What is Docker?

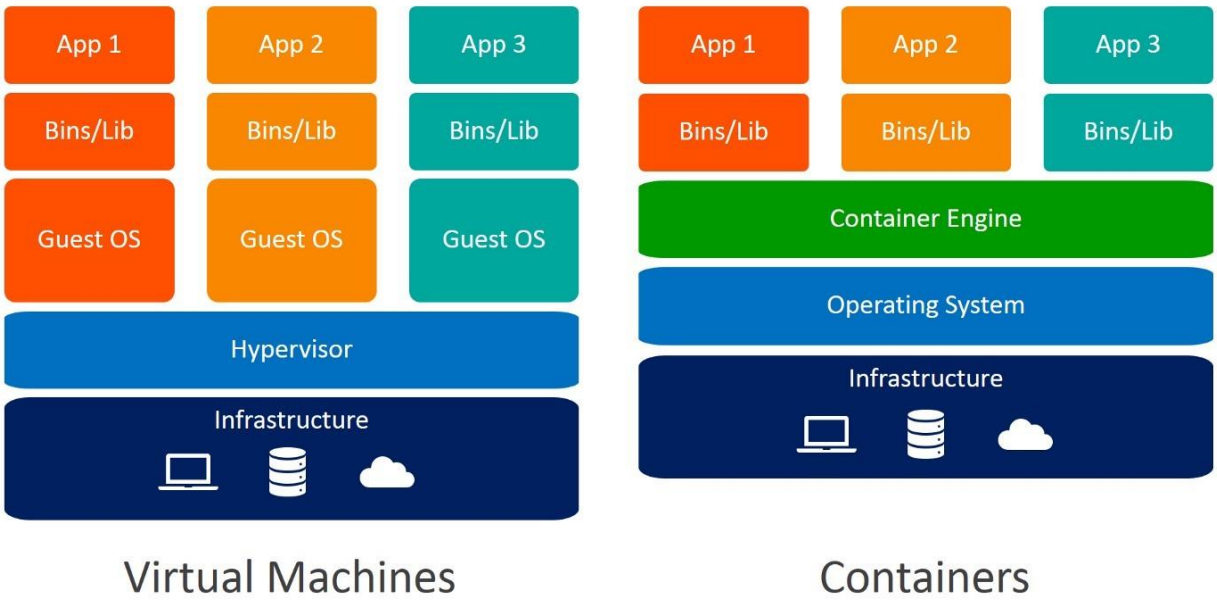
Docker is an open-source containerization platform. It enables developers to package applications into containers—standardized executable components combining application source code with the operating system (OS) libraries and dependencies required to run that code in any environment.

Another Definition:

Docker is a container platform that allows you to build, test and deploy applications quickly. A developer defines all the applications and its dependencies in a Docker file which is then used to build Docker images that defines a Docker container. Doing this ensures that your application will run in any environment.

How does it work?

Containerization vs Virtualization



Virtualization:

An application on a VM requires a guest OS and thus an underlying hypervisor to run. Hypervisor is used to create multiple machines on a host operating system and it manages virtual machines. These virtual machines have their own operating system and do not use the host's operating system. They have some space allocated. Virtualization software: VMWare, Oracle Virtual Box, QEMU, Microsoft Hyper-V

Containerization:

Containerization is an efficient method for deploying applications. A container encapsulates an application with its own operating environment. It can be placed on any host machine without special configuration, removing the issue of dependencies.

Differences:

- VM is hardware virtualization, whereas containerization is OS virtualization.
- Virtualization is the creation of a virtual version of something such as an operating system, server or a storage device or network resources.
- Containerization is a lightweight approach to virtualization.

Previously Issues:

Compatibility with underlying OS is an issue.
Compatibility with library and OS
Dedicated space required
Dependency
Long set up time
Different Dev/Test/Prod environment
Migration of code is an issue.

Docker container solve all these issues, because it is isolated environment have own dependencies, libraries and OS.

What is container?

Containers are completely isolated environment, they have their own network, process and services.
Containers are often referred to as “lightweight,” meaning they share the machine’s operating system kernel and do not require the overhead of associating an operating system within each application.
Containers are inherently smaller in capacity than a VM and require less start-up time, allowing far more containers to run on the same compute capacity as a single VM.

Install Docker Engine on Ubuntu

To install the latest version of Docker on Linux from the “test” channel, run:

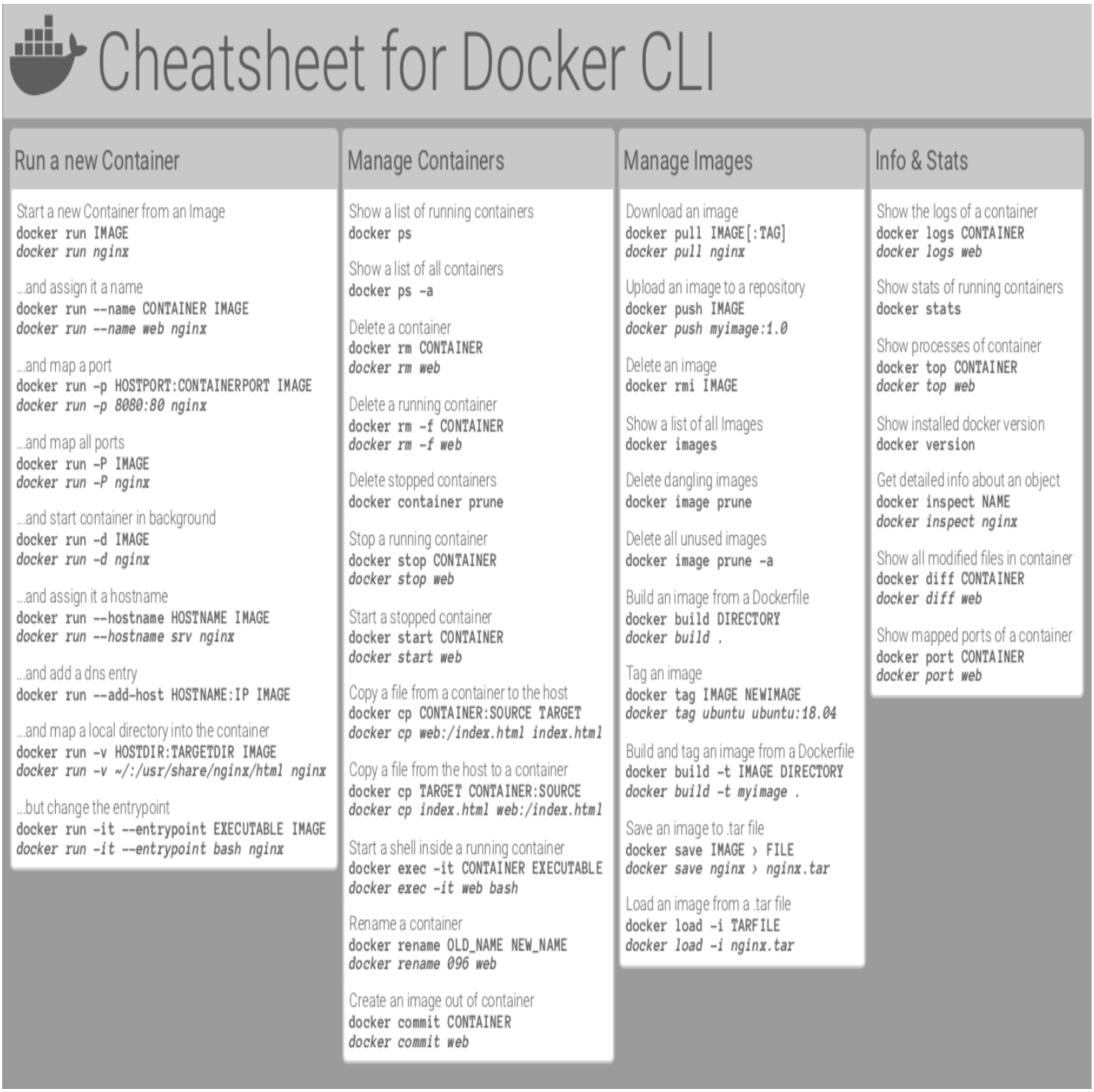
```
$ curl -fsSL https://test.docker.com -o test-docker.sh
$ sudo sh test-docker.sh
```

First Test Docker Image:

Run the command and test docker image.

```
sudo docker run docker/whalesay cowsay Hello-Iftikhar
```

Docker Commands Cheat Sheet



What is Container?
A container is a runnable instance of an image. This is where your application is running. You can manage containers using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state. If we delete a container the data will be lost! Because when the container went down and we brought it back up, the last layer got created again as a new layer. This helps in development if you don't want to store record for each test. To be persistent, use volumes to store data.

How to create my own image?

These are the requirements:

1. OS - Ubuntu
2. Update apt repo
3. Install dependencies using apt
4. Copy source code to /opt folder
5. Run the web server using "flask" command

Docker File:

Docker file consist of **Instruction** and **Argument**

```
FROM Ubuntu

RUN apt-get update
RUN apt-get install python

RUN pip install flask
RUN pip install flask-mysql

COPY ./opt/source-code
ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run
```

For Image build we used this command:

```
docker build Dockerfile -t my-Custom-app

docker push my-Custom-app
```

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.

Network Drivers:

`bridge`: The default network driver. Bridge networks are usually used when your applications run in standalone containers that need to communicate.

