

DOCKER

APPLICATION DEPLOYMENT ARCHITECTURE:

There are two types of application deployment architectures i.e. Monolithic and Microservices architecture.

Monolithic: If an architecture has a single server and a single database to manage all services of an application then it is referred to as a “Monolithic” architecture.

Microservices : If an architecture has an individual server and a database for each specific service in an application then it is referred to as a “Microservices” architecture.

In Monolithic architecture, if any service doesn't work or if there's a bug, we've to stop the server to resolve it and due to this, all the other services would be affected.

Instead, this drawback can be overcome using Microservices architecture as we can stop a single server which has a bug and all other services can run uninterruptedly. However, in Microservices architecture also there's an issue/drawback but it's not a major one when compared to Monolithic, Microservices has multiple servers and databases.

An application has 3 layers, front-end, back-end and database and types of technologies are below.

Front-end: It's the UI of an application. We use react.js (React JavaScript)

Back-end: JavaScript, Python.

Database: MongoDB, MySQL.

If we write a code for an application (version 1.1), we build, test and deploy in our system and if we've to add another service (version 1.2) by writing a code to deploy it, we can't do that because as all the layers have version 1.1 and if there's a service which is currently in our system, we can't install another dependency to run version 1.2.

This is a drawback and in order to resolve this we've come up with virtual machines in 1970's. It's similar to another system but not in physical. It's technically referred as a computer resource that uses software which works as another computer inside a system to run programs and deploy applications. The VM has an O.S, memory, disks. It's kind of a “guest machines” run on a physical host machine. We can only install limited VM's based on the specifications of a physical host machine because it'll affect the system performance, efficiency and this drawback is overcome with the help of a “Docker”.

By using Docker, the complexity of having multiple servers and multiple databases issue is resolved. It helps to deploy multiple applications using a single server.

BEFORE INVENTION OF DOCKER:

If a developer pushes a code into GitHub, we access the code from the GitHub and build and test the code. The war file which we get, needs to be deployed and in order to deploy the war file, we need their dependencies to deploy in different environments (dev, test, prod).

AFTER THE INVENTION OF DOCKER

We access the code from the developers (GitHub), we'll pack all the dependencies in the form of an "Image". This image can be run in containers directly without installing any dependencies and the image works as an operating system for the container.

With the help of a single instance we can deploy and run multiple applications with multiple services in the form of a "containers" (similar to VM's). If an issue arises we can stop that particular container and resolve the issue. So here the Microservices architecture is used by the tool called "Docker".

Container: It's a package of software that includes everything needed to run an application like code, runtime, system tools, system libraries and settings. These are similar to as VM's but these don't need any Operating System, memory and it allocates by itself. The "Docker image" helps to run applications without any OS.

The docker images (**Image = WAR [FILE] + JAVA + TOMCAT + MYSQL [dependencies]**) can be run in a container to deploy and run the application.

Containerization: The process of packing all the required dependencies along with an application (source code) to run in a container is called as Containerization.

ABOUT DOCKER:

- Docker is an open source centralized platform which uses containers on host operating systems to create, deploy and run the applications.
- It was released in March 2013 and written in Go language.
- Docker can be installed in any OS (platform independent) but the docker engine runs on Linux distribution.
- Instead of creating a VM we can create containers to run applications on the host system.

COMPONENTS OF DOCKER: Docker client, Docker Host and Registry (Docker Hub).

Docker Client: This is where we write the commands.

Docker Host: If a system has a docker installed in it and it's called as a Docker Host. It consists of containers, images, volumes and networks.

Registry: This is where we've all images, we can store and share. It's also called as Docker Hub.

Points to be remembered:

1. You can't use the docker directly, we need to start/restart.
2. You can't enter into the container directly, it needs to start first.
3. You need a base image to create a container.
4. A container gets created by default, if an image is created.

DOCKER SETUP:

1) First launch an ec2 instance and install the docker directly as it doesn't need any dependency.

```
~\      #####
NN \    #####\
NN  \    \###|
NN   \    \#/
NN    \    V~'  ->
NNNN   \    /
NN  .-  \    /
      \  /
      /m/'

Amazon Linux 2

AL2 End of Life is 2025-06-30.

A newer version of Amazon Linux is available!

Amazon Linux 2023, GA and supported until 2028-03-15.
https://aws.amazon.com/linux/amazon-linux-2023/

[ec2-user@ip-172-31-84-187 ~]$ sudo su -
[root@ip-172-31-84-187 ~]# yum install docker -y
Loaded plugins: extras_suggestions, langpacks, priorities, update-motd
amzn2-core
Resolving Dependencies
```

```
Installed:
  docker.x86_64 0:25.0.3-1.amzn2.0.1

Dependency Installed:
  containerd.x86_64 0:1.7.11-1.amzn2.0.1      libcgrouper.x86_64 0:0.41-21.amzn2

Complete!
```

2) Ensure once the docker is installed it needs to be started first and then the docker CLI will be enabled for the use.

```
[root@ip-172-31-84-187 ~]# systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; disabled; vendor preset: disabled)
   Active: active (running) since Thu 2024-07-04 10:43:43 UTC; 27min ago
     Docs: https://docs.docker.com
   Process: 3550 ExecStartPre=/usr/libexec/docker/docker-setup-runtimes.sh (code=exited, status=0/SUCCESS)
   Process: 3549 ExecStartPre=/bin/mkdir -p /run/docker (code=exited, status=0/SUCCESS)
  Main PID: 3553 (dockerd)
    Tasks: 10
   Memory: 379.9M
   CGroup: /system.slice/docker.service
           └─3553 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock --default-ulimit nofile=32768:65536
```

DOCKER COMMANDS:

Docker default path: **/var/lib/docker**

Container default path: **/**

- 1) To install docker : **yum install docker -y**
- 2) To start docker: **systemctl start docker**
- 3) To check the status of docker: **systemctl status docker**
- 4) To check all the info about the docker: **docker info**

IMAGE COMMANDS:

- 5) To create an image: **docker pull image name (ubuntu)** (We bring the image from docker hub using pull)
- 6) To get an image of web server: **docker pull web server name (nginx, apache, httpd)**
- 7) To check list of images: **docker images**

```
[root@ip-172-31-84-187 ~]# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
image1	latest	91e7ac150752	2 hours ago	146MB
nginx	latest	fffffc90d343	13 days ago	188MB
ubuntu	latest	35a88802559d	3 weeks ago	78.1MB
shaikmustafa/dm	latest	adc857dee30b	13 months ago	146MB

- 8) To get full details of an image: **docker image inspect image_name**
- 9) To remove an image: **docker rmi image_name**
- 10) To remove all images: **docker rmi \$(docker images)** Whenever you remove an image ensure that the image is not attached to any container.
- 11) To remove image that has <none> as repo and <none> tag (similar to unused images) : **docker image prune**

CONTAINER COMMANDS:

- 12) To create a container: **docker run -it --name my_container_name image_name**

```
[root@ip-172-31-84-187 ~]# docker run -it --name cont-4 ubuntu
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
9c704ecd0c69: Pull complete
Digest: sha256:2e863c44b718727c860746568e1d54afd13b2fa71b160f5cd9058fc436217b30
Status: Downloaded newer image for ubuntu:latest
```

Here **it** - - > i and t are flags. Interactive terminal. You're telling the docker to create an interactive shell to run commands or execute programs in the container.

PuTTY and MobaXterm are used to connect to remote servers over SSH or other protocols. They provide a terminal interface for interacting with the command line of a remote server.

Docker's interactive terminal (it), on the other hand, is used to interact with a local container, not a remote server.

13) To create a container without exiting from the container: **Hold ctrl and press pq and you can create a new one.**

(or)

To exit from the container without stopping it: **Hold ctrl and press pq**

14) To go inside the container: **docker attach (container name or container ID)**

15) To exit from the container: **exit**

16) To start the container: **docker start container_name or container_ID**

17) To start all the containers: **docker start \$(docker ps -a)**

18) To stop the container: **docker stop container_name or container_ID**

19) To stop only a specific containers: **docker stop container-1 container-2 and so on (name or ID)**

20) To stop all containers (logically only running ones, which doesn't include

exited) : **docker stop \$(docker ps)**

21) To stop the container instantly: **docker kill container_name or container_ID**

stop and **kill** both stops the container, i.e. both helps to move the container from up state to exit state but the kill works quickly to stop the container than stop command.

docker client sends the signal to daemon to stop the container quickly and the signal is SIGKILL.

for stop it sends the signal as SIGTERM.

22) To check list of all containers: **docker ps -a (all) or docker container ls -a**

```
[root@ip-172-31-84-187 ~]# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
2395ed6eeda4	nginx	"/docker-entrypoint..."	2 hours ago	Exited (0) 4 seconds ago		cont-1
803e613a235d	ubuntu	"/bin/bash"	2 hours ago	Up 2 hours		cont-4
e7459d976a92	image1	"/docker-entrypoint..."	2 hours ago	Up 2 hours	80/tcp	cont-3
5e78c2569d70	shaikmustafa/dm	"/docker-entrypoint..."	2 hours ago	Up 2 hours	80/tcp	cont-2

23) To check list of running containers: **docker ps (running) or docker container ls**

```
[root@ip-172-31-84-187 ~]# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
2395ed6eeda4	nginx	"/docker-entrypoint..."	2 hours ago	Up 2 hours	80/tcp	cont-1
803e613a235d	ubuntu	"/bin/bash"	2 hours ago	Up 2 hours		cont-4
e7459d976a92	image1	"/docker-entrypoint..."	2 hours ago	Up 2 hours	80/tcp	cont-3
5e78c2569d70	shaikmustafa/dm	"/docker-entrypoint..."	2 hours ago	Up 2 hours	80/tcp	cont-2

24) To check list of exited/stopped containers: **docker ps -f "status=exited" (-f filter)**

```
[root@ip-172-31-84-187 ~]# docker ps -f "status=exited"
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
2395ed6eeda4	nginx	"/docker-entrypoint..."	2 hours ago	Exited (0) 4 minutes ago		cont-1
5e78c2569d70	shaikmustafa/dm	"/docker-entrypoint..."	2 hours ago	Exited (0) 10 seconds ago		cont-2

25) To check latest created container: **docker container ls --latest**

26) To check latest created top two containers: **docker ps -n 2 or docker container ls -a -n 2**

27) To delete the container: **docker rm container_name or ID (rm = remove)**

28) To remove exited containers/remove unused containers/stopped containers: **docker container rm \$(docker container ls -a or ps -a) or docker container prune**

29) To remove a running container: **docker rm -f container name**

30) To remove all running containers: **docker rm -f \$(docker ps)** this leaves the exited ones

31) To check the size of the container: **docker container ls -s**

32) To get all details of a container: **docker inspect container_name**

33) To rename a container: **docker rename container-old-name new-name.**

IMPORTANCE OF -d:

"If we don't use -d" for web server images while creating a container we get the below kind of errors, a container would be created but it will be in an exited state and won't be in up.

