BITS Pilani

Pilani | Dubai | Goa | Hyderabad

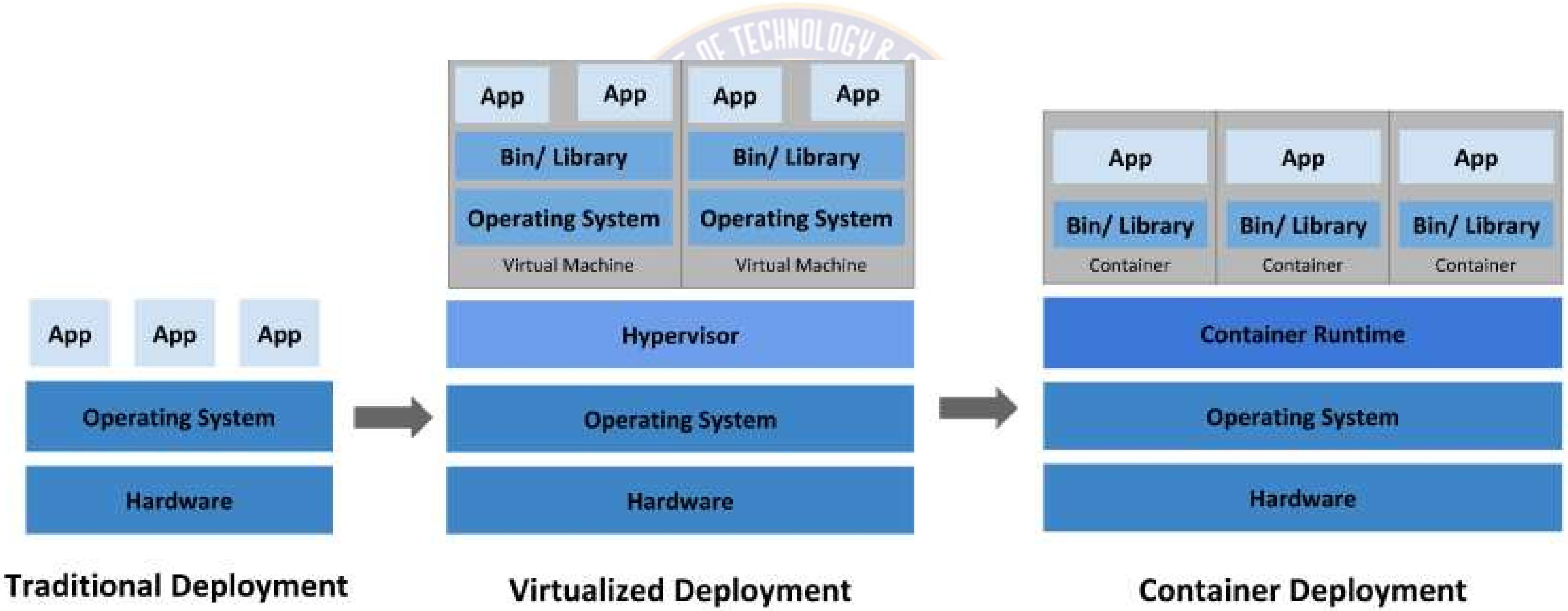
Containers

Aditya Goel

Agenda

- **Containers**
  - ✓What are containers?
  - ✓Containers – History
  - ✓Cgroups
  - ✓Namespaces
  - ✓Virtual Machine vs Containers
  - ✓Containers and Virtual Machines
  - ✓Types of Containers
  - ✓Dockers

