Introduction

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

Docker provides tooling and a platform to manage the lifecycle of your containers:

- Develop your application and its supporting components using containers.
- The container becomes the unit for distributing and testing your application.
- When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

Virtualization

This is the process of running multiple OS's parallelly on a single piece of h/w. Here we have h/w (bare metal) on top of which we have host os and on the host os we install an application called as hypervisor. On the hypervisor we can run any of the OS's as guest OS.

The disadvantage of this approach is that applications running on the guest OS have to pass through a number of layers to access the H/W resources.

Containerization

Here we have bare metal on top of which we install the host Os and on the host OS we install an application called Docker Engine. On the docker engine we can run any application in the form of containers. Docker is a technology for creating these containers. Docker achieves what is commonly called "process isolation".

I.e. all the applications(processes) have some dependency on a specific OS. This dependency is removed by docker and we can run them on any OS as containers if we have Docker engine installed.

These containers pass through less layers to access the h/w resources. Also, organizations need not spend money on purchasing licenses of different OS's to maintain various applications.

Docker can be used at the stages of S/W development life cycle.

Build---->Ship--->Run

Docker comes in 2 flavors.

Docker CE (Community Edition)

Docker EE (Enterprise Edition)

Setup of Docker on Windows

1) Download docker desktop from.

   https://www.docker.com/products/docker-desktop

2) Install it.

3) Once docker is installed we can use Power shell to run the docker commands.

Setup of Docker on an Ubuntu Linux Machine

1) Create Ubuntu linux instances on GCP cloud.

2) Connect to that instance using gitbash.

3 To install docker.

a) Open https://get.docker.com and Copy paste the below two commands.

```
curl -fsSL https://get.docker.com -o get-docker.sh sh get-
docker.sh
```

Docker Components & Architecture

Images and Containers

A Docker image is a combination of bins/libs that are necessary for a s/w application to work. A Docker container is a running instance of a docker image. Any number of containers can be created from one docker image.

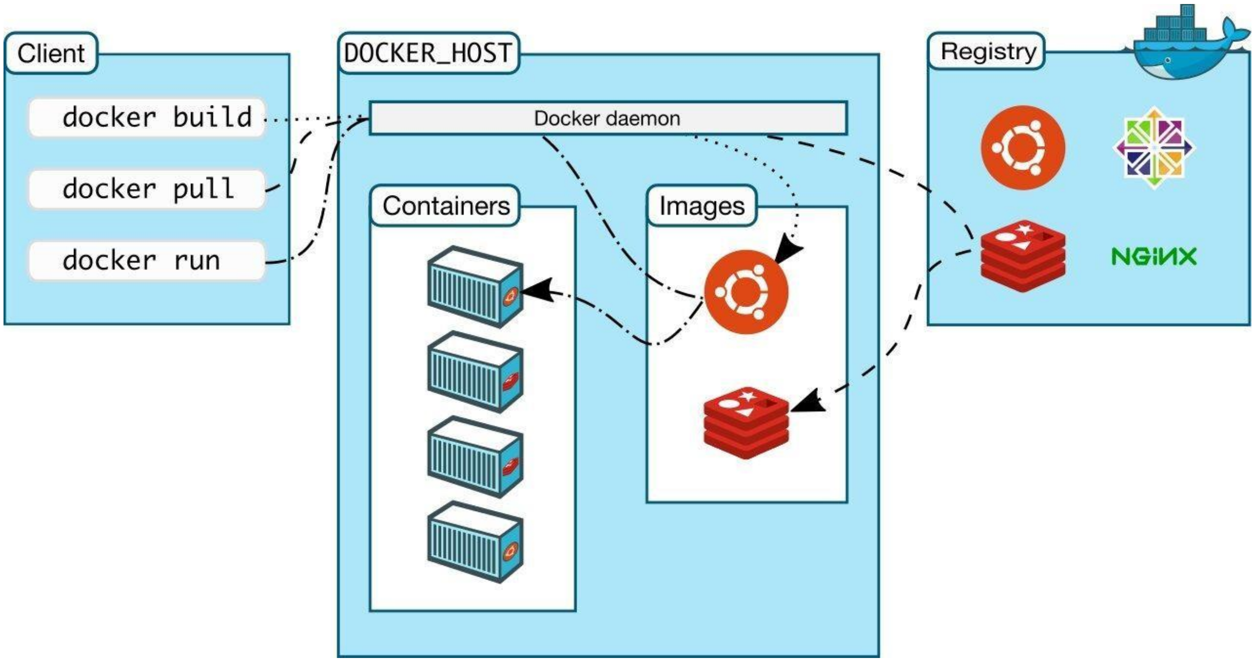
Docker Client: This is the CLI of docker where the user can execute the docker commands, the docker client accepts these commands and passes them to a background process called "docker daemon"

Docker daemon: This process accepts the commands coming from the docker client and routes them to work on docker images or containers or the docker registry.

Docker registry: This is the cloud site of docker where docker images are stored. This is of two types

1 Public Registry (hub.docker.com)

2 Private Registry (Setup on one of our local servers)



Important docker commands

Working on docker images

1) To pull a docker image

```
docker pull image_name
```

2) To search for a docker images

```
docker search image_name
```

3) To upload an image into docker hub

```
docker push image_name
```

4) To see the list of images that are downloaded

`docker images` or `docker image ls`

5) To get detailed info about a docker image

```
docker image inspect image_name/image_id
```

6) To delete a docker image that is not linked any container

```
docker rmi image_name/image_id
```

7) To delete an image that is linked to a container

```
docker rmi -f image_name/image_id
```

8) To save the docker image as a tar file

```
docker save image_name
```

9) To untar the tar file and get image

```
docker load tarfile_name
```

10) To delete all image

```
docker system prune -af
```

11) To create a docker image from a Docker file.

```
docker build -t image_name .
```

12) To create an image from a customized container

```
docker commit container_id/container_name image_name
```

Working on docker containers

1) To see the list of running containers

```
docker container ls
```

2) To see the list of all containers (running and stopped)

```
docker ps -a
```

3) To start a container

```
docker start container_id/container_name
```

4) To stop a container

```
docker stop container_id/container_name
```

5) To restart a container

```
docker restart container_id/container_name
```

To restart after 10 seconds

```
docker restart -t 10 container_id/container_name
```

6)To delete a stopped container

```
docker rm container_id/container_name
```

7) To delete a running container

```
docker rm -f container_id/container_name
```

8) To stop all running container

```
docker stop $(docker ps -aq)
```

9)To delete all stopped containers

```
docker rm $(docker ps -aq)
```

10) To delete all running and stopped containers

```
docker rm -f $(docker ps -aq)
```

11) To get detailed info about a container

```
docker inspect container_id/container_name
```

12) To see the logs generated by a container

```
docker logs container_id/container_name
```

13) To create a docker container

```
docker run image_name/image_id
```

run command options

--name: Used to give a name to the container.

-d: Used to run the container in detached mode.

-it: Used to open an interactive terminal in the container.

-e: Used to pass environment variables to the container.

-v: Used to attach an external device or folder as a volume.

--volume-from: Used to share volume between multiple containers.

-p: Used for port mapping. It will link the container port with the host port. Eg: -p 8080:80 Here 8080 is host port (external port) and
80 is container port (internal port).

-P: Used for automatic port mapping where the container port is mapped with some host port that id greater than 30000.

--link: Used to create a link between multiple containers to create a micro services architecture.

--network: Used to start a container on a specific network.

-rm: Used to delete a container on exit.

-m: Used to specify the upper limit on the amount of memory that a container can use.

