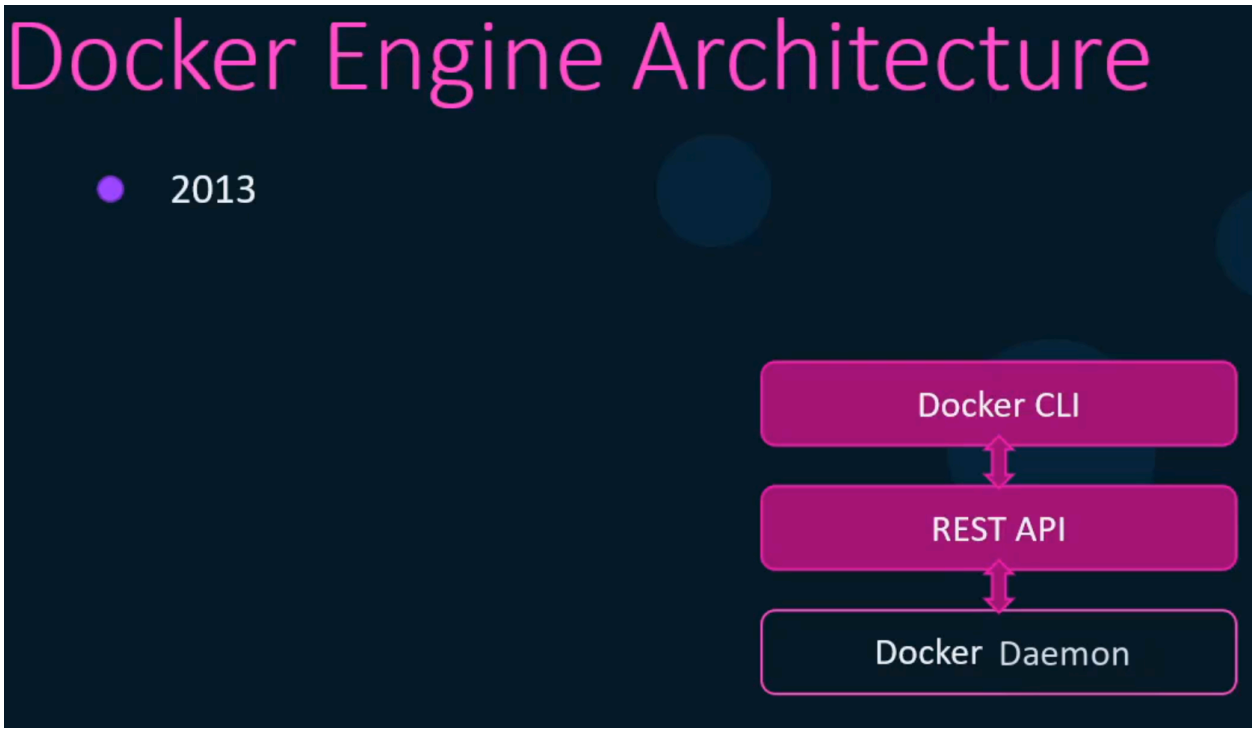
DOCKER CERTIFIED NOTES

1. Docker Engine:

Docker Engine Architecture:



Docker Engine consist of three components:
Docker CLI, REST API Server and Docker Daemon.

The Docker Daemon is the actual server or process that is responsible for creating and managing objects such as images, containers, networks, storage on a host.

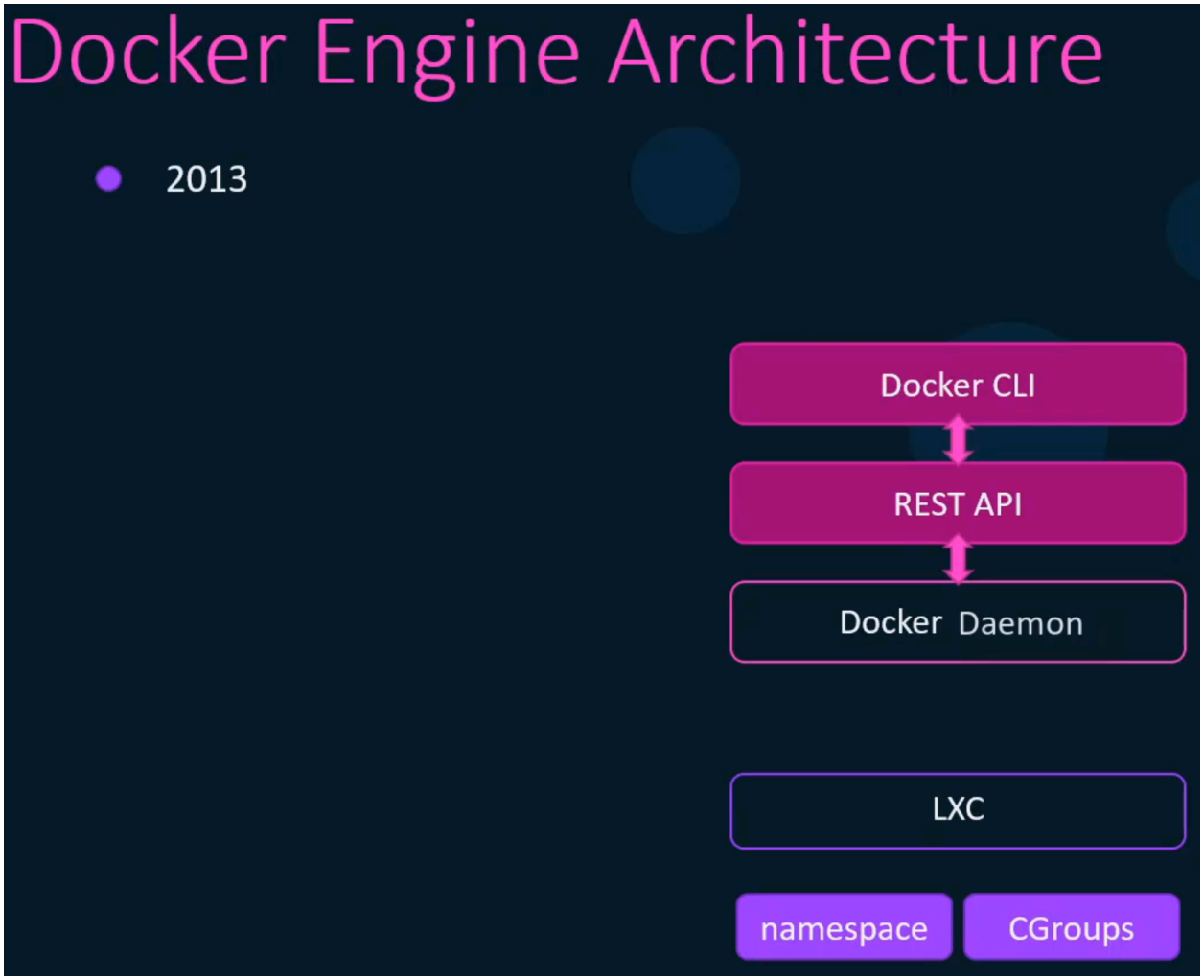
The REST API provides an interface to manage objects in Docker.

The Docker CLI is the command line interface we use to run commands to manage objects in Docker.

So how does Docker manage containers on the host?