(i)

```
[root@ip-172-31-84-187 ~]# docker run -it --name cont-5 httpd
Unable to find image 'httpd:latest' locally
latest: Pulling from library/httpd
f11c1adaa26e: Already exists
3bce494db9a: Pull complete
4f4fb700ef54: Pull complete
0ec6a44b37fe: Pull complete
e4822864e326: Pull complete
65dfcad56f2: Pull complete
Digest: sha256:bc365dbc84877ed9d4c78a6816ad51786e9b961144254282b4d2d66307e7ac6a
Status: Downloaded newer image for httpd:latest
AH00558: httpd: could not reliably determine the server's fully qualified domain name, using 172.17.0.6. Set the 'ServerName' directive globally to suppress this messa
ge
AH00558: httpd: could not reliably determine the server's fully qualified domain name, using 172.17.0.6. Set the 'ServerName' directive globally to suppress this messa
ge
[Thu Jul 04 10:52:09.840222 2024] [mpm_event:notice] [pid 1:tid 1] AH00489: Apache/2.4.61 (Unix) configured -- resuming normal operations
[Thu Jul 04 10:52:09.840783 2024] [core:notice] [pid 1:tid 1] AH00094: Command line: 'httpd -D FOREGROUND'
^C[Thu Jul 04 10:52:34.867129 2024] [mpm_event:notice] [pid 1:tid 1] AH00491: caught SIGTERM, shutting down
```

```
[root@ip-172-31-84-187 ~]# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
3f3113b50562	httpd	"httpd-foreground"	31 seconds ago	Exited (0) 6 seconds ago		cont-5
6aebc525399f	ubuntu	"/bin/bash"	About a minute ago	Up About a minute		cont-4
e7459d976a92	image1	"/docker-entrypoint."	4 minutes ago	Up 3 minutes	80/tcp	cont-3
5e78c2569d70	shaikmustafa/dm	"/docker-entrypoint."	5 minutes ago	Up 5 minutes	80/tcp	cont-2
571e2e5ec26c	nginx	"/docker-entrypoint."	8 minutes ago	Up 7 minutes	80/tcp	cont-1

ii) Here the CLI got stuck till 10:44:08 start worker process 29 and as soon as I press ctrl C (got like this ^C on CLI), then I got to see the container is created but it is in exited state.

```
[root@ip-172-31-84-187 ~]# docker run -it --name cont-1 nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
f11c1adaa26e: Pull complete
c6b156574604: Pull complete
ea5d7144c337: Pull complete
1bbcb9df2c93: Pull complete
537a6cfe3404: Pull complete
767bff2cc03e: Pull complete
adc73cb74f25: Pull complete
Digest: sha256:67682bda769fae1ccf5183192b8daf37b64cae99c6c3302650f6f8bf5f0f95df
Status: Downloaded newer image for nginx:latest
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2024/07/04 10:44:08 [notice] 1#1: using the "epoll" event method
2024/07/04 10:44:08 [notice] 1#1: nginx/1.27.0
2024/07/04 10:44:08 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
2024/07/04 10:44:08 [notice] 1#1: OS: Linux 5.10.219-208.866.amzn2.x86_64
2024/07/04 10:44:08 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 32768:65536
2024/07/04 10:44:08 [notice] 1#1: start worker processes
2024/07/04 10:44:08 [notice] 1#1: start worker process 29
^C2024/07/04 10:44:56 [notice] 1#1: signal 2 (SIGINT) received, exiting
2024/07/04 10:44:56 [notice] 29#29: signal 2 (SIGINT) received, exiting
2024/07/04 10:44:56 [notice] 29#29: exiting
2024/07/04 10:44:56 [notice] 29#29: exit
2024/07/04 10:44:56 [notice] 1#1: signal 17 (SIGCHLD) received from 29
2024/07/04 10:44:56 [notice] 1#1: worker process 29 exited with code 0
2024/07/04 10:44:56 [notice] 1#1: exit
```

It is in exited state, after starting, it then went to up state.


```
[root@ip-172-31-84-187 ~]# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
571e2e5ec26c	nginx	"/docker-entrypoint..."	58 seconds ago	Exited (0) 8 seconds ago		cont-1

```
[root@ip-172-31-84-187 ~]# docker start cont-1
```

```
[root@ip-172-31-84-187 ~]# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
571e2e5ec26c	nginx	"/docker-entrypoint..."	About a minute ago	Up 3 seconds	80/tcp	cont-1

USING -d

1) To create a container using image: **docker run -it -d --name container name nginx**

The importance of -d:

-d - - > detach mode, which means it starts the container and run in the background. So you can use the console to perform other commands instead of coming out of the container (exit).

When you run a container **with the -d flag**, it runs in the background as a daemon. Even if you close your terminal or shell session, the container process itself will continue running. It's decoupled from your terminal session and will persist until you stop it explicitly using a command like `docker stop`.

When you run a container **without the -d flag**, it runs in the foreground. In this case, the container's output is visible in your terminal, and if you close the terminal or interrupt the process (e.g., by pressing Ctrl+C), the container will be stopped. The container's lifecycle is tied to your terminal session.

For application server images, web server images, database server images we need to pass -d (detach) it works in foreground and background.

When you give -d, the container gets created but we don't go inside.

2) To access the web server inside the container: **docker inspect container name**

inspect - - > In order to know the source code of the resource or full info of the container.

3) To access the application inside the container: **curl ip_address of container**

curl > check URL we can access the application internally (you get the default page) and you can't access the application with it's IP directly by searching it in Google through the internet.

We can't assign a host port to the already running or existing container.

docker run -it -d --name container name -p host_port:container_port image_name

docker run -it -d --name cont-2 -p 9090:80 nginx

[or]

docker run - --name cont-2 -p 9090:80 nginx

-p - - > We're publishing a port number to a container and the limit is till 65,536.

9090 - - > It's the host port (the port number of the application running inside the container). Host port is essential to access the app running inside the container. The Container port depends on the image that we take.

80 - - > It's a container port, the default port of the image (so NGINX = 80).

If you don't have an image within the system and still trying to create a container, it is still possible as first it will check within the system and then goes to central, and pulls automatically from the central/docker registry/docker hub.

4) To perform any commands inside a container without going inside the container : **docker exec**

5) To create a file/folder inside a container: **docker exec container_name touch filename**

6) To check the list of files in a container : **docker exec container_name ls**

7) To go inside the container: **docker exec -it container_name bash** or **/bin/bash**

Image layers: It defines the number of actions performed by an image through commands that we mention in the Dockerfile and if there are 3 layers that means there were 3 actions/tasks performed. The actions performed with the help of Dockerfile components is called as a layer and the ones used for documentation purpose doesn't consider as an image layer.

Ex: The below image has only 4 layers.

FROM ubuntu

MAINTAINER: name mahi

RUN apt update -y

RUN apt install apache2 -y

EXPOSE 99

COPY index.htm /var/www/html

To move files, code or anything from a container and if same needs to be in other container, we need to create docker image from the container and once the docker image is run the same data will be moved to another container.

To CREATE AN IMAGE FROM THE CONTAINER:

Steps:

1. First create a container: **docker run -it --name container name image name.**
2. Create an image from the container: **docker commit container_name image name.. docker commit Paytm mahi-image.**

3. To create a container with the created image to get all files from the container Paytm:
docker run -it --name container name created image name docker run -it --name ravi mahi-image.

DOCKER FILE: It's a text file which has a set of instructions. This helps to automate the process of image creation. In "Dockerfile", D should be in capital when you create a docker file and the start components should be in Capital letter to look formal but it can also be in lower case as well. There are a few components in a docker file which are discussed below.

Create a file - - > Build it - - > and - - > Run the image to create a container.

CREATION OF A DOCKER FILE:

We need to create a file using vim like **vim Dockerfile** and here you can use the components to execute the instructions.

IMP: After the file is created, we need to build it and after build we'll get an image and if the image is run then a container will be created.

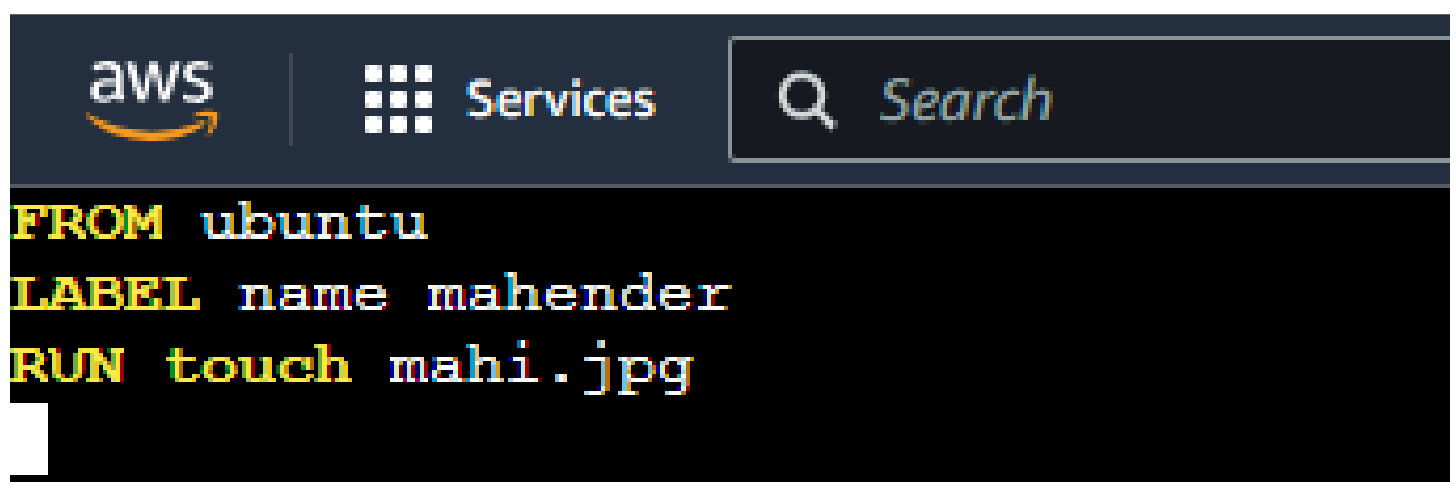
i) First we need to create a Dockerfile using **vim Dockerfile**.

ii) The file will be opened in vim. Go to insert mode "**i**", pass the instructions like below and '**esc**' to exit from insert mode to go to command mode and then use **:wq** (save and quit).

FROM ubuntu

LABEL name mahender

RUN touch mahi.jpg



iii) After passing the instructions, we need to build the image with command syntax "**docker build -t your own image name for the image that you're building**".

COMMAND: docker build -t image-1

