What is Docker ?

Docker is a tool designed to make it easier to deploy and run applications by using containers.

Most Common Problems for Developers :

The code works on developer system but the same code does not work on the {roduction Environment. Docker at a very basic level resolves this issue of an application working on one platform and not working on some other platform.

Containers allow a developer to package up an application with all of the parts it needs, Such as Libraries and other dependencies & ship it all out as one Package. Developer will package all the software and its components into a box which we call it as container & docker will take care of shipping this container to all the possible platforms. This is how we resolve the issue of an application working on one environment or a platform & not working on another.

Advantages of Docker :

Docker enables more efficient use of system resources

Instances of containerized apps use far less memory than virtual machines, they start up and stop more quickly, and they can be packed far more densely on their host hardware. All of this amounts to less spending on IT.

The cost savings will vary depending on what apps are in play and how resource-intensive they may be, but containers invariably work out as more efficient than VMs. It's also possible to save on costs of software licenses, because you need many fewer operating system instances to run the same workloads.

Docker enables faster software delivery cycles

Enterprise software must respond quickly to changing conditions. That means both easy scaling to meet demand and easy updating to add new features as the business requires.

Docker containers make it easy to put new versions of software, with new business features, into production quickly—and to quickly roll back to a previous version

if you need to. They also make it easier to implement strategies like blue/green deployments.

Docker enables application portability

Where you run an enterprise application matters—behind the firewall, for the sake of keeping things close by and secure; or out in a public cloud, for easy public access and high elasticity of resources. Because Docker containers encapsulate everything an application needs to run (and only those things), they allow applications to be shuttled easily between environments. Any host with the Docker runtime installed—be it a developer's laptop or a public cloud instance—can run a Docker container.

Docker shines for microservices architecture

Lightweight, portable, and self-contained, Docker containers make it easier to build software along forward-thinking lines, so that you're not trying to solve tomorrow's problems with yesterday's development methods.

One of the software patterns containers make easier is microservices, where applications are constituted from many loosely coupled components. By decomposing traditional, "monolithic" applications into separate services, microservices allow the different parts of a line-of-business app to be scaled, modified, and serviced separately—by separate teams and on separate timelines, if that suits the needs of the business.

Containers aren't required to implement microservices, but they are perfectly suited to the microservices approach and to agile development processes generally.

Installing Docker on Centos :

#yum install -y yum-utils **(Installing yum-utils package, This package is responsible for #yum-config-manager utility. Using this utility we can manage the repo's)**

#yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo
(Configuring Docker Repository)

#yum-config-manager --enable docker-ce-nightly **(Enabling Docker Repository)**

#yum install docker-ce docker-ce-cli containerd.io -y **(Installing Docker related packages in non-interactive mode)**

#which docker **(Verifying the docker installed location)**

#docker --version **(Displays Docker Version)**

#systemctl status docker **(Verifying Docker Service)**

#systemctl start docker **(Starting Docker Service)**

#systemctl enable docker **(Enabling Docker service to start automatically from the boot time)**

#systemctl status docker **(Ensure docker service is active and Enabled)**

#docker info **(Displays more detailed information about Docker host)**

#docker login **(Login to www.hub.docker.com)**

#docker images **(Lists all the images from the local cache)**

Pulling the Image :

Go to www.hub.docker.com → Repositories → Explore Repositories → Search → ubuntu → Select Ubuntu → Details →docker pull ubuntu (Copy)

#docker pull ubuntu **(Execute this command on docker host to pull ubuntu image to local cache)**

#docker images

#docker images –q **(Lists docker images with Image IDS)**

#docker rmi   < Image ID > **(Deleting a specific Image)**

#docker ps **(Lists all the active containers)**

#docker ps –a **(Lists all the active & Inactive Containers)**

#docker run –it ubuntu **(Creating ubuntu container with interactive Shell)**

#docker start <container Name / Container ID > **(To start a Container)**

#docker stop <Container Name / Container ID > (**To stop a container)**

#docker attach < Containername / Container ID> **(Brings the container from background to Foreground)**