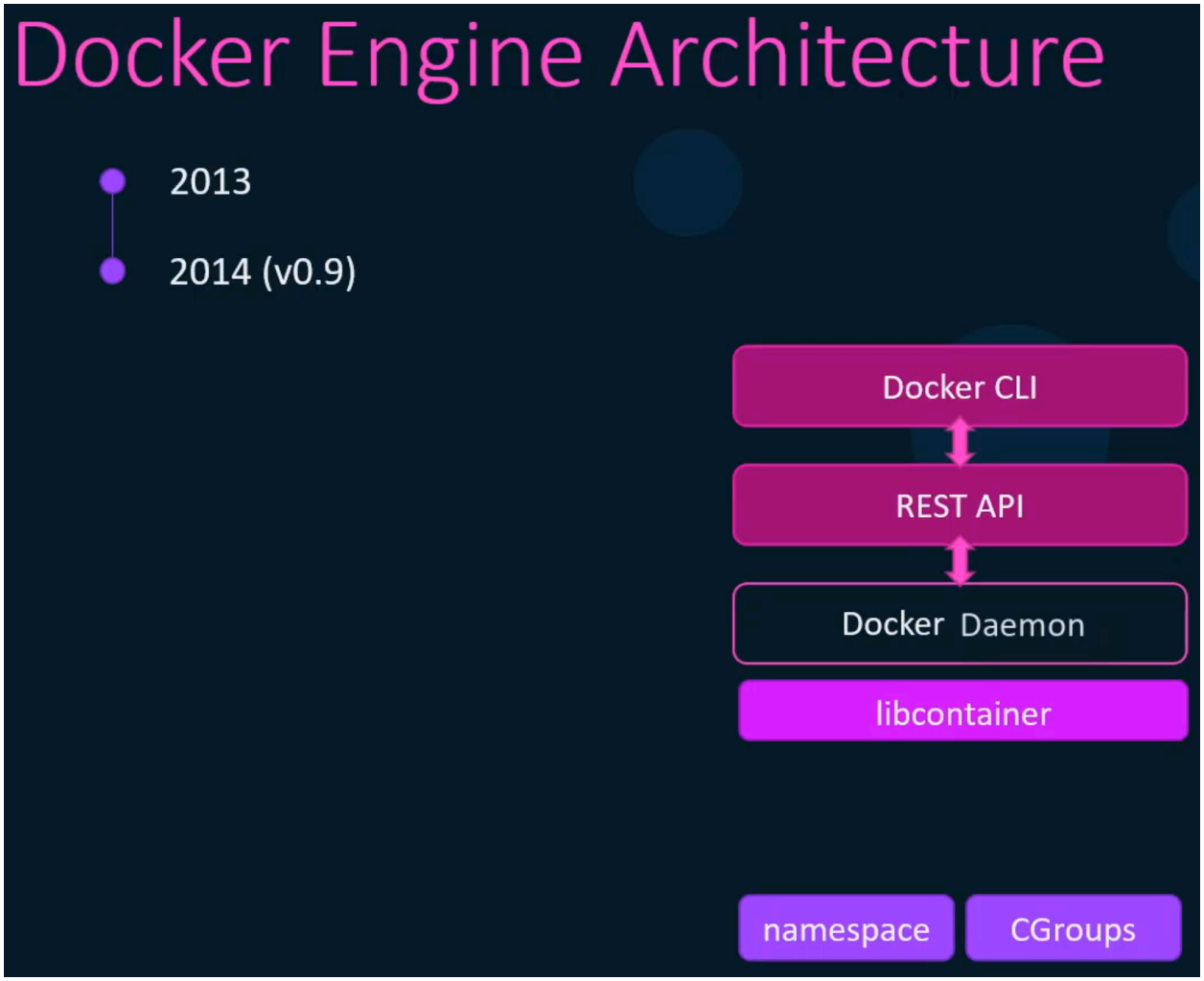
In the past Docker used a technology called Linux containers or LXC to manage containers on Linux. LXC used capabilities of Linux kernel such as **namespaces** and **cgroups** to create isolated environments on Linux known as containers.

It was hard to work directly with LXC so Docker provided a set of tools that made managing containers simpler.

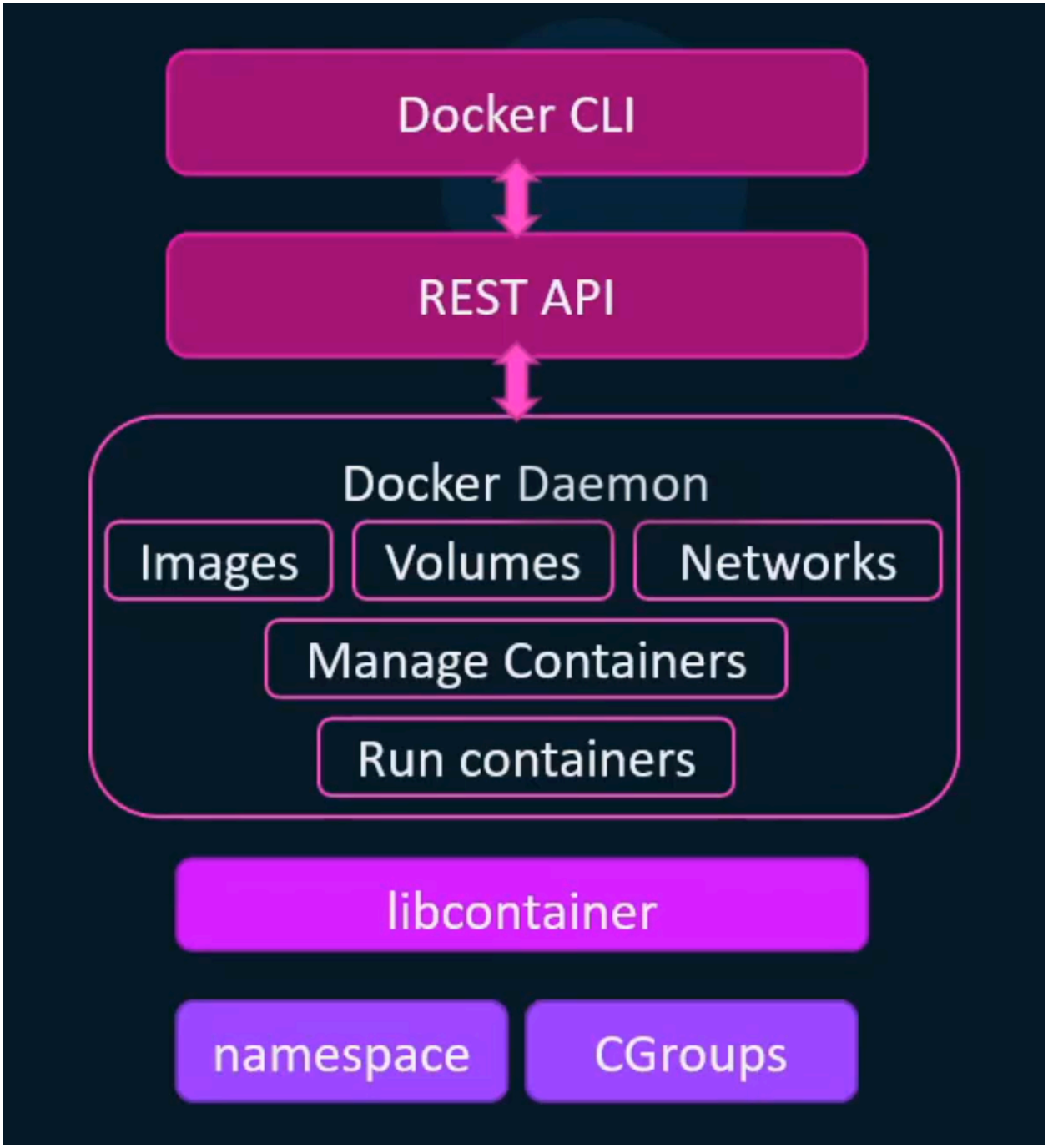


With the release of version 0.9, Docker introduced its own execution environment known as libcontainer.
Libcontainer is written in Go, the same programming language Docker is written in. This reduced Docker's dependency on the kernel's LXC technology.