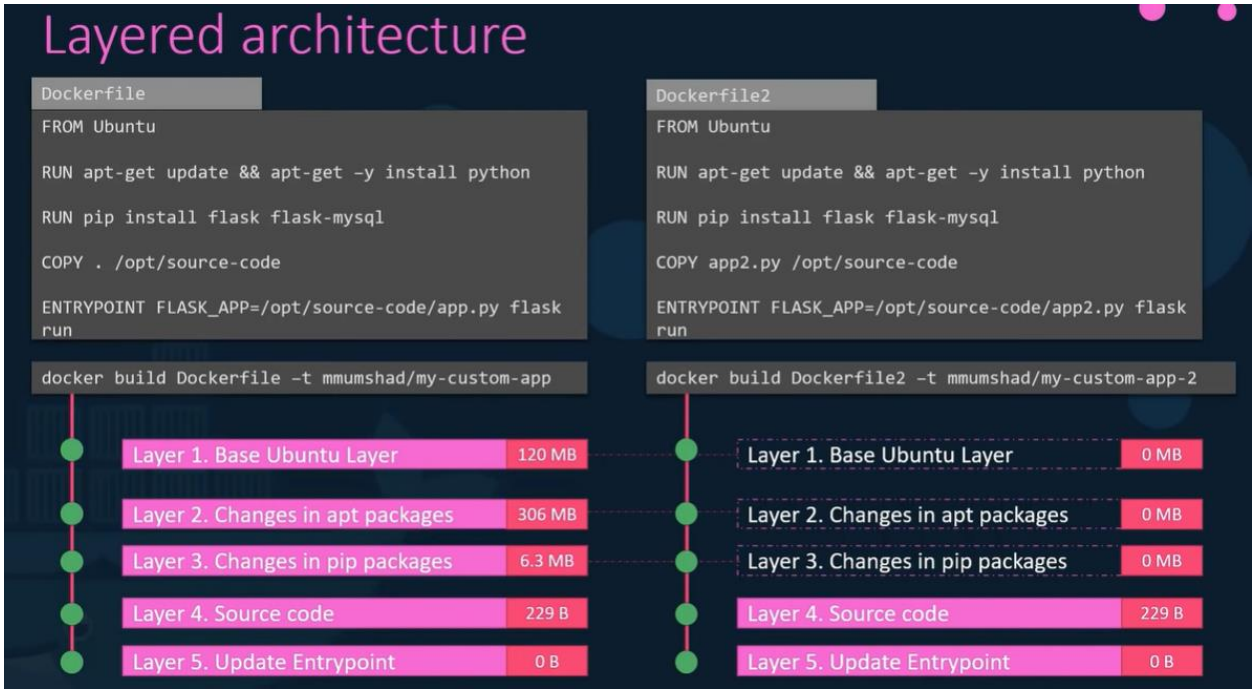
`host`: For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly.

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

What is Docker Layer Architecture?

When docker build images, it builds in the form of layer architecture.

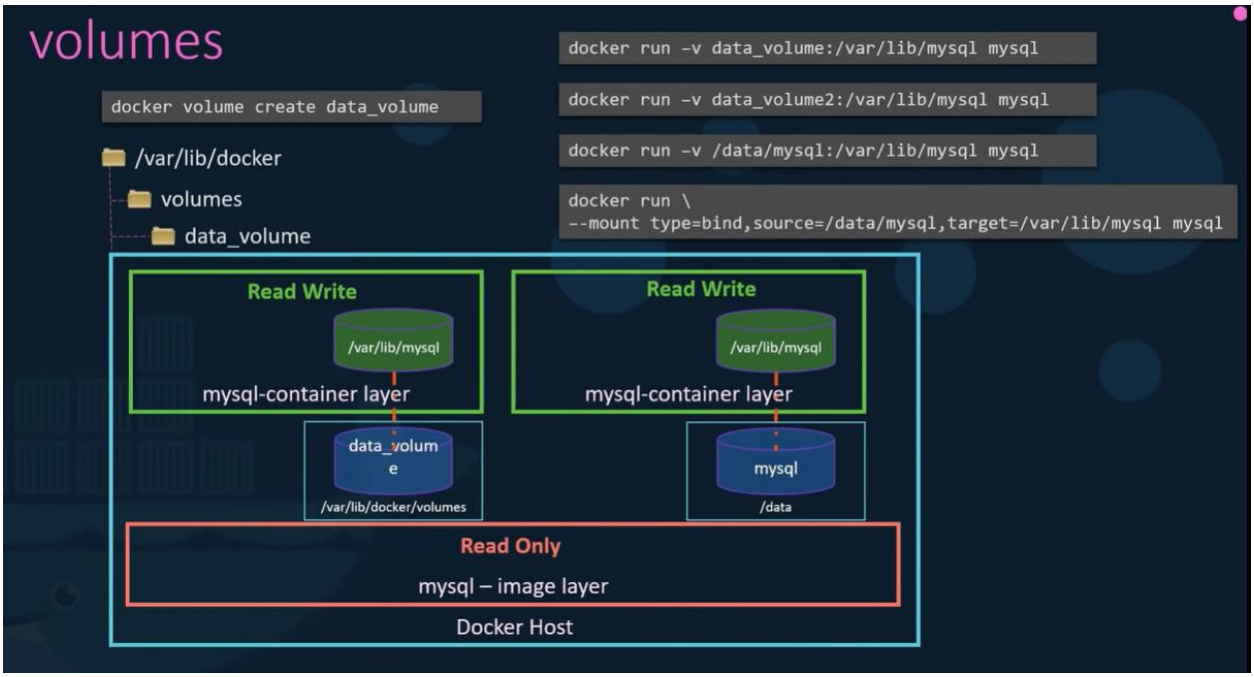
- A Docker image consists of several layers.
- Each layer corresponds to certain instructions in your Dockerfile.
- The following instructions create a layer: RUN, COPY, ADD.
- The other instructions will create intermediate layers and do not influence the size of your image.



