**DOCKER COMMANDS**

1. docker run hello-world [ Docker runs processes in isolated containers. A container is a process which runs on a host. The host may be local or remote. When an operator executes docker run, the container process that runs is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host.] (If you run multiple times then multiple containers will be generated) (It will first pull the image from docker registry/ docker hub and then runs the container)
2. docker run --name CONTAINER\_NAME IMAGE\_NAME
3. docker run –i IMAGE\_NAME (interactive mode… runs in the foreground..)
4. docker run –d IMAGE\_NAME (detached mode… runs in the background..)
5. docker images ( list of images)
6. docker ps (list of current running containers)
7. docker ps –a (list of current running + history (exited) containers)
8. docker rmi IMAGE\_NAME (remove particular image)
9. docker rmi $ (docker images) (removing all images)
10. docker rm CONTAINER\_NAME (to remove particular container)
11. docker rm $(docker ps –a -q) (removing all containers)
12. docker stop CONTAINER\_NAME (to stop paticular running container)
13. docker stop $(docker ps –a –q) (to stop all running containers)
14. docker start CONTAINER\_NAME or CONTAINER\_ID (to start particular container)
15. docker pull IMAGE\_NAME:VERSION (pull the docker image from docker hub with version)
16. docker inspect CONTAINER\_NAME or CONTAINER\_ID (Detailed information about the container)
17. docker exec -i -t CONTAINER\_ID bash ( i -- interactive, t -- terminal , bash -- bash terminal) (to enter into the running container) (to exit press ctrl+pq)

docker exec –u root –it CONTAINER\_ID bash (-u root … login in as a root user with all privilages)

1. docker info (information of the docker software)

**PORT FORWARDING**

1. docker run –p host-port : guest-port –d IMAGE\_NAME …( we have to choose host-port on the host-machine which is available and map to the guest-port which will be the application specific. For example: - apache wants 80 port, tomcat wants 8080 port. And browse by using IP-addr : host-port)
2. docker run –P -d IMAGE\_NAME….( -P mean automatically map any port on host to any port on guest)
3. docker run –p 8000-9000:5000 IMAGE\_NAME (range of IP’s on host m/c)
4. docker run –p 127.0.0.1:80:5000 IMAGE\_NAME
5. docker run –p 127.0.0.1::5000 IMAGE\_NAME

**IMAGE CREATION**

Note :-

1. All the docker containers are stored in /var/lib/docker/containers.
2. All the volumes are stored in /var/lib/docker/volumes
3. First create a directory and then create a file called Dockerfile.
4. Edit the Dockerfile like below :-

FROM ubuntu:16.04 (Take image from a build-in image)

MAINTAINER [s@gmail.com](mailto:s@gmail.com) (Maintainers id)

RUN apt-get update

CMD [“echo”,”container-updated”]

Note:-

* RUN executes command(s) in a new layer and creates a new image. E.g., it is often used for installing software packages. We can have multiple run commands. RUN cannot be overriden
* CMD sets default command or parameters, which can be overidden from command line when docker container runs. We can have only one single CMD command, if we have multiple CMD then the last CMD will be executed. CMD will be overridden.
* ENTRYPOINT configures a container that will run as an executable.

1. docker build –t IMAGE\_NAME:VERSION FOLDER\_OF\_DOCKERFILE

ex: docker build –t myimage:1.1 . (Build Image where my Dockerfile is in the current directory, -t is the tag for image for local machine not for the global docker hub)

**PUSHING IMAGES TO PUBLIC REPOSITORY**

1. First create a repositorty on docker hub as gavasbabu
2. Create a Image as : $ docker build -t gavasbabu/myubuntu:1.1 (repo name / image name: version)
3. $ docker login
4. $ docker push gavasbabu/myubuntu:1.1
5. Create a tag and push for the already build image ( like mysql:latest)
   1. docker tag <image id> <account/repo-name> : <version>

ex:- $ docker tag 1jkkjkk678k gavasbabu/mysql : 1.4

* 1. $ docker push gavasbabu/mysql:1.4

**DOCKER VOLUMES**

Note:- all the volumes are stored in /var/lib/docker/volumes/….We can have multiple containers which share volumes.