With Libcontainer, Docker could now directly interact with the Linux Kernel features such as namespaces and cgroups and thus libcontainer replaced LXC as the default execution environment of Docker.



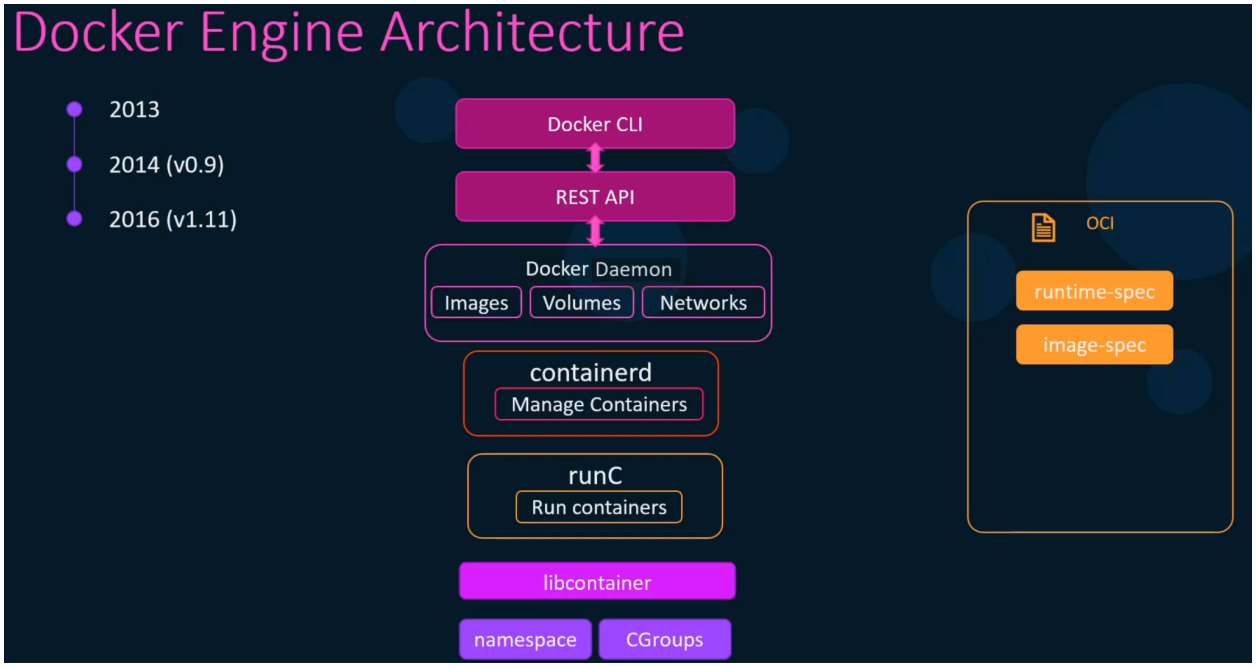
Docker Daemon was a single large monolithic code base that performed multiple functions such as running containers and managing containers and managing networks of volumes and images on the Docker host. As well as the mechanics required for pushing and pulling images from Docker repositories.



But with the version 1.11 the Open Container Initiative set standards on how we should run containers and work with images. They came up with two specifications: runtime-spec and image-sec that have been followed since then.

With the OCI standards in place, the architecture of Docker

Engine was refactored and broken down into smaller reusable components since version 1.11



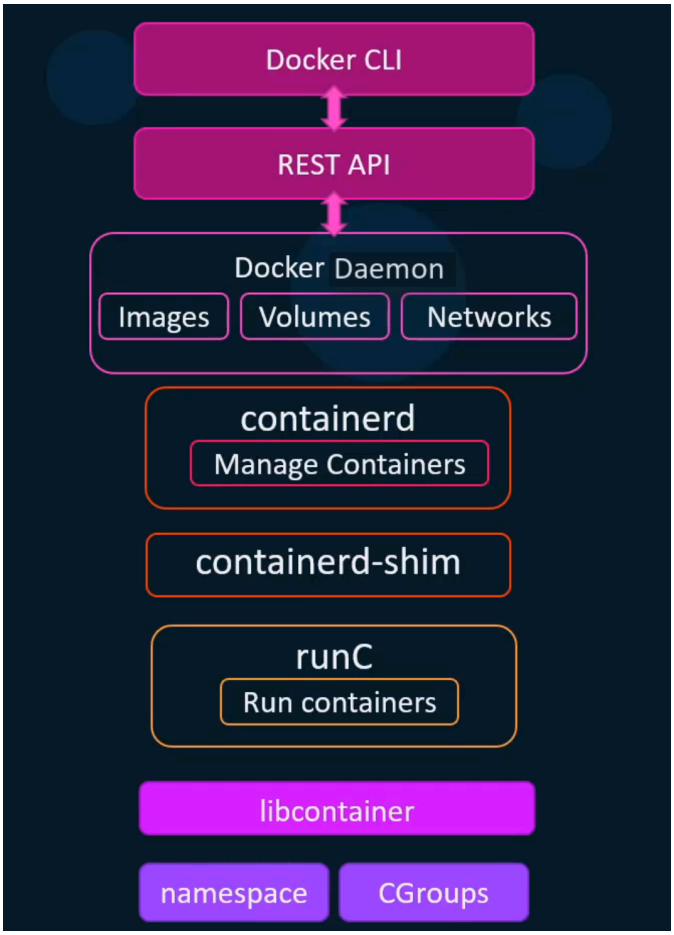
The part that ran containers became an independent component known as runC. runC was the first OCI based technology and was donated by Docker to the Open Compute Project Foundation. Now you could run containers by simply installing runC on your system without installing Docker at all.

The Daemon that managed containers became a separate component known as containerd. It now manages runC which in turn uses the libcontainer to create containers on a host.

Now, what happens when the daemon itself goes down or is restarted?
What happens to the containers and who takes care of the containers when the daemon itself is down?

