

DOCKER

Types of architecture:

- 1.Monolithic
- 2.Microservices

1.Monolithic Architecture:

- It is an Architecture. For all Services, if we use one server and one database we can call it as monolithic.

Eg: Ecommerce SAAS application (or) take Paytm App - movie tickets, bookings, etc., these are called services

Advantage of Monolithic Architecture:

- Maintenance of a single server is easy.

Disadvantage of Monolithic Architecture:

- If there is any bug in a single service, then the whole server should stop. server is down means we have to shutdown entire application to solve that service. So, user is facing problems because it is tightly coupled

To overcome this we go for Microservices Architecture.

2.Microservices Architecture:

- If every service has its own individual servers then it is called microservices
- Every microservice architecture has its own database for each service
- Take same above example. For every service if we keep 1-database, 1-server it is microservice
- It is loose coupling

Advantage of Microservices Architecture:

- As all the services are deployed in individual servers, if there is any bug in a single service, then that particular server can be stopped, so that the other services are not affected.

Disadvantage of Microservices Architecture:

- Maintenance of many servers is difficult
- too cost also

WHY DOCKER:

Let us assume that we are developing an application, and every application has Frontend, Backend and database

To overcome the above monolithic architecture we're using "Docker"

- So while creating the application we need to install the dependencies to run the code
- So, I installed Java11, reactJS and MongoDB to run the code. After sometime, I need another versions of java, react and MongoDB for my application to run the code
- So, it's really a hectic situation to maintain multiple versions of same tool in our system

To overcome this problem we will use "Virtualization"

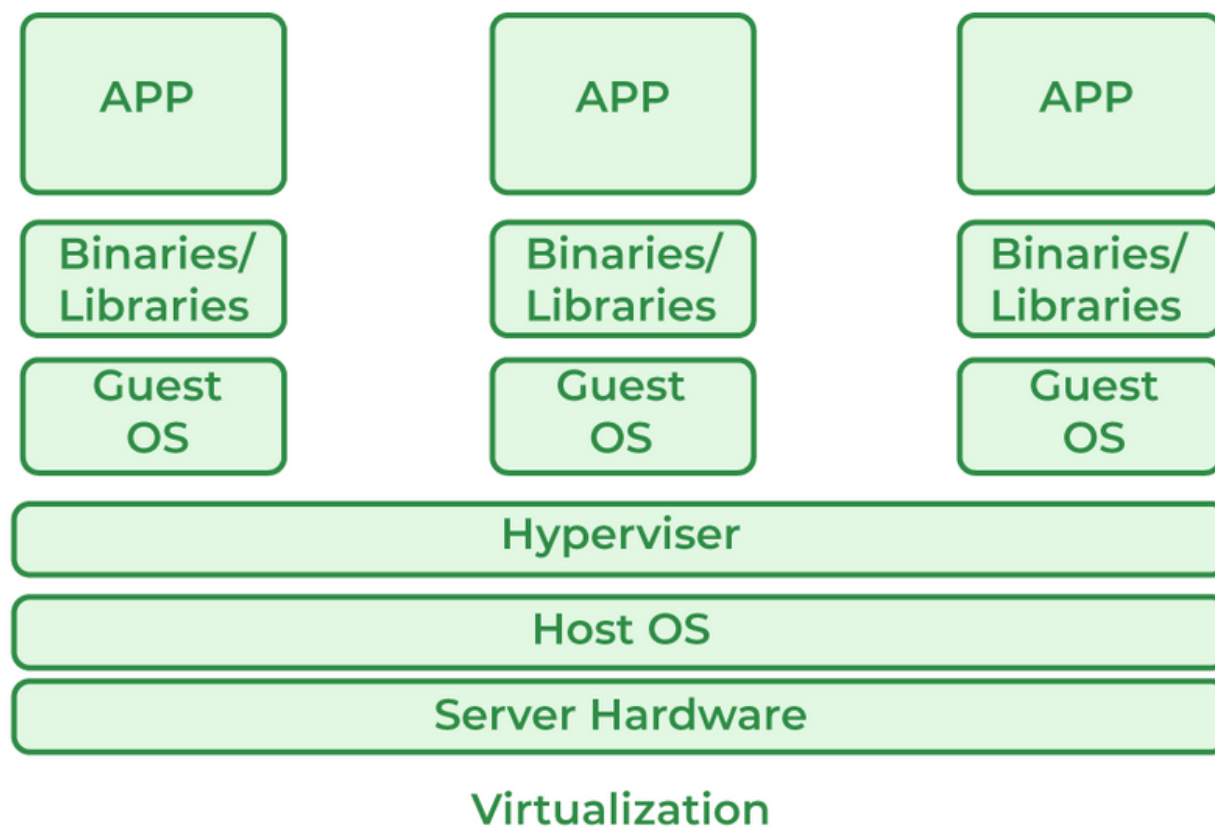
Virtualization:

- **It is used to create a virtual machines inside on our machine b using our local system configurations.** In that virtual machines we can hosts guest OS in our machine
- By using this guest OS we can run multiple application on same machine

- Here, Host OS means our windows machine. Guest OS means virtual machine
- Hypervisor is also known as Virtual machine monitor (VMM). It is a component/software and it is used to create the virtual machines

Disadvantages:

- old method
- If we use multiple guest OS (or) Virtual machines then the system performance is low



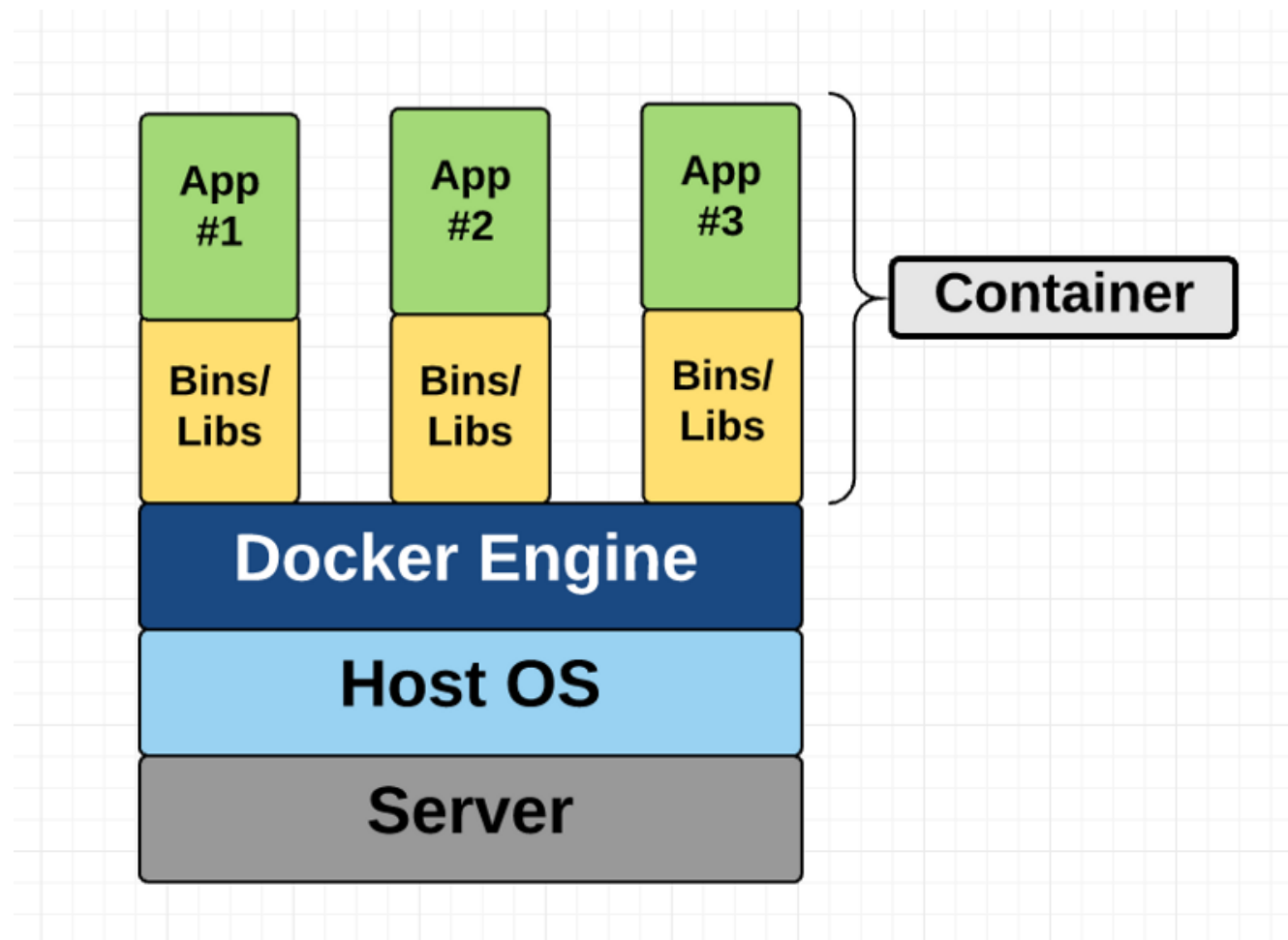
To overcome this virtualization, we are using “Containerization” ie., called Docker

In the docker, we use containerization .

