Docker

Before Docker:

→ Initially only 1 app use to run one one server

→ So, if you want add an app, you basically have to add a server

→ VMWARE solves this problem by introducing Virtual Machines.

→ VM's allows us to run multiple apps on same server

→ Limitations: They occupies their own OS.

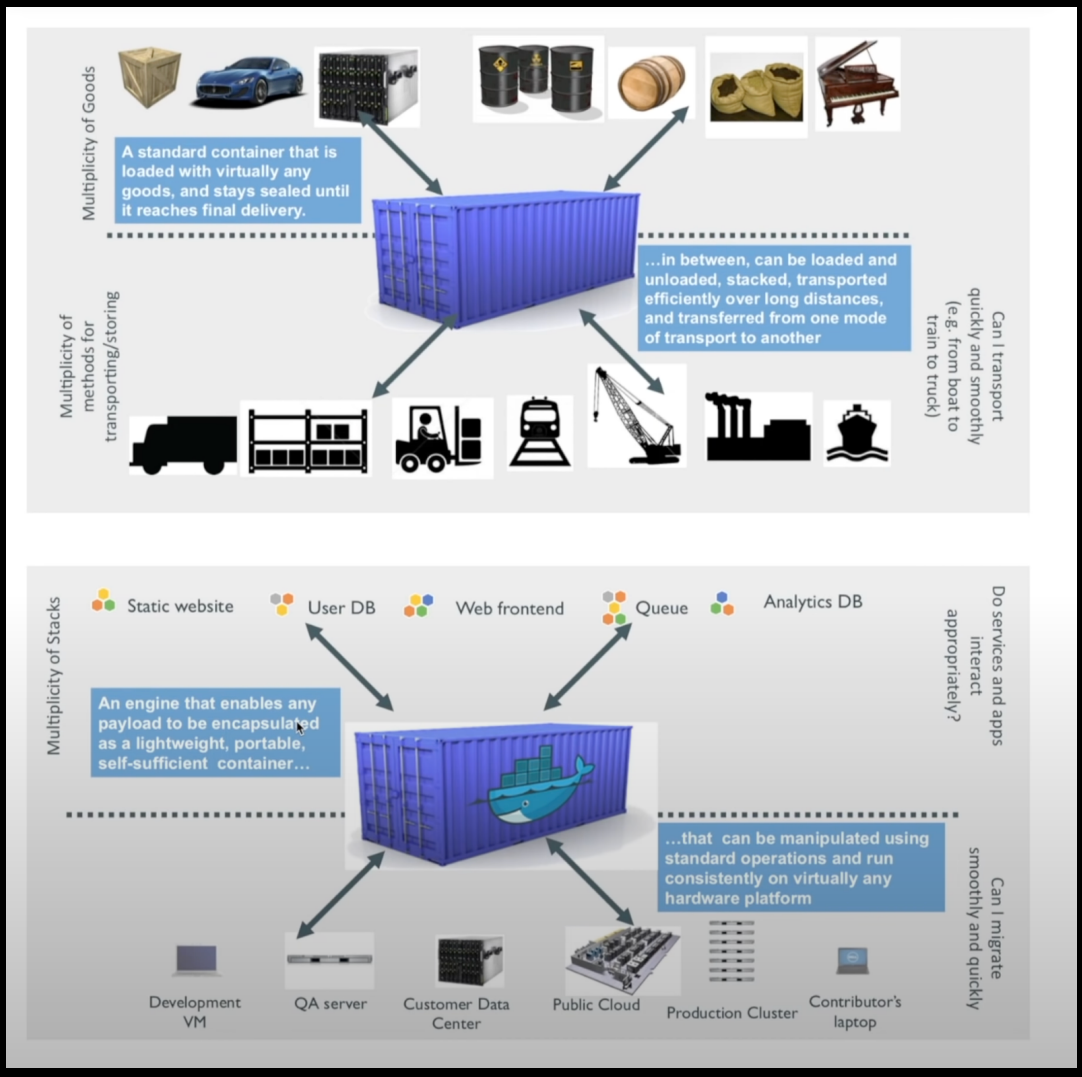
→ Migration is time Consuming

→ "It's work on my machine"!

When you try to send your code to run on your friend's Comp but unable to due to module version & so on

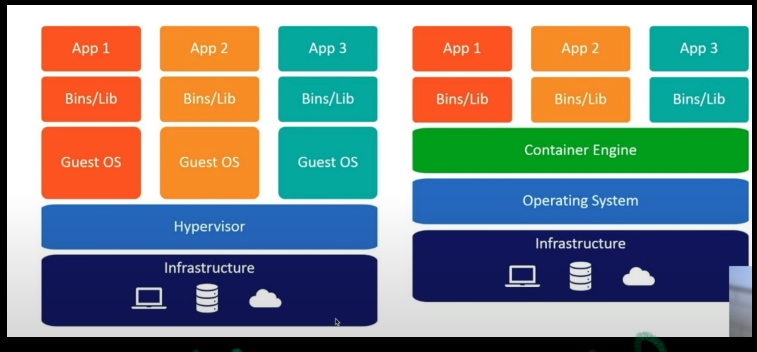
Containers :

→



→ Like on the above example we you want transport to diff Country how all the Box, Piano, Car are put in 1 Box & Ship it

→ Similary you put the Code, depending everything required to run the code you put in Box & give to friend now it will make sure it will run in your friend Computer.



VM Containers

* Hypervisor is used to Create multiple VMs on the host OS & it manages the VM.
→ Every OS will have dedicated hardware

⇒ Containers have one OS only & you have a Container Engine.