Storage Overview:
Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

- Volumes are easier to back up or migrate than bind mounts.
- You can manage volumes using Docker CLI commands or the Docker API.
- Volumes work on both Linux and Windows containers.
- Volumes can be more safely shared among multiple containers.
- Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
- New volumes can have their content pre-populated by a container.
- Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts.

In addition, volumes are often a better choice than persisting data in a container's writable layer, because a volume does not increase the size of the containers using it, and the volume's contents exist outside the lifecycle of a given container.

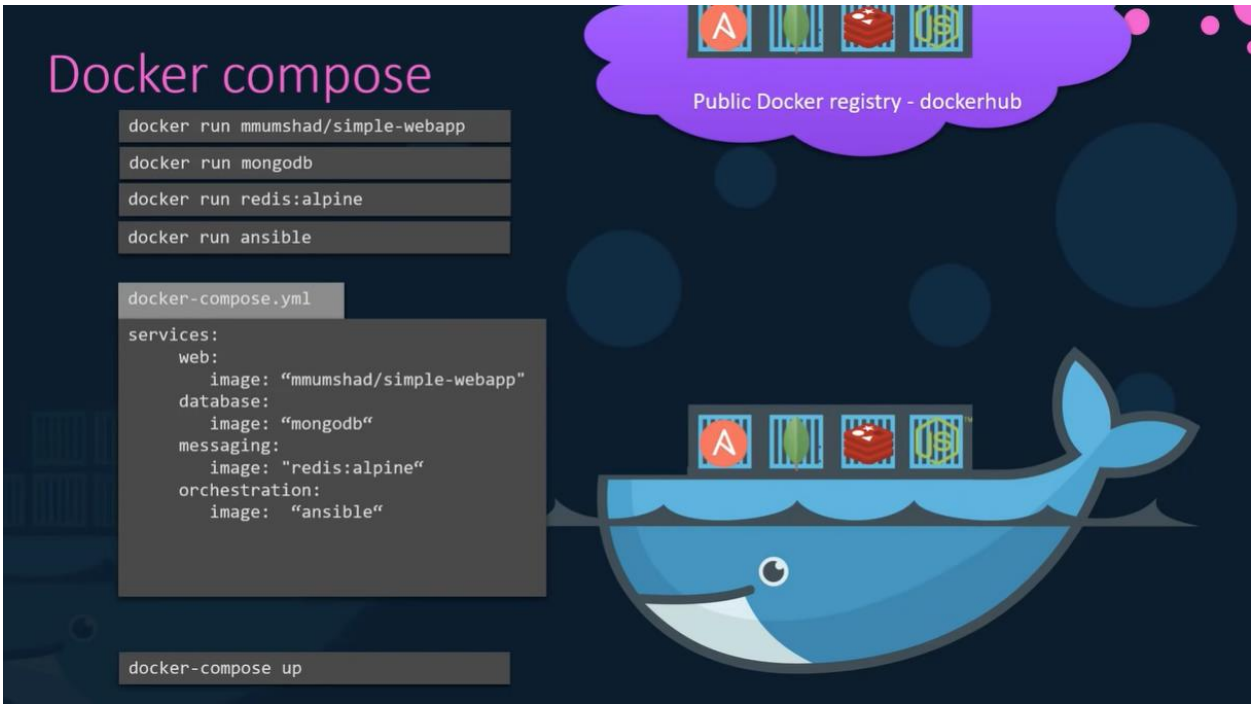


Overview of Docker Compose:

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.

Using Compose is basically a three-step process:

1. Define your app's environment with a `Dockerfile` so it can be reproduced anywhere.
2. Define the services that make up your app in `docker-compose.yml` so they can be run together in an isolated environment.
3. Run `docker compose up` and the Docker compose command starts and runs your entire app. You can alternatively run `docker-compose up` using the docker-compose binary.



Docker daemon

It listens to the API requests being made through the Docker client and manages Docker objects such as images, containers, networks, and volumes.

Docker client

This is what you use to interact with Docker. When you run a command using docker, the client sends the command to the daemon, which carries them out. The Docker client can communicate with more than one daemon.

Docker registries

This is where Docker images are stored. Docker Hub is a public registry that anyone can use. When you pull an image, Docker by default looks for it in the public registry and saves the image on your local system on DOCKER_HOST. You can also store images on your local machine or push them to the public registry.

How to achieve High Availability using Docker – Docker Swarm

It creates multiple containers on multiple hosts.
It does not use any file like YAML etc. to manage but it manages different docker hosts in a cluster.

Docker Swarm can reschedule containers on node failures. Swarm node has a backup folder which we can use to restore the data onto a new Swarm.

We have, mainly two types of nodes in docker swarm:

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

You can set up commands and services to be either global or replicated: a global service will run on every Swarm node, and on a replicated service, the manager node distributes tasks to worker nodes.

Docker Swarm requires two hosts, which can either be Virtual Machine or AWS EC2.