**CREATING VOLUMES WITH MAPPING TO THE HOST VOLUMES**

* + 1. $ docker volume create <volume-name> (creating a directory <volume-name)

ex:- $ docker volume create hello

$ touch /var/lib/docker/volumes/hello/\_data/a.txt ( placing a simple text file in the folder)

* + 1. $ docker run –d –v hello:/myhello –p 89:80 eboraas/apache (creating a container with myhello volume of guest m/c..mapped to hello volume of host m/c.
    2. $ docker exec –it <container-id> bash (enter into the running container and enter into the bash prompt of it)

$ cat /myhello/a.txt (After enter ensure that a.txt is present in myhello folder)

Note:- when creating volume with mapping to the host directory, make sure to give permissions to that directory..

**CREATING VOLUMES WITHOUT MAPPING**

1. $docker run –d –v /myhello2 –p 87:80 eboraas/apache (creating a container with myhello2 volume of guest m/c..automatically mapped to some volume in the host)
2. $ docker exec –it <container-id> bash (enter into the running container and enter into the bash prompt of it)

$ cd /myhello2 (After entering, ensure that myhello2 folder is present)

# $ docker inspect <container-id> and look at the mounts to see the source (volume on the host m/c and destination (on the guest m/c i.e, /myhello2)

# Note:- If we take image from docker hub.. make sure you read Dockerfile because we cannot create volume to the folder which is already created in the Dockerfile. We can ( or we have to ) create mapping to that folder.

$ docker pull Jenkins

$ docker run –p 8888:8080 –d hello:/var/jenkins\_home jenkins

$ docker inspect <container id> (go to mounts and check there will be only one mount)

But do not......

$docker run –p 9999:8080 –d hello:/var (here jenkins\_home will be empty in the host machine.. as jenkins\_home will be mapped to another volume automatically (as it is written in Dockerfile).

$docker inspect <container id> (go to mounts and check there will be two mounts)

**DOCKERFILE**

1. $ mkdir ubuntu\_image2 ; cd ubuntu\_image2; nano Dockerfile

FROM ubuntu:16.04

MAINTAINER s@gmail.com

RUN apt-get update (executes during image build)

**USER** daemon (sets the username)

WORKDIR <folder name> (present working folder)

ARG <name> (defines a variable , user gives a value at the build time of image)

ARG <name>[=<default value>]

ENV <key> <value>

**ENV** <key>=<value> ( we can also.. docker run --env <key>=<value> )

ADD <source> <destination> (copy from source to destination) (source can be remote url as well local machine)(destination is relative to the WORKDIR)

COPY <source> <destination> (copy from source to destination) (source cannot be remote)

EXPOSE [80,89] (expose the port no.s)

ENTRYPOINT [“echo”,”hi”] (executes compulsorily during container build)

CMD [“hello”] (may not execute when command line arguments are given)

1. $ docker build –t myubuntu3:1.1 .
2. $docker run myubuntu3:1.1

o/p:- hi hello

1. $docker run myubuntu3:1.1 bello

o/p:- hi bello

**LINKING CONTAINERS**

**NOTE:- linking containers using –link option behaves different in default docker network (docker0) and user defined network (my\_network1).**

1. **Default docker network (docker0)**
2. $ docker pull mysql (pulling mysql from docker hub)
3. $ docker run --name some-mysql -e MYSQL\_ROOT\_PASSWORD=password -d mysql:latest
4. $ docker pull wordpress (pulling from docker hub)
5. $ docker run --name some-wordpress --link some-mysql:mysql -p 8888:80 -d wordpress

(--link with the container mysql to wordpress, wordpress needs a database so we are liking wordpress container to mysql container)

Now go to browser http://localhost:8888

1. **User-defined network (my\_network1**)

**1.**

**DOCKER COMPOSE**

1. First install Docker or Docker Engine
2. Next install Docker compose like

# curl -L "https://github.com/docker/compose/releases/download/1.10.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