⇒ Running Docker

⇒ Docker windows Containers won't work on docker linux. So, you have to install image.

windows: Docker desktop + WSL

MAC: VMs & Docker Desktop

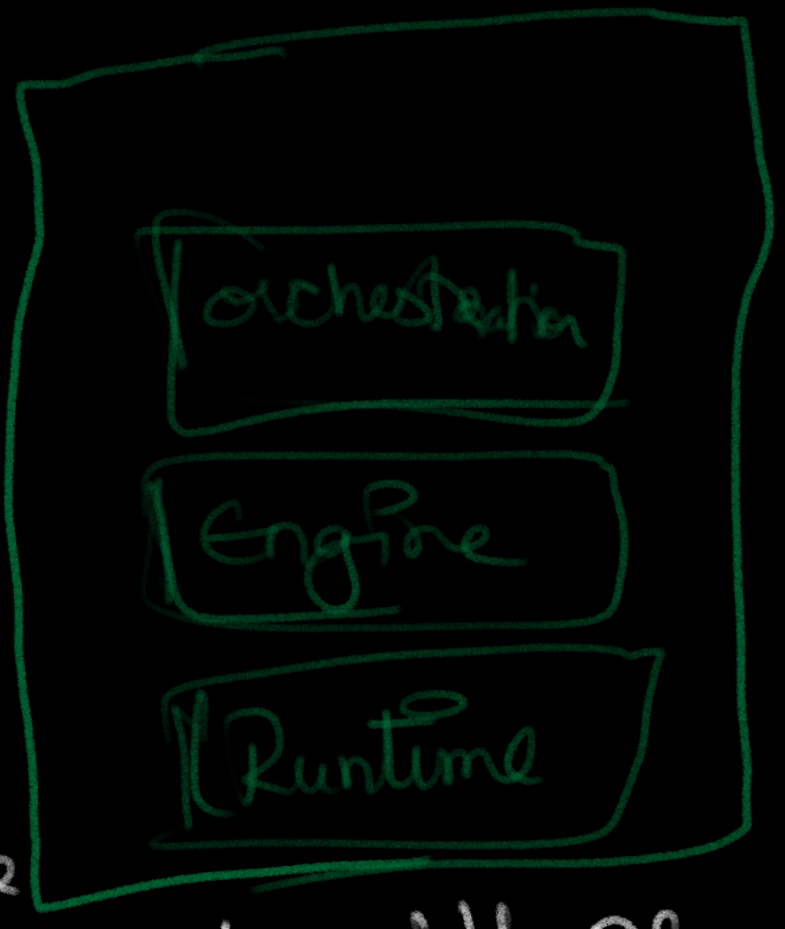
Linux: As it is

What is Docker?

Docker is a Container platform, that allows us to build, test & deploy apps quickly.

⇒ So, far we know Docker is helping us run in an isolated environment.

⇒ It means container doesn't know what's happening outside even in the OS.



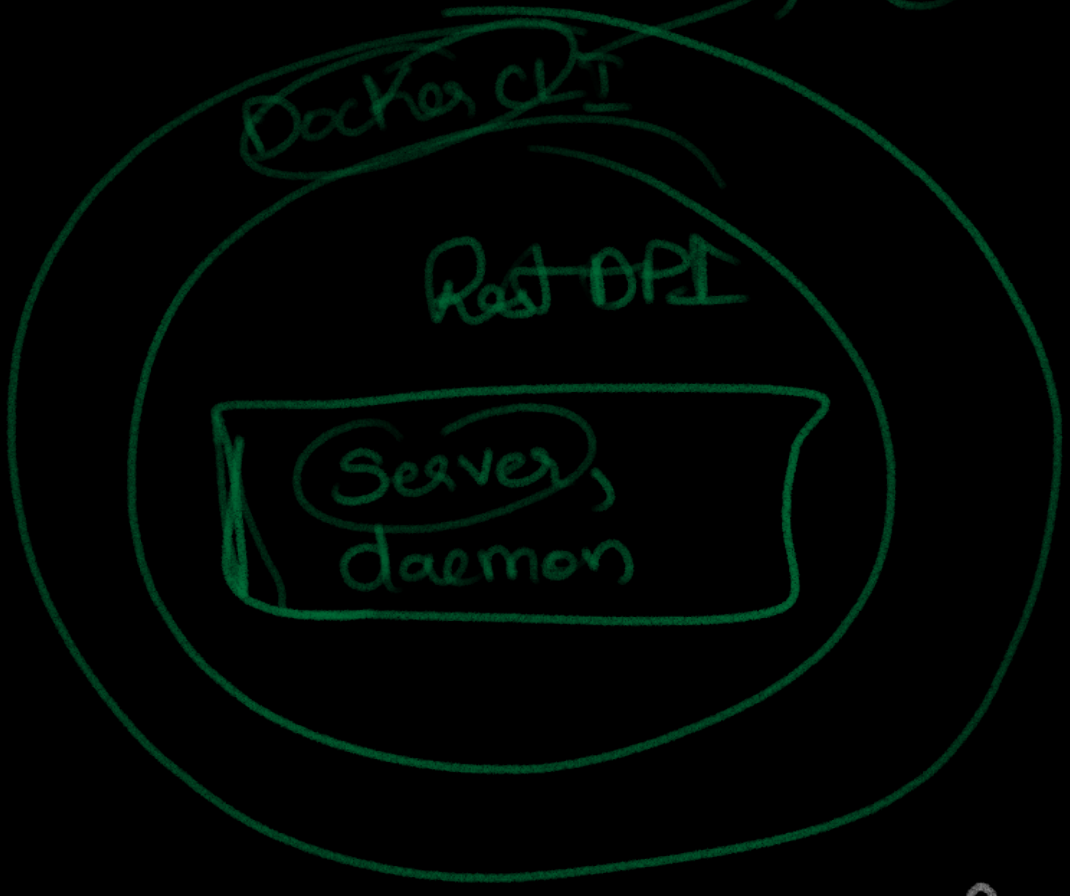
1. Runtime:

It allows us to start & stop the Containers

- Runc
  - ↳ low level runtime
  - ↳ role is to work with OS, start & stop the Containers.
- Containerd
  - ↳ managing runc & Containers
  - ↳ pulling the Images, get data into network.

2. Docker Engine

→ Docker Daemon



→ So, using docker CLI & Rest API, can make an API call to daemon, daemon will work with Containers, images & do it.

→ Uses client-server Architecture

3. Orchestration

Let's say an app is running 10K Containers & you want to scale it to 20K.

→ Assume 500 Containers got expired & you are going to restart

→ It allows us to manage Containers

Container Image

→ Let's say you are making a dish eating something ex: noodles & your friend lives in diff city & you can't just ship noodle soup in Box to your friend when you are far away. It doesn't make sense to ship the running App

⇒ Instead we can share the instructions / receipe.

