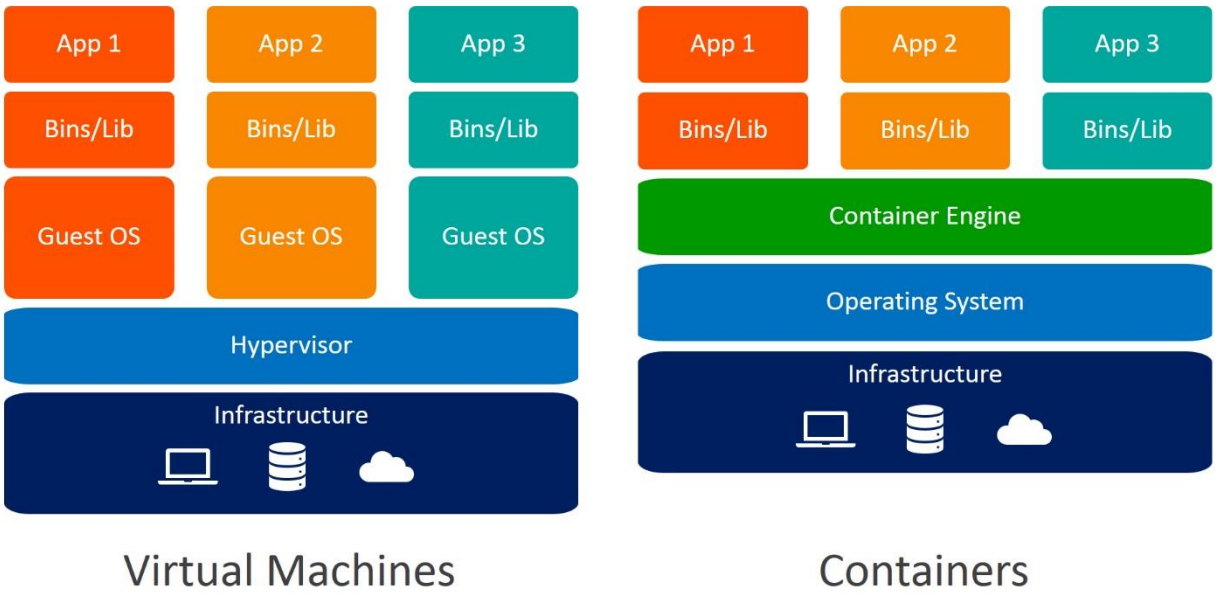
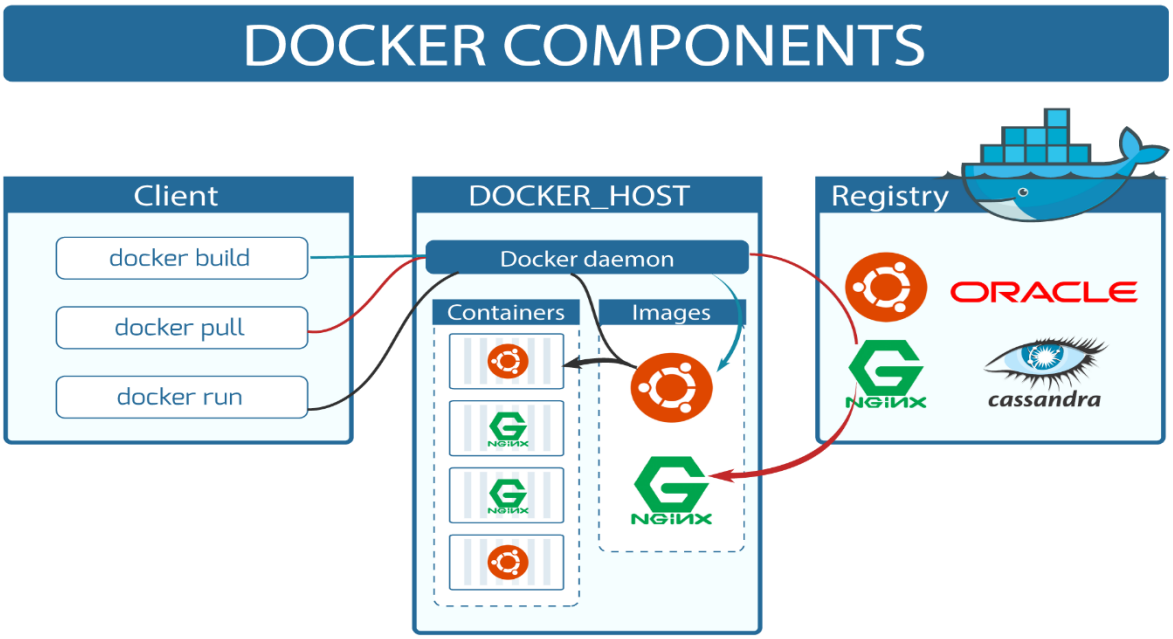
Docker

Docker is a set of coupled software-as-a-service and platform-as-a-service products that use operating-system-level virtualization to develop and deliver software in packages called containers. The software that hosts the containers is called Docker Engine. It was first started in 2013 and is developed by Docker, Inc.



