**Docker**

Docker images are not like pre-compiled commands.

Like base image for Virtualization(windows), docker also needs a base image (ex: Linux).

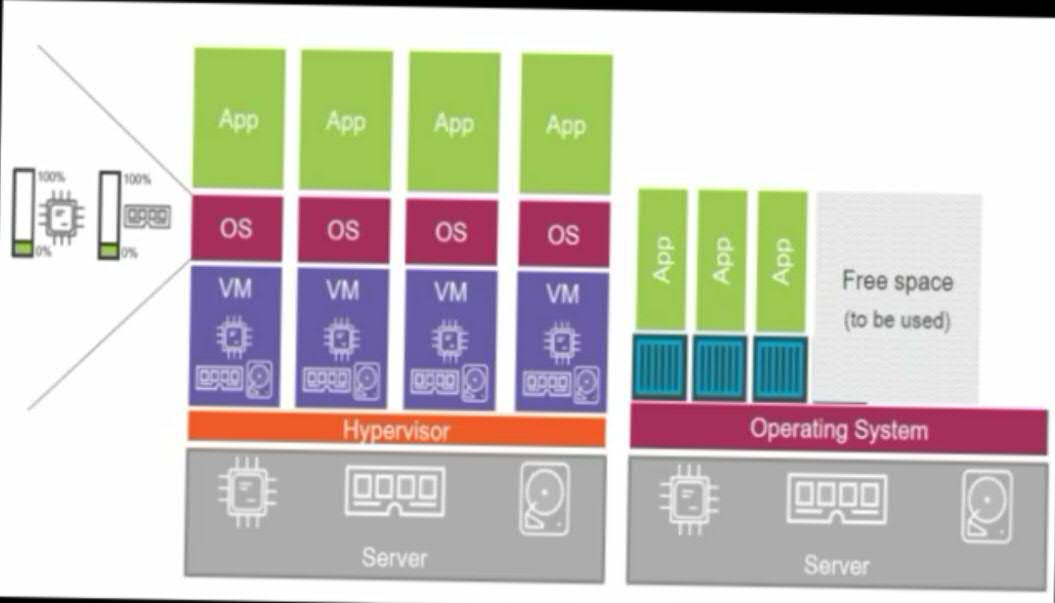
**Few benefits of Docker:**

* Speed
* Memory saving
* Lite weight
* Time saving
* Best resources utilization
* Agile
* Installation free
* Easy to create, start, stop and removing containers.
* Easy to pull, build and remove images
* No need to install OS, container acts as an OS (which is not actually an OS)

<https://www.tecmint.com/install-docker-and-learn-containers-in-centos-rhel-7-6/> 🡪Very good site to learn docker.

DTR-->Docker Trusted Registry, also called **DockerHub**

Docker runs as a Daemon, whereas VM is a running machine.



**What is docker image:**

Docker images are templates used to create containers.

OR

A Docker image is a file, comprised of multiple layers, used to execute code in a Docker container. ... When the Docker user runs an image, it becomes one or multiple instances of that container. Docker is an open source OS-level virtualization software platform primarily designed for Linux and Windows

A single image can be used to create multiple containers.

**What is docker container:**

Container is a running instance of image.

OR

A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Docker containers runs in an isolated state, so will not have any relation with each other.



docker image will have read only permission, but docker container will have read and write permissions.

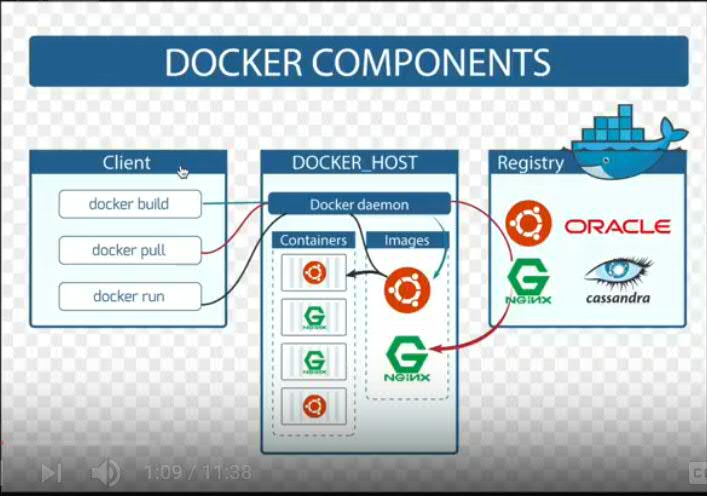
**Container life cycle:**

Container will have 2 states: 1) Running, 2) Stopped or Exited states 3)Paused state

**Where are images stored?**

Registries (e.g Docker Hub or Nexus or AWS ECR)

**Docker Components:**



Docker engine or host

Dockerfile

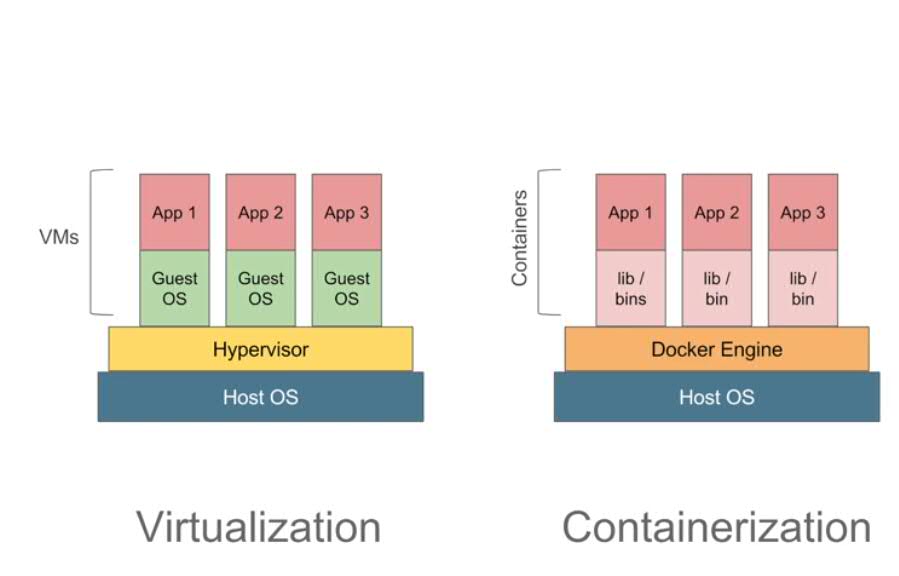
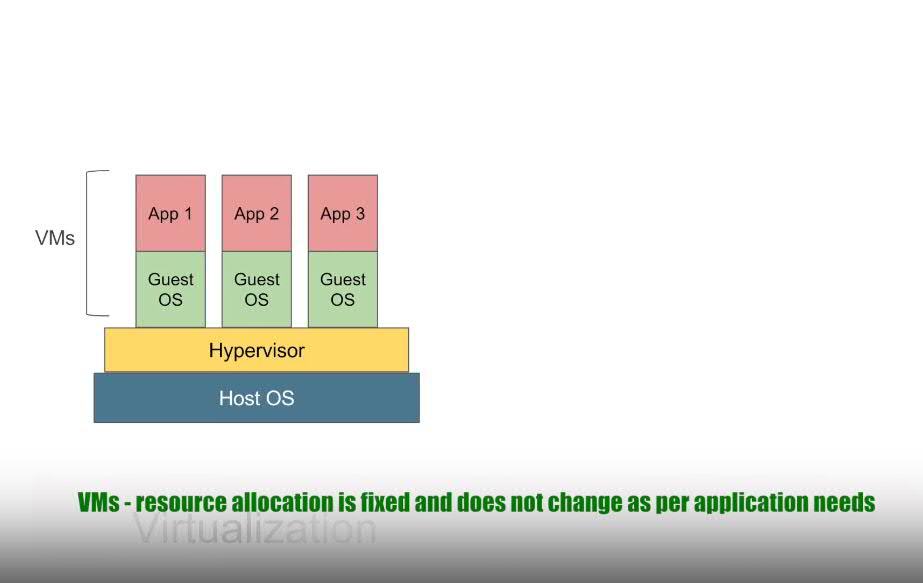
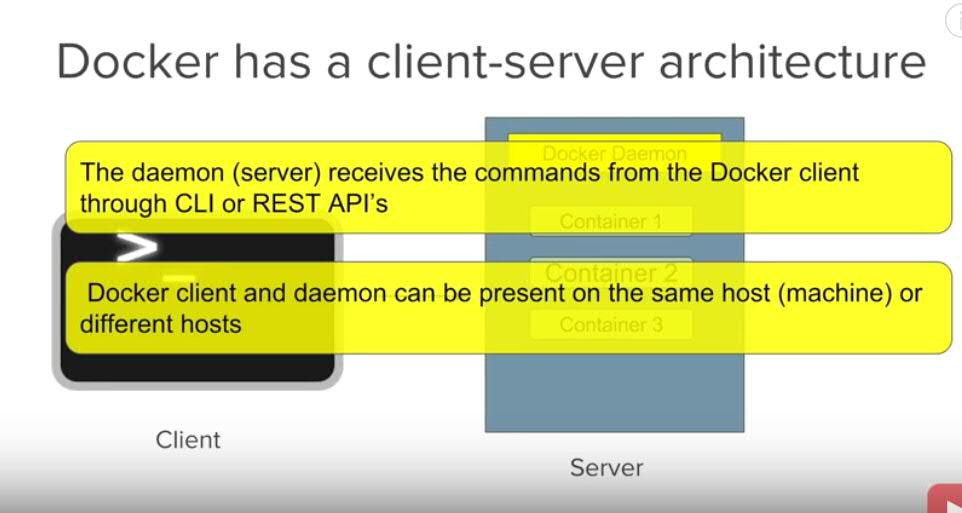
Docker Image --> Docker image is stored in DTR. Docker image is nothing but set of instructions.

Docker container

Docker hub

Docker daemon

Docker client



**Docker life cycle and Architecture:**

**What is docker daemon and client.**

Once you install Docker, then you will get docker daemon and client by default.

Docker client is the terminal where you run docker commands. It understands the docker API. To run docker command user interacts with client.

Docker daemon will communicate between docker client and other components like docker hub and containers.

When you pull an image, then client informs daemon about the image.

Then daemon will search for the image in local repo, if doesn’t exists, then searches in Docker Hub public repo. If does exists, then searches in private repo.

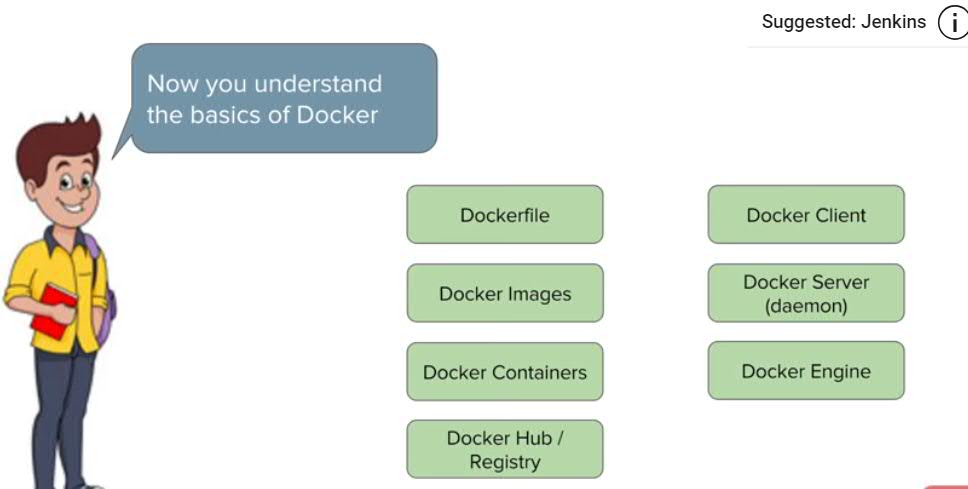
If doesn’t logged in to private repo, then returns access denied message.

If finds the image in local repo, then doesn’t searches in dockerhub. If doesn’t find in local repo, but found in DockerHub, then downloads the image to local repo.

Daemon is a background service. Client speaks with daemon. Daemon check for the images locally, if not found, then tries to search in Docker registry (Docker hub).

Image in docker, is nothing but set of instructions in a file.

Container is running form of image



**Docker commands and working:**

Docker will have 3 types of repositories:

1)Local

2)Private

3)Public

**Docker hub url:** https://hub.docker.com/

**Docker software download url for Windows:** <https://download.docker.com/win/stable/Docker%20for%20Windows%20Installer.exe> (This is just for practice and we don’t use this in realtime).

If installation doesn’t work in your system, then you can use online url to work on Docker. **Use below url:**

https://training.play-with-docker.com/

Open this url, click on the Basics, then click on Your First Linux Containers:

Then login to access, if already not logged in.

Docker has two editions:1) Community edition and 2) Enterprise edition

We use CE for practice and EE in real-time.

To check the docker version, use this command: **docker --version**

**Installing docker on RedHat Linux:** First register in Docker hub.

Then download docker software and install it in your local for practicing.

Use this url to install docker on Linux: <https://docs.docker.com/v1.5/installation/rhel/#red-hat-enterprise-linux-7-installation>

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.

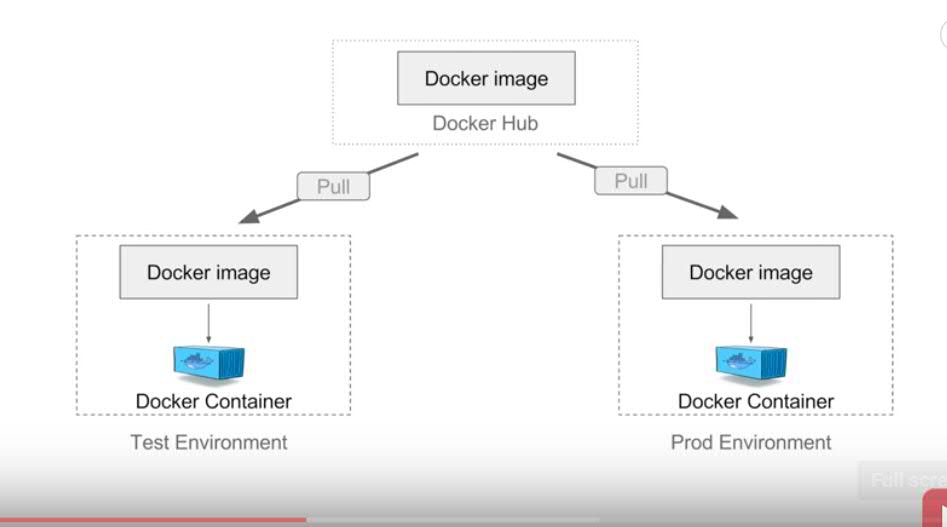
**docker run hello-world**: To create container by running the image. Here, hello-world is the image name.

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 therunning 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 stats 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)

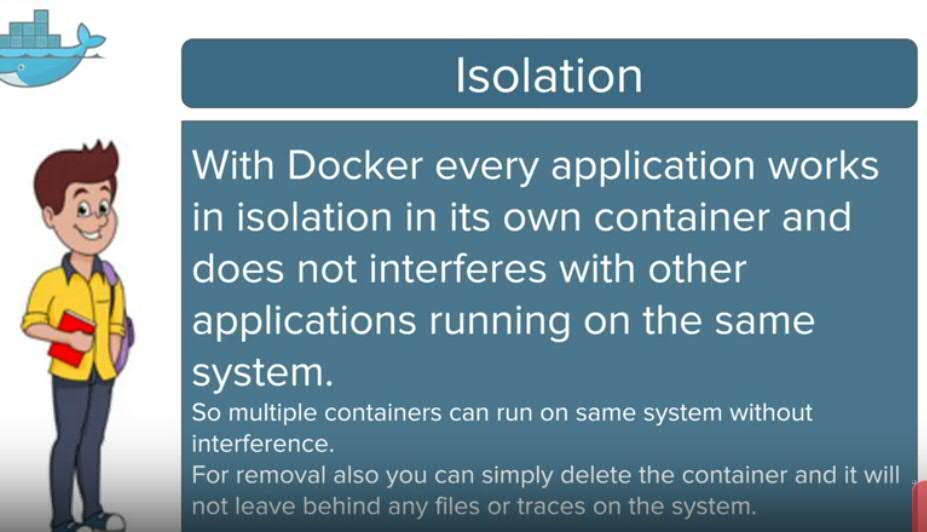
**-it** : Will run in interactive mode and enters in to that container.

**-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.

**detached mode**: means the container will run in backend.

**root user mode**: after completing the running of image, then it will enter into the container.



**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 contaninerName 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:**

To install git on CentOS container, first create centos container in interactive mode, then run below command.

**yum install git** : To download git in CentsOS what we are running in docker with centos image.

**git --version:** To see the installed git version.

**exit:** To come out of centos which is running through docker image.

If you exit without saving, then all the installed data will be lost.

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

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

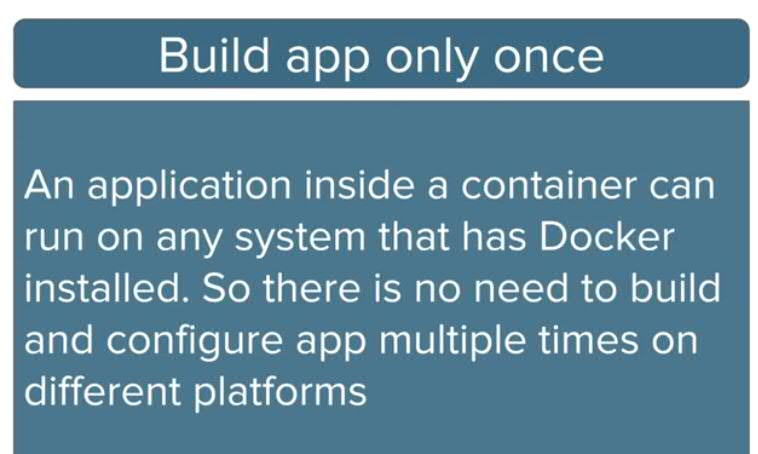
**docker network inspect bridge**: Will give full details about the particular network (ex: bridge network).

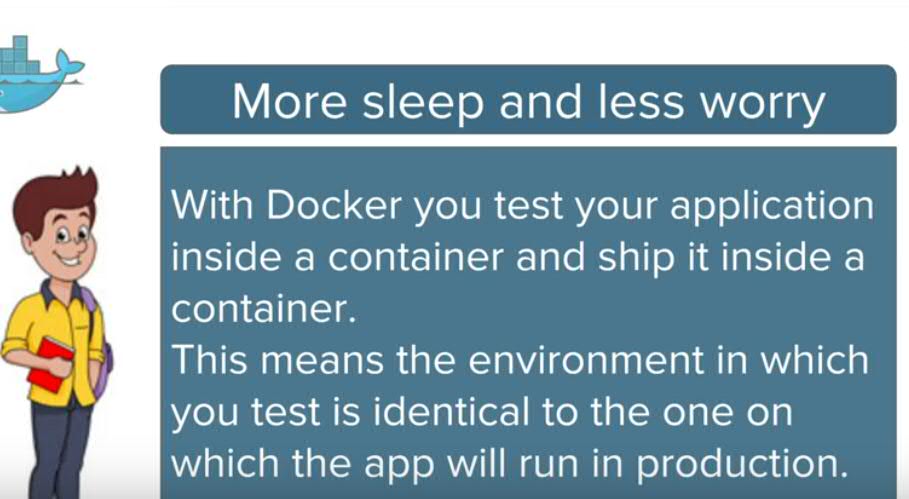
docker network connect: will connect container to the network

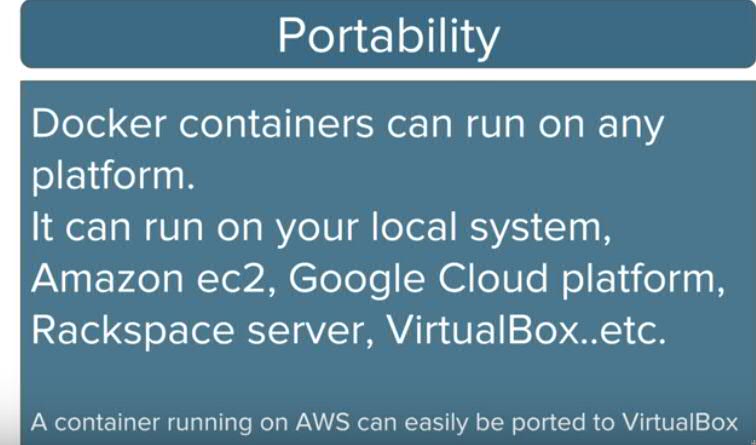
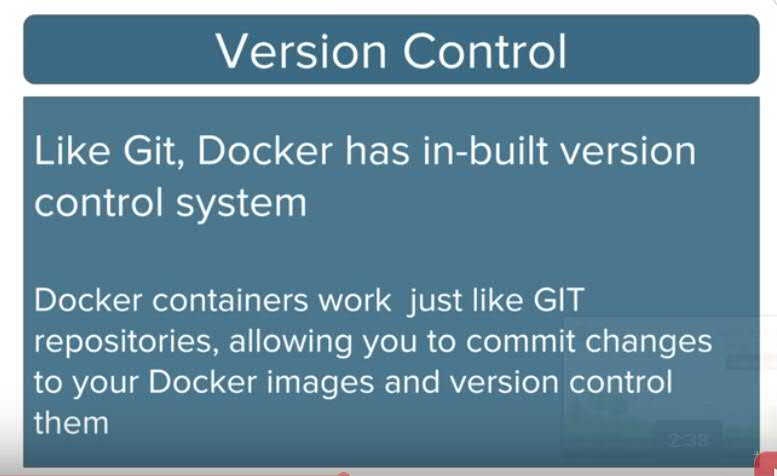
**docker network connect --help:** Will list all possible options for the connect command.

**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.







**Docker Key words:**

**FROM** Centos:7

**MAINTAINER** Krishna Chaitanya kctechnologies@gmail.com

**RUN** yum install java -y

**RUN** mkdir tempdir

OR

**RUN** yum install java\

mkdir tempdir

By using \ at the end of each command, we can write multiple commands. If there is no \, then it indicates it as end of the commands.

Instead of **\** we can use **&&** to use multiple commands.

RedHat commands and centos Commands are same. But those are different for ubuntu.

**FROM** is to give the base image while creating our custom image.

**RUN** command is to run commands to form an image, i.e to run commands

after creating the base image.

**CMD** is used to run on the formed image, i.e CMD will run while starting the container.

**RUN** can be used many times in a Dockerfile, but CMD can be used only once.

**If we are using online Play with docker then create Dockerfile and other files with vim command:**

vim Dockerfile

vim index.html

Then copy paste the text and save.

Then check if the files are existing or not with ls command

./index.html /usr/local/apache2/htdocs/ --------> this is the default location of html files.

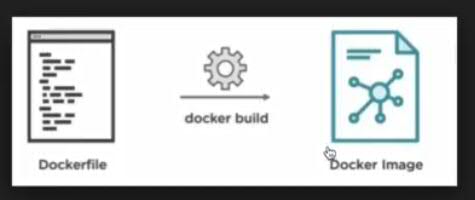
httpd:2.4 creates http server. httpd is an image for http server.

80: default port number for http server

**docker build**: Command to create image from docker file.

Ex: **docker build -t customimagename .**

CustomName can be any name.



. represents Dockerfile exists in current location. If it is in different location, then we have to give the file path.

**docker image ls**

**docker run -p 8009:80 kctechnologies(imagename)**

**Creating custom image to deploy jenkins war into tomcat:**

First check our pre-requisites: They are tomcat 8, java 8

For this you can search for tomcat 8 and java 8 in docker hub. Tag for that is: tomcat:8.0-jre8

ADD -->

COPY -->

**Difference between ADD and Copy**

**Ans:** ADD and COPY are used to get some files to container.

**ADD** has capability to download from internet, whereas **COPY** has ability to copy from our machine (where we are building the image) to the container.

For ADD we give source(host) and destination(cantainer) to copy the stuff. For ADD we can give http also.

While copying jenkins war to tomcat, we need to know the location where our jenkins.war present on our system.

Then need to find where the **webapps** folder present inside the tomcat container. **Ex: /usr/local/tomcat/webapps**

This container path we can find by running the container.

**Ex:** docker run -it tomcat:8.0-jre8 /bin/bash

There you can see webapps path. use that path.

**Dockefile:**

**=======**

**FROM tomcat:8.0-jre8**

**MAINTAINER KrishnaChaitanya kctechnologies@gmail**

**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"]**

**EXPOSE** to set the port number for tomcat.

**CMD** is to start the tomcat. here we are giving in form of array of arguments.

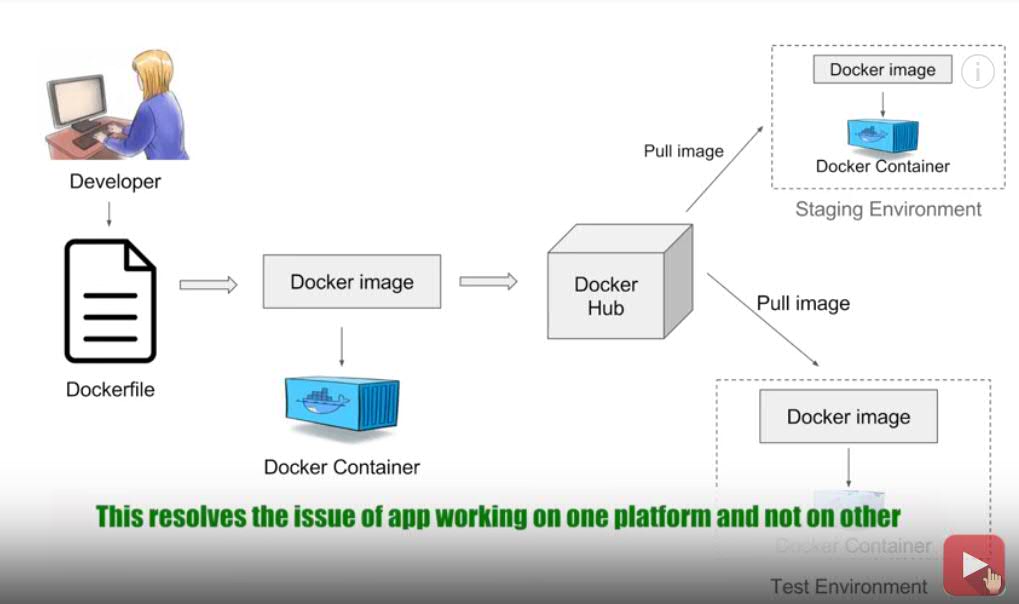
First param in the array says it an executable. Rest of the indexes are arguments.

Then build the image using Dockerfile and run it

**docker build -t myjenkins:1.0 .**

**docker run -p 8080:8080 myjenkins:1.0**

**Running Sample project war and tomcat server image in Docker:**



Create a directory with the name tomcat:

In Dockerfile give FROM as below:

**FROM tomcat:8.0.20-jre8**

In the Dockerfile, FROM is to create base, tomcat is server name and : 8.0.20-jre8 is the version

**COPY ./MyAntProject.war /usr/local/tomcat/webapps/MyAntProject.war**

-> copying the war to webapps with the name MyAntProject.war

To download the war file to play with docker from slack channel.

Use this command: **wget url**

**ex:** wget https://files.slack.com/files-pri/DEVOPS-KCTECH/download/myantproject.war

wget is the command to download files from URL

It will download the war with all small letters. Rename it with capital letters MyAntProject.war

mv mypleantproject.war MyAntProject.war

Then run below command:

**docker build -t javawebdockerimage:tag .** -------> command to convert the java war and tomcat to docker image. **.** represents Dockerfile location.

docker image ls -----> command to check list of docker images.

**docker run -p 9001:8080 javawebdockerimage:latest** -> run this command to download the latest tomcat and run the image javawebdockerimage. Also, we are setting the 8080 tomcat port number to 9001.

Now access the application: http://ipaddress:9001/MyAntProject

To push the created image to Dockerhub, First register in DockerHub, then create an organization and repository.

After that check in Tags, no existing images will be there.

First tag the docker remote hub repo to local and then push from docker to docker hub using below command.

**docker tag imageid orgname/reponame Or usename/imagename**

**docker tag f09c59e7c44d neelkanta1990/neelakanta**

Then push to docker hub

**docker push orgname/reponame**

ex: **docker push neelkanta1990/neelakanta**

If you get access denied issue,

Then login using below command and then try to push

command to login: **docker login**

Then enter username and password of docker hub.

**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 KCTechnologies kctechnologiesdevops@gmail.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 kcjenkinsbatch14:1.0 .

**Step 3) Tag image (docker tag)**

**Ex:** docker tag kcjenkinsbatch14:1.0 kctechnologiesdevops/kcjenkinsbatch14:1.0

**Step 4)** **Login to docker hub (docker login)**

**Ex:** docker login

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

**Ex:** docker push kctechnologiesdevops/kcjenkinsbatch14:1.0

It will automatically take and pushes the tagged images

**docket attach containerId**: To get into the container.

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**

**yum update in redhat is equal to apt-get update in ubuntu.**

yum = apt-get

**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 container instance containerId**: To see the full details of the container like ipAddress etc.

**hostname someHostName**: To change the hostname for the connected ip.

**For creating docker images from a file (Dockerfile):**



Dockerfile is the default file name to create custom images, but we can use any name. But Dockerfile is recommended name.

Create a file with any name in Linux like: Dockerfile

Then add below type of text.

**FROM centos**

**MAINTAINER**

**RUN apt-get update**

**Note**: In a docker file we can have any no.of RUN key words. But should have only 1 CMD keyword.

RUN keyword is to run Linux commands on the given base image.

Then save that file.

Command to build Dockerfile: **docker build -t ouriamgename pathofthedockerfile**

Ex: **docker build -t centos .**

**docker tag imageid orgname/reponame Or usename/imagename**

**docker tag f09c59e7c44d neelkanta1990/neelakanta**

Then push to docker hub

**docker push orgname/reponame**

ex: **docker push neelkanta1990/neelakanta**

**docker push orgname/reponame**

ex: **docker push neelkanta1990/Neelakanta**

**Creating image copy:**

**docker tag existingimageid newimageidname:** Command to create copy of existing image. New image name should be in lower case only.

**Note:** if you give any command after pathofthedockerfile, then the CMD option in the Dockerfile will be overridden by the option we gave after pathofthedockerfile.

Ex: **docker build -t ouriamgename pathofthedockerfile /bin/bash**

--> here /bin/bash will override the **CMD** command inside Dockerfile

Then we can push the image to Docker Hub.

**docker --help**:

**docker stats**: Will show the memory usage of our system from docker.

**docker system df**: To check the disc usage of docker.

**docker system prune --help**: will show info about prune option.

**docker system prune**:

Removes all unused data. This will remove all stopped containers and

Removes all networks which are not used by at least 1 container.

All dangling images.

All build cache.

**Below are options for prune:**

**docker system prune -a or --all**: Removes all unused images not just dangling images (images which are associated or used by at least 1 running container)

**docker system prune -a or --filter**: provides filter values (ex: 'label = <key>=<value>')

**docker system prune -f or --force:** Do not prompt for confirmation.

**docker system prune --volumes**: prune volumes. Means removes all unused volumes, which are not used by at least 1 container.

**docker container prune**: removes all unused containers.

**docker network prune**: removes all unused networks (networks which are not used by at least 1 container)

**One more example of Dockerfile:**

**FROM ubuntu:xenial**

**MAINTAINER KrishnaChaitanya kctechnologies@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/krishna:/opt/krishna ubuntu**

This command will create and runs a new ubuntu container.

here **-v** indicated volumes.

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

This command will create a directory **/opt/krishna** in local. Also creates a **/opt/krishna** 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/krishna** and create a file.   
Then type Ctrl+P+Q. So, you will come back of connected container.

Then go to **/opt/krishna** 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/kctechnologies/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/kctechnologies/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.

**Dockerfile:**

Dockerfile is a simple file, where we give instructions to build docker images.

We use Dockerfile to create our own images.

Dockerfile is the default file name we use; however we can create with different name and can inform docker about the custom file name.

**Step 1:** Create a file named Dockerfile

Create a folder named DockerFiles: Then create a file named **Dockerfile** inside DockerFiles directory.

**Step 2:** Add instructions in Dockerfile

Open the file Dockerfile, Then enter below text and then save it.

**FROM ubuntu**

**MAINTAINER KCTechnologies <email>**

**RUN apt-get update -y**

**CMD ["echo", "Hello, this is my first image creation"]**

After saving, just verify the file content by using cat Dockerfile command.

**Explanation for the above Dockerfile content:**

1)**FROM** indicates the base image. If you don’t want to give a base image, then you can give **FROM scratch**.

**scratch** is am empty image present in Docker Hub, which is used to build images from Scratch.

2)**MAINTAINER** indicates the Name and email of the image creator or maintainer. You can give either Name or email or both.

3) **RUN** indicates to run any command on the command line during the image creation.

4) **CMD** indicates to run any command on the command line during the container creation.

**Difference between RUN and CMD:**

RUN gets executed during the building of the image.

CMD gets executed only when you create a container out of an image.

**Step 3 :** Build Dockerfile to create image

**COMMANDS to build the image**

**docker build**

OR

**docker build -t ImageName:Tag directoryOfDocekrfile**

**Explanation about Docker build Images:**

Go to the location where Dockerfile exists and you want to build.

**docker build .** --> Here dot indicates that Dockerfile exists in current location. And this command is to build/create the image present in current location.

**docker build -t ImageName:Tag directoryOfDocekrfile**: -t indicate tagging of the image. ImageName indicates the name of the image. Tag indicates the Tag/version of the image.

directoryOfDocekrfile indicates the location of the Dockerfile.

Ex: **docker build -t myimage1:1.0 .**

Then click on enter. Then it will create an image as per the sequence of commands given in the Dockerfile.

As per our Dockerfile.

It will pull the ubuntu image from DockerHub if doesn’t exists in local.

Then creates a maintainer.

Then gets apt-get update.

Then runs CMD.

Then verify whether the image is created or not by using the command: **docker images**

**Step 4 :** Run image to create container

**docker run imageName/Id** : This will create the container for the created image.

To know more about Dockerfile commands and creation refer below given docker site or refer my github location:

https://docs.docker.com/engine/reference/builder/#environment-replacement

https://github.com/NeelakantaReddyP/docker-cheat-sheet

**Docker Compose:**

1. What | Why - Docker Compose

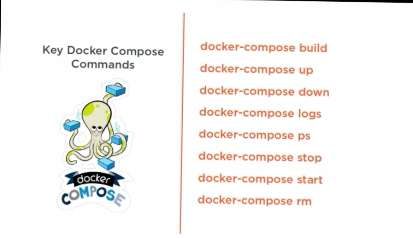
2. How to install

3. How to create docker compose file

4. How to use docker compose file to create services

5. Basic Commands





**Docker compose:**

Docker compose is a tool for defining & running multi-container docker applications.

It use YAML files to configure application services (**docker-compose.yml**)

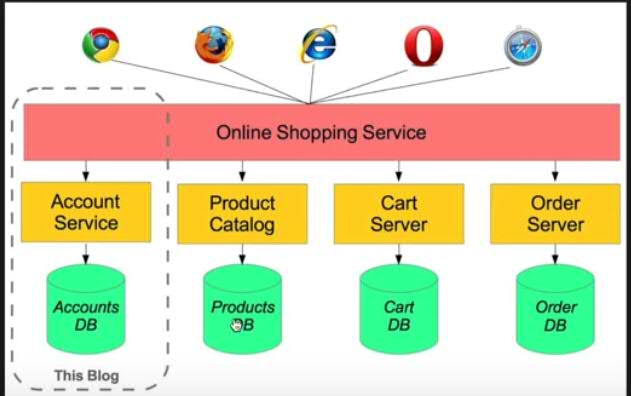
It can start all services with a single command: **docker compose up**

It can stop all services with a single command: **docker compose down**

It can scale up selected services when required



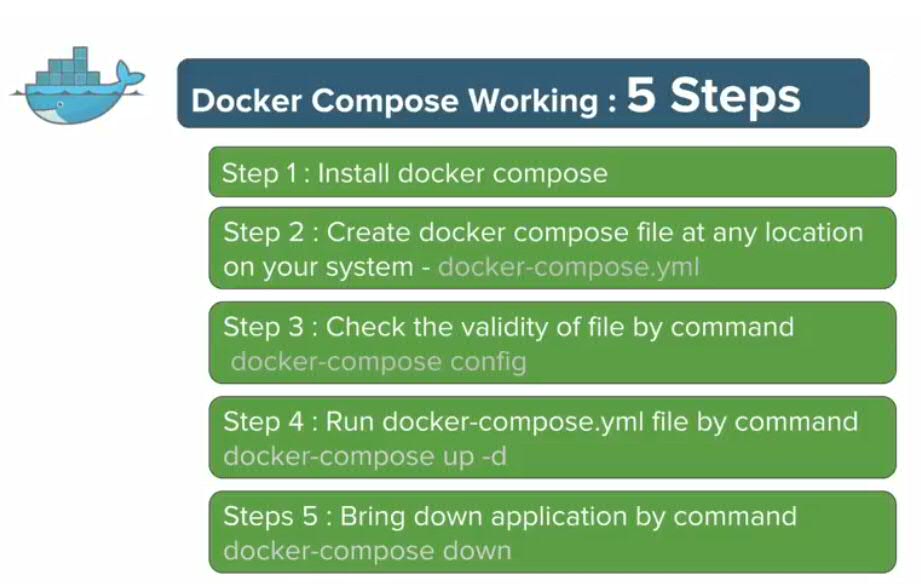
The below image indicates the microservices architecture.



microservices are divided into small modules, so that if modifications are required to one module, then we can modify or scale only that module, without touching any other module.

For example, form the above image, if only one DB size has to be increased.

To achieve these kinds of architecture using docker, we can use Docker compose concept.



**Step 1 :** **install docker compose**

It will be already installed on windows and mac along with docker when you install docker.

Try this command to see the docker compose version: **docker-compose -v** OR **docker compose --version**

If it doesn’t show the version, then it means you haven’t installed docker compose.

**docker compose version**: This will also show the docker compose version along with some other info.

If not installed and you want to install on Linux, then you can install in 2 ways.

**Way 1.** https://github.com/docker/compose/releases

Go to the above GitHub page. Select the latest version and run the curl command.

**Way 2.** Using PIP or Python: If you have installed Python or PIP in your local, then you can use below command.

**pip install -U docker-compose**

**Suggested way to install:**

# wget http://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm

# rpm -ivh epel-release-latest-7.noarch.rpm

**Install Docker Compose**

Install Extra Packages for Enterprise Linux

$ **sudo yum install epel-release**

Install python-pip

$ **sudo yum install -y python-pip**

Then install Docker Compose:

$ **sudo pip install docker-compose**

You will also need to upgrade your Python packages on CentOS 7 to get docker-compose to run successfully:

$ **sudo yum upgrade python\***

To verify a successful Docker Compose installation, run:

$ **docker-compose version**

Step 2 : Create docker compose file at any location on your system. The name of the file will be docker-compose.yml. Remember the extension is yml.

**docker-compose.yml**

You can create a new folder named DockerComposeFiles. Then create docker-compose.yml under DockerComposeFiles directory

Then open that file using command: **vi docker-compose.yml**

Enter the below text in the docker-compose.yml file.

**version: '3'**

**services:**

**web:**

**image: nginx**

**database:**

**image: redis**

Save the above file and check the content of the file using command: **cat docker-compose.yml**

From above file text, we tried to create web application server(ngnix) and database (redis)

Here ngnix and redis are the image names in Docker Hub. You can verify the images names by searching in Docker Hub.

**Step 3:** Check the validity of file by command**: docker-compose config**

Validation is required to check the syntax and version compatibles of compose file.

If you get any errors when you run this command, then there may be a version issue with docker compose or you might be using wrong version of docker compose file.

To solve this, you can specify docker compose version in the first line of docker-compose.yml file like below

**version: '1'**

To know the docker compose version supported by different versions of Docker Engines, we can verify the compatible versions in docker documentation site.

You can directly use this link to check the compatible versions: https://docs.docker.com/compose/compose-file/

If the above command runs successfully, then it list the content of compose file in the terminal.

Printing Content of compose file in the terminal indicates the compose file is validated successfully. Then you can start running the compose file.

**Step 4 :** Run docker-compose.yml file by command: **docker-compose up -d**

**docker-compose up** : Indicates running the compose file. This command must be run from the location where compose file exists.

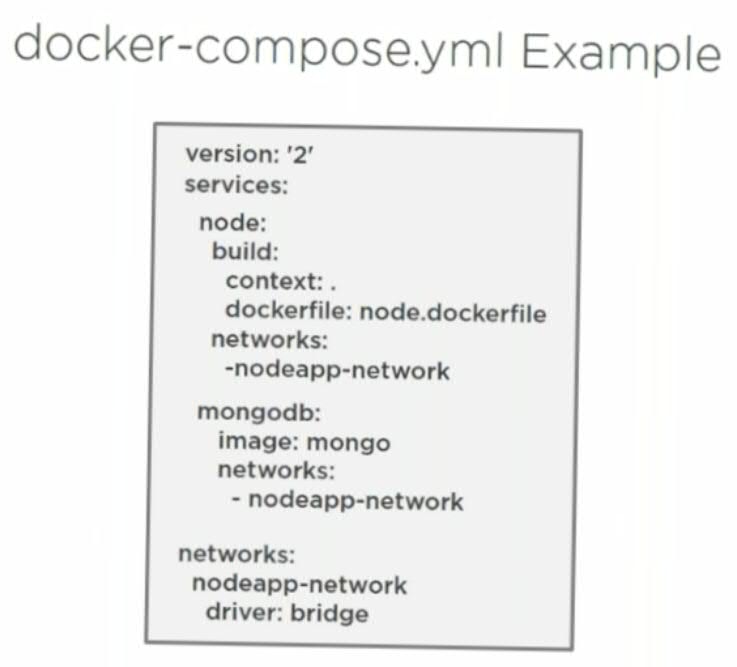
-**d** indicates running the compose file detached state(background).

If you run this command, then it will pull the required images what we mentioned in the compose file, if required.

You can verify the running containers using command**: docker ps**

In your case, we can see ingnix and redis containers created on docker.

**Steps 5 :** Bring down application by command: **docker-compose down**



**How to scale services(—scale):**

Scaling means creating no.of instances of an image while running the docker compose file.

For Example, if we are using microservices like what we have in the diagram and we want to start 4 instances of database, then we can use --scale to create 4 instances for the database.

For example, in our case, we have ngnix and redis images in our compose.

If we want to create 4 instances of redis database, then we can scale by using below command:

**docker-compose up -d --scale database=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.

**Few more important points to remember about docker compose file:**

To know the full details to use any image in docker compose file, then you can search for the image in docker hub, and go to the image details.

There you can find all the details of the image to use in docker compose file.

below is one example to know how the details looks for ngnix image:

Note: you can use below url to search for docker images: <https://hub.docker.com/explore/>

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 we use above sample details and modify our docker-compose.yml, then it looks like below:

version: '3'

services:

web:

image: nginx

ports:

- 9090:80

database:

image: redis

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 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/KCTechnologies/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/KCTechnologies/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.

**VOLUMES and BIND MOUNTS**

Volumes are stored in a part of the host filesystem which is managed by Docker.

Non-Docker processes should not modify this part of the filesystem.

Bind mounts may be stored anywhere on the host system.

Non-Docker processes on the Docker host or a Docker container can modify them at any time.

In Bind Mounts, the file or directory is referenced by its full path on the host machine.

Volumes are the best way to persist data in Docker.

Volumes are managed by Docker and are isolated from the core functionality of the host machine

A given volume can be mounted into multiple containers simultaneously.

When no running container is using a volume, the volume is still available to Docker and is not removed automatically. You can remove unused volumes using docker volume prune.

When you mount a volume, it may be named or anonymous.

Anonymous volumes are not given an explicit name when they are first mounted into a container.

Volumes also support the use of volume drivers, which allow you to store your data on remote hosts or cloud providers, among other possibilities.

**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.

**Few important points:**

**Namespace:**

Docker containers are isolated because of Namespaces.

Each application in a container will have system structure same as our host OS. But that is not actual system structure.

name spaces are: process id name space (pid), net namespace, mnt name space, user namespace.

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

The try ps: it will list all the processes under that jenkins container.

Each application will have process id and this process id will have some other dependent process ids.

**cgroups:**

If we want to set fixed amount of memory to a container, then we set using Control group or CGroup.

Using CGroups we can increase/decrease container memory.

**Capabilities:**

Like sudo users in Linux, we can give access to a user only for specific capabilities like:

For example: **CAP\_NET\_BIND\_SERCIVE** --> Giving only network access to user.

You will get these capabilities from user name space.

**docker run -it jenkins /bin/bash** --> this will run jenkins container in interactive mode. There if you use ls, then you can see the container structure same as our host OS structure.

Go to **cd /etc/** : There you can see all the info how you have in your Host OS etc

It says, containers creates and runs on the OS what they want.

**Ctrl+p+q** : to come out of the container without killing the container.

**Short lived and long lived containers.**

**docker commit 5a8f89adeead newimagename:** To create an image from a containerId/Name

**docker run -ti -v "$PWD/dir1":/dir1 -v "$PWD/dir2":/dir2 newimagename /bin/bash** 🡪 Creating multiple volumes and adding the volume for new image. Stop the existing container and start this new container with this newimage.

**cd /var/lib/docker** ----->Shows the location where docker home and libraries exists.

**Difference between attach and exec.**

**docker attach** depends on the CMD. It opens based on the CMD in docker file.

**docker exec** has the option to give specific path. So, it is always recommended to use docker exec than CMD.

**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.

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

Under this path you can see file called **aufs**. This file is a driver file, which makes docker to work on ununtu OS. It will be different for other Host OS.

**aufs** = augmented file system.

Docker images are downloaded by layers. Containers are combination of image layers. Image layers are reused for another image in Docker.

For every RUN command, a layer will be created

If a same image has to be used by 2 containers, then a separate layer called "**thin read only**" layer for each container will be created.

But no duplication of images layers will be done.

**Data only containers**:

container which will start and kill immediately. But it will have a volume.

Ex: **docker run --name data-only -v /data-mount ubuntu:xenial /run/true**

Then other containers can use this container, so that same volume is used for both.

Ex: **docker run -it --volumes-from data-only ubuntu:xenial /bun/bash**

Here, both the containers use same volume**(/data-mount**).

You can verify the data-mount by running: docker inspect containerId

OR get in to the container interactive mode and verify using : **df -h** command

**docker export:**

**docker checkpoint:**

**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.

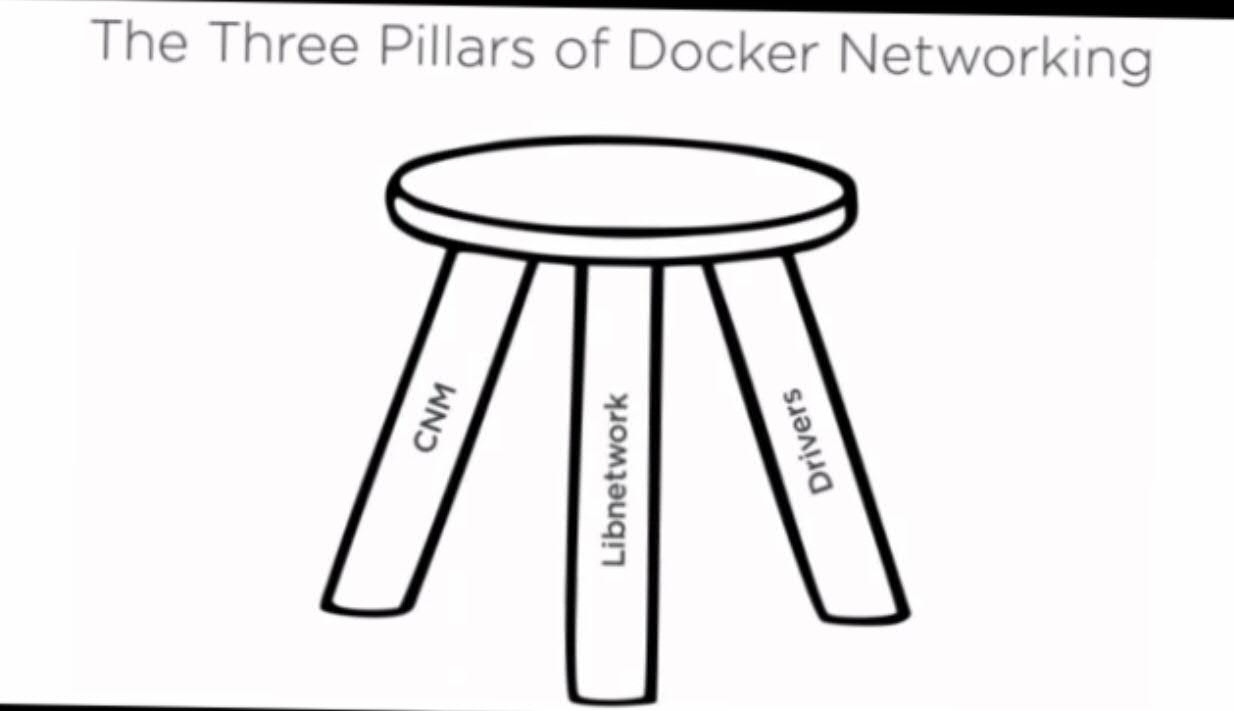
Whenever you install Docker, then an adopter called "**bridge**" will be installed and that will be given to all the containers.

**Bridge** **adopter** will create a connection between docker and HOST.

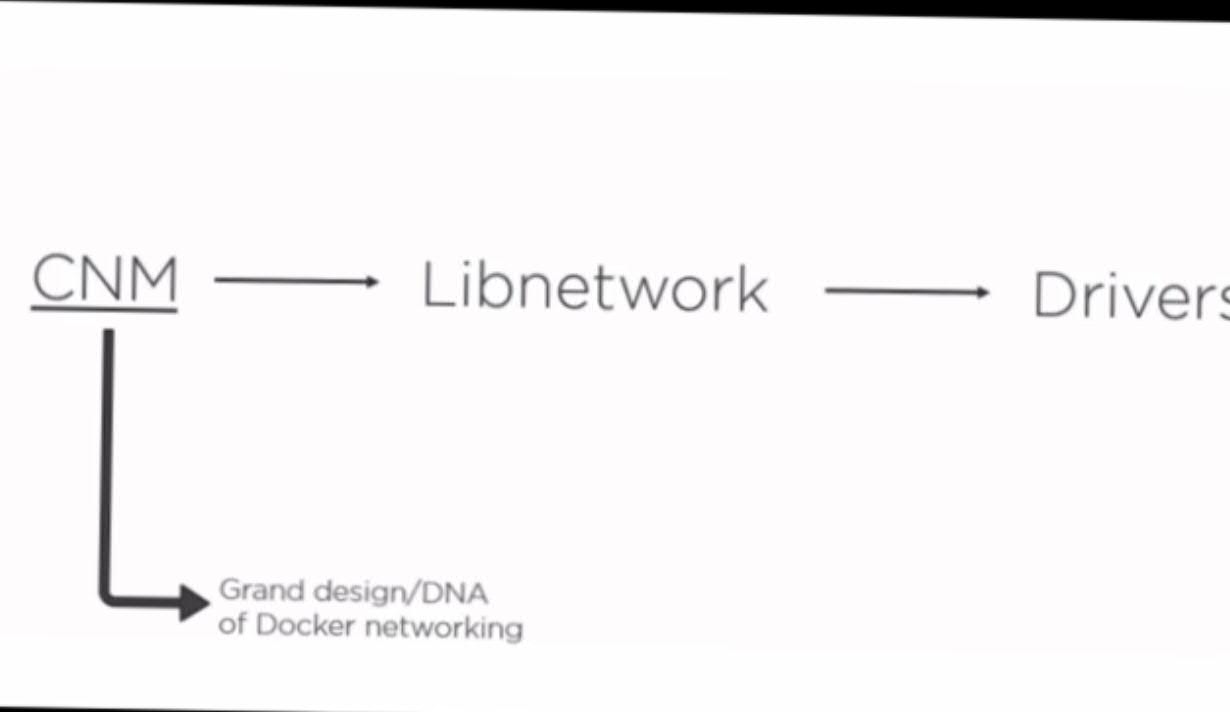
**Loopback** **adopter** is to create connections within the Docker.

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

**3 pillars of Docker Network:**



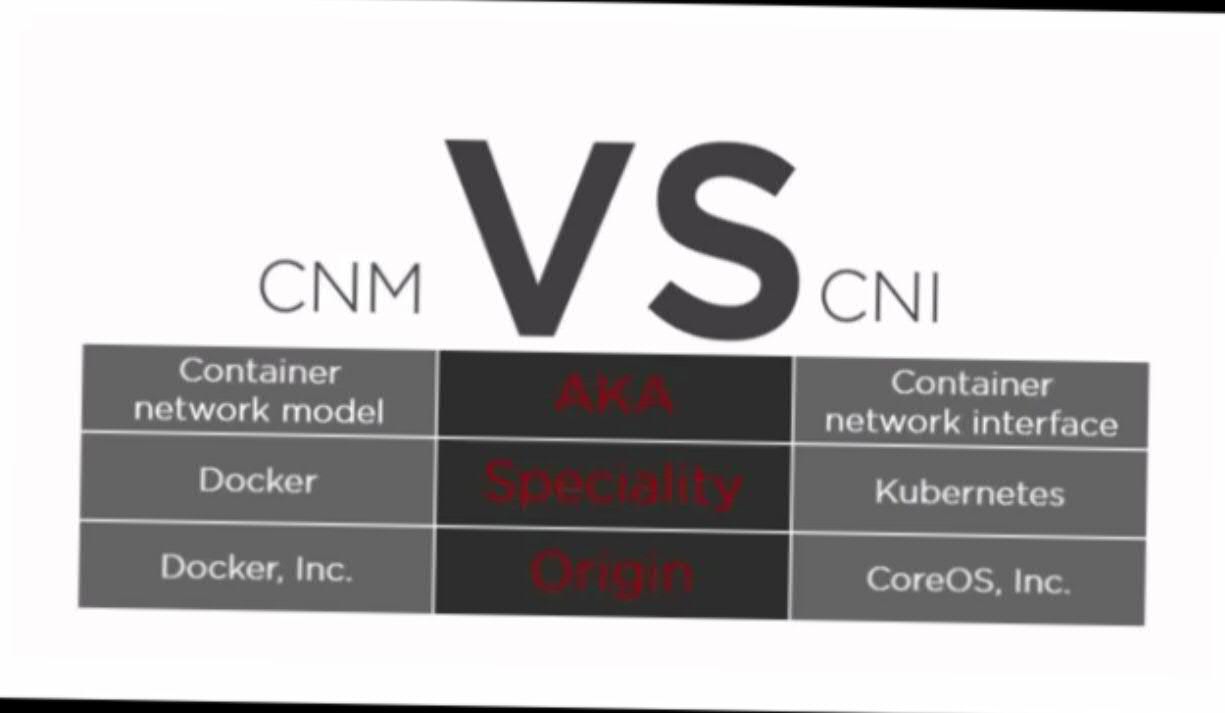
**CNM** : It is not a software. It is requirements documents. It says what has to be done, it was proposed by Docker Inc corporation to maintain thin layers for docker.

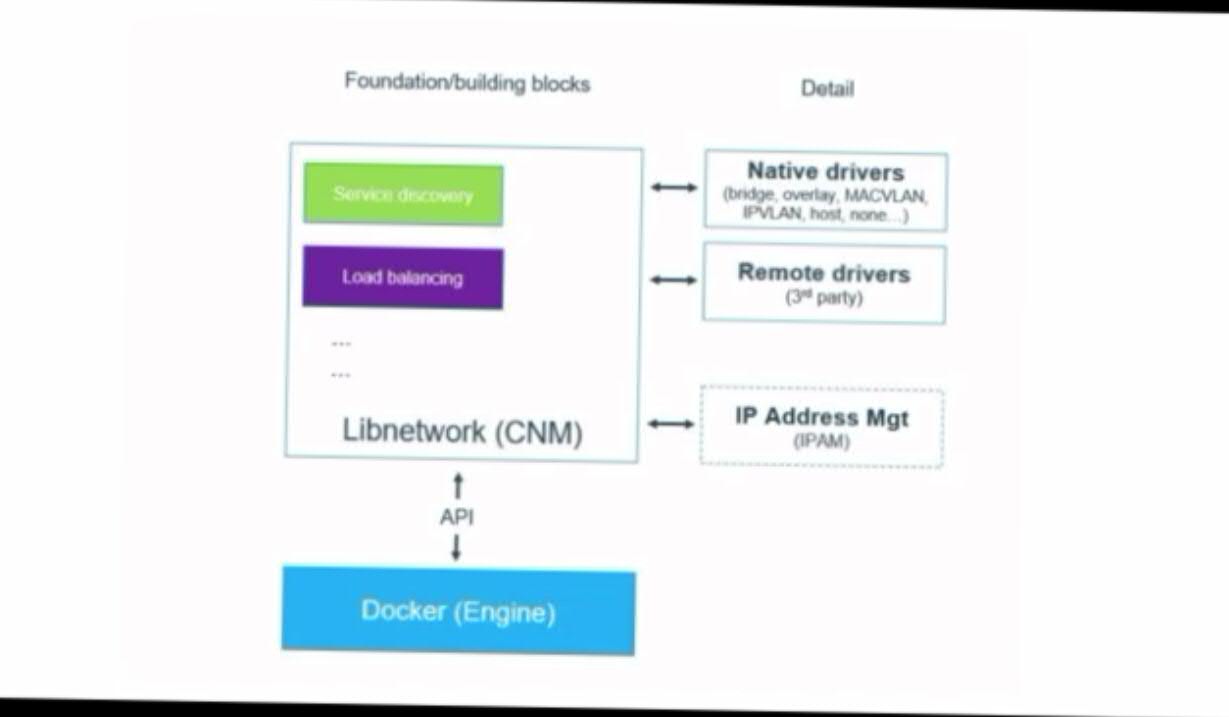


**Libnetwork** : It is the implementation of CNM

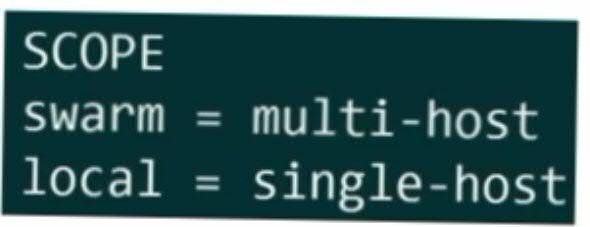
**Driver**: It is specific to networking details, which says what is the kind of network we have to connect to, it is between Host and container, container to container, container to container which presents in different networks.

Drivers are developed in language called GO.





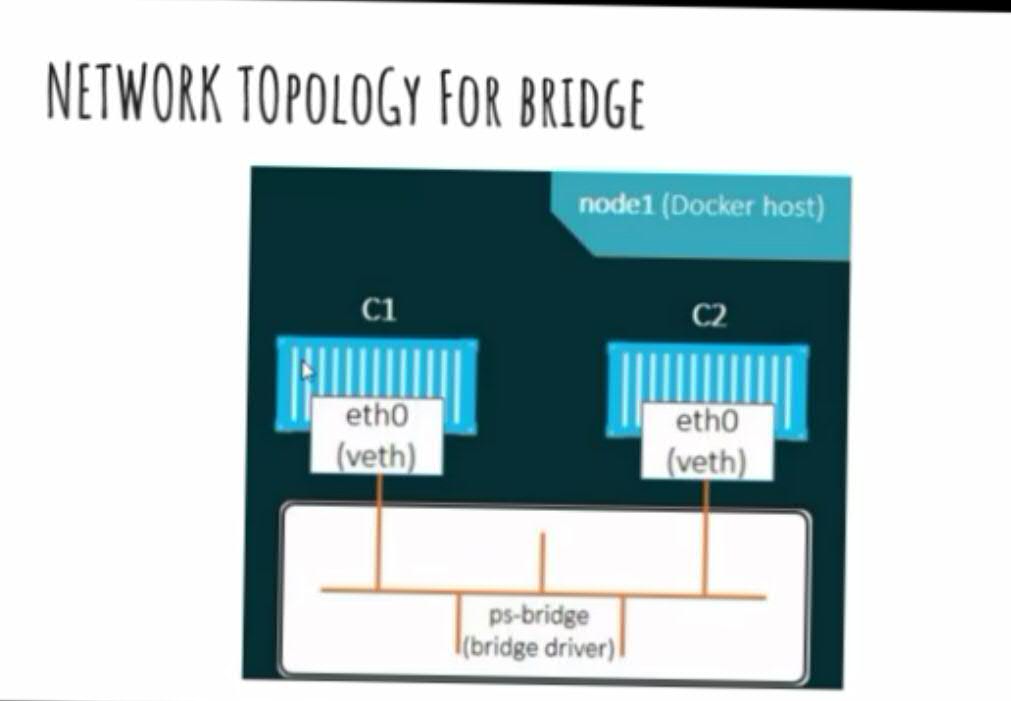
**Networks are of 2 types**: **Single Host and multi Host.**

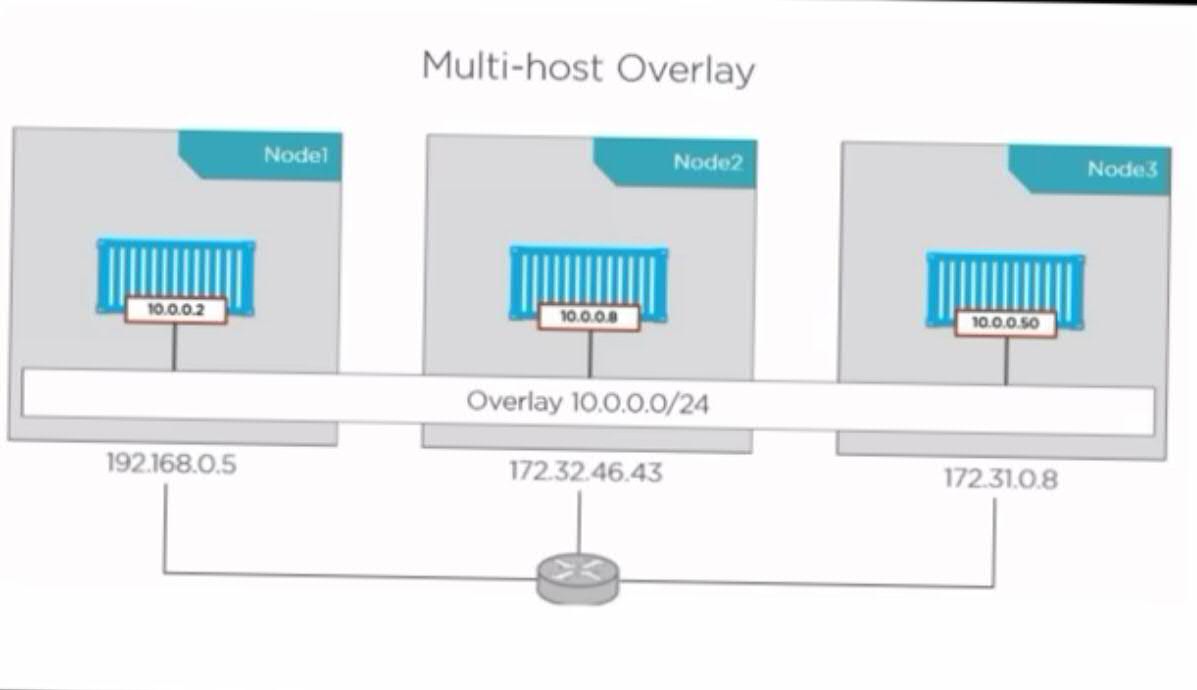


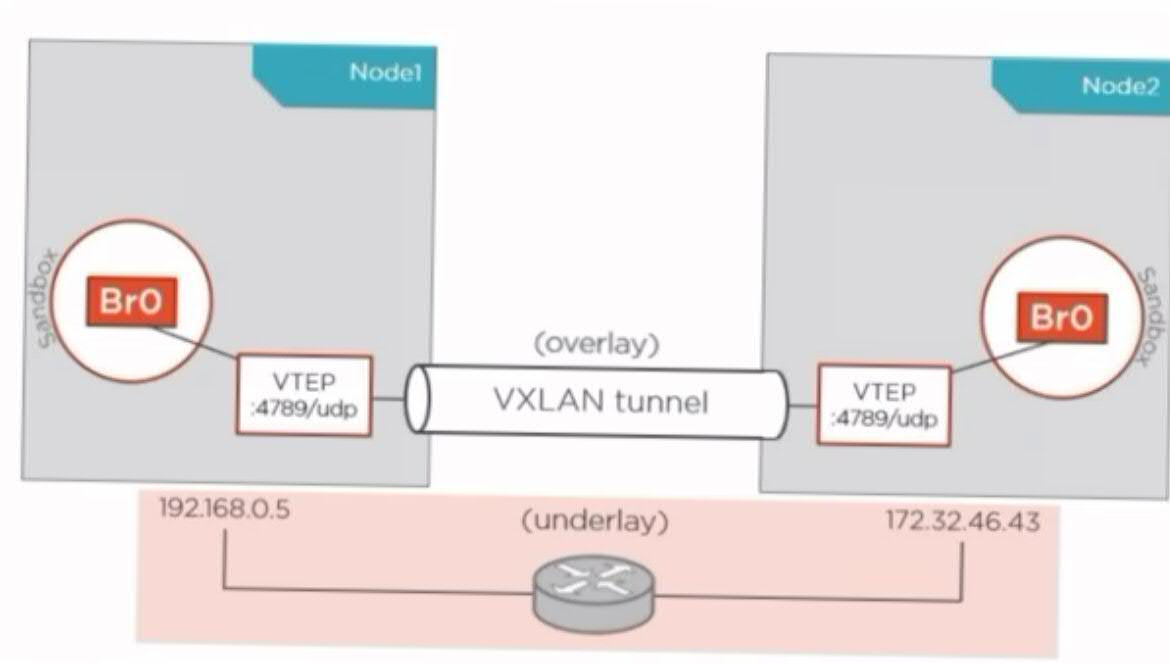
default bridge in Linux is **bridge** and in Windows it is Nat.

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**

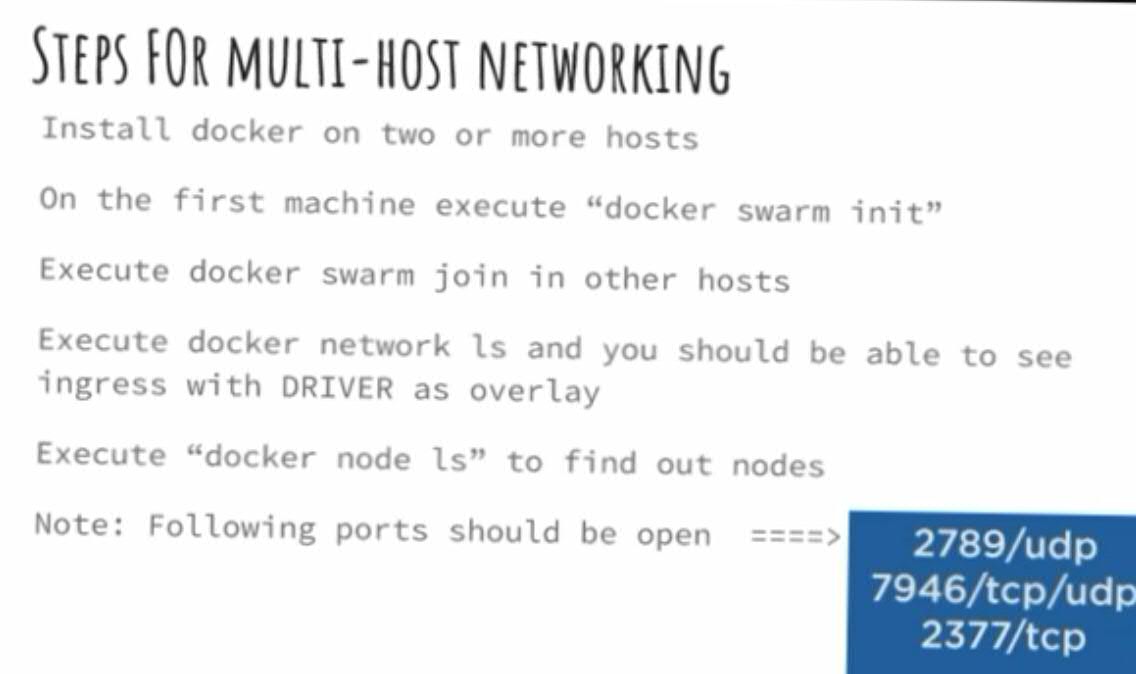
Try**: ping 10.10.0.3** (ContianerName2 IPaddress)

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 kc-tech" --> To create the network

Try executing "docker network ls" on other containers. You will not see the kc-tech as docker follows lazy approach.

To see kc-tech start a container.

"docker service create --name kc-svc --network kc-tech --replicas 3 hello-world sleep 1d"

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

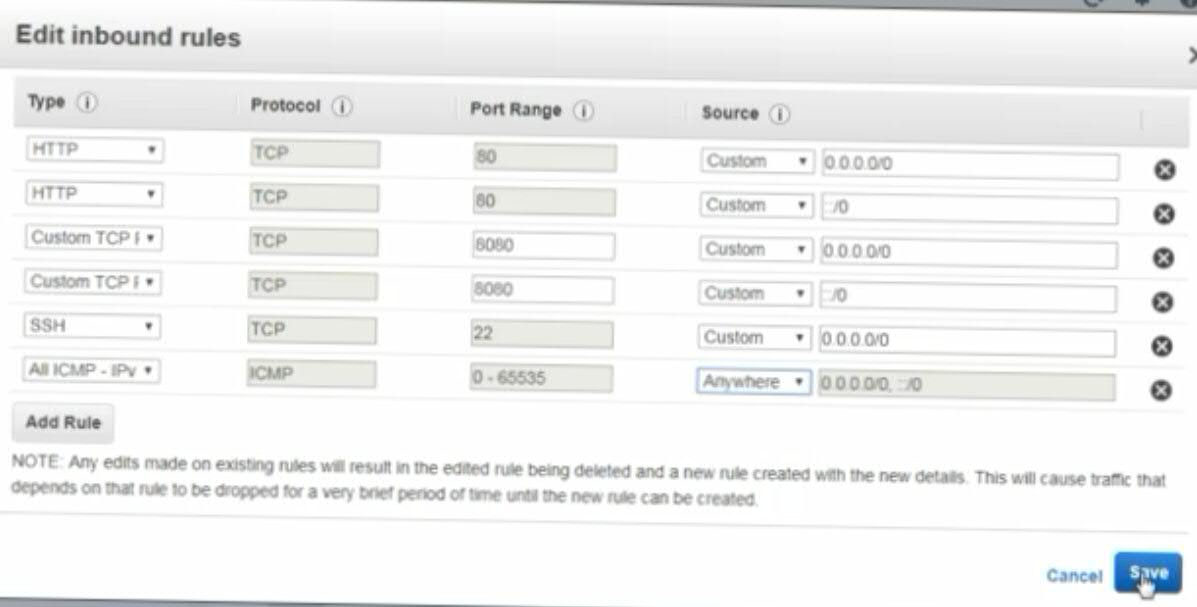
Try to create a service by giving the overlay network created and then try this command: **docker service ls**: This will list the created services, and no.of replicas has been created.

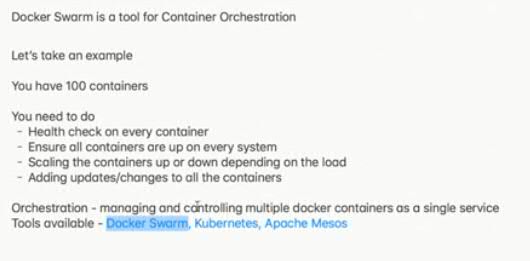
**To inspect:** **docker service inspect serviceName/Id**

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

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

**Ports to be enables on AWS EC2 instance:**





<https://www.youtube.com/watch?v=61Qor_pufn8>

**1) Remove all stopped containers:** docker rm $(docker ps -a -q)

**OR** docker container prune

**2) Remove all running containers:** docker rm $(docker ps -q) -f

**3) Remove all running and stopped containers:** docker rm $(docker ps -a -q) -f

**4) Remove all dangling images:** docker prune