Going back in time

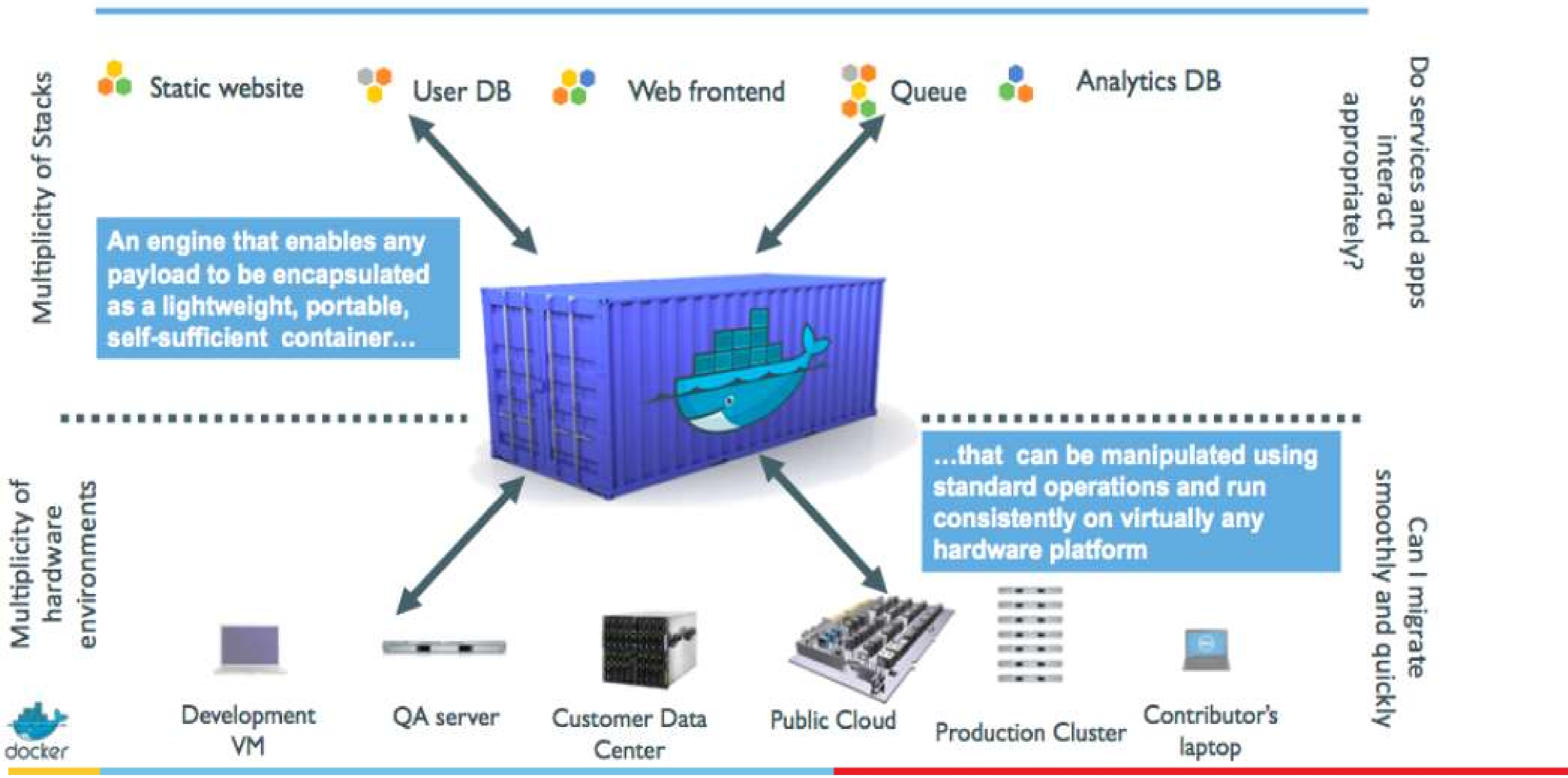


What are Containers?

- A software container is a standardized package of software.
- Everything needed for the software to run is inside the container.
- The software code, runtime, system tools, system libraries, and settings are all inside a single container
- Container is a semi-isolated execution environment
- Managed by the OS kernel running on the host system
- Has its own isolated memory, CPU, storage, process table, and networking interfaces

What are Containers?

A shipping container system for applications



Multiplicity of Stacks

Static website

User DB

Web frontend

Queue

Analytics DB

Do services and apps interact appropriately?

An engine that enables any payload to be encapsulated as a lightweight, portable, self-sufficient container…

…that can be manipulated using standard operations and run consistently on virtually any hardware platform

Multiplicity of hardware environments

Development VM

QA server

Customer Data Center

Public Cloud

Production Cluster

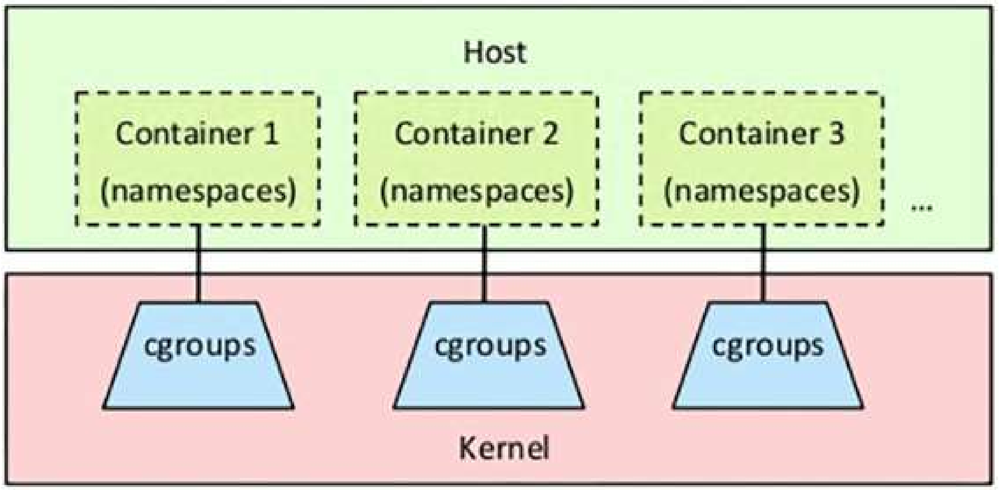
Contributor's laptop

Can I migrate smoothly and quickly

docker

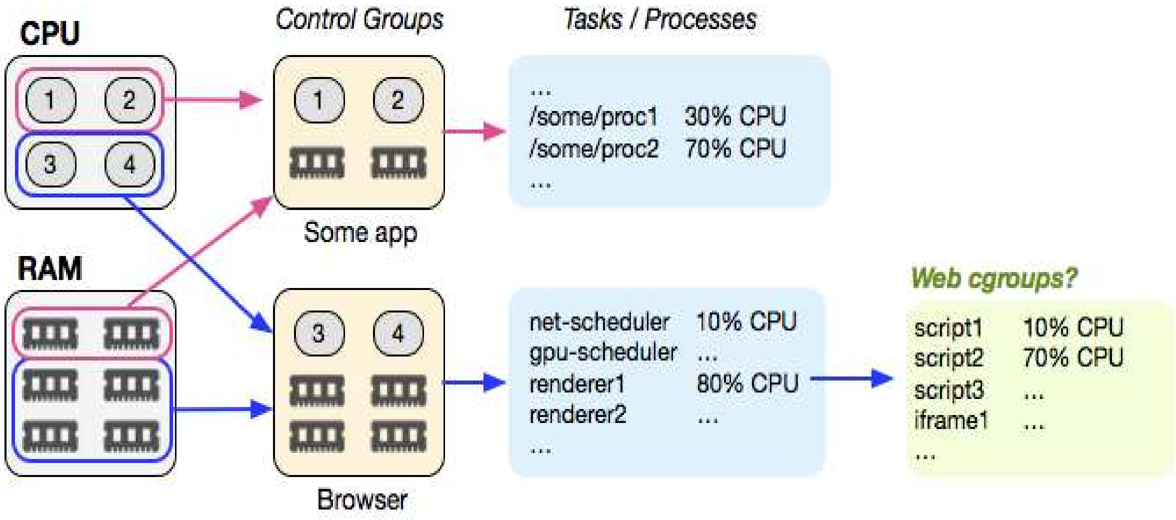
Containers - History

- Containers are powered by two underlying Linux Kernel technologies:
  - cgroups
  - namespaces



