Trouble Shooting: Push Docker Images to Docker Hub

If you are able to run docker login , but encountered the following **unauthorized: authentication required** error while running docker push

1. [root@terry ~]# docker login --username=1972
2. Password:
3. Login Succeeded
4. [root@terry ~]#
5. [root@terry ~]# docker push asiye\_yigit\_tutorial/debian:1.01
6. The push refers to a repository [docker.io/asiye\_yigit\_tutorial/debian]
7. 29303f03b719: Preparing
8. 77a77cd4826d: Preparing
9. fe4c16cbf7a4: Preparing
10. unauthorized: authentication require

**Solution:**

*Creating the repository on Docker hub before running docker push.*

Take a look at

http://stackoverflow.com/questions/36663742/docker-unauthorized-authentication-required-upon-push-with-successful-login

My Docker account

Docker id : pintu12345

Password : Pintu12345&&&&

docker login -u pintu12345 -p Pintu12345&&&&

Git username: p1i2n3t4u5

Password: Pintu\_12345

C:\Users\nirpanig>docker info

Containers: 0

Running: 0

Paused: 0

Stopped: 0

Images: 0

Server Version: 18.09.2

Storage Driver: overlay2

Backing Filesystem: extfs

Supports d\_type: true

Native Overlay Diff: true

Logging Driver: json-file

Cgroup Driver: cgroupfs

Plugins:

Volume: local

Network: bridge host macvlan null overlay

Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog

Swarm: inactive

Runtimes: runc

Default Runtime: runc

Init Binary: docker-init

containerd version: 9754871865f7fe2f4e74d43e2fc7ccd237edcbce

runc version: 09c8266bf2fcf9519a651b04ae54c967b9ab86ec

init version: fec3683

Security Options:

seccomp

Profile: default

Kernel Version: 4.9.125-linuxkit

Operating System: Docker for Windows

OSType: linux

Architecture: x86\_64

CPUs: 2

Total Memory: 1.934GiB

Name: linuxkit-00155d380103

ID: KSQO:N5HY:NTTR:RXRK:X6LC:VRRF:U7HM:CKZD:OLOL:RKVO:A3GK:S7YB

Docker Root Dir: /var/lib/docker

Debug Mode (client): false

Debug Mode (server): true

File Descriptors: 22

Goroutines: 47

System Time: 2019-07-05T07:26:16.5253078Z

EventsListeners: 1

Registry: https://index.docker.io/v1/

Labels:

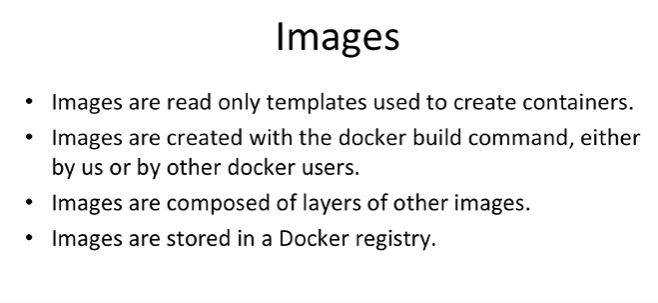
Experimental: false

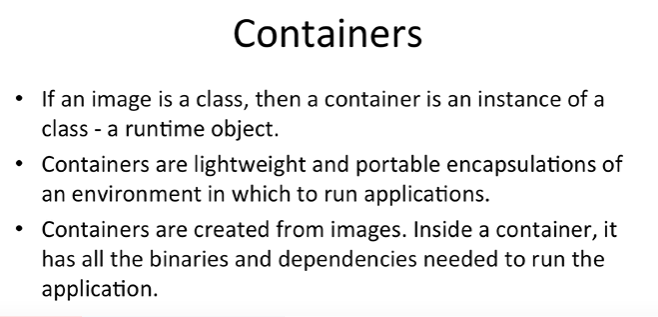
Insecure Registries:

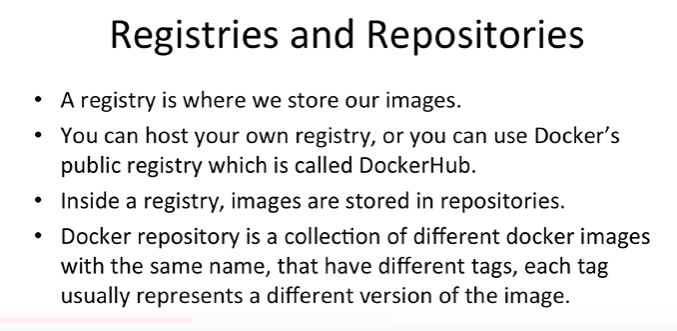
127.0.0.0/8

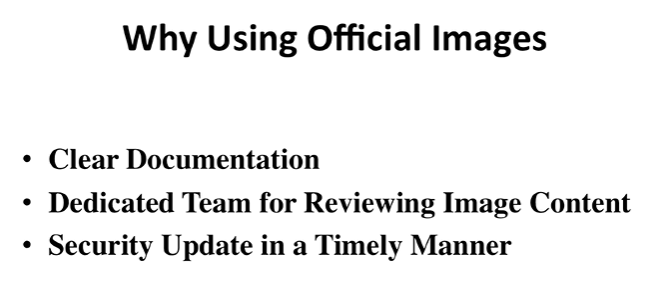
Live Restore Enabled: false

Product License: Community Engine





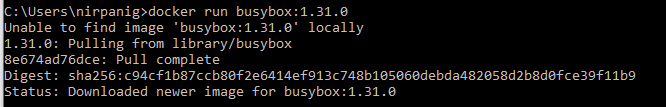




<https://hub.docker.com/>

Docker images command is used to see existing images

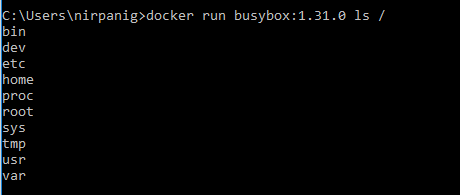


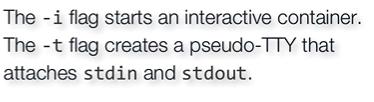


In above command busybox image name 1.31.0 is the tag.

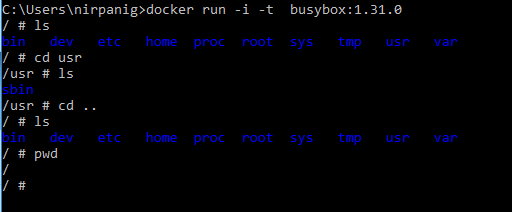
Syntax : docker run imagename:tagname command <argument>

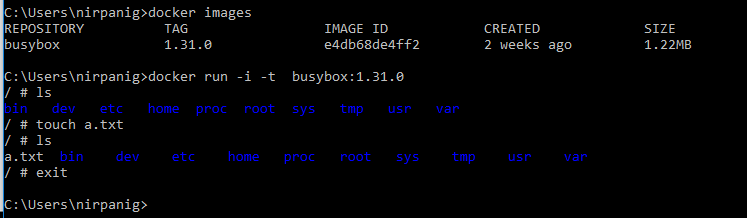






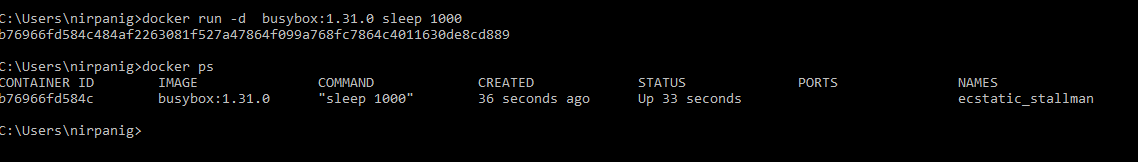
Opens interactive container



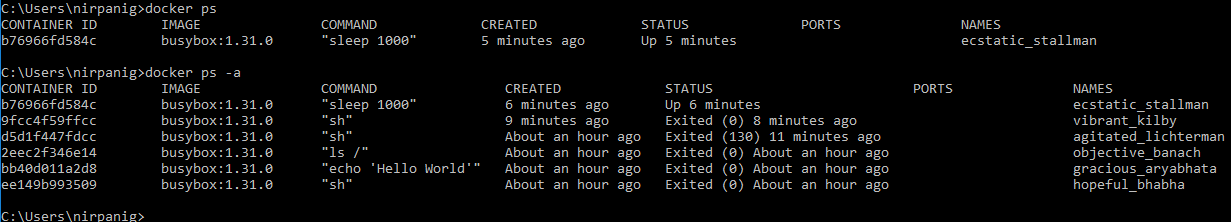


Docker ps command lists all the running docker images

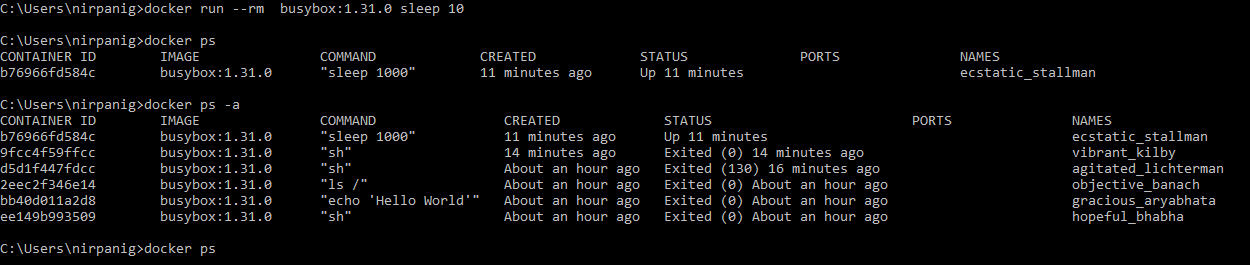
-d option returns long container id



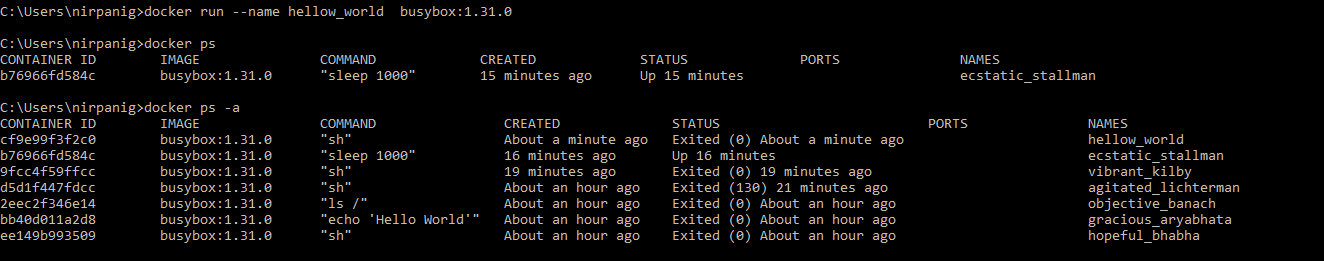
Docker ps –a to list all the previously ran docker images and command

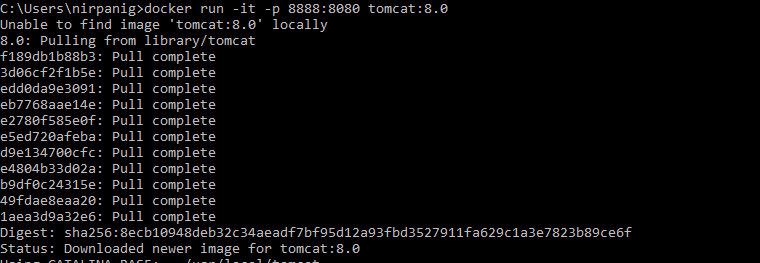


--rm removes the docker container after it exits



--name for changing the container name

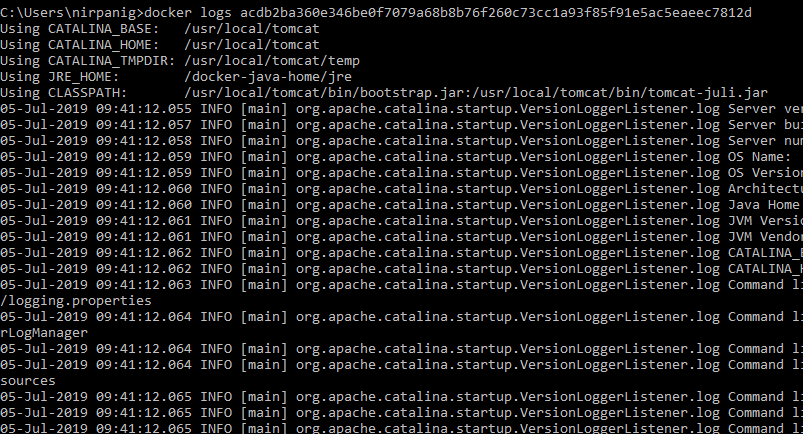


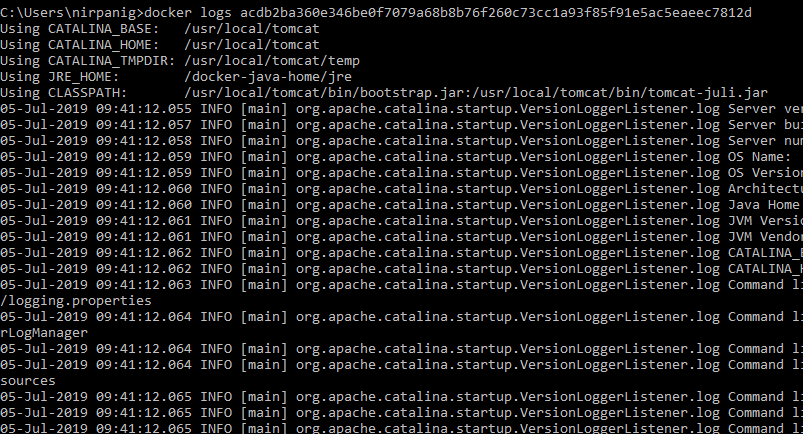




Docker logs container\_id

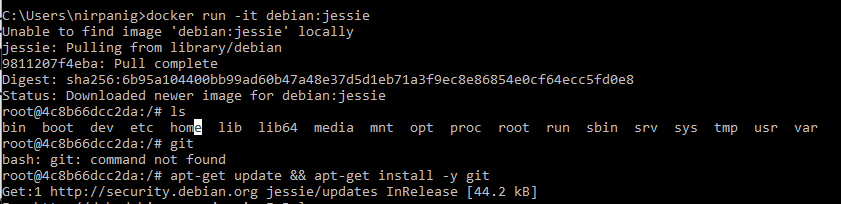






-it is short for --interactive + --tty when you docker run with this command

C:\Users\nirpanig>docker run -it debian:Jessie



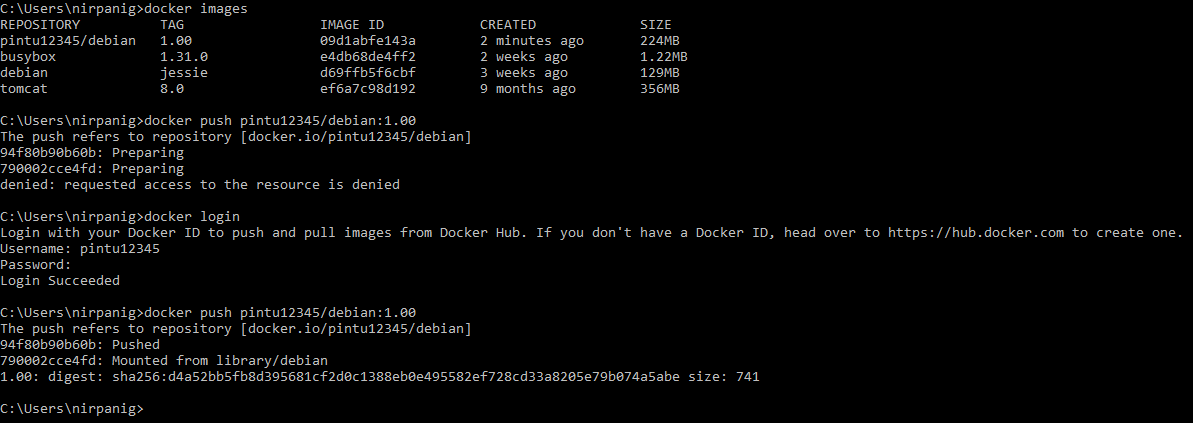
-y option is used to accept automatically prompt for continue

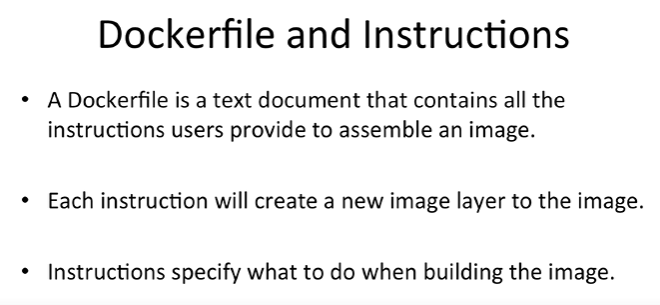
To create an image of our own we have to take an image upon that we have to install other stuff.

We have to create an repository in docker hub to push the new image

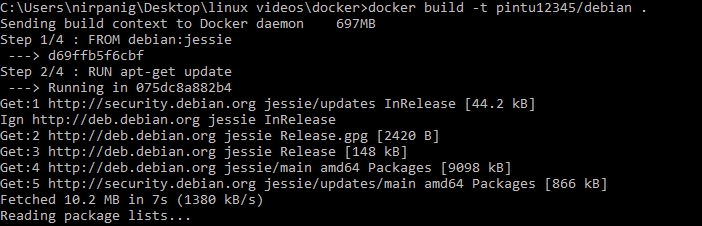


Before pushing the new image to docker hub we mush login from command line

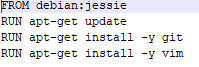




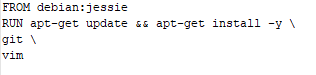
The below command will read the DockerFile from current directory



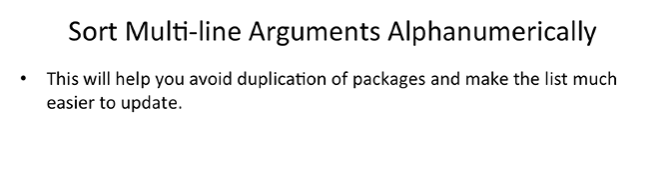
**Multi Line Docker file**

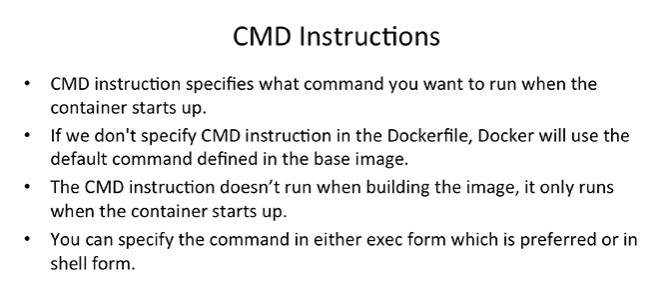


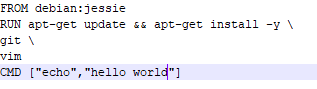
**Single Line Docker file**

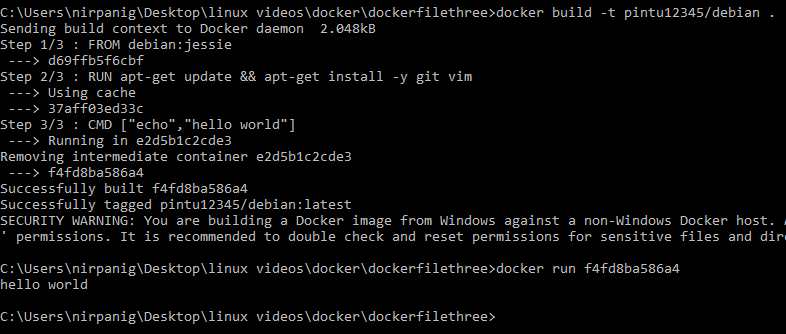


Single line docker file takes lessor number of steps to create an image.

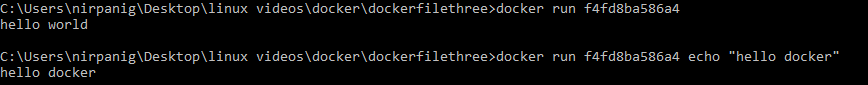


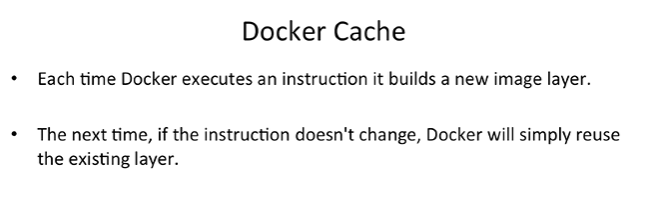


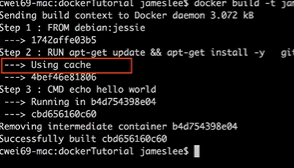


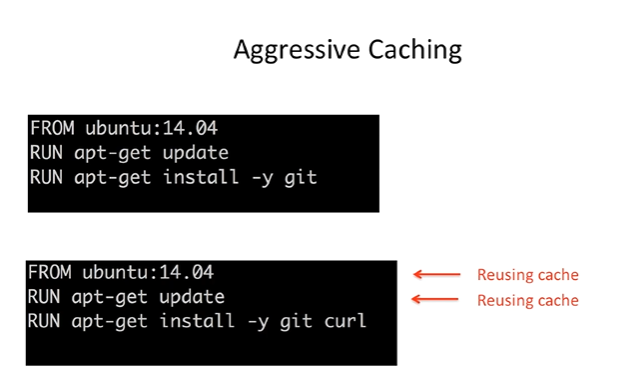


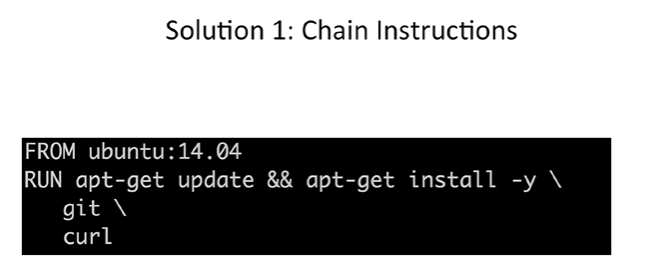
We can override the default command given in DockerFile at run time

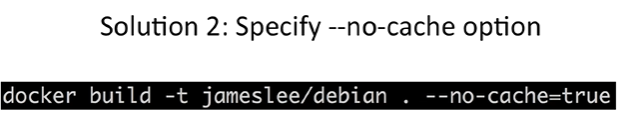


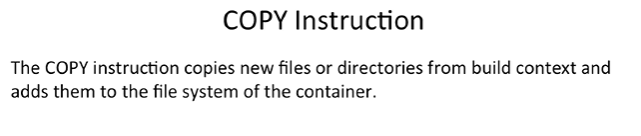


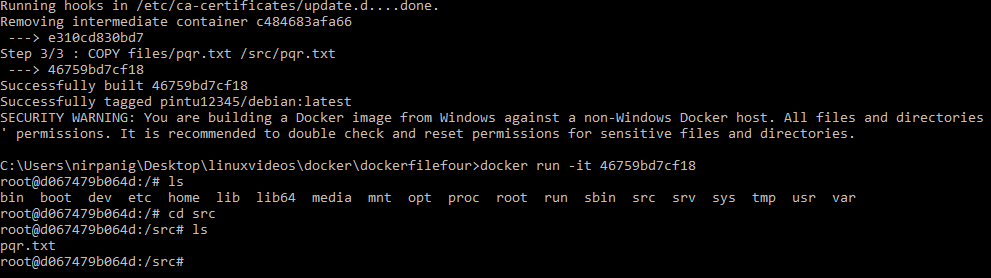


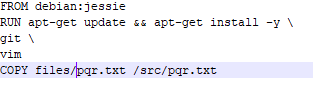




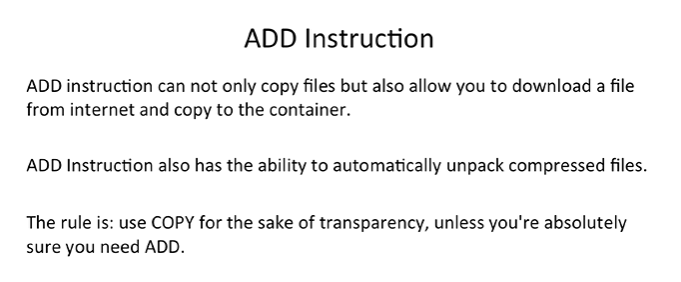


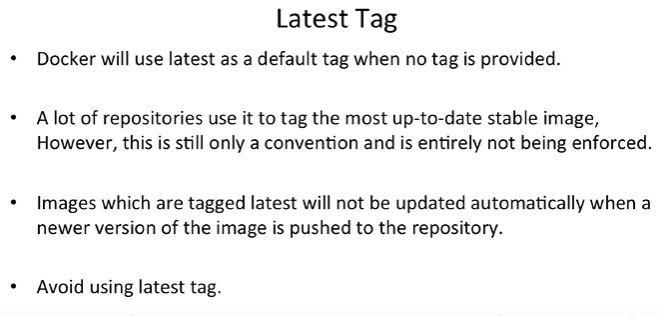




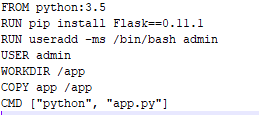


There is some issue with keeping text file directly in the build context.





**First web application**



Line1-> creates image from base image of python

Line2-> installs flask using python pip install

Line3-> creates an user with name admin –ms option creates a home directory for admin user

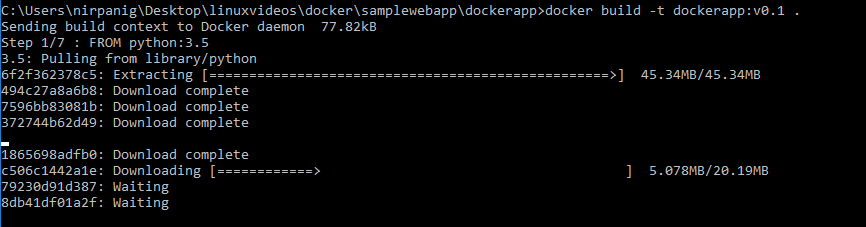
And setting default shell as bash.

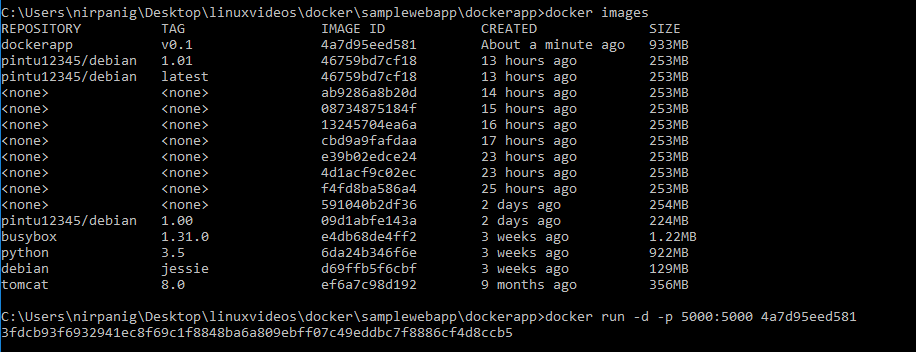
Line4-> this line ensures that the python application and app server will run under admin user.(if not mentioned it will run under root user)

Line5->sets the working directory for RUN, CMD, COPY and entry point application(when we enter the running container it will land in this directory)

Line6-> copy the files

Line7-> run the command as we are already in app folder no need to mention app/app.py

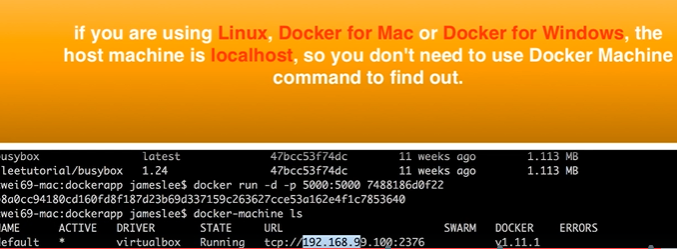




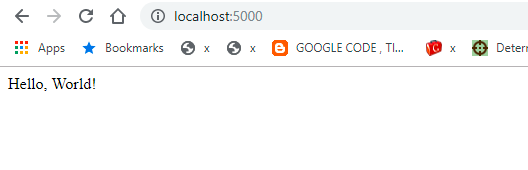
-d option is to run the in the background of container.

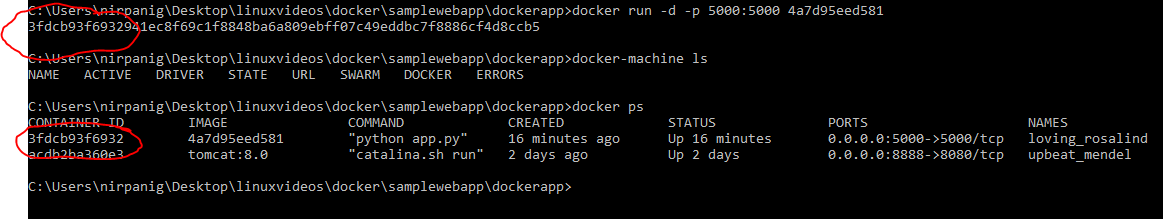
-p option maps container port to host machine port

Smaller alphanumeric value is image id and after starting the image in a container it returns the container id.



Docker-machine ls is used to get the virtual machine ip address where this docker container is running.

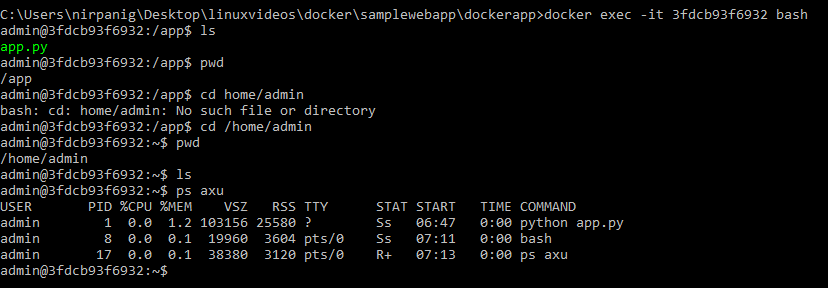




These are long and short container id and short container id.

Using docker exec command we can get in to container running in background and run the bash program

docker exec -it 3fdcb93f6932 bash



Ps axu -> this command will list down all the processes running inside this container.

**Check out source code:**

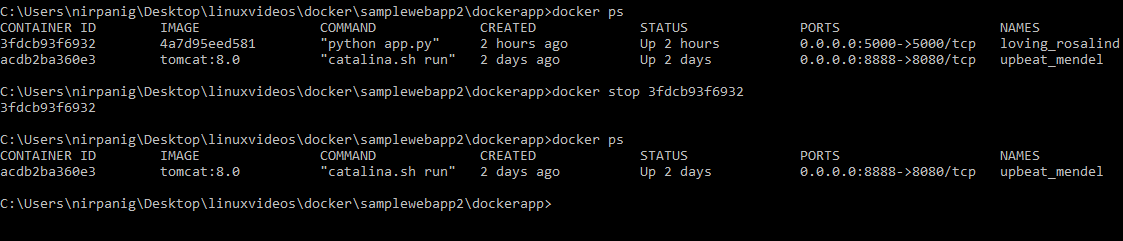
**git clone -b v0.1** <https://github.com/jleetutorial/dockerapp.git>

**Second web application**

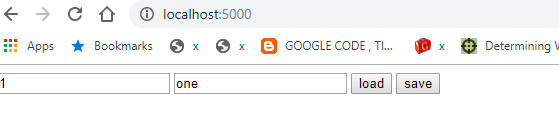
**git clone -b v0.2 https://github.com/jleetutorial/dockerapp.git**



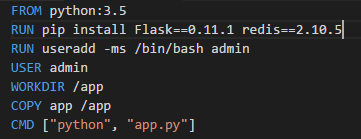
**Stop running container**

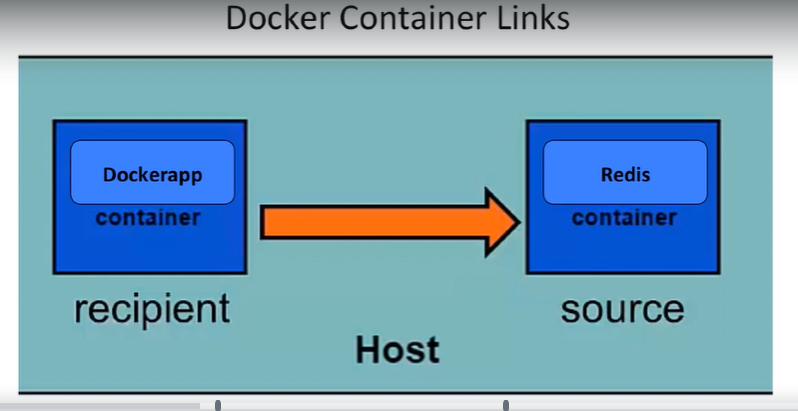






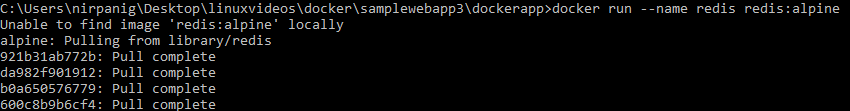






Step 1:

Start the redis container



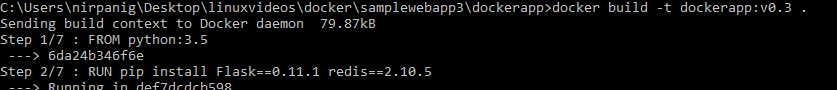
Step 2:

Check whether redis container is running or not



Step 3:

Build the dockerapp



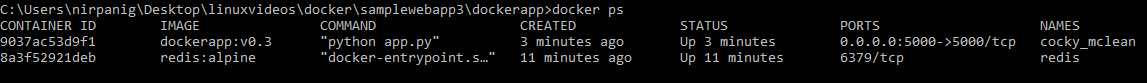
Step 4:

Step run the dockerapp image



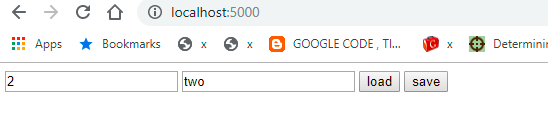
Step 5:

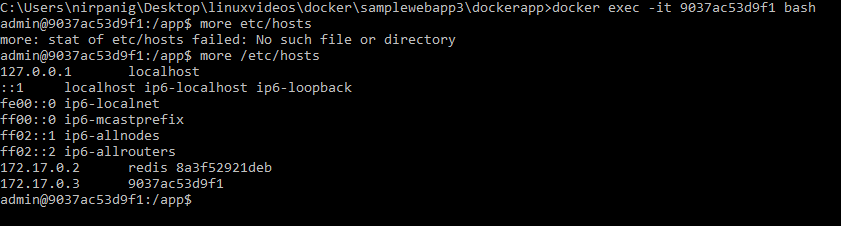
Check both containers are running or not

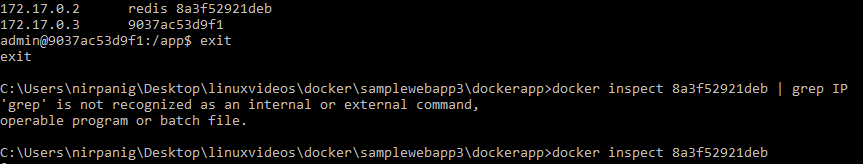


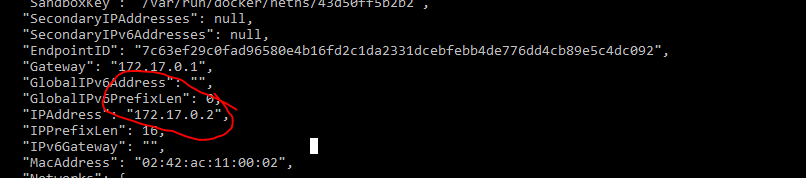
Step 6:

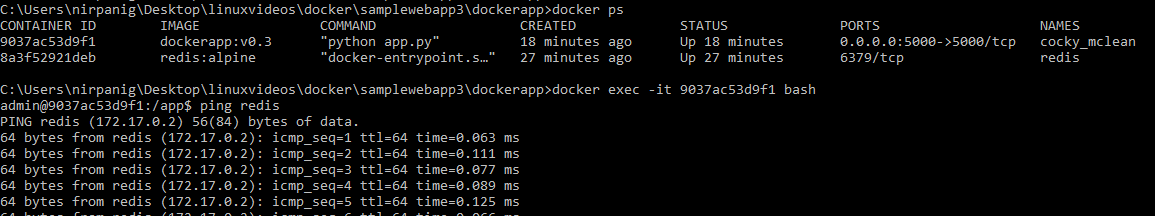
Check app running fine



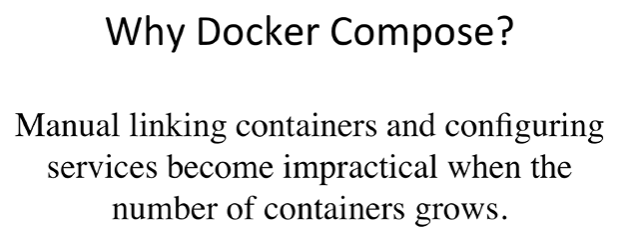




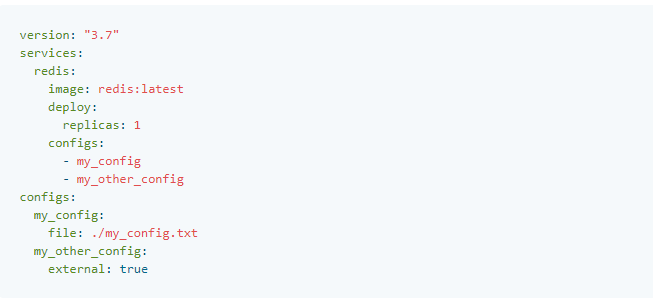


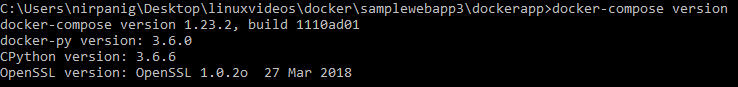


In above example we are connecting from one container to another container with the help container name.



<https://docs.docker.com/compose/compose-file/>

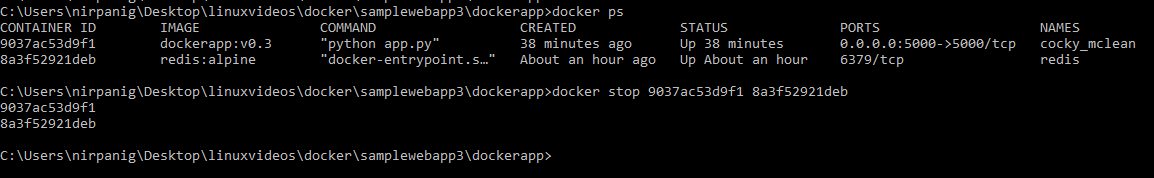




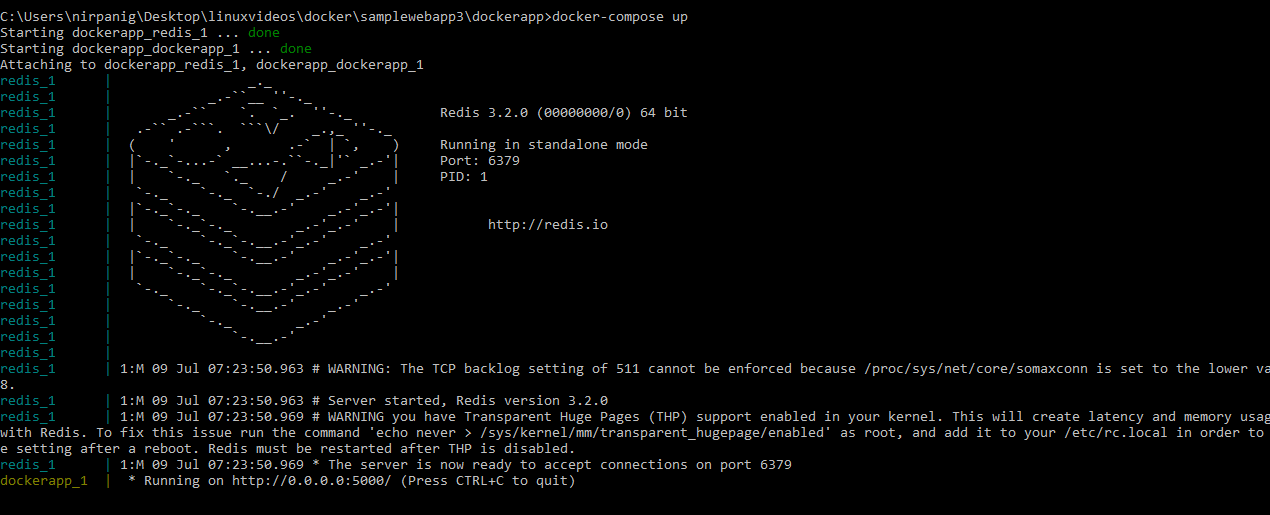


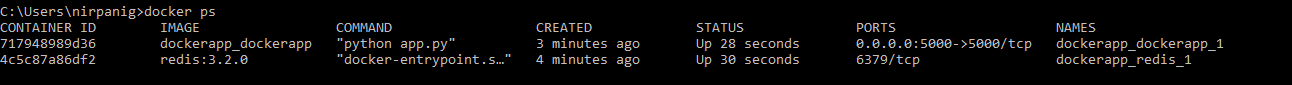
Using docker compose we can create a container from docker file using build step or we can create a container from an image.

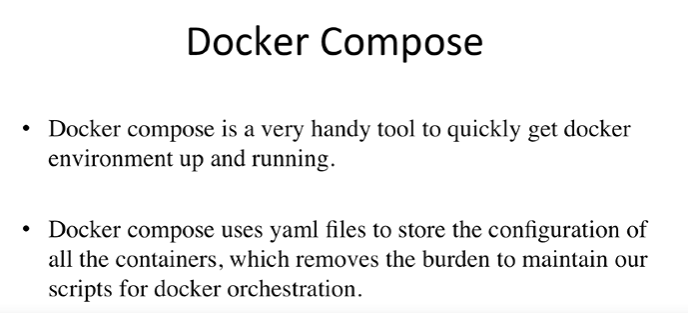
**To stop all running containers**



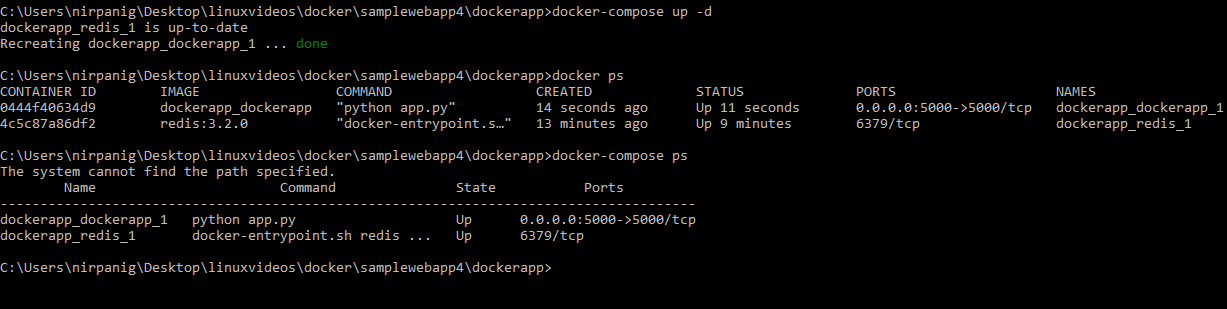
To run docker compose file use command docker-compose up where docker-compose.yml file present





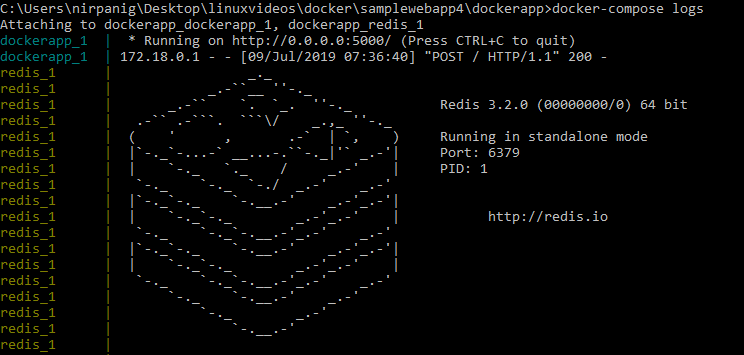


**git clone -b v0.4** <https://github.com/jleetutorial/dockerapp.git>

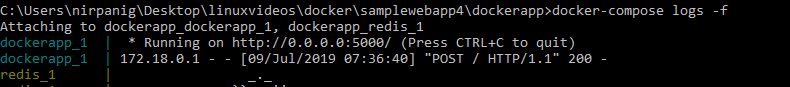


To run the docker compose in background –d option is used

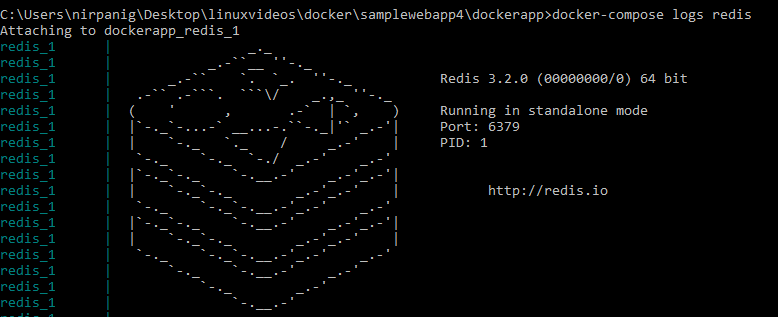
Docker-compose ps -> to see the status of container managed by compose



**docker-compose logs** to see the logs



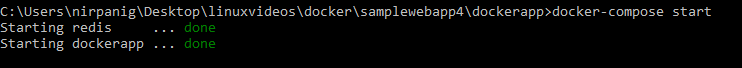
For running logs - f option



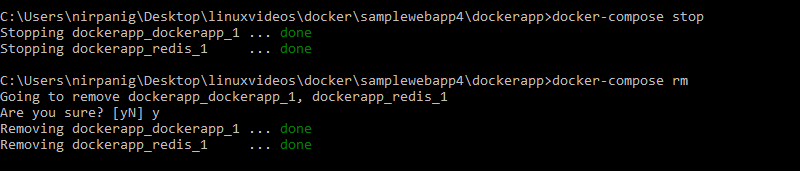
To see the logs of a specific container **docker-compose logs container\_name** is used



**docker-compose stop** to stop all running containers.



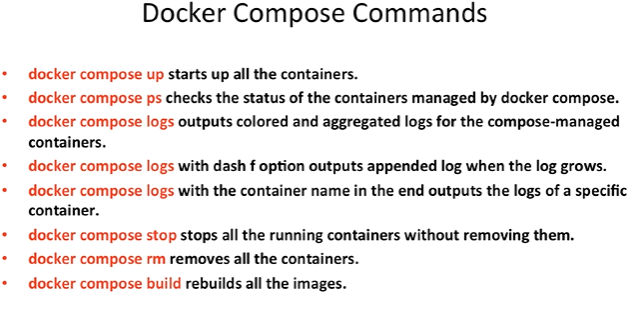
**docker-compose start** to start all stopped containers

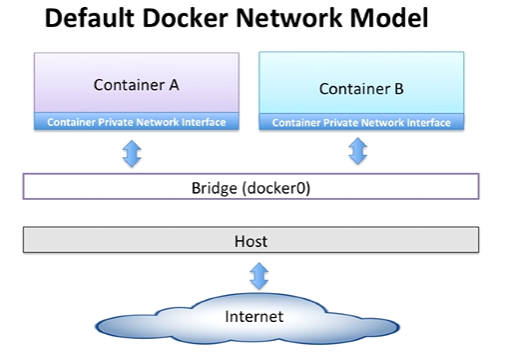


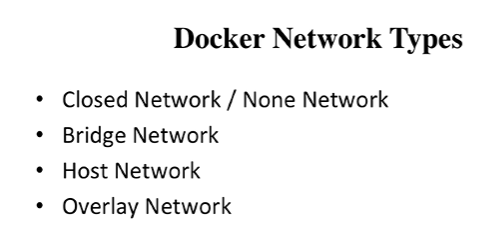
**docker-compose rm** removes stopped containers

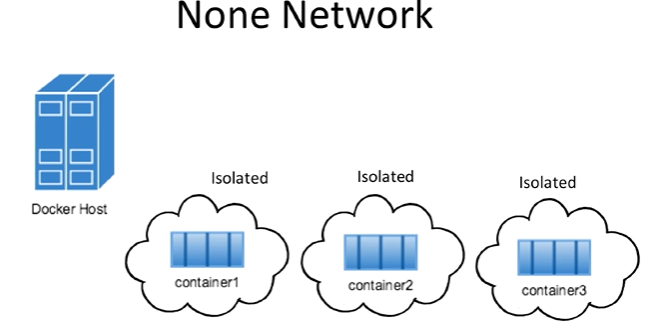
docker-compose up will not rebuild the images

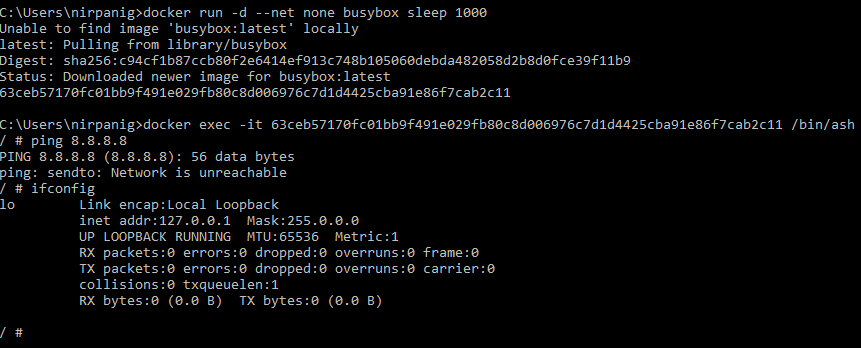
docker-compose build will force for rebuild the images

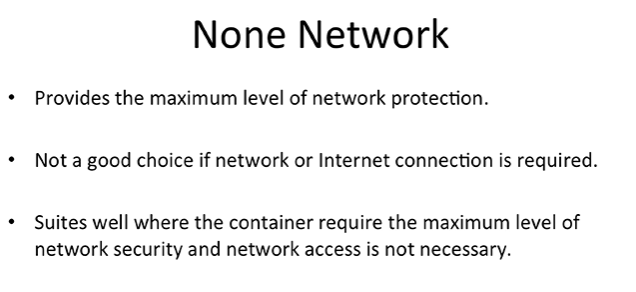


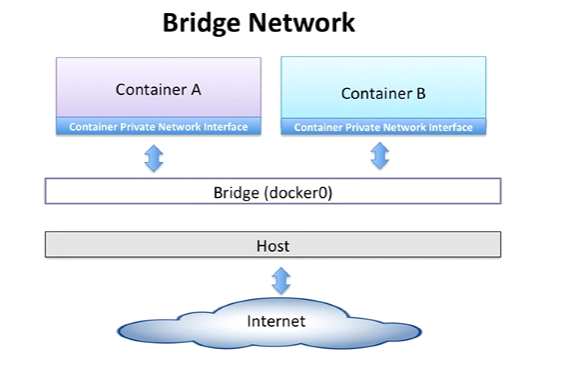


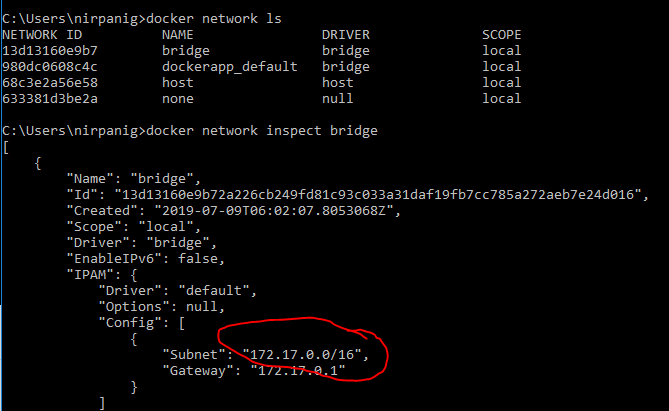




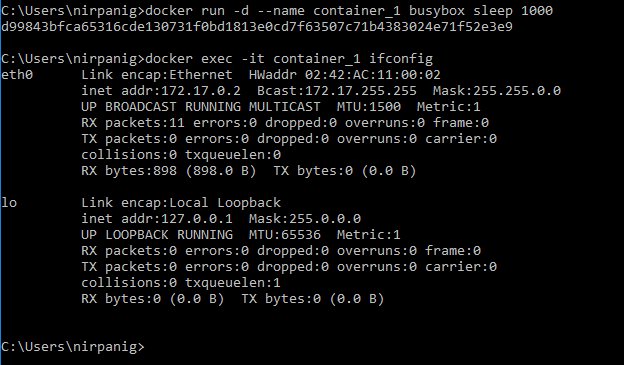




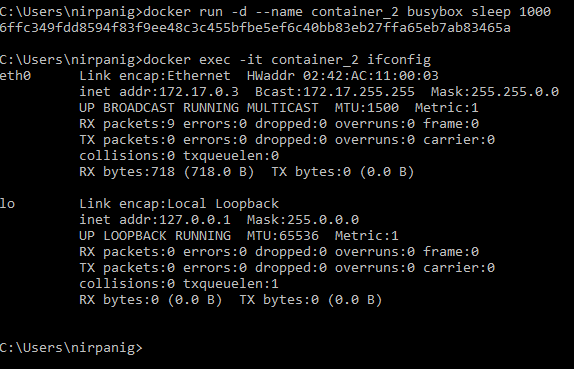




**Container 1**

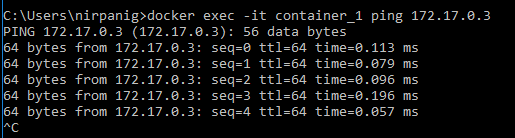


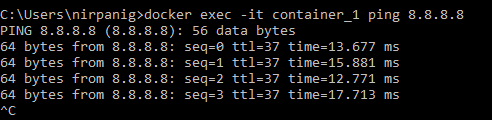
**Container 2**



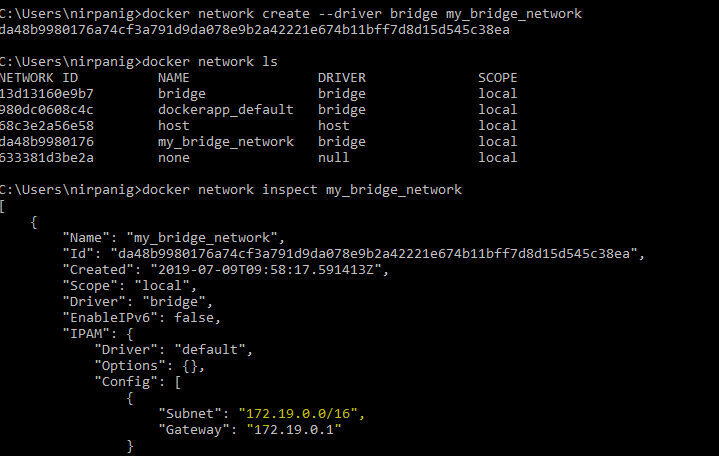
Pinging container\_2 from container\_1 using private network interfaces.

