## Docker:

**Virtualization:** Here we have a bare metal (H/W) on top of which we install the host OS. On the host OS we install an application called hypervisor (VMware, ESXI, Citrix Xen, Hyper-V). On the hypervisor we can install any guest OS and on the guest OS we can install the applications that we want.

The problem here is, these applications must pass through many layers in order to access the h/w resources.

<table>
<tr><td><b>Oracle APP</b></td><td><b>MS SQL</b></td></tr>
<tr><td><b>Guest OS</b></td><td><b>Guest OS (Windows)</b></td></tr>
<tr><td colspan="2"><b>Hypervisor</b></td></tr>
<tr><td colspan="2"><b>Host OS (ubuntu)</b></td></tr>
<tr><td colspan="2"><b>Bare metal</b></td></tr>
</table>

## Containerization:

Here we have a bare metal on which the Host OS is installed, on the host OS we install an application called Docker engine. On the Docker engine we can run any application. These applications have to pass through less number of layers in order to access the hardware resources

<table>
<tr><td><b>Oracle APP</b></td><td><b>MS SQL app</b></td></tr>
<tr><td colspan="2"><b>Docker Engine</b></td></tr>
<tr><td colspan="2"><b>Host OS Ubuntu</b></td></tr>
<tr><td colspan="2"><b>Bare Metal</b></td></tr>
</table>

Docker performs *"process isolation"* ie. it removes the dependency that an application on an OS and it allows that application to run directly on the docker engine. Docker can spin up the necessary environment be it dev environment or testing environment or prod environment in a matter of seconds.

Docker can be used at all the stages of Build, Ship and Run. ie. for developing applications, testing and building them and finally running them in prod environment.

Docker comes in 2 flavors

1. CE(*Community Edition)*
2. *EE(Enterprise Edition)*

### Docker Images and containers

A docker image is a collection of binaries and libraries which are necessary for one software application to run. A running instance of an image is called as a container. Any number of containers can be created from one image.

### Docker Host

The machine on which docker is installed is called as the docker host. It can be Windows, Linux or Mac.

### Docker client

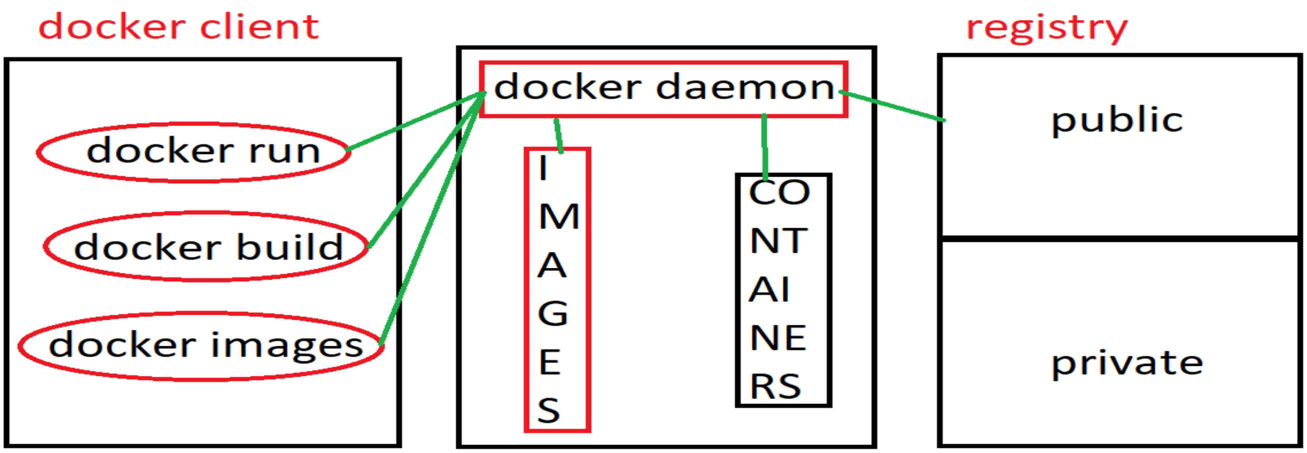
This is an application which is part of the docker engine which is responsible for accepting the docker commands from the user and pass it to docker daemon.

### Docker Daemon

This is a background process which is also a part of docker engine and the responsibility of docker daemon is to accept the docker commands from the docker client and depending on the kind of command route them to docker images or containers or the docker registry

### Docker registry

This is the location where all the docker images are stored. It is of two types. Public and Private. Public registry is hub.docker.com and images uploaded here can be accessed by anyone. Private registry is setup within our servers using a docker image called registry and only our organization team members can access the registry.



**Working on docker images**

To download a docker image

```
docker pull image_name
```

To upload a docker image into the registry

```
docker push image_name
```

To see the list of all docker images available in docker host

```
docker image ls (or) docker images
```

To search for an image in the registry from the command prompt

```
docker search image_name
```

To tag an image with a registry

```
docker tag image_name registry_id/new_image_name
```

To create a new image from a docker container

```
docker commit container_name/container_id new_image_name
```

To create a new image from a dockerfile

```
docker build –t new_image_name .
```

To delete all unused images

```
docker system prune –a
```

To delete a specific unused image

```
docker image rm <imagename>
```

To delete all unused images without atleast one container associated to them

```
docker image prune -a
```

**Working on Docker containers**

To start a stopped container

```
docker start container_name/container_id
```

To stop a running container

```
docker stop container_name/container_id
```

To remove a stopped container

```
docker rm container_name/container_id
```

To remove a running container

```
docker rm –f container_name/container_id
```

To restart a running container

```
docker restart container_name/container_id
```

To restart after 30 seconds

```
docker restart –t 30 container_name/container_id
```

To stop all running containers

```
docker stop $(docker ps –aq)
```

To see all stopped containers

```
docker ps
```

To delete all stopped containers

```
docker rm $(docker ps –aq)
```

To delete all containers (running as well as stopped)

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

To see the logs generated by a container

```
docker logs container_name/container_id
```

To see the ports opened by the container

```
docker port container_name/container_id
```

To come out of the shell of a container without exit

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

To re-enter into the shell of that container

```
docker attach container_name/container_id
```

To get detailed info about any container

```
docker inspect container_name/container_id
```

To execute any command in a container from the docker host

```
docker exec –it  cont_name/cont_id command to be executed
```

Eg: To open interactive bash shell in a container

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

To see the list of all the running containers

```
docker container ls
```

To see the list of all the containers (running as well as stopped)

```
docker ps –a
```