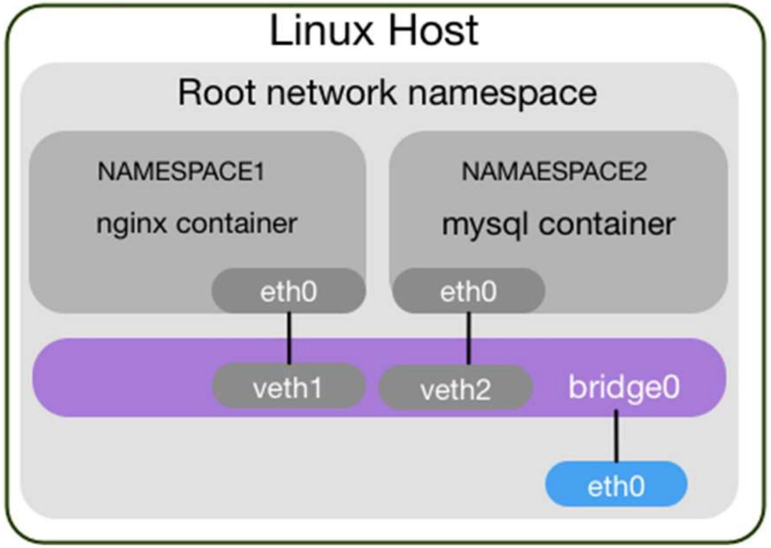
cgroups

- cgroups – Control groups
  - A kernel mechanism for limiting and measuring the total resources used by a group of processes running on a system
  - Processes can be applied with CPU, memory, network or IO quotas
- Cgroup merged into Linux 2.6.24
- Cgroups provides:
  - **Resource limiting**
  - **Prioritization**
  - **Accounting**
  - **Control**

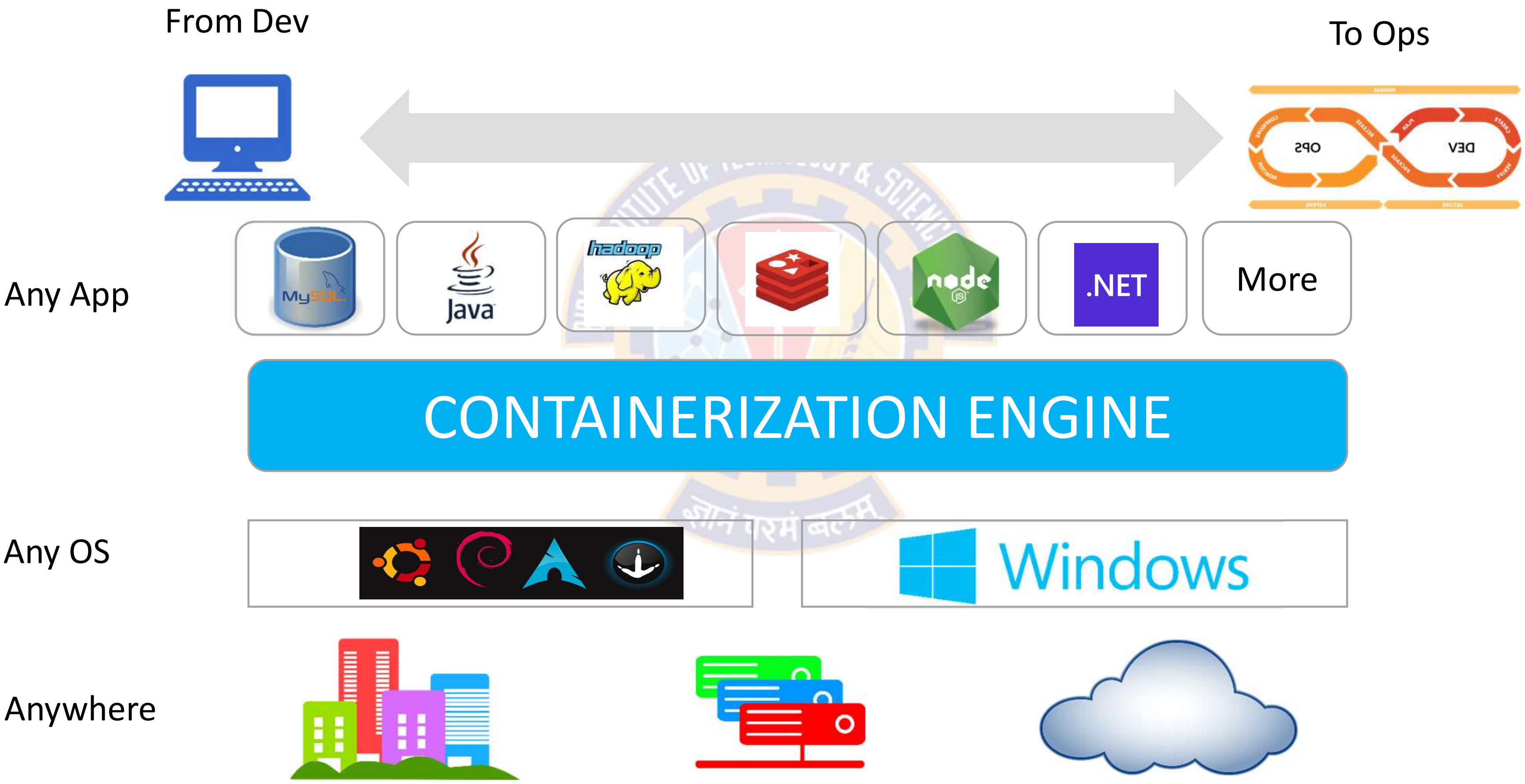


Namespaces

- Namespaces - kernel mechanism for limiting the visibility that a group of processes has of the rest of a system
- Limit visibility
  - Certain process trees
  - Network interfaces
  - User IDs
  - Filesystem mounts
- Namespace merged into Linux 3.8