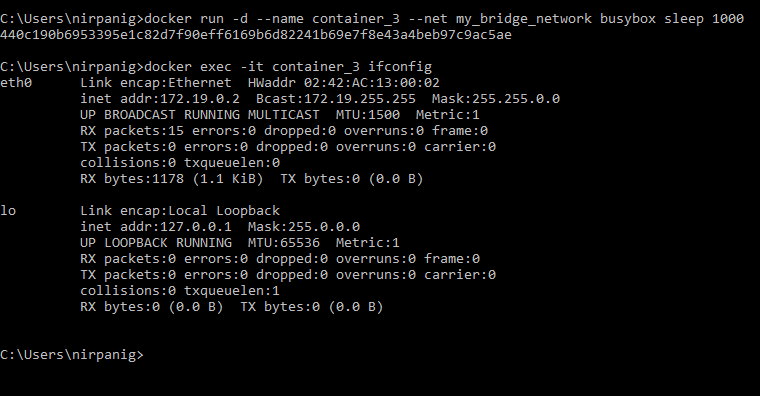
Containers can connect each other using private network interface. They can connect to external world using bridge network interface. 8.8.8.8 is Google’s public IP.



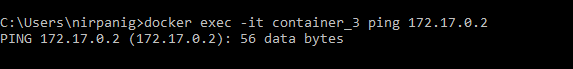


By default different bridge networks are isolated from each other .

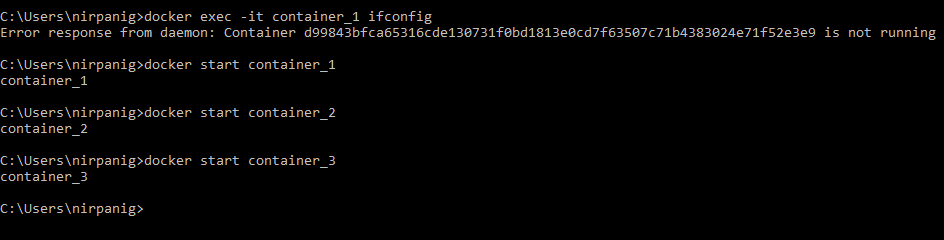




Trying to connect container\_1 from container\_3. it will connect because both are two different bridge network.

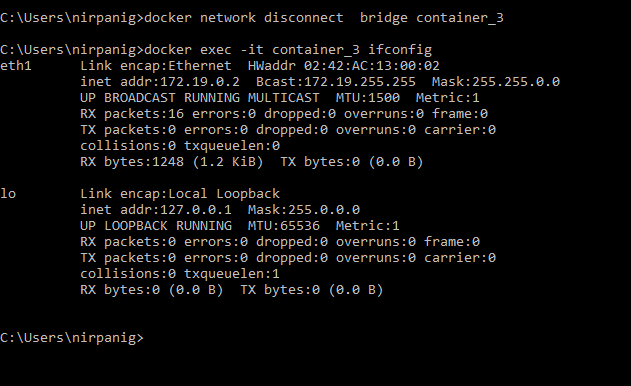


Running again the exited containers

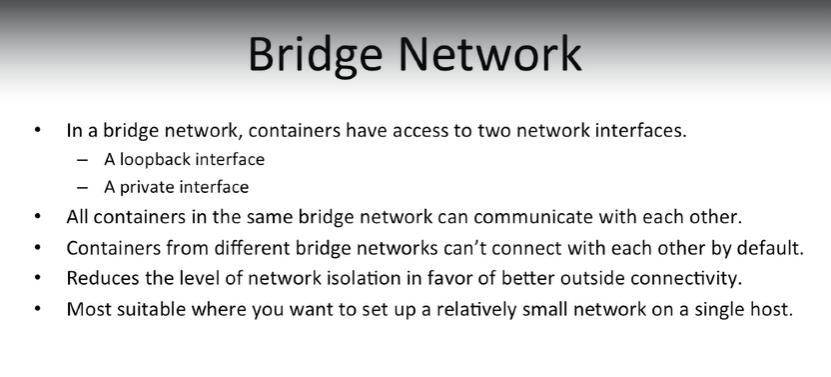


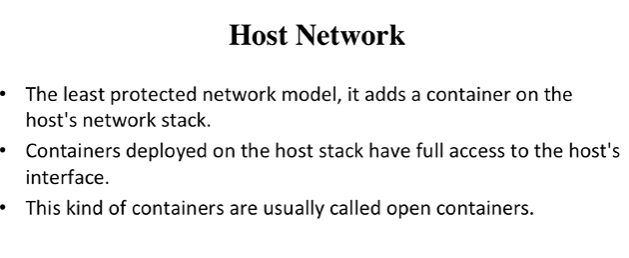


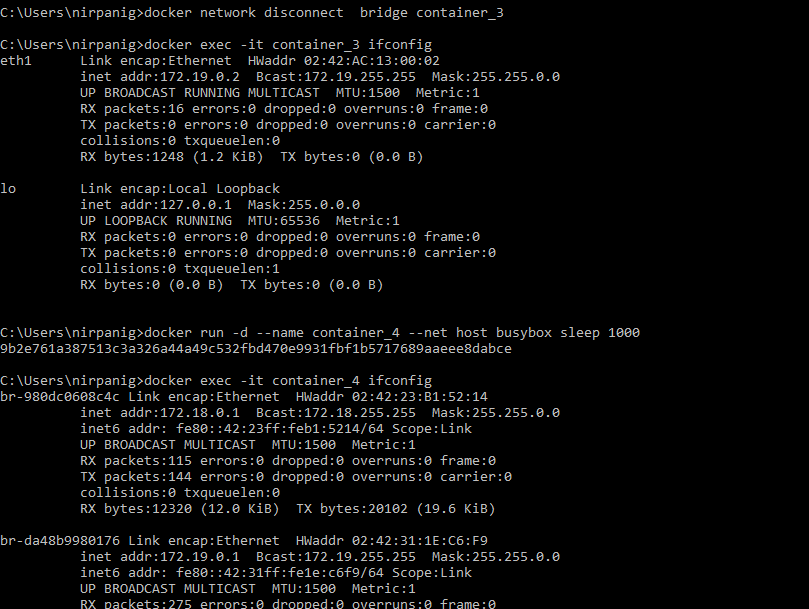
Connecting the default **bridge** network interface to container\_3 ‘s network interface.



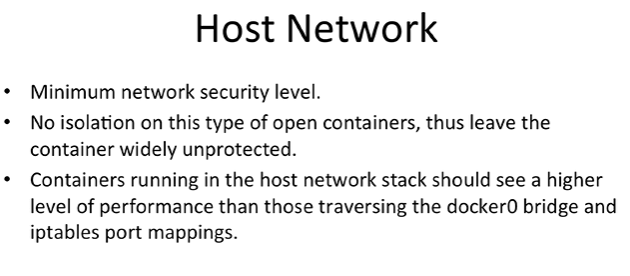
Here we are disconnecting bridge network interface from container\_3 ‘s network interface.

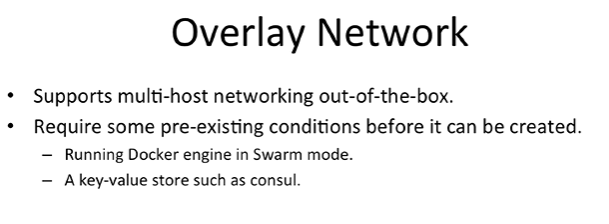


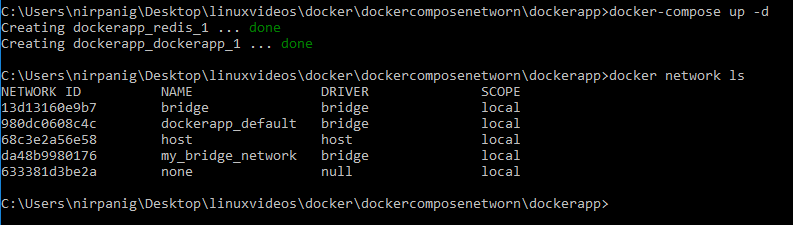




All the available network interface in the host machine are available to the container in this network model.



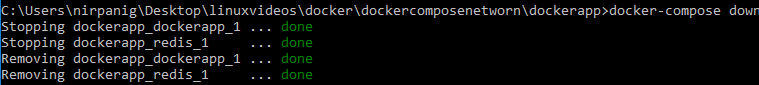




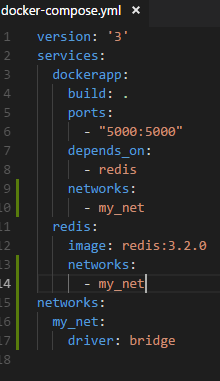
By default docker compose uses default driver to create a bridge network among all the containers.

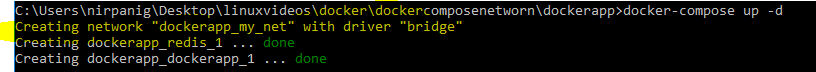
Here dockerapp\_default is the name of the bridge network where dockerapp is the name of the folder where the yml file is present.

Docker-compose down is used to stop the containers and remove all network interface.



Below is the changes to **docker-compose.yml** file to assign user defined network to the containers





**Text Lecture: Overlay Network**

<https://docs.docker.com/engine/userguide/networking/overlay-standalone-swarm/#create-a-swarm-cluster>

Text Direction: Deploy Docker Application to the Cloud with Docker Machine

**Docker Machine Create command**

docker-machine create --driver digitalocean --digitalocean-access-token <xxxxx> docker-app-machine

Extra learning Material: Dockers Monitoring Tools

We didn't cover much about Docker monitoring strategy in this course. However, monitoring tools are handy option for a smooth functioning of your ops. If properly configured, logs and visual data can make a huge difference during troubleshooting.  
We have written a post to summarize some of the popular Docker monitoring tools that you can choose for your environment depending upon your needs.

[Check this up if you are interested.](https://www.level-up.one/dockers-monitoring-tools/)

<https://www.level-up.one/dockers-monitoring-tools/>

**Spotify docker-maven-plugin can’t connect to localhost:2375**

Follow following steps for window 10  
Step 1: Right click on "Dcoker Desktop is running icon "  
Step 2: click on Settings  
Step 3: In “General Tab” you must enable checkbox “Expose Demon on tcp://localhost:2375 without TLS”  
Reference screen attach

**For unix**

export DOCKET\_HOST=unix:///private/var/tmp/docker.sock

COPY and ADD are both Dockerfile instructions that serve similar purposes. They let you copy files from a specific location into a Docker image.

COPY takes in a *src* and *destination*. It only lets you copy in a local file or directory from your host (the machine building the Docker image) into the Docker image itself.

ADD lets you do that too, but it also supports 2 other sources. First, you can use a URL instead of a local file / directory. Secondly, you can extract a tar file from the source directly into the destination.

gcloud auth application-default login

gcloud auth configure-docker

gcloud auth application-default print-access-token

{

"auths": {

"asia.gcr.io": {},

"eu.gcr.io": {},

"gcr.io": {},

"https://index.docker.io/v1/": {},

"marketplace.gcr.io": {},

"staging-k8s.gcr.io": {},

"us.gcr.io": {}

},

"HttpHeaders": {

"User-Agent": "Docker-Client/18.09.2 (windows)"

},

"credsStore": "wincred",

"credHelpers": {

"asia.gcr.io": "gcr",

"eu.gcr.io": "gcr",

"gcr.io": "gcr",

"marketplace.gcr.io": "gcr",

"staging-k8s.gcr.io": "gcr",

"us.gcr.io": "gcr"

},

"stackOrchestrator": "swarm"

**How to pass environment variables to docker container while running**

Make sure your Dockerfile declares an environment variable with ENV:

ENV environment default\_env\_value

ENV cluster default\_cluster\_value

The ENV <key> <value> form can be [replaced inline](https://docs.docker.com/engine/reference/builder/#environment-replacement).

Then you can [pass an environment variable with docker run](https://docs.docker.com/engine/reference/run/#/env-environment-variables)

docker run -p 9000:9000 -e environment=dev -e cluster=0 -d me/app

Or you can [set them through your compose file](https://docs.docker.com/compose/environment-variables/):

node:

environment:

- environment=dev

- cluster=0

Your Dockerfile CMD can use that environment variable, but, as mentioned in [issue 5509](https://github.com/docker/docker/issues/5509#issuecomment-45243772), you need to do so in a sh -c form:

CMD ["sh", "-c", "node server.js ${cluster} ${environment}"]

**What is the difference between EXPOSE and –p**

Basically, you have three options:

1. Neither specify EXPOSE nor -p
2. Only specify EXPOSE
3. Specify EXPOSE and –p

1) If you specify neither EXPOSE nor -p, the service in the container will only be accessible from inside the container itself.

2) If you EXPOSE a port, the service in the container is not accessible from outside Docker, but from inside other Docker containers. So this is good for inter-container communication.

3) If you EXPOSE and -p a port, the service in the container is accessible from anywhere, even outside Docker.

The reason why both are separated is IMHO because:

* choosing a host port depends on the host and hence does not belong to the Dockerfile (otherwise it would be depending on the host),
* and often it's enough if a service in a container is accessible from other containers.

The [documentation](https://docs.docker.com/engine/reference/builder/#expose) explicitly states:

The EXPOSE instruction exposes ports for use within links.

It also points you to how to [link containers](https://docs.docker.com/userguide/dockerlinks/), which basically is the inter-container communication I talked about.

PS: If you do -p, but do not EXPOSE, Docker does an implicit EXPOSE. This is because if a port is open to the public, it is automatically also open to other Docker containers. Hence -p includes EXPOSE. That's why I didn't list it above as a fourth case.

**Docker RUN vs CMD vs ENTRYPOINT**

* RUN executes command(s) in a new layer and creates a new image. E.g., it is often used for installing software packages.
* CMD sets default command and/or parameters, which can be overwritten from command line when docker container runs.
* ENTRYPOINT configures a container that will run as an executable.

### **Shell and Exec forms**

All three instructions (RUN, CMD and ENTRYPOINT) can be specified in shell form or exec form. Let’s get familiar with these forms first, because the forms usually cause more confusion than instructions themselves.

##### **Shell form**

<instruction> <command>

Examples:

RUN apt-get install python3

CMD echo "Hello world"

ENTRYPOINT echo "Hello world"

When instruction is executed in shell form it calls /bin/sh -c <command> under the hood and normal shell processing happens. For example, the following snippet in Dockerfile

ENV name John Dow

ENTRYPOINT echo "Hello, $name"

when container runs as docker run -it <image> will produce output

Hello, John Dow

Note that variable name is replaced with its value.

##### **Exec form**

This is the preferred form for CMD and ENTRYPOINT instructions.

<instruction> ["executable", "param1", "param2", ...]

Examples:

RUN ["apt-get", "install", "python3"]

CMD ["/bin/echo", "Hello world"]

ENTRYPOINT ["/bin/echo", "Hello world"]

When instruction is executed in exec form it calls executable directly, and shell processing does not happen. For example, the following snippet in Dockerfile

ENV name John Dow

ENTRYPOINT ["/bin/echo", "Hello, $name"]

when container runs as docker run -it <image> will produce output

Hello, $name

Note that variable name is not substituted.

### **RUN**

RUN instruction allows you to install your application and packages requited for it. It executes any commands on top of the current image and creates a new layer by committing the results. Often you will find multiple RUN instructions in a Dockerfile.

RUN has two forms:

* RUN <command> (shell form)
* RUN ["executable", "param1", "param2"] (exec form)

(The forms are described in detail in Shell and Exec forms section above.)

A good illustration of RUN instruction would be to install multiple version control systems packages:

RUN apt-get update && apt-get install -y \

bzr \

cvs \

git \

mercurial \

subversion

Note that apt-get update and apt-get install are executed in a single RUN instruction. This is done to make sure that the latest packages will be installed. If apt-get install were in a separate RUN instruction, then it would reuse a layer added by apt-get update, which could had been created a long time ago.

### CMD

CMD instruction allows you to set a default command, which will be executed only when you run container without specifying a command. If Docker container runs with a command, the default command will be ignored. If Dockerfile has more than one CMD instruction, all but last CMD instructions are ignored.

CMD has three forms:

* CMD ["executable","param1","param2"] (exec form, preferred)
* CMD ["param1","param2"] (sets additional default parameters for ENTRYPOINT in exec form)
* CMD command param1 param2 (shell form)

Again, the first and third forms were explained in Shell and Exec forms section. The second one is used together with ENTRYPOINT instruction in exec form. It sets default parameters that will be added after ENTRYPOINT parameters if container runs without command line arguments. See ENTRYPOINT for example.

Let’s have a look how CMD instruction works. The following snippet in Dockerfile

CMD echo "Hello world"

when container runs as docker run -it <image> will produce output

Hello world

but when container runs with a command, e.g., docker run -it <image> /bin/bash, CMD is ignored and bash interpreter runs instead:

root@7de4bed89922:/#

### **ENTRYPOINT**

ENTRYPOINT instruction allows you to configure a container that will run as an executable. It looks similar to CMD, because it also allows you to specify a command with parameters. The difference is ENTRYPOINT command and parameters are not ignored when Docker container runs with command line parameters. (There is a way to ignore ENTTRYPOINT, but it is unlikely that you will do it.)

ENTRYPOINT has two forms:

* ENTRYPOINT ["executable", "param1", "param2"] (exec form, preferred)
* ENTRYPOINT command param1 param2 (shell form)

Be very careful when choosing ENTRYPOINT form, because forms behaviour differs significantly.

##### **Exec form**

Exec form of ENTRYPOINT allows you to set commands and parameters and then use either form of CMD to set additional parameters that are more likely to be changed. ENTRYPOINT arguments are always used, while CMD ones can be overwritten by command line arguments provided when Docker container runs. For example, the following snippet in Dockerfile

ENTRYPOINT ["/bin/echo", "Hello"]

CMD ["world"]

when container runs as docker run -it <image> will produce output

Hello world

but when container runs as docker run -it <image> John will result in

Hello John

##### **Shell form**

Shell form of ENTRYPOINT ignores any CMD or docker run command line arguments.

### **The bottom line**

Use RUN instructions to build your image by adding layers on top of initial image.

Prefer ENTRYPOINT to CMD when building executable Docker image and you need a command always to be executed. Additionally use CMD if you need to provide extra default arguments that could be overwritten from command line when docker container runs.

Choose CMD if you need to provide a default command and/or arguments that can be overwritten from command line when docker container runs.