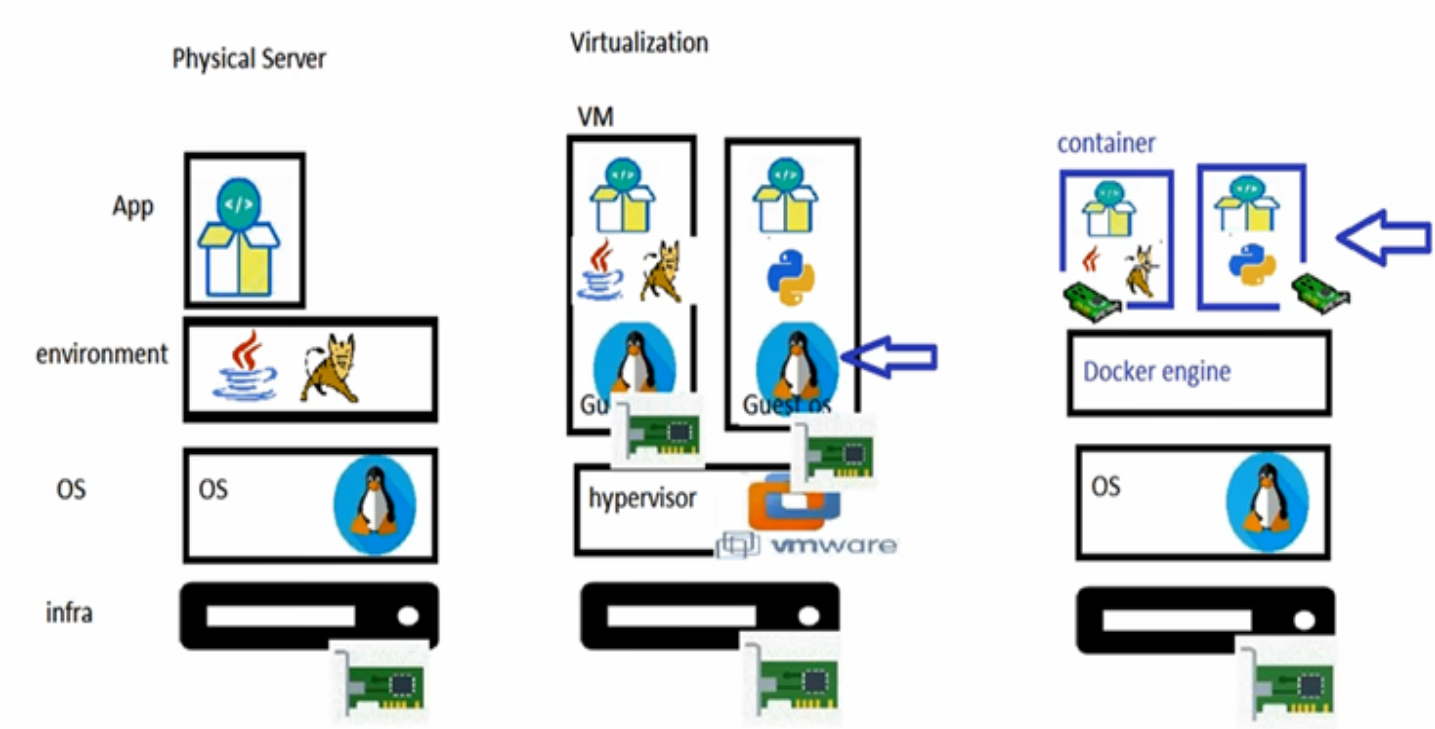
5.1) Docker-DetailedNotes

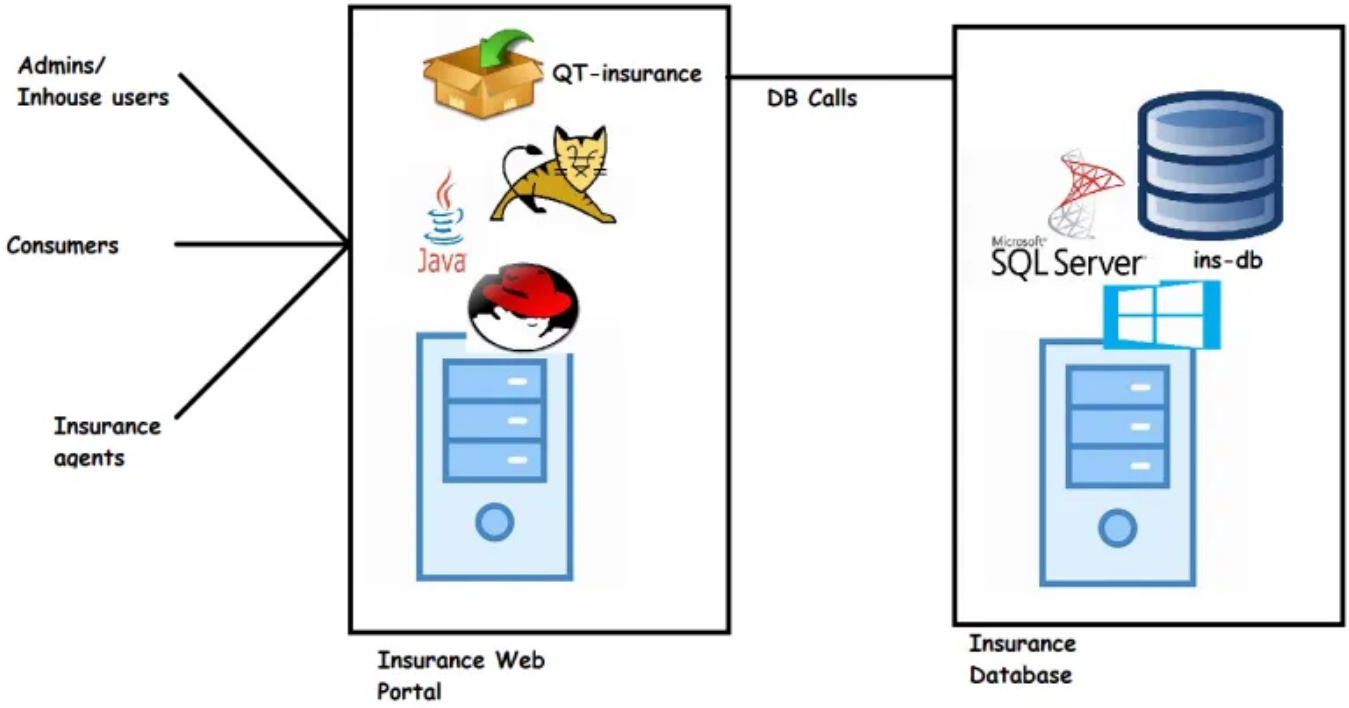


Topic:- Evolution of applications

1. Monolith
2. SOAP
3. Microservices

Monolith

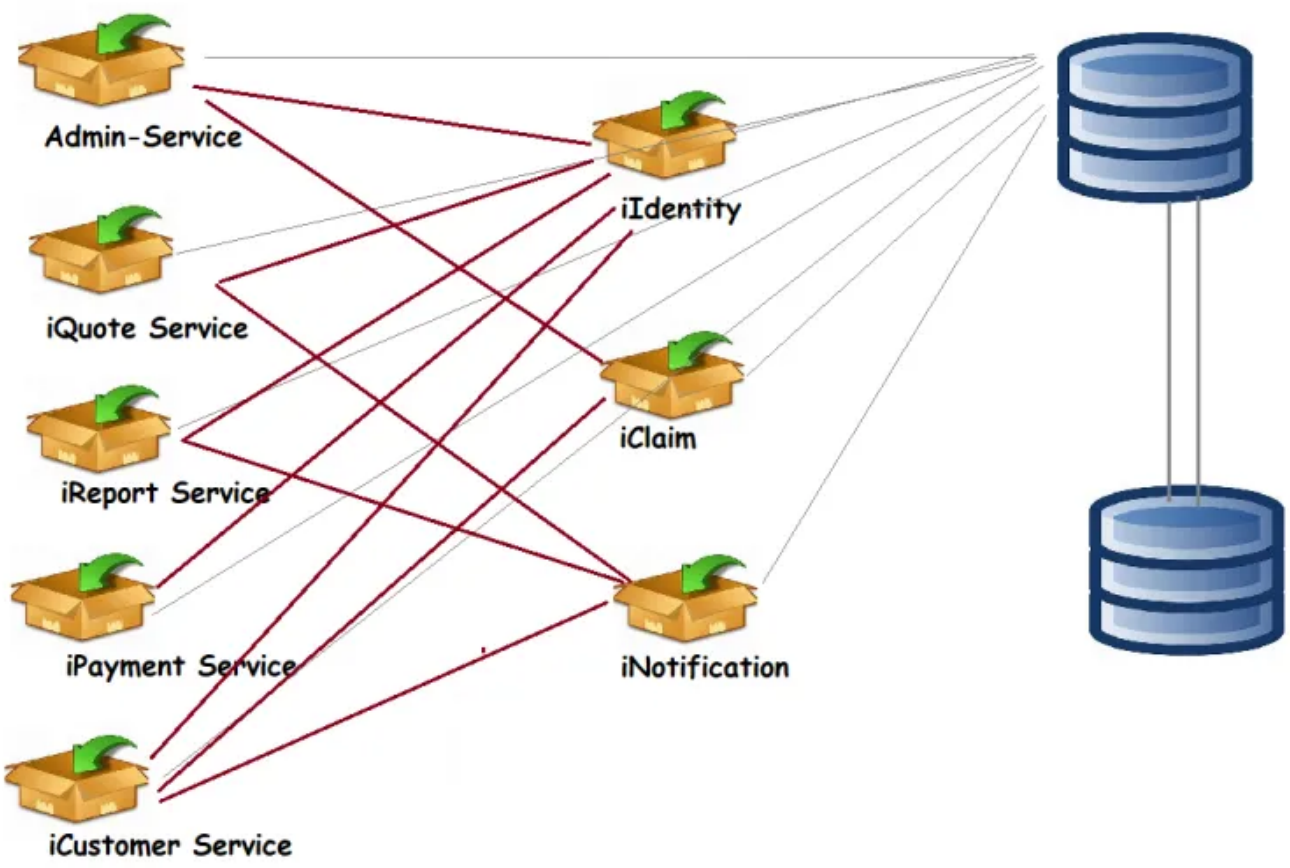
- The complete application is packaged as a unit and deployed on to the server
- The whole insurance application on the server will have one more server to store the data.



- There is an income tax submission season, where there will be a lot of sales
    - Agents will create a lot of quotes
    - Consumer logins might need to be created
- The insurance web portal is running on 3 servers that can handle approx 15000 parallel requests, in this tax season, users will be around 25k. So we need to have at least 5 servers for the web portal.
- Docker containers can be used to run monolith applications

MicroServices

- Application is broken down into multiple individually executable services. With each service offering some functionality



- Microservices can be scaled individually.
- Docker containers are the best fit for microservices.
- Applications will be of two types.
    - Stateless applications:
        - Do Not store state locally, they store the state on external systems.
        - Docker container is the best fit for stateless applications.
    - Stateful application
        - Stores all or some state locally.
        - We can run them on docker, extra processing is required.

TOPIC:- Docker Playground

- Is a virtual environment with virtual machines hosted with Docker pre-installed.
- We can use Docker Playground to experiment with docker.
- To use docker playground, we need docker login Refer Here to create a login

TOPIC: Docker Installation:-

Mainly 2 types i) script based ii) instruction based

I) script based:-

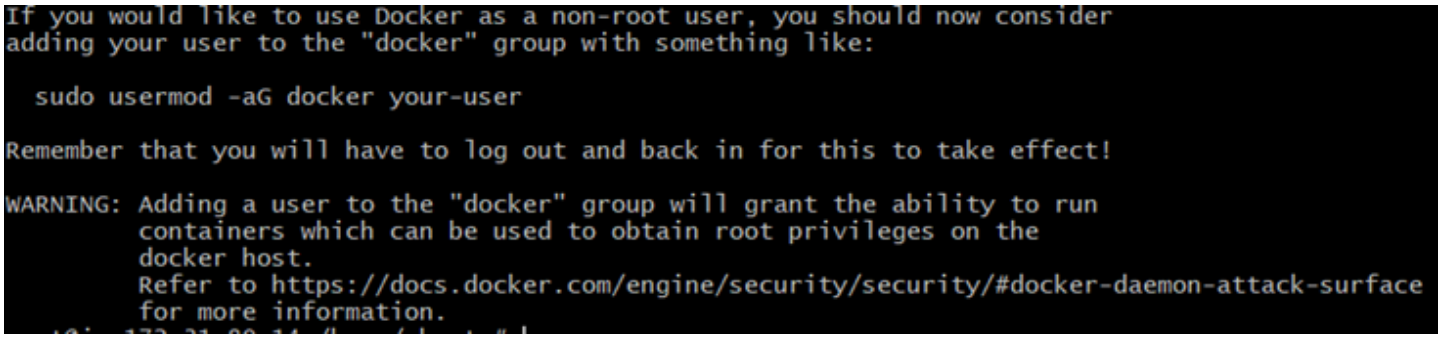
a) Linux/Centos based:-

```
curl -fsSL https://get.docker.com -o get-docker.sh
sh get-docker.sh
sudo service docker start
sudo usermod -aG docker centos
# logout & login
docker --version
docker info
```

b) Ubuntu based:-

```
sudo apt update
curl -fsSL https://get.docker.com -o get-docker.sh
sh get-docker.sh
```

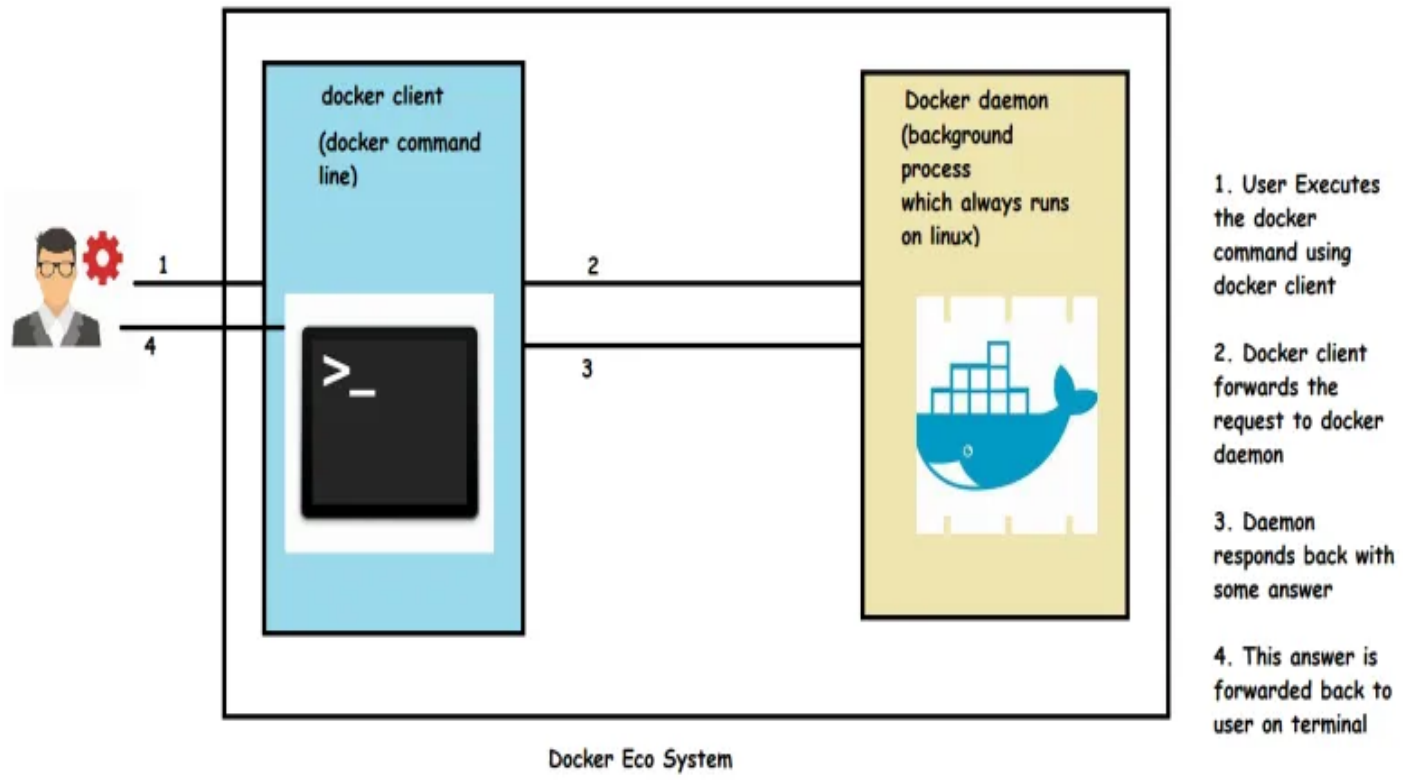
NOTE:- after installing the above script docker user is created and we need to add that used to Ubuntu group



ii) Instruction Based Approach: Refer Here

- Ubuntu (https://docs.docker.com/engine/install/ubuntu/)
- Centos (https://docs.docker.com/engine/install/centos/)

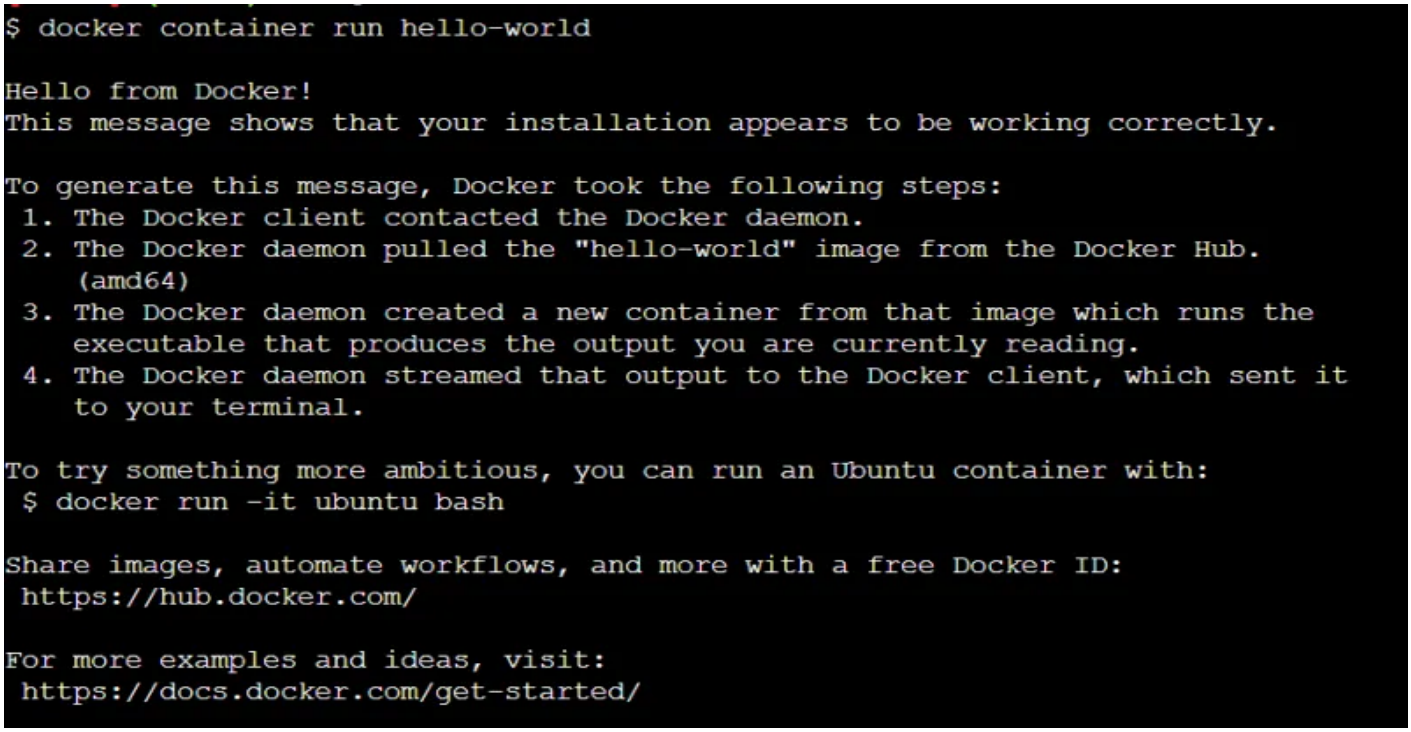
- When we install docker, two major docker components get installed



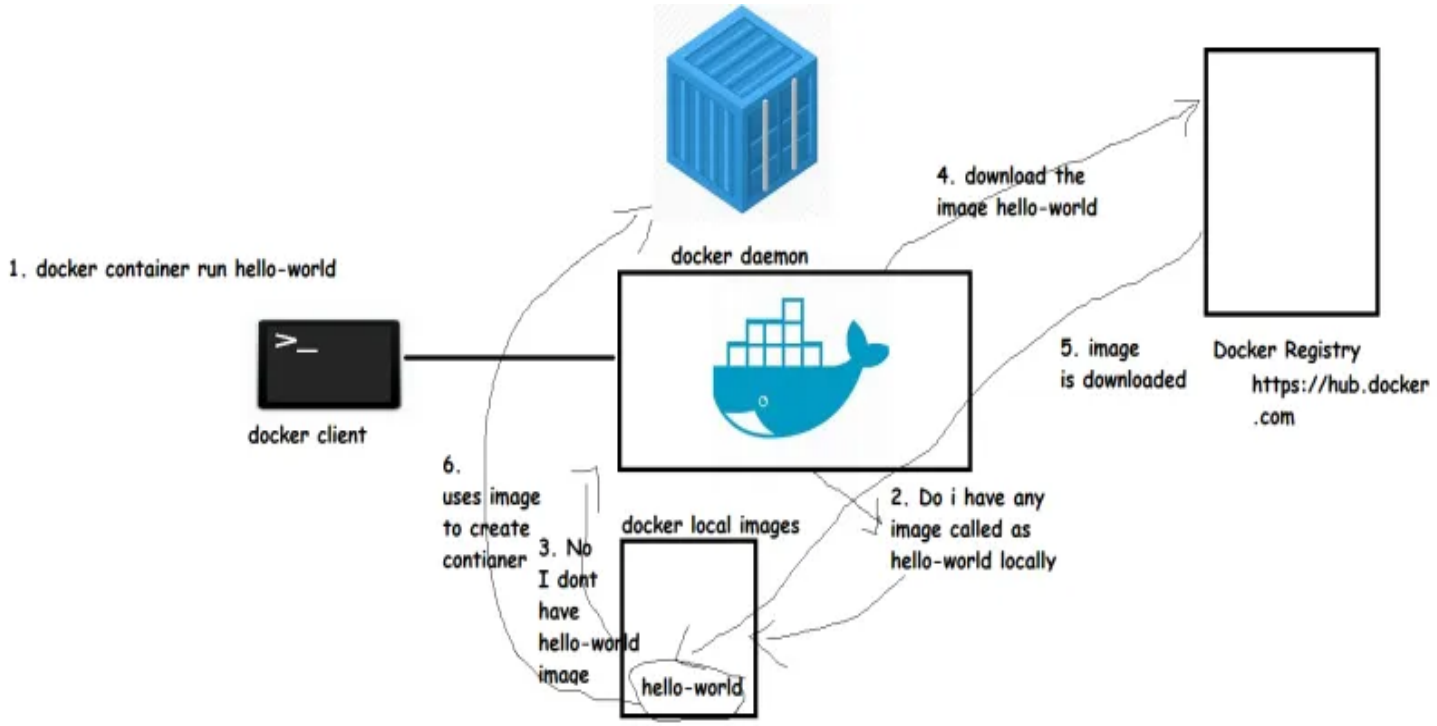
Docker Eco System

Lets execute simple commands in docker and we will check what is happening:-

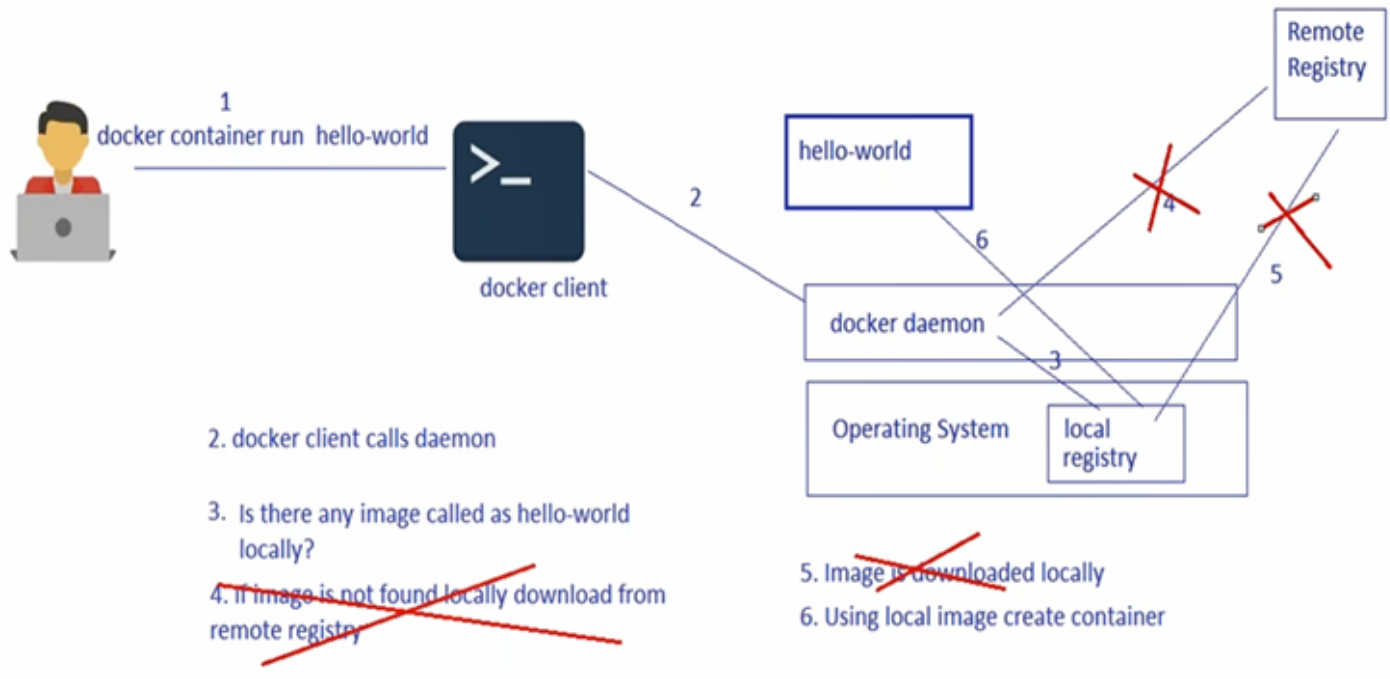
```
$ docker container run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:8c5aeeb6a5f3ba4883347d3747a7249f491766ca1caa47e5da5dfcf6b9b717c0
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```



- When we run docker container hello-world the request is going to demon and it will ask do you have any image called “hello-world” if it is a fresh image and demon doesn’t have any images then it will forward the request to container registry/docker registry, (image is packaging format of docker container)
- If next time it is trying to ask the same image called ‘hello-world’ to the docker daemon then it will give the image.
- For docker containers to run we need docker images. Docker Image is a packaging format which uses all the ingredients to run your application.



When we run the command to create another container with same image the following happens



```
$ docker container run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Note:- Default Docket registry i.e Docket Hub

https://hub.docker.com/

Topic:-Docker commands:-

- Use *help* commands
- Execute docker help
- Docker commands are generally organised as “docker <command> --help”

```
[node1] (local) root@192.168.0.28 ~
$ docker help

Usage:  docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Options:
      --config string      Location of client config files (default "/root/.docker")
  -c, --context string     Name of the context to use to connect to the daemon (overrides DOCKER_HOST env var and default
                           context set with "docker context use")
  -D, --debug              Enable debug mode
  -H, --host list          Daemon socket(s) to connect to
  -l, --log-level string   Set the logging level ("debug"|"info"|"warn"|"error"|"fatal") (default "info")
      --tls                Use TLS; implied by --tlsverify
      --tlscacert string   Trust certs signed only by this CA (default "/root/.docker/ca.pem")
      --tlscert string     Path to TLS certificate file (default "/root/.docker/cert.pem")
      --tlskey string      Path to TLS key file (default "/root/.docker/key.pem")
      --tlsverify          Use TLS and verify the remote
  -v, --version            Print version information and quit

Management Commands:
  app*        Docker App (Docker Inc., v0.9.1-beta3)
  builder     Manage builds
  checkpoint  Manage checkpoints
  config      Manage Docker configs
  container   Manage containers
  context     Manage contexts
  engine      Manage the docker engine
  image       Manage images
  manifest    Manage Docker image manifests and manifest lists
  network     Manage networks
  node        Manage Swarm nodes
  plugin      Manage plugins
  secret      Manage Docker secrets
  service     Manage services
  stack       Manage Docker stacks
  swarm       Manage Swarm
  system      Manage Docker
  trust       Manage trust on Docker images
```

docker management commands:-

```
Management Commands:
  app*        Docker App (Docker Inc., v0.9.1-beta3)
  builder     Manage builds
  checkpoint  Manage checkpoints
  config      Manage Docker configs
  container   Manage containers
  context     Manage contexts
  engine      Manage the docker engine
  image       Manage images
  manifest    Manage Docker image manifests and manifest lists
  network     Manage networks
  node        Manage Swarm nodes
  plugin      Manage plugins
  secret      Manage Docker secrets
  service     Manage services
  stack       Manage Docker stacks
  swarm       Manage Swarm
  system      Manage Docker
  trust       Manage trust on Docker images
  volume      Manage volumes
```

docker commands:-

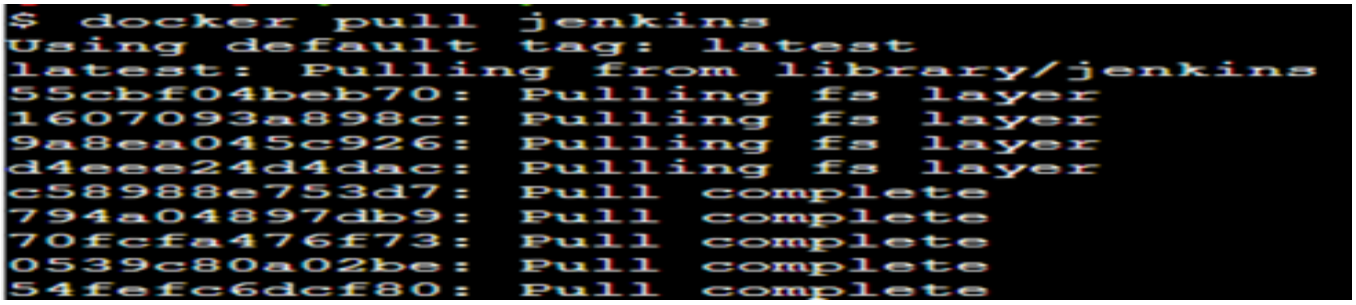
```
Commands:
  attach      Attach local standard input, output, and error streams to a running container
  build       Build an image from a Dockerfile
  commit      Create a new image from a container's changes
  cp          Copy files/folders between a container and the local filesystem
  create      Create a new container
  deploy      Deploy a new stack or update an existing stack
  diff        Inspect changes to files or directories on a container's filesystem
  events      Get real time events from the server
  exec        Run a command in a running container
  export      Export a container's filesystem as a tar archive
  history     Show the history of an image
  images      List images
  import      Import the contents from a tarball to create a filesystem image
  info        Display system-wide information
  inspect     Return low-level information on Docker objects
  kill        Kill one or more running containers
  load        Load an image from a tar archive or STDIN
  login       Log in to a Docker registry
  logout      Log out from a Docker registry
  logs        Fetch the logs of a container
  pause       Pause all processes within one or more containers
  port        List port mappings or a specific mapping for the container
  ps          List containers
  pull        Pull an image or a repository from a registry
  push        Push an image or a repository to a registry
  rename      Rename a container
  restart     Restart one or more containers
  rm          Remove one or more containers
  rmi         Remove one or more images
  run         Run a command in a new container
  save        Save one or more images to a tar archive (streamed to STDOUT by default)
  search      Search the Docker Hub for images
  start       Start one or more stopped containers
  stats       Display a live stream of container(s) resource usage statistics
  stop        Stop one or more running containers
  tag         Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
  top         Display the running processes of a container
  unpause     Unpause all processes within one or more containers
```

Examples

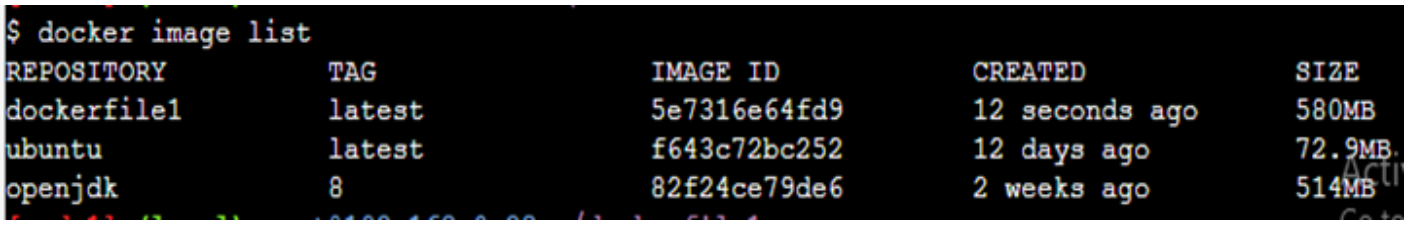
https://docs.docker.com/engine/reference/commandline/cli/

1) docker image build –t <containername> . {This will build the container from the image here . means location of Docker file}

2) docker pull <image-name> {It is used to pull image from docker hub}



3) docker image ls {It will show list of images present in local registry}



Note:- every image we get name,tag,ID,size

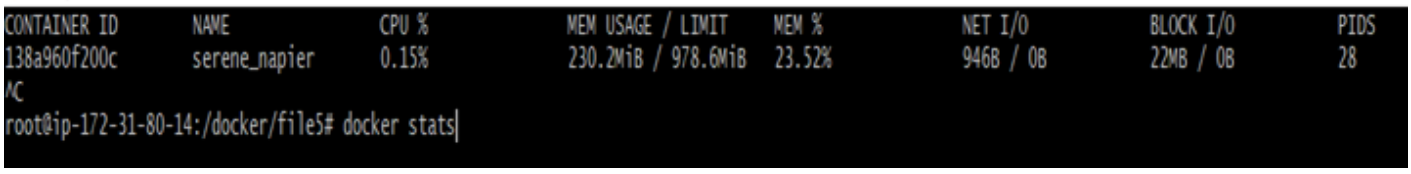
4) docker container run –P –d <image-name>{To run the docker image hear 'P' caps it is responsible for the port forex, if we give EXPOSE 8080 then the 'P' will map to any available port on the host OS/VM . If we want to give custom port we need to use '-p' small p "-p 32790:8080" observe 15 example}

5) docker container ls {it will display running containers}

6) docker image inspect <image name:label> {It will give complete info about specific image}

7) docker image run –it <image-name: version> {It will download image and it will enter into that container}

8) docker stats {It will display running container process memory details }



9) docker container run –help{display all commands }

10) docker container –help

11) docker container stop/start/pause/resume <container ID/container name>

12) docker container –rm <name/id>{to terminate the container }

13) docker container –rm –f <name/id> {-f is forcefully removed }

14) docker container rm -f $(docker container ls -a -q) {to remove all running container at a time}

15) docker container run –p <host-port>:<container -port> <image>{-p is used to define port number }

16) docker container exec <container name> <command > {It will execute commands in side the given container without going inside to that container }

```
root@ip-172-31-80-14:/docker# docker container exec 914cd4425e1e ls
bin
boot
dev
docker-java-home
etc
home
lib
lib64
media
mnt
opt
```

17) docker container exec –it <container name/id> /bin/bash

```
jenkins@914cd4425e1e:/$ ls
bin  boot  dev  docker-java-home  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
jenkins@914cd4425e1e:/$
```

18) docker image inspect -f {{.config.Labels}} spc:slim {to display the LABEL which we mentioned. -f is format type. Also we can use any specific things to display in output}

```
[node1] (local) root@192.168.0.28 ~/testimage
$ docker image inspect -f {{.Config.Labels}} spc:slim
map[author:khaja]
```

19)

DOCKER container modes

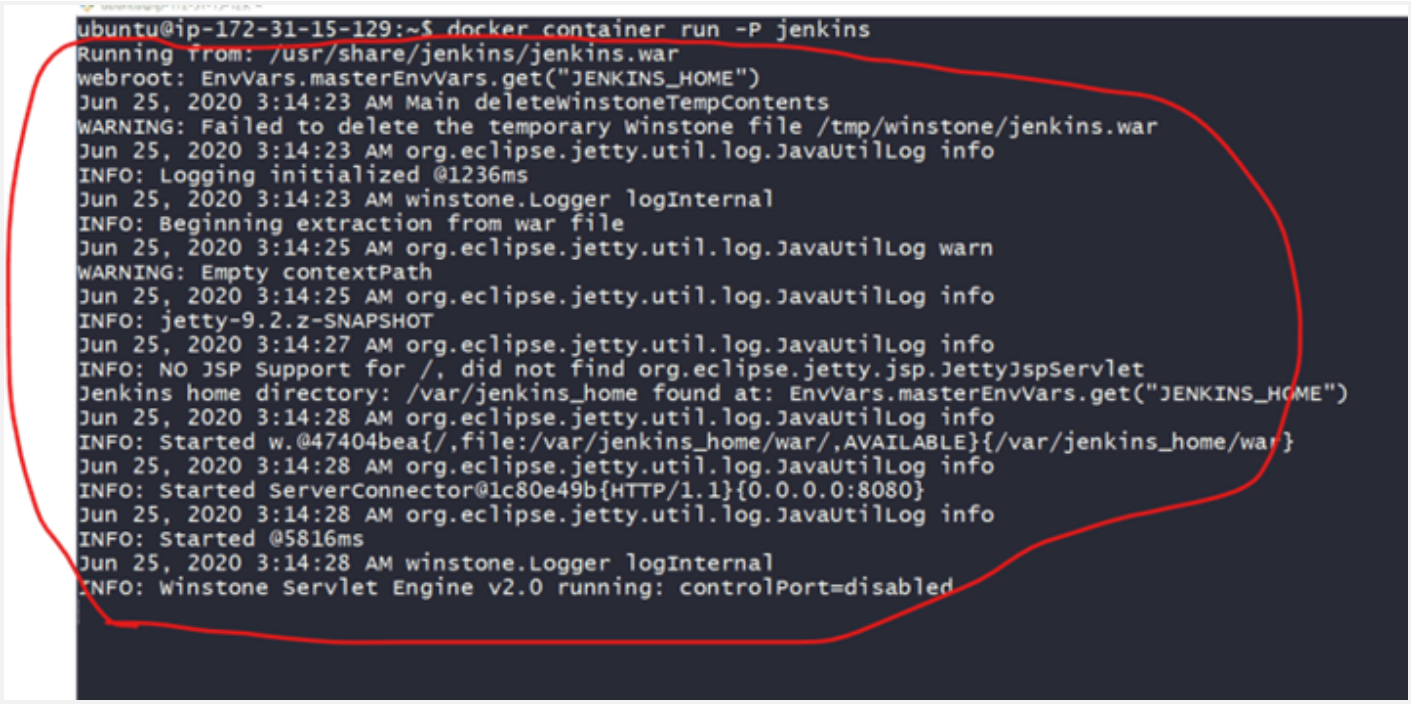
i) Attached mode

ii) detached mode

iii) interactive mode

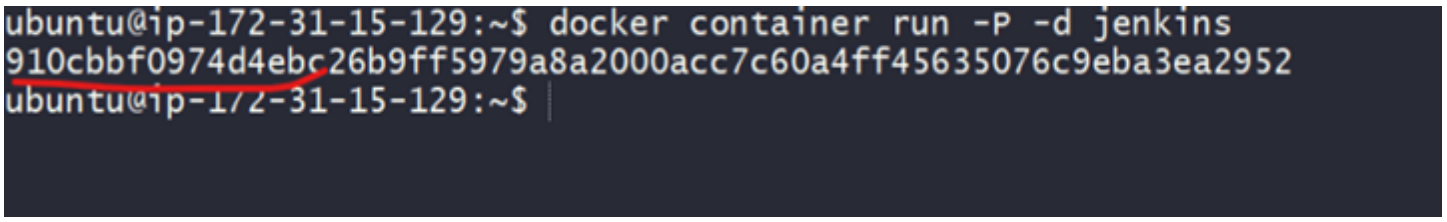
Attached: Container runs in the foreground. The default mode of docker

“Example: docker container run -P Jenkins”



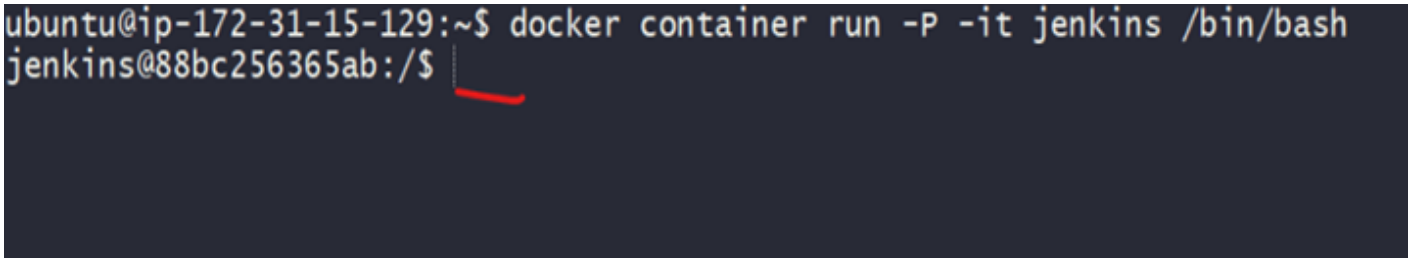
Detached:-

- Docker container runs in the background
- add **-d** to container run to set this mode
- Example: `docker container run -P -d jenkins`



Interactive:-

- In this mode we can interact with docker using terminal (/bin/bash, /bin/sh)
- add **-it** to container run to set this mode
- Example `docker container run -it -P jenkins /bin/bash`



TOPIC: Docker Image:-

Containerization: Process of making your application run on docker containers

Docker image: Docker image is a packing format to run an application on docker engine as a container

What does Docker image have?

It includes everything to run an application. Like code or binaries, runtime, dependencies, other file system objects

Note:- From one Docker image we can create multiple containers

Basic Docker

- When we install docker, we get two components

1. Docker daemon
2. Docker client

- When we run the command to create container the following happens

How can a Docker image be created?

Converting existing container into docker image, Export VMs to Docker Image, Create DockerImage using Dockerfile (Recommended approach)

Docker Registry:-

Docker Hub is Default Registry Refer Here

There are many private docker registries

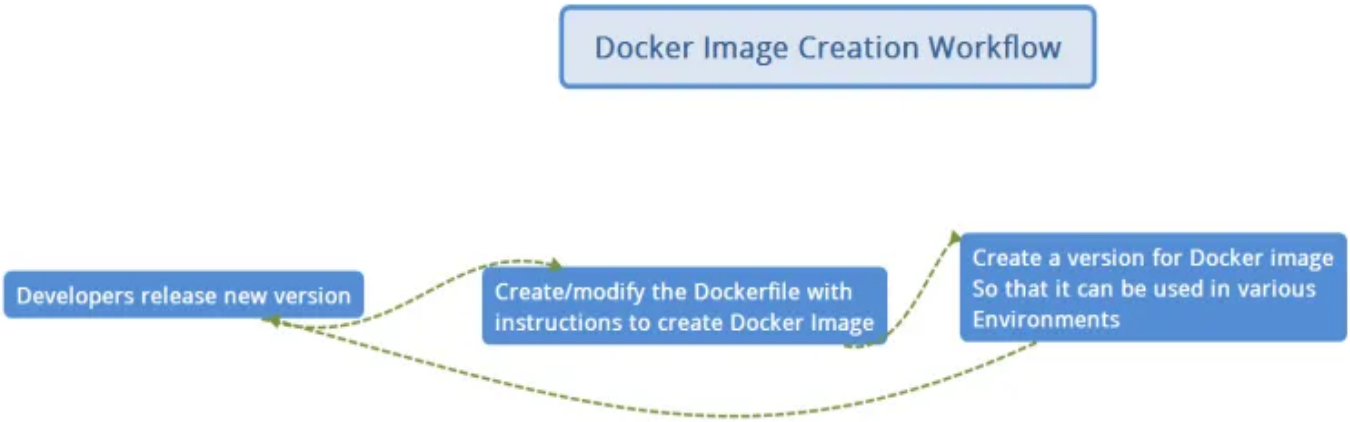
- AWS Elastic Container Registry (ECR)

- Azure Container Registry (ACR)
- Artifactory
- Docker registry (Docker Image) Refer Here
- and many more

Building Docker Images

- Building Docker Images can be done in two ways
- Create a container, do the manual installation & create a docker image from the container.
- This approach is not sensible as the image creation process is manual & cannot be version controlled
- Changes are difficult to handle
- Create a docker container by writing a *Dockerfile*
- This approach helps in maintaining versions of every change which you do
- Changes will be simple as it all about changing one file & create a new version
- *Dockerfile* is a simple plain text with a set of *INSTRUCTIONS*

Workflow:



Basic Steps involved in building a docker image

- choose the right base image
- add steps to install/configure your application
- add information of ports required etc

- add step which should be executed to run your application
- This is exactly what we do to in terms of Dockerfile. In Dockerfile we have lot of instructions. Each instruction performs some activity. So lets look at Dockerfile instructions
- Docker image has the following naming conventions <image-name>:<tag>
- Refer Here for the official documentation.{https://docs.docker.com/engine/reference/builder/}
- Lets try to create a simple dockerfile
- Refer Here for changes {https://github.com/asquarezone/DockerZone/commit/c7dea7d6e776f8e6ec3db4de84d7c011508cb8a7}
- Let's try to build a image

```
[centos@ip-172-31-26-148 hello-docker]$ ls
Dockerfile
[centos@ip-172-31-26-148 hello-docker]$ docker image --help

Usage:  docker image COMMAND

Manage images

Commands:
  build       Build an image from a Dockerfile
  history     Show the history of an image
  import      Import the contents from a tarball to create a filesystem image
  inspect     Display detailed information on one or more images
  load        Load an image from a tar archive or STDIN
  ls          List images
  prune       Remove unused images
  pull        Pull an image or a repository from a registry
  push        Push an image or a repository to a registry
  rm          Remove one or more images
  save        Save one or more images to a tar archive (streamed to STDOUT by default)
  tag         Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE

Run 'docker image COMMAND --help' for more information on a command.
[centos@ip-172-31-26-148 hello-docker]$
```

```
[centos@ip-172-31-26-148 hello-docker]$ docker image build --help

Usage:  docker image build [OPTIONS] PATH | URL | -

Build an image from a Dockerfile

Options:
      --add-host list           Add a custom host-to-IP mapping (host:ip)
      --build-arg list          Set build-time variables
      --cache-from strings      Images to consider as cache sources
      --cgroup-parent string    Optional parent cgroup for the container
      --compress                Compress the build context using gzip
      --cpu-period int          Limit the CPU CFS (Completely Fair Scheduler) period
      --cpu-quota int           Limit the CPU CFS (Completely Fair Scheduler) quota
  -c, --cpu-shares int          CPU shares (relative weight)
      --cpuset-cpus string      CPUs in which to allow execution (0-3, 0,1)
      --cpuset-mems string      MEMs in which to allow execution (0-3, 0,1)
      --disable-content-trust   Skip image verification (default true)
  -f, --file string             Name of the Dockerfile (Default is 'PATH/Dockerfile')
      --force-rm                Always remove intermediate containers
      --iidfile string          Write the image ID to the file
      --isolation string        Container isolation technology
      --label list              Set metadata for an image
  -m, --memory bytes            Memory limit
      --memory-swap bytes       Swap limit equal to memory plus swap: '-1' to enable unlimited swap
```

Now lets try to build our docker image with name my-helloworld

```
[centos@ip-172-31-26-148 hello-docker]$ docker image build -t my-helloworld .
Sending build context to Docker daemon  2.048kB
Step 1/1 : FROM hello-world:latest
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:8c5aeeb6a5f3ba4883347d3747a7249f491766ca1caa47e5da5dfcf6b9b717c0
Status: Downloaded newer image for hello-world:latest
 ---> bf756fb1ae65
Successfully built bf756fb1ae65
Successfully tagged my-helloworld:latest
[centos@ip-172-31-26-148 hello-docker]$
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
nginx               alpine              4efb29ff172a        9 days ago          21.8MB
hello-world         latest              bf756fb1ae65        10 months ago       13.3kB
my-helloworld       latest              bf756fb1ae65        10 months ago       13.3kB
[centos@ip-172-31-26-148 hello-docker]$
```

- When we build docker image an image id, image name & image tag are associated with a image
- Now lets use an instruction LABEL to add metadata Refer Here for the changeset {https://github.com/asquarezone/DockerZone/commit/b3b3d0f36d1f7f12f97eedfb4dd507be0eb7893c}

```
[centos@ip-172-31-26-148 hello-docker]$ docker image build -t my-helloworld .
^[[4~Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM hello-world:latest
 ---> bf756fb1ae65
Step 2/3 : LABEL author="khajaibrahim"
 ---> Running in ac42b83c3ffd
Removing intermediate container ac42b83c3ffd
 ---> 9932ce15331b
Step 3/3 : LABEL organization="learning thoughts"
 ---> Running in 3fe99f6b027b
Removing intermediate container 3fe99f6b027b
 ---> ba68f57b7991
Successfully built ba68f57b7991
Successfully tagged my-helloworld:latest
[centos@ip-172-31-26-148 hello-docker]$
```

```
[centos@ip-172-31-26-148 hello-docker]$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
my-helloworld       latest              ba68f57b7991        28 seconds ago      13.3kB
nginx               alpine              4efb29ff172a        9 days ago          21.8MB
hello-world         latest              bf756fb1ae65        10 months ago       13.3kB
[centos@ip-172-31-26-148 hello-docker]$
```

```
[centos@ip-172-31-26-148 hello-docker]$ docker image --help

Usage:  docker image COMMAND

Manage images

Commands:
  build       Build an image from a Dockerfile
  history     Show the history of an image
  import      Import the contents from a tarball to create a filesystem image
  inspect     Display detailed information on one or more images
  load        Load an image from a tar archive or STDIN
  ls          List images
  prune       Remove unused images
  pull        Pull an image or a repository from a registry
  push        Push an image or a repository to a registry
  rm          Remove one or more images
  save        Save one or more images to a tar archive (streamed to STDOUT by default)
  tag         Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE

Run 'docker image COMMAND --help' for more information on a command.
[centos@ip-172-31-26-148 hello-docker]$ docker image inspect --help
```

```
[centos@ip-172-31-26-148 hello-docker]$ docker image inspect --help

Usage:  docker image inspect [OPTIONS] IMAGE [IMAGE...]

Display detailed information on one or more images

Options:
  -f, --format string   Format the output using the given Go template
[centos@ip-172-31-26-148 hello-docker]$
```

Now lets try find history of image

```
[centos@ip-172-31-26-148 hello-docker]$ docker image history my-helloworld
IMAGE               CREATED             CREATED BY                                      SIZE                COMMEN
ba68f57b7991        6 minutes ago       /bin/sh -c #(nop)  LABEL organization=learni…   0B
9932ce15331b        6 minutes ago       /bin/sh -c #(nop)  LABEL author=khajaibrahim    0B
bf756fb1ae65        10 months ago       /bin/sh -c #(nop)  CMD ["/hello"]               0B
<missing>           10 months ago       /bin/sh -c #(nop) COPY file:7bf12aab75c3867a…   13.3kB
[centos@ip-172-31-26-148 hello-docker]$
```

Lets try to build a gameoflife docker image

- Select base image => tomcat with version 8 of java
- download the file from url into webapps folder
- This application runs on 8080 port
- running this app is running tomcat
- Now lets create a Dockerfile Refer Here for the changes {https://github.com/asquarezone/DockerZone/commit/3ef4e5b112f8e4b7c3bf8915e841f038c3ac67f6}

```
[centos@ip-172-31-26-148 gameoflife]$ docker image build -t gameoflife:1.0 .
Sending build context to Docker daemon  2.048kB
Step 1/4 : FROM tomcat:jdk8-openjdk
 ---> ca11a7a9d0a5
Step 2/4 : LABEL team="qtdevops"
 ---> Running in 9299e4a6c66e
Removing intermediate container 9299e4a6c66e
 ---> 2b764077bc77
Step 3/4 : RUN cd webapps/ && wget https://referenceappkhaja.s3-us-west-2.amazonaws.com/gameoflife.war
 ---> Running in aa998fecc96d
--2020-10-31 13:43:04--  https://referenceappkhaja.s3-us-west-2.amazonaws.com/gameoflife.war
Resolving referenceappkhaja.s3-us-west-2.amazonaws.com (referenceappkhaja.s3-us-west-2.amazonaws.com)... 52.218.20
7
Connecting to referenceappkhaja.s3-us-west-2.amazonaws.com (referenceappkhaja.s3-us-west-2.amazonaws.com)|52.218.2
97|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3192544 (3.0M) [binary/octet-stream]
Saving to: 'gameoflife.war'

     0K .......... .......... .......... .......... ..........  1% 36.8M 0s
    50K .......... .......... .......... .......... ..........  3% 86.4M 0s
```

```
[centos@ip-172-31-26-148 gameoflife]$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
gameoflife          1.0                 9b6c30024a10        2 minutes ago       537MB
my-helloworld       latest              ba68f57b7991        20 minutes ago      13.3kB
tomcat              jdk8-openjdk        ca11a7a9d0a5        4 days ago          534MB
nginx               alpine              4efb29ff172a        9 days ago          21.8MB
hello-world         latest              bf756fb1ae65        10 months ago       13.3kB
[centos@ip-172-31-26-148 gameoflife]$
```

Lets try to build one more image for spring pet clinic

- This application just requires java to be present.
- Copy the jar file and when you want to run the application execute command `java -jar <package>.jar`
- Over here we need to worry about how to start the application.
- Now lets look at steps
    - Try to find a image with java 8 installed
    - Copy the jar file into some path
    - Expose port 8080
    - While starting container execute the command java -jar
- Lets use FROM, LABEL instructions as usual for copying the jar file lets use wget instruction and for running the command during container starts lets use *CMD* instruction
- Refer Here for the changes {https://github.com/asquarezone/DockerZone/commit/80702d24459b6a6b9840f2c11b51377a81024dba}

From the above 3 examples (Lessons Learnt)

- While building images We need to consider
    - picking right base image
    - exposing the ports which application uses
    - commands to install/configure the application
    - commands to start the applicatio

Docker image building steps

1) To build a docker image we should always take some existing docker image as a base
2) Get the steps to configure your application on this docker image
3) Now try to use Docker file Instructions to create Docker image
4) Exercise:- Build a image for java app (spring pet clinic)
5) Lets write the manual steps.

i) Install java 8 (sudo apt-get update && sudo apt-get install openjdk-8-jdk -y)

ii) Download spring petclinic jar file from here (wget https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar)

iii) Run the application(java -jar spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar)

6) The above steps are manual steps same way we will try to create Dockerfile

1. Create Deckerville
2. Find the base image (we will take Ubuntu image)
3. And install java and the spring petclinic application file and start the application

```
{
apt-get update
apt-get install openjdk-8-jdk wget -y
wget
https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar
# Now to start application
java -jar spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar

}
```

Docker image pull and push:-

- The process of downloading the image from docker registry to docker local images is called as *pull*
- The process of uploading the image to docker registry from docker local image is called as *push*

LXC (Linux Containers)

- LXC is an os-level virtualization method for multiple isolated Linux Systems (Containers) using a single Linux Kernel
- Before we understand lxc lets try to understand the process. Process is a program in execution. In OS when we want to use any application a *process* will be launched.

Each Process will have some memory assigned to it and it is executed by cpu. Each Process will have its own Identifier which is called as Process id

- LXC creates isolated environments with each environment getting cpu, memory and network id or whatever is required to run applications in isolated environments.
- This environment is Process to the system which runs lxc and for the application running in container lxc will look like full blown os.

How application in docker is isolated

- Every app running on os will get process id
- Application will be running in a os which will have some ip adress
- Application running in a os which will have storage
- Application running in an os which will have cpu & RAM
- Application running inside docker container also will get all of the above point
- Container is an isolated area of execution with
- storage
- CPU
- RAM
- Process tree
- network interface (ip address)

Is the container new in Linux?

- no
- If you had to create a container it was very difficult and docker has simplified this process

Docker Image

- First Responsibility of DevOps Engineer: Containerize your applications. This means creating docker images for your applications.
- Docker image is collection of layers.
- Lets build a docker image for an application gameoflife
  - Installation:
    - tomcat with Java 8 installed
    - copy the gameoflife.war file into webapps folder of tomcat
  - To make this work we have taken an existing image with java and tomcat installed and copied our package into the folder and then application started working

    - Doing this on running container is not sensible, it would be better if we can have some way of creating images.
    - Steps:
        - Take existing image
        - Copy the necessary files/configurations
        - Create an image with this.
- To Create images docker has *Dockerfile* , we will be using that from our next class. Refer Here for docker file reference.{https://docs.docker.com/engine/reference/builder/}

Consideration for building Docker image for any application is

- Right base image
- Steps for installing and configuring your application
- Expressing details about the ports on which your application works
- Commands to start your application.

TOPIC:- Docker Image layers

Impartment links:-
https://directdevops.blog/2019/09/26/docker-image-creation-and-docker-image-layers/

https://directdevops.blog/2019/09/27/impact-of-image-layers-on-docker-containers-storage-drivers/

- If we can observe two image layers below “spc:0.1 & openjdk:8” images by command “docker image inspect <image name>” the layers are almost the same in spc:0.1 we can observe one extra layer
- Docker image is a collection of layers. Each layer in the Image contains its own data, storage.
- When are layers created?. Mostly they are created when the content is changed (Executing RUN, COPY, ADD instruction)

spc:0.1

```
"Layers": [
    "sha256:8803ef42039dcbe936755e9baae4bb7b19cb0fb6a438eb3992950cd0afef8e4f",
    "sha256:c2c789d2d3c5fc15428921b5316a63907d8c312c5508cbac9a704bd30bcc740d",
    "sha256:527ade4639e08d1b2ba9d0d46f3b43433310a2e8674ae808aeef5502b099ff3c",
    "sha256:2e5b4ca91984c19a75bad687aedebf02609511a033aa0d3d15ac915593279718",
    "sha256:f5181c7ef9028578eb0040a0051ce940c3f9ba62a51d3eb02ad52c3724db3e9b",
    "sha256:68bb2d4221789132994f250b401760cd45255d7f6770874bdc15375592b35e19",
    "sha256:cb8e2372b23c99b24a16f7e92dd62a75e1b46090e5414ca6dd0a72fe1a3e0841",
    "sha256:050bc94248937d7108d3c0b8cec68f30e04324c62fc62c4a834792667c096de3"
]
```

openjdk:8

```
"Layers": [
    "sha256:8803ef42039dcbe936755e9baae4bb7b19cb0fb6a438eb3992950cd0afef8e4f",
    "sha256:c2c789d2d3c5fc15428921b5316a63907d8c312c5508cbac9a704bd30bcc740d",
    "sha256:527ade4639e08d1b2ba9d0d46f3b43433310a2e8674ae808aeef5502b099ff3c",
    "sha256:2e5b4ca91984c19a75bad687aedebf02609511a033aa0d3d15ac915593279718",
    "sha256:f5181c7ef9028578eb0040a0051ce940c3f9ba62a51d3eb02ad52c3724db3e9b",
    "sha256:68bb2d4221789132994f250b401760cd45255d7f6770874bdc15375592b35e19",
    "sha256:cb8e2372b23c99b24a16f7e92dd62a75e1b46090e5414ca6dd0a72fe1a3e0841"
]
```

- For example in existing Dockerfile we will add one extra command like RUN mkdir /helow then build an image

If we can see below pic extra 2 layers are created

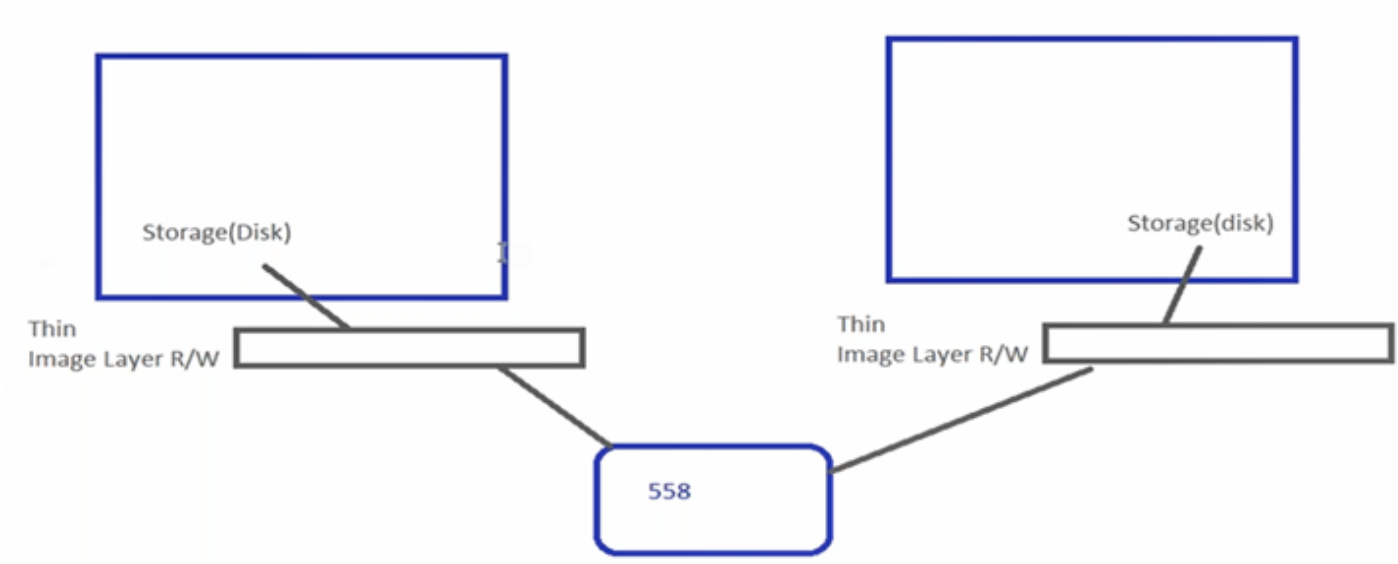
- So creating too many layers is not optimal. Use RUN statement carefully
- The docker image layers are read-only

```
"Layers": [
    "sha256:8803ef42039dcbe936755e9baae4bb7b19cb0fb6a438eb3992950cd0afef8e4f",
    "sha256:c2c789d2d3c5fc15428921b5316a63907d8c312c5508cbac9a704bd30bcc740d",
    "sha256:527ade4639e08d1b2ba9d0d46f3b43433310a2e8674ae808aeef5502b099ff3c",
    "sha256:2e5b4ca91984c19a75bad687aedebf02609511a033aa0d3d15ac915593279718",
    "sha256:f5181c7ef9028578eb0040a0051ce940c3f9ba62a51d3eb02ad52c3724db3e9b",
    "sha256:68bb2d4221789132994f250b401760cd45255d7f6770874bdc15375592b35e19",
    "sha256:cb8e2372b23c99b24a16f7e92dd62a75e1b46090e5414ca6dd0a72fe1a3e0841",
    "sha256:050bc94248937d7108d3c0b8cec68f30e04324c62fc62c4a834792667c096de3",
    "sha256:3765463560f4375856d0795b219921398c0507769453221589 6abac0db30625a"
]
```

openjdk:8

```
"Layers": [
    "sha256:8803ef42039dcbe936755e9baae4bb7b19cb0fb6a438eb3992950cd0afef8e4f",
    "sha256:c2c789d2d3c5fc15428921b5316a63907d8c312c5508cbac9a704bd30bcc740d",
    "sha256:527ade4639e08d1b2ba9d0d46f3b43433310a2e8674ae808aeef5502b099ff3c",
    "sha256:2e5b4ca91984c19a75bad687aedebf02609511a033aa0d3d15ac915593279718",
    "sha256:f5181c7ef9028578eb0040a0051ce940c3f9ba62a51d3eb02ad52c3724db3e9b",
    "sha256:68bb2d4221789132994f250b401760cd45255d7f6770874bdc15375592b35e19",
    "sha256:cb8e2372b23c99b24a16f7e92dd62a75e1b46090e5414ca6dd0a72fe1a3e0841"
]
```

- Forex if we create an image ‘X’ it will take 500MB size and again we created the same image we think it will create an additional 500MB. But here Docker will take images from the existing ‘X’ itself.
- The image layers are read-only. Whenever we create an image it will use the same image layer.
- Whenever we try to create a container Docker will create one extra read-write layer.
- Whenever we change any content in that image. Only this read-write layer will change no more modifications in original read-only layers.
- Whenever we defeat the container this read-write layer also created
- When we create a container, an extra R/w layer is created. All the changes done in that container will be stored in the R/W layer. When we delete the container the R/W layer gets deleted. So in Docker to preserve changes done inside the container we need to take an extra measure (Volume)



Stateful and Stateless Application

- Stateful applications use local storage to preserve application data/config
- Stateless applications don’t use local storage, they rather store the state(application/data) in a database/external system
- Docker is used in both stateful and stateless applications.
- Ensure extra caution in stateful applications.

What are DevOps Engineer Responsibilities over here?

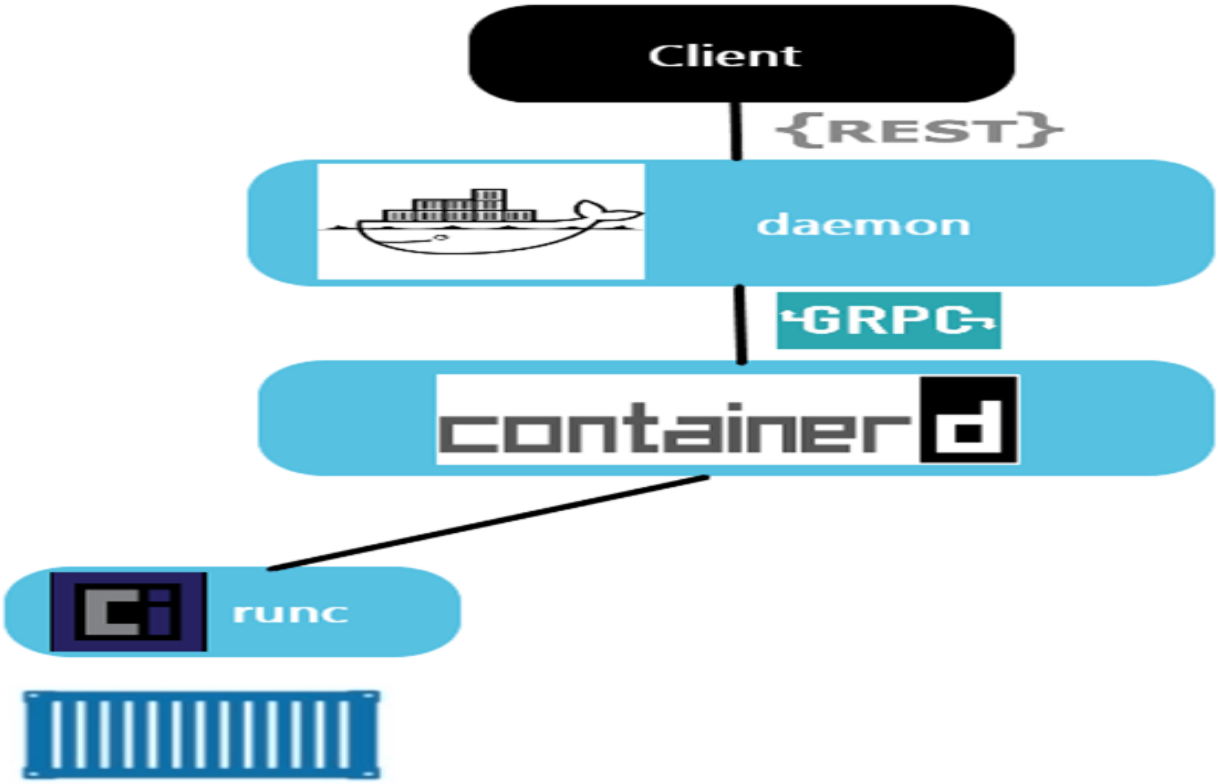
- Ensure you can containerize the applications (To container applications we need to understand the steps of application deployment). DevOps Engineer should be able to use and create new docker images

Topic:- How Docker Works

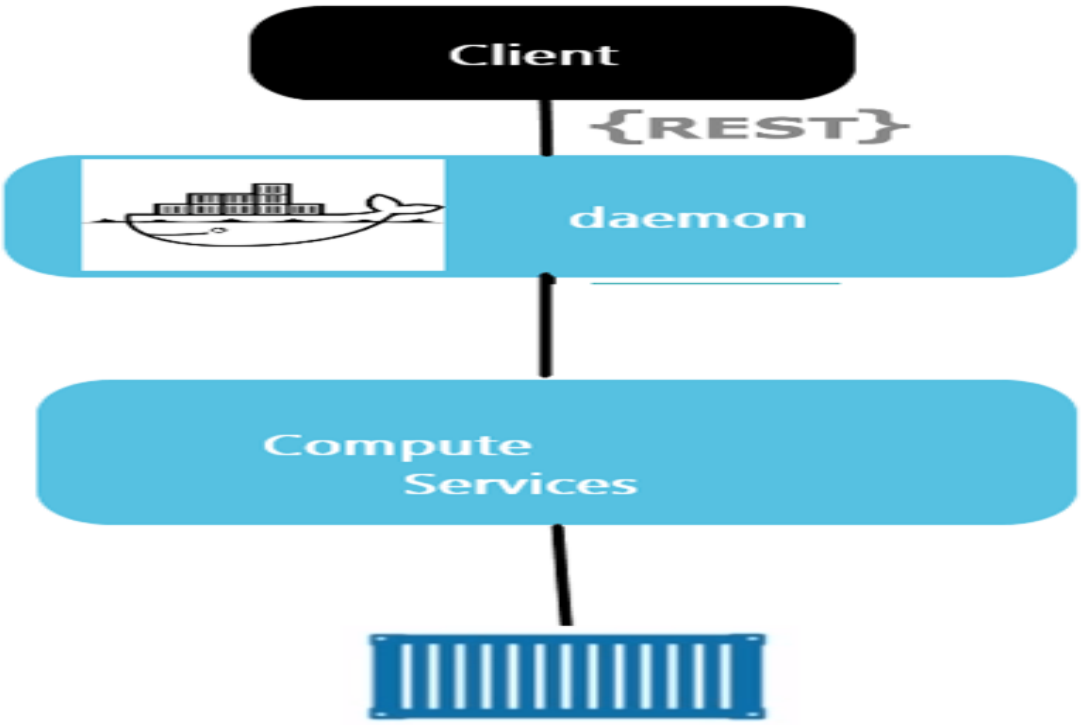
- Linux Container is used to isolate processes. This process isolation will contain all the code, dependencies, binaries and settings required to run the process.
- Docker containers when they started used to create linux containers. Linux releases where impacting docker containers, so they have written their own container creation library (runc)
- Refer Here for understanding docker internals {https://directdevops.blog/2019/01/31/docker-internals/}
- Inside the container (process isolations), it has its own process tree and it has its own filesystem.
- CPU and memory are also allocated to the container
- This isolated process has its own ip address
- Container is getting whatever is needed to run the application. But How?
    - Namespaces
    - CGroups
- Process Isolation: Each container runs with its own kernel namespaces for the following
    - Processes
    - Networks
    - Users
    - Mount
    - IPC

- UTS

- When we make a call to create a container it will ask to docker-demon
- And docker-demon will call to container-d to create a container
- Then it will call to a component called “runc” there a process is created (fork) it will fork “runc” then a container to be created.
- Then an isolations to be created for the container using namespace , cgroups
- With this approach advantage is if we need to upgrade the docker we can upgrade it still container will be running.
- Container will run independent of docker daemon



The Specific Windows Implementation will be as shown below



Topic:- Docker containers

- Docker Container is created from Docker Image. Docker Image is a collection of Image Layers. Now while docker container is created all of the Image Layers are mounted as one file system
- When the docker container is created CMD/ENTRYPOINT gets executed and as long as this command is running the container will be in running state.
- When we create a docker container. And we run the container the container will be live until the CMD/ENTRYPOINT is executing
- For ex:- the container spc:1.0 CMD[“java” “-jar” “spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar”] if we executed docker container run –d spc:1.0 (the container will be in running state
- If we executed like “docker container run –d spec:1.0 echo hollow” then docker will execute only echo command and it will execute caz echo will replace the CMD so docker executed the command and exit
- when we create a docker container every container will have a name and id.
- Every Docker container will have a unique container name and id. if the names are not passed while creating, docker engine gives adhoc names
- If we want to work with particular container we need to know that container name/ID

```
$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS               NAMES
c0c907ec3c29        spc:1.0             "echo hello"             2 minutes ago       Exited (0) 2 minutes ago                        friendly_lamarr
967fe9960109        spc:1.0             "java -jar spring-pe…"   4 minutes ago       Up 4 minutes                8080/tcp            kind_shockley
3f8d87da6a96        spc2:0.1            "/bin/bash"              22 minutes ago      Exited (0) 13 minutes ago                       festive_bartik
7d0bab22e3fe        spc2:0.1            "/bin/bash"              24 minutes ago      Exited (0) 23 minutes ago                       admiring_bell
7a7b60b2de57        test:0.1            "/bin/bash"              39 minutes ago      Exited (0) 39 minutes ago                       optimistic_cori
f05420ead932        tomcat:8            "/bin/bash"              45 minutes ago      Exited (0) 41 minutes ago                       cool_bell
302a08a0441e        jenkins             "/bin/tini -- /usr/l…"   46 minutes ago      Exited (0) 45 minutes ago                       blissful_blackbur
n
8bfef888d4de        openjdk:8           "/bin/bash"              46 minutes ago      Exited (0) 46 minutes ago                       sleepy_goldberg
```

* To create a container with specified name “docker container run --name <name> -d <container name>”

Ex:- docker container run --name mahi -d env:1.0

```
root@ip-172-31-80-14:/docker/file5# docker container run --name mahi -d env:1.0
07b4ac029fe3e165a33729832e959c0f6f96f95517bf1403e8666ffacbb35a08
root@ip-172-31-80-14:/docker/file5# docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
07b4ac029fe3        env:1.0             "java -jar spring-pe…"   17 seconds ago      Up 16 seconds       8080/tcp            mahi
root@ip-172-31-80-14:/docker/file5#
```


- docker container is started in a detached mode. Is there any way to login into it
- execute `docker attach <cont-id or name>`
- execute `docker container exec -it <cont> /bin/bash`

```
[node1] (local) root@192.168.0.13 ~
$ docker container run --name gol-1 -d -P qualitythought/gameoflife:07112020
4eaef41369479f06d2b20dd37f5ea404d060c9c580fd09fa26f683c2de33191d
[node1] (local) root@192.168.0.13 ~
$ docker container exec gol-1 ps
OCI runtime exec failed: exec failed: container_linux.go:349: starting container process cause
le file not found in $PATH": unknown
[node1] (local) root@192.168.0.13 ~
$ docker container exec gol-1 ls
BUILDING.txt
CONTRIBUTING.md
LICENSE
NOTICE
README.md
RELEASE-NOTES
RUNNING.txt
bin
conf
lib
logs
native-jni-lib
temp
webapps
webapps.dist
work
[node1] (local) root@192.168.0.13 ~
```

```
[node1] (local) root@192.168.0.13 ~
$ docker container exec -it gol-1 /bin/bash
root@4eaef4136947:/usr/local/tomcat# ps -aux
bash: ps: command not found
root@4eaef4136947:/usr/local/tomcat# ip addr
bash: ip: command not found
root@4eaef4136947:/usr/local/tomcat# ls
BUILDING.txt     LICENSE  README.md      RUNNING.txt  conf  logs            temp     webapps.dist
CONTRIBUTING.md  NOTICE   RELEASE-NOTES  bin          lib   native-jni-lib  webapps  work
root@4eaef4136947:/usr/local/tomcat#
```

- Remove all docker containers in running and stopped state {docker container rm container id / docker container rm -f $(docker container ls -q) }
- Removing all docker container which are in stop position {docker container rm -f $(docker container ls -q -a) }
- Logs: The logs command will show all the *standard out-put* stream of your container{docker container logs containername:version}
- Also we can give filters to this logs like {docker container logs --since 2020-11-06T05:00 969acaf42cd1}

```
$ docker container logs gol-1
07-Nov-2020 14:30:06.731 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Server version name
e Tomcat/9.0.39
07-Nov-2020 14:30:06.735 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Server built:
 2020 14:11:46 UTC
07-Nov-2020 14:30:06.735 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Server version numb
9.0
07-Nov-2020 14:30:06.735 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log OS Name:
07-Nov-2020 14:30:06.736 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log OS Version:
-193-generic
07-Nov-2020 14:30:06.736 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Architecture:
07-Nov-2020 14:30:06.736 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Java Home:
local/openjdk-8/jre
07-Nov-2020 14:30:06.736 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log JVM Version:
_272-b10
07-Nov-2020 14:30:06.737 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log JVM Vendor:
e Corporation
07-Nov-2020 14:30:06.737 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log CATALINA_BASE:
local/tomcat
07-Nov-2020 14:30:06.737 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log CATALINA_HOME:
```

```
$ docker container logs --since 2020-11-06T05:00   gol-1
07-Nov-2020 14:30:06.731 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Serve
e Tomcat/9.0.39
07-Nov-2020 14:30:06.735 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Serve
 2020 14:11:46 UTC
07-Nov-2020 14:30:06.735 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Serve
9.0
07-Nov-2020 14:30:06.735 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log OS Na
07-Nov-2020 14:30:06.736 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log OS Ve
-193-generic
07-Nov-2020 14:30:06.736 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Archi
07-Nov-2020 14:30:06.736 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Java
local/openjdk-8/jre
07-Nov-2020 14:30:06.736 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log JVM V
_272-b10
07-Nov-2020 14:30:06.737 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log JVM V
e Corporation
07-Nov-2020 14:30:06.737 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log CATAL
local/tomcat
07-Nov-2020 14:30:06.737 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log CATAL
local/tomcat
07-Nov-2020 14:30:06.741 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Comma
a.util.logging.config.file=/usr/local/tomcat/conf/logging.properties
07-Nov-2020 14:30:06.741 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Comma
```

- The *stats* provides real-time information on the specified container or if you don't pass *name* or *id* it shows for all the running containers docker container stats

```
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT     MEM %               NET I/O
  BLOCK I/O           PIDS
4eaef4136947        gol-1               0.15%               244.3MiB / 31.4GiB    0.76%               0B / 0B
  0B / 0B             38
[]
```

```
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT     MEM %               NET I/O
  BLOCK I/O           PIDS
6537824a6ad0        gol-2               0.20%               240.9MiB / 31.4GiB    0.75%               0B / 0B
  0B / 0B             38
4eaef4136947        gol-1               0.43%               244.3MiB / 31.4GiB    0.76%               0B / 0B
  0B / 0B             38
[]
```

- TO put restrictions on docker container on how many CPUs it used and how much ram it want to use {docker container run -d --name gov --cpus 1 --memory 128m -P gol:1.0}

```
[centos@ip-172-31-26-148 ~]$ docker container run -d --name gol-3 --cpu-shares 512 --memory 128M -P gameoflife:071126
20
d33df13848fdb5ac05c2a997486ffc1bc532726d563bb3722e14484af3028cd8
[centos@ip-172-31-26-148 ~]$ docker container stats
```

```
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT   MEM %               NET I/O
   BLOCK I/O           PIDS
d33df13848fd        gol-3               0.19%               84.82MiB / 128MiB   66.26%              656B / 0B
   15.1MB / 0B         28
```

```
[centos@ip-172-31-26-148 ~]$ docker container run -d --name gol-4 -P gameoflife:07112020
3954692443f3b41532b961495431758171a447ada21998b95b183c2b21965214
[centos@ip-172-31-26-148 ~]$ docker container stats
```

```
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT     MEM %               NET I/O
     BLOCK I/O           PIDS
3954692443f3        gol-4               0.18%               78.96MiB / 989.3MiB   7.98%               656B / 0B
     6.64MB / 0B         28
d33df13848fd        gol-3               0.21%               84.87MiB / 128MiB     66.30%              656B / 0B
     15.1MB / 0B         28
```

NOTE:- if we don't give any restrictions then docker will automatically take all the cpus and memory it have and allocate which ever the container wants

- If we need to increase memory and cpu we can achieve it {docker container update --memory 256M gol:3.0 gol:3.0 }

```
[centos@ip-172-31-26-148 ~]$ docker container update --memory 256M gol-3
gol-3
[centos@ip-172-31-26-148 ~]$ docker container stats
```

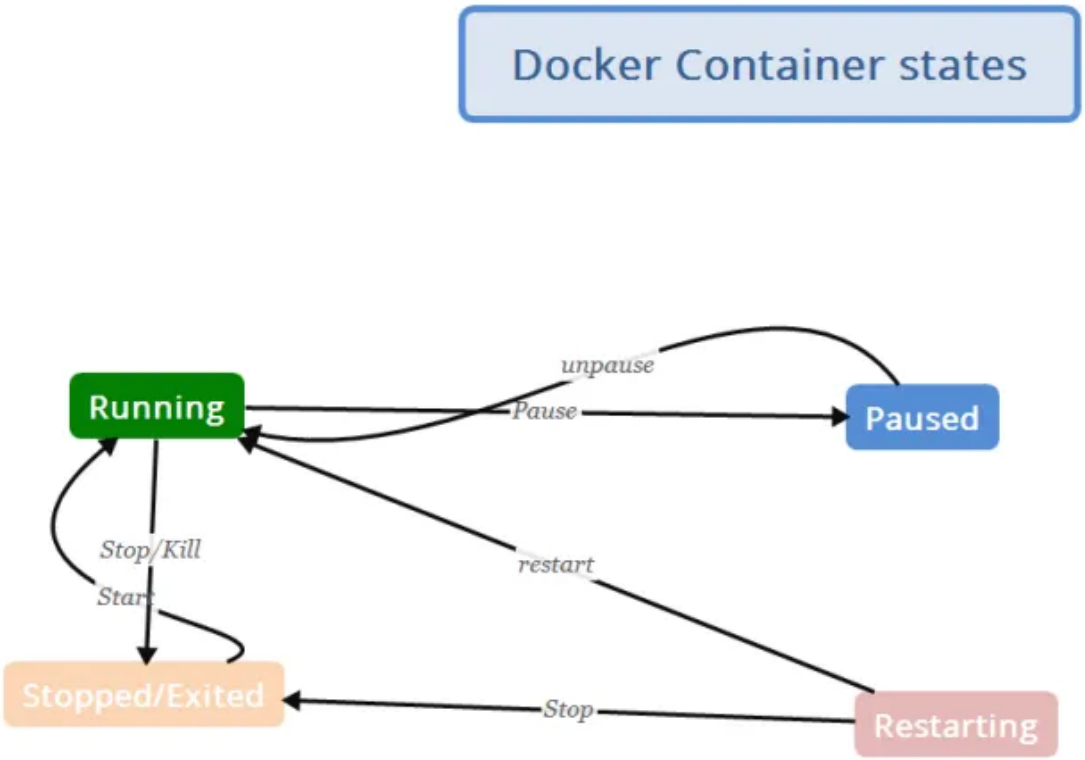
```
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT     MEM %               NET I/O
     BLOCK I/O           PIDS
3954692443f3        gol-4               0.19%               79.12MiB / 989.3MiB   8.00%               656B / 0B
     6.64MB / 0B         28
d33df13848fd        gol-3               0.16%               84.92MiB / 256MiB     33.17%              656B / 0B
     15.1MB / 0B         28
```

- TO inspect docker containers {docker container inspect gol:3.0 }
- To view a specific portion of container for ex memory we can achieve by grep command {docker inspect gol:3.0 | grep -i memory}

```
[centos@ip-172-31-26-148 ~]$ docker container inspect gol-3
[
    {
        "Id": "d33df13848fdb5ac05c2a997486ffc1bc532726d563bb3722e14484af3028cd8",
        "Created": "2020-11-07T14:46:15.185014457Z",
        "Path": "catalina.sh",
        "Args": [
            "run"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 2496,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2020-11-07T14:46:15.854203803Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        "Image": "sha256:24ceb4ef1b9fa48e7f565c7758a227fad9660d3e8ba78baa98e37fbb4cf62bb8",
        "ResolvConfPath": "/var/lib/docker/containers/d33df13848fdb5ac05c2a997486ffc1bc532726d563bb3722e14484af3028
8/resolv.conf",
        "HostnamePath": "/var/lib/docker/containers/d33df13848fdb5ac05c2a997486ffc1bc532726d563bb3722e14484af3028cd
hostname",
        "HostsPath": "/var/lib/docker/containers/d33df13848fdb5ac05c2a997486ffc1bc532726d563bb3722e14484af3028cd8/h
```

```
[centos@ip-172-31-26-148 ~]$ docker container inspect gol-3 | grep -i memory
            "Memory": 268435456,
            "KernelMemory": 0,
            "KernelMemoryTCP": 0,
            "MemoryReservation": 0,
            "MemorySwap": 268435456,
            "MemorySwappiness": null,
```

- Docker container has states like :-
- Running
- Exited/Stopped
- Pause
- Terminated (Deleted)



- TO pause {docker container pause container_name}
- To unpause {docker container unpause container_name}
- To stop the container {docker container stop container_name}
- To start the container {docker container start container_name}

```
$ docker container ls
CONTAINER ID        IMAGE                                  COMMAND             CREATED             STATUS
RTS                     NAMES
78d8cd4b8683        qualitythought/gameoflife:07112020   "catalina.sh run"   18 minutes ago      Up 18 minutes
0.0.0:32775->8080/tcp   gol-3
6537824a6ad0        qualitythought/gameoflife:07112020   "catalina.sh run"   22 minutes ago      Up 22 minutes
0.0.0:32774->8080/tcp   gol-2
4eaef4136947        qualitythought/gameoflife:07112020   "catalina.sh run"   32 minutes ago      Up 32 minutes
0.0.0:32773->8080/tcp   gol-1
[node1] (local) root@192.168.0.13 ~
$ docker container pause gol-1
gol-1
[node1] (local) root@192.168.0.13 ~
$ docker container ls
CONTAINER ID        IMAGE                                  COMMAND             CREATED             STATUS
   PORTS                    NAMES
78d8cd4b8683        qualitythought/gameoflife:07112020   "catalina.sh run"   18 minutes ago      Up 18 minutes
   0.0.0.0:32775->8080/tcp   gol-3
6537824a6ad0        qualitythought/gameoflife:07112020   "catalina.sh run"   22 minutes ago      Up 22 minutes
   0.0.0.0:32774->8080/tcp   gol-2
4eaef4136947        qualitythought/gameoflife:07112020   "catalina.sh run"   33 minutes ago      Up 33 minutes (Paused)
   0.0.0.0:32773->8080/tcp   gol-1
[node1] (local) root@192.168.0.13 ~
$
```

```
$ docker container unpause gol-1
gol-1
[node1] (local) root@192.168.0.13 ~
$ docker container ls
CONTAINER ID        IMAGE                                  COMMAND             CREATED             STATUS
RTS                     NAMES
78d8cd4b8683        qualitythought/gameoflife:07112020   "catalina.sh run"   19 minutes ago      Up 19 minutes
0.0.0:32775->8080/tcp   gol-3
6537824a6ad0        qualitythought/gameoflife:07112020   "catalina.sh run"   23 minutes ago      Up 23 minutes
0.0.0:32774->8080/tcp   gol-2
4eaef4136947        qualitythought/gameoflife:07112020   "catalina.sh run"   34 minutes ago      Up 34 minutes
0.0.0:32773->8080/tcp   gol-1
```

```
$ docker container stop gol-1
gol-1
[node1] (local) root@192.168.0.13 ~
$ docker container ls
CONTAINER ID        IMAGE                                  COMMAND             CREATED             STATUS
RTS                     NAMES
78d8cd4b8683        qualitythought/gameoflife:07112020   "catalina.sh run"   20 minutes ago      Up 20 minutes
0.0.0:32775->8080/tcp   gol-3
6537824a6ad0        qualitythought/gameoflife:07112020   "catalina.sh run"   24 minutes ago      Up 24 minutes
0.0.0:32774->8080/tcp   gol-2
[node1] (local) root@192.168.0.13 ~
$ docker container ls -a
CONTAINER ID        IMAGE                                  COMMAND             CREATED             STATUS
       PORTS                    NAMES
78d8cd4b8683        qualitythought/gameoflife:07112020   "catalina.sh run"   20 minutes ago      Up 20 minutes
       0.0.0.0:32775->8080/tcp   gol-3
6537824a6ad0        qualitythought/gameoflife:07112020   "catalina.sh run"   24 minutes ago      Up 24 minutes
       0.0.0.0:32774->8080/tcp   gol-2
4eaef4136947        qualitythought/gameoflife:07112020   "catalina.sh run"   35 minutes ago      Exited (143) 9 sec
 ago                            gol-1
[node1] (local) root@192.168.0.13 ~
```

```
[node1] (local) root@192.168.0.13 ~
$ docker container start gol-1
gol-1
[node1] (local) root@192.168.0.13 ~
$ docker container ls
CONTAINER ID        IMAGE                                  COMMAND             CREATED             STATUS
RTS                     NAMES
78d8cd4b8683        qualitythought/gameoflife:07112020   "catalina.sh run"   21 minutes ago      Up 21 minutes
0.0.0:32775->8080/tcp   gol-3
6537824a6ad0        qualitythought/gameoflife:07112020   "catalina.sh run"   25 minutes ago      Up 25 minutes
0.0.0:32774->8080/tcp   gol-2
4eaef4136947        qualitythought/gameoflife:07112020   "catalina.sh run"   36 minutes ago      Up 5 seconds
0.0.0:32776->8080/tcp   gol-1
[node1] (local) root@192.168.0.13 ~
$
```

```
attach      Attach local standard input, output, and error streams to a running container
commit      Create a new image from a container's changes
cp          Copy files/folders between a container and the local filesystem
create      Create a new container
diff        Inspect changes to files or directories on a container's filesystem
exec        Run a command in a running container
export      Export a container's filesystem as a tar archive
inspect     Display detailed information on one or more containers
kill        Kill one or more running containers
logs        Fetch the logs of a container
ls          List containers
pause       Pause all processes within one or more containers
port        List port mappings or a specific mapping for the container
prune       Remove all stopped containers
rename      Rename a container
restart     Restart one or more containers
rm          Remove one or more containers
run         Run a command in a new container
start       Start one or more stopped containers
stats       Display a live stream of container(s) resource usage statistics
stop        Stop one or more running containers
top         Display the running processes of a container
unpause     Unpause all processes within one or more containers
update      Update configuration of one or more containers
wait        Block until one or more containers stop, then print their exit codes
```

Docker container running modes:

1) attached: This is default mode

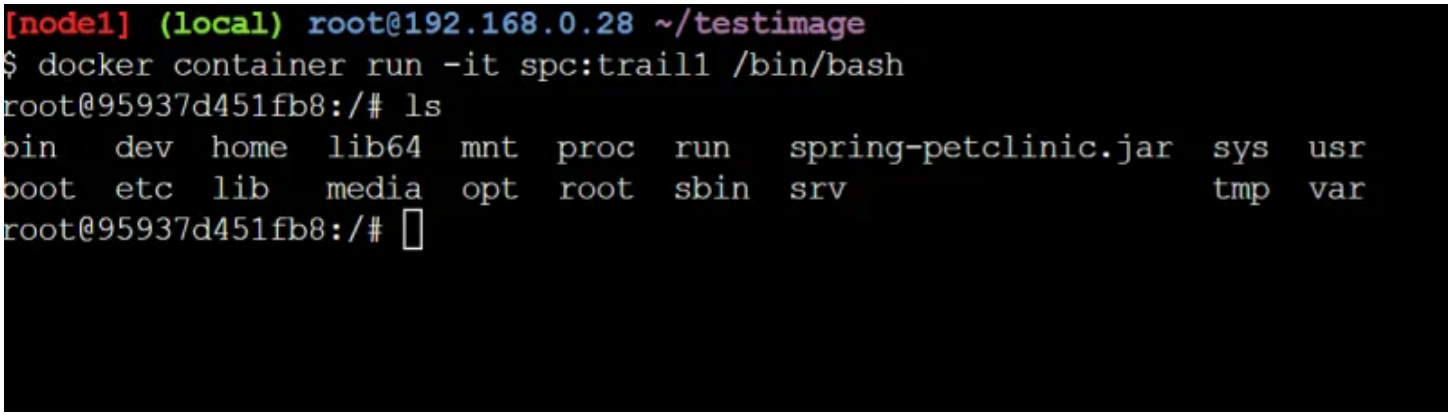
   Ex:- docker container run <tag> <command> {docker container run spc:trail2 java -jar spring.jar}

2) detached/background: in this mode your container will just show the container id and run in background

   EX:- docker container run -d <image>:<tag>

3) interactive: In this mode we login into the terminal of container

   EX:- docker container run -it <image>:<tag> <terminalpath>



Topic:- Dockerfile

https://docs.docker.com/engine/reference/builder/

https://docs.docker.com/engine/reference/builder/

- A container using the base image gets created.
- Then some instructions such as RUN etc gets executed
- Copy of this container is created as a Docker image
- Image is created with the name specified in the command
- When we run a container using our created image, then ENTRYPOINT/CMD gets executed

- While building a docker image divide activities into the following categories

A. Base image with necessary software (if available)
B. execute steps for configuring/deploying application
C. steps for running your application

- docker image build builds an image from '*Dockerfile'* and set of files specified in the PATH (Local directory) or URL (GIT Repository Location) which is called as *context* (Ex:- `docker image build -t hellodocker <PATH or URL>`)
- *"Dockerfile"* is a standard name but if you want to write instruction in a different file that is possible

https://docs.docker.com/engine/reference/builder/

FROM

- This instruction sets the base image

Syntax

```
FROM <image>[:tag] [AS <name>]
```

Examples:

```
FROM ubuntu
FROM openjdk:8
```

- Best practice: always mention tag donot use latest
- also gives and optional way to specify platforms lick linux/amd64, linux/arm64, window/arm64 etc

```
FROM [--platform=<platform>] <image>[:tag] [AS <name>]
```

RUN:

- This instruction execute the commands in the created base image container in a new *layer*
- Run command runs with /bin/bash -c on Linux and cmd on windows

Syntax: "RUN <command>"

```
RUN ["executable", "param1", "param2"]
```

Example:

```
RUN wget https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar

RUN /bin/bash -c 'echo sample'

RUN ["/bin/bash", "-c", "echo sample"]
```

Commands have two forms

- shell =>
    - /bin/bash -c 'echo sample'
    - java -jar spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar

.

- exec => represent in square brackets

  · ["/bin/bash", "-c", "echo sample"]
  ["java", "-jar", "spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar"

ENTRYPOINT

- This is the command that gets executed when starting the container. if ENTRYPOINT is not found then CMD is executed when starting the container
- Syntax:

```
ENTRYPOINT ["executable", "param-1", "param-2" ]
ENTRYPOINT executable  param-1 param-2
```

NOTE:- If both ENTRYPOINT and CMD Exists then ENTRYPOINT will be executable and CMD will be arguments

CMD

Syntax:-

```
CMD ["executable", "param-1", "param-2" ]
CMD executable  param-1 param-2
CMD param-1 param-2
```

ENTRYPOINT & CMD examples:-

EX:1

(Create a image with tag testing:1.0 with the following Dockerfile)

```
FROM openjdk:8
RUN wget
https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPS
HOT.jar
EXPOSE 8080
ENTRYPOINT ["java",  "-jar",  "spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar"]
```

EX:2 (Create a image with tag testing:2.0 with the following Dockerfile)

```
FROM openjdk:8
```

```
RUN wget
https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPS
HOT.jar
EXPOSE 8080
CMD ["java",  "-jar",  "spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar"]
```

EX:3 (Create a image with tag testing:2.0 with the following Dockerfile)

```
FROM openjdk:8
RUN wget
https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPS
HOT.jar
EXPOSE 8080
ENTRYPOINT ["java"]
CMD ["-jar",  "spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar"]
```

We will execute the above Dockerfile and see the images below

```
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
test                3.0                 f917081180e6        21 seconds ago      561MB
test                2.0                 a31e401f7280        2 minutes ago       561MB
test                latest              1ec05366a4ea        4 minutes ago       561MB
```

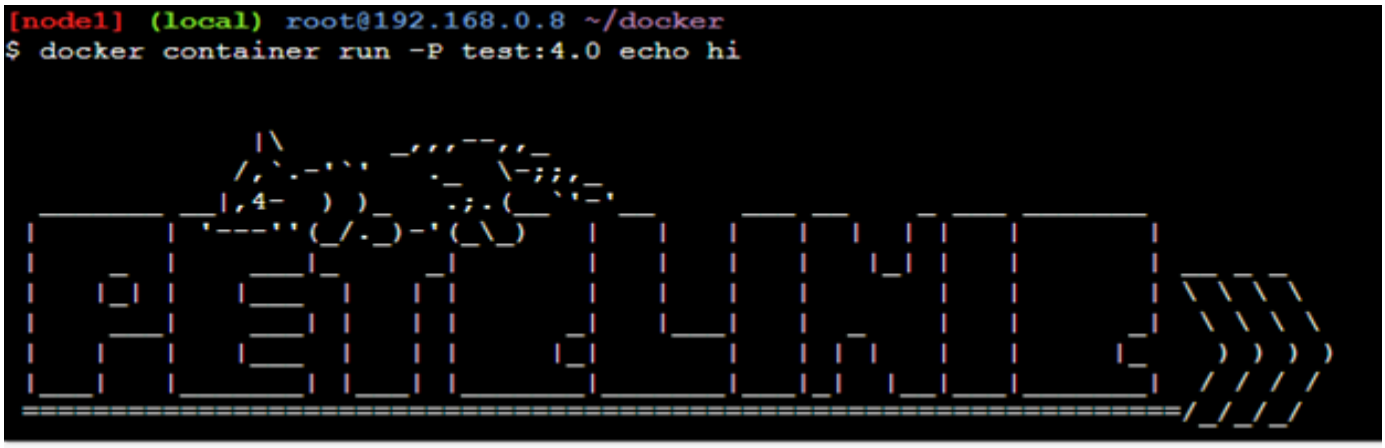
NOTE:- ENTRYPOINT cannot be changed with args but whatever we write in *CMD* can be overwritten with args

```
$ docker container run --help

Usage:  docker container run [OPTIONS] IMAGE [COMMAND] [ARG...]

Run a command in a new container

Options:
      --add-host list                Add a custom host-to-IP mapping (host:ip)
  -a, --attach list                  Attach to STDIN, STDOUT or STDERR
      --blkio-weight uint16          Block IO (relative weight), between 10 and 1000, or 0 to
                                     disable (default 0)
      --blkio-weight-device list     Block IO weight (relative device weight) (default [])
      --cap-add list                 Add Linux capabilities
      --cap-drop list                Drop Linux capabilities
      --cgroup-parent string         Optional parent cgroup for the container
      --cidfile string               Write the container ID to the file
      --cpu-period int               Limit CPU CFS (Completely Fair Scheduler) period
      --cpu-quota int                Limit CPU CFS (Completely Fair Scheduler) quota
      --cpu-rt-period int            Limit CPU real-time period in microseconds
      --cpu-rt-runtime int           Limit CPU real-time runtime in microseconds
  -c, --cpu-shares int               CPU shares (relative weight)
      --cpus decimal                 Number of CPUs
```

For ex:- we have “test” with ENTRYPOINT now we will pass the argument
docker container run –P test echo helloworld
there is no change in execution ENTRYPOINT will not effected
same way in test:2.0 we have CMD so we will pass arguments at run time and see. Below pic the output came hello

```
[node1] (local) root@192.168.0.8 ~/docker
$ docker container run -P test:2.0 echo hellow
hellow
```

In the same way in “test:3.0” ENTRYPOINT [“java”] CMD[] so if we pass argument CMD will execute with argument pass and we have ENTRYPOINT here so only it can execute and



EXPOSE:

- This instruction informs Docker that the application listens on the specific network ports at runtime.
- Two Protocols can be specified TCP and UDP. TCP is default

Syntax:

```
EXPOSE <port> [<port/protocol>]
```

- Example

```
FROM openjdk:8
RUN wget
https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar
EXPOSE 8080
ENTRYPOINT ["java"]
CMD ["-jar",  "spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar"]
```

- Execute the following commands

```
docker image build -t spc:0.1 .

docker container run -P -d spc:0.1

docker container ls
```

LABEL:

- This instruction adds metadata to an image

Syntax:

```
LABEL <key>=<value> <key>=<value>
```

Example:

```
FROM openjdk:8
LABEL author="khaja"
LABEL version="0.2"
LABEL project="QT"
RUN wget
https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar
EXPOSE 8080
ENTRYPOINT ["java"]
CMD ["-jar",  "spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar"]
```

ADD and COPY:

- Both ADD and COPY instructions are used to copy the files into docker image.
- ADD instruction supports copying files from url and local machines
- COPY instruction supports only copying files from local machines

Syntax:

```
ADD <src> <dest>
COPY <src> <dest>
```

Example ADD with URL:

```
FROM openjdk:8
LABEL author="khaja"
LABEL version="0.3"
LABEL project="QT"
ADD https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar
/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar
EXPOSE 8080
ENTRYPOINT ["java"]
CMD ["-jar",  "/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar"]
```

Example Add/Copy with local path

```
FROM openjdk:8
LABEL author="khaja"
LABEL version="0.4"
LABEL project="QT"
COPY spring-petclinic.jar  /spring-petclinic.jar
EXPOSE 8080
ENTRYPOINT ["java"]
CMD ["-jar",  "/spring-petclinic.jar"]
```

ARG

- While building the image. Values of ARG will be changed
- ARG: this instruction allows variables to be passed while building an image.
- ARG is available only while building the docker image.
- While building images we need some customizations we go with ARG.
- ARG: this instruction allows variables to be passed while building image

Syntax:- `docker image build --build-arg url=<otherurl> -t spc:0.1 .`

Syntax:- `docker image build --build-arg download_location="<newlocation>" ...`

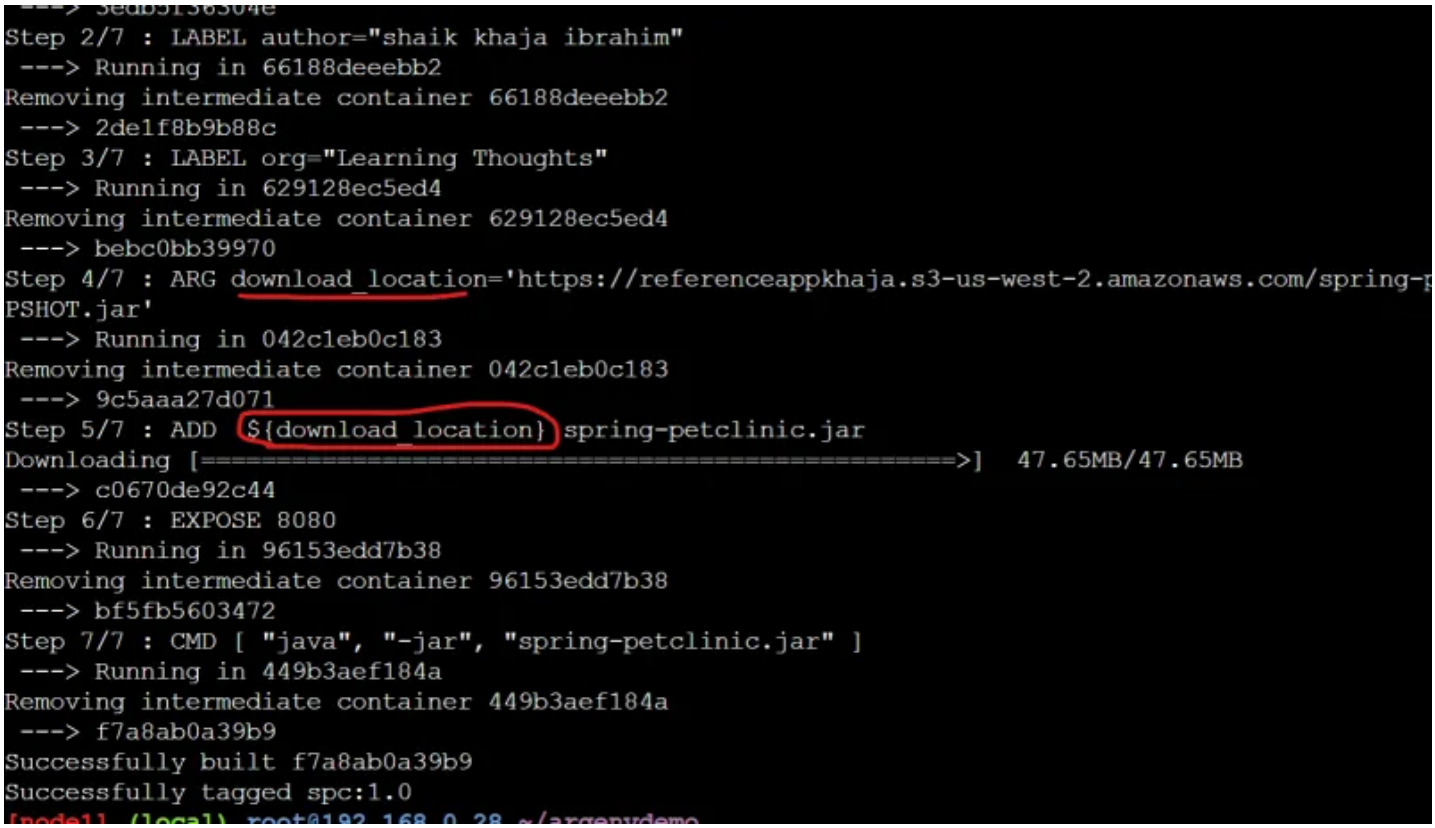
EX:

```
FROM openjdk:8
ARG url=https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar
RUN wget ${url}
EXPOSE 8080
CMD ["java", “-jar", “spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar"]
```

EX:-

```
FROM openjdk:8
LABEL author="shaik khaja ibrahim"
LABEL org="Learning Thoughts"
# Adding an argument with default value
# This value can be changed by passing
# docker image build --build-arg download_location="<newlocation>" ....
ARG  download_location='https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar'
ADD  ${download_location} spring-petclinic.jar
EXPOSE 8080
CMD [ "java", "-jar", "spring-petclinic.jar" ]
```

- If we we can build the image using the above dockerfile, “docker image build --build-arg 'https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar' -t spc:1.0 .



- We have define variable called “download location” and we have used that by using $download_location
- Lets create a container with this image and we will go inside the container and check for the variables “docker container run --name myexp-1 -d -P spc:1.0 ”
- Lets enter into this image and verify the ARG “docker exec -it myexp-1 /bin/bash” and execute set command, set command will show all available Environment variables.

```
$ docker image build --build-arg download_location='https://referenceappkhaja.s3-us-west-2.amazonaws.com/spr
ic-2.2.0.BUILD-SNAPSHOT.jar' -t spc:1.0 .
[node1] (local) root@192.168.0.28 ~/argenvdemo
$ docker exec -it myexp-1 /bin/bash
root@c593bd5a8954:/# set
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:complete_fullquote:expand_aliases:extquote:force_fignore:globasciiranges:hostc
eractive_comments:progcomp:promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=([0]="0")
BASH_ARGV=()
BASH_CMDS=()
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSINFO=([0]="5" [1]="0" [2]="3" [3]="1" [4]="release" [5]="x86_64-pc-linux-gnu")
BASH_VERSION='5.0.3(1)-release'
COLUMNS=78
DIRSTACK=()
EUID=0
GROUPS=()
HISTFILE=/root/.bash_history
HISTFILESIZE=500
HISTSIZE=500
HOME=/root
HOSTNAME=c593bd5a8954
HOSTTYPE=x86_64
IFS=$' \t\n'
JAVA_HOME=/usr/local/openjdk-8
```

https://docs.docker.com/engine/reference/builder/#arg

ENV:

· This instruction will set the environmental variable for all the subsequent instruction and also in containers Build the image

- Environmental variables can be replaced while creating container
- ENV variables are available while building the image and also while the container is running as environmental variables

Syntax:- docker container run –it –e mycontainername=nothing <image name>

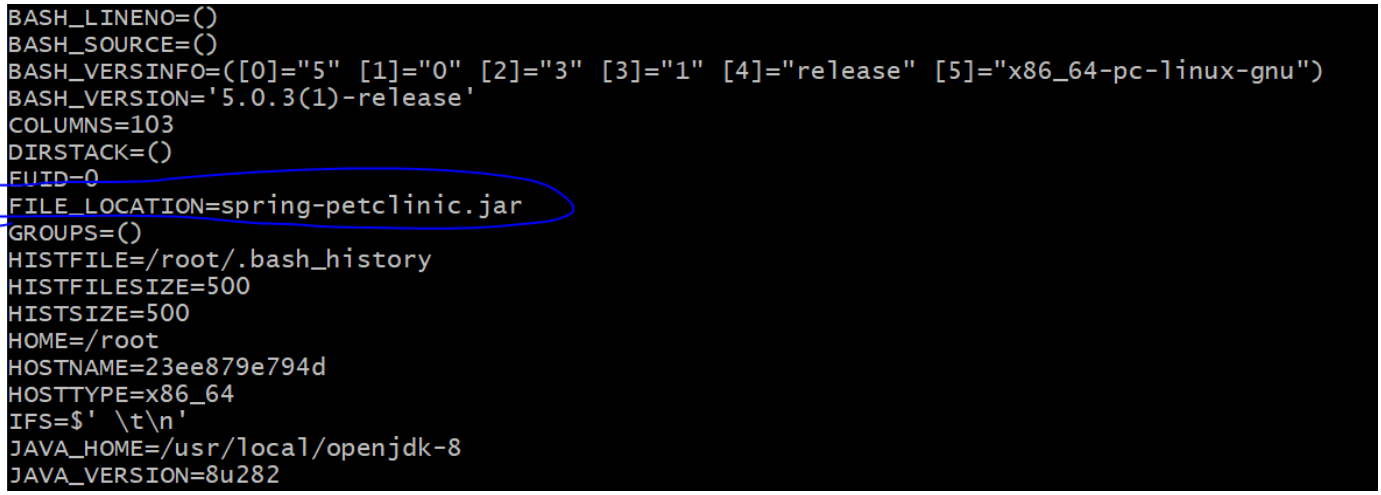
EX;

```
FROM openjdk:8
ARG url=https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar
ENV myfilename=spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar
RUN wget ${url}
EXPOSE 8080
CMD ["java",  "-jar",  "spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar"]
```

EX:-

```
FROM openjdk:8
LABEL author="shaik khaja ibrahim"
LABEL org="Learning Thoughts"
# Adding an argument with default value
# This value can be changed by passing
# docker image build --build-arg download_location="<newlocation>" ....
ARG download_location='https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar'
ENV FILE_LOCATION='spring-petclinic.jar'
ADD ${download_location} ${FILE_LOCATION}
EXPOSE 8080
CMD java -jar ${FILE_LOCATION}
```

- Let's build a docker image with this dockerfile “docker image build --build-arg download_location='https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar' -t spc:2.0 . ”
- And let's create container “docker container run --name envdemo -d -P spc:2.0 ”
- And verify the environment variables “docker container exec -it envdemo /bin/bash”



USER:

- user instruction sets the username for any subsequent RUN, CMD and ENTRYPOINT instructions in Dockerfile. Ensure the user is created before USER instruction in dockerfile. Default username is root

WORKDIR

- workdir instruction sets the working directory for any subsequent RUN, CMD and ENTRYPOINT instructions in Dockerfile and the default value would be /

ONBUILD

- This instruction adds to the image a trigger to be executed at later time when the image is used as the base image for another build

STOPSIGNAL

- The STOPSIGNAL instruction sets the system call signal that will be sent to the container to exit.

HEALTHCHECK

- This instruction tells how to test a container to check if it is still working

Syntax:-

```
HEALTHCHECK --interval=5m --timeout=5s CMD curl -f http://localhost:8080 || exit 1
```

WORKDIR

- WORKDIR sets the default home directory.

USER:

- Instruction will set the user. default user will be root. Before using USER instruction user should be created.

Sample Dockerfile

Ex: 1

```
FROM ubuntu
RUN apt-get update && apt-get install openjdk-8-jdk wget -y
RUN wget https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar
EXPOSE 8080
CMD ["java", "-jar", "spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar"]
```

Activity:

- We can use docker to build applications and package them as containers
- Lets manually build game of life using docker

- For game of life code {https://github.com/wakaleo/game-of-life}
- Commands to build is i) Ensure maven is installed ii) `mvn package` is the command
- Now lets try to create a container with image where maven is installed{docker container run -it maven:3-openjdk-8 /bin/bash}
- Once we enter in to the container clone the code from git hub "git clone https://github.com/wakaleo/game-of-life.git"
- Go to the pom.xml location and execute the command "mvn package"
- Then we can see the '.war' file location in
- Once we execute these commands we can build the application inside the container.
- So, can we container building technique as discussed above completely to build a docker image?
    - While i run my application i might not build the build system capabilities
    - Docker Image should be slim, so we need an approach where we build packages (application binaries) in one container and copy them into a lightweight image to create docker application images.

Note:- The above can be achieved by Docker Multi Staged Builds

EX:-1

```
FROM maven:3-openjdk-8 AS packaging
RUN git clone https://github.com/wakaleo/game-of-life.git
RUN cd /game-of-life/ && mvn package

FROM tomcat:9.0.39-jdk8-openjdk-slim
COPY --from=packaging /game-of-life/gameoflife-web/target/gameoflife.war /usr/local/tomcat/webapps/gameoflife.war
EXPOSE 8080
```

- If we observe the above Dockerfile if the first step we use 'AS' we have given the firs build as one name and we use that in the second stage.
- If we can create an image using "docker image build -t gameoflife:1.0 ."
- And run the container

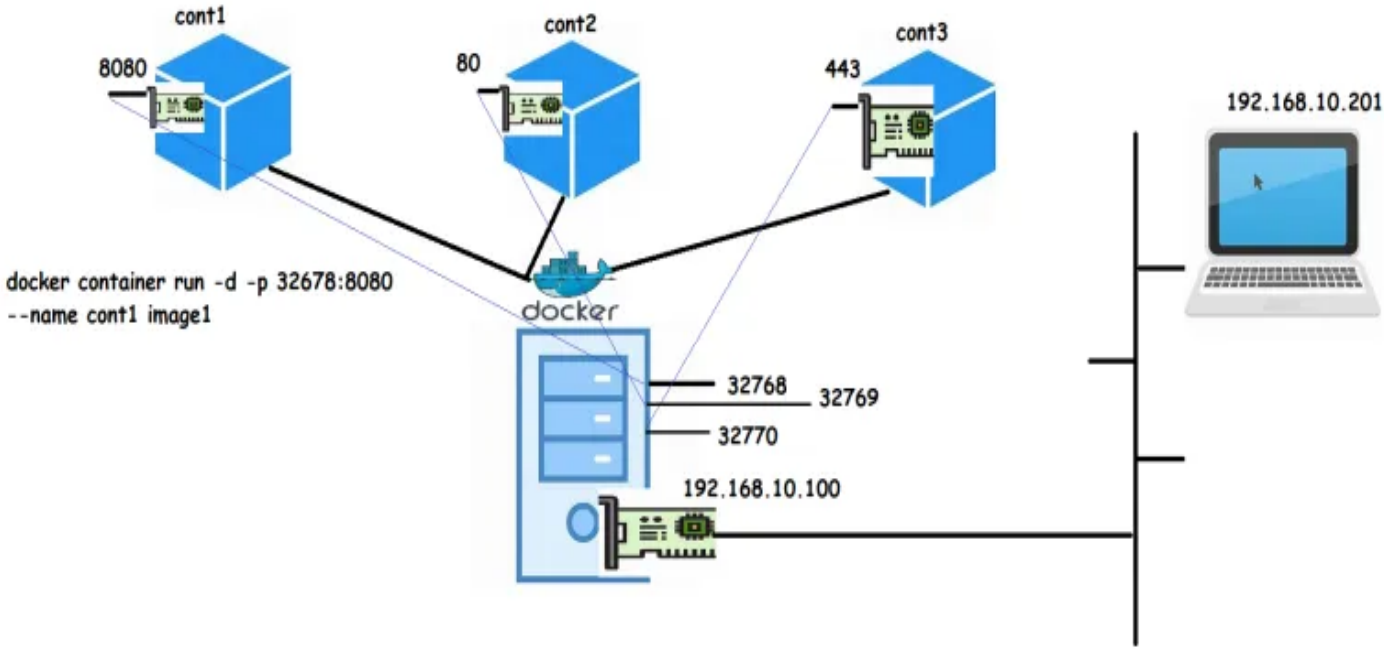
EX:-2

```
FROM maven:3-openjdk-8 AS builder
RUN git clone https://github.com/spring-projects/spring-petclinic.git
RUN cd /spring-petclinic/ && mvn package

FROM  openjdk:8u272-slim
```

COPY --from=builder /spring-petclinic/target/spring-petclinic-2.3.0.BUILD-SNAPSHOT.jar /spring-petclinic.jar
EXPOSE 8080
CMD ["java", "-jar", "/spring-petclinic.jar"]

Topic:- Science behind Docker containerisation

- Docker container is a process isolation.
- It gets cpu and memory
- It gets storage (mount of image layers+read write layer)
- It gets network interface
- Network port mapping b/w docker host and containers is as below pic

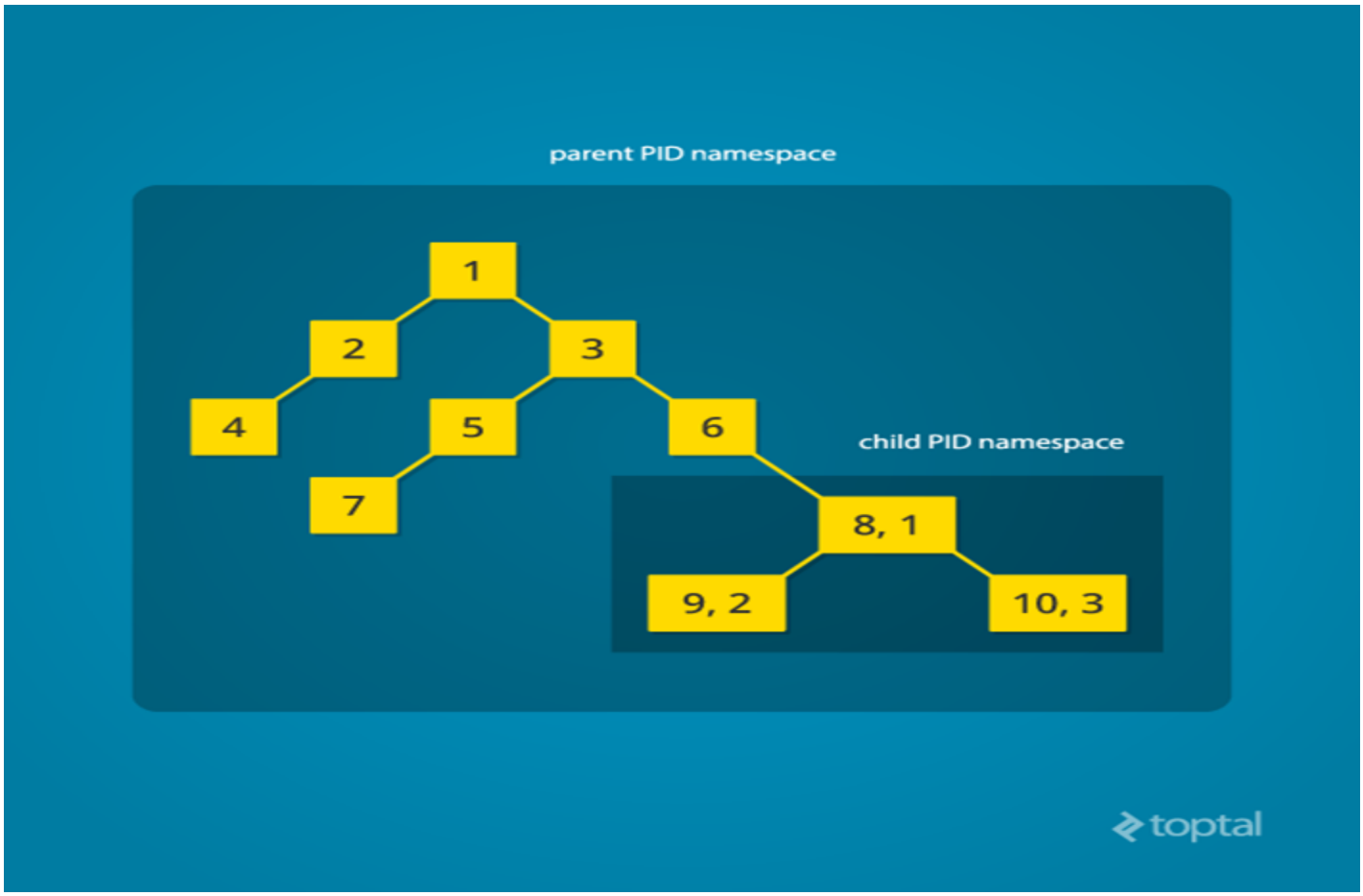


TOPIC:- Docker internals (How the container technology is working)

https://directdevops.blog/2019/01/31/docker-internals/

- Name space is a Linux concept which helps to create Isolation(container) an kernel component of Linux.
- Cgroups helps putting restrictions

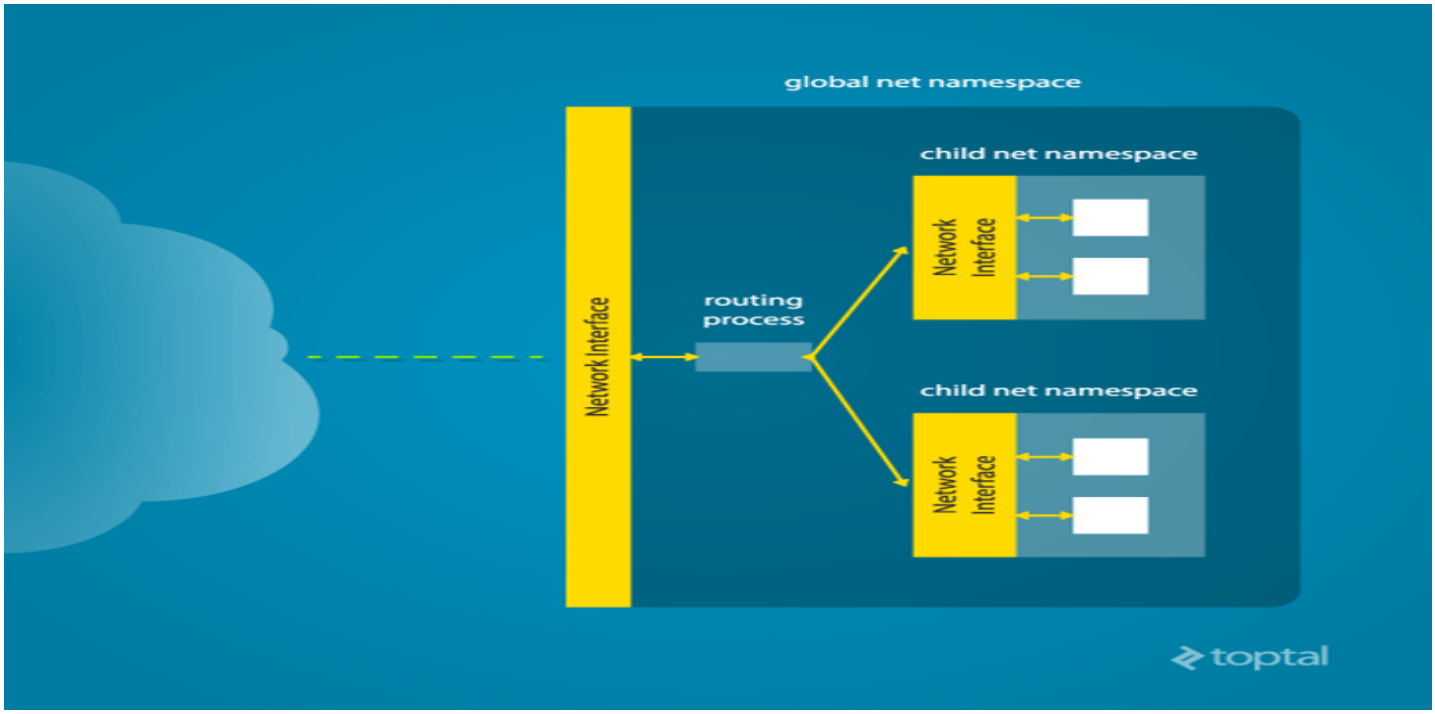
PIDnamespace:-



- PIDname space gives a virtual process tree to ur container to believe it has its own process
- But for your host operating system, your container is another type of process

Network Namespace **:-**

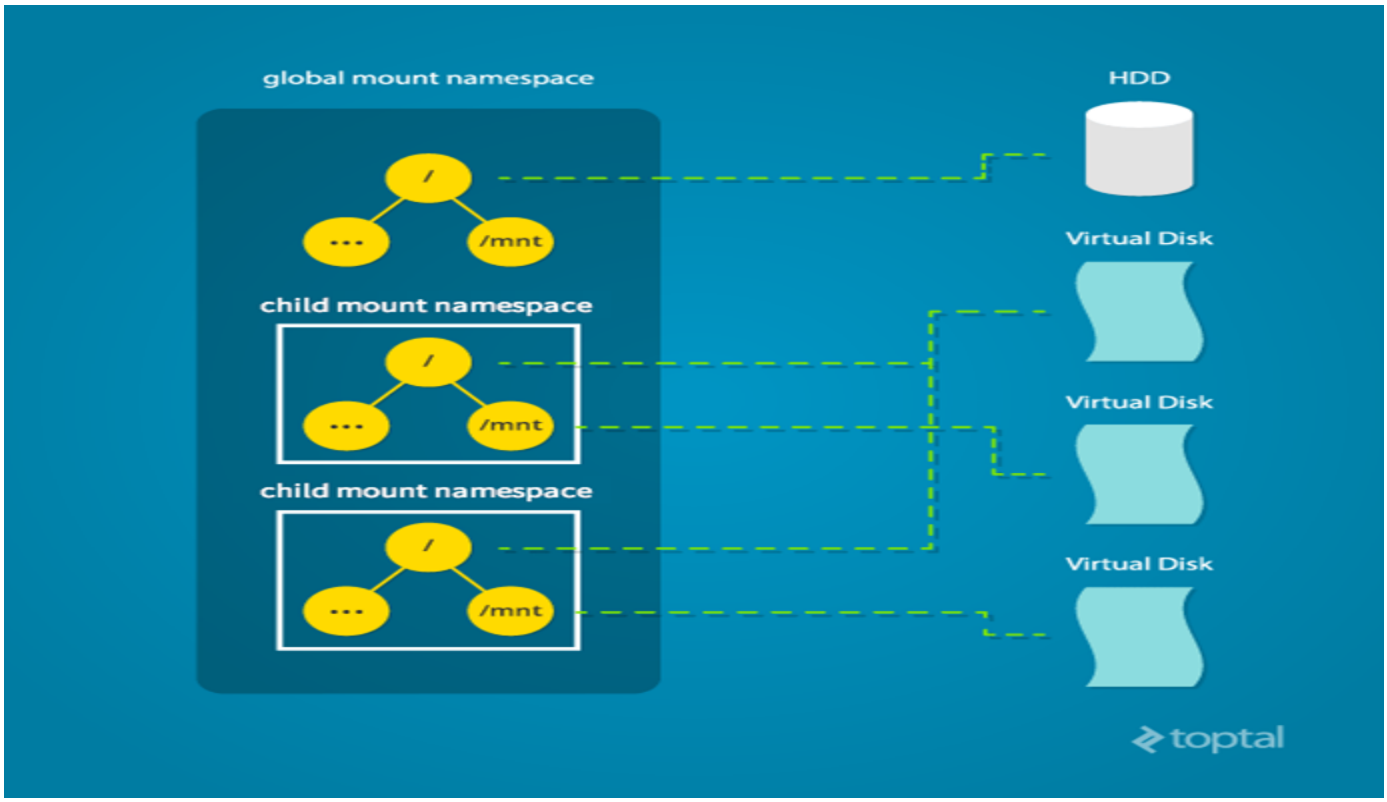
- **net** namespace (Network Namespace) creates the isolated networking for each container with its own network interface.
- So to my container I get process by PID Namespace and net-work by network id namespace



https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces

mount namespace:-

- **mount** namespace creation allows each container to have a different view of entire systems mount point, this allows containers to have their own file system view which starts from root

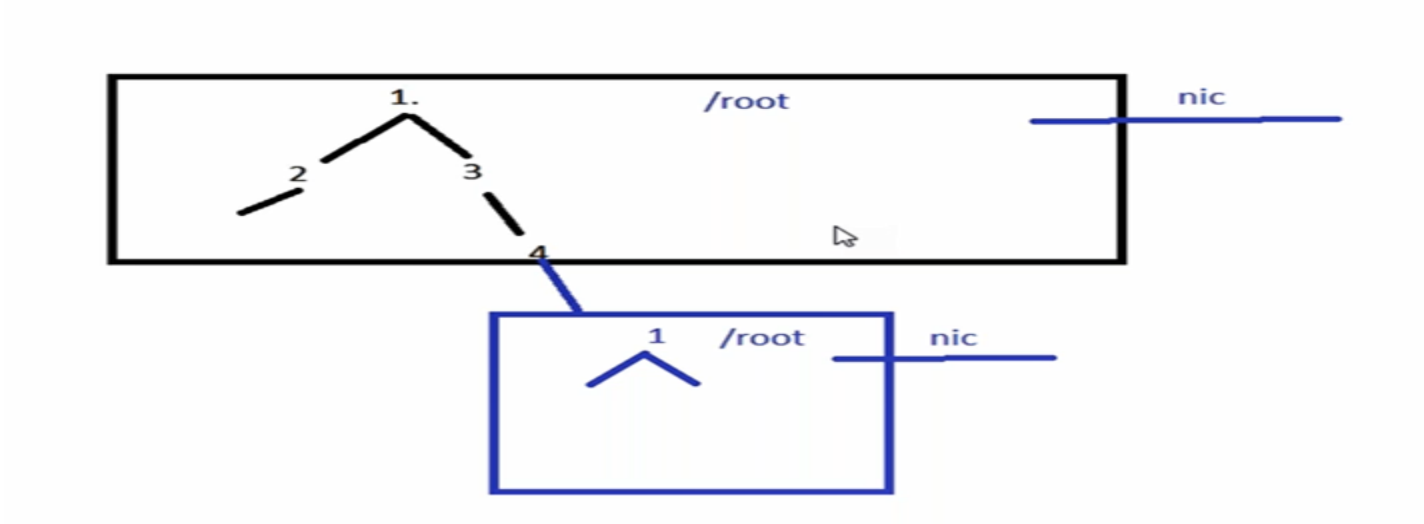


user namespace:-

- **user** namespace allows to create whole new set of user & groups for the containers

· Fortunately even in the Windows world we have namespaces now. The purpose of the namespace is the same but underlying implementation differs. Refer this article

https://docs.microsoft.com/en-us/archive/msdn-magazine/2017/april/containers-bringing-docker-to-windows-developers-with-windows-server-containers

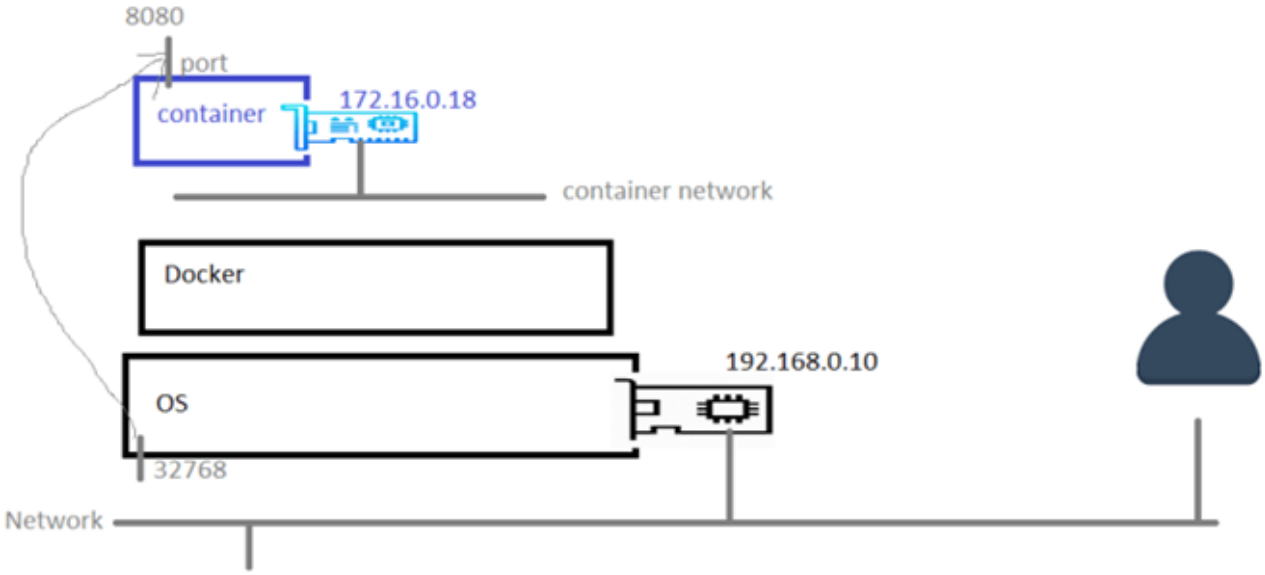


cgroups (control groups)

- Used to allocate CPU, RAM, Disk speed to our containers
- cgroups is a Linux kernel feature
- Control groups are used to impose limits. We can impose limits of disk io, RAM & cpu using Control Groups
- Fortunately even in the Windows world we have control groups now. The purpose of the namespace is the same but underlying implementation differs. Refer this article

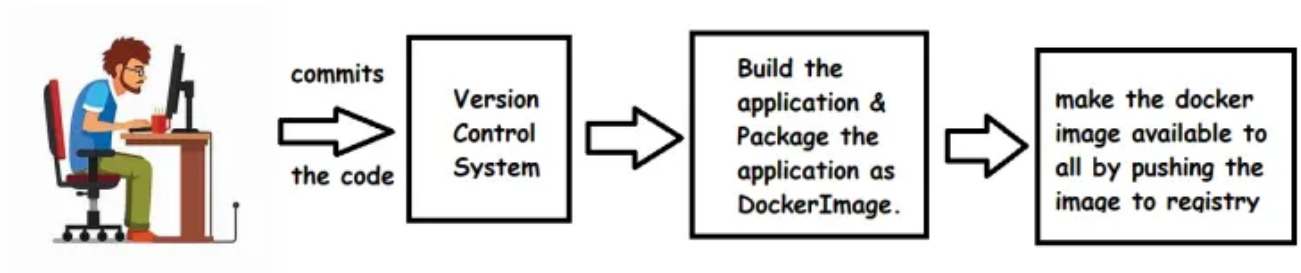
TOPIC:- PORT FORWARDING

To access the app on the container we use port forwarding



Topic:- Storing and distributing docker Images

- Using ACR (Azure Container Registry) to Store Docker images
- Using ECR (Elastic Container Registry) to store docker images



- Now Lets realize this workflow for gameoflife

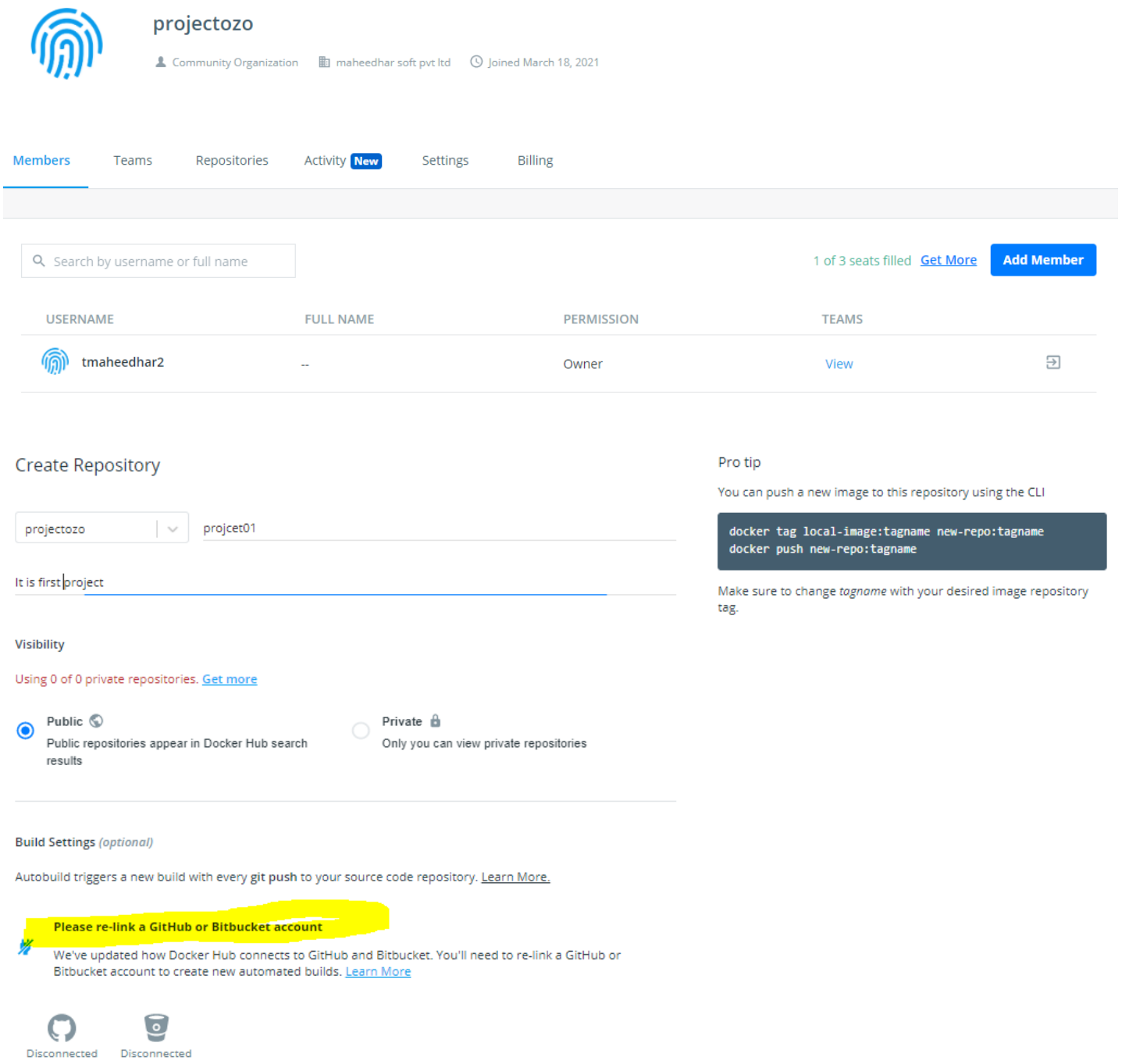
- Whenever a new version of commited a new Docker image can be generated. Dockerfile is created generally in the source-code archive.

```
FROM maven:3-openjdk-8 AS packaging
RUN git clone https://github.com/QT-DevOps/game-of-life.git
RUN cd /game-of-life/ && mvn package

FROM tomcat:9.0.39-jdk8-openjdk-slim
LABEL author="khaja"
LABEL org="learningthoughts"
COPY --from=packaging /game-of-life/gameoflife-web/target/gameoflife.war
/usr/local/tomcat/webapps/gameoflife.war
EXPOSE 8080
```

- Whenever the code is committed we can generate a Docker Image with a new tag. This tag is build locally on some machine, but it should be made available to others (In Enterprise means other members, In public means anyone)
- Let's get Started with using a Default Registry called as *DockerHub*
- Let's build the docker image {docker image build -t gameoflife:1.0 }
- Lets create a tag that matches repository name
- {docker image tag gameoflife:1.0 tmaheedhar2/gameoflife:1.0}

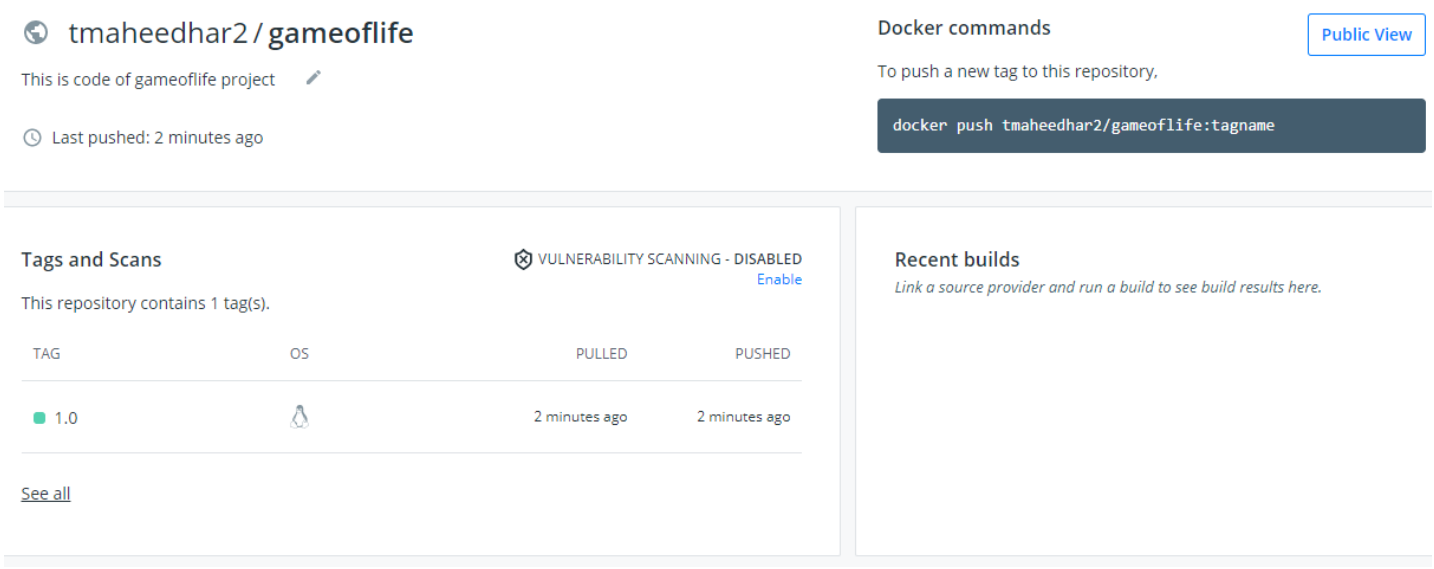
- To push our imge to docker repository we nee to create a repository on docker hub for that login to docker hub → hear we have organisations we can create a separate organisations for separate project. Hear I have put my default organisation ‘tmaheedhar2’ → and create a repository i have given my repository name as ‘gameoflife’ → we can see multiple options in repository creation we can add description, we can assign out git/bitbucket url to docker hub, so that when ever we push the code to git it will automatically create an image and sent to docker hub
- The Important point is we have to give a proper tag to our image that matches out repository neme.
- Then log in to repository via command line and give docker hub credentials {docker login }

- Once login has succeeded then push the image to repository {docker push tmaheedhar2/gameoflife:1.0 }



```
[node1] (local) root@192.168.0.8 /project
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to ht
tps://hub.docker.com to create one.
Username: tmaheedhar2
Password:
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
[node1] (local) root@192.168.0.8 /project
```

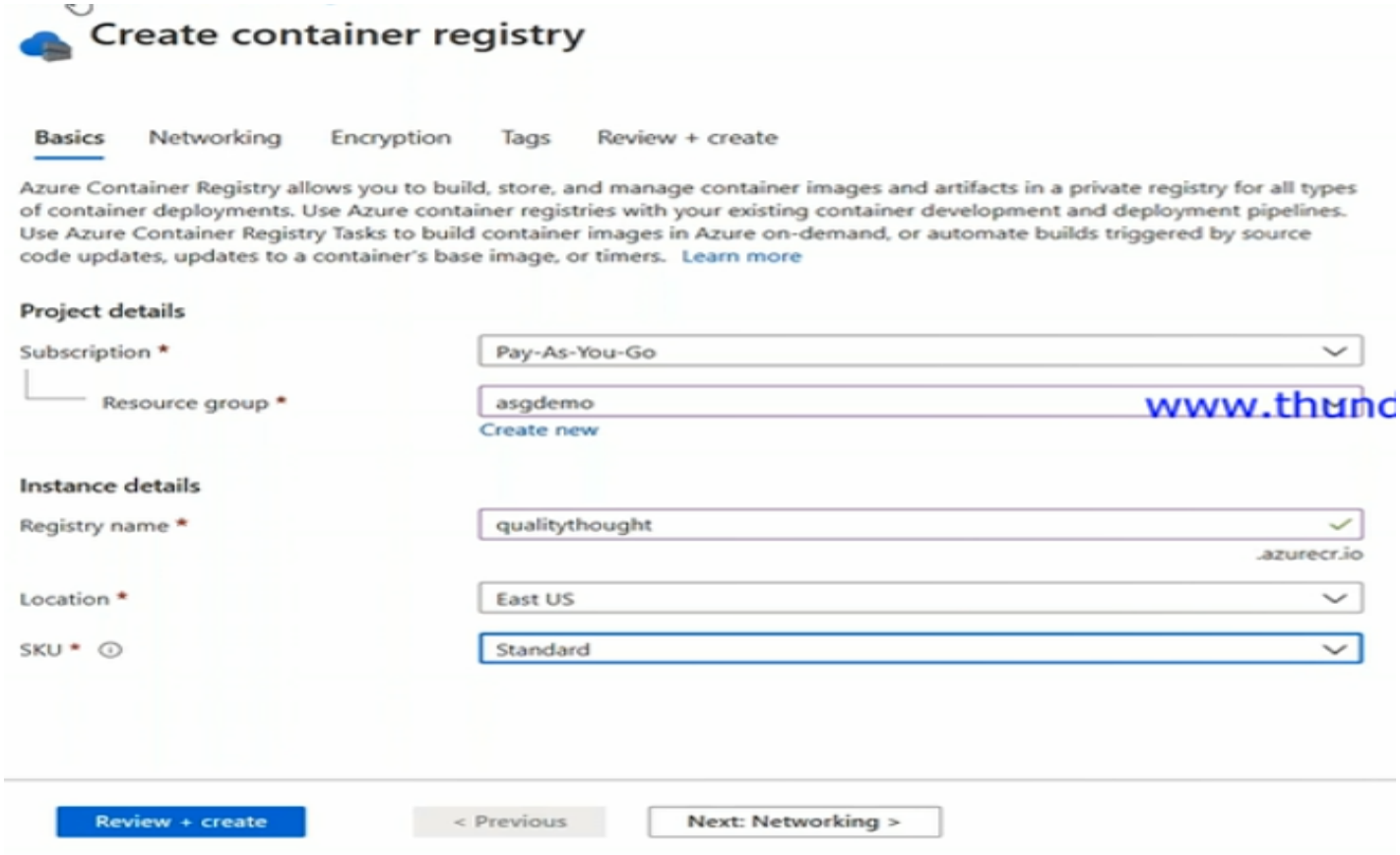
```
[node1] (local) root@192.168.0.8 /project
$ docker push tmaheedhar2/gameoflife:1.0
The push refers to repository [docker.io/tmaheedhar2/gameoflife]
6d51311545c2: Pushed
f25747963018: Mounted from library/tomcat
609892c23047: Mounted from library/tomcat
069c707a9244: Mounted from library/tomcat
a468fa4154af: Mounted from library/tomcat
0c45279c38b9: Mounted from library/tomcat
fee20f1b745d: Mounted from library/tomcat
d0fe97fa8b8c: Mounted from library/tomcat
1.0: digest: sha256:07fe9f0959f9a5a592f8dae51d968b22a69b5a759cdb5821dcdf4078c4bce47a size: 1998
[node1] (local) root@192.168.0.8 /project
```

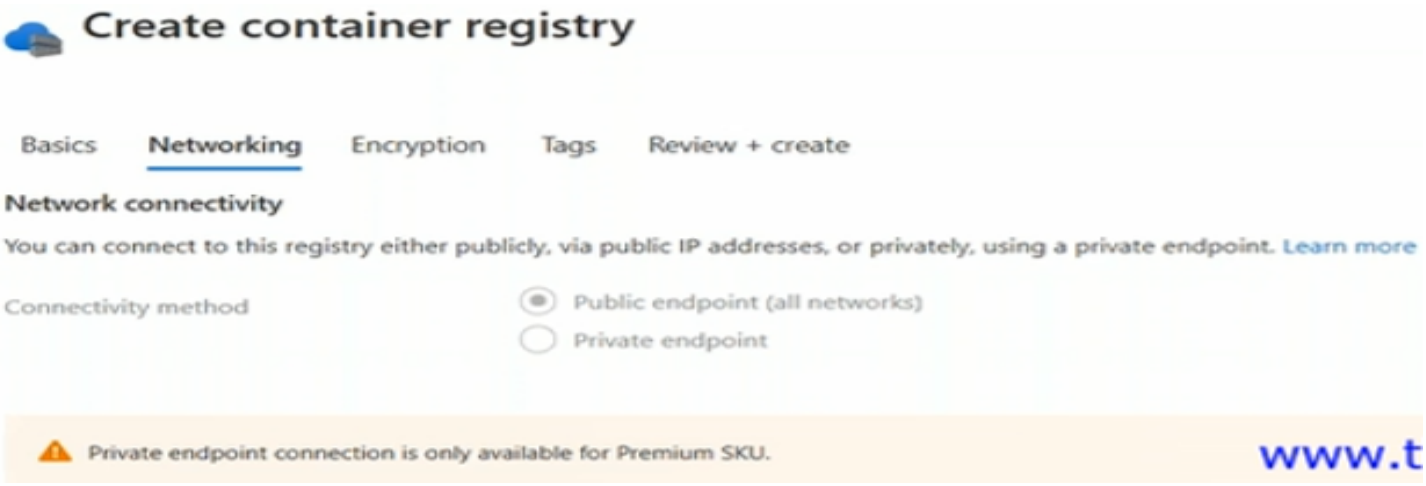


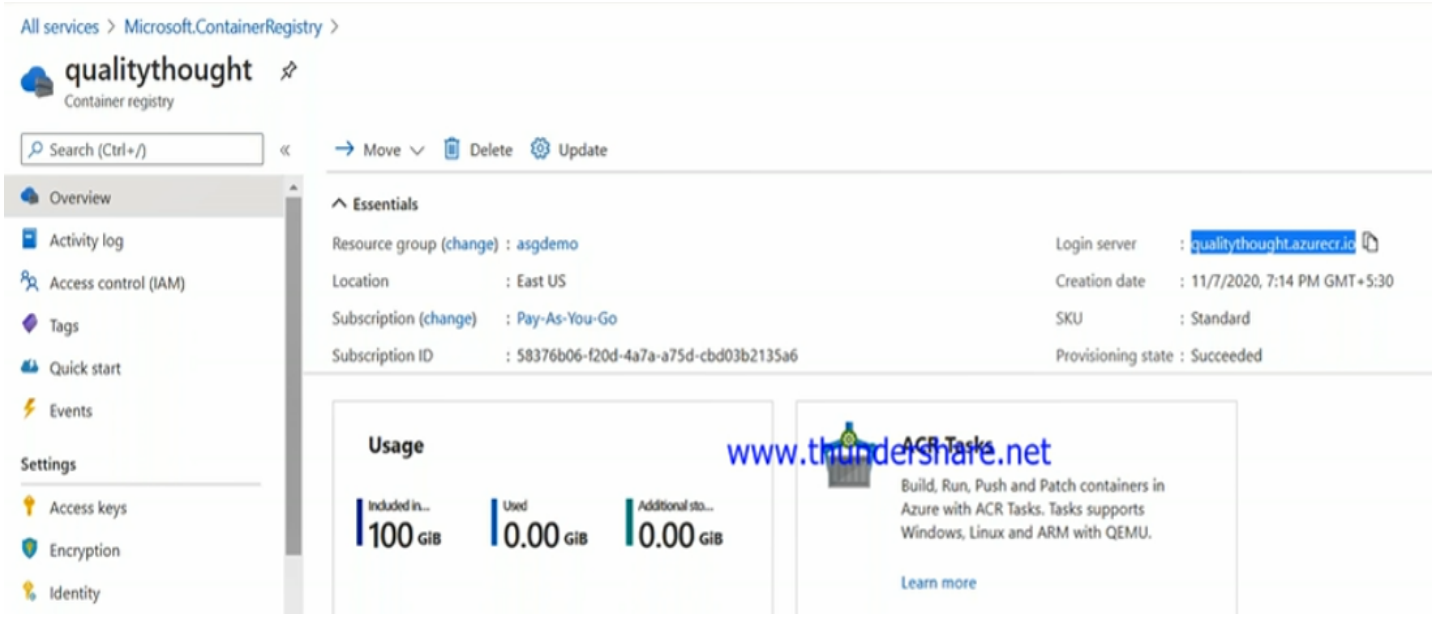
Using ACR (Azure container repository) :-

- Now lets use azure container registry. Refer Here for creating a docker registry in azure cloud
- To login to Azure registry we need to install Azure CLI on our instance{curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash}
- Once Azure CLI is installed we need to do login {az login}
- Then login into azure container registry {az acr login --name nameofouracr}

- Change the image tag to azure registry tag {docker image tag gameoflife:1.0 acrname/repositoryname:version}
- Then push the image to Azure container repository {docker push gameoflife:1.0 acrname/repositoryname:version}
-



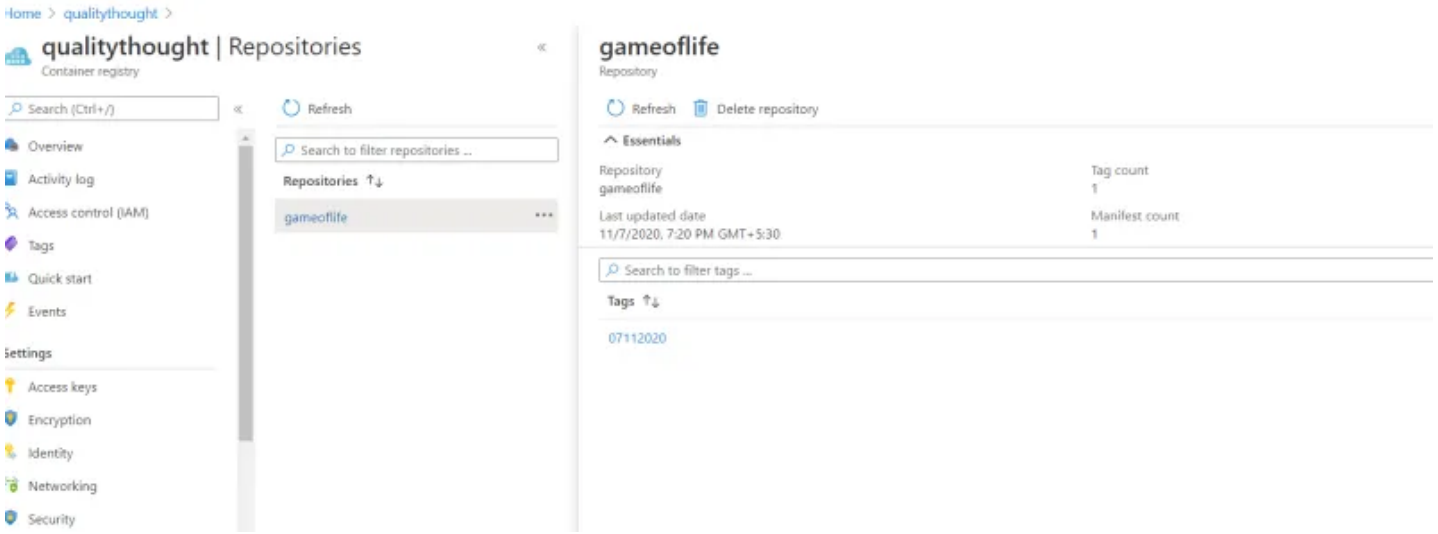





```
qtdevops@qtweb-1:~$ az login
To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code ARPSVSMUG to auth
enticate.
```

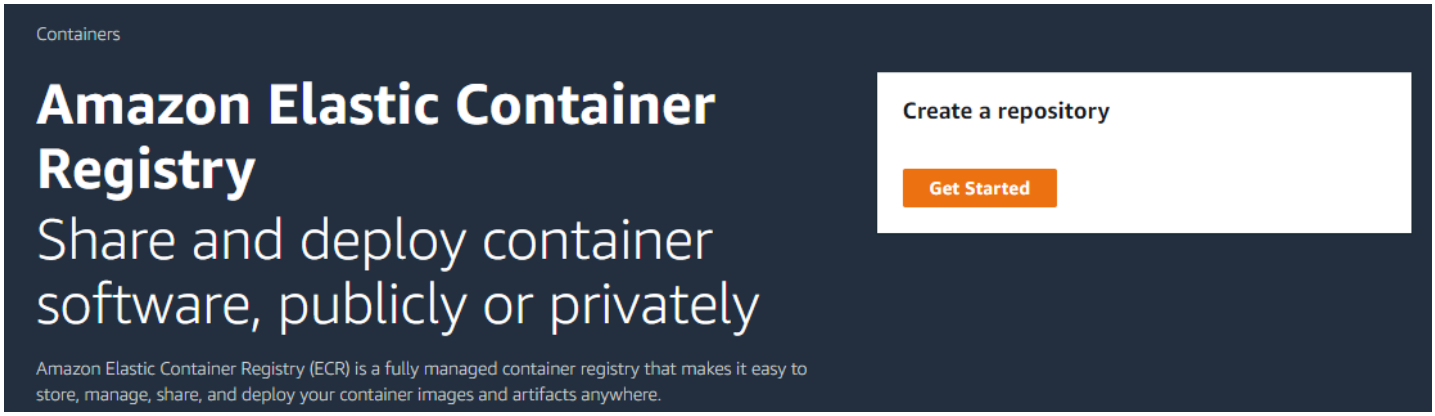
```
qtdevops@qtweb-1:~$ az acr login --name qualitythought.azurecr.io
The login server endpoint suffix '.azurecr.io' is automatically omitted.
Login Succeeded
qtdevops@qtweb-1:~$
```

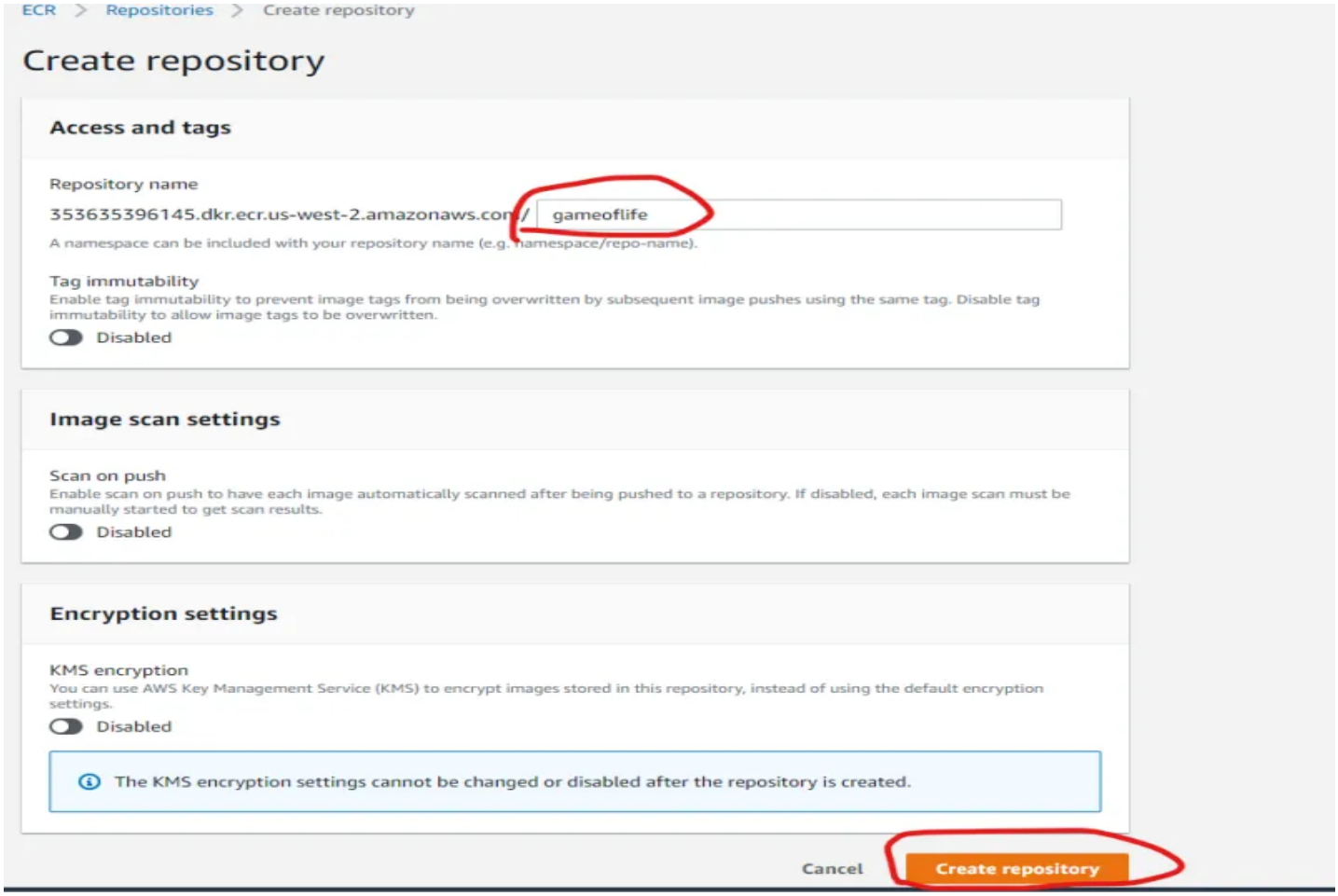
```
qtdevops@qtweb-1:~$ az acr login --name qualitythought.azurecr.io
The login server endpoint suffix '.azurecr.io' is automatically omitted.
Login Succeeded
qtdevops@qtweb-1:~$ docker image tag gameoflife:07112020 qualitythought.azurecr.io/gameoflife:07112020
qtdevops@qtweb-1:~$ docker image push qualitythought.azurecr.io/gameoflife:07112020
The push refers to repository [qualitythought.azurecr.io/gameoflife]
a31e600bedd7: Pushed
d58bacc8f19c: Pushed
329d4f2c486f: Pushed
ec9377a62a47: Pushed
71de13106f17: Pushed
0c45279c38b9: Pushed
fee20f1b745d: Pushed
d0fe97fa8b8c: Pushed
07112020: digest: sha256:d1e9605cd63903c77ab9cb44a08cb3f5c8da6ec88bbabec2fde4fc0b70b419b7 size: 1998
qtdevops@qtweb-1:~$
```

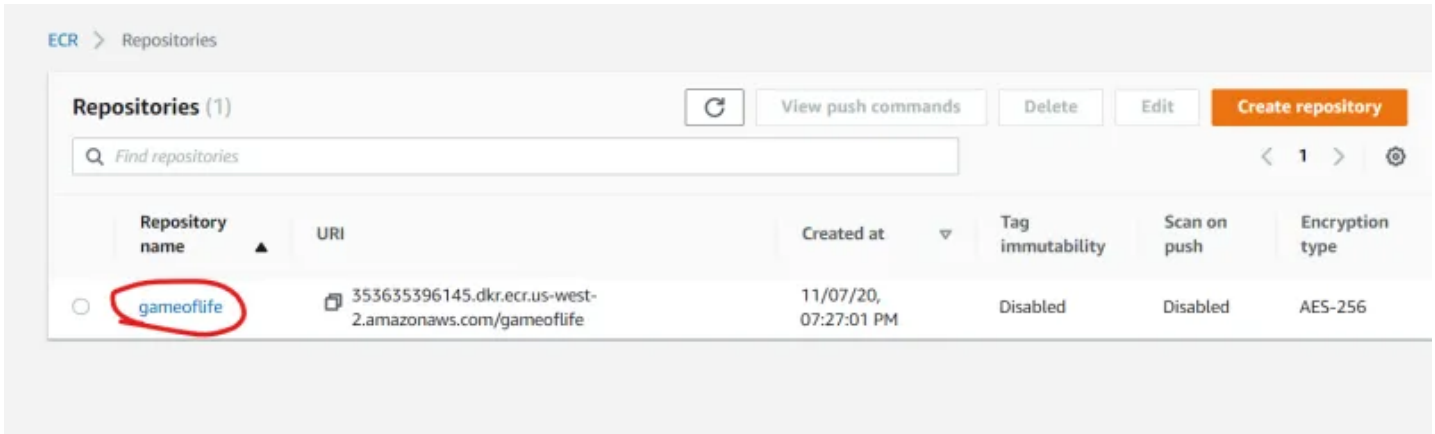


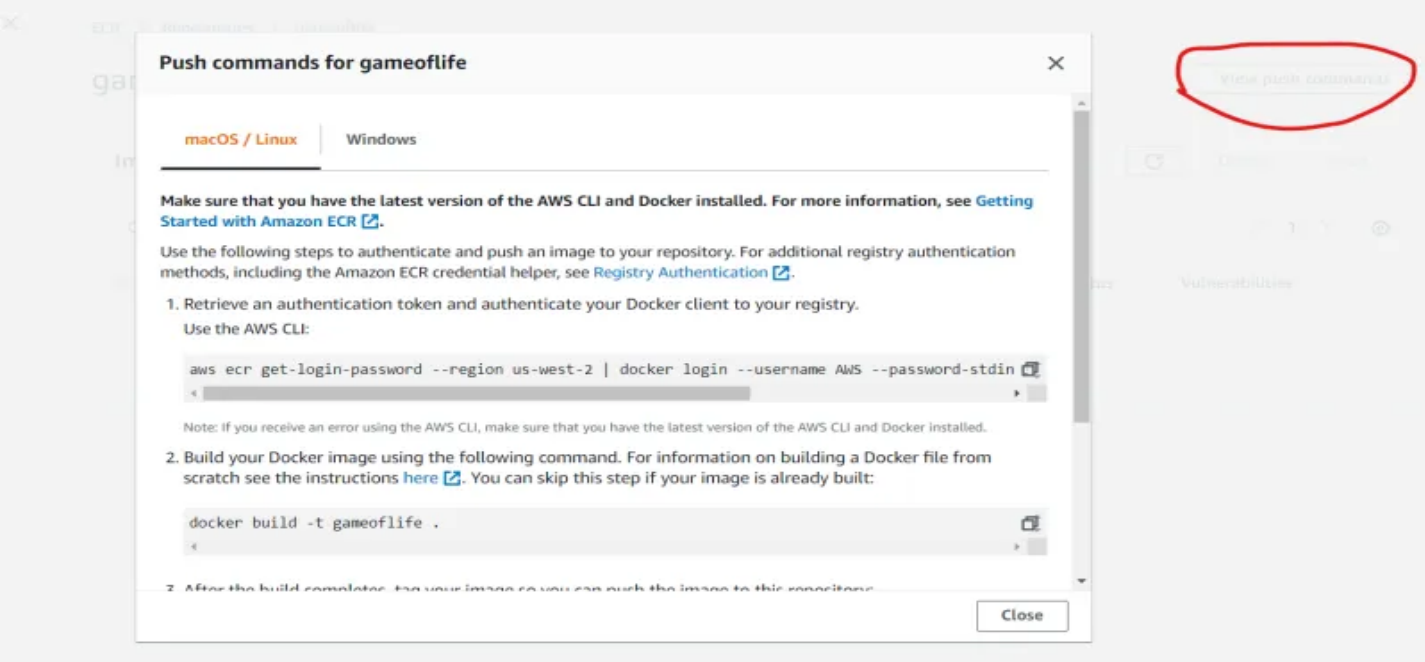
AWS Elastic Container Registry

- Lets create a ECR in aws. Goto ecr service → get started → select repository for public or private → give repository name on (access and tags) → and create repository
- Once repository is created and then open the repository then we will notice ‘view push commands’
- We need to install AWS CLI to our vm {apt-get install awscli && aws -version }
- And if we can see view push commands we can notice authentication {aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 296760598094.dkr.ecr.us-east-1.amazonaws.com}
- If we are not authentication was not success then we need to configure “aws configure”
- We have to build image with keys name {docker tag gameoflife:1.0 296760598094.dkr.ecr.us-east-1.amazonaws.com/gameoflife:0.1}
- Then we need to push the image {docker push 296760598094.dkr.ecr.us-east-1.amazonaws.com/gameoflife:1.0}


Containers
Amazon Elastic Container Registry
Share and deploy container software, publicly or privately
Amazon Elastic Container Registry (ECR) is a fully managed container registry that makes it easy to store, manage, share, and deploy your container images and artifacts anywhere.
Create a repository
Get Started


ECR > Repositories > Create repository
Create repository
Access and tags
Repository name
353635396145.dkr.ecr.us-west-2.amazonaws.com/
gameoflife
A namespace can be included with your repository name (e.g. namespace/repo-name).
Tag immutability
Enable tag immutability to prevent image tags from being overwritten by subsequent image pushes using the same tag. Disable tag immutability to allow image tags to be overwritten.
Disabled
Image scan settings
Scan on push
Enable scan on push to have each image automatically scanned after being pushed to a repository. If disabled, each image scan must be manually started to get scan results.
Disabled
Encryption settings
KMS encryption
You can use AWS Key Management Service (KMS) to encrypt images stored in this repository, instead of using the default encryption settings.
Disabled
The KMS encryption settings cannot be changed or disabled after the repository is created.
Cancel
Create repository

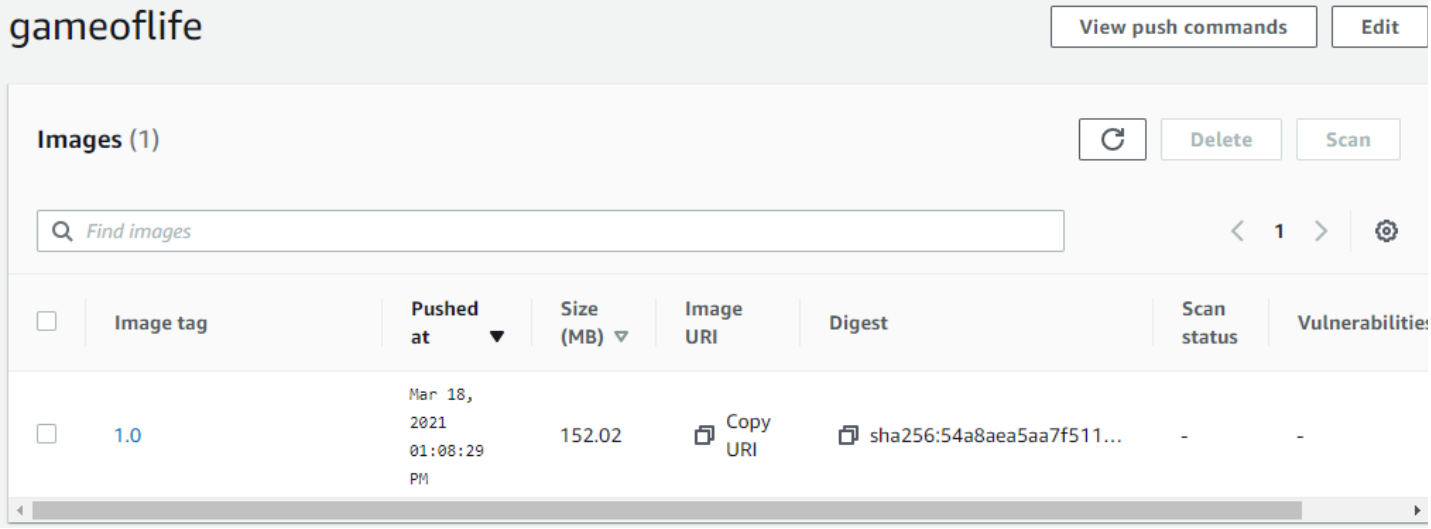

ECR > Repositories
Repositories (1)
View push commands
Delete
Edit
Create repository
Find repositories
1
Repository name
URI
Created at
Tag immutability
Scan on push
Encryption type
gameoflife
353635396145.dkr.ecr.us-west-2.amazonaws.com/gameoflife
11/07/20, 07:27:01 PM
Disabled
Disabled
AES-256




```
root@ip-172-31-57-233:/ecr# aws ecr get-login-password --region us-east-1 | docker login --username AWS --passwo
rd-stdin 296760598094.dkr.ecr.us-east-1.amazonaws.com
Unable to locate credentials. You can configure credentials by running "aws configure".
Error: Cannot perform an interactive login from a non TTY device
root@ip-172-31-57-233:/ecr# aws configure
AWS Access Key ID [None]: AKIAUKGCPLZHGBFPH7QI
AWS Secret Access Key [None]: ^[[A^[[B^C
root@ip-172-31-57-233:/ecr# aws configure
AWS Access Key ID [None]: AKIAUKGCPLZHJDABFU4O
AWS Secret Access Key [None]: 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
Default region name [None]: us-east-1
Default output format [None]:
root@ip-172-31-57-233:/ecr# aws ecr get-login-password --region us-east-1 | docker login --username AWS --passwo
rd-stdin 296760598094.dkr.ecr.us-east-1.amazonaws.com
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

```
root@ip-172-31-57-233:/ecr# docker tag gameoflife:1.0 296760598094.dkr.ecr.us-east-1.amazonaws.com/gameoflife:0.
1
root@ip-172-31-57-233:/ecr# docker image ls
REPOSITORY                                                TAG                       IMAGE ID       CREATED
     SIZE
296760598094.dkr.ecr.us-east-1.amazonaws.com/gameoflife   0.1                       ef044676ffbc   16 minutes a
go   311MB
gameoflife                                                1.0                       ef044676ffbc   16 minutes a
go   311MB
<none>                                                    <none>                    73dfd721aedb   17 minutes a
go   646MB
maven                                                     3-openjdk-8               9a3c0f7f6af8   4 days ago
     525MB
tomcat                                                    9.0.39-jdk8-openjdk-slim  18c86ad30e02   4 months ago
     308MB
```

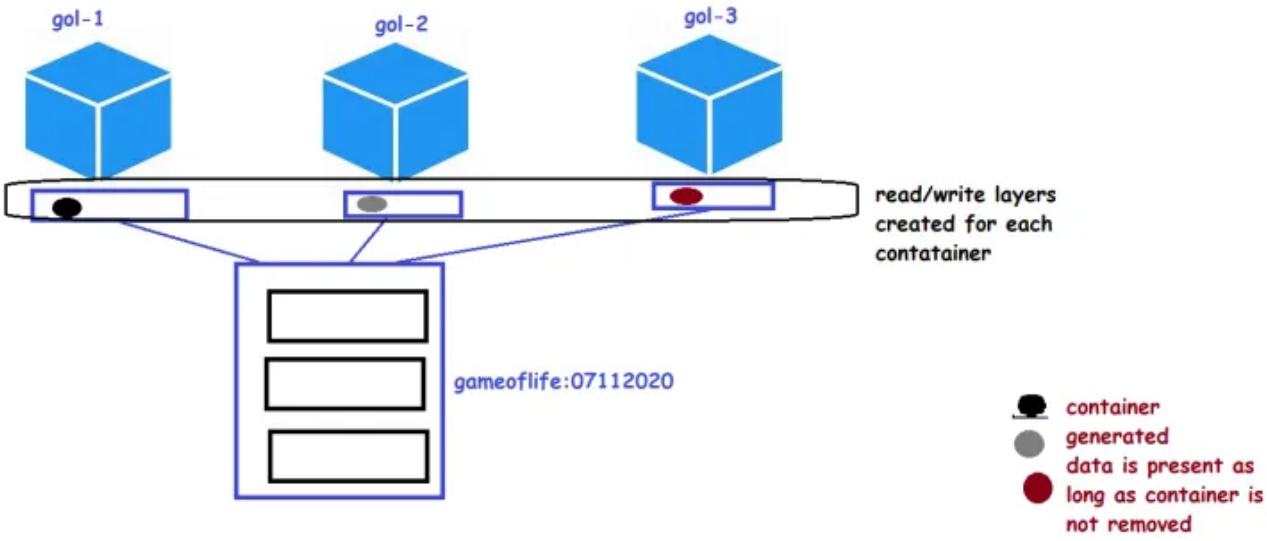
```
root@ip-172-31-57-233:/ecr# docker push 296760598094.dkr.ecr.us-east-1.amazonaws.com/gameoflife:1.0
The push refers to repository [296760598094.dkr.ecr.us-east-1.amazonaws.com/gameoflife]
52a02c8c368f: Pushed
f25747963018: Pushed
609892c23047: Pushed
069c707a9244: Pushed
a468fa4154af: Pushed
0c45279c38b9: Pushed
fee20f1b745d: Pushed
d0fe97fa8b8c: Pushed
1.0: digest: sha256:54a8aea5aa7f511a56e91b38b72bf9a64c6faee9e0d75543a17e2ba440b29a2d size: 1998
root@ip-172-31-57-233:/ecr#
```



- Docker Trusted Registry:
    - This registry is part of Docker Enterprise
    - This registry integrates with LDAP
- Artifactory (Jfrog) Docker Registry
- Docker Registry: This docker image can be used as registry [Refer Here]

Topic:- Docker Volumes

- We can create multiple containers by using one docker image.
- We know every docker container get some layers those are called image layers
- Every Container gets a read write layer extra on top of image layers
- While container is running it will generate some data
- Whatever the data that is generated is stored in R/W layers
- Storage Drivers allows is to create data in the writable layer of the container
- We are already of Image Layers concept. For every RUN,ADD/COPy in the Dockerfile new layers are created.
- When we create a container, a Thin R/w Layer is available for each container to store the data in the container.
- As long as this container is alive this R/W Layer is available once the Once the container dead this layer also deleted
-
-



- Out of all this we have two concerns like i) How docker container mount image layer + R/W layer as one file system (ans was “storage drivers”)&& ii) How to preserve the data which is in R/W layer (ans was “Docker Volume”)

Activity:-

- Let’s Experiment This with the following Dockerfile

```
FROM alpine
RUN mkdir /mycontent && touch /mycontent/1.txt
```

- Build the docker image and give name as layerdemo
- we would be using interactive mode to understand the behavior
- Alpine image has one layer and layerdemo will be adding a new layer. And every container created using layerdemo will add a new Thin R/W layer

```
docker container run --name one -it layerdemo /bin/sh
```

NOTE:- This container is dealing with 3 layers now but inside the container it will look as if it is one file system

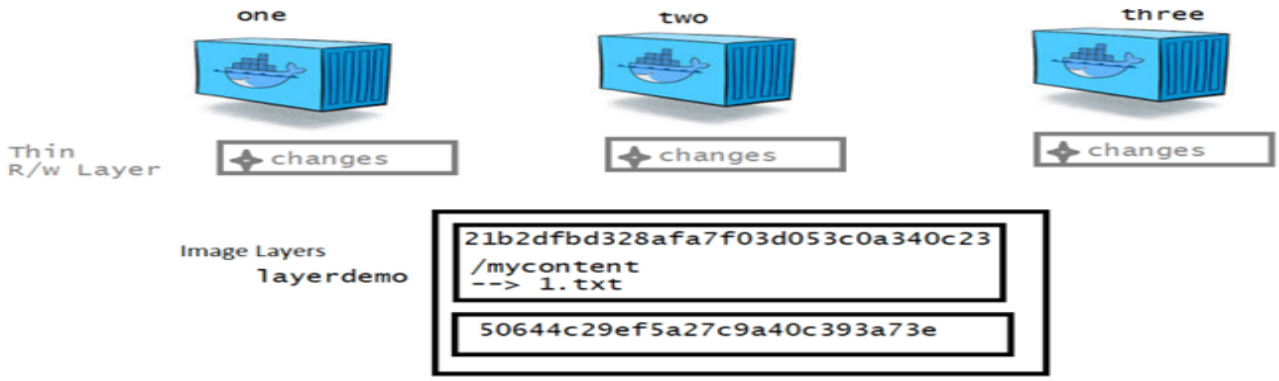
```
ubuntu@ip-172-31-10-209:~/layerdemo$ docker container run --name one -it lay
mo /bin/sh
/ # df -h
Filesystem                Size      Used Available Use% Mounted on
overlay                   7.7G      1.7G      6.0G  22% /
tmpfs                    64.0M         0     64.0M   0% /dev
tmpfs                   491.8M         0    491.8M   0% /sys/fs/cgroup
shm                      64.0M         0     64.0M   0% /dev/shm
/dev/xvda1                7.7G      1.7G      6.0G  22% /etc/resolv.conf
/dev/xvda1                7.7G      1.7G      6.0G  22% /etc/hostname
/dev/xvda1                7.7G      1.7G      6.0G  22% /etc/hosts
tmpfs                   491.8M         0    491.8M   0% /proc/acpi
tmpfs                    64.0M         0     64.0M   0% /proc/kcore
tmpfs                    64.0M         0     64.0M   0% /proc/keys
tmpfs                    64.0M         0     64.0M   0% /proc/timer_list
tmpfs                    64.0M         0     64.0M   0% /proc/sched_debug
tmpfs                   491.8M         0    491.8M   0% /proc/scsi
tmpfs                   491.8M         0    491.8M   0% /sys/firmware
/ #
```

- Internally the layers are combined and presented as one disk to container using storage drivers.
- Now lets try to make some changes

```
# Inside docker container one
cat /mycontent/1.txt
echo "hello" >> /mycontent/1.txt
```

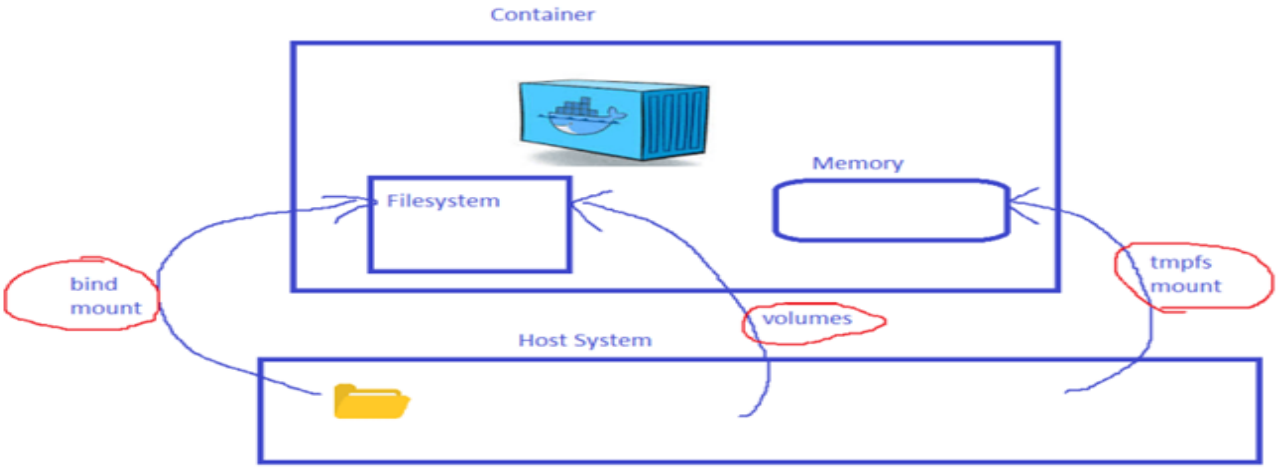
```
/ # cat /mycontent/1.txt
/ # echo "hello" >> /mycontent/1.txt
/ # cat /mycontent/1.txt
hello
/ #
```

Since the layers are read-only whenever you modify any existing content, the file gets copied to read/layer and changes are recorded in the R/W. This approach is called as “copy-on-write”

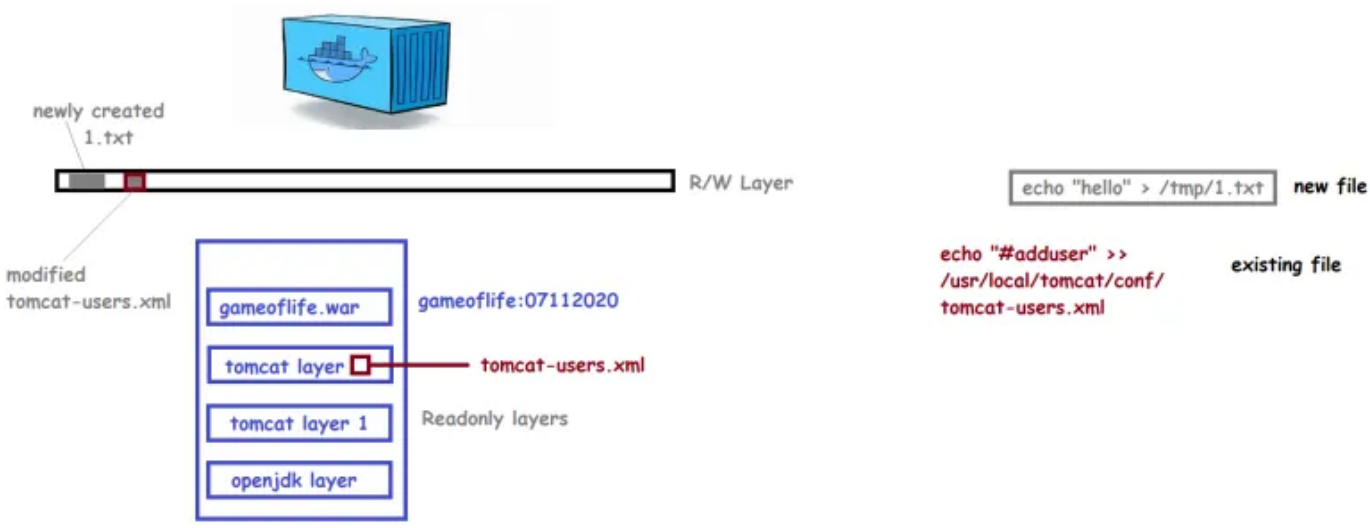


How to preserve the data changed by the docker container

- Docker will store in os “/var/lib/docker”
- If we go to “/var/lib/docker/overlay2” it is the location docker will store all image layers
- Persistence/Preserving the data even after the container is deleted can be achieved by storing the data outside the container. To do this we have 3 options
- Volumes; bind-mounts; tmpfs



Storage Drivers :-



- Copy-on-write strategy
- As we discussed above every docker container has image layers along with R/W layer
- All the changes are copied to the R/W layer for ex. If we modify any files in existing image layers a copy of that will come to the R/W layer. And if we create a file then also stored in an R/W layer. Even if we delete the file also it will maintain a copy
- For this We need a file system which understands image layers and R/W Layers and to the container it should be file system much like any other linux/Windows file system

- Storage Driver understands read only image layers, Read/Write Thin layer of container and combines (union) the layers to give one filesystem to container

Docker has the following storage drivers{to view our storage driver "docker info"}

- **overlay2**: This is preferred storage driver for all linux distributions
- **aufs**: This was preferred storage driver for Docker 18.06 and previous versions
- **devicemapper**
- **btrfs** and **zfs**

https://directdevops.blog/2019/09/27/impact-of-image-layers-on-docker-containers-storage-drivers/

- The above article will tell about "Impact of image layers and storage drivers on docker filesystem"
- If we can see the container, root partition the root volume is overlay
- Internally the is drivers will do lot of work to give this kind of output

```
[node1] (local) root@192.168.0.28 ~
$ docker container run -ti tomcat:8 /bin/bash
root@cc6cfae99cd3:/usr/local/tomcat# df -h
Filesystem      Size  Used Avail Use% Mounted on
overlay         9.4G  540M  8.8G   6% /
tmpfs            64M     0   64M   0% /dev
tmpfs            16G     0   16G   0% /sys/fs/cgroup
shm              64M     0   64M   0% /dev/shm
/dev/sdb         64G   23G   42G  35% /etc/hosts
tmpfs            16G     0   16G   0% /proc/asound
tmpfs            16G     0   16G   0% /proc/acpi
tmpfs            16G     0   16G   0% /proc/scsi
tmpfs            16G     0   16G   0% /sys/firmware
root@cc6cfae99cd3:/usr/local/tomcat#
```

Bind mounts

- are stored anywhere on the host system.
- Oldest option available for persisting the changes
- Create a docker container with bind mount option

Syntax:- docker container run –it –name <name> -v <path in host-os>:<path in container> <image name> /bin/sh

"docker container run –v <some folder on host>:<some folder on container> "

```
docker container run -it --name bindmount -v /home/ubuntu/test:/app alpine /bin/sh
df -h
touch /app/1.txt
exit
docker inspect bindmount
docker container rm bindmount
```

```
/ # df -h
Filesystem                Size      Used Available Use% Mounted on
overlay                   7.7G      1.7G      6.0G  22% /
tmpfs                    64.0M         0     64.0M   0% /dev
tmpfs                   491.8M         0    491.8M   0% /sys/fs/cgroup
shm                      64.0M         0     64.0M   0% /dev/shm
/dev/xvda1                7.7G      1.7G      6.0G  22% /app
/dev/xvda1                7.7G      1.7G      6.0G  22% /etc/resolv.conf
/dev/xvda1                7.7G      1.7G      6.0G  22% /etc/hostname
/dev/xvda1                7.7G      1.7G      6.0G  22% /etc/hosts
tmpfs                   491.8M         0    491.8M   0% /proc/acpi
tmpfs                    64.0M         0     64.0M   0% /proc/kcore
tmpfs                    64.0M         0     64.0M   0% /proc/keys
tmpfs                    64.0M         0     64.0M   0% /proc/timer_list
tmpfs                    64.0M         0     64.0M   0% /proc/sched_debug
tmpfs                   491.8M         0    491.8M   0% /proc/scsi
tmpfs                   491.8M         0    491.8M   0% /sys/firmware
/ #
```

```
ubuntu@ip-172-31-10-209:~$ docker container rm bindmount
bindmount
ubuntu@ip-172-31-10-209:~$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
     PORTS               NAMES
bc1117ee967c        layerdemo           "/bin/sh"           30 minutes ago      Exited (0) 22 minutes a
go                      one
ubuntu@ip-172-31-10-209:~$ ls
get-docker.sh  layerdemo  test
ubuntu@ip-172-31-10-209:~$ cd test/
ubuntu@ip-172-31-10-209:~/test$ ls
1.txt
ubuntu@ip-172-31-10-209:~/test$
```

Let's create a new container using the same storage as mount and the same data should be loaded in the new container

```
ubuntu@ip-172-31-10-209:~/test$ docker container run -it --name bindmount2 -v /home/ubuntu/test:/app al
pine /bin/sh
/ # ls /app
1.txt
/ #
```

"docker container run --mount type=bind,source=/home/ubuntu/test,target=/app <imagename>"

"docker container run --mount type=bind,source=/home/ubuntu/test,target=/app,readonly <imagename>"

NOTE:- By using the 'mount' command we will map the folder location. Hear –v and mount command will do the same job .

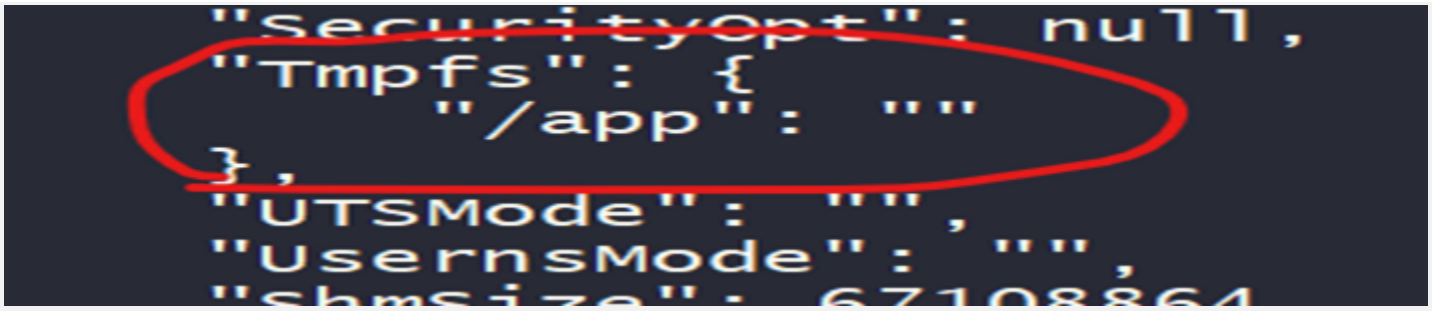
One extra feature with mount command is for ex" docker container run --mount type=bind,source=/home/ubuntu/test,target=/app,**read only** <imagename" we can see readonly option here so we can give read only to folder in container so that docker can't modify files in that location

Note:- In this concept bind-mounts . We need to share a folder which is created by us and which can be used by many processes in systems and they want to modify something which causes a lot of errors. TO avoid that docker has concept called "Volumes"(a mechanism which docker manages)

Tempfs mount:-

- Mounted on to tmpfs in memory
- Tempfs mount can be created using –tmpfs or –mount flag
- Tempfs will be mounted on the RAM of our container

```
docker container run -d --name tmp1 --tmpfs /app alpine sleep 1d
docker container inspect tmp1
docker container run -d --name tmp2 --mount type=tempfs,destination=/app alpine sleep 1d
```

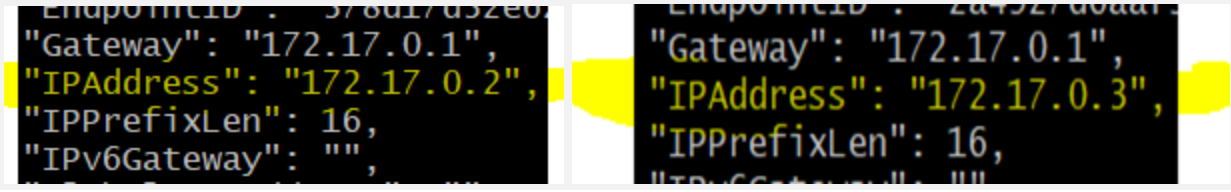


How can one container communicate with another container?

- We will take two containers cont1 & cont2 and every container has IP address lets see

```
docker container run -d --name cont1 alpine sleep 1d
docker container run -d --name cont2 alpine sleep 1d
```

- Every container gets its IP address



- Now let's see if containers have network connectivity b/w then by executing

```
docker container exec cont1 ping cont2
```

```
root@ip-172-31-80-14:/docker/file8# docker container exec cont1 ping cont2
ping: bad address 'cont2'
```

- Now lets check connectivity by pinging ip address

Syntax:- docker container exec <container-1> ping –c <count> <IP address of second container>

docker container exec cont1 ping -c 4 172.17.0.4

```
root@ip-172-31-80-14:/docker/file8# docker container exec cont1 ping -c 5 172.17.0.3
PING 172.17.0.3 (172.17.0.3): 56 data bytes
64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.086 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.090 ms
64 bytes from 172.17.0.3: seq=2 ttl=64 time=0.099 ms
64 bytes from 172.17.0.3: seq=3 ttl=64 time=0.097 ms
64 bytes from 172.17.0.3: seq=4 ttl=64 time=0.114 ms
```

NOTE:- We are able to ping from one container to other by ip addresses but names are not resolving

Docker Volumes:-

- Created and managed by Docker.
- You create docker volumes and use that volume in the container.
- Docker Volumes can be create in the Dockerfile using instruction **VOLUME**
- Refer Here for article on Docker Volumes {https://directdevops.blog/2019/10/03/docker-volumes/}
- By default all the files created by the container are stored on a thin R/w layer. This layer will be existing on a system as long as the container exists. Once the container is deleted this data will be lost
- Docker has two options for storing the files on the host machine, so that the changes done in the container are persisted even after the container is deleted.
    - Volumes
    - Bind Mount
- Create a Docker Image from the file without Volume instruction and with volume instruction Refer Here for changeset

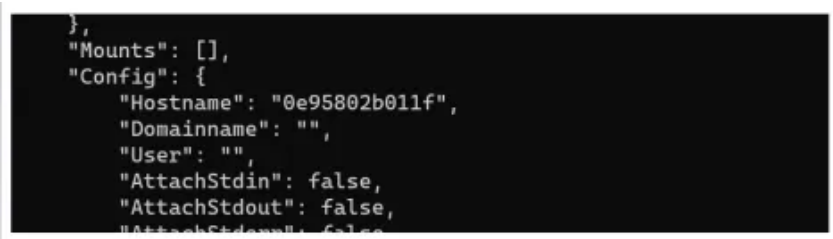
  FROM tomcat:jdk8-openjdk

LABEL team="qtdevops"
RUN cd webapps/ && wget https://referenceappkhaja.s3-us-west-2.amazonaws.com/gameoflife.war
VOLUME /usr/local/tomcat
EXPOSE 8080

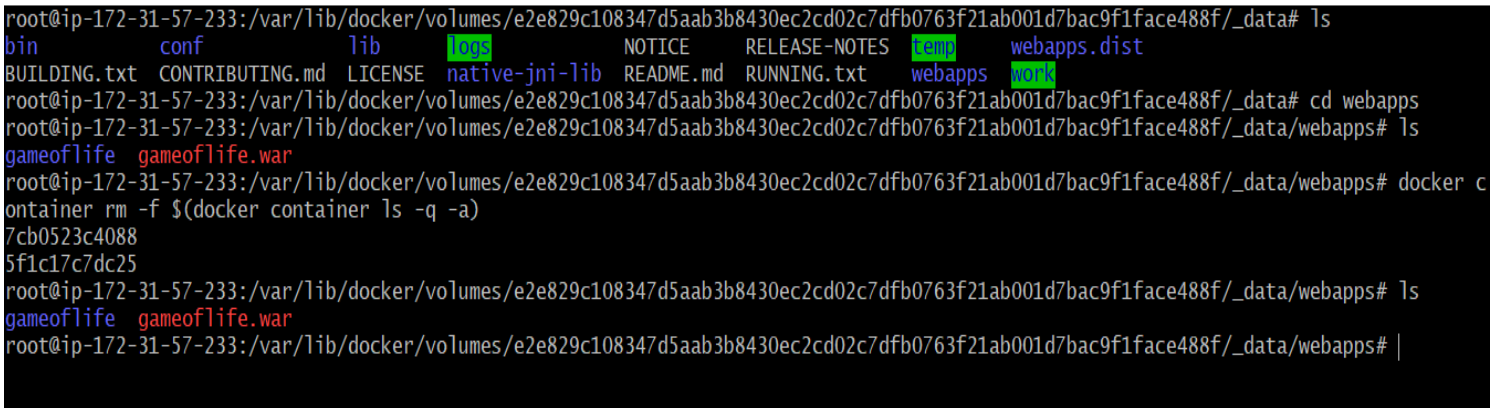
```
ubuntu@ip-172-31-33-175:~$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
spc                 1.0                 5cbdb83acfa7        About a minute ago  336MB
spcwithvol          1.0                 aead398da1f0        2 minutes ago       336MB
openjdk             8u272-slim          485edb6c7a50        12 days ago         288MB
ubuntu@ip-172-31-33-175:~$
```

- Now lets create a docker container from image without volume instruction and other with volume instruction

```
ubuntu@ip-172-31-33-175:~$ docker container run -d --name 'withoutvolume' -P spc:1.0
0e95802b011f8265ffe58a52293e29b33e08555a972f1173a9cda444e7c0bba9
ubuntu@ip-172-31-33-175:~$ docker container run -d --name 'withvolume' -P spcwithvol:1.0
d9a262e9ea198e5867f6d1c73b28ac021830be7608d98e5deffe86e080ac66df
ubuntu@ip-172-31-33-175:~$
```



* if we can see the path “/var/lib/docker/volumes/” the container's data will be present and if we delete the containers also data will be still present.



- “Docker volume ls “ is the best command to know the volume info
- Even if we delete the the docket container the volume/data is still present for that best way is to create a docker volume by volume instrecton or creating docker file with ‘volume’ instruction
- What if the VOLUME instruction is not used in Docker Image?
  https://directdevops.blog/2019/10/03/docker-volumes/
- If your organization is running some kind of third party storage like netapp, vsphere storage etc Refer Here to install volume drivers https://hub.docker.com/search?q=&type=plugin&category=volume
- for cloud related storage drivers http://docs.docker.oeynet.com/docker-for-azure/persistent-data-volumes/#use-a-unique-volume-per-task

Docker volume commands:-

1) docker volume ls {to view all volumes}
2) docker volume prune {to delete all volumes explicitly}
3) docker volume create volume_name {to create a new volume}
4) docker volume inspect volume_name {to inspect the volume}
5)

```
root@ip-172-31-57-233:/# docker volume create volume1
volume1
root@ip-172-31-57-233:/# docker volume inspect volume1
[
    {
        "CreatedAt": "2021-03-19T16:13:12Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/volume1/_data",
        "Name": "volume1",
        "Options": {},
        "Scope": "local"
    }
]
root@ip-172-31-57-233:/# docker volume ls
DRIVER    VOLUME NAME
local     volume1
root@ip-172-31-57-233:/# docker volume prune
WARNING! This will remove all local volumes not used by at least one container.
Are you sure you want to continue? [y/N] y
Deleted Volumes:
volume1