-c: Used to specify the upper limit on the amount of cpu a container can use.

-ip: Used to assign an ip to the container

14) To see the ports used by a container

```
docker port container_id/container_name
```

15) To run any process in a container from outside the container

```
docker exec -it container_id/container_name process_name
```

Eg: To run the bash process in a container

```
docker exec -it container_id/container_name bash
```

16) To come out of a container without exit

```
ctrl+p,ctrl+q
```

17) To go back into a container from where the interactive terminal is running

```
docker attach container_id/container_name
```

18) To see the processes running in a container

```
docker container container_id/container_name top
```

Working on docker networks

1) To see the list of docker networks

```
docker network ls
```

2) To create a docker network

```
docker network create --driver network_type network_name
```

3) To get detailed info about a network

```
docker network insepct network_name/network_id
```

4) To delete a docker network

```
docker network rm network_name/network_id
```

5) To connect a running container to a network

```
docker netowork connect network_name/network_id container_name/container_id
```

6) To disconnect a running container to a network

```
docker netowork disconnect network_name/network_id container_name/container_id
```

Working on docker volumes

1) To see the list of docker volumes

```
docker volume ls
```

2) To create a docker volume

```
docker volume create volume_name
```

3) To get detailed info about a volume

```
docker volume inspect volume_name/volume_id
```

4) To delete a volume

```
docker volume rm volume_name/volume_id
```

Some Use cases

Use Case1:

Create an nginx container in detached mode and name it webserver

```
docker run --name webserver -d -p 9999:80 nginx
```

To access nginx from browser

```
public_ip_of_dockerhost:9999
```

Use Case2:

Create a jenkins container and perform automatic port mapping

```
 docker run  --name myjenkins -d -P jenkins/jenkins
```

To see the port mapping

```
docker port myjenkins
```

To access jenkins from browser

```
public_ip_of_dockerhost:port_from_step2
```

Create a centos container and open interactive terminal in it

```
docker run --name c1 -it centos
```

To come out of the centos container

```
 exit
```

Use Case3:

Create a mysql container and create few tables in it

```
docker run --name mydb -d -e MYSQL_ROOT_PASSWORD=pythonlife mysql
```

To open interactive bash shell in the container

```
docker exec -it mydb bash
```

To login as mysql root user

```
 mysql -u root -p
          Password: pythonlife
```

To see the list of databases

```
 show databases;
```

To move into any of the databases

```
 use sys;
```

Create emp and dept tables

Open https://justinsomnia.org/2009/04/the-emp-and-dept-tables-for-mysql/

Copy the code and paste it

To see the tables

```
select * from emp; select *
          from dept;
```

Linking of Container

To create a multi container architecture we have used the following ways.

1) --link option
2) Docker compose
3) Docker network
4) Python Scripts
5) Ansible playbooks

Use Case4:

Create 2 busybooks containers c1 and c2 and link them.

1) Create a busybox container and name it c1

```
docker run --name c1 -it busybox
```

2) To come out of the c1 container without exit

```
ctrl+p, ctrl+q
```

3) Create another busybox container c2 and link it with c1 container

```
docker run --name c2 -it --link c1:mybusybox busybox
```

3 Check if c2 is pinging to c1`ping c1`

Use Case5:

Setup WordPress and link it with MySQL container

1) Create a MySQL container

```
docker run --name mydb -d -e MYSQL_ROOT_PASSWORD=pythonlife mysql:5
```

2) Create a WordPress container and link with the MySQL container

```
docker run --name mywordpress -d -p 8888:80 --link mydb:mysql wordpress
```

3) To check if wordpress and mysql containers are running

```
docker container ls
```

4) To access wordpress from a browser

```
public_ip_dockerhost:8080
```

5) To check if wordpress is linked with mysql

```
docker inspect mywordpress
```

Use Case6:

Setup CI-CD environment where a Jenkins container is linked with 2 tomcat containers for QAserver and Prod Server.

1)Create a jenkins container

```
docker run  --name myjenkins -d -p 5050:8080 jenkins/jenkins
```

2)To access jenkins from browser

```
public_ip_dockerhost:5050
```

3)Create a tomcat container as qaserver and link with jenkins container

```
docker run --name qaserver -d -p 6060:8080 --link myjenkins:jenkins tomcat
```

4)Create another tomcat container as prodserver and link with jenkins

```
dockerrun --name prodserver -d -p 7070:8080 --link myjenkins:jenkins tomcat
```

6) Check if all 3 containers are running

```
docker container ls
```

Setup LAMP architecture

Create mysql container

```
docker run --name mydb -d -e MYSQL_ROOT_PASSWORD=pythonlife mysql
```

Create an apache container and link with mysql container

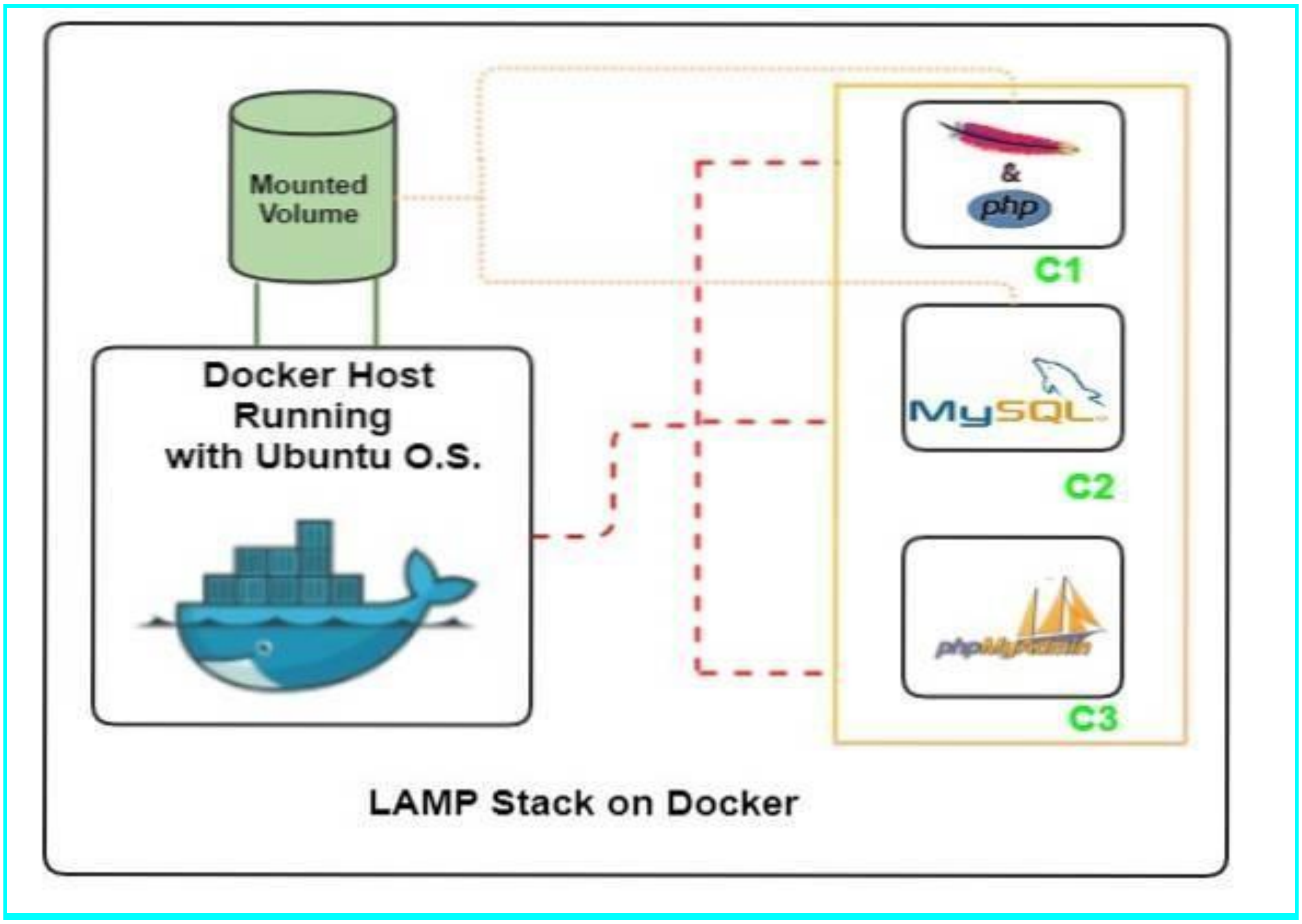
```
docker run --name apache -d -p 9999:80 --link mydb:mysql httpd
```