Containerization: It is used to pack the application along with dependencies to run the application.

containers:

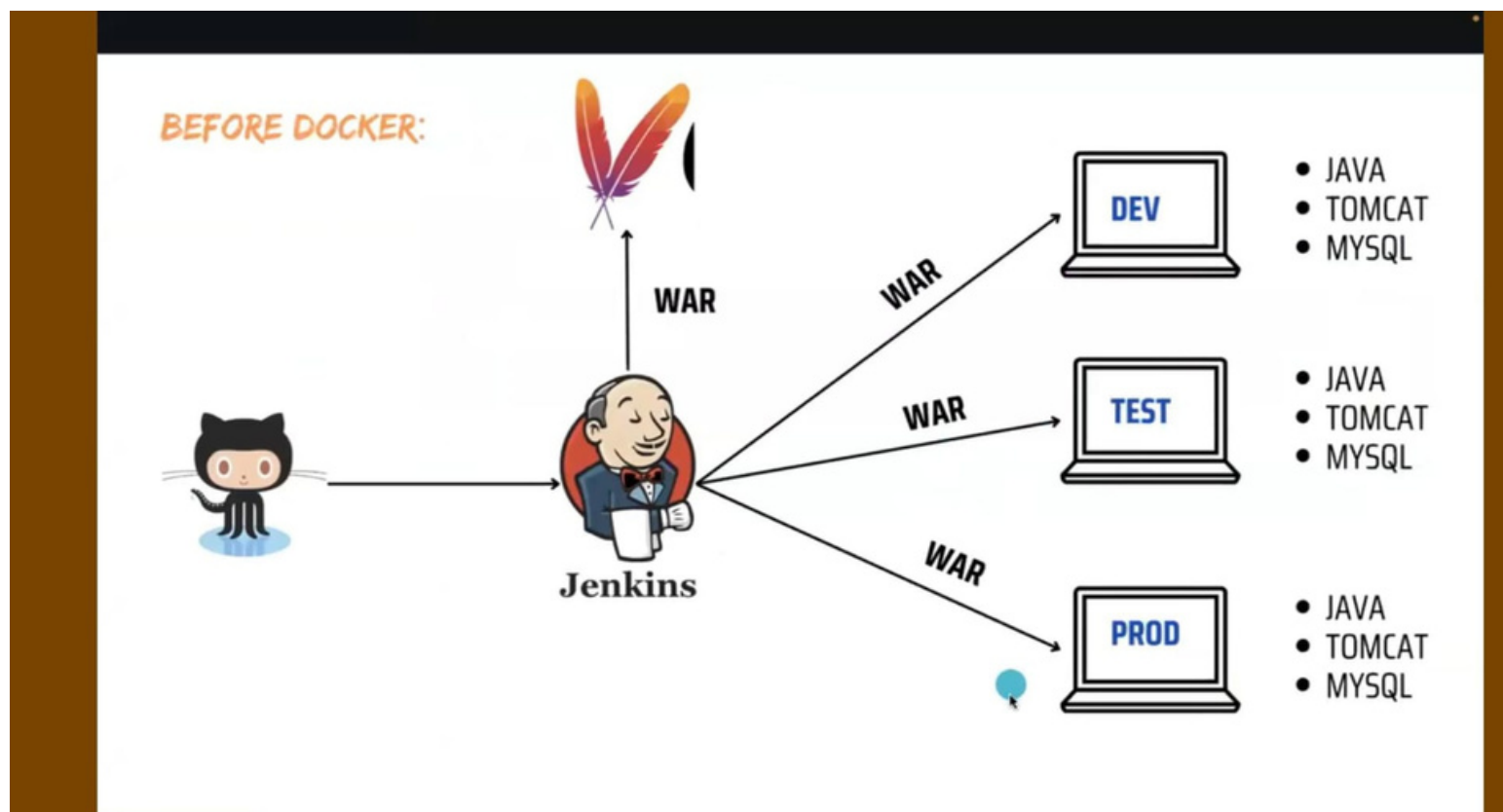
- It's the runtime of the application which is created through docker image
- Container is nothing but it is a virtual machine, which doesn't have any OS
- With the help of images container will run
- Docker is a tool. It is used to create the containers



- It is similar to virtualization architecture, instead of hypervisor we are having Docker Engine
- Through Docker Engine we're creating the containers
- Inside the container we're having the application
- Docker Engine - The software that hosts the container
- In one container we can keep only image

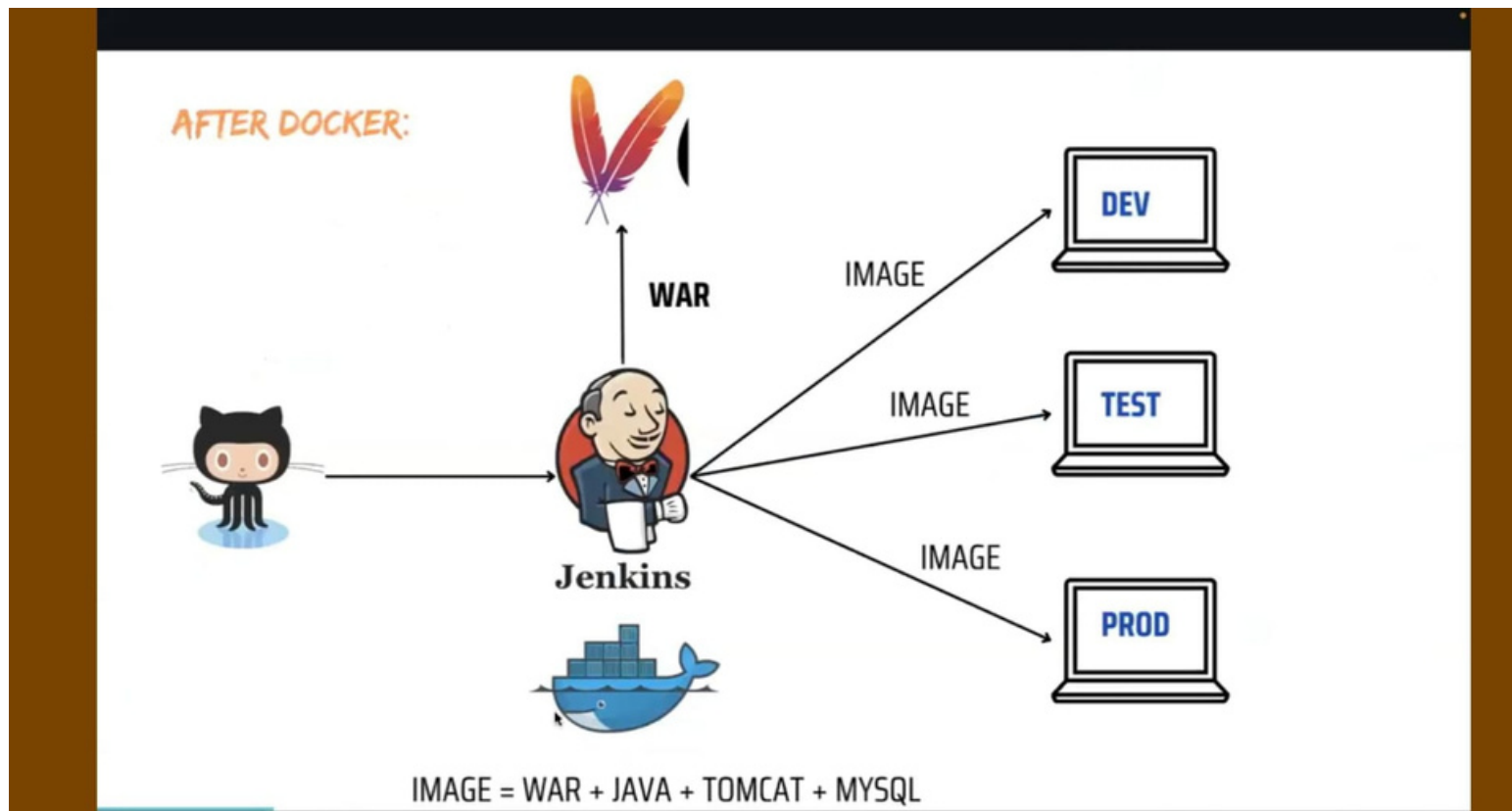
BEFORE DOCKER:

- First, get the code from the GitHub and integrate with Jenkins
- Integrate maven with Jenkins. So, we get War file
- So, that war file we have to deploy in different environments
- So, if you want to deploy war file/application we have to install the dependencies



After Docker:

- we get the code from git hub, build the jar/war fi le in jenkins and we pack the dependencies and war fi le into a image and we will send that image to the server.
- Image= war + java + tomcat + mysql.

