When we talk about orches' → we will talk about Swarm.

This is not that popular. Kubernetes was a better way to do this Orchestn. That's why we have to learn K8s. Docker came up with something as an alternative to K8s → which didn't work out.

* When you are typing commands → you are dealing with client.

whenever you type any command, it goes to the Engine and Engine does the rest.

Runtime (lxd means) → Lowlevel operations

↓

Like → Giving network interface to your container so that it gets an IP address

* There was evolution in this Architecture. The current one is 3rd Architecture. When D^r started, there was one thing which was not good. Then they moved to the 2nd thing — which was also not that great. Then they came to the Current (3rd) architecture.

→ to play with Docker Commands

* manual '--help'

or

* cheat sheet.

Installing Docker on Linux Ubuntu :-

- Create a VM.

$ ssh ubuntu@publicip.

what are the groups available in this Linux VM.

$ cat /etc/group

o/p -
-lxd
ubuntu

$ cat /etc/passwd

o/p. -
-lxd -
ubuntu

Now, install docker. → TO install D^r there are multiple ways

Google : Docker Installation

↓

Install Docker Engine

(or) Google : Docker Install Script

↓

we need to run only 2 commands.

* Docker can be installed by following certain steps

→ Script based Installation :-

curl -fsSL https://get.docker.com -o get-docker.sh

↓ will download the file (get-docker.sh) in my system

sh get-docker.sh

↳ to run the script in

$ curl -fsSL https://get.docker.com -o get-docker.sh

$ ls

o/p get-docker.sh

$ sh get-docker.sh

$ docker version

Version 23.0.3

= =

Permission denied ⟶ ? we got while trying to connect to the Docker daemon socket at Unix.

Client - ✓

Server - ✗

$ docker info

= =

Permission denied ⟶ ?

Docker client needs to speak with Docker Engine.

These communication in a Linux M/c happens over something called as Unix socket.

In the windows M/c, it will happen over something called as named Pipe.

and

for both of these, you need to have Permissions.

$ whoami

Right now, what is the user with which we have logged in? → Ubuntu.

$ cat /etc/group

o/p = =

ubuntu

docker

Docker says

Whoever is part of that Docker Group can speak with that Unix socket.

Generally root user has all the permissions.

$ docker info

o/p : Error

$ sudo docker info

o/p: ⟶ Now its working. client → ✓ Server → ✓

But Everytime adding sudo is also not a good Idea.

So lets add (current user) ubuntu to Docker Group

$ whoami

o/p: ubuntu

$ sudo usermod -aG Docker ubuntu

2 ways ⟨ 1) Use Sudo (or) 2) Add User to Docker Group

$ exit

$ ssh ubuntu@ip

$ docker info

o/p: client → ✓
Server → ✓

I'll try to run very simple command which will check whether the containers work (or) not

↓

$ docker container run hello-world

o/p: Unable to find image 'hello-world : latest' locally

Status: Download newer image for - -

Hello from Docker!

This message shows that your Installation appears to be working Populaly.

- - -
- - -
- - -

This means It is working.

Docker client:-

* client is nothing but any command that I type.
Eg Docker Info

D' client
It Speaks with Servers trying to tell that
client has asked me info —
I can display client Info. Blog
that is something which I know

But Server Info → it will send a request to D' daemon. If D' daemon was not able to respond (socket file)

We have fixed the socket issue (by adding ubuntu user to Docker)

↓

Docker daemon received the request → will send the response → which we see in the output.

* Windows 10/11 (Non-Home Editions) :-

* you are using Windows 10 Home → don't even try installing Docker. It will not work. (as Microsoft's code doesn't have docker).

When you install - what happens is In the backend - what happens is Linux M/c gets created.

Whenever you are typing docker — the stuff is running on Linux.

For non-home Editions:-

Google:- Docker for Windows Desktop. (But it is not recommended to install Docker on Windows Bcoz most of us wud be working on Linux Environments.

* Dokker Playground:

↓

For this create a Docker Hub A/c.

This playground gives a Linux M/c with Docker installed for 4 hrs for free (But no guarantees) (Use this when you want to check something quickly)

* Daemon has all the necessary components so that you can communicate with Docker.

Daemon in Linux means - a program that runs all the time.

* Runtime components are low level components.

When you say $ docker container, runtime speaks with OS and does something. P.

Terms to be aware of :-

containerd runc dockershim OCI

libcontainer appc

rkt containers grpc

All of these components make Docker run (or)

They have impacted Docker in such a way that Docker had to change itself

* Docker looks like a overnight success.

Dotcloud (solomon Hykes), Microsoft altered its code to run D^r on windows

If a startup in San Francisco becomes a overnight success →

and lot of industry attention is given to this →

the thing which many do is → they add lots of feature and

Docker also they added loads of features.