To create a new container

```
docker run image_name
```

Run command options

| | |
|---|---|
| **--name** | **Used to give a name for the container** |
| **-it** | **Used for opening interactive terminal in the container** |
| **-d** | **Used to run the container in detached mode as a background process** |
| **-v** | **Used for attaching an external directory or device as a volume** |
| **--volumes-from** | **Used to create sharable volumes which can be used by multiple containers** |
| **-p** | **Used for port mapping. It will help the container port (internal port) with a port on the docker host (external port)**<br>**Eg: -p 8080:80 Here 8080 is a container port and 80 is the host port** |
| **-P** | **Used for automatic port mapping ie the internal port of the container will be linked with some port number which is greater than 30000** |
| **--link** | **Used for linking multiple containers for creating the micro services architecture** |
| **-e** | **Used for passing environment variables to the container** |
| **-rm** | **This will delete the container on exit** |
| **--network** | **This is used to specify on which network these containers should run** |
| **--memory** | **Used for a fixed amount of memory allocation to the containers** |
| **--cpus** | **Used for specifying how many cpu’s should be used by the container** |

### Working on docker networking

To see the list of networks

```
docker network ls
```

To get detailed info about a network

```
docker network inspect network_id/network_name
```

To create a network

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

To attach a running container to a network

```
docker network connect net_id/net_name cont_name/cont_id
```

To remove a container from a network

```
docker network disconnect net_id/net_name cont_id/cont_name
```

To delete a network

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

**Working on Docker volumes:**

To see the list of all the docker volumes

```
docker volume ls
```

To create a new docker volume

```
docker volume create volume_name
```

To get detailed info about a volume

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

To delete a volume

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

To delete all unused volumes

```
docker system prune –volumes
```

### Usecase-1:

Start tomcat as a container and run it in detached mode. Map the 8080 port of tomcat with 9090 on the docker host machine

```
docker run --name appserver -d -p 9090:8080 tomcat
```

To access the homepage of tomcat, launch any browser,

public-ip-of-docker-host:9090

### Usecase-2:

Start nginx as a container and run it in detached mode. Do automatic port mapping

```
docker run --name webserver -d -P nginx
```

To see the port mapping of nginx container,

```
docker port webserver
```

To access the homepage of nginx, launch any browser

Public-ip-of-docker-host:port_no_captured from above command

### Usecase-3:

Start centos as a container and open interactive terminal in it

```
docker run --name mycentos -it centos
```

Execute Linux commands in the centos container

To come out of the container

Exit

### Usecase-4:

Start mysql as a container and login into mysql database. Create few tables

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

To open interactive terminal in the container

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

To login into the database as a root user

```
mysql -u root -p
```

To see the list of databases

```
show databases;
```

To move into any of the above shown databases,

```
use <database_name>;
```

Eg: use sys

To create an emp and dept table here

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

Copy the code for emp and dept table creation, paste in the mysql container.

To see the data in the database

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

## Usecase-5:

Start Jenkins as a container in detached mode and unlock Jenkins

```
docker run --name myjenkins -d -p 9090:8080 jenkins
```

To access the homepage of Jenkins,

Launch any browser, public-ip-of-dockerhost-machine:9999

To access the initial admin password, open myjenkins bash

```
docker exec -it myjenkins bash
```

Copy the initial password of Jenkins from the path

```
cat /var/jenkins_home/secrets/initialAdminPassword
```

**Creating Multi Container Architecture:**

Bigger applications are broken down into smaller containers and they can be linked with each other for creating a micro services architecture. This linking of containers can be done in the following ways

1. --link
2. docker compose
3. networking
4. Python/Shell programs

**Usecase-6:**

Start 2 busybox containers and name them c1 and c

2 and link both of them. Check if we can ping from c2 to c1

1. Start the 1st busybox container

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

2. Come out of the busybox container without exit

   Ctrl+p, ctrl+q

3. Start another busybox container and link it with the first busybox

   ```
   docker run --name c2 –it --link c1:c1alias busybox
   ```

4. In the c2 container check if we can ping to c1

   ```
   ping c1
   ```

Should get response.

## Usecase-7

Create a development environment where mysql environment should be linked with wordpress container using docker.

Start mysql as a container

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

Start wordpress as a container and link with mysql container

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

To access wordpress, launch any browser,

Public_ip_address_of_dockerhost:8888

Install wordpress -->Developer should be able to create a sample website.

## Usecase-8

LAMP is an opensource development architecture where we use PHP for application development, tomcat as an application server, and mysql as a database. All these should run on Linux operating system. Create this above architecture using docker.

Start php as a container

```
docker run --name myphp –d php:7.2-apache
```

Start tomcat as a container and link with php

```
docker run --name apachetc –p 8899:8080 –d --link  myphp:phpalias
tomcat
```

Start mysql as a container and link both php and tomcat containers

```
docker run --name mydb –d –e MYSQL_ROOT_PASSWORD=root
--link myphp:phpalias --link apachetc:apachealias mysql:5
```

**Usecase-9**

Create master slave setup of jenkins using docker containers

```
docker run --name master –p 7080:8080 –d jenkins
```

Identify the ip address of the container

```
docker inspect master
```

Start ubuntu as a container and link with master

```
docker run --name slave –it --link master:jenkins ubuntu
```

In the slave container install wget and download slave.jar

```
apt-get update
apt-get install –y wget
```

wget http://ipaddressofmaster:8080/jnlpJars/slave.jar (you can give alias name/name of container as well ie., master:8080)

To remove master slave containers

```
docker rm –f master slave
```

**Usecase-10**

Creating a testing environment where a selenium/hub container can be linked with 2 node containers. One node with firefox installed and another with chrome installed.

Start selenium hub as a container

```
docker run --name hub –d –p 4444:4444 selenium/hub
```

Start a chrome node container and link with hub container

```
docker run --name chrome –d –p 5901:5900 --link hub:selenium selenium/node-chrome-debug
```

Start a firefox node container and link with hub container

```
docker run --name firefox –d –p 5902:5900 --link hub:selenium selenium/node-firefox-debug
```

The above 2 containers are GUI containers and we can access them using VNCviewer

Install vnc viewer from

https://www.realvnc.com/en/conncet/download/viewer/windows/

Open vnc viewer

Enter public_ip_of_dockerhost:5901 (or) 5902

Click on continue.... enter password: secret

### Usecase-11

Create lamp architecture using docker containers.

Lamp is an opensource environment where the operating system that is used is linux, the database is mysql, application servers are running on apache and the development language is php.

Start mysql as a container

```
docker run --name mydb –d –e MYSQL_ROOT_PASSWORD=root mysql
```

Start httpd(apache) as a container and link it with mysql

```
docker run --name apache –d --link mydb:mysql –p 8080:80 httpd
```

Start php as a container and link with both mysql and apache containers

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

To check if all three containers are linked or not

```
docker inspect php
```

Search for “Links” section in the JSON output.