# **chmod** +**x** /usr/**local**/bin/docker-compose

# docker-compose *--version*

1. Create a directory called composetest and Enter into it. (IMP:- to run compose commands you should always be in the docker compose directory)
2. Create a file docker-compose.yml ( vi docker-compose.yml)

version: '2'

services:

wordpress:

image: wordpress (pull from dockerhub)

ports:

- 8899:80 (port forwarding)

environment:

WORDPRESS\_DB\_PASSWORD: example (environment variables)

mysql:

image: mysql (pull from dockerhub)

environment:

MYSQL\_ROOT\_PASSWORD: example (environment variables)

1. $ docker-compose up –d ( two containers like composetest\_mysql\_1 , composetest\_wordpress\_1 will run separately and will be linked) (if you run multiple times same containers will be run, but will not create another containers like docker run)
2. $ docker-compose ps (list of container running under compose directory)
3. $ docker-compose **run** <service-name> env (to see list of environment variables of service running)

Ex. $ docker-compose run wordpress env

1. $ docker-compose **stop (to stop the containers under compose)**
2. $ docker-compose down *–volumes (to delete containers with volumes)*
3. $ docker-compose --help (for all compose commands help)

**DOCKER NETWORKING**

***DEFAULT BRIDGE NETWORK (SINGLE HOST)***

1. $docker network ls ( list of all networks and its drivers)
2. $ ifconfig (The bridge network represents the docker0 network present in all Docker installation)
3. $docker run --network=<NETWORK> option (to run with our own network)
4. $docker network inspect bridge ( information of bridge network, with list of running containers under it)

Note:- Containers in this default network are able to communicate with each other using IP addresses. Docker does not support automatic service discovery on the default bridge network. If you want to communicate with container names in this default bridge network, you must connect the containers via the legacy docker run --link option.

***USER-DEFINED BRIDGE NETWORK (SINGLE HOST)***

1. $ docker network **create** -–driver bridge network1 (create a user-defined bridge network called as network1)

(OR)

$docker network create network1 (by default it will be bridge type network)

1. $docker network ls (we see our network (network1) in the list)
2. $docker run –-network= <network name> <container name> (running a container under our own user defined bridge network)
3. $docker network inspect <network name> (we can see the container running in our defined network)

Note:- The containers you launch into this network must reside on the same Docker host. Each container in the network can immediately communicate with other containers in the network. Though, the network itself isolates the containers from external networks.

Within a user-defined bridge network, linking is not supported. You can expose and publish container ports on containers in this network. This is useful if you want to make a portion of the bridge network available to an outside network.

A bridge network is useful in cases where you want to run a relatively small network on a single host. You can, however, create significantly larger networks by creating an overlay network.

When you create a network, Engine creates a non-overlapping subnetwork for the network by default. You can override this default and specify a subnetwork directly using the --subnet option. On a bridge network you can only specify a single subnet. An overlay network supports multiple subnets.

**Note** : It is highly recommended to use the --subnet option while creating a network. If the --subnet is not specified, the docker daemon automatically chooses and assigns a subnet for the network and it could overlap with another subnet in your infrastructure that is not managed by docker. Such overlaps can cause connectivity issues or failures when containers are connected to that network.

You can connect a container to a network even if the container is not running. However, docker network inspect only displays information on running containers.

.

**CONNECTING BRIDGE NETWORKS (SINGLE HOST)**

1. $docker run –d container1 (container1 running on default bridge network (docker0))
2. $docker run –d container2 (container2 running on default bridge network (docker0))
3. $docker network connect my\_network1 container2 (container2 is also running on the user-defined network (my\_network1). So, it has to IP address)
4. $docker run - -network = my\_network1 container2 (container3 is running on userdefined network called my\_network1)

Note :- container1 -> container2 ( container1 can connect to container2)

container3 -> container2 (container3 can connect to container3)

But, container1 cannot connect to container3

**OVERLAY NETWORK WITH EXTERNAL KEY-VALUE STORE**