They have added too many features and it was becoming like a Hypervisor again.

The whole idea of light-weight was going for a toss. (It was becoming fat)

So, at the same time, a company launched core-OS (a Linux OS),

and then they have given something called as Rocket Containers.

They came up with the specification called as appc.

Docker had their own specifications which was not good for industry.

This C^r stuff was itself in ?

So then, all of the people who are creating C^rs & using C^rs → they came to a common forum → and they have agreed to something called as OCI (Open Container Initiate). Whether you create a D^r C^r or rocket C^r (or) Podman C^r all of these C^rs have a same specification.

Now, D^r image which you build can be run on podman also (Bcoz it is OCI compliant image)

In Arch 1
Docker used to rely on Lxc Linux containers

Blog of OCI & rkt containers { D^r changed from architecture 2 to Arch 3 }

OCI is a standard

Lec-40 - Apr-7

Springpetclinic Application — Tomcat — Docker

* We will try to experiment with Containers.

What is in the Container?

* 1st → Lets experiment with normal Linux M/c.

* Applications bring revenue to Enterprises and to run application - we need servers, OS etc.

(If I can directly run an application — I don't need a OS)

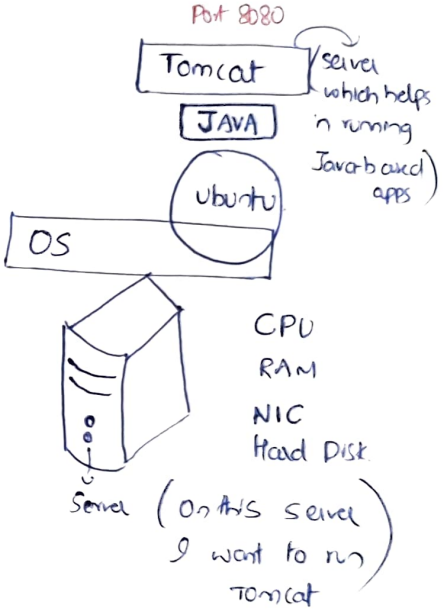
↓ This is basically the mindset for today's class.

We are not concerned about OS.

We are concerned about application which we are trying to build.

* Lets create a Linux Virtual Machine :-

& Install Tomcat in it



Our intention is to run Tomcat.

TO run Tomcat → I need some OS.

For that reason I have taken a server

↓ which can be < Physical (or) Virtual

This Server will have → some CPU

RAM

Network Interface card → to connect to network.

Virtual Hard-Disk.

Lets take Linux (ubuntu) OS.

If you want to run Tomcat → you need Java to be installed.

* Tomcat Application can be accessed over your network on port 8080

* On this ubuntu M/c → lets check System info.

In windows → to see system info → we open task manager — CPU, RAM, Disk, Network (Ethernet) — 4 essential Components.

Login to AWS

↓ (Name: experiment)

Create VM (EC2) ubuntu
t2.micro
mykey
Sec. Gp → Launch Instance.

(dnf)
(yum)

$ ssh ubuntu@public.ip

Ubuntu
$ htop → CPU
Mem

F10 → quit

But we don't see network over here.

there.

To figure out what is network Interface → Execute ifconfig.

$ ifconfig

o/p: command not found. But can be installed with: sudo apt install net-tools

software

$ sudo apt update

$ sudo apt install net-tools -y

$ ifconfig

o/p:- eth0 :- - —
ip: 172.31.40.87

So VM has { CPU, RAM, Storage, Network Interface.

Google: System info. in Linux → $uname
$uname -n
$uname -v
$uname -r — kernel version
$uname -m — architecture.
$ sudo lshw

$ sudo apt-cache search openjdk

$ sudo apt install openjdk-11-jdk -y

$ java -version

$ sudo apt-cache search tomcat

$ sudo apt install tomcat9 -y

$ sudo systemctl status tomcat9

o/p: active (running)

To check if Tomcat is working (or) not ?

systemctl → it is a daemon

* We have experimented in the Linux VM.

→ Network Interface gives network connectivity.

→ CPU, RAM & Disk are available.

→ To install softwares we have used package manager "apt".

Redhat < DNF (dnf) / YUM (yum)

Alpine → apk.

Lets get into container & see if we have CPU, RAM, DISK and Network there.

In ubuntu linux, we can also use

$ sudo lshlk

to see all info.

8Gb → 3 partitions broken into & mounted on root

$ Sudo df -h

Filesystem	Size	Used	Avail	Use%	Mounted on
/dev/root	7.6G	2.4G	5.3G	31%	/

* Let us go with easier way of running docker

Google:

docker playground → Login → start → Add new instance → copy SSH.