#docker system prune –a **(This command removes all the stopped containers, Deletes all the networks not associated to any container, Deletes all the dangling Images)**

#docker pull ubuntu:18.04 **(Downloading docker image of a specific tag)**

#docker images
#docker rm < Container Name / Container ID > **(Deleting Container)**

Networking overview :

One of the reasons Docker containers and services are so powerful is that you can connect them together, or connect them to non-Docker workloads. Docker containers and services do not even need to be aware that they are deployed on Docker, or whether their peers are also Docker workloads or not. Whether your Docker hosts run Linux, Windows, or a mix of the two, you can use Docker to manage them in a platform-agnostic way.

This topic defines some basic Docker networking concepts and prepares you to design and deploy your applications to take full advantage of these capabilities.
Network drivers
Docker's networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality:

bridge:

The default network driver. If you don't specify a driver, this is the type of network you are creating. Bridge networks are usually used when your applications run in standalone containers that need to communicate. See bridge networks.

host:

For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly. See use the host network.

overlay:

Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons. This strategy removes the need to do OS-level routing between these containers. See overlay networks.

macvlan:

Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack. See Macvlan networks.

none:

For this container, disable all networking. Usually used in conjunction with a custom network driver. none is not available for swarm services. See disable container networking.

Network plugins:

You can install and use third-party network plugins with Docker. These plugins are available from Docker Hub or from third-party vendors. See the vendor's documentation for installing and using a given network plugin.

Network driver summary :

User-defined bridge networks are best when you need multiple containers to communicate on the same Docker host.

Host networks are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.

Overlay networks are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.

Macvlan networks are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.

Third-party network plugins allow you to integrate Docker with specialized network stacks.

Differences between user-defined bridges and the default bridge :

User-defined bridges provide automatic DNS resolution between containers.

Containers on the default bridge network can only access each other by IP addresses, unless you use the --link option, which is considered legacy. On a user-defined bridge network, containers can resolve each other by name or alias.

Imagine an application with a web front-end and a database back-end. If you call your containers web and db, the web container can connect to the db container at db, no matter which Docker host the application stack is running on.

If you run the same application stack on the default bridge network, you need to manually create links between the containers (using the legacy --link flag). These links need to be created in both directions, so you can see this gets complex with more than two containers which need to communicate. Alternatively, you can manipulate the /etc/hosts files within the containers, but this creates problems that are difficult to debug.

User-defined bridges provide better isolation.

All containers without a --network specified, are attached to the default bridge network. This can be a risk, as unrelated stacks/services/containers are then able to communicate.

Using a user-defined network provides a scoped network in which only containers attached to that network are able to communicate.

Containers can be attached and detached from user-defined networks on the fly.

During a container's lifetime, you can connect or disconnect it from user-defined networks on the fly. To remove a container from the default bridge network, you need to stop the container and recreate it with different network options.

Each user-defined network creates a configurable bridge.

If your containers use the default bridge network, you can configure it, but all the containers use the same settings, such as MTU and iptables rules. In addition, configuring the default bridge network happens outside of Docker itself, and requires a restart of Docker.

User-defined bridge networks are created and configured using docker network create. If different groups of applications have different network requirements, you can configure each user-defined bridge separately, as you create it.

Linked containers on the default bridge network share environment variables.

Originally, the only way to share environment variables between two containers was to link them using the --link flag. This type of variable sharing is not possible with user-defined networks. However, there are superior ways to share environment variables. A few ideas:

- Multiple containers can mount a file or directory containing the shared information, using a Docker volume.
- Multiple containers can be started together using docker-compose and the compose file can define the shared variables.
- You can use swarm services instead of standalone containers, and take advantage of shared secrets and configs.

Containers connected to the same user-defined bridge network effectively expose all ports to each other. For a port to be accessible to containers or non-Docker hosts on different networks, that port must be published using the -p or --publish flag.