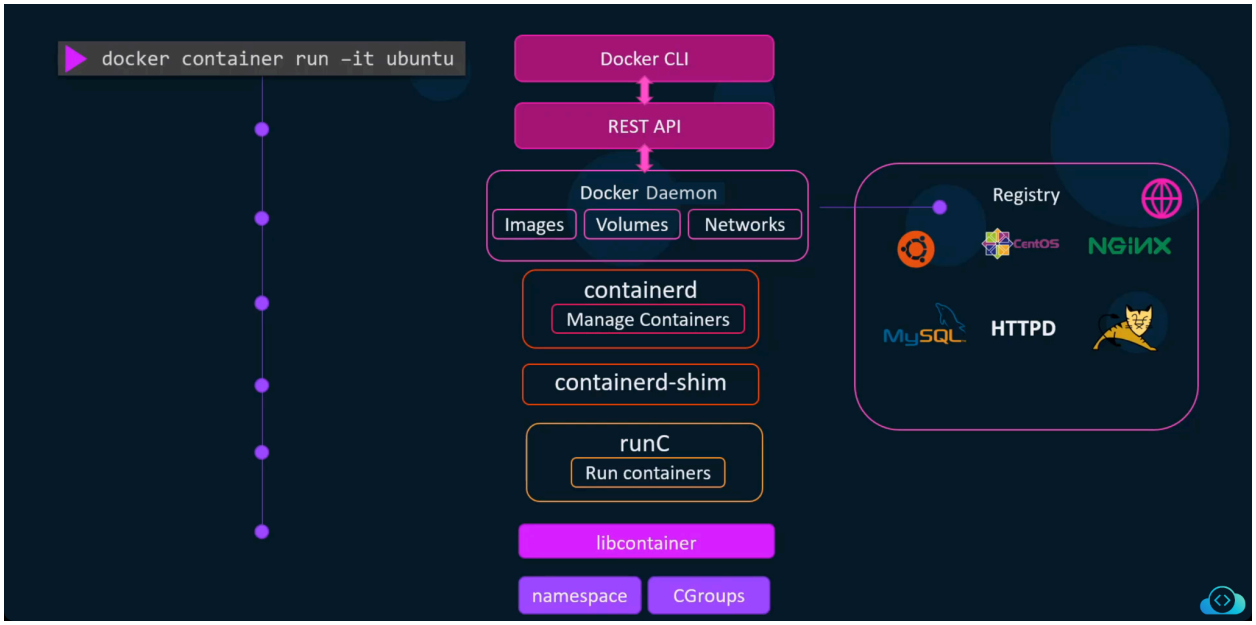
To handle this situation a new component named

containerd-shim was added to make containers daemon less. Instead of containerd creating containers, it's the containerd-shim that does it and monitors the state of the container.



Even if the Docker Daemon is shut down or restarted, the containers run in the background and are attached back when the daemon comes back up.
So to put it all together, when we execute a command using the Docker CLI, to create a container, the Docker Client converts the command into the RESTful API, which is then passed to the Docker Daemon. The Docker Daemon on receiving the request, first checks if the image provided is already available on the local system. If it is not, image is downloaded from the registry.

The default being Docker Hub, once the image is downloaded, it makes a call to the containerd to start the container.



Containerd is responsible for converting the image that was downloaded into an OCI compliance bundle. It then passes the bundle to containerd-shim, which in turn calls runC to start the container. runC interacts with the namespaces and cgroups on Linux Kernel to create a container and that's how a container gets created.

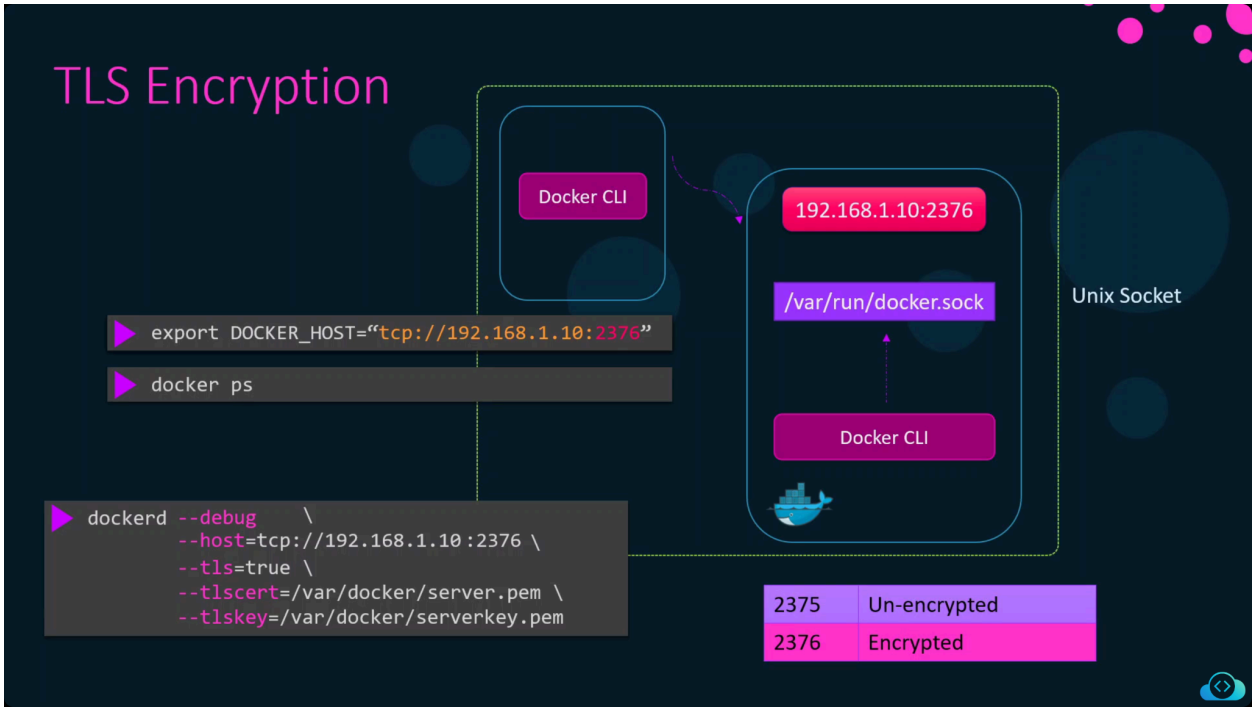
Connecting to Docker externally:

Docker accepts TCP traffic on the port 2375 (non encrypted) and 2376 (encrypted), so to run Docker externally from Docker CLI of another application we can do it using the command:

```
$ dockerd --debug --host=tcp://192.168.0.1:2375
```

OR IN ENCRYPTED WAY:

```
$ dockerd --debug --host=tcp://192.168.0.1:2376 --tls=true
--tlscert=/var/docker/server.pem
--tlskey=/var/docker/serverkey.pem
```



Newer Docker Container Commands:

```
$ docker container create nginx
```

To create a container without running it.

```
$ docker container ls
```

To list all the Docker Containers.

```
$ docker container ls -a
```

To list all the Docker Containers whether running or stopped.

```
$ docker container ls -l
```

To show the last created Container.

```
$ docker container ls -q
```

To show the list of short Container ID of running containers.

```
$ docker container ls -aq
```

To show the list of short Container ID of running and stopped containers.

```
$ docker container start {container_id}
```

To start an already created container.

```
$ docker container run ubuntu
```

This will create and start the container which combines the two commands.

***Note: All the options with the run command like -i, -t, -d,**

--name etc also apply here.

```
$ docker container rename webapp custom-webapp
```

To change the container name from webapp to custom-webapp.

```
$ docker container attach {container_id}
```

To re-attach to the container after running it in detached mode.

```
$ docker container run -it --name=ubuntu ubuntu
```

After running a container in interactive mode you can detach from the container without stopping it using the command: **"Ctrl + P + Q"**

```
$ docker container attach ubuntu
```

To attach back to the container where you left off.

```
$ docker container exec {container_id} {commands}
```

To execute the command on the detached container.

```
$ docker container inspect {container_name}
```

To inspect in depth detail of the container.

```
$ docker container stats
```

It will provide details about all the resources being used by the containers like CPU%, Memory Usage, Memory Limit, Network I/O etc.

```
▶ docker container stats
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK
PIDS
59aa5eacd88c        webapp              50.00%              400KiB / 989.4MiB     0.04%               656B / 0B           0B /
a00b5535783d        epic_leavitt        0.00%               404KiB / 989.4MiB     0.04%               656B / 0B           0B /
616f80b0f026        elegant_cohen       0.00%               404KiB / 989.4MiB     0.04%               656B / 0B           0B /
36a391532e10        charming_wiles      0.01%               8.363MiB / 989.4MiB   0.85%               656B / 0B           0B /
```

```
$ docker container top {container_name}
```

To get details about the processes running inside the container and their details like PID, PPID, TTY, TIME, CMD etc.

```
▶ docker container top webapp
UID                 PID                 PPID                C                   STIME               TTY                 TIME                CMD
root                17001               16985               0                   13:23               ?                   00:00:00            stress
```

```
$ docker container logs {container_name}
```

To get logs of the container.

