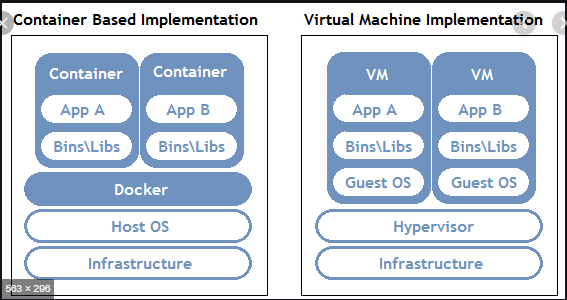
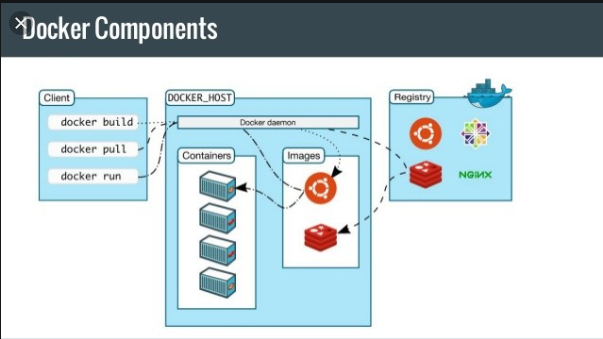
Docker 🡪OS level virtualization. Resource utilization of existing OS.

Bootup time is faster.

Dynamic memory allocation.

Easy integration.





Docker host will receive inputs from client and will perform actions.

Registry is like repository. Images can be stored in registry.

Jfrog can act like docker registry.

Default registry is dockerhub.

Docker images 🡪gives list of images

Docker pull ubuntu:latest 🡪will pull latest Ubuntu image from docker hub

Docker pull centos

Docker run -I –t –d --name=ex1 ubuntu 🡪docker run

Docker ps 🡪to check whether the image is started.

Docker exec –I –t ex1 /bin/bash 🡪can enter into ubuntu container.

Docker commit 🡪Create a new image from a container's changes

Usage: docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Options:

--config string Location of client config files (default "/root/.docker")

-c, --context string Name of the context to use to connect to the daemon (overrides DOCKER\_HOST env var and default context

set with "docker context use")

-D, --debug Enable debug mode

-H, --host list Daemon socket(s) to connect to

-l, --log-level string Set the logging level ("debug"|"info"|"warn"|"error"|"fatal") (default "info")

--tls Use TLS; implied by --tlsverify

--tlscacert string Trust certs signed only by this CA (default "/root/.docker/ca.pem")

--tlscert string Path to TLS certificate file (default "/root/.docker/cert.pem")

--tlskey string Path to TLS key file (default "/root/.docker/key.pem")

--tlsverify Use TLS and verify the remote

-v, --version Print version information and quit

Management Commands:

builder Manage builds

config Manage Docker configs

container Manage containers

context Manage contexts

engine Manage the docker engine

image Manage images

network Manage networks

node Manage Swarm nodes

plugin Manage plugins

secret Manage Docker secrets

service Manage services

stack Manage Docker stacks

swarm Manage Swarm

system Manage Docker

trust Manage trust on Docker images

volume Manage volumes

Commands:

attach Attach local standard input, output, and error streams to a running container

build Build an image from a Dockerfile

commit Create a new image from a container's changes

cp Copy files/folders between a container and the local filesystem

create Create a new container

diff Inspect changes to files or directories on a container's filesystem

events Get real time events from the server

exec Run a command in a running container

export Export a container's filesystem as a tar archive

history Show the history of an image

images List images

import Import the contents from a tarball to create a filesystem image

info Display system-wide information

inspect Return low-level information on Docker objects

kill Kill one or more running containers

load Load an image from a tar archive or STDIN

login Log in to a Docker registry

logout Log out from a Docker registry

logs Fetch the logs of a container

pause Pause all processes within one or more containers

port List port mappings or a specific mapping for the container

ps List containers

pull Pull an image or a repository from a registry

push Push an image or a repository to a registry

rename Rename a container

restart Restart one or more containers

rm Remove one or more containers

rmi Remove one or more images

run Run a command in a new container

save Save one or more images to a tar archive (streamed to STDOUT by default)

search Search the Docker Hub for images

start Start one or more stopped containers

stats Display a live stream of container(s) resource usage statistics

stop Stop one or more running containers

tag Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE

top Display the running processes of a container

unpause Unpause all processes within one or more containers

update Update configuration of one or more containers

version Show the Docker version information

wait Block until one or more containers stop, then print their exit codes

Run 'docker COMMAND --help' for more information on a command.

Docker commit <containerid> <newcustomimagename> :create new image from existing.can be done only if source container is up and running

Docker ps –a : to check stopped containers

Docker rm <containerid> : to delete container. Once done we will not see stopped containers and they will be deleted. So before deleting container has to be stopped. Once stopped we can see stopped containers by docker ps –a . Later we can delete them.

Docker tag maven alokvk1234/maven:latest : Create a tag TARGET\_IMAGE (alokvk1234/maven:latest )that refers to SOURCE\_IMAGE (maven)

Docker login : used to connect to docker hub. With username and password.