1. Update Software Repositories:
   ```
   sudo apt-get update
   ```

2. Uninstall the older docker:
   ```
   sudo apt-get remove docker docker-engine docker.io
   ```

3. Install the new docker:
   ```
   sudo apt install docker.io
   ```

4. Setup the docker:
   ```
   sudo systemctl start docker
   ```

```
sudo systemctl enable docker
```

5. verify the docker version:
```
sudo docker –version
```

6. Run a container:

```
sudo docker pull mysql
sudo docker run -d -p0.0.0.0:80:80 mysql:latest
```

Now, Docker pulls the latest **MySQL** image from the docker hub.
List down all the available Docker containers on your machine by using the following command:

```
sudo docker ps –a
```

Now we will create the Swarm:

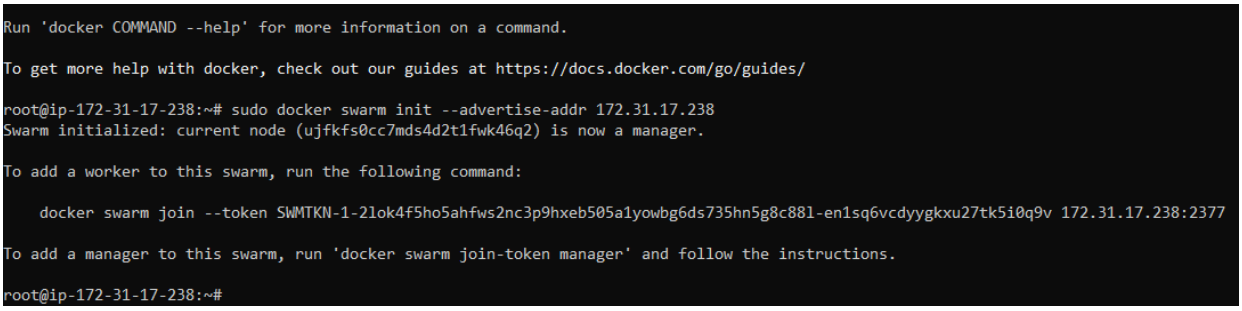
Create a cluster with the IP address of the manager node.

```
sudo Docker swarm init --advertise-addr 192.168.2.151
```

If it's not working fine, install the docker swarm by:

```
sudo apt-get install docker swarm
```

Ok that’s done!



Now, add worker node by copying the command of the “`swarm init`” and paste the output onto the worker node.

The next step is to join our two worker nodes to the Swarm cluster by using the token which was generated earlier.

```
docker swarm join --token <token-id> <ip:port>
```

Note: The docker swarm is a collection of one or more machines (**physical or virtual, called nodes**) that can run your **containers** as services. Nodes in the swarm can be **managers or workers**. Only on **manager** nodes can you see/modify the **swarm status**. **Worker** nodes only run **containers**. In order to run a container in the swarm you must create a service; that service will have zero or more containers depending on the scale that you set for the service.

To **create** a swarm, you run the docker **swarm init** on the machine that will be a manager node. Then, on the other machines that you own you run the **docker swarm join** command in order to add them to the swarm. You cannot add to the swarm a machine that already is on the swarm. In your case, you try to add to the swarm the manager that created the swarm.

When you initiate a swarm (with **docker swarm init**), the machine from that you initiated the swarm is already connected to the swarm, you don't need to do anything else to connect it to the swarm.

After you initiate the swarm, you may (and should) add other machines as managers or workers.

To leave the swarm enter:

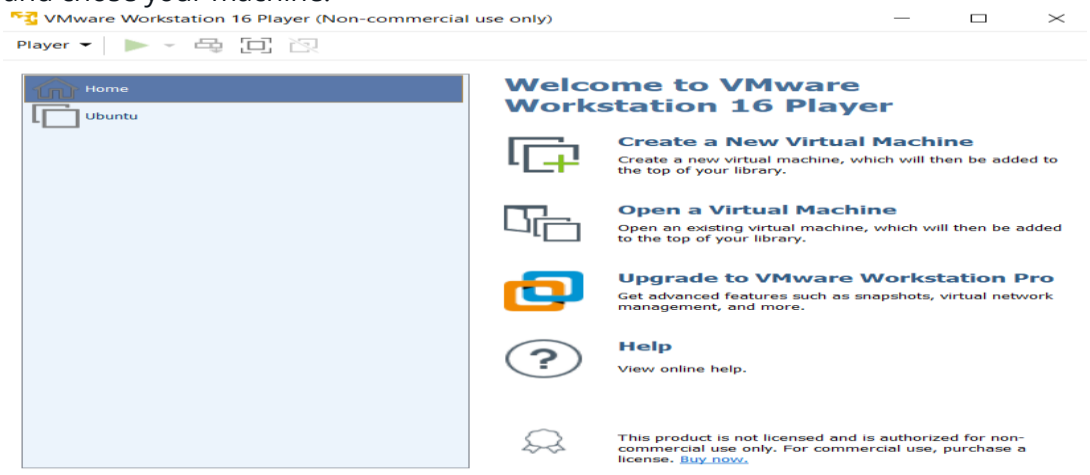
```
docker swarm leave –force
```

Error to solve: Problem will come:

How to install docker-machine:

```
curl -L https://github.com/docker/machine/releases/download/v0.4.0/docker-machine_linux-amd64 /usr/local/bin/docker-machine

chmod +x /usr/local/bin/docker-machine

docker-machine –version

docker-machine create –driver virtualbox manager1
```

