

# Containerization

Sandeep Kumar

# Introduction

Your Name

Total experience

Background – Development / Infrastructure

Experience on Containers/Kubernetes/Git

Your expectations from this training

# Agenda

- Introduction
- Docker Components
- Classroom Environment
- Containers
- Docker – Images
- Docker - Building Images
- Deep Dive – Images
- Deep Dive – Containers
- MicroServices Example
- Container Network Model
- Docker Volumes

# **Session: 1**

## **Introduction**

# What is Container?

Before proceeding with the new terms and technologies lets have a look on the history/traditional approaches used for the application since ages.

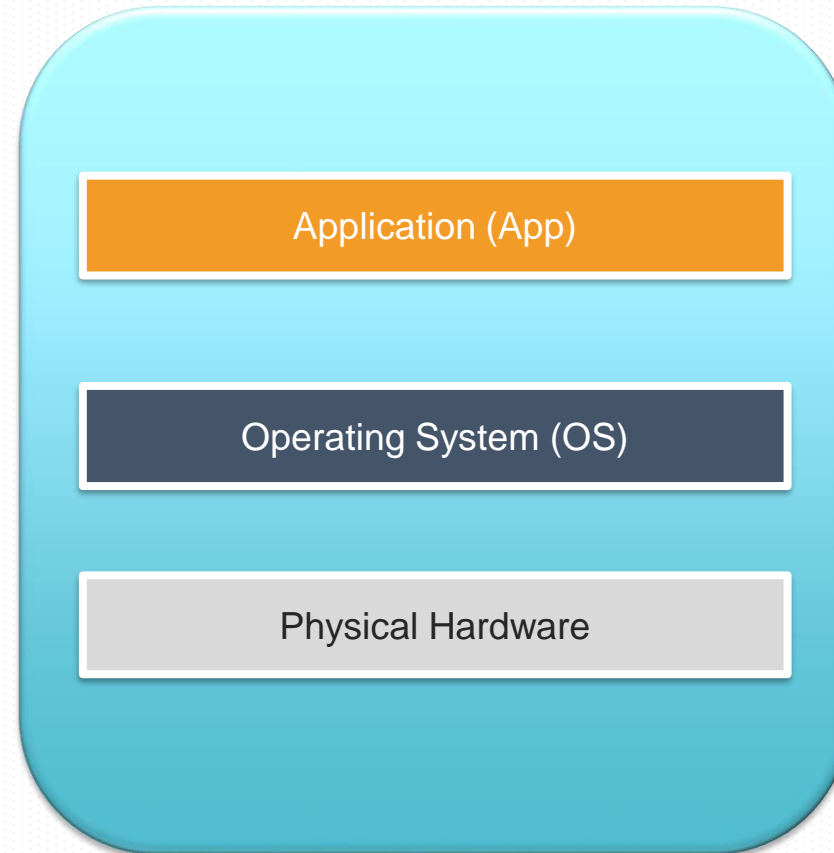
# A History Lesson

- In the traditional ages of development and deployment of application

Unused Resources

Difficult Migrations

Vendor dependency



**Traditional Approach**

Slow Deployment time

Difficult to Scale

Huge cost in Infrastructure

# A History Lesson

- The problem got the solution by a technology called “Hypervisor-Based Virtualization”.
- One physical server can contain multiple running applications.
- Each application need a VM to run the application binaries.

# Virtualization

OverHead

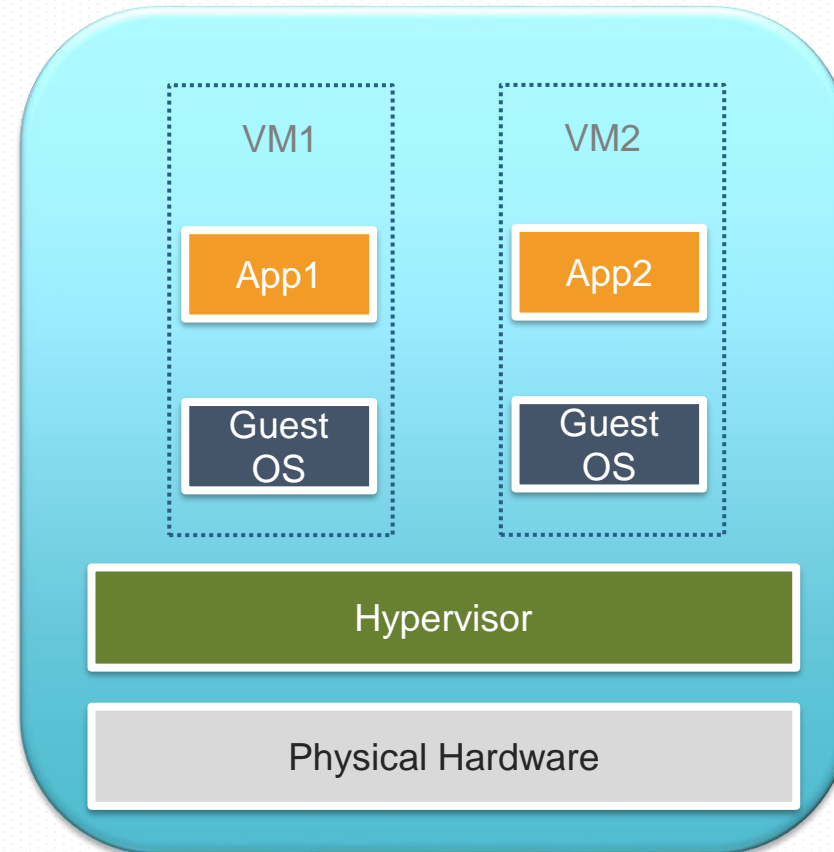
Portability issues

Boot-time still in Minutes

Scaling issue in Hybrid Env

Migrations still failing

Costly Solution



Server Virtualization

Better Resource Pooling

Easier to Scale

Flexibility & Easy Migration

Faster Deployments

Faster Boot time



# Containerization

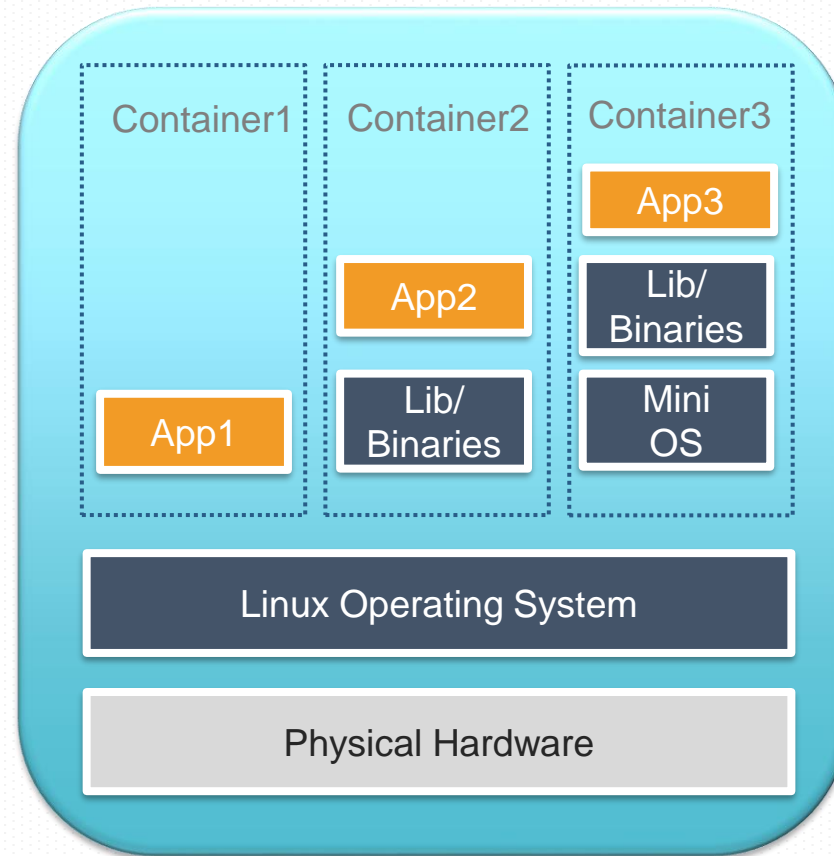
Less OverHead

Highly Portable

Scaling in Hybrid Env

High Migrations success ratio

Cost Effective Solution



Containerization

Much Better Resource Pooling

Extended Scaling

Flexibility & Easy Migration

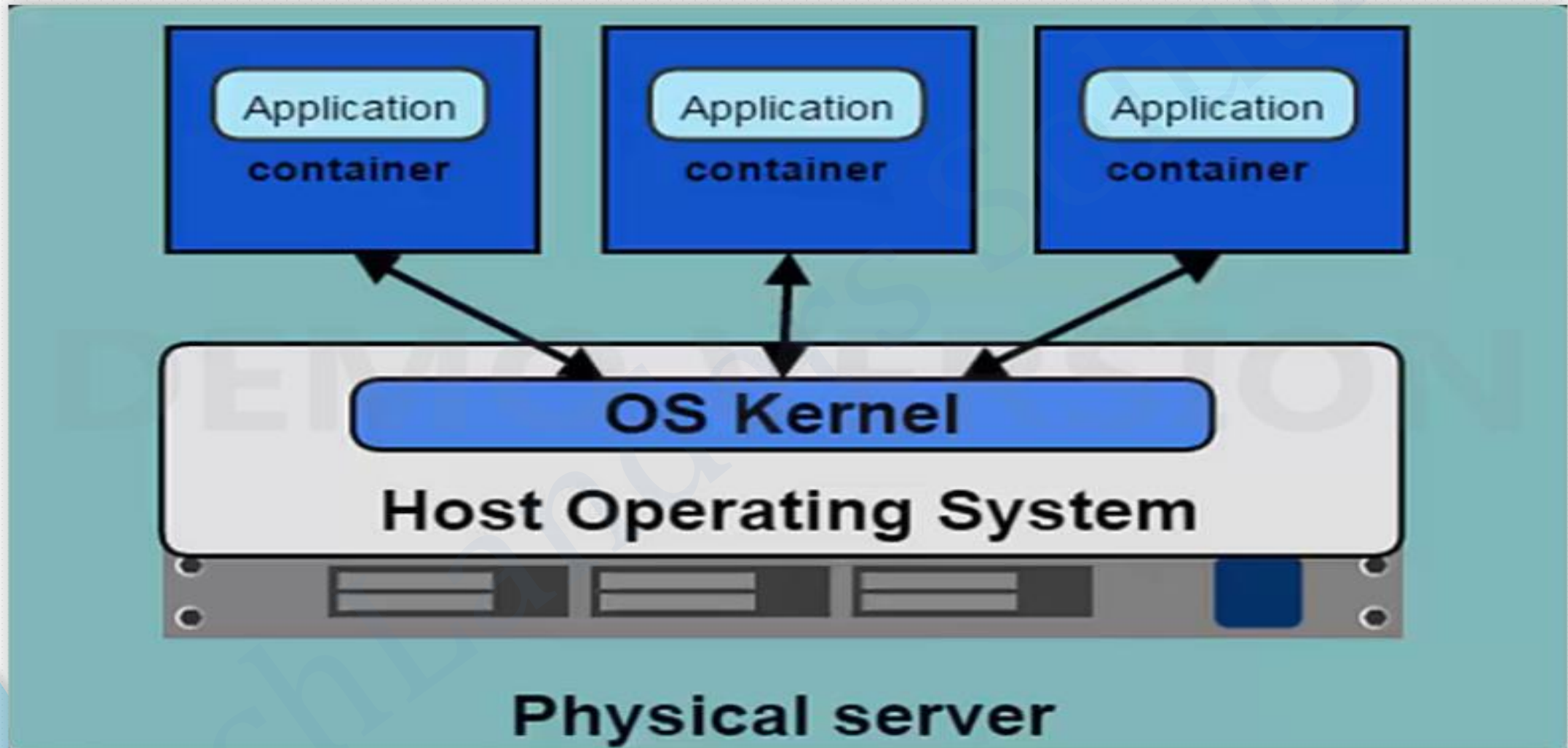
Faster Deployments (Seconds)

Faster Boot time (Seconds)

# Introducing Containers

- Container based virtualization uses the kernel on the host's operating system to run multiple guest instances
- Each guest instance is called a "Container"
- Each container has its own
  - Root Filesystem
  - Processes
  - Memory
  - Devices
  - Network Ports
- From outside its looks like a VM but its not a VM

# Overview of Containers



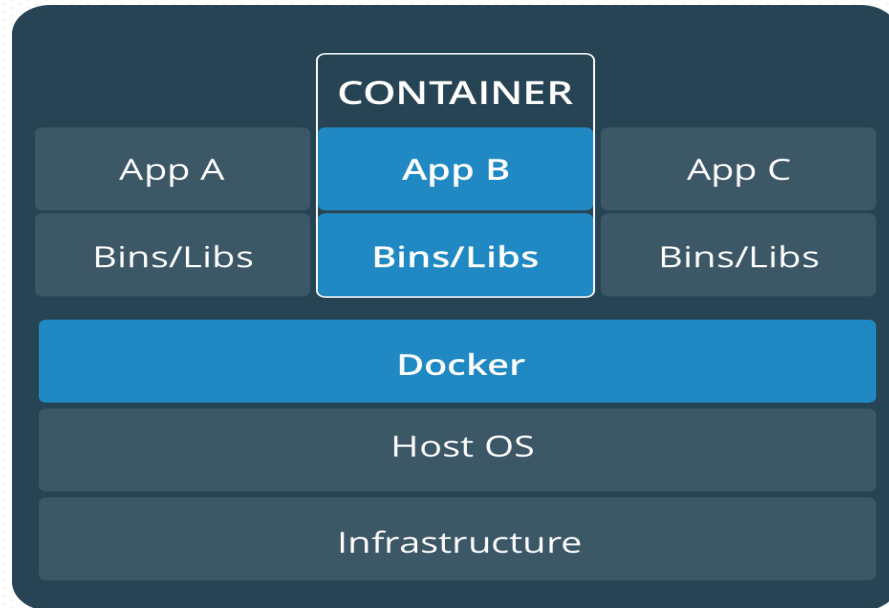
# Overview of Containers

- A question comes in our mind why do we need different containers of our applications.
- A question to audience, I want to install:
  - Three application
  - Pre-requisites all three require different Java Versions
  - What will be the possible solution for this design

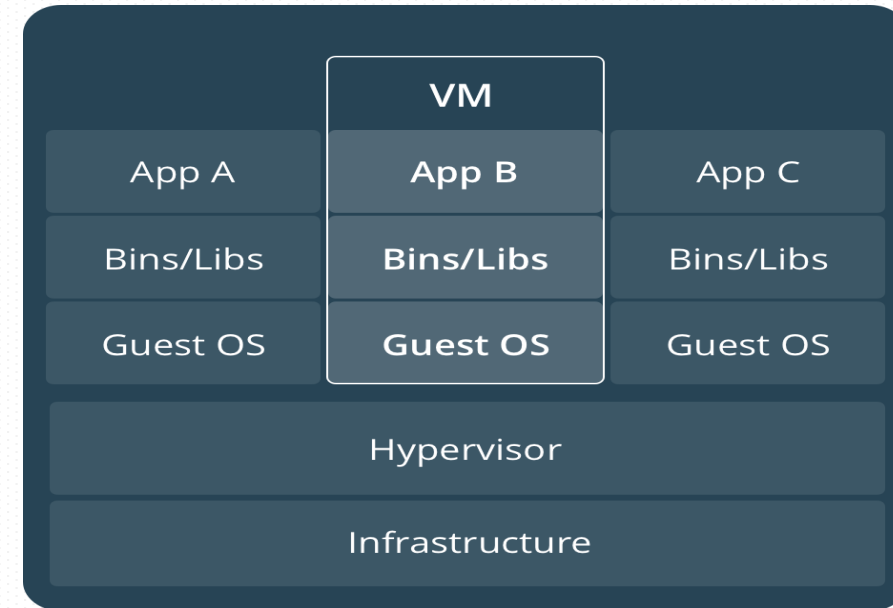
# Containers VS VM's

- Container are more light weight
- No need to install dedicated guest OS, no virtualization like VM is required
- Stop/Start time is very fast
- Less CPU, RAM, Storage Space required
- More containers per machine than VM's
- Great Portability

# Containers VS VM's

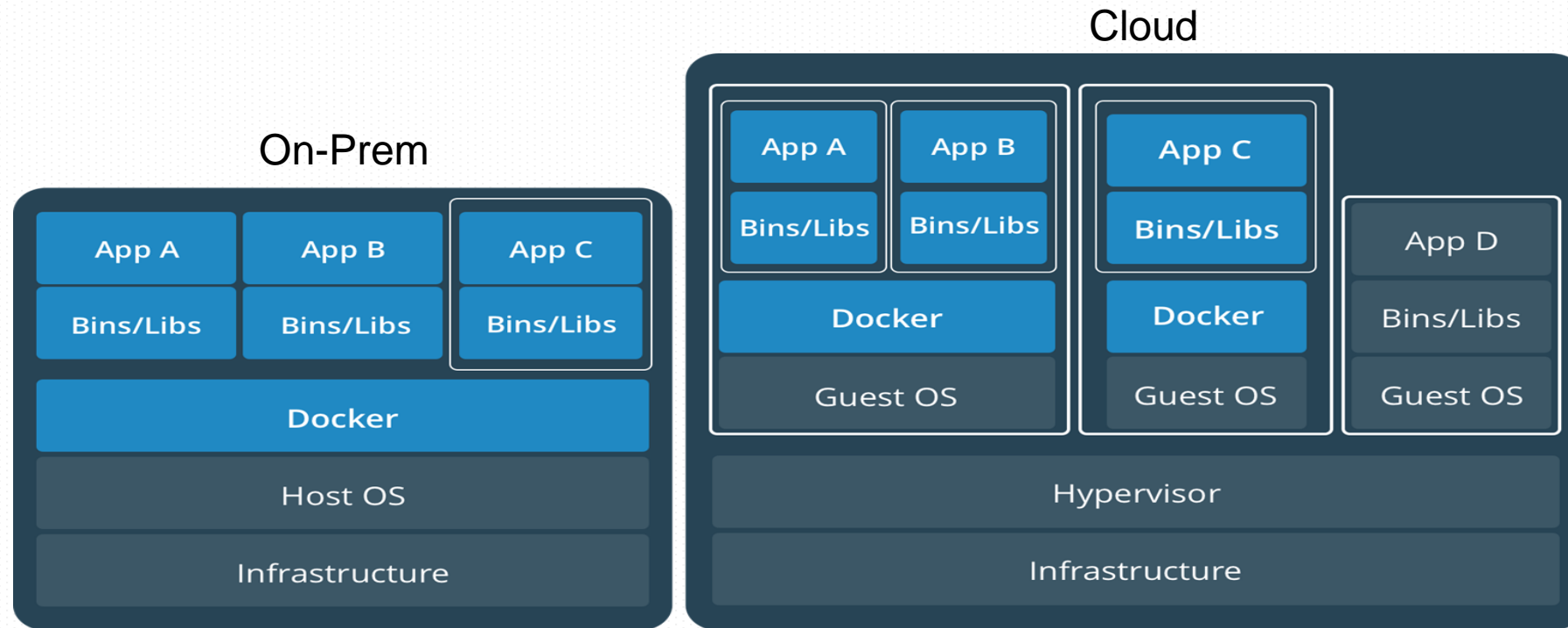


Containers are an app level construct



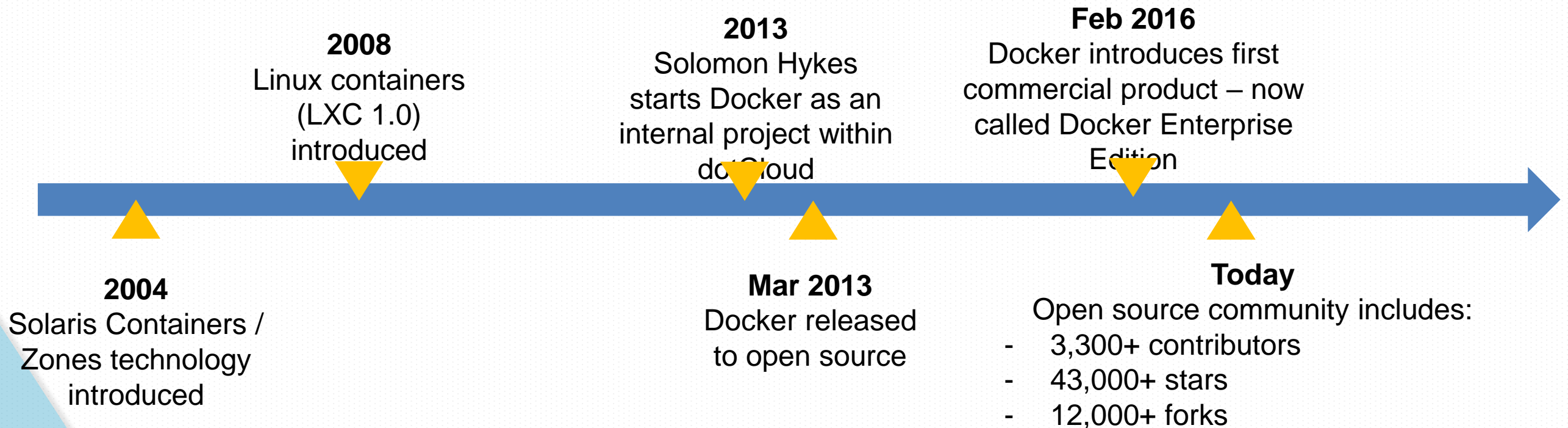
VMs are an infrastructure level construct to turn one machine into many servers

# Containers and VM's together



Containers and VMs together provide a tremendous amount of flexibility for IT to optimally deploy and manage apps.