Create a php container and link with mysql and apache containers

```
docker run --name php -d --link mydb:mysql --link apache:httpd php:7.2-apache
```

To check if php container is linked with apache and mysql

```
docker inspect php
```



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.

Compose works in all environments: production, staging, development, testing, as well as CI workflows.

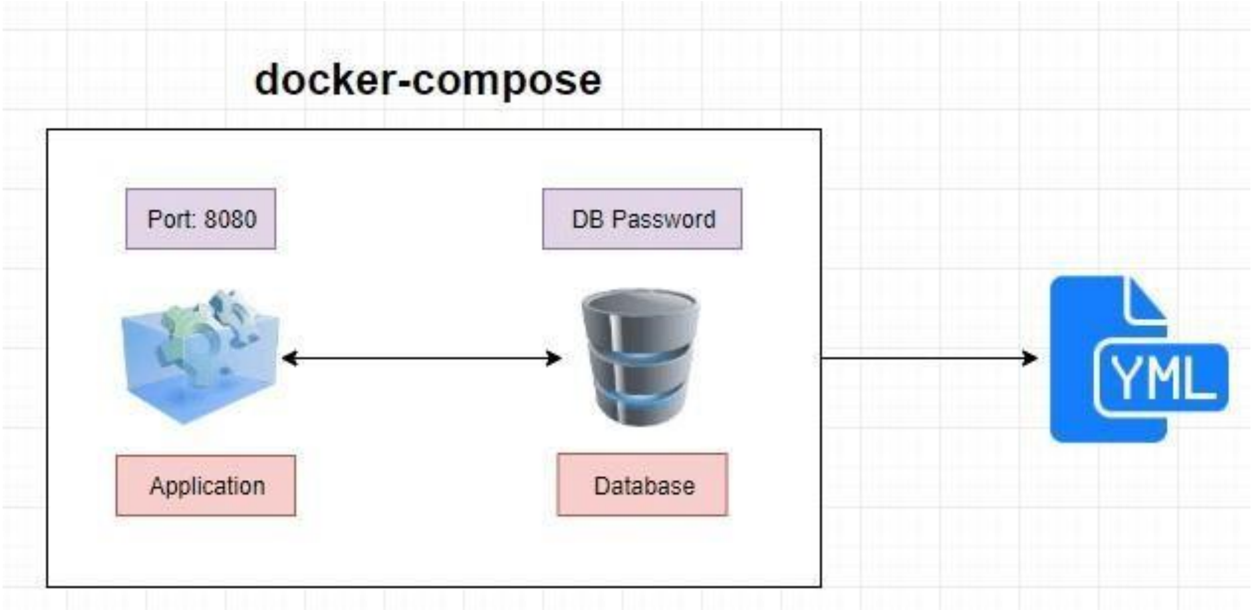
Using Compose is basically a three-step process:

- Define your app’s environment with a Docker file, so it can be reproduced anywhere.

- Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
- Run docker compose up and the Docker compose command starts and runs your entire app. You can alternatively run docker-compose up using Compose stand alone(docker-compose binary).

The disadvantage of the "link" option is it is deprecated and the same individual command has to be given multiple times to set up similar architectures.

To avoid this, we can use docker compose. Docker compose uses yml files to set up the multi container architecture and these files can be reused any number of times.



Setup of docker compose

1) Download the docker compose s/w

```
Sudo curl -L
"https://github.com/docker/compose/releases/download/1.29.0/docke r-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

2) Give execute permissions on it

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

3) To check if docker compose is installed or not

```
docker-compose --version
```

Url: https://docs.docker.com/compose/install/

Use Case7:

Create a docker compose file to setup a mysql and wordpress container and link them. vim docker-compose.yml

```
--version: '3.8'
services:
 mydb:
  image: mysql:5 environment:
   MYSQL_ROOT_PASSWORD: pythonlife
 mywordpress: image: wordpress ports:
   - 8888:80
  links:
   - mydb:mysql
...
```

To set up the containers from the above file

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

To stop all the container of the docker compose file

```
docker-compose stop
```

To start the container

```
docker-compose start
```

To stop and delete

```
 docker-compose down
```

Use Case8:

Create a docker compose file to setup the selenium testing environment where a hub container is linked with 2 node containers.

vim selenium.yml

```
--version: '3.8' services:

hub: image:

selenium/hub ports:

   -    4444:4444container_name:          hub

  chrome:   image:   selenium/node-chrome-

  debug ports:

   -    5901:5900

  links:

   -    hub:seleniumcontainer_name: chrome

 firefox: image: selenium/node-firefox-

  debug ports:

   -    5902:5900

  links:
        -    hub:seleniumcontainer_name: firefox

    ...
```

To create containers from the above file

```
docker-compose -f selenium.yml up -d
```

Docker Volumes

Docker volumes are file systems mounted on Docker containers to preserve data generated by the running container. The volumes are stored on the host, independent of the container life cycle. This allows users to back up data and share file systems between containers easily.

Containers are ephemeral but the data processed by the container should be persistent. This can be done using volumes.

Volumes are categorized into 3 types.They are

1) Simple docker volume
2) Shareable docker volumes
3) Docker Volume containers

Simple Docker Volumes

They are used only for preserving the data on the host machine even when the container is deleted.

1) Create a folder /datamkdir /data

2) Create an ubuntu container and mount the /data as volume

```
docker run --name u1 -it -v /data ubuntu
```

3) In the ubuntu container go into data folder and create some files

```
cd data touch file1 file2
   Exit
```

4) Identify the location where the volume data is stored

```
docker inspect u1
```

Search for the "Mounts" section and copy the "Source" path and check if data was stored or not.

5) Delete the ubuntu container

```
docker rm -f u1
```

6) Check if the data is still present

```
cd "source_path_from_step_4"
  ls
```

Sharable Docker volumes

These volumes are shareable between multiple containers. Create 3 centos containers c1,c2,c3.Mount /data as a volume on c1 container, c2 should use the volume used by c1 and c3 should use the volume used by c2.

1) Create a centos container c1 and mount /data.

```
docker run --name c2 -it -v /data centos
```

2) Go into the data folder and create files in the data folder.

```
cd datatouch
    f1 f2
```

3) Come out of the container without exit.

```
ctlr+p,ctlr+q
```

4) Create another centos container c2 and it should use the volumes used by c1.

```
docker run --name c2 -it --volumes-from c1 centos
```

5) In the c2 container go into the data folder and create some files.

```
cd datatouch
    f3 f4
```

6) Come out of the container without exit.

```
ctlr+p,ctlr+q
```

7) Create another centos container c3 and it should use the volume used by c2.