Total reclaimed space: 0B
root@ip-172-31-57-233:/#
```

Activity:-

- Lets create a volume and mount it to tomcat container {docker container run -d -P --name my-cont-1 --volume volume1:/usr/local/tomcat/volume1 tomcat:jdk8-openjdk}
- Not if we see the container inspect it is mounted
-

```
"Mounts": [
    {
        "Type": "volume",
        "Name": "volume1",
        "Source": "/var/lib/docker/volumes/volume1/_data",
        "Destination": "/usr/local/tomcat/volume1",
        "Driver": "local",
        "Mode": "z",
        "RW": true,
        "Propagation": ""
    }
],
```

Activity 2:-

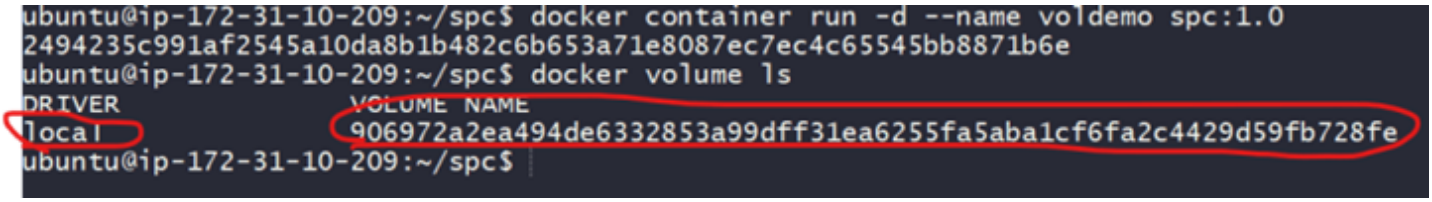
EG:-

```
FROM openjdk:8
ARG url=https://referenceappkhaja.s3-us-west-2.amazonaws.com/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar
ENV myfilename=spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar
```

```
RUN mkdir /app && cd /app && wget ${url}
VOLUME /app
WORKDIR /app
EXPOSE 8080
CMD ["java",  "-jar",  "spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar"
```

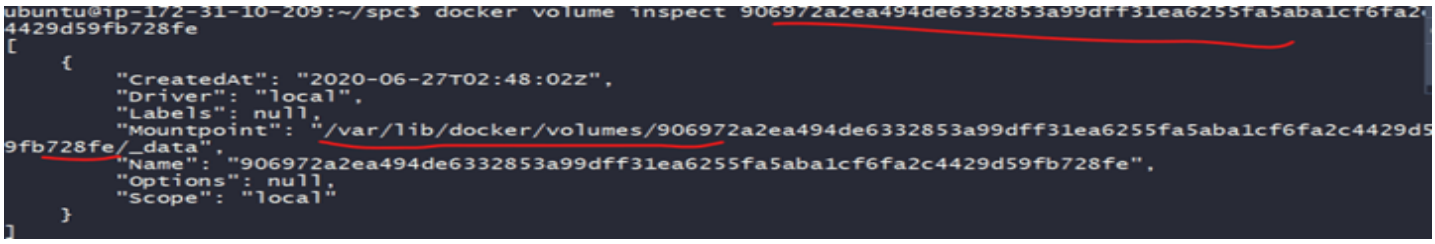
- Build the image
- Now execute following commands

```
docker volume --help
docker volume ls
docker container run -d spc:1.0
docker volume ls
```

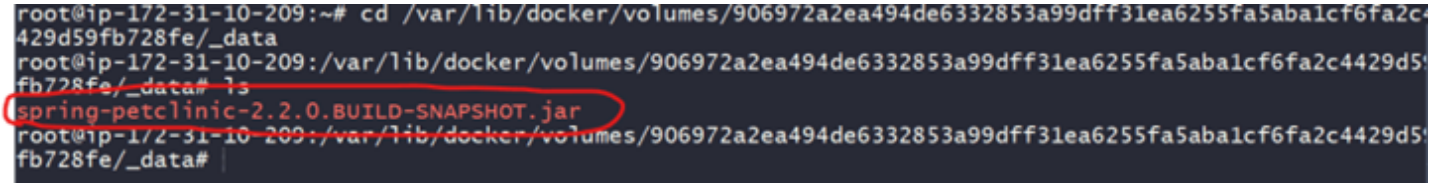


- Lets inspect the volume

```
docker volume inspect <volname>
```



- Lets navigate to the mount point



- Now lets remove the container and look into volume

```
docker container rm -f voldemo
docker volume ls
```

- Volume is still available after the container is deleted, so we don't lose data.
- Planning for volume creation in Dockerfile might not happen in all cases, the we need to create and mount volumes
- Volumes are of two types

    i) anonymous volumes

ii) named volumes

Let's create a named volume:-

My using below command we can create our own volume and default location of the volume is "var/lib/docker/volumes"

docker volume create <volume name>

```
root@ip-172-31-80-14:/docker# docker volume create mahivolume
mahivolume
root@ip-172-31-80-14:/docker# docker volume ls
DRIVER              VOLUME NAME
local               6886296f41fcd9ddcd6c28507c7b9ae062b497500a5476b5e82d1d25ee17c159
local               a208f9da7958cbd538c6cd19175b43eea745a000f32cb5cb195b0b801978e47c
local               d928b7801ad2a4b743bde0d466c62ae65b2853d277e6532678152193337aee7f9
local               d9017d6c1a2fc7e5f0ee3d1a0a1ec8ef0ada7bf51c3ca2db4d36ff0a245a9a65
local               mahivolume
root@ip-172-31-80-14:/docker# docker volume inspect mahivolume
[
    {
        "CreatedAt": "2020-12-11T02:49:46Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/mahivolume/_data",
        "Name": "mahivolume",
        "Options": {},
        "Scope": "local"
    }
]
root@ip-172-31-80-14:/docker#
```

- Lets create an alpine container with this volume (mahivolume)mounted

```
docker container run -d --name alp1 --mount source=myvol,target=/test alpine sleep 1d
docker volume inspect myvol
docker container inspect alp1
docker container exec alp1 df -h
docker container exec alp1 touch /test/1.txt
docker container exec alp1 ls /test
```

```
"Mounts": [
    {
        "Type": "volume",
        "Name": "mahivolume",
        "Source": "/var/lib/docker/volumes/mahivolume/_data",
        "Destination": "/test",
        "Driver": "local",
        "Mode": "z",
        "RW": true,
        "Propagation": ""
    }
```

```
ubuntu@ip-172-31-10-209:~/spc$ docker container exec alp1 df -h
Filesystem                Size      Used Available Use% Mounted on
overlay                   7.7G      2.3G      5.4G  30% /
tmpfs                    64.0M         0     64.0M   0% /dev
tmpfs                   491.8M         0    491.8M   0% /sys/fs/cgroup
shm                      64.0M         0     64.0M   0% /dev/shm
/dev/xvda1                7.7G      2.3G      5.4G  30% /test
/dev/xvda1                7.7G      2.3G      5.4G  30% /etc/resolv.conf
/dev/xvda1                7.7G      2.3G      5.4G  30% /etc/hostname
/dev/xvda1                7.7G      2.3G      5.4G  30% /etc/hosts
tmpfs                   491.8M         0    491.8M   0% /proc/acpi
tmpfs                    64.0M         0     64.0M   0% /proc/kcore
tmpfs                    64.0M         0     64.0M   0% /proc/keys
tmpfs                    64.0M         0     64.0M   0% /proc/timer_list
tmpfs                    64.0M         0     64.0M   0% /proc/sched_debug
tmpfs                   491.8M         0    491.8M   0% /proc/scsi
tmpfs                   491.8M         0    491.8M   0% /sys/firmware
```

```
ubuntu@ip-172-31-10-209:~/spc$ docker container exec alp1 touch /test/1.txt
ubuntu@ip-172-31-10-209:~/spc$ docker container exec alp1 ls /test/1.txt
/test/1.txt
ubuntu@ip-172-31-10-209:~/spc$ docker container exec alp1 ls /test/
1.txt
ubuntu@ip-172-31-10-209:~/spc$
```

- Now lets remove the container alp1 and create a new container alp2 with the same volume mount and see if the data is preserved

```
docker container rm -f alp1
docker volume ls
docker container run -d --name alp2 --mount source=myvol,target=/test alpine sleep 1d
docker container exec alp2 ls /test
```

```
root@ip-172-31-80-14:/docker# docker container run -d --name alp2 --mount source=mahivolume,target=/test alpine sleep 1d
128b92366062a5567cb261577e5e217d085ad9d6b2764f3468b21e183d155ddd
root@ip-172-31-80-14:/docker# docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
128b92366062        alpine              "sleep 1d"          23 seconds ago      Up 22 seconds                           alp2
root@ip-172-31-80-14:/docker# docker container exec alp2 ls /test
1.txt
root@ip-172-31-80-14:/docker#
```

- Let’s try to share the data between two container using the same volume

```
docker container run -d --name alp3 --mount source=myvol,target=/test alpine sleep 1d
docker container exec alp3 touch /test/2.txt
docker container exec alp2 ls /test
```

```
root@ip-172-31-80-14:/docker# docker container run -d --name alp3 --mount source=mahivolume,target=/app alpine sleep 1d
f1a76411e9817a24c68ab66ecc5b0064bc703d3cc43d57c4e111418f83df7814
root@ip-172-31-80-14:/docker# docker container exec alp3 ls /app
1.txt
root@ip-172-31-80-14:/docker# docker container exec alp3 mkdir /app/drive1
root@ip-172-31-80-14:/docker# docker container exec alp2 ls /test
1.txt
drive1
```

- Docker volume has also an option of docker volume drivers which allows volumes to be created in nfs, aws s3 etc

- Now let's remove the containers

```
docker container rm -f $(docker container ls -a -q)
docker volume ls
```

Topic:- Docker Networking

Some imp articles on docker networking

1) For single host networking

https://directdevops.blog/2019/10/05/docker-networking-series-i/

2) For docker swarm

https://directdevops.blog/2019/10/07/docker-swarm-mode/

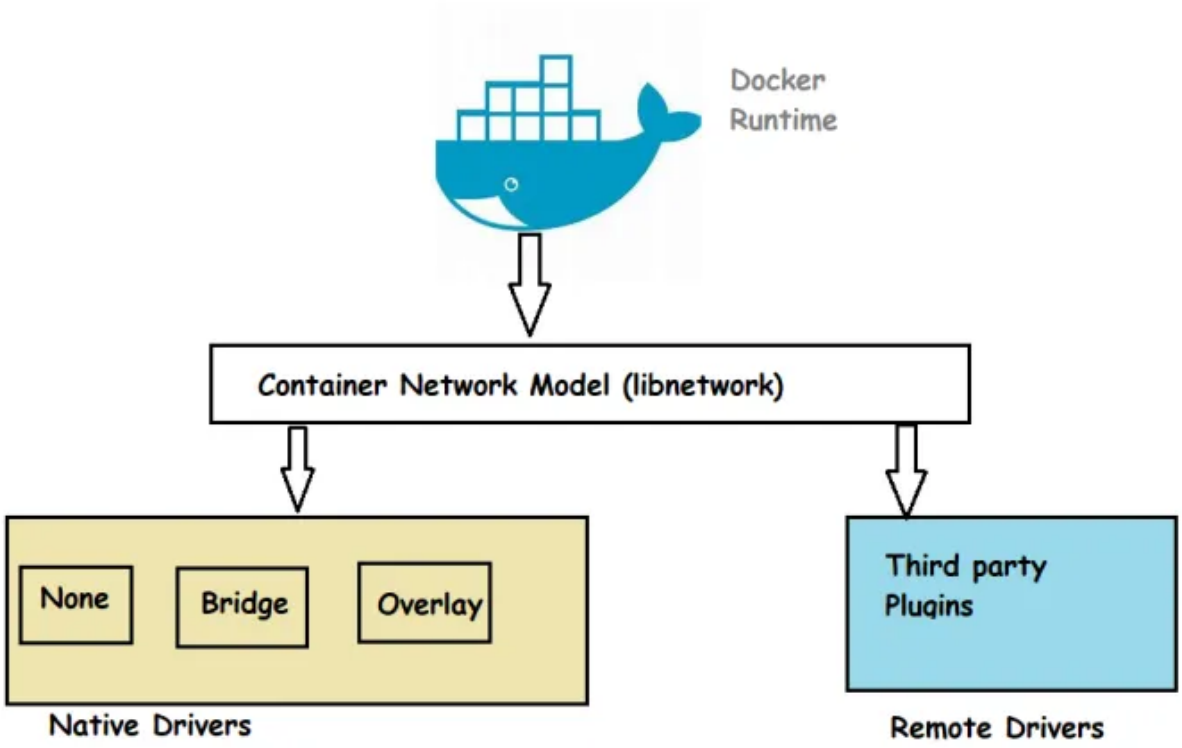
3) For overlay & multi-host networking

https://directdevops.blog/2019/10/07/docker-networking-series-ii-overlay-networks/

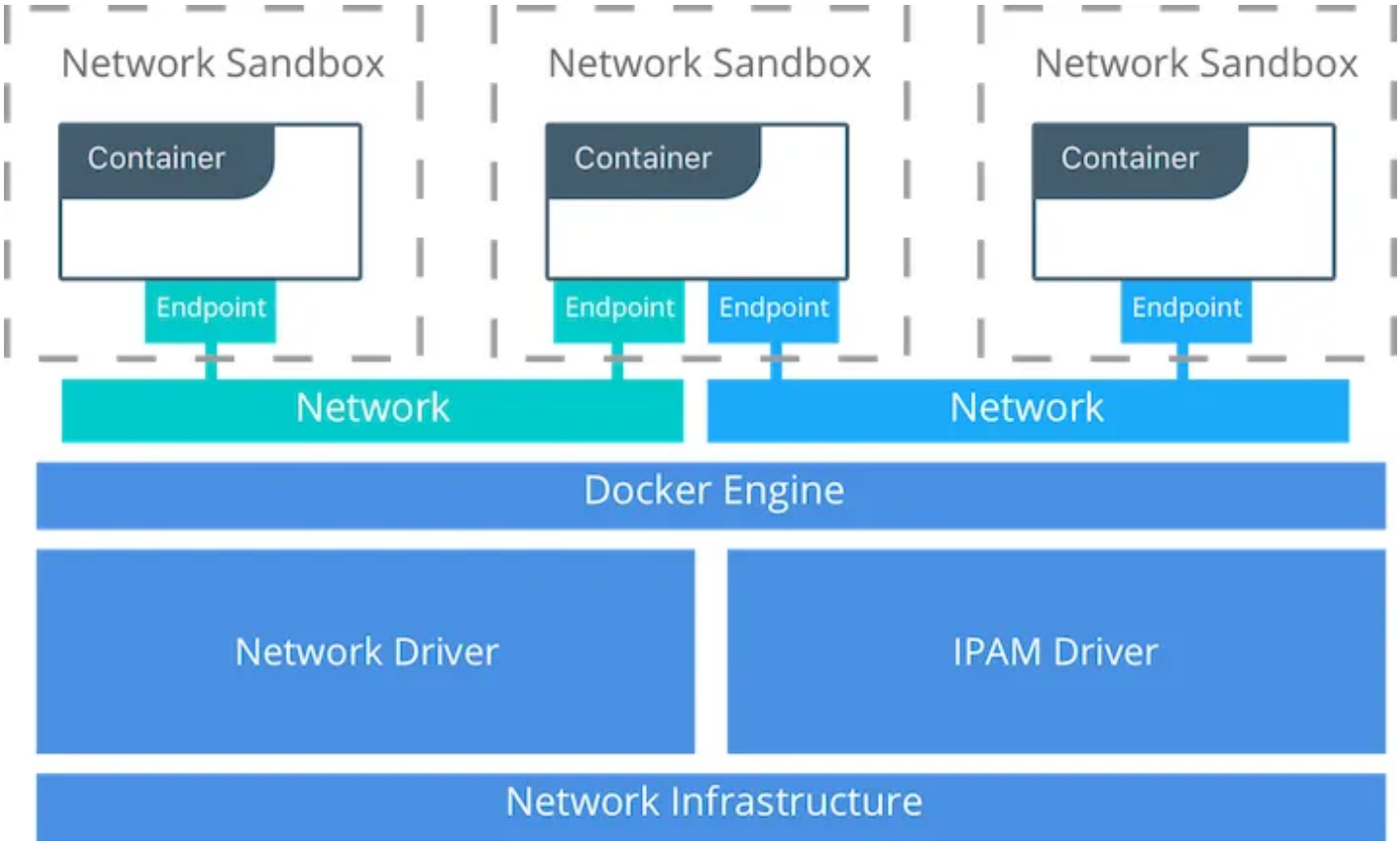
4) Docker networking follows a specification called as Container Network Model (CNM)

https://github.com/moby/libnetwork/blob/master/docs/design.md

- Docker has a opensource project called as *libnetwork* which is implementation of CNM



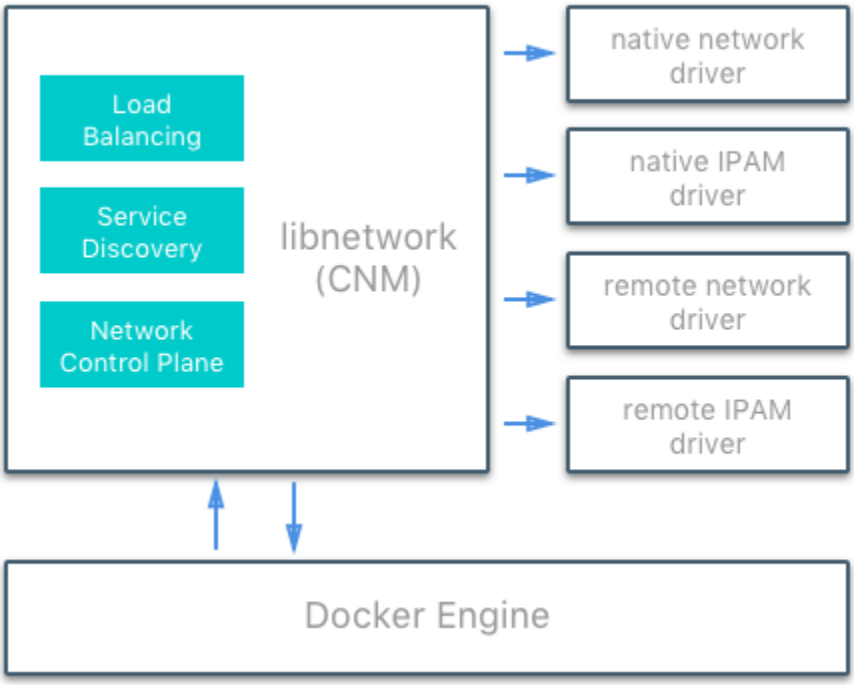
- Docker demon whenever it wants to do anything with networking, It would try to speak with any implementation of container network model.
- Generally it interact with lib-network
- This lib network to implement its networking mechanism it has two different things
- i) Native-drivers ii) Remote drivers (If require third party drivers we download it)
- i) Native-drivers:- None, Drudge, overlay,mack vlan..etc
- So the libnetwork will relay on individual drivers
- Every docker container has “networking sandbox” which means networking stack with in the container
- A network interface which is present inside the docker container is called as “Endpoint”
- Within the docker container we will have one network interface which connects to the network that is created.



CNM constraints:- i) Sand-Box; ii) Endpoint; iii) Network

- 1. Sandbox: this is containers network stack with features such as container network interfaces, routing tables and DNS settings
- 2. Endpoint: joins sandbox to network
- 3. Network: Network is collection of endpoints with connectivity b/w them.
- Whenever we need to connect to network we need whole network to be workable for that Docker need ‘CNM-driver interface’

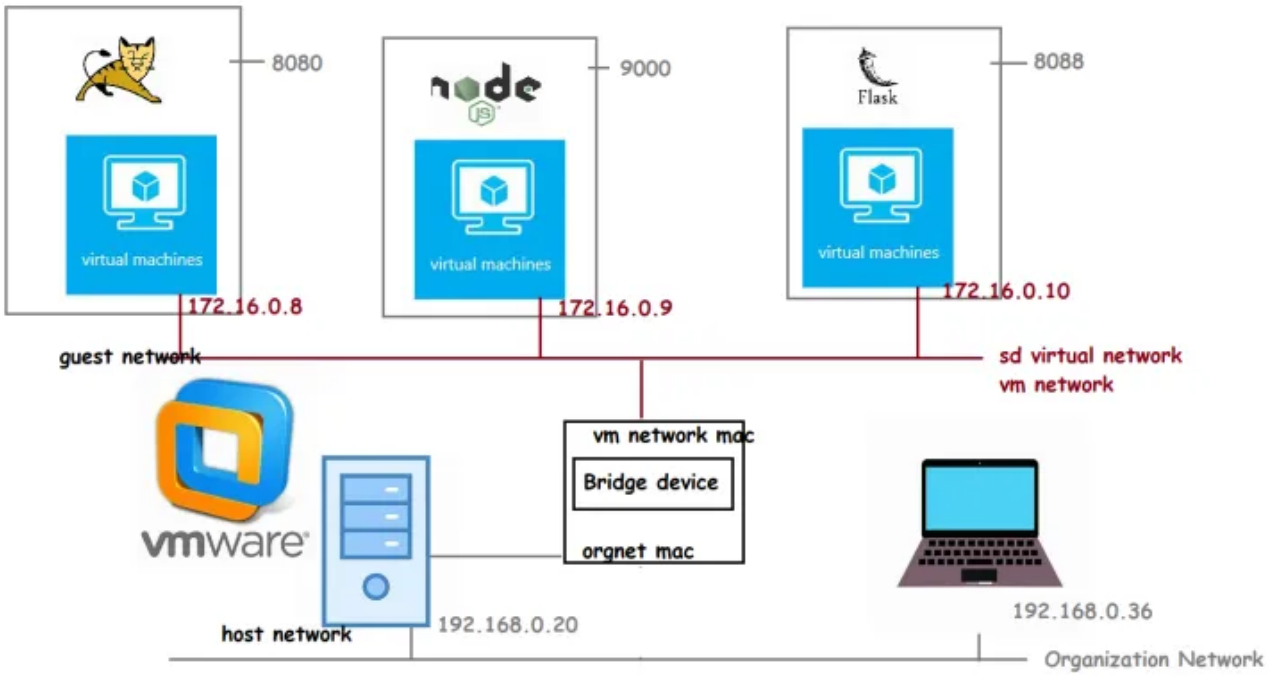
CNM Driver Interfaces:-



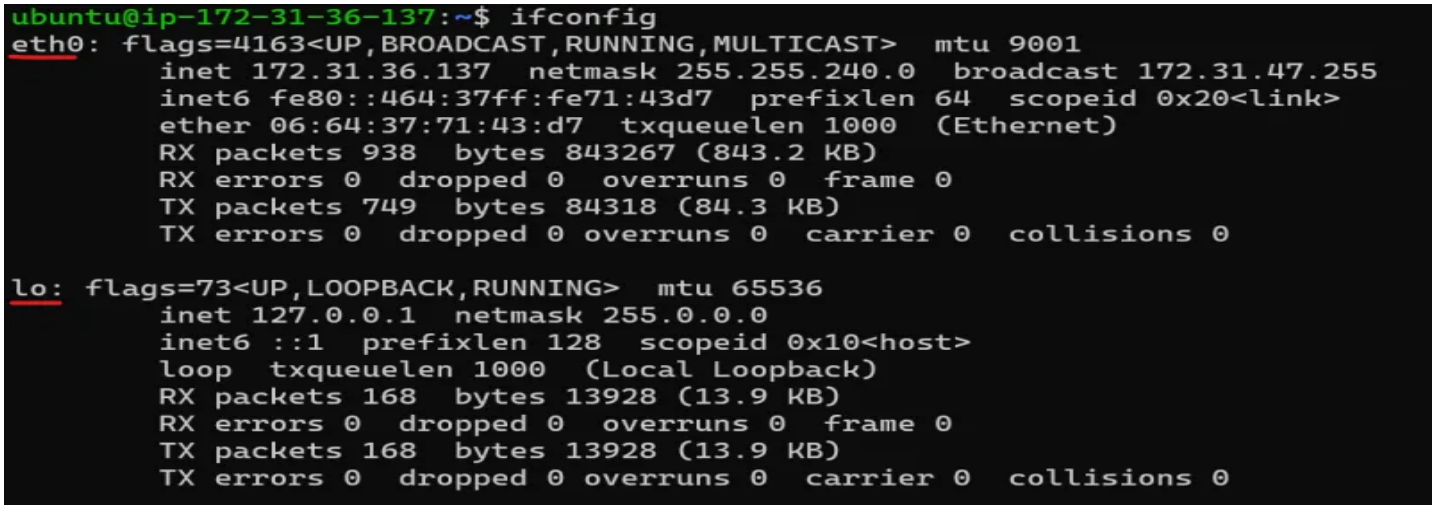
CNM provides two pluggable and open interfaces

- Network Driver
- IPAM (IP Address Management) Drivers

Host Network and Guest Network in Networking



- Docker has created their own pluggable component which is called ‘lib-network’
- We have two concepts in networking i) host network and ii) guest network
- As shown in above figure if we have a server installed with vmware software and 3 vms are hosted on it. If we want to connect to any of those vms, we need a bridevice
- That vms will connect to a vm-network, and it is connected to a bridge device any user from the outer side will access this via that bridge device. This network setup is called “guest-network”
- The network connection which the root vm is obtained is called host-network
- Lets launch a vm with plain os and execute ‘ifconfig’ we can observe “eth0(which enables me to get virtual connection) ” “loopback address”



- And now install docker and execute “ifconfig”. And then execute the ifconfig and we can see ‘docker0 (docker network)’. Once we install docker we are getting an additional interface.

```
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:c5:c2:fb:5d  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 9001
        inet 172.31.36.137  netmask 255.255.240.0  broadcast 172.31.47.255
        inet6 fe80::464:37ff:fe71:43d7  prefixlen 64  scopeid 0x20<link>
        ether 06:64:37:71:43:d7  txqueuelen 1000  (Ethernet)
        RX packets 80053  bytes 118281324 (118.2 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 11095  bytes 797047 (797.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 254  bytes 23968 (23.9 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 254  bytes 23968 (23.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

ubuntu@ip-172-31-36-137:~$
```

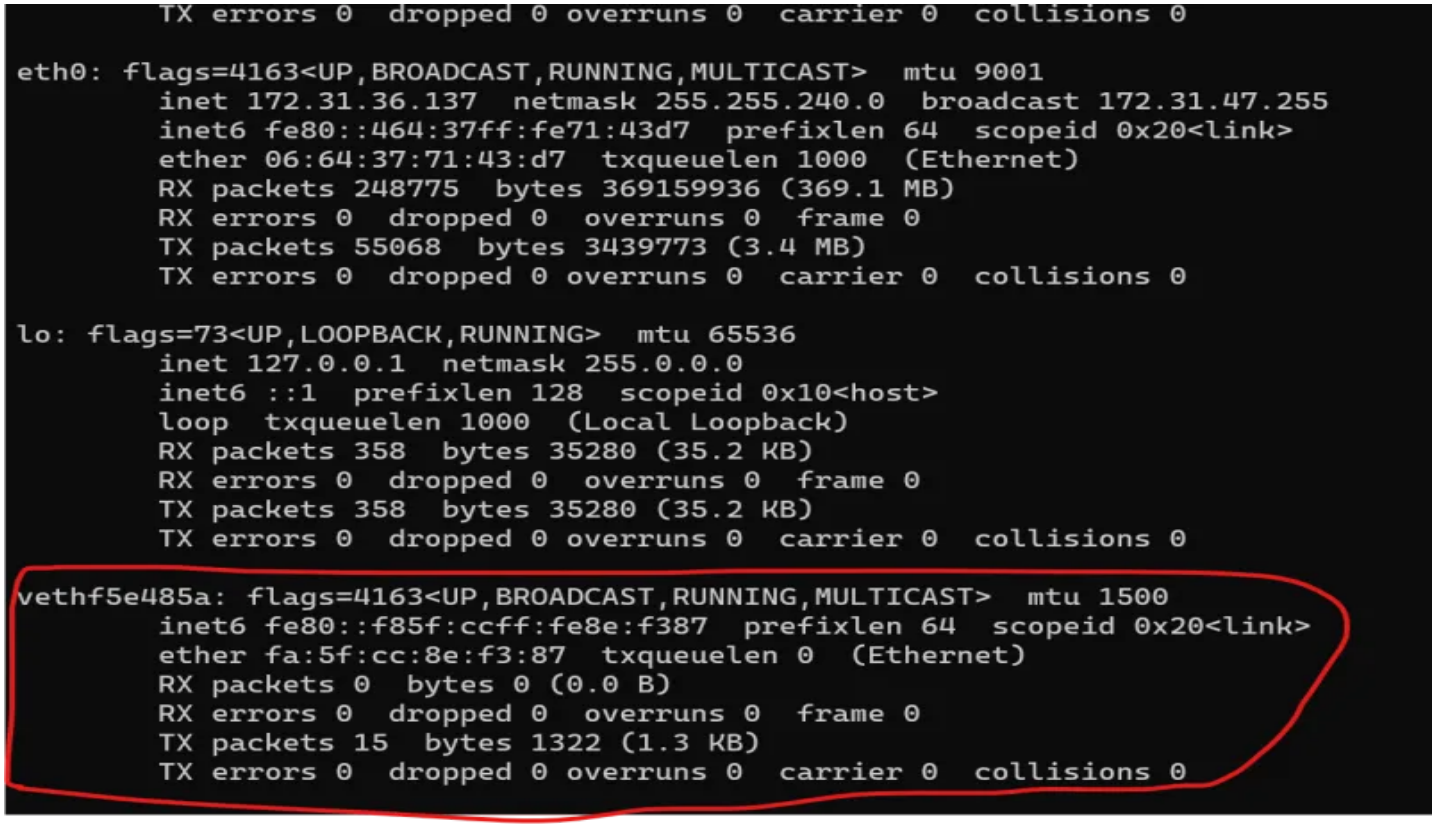
- Run the container and inspect that container

```
docker container run -d --name my-cont-1 tomcat:jdk8-openjdk
docker container inspect my-cont-1
```

- If we observe the newly created container is taking a new IP address with the docker ip address pool and also it is having a separate MAC address.


