Docker Swarm.

In Real time, these docker nodes are distributed across multiple cloud as well as physical machines.

There are 2 kinds of Docker nodes.

1. Manager Node: 2) Worker Node 3) **Drain Node:**

HERE The communication between the nodes is fully encrypted, it will internally uses TLS, PKI authorization

Manager Node is responsible for all Orchestration and container management activities required to keep up the desired system state.

**Worker Node:**

Worker node executes the tasks assigned by Manager node.

##### **Drain Node**

You can set the availability of any node to Drain, if you do not want to execute any task on the node.

Suppose you do not want manager node to process any task, you can set this as Drain node. The scheduler gracefully terminates any more task allocation and moves the node to Drain mode.

**Node** is the Key member of a Docker Swarm.

A Swarm can have **more than one Manager Node**. In this case, they elect their leader using **Raft algorithm** to conduct the Orchestration tasks.

**Manager nodes are also worker nodes.** But you can configure them as Manager only nodes thereby restricting them from working on any tasks.

Assume there is a cluster of 5 Managers running. If 3 upon 5 nodes fail, the system will stop scheduling any more tasks. This can tolerate a maximum loss of two manager.

The existing tasks will continue to run but the scheduler will not be able to re balance tasks and cope with any failures.

**Docker recommendations:**

* Have more than one Master node as well as odd number of Master nodes for High Availability.
* Have maximum of seven manager nodes for a Swarm.

|  |  |
| --- | --- |
| Initialize swarm | docker swarm init  docker swarm init --advertise-addr <ip-address> |
| List the nodes | docker node ls |
|  | docker service create --name testService --replicas 2 --constraint node.labels.disk==ssd tomcat |
|  |  |
|  |  |

Tokens to join swarm

When you initialize a swarm, the current node is marked as the **manager node** and it generates a new root certificate authority (CA) with a key pair. This is used by other nodes to join the swarm.

You can also add your own externally- generated CA, which is added using the flag --external-ca during swarm initialization.

Manager node also generates two tokens,

* **worker node token** (worker node to join swarm)
* **manager node token** (manager node to join swarm)

Swarm commands

|  |  |
| --- | --- |
| Init a swarm | docker swarm init --external-ca protocol=cfssl,url=https://ca.example.com |
| Init swarm | docker swarm init |
|  | docker swarm ca –rotate |
| Create a secret | docker secret create <desired-secret-name> <file-contains-password>  docker secret create ps1 pass.txt |
| List all secrets | docker secret ls |
| Inspect the secret | docker secret inspect <secret-name>  docker secret inspect secret1 |
| Deleting a secret | docker secret rm <secret-name> |
| Creating a service with secret | docker service create --name <container/servicename>  --secret <<existing-secret-name>> <image-name>  docker service create --name n3 --secret ps1 nginx:latest  ex:-2 docker service create --name tomcatService --secret secret1 tomcat |
| To see all services | docker service ls |
| Inspect a service | docker service inspect ngin2  docker service inspect <service name> |
| Scale up or scale down a service | docker service scale <service-name>=5  docker service scale ngin2=5 |
| Delete a service | docker service rm <service-name>  docker service rm n3 |
| To get all services in the node | docker node ps self  //to get all the services in the nodes |
| To get all the nodes | docker node ls |