Go to ubuntu.

ubuntu

$ ssh ip - - - - - - - play-with-docker.com

↓

yes

I can login into the machine.

But

Permission denied. Lets do it on my laptop (on ubuntu VM)

↓

schai > ssh ip172-18-0-7 - - - - play-with-docker.com

↓

Here it connected. So there is some extra work which we need to do (in Linux)

Alt + Enter → Full screen → in Docker PlayGround.

/# apt install openjdk-11-jdk tomcat9 -y

Now, lets try to create a docker container :-

$ docker container run -it ubuntu /bin/bash

o/p: unable to find image 'ubuntu:latest' locally

latest: Pulling from library/ubuntu

Download complete.

So, I'm inside the container

root@c :/# apt update

/# htop

o/p: command not found. (Docker Container is light-weight Area. It will not have all the fancy tools which you will have on OS)

For our convenience lets install htop.

/# apt install htop net-tools -y

Container will exactly have what is necessary to run your Application.

/# htop

o/p: Memory, CPU → Container is having Memory ✓ CPU ✓

* whenever you are asking this information you are asking the Parent (Parent is the Linux M/c on which Docker is running)

/# ifconfig

o/p - ipaddress ✓ (Container) It has its own ipaddress → Network ✓

/# df -h

o/p: Overlay Size 6.7G → It has storage space → Disk ✓

So, once you are inside the @, there is actually no difference. It is exactly as if you are inside the OS.

/# apt install openjdk-11-jdk tomcat9 -y

Inside the C^r we are able to do → whatever we were doing on Linux M/C.

It is giving illusion as if we are working with Linux M/C.

/# systemctl status tomcat9

o/p: Host is down

/# Exit

[node1] (local) root@192.168.0.28 ~

$ docker container stats

$ docker container run -d nginx

o/p:

Download complete

$ docker stats

Yes nginx is running. → This container will have its own CPU, Memory, Disk.

So, Inside the C^r, we are able to install applications.

* We were able to do exactly the similar operations inside the Container as well.

* But what is the diffence v/s Virtual Machine?

↓

Virtual Machine isolates at a hardware level.

$ docker container run -d nginx

o/p: 762 . . . 15

$ docker container run -d nginx

o/p: 8081 . . . . 4f1

$ docker container run -d nginx

o/p: 271 . . . . .

: Totally we are running 6 containers now.

But when you are working on Containers.

↓

$ docker stats

o/p:

Container ID	Name	CPU	Mem
52	—	—	9Mb/31.4GB
73	—	—	95Mb/31.4GB
:	—	—	9.xM/—
a8	—	—	—

By default, all of them are trying to use the shared details of your host system

-p 80,tcp / 80,udp etc

So, the OS that is present inside the container will not make decisions. It will ask the host Kernel.

* The OS layer inside the container is very minimal.

Size of Linux Image = ?

ie. what is the size of Linux OS that we install on a server?

↓

700 - 800mb.

$ docker image pull alpine

$ docker image ls

o/p:

Repository	Tag	Image ID	Size
alpine	latest	9ed4 -	7.05 Mb
nginx	"	080 - -	142 Mb
ubuntu	"	0802 - -	77.8 Mb.

→ Size is 7 mb.

↓

So I can run my appln inside a Linux ~~operating~~ M/c which has size of just 7 mb.

Inside the C^r we are giving importance to the application. (not to OS)

We will not have any extra tool → which is not necessary for the appn to run.

Alpine ↴

* On this if you try to run your appln, your image size is less.

It takes less CPU, less RAM.

So, rather than running it in 1 C^r, run it in 5 or 6 C^rs → your appn will be fast bcoz you are giving importance to appns.

Lets look at Application : Spring Petclinic

Lets try to run this appn inside a Linux M/c.

Launch an Instance (Name: spc)

Git bash

$ ssh ubuntu@public-ip

Dockerfile :

```
FROM amazoncorretto:17-alpine-jdk
LABEL author=Raja
ADD target/spring-petclinic-3.0.0-SNAPSHOT.jar springpetclinic.jar
EXPOSE 8080
CMD ["java", "-jar", "/springpetclinic.jar"]
```

Once you create the Dockerfile

↓

Create the docker packaging format which is docker image.

↓

`docker image build -t spc:1.0 .`

↓

After this whenever I want to run this appln

↓

To create container

↓

`docker container run -d -P spc:1.0`

what I'm trying to tell is ↑ I have built an image called as 1.0

If I run twice → I'll get 2 instances of my appln

Thrice → 3 " "

If you want to send it to the outside world, then you shud give some special names (Ignore for now)

To run this appln — I need Java 17 and I don't need Tomcat.

{ Dockerhub has lot of images which are written by many people which you can reuse.