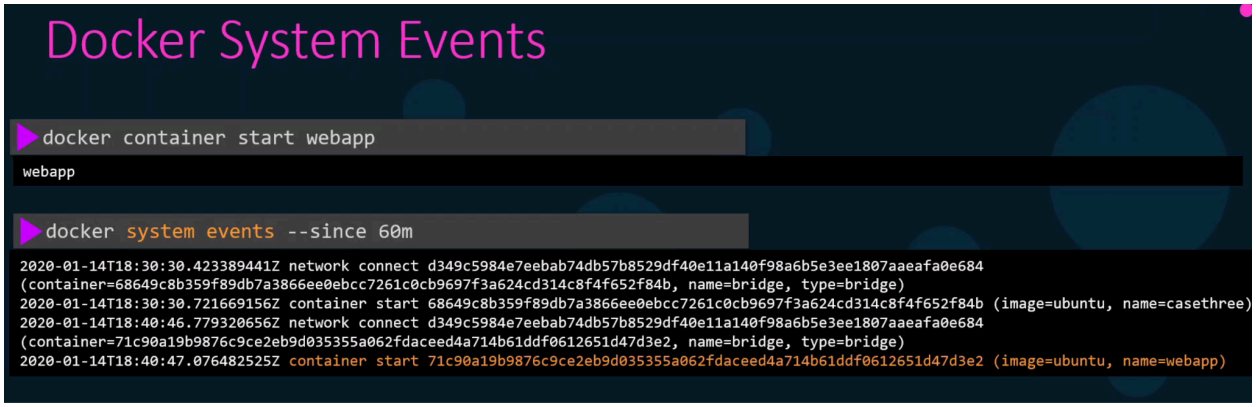
```
▶ docker container logs logtest
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.6. Set the 'ServerName'
directive globally to suppress this message
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.6. Set the 'ServerName'
directive globally to suppress this message
[Tue Jan 14 13:38:15.699310 2020] [mpm_event:notice] [pid 1:tid 140610463122560] AH00489: Apache/2.4.41 (Unix) configured --
resuming normal operations
[Tue Jan 14 13:38:15.699520 2020] [core:notice] [pid 1:tid 140610463122560] AH00094: Command line: 'httpd -D FOREGROUND'
```

```
$ docker container logs -f {container_name}
```

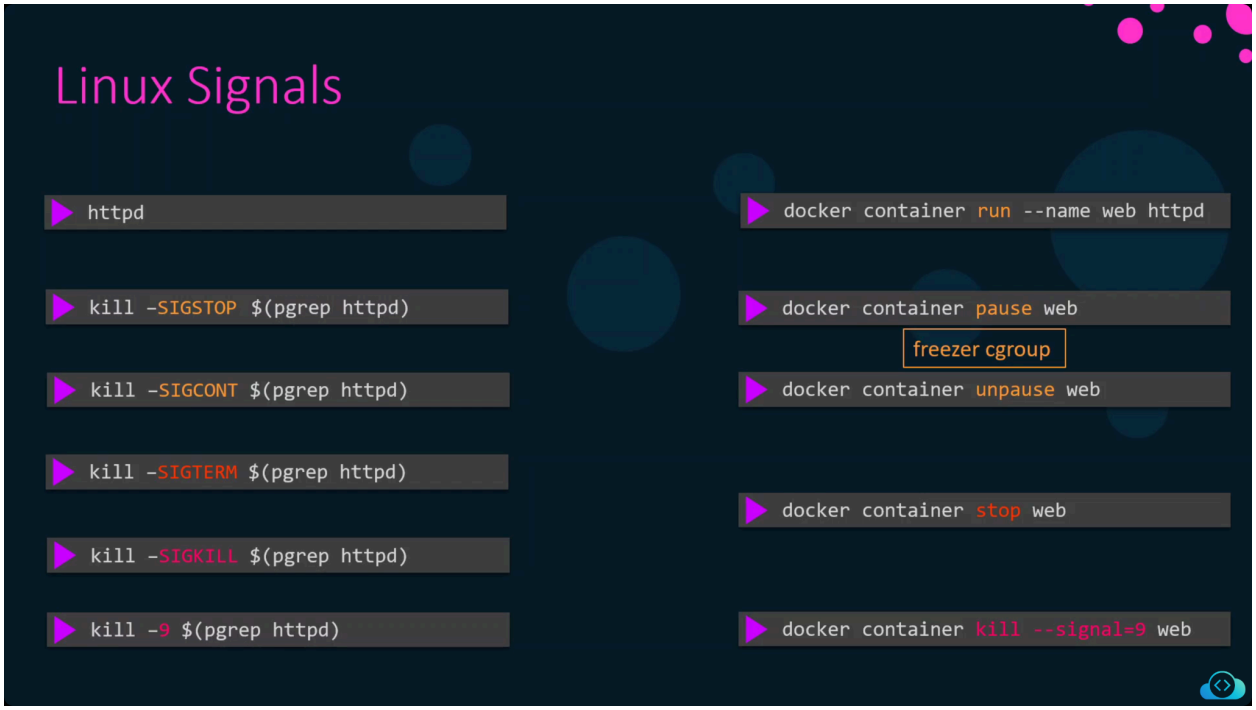
To stream live logs of the application.

```
$ docker system events --since 60m
```

To get all the changes made to the container, the networks related to the container, the storage, etc. All the details in the last 60 mins will be displayed on the screen.



Now let's look at the way to pause, continue, stop and kill a container:



The mechanism uses Linux Signals in the background.

As you can see we can pause the container using the following command:

```
$ docker container pause web
```

We can continue the container using the following command:

```
$ docker container unpause web
```

But there is an issue with this as Docker can't guarantee if the containers will function as expected so it uses freeze cgroups to ensure that they do.

```
$ docker container stop web
```

This command will first pass SIGSTOP to the process letting them know that they need to perform clean up activities and the application is about to shut off. It then sends the SIGKILL to the process just in case if the process didn't stop after receiving the SIGSTOP command.

```
$ docker container kill --signal=9 web
```

You can also pass a specific Signal (9 in above example) in the command to let Docker know that they have to pass this particular signal to the process in the backend.

```
$ docker container rm {container_name}
```

To remove a stopped container from the system to free up the space. You have to stop the container and only then you will be able to remove it.

```
$ docker container stop $(docker container ls -a)
```

To stop all running containers.

```
$ docker container rm $(docker container ls -aq)
```

To remove all the stopped containers.

```
$ docker container prune
```

Also another way to remove all the stopped containers.

```
docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
59aa5eacd88c
a00b5535783d
616f80b0f026
36a391532e10
Total reclaimed space: 1223423
```

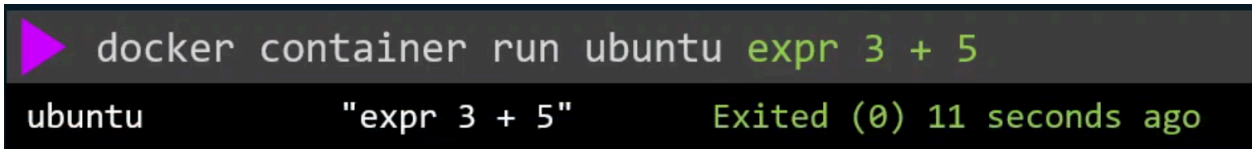
```
$ docker container run -d --name=webapp --hostname=webapp
webapp
```

You can set the hostname for a container using this command.