```
[root@ip-172-31-34-147 ~]# docker build -t image1 .
Sending build context to Docker daemon  9.728kB
Step 1/3 : FROM ubuntu
latest: Pulling from library/ubuntu
aece8493d397: Pull complete
Digest: sha256:2b7412e6465c3c7fc5bb21d3e6f1917c167358449fecac8176c6e496e5c1f05f
Status: Downloaded newer image for ubuntu:latest
--> e4c58958181a
Step 2/3 : LABEL name mahender
--> Running in f75b327e9de8
Removing intermediate container f75b327e9de8
--> c48c49517811
Step 3/3 : RUN touch mahi.jpg
--> Running in af04df73ac27
Removing intermediate container af04df73ac27
--> 4f5ffdb1d351
Successfully built 4f5ffdb1d351
Successfully tagged image1:latest
```

iv) Later run the image to get all the data to a new container with command syntax **docker run -it --name container_name created_image_name**

COMMAND: **docker run -it --name cont-1 image-1**

v) And then give ll inside the container and you'll get the mahi.jpg into the newly created container. This is how the Dockerfile helps in creating a container.

DIFFERENT COMPONENTS OF A DOCKER FILE:

- | | |
|----------|---------------|
| 1) FROM | 7) ENTRYPOINT |
| 2) LABEL | 8) WORKDIR |
| 3) COPY | 9) ENV |
| 4) ADD | 10) ARG |
| 5) RUN | 11) EXPOSE |
| 6) CMD | |

1) **FROM:** It defines the image in a docker file.

ex: FROM ubuntu, FROM nginx, FROM httpd.

2) **LABEL:** It defines the author of the docker file. Instead of Label you can also use maintainer.

ex: name: mahi, e-mail: mahi@gmail.com

3) COPY: It copies the files from our local system (EC2) to a container.

ex: copy source_file (system) destination_file (container).

Here we can download the file and then use COPY command to copy the file to a container and to download the file directly and copy to a container we use ADD.

4) ADD: It copies the files from our local system (EC2) to a container and also it can download the files from internet and send it to a container.

ex: ADD url destination.

5) RUN: It is used to perform a command while we build the image. Multiple RUN's work in the Dockerfile.

ex: RUN touch aws, RUN yum install maven -y

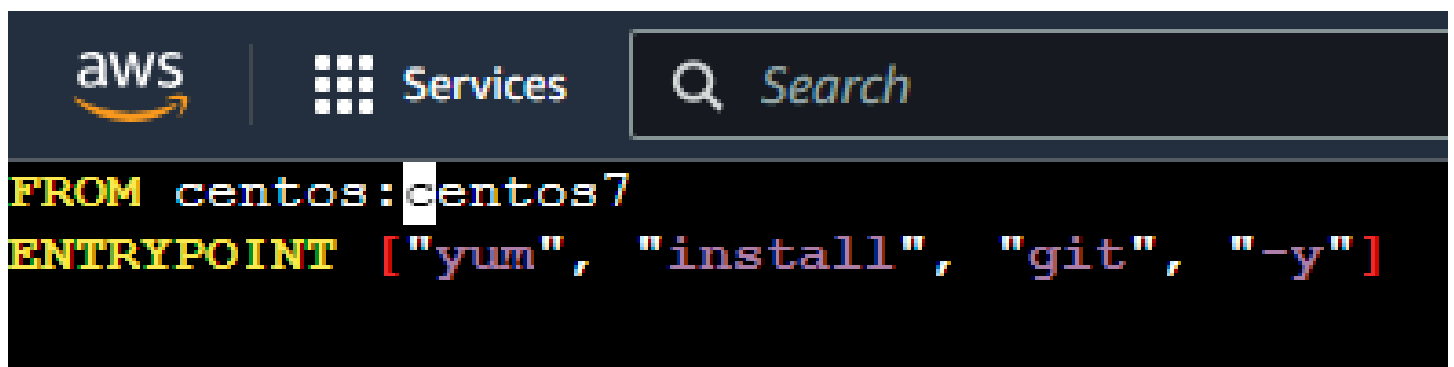
6) CMD: It is used to perform a command while we run a image/creating a container (running image is nothing but creating a container so it can also be defined as while we creating a container). In CMD we can't give values while running an image but in entrypoint it works.

Multiple CMD's doesn't work, only the last one works.

7) ENTRYPOINT: It is also used to perform a command while we run the image (same as CMD) but it has high priority than CMD and it also 'overwrites the values of CMD'.

Multiple ENTRYPOINT's doesn't work in the Dockerfile, only the last works (like if you give git, httpd, tree. only tree gets installed).

Fig-1: VIM FILE

A screenshot of a text editor window showing a Dockerfile. The top bar of the editor has the AWS logo, a 'Services' menu, and a search bar. The code in the editor is as follows:

```
FROM centos:centos7
ENTRYPOINT ["yum", "install", "git", "-y"]
```

This is the picture of building the image, however GIT doesn't gets installed as it is ENTRYPOINT. It only works during CMD.

Fig:2

```
[root@ip-172-31-34-147 ~]# docker build -t image2 .
Sending build context to Docker daemon 30.21kB
Step 1/2 : FROM centos:centos7
--> eeb6ee3f44bd
Step 2/2 : ENTRYPOINT ["yum", "install", "git", "-y"]
--> Using cache
--> 2743de1ed73a
Successfully built 2743de1ed73a
Successfully tagged image2:latest
```

In the file we've given entrypoint and we've mentioned only GIT to install. However, while running an image after building it, we can still pass the values like **httpd, tree** and even these things can be installed as well (Shown in Fig.4). To run the image we're not creating a container just for practical purpose, it's correctly can be shown during deployment.

Fig:3

```
[root@ip-172-31-34-147 ~]# docker run image2 httpd tree
Loaded plugins: fastestmirror, ovl
Determining fastest mirrors
* base: download.cf.centos.org
* extras: download.cf.centos.org
* updates: download.cf.centos.org
Resolving Dependencies
```

Fig:4

```
Verifying : perl-Text-ParseWords-3.29-4.el7.noarch 44/46
Verifying : 4:perl-libs-5.16.3-299.el7_9.x86_64 45/46
Verifying : less-458-9.el7.x86_64 46/46

Installed:
git.x86_64 0:1.8.3.1-25.el7_9 httpd.x86_64 0:2.4.6-99.el7.centos.1
tree.x86_64 0:1.6.0-10.el7

Dependency Installed:
apr.x86_64 0:1.4.8-7.el7
apr-util.x86_64 0:1.5.2-6.el7_9.1
centos-logos.noarch 0:70.0.6-3.el7.centos
```

HIGH PRIORITY and OVERWRITES THE VALUES OF CMD PROOF:

If you do not mention the package name (GIT), it can be still installed during run time as we mentioned in CMD.

Also, the other way is by mentioning a condition which is CMD ['git'] . If you pass Maven during run time (while running an image), maven gets installed and if you do not pass anything by default the CMD will be used, then git will be installed. So here the PRIORITY is HIGH for ENTRYPOINT.

As Maven gets installed, then it means it overwrites the values of CMD (overwriting GIT by Maven if we mention Maven during the run time).

Refer to below images to understanding the above mentioned concept better.

VIM Dockerfile:

```
FROM centos:centos7
ENTRYPOINT ["yum", "install", "-y"]
CMD ["git"]
```

BUILD:

```
[root@ip-172-31-34-147 ~]# docker build -t image22 .
Sending build context to Docker daemon 30.21kB
Step 1/3 : FROM centos:centos7
--> eeb6ee3f44bd
Step 2/3 : ENTRYPOINT ["yum", "install", "-y"]
--> Using cache
--> 91974b509b9b
Step 3/3 : CMD ["git"]
--> Running in c749deb6733e
Removing intermediate container c749deb6733e
--> eccc281772fd
Successfully built eccc281772fd
Successfully tagged image22:latest
[root@ip-172-31-34-147 ~]#
```

i) While running IMAGE if you don't pass GIT it still gets installed as we passed through CMD in VIM:

```
[root@ip-172-31-34-147 ~]# docker run image22
Loaded plugins: fastestmirror, ovl
Determining fastest mirrors
 * base: download.cf.centos.org
 * extras: download.cf.centos.org
 * updates: download.cf.centos.org
Resolving Dependencies
--> Running transaction check
```

```
Verifying      : perl-Getopt-Long-2.40-3.el7.noarch
Verifying      : perl-Text-ParseWords-3.29-4.el7.noarch
Verifying      : 4:perl-libs-5.16.3-299.el7_9.x86_64
Verifying      : less-458-9.el7.x86_64
```

```
Installed:
  git.x86_64 0:1.8.3.1-25.el7_9
```

```
Dependency Installed:
```

ii) While running an image if you pass MAVEN it overwrites the values in CMD in VIM i.e, it overwrites GIT and Maven gets installed:

```
[root@ip-172-31-34-147 ~]# docker run image22 maven
Loaded plugins: fastestmirror, ovl
Determining fastest mirrors
 * base: download.cf.centos.org
 * extras: download.cf.centos.org
 * updates: download.cf.centos.org
Resolving Dependencies
--> Running transaction check
---> Package maven.noarch 0:3.0.5-17.el7 will be installed
```



```
Verifying      : libfontenc-1.1.3-3.el7.x86_64
Verifying      : libtiff-4.0.3-35.el7.x86_64
Verifying      : regexp-1.5-13.el7.noarch

Installed:
  maven.noarch 0:3.0.5-17.el7

Dependency Installed:
```

8) WORKDIR: It is used to create a folder (directory) and it takes directly into the folder in the / default path of the container. In general, whenever you create a container and go inside the container using docker attach command, it takes you into the container's default path / with the container ID but if you use the WORKDIR, instead of taking you to the default path it takes you to the path of the folder you mention in the Dockerfile.

ex: WORKDIR /myapp

9) ENV: It's used to pass the variables inside the containers and can access the values inside a container, we can't overwrite the values in run time.

10) ARG: It is used to pass the variables inside the containers, we can overwrite the values in run time. We can't access the values inside a container.

11) EXPOSE: It is a component which is used to listen the incoming requests on the port 80, let's say we've a web server httpd it works on port 80 and if we mention expose 80 that means the requests goes to web servers in this way useful and it doesn't publish any port it is useful only for documentation purpose.

ex: EXPOSE 80

DEPLOYMENT OF A WEB APPLICATION USING DOCKERFILE

Whenever you take a Ubuntu image or Ubuntu instance it needs to be updated first (**apt update -y**). Without updating no packages would be installed in Ubuntu machine. IN Red Hat and Amazon Linux we use 'yum' and in Ubuntu we use 'apt'. If we take Red Hat operating system httpd gets installed and if it's Ubuntu it's apache2 gets installed.

So usually we create an image by running a command docker pull image_name and it gets downloaded in our local system and then we can use it to build. If you give command or if we just give the name However, pulling it from docker hub or we run when an image is build, image is run, a container will be created and whatever the instructions you've in the Dockerfile will be executed and in the same way if you pass the

instructions to deploy the application. We need to build, run the image and the application gets deployed. The process of deployment using docker file goes as:

So we write a Dockerfile with required instructions in it, build it, run it and it creates a container where a deployment will be performed.

The reason why we use CMD in Dockerfile instead of RUN is because the container gets created after running the image, to run an image we use CMD, can also give ENTRYPOINT. We can't use RUN because RUN is used while building an image.

– 1. With Ubuntu