Google : DockerHub

↓

Search:

To run this Spring Petclinic, I need Java17

So one option is → Launch Linux M/c & install Java on your own

(or)

Directly take a Machine where you have Java.

↓

Search : Correto

amazon correto

↓

This is Java by Amazon.

(Or)

Search : openjdk.

Better version is Corretto

↓

In corretto we are interested in Java17

↓

Select Java-17 alpine-jdk

whenever you see alpine (or) slim

↓

means I want image size to be slim

Ubuntu@ $ Sudo apt update

$ sudo apt install openjdk-17-jdk maven -y

$ git clone https://github.com/spring-projects/spring-petclinic.git

$ cd spring-petclinic/

$ mvn package

$ java -jar target/spring-petclinic-3.0.0-SNAPSHOT.jar

* Docker Way of Working :-

This appln is → Java Package.

- The package in the case of Docker is Docker Image.

  We create a **docker Image**

  ↓

  It is a docker packaging format

  Eg: docker container run nginx

  ↓

  It is a docker Image

- So, we will be creating Image for the application which our developers are building.

  ↓

  For that, we need to create Dockerfile.

  and

  push the Image to Registry (docker hub)

  ↓ means

  (where anyone who is permitted can use your image.)

  ↓

  Create the container using the image anywhere.

* So, containerizing your application means

trying to write some file which is called as Dockerfile

→ in that filling some information

⇓

and Containerizing Application (or) Dockerising Application

whenever you are building the code — you don't build the code anymore

→ you build the docker package.

* To run spring-petclinic → Java 17 is needed.

So, for every app → you will have certain requirements, dependencies.

Name of the jar → spring-petclinic-3.0.0-SNAPSHOT.jar (we got it by mvn package in VM ubuntu)

On VM → Ubuntu —

$ sudo ps aux | grep java

Installing Docker on Ubuntu M/C

we need to install docker in this M/c. !

$ curl -fsSL https://get.docker.com -o get-docker.sh

$ sh get-docker.sh

$ sudo usermod -aG docker ubuntu

(client needs to communicate with server and client-server communication happens on a Unix socket)

⇓

that permission is given to Docker User Group

So, for that reason, we are adding Ubuntu user to Docker Group

$ cd spring-petclinic/

$ vi Dockerfile

```
FROM amazoncorretto:17-alpine-jdk
LABEL author=khaja
ADD target/spring-petclinic-3.0.0-SNAPSHOT.jar /springpetclinic.jar
EXPOSE 8080
CMD ["java", "-jar", "/springpetclinic.jar"]
```

:wq

$ docker image build -t spc:1.0 .

$ docker container run -d -p 8081:8080 spc:1.0 (Now lets try to run it on 8082)

$ docker container run -d -p 8082:8080 spc:1.0 8083

$ docker container run -d -p 8083:8080 spc:1.0

So we are running 3 containers.

$ docker stats

o/p:

CONTAINER ID	NAME	CPU(%)	MEM USAGE/LIMIT
319..	-	69.5%	127.7Mb/7.76GB
2c7..	-	63.4%	153.1Mb/7.76GB
cab..	-	65.75%	217.9Mb/7.76GB

They are taking hardly 250 Mb to run this application

Our Security Group doesn't allow 8081, 8082 -- and all that stuff.

Select the Sec Gp

↓

Edit (Tell → I'm not speaking about 1 port, I'm speaking about all the ports → open it)

All TCP ✓

Chrome Public-ip:8080 → ✓

" :8081 → ✓

8082 → ✓

8083 → ✓

Now, I don't need one complete M/c to run the same appln.

we can run this on a M/c where it hardly takes 250 Mb

* Now in 1 GB of Space - I can run 4 Instances of this Application.

↓

much better than t2.Micro. (in 1 t2.Micro → I can run 4 Spring Apps)

* Now, you publish this image to registry and run it anywhere.

* So, containerizing your application means

* trying to write some file which is called as Dockerfile

→ in that filling some information

⇓

* and Containerizing Application (or) Dockerising Application

whenever you are building the code — you don't build the code anymore

→ → you build the docker package.

* To run spring-petclinic → Java 17 is needed.

So, for every app → you will have certain requirements, dependencies.

: Name of the jar → spring-petclinic-3.0.0-SNAPSHOT.jar (we got it by mvn package in VM Ubuntu)

On VM → Ubuntu —

$ sudo ps aux | grep java

Installing Docker on Ubuntu M/C

we need to install docker in this M/C. ↓

$ curl -fsSL https://get.docker.com -o get-docker.sh

$ sh get-docker.sh

$ sudo usermod -aG docker ubuntu (client needs to communicate with server and client-server communication happens on a Unix socket)

↓

that permission is given to Docker User Group.

So, for that reason, we are adding Ubuntu user to Docker Group

$ cd spring-petclinic/

$ vi Dockerfile

```
FROM amazoncorretto:17-alpine-jdk
LABEL author=Khaja
ADD target/spring-petclinic-3.0.0-SNAPSHOT.jar /springpetclinic.jar
EXPOSE 8080
CMD ["java", "-jar", "/springpetclinic.jar"]
```

:wq

Lec-41- Apr-8

Docker Architecture Evolution

* © gets an Isolated area. It gets its own process. It gets its own storage space. It gets its own cpu & RAM.
* When the D^r started (from that point) to what we have → there has been lot of change in D^rs architecture.

→ How Isolations are created (or) How Containers work ? :-

Go to docker playground
↓
create instance.
↓
Copy ssh.

>ssh . . . → This is a Linux M/c. In this we have Docker.

[node1] (local) root@192.168.0.18 ~
↓
$ ps → means give me all the processes

```
o/p:-  PID   USER   TIME   COMMAND
         1   root   0:00   /bin/sh -c . . . .
        18   root    ..    dockerd
        .    .      .      .
        .    .      .      .
```

} Processes that are running in your M/c.

↓
we can see that certain processes are running inside this M/c.
Lets get into container
$ docker container run -it ubuntu /bin/bash

Now I'm inside the ©. Lets execute ps (process)

root@fb.. ba:/# ps

```
o/p.  PID   TTY
        1          bash
        9          ps
```

} → Processes that are running inside the container.

Container → has processes (PID)

Linux M/c → has processes (PID)

In any process tree (if you go with Linux) → there will be one important Process ID (PID) → which is called as PID 1 (that is the 1st process that gets started when your M/c is created

We can see PID 1 in Linux M/c & also in Container. (So C' also has its own process tree)

Container

```
/# apt update
/# apt install net-tools
/# ifconfig
```

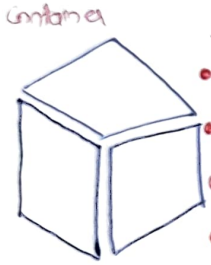
Linux M/c

```
[node1] (local) root@ ~
$ ifconfig
o/p: eth0  192.168.0.18
```

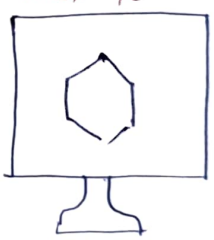
↓

Over here we have an IP address: 172.17.0.2 → it is connected to Network Interface which is eth0.

Linux M/c → Here also we have eth0: 192.168.0.18



Container Components
- Process Tree
- eth0: 172.17.0.2 (Network NIC)
- CPU
- RAM
- Disk Mounts
- Users



Linux M/c Components
- Process Tree
- eth0: 192.168.0.18 (Network NIC)
- CPU
- RAM
- Disk mounts
- Users

Container.

root /# df -h

o/p: Here also we have diff't mount points

Linux M/c

$df -h

o/p. there are different mounts that are available over here

Now, if we create one more C' inside the VM (Linux M/c) - it might get 172.17.0.3 which will have its own process ID, Net Interf, CPU, RAM etc Users etc

↓

How is this actually created?

Docker Internals

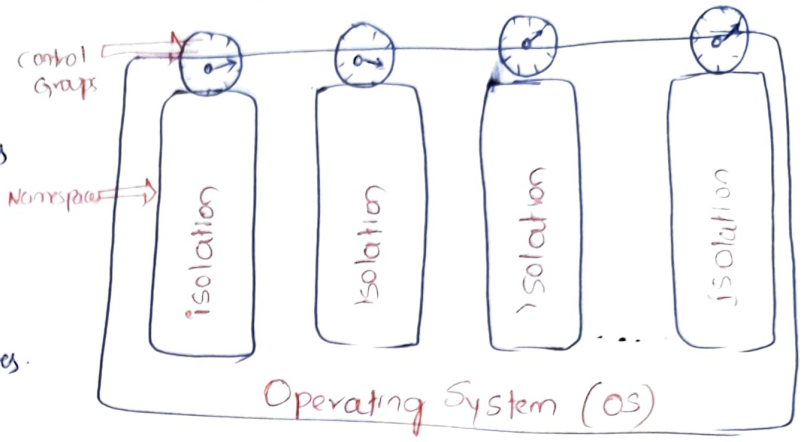
In your OS → you have created an isolation. In this isolation, we get certain resources to run our system

we can create multiple isolations.

with which you can create containers.

* In Linux, whenever you create isolations, you need something called as Namespaces.

* Isolations on the Linux M/C's are created using a linux kernel feature called Namespaces.



* There are different namespaces that are available.

* It gives you an Isolated Area and to that area it gives certain features.

If you dive deeper

* You wud have a namespace which is called as PID Namespace

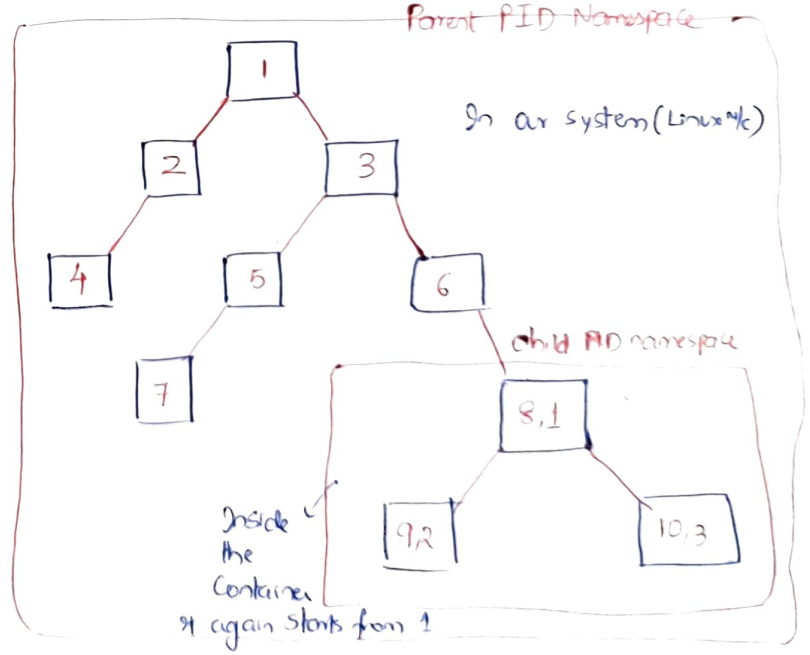
↓

This is a process ID namespace

↓

Creates the isolated process tree inside container.

* Namespaces are a feature of the linux kernel that partitions kernel resources such that One set of processes sees one set of resources while another set of processes sees a diff.t set of resources -



* Similarly we have net namespace

↓

It is responsible for giving network Interface into your containers

* Similarly we have mount namespace which gives disks

* Uses namespace → which will give new set of users inside the (C)r.

Initially

If I had to create a namespace (that is → to write code) → it will be like...

↓

very tough.

So, very few Orgzs were using (C)rs

But Docker has democratised the creation. It gives you very simple commands with which you can create containers.

Apart from namespaces

* We need to apply limits also.

How much of CPU do you want to give to ©ʳ ?

How much of memory " " ?

Dʳ (Engine made) makes it easy to create isolated areas of resource limits

To put those kind of restrictions → there is something called as cgroups (Control Groups)

Eg: If you tell you want ©ʳ only with 512Mb of ram dont give more than that → then cgroups come into play.

2 major Linux concepts that make this Container a reality is
- Namespaces
- Control Groups

This is not a docker concept.

This is Linux concept

It is present in Solaris also. (In solaris, these ©ʳˢ are called as Zones)

* In Linux, it (container) was initially used for handling Guest Users.

Why? → Guest user shud not be able to access other things
So they used to create an Isolated Area. Run everything inside that.
Inside it basically doesn't have any connection with whatever is running in actual machine.

* Windows never had this concept of containers
They have re-written the code in Windows 10 and when 2016 server was created.

Mount
Mount is the way how disks get attached to in the Linux System

* When windows created containers → they also used same naming conventions.
Even in windows → you have same names → which create isolated areas

* Lets look at Container from OS's perspective:-

For OS ⇒ © is a Process

↓ It is nothing but a Process

(so, for the VM in which you are running a ©ʳ, kernel) it is almost like a Process

For the appln inside the Container

For App Inside the Container ⇒ © is ? © looks like OS

But in reality, it is Process with namespaces & Control Groups

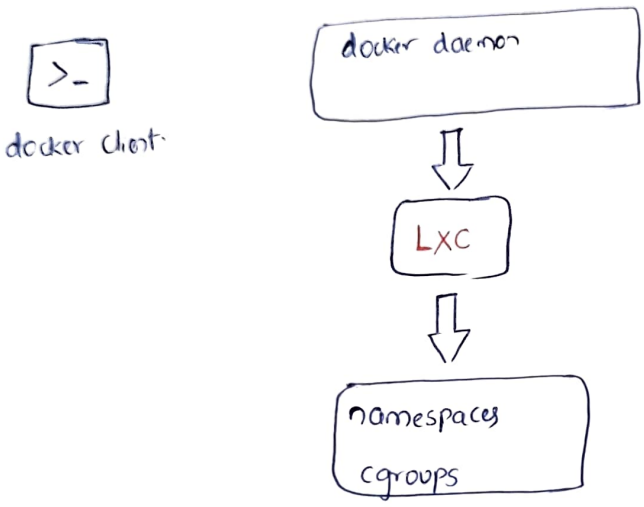
↑ New Kernel is updated → successfully ☺

Docker Architecture

Generation 1 :-

* To create container we need to use 2 features → 1) namespaces
2) control groups.

* This was 1st gen, where docker daemon used lxc (a linux kernel feature) to create containers.



Docker Daemon
↓
whenever you ask daemon to do anything
↓
it used to rely on lxc
It didn't use any ©ʳ techᵍʸ

What it used to do is —
whatever lxc requires to create ©ˢ
- docker daemon used to pass that info.

This info internally uses
namespaces
cgroups
↓
This m/c had to be a linux m/c
bcoz there was no lxc in mac (or) windows

Here also docker had 2 components
1) client
2) daemon

whenever you execute Dˢ command,
Dˢ commands used to speak with lxc (which is linux container - an inbuilt feature of linux kernel)

There are 2 problems

1) To create ©ˢ docker daemon is relying on lxc (∵ lxc is part of linux kernel)
Some features used to work in some linux kernels some features didn't work in some linux kernels
For Eg. If you are updating linux OS → the whole containers used to stop.
Bcoz features that were released by lxc were not matching with what was necessary. → This was becoming a huge roadblock.
2) New kernel is updated → Suddenly ©ˢ stop working.

Initially Linux → docker daemon used to speak with Linux Containers (which is part of Linux kernel).

So, there were some breaking features which were always released and docker had to continuously fix them.

So, you installed Docker → You have created Containers → Suddenly, a new kernel update came → You updated which had some breaking change → Suddenly, your Containers stopped working.

So, what Docker decided was

Let me write my own component for creating containers

that was 2nd Generation.

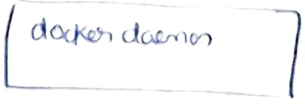
- Generation 2 :-

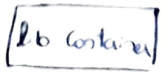
* Since Docker was relying on lxc which was kernel features, updates to kernel frequently used to break containers created by Docker.

So, Docker has created its own component called as libcontainer (libc) to create containers.

* Docker wanted Containers to be multi-OS and lxc was definitely not the way forward.

* Now, in the place of lxc, you have libc





With this release docker became stable.

Now, D' doesn't depend on kernel features, bcoz it is directly speaking with the low level api's of linux to create C's.

* This is where lot of people started adopting Docker.

* Adoption of Docker was drastically increased as it was stable.

* libc uses directly linux api.
  ↓
  It used to speak with Kernel directly (not with lxc)

* Generally, at this phase, D^r was a Overnight success. Microsoft also adopted D^r. and they have started giving native ©^rs in windows.

* The problem was
  Lot of features' requests were coming in and Docker has added too many things. So Docker had started becoming fat. The ©^r techgy which we use to create micro-services → was becoming fat → was becoming like a mini-hypervisor.

At that point → A company created. rkt containers (Rocket containers) was launched. They came up with their own specification of creating containers which was called as appc specification
↓
The reason why this came was, they didn't like the fatness of docker.
Now, this company is acquired by RedHat. (It was revamped & now there is something called as Podman by RedHat)

* Having competition is good. But, having too many standards is not a good idea, for doing the same thing. (Bcoz diff^t Org^zs will have diff^t standards and it will not be portable)

So, what they have done is,
all of them have basically formed a group and they came up with something called as OCI (Open Containers Initiative).

In OCI, they have given 2 specs :-

1) Image Spec
2) Runtime Spec

The basic idea of this is, anyone who runs (or) creates a Container Techgy → you need to follow this specifications.
Now, the adv of following this specification is, I can build an image in docker and try to run it in a diff^t ©^r techgy.

Generation 3:

Reasons for Gen 3 :-

1) Docker becoming fat. It was monolith (Everything was inside the docker Engine)

2) A company started rocket container with appc ↓ spoke with libc.

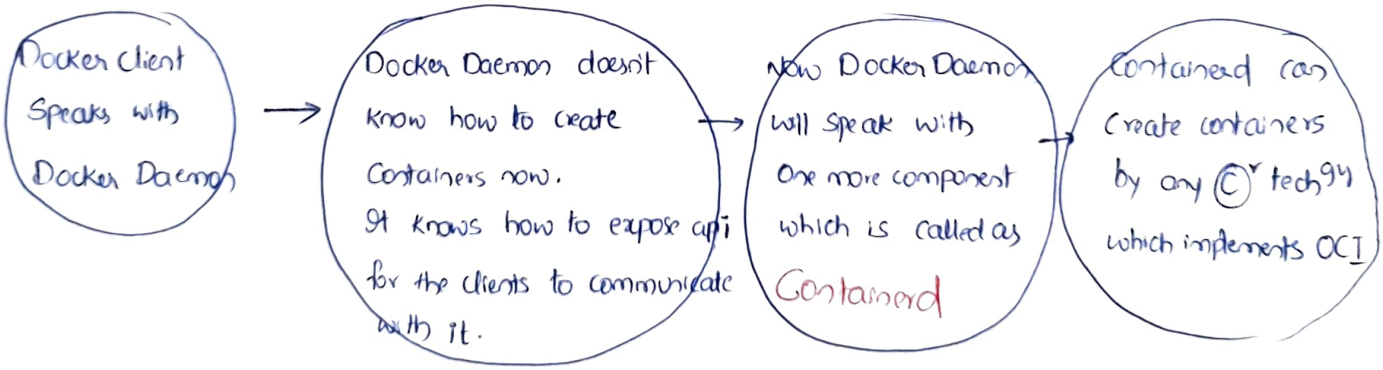
↓ not good for Container World.

* In this Gen, docker Engine was revamped from monolith to multi-component architecture and the images and the containers were according to OCI (open Container Initiative) image spec and runtime spec

* Earlier, all of the intelligence was with docker daemon and whenever you have to create C^rs → it is libC. → apart from this - all the logic was in docker daemon.

This docker daemon is what was becoming fat.

So, at this stage, what docker has done was



When we install docker what we get is runc.

Right now, Current Architecture of D^r has 2 major components { Client Component, Server Component

Server is no longer one docker daemon. Now we have docker daemon which exposes api's and then it speaks with Containerd → which speaks with runc

There is one more component which is docker shim

* runtime is where you have containerd and runc

docker daemon is exposing api's

Orchestration is multi-user scenarios

* when you speak from docker client → docker daemon exposes api's to listen requests from docker client. →

→ It happily passes the requests to containerd → containerd manages the lifecycle of container (how to create, how to stop, how to pause your C' etc) → containerd forks a runc process (which will create runc) which creates container.

(fork means create a new process) and inside that containerd will run runc) runc will create the C'.

→ once the container is created the parent of the container (it is actually runc) will be docker shim → which basically looks into containers and report the status back to your daemon

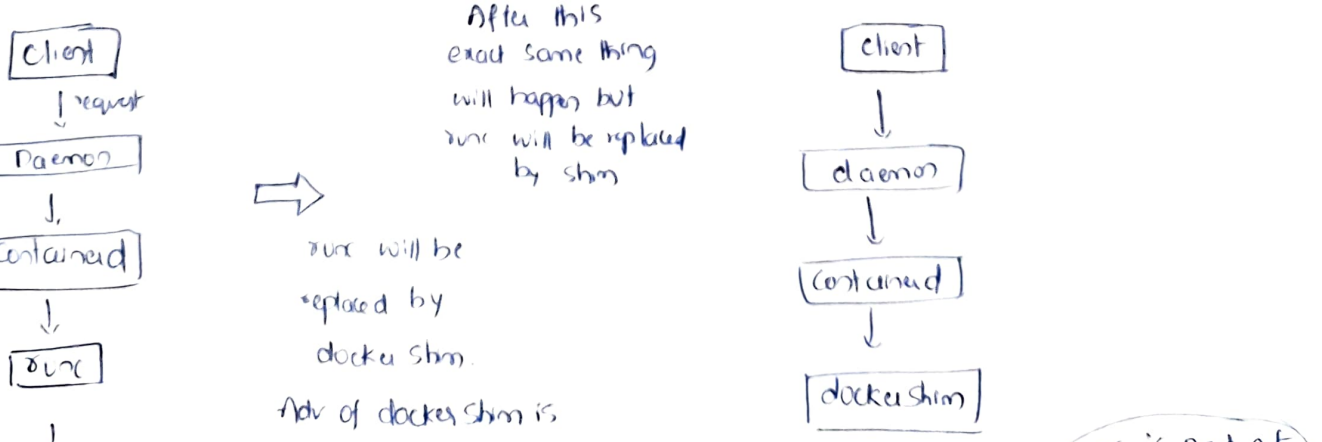
(docker shim is a very light-weight component ↑)

The adv of dockershim is → even while you are updating docker → your containers still run → for that reason they have introduced shim.

If they have used containerd (or) runc there → when you update docker → you need to stop your C' → But Now your C's can run even when you are updating your D'Engine.

why?

Bcoz you are trying to run it with docker shim as its parent but not runc as a parent.



Adv of dockershim is

Once your containers are running I can update docker Engine. Your C's will still be running

runc is part of docker installation

bcoz dockershim has nothing to do with D' installation.