```
docker run --name c3 -it --volumes-from c2 centos
```

8) In the c3 container go into the data folder and create some files.

```
cd datatouch
    f5 f6
```

9) Come out of the container without exit

```
ctlr+p,ctlr+q
```

10) Go into any of the 3 containers and we will see all the files

```
docker attach c1
```

```
cd /data
```

```
ls exit
```

11) Identify the location where the mounted data is stored.

```
docker inspect c1
```

Search for "Mounts" section and copy the "Source" path

13 Delete all containers

```
docker rm -f c1 c2 c3
```

14 Check if the files are still present

```
cd "source_path_from"step12"
```

```
ls
```

Docker volume containers

These volumes are bidirectional, i.e. the changes done on the host will be reflected into the container and changes done by container will be reflected to the host machine.

1. Create a volume.

```
docker volume create myvolume
```

2. To check the location where the mounted volume works.

```
docker volume inspect myvolume
```

3. Copy the path shown in "MountPoint" and cd to that Path.

```
cd "MountPoint"
```

4. Create a few files here.

```
touch file1 file2
```

5. Create an ubuntu container and mount the above volume into the tmp folder.

```
docker run --name u1 -it -v myvolume:/tmp ubuntu
```

6. Change to tmp folder and check for the files.

```
cd /tmp
```

```
ls
```

If we create any files here, they will be reflected to the host machine and these files will be present on the host even after deleting the container.

Use Case9:

Create a volume "newvolume" and create tomcat-users.xml file in it. Create a tomcat container and mount the above volume into it. Copy the tomcat-users.xml files to the required location.

1. Create a volume

```
docker volume create newvolume
```

2. Identify the mount location.

```
docker volume inspect newvolume
```

Copy the "MountPoint" path

3. Move to this path.

```
cd "MountPoint path"
```

4. Create a file called tomcat-users.xml.

```
cat > tomcat-users.xml
```

```
<tomcat-users>
<user username="pythonlife" password="pythonlife" roles="manager-script"/>
</tomcat-users>
```

5. Create a tomcat container and mount the above volume.

```
docker run --name webserver -d -P -v newvolume:/tmp tomcat
```

6. Go into the bash shell of the tomcat container.

```
docker exec -it webserver bash
```

7. Move the tomcat-users.xml file into conf folder

```
mv /tmp/tomcat-users.xml conf/
```

Creating customized docker images

This can be done in 2 ways

1. Using docker commit command
2. Using Docker file

Using the docker commit command

Use Case10:

Create an ubuntu container and install some s/w's in it. Save this container as an image and later create a new container from the newly created image. We will find all the s/w's that we installed.

1. Create an ubuntu container

   ```
   docker run --name u1 -it ubuntu
   ```

2. In the container update the apt repo and install s/w's

   ```
   apt-get update apt-get install -y git tree
   ```

3. Check if git is installed or not

   ```
       git --version
        Exit
   ```

4. Save the customized container as an image

   ```
   docker commit u1 myubuntu
   ```

5. Check if the new image is created or no.

   ```
   docker images
   ```

6. Delete the previously create ubuntu container

```
docker rm -f u1
```

7. Create an new container from the above created image

```
docker run --name u1 -it myubuntu
```

8. Check for git

```
git --version
```

Docker file

Docker file uses predefined keywords to create customized docker images. The Docker engine includes tools that automate container image creation. While you can create container images manually by running the docker commit command, adopting an automated image creation process has many benefits, including:

- Storing container images as code.
- Rapid and precise recreation of container images for maintenance and upgrade purposes.
- Continuous integration between container images and the development cycle.

The Docker components that drive this automation are the Dockerfile, and the docker build command. The Dockerfile is a text file that contains the instructions needed to create a new container image. These instructions include identification of an existing image to be used as a base, commands to be run during the image creation process, and a command that will run when new instances of the container image are deployed.

Docker build is the Docker engine command that consumes a Dockerfile and triggers the image creation process.

Important keyword in Docker file

FROM: This is used to specify the base image from where a customized docker image has to be created.

MAINTAINER: This represents the name of the organization or the author that has created this Docker file.

RUN: Used to run linux commands in the container.Generally, it used to do s/w installation or running scripts.

USER: This is used to specify who should be the default user to login into the container.

COPY: Used to copy files from host to the customized image that we are creating.

ADD: This is similar to copy where it can copy files from host to image but ADD can also download files from some remote server.

EXPOSE: Used to specify what port should be used by the container.

VOLUME: Used for automatic volume mounting, i.e. we will have a volume mounted automatically when the container starts.

WORKDIR: Used to specify the default working directory of the container.

ENV: This is used to specify what environment variables should be used.

CMD: Used to run the default process of the container from outside.

ENTRYPOINT: This is also used to run the default process of the container.

LABEL: Used to store data about the docker image in key value pairs

SHELL: Used to specify what shell should be by default used by the image.

Use Case11:

Create a dockerfile to use nginx as base image and specify the maintainer as pythonlife.

1. Create a docker file.

```
vim dockerfile
        FROM nginx
        MAINTAINER pythonlife
```

2. To create an image from this file.

```
    docker build -t mynginx
```

3. Check if the image is created or not

```
    docker images
```

Use Case12:

Create a dockerfile from the Ubuntu base image and install git in it.

1. Create dockerfile vim dockerfile

```
          FROM ubuntu
          MAINTAINER pythonlife
          RUN apt-get update
          RUN apt-get install -y git
```

2. Create an image from the above file.

   ```
   docker build -t myubuntu
   ```

3. Check if the new image is created.

   ```
   docker images
   ```

4. Create a container from the new image and it should have git installed.

   ```
   docker run  --name u1 -it myubuntu git --
   version
   ```

Cache Busting

When we create an image from a Docker file docker stores all the executed instruction’s in its cache. Next time if we edit the same docker file and add few new instructions and build an image out of it docker will not execute the previously executed statements.

Instead it will read them from the cache. This is a time saving mechanism.The disadvantage is if the docker file is edited with a huge time gap then we might end up installing s/w's that are outdated.

Eg:

```
FROM ubuntu

RUN apt-get update

RUN apt-get install -y git
```

If we build an image from the above Docker file, docker saves all these instructions in the Docker cache and if we add the below statement. “RUN apt-get install -y tree”

Only this latest statement will be executed. To avoid this problem and make docker execute all the instructions once more time without reading from cache we use "cache busting".

```
docker build --no-cache -t myubuntu .
```

Create a shell script to install multiple Softwares and copy this into the docker image and execute it at the time of creating the image.

Use Case13:

1. Create the shell scriptvim script.sh apt-get update for

```
b in tree git wget
      do
         apt-get install -y $b
       done
```

2. Give execute permissions on that file

```
chmod u+x script.sh
```

3. Create the dockerfilevim dockerfile

```
FROM ubuntu
MAINTAINER pythonlife
COPY ./script.sh /
RUN ./script.sh
```

4. Create an image from the dockerfile

```
docker build -t myubuntu .
```

5. Create a container from the above image

```
docker run  --name u1 -it myubuntu
```

6. Check if the script.sh is present in / and also see if tree and git are installed.

```
ls  / git --
version
Tree
```

Use Case14:

Create a dockerfile from base centos image and make /data as the default volume.

1. Create a dockerfilevim dockerfile

```
FROM centos
MAINTAINER pythonlife
VOLUME /data
```

2. Create an image from the above dockerfile.

```
docker build -t mycentos .
```

3. Create a container from the above image and check for the volume.

```
docker run  --name u1 -it mycentos ls (we
should see the data folder)
```

4. Go into the data folder and create some files.

```
cd data touch
```

```
file1 file2 Exit
```

5. Check for the mount section and copy the source path

6. Delete the container

```
docker rm -f u1
```

7. Check if the files are still present

```
cd "source path"
```

Use Case15:

Create a docker file from nginx base image and expose 80 port

1.Create a dockerfile vim

dockerfile

```
FROM nginx

MAINTAINER pythonlife

EXPOSE 90
```

2. Create an image

```
docker build -t mynginx .
```

3. Create a container from above image

```
docker run --name n1 -d -P mynginx
```

4. Check for the ports exposed

```
docker port n1
```

Use Case16:

Create a dockerfile from Ubuntu base image and install java in it, download jenkins. war and make "java -jar jenkins. war" as the default process of the container.

1.Create a dockerfile vim

dockerfile

```
FROM ubuntu
MAINTAINER pythonlife
RUN apt-get update
RUN apt-get install -y openjdk-8-jdk
ADD https://get.jenkins.io/war-stable/2.263.4/jenkins.war /
ENTRYPOINT ["java","-jar","jenkins.war"]
```

2. Create an image from the above file

```
docker build -t myubuntu .
```

3. Create a container from the above image and we will see that it behaves like a jenkins container.

```
docker run  --name u1 -it myubuntu
```

4. Check the default process that is running.

```
docker container ls
```

Use Case17:

Create a dockerfile from centos base image and install “httpd” in it and make “httpd” as the default process.

1. Create the index.html

Vim index.html

```
 <html>
     <body>
          <h1>Welcome to pythonlife</h1>
     </body>
</html>
```

2. Create the dockerfile. vim dockerfile

```
FROM centos
MAINTAINER pythonlife
RUN yum -y update
RUN yum -y install httpd
COPY index.html /var/www/html
ENTRYPOINT ["/usr/sbin/httpd","-D","FOREGROUND"]
EXPOSE 80
```

3. Create an image from the above dockerfile.

```
docker build -t mycentos .
```

4. Create a container from the above image.

```
docker run --name c1 -d -P mycentos
```

5. Check the ports used by the container.

```
docker container ls
```

6. To access the from browser.

```
public_ip_of_dockerhost:port_from_step5
```

CMD and ENTRYPOINT

Both of them are used to specify the default process that should be triggered when the container starts but the CMD instruction can be overridden with some other process passed at the docker run command.

Eg:

```
FROM ubuntu

RUN apt-get update

RUN apt-get install -y nginx

CMD ["/usr/sbin/nginx","-g","daemon off;"]

EXPOSE 80
```

Though the default process is to trigger nginx we can bypass that and make it work on some other process.

```
docker build -t myubuntu .
```

Create a container

```
docker run --name u1 -it -d myubuntu
```

Here if we inspect the default process we will see nginx as the default process.

```
docker container ls
```

On the other hand, we can modify that default process to something else.

```
docker run --name u1 -d -P myubuntu ls -la
```

Now if we do "docker container ls" we will see the default process to be "ls -la".

Docker Networking

For Docker containers to communicate with each other and the outside world via the host machine, there has to be a layer of networking involved. Docker supports different types of networks, each fit for certain use cases.

For example, building an application which runs on a single Docker container will have a different network setup as compared to a web application with a cluster with database, application and load balancers which span multiple containers that need to communicate with each other. Additionally, clients from the outside world will need to access the web application container.

Docker supports 4 types of networks.

1 Bridge

2 Host

3 null

4 Overlay

Bridge Network:

Bridge Bridge networks apply to containers running on the same Docker daemon host. For communication among containers running on different Docker daemon hosts, you can either manage routing at the OS level, or you can use an overlay network.

When you start Docker, a default bridge network (also called bridge) is created automatically, and newly-started containers connect to it unless otherwise specified. You can also create user-defined custom bridge networks. User-defined bridge networks are superior to the default bridge network.

Host Network:

If you use the host network mode for a container, that container's network stack is not isolated from the Docker host (the container shares the host's networking namespace), and the container does not get its own IP-address allocated. For instance, if you run a container which binds to port 80 and you use host networking, the container's application is available on port 80 on the host's IP address.

Null or None Network:

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

Overlay Network:

The overlay network driver creates a distributed network among multiple Docker daemon hosts. This network sits on top of (overlays) the host-specific networks, allowing containers connected to it (including swarm service containers) to communicate securely when encryption is enabled. Docker transparently handles routing of each packet to and from the correct Docker daemon host and the correct destination container.

When you initialize a swarm or join a Docker host to an existing swarm, two new networks are created on that Docker host:

- an overlay network called ingress, which handles the control and data traffic related to swarm services. When you create a swarm service and do not connect it to a user-defined overlay network, it connects to the ingress network by default.
- a bridge network called docker_gwbridge, which connects the individual Docker daemon to the other daemons participating in the swarm.

Use Case18:

Create 2 bridge networks pythonlife1 and pythonlife2. Create 3 busybox containers c1,c2 and c3. c1 and c2 should run on the pythonlife1 network and should ping each other c3 should run on the pythonlife2 network and it should not be able to ping c1 or c2.

Now put c2 on the pythonlife2 network, since c2 is on both pythonlife1 and pythonlife2 networks it should be able to ping to both c1 and c3,but c1 and c3 should not ping each other directly.

1. Create 2 bridge networks

```
docker network create --driver bridge pythonlife1
```

```
docker network create --driver bridge pythonlife2
```

2. Check the list of available networks

```
docker network ls
```

3. Create a busybox container c1 on pythonlife1 network

```
docker run --name c1 -it --network pythonlife1 busybox
```

Come out of the c1 container without exit “ctrl+p,ctrl+q”

4. Identify the ip address of c1

```
docker inspect c1
```

5 Create another busybox container c2 on pythonlife1 network

```
docker run --name c2 -it --network pythonlife1 busybox ping ipaddress_of_c1(It will ping)
```

Come out of the c2 container without exit “ctrl+p,ctrl+q”

6. Identify the ip address of c2.

```
docker inspect c2
```

7. Create another busybox container c3 on the pythonlife2 network.

```
docker run --name c3 -it --network pythonlife2 busybox ping ipaddress_of_c1  (It should not ping) ping ipaddress_of_c2  (It should not ping)
```

Come out of the c3 container without exit “ctrl+p,ctrl+q”

8. Identify the ip address of c3.

```
docker inspect c3
```

9. Now attach pythonlife2 network to c2 container

```
docker network connect pythonlife2 c2
```

10. Since c2 is now on both pythonlife1 and pythonlife2 networks it should ping to both c1 and c3 containers.

```
docker  attach  c2  ping  ipaddress_of_c1   (It should  ping) ping ipaddress_of_c3  (It should ping)
```

Come out of the c2 container without exit “ctrl+p,ctrl+q”

11. But c1 and c3 should not ping each other.

```
docker attach c3 ping ipaddress_of_c1  (It should not ping)
```

Note: To create a network with a specific subnet range.

```
docker network create --driver bridge --subnet=192.168.2.0/24 pythonlife3
```

Use Case19:

Create a custom bridge network and create a docker compose file to start postgres and adminer container on the above created network.

1. Create a custom bridge network.

```
docker network create --driver bridge --subnet 10.0.0.0/24 pythonlife
```

2. Create a docker compose file.

```
vim docker-compose.yml
```

```
--version: '3.8'
services: db:
  image: postgres environment:
   POSTGRES_PASSWORD: pythonlife
   POSTGRES_USER: myuser
  POSTGRES_DB: mydb
adminer: image: adminer
ports:
  - 8888:8080
networks: default:
external: name:
Codingrad

...
```

3. To create the containers

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

4. To see if adminer and postgres containers are created.

```
docker container ls
```

5. To check if they are running on pythonlife network

```
docker inspect container_id_from_Step4
```

Docker compose files to create 2 networks and run containers on different networks.

```
vim docker-compose.yml
```

```
--version: '3.8'
        services: mydb:
          image: jenkins/jenkins ports:
           -    5050:8080network
          s:
```

```
      -    abcqaserver:
     image: tomee ports:
      -    6060:8080network
     s:
      -    xyzprodserver:
     image: tomee ports:
      -    7070:8080network
     s:
      -    xyznetworks: abc:
     {} xyz: {}
...
```

Docker compose files to create 2 containers and also create 2 volumes for both containers.

```
vim docker-compose.yml

--version: '3.8'
    services:
     db:
      image: mysql:5 environment:
       MYSQL_ROOT_PASSWORD: pythonlife
      volumes: mydb:/var/lib/mysql
     wordpress:
      image: wordpress ports:
       - 9999:80 volumes:
      wordpress:/var/www/html
    volumes: mydb:
      Wordpress

...
```

To start the service

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

To see the list of volumes

```
 docker volume ls
```

Create a dockerfile and use it directly in docker-compose

vim dockerfile

```
        FROM jenkins/jenkins

        MAINTAINER pythonlife

        RUN apt-get update

        RUN apt-get install -y git
```

vim docker-compose.yml

```
—
        version: '3.8'
        services:
        jenkins: build: .
        ports:
           -    7070:8080container_name:
          jenkins mytomcat: image:
          tomee ports:
           -    6060:8080container_name:
          mytomcat
...
```

To start the services

```
docker-compose up
```

Working on docker registry

A Docker registry is a place to store and distribute Docker images. It serves as a target for your docker push and docker pull commands. Before we get any further, let's cover some of the basic terminology related to Docker registries.

Image: An image is essentially a template for Docker containers. It consists of a manifest and an associated series of layers.

Layer: A layer represents a filesystem difference. An image's layers are combined together into a single image that forms the base for a container’s filesystem.

Registry: A registry is a content delivery and storage system for named Docker images. It can be thought of as a collection of repositories keyed by name.

Repository: A repository is simply a collection of different versions for a single Docker image. In this way, you can think of it as being similar to a git repository.

Tag: A tag is just a named version of the image. It allows you to identify a specific image later on. You can tag an image (almost) however you want. However, it's probably best to give it a meaningful, human-readable name that can be easily referenced later.

Why Do I Need a Docker Registry?

Imagine a workflow where you push a commit that triggers a build on your CI provider which in turn pushes a new image into your registry. Your registry can then fire off a webhook and trigger a deployment. All without a human having to step and manually do anything. Registries make a fully automated workflow like this much easier.

This is the location where the docker images are saved

This is of 2 types

1 Public registry

2 Private registry

Public Registry:

The registry allows Docker users to pull images locally, as well as push new images to the registry (given adequate access permissions when applicable). By default, the Docker engine interacts with
DockerHub , Docker's public registry instance.

Use Case20:

Create a custom centos image and upload into the public registry.

1) Signup into hub.docker.com

2) Create a custom centos image

   a) Create a centos container and install git init

```
docker run --name c1 -it centos yum -y update
yum -y install git exit
```

b) Save this container as an image

```
docker commit c1 pythonlife/mycentos
```

3 Login into docker hub

```
docker login
```

Enter username and password of docker hub

4 Push the customized image

```
docker push pythonlife/mycentos
```

Private Registry:

This can be created using a docker image called as "registry".We can start this as a container and it will allow us to push images into the registry.

1. Create registry as a container

```
docker run --name lr -d -p 5000:5000 registry
```

2. Download an alpine image

```
docker pull alipne
```

3. Tag the alpine with the local registry

```
docker tag alpine localhost:5000/alpine
```

4. Push the image to local registry

```
docker push localhost:5000/alpine
```

Container Orchestration

This is the process of handling docker containers running on multiple linux servers in a distributed environment.

Advantages:

1. Load Balancing
2. Scaling
3. Rolling update
4. High Availability and Disaster recovery(DR)

LoadBalancing

Each container is capable of sustaining a specific user load. We can increase this capacity by running the same application on multiple containers(replicas).

Scaling

We should be able to increase or decrease the number of containers on which our applications are running without the end user experiencing any downtime.

Rolling update

Applications running in a live environment should be upgraded or downgraded to a different version without the end user having any downtime.

Disaster Recovery

In case of network failures or server crashes still the container orchestration tools maintain the desired count of containers and thereby provide the same service to the end user.

Popular container orchestration tools

1) Docker Swarm

2) Kubernetes

3) OpenShift

4) Mesos

Docker swarm

A Docker Swarm is a group of either physical or virtual machines that are running the Docker application and that have been configured to join together in a cluster. Once a group of machines have been clustered together, you can still run the Docker commands that you're used to, but they will now be carried out by the machines in your cluster. The activities of the cluster are controlled by a swarm manager, and machines that have joined the cluster are referred to as nodes.

Features of Docker Swarm

Some of the most essential features of Docker Swarm are:

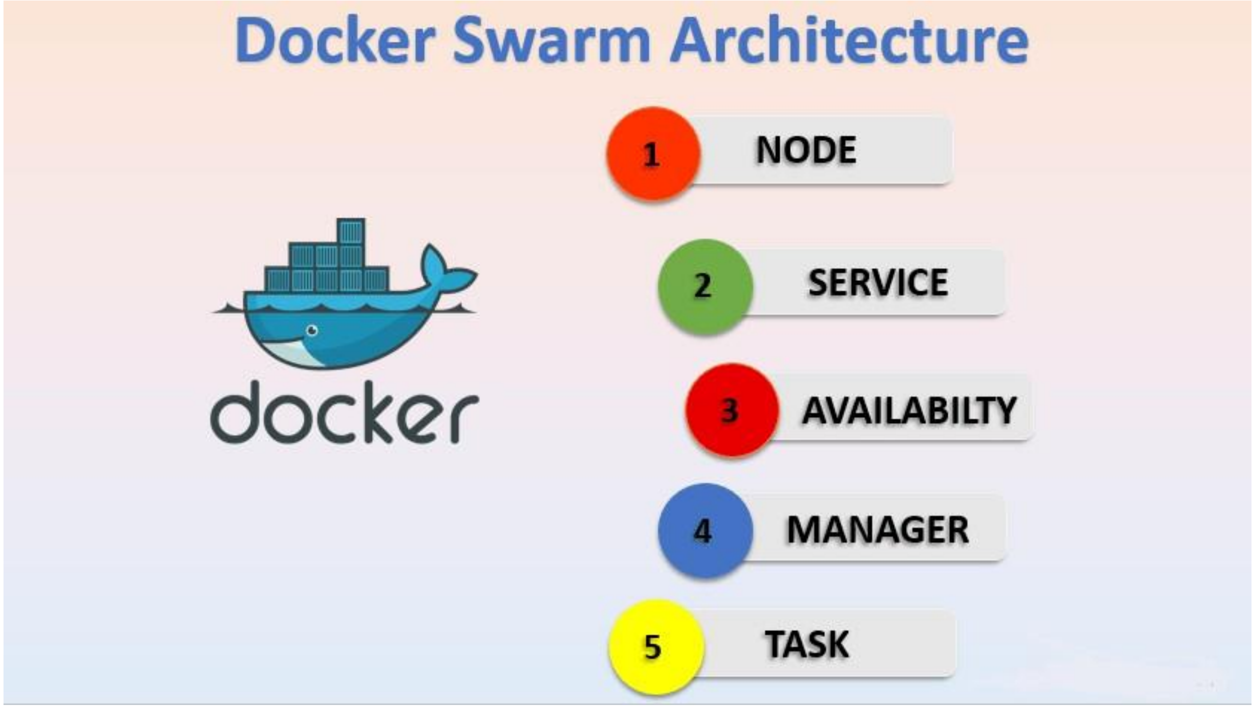
Decentralized access: Swarm makes it very easy for teams to access and manage the environment .

High security: Any communication between the manager and client nodes within the Swarm is highly secure.

Auto load balancing: There is auto load 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.



Setup of Docker Swarm

1. Create 3 GCP ubuntu instances.

2. Name them as Manager, Worker1,Worker2.

3. Install docker on all of them.

4. Change the hostname.

```
vim /etc/hostname
```

Delete the content and replace it with Manager or Worker1 or Worker2

5. Restart

```
init 6
```

6. To initialize the docker swarm, Connect to Manager GCP instance

```
docker swarm init
```

This command will create a docker swarm and it will also generate a tokenid.

7. Copy and paste the token id in Worker1 and Worker2.

Load Balancing:

Each docker container has a capability to sustain a specific user load. To increase this capability we can increase the number of replicas(containers) on which a service can run.

Use Case21:

Create nginx with 5 replicas and check where these replicas are running.

1) Create nginx with 5 replicas

```
docker service create --name webserver -p 8888:80 --replicas 5 nginx
```

2) To check the services running in swarm

```
docker service ls
```

3) To check where these replicas are running

```
docker service ps webserver
```

4) To access the nginx from browser

```
public_ip_of_manager/worker1/worker2:8888
```

5) To delete the service with all replicas

```
docker service rm webserver
```

Use Case22:

Create mysql with 3 replicas and also pass the necessary environment variables.

1.
```
docker service create --name db --replicas 3 -e
MYSQL_ROOT_PASSWORD=codingrad  mysql:5
```

2 To check if 3 replicas of mysql are running.

```
docker service ps db
```

Scaling

This is the process of increasing the number of replicas or decreasing the replicas count based on requirement without the end user experiencing any down time.

Use Case23:

Create tomcat with 4 replicas and scale it to 8 and scale it down to 2.

1. Create tomcat with 4 replicas

```
docker service create --name appserver -p 9090:8080 --replicas4 tomcat
```

2. Check if 4 replicas are running

```
docker service ps appserver
```

3. Increase the replicas count to 8

```
docker service scale appserver=8
```

4. Check if 8 replicas are running

```
docker service ps appserver
```

5. Decrease the replicas count to 2

```
docker service scale appserver=2
```

6. Check if 2 replicas are running

```
docker service ps appserver
```

Rolling updates

Services running in docker swarm should be updated from once version to others without the end user downtime.

Use Case24:

Create redis:3 with 5 replicas and later update it to redis:4 also rollback to redis:3.

1. Create redis:3 with 5 replicas

```
docker service create --name myredis --replicas 5 redis:3
```

2. Check if all 5 replicas of redis:3 are running.

```
docker service ps myredis
```

3. Perform a rolling update from redis:3 to redis:4

```
docker service update --image redis:4 myredis
```

4. Check redis:3 replicas are shut down and in this place redis:4 replicas are running.

```
docker service ps myredis
```

5. Roll back from redis:4 to redis:3

```
docker service update --rollback myredis
```

6. Check if redis:4 replicas are shut down and in its place redis:3 isrunning.

```
docker service ps myredis
```

Some Imp Notes

★ To remove a worker from swarm cluster

```
docker node update --availability drain Worker1
```

★ To make this worker rejoin the swarm

```
docker node update --availability active Worker1
```

★ To make worker2 leave the swarm, Connect to worker2 using git bash.

```
docker swarm leave
```

★ To make manager leave the swarm

```
docker swarm leave --force
```

★ To generate the tokenid for a machine to join swarm as

worker

```
docker swarm join-token worker
```

★ To generate the tokenid for a machine to join swarm as manager

```
docker swarm join-token manager
```

★ To promote Worker1 as a manager

```
docker node promote Worker1
```

★ To demote "Worker1" back to a worker status

```
docker node demote Worker1
```

Failover Scenarios of Workers

Create httpd with 6 replicas and delete one replica running on the manager. Check if all 6 replicas are still running, Drain Worker1 from the docker swarm and check if all 6 replicas are running.

On Manager and Worker2, make Worker1 rejoin the swarm. Make Worker2 leave the swarm and check if all the 6 replicas are running on Manager and Worker1.

1. Create httpd with 6 replicas

```
docker service create  --name webserver -p 9090:80 --replicas 6 httpd
```

2. Check the replicas running on Manager

```
docker service ps webserver | grep Manager
```

3. Check the container id

```
docker container ls
```

4. Delete a replica

```
docker rm -f container_id_from_step3
```

5. Check if all 6 replicas are running

```
docker service ps webserver
```

6. Drain Worker1 from the swarm

```
docker node update --availability drain Worker1
```

7. Check if all 6 replicas are still running on Manager and Worker2

```
docker service ps webserver
```

8. Make Worker1 rejoin the swarm

```
docker node update --availability active Worker1
```

9. Make Worker2 leave the swarm, Connect to Worker2 using git bash

```
docker swarm leave
```

10. Connect to Manager,Check if all 6 replicas are still running

```
docker service ps webserver
```

Failover Scenarios of Managers

If a worker instance crashes all the replicas running on that worker will be moved to the Manager or the other workers.If the Manager itself crashes the swarm becomes headless.

i.e, we cannot perform container orchestration activities in this swarm cluster. To avoid this, we should maintain multiple managers . Manager nodes have the status as Leader or Reachable.If one manager node goes down other manager becomes the Leader.

Quorum is responsible for doing this activity and uses a RAFT algorithm for handling the failovers of managers. Quorum also is responsible for maintaining the min number of managers. Min count of managers required for docker swarm should be always more than half of the total count of Managers.

Total Manager Count	Min Manager Required
1	1
2	2
3	2
4	3
5	3
6	4
7	4

Overlay Networking

This is the default network used by swarm and this network performs network load balancing.

i.e, even if a service is running on a specific worker we can access if from other slave.

Use Case25:

Start nginx with 2 replicas and check if we can access it from browser from the manager and all workers.

1. Create nginx

```
docker service create  --name webserver -p 8888:80 --replicas 2 nginx
```

2. Check where these 2 replicas are running

```
docker service ps webserver
```

These replicas will be running on only 2 nodes and we will have a third node where it is not running.

3. Check if we can access nginx from the third node where it is notpresent.

```
public_ip_of_thirdnode:8888
```

Use Case26:

Create 2 overlay networks codingrad1 and codingrad2. Create httpd with 5 replaces on codingrad1 network.

Create tomcat with 5 replicas on default overlay "ingres" network and later perform a rolling network update to codingrad2 network.

1. Create 2 overlay networks.

```
docker network create  --driver overlay pythonlife1 docker
network create  --driver overlay pythonlife2
```

2. Check if 2 overlay networks are created.

```
docker network ls
```

3. Create httpd with 5 replcias on pythonlife1 network.

```
docker service create  --name webserver -p 8888:80 --replicas 5
--network pythonlife1 httpd
```

4. To check if httpd is running on the pythonlife1 network.

```
docker service inspect webserver
```

This command will generate the output in JSON format. To see the above output in normal text format.

```
docker service inspect webserver --pretty
```

5. Create tomcat with 5 replicas on the default ingres network.