Docker Architecture:



Steps to Install docker on RedHat 7.5:

STEP 1 - Connect to Linux system

STEP 2 - Installing DOCKER using below command:

sudo yum -y update

sudo yum install -y docker

If the above step(sudo yum install -y docker) doesn't work, then try below

```
sudo yum install yum-utils

sudo yum-config-manager --enable rhui-REGION-rhel-server-extras

sudo yum install docker -y
```

STEP 3: checking docker version

docker --version

STEP 4 – Starting DOCKER

sudo service docker start

sudo usermod -a -G docker "user"

Checking docker information: docker info

Running first container:

docker run hello-world : to run hello-world image

docker images : to get list of images present locally

docker ps : to get list of running containers

docker ps -a . : to get list of all(running and stopped) containers

STEP 5 - Stop DOCKER

sudo service docker stop

STEP 6 – Uninstalling DOCKER

sudo yum remove docker

Docker commands:

docker --version: To see the docker version

In community edition we have 2 channels: 1) Stable and 2) Edge release

docker-machine --version: To check docker machine version.

docker-compose --version: To check docker compose version.

docker info: It gives info about docker like, how many containers are running, stopped, paused, how many images and other info.

docker image ls: It gives the info and status of images.

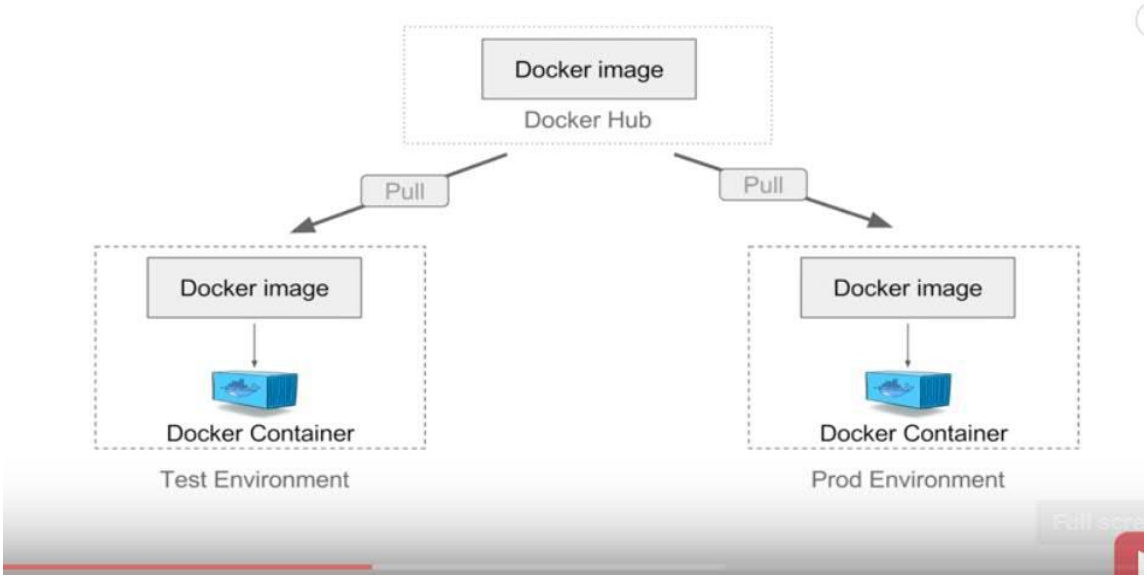
docker run hello-world: To create container by running the image. Here, hello-world is the image name.

First it will check for the image in docker local repo. If not found, then it searches in docker hub repo.

In docker hub repo, we have private and public repos for images. Docker first searches for images in public repo, then tries to search in private repo, If not logged-in to private repo and the image is not found in public repo, then it will throw error saying to login, as it tries to find the image from private access.

You can get the error if you are not using the correct image name (ex: **hellow-world**).

You can use this command (**docker run hellow-word**) to check whether docker is installed correctly or not.



docker pull imageName: It will get the latest image from docker hub repo to your local system. Ex: docker pull hello-world

Docker will pull the images layer by layer.

Difference between docker run and docker pull:

docker pull will load the latest image from docker hub to local docker repo where as

docker run will run the image and creates container, if the image presents in local repo. If the image is not present in local, then it pulls from docker hub and then runs the image and creates container.

docker pull ubuntu:18.04: Here ubuntu is the image name and 18.04 is the tag. So docker will pull the ubuntu image with 18.04 tag.

In docker hub, repository name is nothing but image name.

Each image contains image id. Image name and image id are unique.

docker ps: To list the running containers.

docker ps -a: To list all containers (running and stopped)

docker container ls -a

Using filters while finding the containers or images:

docker ps -a --filter "name=jenkins": Will display all the containers which have jenkins name

docker ps -a --filter 'exited=0': Will list all the containers which has existed state as 0

docker ps --filter status=running: Will list all the running containers

If you pass **status=paused**, then lists all paused containers.

docker pause ContainerName/ID: To pause the container. Paused container will not work until you unpause it.