Restart Policies of Docker:

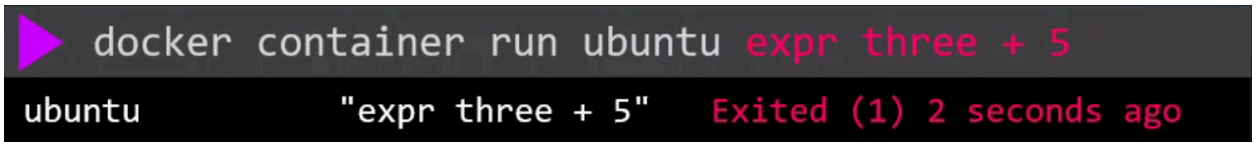
A Docker Container may stop due to various reasons, One is when the job it is supposed to do ends successfully.

For example:

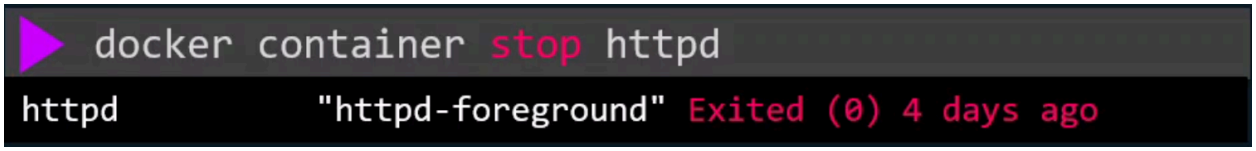


The container was to perform a mathematical calculation and as soon as it does that it stops. The exit code is 0.

A container may stop due to a failure. For instance here is an incorrect parameter thats passed, so it ends with an exit code 1:

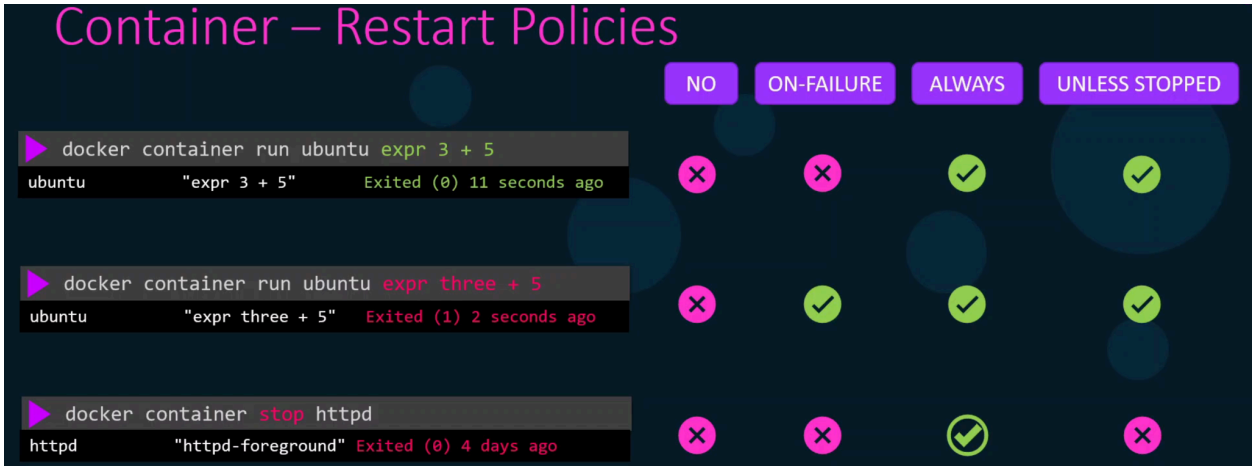


This is a failure.
A running container can also be stopped manually by running the docker container stop command. It first sends a SIGSTOP signal then a SIGKILL signal and results in a forceful termination scenario.



We can configure the container to restart in any of these cases.

There are 4 allowed options to pass as restart parameter for a container while running it, they are explained below:



As you can see in the above image, if you pass “no” as an argument then it won’t restart in any of the scenarios. If you pass “on-failure” argument then it will only restart when it encounters exit code as 1. If you pass “always” argument it will restart in all scenarios but there is a catch, it won’t automatically restart on manual termination case, it will only restart when the Docker Daemon restarts as the DevOps Engineer must have a good reason to manually terminate the container. If you pass an “unless-stopped” argument then it will not restart in the manually terminated scenario.

```
$ docker container run --restart=no ubuntu
```

```
$ docker container run --restart=on-failure ubuntu
```

```
$ docker container run --restart=always ubuntu
```

```
$ docker container run --restart=unless-stopped ubuntu
```

There is also an option to allow your containers to run even after the Docker Daemon is stopped. It can be achieved in the following way:

/etc/docker/daemon.json

```
{
  "debug": true,
  "hosts": ["tcp://192.168.1.10:2376"],
  "live-restore": true
}
```

You have to add a property “live-restore” inside the daemon.json with value true inside the /etc/docker folder.

```
$ docker container run -it --name=ubuntu-test ubuntu
```

```
$ systemctl stop docker
```

```
$ systemctl start docker
```

```
$ docker container attach ubuntu-test
```

You will still be able to connect to the container after the Docker Daemon is back up and get back to the same state of the container before the Docker Daemon was shut down.

Copying Files from and to Container:

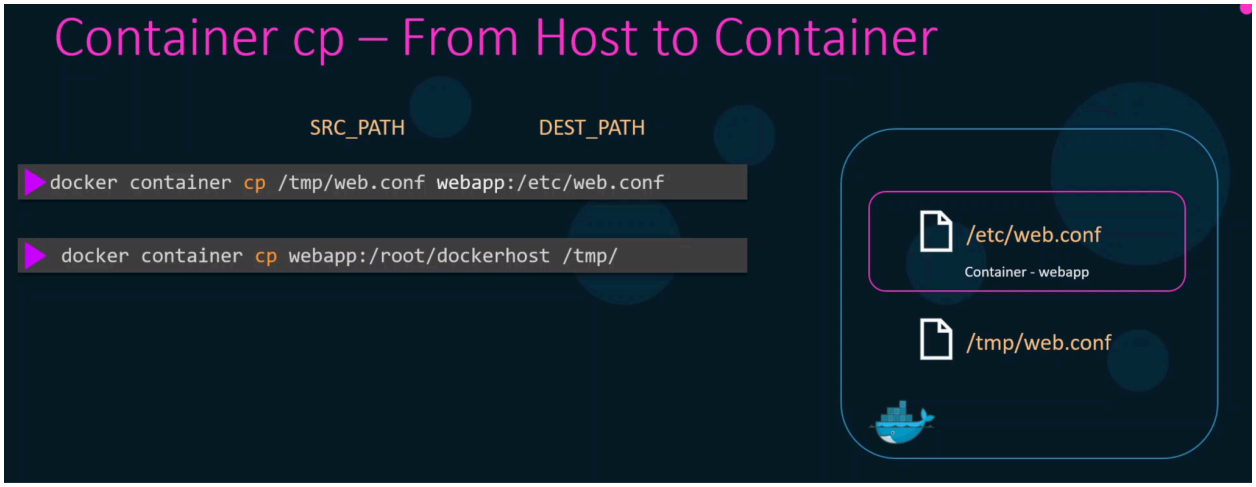
Copying files from Host to Container and from Container to

host is possible using the following command:

```
$ docker container cp {src/path} {destination/path}
```