#docker network ls **(Lists all the available Networks on Docker Host)**

#docker network create --driver bridge --subnet 172.18.0.0/16 bridge1 **(Creating new user/custom defined bridge network)**

#docker network ls

#docker network inspect <network-Name> **(Inspecting a specific network)**

Creating Container with Specific Network :
#docker run -d --name web1 --network bridge1 nginx

#docker network inspect bridge

#docker network connect bridge1 web9 **(Connecting Existing container to another new Network , Assume we have container web9)**

#docker network inspect bridge1
#docker container inspect web9

Disconnecting Container from the Network :
#docker network disconnect <N/W Name> web9
#docker container inspect web9

Connecting Container from the Network :
#docker network connect <N/W Name> web9
#docker container inspect web9

Deleting Network :
#docker network rm bridge1

Removing all the unused Networks :
#docker network prune
#docker network ls

The Docker Engine

First, let us look take a look at Docker Engine and its components so we have a basic idea of how the system works. Docker Engine allows you to develop, assemble, ship, and run applications using the following components:

Docker Daemon :

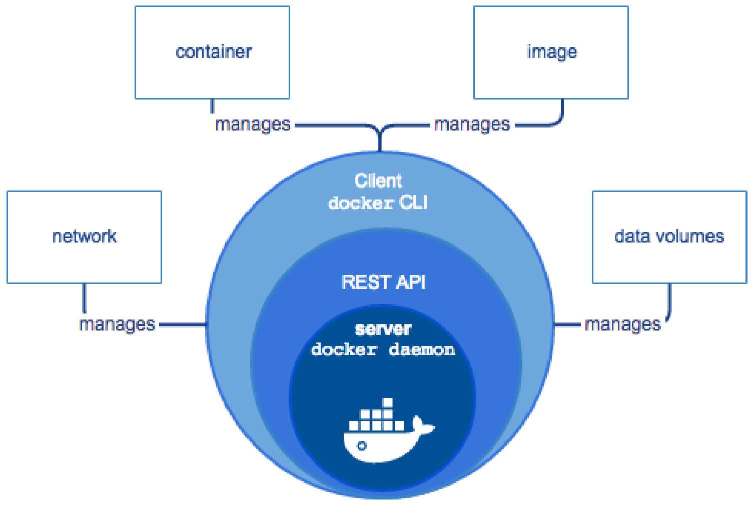
A persistent background process that manages Docker images, containers, networks, and storage volumes. The Docker daemon constantly listens for Docker API requests and processes them.

Docker Engine REST API :

An API used by applications to interact with the Docker daemon; it can be accessed by an HTTP client.

Docker CLI :

A command line interface client for interacting with the Docker daemon. It greatly simplifies how you manage container instances and is one of the key reasons why developers love using Docker.



We will see how the different components of the Docker Engine are used, let us dive a little deeper into the architecture.

Implementation
Docker is available for implementation across a wide range of platforms:

Desktop :