```
FROM ubuntu
RUN apt update -y
RUN apt install apache2 -y
COPY index.html /var/www/html/
CMD ["usr/sbin/apachectl", "-D", "FOREGROUND"]
```

FROM ubuntu - - > Ubuntu Machine or Ubuntu OS

RUN apt update -y - - > Whenever Ubuntu machine is taken it needs to be updated.

RUN apt install apache2 -y - - > Installing apache2 web server in Ubuntu.

COPY index.html /var/www/html/ - - > Copying the source code from our local system to web server.

CMD ["usr/sbin/apachectl", "-D", "FOREGROUND"] - - > Mentioning the C Group to start the webserver.

– 2. With Centos

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

– 3. With nginx

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

After writing the Dockerfile, we need to build it (**docker build -t image name .**) and we need to run the image(**docker run -itd --name cont name -p host-port number:container-port image name**) to create a container to deploy the application that we got the source code in the form of index.html with the help of COPY.

All the above screenshots demonstrates deploying of a web application, by taking an operating system (O.S) and installing a package.

Now we're using direct image without installing any package for deployment which reduces the size of the images. Even in real time, our aim is to reduce the size of an image to decrease the delay in scanning the code and all.

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

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

The default path of web server is /var/www/html.

Difference between RUN vs PULL:

If we give a RUN command it searches whether the image is present in docker host (local system) or not and if there's an image within local then a container will be created and if not, it goes to docker registry and the image gets downloaded in docker host and container gets created.

If PULL command is used it doesn't matter whether the image is available within local or not and it automatically goes to docker registry, downloads the image but it doesn't create a container.

RUN looks in local system and goes to registry to download an image for container creation but PULL only downloads the image into local system.

DOCKER VOLUMES

Docker volume is like a file or a directory where we have the data. Docker volume is mainly used to replicate the changes that have been made to data in one container to another container. It was not possible to do so while an image is created from a container and when later the image is run, the container used to get created and we used to get the data and the application is used to run in the new container as well but, we were unable to see the replication of data when changes are made.

Docker volume can be create in two ways.

a) Command

b) Dockerfile

IMPORTANT POINTS ABOUT DOCKER VOLUMES:

- A volume can be created while creating a container, and can't be created for an existing container. We can declare a directory as a volume while creating a container.

Reference: `docker run -it --name cont-1 -v /MY_VOLUME ubuntu`

- A volume can be shared from one container to another and it needs to be mentioned/declared while creating a new one so the data can be copied.
- First, we've to declare the volume and then share the volume.

Reference: `docker run -it --name cont-2 --privileged=true --volumes-from cont-1 ubuntu`

- If a container is stopped/deleted we can still access the volume.
- Volume (data) will not be included when an image is updated (**means if a container-1 has a volume and if an image is created from container-1 and if we run the created image a container-2 gets created, we get all files from cont-1 to cont-2, but we don't get the data in the volume**).
- A volume looks similar to a directory inside our container(color same as folder).
- We can remove a folder but not volume.

You can delete a directory (folder) inside a container but not volume. The data inside the volume can be shared to another container but not the data in a folder.

DOCKER VOLUME COMMANDS:

1) To create a volume: **docker volume create volume_name**

2) To create a volume along with container: **docker run -it --name container_name -v /volume_path image_name**

3) To share a volume from one container to another: **docker run -it --name new container_name --privileged=true --volumes-from container name from which you want to share the volume to new**

image_name

`docker run -it --name cont-2 --privileged=true --volumes-from cont-1 ubuntu`

`--privileged=true` - - > sharing a volume from one container to another

`--volumes-from` - - > from which container

SHARING A VOLUME OF ONE CONTAINER TO ANOTHER CONTAINER:

i) Created a container with volume.

```
[root@ip-172-31-84-187 ~]# docker volume ls
DRIVER      VOLUME NAME
[root@ip-172-31-84-187 ~]# docker run -itd --name cont1 -v /MY-VOLUME ubuntu
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
9c704ecd0c69: Already exists
Digest: sha256:2e863c44b718727c860746568e1d54afd13b2fa71b160f5cd9058fc436217b30
Status: Downloaded newer image for ubuntu:latest
a082b15884bdaed267630506c57843d71ce1178da7f8e7c8af5034ab325e6e3d
[root@ip-172-31-84-187 ~]# docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS          NAMES
a082b15884bd   ubuntu   "/bin/bash"             5 seconds ago   Up 4 seconds           cont1
[root@ip-172-31-84-187 ~]# docker volume ls
DRIVER      VOLUME NAME
local       3fbb8a18080a82ec8d5c37dbf93b91677a52dd25fe07189e273ec52a843ba3f
```

ii) Go inside the container and you can find the created volume(MY-VOLUME) and there are no files.

```

[root@ip-172-31-84-187 ~]# docker attach cont1
root@a082b15884bd:/# ll
total 0
drwxr-xr-x 1 root root 23 Jul 6 10:50 ./
drwxr-xr-x 1 root root 23 Jul 6 10:50 ../
-rwxr-xr-x 1 root root 0 Jul 6 10:50 .dockerenv*
drwxr-xr-x 2 root root 6 Jul 6 10:50 MY-VOLUME/
lrwxrwxrwx 1 root root 7 Apr 22 13:08 bin -> usr/bin/
drwxr-xr-x 2 root root 6 Apr 22 13:08 boot/
drwxr-xr-x 5 root root 360 Jul 6 10:50 dev/
drwxr-xr-x 1 root root 66 Jul 6 10:50 etc/
drwxr-xr-x 3 root root 20 Jun 5 02:06 home/
lrwxrwxrwx 1 root root 7 Apr 22 13:08 lib -> usr/lib/
lrwxrwxrwx 1 root root 9 Apr 22 13:08 lib64 -> usr/lib64/
drwxr-xr-x 2 root root 6 Jun 5 02:02 media/
drwxr-xr-x 2 root root 6 Jun 5 02:02 mnt/
drwxr-xr-x 2 root root 6 Jun 5 02:02 opt/
dr-xr-xr-x 164 root root 0 Jul 6 10:50 proc/
drwx----- 2 root root 37 Jun 5 02:05 root/
drwxr-xr-x 4 root root 33 Jun 5 02:06 run/
lrwxrwxrwx 1 root root 8 Apr 22 13:08/sbin -> usr/sbin/
drwxr-xr-x 2 root root 6 Jun 5 02:02 srv/
dr-xr-xr-x 13 root root 0 Jul 6 10:50 sys/
drwxrwxrwt 2 root root 6 Jun 5 02:05 tmp/
drwxr-xr-x 12 root root 133 Jun 5 02:02 usr/
drwxr-xr-x 11 root root 139 Jun 5 02:05 var/
root@a082b15884bd:/# cd MY-VOLUME/
root@a082b15884bd:/MY-VOLUME# ll
total 0

```

iii) Now a few files are created and came out of the container with ctrl PQ.

```

root@a082b15884bd:/MY-VOLUME# touch files{1..3}
root@a082b15884bd:/MY-VOLUME# ll
total 0
drwxr-xr-x 2 root root 48 Jul 6 10:51 ./
drwxr-xr-x 1 root root 23 Jul 6 10:50 ../
-rw-r--r-- 1 root root 0 Jul 6 10:51 files1
-rw-r--r-- 1 root root 0 Jul 6 10:51 files2
-rw-r--r-- 1 root root 0 Jul 6 10:51 files3

```

iv) Created another container-2 with sharing the volume of cont-1.

```
[root@ip-172-31-84-187 ~]# docker run -itd --name cont2 --privileged=true --volumes-from cont1 ubuntu
6682d83306e301dfd7b7d968620bbf94a4c78dlac0676ed79ed7fc921707048a
[root@ip-172-31-84-187 ~]# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
6682d83306e3	ubuntu	"/bin/bash"	4 seconds ago	Up 3 seconds		cont2
a082b15884bd	ubuntu	"/bin/bash"	2 minutes ago	Up 2 minutes		cont1

v) Here you can find the files that were in cont-1 are reflecting in cont-2 as well. And also created 5 more files with name new.

```
[root@ip-172-31-84-187 ~]# docker attach cont2
root@6682d83306e3:/# ll
total 0
drwxr-xr-x 1 root root 23 Jul 6 10:52 ./
drwxr-xr-x 1 root root 23 Jul 6 10:52 ../
-rwxr-xr-x 1 root root 0 Jul 6 10:52 .dockerenv*
drwxr-xr-x 2 root root 48 Jul 6 10:51 MY-VOLUME/
lrwxrwxrwx 1 root root 7 Apr 22 13:08 bin -> usr/bin/
drwxr-xr-x 2 root root 6 Apr 22 13:08 boot/
drwxr-xr-x 11 root root 2720 Jul 6 10:52 dev/
drwxr-xr-x 1 root root 66 Jul 6 10:52 etc/
drwxr-xr-x 3 root root 20 Jun 5 02:06 home/
lrwxrwxrwx 1 root root 7 Apr 22 13:08 lib -> usr/lib/
lrwxrwxrwx 1 root root 9 Apr 22 13:08 lib64 -> usr/lib64/
drwxr-xr-x 2 root root 6 Jun 5 02:02 media/
drwxr-xr-x 2 root root 6 Jun 5 02:02 mnt/
drwxr-xr-x 2 root root 6 Jun 5 02:02 opt/
dr-xr-xr-x 162 root root 0 Jul 6 10:52 proc/
drwx----- 2 root root 37 Jun 5 02:05 root/
drwxr-xr-x 4 root root 33 Jun 5 02:06 run/
lrwxrwxrwx 1 root root 8 Apr 22 13:08 sbin -> usr/sbin/
drwxr-xr-x 2 root root 6 Jun 5 02:02 srv/
dr-xr-xr-x 13 root root 0 Jul 6 10:50 sys/
drwxrwxrwt 2 root root 6 Jun 5 02:05 tmp/
drwxr-xr-x 12 root root 133 Jun 5 02:02 usr/
drwxr-xr-x 11 root root 139 Jun 5 02:05 var/
root@6682d83306e3:/# cd MY-VOLUME/
root@6682d83306e3:/MY-VOLUME# ll
total 0
drwxr-xr-x 2 root root 48 Jul 6 10:51 ./
drwxr-xr-x 1 root root 23 Jul 6 10:52 ../
-rw-r--r-- 1 root root 0 Jul 6 10:51 files1
-rw-r--r-- 1 root root 0 Jul 6 10:51 files2
-rw-r--r-- 1 root root 0 Jul 6 10:51 files3
root@6682d83306e3:/MY-VOLUME# touch new{1..5}
```

vi) Now go inside the container-1 and you can find the 5 new files that are created in cont-2 will be there in cont-1 as well. That's how **"the data replication is possible with the docker volume concept"**.


```
[root@ip-172-31-84-187 ~]# docker attach cont1
root@a082b15884bd:/MY-VOLUME# ll
total 0
drwxr-xr-x 2 root root 108 Jul  6 10:52 ./
drwxr-xr-x 1 root root  23 Jul  6 10:50 ../
-rw-r--r-- 1 root root   0 Jul  6 10:51 files1
-rw-r--r-- 1 root root   0 Jul  6 10:51 files2
-rw-r--r-- 1 root root   0 Jul  6 10:51 files3
-rw-r--r-- 1 root root   0 Jul  6 10:52 new1
-rw-r--r-- 1 root root   0 Jul  6 10:52 new2
-rw-r--r-- 1 root root   0 Jul  6 10:52 new3
-rw-r--r-- 1 root root   0 Jul  6 10:52 new4
-rw-r--r-- 1 root root   0 Jul  6 10:52 new5
```

4) To check list of volumes: **docker volume ls**

5) To delete a volume: **docker volume rm volume_name**

6) To know full info about a volume: **docker volume inspect volume_name**

CREATING A VOLUME THROUGH DOCKERFILE:

Create a vim Dockerfile - - > Add the commands

a) FROM ubuntu

b)FROM nginx

VOLUME ["/volume_name"]

VOLUME ["/usr/share/nginx/html"]

Now build the Dockerfile, create a container from image, you'll get the volume.