Build, Ship, Run, Any App Anywhere



From Dev

To Ops

Any App

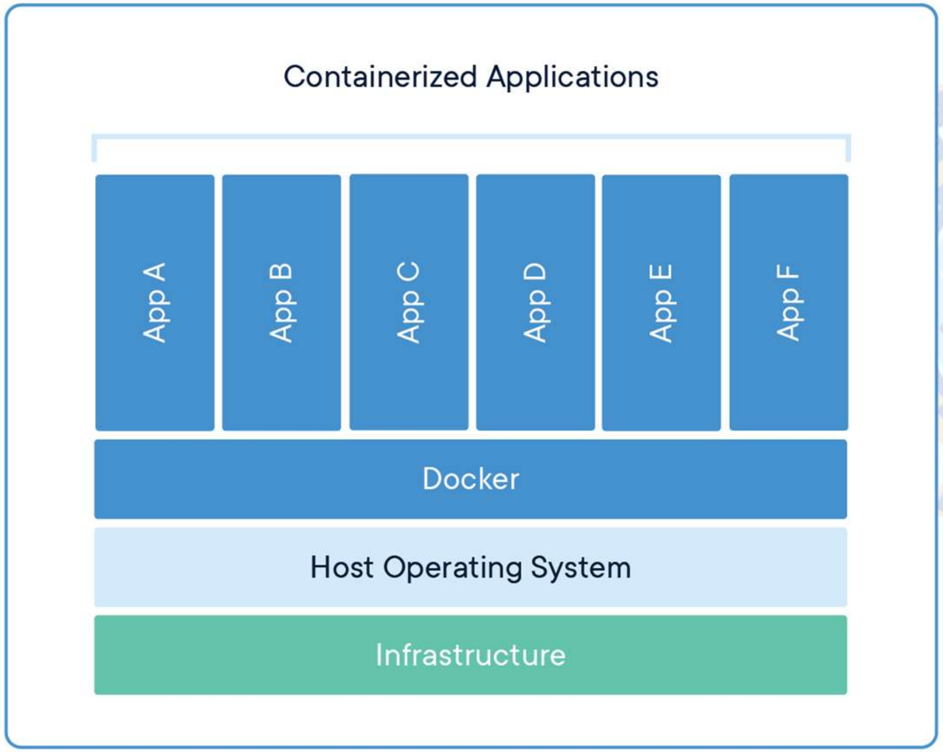
More

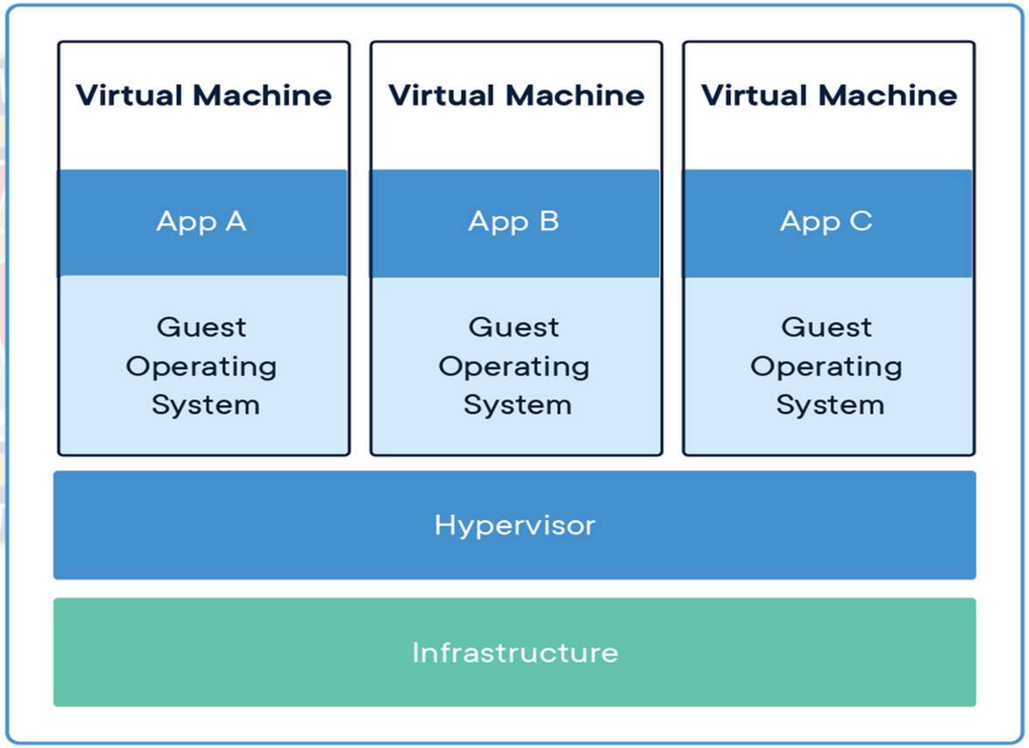
CONTAINERIZATION ENGINE

Any OS

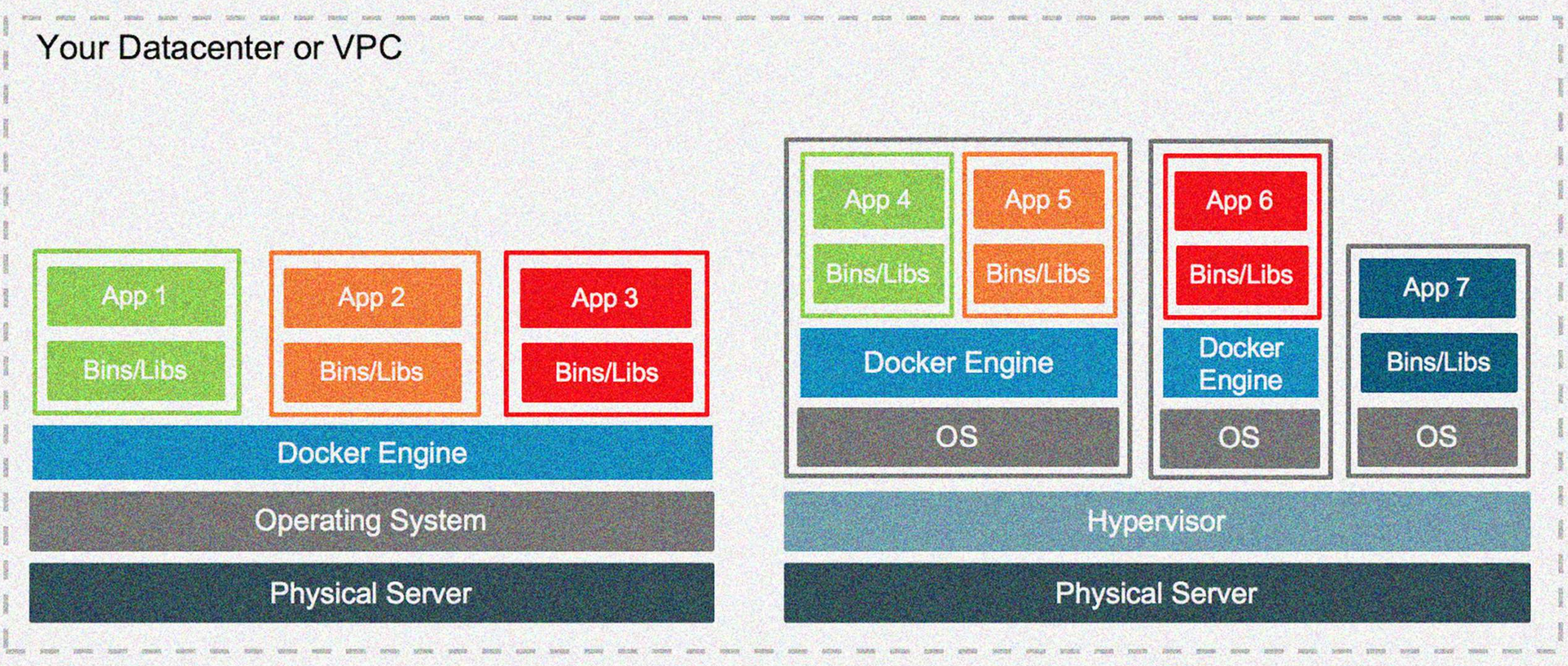
Anywhere

Virtual Machine vs Containers





Containers and Virtual Machines



Types of Containers

- System Containers
- Application Containers

System Containers

- Similar to Virtual or physical machine
- They run a full operating system
- Managing a System Container is as good as managing any virtual or physical machine
  - install packages inside them
  - manage services
  - define backup policies
  - Monitoring
- These containers can be updated – using normal tooling
- They get system and security updates as any other virtual or physical machine
- Oldest type of container - 1999

System Containers - History

- BSD introduced jails, a way of running a second BSD system on the same kernel as the main system
- Linux followed the concept through Linux vServer
- Then Solaris – Zones
- Then OpenVZ project – implemented VPS(Virtual Private Servers) on Linux
- LXC – Linux Containers – mainline Linux implementation
- LXC is a low-level tool that can create both system containers and application containers
- Docker was initially based on LXC
- Goal of LXC: to create an environment as close as possible to a standard Linux installation but without the need for a separate kernel

System Containers - History

- What is LXD?
  - Is a system container and a virtual machine manager
  - Runs on top of LXC
  - Enhances the experience and enabling easier control and maintenance.
  - LXD is image-based and provides images for a wide number of different Linux distributions.
  - A command-line tool that enables easy management to the instances (Containers and VMs)

System Containers - History

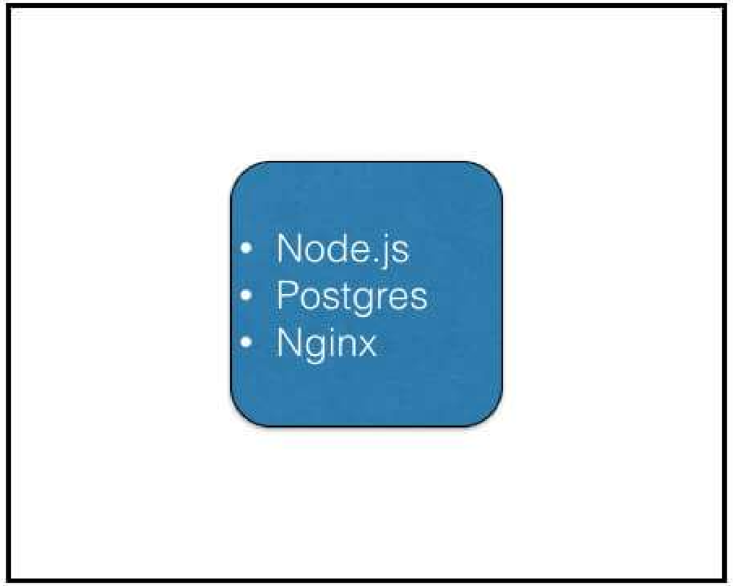
- LXD vs LXC

LXC	LXD
Linux container runtime allowing creation of multiple isolated Linux systems (containers) on a control host using a single Linux kernel	System container and virtual machine **manager** built on top of LXC, enabling easier management, control and integration
Only supports containers	Supports container and VMs
Low-level tool requiring expertise	Better user experience through a simple REST API
Online Demo Tool: https://linuxcontainers.org/lxd/try-it/	

Application Containers

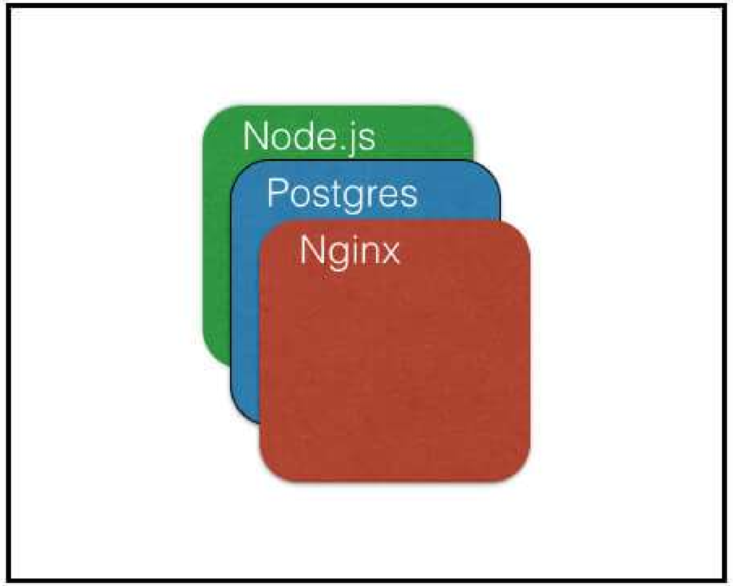
- Containers running a single process per container
- They run stateless types of workloads
- Scale up and down as needed – create new containers and delete them at any time
- No need to care about the lifecycle of those containers – ephemeral
- Example: Docker and rkt from CoreOs

System vs Application Containers



OS containers

- Meant to used as an OS - run multiple services
- No layered filesystems by default
- Built on cgroups, namespaces, native process resource isolation
- Examples - LXC, OpenVZ, Linux VServer, BSD Jails, Solaris Zones



App containers

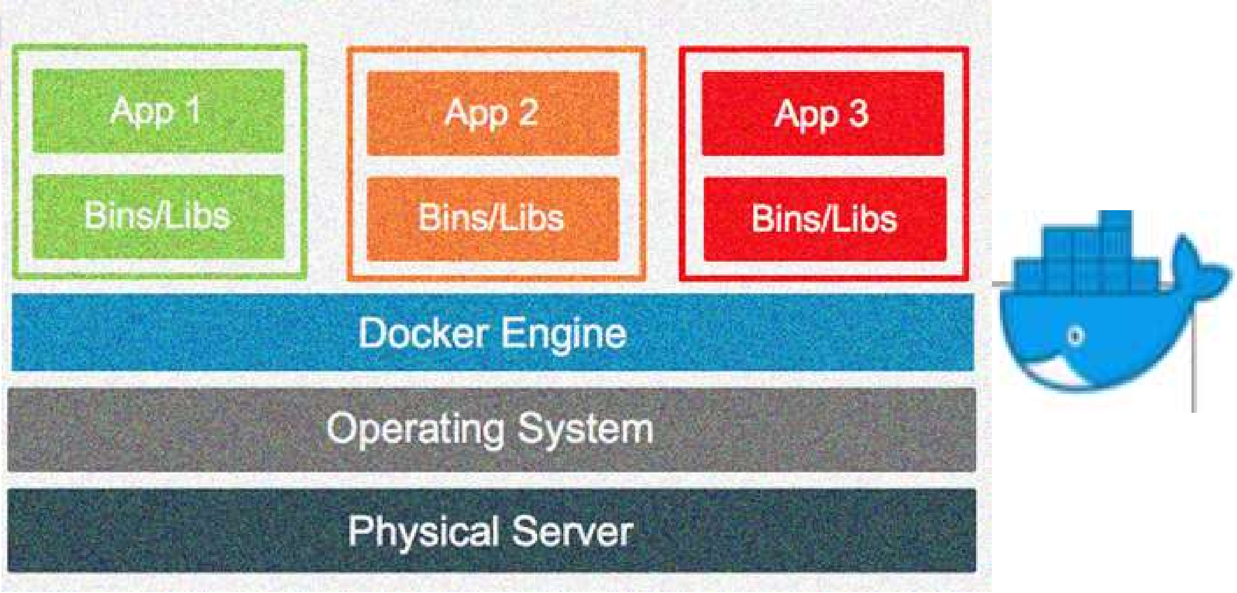
- Meant to run for a single service
- Layered filesystems
- Built on top of OS container technologies
- Examples - Docker, Rocket

Agenda

- **Docker Overview**
  - ✓ Docker Platform
  - ✓ Docker Architecture
  - ✓ Example for running a Docker container
  - ✓ Container Storage
    - ✓ Volumes
    - ✓ Bind mounts
  - ✓ Container Networking
    - ✓ Bridge
    - ✓ Host
    - ✓ Overlay
    - ✓ none

The Docker platform

- Docker is an open platform
- Docker separates applications from hardware infrastructure
- **Containers are used to package and run an application**
- A single host can run many containers simultaneously
- Containers are lightweight and contain everything needed to run the application

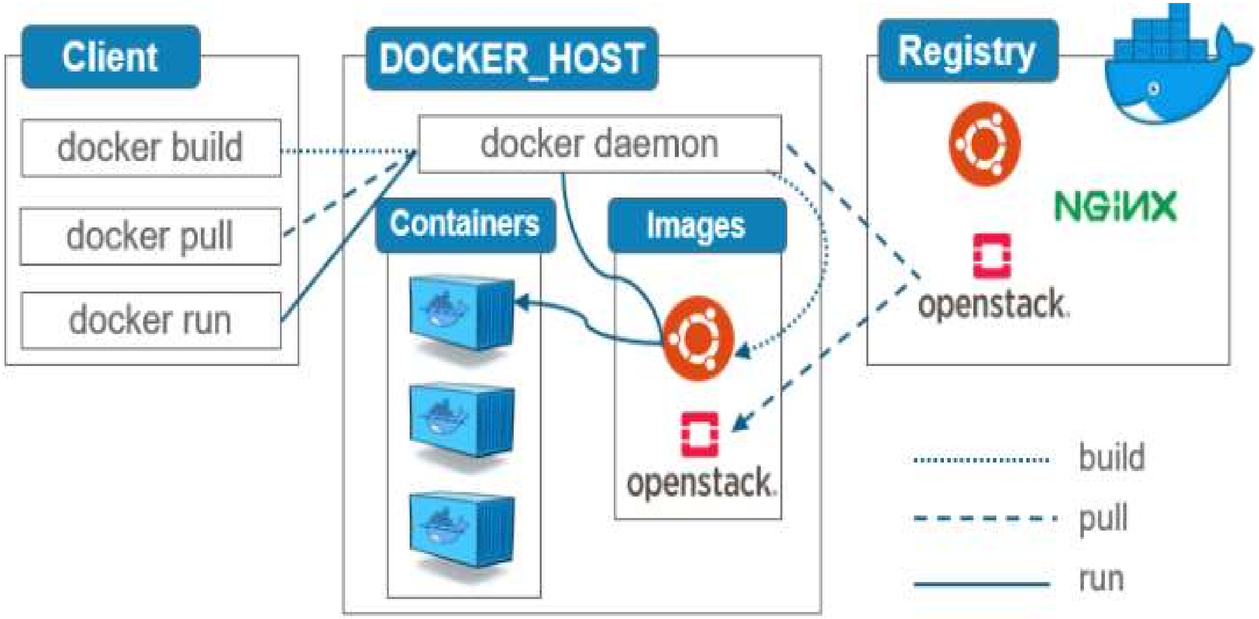


The Docker platform

- Docker provides tooling and a platform to manage the lifecycle of your containers:
  - Develop application using containers.
  - Distributing and test application using containers.
  - Deploy application into production environment, as a container or an orchestrated service.
  - .
- Containers are great for continuous integration and continuous delivery (CI/CD) workflows.

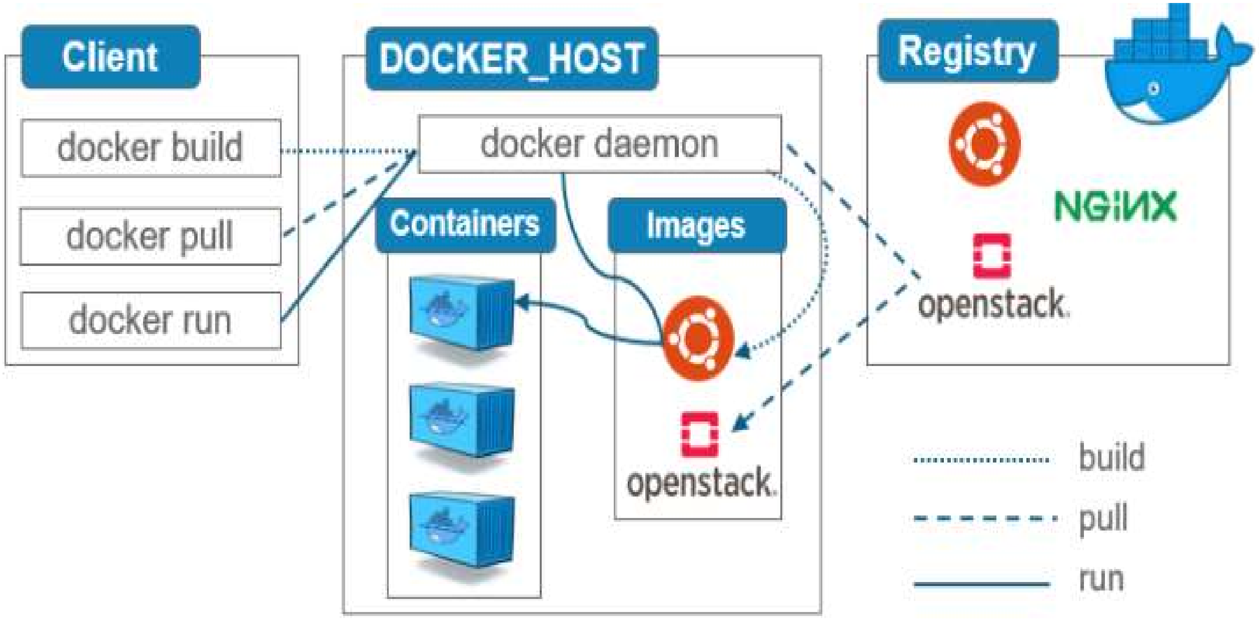
Docker architecture

- Docker uses a client-server architecture.
- The Docker daemon
    - The Docker daemon (dockerd) listens for Docker API requests
    - Manages Docker objects such as images, containers, networks, and volumes.
    - Builds, runs, and distributes containers
- The Docker client
    - The Docker *client* talks to the Docker *daemon*
    - The Docker client and daemon *can* run on the same system
    - The Docker client can communicate with more than one daemon..
- The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.



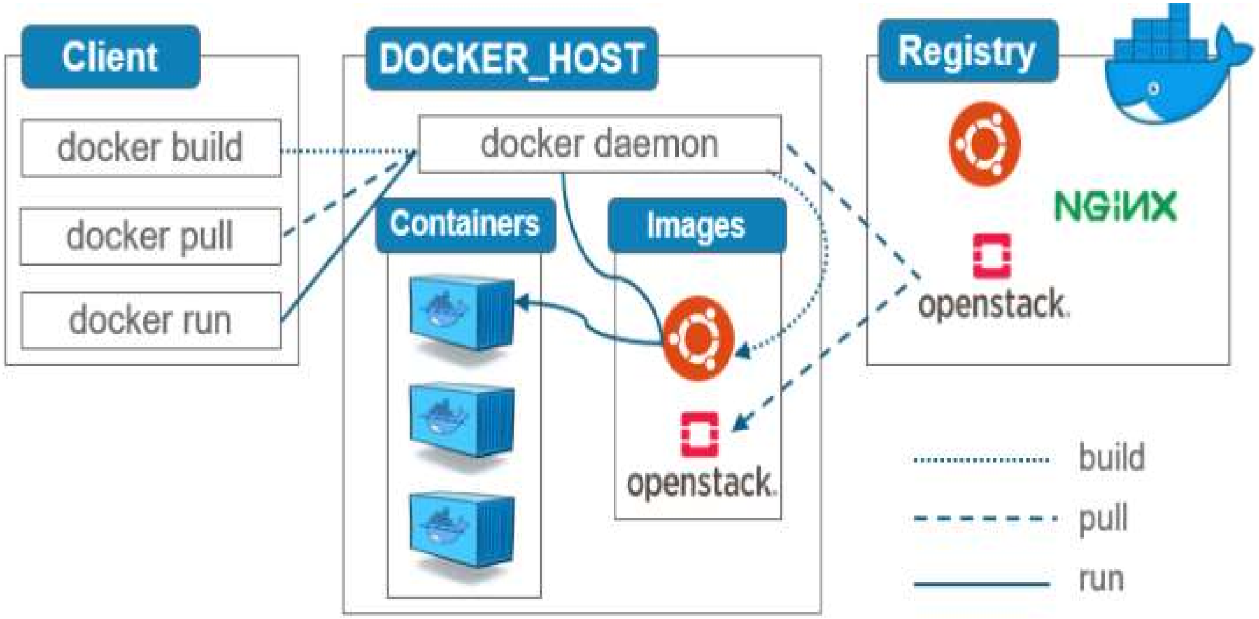
Docker architecture

- Docker registries
  - A Docker *registry* stores Docker images.
  - Docker Hub is a public registry that anyone can use.
  - Docker is configured to look for images on Docker Hub by default