⇒ Similarly, You can send the entire instruction with src code for website & tell your friend run it.

⇒ This instruction part, you follow receipe the dish will be created dish will be the actual app running in servers or Containers.

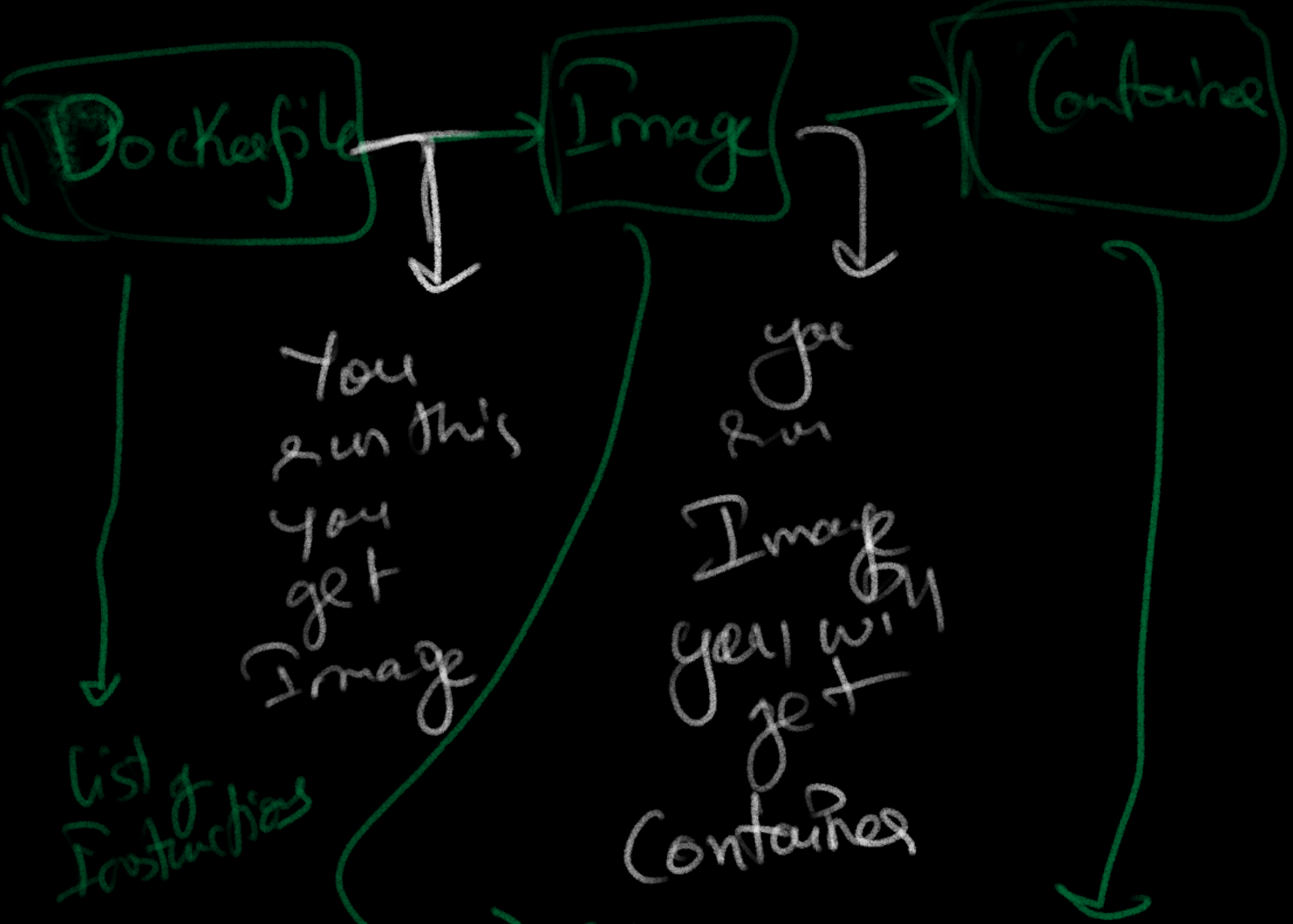
⇒ The thing that leads us to create the dish is known as Docker / Container Image

---

⇒ So, if you want share your app make an docker image & send it & the others can run.

→ Container is a running instance of an Image.

→ You can create docker Images using Dockerfile
↳ Recipe.



→ There are many runtimes, so, how can we choose then they come up with an initiative known as Open Container Initiative

→ OCI contains 2 specs:

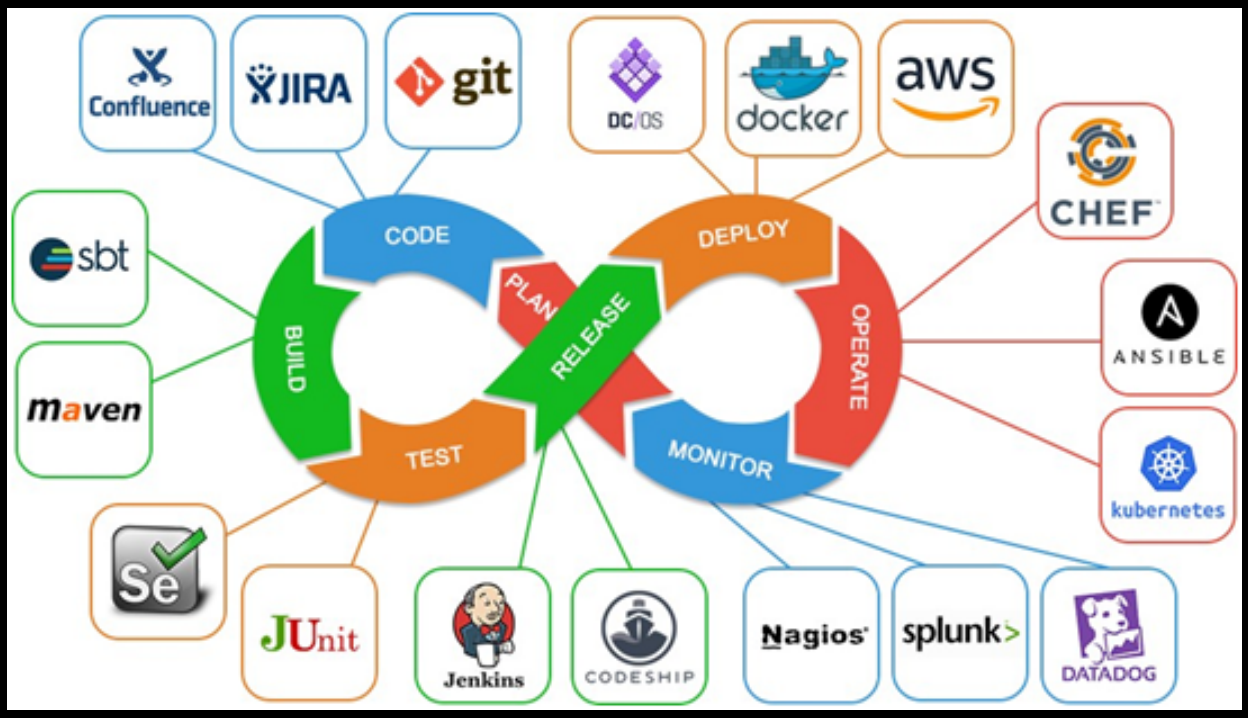
1. Runtime spec
2. Image spec

What are the various Container Runtimes?

Some of the the popular container runtimes are below ones:

- **containerd**: A CNCF project, it manages the complete container lifecycle of its host system that includes image management, storage and container lifecycle, supervision, execution and networking.
- **lxc**: LXC provides OS level virtualization through a virtual environment that has its own process and network space, it uses linux cgroups and namespaces to provide the isolation.
- **runc:** `runc` is a CLI tool for spawning and running containers according to the OCI specification. It was developed by Docker Inc and donated to OCI as the first OCI runtime-spec compliant reference implementation.
- **cri-o:** CRI-O is an implementation of the Kubernetes CRI (Container Runtime Interface) to enable using OCI (Open Container Initiative) compatible runtimes. It is a lightweight alternative to using Docker as the runtime for Kubernetes.
- **rkt:** rkt is an application container engine developed by

DevOps



⇒ Development & operations

Dev OPS

App
Dockerfile
Image
Container

Download Image,
run Image,
Creating Image

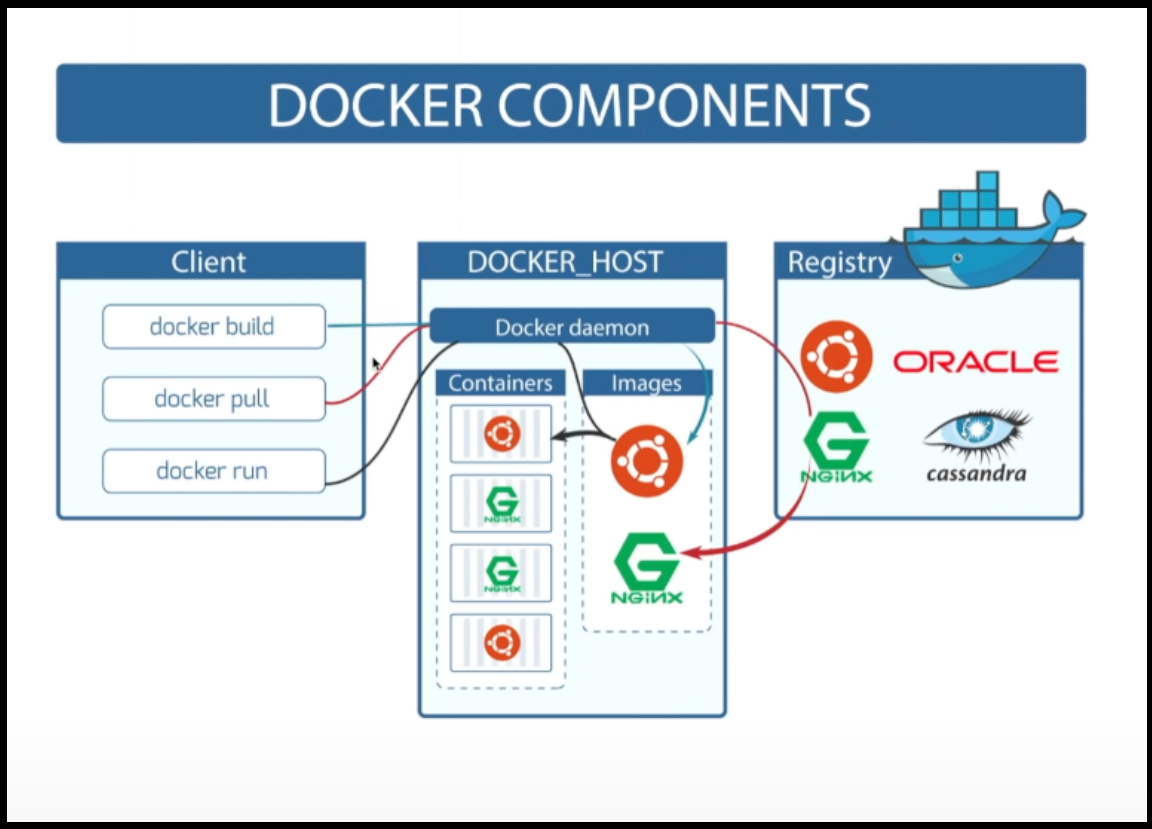
→ If you are not using containers the operation team need to handle hosting & deployement.

→ Since, if you are using Containers you can use node, python, rust anything you want as long as you are creating Images.

Docker CLI

$ Docker run hello-world → Image name

↳ run an Image to create a Container.



⇒ Here when you enter docker run it will talk to docker daemon, daemon will check whether the Image is present in local system if not it will go the registry Dockerhub (Image Library) & it will download from there & runs Image & forms the container.

→ As you know we need Os to run any app like mongo or flask but since Container doesn't interact with outside world then where is the mongo dB running?