```
docker service create --name appserver -p 9999:8080 --replicas
5 tomcat
```

6. Perform a rolling network update from ingres to pythonlife2 network.

```
docker service update --network-add pythonlife2 appserver
```

7. Check if tomcat is now running on intelliqit2 network

```
docker service inspect appserver --pretty
```

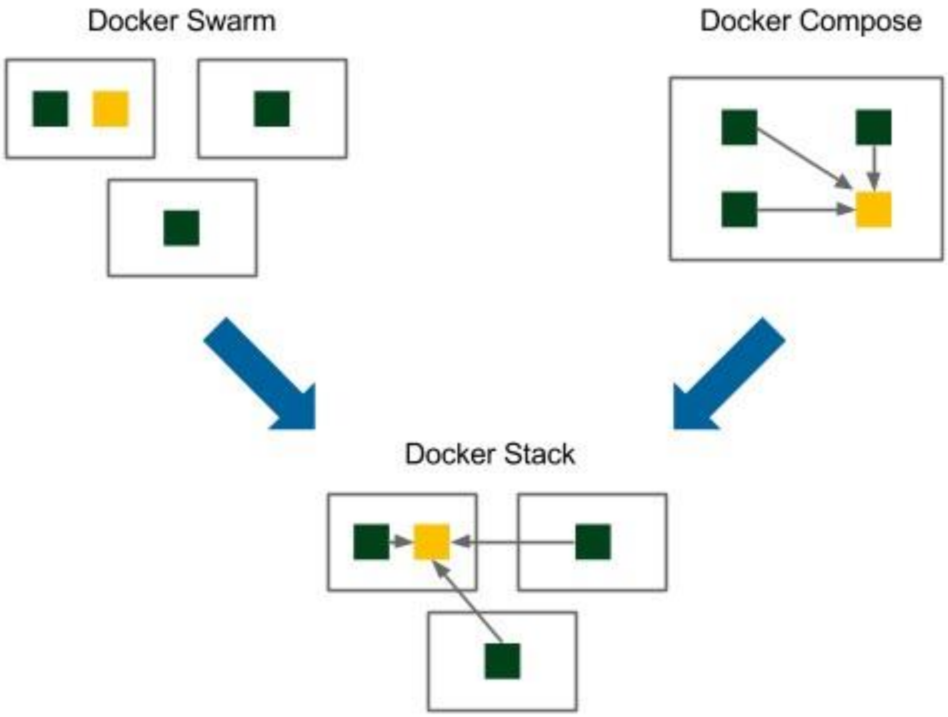
Note: To remove from pythonlife2 network

```
docker service update --network-rm pythonlife2 appserver
```

Docker Stack

Docker compose when implemented at the level of docker swarm it is called docker stack. Using docker stack we can create an orchestrate.a micro services architecture at the level of production servers.

Docker Stack is run across a Docker Swarm, which is essentially a group of machines running the Docker daemon, which are grouped together, essentially pooling resources. Stacks allow for multiple services, which are containers distributed across a swarm, to be deployed and grouped logically.



docker compose + docker swarm = docker stack docker

compose + kubernetes = kompose

1. To create a stack from a compose file

   ```
   docker stack deploy -c compose_filename stack_name
   ```

2. To see the list of stacks created

   ```
   docker stack ls
   ```

3. To see on which nodes the stack services are running

   ```
   docker stack ps stack_name
   ```

4. To delete a stack

   ```
   docker stack rm stack_name
   ```

Use Case27:

Create a docker stack file to start 3 replicas of wordpress and one replica of mysql.

```
vim stack1.yml
```

```
--version: '3.8'
services: db:
  image: "mysql:5" environment:
   MYSQL_ROOT_PASSWORD: pythonlife wordpress:
  image: wordpress ports:
   - "8989:80" deploy:
   replicas: 3
```

...

- To start the stack file

  ```
  docker stack deploy -c stack1.yml mywordpress
  ```

- To see the services running.

  docker service ls

- To check where the services are running.

  docker stack ps mywordpress

- To delete the stack

- docker stack rm mywordpress

Use Case28:

Create a stack file to setup CI-cd architecture where a jenkins container is linked with tomcats for qa and prod environments.

The jenkins containers should run only on Manager, the qaserver tomcat should run only on Worker1 and prod server tomcat should run only on worker2.

vim stack2.yml

```
--version: '3.8'
services:
myjenkins:
  image: jenkins/jenkins ports:
   -    5050:8080deploy:
  replicas: 2 placement:
  constraints:
   -    node.hostname ==
  Managerqaserver: image:
  tomcat ports:
```

```
 -   6060:8080deploy:
replicas: 3 placement:
constraints:
 -   node.hostname ==
Worker1 prodserver: image:
tomcat ports:
 -   7070:8080deploy:
 -
 -   replicas: 4 placement:
constraints:
 -   node.hostname ==
Worker2
```

...

- To start the services

  docker deploy -c stack2.yml ci-cd

- To check the replicas

  docker stack ps ci-cd

Docker secrets

This is a feature of docker swarm using which we can pass secret data to the services running in swarm cluster.

These secrets are created on the host machine and they will be available from all the replicas in the swarm cluster.

1 Create a docker secret.

```
echo " Hello pythonlife" | docker secret create mysecret -
```

2 Create a redis db with 5 replicas and mount the secret.

```
docker service create --name myredis --replicas 5 --secret
mysecret redis
```

3. Capture one of the replica container id.

```
docker container ls
```

4. Check if the secret data is available.

```
docker exec -it container_id cat /run/secrets/mysecret
```

Create 3 secrets for postgres user, password and db and pass them to the stack file.

1 Create secretsecho "pythonlife" | docker secret create
pg_password echo "myuser" | docker secret create pg_user echo
"mydb" | docker secret create pg_db -

2 Check if the secrets are createddocker secret ls

3 Create the docker stack file to work on these secretsvim
stack6.yml

```
--version: '3.1'
        services:
          db:
```

```
          image: postgres environment:
          POSTGRES_PASSWORD_FILE: /run/secrets/pg_password
          POSTGRES_USER_FILE: /run/secrets/pg_user
          POSTGRES_DB_FILE: /run/secrets/pg_db secrets:
          - pg_password
          - pg_user
          - pg_dbadminer:
          image: adminer restart:
          always ports:
          - 8080:8080deploy:
          replicas: 2
     secrets:
          pg_password: external:
          true pg_user:
          external: true pg_db:
          external: true
```

…

- To start the services

  docker Stack deploy -c stack6.yml

  mydb

- To check the replicas docker stack ps

  mydb

Difference B/w Docker Swarm and Kubernetes

Features	**Docker Swarm**	**Kubernetes**
Installation & Cluster Configuration	Installation is very simple;but cluster is not very strong	Installation is complicated;but once setup ,the cluster is very strong
GUI	There is no GUI	GUI is the kubernetes Dashboard
Scalability	Highly Scalable & scale 5x faster than kubernetes	Highly Scalable & Scale fast
Auto-Scaling	Docker Swarm cannot do Autoscaling.	Kubernetes can do auto-scaling.
Load Balancing	Docker Swarm does auto load balancing of traffic between containers in the cluster.	Manual intervention is needed for load balancing traffic between different containers in different pods.
Rolling Updates & Rollbacks	Can deploy rolling updates, but not automatic rollbacks.	Can deploy rolling updates, & does automatic rollbacks.
Data Volumes	Can share storage volumes with any other container.	Can share storage volumes only with other containers in the same pod.

Logging & Monitoring	3rd party tools like ELK should be used for logging and monitoring.	In-built tools for logging & Monitoring.