## Usecase-12

Create CI CD environment using docker containers.

Start devserver Jenkins as a container

```
docker run --name devserver -p 5050:8080 -d jenkins
```

Start tomcat as a container and link with jenkins

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

Start tomcat as a container and name it prodserver and link with devserver

```
docker run --name prodserver -d -p 7070:8080 --link devserver:jenkins tomcat
```

## Docker Compose:

Docker compose is a feature of docker which is used for creating the micro services architecture where multiple containers are linked with each other. Docker compose uses yaml files for performing this activity. The main advantage is reusability.

Installing docker compose

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

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.24.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

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

Check the version of docker-compose

docker-compose --version

**YML Syntax**

```
---
version: '3'
services:
 <nameofyourcontainer>:
     image:
     ports:
     links:
 <nameofyourcontainer2>:
     image:
     ports:
     links:
...
```

## Usecase-1

Create a docker compose file for setting up a development environment where a mysql container is linked with wordpress container.

vim docker-compose.yml

Go into insert mode by pressing i

```
---
version: '3'
services:
mydb:
  image: mysql:5
  environment:
   MYSQL_ROOT_PASSWORD: techupright
mywordpress:
  image: wordpress
  ports:
   - 5050:80
  links:
   - mydb:mysql
…
```

Save and quit   esc:wq

To start the services from the above docker-compose file,

```
docker-compose up
```

To start in detached mode

```
docker-compose up –d
```

To stop the services

```
docker-compose stop
```

To remove the container

```
docker-compose down
```

## Usecase-2

Create a docker-compose file for creating CI CD environment where a jenkins container is linked with 2  tomcat containers

```
version: '3'
services:
```

```
devserver:
  image: jenkins
  ports:
   - 5050:8080
qaserver:
  image: tomcat
  ports:
   - 6060:8080
  links:
   - devserver:jenkins
prodserver:
  image: tomcat
  ports:
   - 7070:8080
  links:
   - devserver:jenkins
```

## Usecase-3

Create a docker-compose file for establishing the lamp architecture where a mysql container should be linked with httpd container and php container.

```
---
version: '3'
services:
mydb:
  image: mysql
  environment:
   MYSQL_ROOT_PASSWORD: root
apache:
  image: httpd
  ports:
   - 8080:80
  links:
   - mydb:mysql
php:
  image: php:7.2-apache
  links:
   - mydb:mysql
   - apache:httpd
...
```

## Usecase-4

Create a docker compose file for setting up a testing environment where a selenium hub container should be linked with 2 node containers. Firefox node and chrome node.

```
---
version: '3'
services:
hub:
  image: selenium/hub
  ports:
   - 4444:4444
```

```
chrome:
  image: selenium/node-chrome-debug
  ports:
   - 5901:5900
  links:
   - hub:selenium
firefox:
  image: selenium/node-firefox-debug
  ports:
   - 5902:5900
  links:
   - hub:selenium
```

...

Note: docker-compose up by default will search for a file called docker-compose.yml. If the file name is anything else, Ex:abc.yml, then we should give

```
docker-compose –f abc.yml up –d
```

## Docker volumes:

Containers are *ephemeral* but the data that it processes should be persistent i.e. it should be available even though the container is deleted. This can be done using docker volumes. Volume is an external folder or device which is mounted onto the container in such a way that the data stored inside the volume will remain even after the container is deleted.

Docker supports two types of volumes.

1. Simple Docker volume

2. Docker volume container

**Simple Docker volume:**

These volumes are used only for preserving the data even after a container is deleted. But this data cannot be reused by other containers.

## Usecase1

Create a directory called /data. Create a centos container and mount this /data on it as a volume. Create some files in this mounted volume and delete the container. Check if the files are still available

1. Create a directory that will act as a mount point

   ```
   mkdir /data
   ```

2. Create a centos container and mount this /data on it

   ```
   docker run --name c1 –it –v /data centos
   ```

3. In the centos c1 container go into the data folder and create new files

   ```
   cd data
   touch file1 file2 file3
   exit
   ```

4. To Identify the mounted location,

   ```
   docker inspect c1
   ```

   Search for “Mounts” section and copy the “source” path

5. Delete the container

   ```
   docker rm –f c1
   ```

6. Check if the data created in volume is still present

   cd “source path copied from step-4" and ls

**Docker volume containers:**
Simple docker volumes are used only for preserving the data even after the container is deleted. But they cannot be shared between other containers. To create volumes that can be shared between multiple containers we can use docker volume containers.

## Usecase-1

Create 3 containers c1,c2 and c3 and mount /data as volume onto the c1 container. C2 container should use the volume used by the c1 container and c3 container should use the volume used by c2 container. Delete all the 3 containers and check if data is still present.

1. Create a centos container c1 and mount /data as a volume on it

   ```
   docker run --name c1 –it –v /data centos
   ```

2. In the c1 container go into the mounted volume and create few files

   cd data -----> touch file1 file2 file3

   Come out of the container without exit (ctrl+p, ctrl+q)

3. Create c2 container and Mount the c1 volume to c2 container

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

4. In the c2 container go into the mounted volume and create few files

   cd data -----> touch file4 file5 file6

   Come out of the container without exit (ctrl+p, ctrl+q)

5. Create c3 container and Mount the c2 volume to c3 container

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

6. In the c3 container go into the mounted volume and create few files

   cd data -----> touch file7 file8 file9

   Come out of the container without exit (ctrl+p, ctrl+q)

7. Go into any of the above three containers and we will see all the files

```
docker attach c1 (or) c2 (or) c3
ls
exit
```

8. Identify the location where the mounted data is present on host machine

```
docker inspect c1 (or) c2 (or) c3 [go to “mounts” and copy path]
```

9. Delete all the trhee containers

```
docker rm –f c1 c2 c3
```

Check if data is still present on docker host

```
cd “Path_of_source_copied_from_step8”
ls
```

**Creating customized docker images**

This can be done in two ways.

1. Using docker commit command
2. Using dockerfile

**Usecase-1**

Start ubuntu as a container and install git in it. Save this container as an image and delete the container. Now create the container from the new image and it should have git already installed on it.

1. Start ubuntu as a container

```
docker run --name c1 –it ubuntu
```

2. Install Git in it

```
apt-get update
apt-get install –y git
git --version
exit
```

3. Save the above container as an image

   ```
   docker commit c1 myubuntu
   ```

4. Delete the ubuntu container

   ```
   docker rm –f c1
   ```

5. Create a new container from the above image

   ```
   docker run --name c1 –it myubuntu
   ```

6. Check if git is already present

   ```
   git –version
   ```

**Dockerfile**

This is a text-based file which uses predefined keywords using which we can create customized docker images.

