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 an CentOS Linux Machine

- Please Visit the below link

  Install Docker Engine on CentOS | Docker Docs

- Install the utils using below command

  ```
  sudo yum install -y yum-utils
  ```

- Add the repo using the command

  ```
  sudo yum-config-manager --add-repo
  https://download.docker.com/linux/centos/docker-ce.repo
  ```

- To install the latest version, run:

  sudo yum install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin

- Start Docker

  sudo systemctl start docker

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.

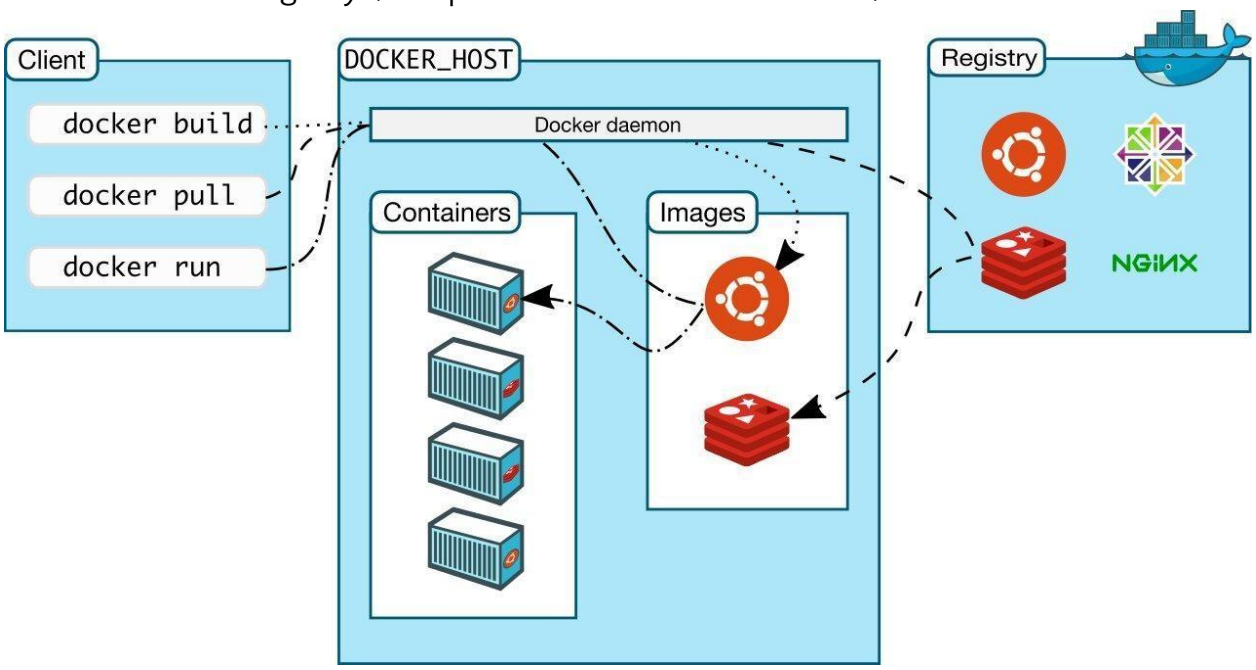
Docker Client: This is the CLI of docker where the user can execute the docker commands, the docker client accepts these commands and passes them to a background process called "docker daemon"