REPLICATING DATA FROM HOST TO CONTAINER:

First create a folder in host (our system) with the help of command `mkdir app - -> cd app/ - -> pwd`
`/root/app/`.

`docker run -it --name container name -v /createdfolderpath:/volumename --privileged=true image_name`

docker run -it --name cont-1 -v /root/app:/my-volume --privileged=true ubuntu [or]

docker run -it --name cont-1 -v \$(pwd):/my-volume --privileged=true ubuntu

/root/app : source, **/my-volume**: the container default path where the volume needs to be created.

--privileged=true: sharing volume.

So whatever the data modifications are done in host it'll replicate in container (cont-1) and viceversa.

HOW TO MOUNT A EXISTING VOLUME TO A CONTAINER FOR DEPLOYMENT:

Create a volume - - > Go to docker default path (cd /var/lib/docker) - - > Go to volumes folder (cd volumes/) - - > Go to _data (cd _data) - - > Create some files and the volume with data needs to be mounted.

SYNTAX:

```
docker run -itd --name container_name --mount
source=created_volume_name,destination=/volume_name image_name
```

```
docker run -itd --name cont-1 --mount source=Swiggy,destination=/My-Volume ubuntu
```

This way whatever the code we've inside the volume can be easily deployed no matter how many times the modifications are done and same can be deployed in all the containers.

HOW TO DEPLOY AN APPLICATION BY MOUNTING A VOLUME THAT HAS A SOURCE CODE:

```
docker run -it -d --name web --mount source=cycle-volume,destination=/usr/local/apache2/htdocs/ -p
3436:80 httpd
```

If we've to directly declare the volume (-V) for deployment instead of mounting then the command is below:

```
docker run -itd --name -container_name -v /usr/share/nginx/html -p 8082:80 nginx
```

since it's an empty volume, we get the default nginx app deployed in order to have another app deployed we need to have the code in html file.

To check what is the volume attached to container, inspect it, docker inspect container_name.

Copy the volume ID and go to default path of volume, cd /var/lib/docker/volumes/volume name/_data/

Here remove the default html data of nginx, and add your own and use the host port and container port and the application gets deployed.

DEPLOYMENT IF CODE IS AT PWD:

If there's a source code in the present working directory (pwd) then the procedure as follows.

Practical steps: First clone a repository from a GitHub to get the source code to your local system at the pwd (root user), open the source code file (cd staticsite docker/), go to html file (cd html/). From html path, create a volume so that the source code that you have in the pwd will be moved from pwd to nginx or web sever default path and it gets deployed as we'll publish the host port and container port.

`docker run -itd --name container_name -v $(pwd):default path of web server -p hostport:containerport`

`docker run -itd --name cont-1 -v $(pwd):/usr/local/apache2/htdocs/ -p 8081:80 httpd`

Default path of nginx image: /usr/share/nginx/html

Default path of httpd image: /usr/local/apache2/htdocs

DOCKER HUB

Docker Hub is a default platform in docker registry under cloud based registry which helps to store the images and also to share with anyone in the world via internet. So, the image that has all the source code, dependencies can be shared to anyone and they can also use the image to run the application at their end.

There are two types of registries in Docker registry.

1. Cloud based registry: Docker Hub, Google Container registry (GCR), Amazon ECR (Amazon Elastic Container Registry).
2. Local Registry: Nexus, Jfrog, Docker Trusted Registry (DTR).

THE PROCESS OF PUSHING AN IMAGE INTO DOCKER HUB:

1. We take source code from the developers.
 2. Write a Dockerfile as per the source code.
 3. Build the Dockerfile to get an image.
 4. Follow the below steps to push the image into docker hub.
- a) Tag the created image (**`docker tag local-image dhub-username/repo-name:new-name`**)
- b) Login to docker hub from server (**`docker login`**)
- c) Before you push the image Push the image (**`docker push dhub-username/repo-name:tagname`**)

DOCKER INTEGRATION WITH JENKINS:

Install Docker and Jenkins in a same server and go to path `/var/lib/systemd/system/docker.service`.

Install Docker plugin in Jenkins to view the containers, images in Jenkins. However, we can't see them directly until below changes are made.

Server configuration:

Make changes vim `docker.service`. At `ExecStart` line, delete until **-H** and add **tcp://0.0.0.0:4243 -H unix:///var/run/docker.sock**.

Reload the daemon with the help of command **systemctl daemon-reload**.

Restart docker **systemctl restart docker** (when docker is restarted the containers goes to exit, so start and check in Jenkins).

Jenkins configuration:

Go to Clouds > New cloud > Give any name > Create.

Go to manage Jenkins and now you can see all the details of containers, images.

In the Jenkins dashboard, we can stop the containers, can only see running containers but not stopped ones.

DOCKER NETWORKS

In a Docker network, network refers to a communication channel over which Docker containers communicate each other or with the host system. This docker networking feature helps to communicate containers with one other to share information and resources from one network to another network.

WHY DOCKER NETWORK WITH AN EXAMPLE?

Let's assume we've an application container (APP) and a database container (DB), if an application container has to communicate with database container it'll be done with the help of IP addresses. If there's any heavy load on frontend container then it'll be down and may get deleted as well.

Ex: IP of APP: **172.13.0.1** and IP of DB: **172.13.0.2**

So if a new container gets created the IP would be **172.13.0.3** so it will not be connected to database container now, because, earlier we connected 0.1 with 0.2.

So here the docker networks gets into picture and with it's help we can maintain separate networks for front-end, back-end or database so that if the container gets deleted a new one will be created in the network and since both networks are connected there won't be any connection issues and the application can run in the new container uninterruptedly.

THE CONCLUDED VERSION OF ABOVE CONCEPT/EXPLANATION:

In real time if a container gets deleted, we make configurations in such a way that a new container gets automatically created and the same application has to run.

In case if there's any hardware failure/malfunction or if a new container is created, we'll get a new IP address so it may cause connection issues between app container and database container because the IP's that we first used to establish connection between APP and DB would differ from the new IP that we got and so in order to resolve the issue we got the docker networking.

With the help of docker networking even if there's any failure and even if a new container gets created because of deletion of an old container which was currently running it can still be communicated with database container by establishing the connection from network to network instead of container to container.

The bridge network is a default network that gets created when docker is installed in our local system, this provides network to all the containers that we create. This helps to assign the IP address to the created containers in a sequential order.

Ex: a) 172.17.0.1, b) 172.17.0.2, c) 172.17.0.3.

The sequential order of IP addresses gets assigned to networks in the below order. We can also assign our own ones in VPC concept.

Ex: a) 172.14.0.1, b) 172.15.0.1, c) 172.16.0.1

In a network we can create 65,534 containers from 65,536 of total containers (2 for broadcasting services) and the IP address range will be like 172.14.0.1, 0.2, 0.3 until 172.14.256.256.

DIFFERENT TYPES OF DOCKER NETWORKS:

There are 4 different types of docker networks. Bridge network, Host network, Overlay network and None network.

a) Bridge network: It's a default network in docker host where a container can communicate with another container within the same host.

b) Host network: It's a network where you want to have the same IP address to EC2 and a container.

c) Overlay network: It's a network that helps to communicate with containers in multiple hosts (docker hosts).

d) None network: If you want to expose your container to anyone then you use none network and no network gets assigned to a container.

DOCKER NETWORK COMMANDS:

1) To create a network: **docker network create network_name**

```
[root@ip-172-31-20-216 ~]# docker network create LOKI
58d4f96f1fcb170919a3abb06dc2ea74c5215e0df3ee7443e61a66319085b6f5
[root@ip-172-31-20-216 ~]# docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
58d4f96f1fcb	LOKI	bridge	local
365d21ec6073	bridge	bridge	local
13c58675d888	host	host	local
b0dd4fddc5cc	none	null	local

2) To know full info of network: **docker inspect network_name/ID**

3) To create a container inside a network: **docker run -it --name cont_name --network network_name image_name**

docker run -it --name cont-1 --network FLM-1 ubuntu

```
[root@ip-172-31-20-216 ~]# docker run -itd --name cont-22 --network LOKI nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
f11c1adaa26e: Already exists
c6b156574604: Already exists
ea5d7144c337: Already exists
1bbcb9df2c93: Already exists
537a6cfe3404: Already exists
767bff2cc03e: Already exists
adc73cb74f25: Already exists
Digest: sha256:67682bda769faelccf5183192b8daf37b64cae99c6c3302650f6f8bf5f0f95df
Status: Downloaded newer image for nginx:latest
1e0c46d4021b76411ad5a95d2a54905c6ab9490b224d4ae365f46670a4661be7
[root@ip-172-31-20-216 ~]# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
1e0c46d4021b	nginx	"/docker-entrypoint..."	6 seconds ago	Up 4 seconds	80/tcp	cont-22
e9d2f526a2ba	99mahi43/leo-repo:kaithi	"/docker-entrypoint..."	32 minutes ago	Up 32 minutes	0.0.0.0:9294->80/tcp, :::9294->80/tcp	cont99
6974f8b8c46e	45ca2f94ff40	"/docker-entrypoint..."	53 minutes ago	Up 53 minutes	0.0.0.0:9293->80/tcp, :::9293->80/tcp	cont100
8cea8b54b895	image:24	"/docker-entrypoint..."	2 hours ago	Up 2 hours	0.0.0.0:2324->80/tcp, :::2324->80/tcp	cont-23

To check if the created container is under the chosen network, then you can inspect (docker inspect cont-22) and you can view as shown in below.

```
"Networks": {
  "LOKI": {
    "IPAMConfig": null,
    "Links": null,
    "Aliases": [
      "1e0c46d4021b"
    ]
  }
}
```