Important keywords in dockerfile.

| | |
|---|---|
| **FROM** | **This represents the base image from which we want to create our customized image** |
| **MAINTAINER** | **This is the name of the author or the organization that has created this dockerfile** |
| **CMD** | **This is used for running any process from outside the container** |
| **ENTRYPOINT** | **Every docker container starts the default process and as long as this process is running that container will be in running state. Entrypoint is used for defining what should be that default process** |

| | |
|---|---|
| RUN | This is used for running commands within the container. It is generally used for running commands related to package management. |
| COPY | Used for copying files from host machine to the container. |
| ADD | This can also be used for copying files from host to container. It is also used for downloading files from some remote servers. |
| VOLUME | This is used for attaching an external device or directory directly onto a container as a default volume. |
| USER | This is used to specify who is the default user that should login into the container |
| WORKDIR | This is used to specify the default working directory |
| LABEL | Useful in giving label to the container |
| STOPSIGNAL | This is used to specify the key sequences that should be given to stop the container |
| ENV | This is used for specifying which variables should be passed as environment variables to the container |
| EXPOSE | This is used for specifying which port should be used as internal port of the container. |
| ARGS | Used to pass external arguments to the container |
| SHELL | Used to specify the default shell of the container |

The advantage of using dockerfile over the commit command is, we can perform version controlling on the dockerfiles.

### Usecase-1

Create a dockerfile from nginx base image and specify the maintainer as techupright. Create a new image from this dockerfile.

vim dockerfile

Go into insert mode by pressing i

```
FROM nginx
MAINTAINER TechUpright
```

Save and quit ---> esc :wq

Create an image from the above dockerfile

```
docker build –t mynginx .
```

(.represents current working dir. Mynginx represents new image name. -t represents tag name)

## Usecase–2

Create a dockerfile from centos base image and execute ls –la as a default command of the container

```
vim dockerfile
```

Go into insert mode by pressing I

```
FROM centos
MAINTAINER TechUPright
CMD [“ls”,”-la”]
CMD [“date”]
```

Save and quit Esc :wq

Create an image from the above dockerfile

```
docker build –t mycentos .
```

Create a container from the above image

```
docker run --name c1 –it mycentos
```

We should see the output of date as only one CMD is executed and the last CMD is only executed. So it wont output ls –la

## Usecase–3

Create a dockerfile from ubuntu base image and install git in it

```
vim dockerfile
```

Go into insert mode by pressing I

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

Save and quit   Esc:wq Enter

Create an image from the above docker file

```
docker build –t myubuntu .
```

Start a container from the above image and we will see git already present

```
docker run --name c1 –it myubuntu
git --version
```

### Cache Busting

Whenever we create an image from a dockerfile, docker stores the executed statements in the docker cache. Next time if we try to create an image from the same dockerfile, it will not re-execute the previously executed statements. This is a time saving mechanism of docker. The disadvantage of this process is when the docker file is edited after a huge timegap, we might end up installing softwares from repositories which are updated long time back

EX: If we create a dockerfile with the below instructions,

```
        FROM ubuntu
        RUN apt-get update
        RUN apt-get install –y git
```

If we create an image from the above dockerfile, it will store all these instructions in the cache. Next time, if we take the same dockerfile and add the below statement,

```
RUN apt-get install –y default-jdk
```

It will execute only this new statement when creating the image. This can result in installing java from an apt repository which was updated long time back. To overcome this problem, we can use cache busting.

1.Create a dockerfile for installing git and java using cache busting

Note: whichever statement has && symbol will not be read from the memory. It will be executed again

Go into insert mode by pressing i

```
FROM ubuntu

MAINTAINER techuprightit

RUN apt-get update && apt-get install –y git default-jdk


Docker build  -–no-cache -t ubuntu .
```

Save and quit  Esc:wq enter

2.Create an image from the above dockerfile

```
docker build –t myubuntu .
```

3.Start a container from the above image and we will see git, java already present

```
docker run --name c1 –it myubuntu
git –version
java -version
```

```
# Rebuild the image
docker build --no-cache
```

```
# Pull the base images again and rebuild
docker build --no-cache --pull

# Also works with docker-compose
docker-compose build --no-cache

# If nothing from the above works for you, you could also prune everything
docker system to prune
```

### Usecase – 1

Create a jenkins container and get into bash and install maven

```
docker run --name j1 –d –p 8080:8080 jenkins
docker exec –it j1
```

The default user in jenkins container is jenkins. So we cannot install maven in this case. Using dockerfile, we can create root user and install maven in jenkins container

```
sudo docker exec -u root -it containername /bin/bash
```

Create a dockerfile from jenkins base image. Change the default user as root and install maven in it

```
vim dockerfile
```

```
FROM JENKINS/JENKINS
MAINTAINER techupright
USER root
RUN apt-get update
RUN apt-get install -y maven
```

Create an image from the above dockerfile

```
docker build -t myjenkins .
```

Start a container from the above image and we will see user as root and maven installed on it

```
docker run --name j1 -d -p 8080:8080 myjenkins
git --version
mvn --version
```

**Usecase-2**

Create a dockerfile with ubuntu as base image and have file in it

Start an ubuntu container which contains jenkins.war file in it

```
        FROM ubuntu
        MAINTAINER techupright
        ADD http://mirrors.jenkins.io/war-stable/latest/jenkins.war /
```

Create an image from the above docker file

```
docker build –t myubuntu .
```

Create a container from the myubuntu image

```
docker run --name myubuntujenkins -it myubuntu
```

In the interactive terminal, go to root account and check the files ls – you will find jenkins.war copied to root folder

## Usecase-3

Create a dockerfile from httpd base image and expose a new port 9090.

```
FROM httpd
MAINTAINER techupright
EXPOSE 9090
```

Save and quit

Create an image from the above dockerfile

```
docker build –t myhttpd .
```

Create a container and check the ports exposed

```
docker run –d --name h1 –P myhttpd
docker container ls
```

## Usecase-4

Create a dockerfile from centos base image and mount /data as the default volume. Create some files within the volume in the container. Delete the container and check if the data is present.

```
FROM centos
MAINTAINER techupright
VOLUME /data
```

Save and quit

Create an image from the above dockerfile

```
docker build –t mycentos .
```

Create a container and create some files in the container

docker run --name c1 –it mycentos

cd data

touch file1 file2 file3

exit

Identify the mount location

docker inspect c1

Search for “Mounts” section and copy the “source” path

Delete the container

docker rm –f c1

Check if the data is still present in the source path

cd “source”_ path_copied_from_“mounts”_section and ls

## Usecase-5

Create a dockerfile from ubuntu base image and make execution of java –jar jenkins.war as a default process.