For example, if you pause any ubuntu container, then you cannot type any commands in that ubuntu container in interactive mode. But you can stop or remove container from another terminal.

docker unpause ContainerName/ID: To unpause the container.

docker top ContainerName/ID: Will list the top processes of the given container.

docker status ContainerName/ID: Will list the status of that container, like memory, CPU usage, etc.

docker attach ContainerName/ID: To get into the container. Ex: to get into ubuntu from docker.

exit: To comeback from container interactive mode to docker.

docker kill ContainerName/ID: To kill any running container.

docker history ImageName/ID: To see the history of image.

docker logs containerId/Name: Will show what are all the commands you ran for the given container id.

This command you can use for auditing the container to check what you have done with that container earlier.

docker images: Gives the list of docker images and its info.

docker images --help: Will list all the options available for docker images command.

docker image ls: Gives the long list of docker images and its info.

docker rmi imageName OR **docker rmi imageId** :This will remove the image from local.

To delete any image, first we have to delete its associated container. But if we want to delete forcibly, then use below command

docker rmi -f imageName/imageId

If we delete image forcibly, then docker will delete the image, but the container remains same and becomes useless.

So, it is always better to remove container and then its image.

Dangling images: Images which are not associated or used by at least 1 running container. Dangling images are layers that have no relationship to any tagged images. They no longer serve a purpose and consume disk space.

docker images -f "dangling=false": Will get all the images which are associated or used by at least 1 running container. Means non-dangling images.

docker images -f "dangling=true": Will get all the images which are not associated or not used by at least 1 running container. Means dangling images.

docker images -f "dangling=false" -q: will get all the images ids for the images which are associated or used by at least 1 running container.

docker images -q OR docker images --quite: will give all the list of docker image ids. Here q means quiet.

docker rmi -f $(docker images -q): To delete all the images at one shot.

docker rm $(docker ps -a -q) --force: This will remove all the containers (running and stopped) forcibly.

docker rm $(docker ps -a -q): This will remove all stopped containers. Any running containers will not be deleted.

docker rm $(docker ps -q) --force: This will remove all running containers. Any stopped containers will not be deleted

docker ps: Will list the running containers.

docker ps -a OR docker ps --all: To see all the containers even if they are running or stopped.

docker stop containerId/Name: To stop the container

docker start containerId/Name: To start the container.

docker restart containerId/Name: To restart the container.

docker rm containerId/Name: To delete the container id from local docker repo.

We cannot delete the running container, so we must delete forcibly with -f OR stop the container and then delete it.

docker run imageName/ID: To run the image and create a container.

docker run --name customCantainerName imageName: To give the custom name to the container while running:

ex: docker run --name mycustomcontainer hello-world

Each image will create a container. Creates multiple containers when we run the same command multiple times.

What is the difference between **docker image ls** and **docker images**?

What is the different between **docker ps -a** and **docker info**?

docker image rm imageId OR **docker rmi imageId** : Both command are same, removes the docker image. To delete forcibly use -f.

docker search imageName: To search the image availability and location.

Also says it is OFFICIAL image or Not.

docker search imageName | head -10: To search the image availability and lists the top 10 images.

Ex: **docker search jenkins | head -10**

docker search --filter=stars=500 jenkins: To search for image based on some condition(filter). Here it will search for Jenkins image which has 500 stars.

docker search --no-trunc jenkins: Will give the image details with full description. --no-trunc means to not to truncate the full details.

docker pull imageName:version: To pull the image with specific version.

ex: docker pull centos:7. It will download docker image for centos7 version.

If :7 is not given, then it will download the latest version by default.

docker run -it centos:7 /bin/bash: This will create the container and gets in to that container.

-it means interactive mode, centos is the image name and :7 is the tag name, so it will download centos 7 version. If :7 is not given, then it will try to pull and run for latest version.

If you use -it(interactive mode), then it will open centos in interactive mode. So, you can use and run centos commands.

i stands for interactive.

t stands for terminal.

d stands for detached or disconnect. (runs in background)

-it : Will run in interactive mode and enters in to that container.

-itd : Will run in interactive mode but doesn't enters into that container.

-d : Will run the image in background, so no logs will print on console.

exec command: To run any Linux command on running container.

Ex: docker exec containerId/Name ls

docker exec containerId/Name hostname

docker exec containerId/Name touch /opt/testfile.txt

If you run the image in detached mode and after that you want to see any details of any container, then use below **exec** command:

docker exec cotaninerName cat path.

For example, after creating jenkins container in detached mode and then you want to see the initial secret password, then use this command

Ex: docker exec containerName cat pathOfTheSecretPasswordFile

Ex: **docker run --name MyUbuntu -it ubuntu bash**: Running the container with custom name. Then getting in to interactive mode with -it (which gets into the container after creation). bash indicate the shell to run the container.

docker inspect imageName or id: it will give all the details of the image like, id, tags of the image and host name, domain name, author, docker version, layers and etc.

docker inspect containerName or id:

docker network create firstnetwork: creating network with the name firstnetwork.

docker network ls: Will show the type of network you are connected.

docker network inspect bridge: Will give full details about the particular network (ex: bridge network).

docker network connect: will connect container to the network

docker network connect --help: Will list all possible options for the connect command.

docker network rm networkName: to remove specified network.

docker network prune: will remove all unused networks, which are not associated to at least 1 container.