For example:

```
$ docker container cp /temp/web.conf webapp:/etc/web.conf
```



2. Docker Image Management:

Searching Images on CLI:

You can run the following commands to search for images on the DockerHub before pulling them or directly running a container on them:

```
$ docker search {image_name}
```

```
$ docker search {image_name} --limit 2
```

```
$ docker search --filter stars=10 {image_name} --limit 2
```

```
$ docker search --filter stars=10 --filter is-official=true
{image_name}
```

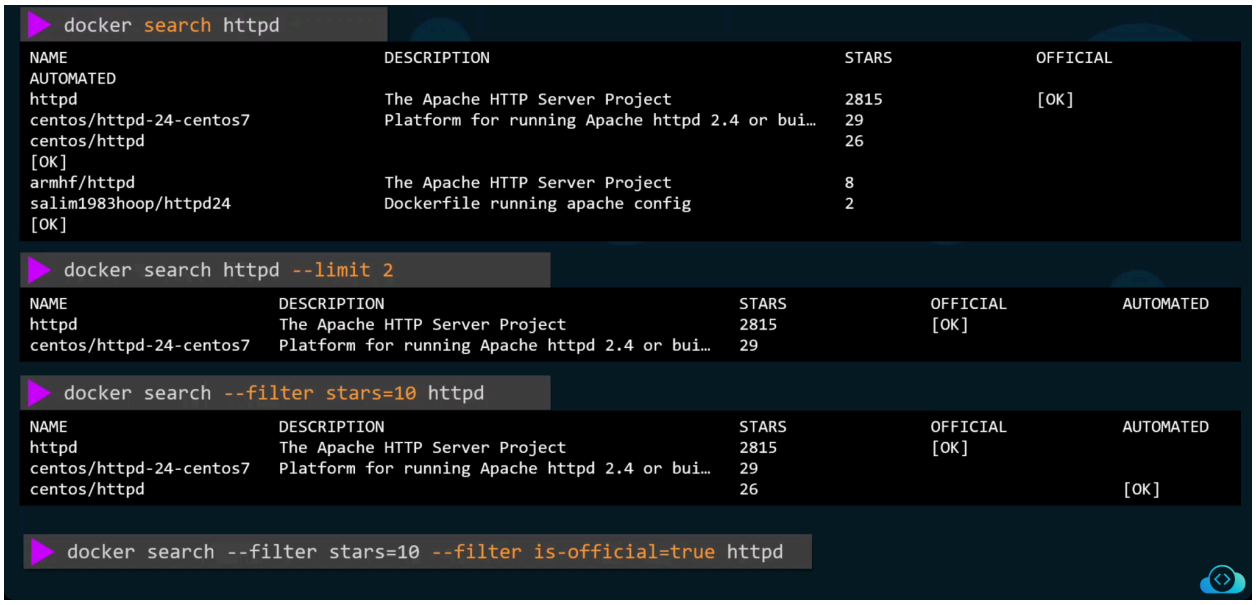
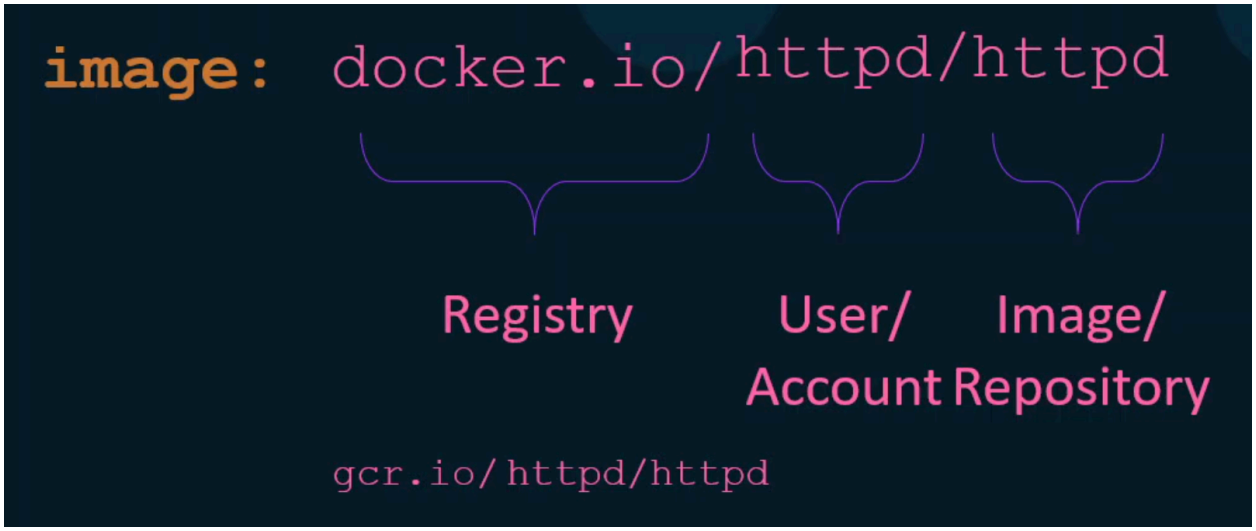


Image Addressing Conventions:

```
$ docker image pull httpd
```



If we don't provide the User Account in the image name then it is assumed to be the same as the image repository name. Which is true in the case of the "httpd" image.
DockerHub (docker.io) is also the default registry the docker looks into while searching for an image.

If you want to get an image from a specific vendor then you have to type the complete name with the registry as well:

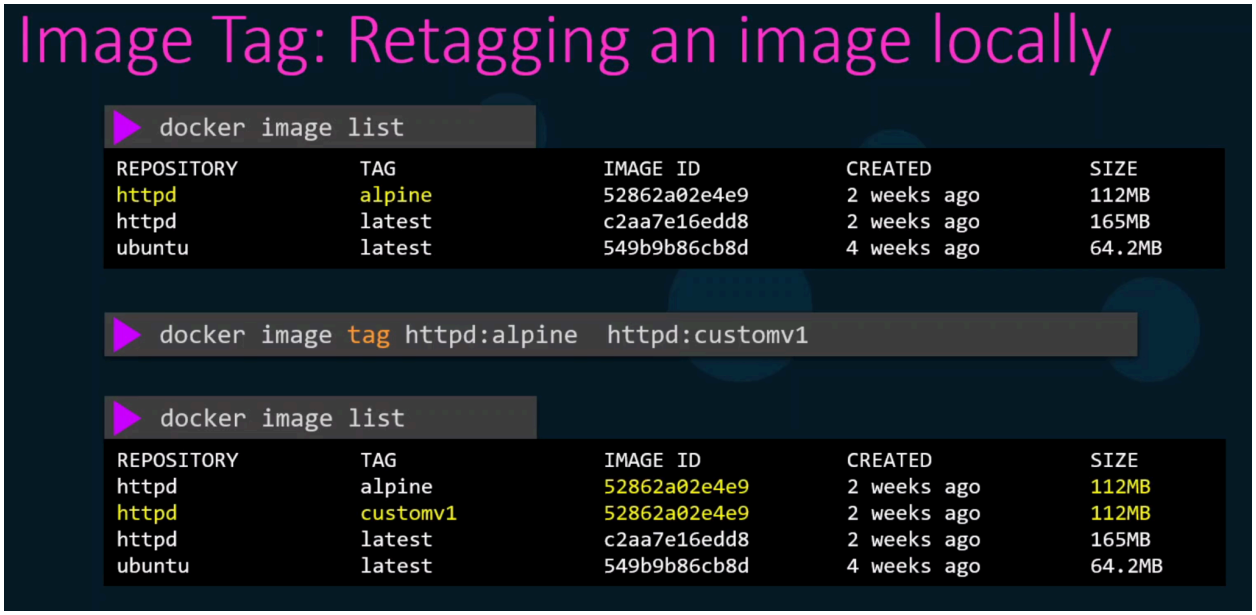
```
$ docker image pull gcr.io/httpd/httpd
```

