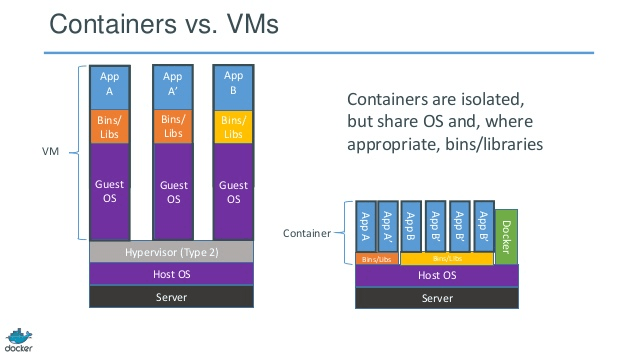
**DOCKER**

* In early days, we use separate systems for every application. Only that application will be running on that particular system. It is hard to maintain all the servers and it is money consuming.
* Later, **virtualization** came into picture, where we can host multiple os and applications in a single system as an independent machine using physical machine’s hardware. We can create n number of vm’s in a single machine, but the performance would be slow.
* Every vm acts as a independent machine with its own cpu, ram, hd taken from physical machine. The problem is, if the application in vm is using only half of the ram from the assigned. The rest is wasted as it can’t be used by any other vm’s.
* But with **docker**, you can avoid this problem. docker will take only as much as it want because it resides on host kernel by sharing kernel hardware. You don’t need to install any other software to use docker. Just you need to install docker.
* Docker is an containerization platform which is open source. It is a light weighted and fast container which can be used to package, run and distribute applications with in seconds. You can create n number of containers as you want in a single system.
* Docker packs applications and its dependencies, libraries together and gives you as one docker image from which you can launch containers. The container is given by docker hence we call it docker container. No other dependencies are required to run docker container hence it is light weighted.
* By providing a light weighted containers, it enables very efficient utilization of hardware resources (cpu, ram, hd) etc.
* You can ship containers to other platforms and run those containers in there. But with VM’S you can’t do these actions.
* Port - 2376



**COMPONENTS**

* **Docker client** - this is where we run any docker commands. It allows us to run these commands where a human can understand.
* **Docker daemon** - it is a service that runs on your host operating system. When you run any docker commands, docker cli translates it into http api call, send it to docker daemon. Then docker daemon evaluates the request , talk to your underlying os and execute your docker command.
* **Docker** **image** - docker images is a template which is needed to build a running container. Images are building blocks of docker. It takes series of instructions from a text based configuration called **dockerfile** and built the infrastructure according to that file. It can run in any environment where docker is installed.
* **Docker Container** - Containers are runtime (or) execution aspect of docker. Docker borrows the concept of normal shipping containers that ship goods and used it in docker, which is used to ship software. it can create, starts, stops, restarts, destroyed the containers.
* Just like shipping containers, docker doesn’t care of what is inside of a container, it performs all the actions in the same way for all the containers either it is a host os (or) a web server (or) a database server. You can ship and run docker containers in any environment where docker is installed.
* **Docker Hub** - Docker registries are the distributed component of docker. Docker stores all the build images in the registry.
* There are 2 types of registeries **public** and **private**. Docker inc provides public registry for images called **docker** **hub**. You have to create an account to store and share your images.
* In docker hub, you can find docker images to download and can run in your local machines.
* These public images are ready with os and applications you needed, you just have to download (pull) those images to your local machine.
* You can create your own images and share them to others by uploading them to docker hub.
* Docker provides a **private** repository called **‘registry’.** You can download this image from hub and push your images to this private registry.

**INSTALLATION**

* To install docker, first you have to add docker yum repository in your linux machine.

**yum-config-manager --add-repo** [**https://download.docker.com/linux/centos/docker-ce.repo**](https://download.docker.com/linux/centos/docker-ce.repo)

* You have to install **container-selinux** repo to install docker.
* Install docker **= yum install docker-ce.**
* Do **yum update** and install docker..**yum install docker**.
* If you are using aws ec2 instance, the docker yum repo is present by default. So, no need to create yum repo, just give yum install docker. It will install automatically.