Here you can see a problem that **VT-X/AMD-v** is not enabled: So, you will open the VMWare and chose your machine:



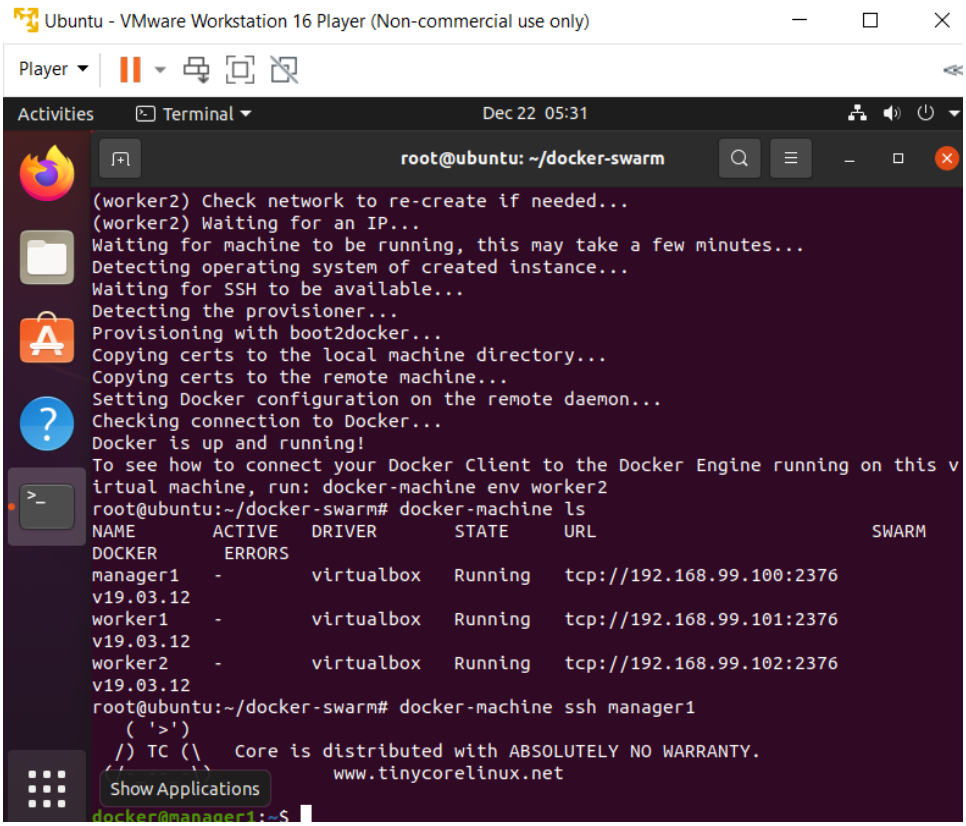
Then in the processors section, enable this.

Make **two** more worker machines here again with the same command:

- worker1
- worker2

Check with the command: `docker-machine ls`

Now, to enter in the machine use command: `docker-machine ssh manager1`



Now, let's initialize the docker swarm;

Enter into the manager machine and write command:

`docker-machine ssh manager1`

`docker swarm init –advertise-addr <manager_IP>`

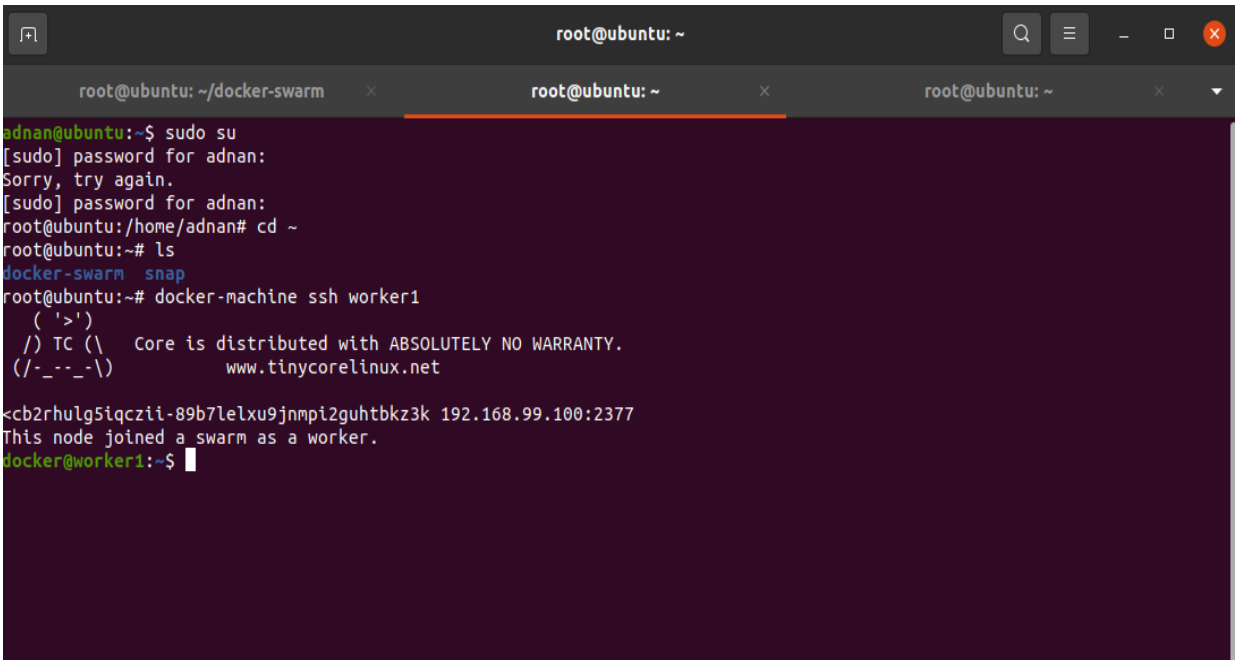
```
docker@manager1:~$ docker swarm init --advertise-addr 192.168.99.100
Swarm initialized: current node (thbazh4jd6inxecbrfb1sp313) is now a manager.

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

    docker swarm join --token SWMTKN-1-4cnz3aztfkiqacs9cyl92uvqtjimemefjatcb2rhulg5iqczii-89b
7lelxu9jnmpi2guhtbkz3k 192.168.99.100:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instruct
ions.

docker@manager1:~$
```

Now go to the other machines by command:

Docker-machine ssh worker1 && worker2

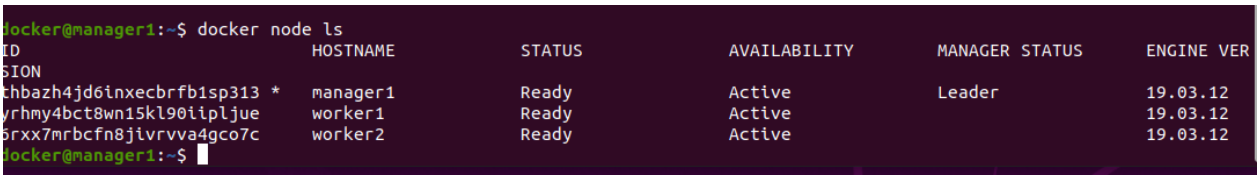
Enter the command generated by manager node to other two machines (worker1 & worker2)

```
docker swarm join --token SWMTKN-1-
4cnz3aztfkiqacs9cyl92uvqtjimemefjatcb2rhulg5iqczii-89b7lelxu9jnmpi2guhtbkz3k
192.168.99.100:2377
```



Now go to the manager machine, and enter the command to show the nodes created;

`docker node ls` (Will work on manager machine)



We have three types of availabilities here:

Active: can assign task to other nodes

Pause: can't assign task to other nodes, but can run existing task

Drain: can't assign task to other nodes, cannot run existing task, and schedules that to other available nodes.

Now, there are three manager statuses here:

Leader: will lead the nodes.

Reachable: It will be leader, if leader dies.

Unavailable: this node is no more available.

Extra commands:

`docker swarm leave` // leave by – force

`docker swarm join-token manager/worker` // IF we want to make another manager (Reachable)

`docker node –help` // to see more commands related to this

So, all done here!

Further we are going to see, how we can use service in docker swarm.

- How to run services in docker swarm.
- How to scale up and scale down services.

`docker node inspect manager1` // to inspect about the node

`docker node promote worker1` // To make it manager

`docker node demote worker1` // To make it again worker

Inside any of these machines, we can now use the docker; let's see by entering the command:

`docker info`

Let's see the services now, we have two types of services there:

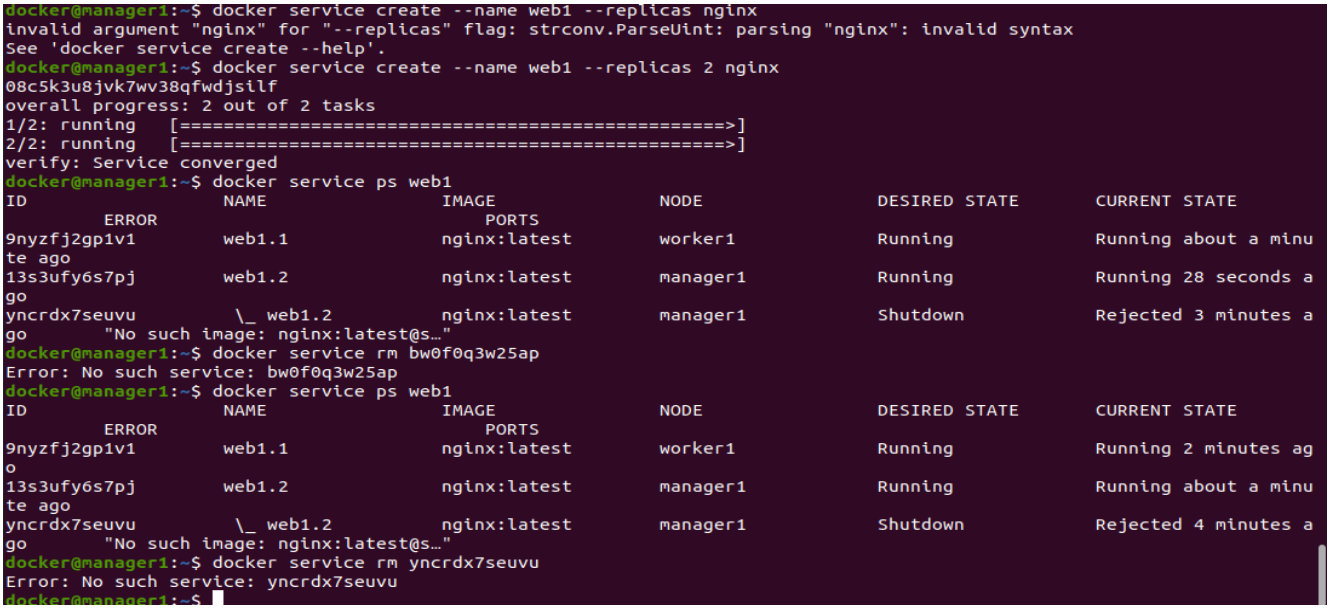
- **Replicated**: You specify the number of identical tasks to run.
- **Global**: It runs one task on every node.