Creating Custom image using Dockefile:

FROM tomcat:8.0-jre8

MAINTAINER Neelakanta neelakanta@in.ibm.com

ADD http://ftp-chi.osuosl.org/pub/jenkins/war/2.143/jenkins.war /usr/local/tomcat/webapps/jenkins.war

EXPOSE 8080

CMD ["catalina.sh", "run"]

Steps and Example to create a docker image and push it to Docker Hub:

Note: Make sure you have registered and have a valid Docker Hub account before pushing.

Step 1) Write Dockerfile

FROM tomcat:8.0-jre8

MAINTAINER Neelakanta neelakanta@in.ibm.com

ADD http://ftp-chi.osuosl.org/pub/jenkins/war/2.143/jenkins.war /usr/local/tomcat/webapps/jenkins.war

EXPOSE 8080

CMD ["catalina.sh","run"]

Step 2) Build docker image using Dockerfile (docker build)

Ex: docker build -t salesbeat:1.0 .

Step 3) Tag image (docker tag)

Ex: docker tag salesbeat:1.0 dsgindia/salesbeat:1.0

Step 4) Login to docker hub (docker login)

Ex: docker login

Step 5) Push Docker image to Docker Hub (docker push)

Ex: docker push dsgindia/salesbeat:1.0

It will automatically take and pushes the tagged images

docker attach containerId: To get into the container.

One more example of Dockerfile:

```
FROM ubuntu:xenial
MAINTAINER Neelakanta neelakanta@gmail.com
RUN apt-get update -y && apt-get install apache2 -y
EXPOSE 80
CMD ["echo", "override this option using /bin/bash"]
```

Then build an image with this docker file and

Then execute this command to run the docker image**: docker run -d -it -p 80:80 myapache:1.0**

After this you have to start apache inside ubuntu.

So get in to ubuntu interactive mode using **docker exec -it containerid /bin/bash**

check apach2 status: **service apache2 status**.

If it is in stop state, then start it: **sebrvice apache2 start**

Then access it from browser: **http://ipaddress**

Ex: **docker exec -it containerid ls /var/jenkins_home**

Ex: **docker attach containerId.**

While using attach, **CMD** option can be overridden with the option we pass while starting the container.

ex: **docker run -d -it imagename:tag /bin/bash**.

Here bin/bash will override CMD option in Dockefile, while starting the container.

Transferring files from host OS to container OS:

Connect to your host OS: and run below command.

docker run -itd -v /opt/myfiles:/opt/myfiles ubuntu

This command will create and runs a new ubuntu container.

here **-v** indicated volumes.

/opt/myfiles:/opt/myfiles --> First path is the path in local host OS. Second path is the path in new container.

This command will create a directory **/opt/myfiles** in local. Also creates a **/opt/myfiles** path in container.

So, these two directories from host OS and Container OS(ubuntu) are in sync. If you create a directory or file in one OS, then it will automatically reflect on another OS.

If you want to see this, then connect to the newly create ubuntu container using below command.

docker attach newcontainerid

Then go to the path: **/opt/myfiles** and create a file.
Then type Ctrl+P+Q. So, you will come back of connected container.

Then go to **/opt/myfiles** in local OS. Then you can see the file what you created in container(ubuntu)

Running Jenkins on docker container with Bind Mount and Volume:

First go to docker hub. Search for jenkins. There you can see jenkins image, Select it. There you see **docker pull jenkins**. Use that command in docker terminal to fetch the image.

docker pull jenkins: pulling latest jenkins image from Docker Hub

docker run -p 8080:8080 -p 50000:50000 jenkins:alpine: Starting docker on the port number 8080. here -p indicates assigning host OS portnumber to docker container(jenkins)

Here left side 8080 is host OS port number, right side 8080 is container port number.

Here we are assigning 8080 port number of Host OS to 8080 of container port number.

Same for 50000 as well, which is for jenkins API.

When you click Enter, it will start jenkins with the port number 8080.

However, with this command, the data or the jenkins directory, jenkins jobs and all the information of jenkins stored will be deleted when you delete the container.

Bind Mount:

docker run --name MyJenkins -p 8080:8080 -p 50000:50000 -v /Users/Mylocation/Desktop/Jenkins_Home:/var/jenkins_home jenkins:alpine

--name MyJenkins --> Indicates custom name for the jenkins container.

If we want to persist the data, then we can use **-v** option, so that the data will be stored on HOST OS.

This data will not be removed, even if you remove the container.

After **-v** the path we have on left side of **:** indicates the local path of user HOST OS.

Right side path after **:** indicates the Jenkins path on docker.

If you run the above command, then it will create a **Jenkins_Home** directory on your host OS.