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 -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
```

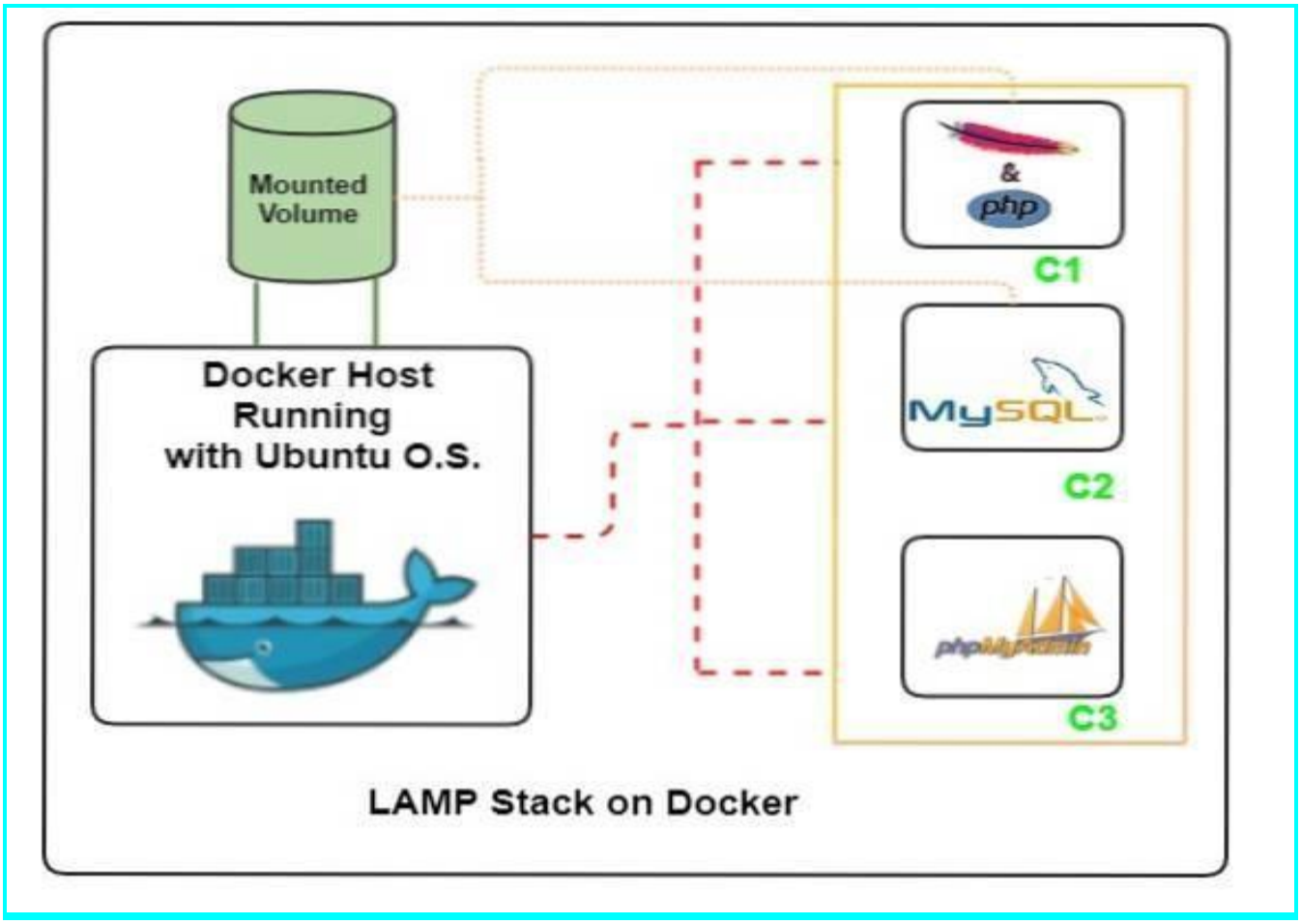
To come out of the centos container

```
exit
```

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



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.

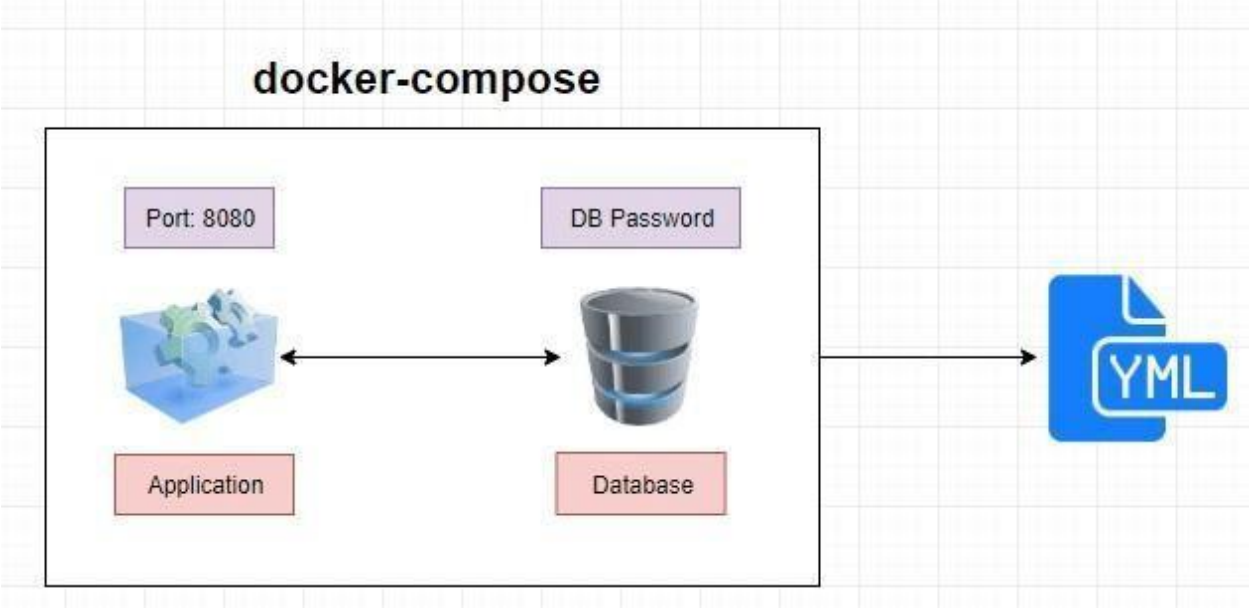
Using Compose is basically a three-step process:

- 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.



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

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 Case:

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

1. Create a docker file