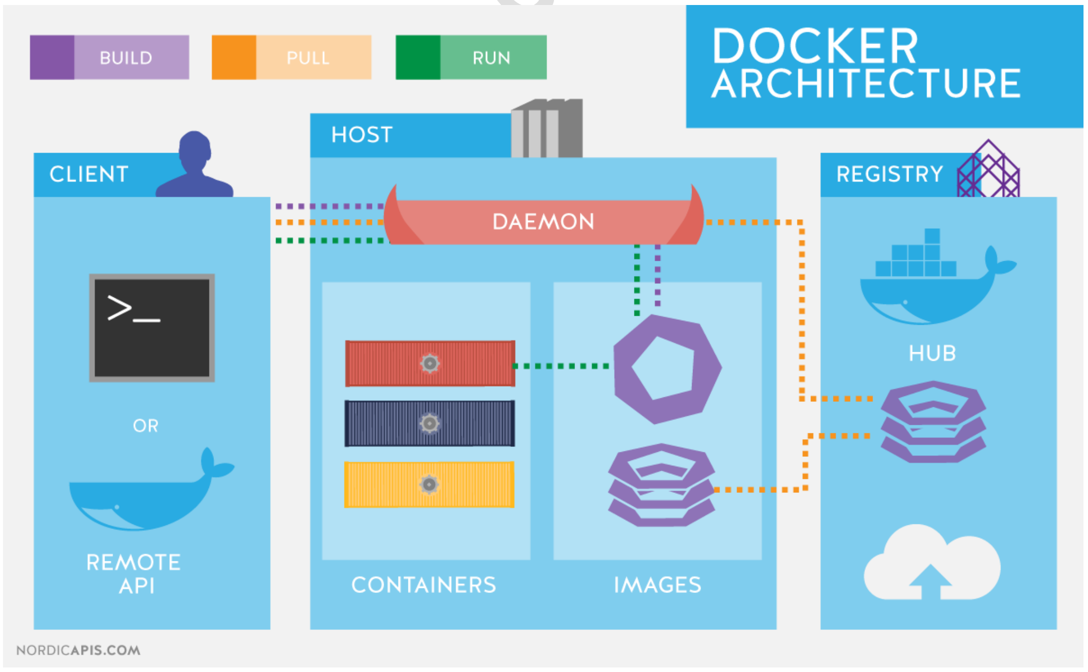
Mac OS, Windows 10.
Server :

Various Linux distributions and Windows Server 2016.

Cloud :
Amazon Web Services, Google Compute Platform, Microsoft Azure, IBM Cloud, and more.

Docker Architecture :

The Docker architecture uses a client-server model and comprises of the Docker Client, Docker Host, Network and Storage components, and the Docker Registry/Hub. Let's look at each of these in some detail.



Docker Client :

The Docker client enables users to interact with Docker. The Docker client can reside on the same host as the daemon or connect to a daemon on a remote host. A docker client can communicate with more than one daemon. The Docker client provides a command line interface (CLI) that allows you to issue build, run, and stop application commands to a Docker daemon.

The main purpose of the Docker Client is to provide a means to direct the pull of images from a registry and to have it run on a Docker host. Common commands issued by a client are:

```
docker build
docker pull
docker run
```

DockerHost :

The Docker host provides a complete environment to execute and run applications. It comprises of the Docker daemon, Images, Containers, Networks, and Storage. As previously mentioned, the daemon is responsible for all container-related actions and receives commands via the CLI or the REST API. It can also communicate with other daemons to manage its services. The Docker daemon pulls and builds container images as requested by the client. Once it pulls a requested image, it builds a working model for the container by utilizing a set of instructions known as a build file. The build file can also include instructions for the daemon to pre-load other components prior to running the container, or instructions to be sent to the local command line once the container is built.

Docker Objects :

Various objects are used in the assembling of your application. The main requisite Docker objects are:

Images :

Images are a read-only binary template used to build containers. Images also contain metadata that describe the container's capabilities and needs. Images are used to store and ship applications. An image can be used on its own to build a container or customized to add additional elements to extend the current configuration. Container images can be shared across teams within an enterprise using a private container registry, or shared with the world using a public registry like Docker Hub. Images are a core part of the Docker experience as they enable collaboration between developers in a way that was not possible before.

Containers :

Containers are encapsulated environments in which you run applications. The container is defined by the image and any additional configuration options provided on starting the container, including and not limited to the network connections and storage options. Containers only have access to resources that are defined in the image, unless additional access is defined when building the image into a container. You can also create a new image based on the current state of a container. Since containers are much smaller than VMs, they can be spun up in a matter of seconds, and result in much better server density.

#docker login **(Login to www.hub.docker.com)**

#docker stats **(Displays information about Running Containers like CPU, Memory Utilization and along with Network / Disk IO statistics)**

#docker system df **(Displays disk usage of Docker)**

#docker system prune –a **(This command removes all the stopped containers, Deletes all the networks not associated to any container, Deletes all the dangling Images)**

#docker pull ubuntu:18.04 **(Downloading docker image of a specific tag)**

#docker images

#docker run –name myubuntu –it ubuntu bash **(Creating container with custom name)**

#docker inspect <image name / Image ID> **(Displays detailed information of an image)**

#docker stop <Contianer Name / Container ID> **(Stopping a container)**

#docker rm < Container Name / Container ID > **(Deleting Container)**

Configuring REST API on Docker Server to manage it from Remote Client :

```
#vim /usr/lib/systemd/system/docker.service
ExecStart=/usr/bin/dockerd -H fd:// -H=tcp://0.0.0.0:8888
:wq!

#systemctl daemon-reload
#systemctl restart docker

#yum install httpd -y
#systemctl start httpd
#systemctl enable httpd
#systemctl status httpd

#firewall-cmd --add-service=http --permanent
#firewall-cmd --add-port=8888/tcp --permanent
#firewall-cmd --reload
#firewall-cmd --list-all
```

Go to Any CLient(Linux) :

#curl http://ip-of-docker-server://8888/imagesg/json (Displays all the available Images on Docker Server)
#curl http://ip-of-docker-server://8888/containers/json (Displays all the available COntainers)
#curl --data "t=5" http://ip-of-docker-server://8888/containers/container-id/stop (To stop a container)
#curl --data "t=5" http://ip-of-docker-server://8888/containers/container-id/start (To start a container)

Working with Containers :

#docker run --name ubuntu1 -it ubuntu **(Creating container from the ubuntu image with custom name)**

#docker stop <Container-ID / Container-Name> **(To stop a container)**
#docker start <Container-ID / Container-Name> **(To start a container)**
#docker pause <Container-ID / Container-Name> **(To pause a container)**
#docker unpause <Container-ID / Container-Name> **(To Unpause a container)**
#docker top <Container-ID / Container-Name> **(Displaying the top process running on Container)**

#docker stats <Container-ID / Container-Name> **(Displays the stats of Container like CPU, RAM, Network and Disk input & outpust statistics)**

#docker attach <Container-ID / Container-Name> **(Bringing the Container from backend to Frontend)**

#docker kill <Container-ID / Container-Name> **(Killing the Container Process)**

#docker ps **(Lists all the active Containers)**

#docker ps -a **(Lists all the active & inactive Containers)**

#docker rm <Container-ID / Container-Name> **(Deleting Container)**

#docker history <Docker-Image> **(Displays the history of a specific docker Image)**

Creating Container by exposing its Ports :

#docker container run --publish 9090:80 nginx **(Creating Container with system defined name from Docker image "NGINX" by exposing the local host port traffic port on 9090 and forwarding it to the executable running inside the container on port 80)**

#docker container run --publish --9091:80 --detach --name nginx2 nginx **(Creating a container with custom name and pushing to run in the background)**

#docker container ls -a **(Lists all the active & Inactive Containers)**

#docker container ls **(Lists all the active containers)**

#docker container logs <Container-ID / Container-Name> **(Displays the logs of a specific container)**

#docker container top <Container-ID / Container-Name> **(Displaying the top process running on Container)**

#docker container stop <Container-ID / COntainer-Name> **(Stopping a Container)**

#docker container rm <Container-ID /Container-Name> **(Deleting a Container)**

Docker File :

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile 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 an automated build that executes several command-line instructions in succession.

Creating file named Dockerfile :

```
#mkdir /images/DockerFiles
#touch /images/DOckerFiles/Dockerfile
#vim /images/DOckerFiles/Dockerfile
FROM centos:centos7
MAINTAINER Mahesh mahesh@xyz.com
RUN yum install java -y
RUN mkdir /opt/tomcat
WORKDIR /opt/tomcat
ADD https://dlcdn.apache.org/tomcat/tomcat-9/v9.0.73/bin/apache-tomcat-
9.0.73.tar.gz .
RUN tar -xvzf apache-tomcat-9.0.73.tar.gz
RUN mv apache-tomcat-9.0.73/* /opt/tomcat
ADD https://github.com/write4mahesh/javapp/blob/master/addressbook.war
/opt/tomcat/webapps/
EXPOSE 8080
CMD ["/opt/tomcat/bin/catalina.sh", "run"]
:wq!
```

Each instruction creates one layer:

FROM : creates a layer from the ubuntu:18.04 Docker image.
MAINTAINER : docker FIle Author Details
COPY : adds files from your Docker client's current directory.
RUN : builds your application with make.
CMD : specifies what command to run within the container.

#docker build /images/DockerFiles **(Creating docker image from the docker File with out specifying name & Tag)**

#docker build -t image1:1.0 /images/DockerFiles **(Creating docke>r image from docker file with required name and Tag)**

Running the Image :
#docker images
#docker run <imageid>

Image Layers :

When you pull a Docker image, you will notice that it is pulled as different layers. Also, when you create your own Docker image, several layers are created. we will try to get a better understanding of Docker layers.

A Docker image consists of several layers. Each layer corresponds to certain instructions in your Dockerfile. The following instructions create a layer: RUN, COPY, ADD. The other instructions will create intermediate layers and do not influence the size of your image.

#docker image ls
#docker history nginx **(Displays the layers of changes made in the Image)**

Image Tagging & Pushing to docker hub :
#docker pull nginx
#docker pull nginx:mainline
#docker image ls

NOTE :
It is actually already knew that based on the imageid, This image already exists in cache.Thats the reason in the output both images of nginx are showing the same Image ID bcz they are not really stored twice in the cache.

How to make new Labels :

#docker image tag nginx useridofdockerhub/nginx **(Since we have not given any tag it takes latest tag name)**

#docker images ls

Lets upload this image to docker hub :

```
#docker login
#docker image push useridofdockerhub/ninx
#docker image tag useridofdockerhub/nginx useridofdockerhub/nginx:testing
#docker image ls
#docker image push useridofdockerhub/nginx:testing
```

NOTE : **Go to www.hub.docker.com , Login as user and verify the pushed images in the repo.**

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.

Prerequisites:

Docker Compose relies on Docker Engine for any meaningful work, so make sure you have Docker Engine installed either locally or remote, depending on your setup.

On desktop systems like Docker Desktop for Mac and Windows, Docker Compose is included as part of those desktop installs.

On Linux systems, first install the Docker Engine for your OS as described on the Get Docker page, then come back here for instructions on installing Compose on Linux systems.

Run this command to download the current stable release of Docker Compose:

```
#curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

Apply executable permissions to the binary:

```
#chmod +x /usr/local/bin/docker-compose
```

Test the installation:

```
#docker-compose --version

#mkdir dockercomposefile
#touch dockercomposefile/docker-compose.yml
#vim dockercomposefile/docker-compose.yml
services :


  web :
    image : nginx
    ports :
    - 8080:80
  database :
    image : redis
:wq!
```

NOTE : We need to check the validity of compose file using the below command

```
#cd dockercomposefile
#docker-compose config
```

NOTE : Use the below link to understand the compatiable compose file versions for docker run time :

https://docs.docker.com/compose/compose-file/

```
#docker-compose up -d
```

up --> to run the yml file and create contianers & applications
-d --> Detatch Mode

```
#docker container ls

#docker-compose down

#docker-compose up -d
```

Scaling up Services :

```
#docker-compose up -d --scale database=5
#docker container ls
```

Scaling down Services :

```
#docker-compose up -d --scale database=1
#docker container ls

#docker-compose down
#docker container ls
```

Docker Volumes :

Docker volumes are a widely used and useful tool for ensuring data persistence while working in containers. Docker volumes are file systems mounted on Docker containers to preserve data generated by the running container.

The data doesn't persist when that container no longer exists, and it can be difficult to get the data out of the container if another process needs it.

A container's writable layer is tightly coupled to the host machine where the container is running. The data cannot be easily moveable somewhere else.

Writing into a container's writable layer requires a storage driver to manage the filesystem.

#docker volume ls (Lists all the available volumes)
#docker volume create myvol1 (Creating new volume)
#docker volume ls
#docker volume inspect myvol1 (Displaying info of a specific volume)
#docker volume rm myvol1 (Deleting volume)
#docker volume prune (Deleting all unused volumes) **Docker Swarm :**

Each node of a Docker Swarm is a Docker daemon, and all Docker daemons interact using the Docker API. Each container within the Swarm can be deployed and accessed by nodes of the same cluster.

Features of Docker Swarm :
Some of the most essential features of Docker Swarm are:

Decentralized access: Swarm makes it very easy for teams to access and manage the environment
High security: Any communication between the manager and client nodes within the Swarm is highly secure
Autoload balancing: There is autoload balancing within your environment, and you can script that into how you write out and structure the Swarm environment
High scalability: Load balancing converts the Swarm environment into a highly scalable infrastructure
Roll-back a task: Swarm allows you to roll back environments to previous safe environments

Swarm Mode Key Concepts :
Service and Tasks :

- Docker containers are launched using services.
- Services can be deployed in two different ways - global and replicated.
- Global services are responsible for monitoring containers that want to run on a Swarm node. In contrast, replicated services specify the number of identical tasks that a developer requires on the host machine.
- Services enable developers to scale their applications.
- Before deploying a service in Swarm, the developer should implement at least a single node.
- Services can be used and accessed by any node of the same cluster.
- A service is a description of a task, whereas a task performs the work.
- Docker helps a developer in creating services, which can start tasks. However, when a task is assigned to a node, the same task cannot be attributed to another node.

Node :

- A Swarm node is an instance of the Docker engine.
- It is possible to run multiple nodes on a single server. But in production deployments, nodes are distributed across various devices.

How Does Docker Swarm Work?
In Swarm, containers are launched using services. A service is a group of containers of the same image that enables the scaling of applications. Before you can deploy a service in Docker Swarm, you must have at least one node deployed.

There are two types of nodes in Docker Swarm:

Manager node : Maintains cluster management tasks
Worker node : Receives and executes tasks from the manager node

Pre-Requisites :
1)Docker 1.13 or Higher
2)DOcker Machine need to be installed ondocker host

Steps to follow :
1)Ensure docker is installed on Based OS
2)Ensure Docker Machine utility is installed
3)Ensure Virtual box hypervisor is installed
4)Create one machine as worker node and others as worker nodes

Installing Docker machine on DOcker host :
#base=https://github.com/docker/machine/releases/download/v0.16.0 && curl -L $base/docker-machine-$(uname -s)-$(uname -m) >/tmp/docker-machine && sudo mv /tmp/docker-machine /usr/local/bin/docker-machine && chmod +x /usr/local/bin/docker-machine
#docker-machine -v **(Displays the docker machine)**

Installing Virtual Box on Base OS :

#yum install wget vim –y **(Installing wget & Vim softwares)**
#wget https://download.virtualbox.org/virtualbox/rpm/rhel/virtualbox.repo -P /etc/yum.repos.d/ **(Configuring Virtual Box Repository)**

#yum install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.**rpm (Configuring EPEL Repository)**

#yum update

#yum install binutils kernel-devel kernel-headers libgomp make patch gcc glibc-headers glibc-devel dkms -y

#Yum install VirtualBox-6.1 **(Installing Virtual Box)**
#which virtualbox
#reboot

#/sbin/vboxconfig **(Stopping & STarting the Virtual Box Services)**
#systemctl status vboxconfig **(Verify Virtual box Service is Running)**

Creating Docker Machines :

```
#docker-machine create --driver virtualbox  dm1
```
#docker-machine ls **(Lists all the docker machines created)**
#docker-machine ip <machinename> **(Displays the IP of docker machine)**

```
                    dm1
#docker-machine create --driver virtualbox  dw1
#docker-machine create --driver virtualbox  dw2
#docker-machine create --driver virtualbox  dm3
#docker-machine ls
```

Connecting to Docker different terminals :

```
#docker-machine ssh dm1
#docker-machine ssh dw1
#docker-machine ssh dw2
#docker-machine ssh dw3
```