Docker push alokvk1234/maven:latest : used to push to docker hub with user name alokvk1234

Docker run -i -t -d -p 8080:8080 -p 50000:50000 --name=jen1 jenkins:latest : 50000 is default port for jenkins agent.

Docker host to container we use private ip and to external world we use bridge network

Docker host and container will ping using bridge network.

Docker network ls :used to check host and bridge network

Docker network inspect bridge : used to check container ip address. Ex if we have 2 to 3 containers we will get ip address of all the containers in docker host.

Docker logs jen1 : used to get the logs

If you don’t allocate bridge network to container you cannot reach your container from external source.Only if you enable bridge network you can reach container from external world. Bridge network uses IGW and not NAT gateway. Bridge network will be allocated by default from docker engine.

Usage: docker network COMMAND

Manage networks

Commands:

connect Connect a container to a network

create Create a network

disconnect Disconnect a container from a network

inspect Display detailed information on one or more networks

ls List networks

prune Remove all unused networks

rm Remove one or more networks

Docker network create mynw :to create own network mynw

Docker network disconnect bridge jen1 : to disconnect from current allocated network

Docker inspect jen1 : used to get meta information of container

Docker network connect mynw jen1 : to connect to new network mynw for container jen1

In single docker host we can create multiple jenkins containers.We can access each jenkins containers using port number.

Docker run -i -t -d --name=jen2 -p 8081:8080 -p 50001:50000 jenkins :we can start another jenkins container by name jen2 with port number 8081 as 8080 is already allocated by jen1 jenkins container.Here 50001 is TCP port for JNLP agents (Java Network Launching Protocol).

Manage networks

Commands:

connect Connect a container to a network

create Create a network

disconnect Disconnect a container from a network

inspect Display detailed information on one or more networks

ls List networks

prune Remove all unused networks

rm Remove one or more networks

docker run -itd --name=jen4 -P jen1 :will allocate dynamic port number (no need to give port number,-P gives dynamic port allocation

Requirement: We have a container which contains jenkins application in that which has 10 jobs running. So you want that container to run only for first 2 days of jan and after 6 months it should again trigger. So you don’t want the data to be stored in docker hub. You can store the data locally and inject it whenever required.

So we have options –v bind mount volume. Here whatever data generated from jenkins can be stored locally in the volume.

Create a directory vol1 and add few files in the directory in docker host.

Now you can mount the directory to the docker container by using below command.

Docker run –itd –name=vol1 –v /root/vol1:/home/ubuntu ubuntu :here it will create new ubuntu container with name vol1 where the directory /root/vol1 is mounted to container with directory /home/ubuntu (if the directory /home/ubuntu is not there then it will be created automatically). Now if we add few more files in /home/ubuntu in container and exit from container. Files which are added in container will be found in docker host /root/vol1. So if we delete the container files will be stored in docker host.

docker run -itd --name=jenv -P -v /root/jenkins:/var/jenkins\_home --user=jenkins jenkins :here all the plugin and configurations files which will be created while initialising jenkins /var/jenkins\_home) will be replicated in docker host(ec2 instance) /root/jenkins. Before this we have to give permissions to Jenkins directory by giving **chmod 777 jenkins/**. By doing this if you delete the container and repoint to another then data will be taken from docker host. We can get the password from docker host.

Docker log containername

Dockerfile: Image as code.

*Dockerfiles*: scripts to build containers, step-by-step, layer-by-layer, automatically from a base image.

**Dockerfile Commands**

1. ADD
2. CMD
3. ENTRYPOINT
4. ENV
5. EXPOSE
6. FROM
7. MAINTAINER
8. RUN
9. USER
10. VOLUME
11. WORKDIR

Sample dockerfile:

############################################################

# Dockerfile to build MongoDB container images

# Based on Ubuntu

############################################################

# Set the base image to Ubuntu

FROM ubuntu

# File Author / Maintainer

MAINTAINER Example McAuthor

################## BEGIN INSTALLATION ######################

# Install MongoDB Following the Instructions at MongoDB Docs

# Ref: http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/

# Add the package verification key

RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10

# Add MongoDB to the repository sources list

RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | tee /etc/apt/sources.list.d/mongodb.list

# Update the repository sources list

RUN apt-get update

# Install MongoDB package (.deb)

RUN apt-get install -y mongodb-10gen

# Create the default data directory

RUN mkdir -p /data/db

##################### INSTALLATION END #####################

# Expose the default port

EXPOSE 27017

# Default port to execute the entrypoint (MongoDB)

CMD ["--port 27017"]

# Set default container command

ENTRYPOINT usr/bin/mongod

docker build -t my\_mongodb . : used to build dockerfile. But then directory where dockerfile is present,in same directory the command should be executed. The **-t [name]** flag here is used to tag the image.

If from other directory docker build command looks like below

Docker build –t abc:ver1.0.1 /root/docker :in /root/docker dockerfile will be present.