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

Save and quit

Create an image from the above dockerfile

docker build –t myubuntu .

Create a container and it will start jenkins

```
docker run --name c1 –it myubuntu
```

We should see the logs of jenkins

## Usecase-6

Create a dockerfile from ubuntu base image and install ansible in it

```
FROM ubuntu
MAINTAINER techupright
RUN apt-get update
RUN apt-get install -y software-properties-common
RUN apt-add-repository ppa:ansible/ansible
RUN apt-get update
RUN apt-get install -y ansible
```

Create image from the above dockerfile

```
docker build –t ansible .
```

Start a container from this image and we will see ansible present in it

```
docker run --name ansible –it ansible
ansible –version
```

Create a shell script which uses the above docker file and creates 5 ansible images.

```
vim script1.sh
#!/bin/bash
for i in {1..5}
do
  docker build -t ansible$i .
done
```

## Docker Networking

Docker supports four types of networks

1. Bridge Network
2. Host Network
3. Null Network
4. Overlay Network

**Bridge Network:** This is the default network of docker when multiple containers run on the same host network machine, we use bridge network

**Host Network:** This is also called as host only network and this is used when we want to create containers which will communicate only with the host machine and not with other containers

**Null Network:** This is also called as None network. This is used when we want to create isolated containers which cannot communicate with the host machine or with other containers. This is used in docker security for preserving sensitive data

**Overlay Network:** This is used when docker containers are running in a distributed environment on multiple servers and they want to communicate with each other. This is used in docker swarm.

### Usecase –1 :

Create 2 bridge networks devops1and devops2. Create 3 busybox containers c1, c2 and c3. C1 and c2 should run on devops1 network and they should communicate with each other.  C3 should run on devops2 network and it should not communicate with either c1 or c2.

Later attach c2 to devops2 network. Because c2 is present on both the networks, it should communicate with c1 and c3. But c1 and c3 should not communicate with each other directly.

Steps:

1. Create 2 bridge networks

```
docker network create --driver bridge techupright1
docker network create techupright2 (default network is bridge)
```

2. Check the list of available networks

```
docker network ls
```

3. Create a busybox container c1 on techupright1 network

```
docker run --name c1 -it --network techupright1 busybox
```

come out of the c1 container without exit ctrl+p,ctrl+q

4. Identify the ipaddress of c1

```
docker inspect c1
```

5. Create another busybox container c2 on techupright1 network

```
docker run --name c2 -it --network techupright1 busybox
ping ipaddress of c1 (it will ping)
```

come out of the c2 container without exit ctrl+p,ctrl+q

6. Identify the ipaddress of c2

```
docker inspect c2
```

7. Create another busybox container c3 on techupright2 network

```
docker run --name c3 -it --network techupright2 busybox
ping ipaddressofc1 (it should not ping)
ping ipaddressofc2 (it should not ping)
```

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

8. Identify the ipaddress of c3

```
docker inspect c3
```

9. Now connect techupright2 network to c2 container

```
docker network connect techupright2 c2
```

10. Since c2 is now on both techupright1 and techupright2 networks, it should ping to both c1 and c3 containers

```
docker attach c2
ping ipaddressofc1 (it should ping)
ping ipaddressofc3 (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 ipaddressofc1 (it should not ping)
```

**Working on Docker Registry**

Docker registry is of two types.

1. Public Registry (hub.docker.com)
2. Private Registry

**Public registry** (hub.docker.com) and images pushed into this registry can be accessed by anyone

**Private registry** is created within our organization servers and the images pushed into this registry can be accessed only by our organization.

**Uploading images into public registry.**

Open hub.docker.com

Sign up for a free account (analyticsbd – my account)

Create a customized docker image

A) docker run –name u1 –it ubuntu

B) Install apache2 in this ubuntu container

```
apt-get update
apt-get install –y apache2
exit
```

C) Save the container as an image

```
docker commit u1 analyticsbd/ubuntu_apache
```

Login into hub.docker.com

docker push analyticsbd/apache2_ubuntu

docker login

Push the docker image

docker push analyticsbd/ubuntu_apache

**Working on Local Registry**

Customized docker images can be uploaded into a local registry from where only the organization team members can access

Usecase-1

Upload an alpine Linux image into the local registry

Start registry as a container

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

Note: The above registry image is provided by the docker community and once it starts as a container it will create a local registry

Download alpine image into our docker host

```
docker pull alpine
```

Tag the image with the local registry

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

Push into the local registry

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

## Container Orchestration

This is a process where we run docker containers in a distributed environment. These containers running on multiple servers should communicate with each other and handle all the production environment issues.

Popular container orchestration tools

1. Docker Swarm
2. Kubernetes
3. Apache Mesos
4. Redhat Openshift

Advantages of container orchestration

1. Load balancing and high availability
2. Scaling of services up or down
3. Performing rolling updates
4. Handling failover scenarios or disaster recovery

### Docker Swarm

### Setup of docker swarm

1. Create 3 AWS instances of ubuntu 20.04
2. Install docker on all of them
3. Change the hostname

   vim /etc/hostname

   Remove the data in this file and replace with

   Manager/Worker1/Worker2

   Save and quit
4. Restart the servers

   init 6
5. Connect to Manager using git bash
6. To setup docker swarm

```
docker swarm init --advertise-addr private_ip_of_manager
```

This command will create the current machine as manager and it will also generate the token-id that we can paste in other machines to join swarm as workers.

**Ports to be opened for docker swarm cluster:**

TCP port 2376 for secure Docker client communication. This port is required for Docker machine to work. Docker Machine is used to orchestrate Docker hosts.

TCP Port 2377. This port is used for communication between the nodes of a Docker Swarm or cluster. It only needs to be opened on manager nodes.

TCP and UDP port 7946 for communication among nodes (container network discovery)

UDP port 4789 for overlay network traffic (container ingress networking).

**Load Balancing in Docker swarm**

Each container running in docker swarm has the capability of withstanding a specific user load without breaking the SLA’s (service level agreements).  When we want to handle bigger user loads, we can create multiple replicas of the same service and deploy them in swarm cluster.

## Usecase-1

Create tomcat with 4 replicas in swarm and check if these replicas are distributed on managers and workers.

