docker container launching commands:
---
docker container run image:tag
docker container run -it image:tag command = if we want to override the default command with which the container should be started and want to change configuration before container begins execution.
docker container run -d image:tag = if it is a long running container
docker container run --name containerName -it image:tag command

container management commands:
---
docker container ls = displays all the currently running containers
docker container ls -a = show all the containers of any status

note:-we can run all the container management commands either by using container id or container name

docker container exec -it containerName command = to go inside the running container

#1 we can attach the terminal to the currently executing container using the below command
docker attach containerId/Name = The current terminal prompt will enter inside the container, so if we terminate or exit from terminal prompt container will be stopped.

#2 how to stop the running docker container
docker container stop containerId/Name

#3 how to restart a container under execution?
docker container restart containerId/Name

#4 What are the possible statuses in which a docker container can exists?
docker container ls -a we can see the statuses of the containers in which those are there. the status column can take one of these values below.
1. created = the container has been created out of the image, but has not yet started execution
2. running = it is up and running and ready for accessing
3. paused = temporarily paused
4. restarting = stop and start container
5. removing = container is removed
6. exited = finished execution successfully
7. dead = forcefully terminated

#5. how to see all the containers in execution
docker container ls = the otherway way we can get the same information is using
docker container ps

#6 how to kill the non-responsive running containers
docker container kill containerId/name

#7 we can see the stats of the running docker container using
docker container stats containerId
shows the statistics of a container like cpu, memory, network i/o, storage etc

docker stats -a = shows the statistics of all the containers that are running

#8 how to access the logs of the running container

docker container logs containerId/name
(or)
docker container logs -f containerId/name

#9 to access information about the docker environment
docker info = will shows the information about the docker workstation, like version, containers in execution/stopped etc.

#10 docker inspect containerId/Name = displays the information about the container.

#11 we can start back the stopped container using
docker container start containerName/Id
---
docker is an containerization technology used for package software applications, its dependent libraries together, so that the programs can be executed isolated from another within the same env also.

Till now we are running docker containers out of the images that are build by someone else. But how to package our applications / dependencies into docker image, so that we can run containers out of our images we build.

The developer has to writing dockerfile with build instructions in packaging the docker application.
The default file in which the developer has to write the build images instructions is "dockerfile" with no extension and should be placed inside a directory.

In this file developer has to write docker directives in asking the docker daemon to build image.
These docker image instructions can be classified into 2 types
1. Image building instructions = are executed at the time of building the docker image, usually gets expanded and added as image layers on docker image

2. Container instructions = executed at the time of starting/running/launching docker container

IMAGE INSTRUCTIONS
---
1. FROM
2. ARG
3. ENV
4. COPY
5. RUN
6. ADD
7. LABEL
8. WORKDIR

CONTAINER INSTRUCTIONS
---
1. VOLUME
2. EXPOSE
3. CMD
4. ENTRYPOINT
5. HEALTHCHECK

Till now we are running docker containers by using the docker images that are already been built and published as part of the docker hub repo. Instead we can build our own docker images with our application and their dependent libraries with which we can create docker containers in running our applications.

For this, we need build our own docker image.
How to build our own docker image by packaging our application and their dependent libraries?
To build our own docker image we need to write docker build file with docker build instructions inside it. We need to create an directory inside which we need to create a docker build file with name Dockerfile (default name) (no extension) and should write docker build instructions which are called Docker Directives inside it.

Using these Docker Directives we can perform the below activities:-
#1 The immutable content that seems to be common across all the containers should be placed within the docker image itself like
- application binaries using which we want to launch container
- software packages, (required to run your application)
- any external static configuration files (required for your application to run)
all of these above are considered as immutable/readonly as we use them to run our application

#2 using docker directives we can write container execution commands that should be executed while launching the container to run the application that are packaged as part of the image.

Important:-
From the above we can understand docker image is a binary, build out of our appliation and their dependencies to execute/run our application, we should not package our source code of the application. because it is neither a repository to distribute source code nor it is an virtualmachine env to use for development. It is meant for shipping our application binaries to execute in a standard workflow process.

Docker containers are the processes that runs on top of our operating system in executing our software applications in an isolated manner, once the program execution has been completed how a process would be terminated, similarly the docker container process also would be terminated once the container program finished execution. so, a docker container is not an virtual machine to package application source code to be used as a machine for development.

We can say now there are 2 types of docker directives or build instructions are available
1. docker image instructions = These directives are executed during the time of building the docker image, these instructions are general used for carrying the activies like
- packaging our application binaries as part of the image
- installing software packages onto the image
- copying external configuration files onto the image

every instruction will produces an layered image on top of the previous instruction being executed and build as a stackable image.

2. docker container instructions
The directives only executes at the time of running the docker container only to lauch or run our application.

Docker image directives / image build instructions
FROM
ENV
COPY

ADD
RUN
ARG
WORKDIR
LABEL

Docker Container Instructions
CMD
EXPOSE
VOLUME
ENTRYPOINT
HEALTHCHECK

What is docker build context?
We write docker file with image/container instructions on a development environment, when we run docker build cli command to build docker image, the docker cli looks for the docker file with the name Dockerfile by default inside the current directory we we are running the command from.

If you have your Dockerfile in a different directory then the current and if the name of the file is different then we should use -f flag for specifying the filelocation to be used for building the image.
docker build img:tag -f /docker/instructions/Dockerfile

When we issue docker build instruction, the docker cli will transfers the entire contents of the directory in which our Dockerfile is there asking the docker daemon to use those instructions in building the docker image. So the directory in which the Dockerfile exists is called "Build Context" and by default it is always "." - indicates current directory.

By the above we can understand always we should be careful in writing Dockerfiles and placing them for eg.. if you have placed your Dockerfile under / (root) of your filesystem and executed docker build command entire system directories will be send to docker daemon. So always avoid placing Dockerfiles into some global locations.
create an empty directory and places the contents you want to ship to the docker daemon in building your image.

A Dockerfile always will start with FROM instruction indicating the base docker image to be used in building our docker image.

but optionally before the FROM instruction we can have parser directives as the beginging line of Dockerfile. The parser directives looks like comments written in docker file as below.
comment
but these are special type of comments interpreter by the docker daemon specially while executing the Dockerfile and there are 2 types of parser directives are there
1. syntax = it is used for referencing another remote dockerfile as part of your docker build
2. escape = used for changing the default escape character to be used inside docker build file

syntax of writing parser directives
syntax=docker/dockerfile:1.2
escape=`

ESCAPE = parser directive
in docker build file the default character used as escape sequence is "\", any character followed

by \ are interpreted specially
\r = carriage return
\n = new line
etc
but while working in widows environment \(backslash) is used as a file separator to specify the directory path, in such case we want to replace escape charater (\) to something else in image file we can do it using ESCAPE parser directive declaring at the top of docker build file as below

Dockerfile

ESCAPE=`

SYNTAX parser directive

SYNTAX is a directive defines the location of the Dockerfile syntax that is used to build the Dockerfile. we use SYNTAX directive to add/import additional docker directives (frontends) in writing our Dockerfile using which we can build docker image.
The docker has provided additional docker directives as part of the docker hub repository which can imported into our Dockerfile used as below.
SYNTAX=docker/docker:1.2.1
The docker SYNTAX directive is only available when we are build docker images with buildkit.

buildkit is a new backend for executing the builds (images) and has lot of benefits when compared with old implementation
1. detects and skip transfer of files from docker context to the daemon
2. use external Dockerfile implementations with new features
3. parallelizes building independent images
3. Detect and skip unused build stages

by default buildkit is installed as part of our docker installation (>18.09) but will not be enabled to build if we want we need to set an env property
DOCKER_BUILDKIT=1
after that if we run docker build command uses new backend for building images.