From git repo:

Docker build -f Dockerfile https://github.com/quickfixtech/docker.git

docker build -t docker https://github.com/quickfixtech/docker.git :here docker is tag name.

docker-compose: Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration.

Require docker-compose.yml.

Docker-compose must be present in docker host as it is utility.

Whatever parameters you pass in docker run command all commands can be passed in docker-compose.yml file.

Define and run multi-container applications with Docker.

Usage:

docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]

docker-compose -h|--help

Options:

-f, --file FILE Specify an alternate compose file

(default: docker-compose.yml)

-p, --project-name NAME Specify an alternate project name

(default: directory name)

--verbose Show more output

--log-level LEVEL Set log level (DEBUG, INFO, WARNING, ERROR, CRITICAL)

--no-ansi Do not print ANSI control characters

-v, --version Print version and exit

-H, --host HOST Daemon socket to connect to

--tls Use TLS; implied by --tlsverify

--tlscacert CA\_PATH Trust certs signed only by this CA

--tlscert CLIENT\_CERT\_PATH Path to TLS certificate file

--tlskey TLS\_KEY\_PATH Path to TLS key file

--tlsverify Use TLS and verify the remote

--skip-hostname-check Don't check the daemon's hostname against the

name specified in the client certificate

--project-directory PATH Specify an alternate working directory

(default: the path of the Compose file)

--compatibility If set, Compose will attempt to convert keys

in v3 files to their non-Swarm equivalent

Commands:

build Build or rebuild services

bundle Generate a Docker bundle from the Compose file

config Validate and view the Compose file

create Create services

down Stop and remove containers, networks, images, and volumes

events Receive real time events from containers

exec Execute a command in a running container

help Get help on a command

images List images

kill Kill containers

logs View output from containers

pause Pause services

port Print the public port for a port binding

ps List containers

pull Pull service images

push Push service images

restart Restart services

rm Remove stopped containers

run Run a one-off command

scale Set number of containers for a service

start Start services

stop Stop services

top Display the running processes

unpause Unpause services

up Create and start containers

version Show the Docker-Compose version information

In docker-compose each and every container will be treated as services.

Ex:

version: '3'

services:

web:

build: .

ports:

- "5000:5000"

redis:

image: "redis:alpine"

here web and redis are 2 services. 2 containers will come up.

docker-compose up –d :here it will take default docker-comose.yml file and create and run containers.

docker-compose –f docker-compose.yml up –d

docker-compose ps : to check container status

docker-compose can work with single host.

Docker-swarm: clustering(orchestration) framework

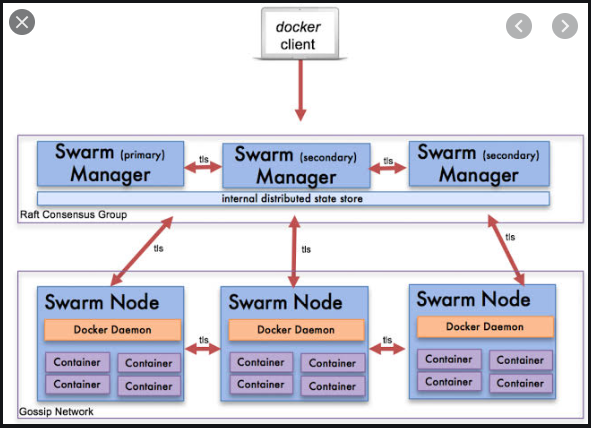
Current versions of Docker include *swarm mode* for natively managing a cluster of Docker Engines called a *swarm*. Use the Docker CLI to create a swarm, deploy application services to a swarm, and manage swarm behavior.

Feature highlights

* **Cluster management integrated with Docker Engine:** Use the Docker Engine CLI to create a swarm of Docker Engines where you can deploy application services. You don’t need additional orchestration software to create or manage a swarm.
* **Decentralized design:** Instead of handling differentiation between node roles at deployment time, the Docker Engine handles any specialization at runtime. You can deploy both kinds of nodes, managers and workers, using the Docker Engine. This means you can build an entire swarm from a single disk image.
* **Declarative service model:** Docker Engine uses a declarative approach to let you define the desired state of the various services in your application stack. For example, you might describe an application comprised of a web front end service with message queueing services and a database backend.
* **Scaling:** For each service, you can declare the number of tasks you want to run. When you scale up or down, the swarm manager automatically adapts by adding or removing tasks to maintain the desired state.
* **Desired state reconciliation:** The swarm manager node constantly monitors the cluster state and reconciles any differences between the actual state and your expressed desired state. For example, if you set up a service to run 10 replicas of a container, and a worker machine hosting two of those replicas crashes, the manager creates two new replicas to replace the replicas that crashed. The swarm manager assigns the new replicas to workers that are running and available.
* **Multi-host networking:** You can specify an overlay network for your services. The swarm manager automatically assigns addresses to the containers on the overlay network when it initializes or updates the application.
* **Service discovery:** Swarm manager nodes assign each service in the swarm a unique DNS name and load balances running containers. You can query every container running in the swarm through a DNS server embedded in the swarm.
* **Load balancing:** You can expose the ports for services to an external load balancer. Internally, the swarm lets you specify how to distribute service containers between nodes.
* **Secure by default:** Each node in the swarm enforces TLS mutual authentication and encryption to secure communications between itself and all other nodes. You have the option to use self-signed root certificates or certificates from a custom root CA.
* **Rolling updates:** At rollout time you can apply service updates to nodes incrementally. The swarm manager lets you control the delay between service deployment to different sets of nodes. If anything goes wrong, you can roll back to a previous version of the service.