4) To check list of networks: **docker network ls**

```
[root@ip-172-31-20-216 ~]# docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
58d4f96f1fcb	LOKI	bridge	local
365d21ec6073	bridge	bridge	local
13c58675d888	host	host	local
b0dd4fddc5cc	none	null	local

5) To remove a network: **docker rm network_name**

6) To remove unused networks: **docker network prune**

7) To connect a container to a network: **docker network connect network_name cont_name**

docker network connect FLM cont-2

8) To disconnect a container from a network: **docker network disconnect network_name container_name**

9) To know if the containers within the same network are communicating or not:

Ex: Let's take a bridge network, if there are 2 containers within the same network and then check if cont-1 (172.17.0.2) is communicating with cont-2 (172.17.0.3). Therefore you can use the below command.

Go inside a container (cont-1) --> **docker attach cont-1** --> and here give command **ping IP_address** of cont-2.

If you get packages which means it's communicating with one another.

ping is used to check if a system is connected to another one or not. If the image is ubuntu then use these two commands **apt update -y** and then again **apt install iputils-ping -y** (ping package) for the ping command to work.

10) To know commands that are available for docker network: **docker network**

It'll give you all the commands that are available like connect, disconnect, create etc.

DOCKER SWARM

Docker swarm is an orchestration service that helps to manage and handle multiple containers at the same time. It helps to manage multiple containers on multiple servers by implementing a cluster.

KEY TERMS IN DOCKER SWARM:

a) Cluster: It's nothing but a group of servers where you can run multiple containers. This cluster is like an alternative for Kubernetes. In real time most of the companies may use either Docker Swarm or Kubernetes.

b) Orchestration: In a general sense, orchestration refers to the coordination and management of multiple tasks or services to achieve a specific outcome. It involves arranging and controlling the elements of a system to work together in a harmonious and organized way. Orchestration can apply to various domains, including music, business processes, and information technology.

c) Orchestration service: It refers to a platform or a service that manages and handles multiple executable tasks or services at the same time.

d) Replication mode: Whenever a container is deleted/removed still the application can be accessed on all servers because a new one gets created in any of the servers inside the cluster. It works as a auto scaling group.

WHAT DOES DOCKER SWARM DO?

The activities inside the cluster are managed by Swarm manager (manager node) and the machines that join the cluster are called as Swarm worker (worker nodes). So this entire service is provided/allowed by Docker swarm. The docker has to be installed in Swarm manager and Swarm worker.

There are two types of nodes in docker swarm. Swarm manager and Swarm worker.

1) Swarm manager: The node is called as a manager node.

2) Swarm worker: The node is called as a worker node.

Swarm manager acts similar to a Master which assigns the work to swarm workers to perform individual service for a better performance. All the worker nodes are assigned to the manager node which is responsible for assigning the work to the worker nodes. Whenever if scaling or update is needed, the work will go to the manager node and manager node assigns the work to the worker nodes.

SERVICE - - > CONTAINER - - > APPLICATION

Whenever a service is created in a manager node, it creates a container automatically and runs the application.

We create only one service (container) in a manager node and not only accessing it from manager node we can also access it from individual worker nodes that are connected to swarm manager. In other words, docker swarm ensures that the application which is running in manager will also run in worker nodes that are linked or assigned with swarm manager. That's the vital role of Docker swarm.

COMPONENTS OF A DOCKER SWARM:

Service: It represents a feature (option) inside an application.

Task: It's a work that needs to be done.

Manager: Who assigns/manages the work to worker nodes.

Worker: Who does the work assigned by manager, which works for a specific purpose of the service.

DOCKER SWARM SETUP:

1) Install the docker on manager and worker nodes.

2) Initialize the docker swarm in manager node (manager ec2 instance) with the help of it's private IP address. (**docker swarm init --advertise-addr 172.31.3.72 [private IP-address]**)

```
[root@ip-172-31-21-1 ~]# docker swarm init --advertise-addr 172.31.21.1
Swarm initialized: current node (jwg2owpzi59eolhkv751yr428) is now a manager.

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

    docker swarm join --token SWMTKN-1-0o795jhbqvds0cy8ubqcotprmn8hix3qle2d94ygk1115r87n3-8ld7652w5bei8ygapihw0wa82 172.31.21.1:2377

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

3) To add a worker to this swarm, copy and paste the command from manager node in the worker nodes (worker ec2 instance).

docker swarm join --token SWMTKN-1-0o795jhbqvds0cy8ubqcotprmn8hix3qle2d94ygk1115r87n3-8ld7652w5bei8ygapihw0wa82 172.31.21.1:2377

```
[root@ip-172-31-22-62 ~]# docker swarm join --token
This node joined a swarm as a worker.
[root@ip-172-31-22-62 ~]#
```

So all we do is, init the swarm, copy paste the token in the worker nodes and a cluster is created (all servers are at one place).

DOCKER SWARM COMMANDS:

If a container gets deleted due to any unavoidable circumstances then another containers gets automatically created either of the servers. This is an advantage of docker swarm.

To init the docker swarm: **docker swarm init --advertise-addr private IP of manager ec2 instance.**

A) SERVICE COMMANDS:

1) To create a service: **docker service create --name service_name --publish host port:container port image_name**

docker service create --name Paytm --publish 8081:80 nginx

Here the above command explains that we're creating a service with a name and publishing a port number to access the application inside the container with the help of an image (web server). We haven't given container name but the service name that we gave will be given to container as well and it gets with an additional ID.

```
[root@ip-172-31-21-1 ~]# docker service create --name MAHI --publish 2526:80 nginx
4f8tbwq0ww3hroq5xd86y6qrf
overall progress: 1 out of 1 tasks
1/1: running [=====>]
verify: Service converged
[root@ip-172-31-21-1 ~]# docker service ls
```

ID	NAME	MODE	REPLICAS	IMAGE	PORTS
4f8tbwq0ww3h	MAHI	replicated	1/1	nginx:latest	*:2526->80/tcp

```
[root@ip-172-31-21-1 ~]# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
479c5c2af8a0	nginx:latest	"/docker-entrypoint..."	23 seconds ago	Up 22 seconds	80/tcp	MAHI.1.24w4zz3qx0o8fquhcakm8pcmr

```
[root@ip-172-31-21-1 ~]#
```

2) To create a service for multiple containers: **docker service create --name service_name --publish host port:container port --replicas mention the number of containers you want to create image_name**

docker service create --name swiggy --publish 8081:80 --replicas 5 nginx

NOTE: When you create your own image, build through docker file and if you create a service with the help of that created image then the container gets created only in that particular instance only which has a docker image. No container will be created in remaining worker servers since we didn't have the same image in worker nodes/worker servers/worker instances. If we pull any public image from docker hub then if we create a service in manager node then the same will be created in worker nodes as well. That's the difference that brings because of an image if it is within the local or from docker hub.

NOTE: If an application is already deployed and if a client get back to us to add an update/add a feature to the app then instead of creating a new service to create a container to run the app with new feature we can just update the image so that the new can be accessed through the app.

All we need to do is to after the developers write the code for the update/new feature/additional feature we'll get the source code, write the Dockerfile as per the source code, build the Dockerfile to get an image, and then update the image to the service which is already running. So that the new update will be directly deployed with the same port that was created for deployment.

docker service update --image image_name service_name

docker service update --name image-2 Paytm

What happens in the backend when the image is updated?

The first image (cycle): container gets created based on replicas.

The second image (game): The above or previous containers gets deleted.

The third image (dm): The above or previous containers gets deleted and new one gets deleted and the same can be accessed when you update the image.

3) To check only list of container created using specific service: **docker service ps service_name**

docker service ps Paytm

4) To check the logs of service: **docker service logs service_name**

5) To inspect a service: **docker service inspect service_name**

6) To check the list of services: **docker service ls**

7) To remove a service: **docker service rm service_name**

If you want to increase the containers for a service, if the service already has 4 and if you need 6 more containers to have overall 10, then you need to give 10 not what you need. So 6 will get created.

8) To scale up the containers for a particular service: **docker service scale service_name=10** **docker service scale Paytm=10**

If you want to decrease the number of containers from 10 to , so it's called scale down.

9) To scale down the containers for a particular service: **docker service scale service_name=5**
docker service scale Paytm=5

10) To rollback: **docker service rollback service_name**

If you've like 3 versions for an application, and if 4th one is updated but if client needs the old version which is 3rd then we need to rollback. Again if we rollback it doesn't go to 2nd instead it goes to 4th again and it's kind of a loop, in order to resolve this issue we use "KUBERNETES".

11) To rollback to a specific version: **docker service rollback image_name**

B) NODE COMMANDS:

In case if a new worker node is created (worker instance is launched) and if you want to add it to the swarm manager or manager node then if you give the command (docker swarm init --advertise-addr private IP address of manager node) to get the previous token that was first generated while initializing the swarm, we'll get an error. So in order to resolve this, we have a command.

If you want to promote worker as a manager, then first we need to remove the node as a worker from the swarm and then generate a token for manager, copy-paste in the worker node and it'll become a swarm manager.

1) To get worker token: **docker swarm join-token worker**

2) To get manager token: **docker swarm join-token manager**

Now you can copy-paste the token from manager node to the worker node and the node will join as a worker in swarm.

3) To check the nodes: **docker node ls** (gives all nodes not only manager or only worker nodes)

4) To leave a node/server from a cluster/swarm: **docker swarm leave**

So with the help of this command, the node will only leave the swarm but it doesn't get removed from cluster, the status changes to "Down" (takes 10 seconds) and then you can **remove/delete** by the below command.

5) To delete/remove the node: **docker node rm node_ID**

DOCKER COMPOSE

Docker compose used to build, run and manage multiple containers in a single server (host) at the same time with the help of a YAML file. YAML file helps to manage the multiple containers as a single service. We can pass parameters in the YAML file to create containers, networks and volumes at the same time. The compose file provides a way to document and configure all of the application service dependencies (databases, queues, caches, web services, API's, etc.)

The docker compose file includes services, networks and volumes. The default path is `./docker-compose.yml`, `compose.yml` (YAML = yml).

The YAML file name that you create must have any one of the names from the following:

docker-compose.yaml

docker-compose.yml

compose.yaml

compose.yml

If you've created a compose file of your own name (**mahi.yml**) with correct extension as yaml or yml, instead of '**docker-compose.yml**' and if you want to execute the file we need to declare "**-f**" in the command. So to execute the compose file command should be as "**docker-compose -f mahi.yml up -d**".

DOCKER COMPOSE FILE COMPONENTS:

1) version: It specifies the version of the docker compose file (currently it's "3").

If you want to know why and which specific version we need to use to mention in the YAML file you can

check it from the official documentation of docker (<https://docs.docker.com>)

2) service: It specifies services (options/features) in your application.

3) networks: You can specify the networking setup of your application (to define where the service needs to create if you do not mention anything it creates in bridge as it's default).

4) volumes: To define the volumes used by your application.

5) configurations: To add external configurations to your containers. Keeping configurations external will make your containers more generic.

DOCKER COMPOSE SETUP:

1) Install docker and start docker.

2) Install compose with the help of script (copy and paste the steps).

```
sudo curl -SL https://github.com/docker/compose/releases/download/v2.28.1/docker-compose-linux-x86_64 -o /usr/local/bin/docker-compose
```

```
sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
```

```
sudo chmod +x /usr/local/bin/docker-compose
```

```
docker-compose version
```

Whenever a developer make changes to the source code, we've to update the image, build it again and run the compose file. So in docker compose file, we remove the image and add build in yaml file. Here build represents the place where we've the dockerfile. So due to this whenever a developer changes the code we need not to update the image and build, instead we use **docker-compose build** so automatically the image gets build in all services. Then, directly run the compose file. The changes that made to the source code will be deployed. Therefore, giving build in the compose file will make job easier.

DOCKER COMPOSE COMMANDS:

1) To execute/run the docker compose file: **docker-compose up -d** (Once the docker file is written, it can start)

2) To build the dockerfile mentioned in yaml file for all services: **docker-compose build**

3) To start all services: **docker-compose up** (The service starts and container gets created)

4) To delete all services: **docker-compose down** (The service gets stopped and the container gets deleted)

5) To start containers of the service: **docker-compose start**

6) To stop containers of the service: **docker-compose stop**

7) To pause the container: **docker-compose pause** (It pauses the container and status will be up but paused status will be added under status and you can't access the application)

8) To unpause the container: **docker-compose unpause**

9) To check the logs of the containers in compose file: **docker-compose logs**

10) To check the list of containers related to compose file only: **docker-compose ps**

11) To check the list of images in the docker compose file: **docker-compose images**

12) To check config details of the compose file: **docker-compose config**

13) To create a database container with a volume:

docker run -d --name container_name -v created volume name:/volume name that want to create inside container -p hostport:containerport image_name -e

docker run -d --name cont-1 -v volume-1:/VOLUME -p 2222:3306 -e MYSQL_ROOT_PASSWORD=mahi mysql/mysql-server:5.7

We're creating a database container with a volume that you create and accessing it by publishing the port numbers and passing a ENV (-e) variable so that the values can be accessed inside a container not ARG with database image name.=bin

14) To go inside the container: **docker exec -it container_name /bin/bash or bash**

After that, mysql -u (default username is root) -p (password) which is mysql -uroot -pmahi@123 then to check queries, SHOW DATABASES; it shows list of databases, to create database CREATE DATABASE name; to select to go into a database USE databasename;. This is how backend team works on databases. We only build the dockerfile, we update that image.

Difference between start and pause?

If you stop the container you can't access the application and the network system and file system inside the container but in pause you can still access the network system and the file system. The application keeps on loading but it doesn't work.

Difference between docker swarm and docker compose?

The only major difference is in docker swarm we create multiple servers multiple applications (or) multiple servers for multiple containers whereas in docker compose we create multiple containers in a single server, like we can run many applications through a single server by mentioning their port

numbers, images, networks, volumes in a yml file. Swarm is an orchestration service but compose is a tool.

Simplified differences between Docker Swarm, Compose, Stack:

Docker **Swarm** → **Single container** on **multiple servers**.

Docker **Compose** → **Multiple containers** on a **Single server**.

Docker **Stack** → **Multiple containers** on **multiple servers**.

DOCKER STACK

Docker Stack is a combination of Docker compose and Docker swarm where we can create multiple containers on multiple servers so that multiple applications can run on all the servers you create.

It's done with the help of a compose file. We mention multiple services (which creates multiple containers), mention the images from dockerhub, publish the port numbers so that we can access multiple applications on multiple servers.

We do not use "build" component in the compose file because if we mention the build by declaring the dockerfile path and it may be in different paths and it gets build only in single server (local) not on multiple servers. The container gets build only in one server not on multiple ones so since we're using Stack to deploy the application on multiple servers we need to use image component in compose file.

The main advantage of Docker stack is we can create Portainer, it's like a GUI to our Docker. However, in real-time we don't completely rely on Portainer for any creation or deletion as it's a third party and we can use it only for monitoring purpose.

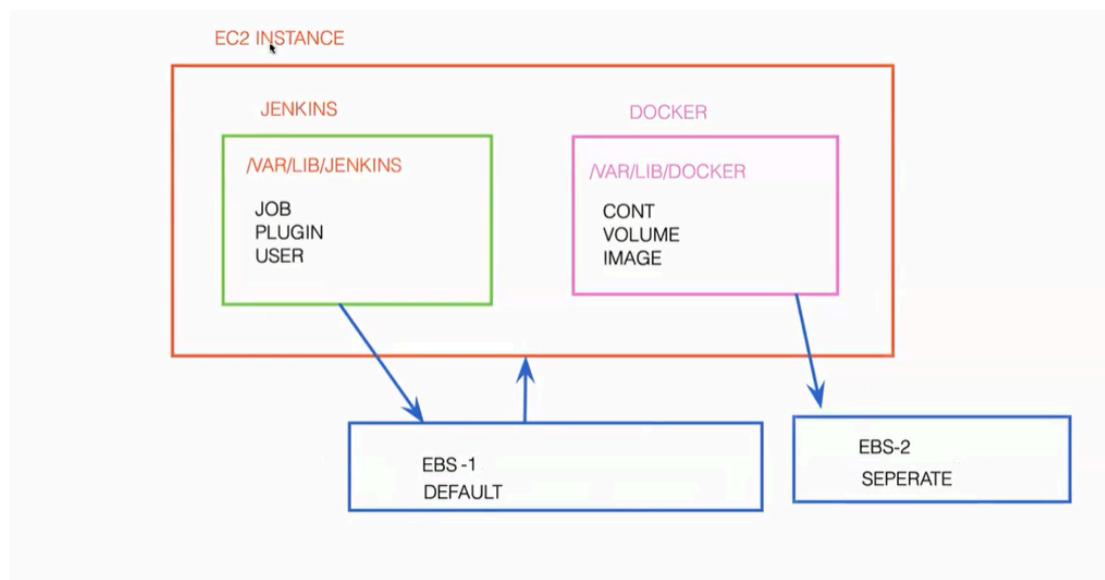
In order to create a Portainer, we need the following requirements:

- 1) Docker swarm mode and all ports should be enable with docker engine.
- 2) `curl -L https://downloads.portainer.io/ce2-16/portainer-agent-stack.yml -o portainer-agent-stack.yml`
- 3) `docker stack deploy -c portainer-agent-stack.yml stack_name (-c - - > compose)`
- 4) `docker ps`
- 5) Use public IP of Swarm master with the port number 9000.

Portainer is a Docker GUI where we can do everything that we did in CLI that can be services, stacks, containers, images, networks, volumes, configs, secrets, swarm.

We've deployed applications through the Dockerfile with html, below are the screenshots of deployment with the help of java and node.js.

DOCKER DIRECTORY DATA



SITUATION:

We've Jenkins and Docker installed in same EC2 instance and they both have two different default paths. All the data related to job, plugin, user etc., in Jenkins and containers, volumes, networks, images etc., in Docker will also be stored in EBS-volume of instance that we take while launching an instance. For example, if we've to maintain separate volume for docker then docker directory data concept comes into picture.

- Uninstall the docker - yum remove docker -y
- remove all the files - rm -rf /var/lib/docker/*
- create a volume in same AZ & attach it to the instance
- to check it is attached or not - fdisk -l
- to format it - fdisk /dev/xvdf --> n p 1 enter enter w
- set a path - vi /etc/fstab (/dev/xvdf1 /var/lib/docker/ ext4 defaults 0 0)
- mkfs.ext4 /dev/xvdf1
- mount -a
- install docker - yum install docker -y && systemctl restart docker
- now you can see - ls /var/lib/docker
- df -h

END TO END DEPLOYMENT OF A NODE.JS APPLICATION USING DOCKER

This deployment is a kind of 'DevSecOps' level because it has more security as we're having a multiple stages in the pipeline like scanning the code, scanning the image and container.

CLEAN WS --> CODE --> SONARQUBE ANALYSIS --> Quality Gates --> Install Dependencies --> OWASP --> Trivy --> Build --> Push --> Container.

1) **Clean WS** : This is used to clean the Workspace (folder) in Jenkins in case if there are any files or data about regarding the previous jobs.

2) **Code**: This stage is to get the source code from GitHub to the CI server.

3) **SonarQube Analysis**:

SonarQube is a Code Quality Assurance tool that collects and analyzes source code, and provides reports for the code quality of your project.

[or]

This stage is for the code quality test which shows if there are any bugs and vulnerabilities in the source code.

4) **Quality Gates**: This is declared in order to know if

5) **Install Dependencies**:

6) **OWASP**:

7) **Trivy**:

8) **Build:** This stage is to build the image.

9) **Tag and Push:** This is to tag the image and push it into docker hub.

9.1) **Scan the image:** This is to scan the image that we've created.

10) **Container:** This is the final stage of the pipeline where the built image with a tag is used to create a container and the container has a Dockerfile that helps to Deploy the application.

PROCEDURE:

STEP-1: Launch an instance with t2.large.

STEP-2: Install JENKINS, GIT, DOCKER & TRIVY.

STEP-3: Install the following plugins in Jenkins.

- SonarQube Scanner
- Node.js
- OWASP Dependency Check
- Docker Pipeline
- Eclipse Temurin Installer Version

STEP-4: Configure all the plugins into Jenkins.

STEP-5: Write a pipeline.

JENKINS INSTALLATION: You can either copy-paste all the steps or can run script (.sh)

```
amazon-linux-extras install java-openjdk11 -y
```

```
sudo wget -O /etc/yum.repos.d/jenkins.repo https://pkg.jenkins.io/redhat-stable/jenkins.repo
```

```
sudo rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io-2023.key
```

```
yum install jenkins -y
```

```
systemctl start jenkins
```

GIT & DOCKER INSTALLATION:

```
yum install git docker -y
```

```
systemctl start docker
```

```
chmod 777 ///var/run/docker.sock
```

TRIVY INSTALLATION:

```
wget https://github.com/aquasecurity/trivy/releases/download/v0.18.3/trivy_0.18.3_Linux-64bit.tar.gz
```

```
tar zxvf trivy_0.18.3_Linux-64bit.tar.gz
```

```
sudo mv trivy /usr/local/bin/
```

```
export PATH=$PATH:/usr/local/bin/
```

```
source .bashrc
```

SONARQUBE SETUP USING DOCKER:

SonarQube needs to be configured into Global tools as well. This can be done only when the SonarQube is accessed through it's 9000 port, instead of following whole procedure to setup a SonarQube, we can access it by using the sonar's image and creating a container from it.

```
docker run -d --name cont_name -p hostport:contport image_name
```

```
docker run -d --name sonar -p 9000:9000 sonarqube:lts-community
```

Login to SonarQube with the help of public IP of our instance and default port number of SonarQube i.e., [100.24.205.196](#):9000 and hit enter. You'll be landed as shown below, enter "admin" as username and also as the password to login.

Log in to SonarQube

[Cancel](#)

Create a new password.

Update your password

This account should not use the default password.

Enter a new password

All fields marked with * are required

Old Password *

New Password *

Confirm Password *

Update

Configure all the plugins with Jenkins means the same which is Integration of NodeJS, SonarQube, Java, OWASP tools by configuring with Jenkins:

Go to Manage Jenkins > Tools in Jenkins.

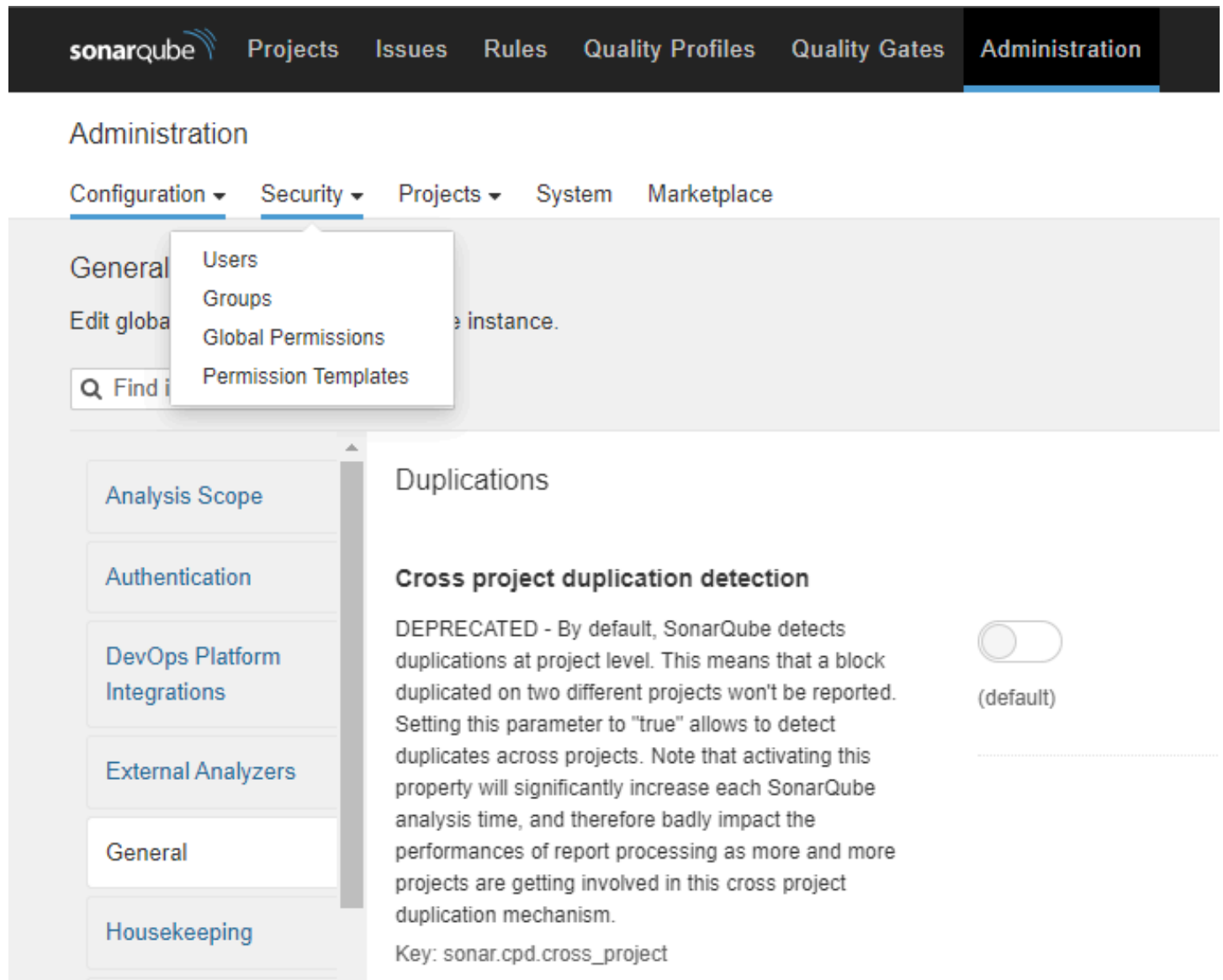
1) - > NodeJS Installations > give name > NodeJS 16.2.0.

2) - > JDK Installations > give a name, check the box '**Install automatically**' > Add installer > Install from adoptium.net > version jdk 17.0.8.1+1.

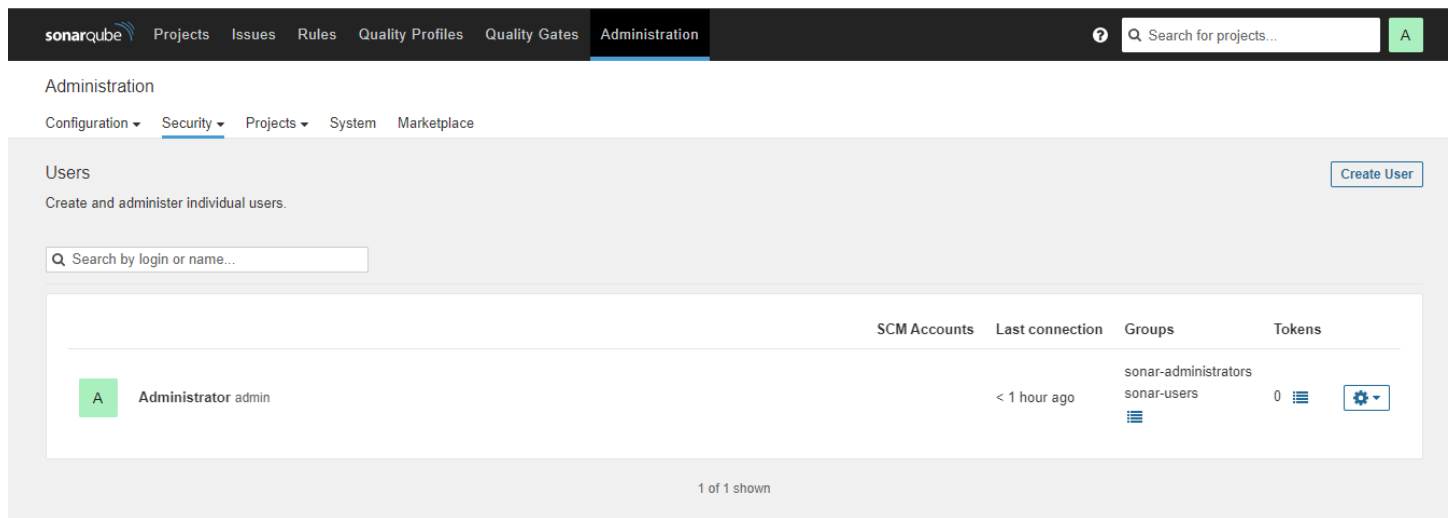
3) - > SonarQube Scanner installations > give a name, check the box 'Install automatically' > leave it to default version itself.

4) -> Dependency-Check installations > give a name > check the box 'Install automatically' > Install from github.com > version 6.5.1.

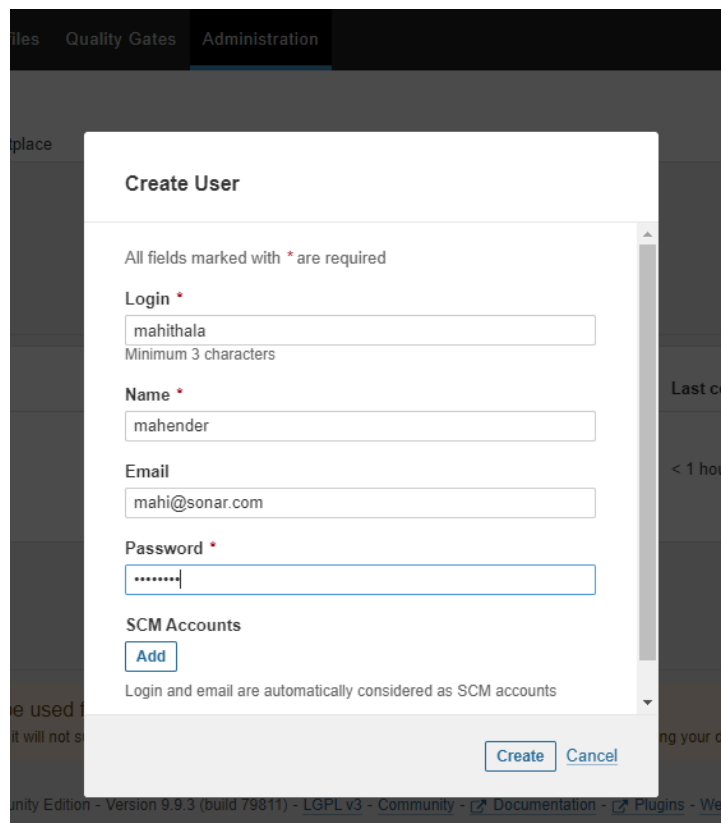
1) This is the homepage of the SonarQube. Go to Administration, Security, click on Users.



2) Click on create user in SonarQube.



3) Fill out the details and click on create.



4) Click on three lines with dots beside settings icon for the created user and then click on “Update Tokens”.

sonarqube Projects Issues Rules Quality Profiles Quality Gates Administration ? Search for projects... A

Administration

Configuration Security Projects System Marketplace

Users Create User

Create and administer individual users.

Search by login or name...

	SCM Accounts	Last connection	Groups	Tokens
A Administrator admin		< 1 hour ago	sonar-administrators sonar-users	0 ⋮ ⚙️
M mahender mahithala mahi@sonar.com		Never	sonar-users	0 ⋮ ⚙️

2 of 2 shown

5) Enter the token name and click on Generate, copy the token ID.

sonarqube Projects Issues Rules Quality Profiles Quality Gates Administration ? Search for projects... A

Administration

Tokens of mahender

Generate Tokens

Name Expires in

Enter Token Name 30 days Generate

! New token "mytoken" has been created. Make sure you copy it now, you won't be able to see it again!

Copy `squ_4af122bb703d9b139547a3e458712f3e38caa273`


Name	Type	Project	Last use	Created	Expiration	
mytoken	User		Never	November 27, 2023	December 27, 2023	Revoke

Done




! Embedded database should be used for evaluation purposes only.
The embedded database will not scale, it will not support upgrading to newer versions of SonarQube, and there is no support for migrating your data out of it into a different database engine.

Copy the token ID and now follow the below steps.

Add the token to credentials. Go to Manage Jenkins > Credentials > system > Global credentials (unrestricted) > Add credentials and then add it.

 Jenkins

Search (CTRL+K)

 1  1 mahender.jangam  log out

Dashboard > Manage Jenkins > Credentials > System > Global credentials (unrestricted) >

New credentials

Kind

Secret text

Scope ?

Global (Jenkins, nodes, items, all child items, etc)

Secret

.....

ID ?

sonar-token

Description ?

Create

Configure SonarQube into Global as well:

Manage Jenkins > System > Search for SonarQube Installations > Click Add Sonar> mention the name > SonarQube website URL in your local > Server authentication token > Select the token name you gave for Sonar while generating a token.

Dashboard > Manage Jenkins > System >

SonarQube servers

If checked, job administrators will be able to inject a SonarQube server configuration as environment variables in the build.

☐ Environment variables

SonarQube installations

List of SonarQube installations

Name

mysonar

Server URL

Default is http://localhost:9000

http://100.24.205.196:9000

Server authentication token

SonarQube authentication token. Mandatory when anonymous access is disabled.

sonar-token

+ Add

Advanced

Save

Apply

Add a Webhook in SonarQube: Administration > Configuration > Webhooks and create a Webhook as shown below.


```
    cleanWs()

  }
}

stage ("Code") {

  steps {

    git branch: 'main', url: 'https://github.com/devops0014/Zomato-Repo.git'

  }

}

stage("Sonarqube Analysis") {

  steps{

    withSonarQubeEnv('my-sonar') {

      sh "' $SCANNER_HOME/bin/sonar-scanner -Dsonar.projectName=zomato \

      -Dsonar.projectKey=zomato '"

    }

  }

}

stage ("Quality Gates") {

  steps {

    script {

      waitForQualityGate abortPipeline: false, credentialsId: 'sonar-token'

    }

  }

}

stage ("Install dependencies") {

  steps {

    sh 'npm install'
```

```

    }

}

stage ("OWASP") {

    steps {

        dependencyCheck additionalArguments: '--scan ./ --disableYarnAudit --disableNodeAudit',
odcInstallation: 'Dp-Check'

        dependencyCheckPublisher pattern: '**/dependency-check-report.xml'

    }

}

stage ("Trivy scan") {

    steps {

        sh "trivy fs . > trivyfs.txt"

    }

}

stage ("Build Dockerfile") {

    steps {

        sh 'docker build -t image1 .'

    }

}

stage("Docker Build & Push"){

    steps{

        script{

            withDockerRegistry(credentialsId: 'docker-password') {

                sh "docker tag image1 99mahi43/loki:mydockerimage"

                sh "docker push 99mahi43/loki:mydockerimage"

            }

        }

    }

}

```

```
    }  
  }  
}  
stage ("Scan image") {  
  steps {  
    sh 'trivy image 99mahi43/loki:mydockerimage'  
  }  
}  
stage ("Deploy") {  
  steps {  
    sh 'docker run -d --name zomato -p 3000:3000 99mahi43/loki:mydockerimage'  
  }  
}  
}  
}
```