The parser directives should be written as the first-lines in docker file if at all if they appear.
if we have an empty line or comment before the parser directive docker daemon treats them as comments and ignore them
the docker parser directives would be ignored if they appear of a build instruction in Dockerfile

wrong
FROM ubuntu:21.04
ESCAPSE=` # this directive is ignored and treats as comment

wrong
Dockerfile
#comment
ESCAPE=` # ignored and treated as comment because we have comment above
FROM ubuntu:21.04

wrong
Dockerfile
ESCAPE=`

SYNTAX=docker\Dockerfile:1.2.1 #ignored because there is an empty line
FROM ubuntu:21.04

A Dockerfile has to must and should start with a docker build image instruction FROM, (optionally) parser directives.

FROM --platform=platform
for eg.. FROM --platform=linux/arm64 ubuntu:21.04

The base images provided works on different platforms like linux/amd64 or linux/arm64 or windows/amd64 and may have same name
image: ubuntu:21.04 = there can be multiple docker images ported on multiple hardware platforms

here the --platform is optional and docker daemon automatically detects the platform based on the hardware you are running the build, if we want to change the default then we need to use platform.

FROM [--platform=platform] image:[tag] as image
by default latest
this is how we need to use an FROM directive.

ENV is a directive used for defining environment varibles with in during docker image so that those are available during container execution.
syntax:-
ENV variable=variableValue
we can define any number of ENV variables all of those are available in container execution.
instead of defining environment variables statically in Dockerfile we can pass env variables during the time of building image as well

varibles.env
JAVA_HOME=/u01/data/jdk11
TOMCAT_HOME=/u01/data/apache-tomcat-9.0

docker build -t img:tag --env-file=variables.env .
---
we can build a docker image with different docker file name and from different directory rather than in the Dockerfile location using below.
docker build -t bctx:1.0 -f buildctx/BuildCtxfile buildctx/.

CMD (container instruction)
CMD = is a Command directive used for running a command/shell command during the lauch of the container. It has total 3 syntax of writing it

1. execute form (recommended)
2. shell form
3. arguments to the ENTRYPOINT directive (later) = until we come to entry point

The CMD acts as a default command to be used while running the docker container, and always the user can override the CMD we specified while launching the container.

unlike CMD an ENTIRYPOINT directive is also used for running a command while launching the docker container, where the user cannot override the ENTRYPOINT command in anycase.
we can write any number of CMD directives in a Dockerfile, but only the last CMD will be executed ignoring the rest.

In case in our Dockerfile if we have CMD with ENTRYPOINT directive, the CMD directive acts as argument to the ENTRYPOINT, only the ENTRYPOINT instruction will be used for lauching the container, more on this we can discuss during ENTRYPOINT

#1 execute form
In this we write CMD we wrote will be passed as json format input to the daemon. if we write CMD in execute form the instructions we wrote will be executed as direct command without any shell. and if we want to execute the instructions with shell we need to specify explicitly.

CMD ["EXECUTABLE", "ARG1", "ARG2"]
CMD ["/bin/bash","-c", "echo", "hi"]

#2. shell form
CMD echo "hi" = The shell form always executes the Command using bash/sh shell only.

RUN directive (image instruction directive)
RUN is used for running any shell command during the time of building docker image. RUN executes on top of docker image and produces a layer image on the base always.

There are 2 forms of writing RUN directive
1. execute form = RUN["executable", "param1","param2"]
2. shell form = RUN command by default it executes the command supplied using /bin/sh -c form

if we want to use bash shell for running command we can use RUN /bin/bash -c "command"

FROM --platform=linux/arm64 image:tag as image
Create our own docker image referring to the base docker image that already exists, it is mandatory and should appear always as 1st line within the Dockerfile (exception: parser directive could exist before FROM).

RUN directive
RUN is an docker image instruction that would be executed at the time of building docker image and we can execute any Shell command using the RUN. Usually it is used for running package managers installing and configuring the software on docker image.
There are 2 forms of RUN command
1. execute form = RUN["executable", "param1", "param2" ...] = here RUN will not runs the command using Shell/bash, it runs the executable we provided directly, so the pipes and redirectors will not work. If we want to run the executable using bash we can use the below syntax
RUN["/bin/bash","-c", "mkdir ~/data"]

2. shell form
The command we supplied always gets executed using shell as default bourne shell
For eg.. /bin/sh -c command

CMD directive
CMD is used for providing default command to be executed while launching the container. We can write any number of CMD directives in a Dockerfile but only the last one would be executed. CMD can be override at the time of launching the container by the user unlike ENTRYPOINT.

In case if we provide both ENTRYPOINT and CMD directives in Dockerfile, then container execution begins with ENTRYPOINT by taking CMD as an argument (will explore later)
We can write CMD in 3 forms
1. execute form
2. shell form
3. arg form to ENTRYPOINT

ENV directive
to define ENV variables as part of the image, so that those values will be available during the container execution.
There are 2 ways we can define env variables.
1. in Dockerfile itself
2. we can pass them as parameters while running the docker container.

#1
Dockerfile
FROM ubuntu:21.04
ENV JDK_HOME=/u01/data/java11

#2
variables.env
JDK_HOME=/u01/data/java11

docker container run --env-file=variables.env envcliimg:1.0
Note:- env variables are not available during build, those are available only at the time of running the container.
---------- ARG
ARG directive is used for passing build arguments at the time of building the image. these acts similar to variables we define any programming.

For eg.. instead of hardcoding the installation of java package with a specific version we can pass the package name as a build argument input

RUN apt install -y openjdk-11-jdk
(instead of the above)

Dockerfile
FROM ubuntu:21.04
ARG JDK_PKG

RUN apt install -y $JDK_PKG

in the above Docker build script we declared an argument to be passed while running the docker image build using which we are installing the jdk package. here we parameterized the image build.

How to declare the arguments in Dockerfile?
There are 2 ways we can use ARG in Dockerfile.

#1 The only directive that can exists before FROM in a Dockerfile is ARG
Dockerfile
ARG IMG
FROM $IMG

IMG ARG cannot be used

In the above we are parameterizing the base image to be used for the build. In this case once the FROM directive has finished execution the ARG value will dis-appear and cannot be used further.

#2 We can declare ARG after the FROM directive to use the ARG as input to perform image building
FROM ubuntu:21.04
ARG JDK_PKG

at the time of declaring the ARG we can supply default value
For eg..
ARG JDK_PKG=openjdk-11-jdk

we can refer the ARG in Dockerfile using ${ARG_NM} expression
For eg.. RUN apt install -y ${JDK_PKG}

In case if we declared ARG without a default value, then at the time of lauching the image build, we should mandatorily supply the values.

Dockerfile (with no default value for ARG)
FROM ubuntu:21.04
ARG JDK_PKG

docker build -t argimg:1.0 . = gives error, ARG JDK_PKG is mantory
docker build -t argimg:1.0 --build-arg JDK_PKG=openjdk-12-jdk .

ARG directive

We can pass data during the time of building the docker image by using ARG, we can think of them as variables in a typical program, where we can substitute them with corresponding values while running the program.
Similar we can declare placeholders in dockerfile and can substitute values while running the docker build, this help us in keeping the changes minimal in a dockerfile.