# Origins of Docker Project





# Origins of Docker Project

- 'dotCloud' was operating a PaaS platform, using a custom container engine.
- This engine was based on 'OpenVZ' (and later, LXC) and AUFS.
- It started (circa 2008) as a single Python script.
- By 2012, the engine had multiple (~10) Python components. (and ~100 other micro-services!)
- End of 2012, 'dotCloud' refactors this container engine.
- The codename for this project is "Docker."

# Docker becomes a platform


- The initial container engine is now known as "Docker Engine".
- Other tools are added:
  - Docker Compose (formerly "Fig") (YAML file)
  - Docker Machine (SDK)
  - Docker Swarm (Cluster)
  - Docker Cloud (formerly "Tutum")
  - Docker datacenter (Management Layer)
- Docker Inc. launches commercial offers.


# About Docker Inc.


- Docker Inc. Formerly 'dotCloud' Inc, used to be a French company
- Docker Inc. is the primary sponsor and contributor to the Docker Project:
  - Hires maintainers and contributors.
  - Provides infrastructure for the project.
  - Runs the Docker Hub.
- HQ in San Francisco.
- Backed by more than 100M in venture capital.


# Deployment Problem


Multiplicity of Stacks

 **Static website**  
`nginx 1.5 + modsecurity + openssl + bootstrap 2`


 **Background workers**  
`Python 3.0 + celery + pyredis + libcurl + ffmpeg + libopencv + nodejs + phantomjs`

 **User DB**  
`postgresql + pgv8 + v8`

 **Queue**  
`Redis + redis-sentinel`

 **Analytics DB**  
`hadoop + hive + thrift + OpenDK`

 **Web frontend**  
`Ruby + Rails + sass + Unicorn`

 **API endpoint**  
`Python 2.7 + Flask + pyredis + celery + psycopg + postgresql-client`

Do services and apps  
interact  
appropriately?

Multiplicity of  
hardware  
environments

 **Development VM**

 **QA server**

**Customer Data Center**



**Public Cloud**

**Disaster recovery**

**Production Servers**



**Production Cluster**
















**Contributor's laptop**



Can I migrate  
smoothly and  
quickly?



# Matrix Checks

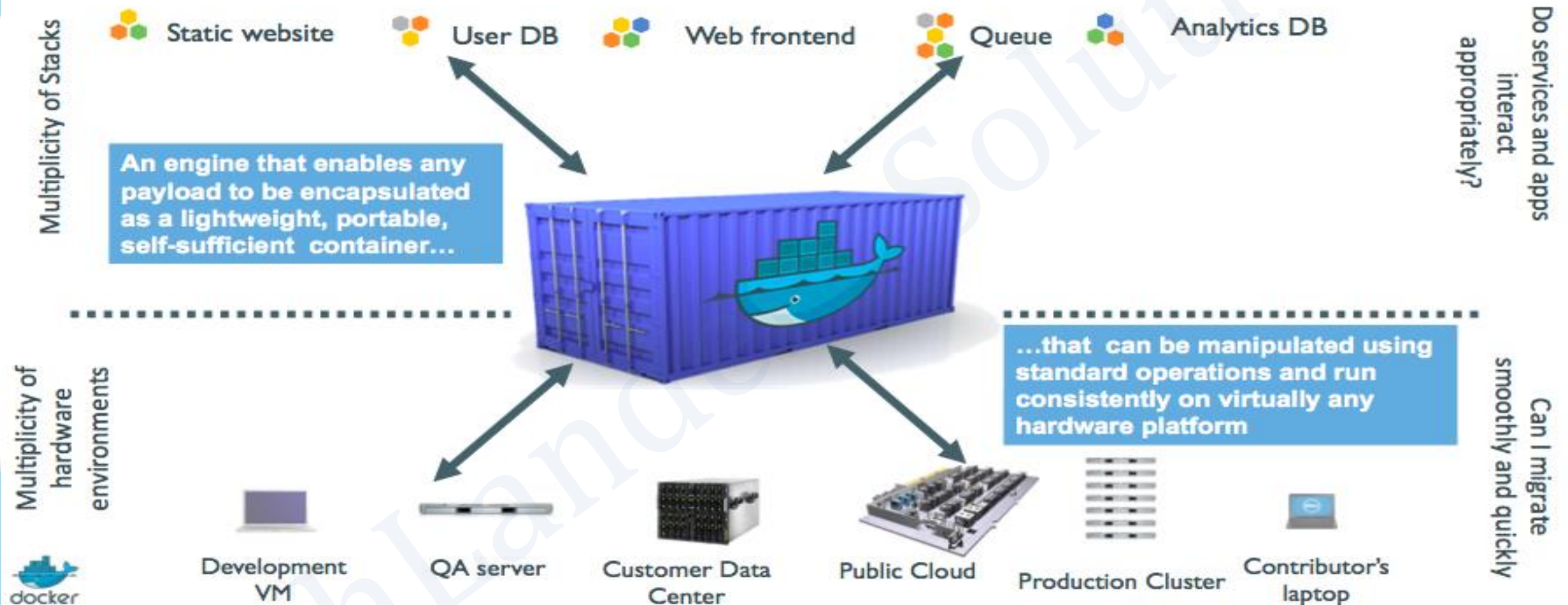
	Static website	?	?	?	?	?	?	?
	Web frontend	?	?	?	?	?	?	?
	Background workers	?	?	?	?	?	?	?
	User DB	?	?	?	?	?	?	?
	Analytics DB	?	?	?	?	?	?	?
	Queue	?	?	?	?	?	?	?
		Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers
								

# Intermodal Shipping Containers

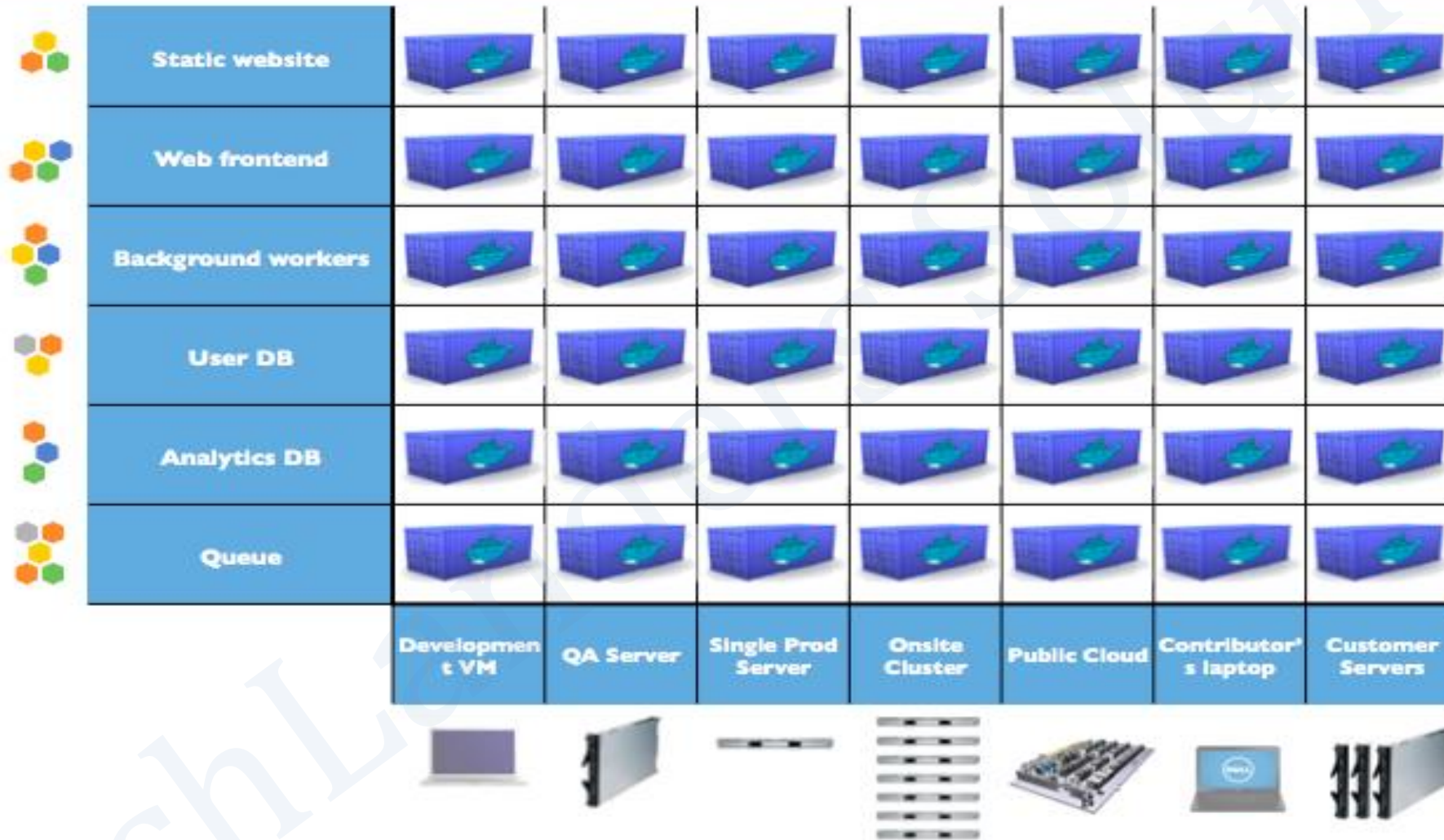




# Shipping Container for Applications



# Eliminate the Matrix





# Results

## Speed

- No OS to boot = applications online in seconds

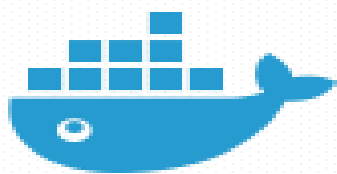
## Portability

- Less dependencies between process layers = ability to move between infrastructure

## Efficiency

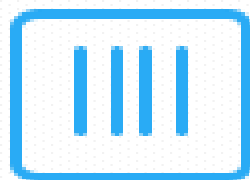
- Less OS overhead
- Improved VM density

# Adoption in Just 4 years



**14M**

Docker  
Hosts



**900K**

Docker  
apps



**77K%**

Growth in  
Docker job  
listings



**12B**

Image pulls  
Over 390K%  
Growth



**3300**

Project  
Contributors

# Why Docker?



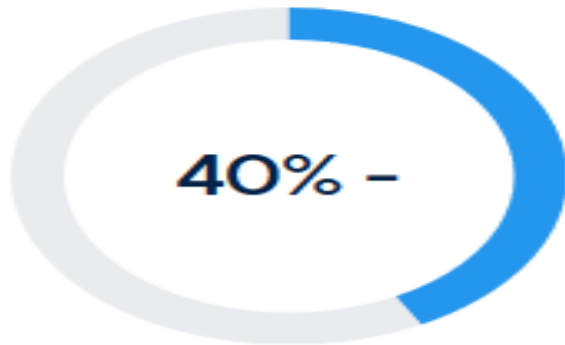
Faster Time to Market



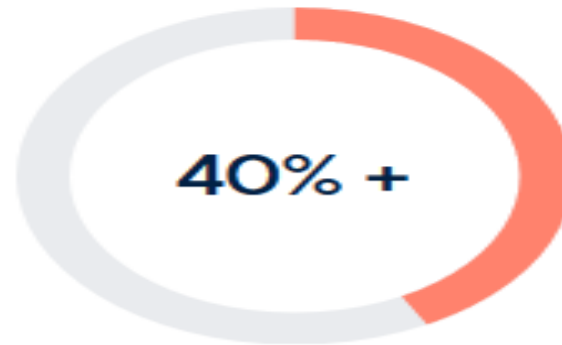
Developer Productivity



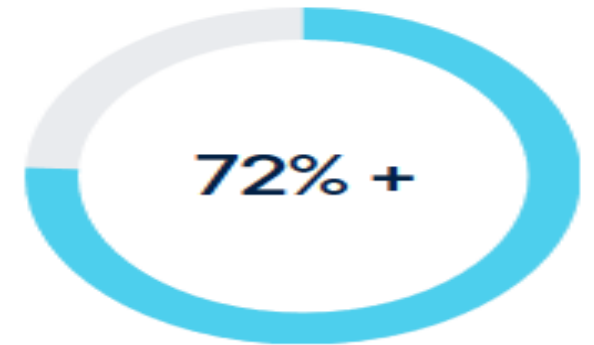
Deployment Velocity



IT Infrastructure Reduction



IT Operational Efficiency



Faster Issue Resolution

# **Session: 2**

## **Classroom Environment**

# Docker Engine Install Demo

- ▶ Docker Engine/Client would be installed on Training Environment as demo LAB.
- ▶ <https://docs.docker.com/engine/installation/linux/centos/>

# Docker Installation Key Points

- ▶ docker.service Systemd File:

```
[root@TechLanders lib]# more /usr/lib/systemd/system/docker.service
```

- ▶ Docker Socket file:

```
[root@TechLanders lib]# file /var/run/docker.sock  
/var/run/docker.sock: socket
```

- ▶ Docker PID file and Docker Container PID file (This is Docker Daemon which will have pid1)

```
root@TechLanders run]# cat /var/run/docker.pid  
3715
```

```
[root@TechLanders libcontainerd]# cat /var/run/docker/containerd/docker-containerd.pid  
4251
```

- ▶ Docker Detailed Information:

```
[root@TechLanders libnetwork]# docker info
```

# Manage Docker as a non-root user

- ▶ The docker daemon binds to a Unix socket instead of a TCP port.
- ▶ By default that Unix socket is owned by the user "root" and other users can only access it using sudo.
- ▶ The docker daemon always runs as the root user.
- ▶ If you don't want to use sudo when you use the docker command, add users to Unix group called "docker"  
example: `usermod -aG docker <user-name>`
- ▶ When the docker daemon starts, it makes the ownership of the Unix socket read/writable by the docker group.

# Docker Command Reference

- Below is the official Docker command reference link:
- <https://docs.docker.com/engine/reference/commandline/container/#child-commands>



# Session: 3

## Docker Components

# Docker Overview

- ▶ Docker is an open platform for developing, shipping, and running applications.
- ▶ Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.
- ▶ With Docker, you can manage your infrastructure in the same ways you manage your applications.
- ▶ By using Docker's methodologies for shipping, testing, and deploying, you can reduce time of customer delivery.

# Docker Platform

- ▶ Docker provides the ability to package and run an application in a loosely isolated environment called a container. The **isolation** and **security** allow you to run many containers simultaneously on a given host.
- ▶ Because of the **lightweight** nature of containers, which run without the extra load of a hypervisor, you can run more containers on a given hardware combination than if you were using virtual machines.

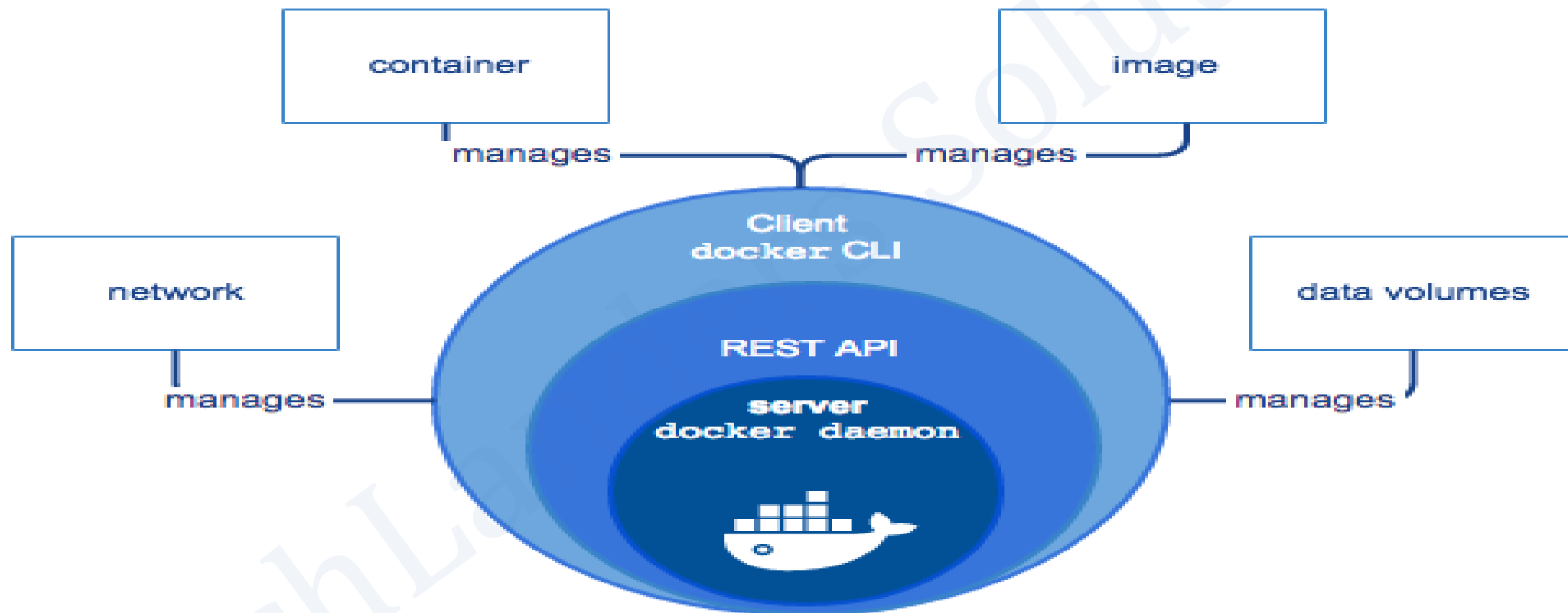
# Docker Test

- ▶ `docker --version`
- ▶ Test the docker functioning: `docker run hello-world`

# Docker Platform

- ▶ Docker provides tooling and a platform to manage the lifecycle of your containers:
  - Encapsulate your applications (and supporting components) into Docker containers
  - Distribute and ship those containers to your teams for further development and testing
  - Deploy those applications to your production environment, whether it is in a local data center or the Cloud

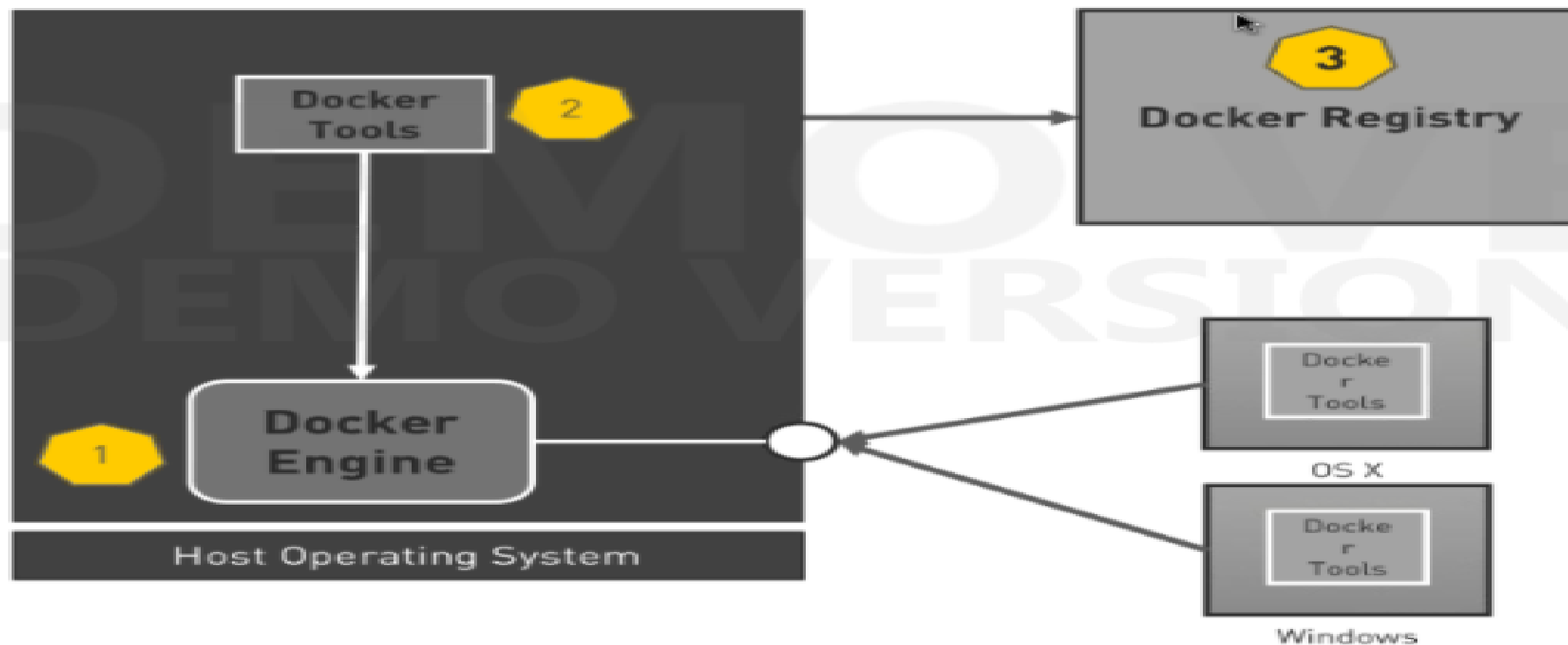
# Docker Engine



# Docker Engine

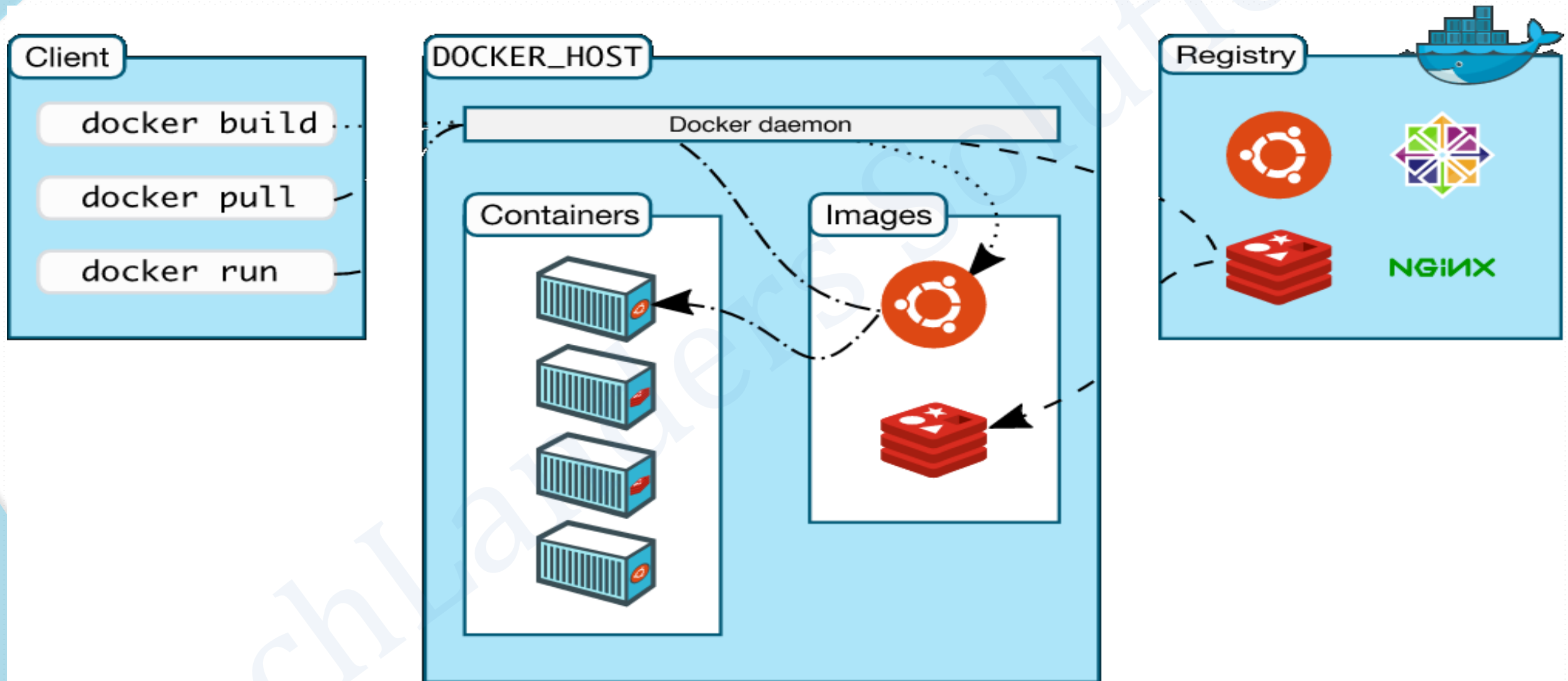
- ▶ Docker Engine is a client-server application with these major components:
  - A server which is a type of long-running program called a daemon process.
  - A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
  - A command line interface (CLI) client.
- ▶ The CLI uses the Docker REST API to control or interact with the Docker daemon through scripting or direct CLI commands
- ▶ The daemon creates and manages Docker objects, such as images, containers, networks, and data volumes

# Docker Architecture





# Docker Architecture



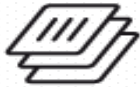
# Docker Architecture

- ▶ Docker uses a client-server architecture.
- ▶ The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers.
- ▶ The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon.
- ▶ The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.

# Docker Architecture

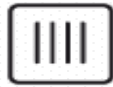
- ▶ The Docker Daemon
  - The Docker daemon runs on a host machine. The user uses the Docker client to interact with the daemon.
- ▶ The Docker Client
  - The Docker client, in the form of the docker binary, is the primary user interface to Docker.
  - It accepts commands and configuration flags from the user and communicates with a Docker daemon.

# Docker Architecture



## **Image**

The basis of a Docker container. The content at rest.



## **Container**

The image when it is 'running.' The standard unit for app service



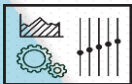
## **Engine**

The software that executes commands for containers. Networking and volumes are part of Engine. Can be clustered together.



## **Registry**

Stores, distributes and manages Docker images



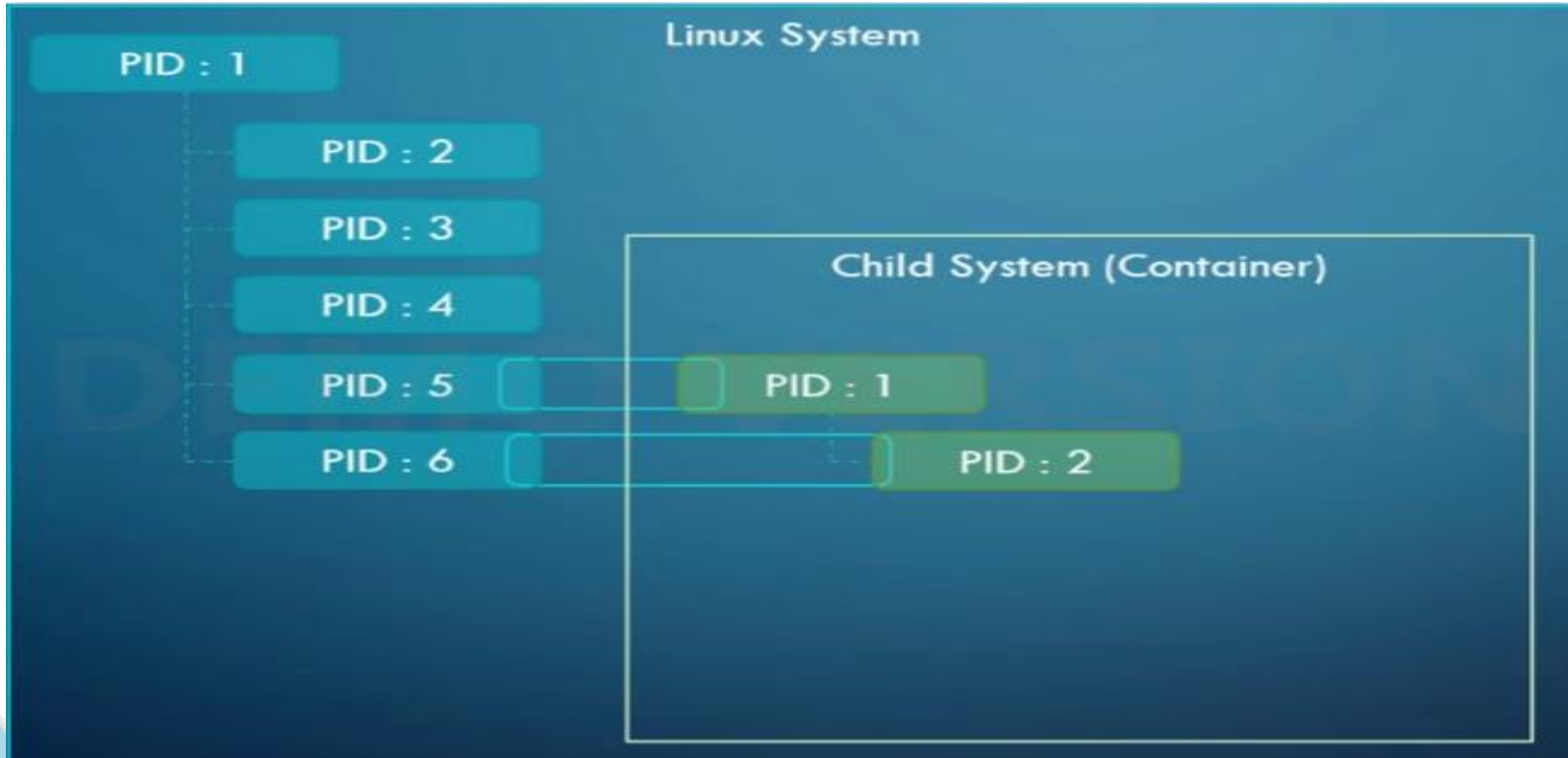
# Docker Architecture

- Lets understand how does applications works in isolation under the hood.
- Docker uses namespace which is a nothing but an isolated environment like VM, but on top of VM called container.



# Docker Architecture

- The demonstration will be given once we understand the container operations.



# Docker Images

- ▶ A Docker image is a read-only template with instructions for creating a Docker container.
- ▶ For example, an image might contain an Ubuntu operating system with Apache web server and your web application installed. You can build or update images from scratch or download and use images created by others.
- ▶ A docker image is described in text file called a Dockerfile, which has a simple, well-defined syntax.
- ▶ Docker images are the build component of Docker.

# Docker Containers

- ▶ A Docker container is a running instance of a Docker image.
- ▶ You can run, start, stop, move, or delete a container using Docker API or CLI commands.
- ▶ When you run a container, you can provide configuration metadata such as networking information or environment variables.
- ▶ Each container is an isolated and secure application platform, but can be given access to resources running in a different host or container, as well as persistent storage or databases.
- ▶ Docker containers are the run component of Docker.



# Docker Registries

- ▶ A docker registry is a library of images.
- ▶ A registry can be public or private, and can be on the same server as the Docker daemon or Docker client, or on a totally separate server.
- ▶ Docker registries are the distribution component of Docker.
- ▶ “Docker Hub” is known as global registry.

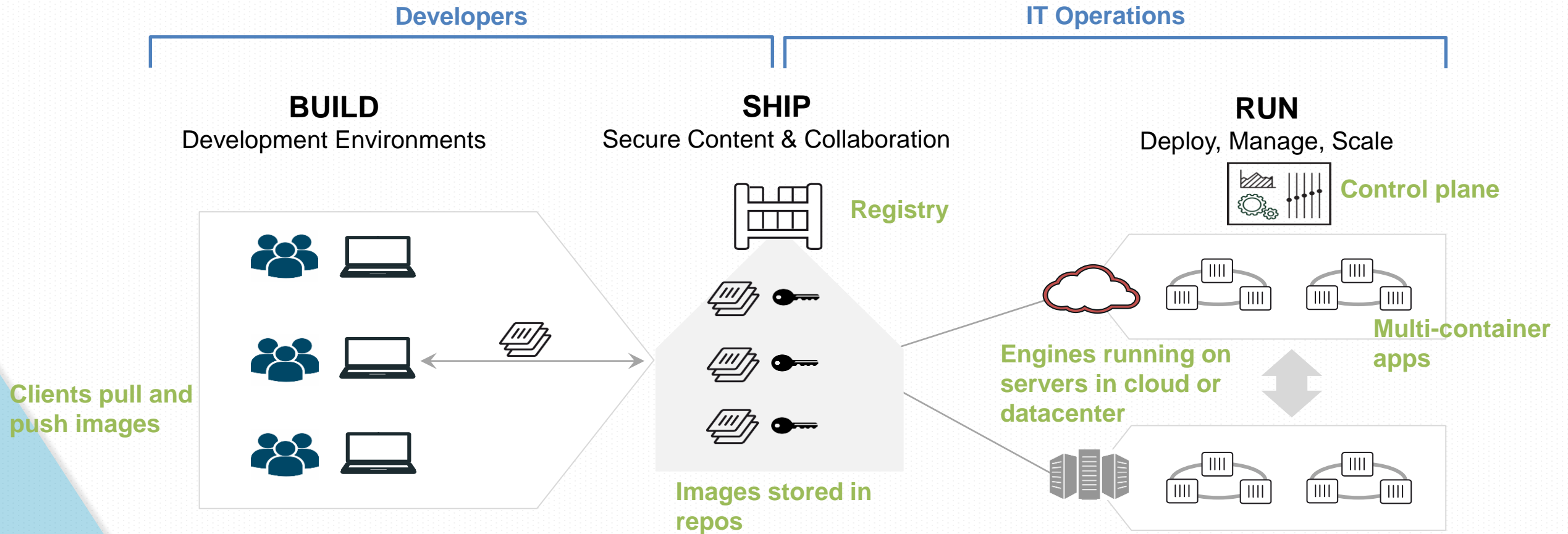
# Docker Orchestration

- Three tools for orchestrating distributed application with Docker
- Docker Machine
  - This is a tool that provisions Docker hosts and install the Docker Engine on them
- Docker Swarm
  - This is a tool that clusters multiple Docker Engines and do the Scheduling of containers
- Docker Compose
  - This is a tool that create and manage multi-container application i.e combine different container for an application

# Docker Features

- Lightweight
  - Containers running on a single machine all share the same operating system kernel so they start instantly and make more efficient use of RAM. Images are constructed from layered filesystems so they can share common files, making disk usage and image downloads much more efficient.
- Open
  - Docker containers are based on open standards allowing containers to run on all major Linux distributions and Microsoft operating systems with support for every infrastructure.
- Secure
  - Containers isolate applications from each other and the underlying infrastructure while providing an added layer of protection for the application.

# Container as a Service



# Session: 4

## Containers

# How Container works

- ▶ A container uses the host machine's Linux kernel, and consists of any extra files you add when the image is created, along with metadata associated with the container at creation or when the container is started.
- ▶ Each container is built from an image.
- ▶ The image defines the container's contents, which process to run when the container is launched, and a variety of other configuration details.
- ▶ The Docker image is read-only. When Docker runs a container from an image, it adds a read-write layer on top of the image (using a UnionFS) in which your application runs.

# How Container works

- ▶ When you use the "docker run" CLI command, the Docker Engine client instructs the Docker daemon to run a container.

```
docker run ubuntu ps ax
```

- ▶ This example tells the Docker daemon to run a container using the centos Docker image, to remain in the foreground in interactive mode (-i), provide a tty terminal (-t) and to run the /bin/bash command.

```
docker run -i -t centos /bin/bash
```

Because if you exit the current running process /bin/bash (pid 1), container will stop/exit. Use "Ctrl pq" to safe exit without stopping the container.

# How Container works

```
[root@TechLanders libcontainerd]# docker run -i -t centos /bin/bash
Unable to find image 'centos:latest' locally
latest: Pulling from library/centos
d9aaf4d82f24: Pull complete
Digest: sha256:eba772bac22c86d7d6e72421b4700c3f894ab6e35475a34014ff8de74c10872e
Status: Downloaded newer image for centos:latest
[root@9a06b1a61fc5 /]#
```

```
[root@TechLanders overlay]# ls -lrt /var/lib/docker/image/overlay2/repositories.json
-rw-----. 1 root root 545 Sep 18 03:17 /var/lib/docker/image/overlay2/repositories.json
```



# How Container works

```
[root@TechLanders overlay]# cat repositories.json
```

```
{"Repositories":{"centos":{"centos:latest":"sha256:196e0ce0c9fbb31da595b893dd39bc9fd4aa78a474bbdc21459a3ebe855b7768","centos@sha256:eba772bac22c86d7d6e72421b4700c3f894ab6e35475a34014ff8de74c10872e":"sha256:196e0ce0c9fbb31da595b893dd39bc9fd4aa78a474bbdc21459a3ebe855b7768"},"hello-world":{"hello-world:latest":"sha256:05a3bd381fc2470695a35f230afefd7bf978b566253199c4ae5cc96fafa29b37","hello-world@sha256:1f19634d26995c320618d94e6f29c09c6589d5df3c063287a00e6de8458f8242":"sha256:05a3bd381fc2470695a35f230afefd7bf978b566253199c4ae5cc96fafa29b37"}}}}
```

```
[root@TechLanders overlay]# docker image list
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
centos	latest	<b>196e0ce0c9fb</b>	3 days ago	197MB
hello-world	latest	05a3bd381fc2	5 days ago	1.84kB

```
[root@TechLanders overlay]# docker image inspect centos
```

# More Useful - Container

- Do something in our container:

Lets suppose we try to use “talk” for communication.

- Let’s check how many packages are installed:

```
rpm -qa | wc -l
```

- Install a package in our container

```
yum -y update  
yum install talk
```

# A non interactive - Container

- ▶ In your Docker environment, just run the following command:

```
docker run jpetazzo/clock
```

- This container just displays the time every second.
- This container will run forever.
- To stop it, press ^C.
- Docker has automatically downloaded the image jpetazzo/clock.

# Run in background - Container

- ▶ Containers can be started in the background, with the -d flag (daemon mode):

```
docker run -d jpetazzo/clock
```

- We don't see the output of the container.
- But don't worry: Docker collects that output and logs it!
- `docker ps -a`
- `docker logs <container-id>`
- Docker gives us the ID of the container.

# List Running Containers

- ▶ With `docker ps`, just like the UNIX `ps` command, lists running processes.

`docker ps`

`docker ps -l`

`docker ps -a`

`docker ps -q`

- The (truncated) ID of our container.
- The image used to start the container.
- That our container has been running (Up) for a couple of minutes.
- Now, start multiple containers and use “`docker ps`” to list them.

# Stop our Container

- There are two ways we can terminate our detached container.
  - Killing it using the docker “kill” command.
  - Stopping it using the docker “stop” command.
- The first one stops the container immediately, by using the KILL signal.
- The second one is more graceful. It sends a TERM signal, and after 10 seconds, if the container has not stopped, it sends KILL.

# Removing Container

- Let's remove our container:

```
docker rm <yourContainerID>
```

# Attaching to a Container

- You can attach to a container:  
  
    `docker attach <containerID>`
- The container must be running.
- *There can be multiple clients attached to the same container.*



# Restarting a Container

- When a container has exited, it is in stopped state.
- It can then be restarted with the “start” command.

`docker start <containerID>`

- The container must be running.
- You can also use restart command  
`docker restart <container-id>`

# LAB1

- Create one Ubuntu Container in interactive and terminal mode:
- `docker run -it ubuntu`
- Stop the container, here we have one option to execute “exit”.
- Check the status of your container
- `docker ps -a`
- Create a new container, interactive and terminal mode
- `docker run -it ubuntu`

# LAB1

- Come out of it, this time without exit, user escape sequence control+P+Q
- From host stop your container
- `docker stop <container-ID>`
- Check the details status of your container and start it again.
- `docker ps -a`
- `docker start <container-id>`
- Try to access the container and then come out it and exit
- `docker attach <container-ID>`
- Exit

# LAB1

- Create a new container using ( -d ) option.
- Docker run -d centos (THIS WILL NOT RUN)
- Docker run -dt centos (YOU NEED TO PROVIDE A TERMINAL HERE) → THIS IS TRUE WITH BASE IMAGES
- Now check the difference using docker ps -a command
- Try to access this container
- Docker attach <container-ID> (THIS WILL NOT WORK) (BECAUSE WE ARE NOT USING INETRACTIVE IN THE CREATION)
- Use docker exec -it <CONATINER-ID> /bin/bash
- NOTE: Alternatively you could have used “docker run -dit centos and accessed using docker attach <Conatiner-ID>
- Remove everything
- Docker stop
- Docker rm

# LAB2

- *On your host machine install httpd package*
  - `Yum install httpd -y`
- *Verify installation:*
  - `rpm -qa | grep -i httpd`
- *Create a container using centos:*
  - `docker run -it centos`
- *Check the httpd package inside it:*
  - `rpm -qa | grep -i httpd`
- *Install ntp package*
  - `yum install ntp`
- *Logout from container and check the ntp package inside host machine*
  - `rpm -qa | grep -I ntp`

Learning - Containers have isolated independent root filesystem

# Session: 5

## Docker – Containers Advanced

# Checking Info

*Checking System usage:*

\$ docker system df

// use **-v** for verbose run

*Checking Docker Information:*

\$ docker system info

*Checking Docker events:*

\$ docker system events

//Checking runtime events

Checking Logs:

\$ docker logs {containerid}

*Cleaning up Unused resources:*

\$ docker system prune

//use **-a** for deleting all unused images **-volume** for deleting volume

# Container

- *Set a Name to container*  
docker run --name my-redis -d redis
- *Setting a hostname from outside:*  
docker run -dit --name container1 --hostname container1 centos
- *Finding stats for container:*  
  
docker top container-name  
docker stats
- *Inspecting Docker containers*  
  
docker inspect container-name
- *Homedirectory*  
  
/var/lib/docker



# Resource binding to a Container

*Limiting memory(Quota):*

```
$ docker run -it -m 300M ubuntu:14.04 /bin/bash
```

Setting Memory reservation

```
docker run -it -m 500M --memory-reservation 200M ubuntu:14.04 /bin/bash
```

*Check Reservation using docker stats command and docker inspect {container-id} command*

*Limiting CPUs:*

```
docker run -it --cpus 0.02 ubuntu:14.04 /bin/bash
```

# **Session: 5**

## **Ports and Volumes**

# Working with Volumes

- Containers have ephemeral storage
- Data shared within the container is not accessible after it is terminated
- Volumes provide persistence to containers
- Each volume is implicitly or explicitly mapped to a host directory
- Method1: (Default Mount Method or Volume Mount)

`docker volume create data`

`docker run -v data:/data <image>`

**Note:** data is a volume present under `/var/lib/docker/volumes/data`, but will happen if we don't create volume explicitly and use `docker run -c data1:/data1 <image>`. Docker will automatically create a volume for you!.

- Method2: (Bind Mounting Method)

`docker run -d -v /var/tmp/data:/data <image>`

`docker volume ls`

# Working with Volumes

- Question is who is responsible for performing all these operations in Docker.
- And the answer is “Storage Drivers”, some of the common drivers are like “aufs, zfs, overlay, overlay2, device mapper etc.)
- The choice of device driver depends upon the underlying OS.
- “aufs” is the default Storage Driver for Ubuntu, however this is not present in Centos, in such cases “device mapper” is a better option to go with.
- The key part is “docker will pick the best storage driver for you automatically”.
- Execute `docker info` | more to know about the drivers used by the docket for your host.

# Working with Volumes

- Volumes can be declared in two different ways.
  - Within a Dockerfile, with a VOLUME instruction.  
VOLUME /uploads
  - On the command-line, with the -v flag for “docker run”.  
docker run -d -v uploads:/uploads <myapp-image>

# Working with Volumes

- Add storage

```
docker volume ls
```

```
docker volume create data
```

```
docker volume ls
```

```
docker run -p 80:80 --name web1 -d -v data:/data httpd
```

OR

```
docker run --name web1 -dt -v data:/data ubuntu
```

```
docker inspect web
```

```
docker exec -it web /bin/bash
```

- Inside container run below commands to check volumes

```
cd /data
```

```
echo "Creating file from container" > data
```

```
exit
```

- Now kill this container and launch a new container

```
docker rm -f yogeshweb
```

```
docker run -it -v data:/data ubuntu /bin/bash
```

# Working with Volumes

- Add volume from a directory from HOST

```
mkdir htdocs
```

```
docker run -p 80:80 --name yogeshweb -dt -v $PWD/htdocs:/usr/local/apache2/htdocs httpd
```

OR

```
docker run -dt --name test -v $PWD/yogesh:/yogesh ubuntu
```

```
docker exec -it yogeshweb /bin/bash
```

- Now run below commands within the containers

```
echo '<h1> Hello World from Container itself</h1>' > index.html
```

```
exit
```

- Test it

```
echo '<h1> Hello World from host itself</h1>' > index.html
```

```
docker rm -f web
```

# Listing Volumes

- If a container is stopped, its volumes still exist and are available.
- Since Docker 1.9, we can see all existing volumes:  
docker volume ls



# Docker Ports and Volumes

- Two most important things in docker are Ports and Volumes.
- docker images
- docker run -d nginx:latest
- docker ps
- docker inspect <container-id>| grep -i ip
- elinks http://<IP>
- elinks http://localhost ==> No output
- Because nothing exposed to the localhost
- docker ps
- docker stop <container-id>
- docker rm `docker ps -a -q`

# Docker Ports and Volumes

- Lets expose the containers to a random port by docker itself
- `docker run -d --name=nginxserver -P nginx:latest`
- `docker ps` (you will get a port mapped by the docker for you)
- `elinks http://<container-IP>`
- `elinks http://localhost:<mapped port>`
- `docker port <container-name> $CONTAINERPORT`
- `docker stop server1`

# Docker Ports and Volumes

- But how to bound the container with a particular port:
- `docker run -d -p 8080:80 nginx`
- Even you can bound multiple ports:
- `docker run -d -p 8080:80,8081:443 nginx`
- `elinks http://<container-ip>`
- Once done stop and remove the container

# Docker Ports and Volumes

- Actual (Bind Mounting Way):
  - `cd /var/tmp`
  - `mkdir www`
  - `cd www`
  - `vi index.html`
- `<HTML>`
- `<HEAD></HEAD>`
- `<BODY>`
- `<HR/>`
- `THIS IS A TEST PAGE CREATED BY YOGESH`
- `<HR/>`
- `</BODY>`
- `</HTML>`

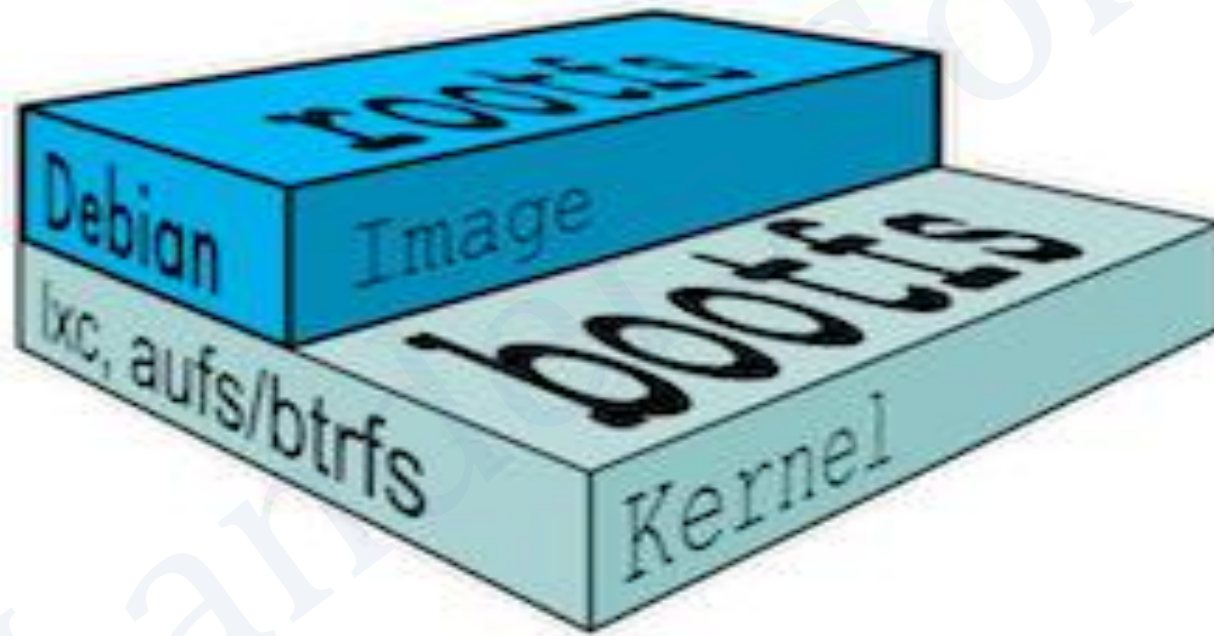
# Docker Ports and Volumes

- `docker run -d -p 8080:80 --name=server1 -v /var/tmp/www:/usr/share/nginx/html nginx:latest`
- `docker ps`
- `elinks http://localhost:8080`
- CHANGE THE INDEX file and see the results
- `elinks http://localhost:8080`
- We don't have to do anything now inside container, just deal with the VOLUMES mounted outside the container

# Session: 6

## Docker - Images

# Docker Images



# What is an image?

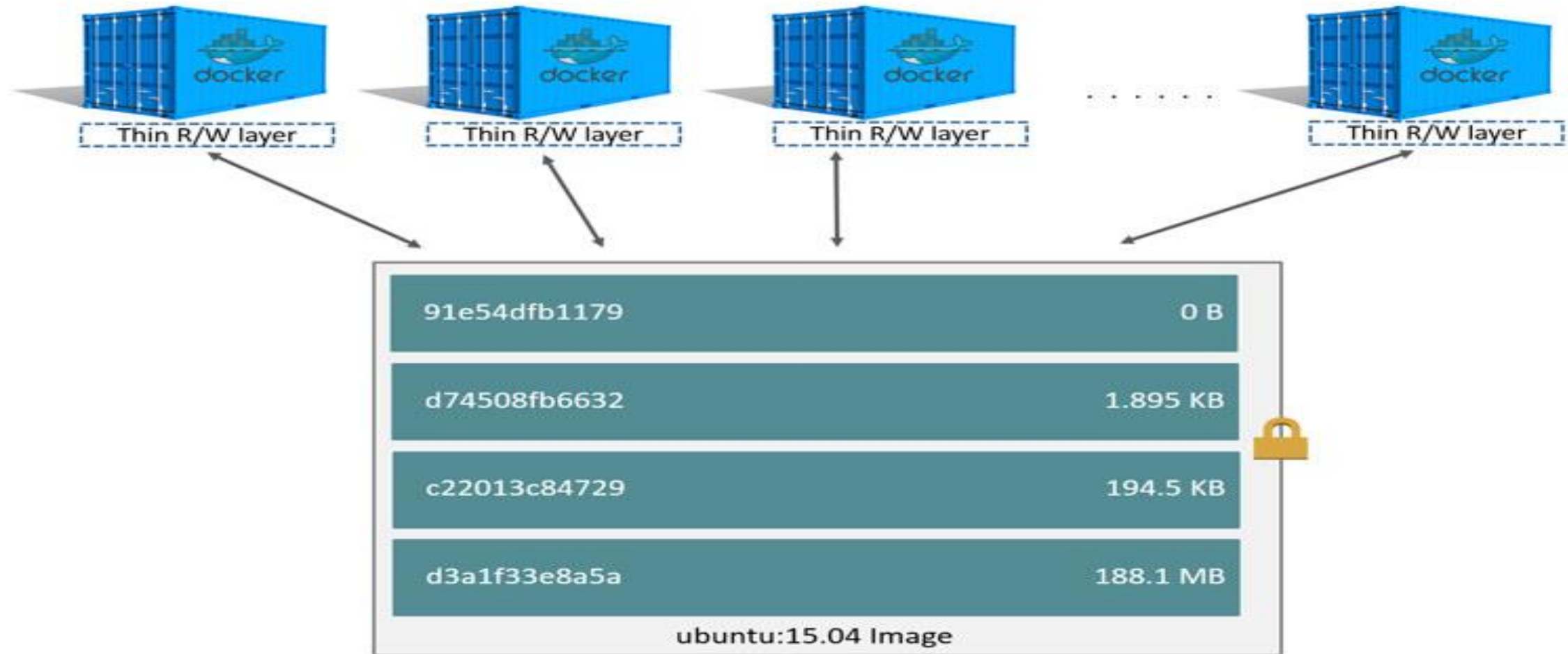
- An image is a collection of files + some meta data.
- Images are made of *layers, conceptually stacked on top of each other*.
- Each layer can add, change, and remove files.
- Images can share layers to optimize disk usage, transfer times, and memory use.
- Example:
  - CentOS
  - JRE
  - Tomcat
  - Dependencies
  - Application JAR
  - Configuration



# Container vs Image

- An image is a read-only filesystem.
- A container is an encapsulated set of processes running in a read-write copy of that filesystem.
- docker run starts a container from a given image.
- Its “A chicken-and-egg problem”.
- The only way to create an image is by "freezing" a container.
- The only way to create a container is by instantiating an image.

# Container vs Image



# Store & manage images

- Images can be stored:
  - On your Docker host.
  - In a Docker registry.
- You can use the Docker client to download (pull) or upload (push) images.
- To be more accurate: you can use the Docker client to tell a Docker server to push and pull images to and from a registry.
- Lets explore docker public registry called “docker hub”

# Showing current images

- Let's look at what images are on our host now.

docker images

docker image list

# Searching for images

- We cannot list all images on a remote registry, but we can search for a specific keyword:

docker search zookeeper

- "Stars" indicate the popularity of the image.

# Downloading images

- There are two ways to download images.
  - Explicitly, with “docker pull”.
  - Implicitly, when executing “docker run” and the image is not found locally.

- Pulling an image.

`docker pull debian:jessie`

- Images can have tags.
- Tags define image versions or variants.
- “docker pull ubuntu” will refer to “ubuntu:latest”.
- The :latest tag is generally updated often.

# Docker Image LifeCycle

- To list docker images: `docker images`
- To remove docker images locally: `docker rmi <image-name>`
- Special Cases:
- Lets try to remove docker images used by current container: `docker rmi <image-name>`
- This time you will get an error, but the image can be removed forcefully using “-f” option.
- `Docker rmi -f <image-name>` (Note: It wont work with Image ID)

# Docker Image Information

- Docker Image details:
- docker image list
- Detailed Information about docker image:
- `ls -lrt /var/lib/docker/image/overlay2/imagedb/content/sha256`
- High level certificate sign information (SHA):
- `cat /var/lib/docker/image/overlay2/repositories.json`
- All details with command line:
- `docker image inspect <image-name>`
- Even you can check all of the information about the docker image:
- `docker history <image-id>`



# Docker Image and Container LifeCycle

- To list docker: `docker ps -a`
- To list docker images: `docker images`
- To remove container: `docker rm <container-id> or <Name>`
- To remove multiple container: `docker rm `docker ps -a -q``
- To remove docker images locally: `docker rmi <image-name>`

# Docker Image and Container LifeCycle

- Special Cases:
- Lets try to remove docker images used by current container: `docker rmi <image-name>`
- This time you will get an error, but the image can be removed forcefully using “-f” option.
- `Docker rmi -f <image-name>` (Note: It wont work with Image ID)
- The best part is though you have removed the image forcefully, but this will not impact the current container as the container preserves the metadata in containers folder.
- You containers are safe!.

# Session: 7

## Building Images

# Building Images Interactively

- Let's have a Use Case:
  - We will build an image that has httpd.
  - First, we will do it manually with docker commit.
  - Then, we will use a Dockerfile and “docker build”.

# Create a new container

- Let's start from base image "centos":

```
docker run -it centos
```

```
yum install -y httpd
```

```
exit
```

- Inspect the changes:

```
docker diff <yourContainerId>
```

- Commit the changes:

```
docker commit <yourContainerId>
```

Real way:

```
docker commit -m "Added HTTPD" -a "Gagandeep Singh" 1bb937745e8e gagandeep/centostesting:v1
```

# Run & Tag the image

- Let's run the new images:  
`docker run -it <newImageId>  
rpm -qa | grep -i httpd`
- Tagging images:  
`docker tag <newImageId> newhttpd`
- Run it using Tag:  
`docker run -it newhttpd`

# Dockerfile overview

- A Dockerfile is a build recipe for a Docker image.
- It contains a series of instructions telling Docker how an image is constructed.
- The “docker build” command builds an image from a Dockerfile.

# First Dockerfile

- Create a directory to hold our Dockerfile.

```
mkdir myimage
```

- Create a Dockerfile inside this directory.

```
cd myimage
```

```
vim Dockerfile
```

- Write below in our Dockerfile

```
FROM centos
```

```
RUN yum update -y
```

```
RUN yum -y install httpd
```



# First Dockerfile

- “FROM” indicates the base image for our build.
- Each “RUN” line will be executed by Docker during the build.
- Our “RUN” commands **must be non-interactive**.
- No input can be provided to Docker during the build.

# RUN – collapsing layers

- It is possible to execute multiple commands in a single step:

RUN apt-get -y update && apt-get install -y wget && apt-get clean

# First Dockerfile

- Build the Dockerfile:

```
docker build -t httpd .
```

- -t indicates the tag to apply to the image.
- . indicates the location of the Directory of Dockerfile.

# Run & Tag the image

- Let's run the new images:  
docker run -it <newImageId>  
rpm -qa | grep -i httpd

# Using Image & viewing history

- The history command lists all the layers composing an image.
- For each layer, it shows its creation time, size, and creation command.
- When an image was built with a Dockerfile, each layer corresponds to a line of the Dockerfile.

`docker history httpd`

# Using JSON Syntax - Dockerfile

- Change our Dockerfile:  
FROM centos  
RUN yum update  
RUN ["yum", "install", "-y", "httpd"]
- Now, Build & run it.

# Dockerfile

- Dockerfile:

```
FROM centos
```

```
MAINTAINER Sandeep Sandeep.kumar@TechLanders.com
```

```
RUN mkdir /data
```

```
RUN yum -y install httpd php
```

```
RUN echo " TechLanders Solutions Deals in DevOps and Cloud" > /var/www/html/index.html
```

```
EXPOSE 80
```

```
VOLUME /data
```

```
RUN echo "httpd" >> /root/.bashrc
```

```
CMD ["/bin/bash"]
```

- Build the image:

```
docker build -t TechLandersdevops .
```

# COPY Instruction

- For Use Case, let's build a container that copy file from localhost

```
RUN echo " TechLanders Solutions Deals in DevOps and Cloud" >  
/var/www/html/index.html
```

Dockerfile content

```
FROM centos  
RUN yum -y install httpd  
COPY ./index.html /var/www/html/index.html  
EXPOSE 80  
CMD ["httpd","-D","FOREGROUND"]
```



# COPY Instruction

- For Use Case, let's build a container that compiles a basic "Hello world" program in C.
- hello.c

```
[root@TechLanders yogesh]# cat hello.c
#include<stdio.h>
int main () {
    puts("Hello, TechLanders!");
    return 0;
}
```

- Dockerfile

```
[root@TechLanders yogesh]# cat Dockerfile
FROM ubuntu
RUN apt-get update
RUN apt-get install -y build-essential
COPY hello.c /
RUN make hello
CMD /hello
```

**Note:** Using COPY keyword we can copy the files from Docker Host to a container.

# COPY Instruction

- `[root@TechLanders yogesh]# docker build -t fifthfile .`
- `[root@TechLanders yogesh]# docker run -it fifthfile /bin/bash`
- `root@519a9d815a29:/# ls -lrt /hello`
- `-rwxr-xr-x. 1 root root 8600 Sep 18 11:54 /hello`
- `root@519a9d815a29:/# /hello`
- `Hello, world!`
- `root@519a9d815a29:/#`

# Environment Setting

FROM debian:stable

RUN apt-get update && apt-get install -y apache2

**ENV MYVALUE Sandeep Kumar**

EXPOSE 80

CMD ["/usr/sbin/apache2ctl","-D","FOREGROUND"]

# Real Example-2

- vi Dockerfile
- FROM debian:stable
- MAINTAINER Gagandeep <gagandeep.singh@TechLanders.com>
- RUN apt-get update && apt-get upgrade -y && apt-get install -y apache2 telnet elinks openssh-server
- ENV MYVALUE gagandeep
- EXPOSE 80
- CMD ["/usr/sbin/apache2ctl","-D","FOREGROUND"]
- docker build -t yogeshraheja/myapache .
- docker run -d -p 8989:80 <image>
- docker ps

# Real Example-2

- `docker inspect <container> | grep -i ipad`
- `elinks http://<IP>` (if not given dedicated IP) OR Access via public IP
- `docker exec -it <container> /bin/bash`
- `ps aux | grep -i apache`
- `echo $MYVALUE`

# Real Example-3

- vi Dockerfile
- FROM debian:stable
- MAINTAINER Gagandeep <gagandeep.singh@TechLanders.com>
- RUN apt-get update && apt-get upgrade -y && apt-get install -y apache2 telnet elinks openssh-server
- ENV MYVALUE yogesh-raheja
- EXPOSE 80
- CMD ["/usr/sbin/apache2ctl","-D","FOREGROUND"]
- docker build -t yogeshraheja/myapache .
- docker run -d -p 8989:80 <image>
- docker ps

# Docker Restart Policy

```
$ docker run -dit --restart unless-stopped centos
```

Flag	Description
no	Do not automatically restart the container. (the default)
on-failure	Restart the container if it exits due to an error, which manifests as a non-zero exit code.
always	Always restart the container if it stops. If it is manually stopped, it is restarted only when Docker daemon restarts or the container itself is manually restarted. (See the second bullet listed in <a href="#">restart policy details</a> )
unless-stopped	Similar to always, except that when the container is stopped (manually or otherwise), it is not restarted even after Docker daemon restarts.

# Docker Registries

Local Registry (Local to Host)

Remote Registry (Private)

Global Registry (Public)



# Docker Trusted Registry

- Enterprise-grade image storage solution from Docker
- Highly Secure
- Image and job management with CICD
- HA Availability
- Efficiency with Near to user storage and bandwidth sharing
- RBAC
- Security Scanning
- Similar tools in market Sonatype Nexus (open source), AWS ECR (PaaS), Azure Container Registry, GCP container Registry etc

# Docker Trusted Registry

- Enterprise-grade image storage solution from Docker
- Highly Secure
- Image and job management with CICD
- HA Availability
- Efficiency with Near to user storage and bandwidth sharing
- RBAC
- Security Scanning
- Similar tools in market Sonatype Nexus (open source), AWS ECR (PaaS), Azure Container Registry, GCP container Registry etc

# Docker Compose

- Compose is a tool for defining and running multi-container Docker applications
- Write syntax in yaml - docker-compose.yml to configure the services

```
version: '2.0'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - ./code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

# Docker Swarm

- Clustering (basically Orchestration) of Docker Hosts
- Provide High availability for node failure and container failures
- Load balancing and service distribution
- Role based access control
- Scalability

# Universal Control Plane

- Enterprise-grade cluster management solution from Docker
- Centralized cluster management
- Deploy, manage, and monitor via GUI
- Built-in security and access control
- Integrated with DTR
- Can utilize CLI

Note: Better tools available in market like Rancher, Openshift

# Docker Limitations

- Hardware Issues? High Availability?
- How IP address will be managed for failover?
- Scaling?
- Auto Healing?
- Autoscaling?
- No Application Management - Only Containerization
- Updation of application/management