When we talk about docker Image they also contain operating system files. It contains smaller version of Os in the Base Image

→ If you don't want to run you can directly download using `$ docker pull`

→ To view the running containers
`$ docker ps`

→ `$ docker Images` → view all the images

→ docker ps

Shows all the running containers

```
docker run -it ubuntu:16.04

root@90647413e3b2:/# ls
bin  boot  dev  etc  home  lib  media  mnt  opt  pr
oc  root  run  sbin  srv  sys  tmp  usr  var
root@90647413e3b2:/# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT
 START   TIME COMMAND
root         1  0.0  0.0   3544  2560 pts/0    Ss
 04:47   0:00 /bin/bash
root        12  0.0  0.0   5472  2304 pts/0    R+
 04:48   0:00 ps aux
root@90647413e3b2:/# ls
bin   dev  home  media  opt   root  sbin  sys  usr
boot  etc  lib   mnt    proc  run   srv   tmp  var
root@90647413e3b2:/# touch hey.txt
root@90647413e3b2:/# ls
bin   etc      lib    opt   run   sys  var
boot  hey.txt  media  proc  sbin  tmp
dev   home     mnt    root  srv   usr
```

→ On a shell, running Ubuntu & created a new file hey.txt &

→ Now using the docker ps Container ls to know Container & we did below

```
base ~ (0.057s)
docker container ls
CONTAINER ID   IMAGE          COMMAND       CREATED
              STATUS              PORTS     NAMES
90647413e3b2   ubuntu:16.04   "/bin/bash"   About a
 minute ago   Up About a minute             vigorou
s_ellis
```

```
base ~
docker container exec -it 90647413e3b2 bash
root@90647413e3b2:/# ls
bin   etc       lib    opt   run   sys  var
boot  hey.txt   media  proc  sbin  tmp
dev   home      mnt    root  srv   usr
root@90647413e3b2:/#
```

→ It means the Bash shell should be attached to this running Container

→ This Command won't work if the Container is not running

→ As you can see hey.txt is present here as well.

→ Both shells are attached to same Container

→ You can also copy Container ID & use # docker stop ID to stop running

→ $ Docker ps -a → shows all the stopped Containers

→ To remove Containers $ docker rm ID

→ $ docker inspect (ex: Ubuntu or ID)
gives all info about Containers
↳ Image state, path, ID, when created, running (or) note, process ID, network Info.

→ $ docker logs ID to check history

→ $ Docker Container prune -f
means deletes all the Containers that
are stopped.

Prune - deletes all Containers that stopped
-f - don't ask again, just do it

```
base ~ (0.458s)
docker run -d alpine ping www.apple.com

616cf8fc5bdc16a55d638074bc736c13a538bbe7a9ffc0ed95da75e0923d95c9
```

↳ -d means detached mode
↳ runs in Background,
↳ returned Id of Container

→ Alpine is Image based on alpine linux, Since all Images are based on Linux Kernels you can run Linux Commands like Pstg.

```
base ~ (0.489s)
docker run ubuntu echo hey

hey
```

whatever you put outside like this will be running on linux terminal

```
base ~ (0.062s)
docker logs 616c

PING www.apple.com (23.48.49.37): 56 data bytes
64 bytes from 23.48.49.37: seq=0 ttl=63 time=122.146 ms
64 bytes from 23.48.49.37: seq=1 ttl=63 time=32.962 ms
64 bytes from 23.48.49.37: seq=2 ttl=63 time=32.531 ms
64 bytes from 23.48.49.37: seq=3 ttl=63 time=30.907 ms
64 bytes from 23.48.49.37: seq=4 ttl=63 time=32.498 ms
64 bytes from 23.48.49.37: seq=5 ttl=63 time=37.084 ms
64 bytes from 23.48.49.37: seq=6 ttl=63 time=38.257 ms
64 bytes from 23.48.49.37: seq=7 ttl=63 time=38.093 ms
64 bytes from 23.48.49.37: seq=8 ttl=63 time=34.407 ms
64 bytes from 23.48.49.37: seq=9 ttl=63 time=40.316 ms
64 bytes from 23.48.49.37: seq=10 ttl=63 time=84.799 ms
64 bytes from 23.48.49.37: seq=11 ttl=63 time=105.360 ms
64 bytes from 23.48.49.37: seq=12 ttl=63 time=31.836 ms
64 bytes from 23.48.49.37: seq=13 ttl=63 time=31.997 ms
64 bytes from 23.48.49.37: seq=14 ttl=63 time=29.808 ms
64 bytes from 23.48.49.37: seq=15 ttl=63 time=31.567 ms
64 bytes from 23.48.49.37: seq=16 ttl=63 time=33.922 ms
64 bytes from 23.48.49.37: seq=17 ttl=63 time=39.026 ms
```

1st 4 digits will suffice

```
base ~ (0.069s)
docker logs --since 5s 616c

64 bytes from 23.48.49.37: seq=508 ttl=63 time=38.362 ms
64 bytes from 23.48.49.37: seq=509 ttl=63 time=37.803 ms
64 bytes from 23.48.49.37: seq=510 ttl=63 time=36.373 ms
64 bytes from 23.48.49.37: seq=511 ttl=63 time=33.058 ms
64 bytes from 23.48.49.37: seq=512 ttl=63 time=37.693 ms
```

↳ Here it gives logs for only last 5 seconds

```
base ~ (0.069s)
docker rmi alpine -f

Untagged: alpine:latest
Untagged: alpine@sha256:b89d9c93e9ed3597455c90a0b88a8bbb5cb7188438f70953fede212a0c4394e0
Deleted: sha256:092561eea88f9f28654bbade209576f5f93efeb8c7ba66a07ac2033c0ddc8ae7
```

↳ force Remove Image

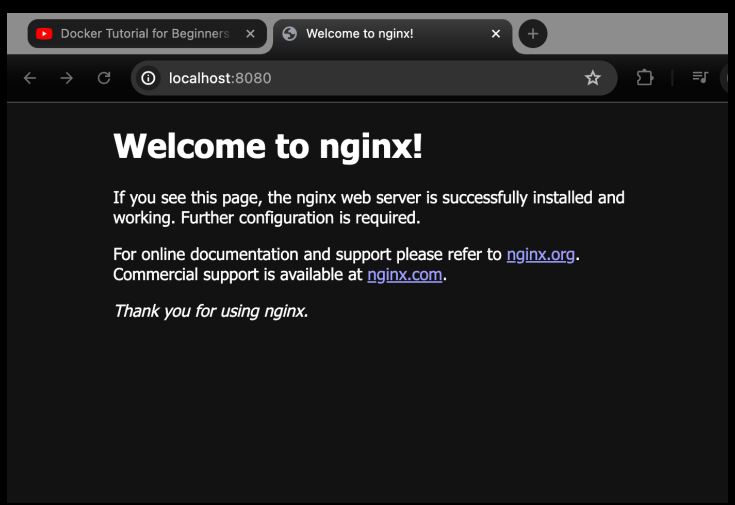
```
base ~ (0.363s)
docker run -d -p 8080:80 nginx

24b5fda024cd7e28b1edcc103dd9ebac9108d7e90e7732c0a015c6944ebdd1b1
```

↳ p is taking 2 arguments the ports you want to expose it to & port of Container

8080 → exposed port localhost:8080

80 → Port of Container (default Port of nginx → 80)

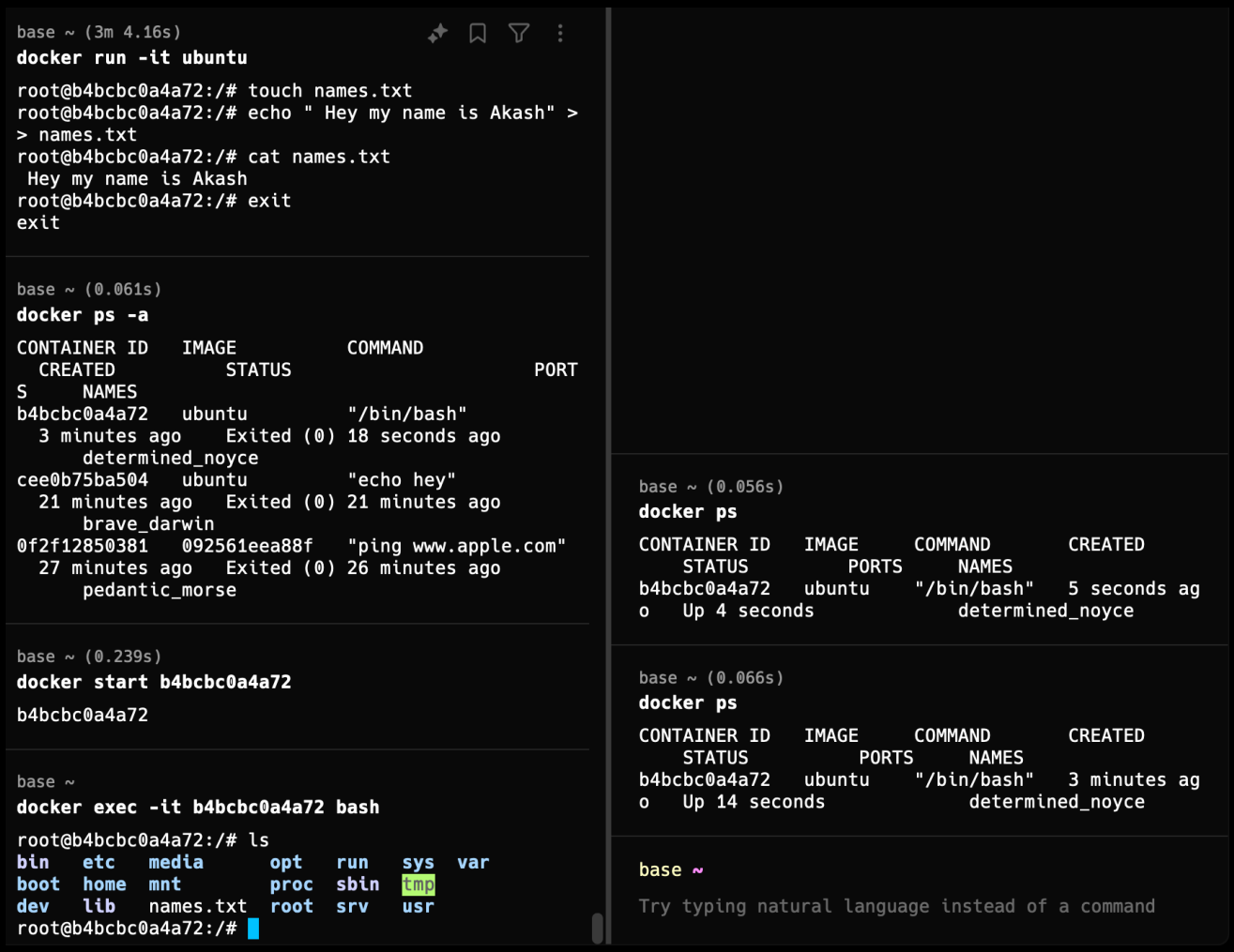


→ So, for accessing Container locally we do as above in the image, it does port forwarding

```
base ~
docker run -it ubuntu

root@b4bcbc0a4a72:/# touch names.txt
root@b4bcbc0a4a72:/# echo " Hey my name is Akash" >
> names.txt
root@b4bcbc0a4a72:/# cat names.txt
 Hey my name is Akash
root@b4bcbc0a4a72:/#
```

↳ If in a new shell again you run the above command names.txt won't not be there because it will be creating a new Container



↳ once you exit (container stops

So using docker ps (or) docker ps -a

find the container ID & then

use # Docker start ContainerID &

then use docker exec -it pid bash

to take to terminal

```
base ~ (0.13s)
docker commit -m "added names.txt file" b4bcbc0a4a72 names_ubuntu:1.01
sha256:4710915979016309398 4ac6835ed1bc625c22684a106b17d4ae4cc98dac31c7f

base ~ (0.063s)
docker images
REPOSITORY                                TAG
                  IMAGE ID       CREATED          SIZE
names_ubuntu                              1.01
                  471091597901   5 seconds ago    101MB
```

Container ID

Image name, Version

↳ By using this Command

```
base ~ (31.286s)
docker run -it names_ubuntu:1.01
root@e67286863357:/# ls
bin  boot  dev  etc  home  lib  media  mnt  names.txt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@e67286863357:/# exit
exit

base ~ (0.063s)
docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES

base ~ (0.056s)
docker ps -a
CONTAINER ID   IMAGE               COMMAND               CREATED          STATUS                      POR
TS     NAMES
e67286863357   names_ubuntu:1.01   "/bin/bash"           43 seconds ago   Exited (0) 12 seconds ago
       infallible_chaum
b4bcbc0a4a72   ubuntu              "/bin/bash"           13 minutes ago   Exited (137) 3 minutes ago
       determined_noyce
cee0b75ba504   ubuntu              "echo hey"            31 minutes ago   Exited (0) 31 minutes ago
       brave_darwin
0f2f12850381   092561eea88f        "ping www.apple.com"  37 minutes ago   Exited (0) 36 minutes ago
       pedantic_morse

base ~ (0.052s)
docker stop e67286863357
e67286863357

base ~ (0.061s)
docker images -q
471091597901
bd1f209b12c1
26af66f0ce61
60e5a9498560
23e08b4032bf
d5e32897ae01
3970bf581bd2
d49959718993
a3b99b6de3e7
5784497e97da
f1f044275595
89424ddaf8a6
6f8fb6bbeb9f
7c5ce99f18b3
b897f7f0612e
a5d57a9e6115
bec7a14748a1
2d9e1b1f31ce
695e39d4c41b
```

↳ As you can see when we run names.txt exists

↳ -q only gives ID of all Images

=>

↳ removes all the Images

↳ Also you can use -f to force it

→ But remember it won't delete Running Containers

```
→  ~ docker run ubuntu:16.04
Unable to find image 'ubuntu:16.04' locally
16.04: Pulling from library/ubuntu
828b35a09f0b: Downloading  2.971MB/41.24MB
238e9b8fdf46: Download complete
1d5b1b491959: Download complete
269a6c6175ba: Download complete
```

↳ These 4 things are called layers

→ Images are built in layers.
→ Each layer is immutable file, A collection of files.

Ex: Here Ubuntu Image is Collection of 4 files

Why?

⇒ Since, Everything is based on Linux Kernel there may be a possibility of you downloading another Image let's say mongoDB It may you some files that are also being used by Ubuntu for ex

⇒ Since you downloaded Ubuntu Image, It does not make sense to download same files for other Image.

⇒ So, Images are built in layers Every Images has a hash Value calculated by SHA 256

→ Command to get hash values.

```
base ~ (0.069s)
docker images -q --no-trunc
```

```
sha256:471091597901630939840ac6835ed1bc625c22684a106b17d4ae4cc98dac31c7f
sha256:bd1f209b12c1d2acfac2c4d063c093a39499e8db44475073cb7ab427b09012d5
sha256:26af66f0ce61786c7c80a4c3df3fc82449ba62eb649d2a456841697910e1e988
sha256:60e5a9498560419431c1dd00c4001c5496563909999976d2bbeb46afcf0a655bc
sha256:23e08b4032bfd4d5865f8d37fdb121697e6eb335fa26e08683d1bd26cb3a72c1
sha256:d5e32897ae0120e97b2e5bf7d8cabc3c71b80db2e8ba50f9047cc16b4abe638b
sha256:3970bf581bd22fd83c054b77e726be5a8b716006990eaaf88dfcbf6e73e1ab68
sha256:a3b99b6de3e7668ded16f3bfbbf3d07ffd9ac61853671d684a7f682932e39b8f
sha256:d49959718993a2a576792eefa24b3ae217882bf190899db3bfa9cf3478826676
sha256:5784497e97da1d5a239a59b545575d62fd2249b8291931d5fbc8f22e37f7ba8b
sha256:f1f044275595b8324130d12b8d97becf965360f899051ebff2486eeb4b9d4541
sha256:89424ddaf8a6a51b507390594588643e452092bc8d92fbfb888cbcee2656c4af
sha256:6f8fb6bbeb9f534c1acb64cdccfedbe0d8994188b4a8c73b2877ca910e929896
sha256:7c5ce99f18b31c735359a52d165caf562f1f5b99983d43c71385bff7877bb747
sha256:b897f7f0612ef880127b7624e6f649b30afd036d9a2709af8949298cfa622b3e
sha256:a5d57a9e6115b2505c8c5a84fe05b40f751f561e18153bdb1e6f4ddaf03d7d0e
sha256:bec7a14748a14a45b4b43e484cb8f07cc329eae1a1742afd5c93a05780b8aec4
sha256:2d9e1b1f31ce77ba4425293c3585b4885593b86cccb9e6856e89a8749c841d6e
sha256:695e39d4c41b769cd71053d787aa47785ed3d7231d5c3056faa03b4c8c085b2f
sha256:9572b90f89886750618efaade1613e782e421156a23fbe4aa38c9a5fb15d6601
sha256:ffb64c9b7e8b9f1891f7a72609d4e691c4671dc4aa83084fc7b5774958d827de
sha256:11ceee7cdc57225711b8382e1965974bbb259de14a9f5f7d6b9f161ced50a10a
sha256:c8405f018c81730ffbfd948a84e63d825d259df975b5b9500fd6426ea48fbd27
sha256:2bceedbd3273e5cc709fc640b29d04fb7dde67b9261dcdf3f69d4a02efca7219
sha256:32eb28f276a5415919c40d989d9a5d49852dcbb8d18d8b0f4510102f9a0c1d12
sha256:96e2c08ad32ed230afea843bed0bebdfc0bc379775f12615466645dac97a2ecf
sha256:6241433e83b5a5599bb4d5c54927bcc77d330f52504cedb7eb7685b9b5d6bce4
sha256:ce59d6220a4de14dfa902c338d6025f55bfe27495b069fd37993082bf521370f
sha256:21efaf670a9c106c4db8748e3ff5e2b7cc95d824ee712522954f0c1d965412f2
sha256:499fa48260b8bd955d5891d6a1dc82d6e2ce7ca908b72ed66ee4a9939018ed9b
sha256:79f8d13ae8b8839cadfb2f83416935f5184206d386028e2d1263577f0ab3620b
sha256:789de7d5f83dc4ef225bced5637e49dd5edf4e22190546246045a513df6218a1
sha256:2437cf762177702dec2dfe99a09c37427a15af6d9a57c456b65352667c223d93
sha256:3750dfec169f630ab8042ca29bca08cbf436ccf1a74ae6f2778f31d0235651d5
sha256:ee301c921b8aadc002973b2e0c3da17d701dcd994b606769a7e6eaa100b81d44
sha256:d125c6a1fe22504f84552b7eb11e353e88691875c18caf68847843892b190ccc
sha256:829e9de338bd5fdd3f16f68f83a9fb288fbc8453e881e5d5cfd0f6f2ff72b43e
sha256:c027a58fa0bb1f8ff35555e068fc8d1a67173782978e25d0450bac0eb5848cd7
sha256:8a85ac985999aae3427246efaa562cc8d6ec4911721b245b71958a8aef693e1d
```

You can see Image ID is equal to the 1st few hashed value.

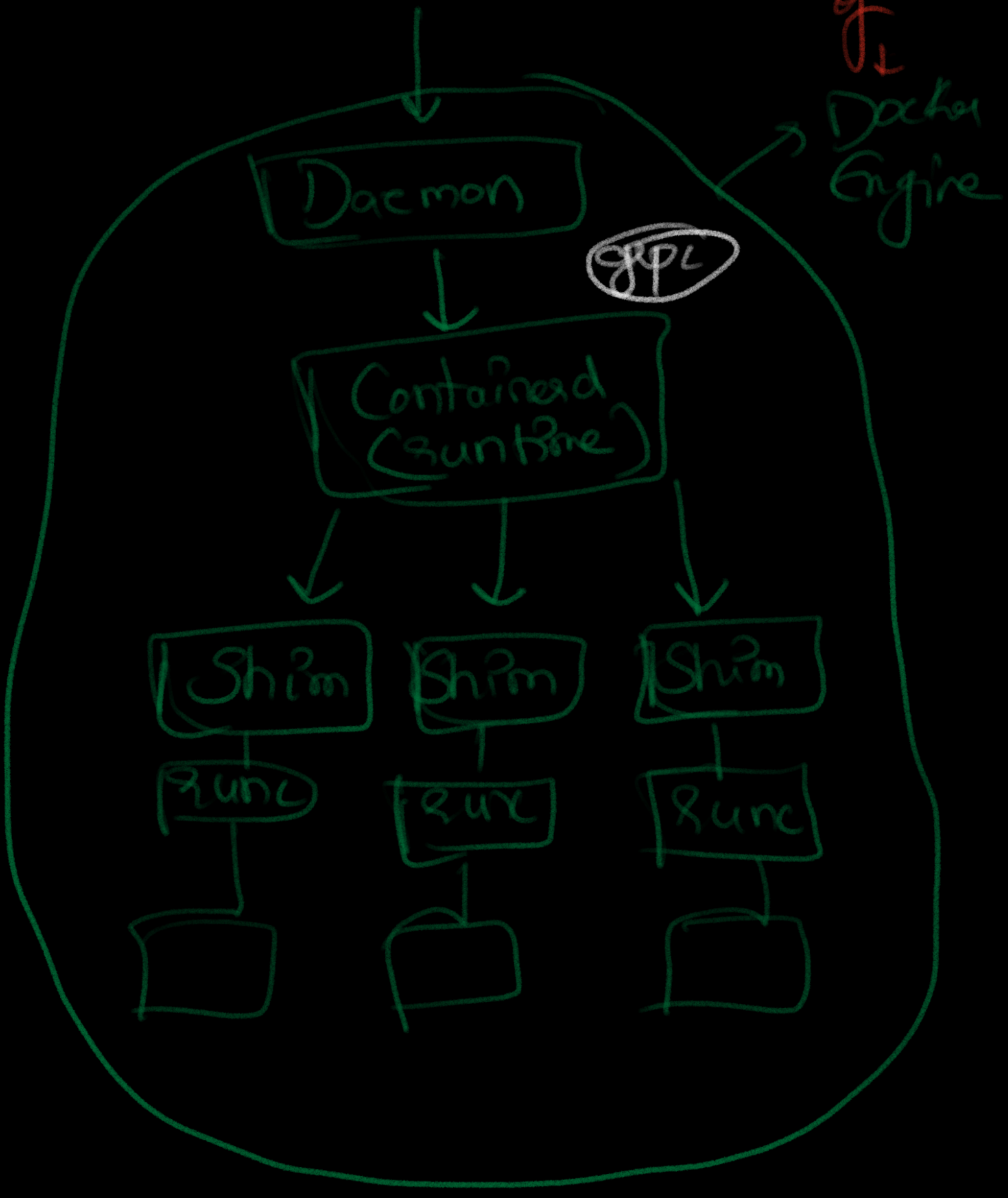
```
base ~ (0.063s)
docker images
```

REPOSITORY	IMAGE ID	CREATED	TAG SIZE
names_ubuntu	471091597901	21 minutes ago	1.01 101MB
mongo	bd1f209b12c1	3 days ago	latest 761MB

→ Also using # docker Inspect, you can check the layers.

→ If there is a particular layer of image that is already downloaded on your system, it will not download again. That's how it makes it fast

Architecture of Docker Engine



⇒ When I run the $ Docker run command docker client, it talks to docker daemon talks to containerd using CRUD operation API's over grpc (it's a protocol used by clients & servers to communicate locally)

⇒ So, Containerd is not the one creating Containers. Runc does that.

⇒ Containerd will take the image & pass it Runc & ask it to create a Container then Runc will talk to your OS kernels & pull resources & create a Container.

⇒ Once the Daemon is stopped all the Containers running will also be stopped.

⇒ If you want to update your daemon than it will impact running Containers.

⇒ So, Once the Container is Created RunC will be removed, RunC role is just deleting, starting Containers.

⇒ Shim takes over RunC once it goes, these are known as daemon less containers.

⇒ client $\xrightarrow{\text{HTTP(:2375)}}$ Server

TLS - Transport layer security ← you can allow only which these exists.