Use master-slave architecture.

Master 🡪workernodes



Docker swarm :to bring up the framework

Docker node : list how many nodes

Docker service :alias of docker run but docker run will execute on single host but docker service in distributed environment.

Docker stack is like docker compose

Usually in dockers communication takes place in bridge network. But in Swarm communication takes place in overlay network (establish tunnel between manager and worker node).

Each and every manager and worker node acts like proxy. All nodes can be added in load balancer.

Management Commands:

builder Manage builds

config Manage Docker configs

container Manage containers

context Manage contexts

engine Manage the docker engine

image Manage images

network Manage networks

node Manage Swarm nodes

plugin Manage plugins

secret Manage Docker secrets

service Manage services

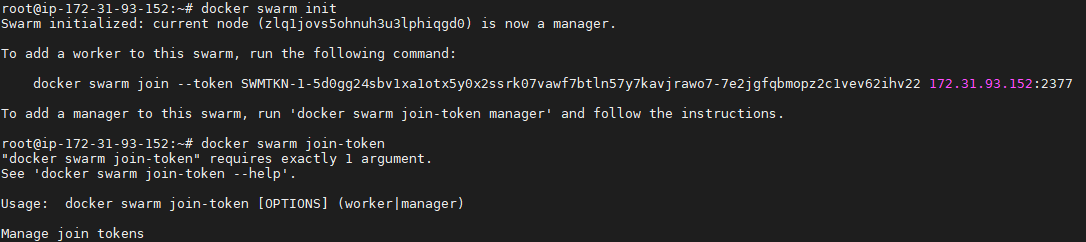
stack Manage Docker stacks

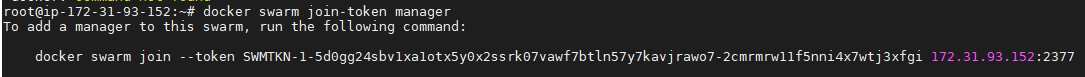
swarm Manage Swarm

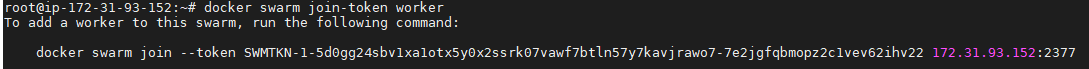
system Manage Docker

trust Manage trust on Docker images

volume Manage volumes



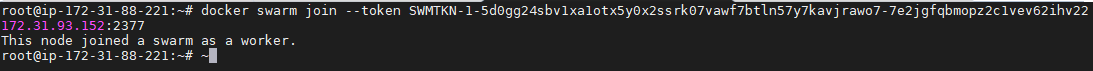




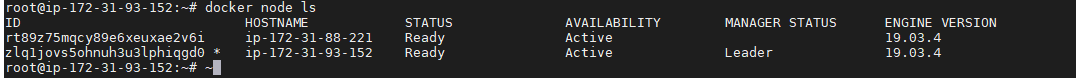


Docker node ls :gives status of nodes and its metadata

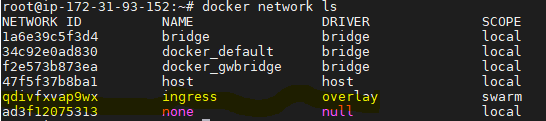
Now in worker node install docker and issue command to add worker node to swarm manager which you got



Now in manager issue docker node ls you will see below output.

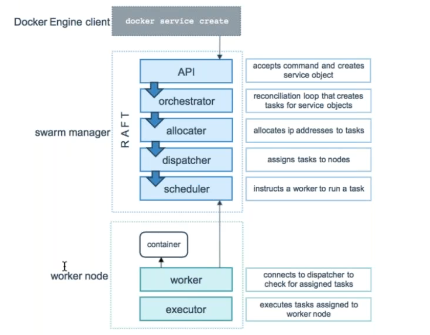


Docker network ls



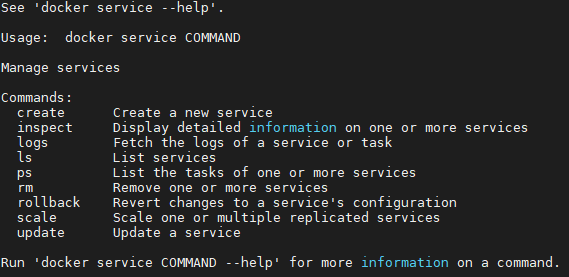
Docker service :If you want to start single network in worker node use below command in manager

Components in docker swarm:



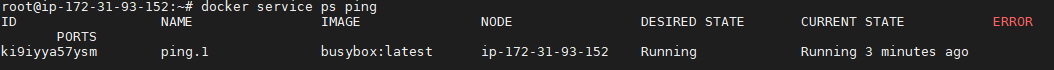
Usage: docker service create [OPTIONS] IMAGE [COMMAND] [ARG...]

docker service create --name=ping busybox ping google.com

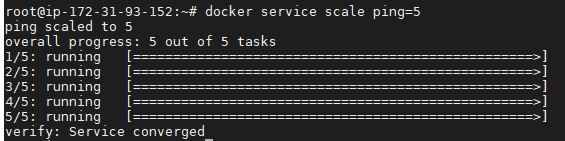


docker service ls :list services

docker service ps ping :gives where the service is started whether in manager or worker



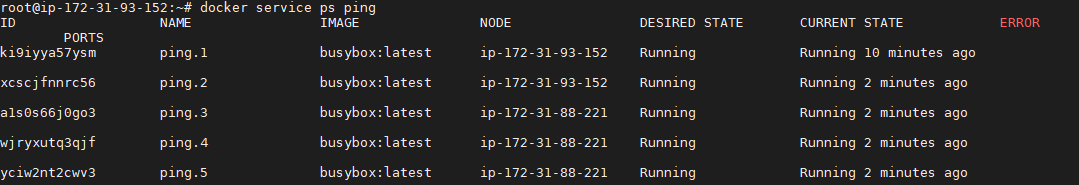
docker service scale ping=5 :here 5 replicas will be created



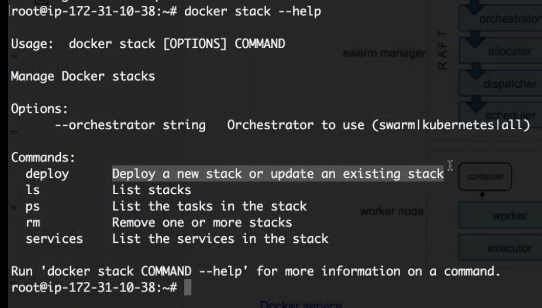
if we do docker service ls we can see 5 replicas created.



docker service ps ping :here we can see 2 containers are created in manager and 3 in worker.

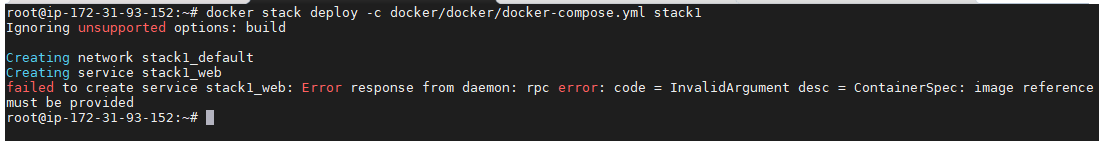


docker stacks :

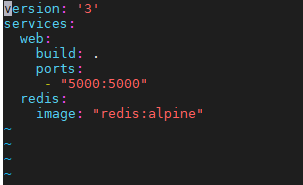


docker stack deploy -c docker/docker/docker-compose.yml stack1

here docker-compose.yml file should have image reference else it will throw error as below.

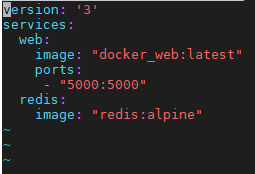


If like below is docker-compose.yml file

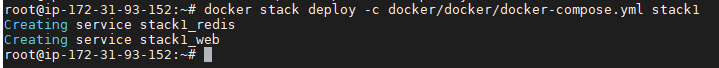


It will throw error image reference must be provided.

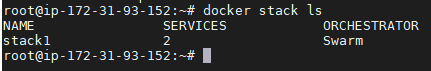
So edit docker-compose.yml file. Here instead of build option we are directly pointing created image docker\_web:latest.



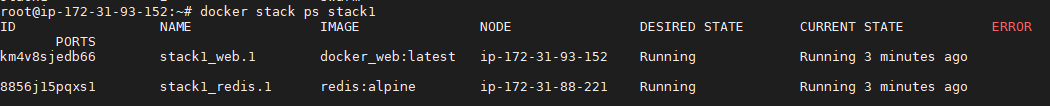
now it will create successfully.



if we give **docker stack ls** below is output.



docker stack ps stack1 :gives where the services are running.



Now in order to make docker client we need to install docker-machine in client so that all the commands can be executed through docker client.