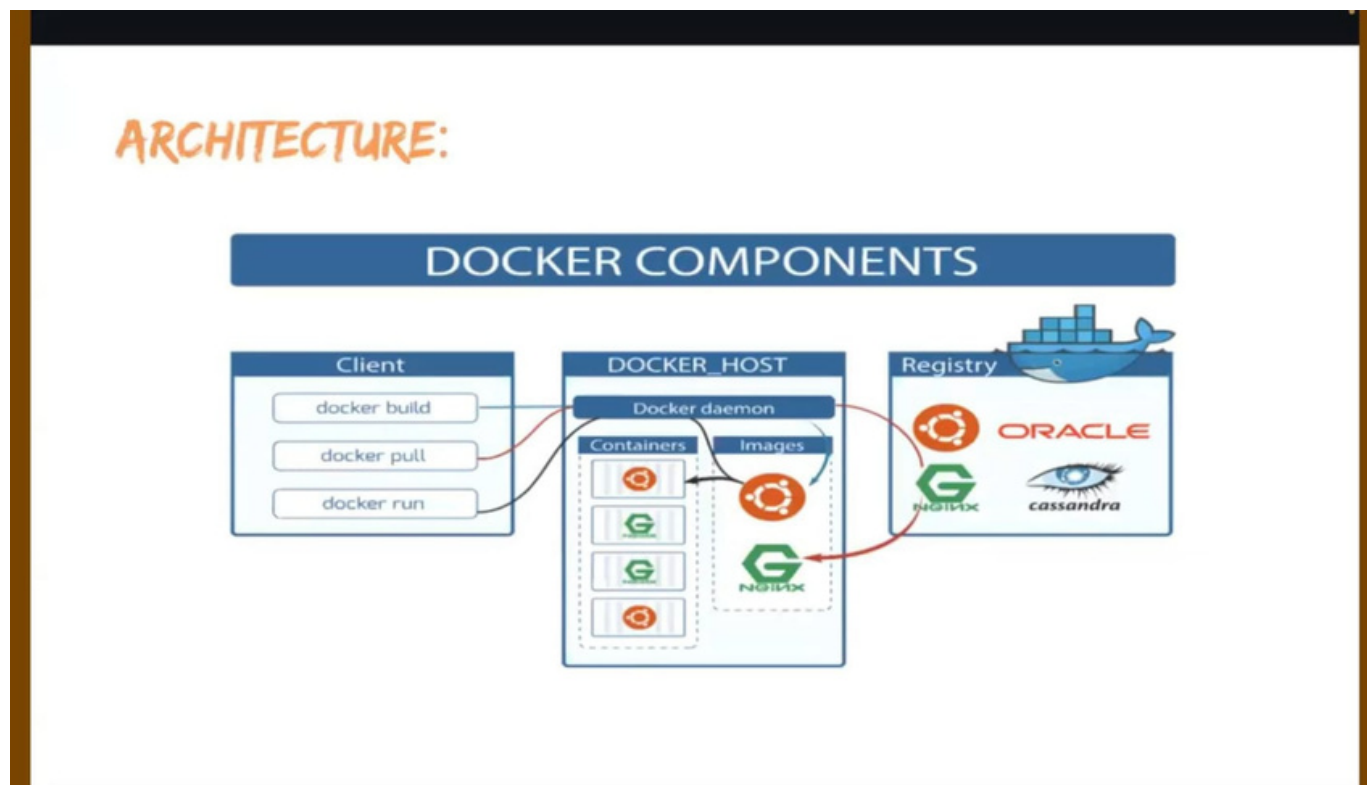


- So, overall after docker, In any environment no need to install dependencies. We can just run the images in that particular environment. If container is created means application created

DOCKER:

- It is an open source centralized platform designed to create , deploy, and run applications.
- It is written in the Go Language.
- it enables developers to package applications into containers.
- Before Docker , many users face the issue that particular code is not running in the user system but running in developers system which is because of dependency issue. Due to containerization the issue is resolved.

DOCKER ARCHITECTURE:



We are having 4 components

1. **Docker Client**

- a. It is a primary way that many docker users interact with docker. When you use commands such as docker run, the client sends these commands to docker daemon, which carries them out
- b. The docker commands use the docker API
- c. Overall, here we perform the commands

2. **Docker Host**

- a. It contains containers, images, volumes, networks
- b. It is also a server, where we install the docker in a system

3. **Docker Daemon**

- a. Docker daemon runs on the host OS.
- b. It is responsible for running containers to manage docker services
- c. Docker daemon communicates with other daemons
- d. It offers various docker objects such as images, containers, networking and storage

4. **Docker Registry:**

- a. A Docker registry is a scalable open-source storage and distribution system for docker images
- b. It is used for storing and sharing the images
- c. Eg: For git we had GitHub. Same like for docker we had Docker registry

Advantages of Docker:

- Caching a cluster of containers
- Flexible resources sharing
- Scalability - Many containers can be placed in a single host
- Running your service on network that is much cheaper than standard servers

DOCKER COMMANDS:

INSTALLATION & SYSTEM COMMANDS:

- To install a docker : **yum install docker -y**
- To check the docker version: **docker version**
- To check docker information: **docker info**
- To start the docker services: **systemctl start docker**
- To check the status: **systemctl status docker**

IMAGE COMMANDS:

- To see a list of downloaded images: **docker image**
- To pull image from Docker Hub : **docker pull image-name**

CONTAINER COMMANDS:

- To view the list of running containers:
docker ps / docker container ls
- List all the running and stopped containers:
docker ps -a / docker container ls -a
- create&run the container:
docker run -it --name <container_name> <image_name>
(**it** -> {interactive terminal}, it will creates a shell inside the container, which is used to perform commands or execute the programs)

- To start an existing container: **docker start <my-container>**
- To stop the container: **docker stop <my-container>**
- To start a stopped container: **docker start <container>**
- To Stop a running container: **docker stop <container name>**
- To attach with a container: **docker attach <container name>**
- To exit from the container: **exit**
- Return from a running container without stopping : **Ctrl+P+Q**
- To view the list of containers that have exited:
docker ps -f "status=exited"
- To Remove the stopped container: **docker rm <my-cont>**
- To Stops the specified containers:
docker stop cont1 cont2 cont-3
- To Stop the all running containers: **docker stop \$(docker ps)**
- To Starts the specified containers:
docker start cont-1 cont-2 cont-3
- To Start all stopped or exited containers:
docker start \$(docker ps -a)
- Rename the container: **docker rename <old_name> <new_name>**
- Forcefully stops the specified container: **docker kill my-container**
- Lists the last 2 created containers: **docker ps n -2**

- To Lists the last 2 created containers, including stopped ones:
docker container ls -a -n 2
- To Shows the latest created container:
docker container ls --latest
- To Removes all containers, including stopped ones:
docker container rm \$(docker container ls -a)
- To delete all stopped containers:
docker container prune
- to inspect the image:
docker image inspect <image>
- to remove the images:
docker rmi \$(docker images) or docker images prune

Day-3

- `yum install docker -y && systemctl start docker`
- `systemctl status docker`
- `docker pull nginx`
- `docker run -it -d --name cont1 nginx :`
 - d ---> detach ---> when we use webserver images or application server images or database related images then we can use -d .
- `docker ps -a`
- `docker inspect <Cont> :` Display detailed information on one or more containers
- `curl 172.17.0.2`
- `docker run -it -d --name cont2 -p 9090:80 nginx :`
 - 9090: hostport ---> used to access the application which is running inside the container.
 - 80: containerPort ---> depends on the image that we are attaching to the container.
 - nginx default port number is =80
- `docker run -it -d --name cont3 -p 9091:80 httpd`
- `docker run -it -d --name cont4 -p 8089:80 shaikmustafa/mygame`
- `docker exec <cont1>ls :`
 - exec : which is used to perform the commands inside the container without going into the container
- `docker exec cont1 mkdir flm :` creating folder inside the container
- `docker exec -it cont1 /bin/bash :`
 - to go into the container .sometimes /bin/bash will won't work instead of that we can use only bash

Docker images:

A Docker image is a file used to execute code in a Docker container. Docker images act as a set of instructions to build a Docker container, like a template. Docker images also act as the starting point when using Docker. An image is comparable to a snapshot in virtual machine (VM) environments.

