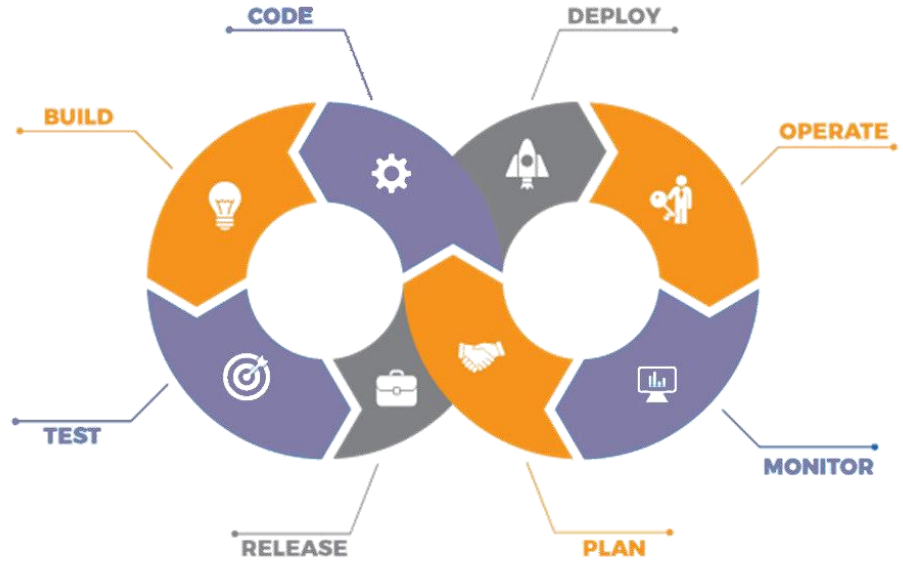


# Containerization Using Docker



# Agenda

01

What is  
Virtualization?

02

What is  
Containerization?

03

Containerization  
Tools

04

Components of  
Docker

05

Installing Docker

06

Common Docker  
Commands

07

Creating a Docker Hub  
Account

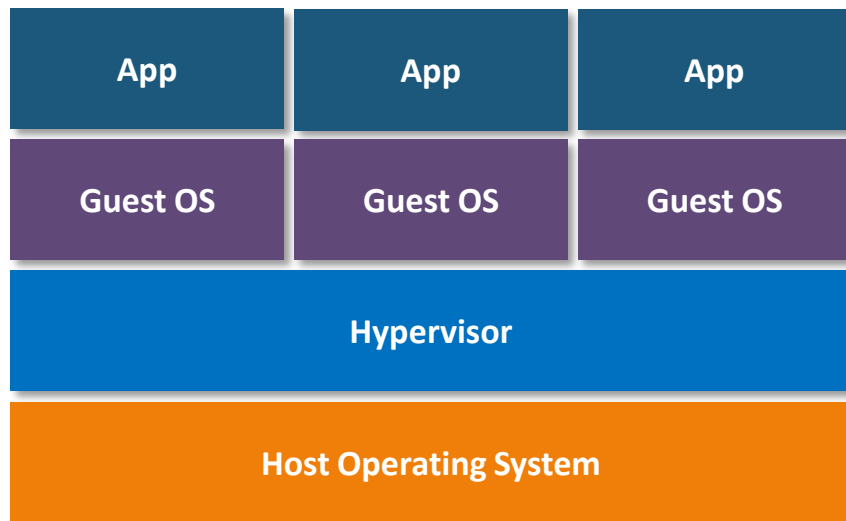
08

Introduction to  
Dockerfile

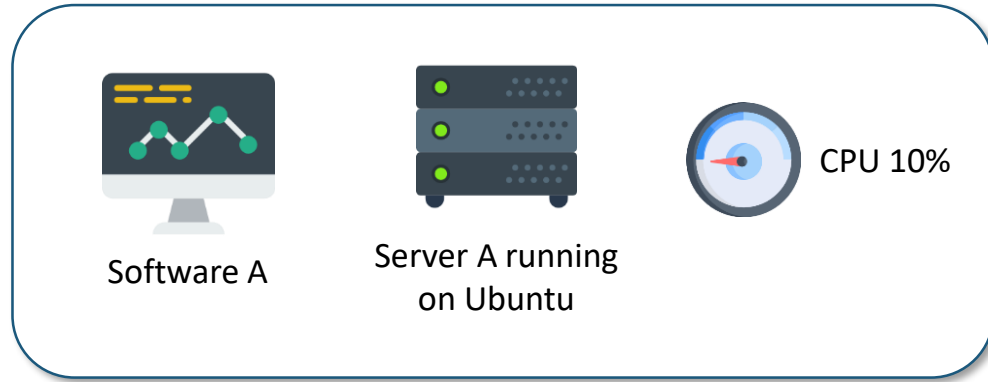
What is Virtualization?

# What is Virtualization?

Virtualization is the process of running multiple virtual systems or resources on top of a single physical machine. These resources could be a storage device, network or even an operating system!

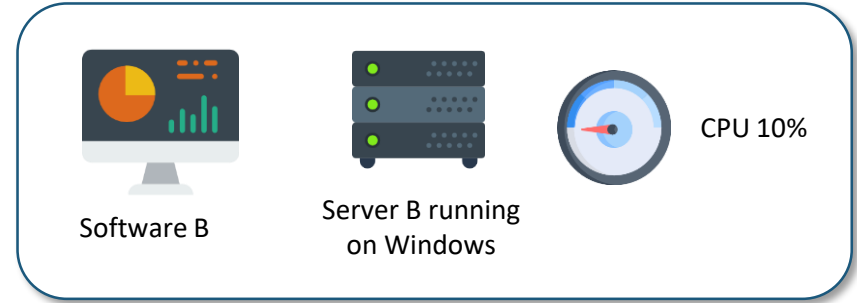
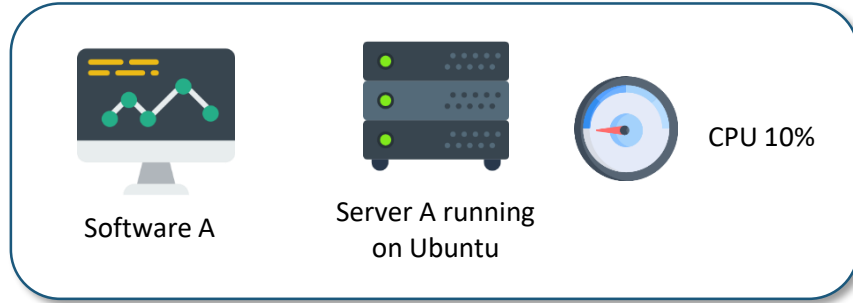


# Problems before Virtualization



Imagine Software A running on Server A which has Ubuntu running on it. This software can only run in the Ubuntu environment.

# Problems before Virtualization



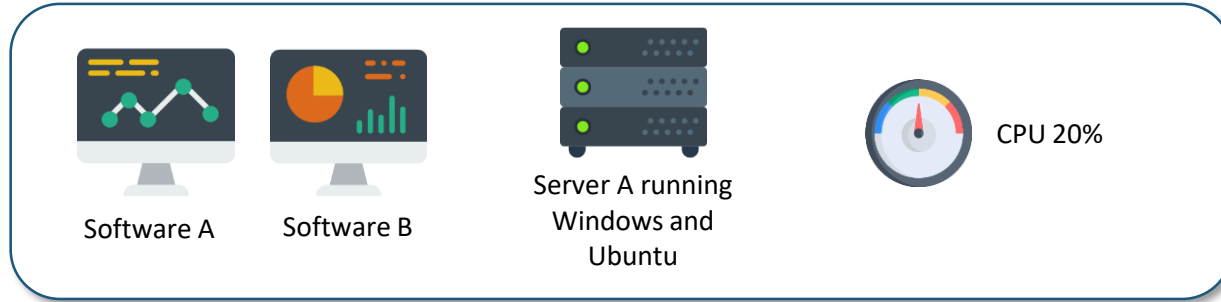
Some time later, we needed Software B which can only run on Windows. Therefore, we had to buy and run a Server B which had windows running on it. The software took only 10% of the CPU resources.

# Problems before Virtualization



- ❌ Buying servers was expensive.
- ❌ Resources were not being utilized at their full potential.
- ❌ The process of getting any software up and running was time consuming.
- ❌ Disaster recovery was difficult.

# After Virtualization



Windows and Ubuntu OS now are running on the same server in parallel using the Virtualization technology. This accounts for better CPU utilization and cost savings!



# Advantages of Virtualization

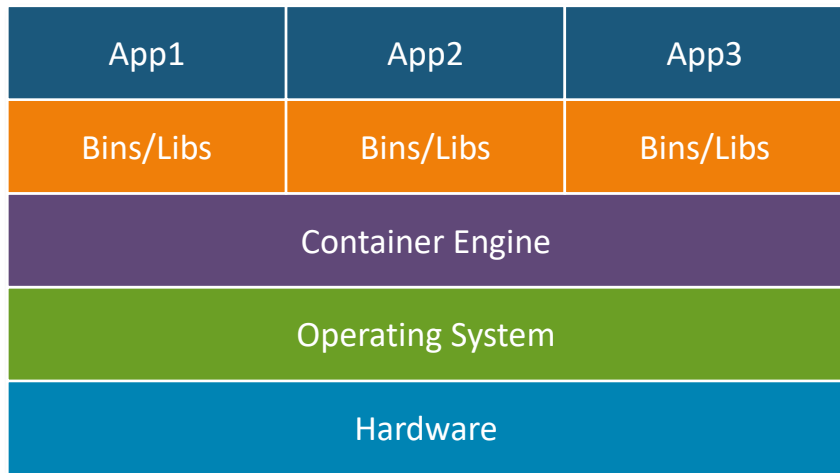


- ✓ It results in reduced spending.
- ✓ Resources are utilized more efficiently.
- ✓ Process of getting software up and running is shorter.
- ✓ Easier backup and disaster recovery is available.

What is Containerization?

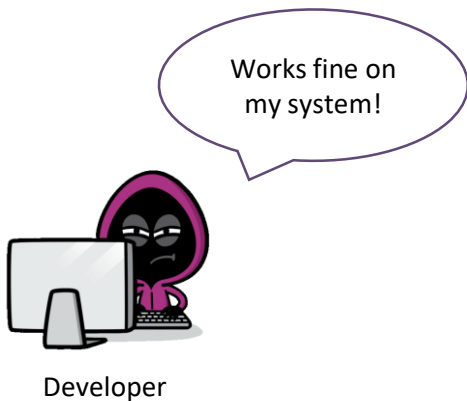
# What is Containerization?

Application **containerization** is an OS-level virtualization method used to deploy and run distributed applications without launching an entire virtual machine (VM) for each app.



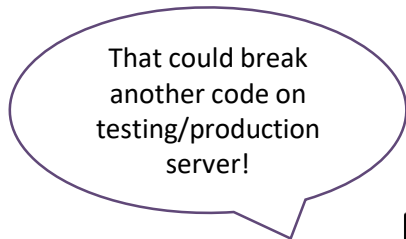
# Problems before Containerization

Developers when run the code on their system, it would run perfectly. But the same code would not run on the operations team's system.



# Problems before Containerization

The problem was with the environment the code was being run in. Well, a simple answer could be, why not give the same VM to the operations/testing team to run the code.



# Problems before Containerization



VMs took too many resources to run.



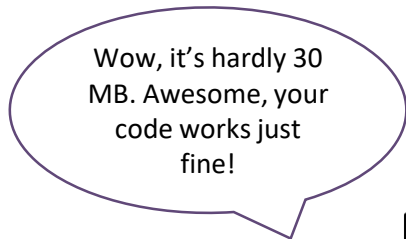
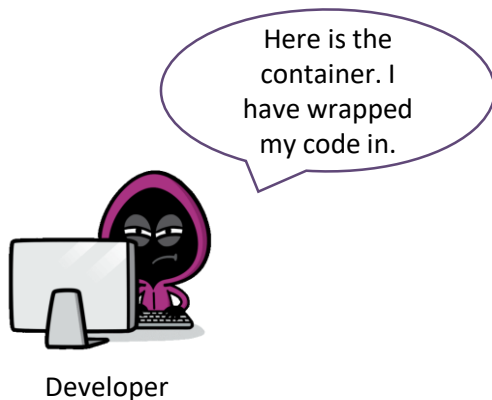
VMs were too big in size to be portable.



VMs were not developer friendly.

# How did containers solve the problems?

With containers, all the environment issues were solved. The developer could easily wrap their code in a lightweight container and pass it on to the operations team.



# Advantages of Containers



- ✓ Containers are not resource hungry.
- ✓ They are lightweight and hence portable.
- ✓ They are developer friendly and can be configured through the code.

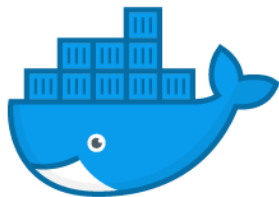


# Containerization Tools

# Containerization Tools



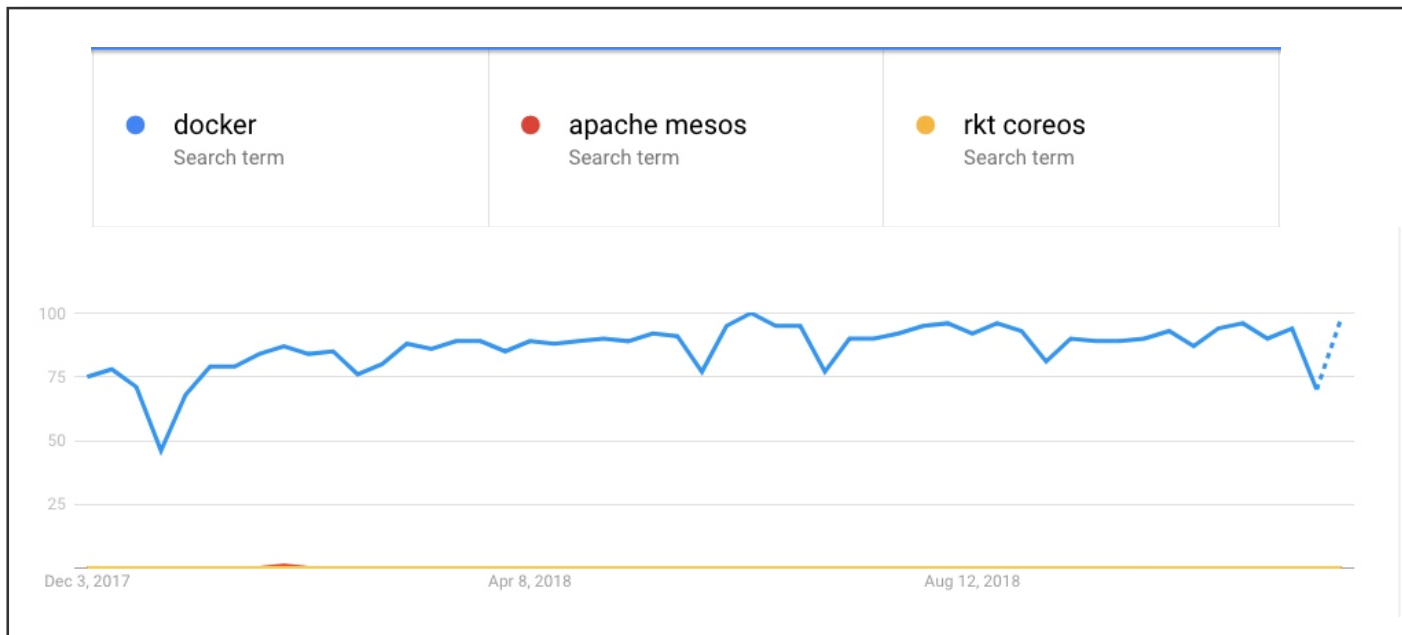
MESOS



docker

# Containerization Tools

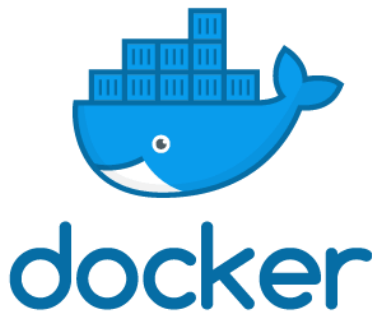
Docker is clearly the most famous among them all!



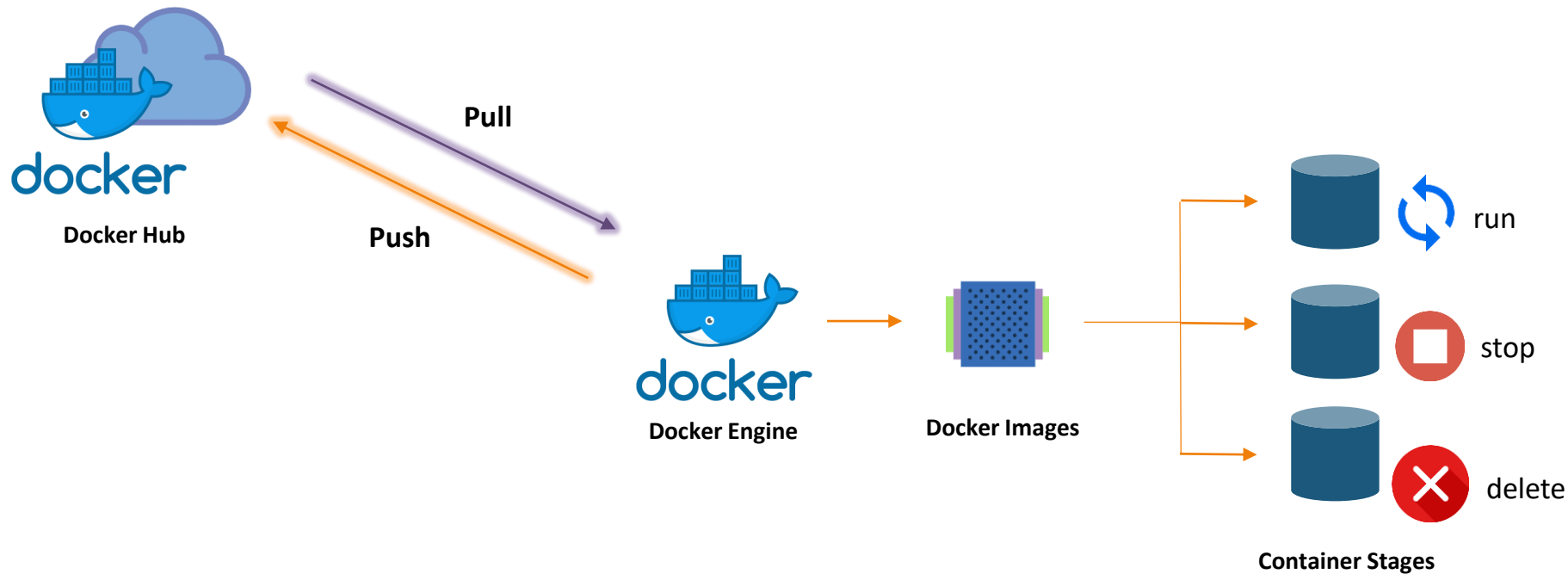
# What is Docker?

# What is Docker?

Docker is a computer program that performs operating-system-level virtualization, also known as "containerization". It was first released in 2013 and is developed by Docker, Inc.  
Docker is used to run software packages called "containers".



# Docker Container Life Cycle



# Components of Docker Ecosystem

# Components of Docker Ecosystem



Docker Hub



Docker Engine



Docker Images



Containers



Docker Volumes



Docker File



# Components of Docker Ecosystem



Docker Hub



Docker Engine



Docker Images



Containers



Docker Volumes



Docker File



Docker Hub is a central public docker registry.



It can store custom docker images.



The service is free, but your images would be public.



It requires username/password.



# Components of Docker Ecosystem



Docker Hub



Docker Engine



Docker Images



Containers



Docker Volumes



Docker File



Docker Engine is the heart of the docker ecosystem.



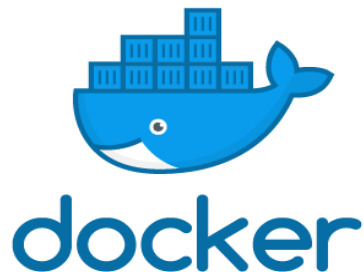
It is responsible for managing your container runtimes.



It works on top of operating system level.



It utilizes the kernel of the underlying OS.



# Components of Docker Ecosystem



Docker Hub



Docker Engine



Docker Images



Containers



Docker Volumes



Docker File



Docker Image is like the template of a container.



It is created in layers.



Any new changes in the image results in creating a new layer.



One can launch multiple containers from a single docker image.



# Components of Docker Ecosystem



Docker Hub



Docker Engine



Docker Images



Containers



Docker Volumes



Docker File



A Docker Container is a lightweight software environment.



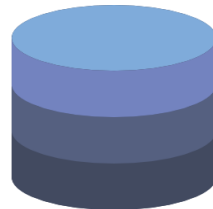
It works on top of the underlying OS kernel.



It is small in size and therefore is highly portable.



It is created using the docker image.



# Components of Docker Ecosystem



Docker Hub



Docker Engine



Docker Images



Containers



Docker Volumes



Docker File



Docker Containers cannot persist data.



To persist data in containers, we can use Docker Volume.



A Docker Volume can connect to multiple containers simultaneously.



If not created explicitly, a volume is automatically created when we create a container.



# Components of Docker Ecosystem



Docker Hub



Docker Engine



Docker Images



Containers



Docker Volumes



Dockerfile



Dockerfile is a Text file, which is used to create custom image



It can include commands that have to be run on the command line



This Dockerfile can be used to build custom container images



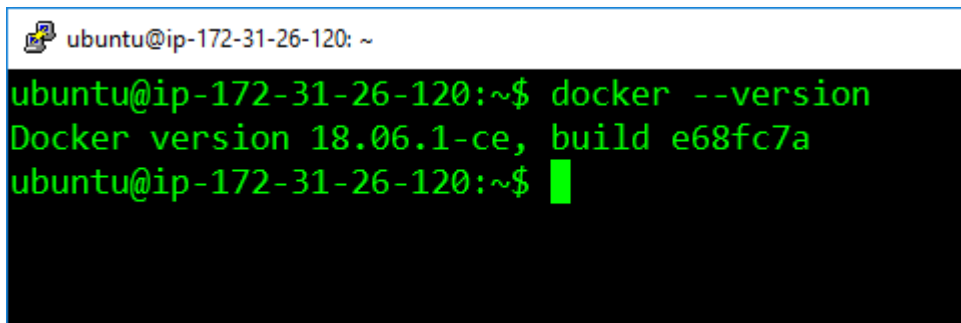
# Installing Docker

# Common Docker Commands



# Common Docker Commands

```
docker --version
```

A terminal window with a white title bar showing 'ubuntu@ip-172-31-26-120: ~'. The terminal has a black background with green text. The command 'docker --version' is entered at the prompt, and the output 'Docker version 18.06.1-ce, build e68fc7a' is displayed on the next line. A green cursor is visible at the end of the second prompt line.

```
ubuntu@ip-172-31-26-120: ~  
ubuntu@ip-172-31-26-120:~$ docker --version  
Docker version 18.06.1-ce, build e68fc7a  
ubuntu@ip-172-31-26-120:~$ █
```

This command helps you know the installed version of the docker software on your system.

# Common Docker Commands

```
docker pull <image-name>
```

```
ubuntu@ip-172-31-26-120: ~  
ubuntu@ip-172-31-26-120:~$ docker pull ubuntu  
Using default tag: latest  
latest: Pulling from library/ubuntu  
32802c0cfa4d: Pull complete  
da1315cffa03: Pull complete  
fa83472a3562: Pull complete  
f85999a86bef: Pull complete  
Digest: sha256:6d0e0c26489e33f5a6f0020edface2727db948974  
Status: Downloaded newer image for ubuntu:latest  
ubuntu@ip-172-31-26-120:~$
```

This command helps you pull images from the central docker repository.

# Common Docker Commands

`docker images`

```
ubuntu@ip-172-31-26-120: ~  
ubuntu@ip-172-31-26-120:~$ docker images  
REPOSITORY          TAG                 IMAGE ID  
SIZE  
ubuntu              latest             93fd78260bd1  
86.2MB  
ubuntu@ip-172-31-26-120:~$
```

This command helps you in listing all the docker images downloaded on your system.

# Common Docker Commands

```
docker run <image-name>
```

```
ubuntu@ip-172-31-26-120: ~
```

```
ubuntu@ip-172-31-26-120:~$ docker run -it -d ubuntu  
233e926091f338a18d3ba915ad34a6b1bc868642d7f3eb120f91  
ubuntu@ip-172-31-26-120:~$
```

This command helps in running containers from their image name.

# Common Docker Commands

`docker ps`

```
ubuntu@ip-172-31-26-120: ~  
ubuntu@ip-172-31-26-120:~$ docker ps  
CONTAINER ID      IMAGE      COMMAND  
STATUS            PORTS      NAMES  
233e926091f3      ubuntu    "/bin/bash"  
Up About a minute  angry_jennings  
ubuntu@ip-172-31-26-120:~$
```

This command helps in listing all the containers which are **running** in the system.

# Common Docker Commands

```
docker ps -a
```

```
ubuntu@ip-172-31-26-120: ~  
ubuntu@ip-172-31-26-120:~$ docker ps -a  
CONTAINER ID        IMAGE               COMMAND  
STATUS              PORTS              NAMES  
f0a5fa001b0e        ubuntu             "/bin/bash"  
Exited (0) 5 seconds ago          relaxed_clark  
233e926091f3        ubuntu             "/bin/bash"  
Up 4 minutes                angry_jenning  
ubuntu@ip-172-31-26-120:~$
```

If there are any stopped containers, they can be seen by adding the `-a` flag in this command.

# Common Docker Commands

```
docker exec <container-id>
```




```
root@233e926091f3: /  
ubuntu@ip-172-31-26-120:~$ docker exec -it 233e926091f3 bash  
root@233e926091f3:/#
```

For logging into/accessing the container, one can use the **exec** command.

# Common Docker Commands

```
docker stop <container-id>
```

 ubuntu@ip-172-31-26-120: ~

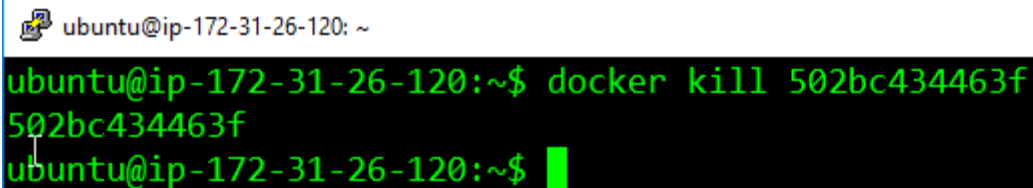
```
ubuntu@ip-172-31-26-120:~$ docker stop 233e926091f3
233e926091f3
ubuntu@ip-172-31-26-120:~$ █
```

For stopping a running container, we use the **stop** command.



# Common Docker Commands

```
docker kill <container-id>
```

A terminal window with a black background and green text. The prompt is 'ubuntu@ip-172-31-26-120: ~'. The command 'docker kill 502bc434463f' is entered and executed. The output shows the container ID '502bc434463f' on the next line, followed by another prompt 'ubuntu@ip-172-31-26-120: ~\$' with a green cursor.

```
ubuntu@ip-172-31-26-120: ~  
ubuntu@ip-172-31-26-120:~$ docker kill 502bc434463f  
502bc434463f  
ubuntu@ip-172-31-26-120:~$
```

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

# Common Docker Commands

```
docker rm <container-id>
```

```
ubuntu@ip-172-31-26-120: ~  
ubuntu@ip-172-31-26-120:~$ docker rm 502bc434463f  
502bc434463f  
ubuntu@ip-172-31-26-120:~$
```

To remove a stopped container from the system, we use the **rm** command.

# Common Docker Commands

```
docker rmi <image-id>
```

```
ubuntu@ip-172-31-26-120: ~  
ubuntu@ip-172-31-26-120:~$ docker rmi 93fd78260bd1  
Untagged: ubuntu:latest  
Untagged: ubuntu@sha256:6d0e0c26489e33f5a6f0020edface27  
71f23c49  
Deleted: sha256:93fd78260bd1495afb484371928661f63e64be3  
Deleted: sha256:1c8cd755b52d6656df927bc8716ee0905853fad  
Deleted: sha256:9203aabb0b583c3cf927d2caf6ba5b11124b0a2  
Deleted: sha256:32f84095aed5a2e947b12a3813f019fc69f159c  
Deleted: sha256:bc7f4b25d0ae3524466891c41cefc7c6833c533  
ubuntu@ip-172-31-26-120:~$
```

To remove an image from the system, we use the **rmi** command.

# Creating a Docker Hub Account

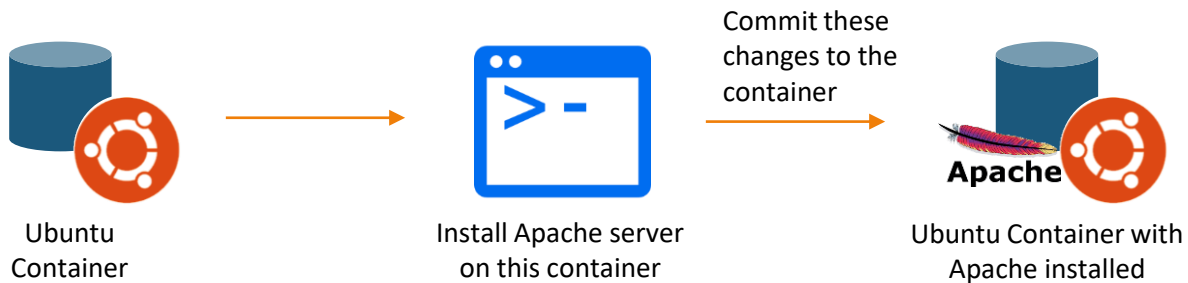
# Creating a Docker Hub Account

1. Navigate to <https://hub.docker.com>
2. Sign up on the website
3. Agree to the terms and conditions
4. Click on Sign up
5. Check your email, and verify your email by clicking on the link
6. Finally, login using the credentials you provided on the sign up page

Committing Changes to a  
Container

# Committing Changes to a Docker Container

Let's try to accomplish the following example with a container and see how we can commit this container into an image.



# Committing Changes to a Docker Container

1. Pull the Docker Container using the command:

```
docker pull ubuntu
```

```
ubuntu@ip-172-31-26-120: ~  
ubuntu@ip-172-31-26-120:~$ docker pull ubuntu  
Using default tag: latest  
latest: Pulling from library/ubuntu  
32802c0cfa4d: Pull complete  
da1315cffa03: Pull complete  
fa83472a3562: Pull complete  
f85999a86bef: Pull complete  
Digest: sha256:6d0e0c26489e33f5a6f0020edface2727db9489  
Status: Downloaded newer image for ubuntu:latest  
ubuntu@ip-172-31-26-120:~$
```

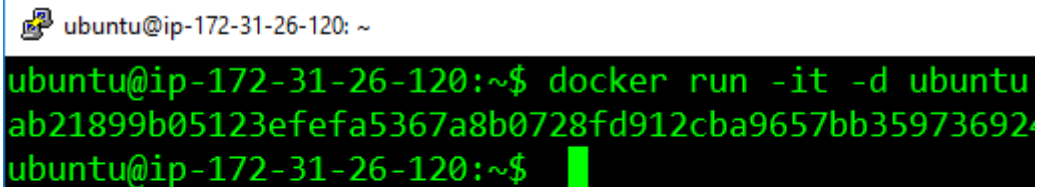
In our case, the image name is “ubuntu”.



# Committing Changes to a Docker Container

2. Run the container using the command:

```
docker run -it -d ubuntu
```



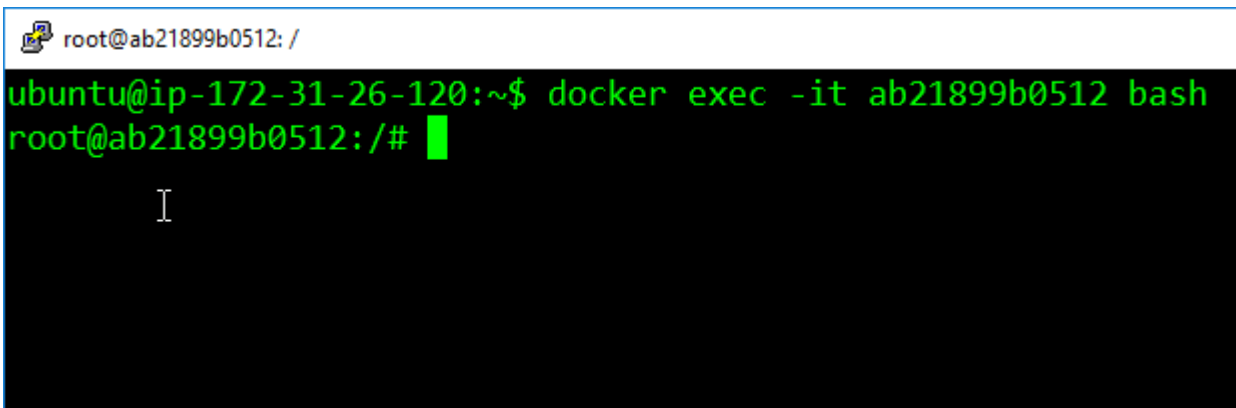
A terminal window with a black background and green text. The prompt is 'ubuntu@ip-172-31-26-120: ~'. The command 'docker run -it -d ubuntu' has been entered and executed. The output is a long alphanumeric string: 'ab21899b05123efefa5367a8b0728fd912cba9657bb359736924'. The prompt is now 'ubuntu@ip-172-31-26-120:~\$' followed by a green cursor block.

```
ubuntu@ip-172-31-26-120: ~  
ubuntu@ip-172-31-26-120:~$ docker run -it -d ubuntu  
ab21899b05123efefa5367a8b0728fd912cba9657bb359736924  
ubuntu@ip-172-31-26-120:~$
```

# Committing Changes to a Docker Container

3. Access the container using the command:

```
docker exec -it <container-id> bash
```




```
root@ab21899b0512: /  
ubuntu@ip-172-31-26-120:~$ docker exec -it ab21899b0512 bash  
root@ab21899b0512:/#
```

# Committing Changes to a Docker Container

4. Install Apache2 on this container using the following commands:

```
apt-get update  
apt-get install apache2
```

 root@ab21899b0512: /

```
root@ab21899b0512:/# apt-get install apache2  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
The following additional packages will be installed:  
  apache2-bin apache2-data apache2-utils file libapr1  
  libaprutil1-dbd-sqlite3 libaprutil1-ldap libasn1-8-
```

# Committing Changes to a Docker Container

5. Exit the container and save it using this command. The saved container will be converted into an image with the name specified.

```
docker commit <container-id> <username>/<container-name>
```

The **username** has to match with the username you created on DockerHub.  
The **container-name** can be anything.

# Pushing the Container on DockerHub

# Pushing the Container on DockerHub

1. The first step is to login. It can be done using the following command:

```
docker login
```

# Pushing the Container on DockerHub

2. For pushing your container on DockerHub, use the following command:

```
docker push <username>/<container-id>
```

# Pushing the Container on DockerHub

3. You can verify the push on DockerHub.

Now anyone, who wants to download this container, can simply pass the following command:

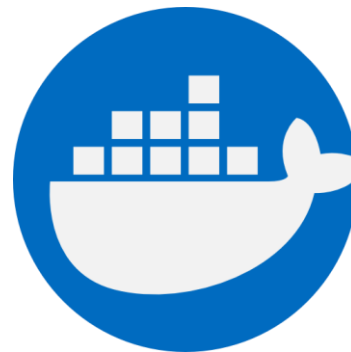
```
docker pull username/apache
```



# Private Registry for Docker

# Private Registry for Docker

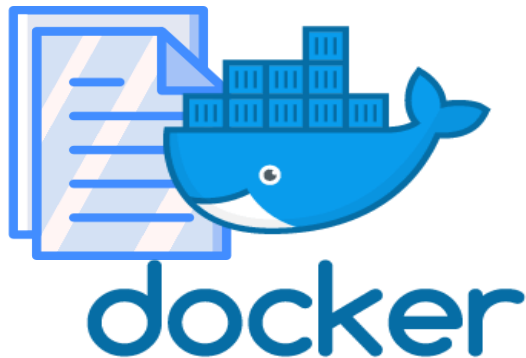
- ✓ DockerHub is a publicly available Docker Registry
- ✓ You may want to create a Private Registry for your company or personal use
- ✓ The registry is available on DockerHub, as a container named 'registry'



# Introduction to Dockerfile

# Introduction to Dockerfile

A **Dockerfile** is a text document that contains all the commands a user could call on the command line to assemble an image. Using the **docker** build, users can create an automated build that executes several command-line instructions in succession.



# Various Commands in Dockerfile

**FROM**

**ADD**

**RUN**

**CMD**

**ENTRYPOINT**

**ENV**

The **FROM** keyword is used to define the base image, on which we will be building.

Example

```
FROM ubuntu
```

Dockerfile

# Various Commands in Dockerfile

FROM

ADD

RUN

CMD

ENTRYPOINT

ENV

The **ADD** keyword is used to add files to the container being built. The syntax used is:

**ADD <source> <destination in container>**

## Example

```
FROM ubuntu  
ADD . /var/www/html
```

Dockerfile

# Various Commands in Dockerfile

FROM

ADD

RUN

CMD

ENTRYPOINT

ENV

The **RUN** keyword is used to add layers to the base image, by installing components. Each RUN statement adds a new layer to the docker image.

## Example

```
FROM ubuntu
RUN apt-get update
RUN apt-get -y install apache2
ADD . /var/www/html
```

Dockerfile

# Various Commands in Dockerfile

FROM

ADD

RUN

**CMD**

ENTRYPOINT

ENV

The **CMD** keyword is used to run commands on the start of the container. These commands run only when there is no argument specified while running the container.

## Example

```
FROM ubuntu
RUN apt-get update
RUN apt-get -y install apache2
ADD . /var/www/html
CMD apachectl -D FOREGROUND
```

Dockerfile



# Various Commands in Dockerfile

FROM

ADD

RUN

CMD

**ENTRYPOINT**

ENV

The **ENTRYPOINT** keyword is used strictly to run commands the moment the container initializes. The difference between CMD and ENTRYPOINT: ENTRYPOINT will run irrespective of the fact whether the argument is specified or not.

## Example

```
FROM ubuntu
RUN apt-get update
RUN apt-get -y install apache2
ADD . /var/www/html
ENTRYPOINT apachectl -D FOREGROUND
```

Dockerfile

# Various Commands in Dockerfile

FROM

ADD

RUN

CMD

ENTRYPOINT

ENV

The **ENV** keyword is used to define environment variables in the container runtime.

## Example

```
FROM ubuntu
RUN apt-get update
RUN apt-get -y install apache2
ADD . /var/www/html
ENTRYPOINT apachectl -D FOREGROUND
ENV name Devops
```

Dockerfile

# Running the Sample Dockerfile

# Running the Sample Dockerfile

Let's see how we can run this sample Dockerfile now.

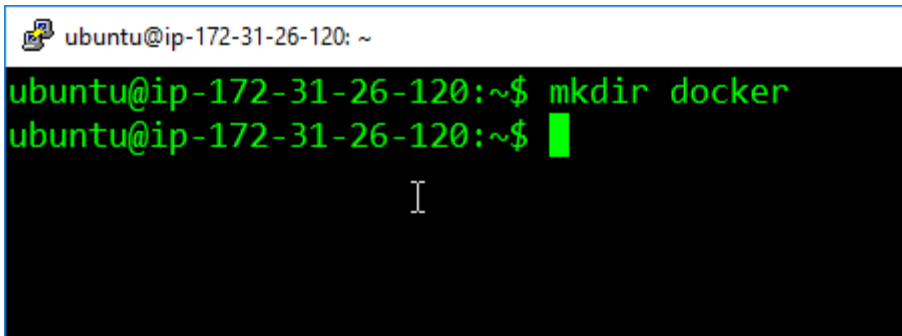
## Example

```
FROM ubuntu:16.04
RUN apt-get update
RUN apt-get -y install apache2
ADD . /var/www/html
ENTRYPOINT apachectl -D FOREGROUND
ENV name Devops
```

Dockerfile

# Running the Sample Dockerfile

1. First, create a folder docker in the home directory.

A terminal window with a black background and green text. The title bar shows a terminal icon and the text 'ubuntu@ip-172-31-26-120: ~'. The terminal content shows the command 'mkdir docker' being executed successfully, followed by a new prompt line.

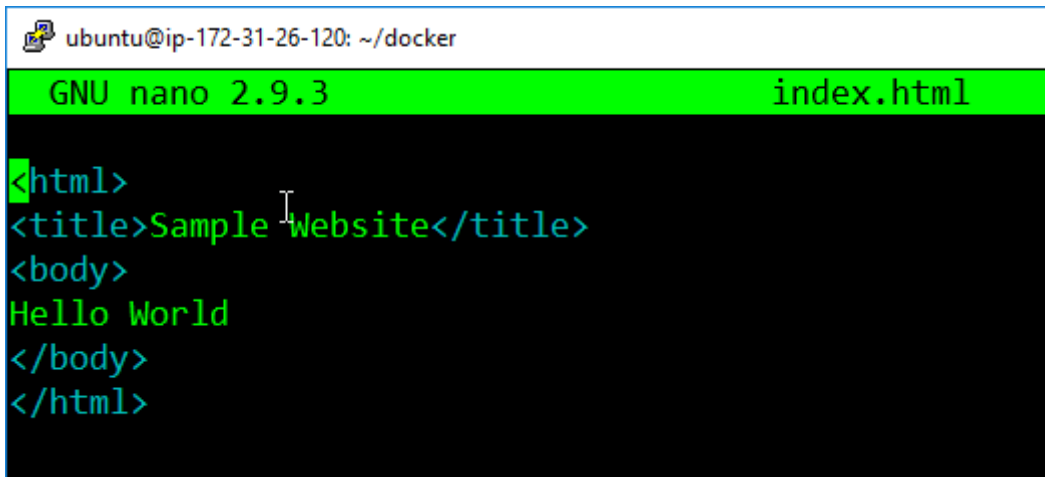
```
ubuntu@ip-172-31-26-120: ~  
ubuntu@ip-172-31-26-120:~$ mkdir docker  
ubuntu@ip-172-31-26-120:~$
```

# Running the Sample Dockerfile

2. Enter into this directory and create a file called 'Dockerfile', with the same contents as the sample Dockerfile.

# Running the Sample Dockerfile

3. Create one more file called 'index.html' with the following contents.



```
ubuntu@ip-172-31-26-120: ~/docker
GNU nano 2.9.3 index.html
<html>
<title>Sample Website</title>
<body>
Hello World
</body>
</html>
```

The screenshot shows a terminal window with the title bar 'ubuntu@ip-172-31-26-120: ~/docker'. The terminal is running the GNU nano 2.9.3 text editor, editing a file named 'index.html'. The editor's status bar at the top is green and displays 'GNU nano 2.9.3' on the left and 'index.html' on the right. The main editing area has a black background with green text. The content of the file is as follows: the first line is '<html>', the second line is '<title>Sample Website</title>', the third line is '<body>', the fourth line is 'Hello World', the fifth line is '</body>', and the sixth line is '</html>'. A white cursor is positioned at the end of the second line, after the word 'Website'.

# Running the Sample Dockerfile

4. Now, pass the following command:

**docker build <directory-of-dockerfile> -t <name of container>**

```
ubuntu@ip-172-31-26-120: ~/docker
ubuntu@ip-172-31-26-120:~/docker$ docker build . -t intellipaat/custom
Sending build context to Docker daemon  3.072kB
Step 1/6 : FROM ubuntu
--> 93fd78260bd1
Step 2/6 : RUN apt-get update
--> Using cache
--> 8ce3e5e6548b
Step 3/6 : RUN apt-get -y install apache2
--> Using cache
--> 296859cef2f0
Step 4/6 : ADD . /var/www/html
--> a3dba497063b
Step 5/6 : ENTRYPOINT apachectl -D FOREGROUND
--> Running in e93d78e6de9d
Removing intermediate container e93d78e6de9d
--> 2a0995664eba
Step 6/6 : ENV name Devops Intellipaat
--> Running in 7497da476b3c
Removing intermediate container 7497da476b3c
--> 73370339b1d4
Successfully built 73370339b1d4
Successfully tagged intellipaat/custom:latest
ubuntu@ip-172-31-26-120:~/docker$
```



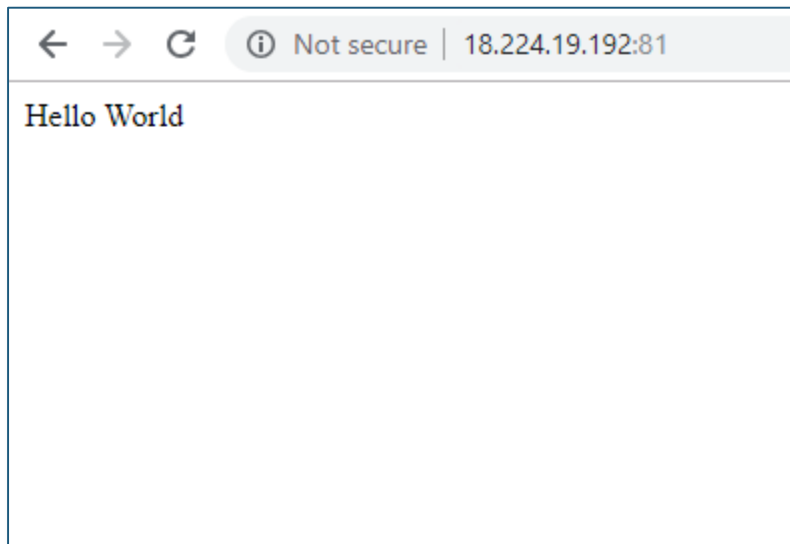
# Running the Sample Dockerfile

5. Finally, run this built image, using the following command:

```
docker run -it -p 81:80 -d username/custom
```

# Running the Sample Dockerfile

6. Now, navigate to the server IP address on port 81.



# Running the Sample Dockerfile

**7.** Finally, login into the container and check the variable `$name`. It will have the same value as given in the Dockerfile.