docker-machine is utility to connect to remote machine

in order to install docker machine use below command

base=https://github.com/docker/machine/releases/download/v0.16.0 &&

curl -L $base/docker-machine-$(uname -s)-$(uname -m) >/tmp/docker-machine &&

sudo install /tmp/docker-machine /usr/local/bin/docker-machine

once done we need to prepare below URL

docker-machine create --driver generic --generic-ip-address=172.31.93.152 --generic-ssh-key ".ssh/id\_rsa" --generic-ssh-user=root 172.31.93.152

we can connect to remote by issuing below command in docker client

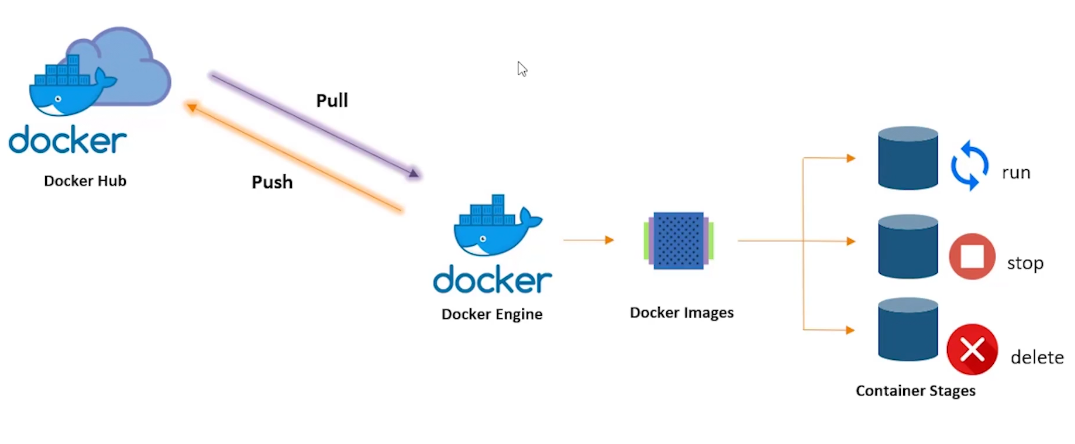
for password less authentication copy the public key of docker client where docker engine is installed and copy it to remote (can be swarm manager or swarm workernode)

eval $(docker-machine env 172.31.93.152)

**Docker:**

Container does not contain kernel in place but it shares the kernel of OS. But container has minimum binaries to run the centos or any other application.

**Docker Lifecycle:**



When you run an image it becomes container.

**Common Docker commands used:**

**docker --version** : to check which version of docker your using.

**docker pull <image\_name>** :to pull an image from docker hub.

**docker images** : List of images pulled from docker hub.

**docker run -itd <image\_name> :** to run docker image which will be converted to container.

Here **-it** is interactive mode and **-d** is run in background or detached.

**docker ps** : to view all the running containers

**docker ps -a** :to view running and stopped containers

**docker** **exec -it <container\_id> /bin/bash/** :to get into the container.

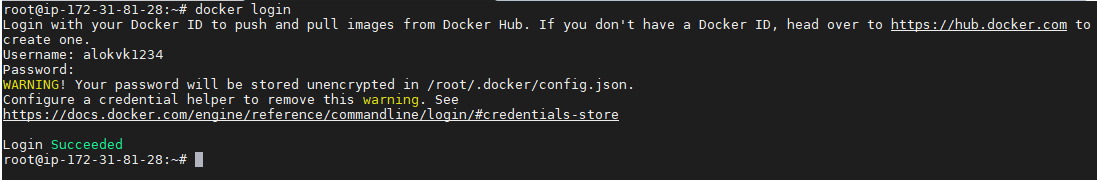
**docker stop <container\_id> :** to stop container. You will not be able to see the container when you issue docker ps.

**docker kill <container\_id>** : to kill container by stopping its execution immediately.

**docker rm <container\_id>** : will remove container. This is used to remove stopped containers. So when you issue docker ps -a then no stopped containers will be found.

**docker rmi <image\_id>** : To remove images. So when you issue docker images then no images will be found if you issue this command.

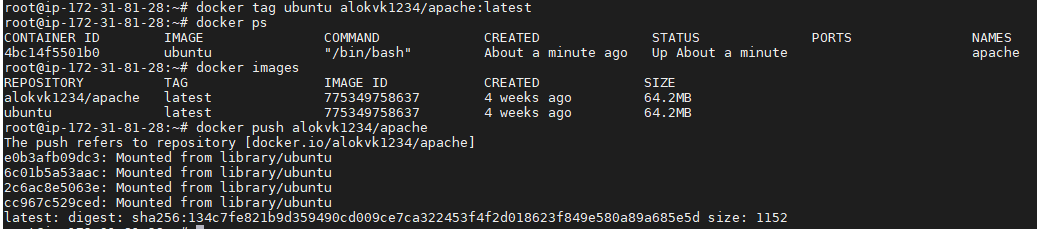
**docker login** :used to login to docker hub account.



**docker tag <source\_image> <target\_image>** : Create a target image (alokvk1234/apache:latest) which refers to source image ubuntu.

docker tag ubuntu alokvk1234/apache:latest

**docker push alokvk1234/apache** :Pushes docker images to docker hub.



**Dockerfile:**

It is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create automated build that executes several command line instructions in sequence.

**FROM** :Key word used to define the base image on which we will be building.

Ex: FROM ubuntu

**ADD <source> <destination in container>** : Add files inside a container.

Ex:

FROM ubuntu

ADD . /var/www/html

**RUN** : Used to add layers to the base image by installing components. Each RUN statement adds a new layer to docker image.

Ex:

FROM ubuntu

RUN apt-get update

RUN apt-get -y install apache2

ADD . /var/www/html

**CMD** : Is used to run commands on the start of the container. These commands run only when there are no arguments specified while running the container.

Ex:

FROM ubuntu

RUN apt-get update

RUN apt-get -y install apache2

ADD . /var/www/html

CMD apachectl -D FOREGROUND

**ENTRYPOINT** : used to run commands on the start of the container. These commands run when there are arguments specified or no while running the container. Only difference between CMD and ENTRYPOINT is ENTRYPOINT is used is there are arguments specified while running a container.

Ex:

FROM ubuntu

RUN apt-get update

RUN apt-get -y install apache2

ADD . /var/www/html

ENTRYPOINT apachectl -D FOREGROUND

**ENV <name of variable> <value>** : Used to define environment variables while container runtime.

Ex:

FROM ubuntu

RUN apt-get update

RUN apt-get -y install apache2

ADD . /var/www/html

ENTRYPOINT apachectl -D FOREGROUND

ENV name Devops Intellipaat

Note: Name of docker file should be **Dockerfile**.

Note: To run docker without sudo

**sudo usermod -aG docker $USER**

**docker build -t abc:ver1.0.0 /root/dockerfile** : Used to build dockerfile.

**docker run -itd -p 84:80 <image\_name>** : this is used to run docker container. Here we use port forwarding 84:80. So we can access this container from external world through port 84.

**docker rm -f $(docker ps -a -q)** will delete all the containers.

**Docker Storage:**

Data exists only when container is active. If container is deleted or crashed the data is also deleted along with the container. So it is stateless.

Due to this data on the writable layer in the container is written using storage driver.

To persist data inside the container, even after it is deleted, we have 2 options:

1. **Docker Volumes**: It is mountable entity which can be used to store data in docker file system.

Command:

**docker volume create <volume\_name>** to create volume

**docker volume ls** used to list all the volumes in your system

**docker run -itd --mount source=<source\_folder>,destination=<destination\_folder> <container\_name>** to attach a volume to container

Ex: docker run -itd --mount source=demo\_vol,destination=/app ubuntu

For example if you create a container and attach the volume demo\_vol and once the container is deleted, data inside demo\_vol will not be deleted. So you can launch another container and again attach the volume demo\_vol and you can see the data stored.

If you attach a volume and add 2 files in the volume, later if you attach same volume to another container which already have 2 files in it and if you add another file to it and go back to first container then you will find 2 files along with newly added file from 2nd container. So this is awesome feature of file sharing.

1. **Bind mounts:** Mount the directory of host machine to the docker container.

Command:

Syntax:

docker run -it -v <source\_directory>:<destination\_directory> -d <container-name>

Ex:

docker run -it -v /home/ubuntu/mount:/demo -d ubuntu

**Linking docker container:**

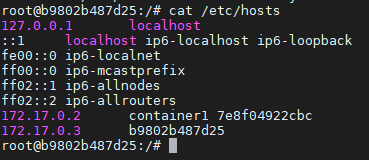
Linking is a legacy feature of docker, which is used to connect multiple containers. With linking, containers can communicate among each other. Name of container is important aspect while linking containers.

Command:

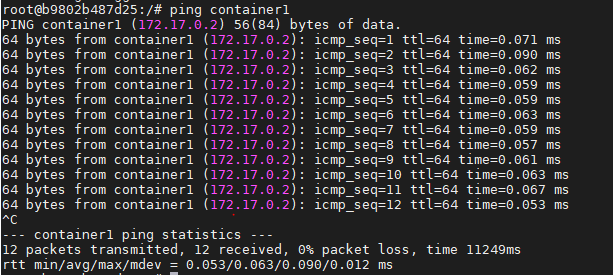
**docker run -it --name container1 -d ubuntu**

**docker run -it --name container2 --link container1 -d ubuntu**

Now if you got to hosts file in container2 then you will find container1 ip-address linked with container2. So in below screenshot we can see container1 ip-address in container2.



So we can ping containers.



**Monolithic**: A monolithic application is a single tiered software application in which different components are combined into single program which resides in a single platform. Performance is better because they don’t have latency.

If one service goes done then it may effect to the other services. If any issue on code for a particular service then we have to debug the entire code. Bug in any module can bring down entire application.

**Micro services** are software development architectural style that structures an application as a collection of loosely coupled services. All the services are independent.

**Docker Compose:** Is a tool for defining and running multi container docker applications. With compose you use YAML file to configure your application services. Then with single command, you create and start all the services from your configuration. Run docker-compose up and compose starts and run your entire app.

Below are commands to install docker compose:

**sudo curl -L "https://github.com/docker/compose/releases/download/1.23.1/dockercompose-$(**

**uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose**

**sudo chmod +x /usr/local/bin/docker-compose**

**docker-compose --version**

Refer below link:

<https://computingforgeeks.com/how-to-install-latest-docker-compose-on-linux/>

There are only 2 types of structures in YAML which you need to know:

* Maps : When we map a key to a value in YAML files, they are termed as MAPS

<keys> : <value>

Ex: Name : Intellipaat

Course : Devops

* Lists : YAML lists are sequence of objects.

Ex:

args

- sleep

- “100”

- message

Also indentation is very important in YAML files.

Sample docker compose YAML file:

version: '3'

services:

sample1:

image: 'httpd'

ports:

- "5000:5000"

sample2:

image: 'nginx'

To build docker compose file below is command.

**docker-compose up -d**

To remove all the containers which are added using docker-compose. It will not only stop but it will delete the containers.

**docker-compose down**

Sample docker-compose file for wordpresss:

version: '3.3'

services:

db:

image: mysql:5.7

volumes:

- db\_data:/var/lib/mysql

restart: always

environment:

MYSQL\_ROOT\_PASSWORD: somewordpress

MYSQL\_DATABASE: wordpress

MYSQL\_USER: wordpress

MYSQL\_PASSWORD: wordpress

wordpress:

depends\_on:

- db

image: wordpress:latest

ports:

- "8000:80"

restart: always

environment:

WORDPRESS\_DB\_HOST: db:3306

WORDPRESS\_DB\_USER: wordpress

WORDPRESS\_DB\_PASSWORD: wordpress

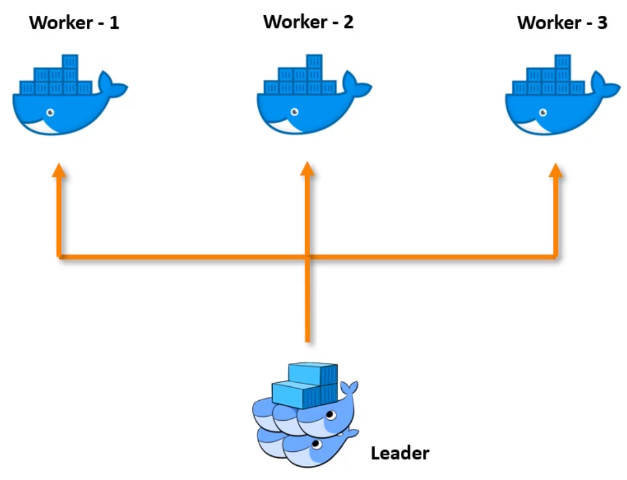
volumes:

db\_data: {}

**Docker Swarm**:

Docker Swarm is container orchestration tool which is scheduling and clustering tool for docker containers. With Swarm IT administrators and developers can establish and manage a cluster of docker nodes as a single virtual system.

**Architecture of Swarm:**



Docker swarm will be installed along with docker installation. No need to install it again as like docker-compose.

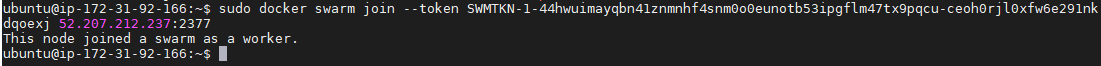
To create docker swarm cluster use below command.

**docker swarm init --advertise-addr=<ip-address-of-leader**>

**docker swarm leave --force** to leave the node forcefully from cluster.

To add a worker to this swarm, run the following command:

**docker swarm join --token SWMTKN-1-44hwuimayqbn41znmnhf4snm0o0eunotb53ipgflm47tx9pqcu-ceoh0rjl0xfw6e291nkdqoexj 52.207.212.237:2377**

****

**docker node ls** gives which is manager and worker node list.



**Service:**

An application can be deployed in docker swarm using service.

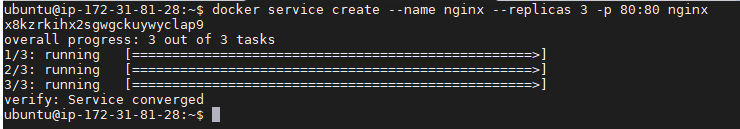
Containers on the cluster are deployed using services on docker swarm. A service is a long-running docker container that can be deployed to any worker node.

Services are additional layer which deals with networking part of containers.

Service is one end point through which we can reach any number underlying containers.

Creating service:

**docker service create --name <name\_of\_the\_service> --replicas <number\_of\_replicas> <image\_name>**

****

To check whether service is running on cluster use below command.

**docker service ls**

****

Now you can check whether application nginx is running on both master as well as slave. So 3 replicas of containers are randomly accessible from any of the nodes. So whenever you deploy a service on cluster then that service can be accessed by any of the nodes in cluster.

If any container is deleted then service will automatically bring up another container and maintain the value of replicas. So end user will not experience the down of application.

**Docker network:**

Docker network is used to have interactions between docker containers. This is the reason docker is very popular.

Types of networks:

**Bridge**: Default network. These are usually used when your applications run in standalone containers that need to communicate.

**host**: This is used when you want to interact with applications outside docker ecosystem.

**Overlay**: Overlay networks connect multiple docker daemons together and enable swarm services to communicate with each other. This can be used to establish communication between swarm service and standalone container or between 2 standalone containers on different docker daemons.

**macvlan**: Allow you to assign MAC address to a container, making it appear as a physical device on your network. Docker daemons routes traffic to containers by their MAC addresses.

**none:** Disable all networking. Usually used in conjunction with a custom network driver. This is not available for swarm services.