CREATE IMAGE FROM CONTAINER:

- First it should have a base image - `docker run nginx`
- Now create a container from that image - `docker run -it --name container_name image_name /bin/bash`
- Now start and attach the container
 - go to tmp folder and create some files (if you want to see the what changes has made in that image - `docker diff container_name`)
- exit from the container
- now create a new image from the container - `docker commit container_name new_image_name`
- Now see the images list - `docker images`
- Now create a container using the new image
- start and attach that new container
- see the files in tmp folder that you created in first container.

image commands:

```
docker run -itd --name <paytm> <ubuntu>
docker exec -it <paytm> /bin/bash
docker commit < paytm> < myimage>
docker run -it --name <cont33> <myimage>
```

Day-4

Docker File:

- docker file--basically a text file which contains some set of instructions.
- always D capital on Docker file

Docker file components:

1.FROM : this component is used to define images in docker file

ex:FROM ubuntu

FROM nginx

FROM https

2.LABEL : used to define the author of the docker file

ex: name: mustafa

3.COPY : used to copy the files from our local(ec2) system to container

ex:copy source_file destination_file

4.ADD : used to copy the files from our local to container and it will download the files from internet and send to container.

ex: ADD url destination(container)

5. RUN : it is used to run a command or else to perform a command while we build the image.

ex: RUN touch aws

RUN yum install git -y

```
FROM centos:centos7
LABEL name flm
RUN mkdir lovely
WORKDIR /chary
RUN touch azure.txt
```

6.CMD : used to perform a command while we run the image.

```
root@ip-172-31-84-206:~  
FROM centos:centos7  
CMD ["yum", "install", "httpd", "-y"]
```

7.ENTRYPOINT: used to perform a command while we run the image.

entrypoint will have high priority than CMD

entrypoint will overwrite the values of CMD

```
root@ip-172-31-84-206:~  
FROM centos:centos7  
ENTRYPOINT ["yum", "install", "nginx", "-y"]
```

8.WORKDIR : used to create a path inside a container and we will directly go to that path.

ex: WORKDIR/myapp

9.ENV : used to pass variables .

it will not overwrite the values in runtime.

we can access these values inside a container.

10.ARG: used to pass variables.

we can overwrite the values in runtime.

we cannot access these values inside a container.

11.EXPOSE: used to publish a container port.

note: this is only for documentation purpose.

ex: EXPOSE 80

```
FROM centos:centos7
RUN ["yum", "install", "git", "-y"]
RUN ["yum", "install", "tree", "-y"]
RUN ["yum", "install", "maven", "-y"]
```



root@ip-172-31-84-206:~

```
FROM centos:centos7
ARG cloud=aws
RUN echo i am learning $cloud
ENV course=devops
RUN echo i am learnig $course
~
~
```

TO BUILD : **docker build -t <image> .** (“.” is present directory)
image is used to create the new container then go to the container see the files.

day-5

creating a dockerfile to deploy the application in various types :

in apache

```
root@ip-172-31-88-52:~  
FROM ubuntu  
RUN apt update -y  
RUN apt install apache2 -y  
COPY index.html /var/www/html/  
CMD ["/usr/sbin/apachectl", "-D", "FOREGROUND"]  
~
```

in nginx

```
FROM ubuntu  
RUN apt update -y  
RUN apt install nginx -y  
COPY index.html /var/www/html/  
EXPOSE 80  
CMD ["nginx", "-g", "daemon off;"]  
~
```

httpd

```
FROM centos:centos7  
MAINTAINER name chary  
RUN yum install httpd -y  
COPY index.html /var/www/html/  
EXPOSE 80  
CMD ["/usr/sbin/httpd", "-D", "FOREGROUND"]  
~
```

we need to use images instead of using operating system:

```
root@ip-172-31-88-52:~
```

```
FROM nginx
COPY index.html /use/share/nginx/html
```

```
root@ip-172-31-88-52:~
```

```
FROM httpd
COPY index.html /usr/local/apache2/htdocs/
```

To build:: `docker build -t <image>:<tag> .`

TO RUN : `docker run -itd --name <con_name> -p 5555:80 <image>`

day-5

DOCKER VOLUMES :

If we have two containers , then the changes made in container2 will not be reflected in container1 .
that'swhy we are using DOCKER VOLUMES.

We can map volumes in two ways:

- Container -----container
- Host -----container

USES OF VOLUMES:

- Decoupling Container from storage.
- Share Volume among different Containers.
- Attach Volume to Containers.
- On deleting Container Volume will not be deleted.

CREATING VOLUMES FROM DOCKER FILE:

- Create a Docker file and write
FROM ubuntu
VOLUME["/myvolume"]
- build it - `docker build -t image_name .`
- Run it - `docker run -it --name container1 ubuntu /bin/bash`
- Now do `ls` and you will see myvolume-1 add some files there
- Now share volume with another Container - `docker run -it --name container2(new) --privileged=true --volumes-from container1 ubuntu`
- Now after creating container2, my volume1 is visible
- Whatever you do in volume1 in container1 can see in another container
- `touch /myvolume1/samplefile1` and exit from container2.
- `docker start container1`
- `docker attach container1`
- `ls/volume1` and you will see your samplefile1

```
FROM ubuntu
ADD file1 /ubuntu1/file
VOLUME /ubuntu1
```

creating volumes from command

- `docker run -it - --name container3 -v /volume2 ubuntu /bin/bash`
 - now do `ls` and `cd volume2`.
 - Now create one file and exit.
 - Now create one more container, and share Volume2 - `docker run-it - --name container4 - - --privileged=true - -volumes-from container3 ubuntu`
 - Now you are inside container and do `ls`, you can see the Volume2
 - Now create one file inside this volume and check in container3, you can see that file
-
- `docker run -it --name cont1 -v /flm ubuntu`
 - `docker run -it --name cont2 --privileged=true --volumes-from cont1 ubuntu`

HOST TO CONTAINER:

- Verify files in `/home/ec2-user`
- `docker run -it - --name hostcont -v /home/ec2-user:/raham - --privileged=true ubuntu`
- `cd raham` [raham is (container-name)]
- Do `ls` now you can see all files of host machine.
- Touch file1 and exit. Check in ec2-machine you can see that file

To create a container with that folder:

`docker run -it --name container_name -v`
`source_path:destination_path --privileged=true image_name`
(or)

`docker run -it --name container_name -v $(pwd):destination_path --`
`privileged=true image_name`

- To delete the volume: **docker rm volume volume_name**
- To delete the list of all unused volumes :**docker volume prune**

Note: If the volume is attached to any container , we cannot delete that volume.

- To create the volume:**docker volume create volume_name**
- To see the list of all volumes:**docker volume ls**

MOUNTING THE VOLUMES:Creating a new container with the existing volume.

attach a volume to a container: `docker run -it --name=example1 --mount source=vol1,destination=/vol1 ubuntu`

send some files from local to container:

› create some files

› `docker run -it --name cont_name -v "$(pwd)":/my-volume ubuntu`

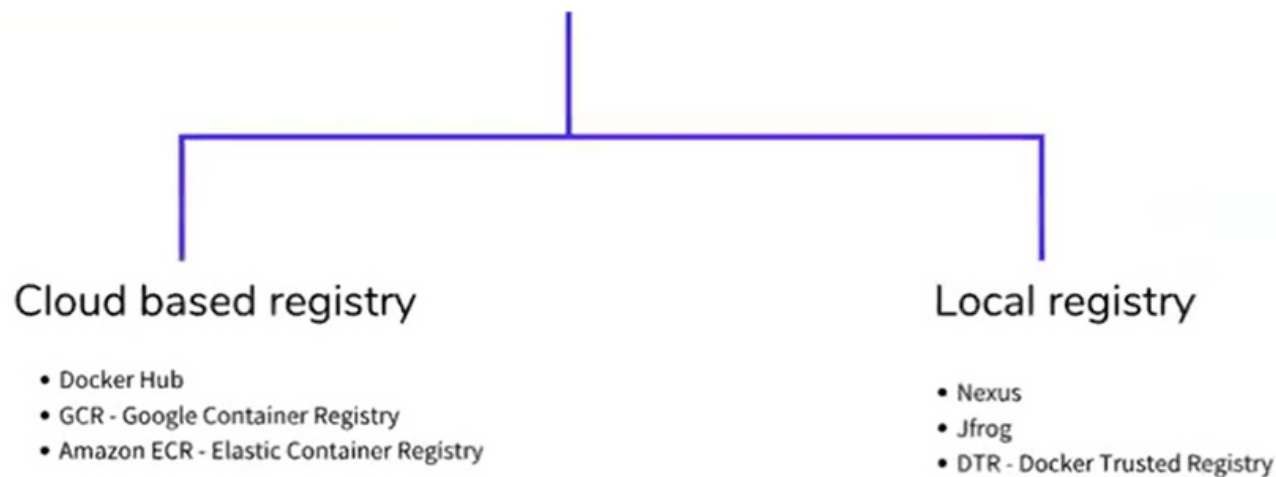
remove the volume: `docker volume rm volume_name`

remove all unused volumes: `docker volume prune`

- `docker run -itd --name swiggy --mount source=swiggy,destination=/usr/share/nginx/html -p 8081:80 nginx`
- `docker run -itd --name cont1 -v /usr/share/nginx/html -p 8082:80 nginx`

DOCKER HUB/ DOCKER REGISTRY:

It is used to store the images. Docker hub is the default registry



Launch an EC2 instance, install the docker and start the docker.
Install git and clone the code from git hub to our server.
Go to the source code and write the docker fi le.