***Note: You will have to login to pull images from a certain registry (grc.io). Only the DockerHub registry is public so you can pull images from there without any issues.**

Image Tagging:

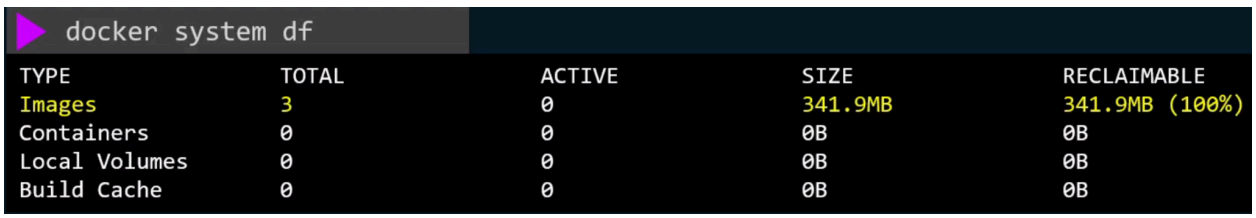
If you want to create a soft copy of an image you can do it using the following command:

```
$ docker image tag {original_image_name}
{tagged_image_name}
```



You can also view system resources information using:

```
$ docker system df
```



As you can see, docker is smart enough to not show tagged images as a unique image and only shows 3 images instead of 4.

Removing images:

```
$ docker image rm
```

You cannot remove images that have containers running on them. You have to stop those containers first and only then you can remove the image.

***Note: If you remove an original image that has tagged images to itself then docker will first remove the tag from the tagged image and then remove the original image you have selected but the tagged images will still exist without the original image.**

```
root@fkyounvidia:/home/gaurav# docker image list
REPOSITORY   TAG       IMAGE ID       CREATED        SIZE
httpd        latest    4ce47c750a58   5 months ago   147MB
root@fkyounvidia:/home/gaurav#
root@fkyounvidia:/home/gaurav#
root@fkyounvidia:/home/gaurav# docker image tag httpd mynewhttpd
root@fkyounvidia:/home/gaurav#
root@fkyounvidia:/home/gaurav#
root@fkyounvidia:/home/gaurav# docker image list
REPOSITORY   TAG       IMAGE ID       CREATED        SIZE
mynewhttpd   latest    4ce47c750a58   5 months ago   147MB
httpd        latest    4ce47c750a58   5 months ago   147MB
root@fkyounvidia:/home/gaurav#
root@fkyounvidia:/home/gaurav#
root@fkyounvidia:/home/gaurav# docker image rm httpd
Untagged: httpd:latest
Untagged: httpd@sha256:72f6e24600718dddef131de7cb5b31496b05c5af41e9db
root@fkyounvidia:/home/gaurav#
root@fkyounvidia:/home/gaurav#
root@fkyounvidia:/home/gaurav# docker image list
REPOSITORY   TAG       IMAGE ID       CREATED        SIZE
mynewhttpd   latest    4ce47c750a58   5 months ago   147MB
```

If you want to remove all the images that don't have containers running on them you can use the following command:

```
$ docker image prune -a
```

You can use the following command to inspect an image in detail:

```
$ docker image inspect {image_name}
```

For example:

```
$ docker image inspect httpd
```

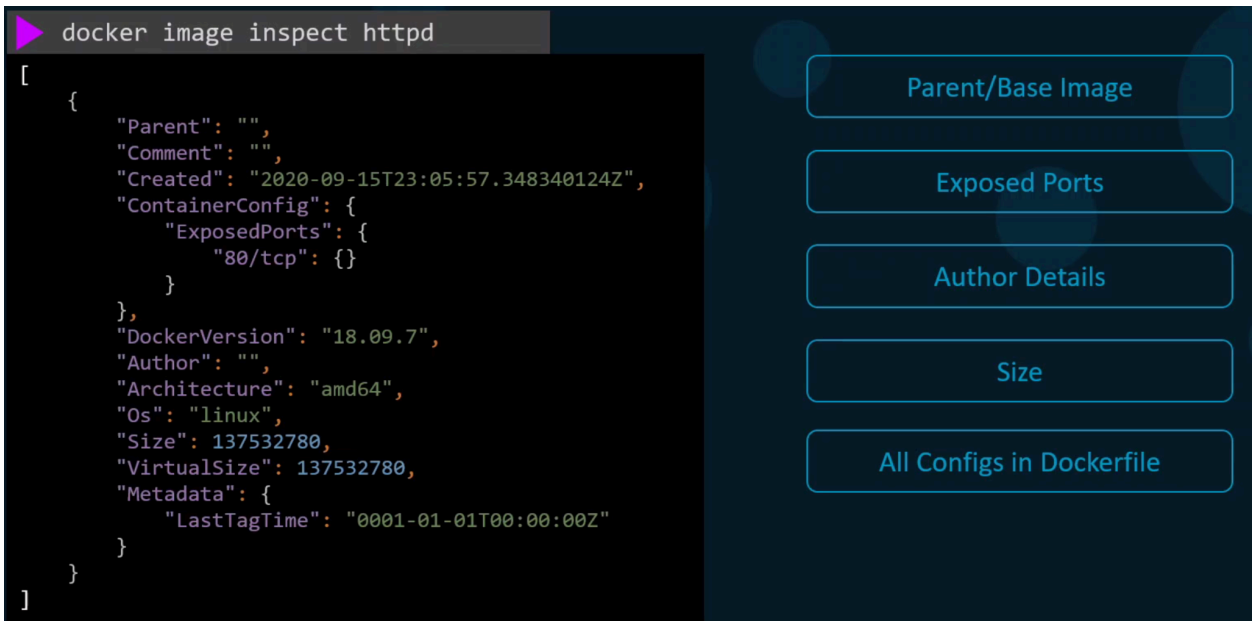


Image Save and Load (Alternative to Pull):

If you are working on a system that doesn't allow internet access due to security reasons, you are unable to pull images from DockerHub Registry. In that case you can use the following method:

Step 1: Pull the image on a different machine.

```
$ docker image pull httpd:latest
```

Step 2: Create a tar file of that pulled image.

```
$ docker image save httpd:latest -o httpd.tar
```

Step 3: Transfer the tar file to the restricted environment.

Step 4: Now load the image using the tar file.

```
$ docker image load -i httpd.tar
```

```
$ docker image list
```

Now you will be able to view the image on your system.

Image Import and Export (For Containers):

You can make an image from a running Container using import and export commands:

```
$ docker export {container_name} > testimage.tar
```

This will create a tar file of the running container that can be used to import an image of that container.

```
$ docker image import testimage.tar newimage:latest
```

3. Docker Engine - Security:

4. Docker Engine - Networking:

5. Docker Engine - Storage:

6. Docker Compose:

7. Docker SWARM:

8. Kubernetes:

9. Docker Engine Enterprise:

10. Docker Trusted Registry:

11. Docker Recovery: