Docker

What is Docker ?

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. Docker provides the ability to package and run an application in a loosely isolated environment called a container.

CONTAINERS

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.

VIRTUAL MACHINES

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries – taking up tens of GBs. VMs can also be slow to boot.

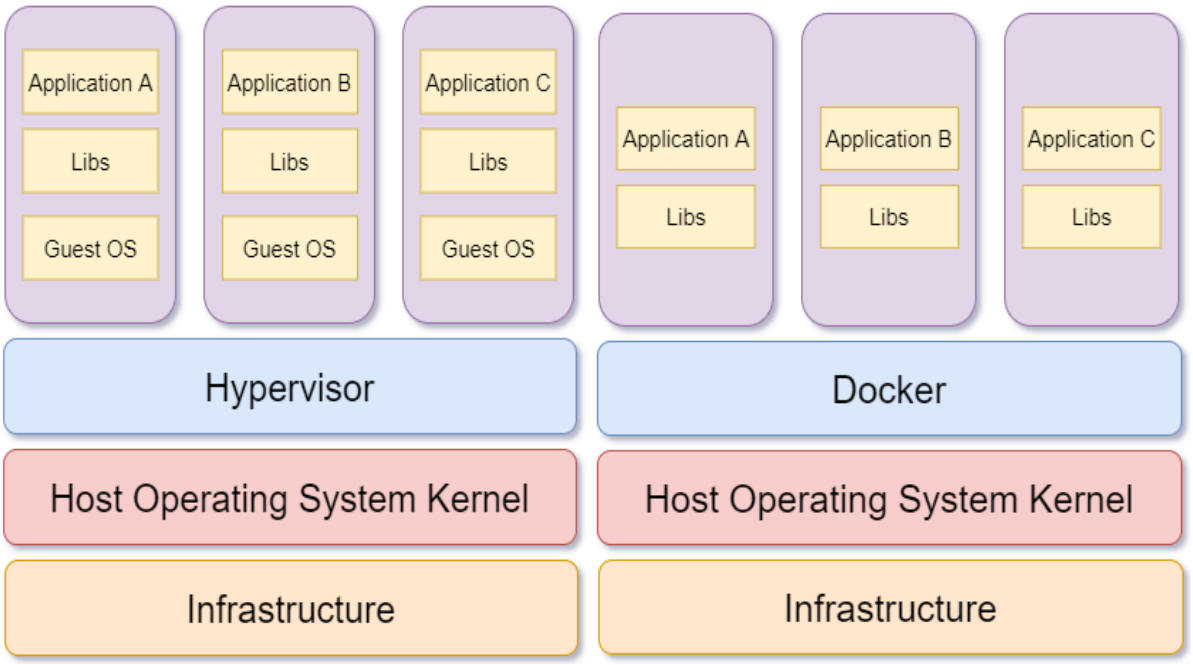
What is Image ?

A Docker image is a read-only template that contains a set of instructions for creating a container that can run on the Docker platform. It provides a convenient way to package up applications and preconfigured server environments, which you can use for your own private use or share publicly with other Docker users. Docker images are also the starting point for anyone using Docker for the first time.

Virtualization - Hardware is dedicated upfront

Containerization - No dedicated hardware upfront

Containerization involves encapsulating or packaging up software code and all its dependencies so that it can run uniformly and consistently on any infrastructure.



++
Let's consider a real life scenario here
Book Management Application
store information regarding all the books you own, and can also serve the purpose of a book lending system for your friends.
If you make a list of the dependencies, that list may look as follows:

- Node.js
- Express.js
- SQLite3

Taking all these into account, the final list of dependencies is as follows:

- Node.js
- Express.js
- SQLite3
- Python 2 or 3
- C/C++ tool-chain

What if you have a teammate who uses Windows while you're using Linux.
That is the idea behind containerization: putting your applications inside a self-contained package, making it portable and reproducible across various environments.Or the fact that popular technologies like nginx are not well optimized to run on Windows. Some technologies like Redis don't even come pre-built for Windows.

All these issues can be solved if only you could somehow:

- Develop and run the application inside an isolated environment (known as a container) that matches your final dPut your application inside a single file (known as an image) along with all its dependencies and necessary deployment configurations.
- And share that image through a central server (known as a registry) that is accessible by anyone with proper authorization.
- deployment environment.

++++++++++++++++++++++++++

In comparison to the traditional virtualization functionalities of hypervisors, Docker containers eliminate the need for a separate guest operating system for every new virtual machine.

Docker implements a high-level API to provide lightweight containers that run processes in isolation.

A Docker container enables rapid deployment with minimum run-time requirements. It also ensures better management and simplified portability.
This helps developers and operations team in rapid deployment of an application.

++++++++++++++++++++++++++

Create Ubuntu Machine on AWS

All Traffic - anywhere

Connect using MobaxTerm
Downloading Mobaxterm
https://mobaxterm.mobatek.net/download-home-edition.html

https://get.docker.com/

Go to Root Account
$ sudo su -
# curl -fsSL https://get.docker.com -o get-docker.sh
(this will download shell script in the machine)

sh get-docker.sh (This will execute the shell script, which will install docker)

How to check the docker is installed or not

docker --version

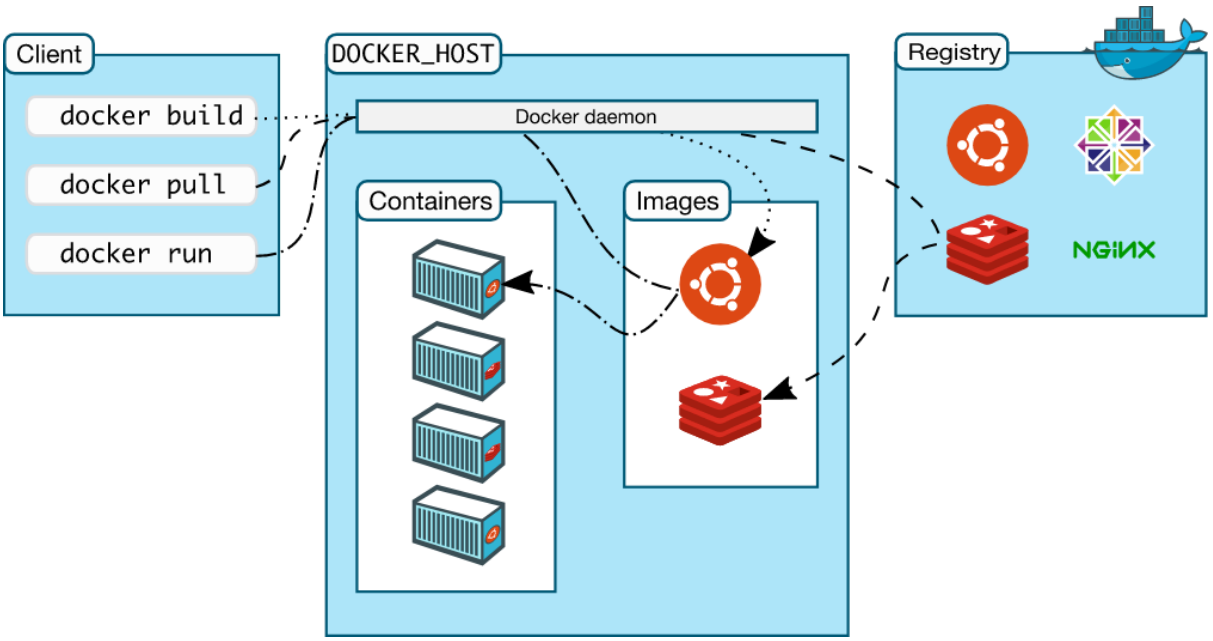
Terminalogy:
1) Docker Images
Combinations of binaries / libraries which are necessary for one software application.

2) Docker Containers
When image is executed comes into running condition, it is called container.

3) Docker Host
Machine on which docker is installed, is called as Docker host.

4) Docker Client
Terminal used to run docker run commands (Git bash)

On linux machine, git bash will work like docker client.



Docker Commands

Working on Images

1 To download a docker image
docker pull image_name

2 To see the list of docker images
docker image ls
(or)
docker images

3 To delete a docker image from docker host
docker rmi image_name/image_id

4) To upload a docker image into docker hub
docker push image_name

5) To tag an image
docker tag image_name ipaddress_of_local_registry:5000/image_name

6) To build an image from a customised container
docker commit container_name/container_id new_image_name

7) To create an image from docker file
docker build -t new_image_name

8) To search for a docker image
docker search image_name

9) To delete all images that are not attached to containers
docker system prune -a

Working on containers

---

10) To see the list of all running continers
docker container ls

11) To see the list of running and stopped containers
docker ps -a

12) To start a container
docker start container_name/container_id

13) To stop a running container
docker stop container_name/container_id

14) To restart a running container
dock restart container_name/container_id
To restart after 10 seconds
docker restart -t 10 container_name/container_id

15) To delete a stopped container
docker rm container_name/container_id

16) To delete a running container
docker rm -f container_name/container id

17) To stop all running containers
docker stop $(docker ps -aq)
(q will give image ids)
18) To restart all containers
docker restart $(docker ps -aq)

19) To remove all stopped containers

docker rm $(docker ps -aq)

20) To remove all contianers(running and stopped)
docker rm -f $(docker ps -aq)

21) To see the logs generated by a container
docker logs container_name/container_id

22) To see the ports used by a container
docker port container_name/container_id

23) To get detailed info about a container
docker inspect container_name/container_id

24) To go into the shell of a running contianer which is moved into background
docker attach container_name/container id

25) To execute anycommand in a container
docker exec -it container_name/container_id command
Eg: To launch the bash shell in a contianer
docker exec -it container_name/container_id bash
26) To create a container from a docker image (imp)
docker run image_name

++

Run command options

-it for opening an interactive terminal in a container

--name Used for giving a name to a container

-d Used for running the container in detached mode as a background process

-e Used for passing environment variables to the container

-p Used for port mapping between port of container with the docker host port.

-P Used for automatic port mapping i.e., it will map the internal port of the container with some port on the host machine.

This host port will be some number greater than 30000

-v Used for attaching a volume to the container

--volume-from Used for sharing volume between containers

--network Used to run the contianer on a specific network
--link Used for linking the container for creating a multi container architecture

--memory Used to specify the maximum amount of ram that the container can use

++=

docker images (There are no images)

To download tomcat image

docker pull tomee

docker images

docker pull ubuntu

If you do not specify the version, by default, we get latest version

I want to download jenkins
docker pull jenkins

To create a container from an image

docker run --name mytomcat -p 7070:8080 tomee

docker run --name c1 -p 7070:8080 tomee

docker exec -it mytomcat bash
To check the tomcat is running or not
http://13.126.59.69:7070

(7070 is port number mapped in docker host)

Lets remove the container (Open another gitbash terminal)

docker stop c1

docker rm -f c1

docker run --name   mytomcat -p 7070:8080   -d   tomee

(The above command runs tomcat in detached mode , so we get out # prompt back)

docker container ls

TO start jenkins
docker run --name myjenkins -p 9090:8080   -d   jenkins/jenkins

To check for jenkins (Open browser)
 http://13.126.59.69:9090
To create ubuntu container
docker run --name myubuntu -it   ubuntu

Observation: You have automatically entered into ubuntu
ls (To see the list of files in ubuntu)
exit (To comeout of container back to host or us ctrl + p + q)

++++++++++++

Scenario:
Start tomcat as a container and name it as "webserver". Perform port mapping and run this container in detached mode

docker run --name webserver -p 7070:8080 -d tomee

To access homepage of the tomcat container
Launch any browser
public_ip_of_dockerhost:7070

++++++++++++++++++++++++++++++++

Scenario:
Start jenkins as a container in detached mode , name is as "devserver", perform port mapping

docker run -d --name devserver -p 9090:8080 jenkins/jenkins

To access home page of jenkins (In browser)
public_ip_of_dockerhost:9090

++

Scenario : Start nginx as a container and name as "appserver", run this in detached mode , perform automatic port mapping

Generally we pull the image and run the image

Instead of pulling, i directly

docker run --name appserver -P -d nginx
(if image is not available, it perform pull operation automatically)
(Capital P , will perform automatic port mapping)

How to check nginx is running or not? (we do not know the port number)

To know the port that is reserved for nginx)
docker port appserver
80/tcp -> 0.0.0.0:32768

80 is nginx port
32768 is dockerhost port

or

docker container ls   (to see the port of nginz and docker host)

To check nginx on browser
13.126.59.69:49153

++++++++++++++++++++++++++
To start centos as container

docker run --name mycentos -it centos
exit (To come back to dockerhost)

++++++++++++++

To start mysql as container, open interactive terminal in it, create a sample table.

docker run --name mydb -d -e MYSQL_ROOT_PASSWORD=laxman mysql:5

docker container ls

I want to open bash terminal of mysql
docker exec -it mydb bash

To connect to mysql database
mysql -u root -p

enter the password, we get mysql prompt

TO see list of databases
> show databases;

TO switch to a databse
> use db_name
> use mysql

> exit
exit
exit

Multi container architecture using docker

This can be done in 2 ways
1) --link / network
2) docker-compose

1) --link option

Use case:

Start two busybox containers and create link between them

(**BusyBox** combines tiny versions of many common UNIX utilities into a single small executable)

Create 1st busy box container
docker run --name busybox1 -it busybox

/ #

How to come out of the container without exit
(ctrl + p + q)

Create 2nd busy box container and establish link to c1 container
docker run --name busybox2 --link busybox1:busybox1-alias -it busybox (busybox1-alias is alias name)

/ #

How to check link is established for not?