**DOCKER COMMANDS**

* **docker --version (or) -v =** To see docker version.
* **/var/lib/docker/ =** docker root directory.
* **service docker start =** To start docker**.**
* **docker login =** login to docker with docker id to push and pull images to docker hub.
* **docker info =** shows info about docker installation, includes kernel version, containers running, stopped, images, ram(**using by docker**), cpu, os type etc.
* **docker images =** To see if you have any docker images in your server**.**
* **docker search image-name** = To search if the specific app image is there in docker hub (or) not.
* **docker pull image-name =** To download a docker image to your local system.
* **docker run –it image**  = To create and login to docker container from docker image.

**i – interactive, t = terminal**

* **docker run –itd** **image** = To create container without login into that (run in background).

**d – detached mode.**

* We can create multiple containers from an image. Every container differs from each other.
* **exit** = To exit from container. It will stop the container.
* **docker run –d --name=NAME image =** To specify a name to container.
* **docker run –d --hostname=HOSTNAME image =** To give a hostname to container.
* **Ctrl + p + q** = To leave docker in running state. Container will be running.
* **docker create –p 80:80 IMG-Name =** To create a docker container (without starting).
* **docker start con-ID =** To start a created container.
* **docker attach container-id** = To get back to the running container.
* **docker exec –it con-ID /bin/bash =** To execute the shell and login to container.
* **docker stop container-id** = To stop running container.
* **docker pause con-ID =** To pause a container.
* When you pause a container, it suspends all the process. In linux, it suspends the process by **cgroups freezer**.
* **docker unpause con-ID =** To unpause a container.
* **docker rename con-ID new-name** = To rename a docker container.
* **docker start container-id =** To start stopped container.
* **docker restart con-ID =** To restart a container.
* **docker rm container-id –** To delete a container.
* **docker rm –f container-ID** = To delete a running container.
* **docker top container-id** = Shows top running process in that container.
* **docker stats container-id =** Shows container statistics like cpu usage, memory usage & limit.
* **docker rmi image** = To delete an docker image.
* **docker rmi –f image** = To delete docker image forcefully.
* **docker images –q =** Shows images with only image id.
* **docker** **ps** = Shows running docker images.
* **docker ps –a** = shows running and exited containers.
* **docker inspect =** shows low-level information on docker objects(i.e containers, volumes, images, n/w).

**docker inspect image-name =** show info about that specific image.

**docker inspect container-name =** show info about that specific container.

* **docker history image** = Shows commands run on that image.
* **docker cp filepath container-id:/destpath** – To copy a file from local to docker running container. It will replace the old file with this new file.
* **docker cp con-ID:/filepath localpath** = To copy file from container to local.
* **docker events --since ‘year-month-date’ =** shows realtime events from the docker.
* **docker diff =** shows the changes in files and directories. The changes includes adding, deleting represented by A,D,C flags.

**docker diff con-ID =** shows the changes in this container**.**

* **docker logs con-ID --since 1h =** To see logs of that specific container for last 1 hour**.**
* **docker logs con-ID --since ‘y-m-d’ =** To see logs of that container on that day.
* **docker logs con-ID --since 39m =** To see logs of that container for last 30 min**.**
* **docker logs con-ID --until 30m =** shows logs upto that point**.**
* **docker logs –f con-ID =** To see and follow the container logs.
* **docker logs --follow con-ID =** To follow docker logs output (**-f**).
* **docker logs --tail 5 con-ID** = To see last 5 lines logs of a container.

**SAVE & LOAD**

* you can copy your containers and images to remote servers with docker save and load commands. By taking backup to a file, you can share the containers and images.
* **docker save image-name > file.tar** = To take backup of an docker image to tar file.
* After backingup image to file, you can share that file to remote servers and use those images to start containers.
* **docker load < file.tar** – To load the docker image from the tar file.
* **Export** – didn’t export history and layers.
* **Save** – exports layers and history.
* We can rollback to previous layers if we exported image with **save** cmd. Because, it save all the layers and history. You can’t rollback with **export**, because it doesn’t save layers & history.
* If you want to share a container to remote machines, commit the running container as a image, save that image and share the image.

**DOCKER HUB**

* you can push your images to docker hub by following these steps.

**Commit your docker image with your repo and tag.**

**Login to docker hub from command line.**

**Push the image to hub.**

* The repository name and user name are important as they should be in order like **user/repo.**
* **docker login** – To login to hub from cmd line. Enter you username and password.
* Now you are loggedin to your dockerhub account.
* **docker logout** - to logut from dockerhub account.
* While commiting image to repo, specify **tag** after the repository name to identify in hub.

**For ex: username/repositoryname:tag**

* **Tag** is nothing but a **name** to the images which are pushing to docker hub. With this tag you can identify your images in docker hub. So while pulling image from docker hub you can specify the tag to download to local.
* If you didn’t specified any tag, it will take the default tag(**latest**) for all the images.
* **docker commit con-ID user/repo:tag** – to commit along with tag.
* After commiting an image check the docker images in your system, you will find your newly commited container as image. You can use this image to launch docker containers.
* **docker push user/repo:tag** – To push an image to your docker hub..
* **docker pull user/repo:tag =** To pull an image from you docker hub.
* **docker run -itd –restart unless-stopped redis**

-no : Do not automatically restart the container. (the default)

- on-failure:

- unless-stopped: Restart the container unless it is explicitly stopped or Docker itself is stopped or restarted

- always : Always restart the container if it stops.

**DOCKER PRIVATE REGISTRY**

* We can push our own images to docker hub, which is a public repository. Everyone can see our images.
* There might be a situation, where we need to maintain our own private repository for our organizations/teams. For that, we use a private repository called **registry**.
* Download the registry image from hub and run that image.

**docker run –d –p 5000:5000 --name localregistry registry**

* **registry** – It is the container name to host private repository.
* **5000:5000** – mapping the port of docker registry to localhost.
* Check the docker images in your system, to know whether registry in downloaded (or) not and check docker ps to verify its running (or) not.
* To push images to private repo, first we have to commit containers to localhost and tag those images with registry.
* **docker commit conid imgname** - to commit an container locally.
* **docker tag image-id localhost:5000/image-name** – To tag an image to localhost private registry.
* **docker push localhost:5000/image-name** – To push the tagged image to private registry.
* Now, delete the pushed image from your host system and try to pull it from private registry.
* **docker pull localhost:5000/image-name** – To pull an image from private registry.
* The docker private registry container should be running to do all these actions(pull, push).
* If you delete the registry container all your images pushed to pricate repository will be deleted along with the container.

**DOCKER VOLUMES**

* Basically, containers are ephermeral, means once a container is removed it is gone and all the data will be gone inside a container.
* With the help of volumes, we can separate the data from container life cycle. Means, even after the container is deleted, the volume will be present, which we can reuse.
* **Data volumes –**

**It is a specially designed directory in containers.**

**It is created when container is created.**

**It won’t delete when container is deleted.**

**Data volumes can be shared across containers too.**

* **docker volume create vol-name** – To create a volume.
* **docker run –it –v vol-name:/vol-name image-name** – To create and mount vol to container and create container.
* It will mount the volume in container and launch the container in interactive mode. Write some data in the volume dir and exit from container.
* Once exiting from the container, it will be no longer available, but the volume will be present. You can check by command below.
* **docker volume ls** – To check all the volumes present. Find your volume in the list by your vol name.
* **docker volume inspect vol-name** – To see what we did in the volume. It will show you the data you added to that volume.
* Create a new container and mount the existing volume to that container. Afetr entering into container you will see the data, which saved in your volume.
* **docker volume rm vol-name** – To delete a volume.
* Upto now, we have created new volumes and attached it to containers. We can also create new volumes and copy the existing volume dir data to that new volumes.
* **docker run –it –v old-vol:/new-vol image-name** – Creates a new vol and copies the data from old vol to new vol in the container. New volume name can be anything.
* You can share the volumes between multiple containers. Just mount the volume to different containers while creating them. That’s it.
* If you have updated the data in oldvol, the newvol will be updated automatically and viceversa.
* **docker run –it –v oldvol:/newvol:ro image-name** – To attach a vol with read only permissions.
* If you stopped a container from running, the volume will be unmounted and once the container starts, again the volume will mount automatically.
* You can also mount a local dir in docker container.
* **docker run –it –v localdirpath:/mountvolname image-name** – To copy and mount a local dir to docker container.
* It will update the data automatically, once the change occurs either in local dir (or) in docker container.

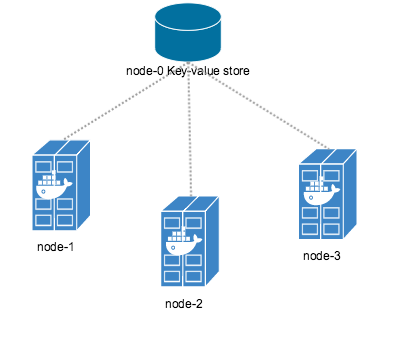
**LINKING CONTAINERS**

* In order to connect multiple containers together and transfer information between them securely, we link containers.
* For example, we have a mysql container and web server container, we have to link these two container to talk to each other.
* **Docker Link flag** – This method is deprecated. No one uses this method now. Start container 1 normally and start container 2 with **--link** option.
* **docker run –d --link --name con1-name con2-name** = To link container 1 and 2.
* You can also link container using docker **network**. By launching docker containers in same network you can make communication between them.

**Create your own network.**

**Launch containers in your own network. Try to ping.**

* **docker network ls** – To see all networks in docker.
* **docker network create --driver bridge n/w-name** – To create a bridged network with a name.
* you can also specify ip address range, subnet, gateway while creating network.
* **docker network create –d bridge --subnet=192.168.0.0/16 --ip-range=192.168.20.0/24 --gateway=192.168.20.1 n/w-name** = To create a network with specific subnets.
* Now, while starting a container, specify the network to run container in this specific network.
* **docker run –d –network=n/w-name img-name** – To launch a container in specified network.
* You can also connect and disconnect network from an container.
* **docker network connect n/w-name con-ID** = To connect a network to container.
* **docker network disconnect n/w-name con-ID** = To disconnect a network from container.
* **docker network rm n/w-name =** To delete a network.
* You can’t delete a network if it is connected it to a running container. You have to disconnect it from container first and remove the network.
* Bridge network is only works for local communication. You can’t communicate with other docker host using bridge network.
* Inorder to make communication between multiple docker host, we have to create and use **overlay** network.
* We can create overlay network only if we have initialized swarm. So initialize swarm first.
* Create a overlay network **docker network create –d overlay n/w-name**
* All swarm traffic is encrypted by default using **AES** algorithm. Swarm nodes rotates the key using for encryption for every 12 hours.
* To encrypt data as well, use **--opt encrypted** while creating overlay network. This enables **IPSEC** encryption at the level of **vxlan**.
* Docker will creates a IPSEC tunnels between nodes, where tasks are scheduled for services attached to overlay network. These tunnels also uses AES algorithm.
* By default, swarm uses ingress overlay network, you have to specify your custom overlay network to use. **(--network overlayname).**
* Above we have create overlay network after initializing swarm, now we will create overlay network **without** **swarm**.
* To setup this, we need key-value store used by docker daemon which holds info about network state which includes network **discovery**, **endpoints**, **ip** addrss etc and enables cross-host container to container communication.
* Docker supports **consul**, **zookeeper**, **etcd** key-value stores. But, in this example we use **consul**.
* Download consul docker image and run it.
* Allow consul port **8500**.
* Docker daemon port = **2375**
* The below diagram shows node0 configured as keyvalue store.



* Now, we have to configure other docker hosts to listen to the node0 key-value store.
* Edit /etc/default/docker and paste the following content.

**DOCKER\_OPTS="-H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock --cluster-store=consul://<NODE-0-PRIVATE-IP>:8500/network --cluster-advertise=eth0:2375"**

* Replace NODE-0-PRIVATE-IP with your server private ip.
* Restart the docker service in all the other hosts.
* Now you can create overlay networks and run containers in that network. The containers from multiple hosts can ping from one another now.

**DOCKER FILE**

* A docker file is a script which contains commands and instructions, which executes them sequentially in order to create an docker image. You have to build the image from docker file. The build is run by docker daemon. It will transfer the whole context to docker daemon.
* Here are some dockerfile commands we use.
* **FROM** – The base image to create our new image. It should be on top of the dockerfile.
* **MAINTAINER** – Contains the name of the maintainer of the image.
* **RUN** – Used to execute commands during build process of docker image.
* **ADD** – Used to copy file from host machine to docker container. It have decompression feauture. You can use url to download files. It will download from the url and send context to daemon.
* **COPY** – same as ADD. But, doesn’t have decompression feauture.
* **EXPOSE** – map a port to docker container.
* **ENV** – Define an environment variable.
* **CMD** – means what should happen when you run a container. Allows you to set a command, which will execute when you run a container. Once the CMD command gets executed, the container will stop running, thinking that job is done.
* You can pass a parameter while running the container, cmd will take this new parameter ignoring old one.
* **ENTRY** **POINT** – Allows you to configure container as executable. It is similar to CMD but allows you to specify a command with parameters.
* **WORKDIR** – Directive for commands to execute.
* **USER –** Set the user for container created from the image.
* **VOLUME –** Enable access between local machine and docker.
* We write a sample file to install httpd.

**FROM centos**

**MAINTAINER** [**name@mail.com**](mailto:name@mail.com)

**RUN yum update –y && yum install httpd –y**

**EXPOSE 80**

**ADD /root/index.html /var/www/html**

**ENTRYPOINT [/usr/sbin/httpd,”-D”,”FOREGROUND”]**

* Execute the above docker file with docker build command.
* **docker build –t imagename:tag dockerfilepath** – To build image from docker file.
* **-t** – Giving tag to that image.
* After building an image run the image to launch container with httpd. Check the browser with your host ip and port you mentioned in dockerfile.
* **docker run –d –p 8080:80 imagename:tag** – To launch container from your custom image.
* **8080:80** is used to bind the port with localhost port.
* You can start the http server inside the container with **/usr/sbin/httpd &** command.
* If you want to install nginx with docker file. Just change the **entrypoint** to **[“nginx”, “-g daemon off;”].**
* You can add volumes in docker file **VOLUME /sourcepath /mntpoint**.
* We can deploy war files by using tomcat docker container. We can write dockerfile by using tomcat image to login to manager host.
* Download tomcat docker image.
* Write a dockerfile to copy users file to docker container.

**FROM tomcat:latest**

**MAINTAINER name**

**ADD tomcat-users.xml /usr/local/tomcat/conf/**

**EXPOSE 8080**

* Execute the docker file to build a new image from tomcat image with the additional settings as you specified.
* **docker build –t image\_name:tag .** – To create a new image from docker file.
* Run the docker image to start tomcat - **docker run –d –p 8080:8080 imagename:tag**
* It will start your custom tomcat container. You can login to tomcat manager console with your credentials.
* If you want to deploy war file to tomcat container. Just Add copy to dockerfile and build it.

**COPY warfilepath /usr/local/tomcat/webapps/name.war**

* It will deploy war to tomcat. You can search by url with port and app name.
* If you want to use oracle jdk, you have to use **dordoka/tomcat** image, which contains oraclejdk and tomcat. Official oracle jdk image is not available in docker.

**DOCKER COMPOSE**

* Compose is a tool for defining and running multi-container applications. With compose, you use a yaml file to configure your application services. Then, with the single command you can create, link and start all the services from the configuration.
* Compose is basically three steps.

**Define your app environment in a dockerfile.**

**Define the services that make up your app in compose.yml file. So, that they can run together in isolated environment.**

**Run docker-compose up and docker starts and run your entire app.**

* To install docker compose, you need python-pip.
* Install pip first and install docker compose.
* **pip install docker-compose** – To install compose.
* **docker-compose --version** – To check version. (if output shows, you have installed compose successfully).
* The yaml file should be named as docker-compose.yml. Otherwise docker compose won’t read the file.
* **docker-compose up** – To run the containers.
* **docker-compose stop** – To stop all the containers.
* **docker-compose ps** – To see all running containers started with compose tool.
* **docker-compose start** – To start all containers which created by compose tool.
* For ex:

**version: ‘3’**

**services:**

**nginx:**

**image: nginx**

**ports:**

**- 8080:80**

**volumes:**

**- “./localdir:/dockerpath”**

**networks:**

**- networkname**

**mysql:**

**image: mysql**

**ports:**

**- 3306:3306**

**environment:**

**MYSQL\_ROOT\_PASSWORD: password**

**networks:**

**- networkname**

**wordpress:**

**image: wordpress**

**ports:**

**- 8008:80**

**environment:**

**WORDPRESS\_DB\_HOST: mysql**

**WORDPRESS\_DB\_PASSWORD: password**

**networks:**

**- networkname**

**networks:**

**networkname**

**driver: bridge**

* In the above example, we have created nginx, mysql and wordpress in the same network.
* We have made an connection between mysql and wordpress by entering wordpress host as mysql.
* Check the connections with localhost and ip whether services are working (or) not.
* Normally we have to write separate dockerfiles for each service and build those images one after the other which wil take more time. But with docker compose, we can up the 3 servcies with one command by writing all of the services in docker-compose file.
* You can use your dockerfile with docker compose. So, it will build from dockerfile and start the services from compose file.

**DOCKER SWARM:**

* It is a technique to create and maintain clusters of docker engines.
* A swarm is nothing but a group of machines that are running docker and joined as a cluster. After this, you can use docker commands but now they will execute on cluster by swarm manager. The machines in swarm can be physical (or) virtual. After joining swarm they called as nodes.
* You can run swarm services and standalone containers on same docker machine.
* The swarm manager automatically assign ip address to thre containers in all nodes.
* Swarm supports scaling, for each service you can declare the no of tasks to run. So when you scale up and down, it automatically adds and removes the tasks to maintain desired state.

**FEAUTURES:**

* **HA -** if one node fails, other nodes takes its place automatically without interrupting the cluster.
* **AUTO LOAD BALANCING –** automatically share the load.
* **DECENTRALIZED ACCESS -** can access from anywhere with ssh**.**
* **EASY TO SCALE-UP DEPLOYMENTS**
* **ROLLING UPDATES**

**SETUP SWARM:**

* Take 3 servers(in my case) one for manager node and 2 for worker nodes.
* Install , enable and start docker on those 3 servers. Make sure the network communication between manager and nodes is working properly by pinging.
* Open these following ports in those 3 servers.
* **2377** - For cluster management communications.
* **7946** - For communication among swarm nodes.
* **4789** - For overlay network traffic.
* By default, swarm mode is disabled you have to enable it.
* To enable swarm on a machine, run this command.

**docker swarm init --advertise-addr <manager ip>**

* When you initialize swarm, two networks are created in that host.
* **Ingress** - it is an overlay network which controls data traffic between swarm services.
* **Docker\_gwbridge** - it is an bridge network, which connects the individual docker daemon to other daemons participating in swarm.
* You can launch your containers in default overlay network (or) you can create your own overlay network and launch container in it.
* After running init command, now your docker machine acts as swarm manager and you can add other docker machines to this manager as worker nodes.
* When you init the swarm, it generates a **token** . this token is useful for worker nodes and manager nodes to join to this swarm. Make a note of that token to join docker machines to this swarm.
* This node is designated as manager node and its starts listening on advertise ip over port **2377**.
* Go to your nodes, enter the token command which is generated by manager in all the nodes.

**docker swarm join \**

**--token SWMTKN-1-4o2koou6ia0omjy3p9ws9xugyqhv66pcxp2xjzxoye1rk0s8rl- 7ou2c6w4pwbud5ss1mrfveypb \**

**35.154.139.30:2377**

* This is the token along with your **manager** **ip** and **port**, which is generated by my docker swarm. For you it will be different.
* After entering the token in nodes, you will see a message like “**This node joined a swarm as a worker”**. Means that this docker machine is joined docker swarm as worker node successfully.
* You can join as many docker machines as you want to swarm as worker nodes.
* The token will be different for manager nodes and worker nodes. You can check the token by entering these commands.

**docker swarm join-token manager** - shows manager token

**docker swarm join-token worker -** shows worker token

* Once you join the nodes, you can check the list of nodes in swarm.

**docker node ls**

* From the output, you will see **id** of nodes, **hostname**, **status**, **availability**, **manager** **status**.
* Availability has 3 states,
* **Active** - Scheduler can assign tasks to the node.
* **Pause** - Scheduler can’t assign new tasks, but existing tasks are running.
* **Drain** - Scheduler can’t assign new tasks, existing tasks move to other nodes.
* Manager status has 4 states,
* **No** **Value** - It’s a worker node and does not participate in swarm management.
* **Leader** - It’s the leader and makes the swarm management and decisions.
* **Reachable** - It’s a manager but not leader. Will become leader if current leader is down.
* **Unavailable** - It’s a manager, but not able to communicate with other managers.
* Once you joined a machine to swarm as worker, you can check its details by inspect.

**docker inspect <worker hostname>**

* To update the availability status of a node

**docker node update --availability <state> <node-hostname, id>.**

* To promote a node from worker to manager

**docker node promote <node-hostname,id>**

* To demote a node from manager to worker

**docker node demote <node-hostname,id>**

* To assign a label to manager/node

**docker node update --label-add Env=labelname <nodeid>**

* To leave a node from cluster. Type this from the node

**docker swarm leave**

* When you type this command, worker just leaves from the cluster. You have to delete the node from manager to completely remove from the cluster.
* To delete a node form cluster

**docker node rm <node-id/hostname>**

* By default, all manager nodes acts as worker nodes too. To prevent this, set manager availability to drain. Then docker stops tasks on drain node and pass to active nodes.

**docker node update --availability drain node\_name**

**SWARM SERVICES:**

* Service is the definition of the task that run on worker nodes. When you create a service you specify which image to use and which commands to execute inside running containers.
* There are 2 types of swarm service modes.
* **Replicate** - you can pass number of replicas for the services and swarm maintains the count. For ex, if you specify 2 replicas and if 2 replica is down swarm will create new replica automatically to maintain the count. You can specify replicas count while creating a service **(--replicas 3).** By default, it will take replicated mode.
* **Global** - the swarm runs one task for the service in every available node. You can specify like this while creating a service **(--mode global)**
* To create a service

**docker service create –p 8080:80 --name <name/of/service> imagename**

* To see what services are running totally

**docker service ls**

* To see service running on which node

**docker service ps servicename**

* To scale up services

**docker service scale service\_name=5**

* To inspect a service

**docker service inspect --pretty service\_name**

* To delete a service

**docker service rm service\_name (or) ID**

* To update an image which is used by running service

**docker service update --image image\_name service\_name**