Note: ARG are build-time parameters,those are not accessible during the runtime of the container, if we want runtime parameters, then we should use ENV

There are 2 ways we can use ARG
#1 before FROM statement
In this case it acts as argument for FROM statement indicating which base image should be used for building the docker image. after the FROM statement, the ARG value will not be available.

#2 after the FROM statement
The ARG defined after FROM directive will be available through out the build file during the docker image build.
while declaring the ARG we can assign default value in which case, the build works without expecting any input to be passed during build
ARG ARG_NM=ARG_VALUE
if case if we have decared the ARG without default values, we should pass values as input while launching the image building
docker build -t img:1.0 --build-arg NAME=VALUE --build-args NAME=VALUE .

LABEL directive
---
just for adding documentation in the image metadata. A LABEL is a key-value pair. where any number of labels can be declared in a Dockerfile

LABEL "VERSION"="1.0"
LABEL "AUTHOR"="SRIMAN"

docker inspect image:tag = displays all the labels you defined

MAINTAINER [deprecated] directive

Just to tell the author of the image, or who is maintaining this docker image.
MAINTAINER NAME
(instead use below)
LABEL maintainer="SRIMAN"

COPY directive
---
if we want to copy any file from docker build context into the docker image we can use copy directive.

railreservation
|-Dockerfile
|-pom.xml
|-target
|-railreservation.war

how to copy the project artifacts into the docker image, so that we can use it?

Dockerfile

FROM ubuntu:21.04
WORKDIR /u01/data
COPY target/railreservation.war apache-tomcat-9.0/webapps

ADD directive

COPY and ADD both works in the same way, but ADD not only help us in copying the building context files into Docker image, it can be passed with remote url, so that it downloads and copies to the image.

Dockerfile

FROM ubuntu:21.04
RUN mkdir -p /u01/data
WORKDIR /u01/data
ADD https://download.java.net/openjdk/jdk11/ri/openjdk-11+28_linux-x64_bin.tar.gz --output openjdk-11+28_linux-x64_bin.tar.gz .

the ADD directive above downloads the tar.gz file and places in WORKDIR

WORKDIR directive
we can set WORKDIR location to a path pointing in the docker image, so all the image instructions written after the WORKDIR works relative from the WORKDIR location only.
we can think of it as a replacment of "cd", we can use WORKDIR directive any number of times in a Dockerfile.

EXPOSE directive

EXPOSE 80/tcp (protocol is optional)

EXPOSE doesnt exposes any of the ports of the container, rather it serves as documentation to the user who is using our docker image, letting him understand which ports of this container should be exposed while running.

The application that is running within the docker container will produce some data during its execution, and by default it will be written on container writable layer. It is not recommended to write the data onto the Container Writable Layer, because of the below.
1. The container writable layer would be destroyed at the end of the container, so we cannot access the data outside the container.
2. The performance will be degraded when we write the data onto Container Writable Layer

It is always recommended to write the data onto the external storage volumes hosted by the docker host machine.
advantages:
1. scalability, we can share the data across multiple instances of the container easily
2. crash recovery: with minimal efforts we can restore another container with the same volume mount, in the event of a container crash.
So it is always recommended to design docker containers as immutable (read-only).

There are 2 types of mounts are supported by the docker
1. bind mounts = we can mount a Filesystem location of a host computer of the docker daemon on to the container physical directory location, so that the programs can store the data into the bind mount location. There are few notable things about bind mounts
- These are completely depenedent on Host Filesystem
- The bind mounts directories can be used by the host computer applications in order to process, apart from containers.

2. bind volumes =

Instead of writing the data onto the Container Writable layer of the docker container, it is always recommended to keep the data externalized by writing into docker volumes. There are 2 reasons for making containers immutable.
1. We can scaleup the containerized application by sharing the common data across the instances through docker volumes
2. crash-recovery: with minimal efforts we can replace a failed container with a newly created container quickly by restoring the state from volume

How many types of volumes are supported by docker?
There are 2 types of volumes are supported by docker.
1. bind mounts
2. docker volumes

#1 bind mounts
we mount a Filesystem directory of the docker host onto a specific location of the docker container.
advantages:-
1. container and other process running on the docker host can access data directly out of docker
dis-advantages:-
1. bind mounts works based on host Filesystem, so that there is a chance of compatible issues might creepup
2. these are poor performant when compared with docker volumes
3. developer has to manually take care of managing the bind mounts like
- create
- migrate
- backup

How to work with bind mounts?
#1 create a directory/folder on the docker host which you would like to mount on docker container.
mkdir -p /u01/data (ensure proper permissions are there)

There are 2 syntaxes in mouting the directory onto the container while running the container
1. -v or --volume (old) (less readable)
sourcedir:targetdir:propagationSettings
the third option is optional but it take possible 3 values
1. ro = readonly
2. z = indicates the same bind mount is shared across multiple containers
3. Z = bind mount content is private and unshared
in case the source directory we specified doesnt exists on host, it creates and lauches the container

2. --mount (newly) (readable) (key=value pairs)
--mount type=bind,source=/u01/data,target=/u01,readonly

propagation settings
shared
private
incase the source directory we specified doesnt exists on host, it gives an error
In both of these cases the destination directory we specified if already exists in the container, then it removes the directory contents and loads the bind volume data only.

#2 docker volumes
Docker volumes are preffered of persisting the data that is generated by the running docker

container. Docker Volumes has several advantages when compared with bind mounts:
advantages:
1. volumes are easier to backup and migrate than bind mounts
2. You can manage docker volumes through docker cli commands
3. Volumes work with both windows and Linux operating system
4. Volumes can be more safely shared among multiple containers.
5. Volume drivers can be used for storing the volumes at different places like cloud or nfs location etc
6. docker volumes are highly performant when compared with bind mounts.

There are 2 ways of working with docker volumes are there
1. we can create docker volume manually on the docker host and can mount on the container, these are called named volumes.
2. we can declare directly the volumes as part of the Dockerfile, in this case the volumes are automatically created by daemon using an random hashvalue to identify rather than use a volume name.

How to create docker volumes and manage them?
docker volume create volumename = creates an named volume under /var/lib/docker/volumes
docker volume ls = to list all the docker volumes managed docker daemon
docker volume inspect volumename = to see the details of the volume
docker volume rm volumename = to delete a volume

For eg.. we can create a volume as below
docker volume create vol1 (default: /var/lib/docker/volumes/vol1)

#1 approach one of working with docker volume (manually creating)
To create container using this volume we can use 2 syntaxes
1. -v or --volume
volname:targetDirectory/propagationsettings (ro/z/Z)
docker container run -v vol1:/u01 image:tag

2. --mount
--mount type=volume,source=vol1,target=/u01

#2 directly we can declare volumn in Dockerfile using VOLUME directive.

FROM ubuntu:21.04
VOLUME["/u01"]

always VOLUME directive creates a new volume on the host,so avoid using it.

How to write the data into docker volume?
#1 create a container out of the volume we created
for eg..
docker container run -v vol1:/u01 --name=container1 img:tag
while the container is running we can use docker cp command to copy data into that volume of the container in which we mounted.
syntax:-
docker cp source containerName:location

docker cp sourcefile container1:/u01

ENTRYPOINT directive
ENTRYPOINT is a docker directive or docker instruction we can write as part of Dockerfile to build image. we use ENTRYPOINT directive to build executable docker containers, which means when the end-user lauches a docker container which has ENTRYPOINT instruction as part of it, the container execution will begin with executing the ENTRYPOINT only.

In case of CMD we can always override as known earlier. but an ENTRYPOINT cannot be overriden while lauching the container.

Along with ENTRYPOINT we can use CMD also as part of a Dockerfile, there are many combinations of using these 2 directives togethers lets explore.

#1 CMD only without ENTRYPOINT
CMD can be used directly as part of Dockerfile without using ENTRYPOINT, in case if we use CMD without ENTRYPOINT, when we launch docker container, it begins the execution from CMD directly, few notable things here in using CMD
1. We can have multiple CMD directives in a Dockerfile, but always only the last one will be executed
2. We can always override the CMD instruction while launching a docker container by passing command
3. In case if we want to run multiple instructions in CMD we have 2 options
3.1 place them as part of shellscript and let us invoke or launch the shellscript within CMD
3.2 we can concatenate multiple shell commands using & in CMD as shown below
CMD sh1.sh && sh2.sh

#2 CMD along with ENTRYPOINT
We have 2 options while using CMD along with ENTRYPOINT,
2.1 we can write CMD before ENTRYPOINT
if we write CMD before the ENTRYPOINT the CMD will be ignored and will not be executed.

2.2 we can write CMD after ENTRYPOINT
if we write CMD after the ENTRYPOINT, the CMD acts as an argument into the ENTRYPOINT instruction.

2.2 we can write CMD after ENTRYPOINT = EXAMPLE
Dockerfile
FROM ubuntu21.04
ENTRYPOINT ["vmstat"]
CMD ["-s"]

#3 How to build an docker image with ENTRYPOINT instruction and make the container accessible interactively?

Dockerfile
FROM ubuntu:21.04
apt update -y
apt install -y openjdk-11-jdk
RUN mkdir -p /u01/data
WORKDIR /u01/data
COPY target/app.jar .
ENTRYPOINT["java","-jar", "app.jar"]

docker build -t japp:1.0

docker container run japp:1.0
docker container run -it japp:1.0 /bin/bash = In this case since we are using ENTRYPOINT for running the container, we cannot override the ENTRYPOINT by using -it and command

But along with ENTRYPOINT we can execute additional instructions or grab container iteractively by using the below technic

startup.sh
set -e
nohup java -jar app.jar &
exec "$@" # all arguments we passed

Dockerfile
FROM ubuntu:21.04
apt update -y
apt install -y openjdk-11-jdk
RUN mkdir -p /u01/data
WORKDIR /u01/data
COPY target/app.jar .
COPY startup.sh .
ENTRYPOINT ["startup.sh"]

docker container run -it japp:2.0 /bin/bash

ENTRYPOINT directive

ENTRYPOINT instruction written in the Dockerfile will be executed at the lauch of the docker container, it is used for building executable containers, which means if we have packaged an application which always has to start executing with entry instruction always generally we use ENTRYPOINT.

ENTRYPOINT instruction cannot be overriden by the developer at the time launching the container unlike CMD.

We can use both ENTRYPOINT and CMD together in our Dockerfile, but behaviour interms of execution would result differently by the way those are applied. Lets see the combination of these 2 instructions

1. CMD only without ENTRYPOINT
- it acts as a default command in launching the container
- we can write multiple CMD but only the last one will be executed and others are ignored
- we can override the CMD during the launch of container

2. CMD with ENTRYPOINT
Here there are 2 combinations of using CMD with ENTRYPOINT
1. Before ENTRYPOINT use CMD
- CMD will be ignored

2. After ENTRYPOINT use CMD
- If we use CMD after the ENTRYPOINT, then it acts as an parameter to the ENTRYPOINT instruction

Note:- we should write ENTRYPOINT and CMD in executable form to make CMD as argument ENTRYPOINT instruction.

Dockerfile

```
FROM ubuntu:21.04
ENTRYPOINT ["vmstat"]
CMD ["-s"]
```

ENTRYPOINT with interactive shell
While using ENTRYPOINT instruction we cannot override by running container iteractively. even we pass command while lauching the container it acts as arugment to the ENTRYPOINT instruction.
Then how to launch docker container interactively?

startup.sh
```
#!/bin/bash
set -e
// write the code you want to execute first but launch it as a daemon
echo "$@"
```

Dockerfile
```
FROM ubuntu:21.04
ENTRYPOINT ["startup.sh"]
```

How to install apache2 server and access the pages from the host?

Dockerfile

```
FROM ubuntu:21.04
RUN apt update -y
RUN apt install -y apache2
ENTRYPOINT ["systemctl","start","apache2"]
```

```
docker build -t apache2:1.0 .
docker container run apache2:1.0
```

Docker container?
A software process/program packaged into a docker image , which is under execution. once the software program within the container finished execution, the container automatically terminates

The above behaviour holds good when we are running standalone applications. but what about the applications deployed on the containers and started as daemon services (background process)

For eg.. we installed tomcat server and deployed our war into webapps directory and started the tomcat as a systemd (daemon). we expect the container to run for infinite time until we stop, but as we launched tomcat as systemd service, once it started the container terminates as nothing is holding container to wait for termination.

So how to run daemon servers within the docker container for infinite time?
In linux we have a file called "/dev/null" = blackhole file or null file
anything we write into file will be discarded. similarly we can read the data from this file for infinite time and will return infinite NULL

```
FROM ubuntu:21.04
RUN apt update -y
RUN apt install -y apache2
EXPOSE 80/tcp
ENTRYPOINT ["systemctl apache2 start"]
CMD ["tail -f /dev/null"]
```

Now we can run this container as a daemon container as it runs for infinite amount of time.

Here by default apache2 executes on 80 port and apache2 is available and only accessible within the container only. if we want to access the containerized applications using their port outside the container from docker host then we need to publish the port to the host

container publish port (port forwarding)

When we run containerized application within the docker container, by default the application is accessible within the container via the port. but if we want the application to accessed from docker host then we expose/publish the port to the host using -p option.

```
docker container run -p 80:8080 apache2:1.0
```

http://localhost:8080

HEALTHCHECK directive
The HEALTHCHECK instruction tells the docker daemon how to verify the containerized application is still working or not. Many times when we package and start the application in a container, the container reports the status as UP. but the underlying application may not be accessible due to various reasons like
1. application deployment might be failed
2. server might went into unresponsive state due to stuck threads

identifying such containerized applications and monitoring them would be tricky, the docker help us in monitoring using HEALTHCHECK CMD we supplied.

application development:-
The developer of the application should be responsible for exposing an (for eg.. /health) HealthCheck endpoint upon accessing should give us a success response with status : 200 (OK), so that we can use this endpoint to determine application is working or not.

devops engineer:-
while building the docker image, in Dockerfile instruction file he has to write a HEALTHCHECK instruction to access the above endpoint provided and verify the status of the actual application running in the container.

if a docker container with HEALTHCHECk endpoint written along with UP as a container status beside it starts displaying (healthy) (unhealthy)

There are 2 forms of HEALTHCHCK instruction:
1. HEALTHCHECK [options] CMD command
2. HEALTHCHECK NONE (disable the healthcheck if anything comes from base image)

The options can be as below.
1. --interval=DURATION (default: 30s) = repeated interval of time to run the CMD command
2. --timeout=DURATION (default: 30s) = how long we should wait for a response before making as failure
3. --start-period=DURATION (default: 0s) = how long we should wait after the container is up before sending executing our HEALTHCHECK CMD to verify the health
4. --retries=N (default: 3) = number of tries/attempts to be made before reporting the container as unhealthy

for eg..
HEALTHCHECK --interval=2m --timeout=30s CMD curl -f http://localhost:8081/health || exit 1

docker networking

The important feature that makes the docker more powerful is the networking. using docker networking we can inter-connect a docker container with host or other docker containers and can build clustered containerized applications.

The docker supports 5 types of docker networking, and the support for networking works in docker based network drivers. if we want we can write our own driver and we can plugin our networking configuration into docker daemon.

by default when we install docker on our machine the bridge and host network will be configured and available for usage.

1. bridge = when we run a docker container without any network mode specified by default the container will be launched under bridge network.
What is bridge network?
If we want to isolate our container from other containers on the docker daemon we can use bridge network. So, the group of containers connected to the same bridge can communicate and can be isolated from the rest.
As discussed above by default during docker installation one default bridge will be created. if we want developer can create his own bridge with which he can launch a docker container

What is the different in using default bridge and user-defined bridge in running containers?
- if we run a docker container without any network mode, by default the container will be launched on default bridge network, so that all such containers which are created attached with default bridge can communication with each other (security problem). instead it is always recommended to create your own bridge network and launch the container on your network.

- in case of default bridge, the containers has to access each other using ip address by running docker container inspect and see the ip address and use it.
java container -> mysql container (through ip)
if we use our user-defined bridge network dns resolution will takes place automatically, lets ways we launched 2 containers with name java and web, the each of these containers can talk directly using their name no need of ip address.
- we can attach and detach a container on fly if we use user-defined bridge network but we cannot do the same with default bridge.
- if we use user-defined bridge network we can modify the iptables and network configuration with which we can launch container based on application requirement, where as if we use default bridge even though we can configure the network but the changes we made will affect all the containers running on the same bridge.

How to manage a user-defined bridge?
1. docker network create bridgeName
2. docker network rm bridgeName
3. docker network disconnect bridgeName containerName
4. docker network connect bridgeName containerName

how to launch a container in existing bridge network?
docker container run --network bridgeName --publish 8081:8080 healthcare:1.0

2. host = if we want to run docker container directly on the host networking only without isolating from the host network, we can use host. this means on the same network or ip address of the host computer the container is exposed. If we have standalone containers we can remove network isolation from the host using host network.

if we are using the host network the -p or --publish options would void.

docker container run --network host img:tag

3. overlay
If we have 2 different docker containers running across the docker daemons we can use overlay network driver for inter-connecting these containers across the workstations.

4. mcvlan = allow you to assign a mac address to a container making it appear as a physical device on your network. now docker daemon routes the traffic to the container based on mac address.

5. none = disable network for this container.

jfrog artifactory
- In a organization we setup our own corporate repository rather than allowing the developers/ others to directly connect to public repositories to manage the project dependencies
- distribute artifacts/dependencies across the teams
- public and distribute internal dependencies between the teams of the organization
- release management

jfrog artifactory is a repository management software, using which we can create repositories for different teams and distribute artifacts and publish the different stages of the produced artifacts at
- dev
- stage
- production
for facilitating the release management

jfrog artifactory supports three types of repositories
- local repositories = publishing the produced artifacts of the project
- remote repositories = developers during their development they need dependencies as part of their project which can downloaded/pulled from remote repository -> always proxies to an public repository (maven central, jcentral), helps us in speeding up the build since the dependencies are cached and are distributed across the team locally
- virtual repositories = virtual repository aggregates both local and remote repositories

rconnect-localdev-repo
rconnect-qa-repo (local)
rconnect-stage-repo (local)
rconnect-release (local)
rconnect-remote-repo

rconnect-distribution-repo
-rconnect-local-dev-repo
-rconnect-remote-repo

jfrog = The provide software tools for release management, end-end application delivery. using these tools an organization can implement ci/cd workflow easily. There are many tools provided by jfrog

jfrog platform (full suite)
- dependency management of various remote repositories
- container registry for distributed container images
- release management pipelines
- security scanner
- distribution

jfrog artifactory (commercial) | jfrog artifactory oss (only jcentral, maven remote repository proxying)
- dependency management of various remote repository
maven, jcentral, bitnary

jfrog container registry (open source)
- container images of the organization can be distributed locall across
(docker, helm)

jfrog artifactory / container registry supports three types of repositories
- local repository = publishing the produced artifacts so that it can be distributed across various stages of our project delivery
-local-dev (alpha/beta)
-qa (beta)
-stage
-release

- remote repository = proxy the public repository and cache locally, so that we can improve build performance

- virtual repository = hybrid of both,

---

to setup artifactory-oss or artifactory-jcr we need a hosted domain only. if we dont have the machine attached domain name, we can install apache2 server with virtual host configured locally on the same machine where we are running the above softwares.

setting up artifactory-oss.
#1 download jfrog artifactory-oss software
https://jfrog.com/open-source/

#2 guznip and tar extract into /u01/app directory
before proceeding further in running the artifactory setup the apache2 server with virtualhost configuration. repo.rconnect.org

#3 install apache2 server
sudo apt update -y
sudo apt install -y apache2

now we need to write virtualhost configuration on the ServerName: www.repo.rconnect.org with proxyPass configuration pointing to the ip and port on which artifactory oss server is running.

here apache2 is not playing the role of webhosting server rather he acts as a proxy for a back-end server.

We configure apache2 server on port: *.80 with ServerName: www.repo.rconnect.org up receiving the request proxy this request (pass) the request to backend server (artifactory oss) running on http:

sudo a2enmod proxy
sudo a2enmod proxy_http
sudo a2enmod proxy_balancer
sudo a2enmod lbmethod_byrequests

/etc/apache2/sites-available/repo-rconnect.conf
<VirtualHost *:80>
ServerName www.repo.rconnect.org
ServerAdmin webmaster@localhost
ProxyPreserveHost on
ProxyPass / http://127.0.0.1:9091
ProxyPassReverse / http://127.0.0.1:9091
</VirtualHost>

How to install and use Jfrog Artifactory oss?
To install jfrog artifactory oss and use it we need apache2 with virtual domain configured with proxy pass should be setup.

steps for installing and configuring apache2 server.
---
1. sudo apt update -y
2. sudo apt install -y apache2

3. enable proxy modules in apache2 server
- sudo a2enmod proxy
- sudo a2enmod proxy_http
- sudo a2enmod proxy_balancer
- sudo a2enmod lbmethod_byrequests

4. create apache2 virtual host configuration file for proxy pass configuration
/etc/apache2/sites-available/
- create a file for eg.. repo-rconnect.conf with below contents
<VirtualHost *:80>
ServerName www.repo.rconnect.org
ServerAdmin webmaster@localhost
ProxyPreserveHost on
ProxyPass / http://127.0.0.1:9092/
ProxyPassReverse / http://127.0.0.1:9092/
</VirtualHost>
the port on which the jfrog artifactory ui is configured by us is "9092" always it should point to frontend port of jfrog artifactory
5. vim /etc/hosts
add domainname with loopback ip
127.0.0.1 repo.rconnect.org
6. sudo a2ensite repo-rconnect
7. sudo apt reload apache2
8. sudo apt restart apache2

Steps for setting up JFrog Artifactory OSS Repository.
#1 create /u01/app directory
mkdir -p /u01/app

#2 download JFrog Artifactory OSS software binary
https://jfrog.com/open-source/#artifactory
https://releases.jfrog.io/artifactory/bintray-artifactory/org/artifactory/oss/jfrog-artifactory-oss/[RELEASE]/jfrog-artifactory-oss-[RELEASE]-linux.tar.gz

#3 copy the tar.gz into app directory, gunzip and tar extract it
/u01/app$ gunzip jfrog-artifactory-oss-[RELEASE]-linux.tar.gz
/u01/app$ tar -xvf jfrog-artifactory-oss-[RELEASE]-linux.tar
it will extracts into below directory.
artifactory-oss-7.18.6

#4 change default port configuration
by default we dont have system.yml in artifactory-oss-7.18.6/var/etc. but there is a template file with name system.basic-template.yaml.
we should copy the contents of the file into system.yml under artifactory-oss-7.18.6/var/etc and add the below configuration at the end of the file

system.yml

@formatter:off
JFROG ARTIFACTORY SYSTEM CONFIGURATION FILE
HOW TO USE: comment-out any field and keep the correct yaml indentation by deleting only the leading '#' character.
configVersion: 1
artifactory:
port: 9091
router:
entrypoints:
externalPort: 9092

#5 goto /u01/app/artifactory-oss-7.18.6/app/bin
./artifactoryctl start/stop to start and stop the artifactory oss server

#6 open the browser and type http://repo.rconnect.org -> should open the jfrog artifactory home page asking for changing the default admin user and password
default: admin/password

Creating Repositories for the project
once after the login into jfrog artifactory ui, click on administrator in the left panel. and expand repositories and click on repositories.

In general There are 2 types of repositories are required for maven projects.
1. artifact (just) repositories
The dependencies that we use as part of our project, those are jar dependencies usually. the artifact repositories are nothing but jar dependent repositories

when you build a java project, you produce the project as an published or distributable artifact, which is for eg.. rconnect.war. if we want to distribute this war within our project during different stages of delivery we need to publish into artifact repositories

2. plugin repositories
building blocks of maven build system, they perform actions as part of the maven build like maven-compiler-plugin compiling the source code, maven-jar-plugin for packaging the project as jar etc.

sometimes based on the requirement, in a project we might build a custom maven plugin of our own, how to distribute this maven-plugin during various stages of deployment and delivery of the project. publish the plugin into plugin repositories

While setting up repositories for the project in artifactory we need to create 2 types of repositories as described about
1. repositories
2. plugin-repositories
These can be distinguished only through name, underlying both the repositories are same.

To publish and distribute project artifacts of several stages of our project we need to create 2 repositories for each of the above
1. snapshots
2. release repositories

In JFrog there are 3 types of repositories are there
1. local repository = publishing the project artifacts so that it can be distributed across the delivery stages
2. remote repository = proxy the remote repositories
3. virtual repository = combination of local and remote, so that first time when we conduct the build virtualrepo redirect to remote and cache it locally, 2nd time onwards the builds will be done from local repo.

For a project we need to setup
repository, plugin repositories for both snapshot and releases of type local, remote, and virtual repo as shown below.

For eg.. lets a project rconnect we can setup below repositories

artifact repositories
---
local-repo-rconnect-snapshot
local-repo-rconnect-release

plugin repositories
---
local-pluginrepo-rconnect-snapshot
local-pluginrepo-rconnect-release

remote-repo-rconnect-snapshot
remote-repo-rconnect-release

virtualrepo-rconnect-snapshot
local-repo-rconnect-snapshot
remote-repo-rconnect-snapshot

virtualpluginrepo-rconnect-snapshot
local-pluginrepo-rconnect-snapshot
remote-repo-rconnect-snapshot

virtualrepo-rconnect-release
local-repo-rconnect-release
remote-repo-rconnect-release

virtualpluginrepo-rconnect-release
local-pluginrepo-rconnect-release
remote-repo-rconnect-release
---
after creating these repositories goto application tab on left navigation bar expand aritfacts and click on setup right top corner
select appropriate release, snapshot repositories as we created above, enter password in box and click on generate. will create settings.xml

when we run the maven build in our project
~/railworkspace
|-rconnect
|-pom.xml
mvn clean verify = by default maven build system goes to maven central repositories for resolving dependencies and plugins we defined in pom.xml.

to let the maven build goto artifactory repositories we setup now we need place settings.xml in ~/.m2 directory of the user

In the settings.xml we define repositories which are repositories and plugin repositories pointing to the jfrog repositories we created along with username and password to connect.

now after setting up the settings.xml in ~/.m2 directory if we perform maven build it starts pulling dependencies and plugins from our artifactory repositories only.

if we want to publish the build artifacts into local repositories of the artifactory we need to configure distributionManagement section under pom.xml, for different stages we can create different profiles pointing to snapshot and release repositories.

<profiles>
<profile>
<id>snapshot</id>
<distributionManagement>
<repository>
<id>snapshots</id>
<uri>local-repo-rconnect-snapshot</uri>
</repository>
</distributionManagement>
</profile>
<profile>
<id>release</id>
<distributionManagement>
<repository>
<id>central</id>
<uri>local-repo-rconnect-release</uri>
</repository>
</distributionManagement>
</profile>
</profiles>

mvn clean deploy -P snapshot/release

How can we distribute docker images that are build out of the project, to various different stages of our application development and delivery. how can we achieve ci/cd for an application?
The one way we can achieve is through docker hub repository, the developer build container images can be published in docker hub repository, so that it can be pulled at various different stages of the development/delivery phase and use it.

But there are several problems with this approach.
1. by default all the images that are published into docker hub are public and anyone can use it
2. only one single private repository is permitted for one account and that is of non-commercial usage by docker hub
3. pushing and pulling the container images onto the docker hub takes lot of time due to the internet connection and bandwidth speeds and delays the delivery of application.

So, instead of using docker hub repository, we can setup our own internal organizational container registries and there are several container registry products are available in opensource and commercial as well.
We can host these container registries within organization, publish and distribute project container images internal and for implementing ci/cd.

JFrog artifactory fcr = it is mandatory to have domain name associated to do dns resolution while pushing and pulling container images from the container registry. If we are hosting the container registry locally, then we need to atleast setup virtualhost with /etc/hosts domain name resolution.

JFrog Container Registry supports 3 types of image repositories
1. local repository = distributed container images we produced out of the project locally
2. remote repository = proxy of the docker hub repository
3. virtual repository = combination of both local and remote repository

8081/8082

Managing the IT infrastructure is always challenging in the software engineering world. during the time of deployment of an application, the IT Engineering team has to procure the Machine to run the application.
In most of cases the Engineering team is forced to take big machines of huge capacity where in reality only 5% - 10% machine capacity is being used in running the application.
We can run only one single environment on one machine, which indicates we cannot run multiple software applications on the same computer, so for another application procure one more machine, this increases the IT Infrastructure cost in multiplication factor of number of application to be deployed.

To rescue or solve from the above problem, the virtualization technology came into market. Using Virtualization we can run multiple different software applications of various technologies on a single server environment.

Virtualization works on a software called Hypervisor, which partitions and allocates the hardware resources of the computer across multiple Virtual Machines running, so that in a isolated environment each of the software application can run on its own operating system.

Being the fact that every virtual machines has an operating system installed and running on it, which makes the overloaded. Due to an full operating system running in each virtual machine the number of software applications we can run in parallel on a computer system is being Limited.

How to solve the above problem, that is where Containerization technology has been brought. From the above it looks like Containerization tools are light weight and has lot of advantages when compared with Virtualization technology, but from reality each has their own advantages and disadvantages.

Virtualization
advantages:-
1. Each of the Virtual Machines can have their own operating system being installed and can run irrespective of the host operating system on top of which there are running.
2. The virtual machines running in virtualization technology are highly secured because every vm run out of its own operating system independent of the host and others. So the host has been compromised, there is no impact on the virtual machines running on the host.
3. Ideally suitable for software applications that requires a full operating system services.
4. Suitable for running appliations for longer time.

dis-advantages:-
1. as each virtual machines runs out of its own operating system, these are resourceful and are considered to be heavy weight. Each operating system consumes heavy system resources in running byitself rather than the application programs.

2. Patching and upgradation has to be done for each of the machines as they run on independent operating systems which increases the maintaince cost and efforts

3. The virtual machines runs out of virtual machine disk images, as the full operating system is being installed on a virtual machine, the disk images are usually huge in size, due to this we cannot carry a vm disk image from one env to another in recreating the ci/cd pipeline infra.

4. The virtual machines takes huge time in bootup, as to start running the programs on the virtual machine first the operating system should be started which itself takes lot of time and are not ideally suitable for high scalable solutions.

Containerization:-

advantages:-
1. The containers are light weight packaged with only required software packages and applications to run in an isolated env one from another, each container doesnt have operating system running as part of it, so when compared with vm the containers takes less number of system resources allowing us to run more software applications on the same hardware.
2. Containers runs out of container images. as the operating system is not packages as part of the container, the images are very small in size and can be easily shippable. For eg.. a container image can be as less as 2 mb also. implementing ci/cd is very easy in containerization technology
3. Quickly bootable makes them suitable for scalability
4. patching and upgradation is easy, if we patch the host operating system, it reflects across all the running containers.

dis-advantages:-
1. Not suitable for applications which requires an dedicated operating system
2. Not secured, if the host has been compromised then all the containers will be exposed
3. Usually not meant for running the applications of longer duration

In most of the cases the combination the above 2 are being used for the IT organizations right now.
To run the containers we need Containerization Tool/Engine to be installed on a computer. How can we have several computers in hand with Containerzation engine installed that is where people use Virtualization Technology to create computers.
To run software applications on top these vms people use containerization technology.

What is Virtualization, what is the purpose of it?
We can run multiple parallel isolated environment on a single computer system hardware by sharing the underlying hardware resources of the computer to multiple virtual machines through hypervisor software is called "Virtualization".
In each of the Virtual Machines a dedicated operation system will be installed and used.

What is Containerization, what is the purpose of it?
We can package software libraries and software applications and can execute in an isolated manner from one program to the another program within the environment is called "Containerization".
In each container it has only the bins/libs enabling it to talk to the host operating system, due to this containers are considered as light weight processes running on the host computer.

virtualization:
advantages:
1. dedicated operating system so suitable for applications requires an entire operating system to run
2. we can run any operating irrespective of the host operating system
3. highly secured
4. must suitable for longer time of execution

dis-advantages:-
1. heavy weight because of dedicated operating system
2. not carriable across the env because of the vm images are of huge in size, so tough to achieve ci/cd
3. upgrading and patching takes more time as individual virtual machine operating systems has to be patched separately
4. virtual machines takes more time in booting, so those are not ideally suitable for achieving high-scalability.

containerization:
advantages:-
1. very light weight, which allows us to run multiple programs on the underlying machine
2. the images of the containers are very small in size that make them easily shippable, due to which we can achieve ci/cd easily
3. quickly bootable, so suitable for scalability
4. easy to patch and upgrade, if you patch the host operating system , it reflects across all containers
dis-advantages:-
1. cannot run a different operating system across the containers independent of the host
2. less secured, if host compromises the containers as well
3. not suitable to be used for long-period execution.

What is docker?
Instead of packaging and running a dedicated operating system on each the virtual machine environments, is there a way we can just only package software application and dependent software libraries and run the software application isolated from another software program on the same platform, that is where "Containerization Technology" has been brought.

Docker is a word derived from "Dock Worker", a person who employed in a port to load and unload the ships. and the company has been transformed into Docker Inc.

Docker is a containerization tool using which we can package software programs and their

dependent software libraries together and can ship across environments. There are 2 main advantages of containerization technology.
1. These are easily shippable
The container images are packaged with libraries, software application and instructions in running the software application, thus helps us in abstracting the software delivery process.
1.1 Now the devops engineer can easily run any application without bothering about the technology in which the application was built, standardized workflow in running the application
1.2 since container images are very small in size, we can achieve ci/cd easily

2. Software programs can be ran isolated from another software programs of the same env
2.1 with minimal hardware resources required a software application can be executed in an islated manner using containerization technology. so we can effectively use the underlying hardware resources of the computer.

How does containerization technology works?
The Linux operating system plays a big role towards achieving the containerization technology, many software gaints in the market especially Google Llc. has contributed much of its core to the Linux operating system in build Containerization technology into reality.

What is docker?
Docker is a containerization tool that will allow us to package software applications with their dependent libraries so that those can shipped across the environments. it abstracts the software delivery process of various different types of applications by providing an uniform workflow in running the docker containers. In addition to the above docker allows us to run multiple applications on the same env isolated from one with the another.

In order to run programs isolated from one with another in the same environment, docker requires/depends on few features of Linux operating systems, organizations like Google and others has contributed alot in adding necessary features to the Linux kernal to make containerization technology work.

Following are the features of Linux that docker uses to work in isolation manner.
1. Linux Namespaces = docker containers works on the concept of Linux namespaces, so that it can run mutiple programs isolated from each other. when we run a docker container, docker creates a separate linux namespace in which the container will be executed and limited to be accessible within the namespace only. as each containers runs within its own linux namespace, the containers are isolated from each other.

Docker uses the below namespaces of Linux
1. pid = Runs the container process in separate namespace
2. net = Managing network interfaces
3. ipc = inter-process communication
4. mnt = filesystem mount points
5. uts = unix timesharing

2. Control Groups (cgroups) = docker engine uses the cgroups in controlling and limiting the applications of the container to use specific resources of the machine

3. Union Filesystem = container images of the docker are assembled or created based on Union Filesystem

---

What is the Architecture of docker?
There are 3 parts are there in docker
1. docker engine / daemon = its a long running process that runs a machine which is called "docker workstation". it is responsible for creating and running docker images as container processes on the computer.
It again containers 3 subparts inside it
1.1 docker daeomon = exposes rest api to the clients so that they can communicate with docker engine
1.2 containerd = is a component that is responsible for passing docker images to runc in a oci {open container intiative} image format. it takes care of various other responsibilities like pulling docker images from repository, managing the containers etc
1.3 runc = is the low-level component interacts with the Linux operating system in creating and destroy containers out of a image

2. docker hub repository
The place where docker images are kept and distributed across

3. docker cli
Command-Line interface tools through which we can interact with docker engine from an remote computer.

How does docker works?
For docker to run containers in isolated from one to the other it uses many features of the Linux operating system.
1. Linux Namespaces = below namespaces of the Linux are used by the docker to run containers in an isolated manner
- pid
- net
- uts
- ipc
- mnt

2. CGroups = helps the docker in controlling the allocation of hardware resources to the containers
3. Union Filesystem = is used for building and distribute docker images
---
Architecture of docker
---

1. Docker Engine
The machine on which the docker is running is called "Docker Workstation". It is the linux daemon process that is running on the Linux machine and manages the docker containers.
There are three components are there in docker engine
1.1 docker daemon = exposes rest api to let the docker clients to communicate with docker engine
1.2 containerd = is responsible for converting docker images into oci (open container initiative) specification image format and passes to runc to create a container. it takes care of managing the images, pull from docker hub repo and managing the running containers.
1.3 runc = is responsible for starting/creating docker containers and stoping/destroying docker containers on linux
2. docker hub repository = is a place where docker images are published and distributed
3. docker cli = a command-line interface which will be used by clients/ops engineers in communicating with docker engine asking to run containers and manage them
---
What are docker objects?
There are 2 main objects are there in docker

1. docker image
docker image is a file (similar to a virtual machine image file) on which the software application and its dependent software libraries along with libs/bins required to communicate with host operating system of the docker workstation are packaged inside it. From a docker image we run a docker container out of it.

docker images are mostly created from a base docker image, there are several base images pre-baked and are published as part of docker hub repository. The quickest way to package and ship our applications are extending from one of the docker base images.

The docker base images containers low-level bits like libs/bins that are required for running the container on the docker workstation. There are plenty of docker images are available on docker hub be cautious while choosing the base images
We can categorize the images into 2 groups
1. Official images = trusted and can be taken as base images in baking our applications
2. Open source images = these images are contributed by any third-parties or individuals and cannot be trusted and should use at your own risk.

docker team has provided a base image called "alpine" which is of 2mb in size, which can be considered as a the best source in creating our own docker image. similarly the debain has contributed in publishing their own images which is ubuntu-20.04 image can be used for baking our images.

one of the key characteristics of docker images are once they are build those are read-only unlike your virutal machine images.
Refer:- to vmimage-vs-dockerimg.jpg

docker images are layered and stacked one ontop of another:- , so that docker images can built by sharing common base image layer and can save storage space in keeping images on disks. apart from that we can distribute docker images quickly.

There are 2 aspects of docker image
docker images are read-only:- to share same images in running multiple containers. The running containers might produce some program data and those are written ontop of "Container writable layer". Docker uses storage drivers in writing the data produced by the programs into writable layer.

docker images are stackable :- so that we can save storage space in keeping multiple images on disk and quick to distribute. docker uses a special filesystem format called "Union Filesystem" for stacking the docker images.

2. docker container
a running process/instance that is created out of a docker image is called docker container. using docker cli we can create/start/stop and destroy a docker container.

How to install docker?
docker supports both windows/linux/mac platforms as well.
1. earlier docker doesnt works on windows as it greatly depends on Linux features like namespaces, cgroups and filesystem formats. Microsoft has worked heavily on bringing docker on to the Windows platform. So from Windows 10 premium >above versions we can run docker

we need to install docker-desktop-windows to run docker on windows. the cli commands and docker images differs greatly from linux (note). usually the size of the docker images are also heavy when compared with linux images.

2. if we have windows 7.1/windows 8/windows 10 (non premium/ultimate) editions how to install docker.
docker-toolbox-for-windows software is available.
docker cli is installed on host-machine and docker daemon is installed on ubuntu linux operating system, so that when we run the docker command a docker daemon will be started and will run container inside it.
---
3. For linux platforms docker has provided docker-desktop-ubuntu
4. For mac docker-desktop-mac

Let us install docker on ubuntu virtualmachine.
We have to install Docker from Official Docker Repository. To do that, we will add a new package source (repository), add the GPG key from docker to ensure all the downloads are valid.

#1
sudo apt update

#2 install pre-requisite software packages for installing docker. will let us download apt packages over https
sudo apt install apt-transport-https ca-certificates curl software-properties-common

#3 Then add the GPG key for the official docker repository to our system.
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

#4 Add the docker repository to the apt sources:
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu focal stable"

#5 update the package database with the newly added docker repo
sudo apt update
sudo apt-cache policy docker-ce (optional)

#6 install docker packages
sudo apt install -y docker-ce (default it install all docker packages if req)
docker-ce-cli and containerd.io

docker has been distributed in 2 additions
1. community edtion = open source
2. enterprise edition = commercial

#7 check docker is running as a service
systemctl status docker

by default docker is installed and configured to be used with root permissions only, so we need to add our system user/logged in user to docker group.

when we install docker it creates a default group called "docker" and gives permissions to lauch process of these groups of users only. rwx-rwx- - - -
we need to add the user to the docker group.
sudo usermod -aG docker $USER

if you still face an issue while running docker we can give permission to the sock file.
sudo chmod 777 /var/run/docker.sock

sudo systemctl status docker
sudo usermod -aG docker $USER
sudo chmod 777 /var/run/docker.sock

docker image ls

How to work with docker or docker workflow?
There are lot of pre-built docker images are there published in docker hub repository, to start with, we can easily use one of docker images that are already available to understand workflow.

#1 docker container is nothing but a process under execution out of a docker image, so when we run a docker container from an image, the docker daemon check for the image is already available in the local image cache (local machine), if present, the it quickly runs a container out of the image.

#2 if the image is not found in local image cache, then it by default connects to docker hub repo and download the image into local system cache

#3 creates an container instance on the image, so that container runs the program based on the instructions written as part of the image.

#4 once the software program finishes executing within the container automatically the docker container will be stopped and status will reported as exited.

lifetime of the docker container:= container exists until the program running inside it completes execution.

How to run a docker container from an image?
We can launch a docker container using docker container cli command.

docker container run -it image:tag command
-it = interactively (let me get inside the container)
image:tag = look for the docker image in docker hub repo, tag specifies the version of the image
command = which command we want to run inside the container

docker container run -it ubuntu:21.04 /bin/bash
The above command downloads ubuntu:21.04 from docker hub repository https://hub.docker.com , and places the image into local image cache
creates the container out of the image and takes us into the bash prompt of the container. once we exit from the bash prompt, as the (bash) program exited, it terminates container automatically.

How to see a running container?
docker container ls = is the command that displays the containers currently "Up".
if we want to see all the containers which are Up and Exited we need to use -a flag
docker container ls -a

These are the basic commands in running a docker container.

How to manage images in the local image cache?
image management commands:-
---

1. docker image ls = shows all the images that are there in the local system image cache. It displays the below information about the image that exists

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	21.04	01244e8938d1	3 days ago	74.1MB

REPOSITORY:= repository name
TAG:= version of the image, every time when a image has been updated, the author publishes the image with updated version (tag), without overwriting the previous image.
by default for every image, it is common practise to create a tag called "latest" and always the latest will be pointed to latest version of the image. but there is no guarantee that latest always points to latest version.
IMAGE ID:= is a hashvalue generated by docker based on the contents image to identify whether 2 images are same or not.
CREATED:= when does this image has been created by author
SIZE:= size of the image

2. we can pull the image from docker hub repo without running a container on it.
docker image pull image:tag - this command pulls the image and stores in image cache
for eg.. docker image pull ubuntu:21.04

3. we can remove a docker image from the local system image cache
docker image rm image:tag, for eg.. docker image rm ubuntu:21.4.
For the above image if there are any containers exists out of it the command fails. so first we need to remove containers and then remove the image.
docker container rm containerNm/containerId
(or)
use -f = force flag to remove containers along with image
docker image rm -f image:tag = removes containers on that image and image as well.

but if there exists any container in Up status again the command will fail.

4. docker image save image:tag -o image.tar = to export the current image in the local system cache in to a file, we use the this. so that we can copy locally into another machine and can import back. with this we can avoid the route of docker hub repo

5. how to load a docker image into the local system image cache from a docker image file
docker image load -i image.tar
(or)
docker image load < image.tar

6. how to add a tag to the existing docker image.
docker image tag image:tag newimage:newtag
to an existing image it gives another name

ODWA127042021 = wW4[wu