```
FROM almalinux:8

RUN yum install nginx -y

RUN echo "welcome to Dockerfile" > /usr/share/nginx/html/hello.html

CMD ["nginx", "-g", "daemon off;"]
```

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.

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:

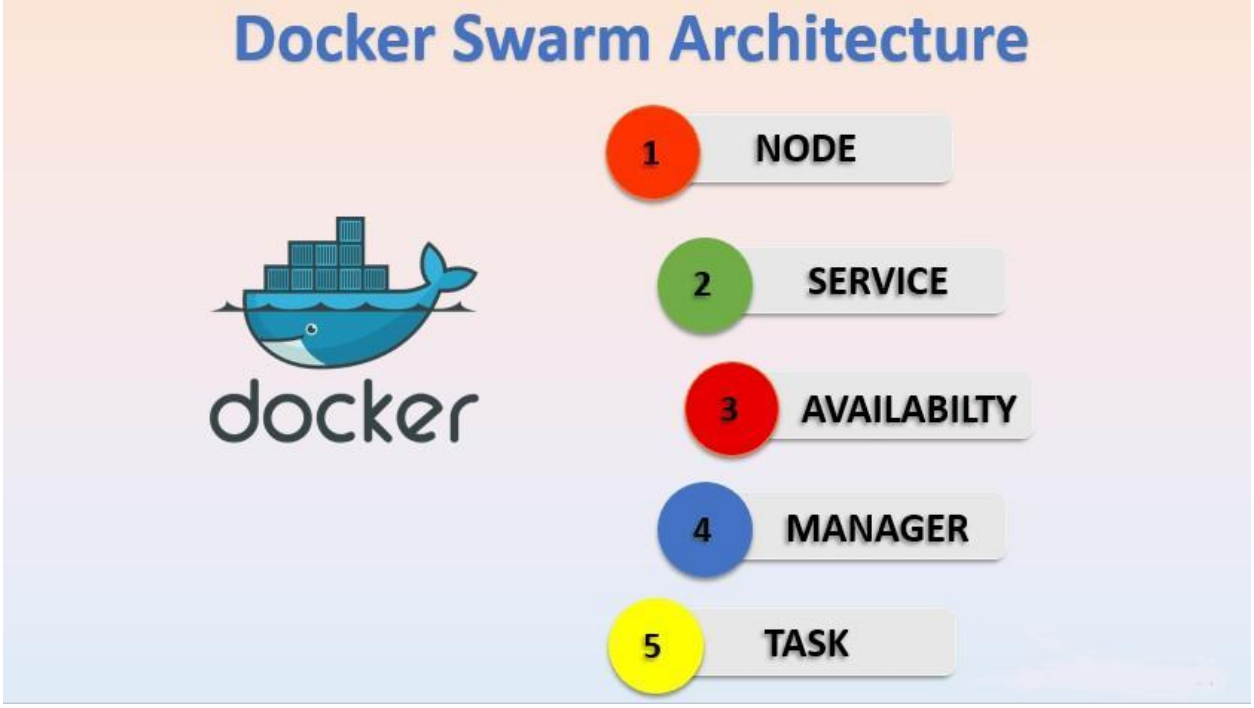
Decentralized access: Swarm makes it very easy for teams to access and manage the environment .

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.



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

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.