```
"Networks": {
    "bridge": {
        "IPAMConfig": null,
        "Links": null,
        "Aliases": null,
        "NetworkID": "6da03dd811ef9f5632811ba5d4ba83c76f745d9f9c41cd2d18fe287250653fba",
        "EndpointID": "9b8e08d4fb86338629b2e698d431321da86966b27cab3f219a7f8a357e14ddc4",
        "Gateway": "172.17.0.1",
        "IPAddress": "172.17.0.2",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:ac:11:00:02",
        "DriverOpts": null
    }
}
```


- Now execute ifconfig. We can see extra is added “vethf5e485a”



Docker native network drivers:

Host: with a host driver, a container uses the networking stack of the host.
Bridge: This is the default driver. Connecting multiple containers on the same bridge network in a single host. The containers created by default are connected to bridge network
Overlay: This driver creates an overlay network to support multi-host containers.
MACVLAN:- Establish a connection b/w container interfaces and host interfaces. It provides Private ip. It will use a MAC address to establish a connection. A container which is created using MACVLAN will have a MAC address.
None:- used to create containers with its own network stack. It doesn't configure any interfaces inside the container so it can't establish any sort of communication b/w containers

Activity:-

- Lets experiment with default bridge network
- Create two containers dc1 and dc2

```
docker container run --name dc1 -d alpine sleep 1d
docker container run --name dc2 -d alpine sleep 1d
```

```
ubuntu@ip-172-31-36-137:~$ docker container run --name dc1 -d alpine sleep 1d
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
188c0c94c7c5: Pull complete
Digest: sha256:c0e9560cda118f9ec63ddefb4a173a2b2a0347082d7dff7dc14272e7841a5b5a
Status: Downloaded newer image for alpine:latest
3645b6d733e55128b82f8c3ba7c9de955a8f671443934645a0b64f129e642bc5
ubuntu@ip-172-31-36-137:~$ docker container run --name dc2 -d alpine sleep 1d
2342a0a139d813e235111d3a4f2f18e84f17e7e52635f26a37b4b9eb1b2101bf
ubuntu@ip-172-31-36-137:~$ docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
   NAMES
2342a0a139d8        alpine              "sleep 1d"          8 seconds ago       Up 8 seconds
   dc2
3645b6d733e5        alpine              "sleep 1d"          15 seconds ago      Up 15 seconds
   dc1
ubuntu@ip-172-31-36-137:~$
```

* Now lets login into dc1 and ping dc2

```
ubuntu@ip-172-31-36-137:~$ docker container exec dc1 ping -c 4 dc2
ping: bad address 'dc2'
ubuntu@ip-172-31-36-137:~$ docker container exec dc2 ping -c 4 dc1
ping: bad address 'dc1'
ubuntu@ip-172-31-36-137:~$
```

Now let's find IP addresses of dc1 and dc2 . try pinging with ip address

```
  "Containers": {
      "2342a0a139d813e235111d3a4f2f18e84f17e7e52635f26a37b4b9eb1b2101bf": {
          "Name": "dc2",
          "EndpointID": "bebea7462e120698b399cb4561cb0ed728d9d1131cbb31c32068f3f20ea90752",
          "MacAddress": "02:42:ac:11:00:03",
          "IPv4Address": "172.17.0.3/16",
          "IPv6Address": ""
      },
      "3645b6d733e55128b82f8c3ba7c9de955a8f671443934645a0b64f129e642bc5": {
          "Name": "dc1",
          "EndpointID": "2755b4dd2ccf177130dd5ff715d07d6a0489fb1e0395def45d3c6330e14e57de",
          "MacAddress": "02:42:ac:11:00:02",
          "IPv4Address": "172.17.0.2/16",
          "IPv6Address": ""
      }
  },
  "Options": {
      "com.docker.network.bridge.default_bridge": "true",
```

```
ubuntu@ip-172-31-36-137:~$ docker container exec dc2 ping -c 4 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.097 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.081 ms
64 bytes from 172.17.0.2: seq=2 ttl=64 time=0.081 ms
64 bytes from 172.17.0.2: seq=3 ttl=64 time=0.083 ms

--- 172.17.0.2 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.081/0.085/0.097 ms
ubuntu@ip-172-31-36-137:~$ docker container exec dc1 ping -c 4 172.17.0.3
PING 172.17.0.3 (172.17.0.3): 56 data bytes
64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.062 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.076 ms
64 bytes from 172.17.0.3: seq=2 ttl=64 time=0.086 ms
64 bytes from 172.17.0.3: seq=3 ttl=64 time=0.079 ms

--- 172.17.0.3 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.062/0.075/0.086 ms
ubuntu@ip-172-31-36-137:~$
```

- As per the observations we can ping from one container to another container using ip address in default bridge network
- Now lets create user defined bridge network

```
ubuntu@ip-172-31-36-137:~$ docker network create --driver bridge my-network-1
b214436d75064348a2a9243a4a757d39ea2cdc3cea0c62ac900777b79bcae459
ubuntu@ip-172-31-36-137:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
6da03dd811ef        bridge              bridge              local
eaa8cef76806        host                host                local
b214436d7506        my-network-1        bridge              local
d818b187f421        none                null                local
ubuntu@ip-172-31-36-137:~$
```

Now let's inspect the created network `docker network inspect <network-name>`

```
ubuntu@ip-172-31-36-137:~$ docker network inspect my-network-1
[
    {
        "Name": "my-network-1",
        "Id": "b214436d75064348a2a9243a4a757d39ea2cdc3cea0c62ac900777b79bcae459",
        "Created": "2020-11-09T02:59:45.160043336Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
```

* Lets create two containers as shown above in user-defined network

```
ubuntu@ip-172-31-36-137:~$ docker container run -d --name udc1  --network my-network-1 alpine sleep 1d
44fd00599d8f094f4d694c09f67623b8dc36643cdc5bd79e440774d71976817c
ubuntu@ip-172-31-36-137:~$ docker container run -d --name udc2  --network my-network-1 alpine sleep 1d
c227b5eb3b49d4c837ee802496662ae470ddfd390ae5a6e307df3ad87747417b
ubuntu@ip-172-31-36-137:~$ docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
   NAMES
c227b5eb3b49        alpine              "sleep 1d"          6 seconds ago       Up 6 seconds
   udc2
44fd00599d8f        alpine              "sleep 1d"          15 seconds ago      Up 14 seconds
   udc1
2342a0a139d8        alpine              "sleep 1d"          13 minutes ago      Up 13 minutes
   dc2
3645b6d733e5        alpine              "sleep 1d"          13 minutes ago      Up 13 minutes
   dc1
ubuntu@ip-172-31-36-137:~$
```

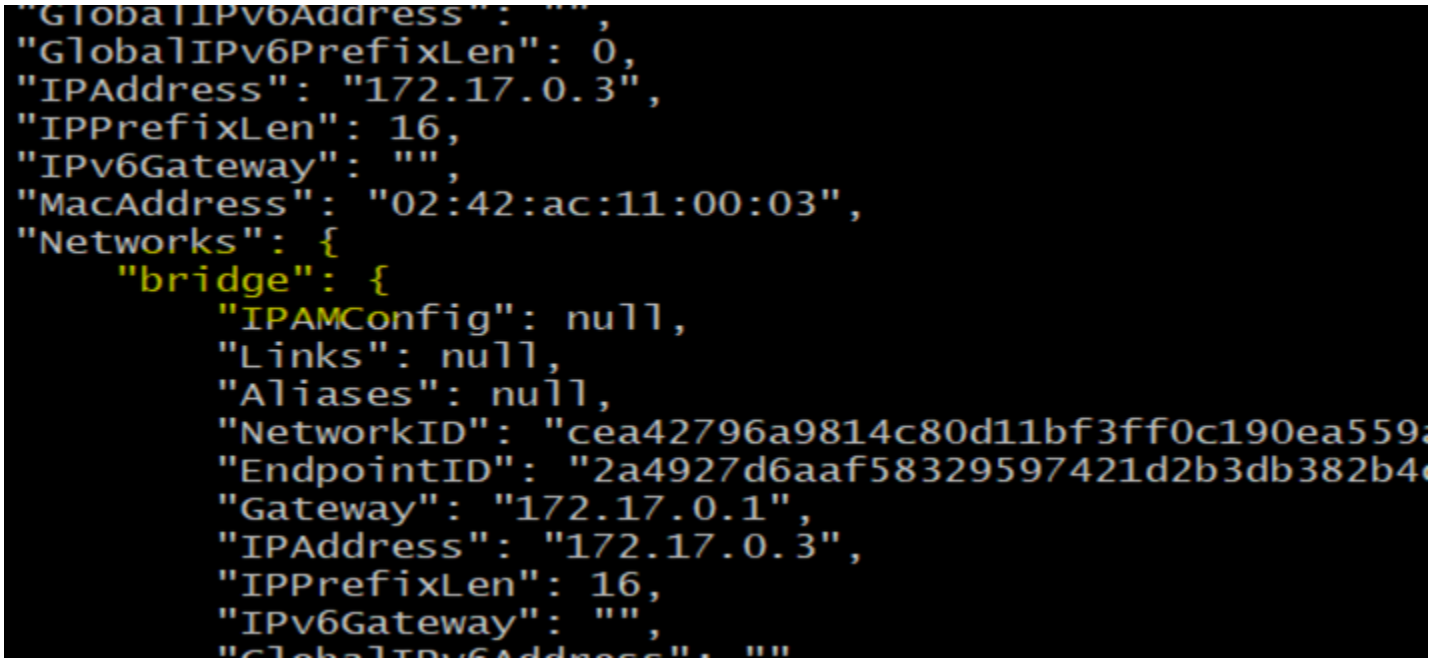
Now let's try to ping from one container to another.

```
ubuntu@ip-172-31-36-137:~$ docker container exec udc1 ping -c 4 udc2
PING udc2 (172.18.0.3): 56 data bytes
64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.128 ms
64 bytes from 172.18.0.3: seq=1 ttl=64 time=0.086 ms
64 bytes from 172.18.0.3: seq=2 ttl=64 time=0.082 ms
64 bytes from 172.18.0.3: seq=3 ttl=64 time=0.096 ms

--- udc2 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.082/0.098/0.128 ms
ubuntu@ip-172-31-36-137:~$ docker container exec udc2 ping -c 4 udc1
PING udc1 (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.060 ms
64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.086 ms
64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.095 ms
64 bytes from 172.18.0.2: seq=3 ttl=64 time=0.091 ms

--- udc1 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.060/0.083/0.095 ms
ubuntu@ip-172-31-36-137:~$
```

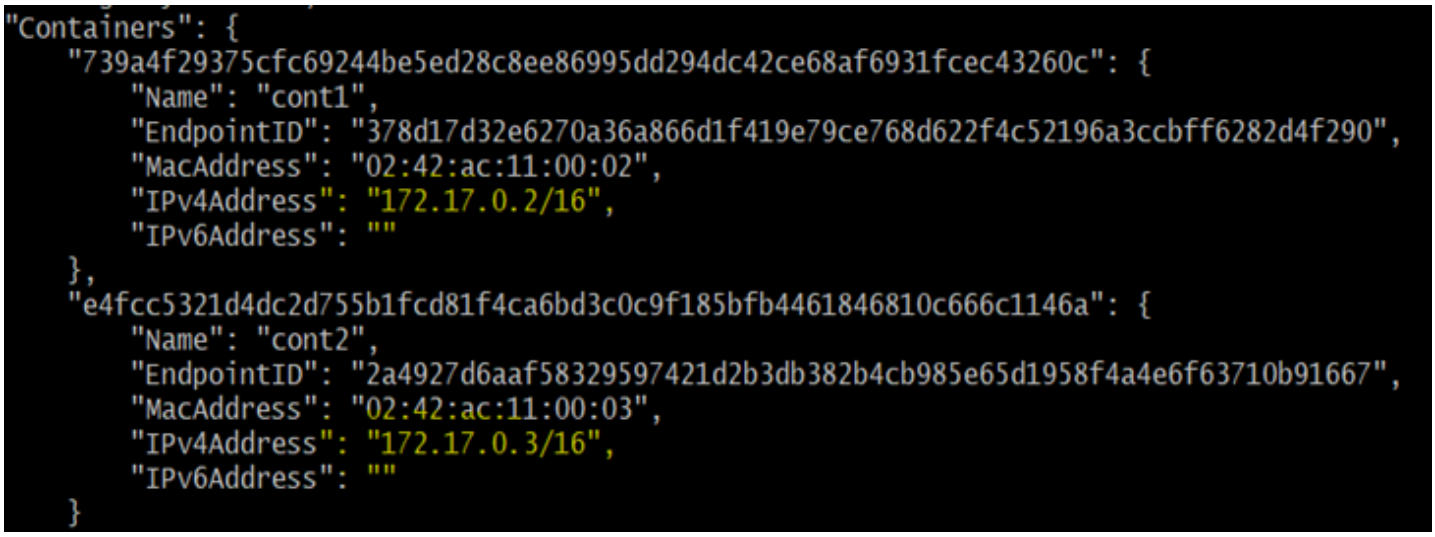
- User defined bridge networks have name based resolution enabled.
- Running applications on a single host might lead to a single point of failure, we need to run our containers on multiple machines and there should be connectivity.
- This is where we will be learning Docker multi-host networking (Overlay driver) and We will introduce Docker Swarm

Activity:-

Create a nginx container and execute docker inspect <container-id>



now let's inspect bridge network" docker network inspect bridge"



Bridge Network

- In docker by default we will have a default bridge network with name '**bridge**'
- Let's create a new bridge (user-defined bridge)

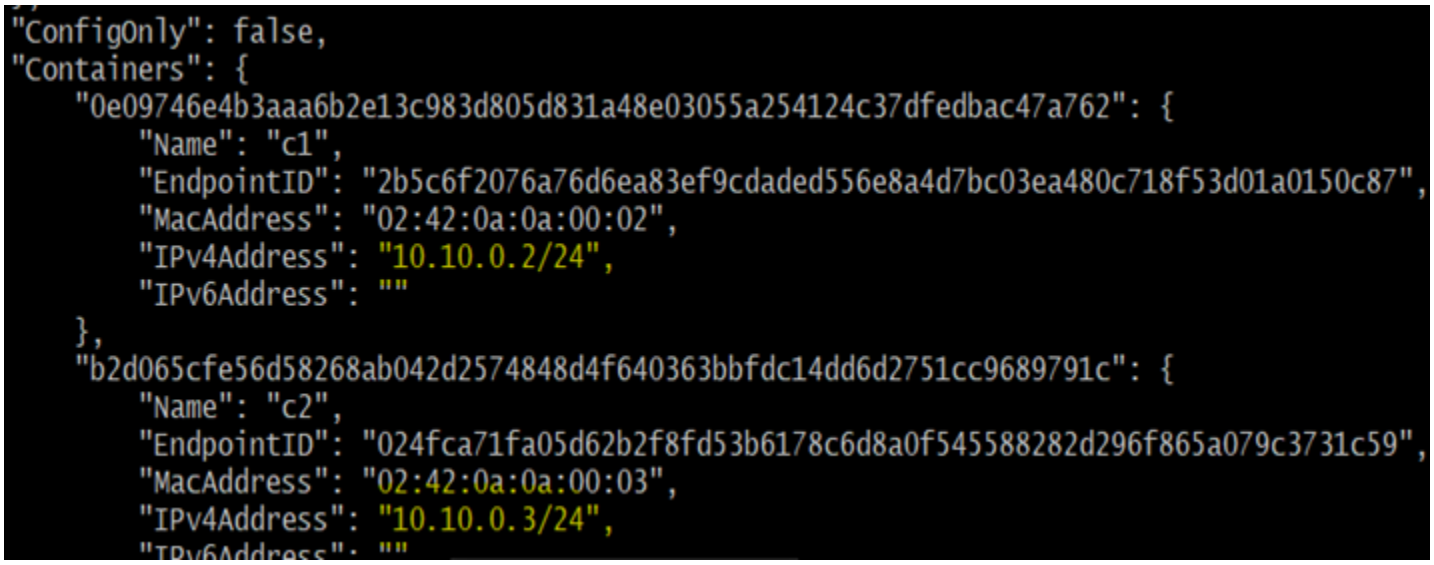
docker network create --help
docker network create --driver bridge --subnet 10.10.0.0/24 mybridge

```
ubuntu@ip-172-31-10-209:~$ docker network create --driver bridge --subnet 10.10.
0.0/24 mybridge
d3316cef5e151a28ebce6a15e8b3f58c06a61b67ed684ea7ef5da6e496c5863b
ubuntu@ip-172-31-10-209:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
f7aaae1d88e9        bridge              bridge              local
5399639e2878        host                host                local
d3316cef5e15        mybridge            bridge              local
1cf82ac2db95        none                null                local
```

Let's create two containers in side "mybridge" network

```
docker container run -d --name c1 --network mybridge alpine sleep 1d
docker container run -d --name c2 --network mybridge alpine sleep 1d
```

Let's find the ip address of the container (docker network inspect <mybridge>)



Even the containers are able to pingable to each other

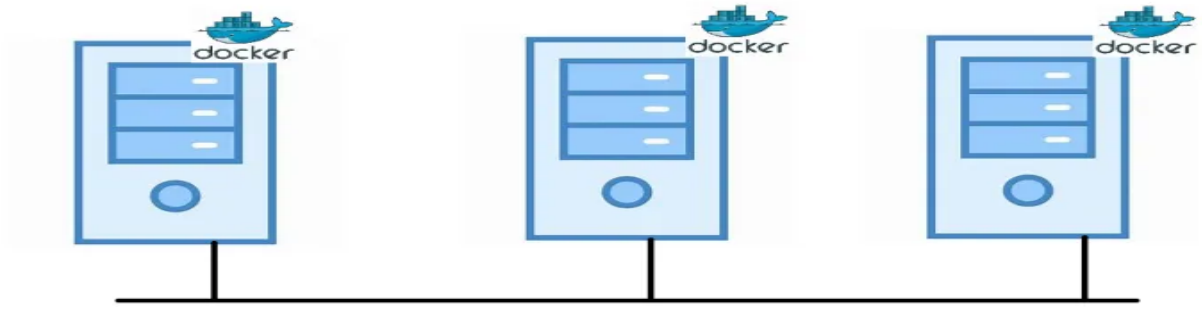
```
root@ip-172-31-80-14:/docker/file8# docker container exec c2 ping -c 4 c1
PING c1 (10.10.0.2): 56 data bytes
64 bytes from 10.10.0.2: seq=0 ttl=64 time=0.402 ms
64 bytes from 10.10.0.2: seq=1 ttl=64 time=0.095 ms
64 bytes from 10.10.0.2: seq=2 ttl=64 time=0.093 ms
64 bytes from 10.10.0.2: seq=3 ttl=64 time=0.127 ms

--- c1 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.093/0.179/0.402 ms
root@ip-172-31-80-14:/docker/file8# docker container exec c1 ping -c 4 c2
PING c2 (10.10.0.3): 56 data bytes
64 bytes from 10.10.0.3: seq=0 ttl=64 time=0.065 ms
64 bytes from 10.10.0.3: seq=1 ttl=64 time=0.097 ms
64 bytes from 10.10.0.3: seq=2 ttl=64 time=0.112 ms
64 bytes from 10.10.0.3: seq=3 ttl=64 time=0.097 ms

--- c2 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.065/0.092/0.112 ms
```

Docker Multi Host Networking

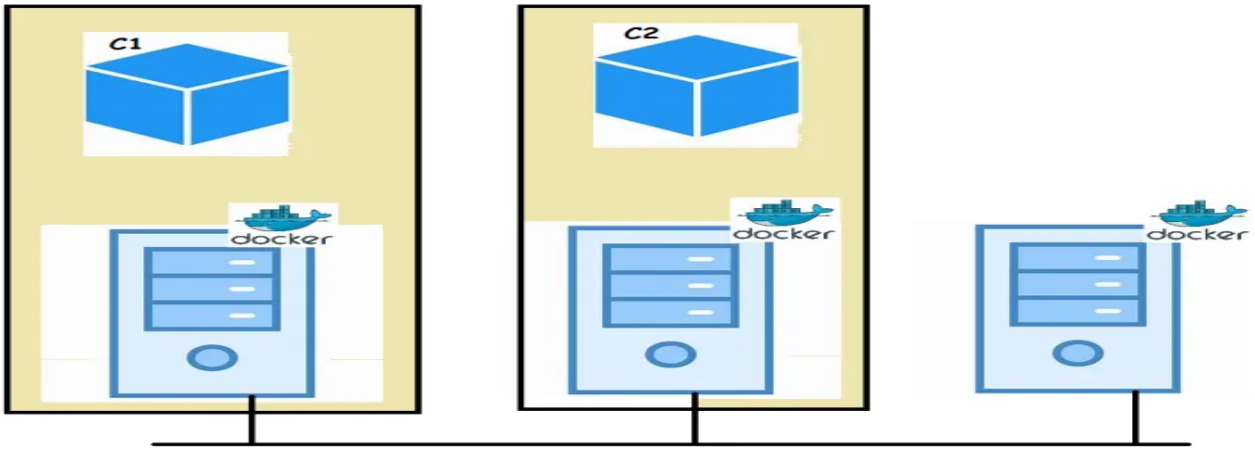
- Communication b/w containers if they are on different servers which are in the same network.

Activity:-

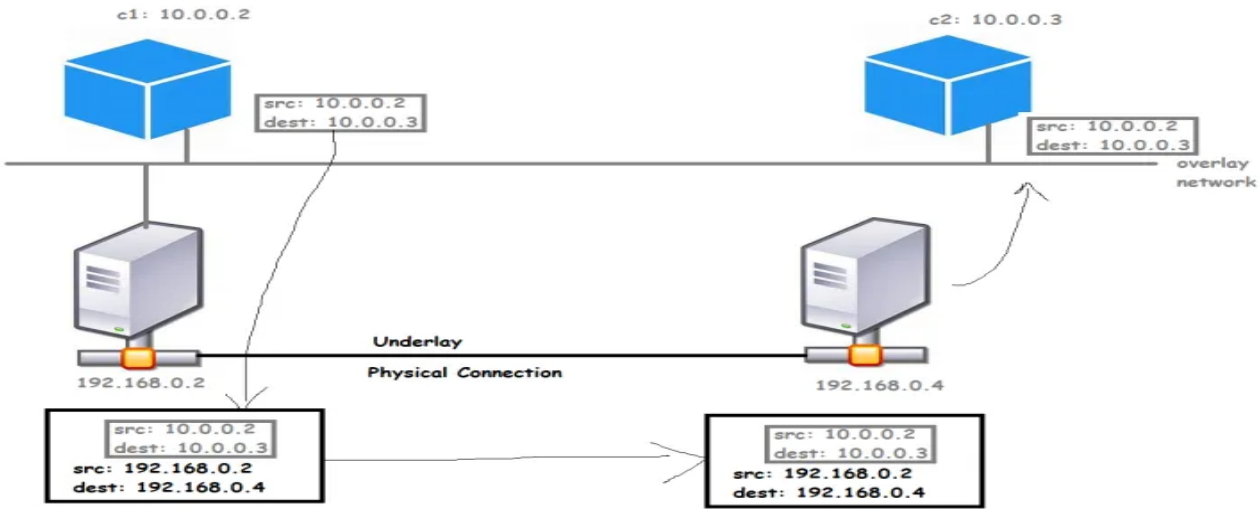
- Create 3 servers connected to each other within a network and install docker on all the three servers



- How can we enable communication b/w two container running on different hosts



- Docker has network driver which is called as overlay which can help in connecting two containers running on different hosts



- to understand overlay networks {https://directdevops.blog/2019/10/07/docker-networking-series-ii-overlay-networks/}
- When we are working with multiple docker hosts. It becomes extremely important for us to ensure
  - ★ avoid failures
  - ★ Scale the containers
  - ★ Simplicity
  - ★ Zero Downtime deployments
- What we are looking for is some software which manages this cluster and Orchestrate. This is exactly where Docker Swarm, Kubernetes and Apache mesos (many more) come into play.

Docker networking commands:-

1) docker network ls {to view all available networks }

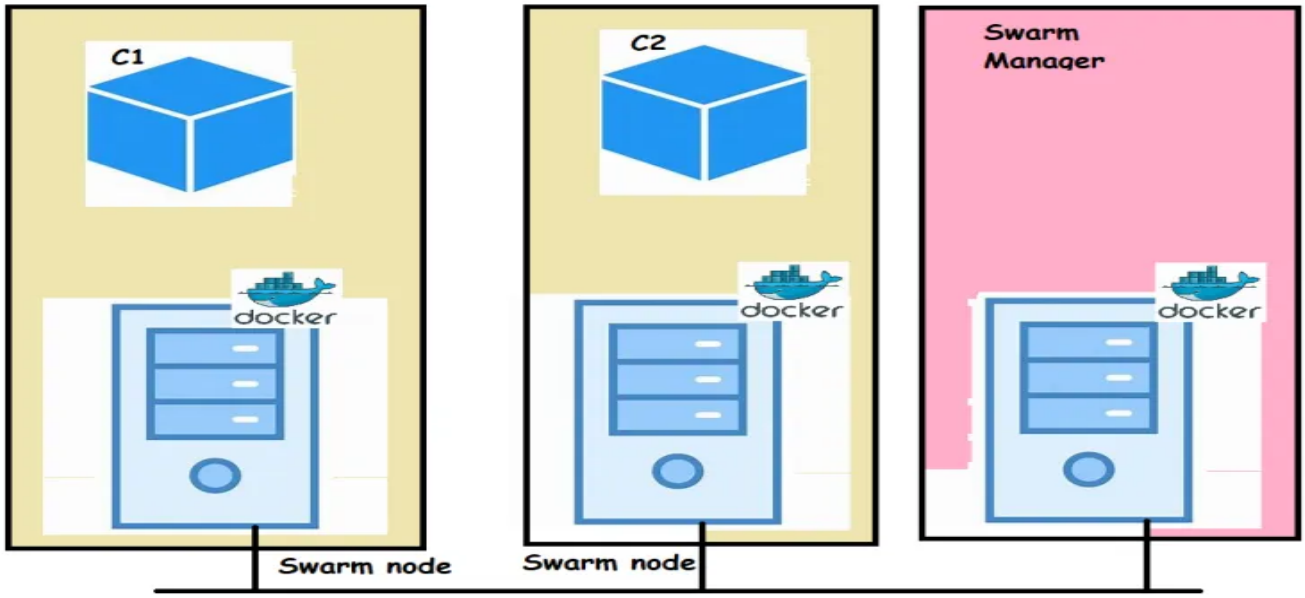
2) docker network create --driver drivername network-name {To create a customised network for container}
3) docker network inspect network-name {To inspect our network}
4) docker network connect network-name container-name

Topic:- Docker Swarm

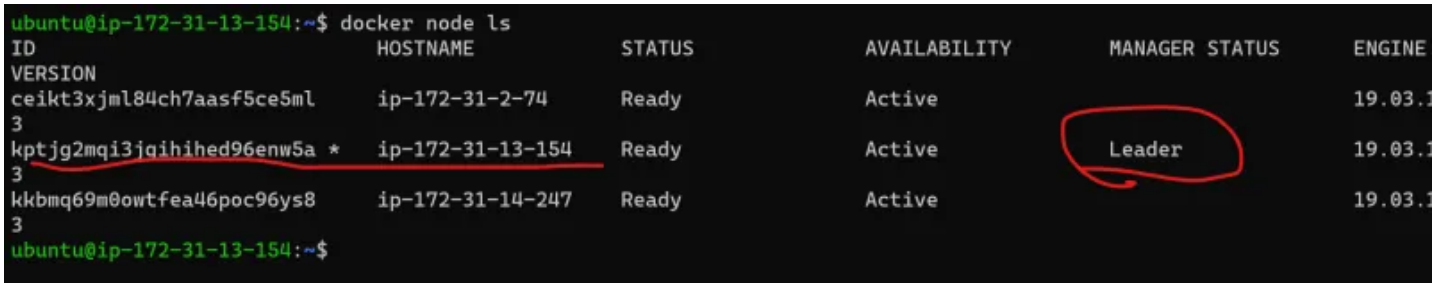
Docker Swarm creates an overlay network b/w multiple nodes
https://directdevops.blog/2019/10/07/docker-swarm-mode/

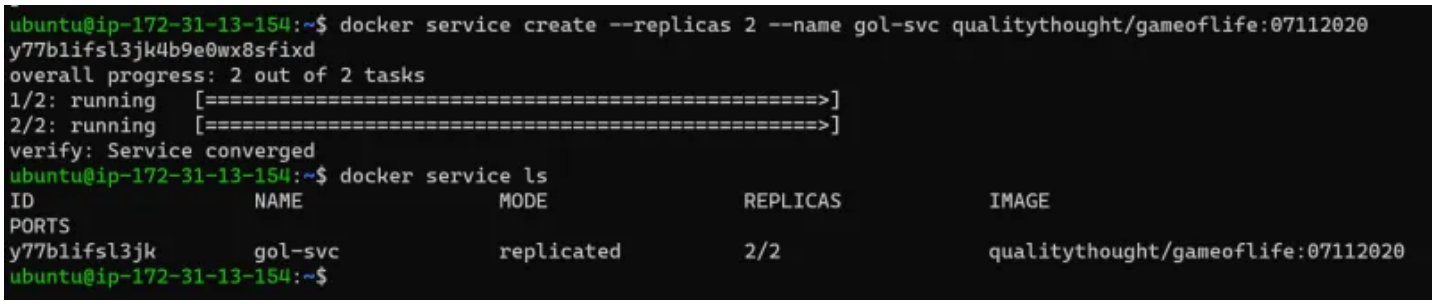
Docker Swarm is a cluster and it has two types of nodes

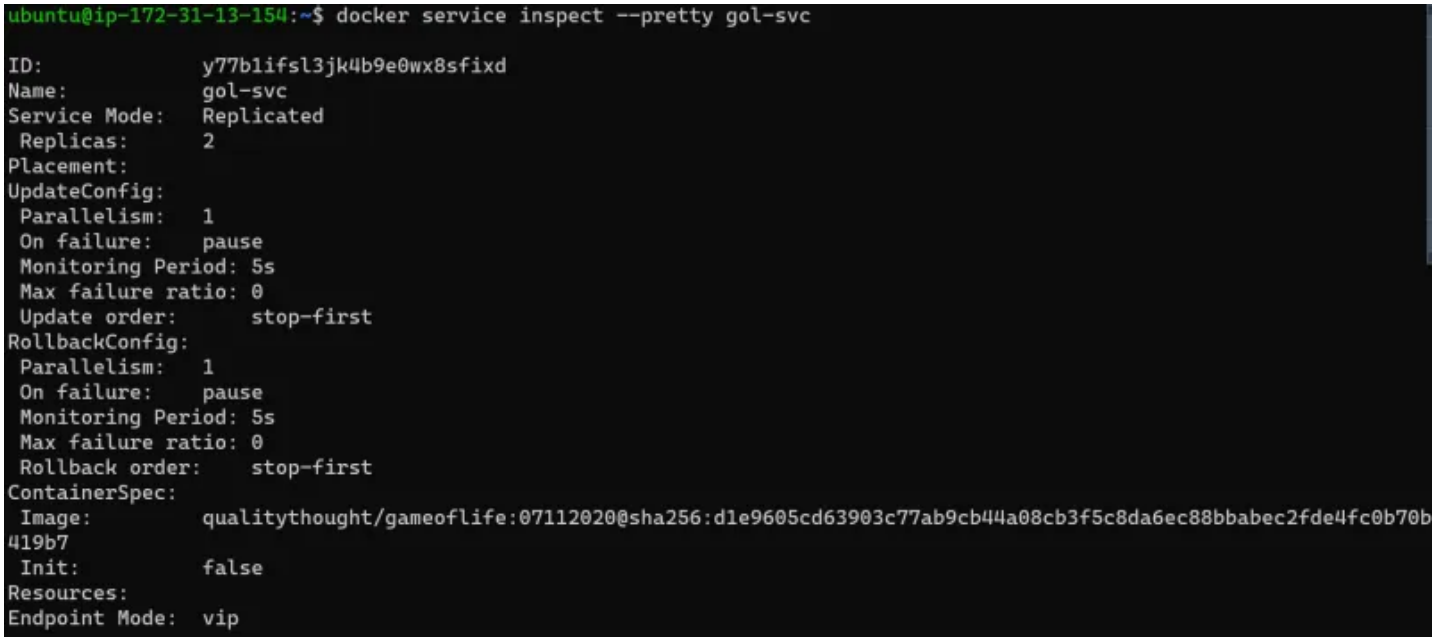
- Manager Node:
  - This is a point of contact for us to execute the desired state (Service).
  - This nodes dispatches a unit of work (task) to the worker nodes
  - You can have multiple manager nodes (only one will be primary manager)
- Worker Node:
  - They receive and execute the tasks dispatched from manager nodes
  - Multiple Nodes can be created
  - Agent that runs on this node reports the status of tasks assigned to it back to manager
  - Docker Swarm Representation:-



- Create the docker swarm cluster as mentioned in the above link

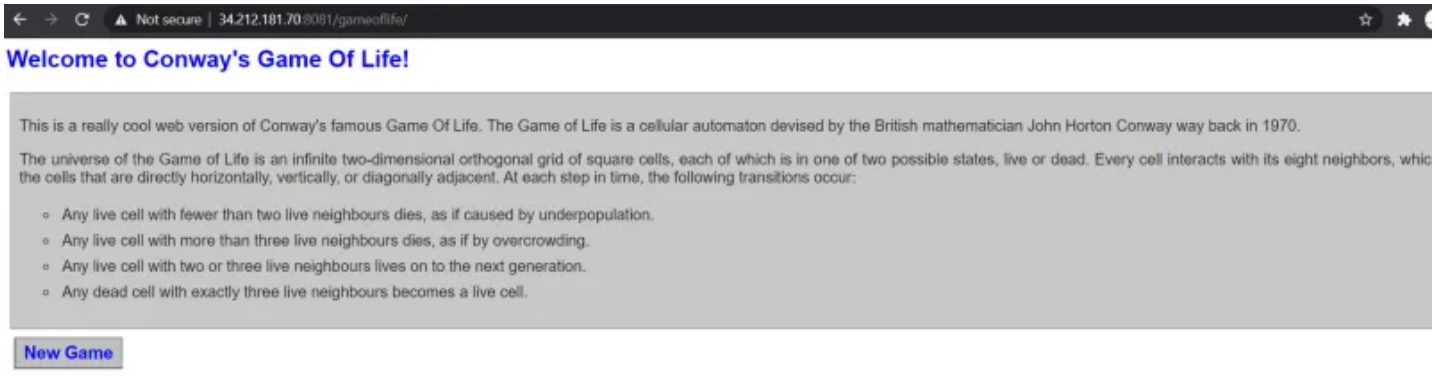






```
ubuntu@ip-172-31-13-154:~$ docker service ps gol-svc
ID                  NAME                IMAGE                                   NODE                DESIRED STATE
CURRENT STATE           ERROR               PORTS
k7jks7j1h7wd        gol-svc.1           qualitythought/gameoflife:07112020   ip-172-31-13-154    Running
Running 2 minutes ago
rt43jk6yitba        gol-svc.2           qualitythought/gameoflife:07112020   ip-172-31-2-74      Running
Running 2 minutes ago
ubuntu@ip-172-31-13-154:~$ docker service scale gol-svc=3
gol-svc scaled to 3
overall progress: 3 out of 3 tasks
1/3: running   [==================================================>]
2/3: running   [==================================================>]
3/3: running   [==================================================>]
verify: Service converged
ubuntu@ip-172-31-13-154:~$ docker service ps gol-svc
ID                  NAME                IMAGE                                   NODE                DESIRED STATE
CURRENT STATE           ERROR               PORTS
k7jks7j1h7wd        gol-svc.1           qualitythought/gameoflife:07112020   ip-172-31-13-154    Running
Running 2 minutes ago
rt43jk6yitba        gol-svc.2           qualitythought/gameoflife:07112020   ip-172-31-2-74      Running
Running 2 minutes ago
0s4cb1srxqbn        gol-svc.3           qualitythought/gameoflife:07112020   ip-172-31-14-247    Running
Running 8 seconds ago
ubuntu@ip-172-31-13-154:~$ docker service rm gol-svc
gol-svc
ubuntu@ip-172-31-13-154:~$ docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
ubuntu@ip-172-31-13-154:~$
```

```
ubuntu@ip-172-31-13-154:~$ docker service create --replicas 2 --name gol-svc  --publish published=8081,target=8080 qu
alitythought/gameoflife:07112020
xa49ksmjb2kfpub8vw0j4h77i
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged
ubuntu@ip-172-31-13-154:~$ docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE
PORTS
xa49ksmjb2kf        gol-svc             replicated          2/2                 qualitythought/gameoflife:07112020
*:8081->8080/tcp
ubuntu@ip-172-31-13-154:~$
```



- Generally docker services will be exposed to the outside world using load balancers.
- Experiment by doing
    - docker network ls (on manager and worker nodes)
- Some reference applications for conterization

    - openmrs
    - shopizer
    - nopcommerce (linux)

Topic:- Important points:-

- Our Linux operating system will store all docker related stuff in “var /lib/docker”
-