Creating customized docker Images: This can be done in 2 ways

1. Using docker commit command
2. Using docker file

Docker commit:

Use case: Create an Ubuntu container and install git and java in it, Save this container as an image and then delete the container.

Create a new container from the saved image and check if git and java are already installed in it

1. Create an Ubuntu Container

```
docker run --name U1 -it ubuntu
```

2. In the Ubuntu U1 Container install git and java

```
apt-get update
apt-get install -y git default-jdk
```

3. Check if git and java are installed

```
git --version
java -version
exit
```

4. Save the container as an image

```
docker commit U1 myubuntu
```

5. Delete the Ubuntu U1 Container

```
docker rm -f U1
```

6. Create a new container from the newly created image

```
docker run --name U1 -it myubuntu
```

① We should see git and java already installed.

git --version

java -version

Docker Session 10 (9-03-20):

Docker file: This is a text based file which can be used for creating customized docker images. This file uses certain predefined keywords using which docker images are created.

Important keywords used in Docker File:

① FROM: This is used to specify the base image from which customized docker image should be created.

② MAINTAINER: This is used to specify the name of the Organisation or the Author who created this docker file.

③ CMD: This is used for running an application in a container by keeping the control outside the container.

④ ENTRYPOINT: Every docker container triggers a default process when it runs. We can specify what should be the default process using this entry point instruction

⑤ RUN: This is used generally for running commands in a container.
ex: updating repositories, Installing softwares - etc

⑥ EXPOSE: This is used to open a port in the container. This port can be mapped with some other port on the host machine.

7. VOLUME : This is used for Mounting an External folder as a default volume when the container starts.

8. COPY : Used for copying files from host machine to the container

9. ADD : This can also be used for copying files from host to container. It can also be used for downloading files from an external server into the Container.

10. USER : This is used to specify who should be the default used that should login into the Container.

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

12. LABEL : This is used for ingiving default LABEL to the container.

13. STOPSIGNAL : This is used to specify the key sequence that should be passed to stop the Container.

14. ENV : This is used to specify which variable should be passed as Environment variables.

15.

⇒ Create a docker file from nginx base image and specify the maintainer as intelliq

① vim dockerfile

```
FROM nginx
MAINTAINER intelliqit
:wq
```

② Create an image from the above file

```
docker build -t myngingx .
```

myngingx → new image name
(.) → current working directory

③ Check if the image is created or not

```
docker images
```

⇒ Create a docker file from ubuntu base image and install git and java in it

① vim dockerfile

```
FROM ubuntu
MAINTAINER intelliq
RUN apt-get update
RUN apt-get install -y git default-jdk
:wq
```

② Create an image from the above file

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

③ Check if the image is created or not

```
docker images
```

④ Create a container from the above image and check if git and java are present

```
docker run --name U1 -it myubuntu
git --version
java -version
```

Cache Busting: Whenever we create an image from a docker file docker stores all those instructions in its memory called as docker cache. If we modify the docker file and try to create an image it will read the previous instructions from the cache and only the new instructions will be executed. This is a time saving mechanism of docker.

ex:

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

If we create an image from the above docker file docker will store all those instructions in the cache. and it will execute only those instructions that are added later.

To over come this we can use Cache Busting by using (&&) whichever instruction has && will not be read from memory but it will be executed.

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

Docker Session 11 (10-03-20)

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

① Vim dockerfile

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

:wq

② create an image from a dockerfile

docker build -t ansible .

③ Create a container from the above image

docker run --name a1 -it ansible

④ Check if ansible is present or not-

ansible --version

⇒ create a dockerfile from ubuntu base image and download jenkins.war into ~~[illegible]~~ it.

① Vim dockerfile

```
FROM ubuntu
MAINTAINER intelliq
ADD http://                    /tmp
```

② create an image from a dockerfile

docker build -t myubuntu .

③ create a container from the above image

docker run --name u1 -it myubuntu

④ Check if Jenkins is present in it

cd tmp

ls

⇒ Create a dockerfile from jenkins base image and make the default user as root. Install git and maven in it.

① Vim dockerfile

FROM jenkins

MAINTAINER intelliq

USER root

RUN apt-get update

RUN apt-get install -y git maven

:wq

② Create an image from a dockerfile

docker build -t myjenkins .

③ Create a container from the above image

docker run --name j1 -d -P myjenkins

④ Open interactive terminal in the container and check if the default user is root and also if git and maven are installed

docker exec -it j1 bash

whoami

git --version

mvn --version

11-03-20 Class cancelled

Docker Session12(12-03-20):

⇒ create a dockerfile from ubuntu base image install java on it and make execution of "java -jar jenkins.war" as default process.

(1) Vim dockerfile

```
FROM ubuntu
MAINTAINER intellig
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"]
```

:wq!

(2) create an image from the above file

docker build -t myubuntu .

(3) Create a container from above image

docker run --name v1 -it myubuntu

we should see the logs of jenkins coming up

(4) To check the default process of the above container

docker container ls

We will see "java -jar /jenkins.war" as the default process instead of the regular /bin/bash

⇒ Create a dockerfile from ~~nginx~~ Ubuntu base image and make nginx as default process of this docker file.

① Vim dockerfile

```
FROM ubuntu
MAINTAINER intellig
RUN ~~sg~~ apt-get update
RUN apt-get install -y nginx
ENTRYPOINT ["/usr/sbin/nginx", "-g", "daemon off;"]
EXPOSE 80
:wq!
```

② Create image from the above file

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

③ Create a container from above file

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

④ To check the default process of the above container

```
docker container ls
```

We will see "nginx -g daemon off" as the default process instead of the regular /bin/bash

⇒ Create a dockerfile from centos base image and install httpd in it. Make execution of httpd as default process.

① Vim dockerfile

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

:wq!

⇒ Create a dockerfile from centos base image and it should mount /data as default volume.
Create a container from this image and create some files in container in /data folder. Delete the container and check if the files are still available on host machine.

① Vim dockerfile

```
FROM Centos
MAINTAINER intelliq
VOLUME /data
```

② Create an image from the above file

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

③ Create a container from the above image

docker run --name c1 -it mycentos

④ Go to data folder and create files

cd /data

touch file1 file2

⑤ Identify the location where the mounted files are present

docker inspect c1

Go to "Mounts" section and copy the Source path

⑥ Delete the container

docker rm -f c1

⑦ Check if the files are still available

cd "source_path_copied_from_step 5"

Docker Session 13 (13-03-20):

Docker Networking: Docker uses 4 types of Networks

① Bridge ② Host Only ③ Null ④ Overlay

Bridge Networks: When multiple containers are created on the same host machine. They will automatically runs on the Bridge Network. This is the default Network of docker

Host Network: Containers Running in the host network can Communicate only with the host machine and not with other Containers.

Null Network : This is used for creating isolated containers. i.e, These containers cannot communicate with the host machine or with other containers

Overlay Network: This is used when docker containers are running in a distributed Environment. i.e; on multiple servers and they want to communicate with Each other. Then we can use Overlay Network. This is used in Container orchestration.

Use Case

→ Create 2 bridge networks Intelliq1 and Intelliq2

Create 3 busybox containers C1, C2 and C3

C1, C2 should run on Intelliq1 network. So they should ping each other.

C3 should run on Intelliq2 network so it should not ping to Either C1 or C2 Containers.

Now connect C2 Container to intelliq2 network. Because C2 container is present on both the Networks, it should be able to communicate with C1 and C3. But C1 and C3 should not communicate with Each other directly.

① Create 2 bridge Networks

```
docker network create --driver bridge intelliq1
docker network create --driver bridge intelliq2
```

② Check the list of available networks

```
docker network ls
```

(3) Create a busybox container C1 on intelliq1 network

docker run --name C1 -it --network intelliq1 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 intelliq1 network

docker run --name C2 -it --network intelliq1 busybox

Ping ipaddress_of_C1 (It will ping)

(6) Identify the ipaddress of C2

docker inspect C2

(7) Create another busybox container C3 on intelliq2 network

docker run --name C3 -it --network intelliq2 busybox

Ping ipaddress_of_C1 (It should not ping)

C2

come out of the C3 Container without Exit Ctrl+P, Ctrl+Q

(8) Identify the ipaddress of C3

docker inspect C3

(9) Now attach intelliq2 network to C2 Container

docker network connect intelliq2 C2

(10) Since C2 is now on both intelliq1 and intelliq2 networks it should ping to both C1 and C3 Containers

docker attach C2

Ping C1 (It should ping)

Ping C3 (It should ping)

come out of C2 Container without Exit Ctrl+P, Ctrl+Q

(11) But C1 and C3 should not ping each other

docker attach C3

ping ipaddress_of-

Working on Docker Registry:

Registry is the location where all the docker images are saved. This is of 2 types

Public
Private

→ Public registry is hub.docker.com

→ Private registry is setup within our own organisation servers. And it can be accessed by our team members only.

Uploading images into Public Registry:

Open hub.docker.com and signup for a free account

Create a customized docker image

(a) start ubuntu as a container

docker run --name U1 -it ubuntu

(b) Install git in it

apt-get update

apt-get install -y git

exit

(c) Save this container as an image

docker commit U1 intelliqit/myubuntu

dockerID

(d) Login into dockerhub

docker login

(e) Push the image

docker push intelliqit/myubuntu

Docker 14th Session (14-03-20)

Working on Local Registry: Local registry can be setup using the docker image called registry. If we start this registry as a container, we will be able to upload images and download images from it.

⇒ Start registry image as a container, tag an alpine image with this local registry and push the image into the local registry.

① Start registry as a container

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

② Tag alpine image (links the image with the local registry)

docker tag alpine localhost:5000/alpine

③ Push the image onto local registry

docker push localhost:5000/alpine

Container Orchestration: This is a process of running docker containers in a distributed Environment on multiple servers. ~~Throug~~ Through Container Orchestration we can handle the production related challenges like High availability, Load Balancing, Scaling, Rolling updates, Disaster recovery .. etc.

Load Balancing: Each docker container is capable of withstanding a specific user load. We can increase the count of these replicas on the Run time and handle higher loads.

Scaling: Based on the business requirement we should be able to scale up or Scaledown the no. of containers without the End user experiencing any down time.

Rolling Updates: The services running in a production Environment should be upgraded or downgraded without downtime for a End user.

Disaster Recovery: If a container stops or if entire server where the containers are running is crashes still we should be able to maintain the desired count of Containers on the remaining servers. to acheive high availability.

Popular container Orchestration Tools:

1. Docker Swarm
2. Kubernetes
3. Apache Mesos
4. Open Shift

Setup of Docker Swarm: Create 3 ubuntu aws instances

1. Create 3 Ubuntu18 AWS instances
2. Name the first one as Manager and others as workers
3. Install docker in all of them
4. Change the hostname (not mandatory)

   Vim /etc/hostname

   Remove the data in the file and replace it manager (or) worker1 (or) worker 2

⑤ Restart the AWS instances

init 6

Install docker swarm:

ⓐ Connect to the Manager AWS Instances

docker swarm init --advertise-addr private_ip_of_Manager

This command will convert the current machine as a Manager and it will also generate a token id. Copy and paste the token id in the worker AWS instances.

2 Ports to be opened on AWS instances for docker swarm to work

- 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 communicate 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:

Start nginx with 5 replicas in docker swarm check if these replicas are distributed on Manager and workers. Also check if the nginx homepage can be accessed from Manager and Workers.

① Start nginx with 5 replicas in swarm

docker service create --name webserver -p 8989:80 --replicas 5 ngi

② To check on which node the 5 replicas of nginx are running

docker service ps webserver

③ To access these replicas from Manager or Workers

Launch any browser

public_ip_of_manager (or) workers: 8989

⇒ Start MySQL database with 3 replicas and check all replicas are running.

① Start mysql with 3 replicas in swarm

docker service create --name mydb --replicas 3
-e MYSQL_ROOT_PASSWORD=intellig mysql

② To check on which nodes the 3 replicas of mysql are running

docker service ps mydb

docker service rm webserver mydb — To delete services

↳ services Name (Whatever you create)

Scaling in swarm:

Create tomcat with 3 replicas in docker swarm and then ~~Se~~ Scale it up to 8 replicas. Later Scale it down to 2 replicas.

① Start tomcat with 3 replicas

docker service create --name appserver -P 9090:8080
--replicas 3 tomcat

② Check if 3 replicas are running

docker service ps appserver

③ Increase the count of replicas to 8

docker service scale appserver=8

④ Check if 8 replicas are running

docker service ps appserver

⑤ Decrease the count to 2

docker service scale appserver=2

⑥ Check if only 2 replicas are running

docker service ps appserver

Docker Session (16-03-20)$

Rolling Updates:

Use case: Start redis 3 service with 5 replicas in docker swarm. Perform a rolling update from redis 3 to redis 4. Later perform a rolling roll back from redis 4 to redis 3

① Create redis:3 with 5 replicas in swarm

docker service create --replicas 5 --name mydb redis:3

② To check if redis:3 is running with 5 replicas

docker service ps mydb

③ Perform a rolling update --image redis:4 mydb

docker service update --image redis:4 mydb

④ Check if redis:3 replicas are shutdown and 5 replicas of redis:4 are running

docker service ps mydb

⑤ Roll back from redis:4 to redis: 3

docker service update --rollback mydb

⑥ Check if redis:4 replicas are shutdown and in its place redis:3 is running

docker service ps mydb

Commands:

① To drain worker1 from the swarm

docker node update --availability drain worker1

② To make a drained "worker1" rejoin the swarm

docker node update --availability active worker1

③ To make worker2 leave the swarm

connect to worker2 using gitbash

docker swarm leave

④ To make the manager leave the swarm

Connect to Manager

docker swarm leave --force

⑤ To generate a token id for a machine to join swarm as a ~~manager~~ worker

docker swarm join-token worker

⑥ To generate the token id for a machine to join as Manager

docker swarm join-token manager

⑦ To make "worker1" as a Manager

docker node promote worker1

⑧ To make "worker1" back as worker

docker node demote worker1

Failover Scenarios:

Use case: Start httpd with 3 replicas. Delete one replica running on the Manager. Check if that replica is recreated.

- Drain worker1 from the docker swarm and check if all the replicas running on worker1 have moved to Manager and other worker.
- Make worker1 rejoin swarm
- Connect to worker2 and make it leave the swarm and check if the instance is running on worker2 have migraded to manager and worker1

① Start httpd with 3 replicas

docker service create --name webserver -p 8888:80 --replicas 3 httpd

② Check if replicas are running

docker service ps webserver

③ Delete a replica running on Manager

docker container ls

docker rm -f container_id_from_above_command

④ Check if 3 replicas are still running

docker service ps webserver

⑤ Drain worker1 from the swarm

docker node update --availability drain worker1

⑥ Check if 3 replicas are still running on Manager and worker2

docker service ps webserver

⑦ Make worker1 rejoin swarm

docker node update --availability active worker1

⑧ Make worker2 leave the swarm

Connect to worker2 machine

docker swarm leave

⑨ Check if all 3 replicas are still running on Manager and worker1

docker service ps webserver