It will write down all the Jenkins related data to **Jenkins_Home** directory

Now Jenkins will be up and running. From browser you can access Jenkins with http://localhost:8080

docker ps: shows running containers.

Then try,

docker stop MyJenkins: This will stop MyJenkins container.

Then remove MyJenkins container: **docker rm MyJenkins**

Then re-create the container again with different name (with same or different port)

docker run --name MyJenkins2 -p 9090:8080 -p 50000:50000 -v /Users/Mylocation/Desktop/Jenkins_Home:/var/jenkins_home Jenkins:alpine

If you run this command, still jenkins uses the physical location on jenkins from HOST OS. Whatever the jobs you have created with MyJenkins will still exists.

So, if we use **-v**, then, even if we remove the container, still its data will be persisted on HOST OS.

Volume:

docker volume create myjenkins: This creates a volume with the name myjenkins.

docker volume ls: Will list the created volumes

docker volume inspect myjenkins: This will give the details of the volume and the mount point where the volume is available.

docker run --name MyJenkins3 -p 9090:8080 -p 50000:50000 -v myjenkins:/var/jenkins_home Jenkins:alpine

Here we are giving the volume name, instead of giving the full path(mount) of HOST OS.

Advantage here is:

Although the volume is present in HOST OS, you will not be able to see it, or the functions will not be able to touch or change this volume.

It will be easy when we want to share values between the containers and also want to persist our data.

The above command will run the jenkins as usual. Here, it will use docker volume instead of using physical location.

docker inspect MyJenkins3: This will give the full details of the container, including volumes, mounts ..etc... in a json file format.

Run docker in interactive mode. Then type ls. you will be able to see the underlying structure of your jenkins container application.

For example, if we run docker run ubuntu. Then it will download and run ubuntu. If you want work on that ubuntu machine (container), then use **docket attach ubuntucontainerid**, so you will connect to ubuntu and we can work on that.

You can check whether you are connected to ubuntu by using ubuntu command. Ex: **hostname**

Ctrl+p+q : To get back from our connected container(ubuntu) to docker.

docker image will have read only permissions. But docker container will have read and write permissions.

We can create any no.of containers for same image.

docker login: command to login to dockerhub.

username and emailid are different in docker.

After you login with correct credentials, then your credentials will be saved under: **/root/.docker/config.json** file

logout: is the command to logout from logged in docker hub

Docker volumes:

1. What are Volumes
2. How to create / list / delete volumes
3. How to attach volume to a container
4. How to share volume among containers
5. What are bind mounts

Volumes definition: Volumes are the preferred mechanism for persisting data generated by and used by Docker containers

1) Volumes are used to persist docker container data on Host OS, so we can have the data even if the container is deleted.

If we don't provide the physical location for persisting the data, then data will be stored inside the container and data will be deleted when we delete the container.

Volumes can be created in 2 ways:

1)**Data volume:** Here we persist the data inside the container. But data will not be deleted when we delete the container.

ex: **$ docker run -it -v /data --name container1 busybox**

2)**Data volume containers**: Here we persist the data in a physical location of Host OS. So, data will not be deleted when we delete the container.

Here we are going to discuss only about Data volume containers:

docker volume //get information

docker volume --help: Will give all the possible options for docker volume and their brief description.

docker volume create: to create volume

docker volume ls: To list the created volumes

docker volume inspect volumeName: To display details information for one or more volumes.

docker volume rm volumeName: To remove one or more volume

docker volume prune: To remove all local unused volumes

Use of Volumes

Decoupling container from storage.

Share volume (storage/data) among different containers.

Attach volume to container.

On deleting container volume does not delete.

Example commands:

1)**docker volume create myvol1**

Here we are creating a volume with the name myvol1.

2)**docker volume ls**: Will list the created volumes.

Here it shows myvol1, as we created only 1 volume (myvol1).

3)**docker volume inspect myvol1**: Will list the details of the volume myvol1.

Mountpoint in the details indicates the location where the volume got created locally and functions cannot edit it.

4)**docker volume rm myvol1**: This will remove the volume myvol1.

5)**docker volume prune**: This will remove the unused volumes which are not used by at least 1 container.

Let's see how to use our volume myvol1:

Let's try to pull a jenkins image : **docker pull jenkins**

Then start the container: **docker run --name MyJenkins1 -v myvol1:/var/jenkins_home -p 8080:8080 -p 50000:50000 jenkins**

So, the Jenkins container will be created with 8080 and 50000 port numbers, here we are giving the container name as MyJenkins1

-v is to attach a volume.

-**v myvol1:/var/jenkins_home** : here we are saying Jenkins home on docker should correspond to volume myvol1.

So, Jenkins will be started and available with the port number 8080.

Then you can start one more container with different container name and port numbers using below command:

docker run --name MyJenkins2 -v myvol1:/var/jenkins_home -p 9090:8080 -p 60000:50000 jenkins

If you try to access jenkins with loclahost:9090, it will try to refer to the MyJenkins1 jenkins as the volume (myvo1) is shared by MyJenkins1 and MyJenkins2

Bind Mount: A file or directory on the host machine is mounted into a container.

It means, instead of using volumes we use physical location to persist container data.

Use below command to use Bind Mount:

docker run --name MyJenkins3 -v /Users/MyLocation /Desktop/Jenkins_Home:/var/jenkins_home -p 9191:8080 -p 40000:50000 jenkins

Try to access jenkisn with 9191 port number. You will observe that jenkins files are created under /Users/MyLocation /Desktop/Jenkins_Home directory.

Try **docker ps:** will show 3 docker containers running.

We can try to stop and remove the containers. But still the containers data exists in Mount and volumes.

Important points to remember about Docker volumes:

By default, all files created inside a container are stored on a writable container layer.

The data doesn't persist when that container is no longer running

A container's writable layer is tightly coupled to the host machine where the container is running. You can't easily move the data somewhere else.

Docker has two options for containers to store files in the host machine

So that the files are persisted even after the container stops.

Creating link between containers:

Will try to create wordpress and mysql containers.

Wordpress needs mysql to work. So, both the containers need to be communicated.

So, to create communication between these 2 containers we need to link both the containers.

To containers use Protocols like TCP/HTTP etc to communicate with each other.

First pull and run mysql container:

docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql:tag

Remove tag if you don't want any tag.

Then pull and run wordpress image:

docker run --name some-wordpress --link some-mysql:mysql -p 8080:80 -d wordpress

To find these commands, you can search in docker hub

With **--link** we are creating link between wordpress and mysql container

Verify both the containers are running or not, then

Now try to access wordpress with 8080 port number.

Here we created dependency/communication/link between 2 containers.

Docker Keywords or directives:

==========

FROM --> To specify the base image. If there is not base image, then give scratch as the base image. SARATCH is also an image which has nothing.

MAINTAINER --> maintainer of the Dockerfile.

COPY --> To copy files from source container to destination container path.

ADD -> To download files from remote location and copy to the container path.

RUN -> To run Linux commands on the base image.

CMD -> To run a command on the container, at the starting time of the container.

EXPOSE -> To expose the container accessible port number.

ENV -> To set environmental variables. If you set environmental variable in one file. Then you can reuse it from any other file.

ARG ->It is an argument, also called as variable. If we write any ARG, then we can use it anywhere in the Dockerfile by using $.

WORKDIR -> To define from which location the command given in CMD has to run.

VOLUME -> To specify on which volume the container has to run.

USER -> If we give user here, then that userid becomes as the default userid, and the owner of all the files in that application will be assigned with this user id.

If we use USER directive in the Dockerfile, then it's our responsible to create that user, that can be by using RUN command, like RUN **adduser** admin, or use the existing user.

If not given any user, then root user becomes the default user.

ENTRTPOINT ->Entry point will contains only an executable, not command. ENTRYPOINT has 2 forms, Shell form and exec form. Whatever the params you write in CMD, that will become as argument to ENTRTPOINT

exec form example**: CMD ["executable", "param1", "param2"]**

Shell form example**: CMD command param1 param1**

Always exec form is recommended to use.

Dockefile example with ENTRYPOINT:

```
FROM ubuntu:xenial

RUN apt-get update -y

ENTRYPOINT ["echo"]

CMD ["hello"]
```

build and run the image, It will print **hello** in the console.

/var/lib/docker/containers ---> This is the place where all the containers data is stored.

Take any container id and go to this path run cd containerId, then ls, there you can see all the container related files and data.

Docker Networking:

docker network ls: Will list all the used and supported networks in Docker.

Basically, it has bridge, host and none docker adopters.

None indicates no network. You use it when you don't want a container to run on any network.

docker inspect bridge: Will give the complete info of bridge adopter.

Under this you can see a **Gateway**, which is used to speak to the other networks.

Subnet indicates the range, i.e 2 power 16- 2.

Subnet indicates the communication ipAddress range.

If you get into docker ubuntu container and check for the ipconfig, there you can see **Loopback** and **eth0** adopters.

eth0 will be the default bridge network for any container.

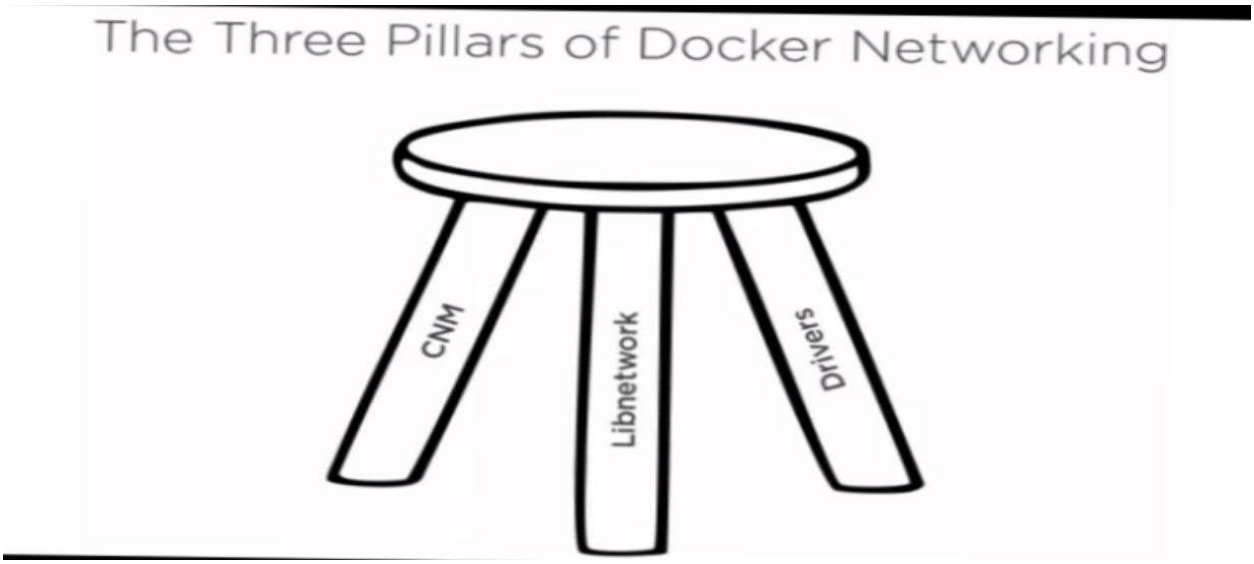
Whenever you install Docker, then an adopter called "**bridge**" will be installed and that will be given to all the containers.

Bridge adopter will create a connection between docker and HOST.

Loopback adopter is to create connections within the Docker.

Containers can communicate to other container using ipAddress and this happens using bridge network.

3 pillars of Docker Network:



CNM : It is not a software. It is requirements documents. It says what has to be done, it was proposed by Docker Inc corporation to maintain thin layers for docker.

Libnetwork : It is the implementation of CNM

Driver: It is specific to networking details, which says what is the kind of network we have to connect to, it is between Host and container, container to container, container to container which presents in different networks.

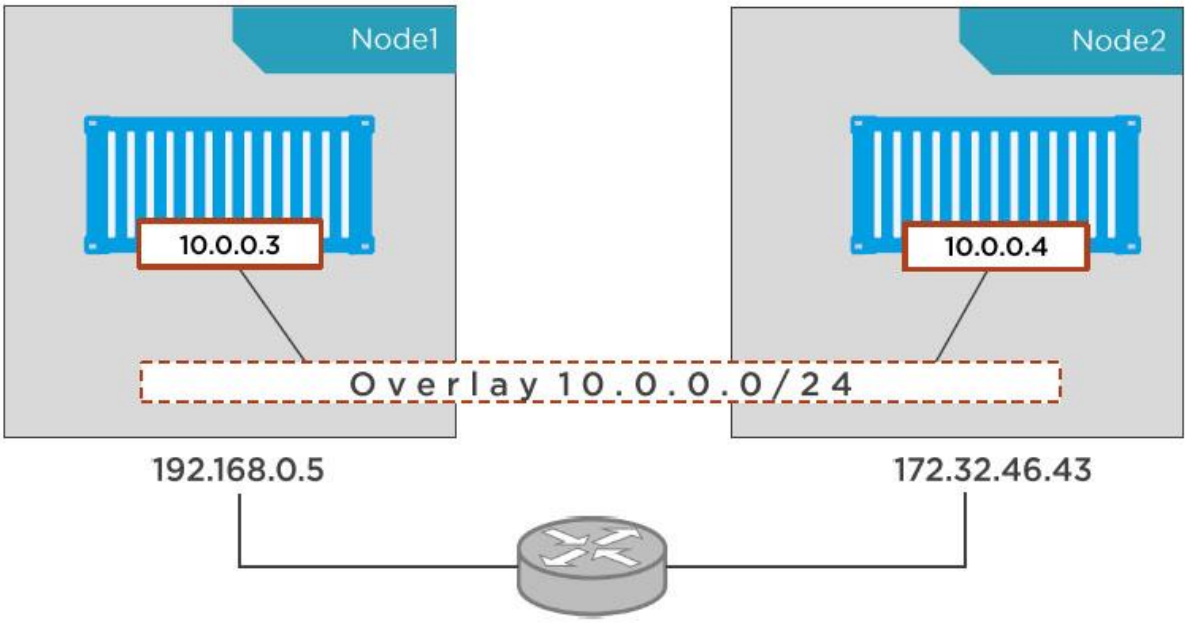
Drivers are developed in language called GO.

Networks are of 2 types: **Single Host and multi Host.**

default bridge in Linux is **bridge** and in Windows it is Nat.

Docker relies on CNM (Common Network Module) and Kubernetes relies on CNI (Common Network Interface).

Both are meant for enabling Multi Host networks.



CNM contains:

Sandbox: Is nothing but a NetworkNamespace, which contains the full information of Network.

Endpoint: In a Network interface like eth0. It is an adopter like LAN.

Network: This is a connected end points which we want to create.

Network commands:

Options for the command **docker network**:

connect: connect a container to a network

create: create a network

disconnect: disconnect a container from a network

inspect: display detailed information on one or more networks

ls: lists networks

prune: Remove all unused networks

rm: Remove one or more networks

docker network ls: Will list the default networks available. (bridge, host, null). bridge is the default driver.