`docker service create –name web1 –replicas 2 nginx`

// here we are using replica service and it will run only two nodes

Now, check the service by:

`docker service ps web1`



`docker service scale web1=4` //means rather than 2 times, it will run 7 times total.

```
root@ubuntu: ~/docker-swarm
root@ubuntu: ~/docker-swarm   root@ubuntu: ~   root@ubuntu: ~
go
yncrdx7seuvu          \_ web1.2          nginx:latest        manager1            Shutdown            Rejected 3 minutes a
go       "No such image: nginx:latest@s…"
docker@manager1:~$ docker service rm bw0f0q3w25ap
Error: No such service: bw0f0q3w25ap
docker@manager1:~$ docker service ps web1
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE
          ERROR                              PORTS
9nyzfj2gp1v1        web1.1              nginx:latest        worker1             Running             Running 2 minutes ag
o
13s3ufy6s7pj        web1.2              nginx:latest        manager1            Running             Running about a minu
te ago
yncrdx7seuvu          \_ web1.2          nginx:latest        manager1            Shutdown            Rejected 4 minutes a
go       "No such image: nginx:latest@s…"
docker@manager1:~$ docker service rm yncrdx7seuvu
Error: No such service: yncrdx7seuvu
docker@manager1:~$ docker service scale web1=4
web1 scaled to 4
overall progress: 4 out of 4 tasks
1/4: running   [==================================================>]
2/4: running   [==================================================>]
3/4: running   [==================================================>]
4/4: running   [==================================================>]
verify: Service converged
docker@manager1:~$ docker service ps web1
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE
          ERROR                              PORTS
9nyzfj2gp1v1        web1.1              nginx:latest        worker1             Running             Running 12 minutes a
go
13s3ufy6s7pj        web1.2              nginx:latest        manager1            Running             Running 10 minutes a
go
yncrdx7seuvu          \_ web1.2          nginx:latest        manager1            Shutdown            Rejected 13 minutes
ago    "No such image: nginx:latest@s…"
gvsa878v62bm        web1.3              nginx:latest        worker2             Running             Running 44 seconds a
go
m4p6gc8b0j2h        web1.4              nginx:latest        worker2             Running             Running 44 seconds a
go
docker@manager1:~$
```

`docker service scale web1=0` // to remove all

Let's check if our manager node is not working or exit, if its working on the worker1 that's is reachable:

```
docker@manager1:~$ docker node promote worker1
Node worker1 promoted to a manager in the swarm.
docker@manager1:~$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VER
SION
thbazh4jd6inxecbrfb1sp313 *   manager1            Ready               Active              Leader              19.03.12
yrhmy4bct8wn15kl90iipljue     worker1             Ready               Active              Reachable           19.03.12
6rxx7mrbcfn8jivrvva4gco7c     worker2             Ready               Active                                  19.03.12
docker@manager1:~$ docker service ps web1
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE
          ERROR                              PORTS
9nyzfj2gp1v1        web1.1              nginx:latest        worker1             Running             Running 17 minutes a
go
13s3ufy6s7pj        web1.2              nginx:latest        manager1            Running             Running 16 minutes a
go
yncrdx7seuvu          \_ web1.2          nginx:latest        manager1            Shutdown            Rejected 19 minutes
ago    "No such image: nginx:latest@s…"
gvsa878v62bm        web1.3              nginx:latest        worker2             Running             Running 6 minutes ag
o
m4p6gc8b0j2h        web1.4              nginx:latest        worker2             Running             Running 6 minutes ag
o
docker@manager1:~$ exit
logout
root@ubuntu:~/docker-swarm#
```

Now, worker1 is promoted, lets exit from the manager1
And see if our worker1 is now leader and service is working finely;

```
docker@worker1:~$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VER
SION
thbazh4jd6inxecbrfb1sp313     manager1            Ready               Active              Leader              19.03.12
yrhmy4bct8wn15kl90iipljue *   worker1             Ready               Active              Reachable           19.03.12
6rxx7mrbcfn8jivrvva4gco7c     worker2             Ready               Active                                  19.03.12
docker@worker1:~$
```

So, it's working fine and now the worker1 is leading it.