1. Create tomcat with 4 replicas in swarm

```
docker service create --name webserver -p 9090:8080 --replicas 4 tomcat
```

2. To see the home page of the tomcat service that is distributed in swarm,

Launch any browser, public_ip_of_manager/worker1/worker2:9090

3. To see the list of nodes where these 4 tomcat replicas are running

```
docker service ps
```

4. To see a specific servicedo,

```
docker service ps <service_name> eg. mydb, webserver etc.,
```

## Usecase-2:

Start  mysql5 with 3 replicas and check the nodes on which these replicas are running.

1. Start mysql with 3 replicas

```
docker service create --name mydb -e MYSQL_ROOT_PASSWORD=techupright --replicas=3 mysql:5
```

2. To check on which nodes these three replicas of mysql are running

```
docker service ps mydb
```

To see the list of services deployed in swarm

```
docker service ls
```

To get detailed information about any service

```
docker service inspect service_name/service_id
```

This command will show the service info in JSON file format

To display the output in normal format

```
docker service inspect service_name/service_id --pretty
```

To delete a service

```
docker service rm service_name/service_id
```

### Scaling of replicas in swarm

Depending on the business requirement, we should be able to increase the replica count or decrease the count without experiencing any downtime.

Create httpd with 5 replicas and then scale it to 9. Later scale down to 3.

1. Start httpd with 5 replicas in swarm

```
docker service create --name appserver -p 8888:88 --replicas 5 httpd
```

2. To see the list of nodes where these replicas are running

```
docker service ps appserver
```

3. To scale the replcias count to 9

```
docker service scale appserver=9
```

4. Check if 9 replicas are running in the cluster

```
docker service ps appserver
```

5. To scale down the replicas count to 3

```
docker service scale appserver =3
```

6. Check if only 3 replicas are running

```
docker service ps appserver
```

**Performing Rolling updates**

The services running in docker swarm can be upgraded to a higher version or roll back to an older version without the enduser experiencing any downtime. This is done by docker by affecting the update operation on one replica after another.

Usecase-1

Start redis:3 with 5 replicas in dockerswarm and upgrade to redis4. Later roll back to redis:3

1. Start redis:3 with 5 replicas in swarm

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

2. Check if 5 replicas are running in swarm with redis:3

```
docker service ps myredis
```

3. Perform a rolling update to redis:4

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

4. Check if 5 replicas of redis:4 are running and redis:3 is shut down

```
docker service ps myredis
```

5. Perform a rolling rollback from redis:4 to redis:3

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

6. Check if 5 replicas of redis:3 are running and redis:4 has shut down

```
docker service ps myredis
```

**Docker Cluster:**

To see all the nodes in the cluster

```
docker node ls
```

Manager status: Leader

Removing the node from the docker swarm cluster from the level of manager [availability changes : active -> drain ]

```
docker node update --availability drain <node_name>
```

```
docker node update --availability drain worker1
```

To make this node rejoin the docker swarm,

```
docker node update --availability active <node_name>
```

```
docker node update --availability active worker1
```

If the worker wants to leave docker swarm, go to the worker machine and give the command

```
docker swarm leave]
```

1. To generate the token id for nodes to join as workers

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

2. To generate the token id for nodes to join as manager

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

3. To promote a node(worker-1) and make him as manager

```
docker node promote worker1
```

The status become reachable

4. To demote a manager (worker-1) and make him as worker

```
docker node demote worker1
```

The status reachable gets removed

5.If the manager wants to leave forcibly

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

Handling Failover scenarios

**Usecase-1**

Create tomcat with 5 replicas

Delete 1 replica and check if all 5 replicas are still running

Drain a worker 1 from swarm and check if the replicas are running on worker1 have migrated to manager and worker2.

Make worker1 rejoin the docker swarm

Go to worker2 and make it leave the swarm. Check if all replicas from worker2 have migrated to manager and worker1

1. Start tomcat with 5 replicas in swarm

```
docker service create --name appserver -p 8888:8080 --replicas 5 tomcat
```

2. To check if 5 replicas are running on the swarm cluster

```
docker service ps appserver
docker service ps appserver | grep running
```

3. To delete a replica from manager

```
docker container ls
```

Select the container id of the replica that we want to delete

```
docker rm -f container_id
```

4. Check if we still have 5 replicas running in the swarm cluster

```
docker service ps appserver
```

5. Drain worker1 from the swarm

```
docker node update --availability drain worker1
```

6. Check if 5 replicas are still running on Manager and worker2

```
docer service ps appserver
```

7. Make worker1 rejoin the swarm

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

8. Go to worker2 and make it leave swarm

Connect to worker2 using git bash

```
docker swarm leave
```

9. Check if 5 replicas of tomcat are still running on Manager and worker1

connect to manager using git bash

```
docker service ps appserver
```

To remove appserver service,

```
docker service rm appserver
```

**Overlay Networking**

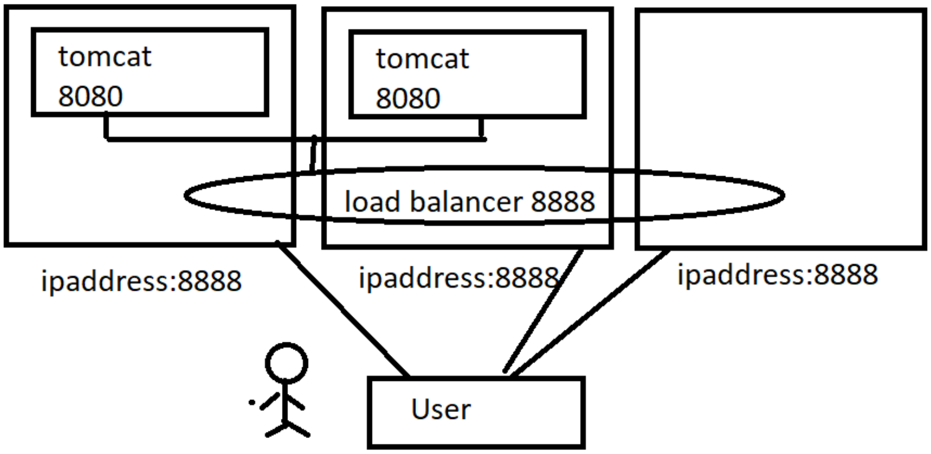
This is a default network that docker swarm uses. This helps in communication between containers running on different host machines. The default overlay network is called as ingress network.

**Usecase:**

If we have 3 machines in the swarm cluster, and we start tomcat with 2 replicas with the below command

```
docker service create –name webserver –p 8888:8080 –replicas 2 tomcat
```

What docker swarm does is, it will create these two replicas and expose their 8080 port to a network load balancer. This network load balancer will route to port number 8888 from where it is exposed to the external world.



Though the tomcat server is not present on the third server, still when the request comes to the third server, it will be routed to the tomcat+ container automatically. This is the purpose of using ingress network.