/ # ping c10

Ctrl +c (to come out from ping)

(ctrl + p + q)
++++++++++++++++++++++++++++++++

Ex : Creating development environment using docker

Start mysql as container and link it with wordpress container.

Developer should be able to create wordpress website

1) TO start mysql as container

docker run --name mydb -d -e MYSQL_ROOT_PASSWORD=laxman mysql:5

(if container is already in use , remove it
docker rm -f mydb)

Check whether the container is running or not
docker container ls

2) TO start wordpress container
docker run --name mysite -d -p 5050:80 --link mydb:mysql wordpress

13.232.183.233:5050

Check wordpress installed or not
Open browser
public_ip:5050
18.138.58.3:5050

Lets delete all the container
docker rm -f $(docker ps -aq)

To start jenkins as a container
docker run --name devserver -d -p 7070:8080 jenkins/jenkins

to check jenkins is running or not?
Open browser
public_ip:7070
http://18.138.58.3:7070

We need two tomcat containers (qa server and prod server)
docker run --name qaserver -d -p 8080:8080 --link devserver:jenkins tomee

to check the tomcat   use public_ip but port number will be 8080
http://18.138.58.3:8080

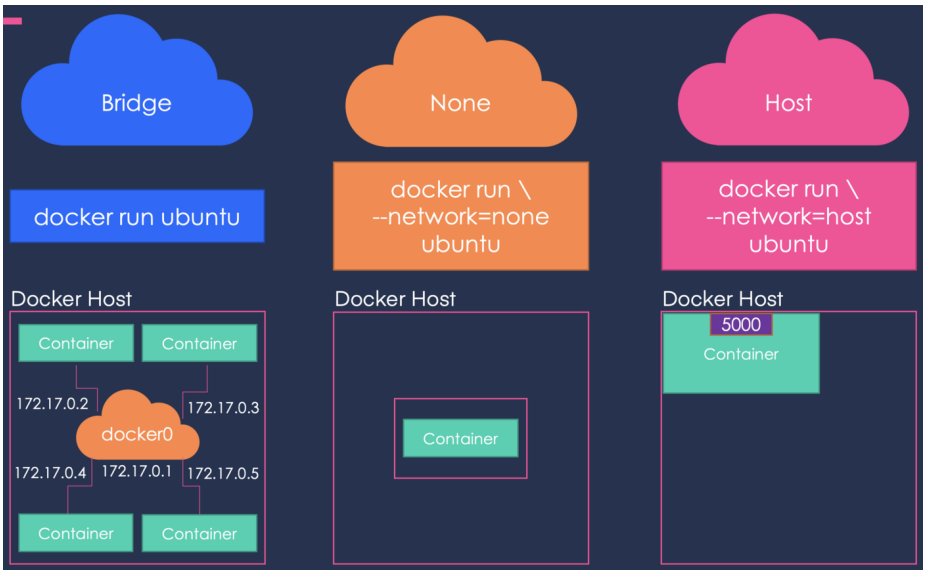
docker run --name prodserver -d -p 9090:8080 --link devserver:jenkins tomee
to check the tomcat of prodserver
http://18.138.58.3:9090

All the commands we learnt till date are adhoc commands.

In the previous usecase we have installed two containers (chrome and firefox)
Lets say you need 80 containers?
Do we need to run 80 commands?

Instead of 80 commands, we can use docker compose which we will see in coming sections.
==========

Docker Networking



One of the reasons Docker containers and services are so powerful is that you can connect them together.

Network Drivers

Docker supports networking for its containers via network drivers.
In this article, we will be discussing how to connect your containers with suitable network drivers. The network drivers used in Docker are below:

- Bridge
- Host
- None
- Overlay
- Macvlan

Bridge

It is a private default network created on the host
Containers linked to this network have an internal IP address through which they communicate with each other easily
The Docker server (daemon) creates a virtual ethernet bridge docker0 that operates automatically, by delivering packets among various network interfaces
These are widely used when applications are executed in a standalone container

Host

It is a public network
It utilises the host's IP address and TCP port space to display the services running inside the container
It effectively disables network isolation between the docker host and the docker containers.

None
In this network driver, the Docker containers will neither have any access to external networks nor will it be able to communicate with other containers
This option is used when a user wants to disable the networking access to a container
In simple terms, None is called a loopback interface, which means it has no external network interfaces

Overlay
This is utilised for creating an internal private network to the Docker nodes in the Docker swarm cluster
Note: Docker Swarm is a service for containers which facilitates developer teams to build and manage a cluster of swarm nodes within the Docker platform
It is an important network driver in Docker networking. It helps in providing the interaction between the stand-alone container and the Docker swarm service

Macvlan
It simplifies the communication process between containers
This network assigns a MAC address to the Docker container. With this Mac address, the Docker server (daemon) routes the network traffic to a router
Note: Docker Daemon is a server which interacts with the operating system and performs all kind of services
It is suitable when a user wants to directly connect the container to the physical network rather than the Docker host

When you start Docker, a default bridge network (also called bridge) is created automatically, and newly-started containers connect to it unless otherwise specified. You can also create user-defined custom bridge networks. User-defined bridge networks are superior to the default bridge network.

Containers on the default bridge network can only access each other by IP addresses, unless you use the --link option, which is considered legacy. On a user-defined bridge network, containers can resolve each other by name or alias.

To list the networks
docker network ls

docker run --name nginx -d -p 8080:80 nginx
It will automatically attached to default bridge network
Access using hostip
docker inspect nginx

docker network ls
docker network create --driver bridge my-network
If we don’t specify the driver, by default it takes bridge driver only.

docker network ls

Running nginx and connecting it to my-network
docker run --name nginx2 -d --network my-network -p 7070:80 nginx

docker inspect nginx2
docker inspect my-network

docker run -it --name ubuntu1 ubuntu

docker network inspect bridge

docker run -it --name ubuntu2 ubuntu

docker network inspect bridge

docker exec -it ubuntu1 bash

ping <internal-ip-of-ubuntu2>

and also try to ping using container name that is
ping ubuntu2

To install ping in the docker container
apt-get update
apt-get install iputils-ping

docker network ls
docker inspect my-network
docker run -it --name ubuntu1 --network my-network ubuntu
docker inspect my-network

docker run -it --name ubuntu2 --network my-network ubuntu
docker inspect my-network

Try to ping one container from other with their ip addresses and DNS names(container names)

To install ping in the docker container
apt-get update
apt-get install iputils-ping
To connect a running container to a network:

docker network connect <network-name> <container-name>
docker network disconnect <network-name> <container-name>

To delete the network
docker network rm my-network
docker network ls

Networking using the host network

To start a nginx container which binds directly to port 80 on the Docker host. From a networking point of view, this is the same level of isolation as if the nginx process were running directly on the Docker host and not in a container. However, in all other ways, such as storage, process namespace, and user namespace, the nginx process is isolated from the host.
Attaching a container to host network:

docker run --name nginx --network host -d nginx
Observe, here we are not performing port mapping.
Access the application using docker-host-ip

Docker volumes

Docker containers are ephemeral (temporary)
Where as the data processed by the container should be permanent.

Generally, when a container is deleted all its data will be lost.
To preserve the data, even after deleting the container, we use volumes.

Volumes are of two types
1) Simple docker volumes
2) Sharable volumes

Simple docker volumes

These volumes are used only when we want to access the data,
even after the container is deleted.

But this data cannot be shared with other containers.

Docker has 3 options for containers to store files in the host machine, so that the files are persisted even after the container stops:

docker volume types:

1. anonymous volumes
2. named volumes
3. host volume or bind volumes

Anonymous Volumes

Create a container with an anonymous volume which is mounted as /data01 on container. in this case we mention container directory name. On host system it maps to a random-hash directory under /var/lib/docker directory.

```
docker run -it --name webuat01 -v /data01 nginx /bin/bash
```

On Host to verify volume

```
docker volume ls
docker inspect <volume_name>
```

watch -n 1 ls /var/lib/docker/volumes/_data

Named Volumes

Create a container with a named volume name which is mounted as `/data01` on container. You can see volume name as `vtwebuat02_data01_val`

```
docker run -it --name webuat02 -v webuat02_data01_val:/data01 nginx /bin/bash
```

Create a named volume then attach volume to a container

```
docker volume create uatweb03_data01_vol
docker run -it --name uatweb03 -v uatweb02_data01_vol:/data01 nginx /bin/bash
```

Host Volumes

Create a host volume

```
mkdir /opt/data02
docker run -it --name c1 -v /opt/data02:/cont-dir nginx bash
```

Use case
---
1) Create a directory called /data ,
start centos as container and mount /data as volume.
Create files in mounted volume in centos container,
exit from the container and delete the container. Check if the files are still available.

Lets create a folder with the name
mkdir /data

docker run --name c1 -it -v /data centos (v option is used to attach volume)

ls (Now, we can see the data folder also in the container)

cd data
touch file1 file2
ls
exit (To come out of the container)
docker inspect c1

We can see under mounts "data" folder it located in the host machine.
Copy the path

/var/lib/docker/volumes/c5c85f87fdc3b46b57bb15f2473786fe7d49250227d1e9dc537bc594db001fc6/_data

Now, lets delete te container
docker rm -f c1

After deleting the container, lets go to the location of the data folder

cd
/var/lib/docker/volumes/d867766f70722eaf8cba651bc1d64c60e9f49c5b1f1ebb9e781260f777f3c7e8/_data

ls (we can see file1 file2)

(Observe , the container is deleted but still the data is persistant)

++++++++++++

Sharable Volumes

These are also known as reusable volume.
The volume used by one container can be shared with other containers.
Even if all the containers are deleted, data will still be available on the docker host.

Ex:

sudo su -

Lets create a directory /data
mkdir /data

Lets Start centos as container
docker run --name c1 -it -v /data:/data centos
ls (we can see the list of files and dir in centos)

cd data
ls (currently we have no files)

Lets create some files
touch file1 file2 (These two files are available in c1 container)

Comeout of the container without exit
Ctrl +p +q (container will still runs in background)

Lets Start another centos as container (c2 container should use the same volume as c1)
docker run --name c2 -it --volumes-from c1 ubuntu

cd data
ls (we can see the files created by c1)

Lets create some more files
touch file3 file4
ls (we see 4 files)

Comeout of the container without exit
Ctrl +p Ctrl +q (container will still runs in background)

Lets Start another centos as container
docker run --name c3 -it --volumes-from c2 centos

cd data
ls (we can see 4 files)
touch file5 file6
ls

Comeout of the container without exit
Ctrl +p Ctrl +q (container will still runs in background)

Now, lets connect to any container which is running in the background
docker attach c1
ls (you can see all the files)
exit

Identify the mount location
$ docker inspect c1
(search for the mount section)

Take a note of the source path

/var/lib/docker/volumes/e22a9b39372615727b964151b6c8108d6c02b13114a3fcce255df0cee7609e15/_data

Lets remove all the container
docker rm -f c1 c2 c3

Lets go to the source path

cd
/var/lib/docker/volumes/e22a9b39372615727b964151b6c8108d6c02b13114a3fcce255df0cee7609e15/_data

ls (we can see all the files)

Creating customised Docker images

Whenever docker container is deleted,
all the softwares that we have installed within the container will also be deleted.

If we can save the container as an image, then we can preserve the softwares.

This creation of custom docker images can be done in two ways.

1) using docker commit command
2) using docker file

Using docker commit
---

docker run --name c11 -it ubuntu

Update apt repository

apt-get update
apt-get install git

TO check the git
git --version
exit

TO save the container as image (snapshot)

docker commit c11 myubuntu

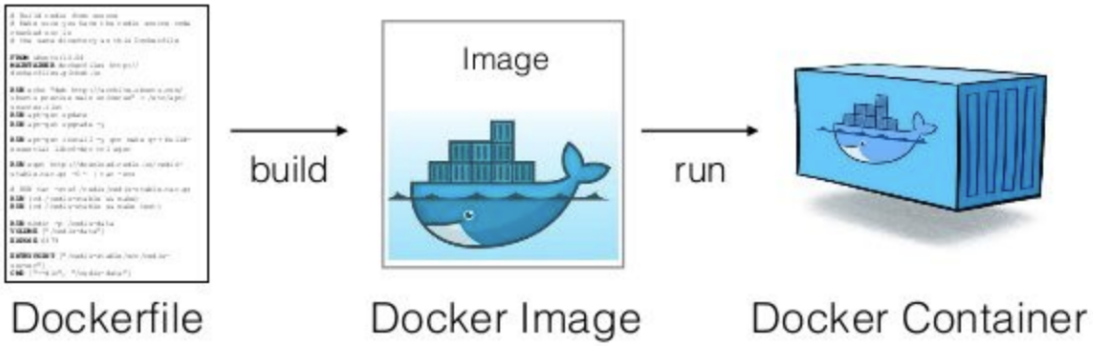
To see the list of images
docker images (you can see the image which you have created)

Now lets run the image which we have created
docker run --name c22 -it myubuntu

git --version (git is pre installed)

Saving docker images into a tar file
docker save -o <file-name.tar> <image-name>
Loading Image from tar file
docker load -i < <file-name.tar>
—--

Dockerfile



Using docker file

This is a simple text file, which uses predefined keywords for creating customised docker images.

Syntax: <Instruction> <command/arguments>

Key words used in docker file (case sensitive)

1) FROM -- used to specify the base image from which the docker file has to be created.

2) MAINTAINER -- This represents name of the organisation or the author who created this docker file.

3) CMD -- This is used to specify the initial command that should be executed when the container starts.

4) ENTRYPOINT - used to specify the default process that should be executed when container starts.
It can also be used for accepting arguments from the CMD instruction.

5) RUN -- Used for running linux commands within the container. It is generally helpful for installing the software in the container.

6) USER -- used to specify the default user who should login into the container.

7) WORKDIR --
Used to specify default working directory in the container

8) COPY -- Copying the files from the host machine to the container.

9) ADD -- Used for copying files from host to container, it can also be used for downloading files from remote servers.

10) ENV -- used for specifying the environment variables that should be passed to the container.

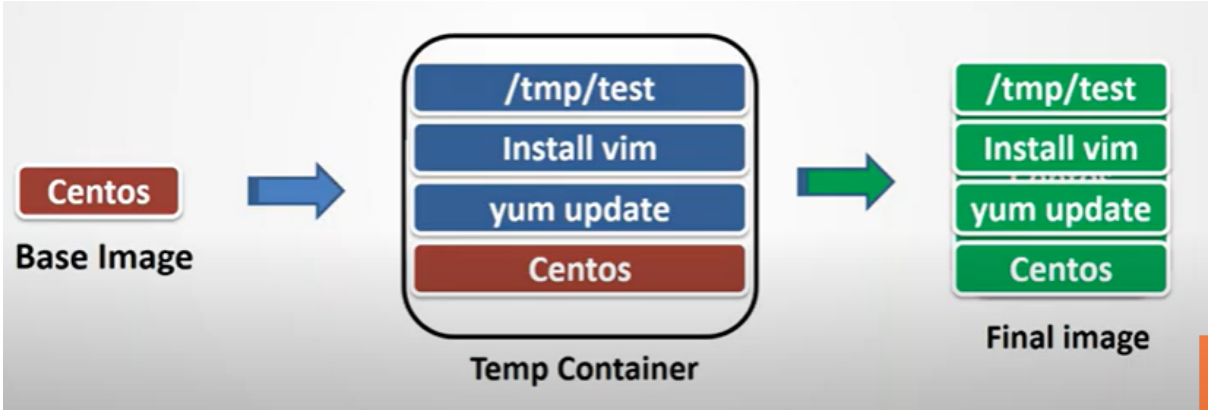
EXPOSE -- Used to specify the internal port of the container
VOLUME -- used to specify the default volume that should be attached to the container.
LABEL -- used for giving label to the container

What is difference between ARG and ENV in Dockerfile
Read https://vsupalov.com/docker-env-vars/

To list out all the installed packages in linux we can use **dpkg -l**

CMD will be overridden by the argument what user has given in the command, But if we use ENTRYPOINT, whatever the user is given in the command, it will be added as an argument to the default command.
++++++++++++++++++++

Image Creation:



Create a dockerfile by taking nginx as the base image
and specify the maintainer as myord Construct an image from the dockerfile.

Creating customized docker images by using docker file.

$ sudo su -
# vim dockerfile

```
FROM nginx
LABEL app-env=dev
```

:wq

TO build an image from the dockerfile
docker build -t mynginx .

(t stands for tag,
. stands for current working dir
mynginx is the new image name)

https://stackoverflow.com/questions/64804749/why-is-docker-build-not-showing-any-output-from-commands
If you don't want to use buildkit, you can revert to the older build engine by exporting
`DOCKER_BUILDKIT=0`

```
export DOCKER_BUILDKIT=0
```

TO see the image
docker images

++

When ever i start my container, i want a program to get executed.

vim dockerfile

```
FROM ubuntu
MAINTAINER Name=Laxman
LABEL type=practice
CMD ["date"]
```

:wq

TO build an image from the dockerfile
docker build -t myubuntu .

TO see the image
docker images

Running conainer from the image
docker run -it   myubuntu

In one docker file, we can have one CMD instruction.
If we give two CMD instruction, it executes the latest one
Lets try

vim dockerfile

```
FROM ubuntu
MAINTAINER myorg
CMD ["date"]
CMD ["ls", "-la"]
```

:wq

docker build -t myubuntu .

docker run -it   myubuntu
(Observation, we get ls -la output)

RUN instruction is used execute linux commands in dockerfile(during image building process)
CMD is used to execute some default command whenever the container starts running from the image.
If we write multiple CMD instructions, only latest CMD instruction will be exected.
CMD command can be overridden with the command line argument.
ENTRYPOINT is used to execute some default command whenever the container starts. But commands or arguments that we provide in the docker run command will be taken as an argument to the ENTRYPOINT command.
ENTRYPOINT command can't be overridden from the command line. Instead that command line argument will be taken as an argument to the ENTRYPOINT command.

* If we have both CMD and ENTRYPOINT instructions in the dockerfile, then in this case also CMD command will be taken as an argument to the ENTRYPOINT command.

++
In ubuntu container, I want to install git in it.

Lets remove the docker file
rm dockerfile
vim dockerfile

```
FROM ubuntu
LABEL maintainer=myorg
RUN apt-get update
RUN apt-get install -y git
```

:wq

Note: CMD -- will run when container starts.
       RUN -- will executed when image is created.

docker build -t myubuntu .

Lets see the images list and space consumed by our image
docker images

docker run -it   myubuntu
git --version
exit

Cache busting

Whenever an image is build from a dockerfile, docker reads its memory and checks which instructions were already executed.
These steps will not be re executed.
It will execute only the latest instructions. This is a time saving mechanism provided by docker.

But, the disadvantage is, we can end up installing software packages from a repository which is updated long time back.

Ex:

cd docker
vim dockerfile

Lets just add one more instruction

```
FROM ubuntu
MAINTAINER laxman

RUN apt-get update
RUN apt-get install -y git
RUN apt-get install -y tree
```

:wq

Lets build an image
docker build -t myubuntu .

(Observe the output, Step 2, 3, 4 is using cache. Only step 5 is executed freshly)

Advantage: time saving mechanism

Disadvantage : Lets say, you are running after 4 months, We are installing tree from apt which is updated long time back.)
Note: If we you don't want the cache memory to be used while building the image use --no-cache
Eg: docker build -t mypythonapp:v2 --no-cache .

TO avoid this disadvantage we use cache busting

vim dockerfile

```
FROM ubuntu
MAINTAINER name=Laxman

RUN apt-get update && apt-get install -y git tree
```

:wq

Lets build an image
docker build -t myubuntu .

==========

```
FROM ubuntu
LABEL maintainer=laxman
RUN apt-get update
RUN apt-get install -y git
CMD ["date"]

FROM ubuntu
LABEL maintainer=laxman
RUN apt-get update
RUN apt-get install -y git --no-cache #This instruction won't use cache memory
ENTRYPOINT ["date"]

FROM ubuntu
LABEL maintainer=laxman
```

```
RUN apt-get update
RUN apt-get install -y git
ENTRYPOINT [“ls”]

FROM ubuntu
LABEL maintainer=laxman
RUN apt-get update
RUN apt-get install -y git
ENTRYPOINT [“ls”]
CMD [“date”]

FROM ubuntu
LABEL maintainer=laxman
RUN apt-get update
RUN apt-get install -y git
ENTRYPOINT [“ls”]
CMD [“-lrt”]
```

What is ARG ?

ARG instruction defines a variable that can be used to build a Docker image.
ARG values are not available after the image is built.
In the running container you can’t access the ARG variables.
Once ARG variable is defined in the Dockerfile, you can always override the values by passing command line argument as shown below
If you need to pass two variables using command line then you need to use --build-arg two times

To pass arguments to our docker image at build time we utilise ARG. ARG is only available during the build of a docker image. Once the image is built we cannot utilise ARG.

What is ENV?

ENV is to provide default values for your future environment variables inside the container.
We can’t change the ENV variable using command line argument directly
If we need to change the ENV variable using command line then we have to use ARG and place ARG variable in ENV variable

```
FROM alpine
ENV environment="production"
ARG app_dir
RUN echo ${app_dir}
```

```
COPY myfile.txt ${app_dir}/myfile.txt
WORKDIR ${app_dir}
```

docker build --build-arg app_dir=mydir -t image:v1 .

docker run -it -e environment="development" image:v2 sh

```
FROM alpine:3.7
ARG VARIABLE_1=5
ENV VARIABLE_2=$VARIABLE_1
RUN echo "print variable value:" $VARIABLE_1
RUN echo " print ENV variable : " $VARIABLE_2
```

Build the docker image directly
You will both variable having same values as defined for ARG variable
docker build .

Build the docker image using CMD line argument
--build-arg VARIABLE_1=7
Output will show 7 for both the variables
docker build -t test --build-arg VARIABLE_1=7 .

Build image by using ENV variable as CMD line argument
Build will fail and you will see the error “[Warning] One or more build-args [VARIABLE_2] were not consumed

docker build -t test --build-arg VARIABLE_2=7 .

docker run -it test

echo $VARIABLE_1
echo $VARIABLE_2

ADD Instruction:

The ADD command is used to copy files/directories into a Docker image. It can copy data in three ways:

Copy files from the local storage to a destination in the Docker image.

Copy a tarball from the local storage and extract it automatically inside a destination in the Docker image.

Copy files from a URL to a destination inside the Docker image.
The ADD command requires a source and a destination.

ADD source destination
If source is a file, it is simply copied to the destination directory.

If source is a directory, its contents are copied to the destination, but the directory itself is not copied.

source can be either a tarball or a URL (as well).

source needs to be within the directory where the docker build command was run.

The Dockerfile specification provides two ways to copy files from the source system into an image: the COPY and ADD directives.
Here we will look at the difference between them and when it makes sense to use each one of them.

Sometimes you see COPY or ADD being used in a Dockerfile, but you should be using COPY 99% of the time. Here's why.

COPY and ADD are both Dockerfile instructions that serve a similar purpose. They let you copy files from a specific location into a Docker image.
COPY takes in a source and destination. It only lets you copy in a local or directory from your host (the machine-building the Docker image) into the Docker image itself.

COPY <src> <dest>
ADD does that same but in addition, it also supports 2 other sources.

A URL instead of a local file/directory.
Extract tar from the source directory into the destination.
ADD <src> <dest>

```
FROM ubuntu
ADD https://dl.k8s.io/release/v1.25.0/bin/windows/amd64/kubectl.exe  /my-app/
```

Dockerizing Python Flask Application

GitHub Url:
https://github.com/clouddevopscoach/python-flask-docker-hello-world/blob/master/Dockerfile

```
FROM python:3.6
MAINTAINER Laxman "laxman@gmail.com"
COPY . /app
WORKDIR /app
```

```
RUN pip install -r requirements.txt
ENTRYPOINT ["python"]
CMD ["app.py"]
```

Build the image using the following command

$ docker build -t simple-flask-app:latest .

Run the Docker container using the command shown below.

$ docker run --name simple-app -d -p 5000:5000 simple-flask-app

The application will be accessible at http:127.0.0.1:5000 or if you are using boot2docker then first find ip address using `$ boot2docker ip` and the use the ip `http://<host_ip>:5000`

Dockerizing Spring Boot Web Service

GitHub Url:
https://github.com/clouddevopscoach/docker-spring-boot-java-web-service-example

```
# Use an official OpenJDK runtime as a parent image
FROM openjdk:8-jdk-alpine

RUN apk update && apk add bash

# Set the working directory to /app
WORKDIR /app

# Copy the fat jar into the container at /app
COPY /target/docker-java-app-example.jar /app

# Make port 8080 available to the world outside this container
EXPOSE 8080

# Run jar file when the container launches
CMD ["java", "-jar", "docker-java-app-example.jar"]

—------
Containerising maven also

FROM openjdk:8-jdk-alpine
RUN apk update && apk add bash
```

```
# Set the working directory to /app
WORKDIR /app
COPY . .
RUN wget
https://mirrors.estointernet.in/apache/maven/maven-3/3.6.3/binaries/apache-maven-3.6.3-bin.tar.gz
RUN tar -xvf apache-maven-3.6.3-bin.tar.gz
RUN mv apache-maven-3.6.3 /opt/
ENV M2_HOME='/opt/apache-maven-3.6.3'
ENV PATH="$M2_HOME/bin:$PATH"
RUN export PATH
RUN mvn clean install
# Make port 8080 available to the world outside this container
EXPOSE 8080

# Run jar file when the container launches
CMD ["java", "-jar", "target/docker-java-app-example.jar"]



—----------
FROM ubuntu
RUN apt update && apt install openjdk-8-jdk -y
RUN apt update && apt install maven -y
COPY . /my-dir/
WORKDIR /my-dir
RUN mvn clean install
WORKDIR /my-dir/target
CMD ["java", "-jar", "docker-java-app-example.jar"]
```

Before building the image, we need to create the Uber Jar (aka Fat Jar) file, to do so, just clean install with maven as below;

```
mvn clean install
```

```
docker build --tag=docker-java-hello-world-app .
```

```
docker run -p 80:8080 docker-java-hello-world-app
```

As you run the command, you can open up your browser, you can go to this url http://publicip_of_docker_host:80/docker-java-app/test

Multi Stage Build

```
FROM ubuntu AS builder-stage
RUN apt-get update && apt-get install maven -y
WORKDIR /app
COPY . /app
RUN mvn clean install


FROM openjdk:8-jdk-alpine
RUN apk update && apk add bash
# Set the working directory to /app
WORKDIR /app
# Copy the fat jar into the container at /app
COPY --from=builder-stage /app/target/docker-java-app-example.jar /app
# Make port 8080 available to the world outside this container
EXPOSE 8080
# Run jar file when the container launches
CMD ["java", "-jar", "docker-java-app-example.jar"]
```

docker build --tag=docker-java-hello-world-app .
docker run --name spring-boot-app -p 80:8080 docker-java-hello-world-app
As you run the command, you can open up your browser, you can go to this url
http://publicip_of_docker_host:80/docker-java-app/test

Dockerizing-a-NodeJS-web-app

GitHub Url: https://github.com/clouddevopscoach/Dockerizing-a-NodeJS-web-app

```
FROM node:10

# Create app directory
WORKDIR /usr/app

# Install app dependencies
# A wildcard is used to ensure both package.json AND package-lock.json are copied
# where available (npm@5+)
```

```
COPY package*.json ./

RUN npm install
# If you are building your code for production
# RUN npm ci --only=production

# Bundle app source
COPY . .

EXPOSE 8080
CMD [ "node", "server.js" ]

docker build -t mynodejs:v1 .

docker run -p 7070:8080 -d mynodejs:v1



—----------------------------------------------
Installing npm in our ubuntu base image

FROM ubuntu
RUN apt-get update && apt-get install -y npm
#FROM node:10

# Create app directory
WORKDIR /usr/app

# Install app dependencies
# A wildcard is used to ensure both package.json AND package-lock.json are copied
# where available (npm@5+)
COPY package*.json ./

RUN npm install
# If you are building your code for production
# RUN npm ci --only=production

# Bundle app source
COPY . .

EXPOSE 8080
CMD [ "node", "server.js" ]
```

docker build -t mynodejs:v1 .

docker run -p 9090:8080 -d mynodejs:v1

==========
==========
Working on docker registry
Registry is a location where docker images are saved.
Types of registry
1) public registry
2) private registry

public registry is hub.docker.com
Images uploaded here are available for everyone.

Usecase: Create a customized ubuntu image, by installing tree in it.

Save this container as an image, and upload this image in docker hub.

Step 1: Create a new account in hub.docker.com

Step 2: Creating our own container
docker run --name c5 -it ubuntu

Lets install tree package in this container
/# apt-get update
/# apt-get install tree
/# exit

Step 3: Save the above container as an image
docker commit c5 clouddevopscoach/ubuntu_img26
(clouddevopscoach/ubuntu_img15 -- is the image name)

Note: Image name should start with docker_id/

To see the list of images
docker image ls (we can see the new image)

TO upload the image to hub.docker.com (docker login command is used)
---
docker login (provide docker_id and password)

To upload the image
docker push <image_name>
docker push clouddevopscoach/ubuntu_img26

login to docker hub to see your image

Using Elastic Container Registry
Login into aws.amazon.com
Go to ECR (Elastic Container Registry)
Create a Private Repository
Install aws cli in docker host
apt-get update
apt-get install awscli -y
aws --version
aws configure
 Now, go to IAM service in aws
Go to my security credentials
Go to Access keys (access key ID and secret access key)
Create New Access key if it is not there
Or else use existing access keys.
Now come to docker host
aws configure
Give Access Key ID: and Access Key ID:
Give AWS region ap-south-1

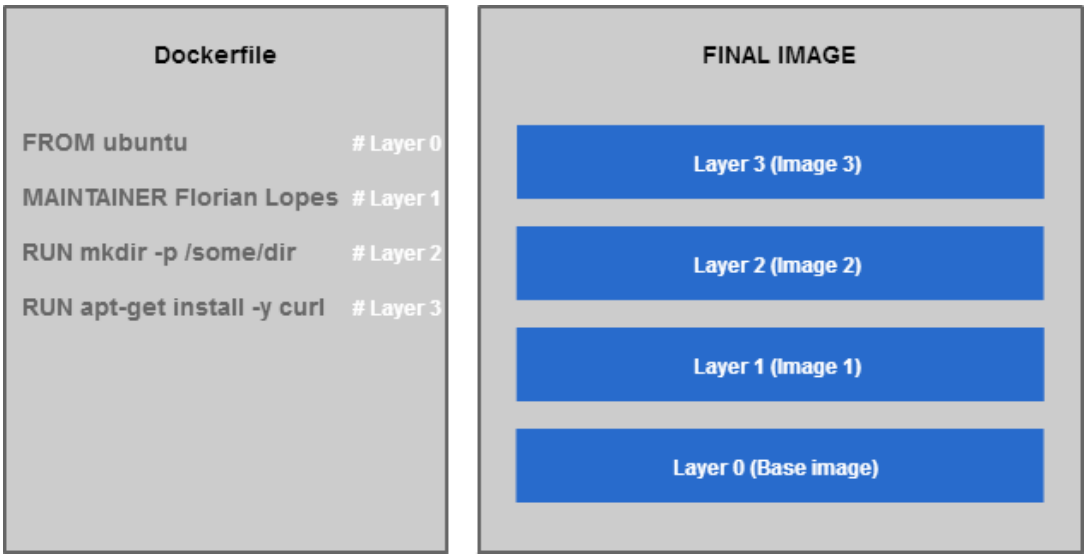
aws ecr get-login --no-include-email --region ap-south-1
You will be getting docker login command and credentials
Copy and paste it in command line
Login will be succeeded.
Tag the docker images with URI
docker tag <local-image-name> <URI:v1>
docker tag myimage:later
494598202083.dkr.ecr.ap-south-1.amazonaws.com/clouddevopscoach:v1

docker push <URI:v1>
docker push 494598202083.dkr.ecr.ap-south-1.amazonaws.com/clouddevopscoach:v1

==========

Docker Layers

A Docker image consists of several layers. Each layer corresponds to certain instructions in your Dockerfile.

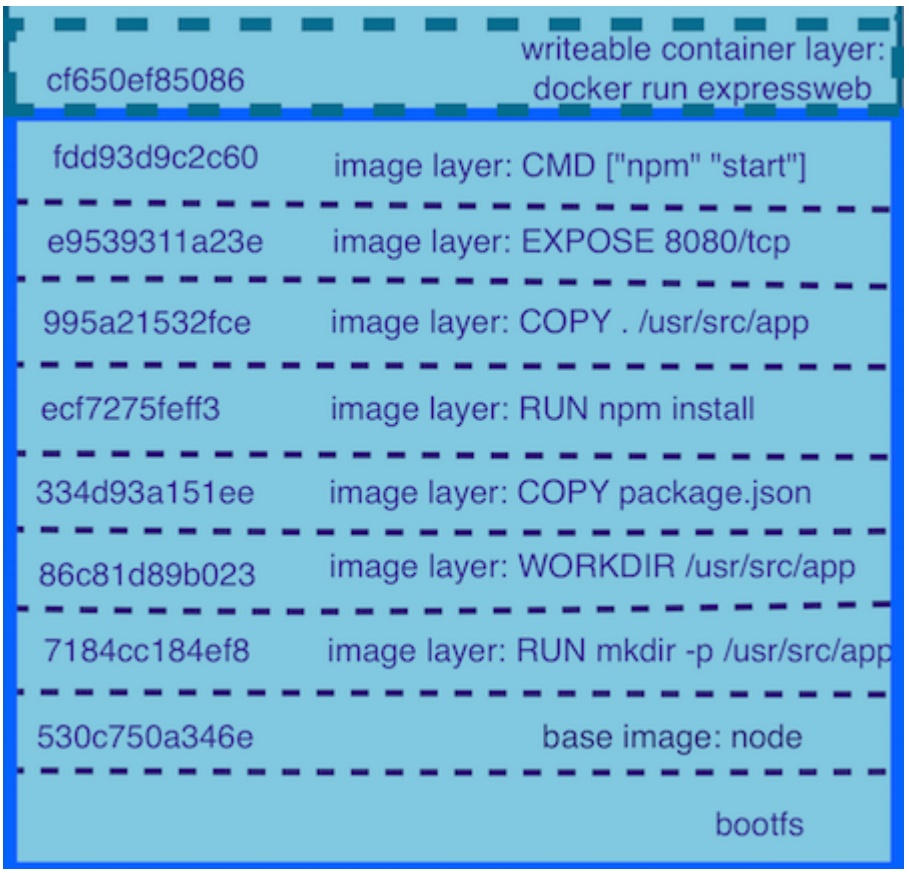


When you pull a Docker image, you will notice that it is pulled as different layers. Also, when you create your own Docker image, several layers are created.

```
FROM node:argon
# Create app directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
# Install app dependencies
COPY package.json /usr/src/app/
RUN npm install
# Bundle app source
COPY . /usr/src/app
EXPOSE 8080
CMD [ "npm", "start" ]
```

```
$ docker build -t expressweb .
Step 1 : FROM node:argon
argon: Pulling from library/node...
...
Status: Downloaded newer image for node:argon
 ---> 530c750a346e
Step 2 : RUN mkdir -p /usr/src/app
 ---> Running in 5090fde23e44
 ---> 7184cc184ef8
Removing intermediate container 5090fde23e44
Step 3 : WORKDIR /usr/src/app
```

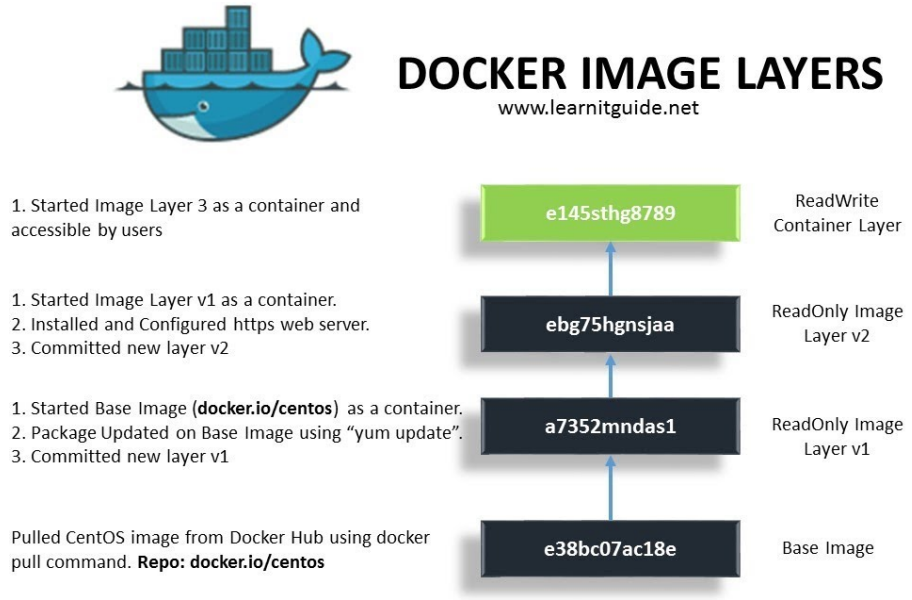
```
 ---> Running in 2987746b5fba
 ---> 86c81d89b023
Removing intermediate container 2987746b5fba
Step 4 : COPY package.json /usr/src/app/
 ---> 334d93a151ee
Removing intermediate container a678c817e467
Step 5 : RUN npm install
 ---> Running in 31ee9721cccb
 ---> ecf7275feff3
Removing intermediate container 31ee9721cccb
Step 6 : COPY . /usr/src/app
 ---> 995a21532fce
Removing intermediate container a3b7591bf46d
Step 7 : EXPOSE 8080
 ---> Running in fddb8afb98d7
 ---> e9539311a23e
Removing intermediate container fddb8afb98d7
Step 8 : CMD npm start
 ---> Running in a262fd016da6
 ---> fdd93d9c2c60
Removing intermediate container a262fd016da6
Successfully built fdd93d9c2c60
```



https://github.com/clouddevopscoach/docker-layers.git

Branch: feature/dockerbenchsecurity

```
FROM openjdk:10-jdk

VOLUME /tmp

RUN useradd -d /home/appuser -m -s /bin/bash appuser

USER appuser

ARG JAR_FILE

COPY ${JAR_FILE} app.jar

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```



docker image ls
docker history <image-id/image-name>

The following instructions create a layer: RUN, COPY, ADD. The other instructions will create intermediate layers and do not influence the size of your image.
What happens here? We notice that layers are created and most of them are removed (removing intermediate container). So why does it say removing intermediate container and not removing intermediate layer? That's because a build step is executed in an intermediate container. When the build step is finished executing, the intermediate container can be removed.

Recreate the Docker Image

We notice that the first layers are identical to our previous build. The layer ID's are the same. In the log, we notice that the layers are taken from the cache.

==========