- Build the docker fi le : **docker build -t image_name .(image1)**
- Now check the list of images: **docker images**
- Create a repository in docker hub with any name : **chary-repo**
- Now give a tag name to the image :

docker tag image1 username/created_repo:image_name

ex: (docker tag image1 charytarak/baby:naimage)

after that login to the dockerhub:



```
[root@ip-172-31-85-187 html]# docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have
Username: charytarak
Password:
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

Now push the tagged image to the docker hub :




docker push username/repo:image_name
ex: (docker push charytarak/baby:naimage)

Now we can see that the image is in the docker hub.

Tags				
This repository contains 1 tag(s).				
Tag	OS	Type	Pulled	Pushed
 naimage		Image	---	24 minutes ago
See all				


Using Jenkins we are automating the creating the docker image and pushing to docker hub:


- Set up jenkins in the server .
- create a pipeline job in the jenkins with the descriptive pipeline:


S	W	Name ↓	Last Success	Last Failure	Last Duration
		dockermama	6 min 19 sec #4	7 min 58 sec #3	14 sec 

Icon: S M L

Icon legend

 Atom feed for all

 Atom feed for failures

 Atom feed for just latest builds

Stage1 : Getting the code from git hub.

Stage2: Build the code:

We have to give full permissions to the docker.sock file .

To give full permissions to the docker.sock file :

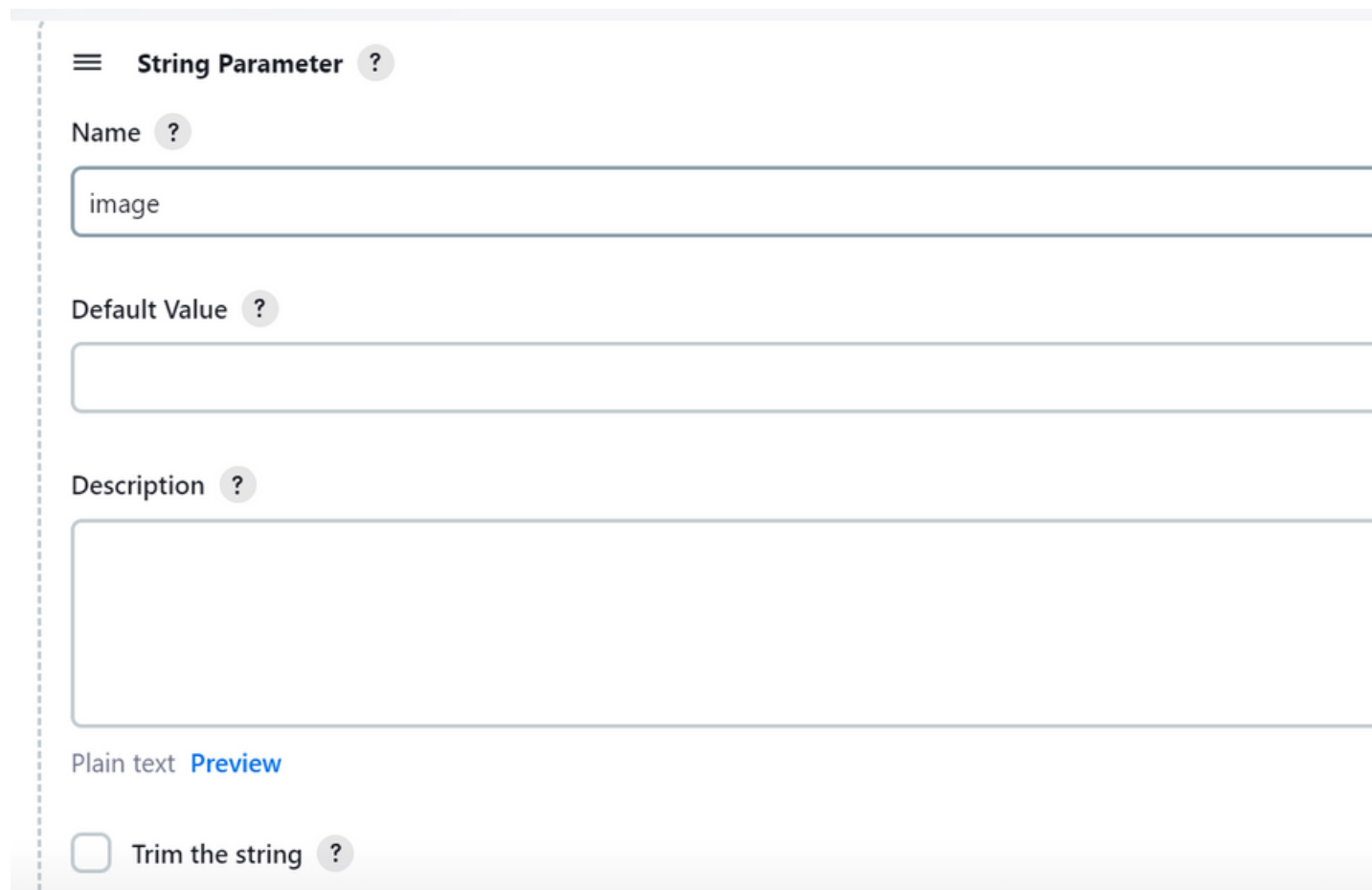
```
jenkins@jenkins-15261-200-20-11703-newshtml-31-86-110.html:~$ # chmod 777 ///var/run/docker.sock
```

Stage3: Deploy in a container:

Script ?

```
1 pipeline {
2   agent any
3
4   stages {
5     stage('Hello') {
6       steps {
7         git 'https://github.com/charytarak/staticsite-docker.git'
8       }
9     }
10    stage('build') {
11      steps {
12        sh 'docker build -t $image:$tag html/'
13      }
14    }
15    stage('deploy') {
16      steps {
17        sh 'docker run -itd --name $cname -p $hport:80 $image:$tag '
18      }
19    }
20  }
21 }
22
```


We cannot configure the port number, container name every time. So we go for String parameters to give input at the build time. Now add String parameters for the 4 strings : image , tag , cname, hport Do same like this.



The image shows a configuration form for a 'String Parameter'. At the top, there is a title 'String Parameter' with a help icon. Below the title, there are three input fields: 'Name' (containing 'image'), 'Default Value' (empty), and 'Description' (empty). At the bottom, there is a 'Trim the string' checkbox, which is currently unchecked. The form is styled with a light gray background and rounded corners.

String Parameter ?

Name ?

image

Default Value ?

Description ?

Plain text [Preview](#)

☐ Trim the string ?

Do same for other 3 strings.
Now save and build .

Pipeline Docker-pipeline

This build requires parameters:

image

tag

cname

hport

▶ Build

Cancel

To see the list of images and containers we have created in jenkins:

Install a plugin : docker

In the manage jenkins we have the docker

Uncategorized



Docker

Plugin for launching build Agents as Docker
containers

Integrate the docker with jenkins:

Go to server → Go to the path `vim /lib/systemd/system/docker.service`

In the docker.service file: go to the 19th line

erase and modify the data by:

```
# for containers run by docker  
ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:4243 -H unix:///var/run/docker.sock
```

Daemon reload: **systemctl daemon-reload**

Restart the docker : **systemctl restart docker**

After restarting the docker , all the containers will be in exited state.

So start all the containers: **docker start container_names**

Now configure in jenkins :

Manage jenkins → clouds → New cloud → Give any name → check the docker → create.

In the docker cloud details

The screenshot shows the 'Cloud docket Configuration' page in Jenkins. The 'Name' field is set to 'docket'. The 'Docker Cloud details' section is expanded, showing the 'Docker Host URI' set to 'tcp://0.0.0.0:4243' and 'Server credentials' set to '- none -'. There are 'Save' and 'Apply' buttons at the bottom.

Cloud docket Configuration

Name ?
docket

Docker Cloud details ^ Edited

▼

Docker Host URI ?
tcp://0.0.0.0:4243

Server credentials
- none -

Save Apply

Test the connection and save.

Now in **Manage Jenkins** → **Docker** → we can see all the images, running containers, we can stop the containers

Docker Server

Running Containers

Container Id	Image	Command	Created	Status	Ports	
f36a88ab41aa3da5ee6414297551d77b1bcc59508725e204207437c5b00e61bd	chaey:88	/docker-entrypoint.sh nginx -g 'daemon off;'	Tue Dec 05 01:51:58 UTC 2023	Up About a minute	ContainerPort(ip=0.0.0.0, privatePort=80, publicPort=8090, type=tcp) ContainerPort(ip=::, privatePort=80, publicPort=8090, type=tcp)	<div>stop</div>

task1:Using Jenkins to push docker image into docker hub with declarative script:

```
1 pipeline {
2   agent any
3
4   stages {
5     stage('Hello') {
6       steps {
7         git "https://github.com/charytarak/staticsite-docker.git"
8       }
9     }
10    stage('build') {
11      steps {
12        sh 'docker build -t image1 html/'
13      }
14    }
15    stage('registry') {
16      steps {
17        script {
18          withDockerRegistry(credentialsId: 'dockerhub') {
19            sh 'docker tag image1 charytarak/lcu:chary-image'
20            sh 'docker push charytarak/lcu:chary-image'
21          }
22        }
23      }
24    }
25    stage('deploy') {
26      steps {
27        sh 'docker run -itd --name mygame -p 8082:80 charytarak/lcu:chary-image'
28      }
29    }
30  }
31 }
```


we need one plugin for pushing the docker image into docker hub

Docker Pipeline 572.v950f58993843


Build and use Docker containers from pipelines.
[Report an issue with this plugin](#)


✓

after executing the pipeline , check the dockerhub:

 **charytarak / lcu**



Description

This repository does not have a description 

 Last pushed: a few seconds ago

Tags

This repository contains 1 tag(s).

Tag	OS	Type	Pulled	
 chary-image		Image	---	2 mi

[See all](#)

DOCKER NETWORK:

Docker Network is used to make a communication between the multiple containers that are running on same (or) different docker hosts

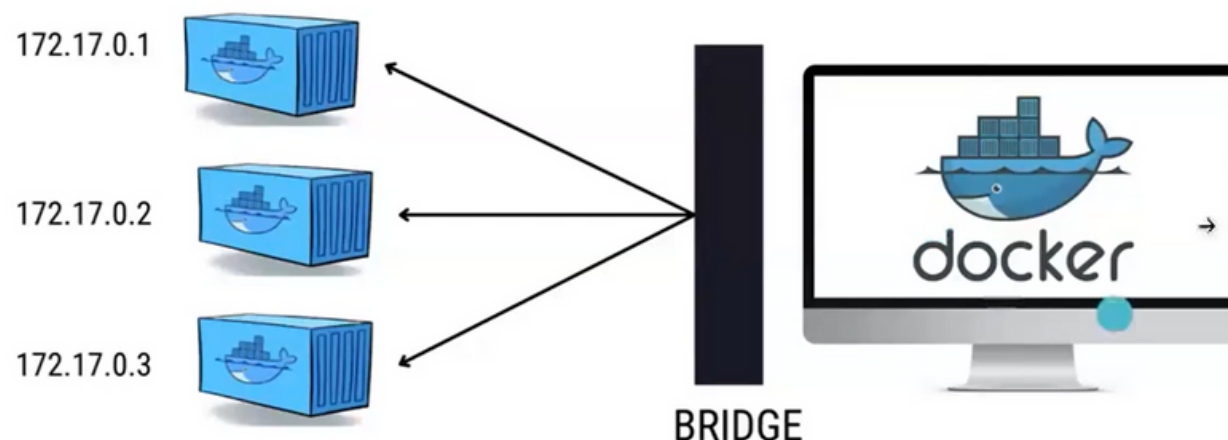
why network?

Let's assume we are having 2 containers like APP and DB container. This App container has to communicate with DB container. So, the developer will write a code to connect the application to the DB container.

So, here the IP address of a container is not permanent. If a container is removed due to hardware failure, a new container will be created with a new IP, which can cause connection issues

To resolve this issue, we are creating our own network. i.e. we are using docker networks to create our custom/own network

DOCKER NETWORK:



Now, Create a container and do inspect. So in inspect you can see the full networks data. i.e. you can see IP address and everything.

creating a container: docker run -it --name con1 ubuntu

inspect the container: docker inspect con1

- **See the all networks → docker network ls**

```
[root@ip-172-31-37-40 ~]# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
c26d25e7f0d1        bridge             bridge              local
15cfcfefdf22        host               host                local
96d9db69dc35        none               null                local
```

Each container will contain multiple networks. So, we have different types of docker networks.

1. **BRIDGE NETWORK:**

- **It is a default network that container will communicate with each other within the same host.**
- Create one container, and inspect the container

```
"Networks": {
  "bridge": {
    "IPAMConfig": null,
    "Links": null,
    "Aliases": null,
    "NetworkID": "75a3a7f0dbe35acc48b71af97616ec45cbad6fb77cadf0bdc8b2f77d3b2055a1",
    "EndpointID": "2579bc8c6986c9888aa8a808b398477c12879f379d71ef91bb2407cd1619df0d",
    "Gateway": "172.17.0.1",
    "IPAddress": "172.17.0.4",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "02:42:ac:11:00:04",
    "DriverOpts": null
  }
}
```

2.HOST NETWORK:

When you need, your container IP and EC2 instance IP same than we have to use host network.

3.none network:

When you don't want the containers to get exposed to the world, we use none network.

It will not provide any network to our container.

```
"Networks": {
  "none": {
    "IPAMConfig": null,
    "Links": null,
    "Aliases": null,
    "NetworkID": "dd72366ca9bd79cabbd5a239cf938c48ce73941d7b18d9caf32132c588fbc35",
    "EndpointID": "6fd33dbb3d0f07fdf4b6bb92581d108ff9558ab474784c4ca46c1540648febe0",
    "Gateway": "",
    "IPAddress": "",
    "IPPrefixLen": 0,
    "IPv6Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "",
    "DriverOpts": null
  }
}
```

4.OVERLAY NETWORK:

- If you want to establish the connection between the different containers which are present in different servers.

If we have multiple networks to our container means communication will increased

So, these are the Docker networks. first 3 networks are default. Normally, we're using bridge network

- **Create Custom Network → docker network create chary**
- **see the list of networks → docker network ls**

Now, we have to attach the custom network to our container. the **command is**

- **docker run -it --name cont2 --network chary ubuntu**
- **docker inspect cont2**
- **So, now you got one IP address for "chary" network**

```
"Networks": {
  "sandeep": {
    "IPAMConfig": null,
    "Links": null,
    "Aliases": [
      "b0e650ebe611"
    ],
    "NetworkID": "b5b57098311f382b37ea05246f4ee41c112457000526247dffb7188552fc65f9",
    "EndpointID": "520a341106482c33f420082a0727115bd8ace874c8ab9b9ea0311440b6316670",
    "Gateway": "172.18.0.1",
    "IPAddress": "172.18.0.2",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "02:42:ac:12:00:02",
    "DriverOpts": null
  }
}
```

Now, if you create a network inside "**Chary**"

If we use the different networks inside a same container the IP address range is

IP address will - 172.18.0.1, 172.19.0.1, 172.20.0.1, ... upto 172.256.256.256
if we use the same network in different containers the IP address range is

- 172.18.0.1, 172.18.0.2, 172.18.0.3, upto 172.256.256.256

docker network connect <n1> <n2>: Connect a container to a network(n)

docker network create <name> : Create a network

docker network disconnect chary cont4 : Disconnect a container from a network

docker inspect <name>: Display detailed information on one or more networks

docker network ls : List networks

docker network prune : Remove all unused networks

docker network rm <name>: Remove one or more networks
if that network is attached to a container we can't delete

ping: This command is mainly used for checking the network connectivity among host/server and host.

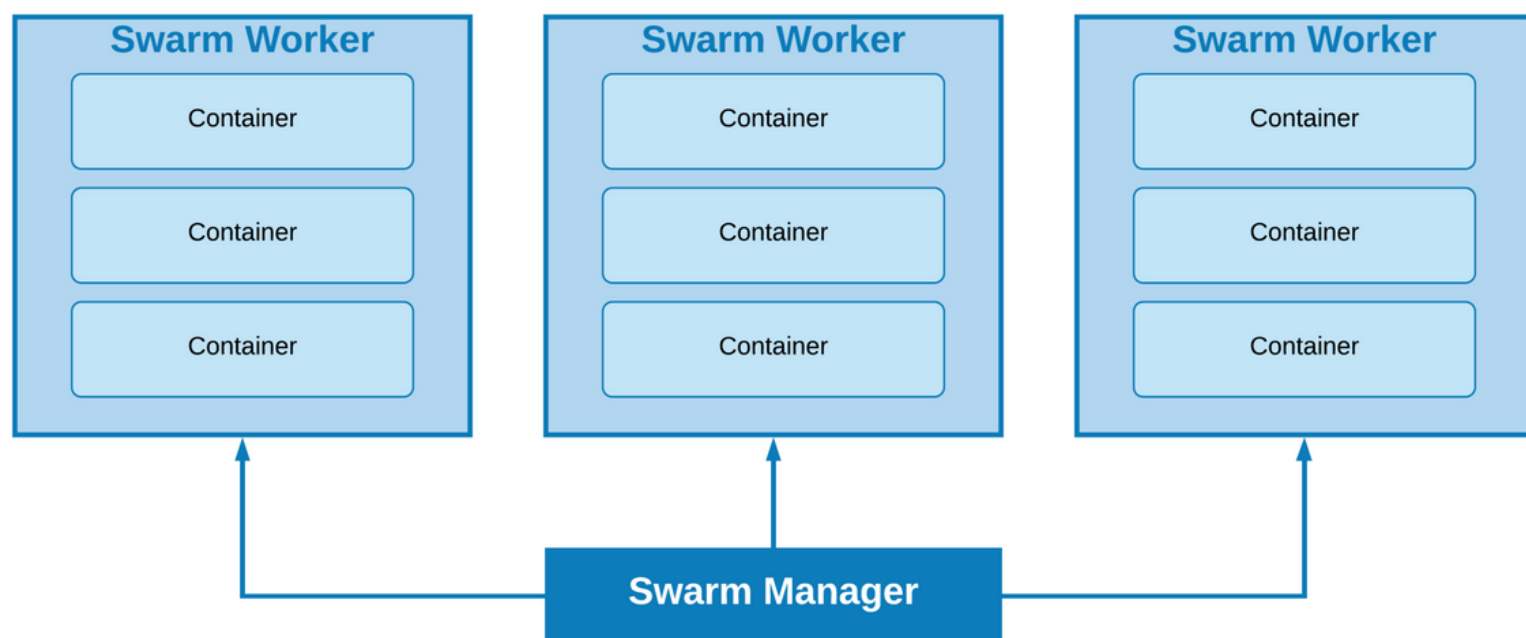
ping command: **ping <ip.adress>**

to install ping command in ubuntu: **apt install iputils-ping**

DOCKER SWARM:

Docker Swarm is an Orchestration service (or) group of service.

- It is similar to master-slave concept
- Within the docker that allows us to manage and handle multiple containers at the same time
- Docker Swarm is a group of servers that runs the docker application
- i.e. for running the docker application, in docker swarm we're creating group of servers
- We used to manage the multiple containers on multiple servers.
- This can be implemented by the "Cluster"
- The activities of the cluster are controlled by a "Swarm Manager" and machines that have joined the cluster is called "Swarm Worker"
- Here, it is the example of master and slave.



- Docker Engine helps to create Docker Swarm
i.e. if you want to implement Docker Swarm. In that system we have to install docker for 2 servers. i.e. Swarm Manager & Swarm Worker
- In the cluster we are having 2 nodes
 - a. Worker nodes
 - b. Manager nodes
- The worker nodes are connected to the manager nodes
- So, any scaling i.e. containers increase (or) updates needs to be done means first, it will go to the manager node
- From the manager node, all the things will go to the worker node
- Manager nodes are used to divide the work among the worker nodes
- Each worker node will work on an individual service for better performance
- i.e. 1 - worker node, 1 - service.

components in docker swarm:

- 1.SERVICE:It represents a part of the feature of an application
- 2.TASK:A Single part of work (or) Work that we are doing
- 3.MANAGER:This manages/distributes the work among the different nodes
- 4.WORKER:which works for a specific purpose of the service

- Take 1 normal server named as manager and inside the server install & restart the docker

- Initializing Swarm:

`docker swarm init --advertise-addr PrivateIp`

```
[root@ip-172-31-6-178 ~]# docker swarm init --advertise-addr 172.31.6.178
Swarm initialized: current node (lmleor2xcefp3dijmd9munano) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-3rnz2zfy0qav9ira8rlwp9ajirmd5eb3k2clpvnmlwj4e0q6v-c62h8q9n3xw4552u99tevv9b8 172.31.6.178:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

Here, you got all details, how to connect with worker node i.e. in master the token is generated. If we give this token in another server it will be work as a worker node

Now, take 2 normal servers named as worker-1,2 and install & restart the docker here in 2 servers

Now, copy the token from manager server and paste in 2nd server. It will joined a swarm as a worker.

```
[root@ip-172-31-13-95 ~]# docker swarm join --token SWMTKN-1-4wxsky3fri9o0t4dzu4h0wseq456lc0yfb6wio8mzelvk2ni6h-c089yjjz6bvqr9srkbfpxwrkw 172.31.4.67:2377
This node joined a swarm as a worker.
```

See the list of nodes in manager: `docker node ls`.

```
[root@ip-172-31-4-67 ~]# docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER
5zh25v2k57uq82be9jxv2el8d *	ip-172-31-4-67.ap-south-1.compute.internal	Ready	Active	Leader
x7nmp9kjek709708m52ni9gu0	ip-172-31-13-95.ap-south-1.compute.internal	Ready	Active	

create a service/container:

docker service create --name chary --replicas 3 --publish 8081:80 httpd

here, chary → service name

replicas → duplicates i.e. if the container is stopped/deleted means automatically another container will be created with the same configuration

3 → duplicate containers, like how many containers you need, just give the number

Now, 1 container is created

See the list of Services: docker service ls

It will work in only manager, not in slaves

- Now, if you perform → `docker ps -a`
- Now, check in slave server, you will get the remaining containers
- Now, check it's working (or) not

Go to → copy publicIP:8081 in browser → it works

- same like check in slave servers, you will get it.

Note :

Here, If the worker node contains less containers. Manager will send the containers to that worker node. It balances the work load

i.e. Now take another service with 2 replicas. This time, the container will add in Worker-2

Create Docker file, and we have to run the image from the Docker file:

we need to create docker file & index.html file:

vim dockerfile:

```
FROM nginx
COPY . /usr/share/nginx/html/
~
~
```

vim index.html:

```
<h1>THIS IS MY WEB AAPPLICATION </h1>
~
~
```

after that build the image: **docker build -t image1 .**

can we implement the swarm by using image1

docker service create --name leo -p 8087:80 --replicas 5 image1

Now, check it's working (or) not

Go to → copy publicIP:8087 in browser → it works

same like check in slave servers, you will get it.

- If we stop/delete a container in worker node. Automatically another container will created

docker stop contID

docker ps -a → you can see container exited and created

Because of the Auto/self healing

- removing the manager in docker swarm: **docker swarm leave --force**

UPDATE THE IMAGE FROM SERVICE:

1.Update Dockerfile

2.Build the Dockerfile

3.docker build -t <image >

4.So, right now, present service we need to update the image

docker service update --image ImageName ServiceName

Check in browser, whether it's working/not

ROLLBACK TO PREVIOUS SERVICE:

Here, I want to go back to the previous image means. Without updating the image you can't rollback to previous image.

docker service rollback ServiceName

So, here the common query is we already update the image. How we can get the previous image

It will stores the log files. So, we can rollback to any image.

TO CREATE A SERVICE: `docker service create --name service_name --publish 8081:80 --replicas 3 image_name`

TO SEE LIST OF SERVICES : `docker service ls`

TO UPDATE IMAGE : `docker service update --image image_name servicename`

TO ROLL BACK : `docker service rollback servicename`

TO SCALE : `docker service scale service_name=count`

TO GET LOGS : `docker service logs service_name`

TO GET THE CONTAINERS OF A SERVICE : `docker service ps service_name`

TO INSPECT : `docker service inspect service_name`

TO REMOVE : `docker service rm service_name`

Scaling in Docker Swarm:

If you want to increase/decrease the replicas for containers we can use
Scaling

Here Scaling is 2 types

1.Container Scaling → using docker

2.Server Scaling → using aws, Based on the users request it will increase

In Manager Server → **docker service ls**

Now, I want to increase upto 10 replicas

docker service scale ServiceName=10

scale down to 5 replicas

docker service scale ServiceName=5

Now, check the containers in Manager & Worker servers

NODE COMMANDS:

TO INIT THE SWARM : `docker swarm init --advertise-addr private-ip`

TO SEE LIST OF NODES : `docker node ls`

TO GET MANAGER TOKEN : `docker swarm join-token manager`

TO GET WORKER TOKEN : `docker swarm join-token worker`

TO LEAVE THE NODE FROM SWARM: `docker swarm leave`

TO REMOVE THE NODE : `docker node rm node-id`

DOCKER COMPOSE:

- Docker compose is a tool used to build, run and ship the multiple containers for application
- It is used to create multiple containers in a single host/server
- It used YAML file to manage multi containers as a single service
- i.e. In docker compose file we are writing the container configurations, that should be written in YAML format and the compose file extends with ".yaml"
- The compose file provides a way to document and configure all of the applications service dependencies like - databases, queues, caches, web service API's, etc.,
- In one directory, we can write only one docker-compose file

practical:

1. Install & Restart the docker in a server.

2. install docker compose

- `sudo curl -L https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m) -o /usr/local/bin/docker-compose`
- `sudo chmod +x /usr/local/bin/docker-compose`
- `ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose`
- `docker-compose version`

3. Write the Docker compose file
vim docker-compose.yml

```
version: '3'
services:
  paytm:
    image: nginx
    ports:
      - "8081:80"
```

- Version → It specifies the version of the compose file, default 3
- Services → if you want to create a service, first mention services
- paytm → service name
- Here, service acts as a container
- Inside the service we are having containers, and we have to give the image name

4. Execute the compose file:

docker-compose up -d → here, d is for detach mode

- After performing this command, automatically containers will get created which are present in docker compose file

docker ps -a

So, here with the name of paytm, container will be created → **root.paytm-1**

5. Access the application in browser

publicIP:8081 → you will get the output in the browser

If Creating Multiple Services in Docker Compose

```
version: '3'
services:
  paytm:
    image: shaikmustafa/mygame
    ports:
      - "8082:80"

  zomto:
    image: shaikmustafa/dm
    ports:
      - "8083:80"

  swiggy:
    image: shaikmustafa/cycle
    ports:
      - "8084:80"
~
```

Here, in docker-compose we have to define 4 components for containers

- P → ports
 - I → Image
 - V → Volume
 - N → Network
- Now, we have to add these components inside a docker-compose

```
version: '3'
services:
  app:
    image: nginx
    ports:
      - "7000:80"
    volumes:
      - /home/ec2-user
    networks:
      - mynetwork

networks:
  mynetwork:
    driver: bridge
```

Now, execute the docker-compose

docker-compose up -d

perform inspect in container

docker inspect contID

Now, you will see everything

For defining volumes in docker-compose, give like this..

This is the way we can give all components inside a docker compose file..

(FAQ) Suppose, in one service, I got issue in docker compose. how to resolve ?

(Here, We're going into particular docker file and update the docker file.
Build the compose file. Old containers are running, new containers will
deploy)

DOCKER-COMPOSE COMMANDS:

1.Create the containers in docker compose:**docker-compose up -d**

2.Stop and remove the composed containers:**docker-compose down**

3.Stop the containers from the compose file:docker-compose stop

docker ps -a

start the stopped containers → **docker-compose up -d**

4.See the list of images in docker-compose file:**docker-compose images**

5.See the compose containers

docker-compose ps

docker ps -a → we see manual created containers

6. See the logs in docker compose. Inside the logs containers start & end details are present here

docker-compose logs

7. See the code configuration in compose file, not from vim editor

docker-compose config

8. Pause & UnPause in container

docker-compose pause

i.e. no updates will happen in that container. that means container will be stuck

docker-compose unpause

Used to unpause the container

- Usually, we're maximum using this "**docker-compose.yml**" file name. If we use another name we're getting this error

eg:

- **vim docker-compose.yml (this is the standard way)**
- **vim chary.yml (docker won't execute this file, it shows error)**

For that, the command is → **docker-compose -f chary.yml up -d**

This is the code we're using for another file names

DOCKER STACK:

If you want to deploy multiple services in multiple servers we're using docker stack.

- It is the combination of Docker Swarm + Docker Compose
 - Here, first we have to initiate the Swarm, otherwise stack doesn't work here
 - Here, we have to write compose files
 - Docker stack is used to create multiple services on multiple hosts
 - i.e. it will create multiple containers on multiple servers with the help of compose file
 - To use the docker stack we have initialized docker swarm, if we are not using docker swarm, docker stack will not work
 - Once we remove the stack automatically all the containers will gets deleted
 - We can share the containers from manager to worker according to the replicas
 - In docker stack, we are using overlay network

Eg: Let's assume, if we have 2 servers which is manager and worker. If we deployed a stack with 4 replicas. 2 are present in manager and 2 are present in worker

- Here, manager will divide the work based on the load on a server

practical:

Take 3 servers named as manager, worker-1 & worker-2

step1: Install & restart the docker

step2: `docker swarm init --advertise-addr privateIP`

copy the token, paste in worker-1&2

See the list of nodes → `docker node ls`

step3:

Write the docker-compose file, for that install the docker compose

- `vi docker-compose.yml`

```
version: '3'
services:
  paytm:
    image: nginx
    ports:
      - "8081:80"
  phonepe:
    image: httpd
    ports:
      - "8888:80"
```

step4: execute the file

docker stack deploy --compose-file docker-compose.yml <stackName>

docker ps -a → containers will be present in worker nodes

step5: access the file

search in web browser port number 8081

Use case Scenario for Docker Stack:

Suppose, we have paytm app. Usually, daily 1k people accessing the paytm. Due to festivals, this time 100k people use this application. Due to multiple requests, server can't handle the capacity.

So, that's why i want to run my application in multiple servers. For that thing we're using cluster

Here, if you need multiple containers, we can use replicas for this. replicas used for high availability and application performs well

COMMANDS:

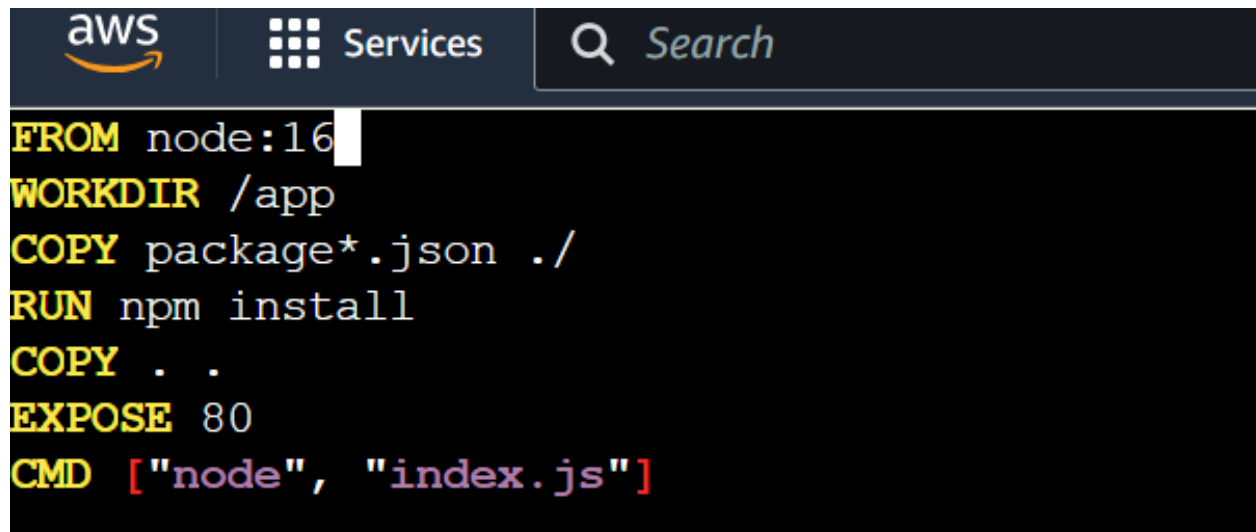
1. See the list of stacks: **docker stack ls**

2. Remove the stack: **docker stack rm stackName**

3. See the stack related services: **docker stack services stackName**

deploying the node js file by using docker :

- install docker& restart the docker
- install the git
- clone the nodejs repo from github
- write the dockerfile:

A screenshot of a Dockerfile editor in the AWS console. The interface shows the AWS logo, a 'Services' menu, and a search bar. The Dockerfile content is as follows:

```
FROM node:16
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 80
CMD ["node", "index.js"]
```

- build the image: `docker build -t <image> .`
- run the container:
`docker run -itd --name chary -p 9090:80 <imagename>`
- check the webserver :port--9090
it will work..

here we facing with volume(SIZE) issue

image2	latest	d6bf135a4f3c	18 minutes ago	909MB
--------	--------	--------------	----------------	-------

TO overcome that issue we need to use **multi stage build in docker**:
ntg but it allows you to build the docker container in multiple stages
allowing to copy artifacts from one stage to another stage.

the major advantage of this is to build light weight containers

- it is used to reduce the image size

go to docker file ,change like this by using two stages

```
FROM node:19-alpine AS base
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
CMD npm start

#second stage
FROM base AS final
RUN npm install --production
COPY . .
CMD ["node", "index.js"]
```

- after deploying this one you will get less image size:

devops	latest	e77266cc41e3	About a minute ago	177MB
image2	latest	d6bf135a4f3c	18 minutes ago	909MB