**Usecase-1**

Create 2 overlay networks techupright1 and techupright2. Start tomcat with 3 replicas on techupright1 network. Start nginx with 2 replicas on the default overlay network and later assign it to techupright2 network (rolling network update)

1. Create 2 overlay networks

```
docker network create --driver overlay techupright1
docker network create --driver overlay techupright2
```

2. To see the list of all networks

```
docker network ls
```

3. Create tomcat with 3 replicas on techupright1 network

```
docker service create --name webserver -p 8888:8080 --replicas 3
--network techupright1 tomcat
```

4. To check if tomcat service is running on techupright1 network

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

5. Start nginx with 2 replicas

```
docker service create --name appserver -p 9999:80 --replicas 2 nginx
```

6. Perform a rolling network update for nginx to techupright2

```
docker service update --network-add techupright2 appserver
```

7.To check if nginx service is running on techupright2 network

```
docker service inspect appserver –pretty
```

docker-compose+swarm = stack

docker-compose+kubernetes = kompose

**Docker Stack**

A stack is a group of inter-related services that share dependencies with each other in such a way that they can be scaled and orchestrated together.  A single stack can define the functionality of the complete application.

To deploy a stack in *swarm*

```
docker stack deploy –c docker-compose_filename/stack_filename stackname
```

To see the list of all the stacks

```
docker stack ls
```

To see the list of nodes where the stack services are running

```
docker stack ps stackname
```

To delete a stack

```
docker stack rm stackname
```

To see the list of services in a stack

```
docker stack services
```

Usecase-1

Create a stack file for starting wordpress with 3 replicas and link it with 1 replica of mysql.

```
vim stack1.yml
---
version: '3'
services:
mydb:
  image: mysql:5
  environment:
   MYSQL_ROOT_PASSWORD: techupright
mywordpress:
  image: wordpress
  ports:
   - 5050:80
  deploy:
   replicas: 3
…
```

To deploy this stack file in swarm

```
docker stack deploy –c stack1.yml wordpress_mysql
```

To see the list of nodes where these stack services are running

```
docker stack ps wordpress_mysql
```

To delete the stack

```
docker stack rm wordpress_mysql
```

**Usecase-2**

Create a stack file for starting jenkins with 1 replica on the manager tomcat as qaserver with 2 replicas on worker-1 and tomcat as prodserver with 3 replicas on worker-2

```
---
version: '3'
services:
devserver:
  image: jenkins
  ports:
   - 5050:8080
  deploy:
   placement:
    constraints:
     - node.hostname == Manager
qaserver:
image: tomcat
ports:
  - 6060:8080
deploy:
  replicas: 2
  placement:
   constraints:
    - node.hostname == Worker1
prodserver:
image: tomcat
ports:
  - 7070:8080
deploy:
  replicas: 3
  placement:
   constraints:
    - node.hostname == Worker2
```

Save and quit

To deploy this stack in swarm

```
docker stack deploy -c stack2.yml ci-cd
```

To see the list of nodes where these stack services are running

```
docker stack ps ci-cd
```

To delete the stack

```
docker stack rm ci-cd
```

---

Docker stack with limited resources

Create a stack file for setting up the lamp architecture where mysql runs with 2 replicas. Each replica should not consume beyond 100mb of ram and 0.01 cpu.

```
---
version: '3'
services:
mydb:
  image: mysql
  environment:
   MYSQL_ROOT_PASSWORD: techupright
  deploy:
   replicas: 2
   resources:
    limits:
 cpus: "0.01"
 memory: 100M
apache:
  image: httpd
  ports:
   - 9999:80
  deploy:
   replicas: 3
   resources:
    limits:
 cpus: "0.02"
 memory: 150M
php:
  image: php:7.2-apache
  deploy:
   resources:
    limits:
 cpus: "0.01"
 memory: 150M
…
```

```
docker stack ls
```

docker stack deploy –c stack3.yml lamp

To specify a range for the network specify a subnet range and assign this network to a service in swarm.

Create a overlay network with a subnet range

1. docker network create –driver overlay –subnet 172.168.0.1/24 techupright

Create a service with 5 replicas on techupright1 network.

2. docker service create –name webserver –replicas 5 –network techupright1 tomcat

Dind – in hub.docker.com (Docker in Docker)

## Kubernetes

This is also a container orchestration tool similar to docker swarm. It was a product of google but currently its an opensource tool. Kubernetes can be used for handling all the production related issues like high availability, load balancing, scaling, performing rolling updates, disaster recovery etc.,

Pod: This is the smallest kubernetes object and it is used for storing containers . Kubernetes does not deploy docker containers directly instead it deploys the containers within the pods. A single pod can contain one or more containers. But generally containers and pods share one to one relationship.

To create a kubernetes cluster,

Signup for a gcloud account

Signin into that account

Click on menu icons on theh top left corner

Click on kubernetes engine

Click on create a project --> enter some project name

Click on create cluster --> Enter some cluster name

Click on create

To connect to this cluster --> Click on connect --> Copy the google cloud connect command --> paste in the google cloud shell seen at the top right corner

====================================================

To see the list of nodes in the cluster

```
kubectl get nodes
```

To see the ipaddresses(public and private) of the nodes

```
kubectl get nodes –o wide
```

To get detailed info about each node

```
kubectl describe nodes node_name
```

====================================================

**UseCase:**

Start nginx as a pod in the kubernetes cluster and name the pod as a webserver

```
kubectl run --image nginx webserver
```

To see the list of pods

```
kubectl get pods
```

To get detailed info about the pod

```
kubectl describe pods pod_name
```

To delete the pod

```
kubectl delete pods pod_name
```

UseCase:

Start tomcat in the kubernetes cluster with 3 replicas and name it appserver.

```
kubectl run --image tomcat appserver --replicas=3
```

```
kubectl get pods
```

To see the list of pods related to tomcat

```
Kubectl get  pods –o wide | grep appserver
```

UseCase

Start mysql in the kubernetes cluster with 2 replicas

```
Kubectl run –image mysql:5 mydb --env MYSQL_ROOT_PASSWORD
–replicas 2
```

To see the list of pods related to mydb

```
kubectl get pods –o wide | grep mydb
```

To delete kubectl appserver completely from the cluster

```
kubectl delete deployment appserver
```

To scale the above mysql from 2 replicas to 4

```
kubectl scale deployments/mydb --replicas=4
```

To see all the deployments in kubernetes cluster

```
kubectl get deployment
```

Implementing kubernetes objects using yaml files

Each yaml files contain 4 top level fields

```
apiVersion:
kind:
metadata
spec:
```

apiVersion: This is the verison of kubernetes api that is used for creating the objects.

kind: This is used to specify the type of kubernetes object that we want to create.

metadata: This contains information like names, labels etc., Labels is again a dictionary object which can contain any key value pairs.

spec: This contains exact information about docker images, container names, port mapping env variables etc.,

| Version | Kind |
|---|---|
| v1 | Pod |
| v1 | Service |
| v1 | Replication Controller |
| apps/v1 | ReplicaSet |
| apps/v1 | Deployment |

Create a pod definition file for creating a pod with a name mywebserver. Within this pod it should deploy a httpd container with the name myhttpd.

```
---
apiVersion: v1
kind: Pod
metadata:
name: mywebserver
labels:
  author: techupright
  type: frontend
spec:
containers:
  - name: myhttpd
    image: httpd
…
```

To create the pods using the above files

```
Kubectl create –f pod-definition.yml
```

To see the list of pods created in the kubernetes cluster

```
Kubectl get pods –o wide
```

To delete the pods created using the above file

```
Kubectl delete –f pod-definition.yml
```

Create a pod definition file which will start postgres as a container and the container name should be mypostgres. The pod name should be postgres. In the labels specify the type as backend and author as prashanth.

```
---
apiVersion: v1
kind: Pod
metadata:
name: postgres #name of the pod
labels:
  author: prashanth
  type: backend
```

```
spec:
containers:
  - name: mypostgres
    image: postgres
…
```

```
Kubectl create –f kubeplay2.yml
```

To see the list of pods created in the kubernetes cluster

```
Kubectl get pods –o wide
```

To delete the pods created using the above file

```
Kubectl delete –f kubeplay2.yml
```

USeCase4:

Start mysql file as acontainer and name the container as mydb. Pass the necessary environment variables and name the pod as mysqldb.

```
---
apiVersion: v1
kind: Pod
metadata:
name: mysqldb
labels:
  author: Prashanth
spec:
containers:
  - name: mysql
    image: mysql:5
    env:
     - name: MYSQL_ROOT_PASSWORD
       value: hellothere
…
kubectl describe pods mysqldb
Kubectl create –f poddefinition3.yml
Kubectl delete –f pod-definition3.yml
```

Create a pod definition file by starting :

Start jenkins as a container and the container name should be myjenkins. Add The jenkins port 8080 with 5050 on the host and the name of pod as DevServer.

```
---
apiVersion: v1
kind: Pod
metadata:
name: devserver
labels:
  author: Prashanth
spec:
containers:
  - name: myjenkins
    image: jenkins
    ports:
     - containerPort: 8080
       hostPort: 5000
…
```

Replication controller is the next level of kubernetes object and this is used not only for deploying pods.

Create a replication controller file where the name of the replication controller is rc-tomcat. It should contain three pods with the name tomcat-pod and this pod should have the tomcat container with the name called webserver.

```
---
apiVersion: v1
kind: ReplicationController #not pod
metadata:
name: rc-tomcat
spec:
replicas: 3
template:
  metadata:
   name: tomcat-pod
   labels:
    author: techupright
    type: frontend
  spec:
   containers:
    - name: webserver
      image: tomcat
      ports:
      - containerPort: 8080
```

```
        hostPort: 7070
...
```

**Replica Set:**

A replica set is similar to a replication controller and this object is an encapsulation on the pod. i.e in the replica set we have pod and in the pod we have containers. But the replica set has an additional field called as selector which helps in searching for pods based on specific label and add them to the replica set object.

Create a replicaset file which creates a replica set called rc-httpd. Within this replicaset it should run 4 pods with the name httpd-pod and within the pod it should run httpd with the container name appserver.

```
---
apiVersion: apps/v1
kind: ReplicaSet
metadata:
name: rc-httpd
labels:
  author: techuprightit
  type: frontend
spec:
replicas: 4
selector:
  matchLabels:
   type: frontend
template:
  metadata:
   name: httpd-pod
   labels:
    type: frontend
  spec:
   containers:
    - name: appserver
      image: httpd
…
```

To scale the number of replicas, open the replica set file and change the replica count from 4 to 6. For these changes to get affected

```
Kubectl replace –f <filename>
```

To change the replica count without modifications in the file..

```
Kubectl scale –replicas=8 –f <filename>
```

**Implementing docker-compose in kubernetes**

This can be done by installing a software called Kompose.

https://github.com/kubernetes/kompose/blob/master/docs/installation.md

Install chcolatey and install kompose

```
---
version: '3'
services:
mydb:
  image: u
  environment:
   POSTGRES_PASSWORD: techupright
webserver:
  image: nginx
  ports:
   - 8899:80
…
```

Kompose file to create the CICD environment

```
---
version: '3'
services:
devserver:
  image: jenkins
  ports:
   - 5050:8080
qaserver:
  image: tomcat
  ports:
   - 6060:8080
prodserver:
  image: tomcat
  ports:
   - 7070:8080
…
```

Kompose-up / kompose-down

Kubectl get all/  kubectl get pods

Deployment:

This is a kubernetes object which is higher than a replica set and it can perform all activities like load balancing, scaling and rolling updates.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
name: nginx-deployment
labels:
  author: techuprightit
  type: frontend
spec:
replicas: 3
selector:
  matchLabels:
   type: frontend
template:
  metadata:
   labels:
    type: frontend
  spec:
   containers:
    - name: mywebserver
      image: nginx:1.7.9
      ports:
       - containerPort: 80
         hostPort: 9090
…
```

```
Kubectl create –f deployment-definition.yml
Kubectl get pods
Kubectl describe pods <podname>
```

To upgrade nginx from 1.7.9 to 1.9 version,

```
kubectl --record deployment.apps/nginx-deployment set image deployment.v1.apps/nginx-deployment mywebserver=nginx:1.9.1
```

To delete the definition file

```
Kubectl delete –f deployment-definition.yml
```

**Services**:

This is a kubernetes object which is used if we want to map the pods running within the kubernetes cluster with the external world. Service object works like a router which enables us to connect the pods with the service object and from the service object to the external world.

The service object works like a router which enables us to connect the pods with the service object and from the service object to the external world.

Create a service definition file to map tomcat container port 8080 with the service port 8080 and the node port 30008

```
---
```

```
apiVersion: v1
kind: Service
metadata:
name: app-service
spec:
type: NodePort
ports:
  - targetPort: 8080
    port: 8080
    nodePort: 30008
selector:
  tier: frontend
```

…

Target port is the container port

Port is service port

Node port is the Linux server port

```
Kubectl create –f pod-definition.yml
Kubectl crate –f service-definition.yml
```

To open a port on the gcloud cluster

Gcloud compute firewall