Continuous Delivery

Continuous delivery

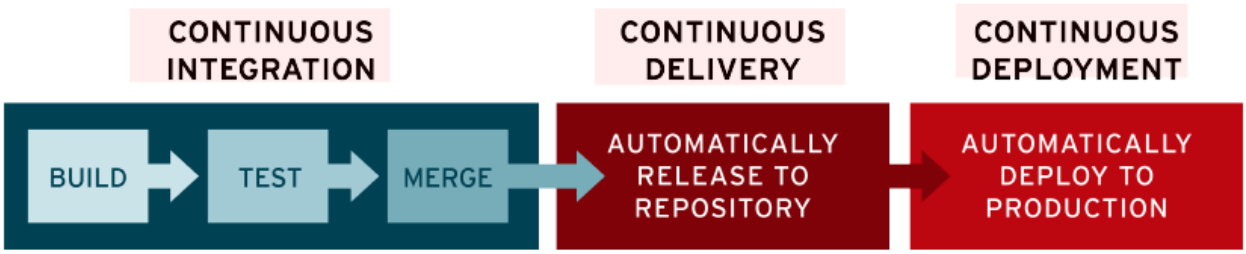
Continuous delivery is an extension of continuous integration since it automatically deploys all code changes to a testing and/or production environment after the build stage.
This means that on top of automated testing, you have an automated release process and you can deploy your application any time by clicking a button.

In theory, with continuous delivery, you can decide to release daily, weekly, fortnightly, or whatever suits your business requirements. However, if you truly want to get the benefits of continuous delivery, you should deploy to production as early as possible to make sure that you release small batches that are easy to troubleshoot in case of a problem.

Continuous deployment

Continuous deployment goes one step further than continuous delivery. With this practice, every change that passes all stages of your production pipeline is released to your customers. There's no human intervention, and only a failed test will prevent a new change to be deployed to production.

Continuous deployment is an excellent way to accelerate the feedback loop with your customers and take pressure off the team as there isn't a "release day" anymore. Developers can focus on building software, and they see their work go live minutes after they've finished working on it.



CI and CD stand for continuous integration and continuous delivery/continuous deployment. In very simple terms, CI is a modern software development practice in which incremental code changes are made frequently and reliably. Automated build-and-test steps triggered by CI ensure that code changes being merged into the repository are reliable. The code is then delivered quickly and seamlessly as a part of the CD process. In the software world, the CI/CD pipeline refers to the automation that enables incremental code changes from developers' desktops to be delivered quickly and reliably to production.

Why is CI/CD important?

CI/CD allows organizations to ship software quickly and efficiently. CI/CD facilitates an effective process for getting products to market faster than ever before, continuously delivering code into production, and ensuring an ongoing flow of new features and bug fixes via the most efficient delivery method.

What is the difference between CI and CD?

Continuous integration (CI) is practice that involves developers making small changes and checks to their code. Due to the scale of requirements and the number of steps involved, this process is automated to ensure that teams can build, test, and package their applications in a reliable and repeatable way. CI helps streamline code changes, thereby increasing time for developers to make changes and contribute to improved software.

Continuous delivery (CD) is the automated delivery of completed code to environments like testing and development. CD provides an automated and consistent way for code to be delivered to these environments.

Continuous deployment is the next step of continuous delivery. Every change that passes the automated tests is automatically placed in production, resulting in many production deployments.

Continuous deployment should be the goal of most companies that are not constrained by regulatory or other requirements.

In short, CI is a set of practices performed *as developers are writing* code, and CD is a set of practices performed *after* the code is completed.

Containerization using Docker

Docker is the containerization platform that is used to package your application and all its dependencies together in the form of containers to make sure that your application works seamlessly in any environment which can be developed or tested or in production. Docker is a tool designed to make it easier to create, deploy, and run applications by using containers.

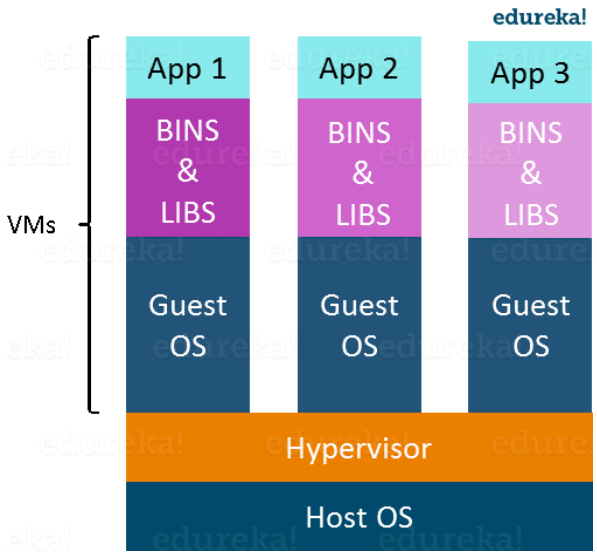
Docker is the world's leading software container platform. It was launched in 2013 by a company called Dotcloud, Inc which was later renamed Docker, Inc. It is written in the Go language. It has been just six years since Docker was launched yet communities have already shifted to it from VMs. Docker is designed to benefit both developers and system administrators making it a part of many DevOps toolchains. Developers can write code without worrying about the testing and production environment. Sysadmins need not worry about infrastructure as Docker can easily scale up and scale down the number of systems. Docker comes into play at the deployment stage of the software development cycle.

What is Virtualization?

Virtualization is the technique of importing a Guest operating system on top of a Host operating system. This technique was a revelation at the beginning because it allowed developers to run

multiple operating systems in different virtual machines all running on the same host. This eliminated the need for extra hardware resource. The advantages of Virtual Machines or Virtualization are:

- Multiple operating systems can run on the same machine
- Maintenance and Recovery were easy in case of failure conditions
- Total cost of ownership was also less due to the reduced need for infrastructure



In the above diagram, you can see there is a host operating system on which there are 3 guest operating systems running which is nothing but the virtual machines.

As you know nothing is perfect, Virtualization also has some shortcomings. Running multiple Virtual Machines in the same host operating system leads to performance degradation. This is because of the guest OS running on top of the host OS, which will have its own kernel and set of libraries and dependencies. This takes up a large chunk of system resources, i.e. hard disk, processor and especially RAM.

Another problem with Virtual Machines which uses virtualization is that it takes almost a minute to boot-up. This is very critical in case of real-time applications.

Following are the disadvantages of Virtualization:

- Running multiple Virtual Machines leads to unstable performance
- Hypervisors are not as efficient as the host operating system
- Boot up process is long and takes time

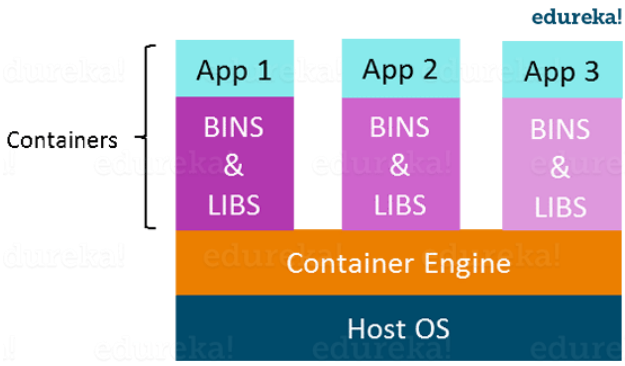
These drawbacks led to the emergence of a new technique called Containerization.

What is Containerization?

Containerization is the technique of bringing virtualization to the operating system level. While Virtualization brings abstraction to the hardware, Containerization brings abstraction to the operating system. Do note that Containerization is also a type of Virtualization. Containerization is however more efficient because there is no guest OS here and utilizes a host's operating system, share relevant libraries & resources as and when needed unlike virtual machines. Application specific binaries and libraries of containers run on the host kernel, which makes processing and execution very fast. Even booting-up a container takes only a fraction of a second. Because all the containers share, host operating system and holds only the application related binaries & libraries. They are lightweight and faster than Virtual Machines.

Advantages of Containerization over Virtualization:

- Containers on the same OS kernel are lighter and smaller
- Better resource utilization compared to VMs
- Boot-up process is short and takes few seconds

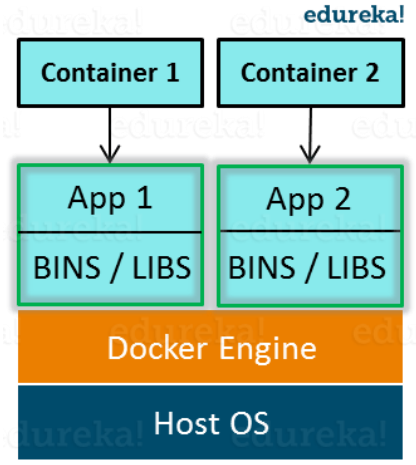


In the above diagram, you can see that there is a host operating system which is shared by all the containers. Containers only contain application specific libraries which are separate for each container and they are faster and do not waste any resources.

All these containers are handled by the containerization layer which is not native to the host operating system. Hence software is needed, which can enable you to create & run containers on your host operating system.

Introduction to Docker

Docker is a containerization platform that packages your application and all its dependencies together in the form of Containers to ensure that your application works seamlessly in any environment.



As you can see in the diagram on the right, each application will run on a separate container and will have its own set of libraries and dependencies. This also ensures that there is process level isolation, meaning each application is independent of other applications, giving developers surety that they can build applications that will not interfere with one another.

As a developer, I can build a container which has different applications installed on it and give it to my QA team who will only need to run the container to replicate the developer environment

Benefits of Docker

Now, the QA team need not install all the dependent software and applications to test the code and this helps them save lots of time and energy. This also ensures that the working environment is consistent across all the individuals involved in the process, starting from development to deployment. The number of systems can be scaled up easily and the code can be deployed on them effortlessly.

Features of Docker

- Docker has the ability to reduce the size of development by providing a smaller footprint of the operating system via containers.
- With containers, it becomes easier for teams across different units, such as development, QA and Operations to work seamlessly across applications.
- You can deploy Docker containers anywhere, on any physical and virtual machines and even on the cloud.
- Since Docker containers are pretty lightweight, they are very easily scalable.

Virtualization vs Containerization

Virtualization and Containerization both let you run multiple operating systems inside a host machine.

Virtualization deals with creating many operating systems in a single host machine. Containerization on the other hand will create multiple containers for every type of application as required.

The major difference is that there are multiple Guest Operating Systems in Virtualization which are absent in Containerization. The best part of Containerization is that it is very lightweight as compared to the heavy virtualization.

Advantages of Docker –

Docker has become popular nowadays because of the benefits provided by Docker containers.
The main advantages of Docker are:

1. **Speed** – The speed of Docker containers compared to a virtual machine is very fast. The time required to build a container is very fast because they are tiny and lightweight. Development, testing, and deployment can be done faster as containers are small. Containers can be pushed for testing once they have been built and then from there on to the production environment.
2. **Portability** – The applications that are built inside docker containers are extremely portable. These portable applications can easily be moved anywhere as a single element and their performance also remains the same.
3. **Scalability** – Docker has the ability that it can be deployed on several physical servers, data servers, and cloud platforms. It can also be run on every Linux machine. Containers can easily be moved from a cloud environment to a local host and from there back to the cloud again at a fast pace.
4. **Density** – Docker uses the resources that are available more efficiently because it does not use a hypervisor. This is the reason that more containers can be run on a single host as compared to virtual machines. Docker Containers have higher performance because of their high density and no overhead wastage of resources.

Components of Docker

1. **Docker Image**
   - It is a file, comprised of multiple layers, used to execute code in a Docker container.
   - They are a set of instructions used to create docker containers.
2. **Docker Container**
   - It is a runtime instance of an image.
   - Allows developers to package applications with all parts needed such as libraries and other dependencies.
3. **Docker file**
   - It is a text document that contains necessary commands which on execution helps assemble a Docker Image.
   - Docker image is created using a Docker file.
4. **Docker Engine**
   - The software that hosts the containers is named Docker Engine.
   - Docker Engine is a client-server based application

     The docker engine has **3 main** components:

- **Server**: It is responsible for creating and managing Docker images, containers, networks, and volumes on the Docker. It is referred to as a daemon process.
- **REST API**: It specifies how the applications can interact with the Server and instructs it what to do.
- **Client**: The Client is a docker command-line interface (CLI), that allows us to interact with Docker using the docker commands.

5. **Docker Hub**
   - Docker Hub is the official online repository where you can find other Docker Images that are available for use.
   - It makes it easy to find, manage, and share container images with others.

How Docker works?

Steps :

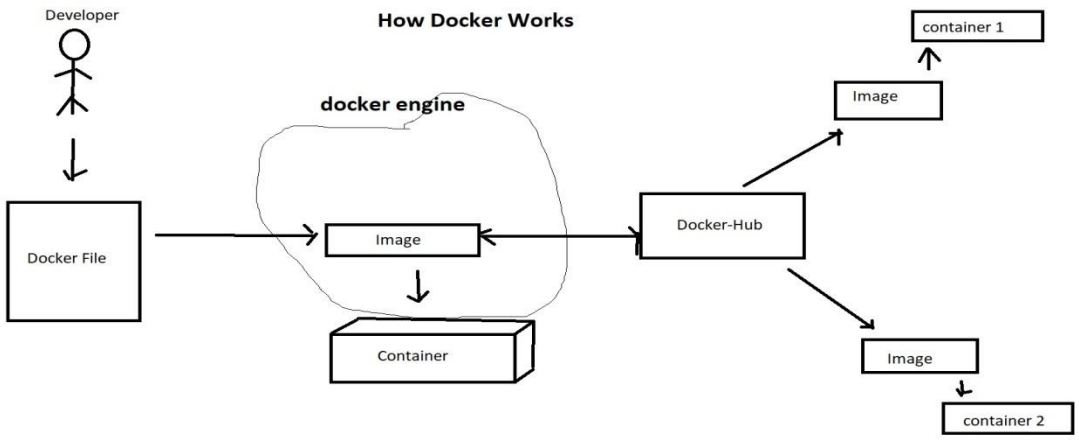
1) Developer creates a docker file

2) Run docker file on Docker Engine then it creates image

3) from image container is created

4) upload image on Docker-Hub for reuse

Fig: How Docker works



Basic commands in docker:

1) To see all images present in our local machine

#docker images

2) To find images on docker-hub

docker search jenkins

3) To install docker

yum install docker -y

4) To download images from docker hub

docker pull jenkins

5) To create and run (start) container from image

docker run -it jenkins /bin/bash

6) To rename container

docker run -it --name xyz jenkins /bin/bash

7) To check docker service (start or stop)

service docker status

8) To start container

docker start container_name

9) To go inside container

docker attach container_name

10) To see all containers

docker ps -a

11) To see running containers

docker ps

12)To stop container

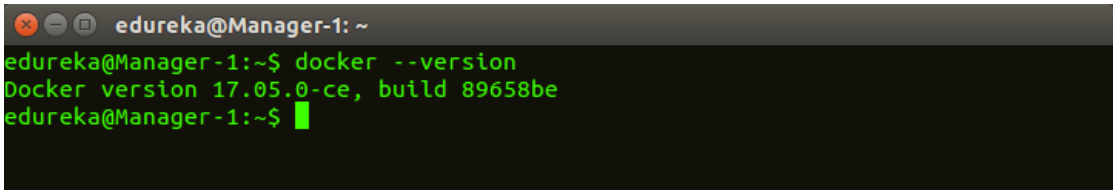
docker stop container_name

13) To delete container

docker rm container_name

1. **docker –version**

- This command is used to get the currently installed version of docker



2. **docker pull**

Usage: docker pull <image name>

This command is used to pull images from the **docker repository**(hub.docker.com)

```
edureka@Manager-1: ~
edureka@Manager-1:~$ docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
ae79f2514705: Pull complete
c59d01a7e4ca: Pull complete
41ba73a9054d: Pull complete
f1bbfd495cc1: Pull complete
0c346f7223e2: Pull complete
Digest: sha256:6eb24585b1b2e7402600450d289ea0fd195cfb76893032bbbb3943e041ec8a65
Status: Downloaded newer image for ubuntu:latest
edureka@Manager-1:~$
```

3. **docker run**

Usage: docker run -it -d <image name>

This command is used to create a container from an image

```
edureka@Manager-1: ~
edureka@Manager-1:~$ docker run -it -d ubuntu
f49b58b66d1ebc7c2d9e42280c1e24019b3202fb3e69050c45e806e6f9b65f71
edureka@Manager-1:~$
```

4. **docker ps**

- This command is used to list the running containers

```
edureka@Manager-1: ~
edureka@Manager-1:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
              PORTS               NAMES
f49b58b66d1e        ubuntu              "/bin/bash"         6 minutes ago       Up 6 mi
nutes                           angry_knuth
edureka@Manager-1:~$
```

5. **docker ps -a**

- This command is used to show all the running and exited containers

```
edureka@Manager-1: ~
edureka@Manager-1:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
                      PORTS               NAMES
fe6e370a1c9c        ubuntu              "/bin/bash"         12 seconds ago      Up 11
seconds                                   boring_ritchie
2b86a0703d4f        ubuntu              "/bin/bash"         About a minute ago  Exited
 (0) 21 seconds ago                       infallible_galileo
edureka@Manager-1:~$
```

6. **docker exec**

   Usage: docker exec -it <container id> bash

   This command is used to access the running container

```
root@fe6e370a1c9c: /
edureka@Manager-1:~$ docker exec -it fe6e370a1c9c bash
root@fe6e370a1c9c:/#
```

7. **docker stop**

   Usage: docker stop <container id>

   This command stops a running container

```
edureka@Manager-1: ~
edureka@Manager-1:~$ docker stop fe6e370a1c9c
fe6e370a1c9c
edureka@Manager-1:~$
```

8. **docker kill**

   Usage: docker kill <container id>

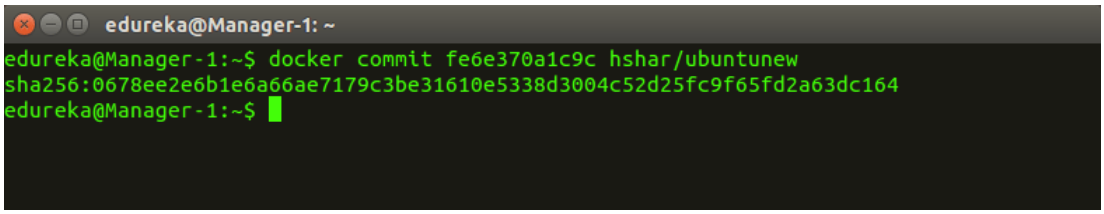
   This command kills the container by stopping its execution immediately. The difference between ‘docker kill’ and ‘docker stop’ is that ‘docker stop’ gives the container time to shutdown gracefully, in situations when it is taking too much time for getting the container to stop, one can opt to kill it

```
edureka@Manager-1: ~
edureka@Manager-1:~$ docker kill d611cbc3789c
d611cbc3789c
edureka@Manager-1:~$
```

9. **docker commit**

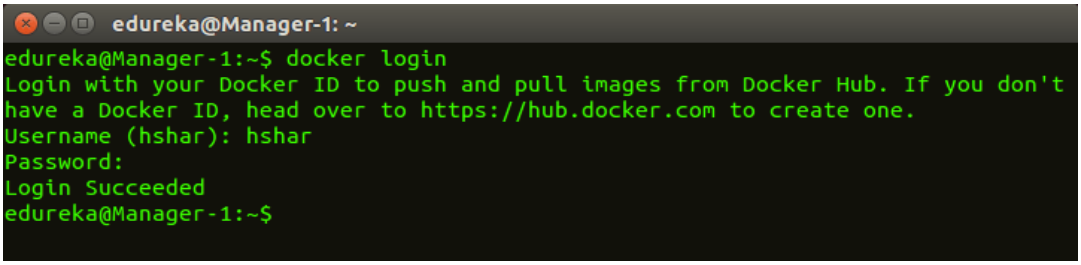
Usage: docker commit <conatainer id> <username/imagename>

This command creates a new image of an edited container on the local system



10. **docker login**

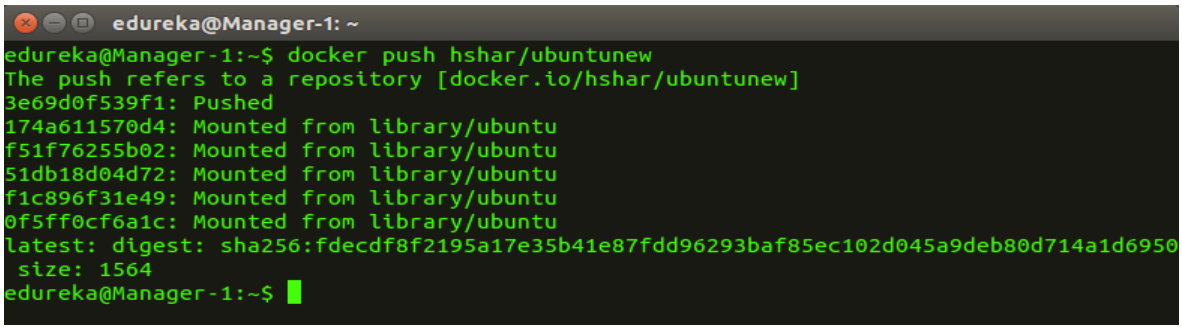
This command is used to login to the docker hub repository



11. **docker push**

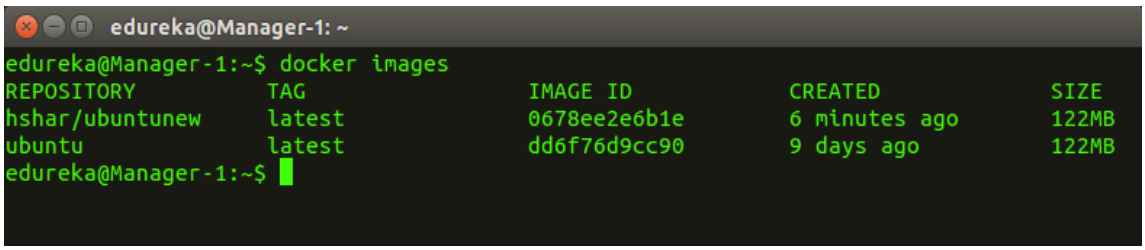
Usage: docker push <username/image name>

This command is used to push an image to the docker hub repository



12. **docker images**

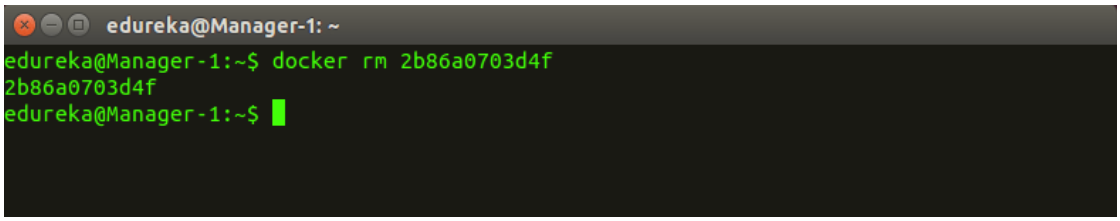
This command lists all the locally stored docker images



13. **docker rm**

Usage: docker rm <container id>

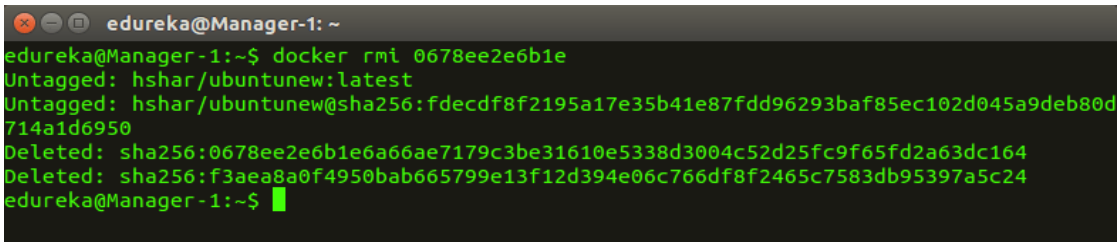
This command is used to delete a stopped container



14. **docker rmi**

    Usage: docker rmi <image-id>

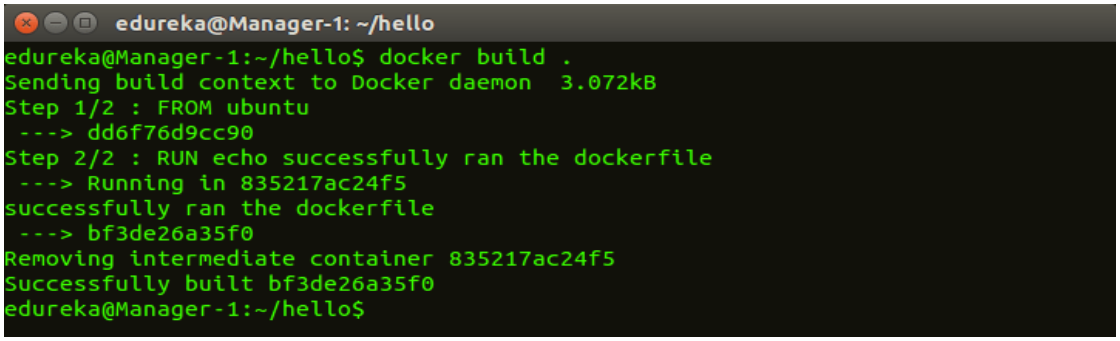
    This command is used to delete an image from local storage



15. **docker build**

    Usage: docker build <path to docker file>

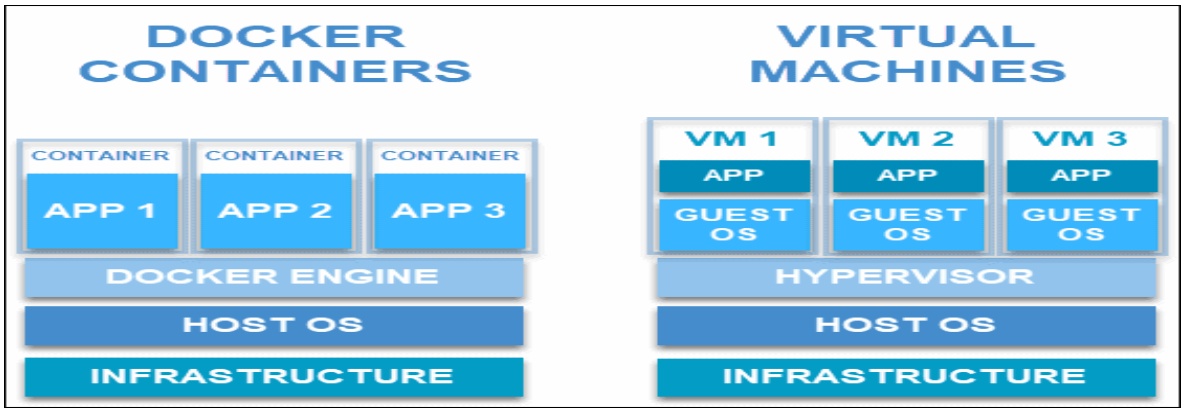
    This command is used to build an image from a specified docker file



Docker Image: The concept of Image and Container is like class and object, in which an object is an instance of a class, and a class is the blueprint of the object. Images are different in Virtual Machines and Docker. In virtual machines, images are just snapshots of running virtual machines at different points of time, but Docker images are a little bit different. The most important and major difference is that Docker images are immutable (they cannot be changed). It contains the source code, libraries, dependencies, tools, and other files needed for an application to run. In the real world, it happens a lot that software works on one computer but it does not work on others due to different environments. This issue is completely solved by

docker images and using this, the application will work the same on everyone's PC. Every developer on a team will have the exact same development instance. Each testing instance is exactly the same as the development instance. Your production instance is exactly the same as the testing instance. Also, developers around the world can share their Docker images on a platform called Docker HUB.

Docker Container: They are actually Docker Virtual Machines but are commonly called Docker Containers. If a Docker image is a map of the house, then a Docker container is an actual built house, or in other words, we can call it an instance of an image. As per the official website, a container is a runnable instance of an image. You can create, start, stop, move, or delete a container using Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state. An application runs using a cluster of containers which are self isolated from one another and also from the host machine where they are running. **Example:** If a backend application is running on a Docker container at port 8000 and you tried to access it from the host machine, you will not be able to access it, as containers are self-isolated and in that case you have to explicitly expose your application at a certain port and connect your machine port to that port.



Difference between Docker Image and Docker Container:

S.no.	Docker Image	Docker Container
1	It is a blueprint of the Container.	It is an instance of the Image.
2	Image is a logical entity.	Container is a real world entity.

3	Images are created only once.	Containers are created any number of times using an image.
4	Images are immutable. One cannot attach volumes and networks.	Containers change only if the old image is deleted and a new one is used to build the container. One can attach volumes, networks etc.
5	Images do not require computing resources to work.	Containers require computing resources to run as they run with a Docker Virtual Machine.
6	To make a docker image, you have to write script in a Dockerfile.	To make a container from an image, you have to run “docker run <image>” command
7	Docker Images are used to package up applications and pre-configured server environments.	Containers use server information and a file system provided by an image in order to operate.
8	Images can be shared on Docker Hub.	It makes no sense in sharing a running entity, always docker images are shared.
9	There is no such thing as a running state of a Docker Image.	Containers use RAM when created and in running state.
10	An image must not reference to any state to remove the image.	A container must be in a running state to remove it.
11	One cannot connect to the images as these images are like snapshots.	In this, one cannot connect them and execute the commands.
12	Sharing of Docker Images is possible.	Sharing of containers is not possible directly.
13	It has multiple read-only layers.	It has a single writable layer.
14	These image templates can exist in isolation.	These containers cannot exist without images.

Dockerfile

The Dockerfile is a simple text file that consists of instructions for generating a Docker image. Dockerfile will define the processes to quickly produce an image. While creating your application, you should create a Dockerfile in order since the Docker daemon runs all of the instructions from top to bottom.

Steps To Create a Dockerfile

- Create a file named Dockerfile.
- Add instructions in Dockerfile.
- Build Dockerfile to create an image.
- Run the image to create a container.

Important Dockerfile Keywords

1. FROM: Represents the base image(OS), which is the command that is executed first before any other commands.

Syntax:

```
FROM <ImageName>
```

Example: The base image will be ubuntu:19.04 Operating System

```
FROM ubuntu:19.04
```

2. COPY: The copy command is used to copy the file/folders to the image while building the image.

Syntax:

```
COPY <Source>   <Destination>
```

Example: Copying the .war file to the Tomcat webapps directory

```
COPY target/java-web-app.war  /usr/local/tomcat/webapps/java-web-app.war
```

3. ADD: While creating the image, we can download files from distant HTTP/HTTPS destinations using the ADD command.

Syntax:

```
ADD <URL>
```

Example: Try to download Jenkins using ADD command

```
ADD https://get.jenkins.io/war/2.397/jenkins.war
```

4. RUN: Scripts and commands are run with the RUN instruction. The execution of RUN commands or instructions will take place while you create an image on top of the prior layers (Image).

Syntax:

```
RUN < Command + ARGS>
```

Example:

```
RUN touch file
```

5. CMD: The main purpose of the CMD command is to start the process inside the container and it can be overridden.

Syntax:

```
CMD [command + args]
```

Example: Starting Jenkins

```
CMD ["java","-jar", "Jenkins.war"]
```

6. ENTRYPOINT: A container that will function as an executable is configured by ENTRYPOINT. When you start the Docker container, a command or script called ENTRYPOINT is executed. It can't be overridden.

Syntax:

```
ENTRYPOINT [command + args]
```

Example: Executing the **echo command.**

```
ENTRYPOINT ["echo","Welcome to GFG"]
```

7. MAINTAINER: By using the MAINTAINER command we can identify the author/owner of the Dockerfile and we can set our own author/owner for the image.

Syntax:

```
MAINTAINER <NAME>
```

Example: Setting the author for the image as a GFG author.

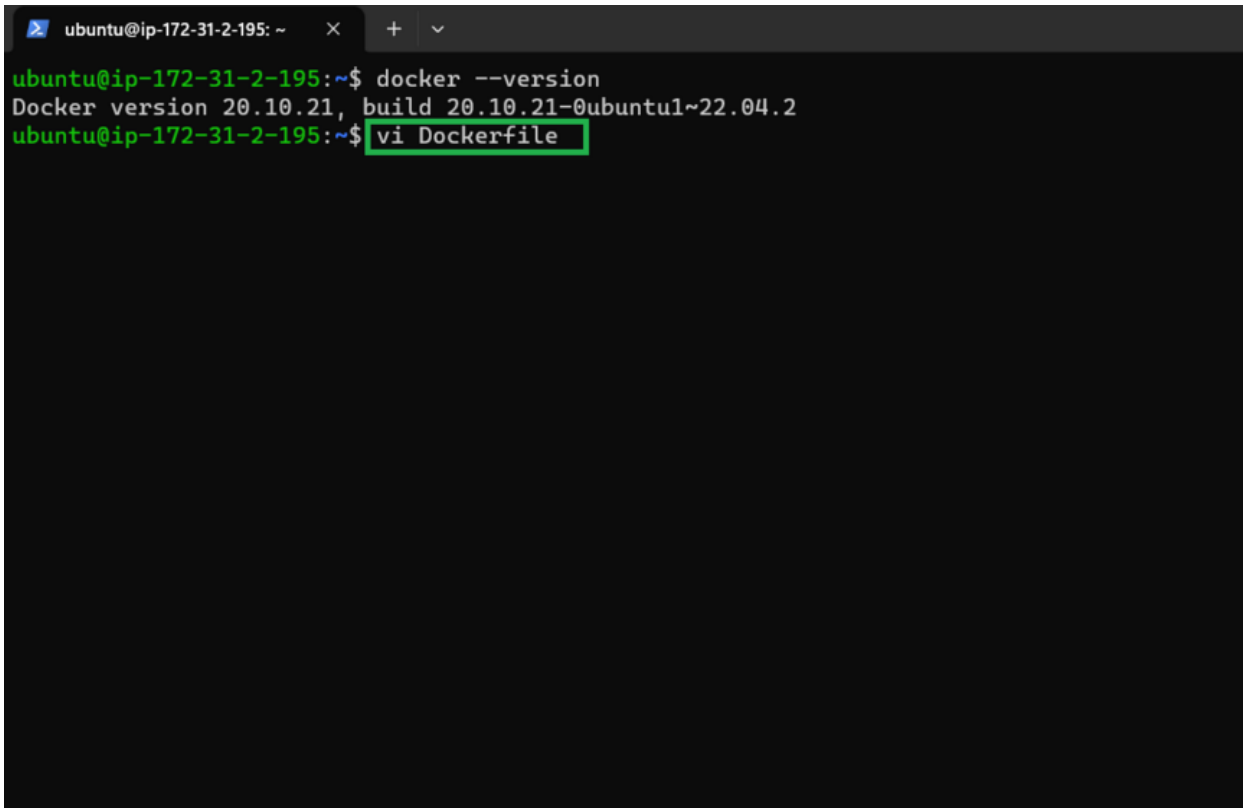
```
MAINTAINER  GFG author
```

Example 1: Steps To Create Dockerfile With Example(Jenkins)

In this example, we will write the Dockerfile for Jenkins and build an image by using Dockerfile which has been written for Jenkins and we will run it as a container.

Step 1: Open Docker and create a file with the name **Dockerfile.**

Step 2: Open the Dockerfile by using the vi editor and start writing the command that is required to build the Jenkins image.

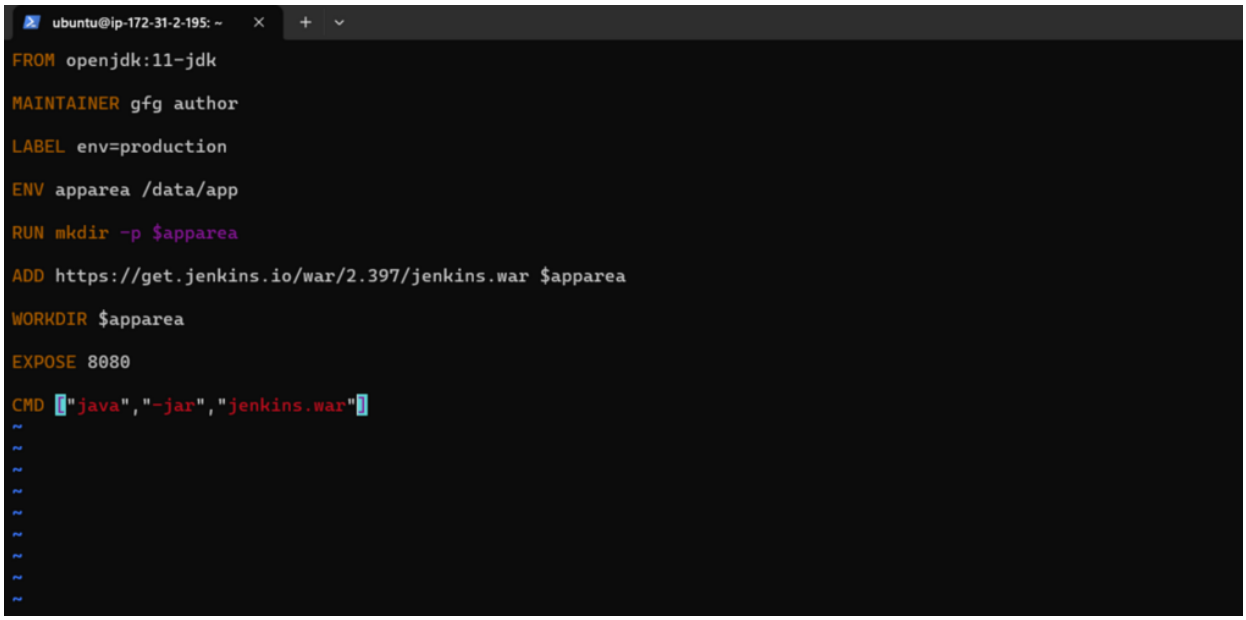


Dockerfile for Jenkins image

We used JDK as a base image because Jenkins's pre-requisite is JDK after that we added a command called **MAINTAINER** which indicates the author or owner of the docker file and we added the **ENV** variable where we set the path for the Jenkins and by using **RUN** command we are creating the path and by using **ADD** we are downloading the Jenkins and starting the **.war** file with the help of **CMD** command.

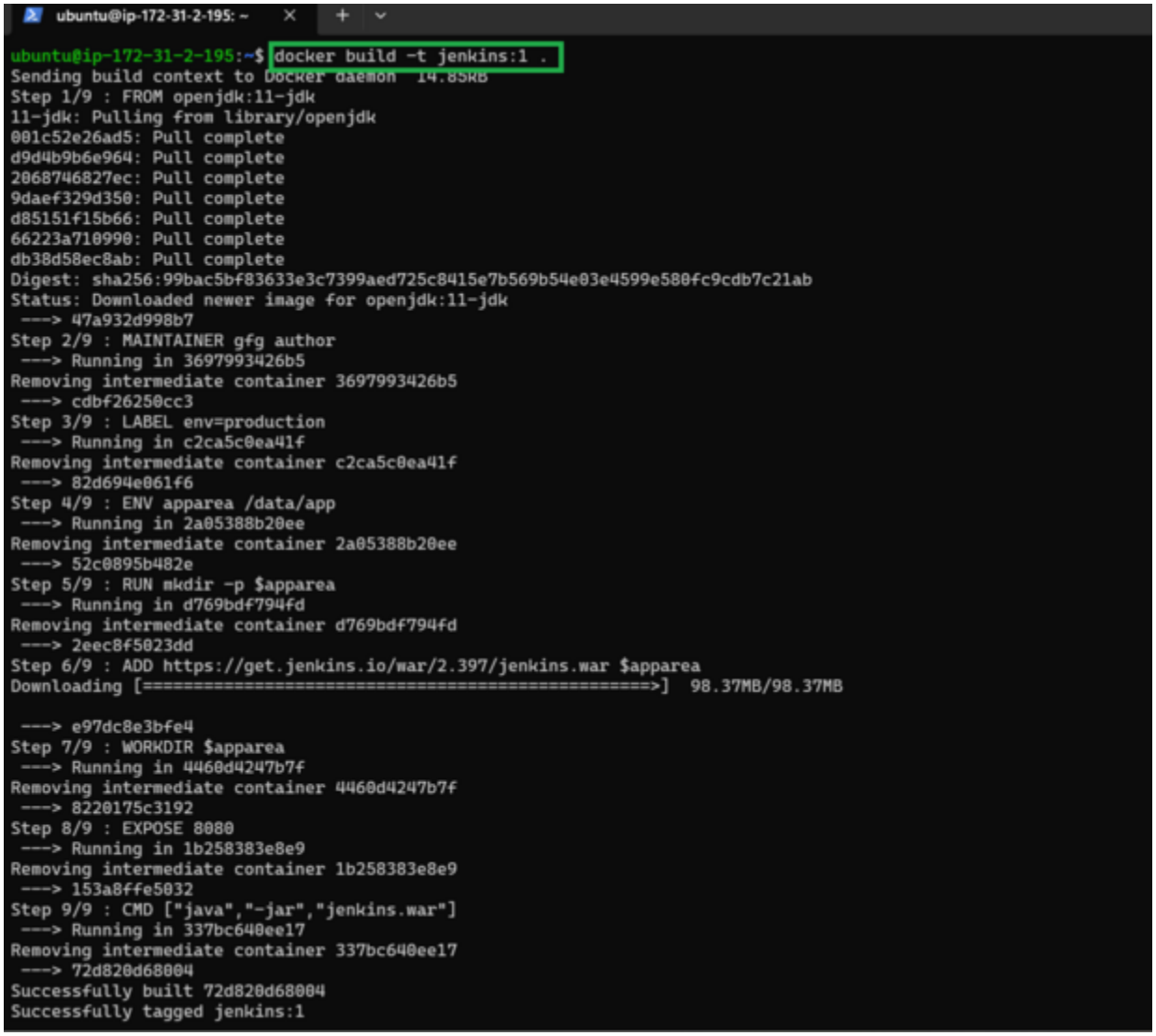
```
FROM openjdk:11-jdk

MAINTAINER GFG author

LABEL env=production

ENV apparea /data/app

RUN mkdir -p $apparea

ADD https://get.jenkins.io/war/2.397/jenkins.war $apparea
```

```
WORKDIR $apparea
EXPOSE 8080
CMD ["java","-jar","jenkins.war"]
```



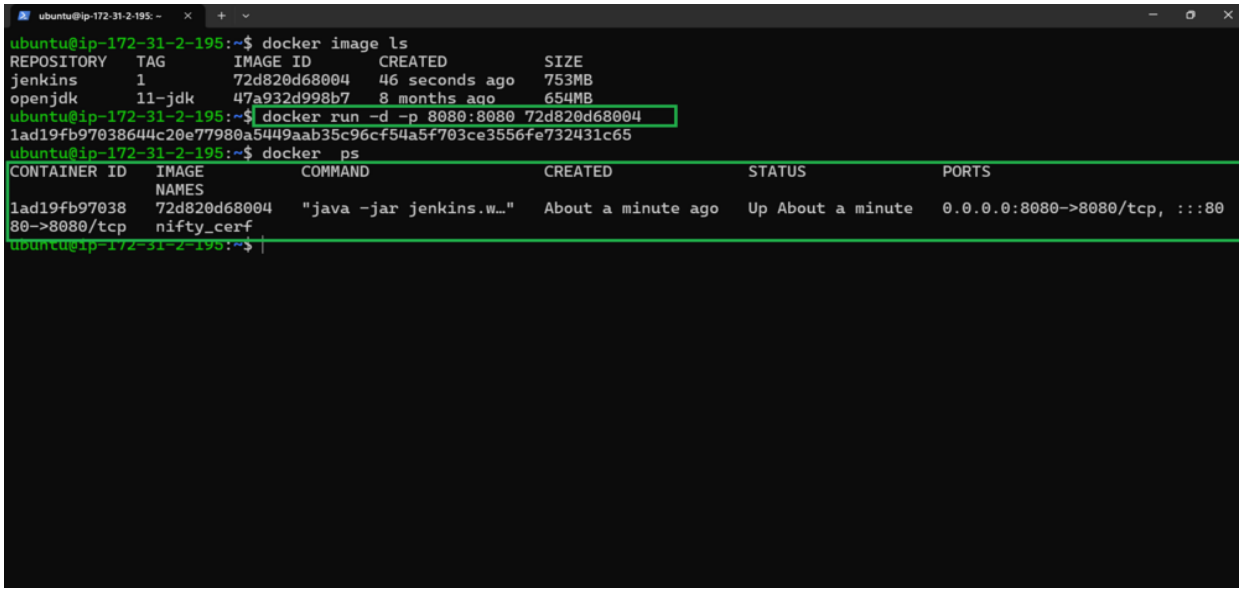
Step 3: Build the image by using the below command with the help of Dockerfile and give the necessary tags. and the dot(.) represents the current directory which is a path for Dockerfile.

```
docker build -t jenkins:1 .
```

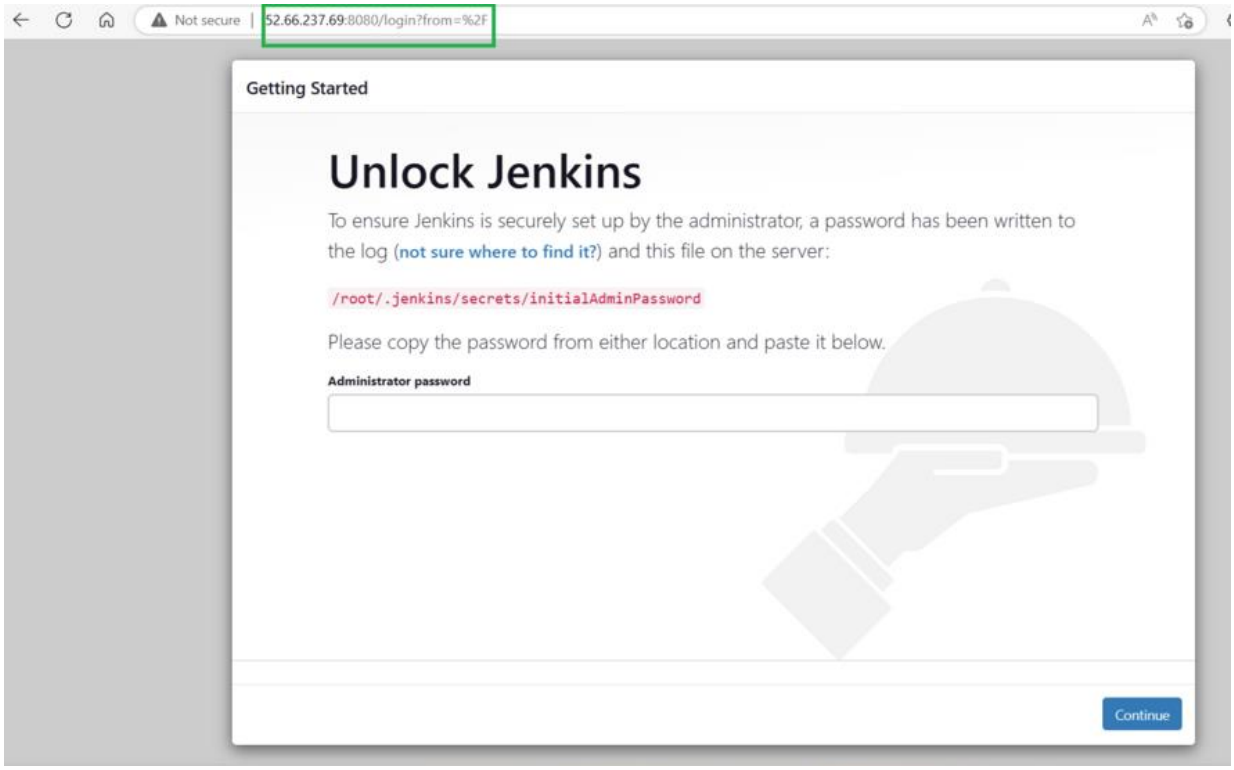


Step 4: Run the container with the help image ID or tag of the image by using the below command.

```
docker run -d -p 8080:8080 <Imagetag/ID>
```

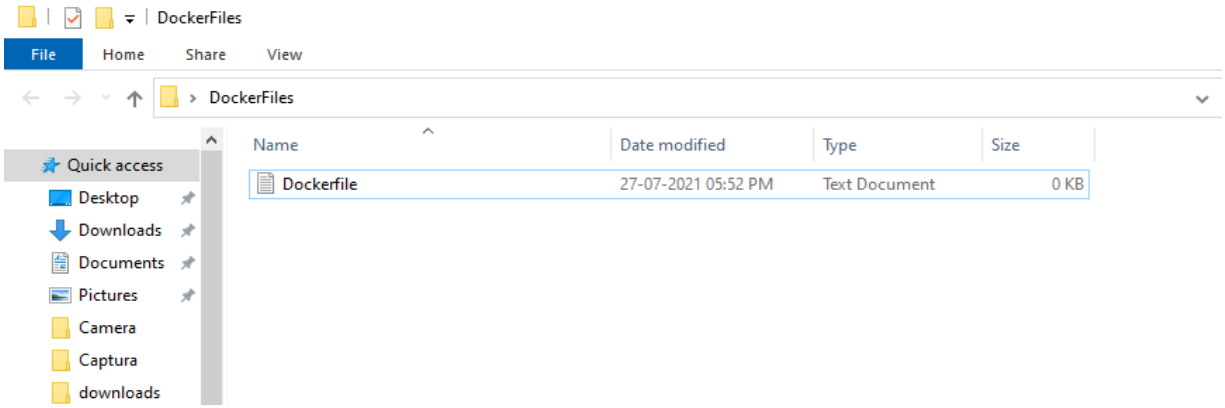


Step 5: Accesses the application (Jenkins) from the internet with the help of host port and hostIP (HostIP: Port)



Example 2: Steps To Create Dockerfile

Step 1: Create a file name called **"Dockerfile"**.By default when you run the docker build commands docker searches for a file named Dockerfile. However, it is not compulsory, you can also give some different names, and then you can tell the docker that this particular file is local but for now we will go with the Dockerfile.



Step 2: The very first instruction that a docker file starts with is FROM. Here you have to give a base image. So for example, if you want to get a base image from Ubuntu we will use FROM Ubuntu.

```
FROM ubuntu
```

Then the other instruction is you have to give a **MAINTAINER.** This is optional but it's a best practice that you give the maintainer of this image so that it is very easy to find out who is the maintainer and you can give your name and email as well. And if you want you can just give the email as well without giving the name. But here we are giving the entire thing.

```
MAINTAINER YOUR_NAME <YOUR_EMAIL_ID>
```

Next, we want to run something so we will say run any command we can use **RUN** and add the command that you need to run.

```
RUN apt-get update
```

And if you want to run something on the command line during container creation you can give **CMD** and inside square brackets, and we add the command. Here it is as shown below:

```
CMD ["echo", "Image Created"]
```

At this point the file will have the following commands:

```
FROM ubuntu

MAINTAINER YOUR_NAME <YOUR_EMAIL_ID>

RUN apt-get update

CMD ["echo", "Image Created"]
```

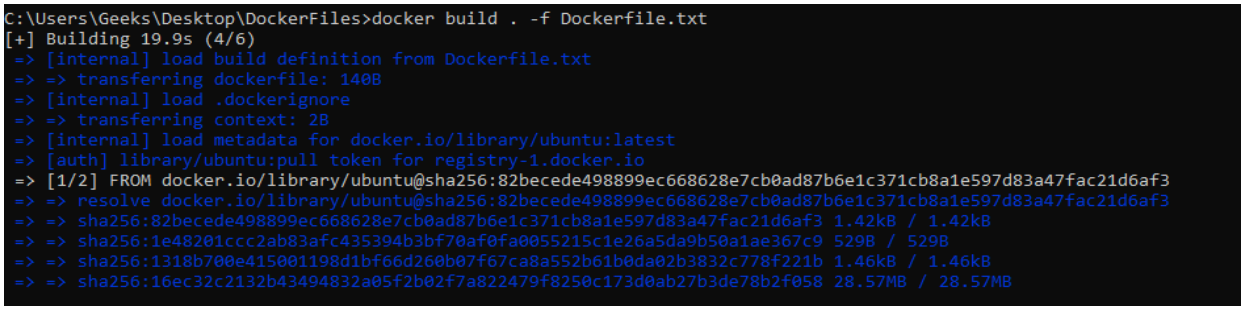
Step 3: Now we have to build the image so here are the commands you can use:

```
docker build /<FILE_LOCATION>
```

Or,

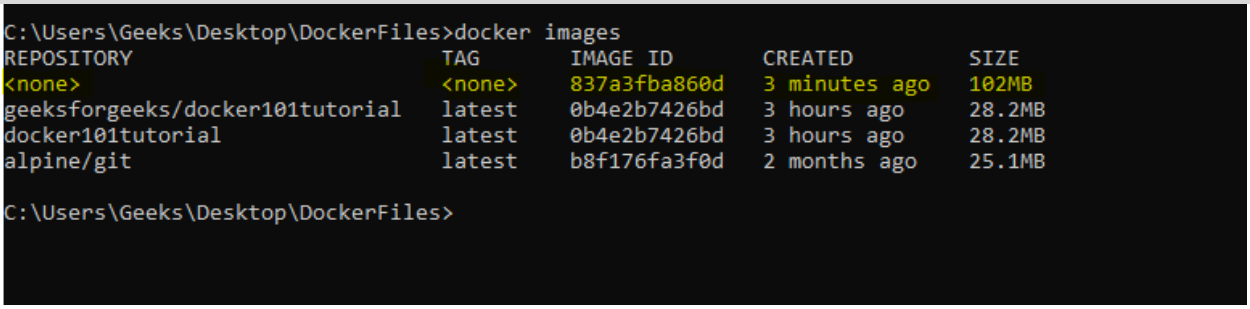
```
docker build . -f Dockerfile.txt
```

It says docker build and you have to give the location of your docker file. This will start building the image.



Command to list the images

```
docker image ls / docker images
```



Running a Container

Running of containers is managed with the Docker **run** command. To run a container in an interactive mode, first launch the Docker container.

```
sudo docker run –it centos /bin/bash
```

Listing of Containers

One can list all of the containers on the machine via the **docker ps** command. This command is used to return the currently running containers.

```
docker ps
```

Syntax

```
docker ps
```

Options

None

Return Value

The output will show the currently running containers.

Example

```
sudo docker ps
```

Output

When we run the above command, it will produce the following result −

```
demo@ubuntuserver:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
 STATUS              PORTS               NAMES
9f215ed0b0d3        centos:latest       "/bin/bash"         About a minute ago
 Up About a minute                       cocky_colden
demo@ubuntuserver:~$
```

Let’s see some more variations of the **docker ps** command.

```
docker ps -a
```

This command is used to list all of the containers on the system

Syntax

```
docker ps -a
```

Options

- **—a** − It tells the **docker ps** command to list all of the containers on the system.

Return Value

The output will show all containers.

Example

```
sudo docker ps -a
```

Output

When we run the above command, it will produce the following result −

docker history

With this command, you can see all the commands that were run with an image via a container.

Syntax

```
docker history ImageID
```

Options

- **ImageID** − This is the Image ID for which you want to see all the commands that were run against it.

Return Value

The output will show all the commands run against that image.

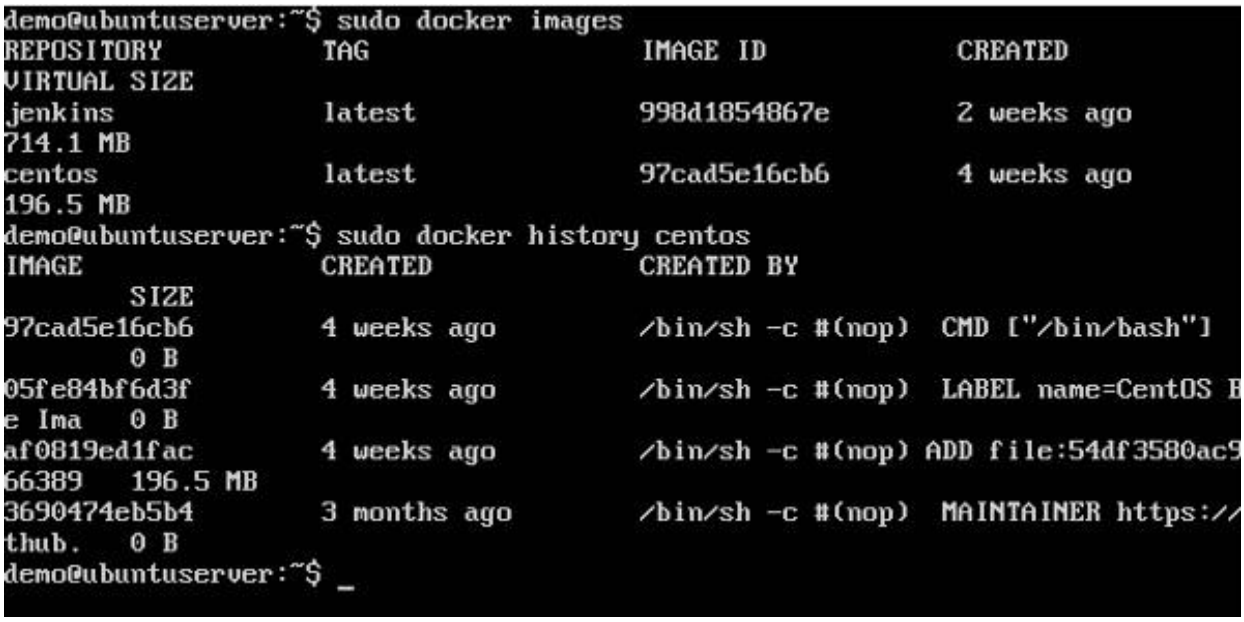
Example

```
sudo docker history centos
```

The above command will show all the commands that were run against the **centos** image.

Output

When we run the above command, it will produce the following result −



docker top

With this command, you can see the top processes within a container.

Syntax

```
docker top ContainerID
```

Options

- **ContainerID** − This is the Container ID for which you want to see the top processes.

Return Value

The output will show the top-level processes within a container.

Example

```
sudo docker top 9f215ed0b0d3
```

The above command will show the top-level processes within a container.

Output

When we run the above command, it will produce the following result −

```
demo@ubuntuserver:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
9f215ed0b0d3        centos:latest       "/bin/bash"         12 minutes ago
Up 12 minutes                           cocky_colden
demo@ubuntuserver:~$ sudo docker top 9f215ed0b0d3
UID                 PID                 PPID                C
STIME               TTY                 TIME                CMD
root                1606                678                 0
18:13               pts/0               00:00:00            /bin/bash
demo@ubuntuserver:~$
```

docker stop

This command is used to stop a running container.

Syntax

```
docker stop ContainerID
```

Options

- **ContainerID** − This is the Container ID which needs to be stopped.

Return Value

The output will give the ID of the stopped container.

Example

```
sudo docker stop 9f215ed0b0d3
```

The above command will stop the Docker container **9f215ed0b0d3**.

Output

When we run the above command, it will produce the following result −

```
demo@ubuntuserver:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
9f215ed0b0d3        centos:latest       "/bin/bash"         22 minutes ago
Up 22 minutes                           cocky_colden
demo@ubuntuserver:~$ sudo docker stop 9f215ed0b0d3
9f215ed0b0d3
demo@ubuntuserver:~$ sudo docker rm 9f215ed0b0d3
9f215ed0b0d3
demo@ubuntuserver:~$ _
```

docker rm

This command is used to delete a container.

Syntax

```
docker rm ContainerID
```

Options

- **ContainerID** − This is the Container ID which needs to be removed.

Return Value

The output will give the ID of the removed container.

Example

```
sudo docker rm 9f215ed0b0d3
```

The above command will remove the Docker container **9f215ed0b0d3**.

Output

When we run the above command, it will produce the following result −

```
demo@ubuntuserver:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
9f215ed0b0d3        centos:latest       "/bin/bash"         22 minutes ago
Up 22 minutes                           cocky_colden
demo@ubuntuserver:~$ sudo docker stop 9f215ed0b0d3
9f215ed0b0d3
demo@ubuntuserver:~$ sudo docker rm 9f215ed0b0d3
9f215ed0b0d3
demo@ubuntuserver:~$ _
```

docker stats

This command is used to provide the statistics of a running container.

Syntax

```
docker stats ContainerID
```

Options

- **ContainerID** − This is the Container ID for which the stats need to be provided.

Return Value

The output will show the CPU and Memory utilization of the Container.

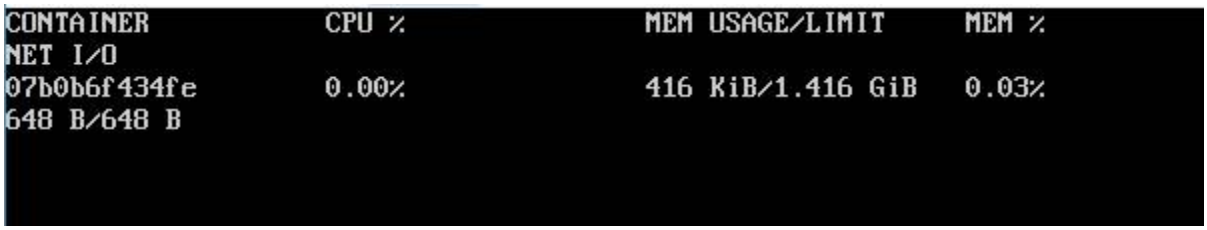
Example

```
sudo docker stats 9f215ed0b0d3
```

The above command will provide CPU and memory utilization of the Container **9f215ed0b0d3**.

Output

When we run the above command, it will produce the following result −



docker attach

This command is used to attach to a running container.

Syntax

```
docker attach ContainerID
```

Options

- **ContainerID** − This is the Container ID to which you need to attach.

Return Value

None

Example

```
sudo docker attach 07b0b6f434fe
```

The above command will attach to the Docker container **07b0b6f434fe**.

Output

When we run the above command, it will produce the following result −

```
demo@ubuntuserver:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
07b0b6f434fe        centos:latest       "/bin/bash"         3 minutes ago
Up 3 minutes                            cocky_pare
demo@ubuntuserver:~$ sudo docker attach 07b0b6f434fe

[root@07b0b6f434fe /]# _
```

Once you have attached to the Docker container, you can run the above command to see the process utilization in that Docker container.

```
top - 15:24:06 up  2:06,  0 users,  load average: 0.00, 0.01, 0.02
Tasks:   2 total,   1 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.
KiB Mem :  1484856 total,  1057152 free,    52368 used,   375336 buff/cache
KiB Swap:  1519612 total,  1519612 free,        0 used.  1403868 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
    1 root      20   0   11784   2992   2644 S  0.0  0.2   0:00.01 bash
   15 root      20   0   51864   3772   3272 R  0.0  0.3   0:00.00 top
```

docker pause

This command is used to pause the processes in a running container.

Syntax

```
docker pause ContainerID
```

Options

- **ContainerID** − This is the Container ID to which you need to pause the processes in the container.

Return Value

The ContainerID of the paused container.

Example

```
sudo docker pause 07b0b6f434fe
```

The above command will pause the processes in a running container **07b0b6f434fe**.

Output

When we run the above command, it will produce the following result −

```
demo@ubuntuserver:~$ sudo docker ps
[sudo] password for demo:
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
07b0b6f434fe        centos:latest       "/bin/bash"         18 minutes ago
Up 18 minutes                           cocky_pare
demo@ubuntuserver:~$ sudo docker pause 07b0b6f434fe
07b0b6f434fe
demo@ubuntuserver:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS                  PORTS               NAMES
07b0b6f434fe        centos:latest       "/bin/bash"         19 minutes ago
Up 19 minutes (Paused)                      cocky_pare
demo@ubuntuserver:~$ _
```

docker unpause

This command is used to **unpause** the processes in a running container.

Syntax

```
docker unpause ContainerID
```

Options

- **ContainerID** − This is the Container ID to which you need to unpause the processes in the container.

Return Value

The ContainerID of the running container.

Example

```
sudo docker unpause 07b0b6f434fe
```

The above command will unpause the processes in a running container: 07b0b6f434fe

Output

When we run the above command, it will produce the following result −

```
demo@ubuntuserver:~$ sudo docker unpause 07b0b6f434fe
07b0b6f434fe
demo@ubuntuserver:~$
```

docker kill

This command is used to kill the processes in a running container.

Syntax

```
docker kill ContainerID
```

Options

- **ContainerID** − This is the Container ID to which you need to kill the processes in the container.

Return Value

The ContainerID of the running container.

Example

```
sudo docker kill 07b0b6f434fe
```

The above command will kill the processes in the running container **07b0b6f434fe**.

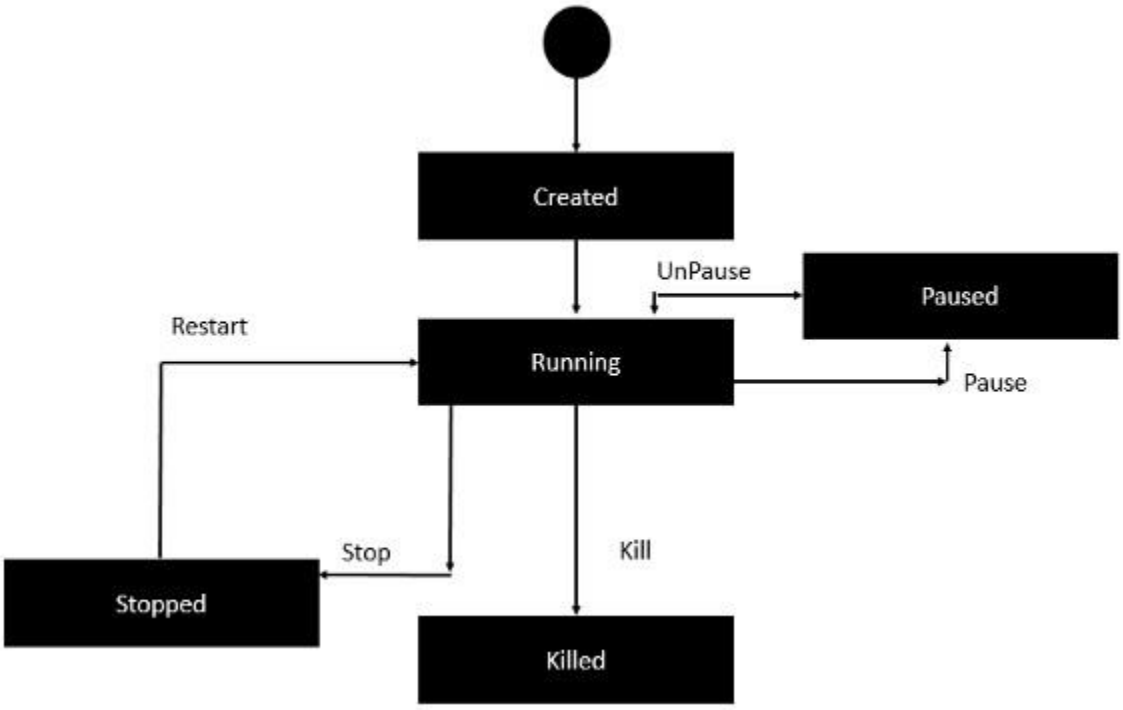
Output

When we run the above command, it will produce the following result −

```
demo@ubuntuserver:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
07b0b6f434fe        centos:latest       "/bin/bash"         23 minutes ago
Up 23 minutes                           cocky_pare
demo@ubuntuserver:~$ sudo docker kill 07b0b6f434fe
07b0b6f434fe
demo@ubuntuserver:~$
```

Docker – Container Lifecycle

The following illustration explains the entire lifecycle of a Docker container.

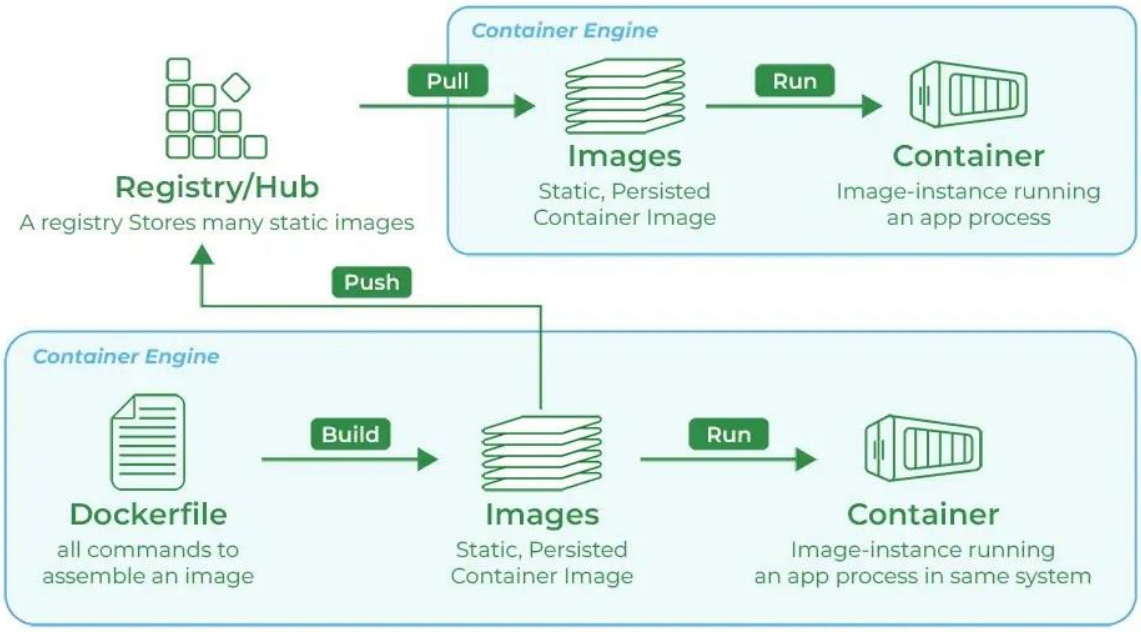


- Initially, the Docker container will be in the **created** state.
- Then the Docker container goes into the running state when the Docker **run** command is used.
- The Docker **kill** command is used to kill an existing Docker container.
- The Docker **pause** command is used to pause an existing Docker container.
- The Docker **stop** command is used to pause an existing Docker container.
- The Docker **run** command is used to put a container back from a **stopped** state to a **running** state.

What is Docker Hub?

Docker Hub is a repository service and it is a cloud-based service where people push their Docker Container Images and also pull the Docker Container Images from the **Docker Hub** anytime or anywhere via the internet. It provides features such as you can push your images as private or public. Mainly DevOps team uses the Docker Hub. It is an open-source tool and freely available for all operating systems. It is like storage where we store the images and pull the images when it is required. When a person wants to push/pull images from the Docker Hub they must have a basic knowledge of Docker.

When a Developer team wants to share the project with all dependencies for testing then the developer can push their code on **Docker Hub** with all dependencies. Firstly create the **Images** and push the Image on Docker Hub. After that, the testing team will pull the same image from the Docker Hub eliminating the need for any type of file, software, or plugins for running the Image because the Developer team shares the image with all dependencies.



Docker Hub Features

- Storage, management, and sharing of images with others are made simple via Docker Hub.
- Docker Hub runs the necessary security checks on our images and generates a full report on any security flaws.
- Docker Hub can automate the processes like Continuous deployment and Continuous testing by triggering the Webhooks when the new image is pushed into Docker Hub.
- With the help of Docker Hub, we can manage the permission for the users, teams, and organizations.
- We can integrate Docker Hub into our tools like GitHub, Jenkins which makes workflows easy

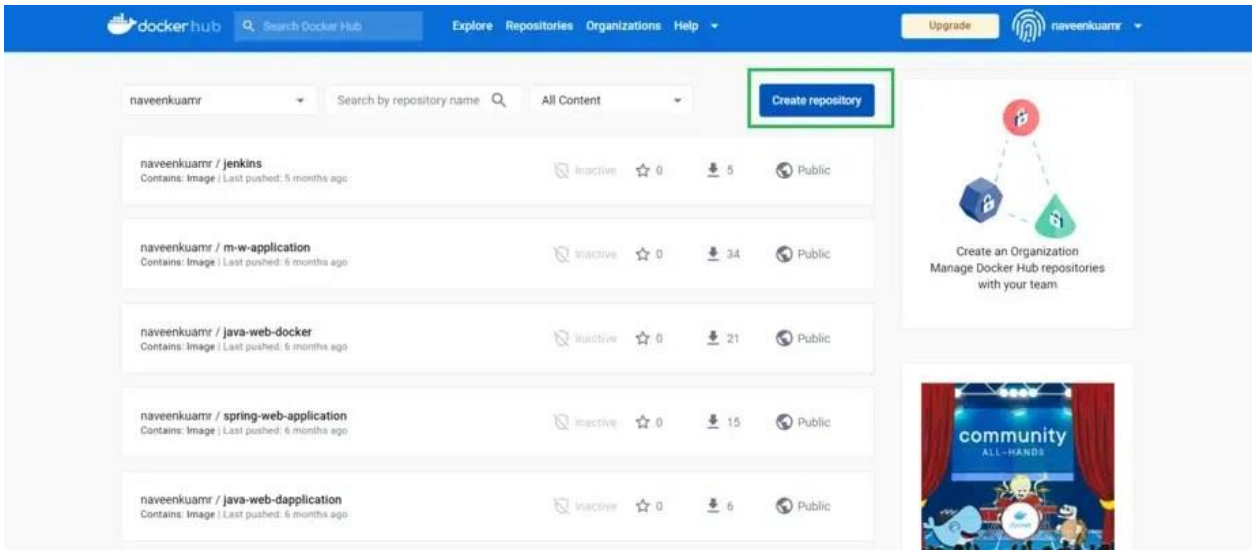
Advantages of Docker Hub

- Docker Container Images are light in weight.

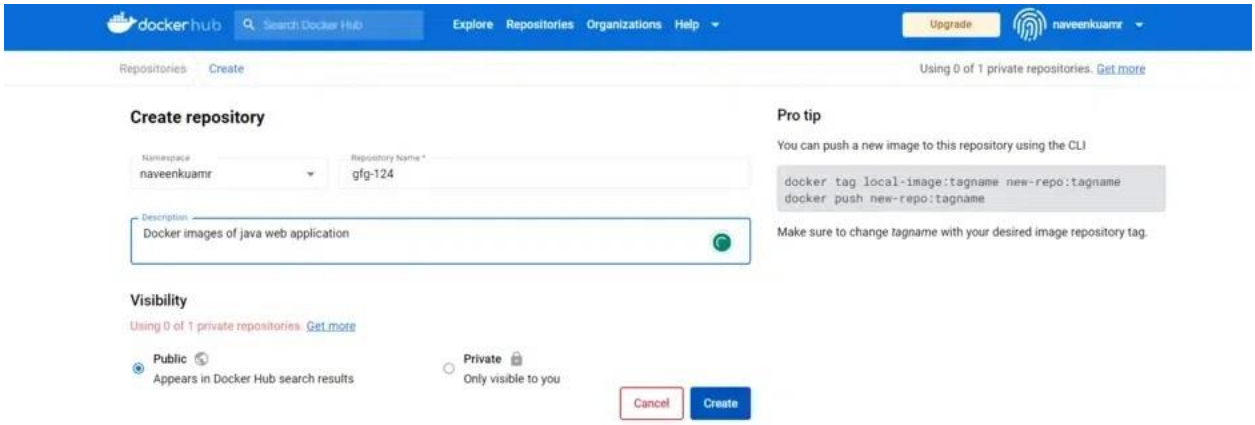
- We can push the images within a minute and with help of a command.
- It is a secure method and also provides a feature like pushing the private image or public image.
- Docker hub plays a very important role in industries as it becomes more popular day by day and it acts as a bridge between the developer team and the testing team.
- If a person wants to share their code, software any type of file for public use, you can just make the images public on the docker hub.

Creating First Repository in Docker Hub Using GUI

Step 1: We must open Docker Hub first, then select Create Repository.

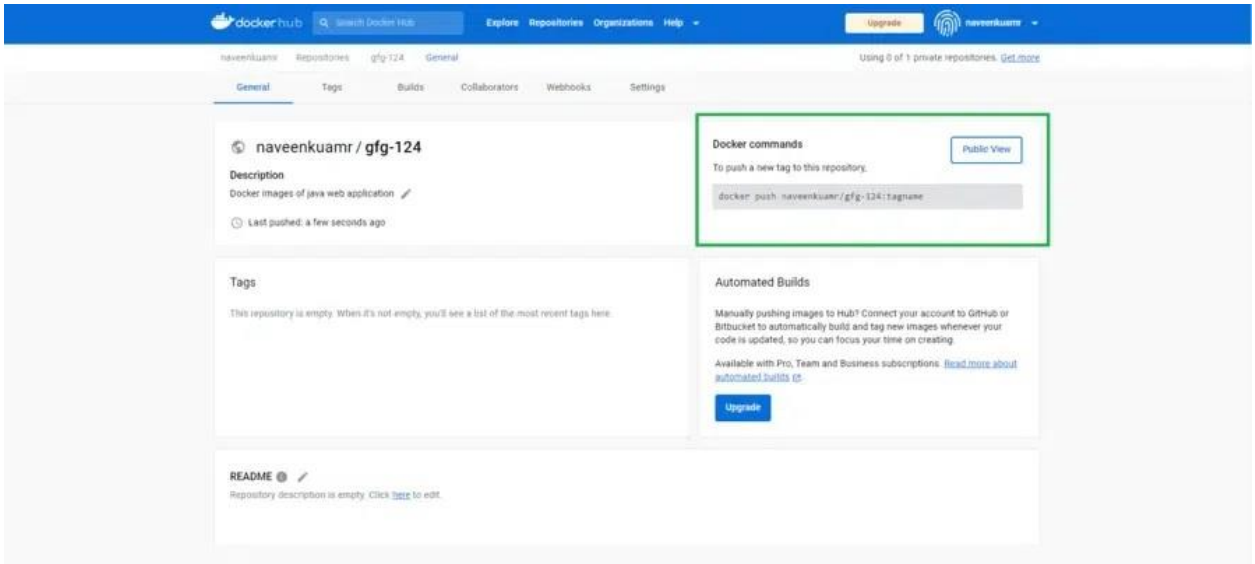


Step 2: After that, we will be taken to a screen for configuring the repository, where we must choose the namespace, repository name, and optional description. In the visibility area, as indicated in the picture, there are two options: Public and Private. We can choose any of them depending on the type of organization you are in. If you chose Public, everyone will be able to push-pull and use the image because it will be accessible to everyone. If you select the private option, only those with access to that image can view and utilize it. it.



Step 3: At finally repository is created with the help of the Docker Commands we can push or pull the image.

```
docker push <your-username>/my-testprivate-repo>.
```



How To Push or Pull Images from Docker Hub?

To get started with Docker Hub you should be able to get familiar with the below two commands:

1. Push Command

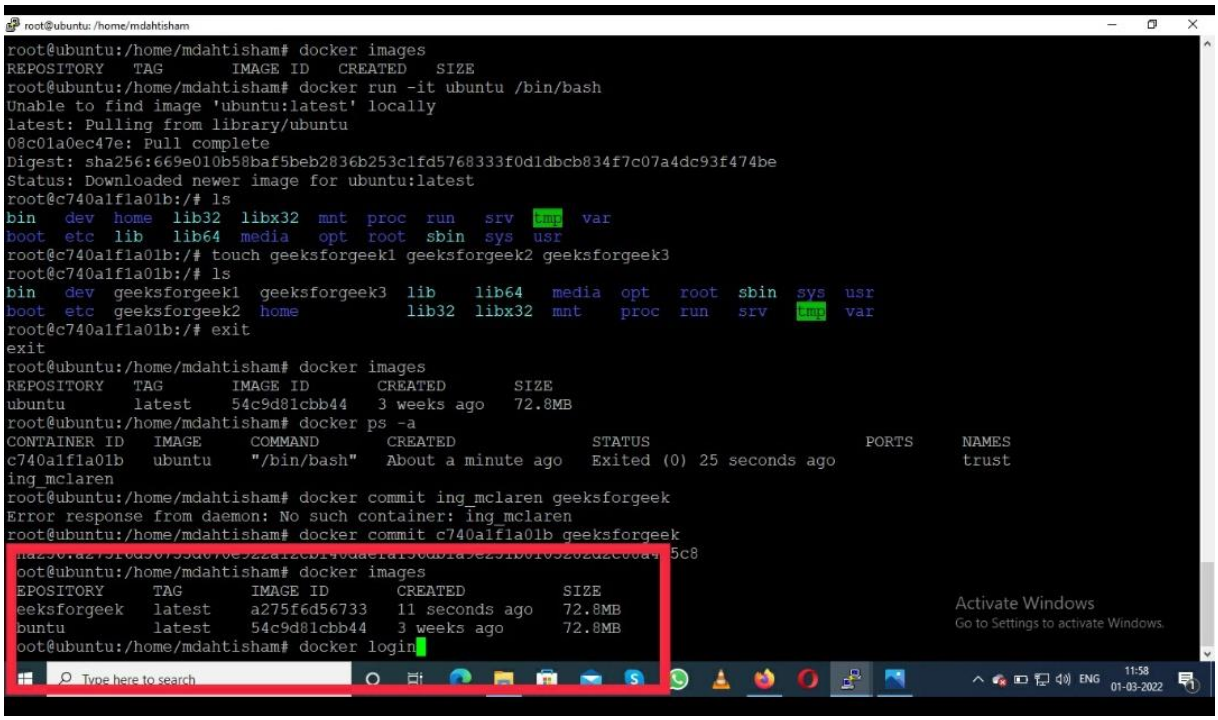
This command as the name suggests itself is used to pushing a docker image onto the docker hub.

Implementation

Follow this example to get an idea of the push command:

- Open Docker in your system.
- Locate the Images that you want to push using the below command:

```
# docker images
```



The above command will list all the images on your system.

Step 1: Go to the browser and search *hub.docker.com.*

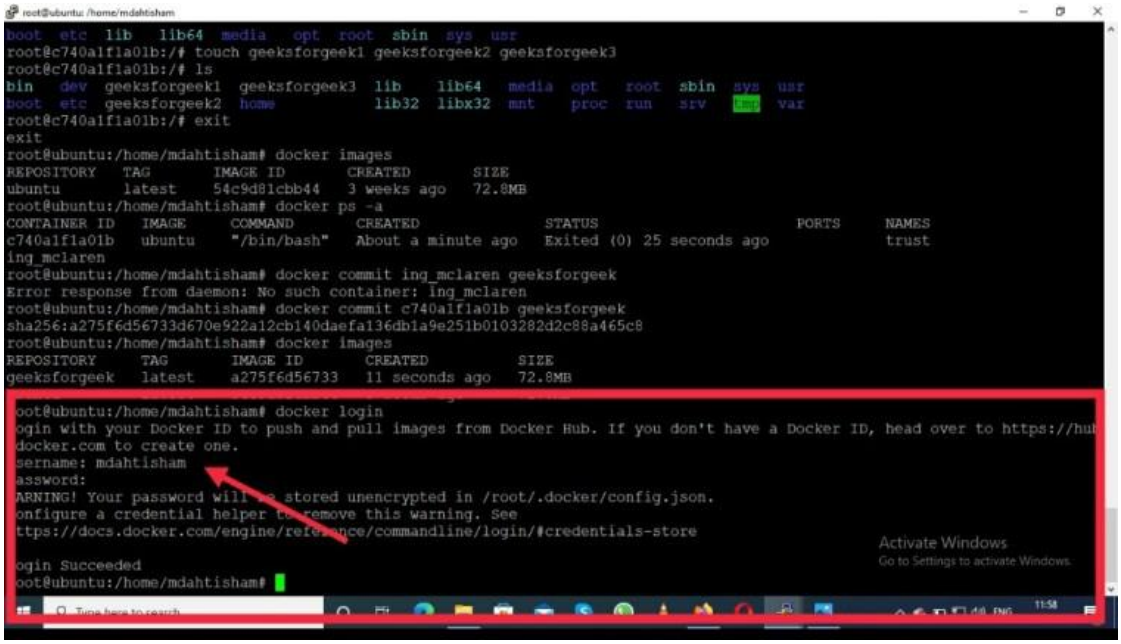
Step 2: Sign up on the docker hub if you do not have a docker hub account, after login on to docker hub.

Step 3: Back to the docker terminal and execute the below command:

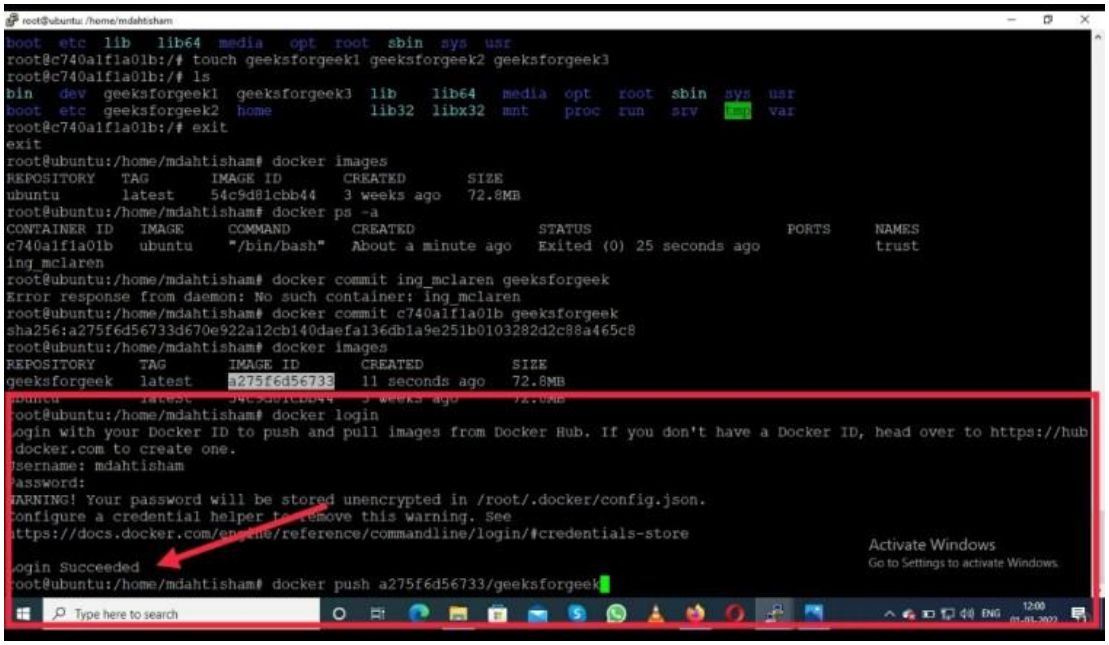
```
# docker login
```

Step 4: Then give your credential and type in your docker hub username or password.

- username
- password



Step 5: After that hit the Enter key you will see login success on your screen.



Step 6: Then type the tag images name, docker hub username, and give the name it appears on the docker hub using the below command:

```
# docker tag geeksforgeek mdahtisham/geeksimage
```

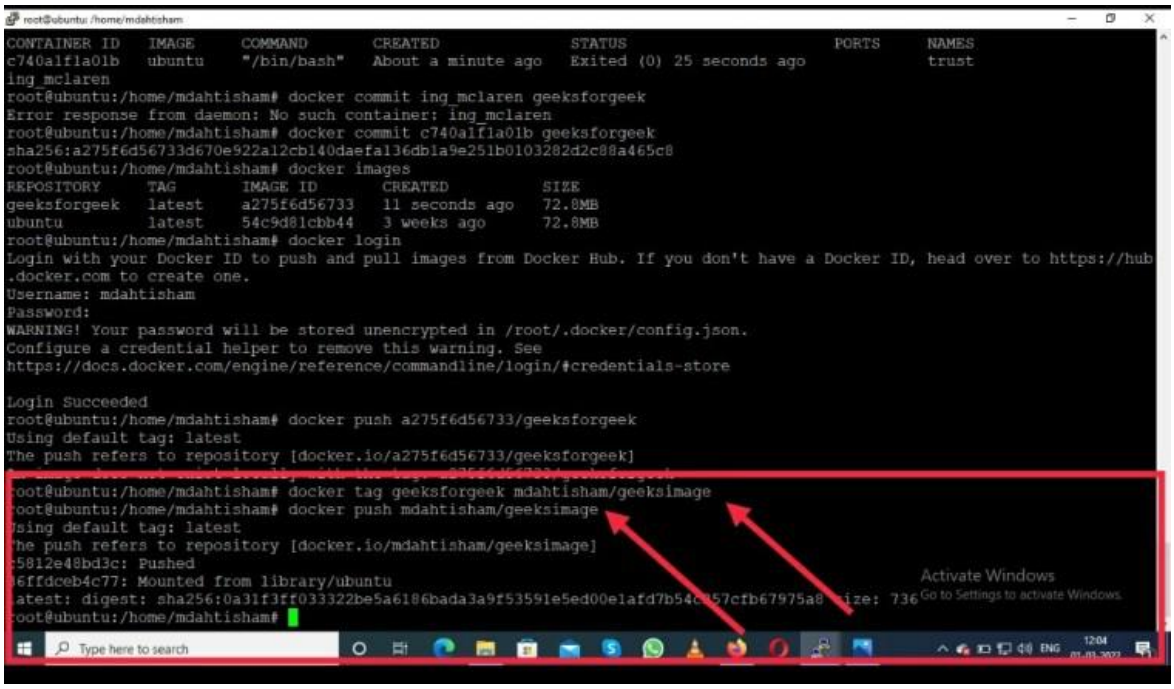
geeksforgeek - Image name

mdahtisham - Docker hub username

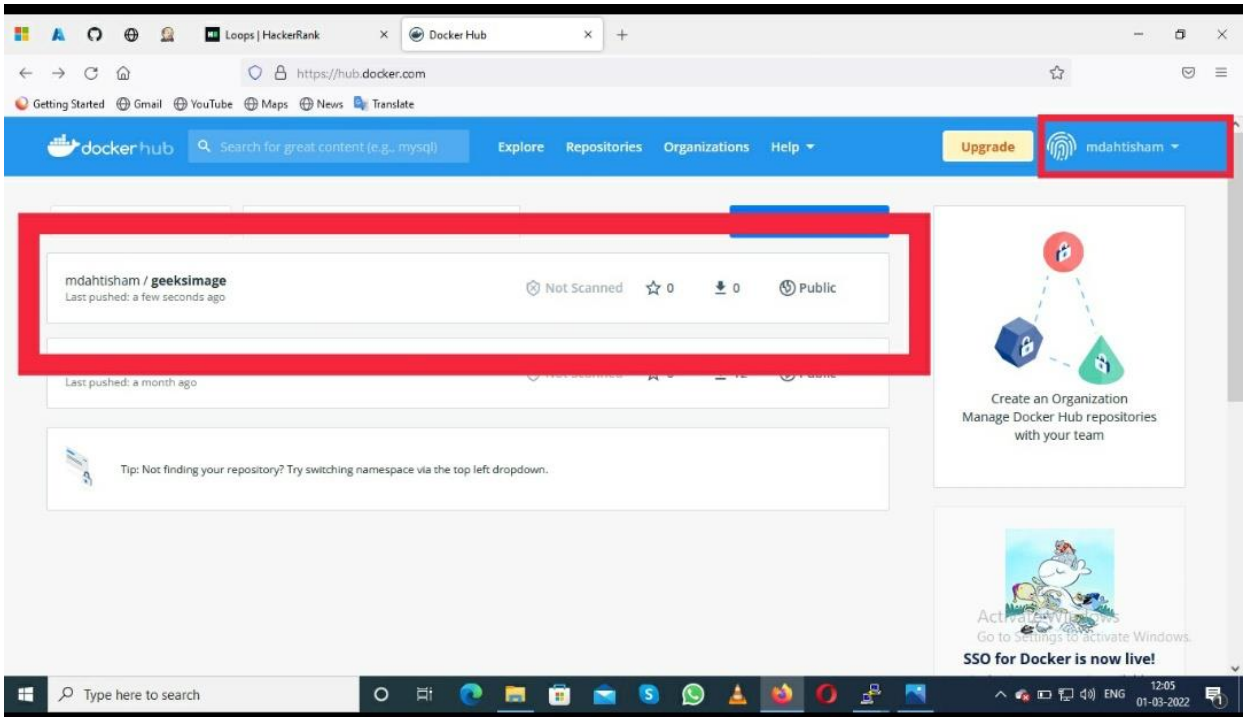
geeksimage - With this name Image will appear on the docker hub

Step 8: Now push your image using the below command:

```
# docker push mdahtisham/geeksimage
```



Note: Below you can see the Docker Image successfully pushed on the docker hub: mdahtisham/geeksimage



2. Pull Command

The pull command is used to get an image from the Docker Hub.

Implementation:

Follow the example to get an overview of the pull command in Docker:

Step 1: Now you can search the image using the below command in docker as follows:

```
# docker search imagename
```

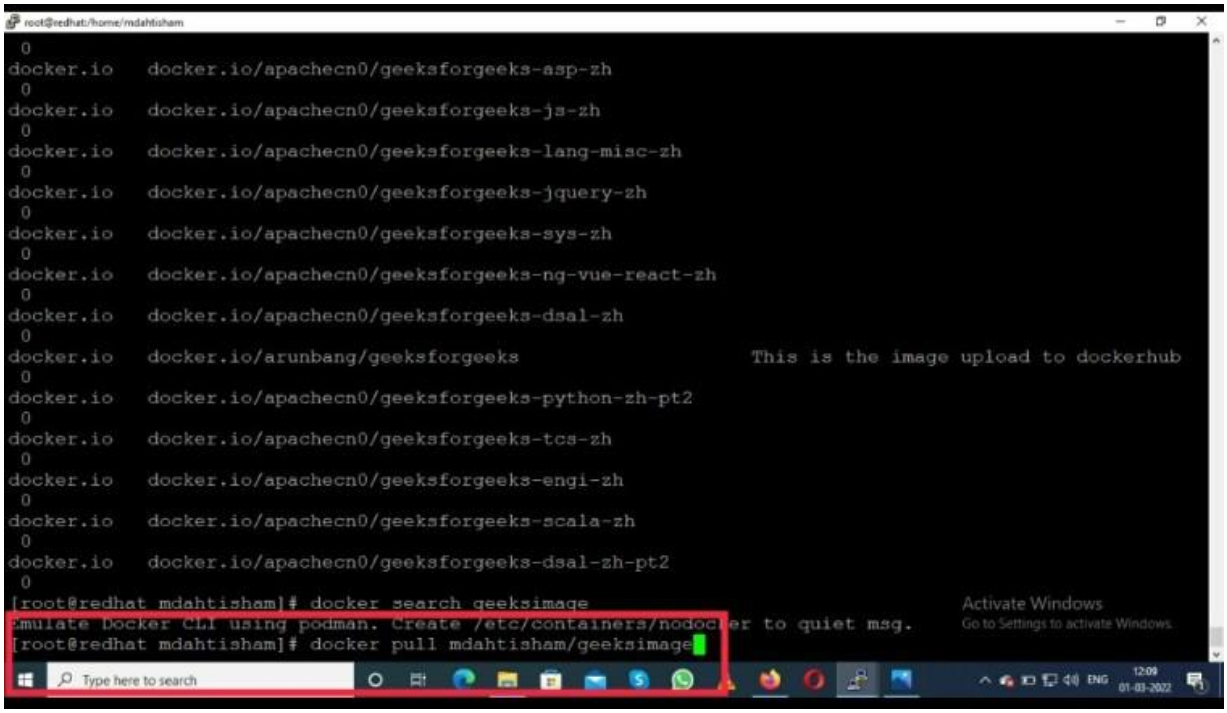
One can see all images on your screen if available images with this name.One can also pull the images if one knows the exact name

Step 2: Now pull the image see the below command.

```
# docker pull mdahtisham/geeksimage
```

mdahtisham - Docker Hub username

geeksimage - With this name Image will appear on the docker hub



Step 3: Now check for the pulled image using the below command as follows:

```
# docker images
```