Under this command, NetworkId is network id, NAME is the network name, DRIVER is the name, SCOPE says the scope of Network.

By default, Network Name and driver name are same, but those can be different.

docker network inspect bridge: Will give the full information of bridge driver.

Run this command and see the gateways. It has computer power to connect 2 power 16-2 machines on your network.

If the SCOPE is local, that means it is Single host network.

If the SCOPE is swarm, then it means it is a Multi-host network, which means to connect between to docker engines running on different machines.

docker network create -d bridge --subnet 10.1.0.0/24 mybridge

Here, **docker network create** is the command to create a network. **mybridge** is the network name.

-d is to specify the type of driver we want to use for this network. bridge is the type of driver.

--subnet 10.1.0.0/24 is the CIDR range. So 2 power 32-24 = 254 containers can be connected to this network.

docker info: Will give the full info of docker.

Then you can see plugin Network: **bridge**, **host**, **macvlan**, **null**, **overlay**

These are all the drivers that are already installed on your machine.

You can also get 3rd party drivers, you can download and use those.

Then type **docker network ls**: This will list the newly created network

While creating a container, if you don't specify any network, then by default docker creates the container in bridge network.

docker network inspect mybridge/NetworkId: This will give full details of the network like Gateway and the containers under this network.

Now create 2 containers under our newly created network:

docker run -dt --name ContainerName1 --network mybridge ubuntu sleep 1d

docker run -dt --name ContainerName2 --network mybridge ubuntu sleep 1d

Now try, **docker network inspect mybridge**: This will show the ContianerName1 and ContianerName2 are associated to the mybridge network.

Now try: **docker attach ContianerName1**

Then install ping in ContianerName1, if it doesnt exists: **apt-get install iputils-ping**

Try**:** **ping 10.10.0.3** (ContianerName2 IPaddress)

Try: **ping ContianerName2**

In default, 2 containers cannot communicate each other with Name, but works with ipaddress.

But, with custom networks we create, they can communicate. This be because of DNS service discovery.

To create multi Host networks, we use overlay bridge:

For this, create 2 ubuntu instances on AWS EC2 instances.

connect as root user using **sudo -i**

Install docker on both instances.

Try to ping from one to other and check they are reachable to connect or not. They will be able to connect.

Then enable port numbers as given in the screen shot. If we use vagrant, then we no need to enable port numbers as all port numbers are opened by default.

Then go to docker machine, Type **docker swarm init**. Then it will generate a command with secret key to enter in other machines.

The machine where you use docker swarm init is called Manager. Manager assigns the work. We can have more than 1 Manager.

Where ever the secret command you execute is called Worker. Worker executes the work.

Copy the command generated by Master and execute that on the Other Host (Worker).

So that Worker node will be joined as a swarm as a worker.

Then try, **docker network ls**: There you can see bridge and overlay networks created. overlay network will have **SCOPE** as swarm(multi-host).

Then go to the Master machine and try, **docker network ls**. There also you can see 2 networks created (**bridge and overlay**), so that 2 machines can communicate through docker networks(overlay).

When you use SWARM, you cannot run container with run command any more. Container in SWARM are called "service"

CREATION OF OVERLAY NETWORK

"docker network create -d overlay **mymultihost**" --> To create the network

Try executing "docker network ls" on other containers. You will not see the **mymultihost** as docker follows lazy approach.

To see **mymultihost** start a container.

"docker service create --name my-svc --network **mymultihost** --replicas 3 hello-world sleep 1d"

To see the status, execute on host1 "docker service ps my-svc".

You should run all the container(service) creation commands on master and not on Worker. It will not work on Worker.

--replicas means, no.of container instances to be created.

Try to create a service by giving the overlay network created and then try this command: **docker service ls**: This will list the created services, and no.of replicas has been created.

To inspect: docker service inspect serviceName/Id

To see the status, execute on Host1: **docker service ps serviceName/Id**

Docker Compose:

Here is an example using docker-compose.yml:

```
version: "3.3"

services:
  wordpress:
    image: wordpress
    ports:
      - "8080:80"
    networks:
      - overlay
    deploy:
      mode: replicated
      replicas: 2
      endpoint_mode: vip

  mysql:
    image: mysql
    volumes:
       - db-data:/var/lib/mysql/data
    networks:
       - overlay
    deploy:
      mode: replicated
      replicas: 2
```

```
    endpoint_mode: dnsrr

volumes:
  db-data:

networks:
  overlay:
```

If you run the command: **docker-compose up -d**

Then nginx will be created and will be available to access from browser with the port number 9090.

You can try using: **http://localhost:9090**

docker-compose up -d --scale wordpress=4

You can verify the 4 instances by running command: **docker ps**

So you can see 1 ingnix and 4 redis containers.

When you use **docker-compose down** command, then all containers will be down including 4 instances.

References: https://rominirani.com/docker-tutorial-series-part-7-data-volumes-93073a1b5b72