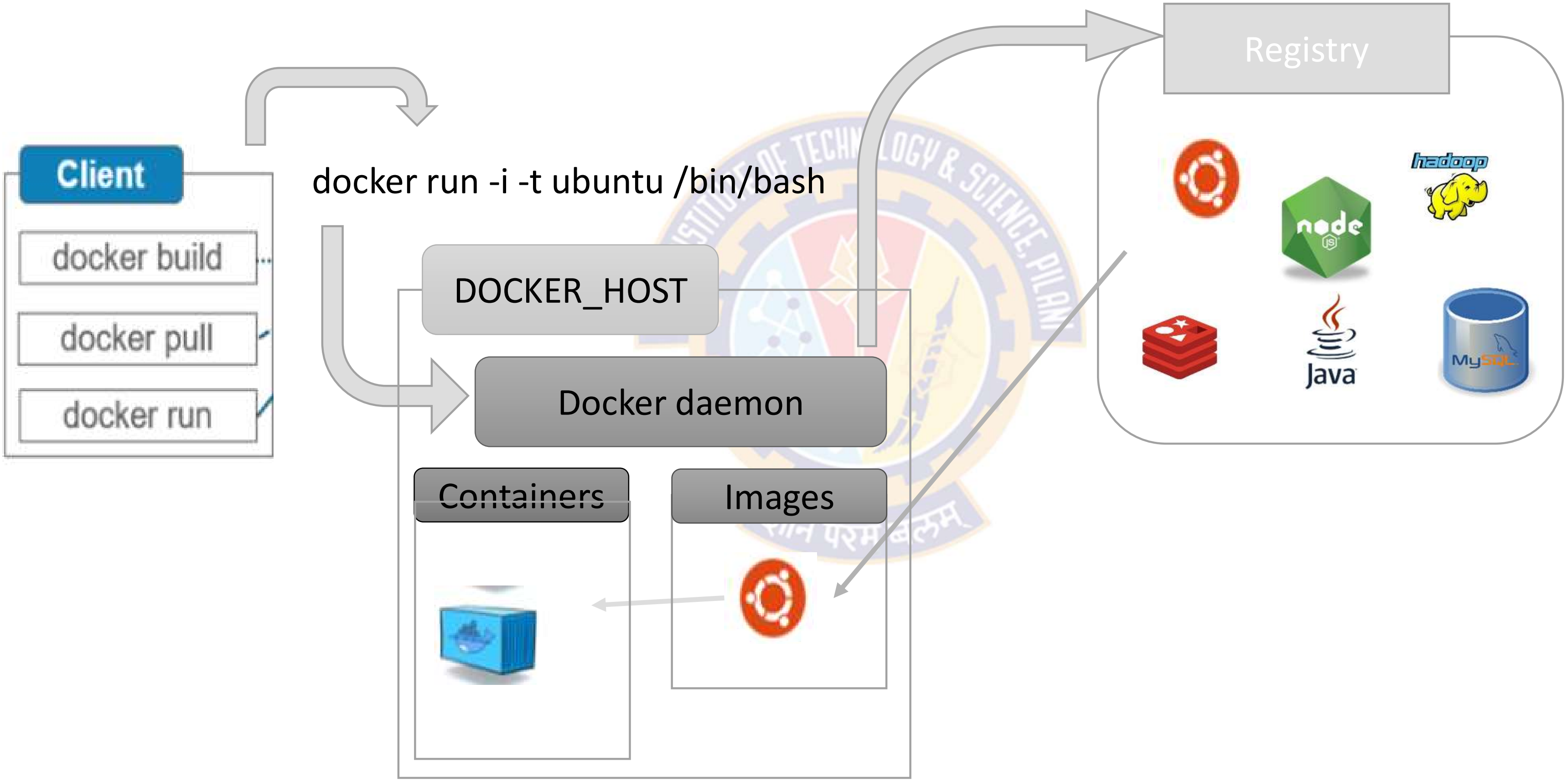


Docker architecture

- Docker objects
  - IMAGES: AN *IMAGE* IS A READ-ONLY TEMPLATE WITH INSTRUCTIONS FOR CREATING A DOCKER CONTAINER.
  - CONTAINERS: A CONTAINER IS A RUNNABLE INSTANCE OF AN IMAGE



Example for running a Docker container



Docker Commands

Containers

- A container is a runtime instance of a docker image
- Create and run a container from an image, with a custom name:
  - docker run –name <container name> <image name>

  docker run –name mylinuxserver Ubuntu
- Run a container with and publish a container's port(s) to the host.
  - docker run -p <host port>:<container port> <image name>

  docker run –p 8080:80 nginx
- Run a container in the background
  - docker run –d <image name>

  docker run –d –p 8080:80 nginx
- Start or stop an existing container:
  - docker start/stop <container name> (or <container id>)

  docker stop 11ed or mynginx
- Remove a stopped container:
  - docker rm <container name> (or <container id>)

  docker rm –f 11ed or mynginx
- Open a shell inside a running container:
  - docker exec -it <container name> sh

  docker exec –it myubuntu bash

Docker Commands

Container

- Fetch and follow the logs of a container: ***docker logs -f <container name>***
- To inspect a running container: ***docker inspect <container id> (or) <container name>***
- To list currently running containers: ***docker ps***
- List all docker containers (running and stopped): ***docker ps --all***
- View resource usage stats: ***docker container stats***

Docker Commands

Images: Docker images are a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings

- Build an Image from a Dockerfile: ***docker build -t <image name>***
- Build an Image from a Dockerfile without the cache: ***docker build -t <image name>. –no-cache***
- List local images: ***docker images ls***
- Delete an Image: ***docker rmi <image name>***
- Remove all unused images: ***docker image prune***

Build and run customised Docker image

Dockerfile

```
FROM ubuntu:lastest
RUN mkdir /app
RUN apt update
RUN apt install vim gcc -y
WORKDIR /app
ENTRYPOINT ["/bin/bash"]
```

- docker build -t mycustomimage .
- docker run -it mycustomimage

Container Storage

- The container’s filesystem
- Each container also gets its own “scratch space” to create/update/remove files.
- Any changes won’t be seen in another container, *even if* they are using the same image
- Docker containers use the following for persistent storage of Date
  - Volumes
  - Bind mounts

Container Storage

Volumes

- Docker containers use ***Volumes*** as the ***preferred mechanism*** to ***store persisting data*** generated by and used by containers
- Volumes work on both Linux and Windows containers.
- Volumes can be more safely shared among multiple containers.
- Volume drivers let you store volumes on remote hosts or cloud providers
- Volumes provide the ability to connect specific filesystem paths of the container back to the host machine

Container Storage: Volumes

Steps to create and mount Volume:

- Create a volume by using the docker volume create command

```
docker volume create mydb
```

- Start the container with mount

```
docker run -dp 3000:3000 --mount type=volume,src=mydb,target=/etc/myappdb getting-started
```

- docker volume inspect

```
docker volume inspect mydb
```

OUTPUT

```
[
   {
      "CreatedAt": "2019-09-26T02:18:36Z",
      "Driver": "local",
      "Labels": {},
      "Mountpoint": "/var/lib/docker/volumes/mydb/_data",
      "Name": “mydb",
      "Options": {},
      "Scope": "local"
   }
]
```

Container Storage: bind mounts

- A bind mount - share a directory from the host’s filesystem into the container.
- When working on an application, you can use a bind mount to mount source code into the container.
- The container sees the changes you make to the code immediately, as soon as you save a file.
- This means that you can run processes in the container that watch for filesystem changes and respond to them.

docker run -it --mount type=bind,src="$(pwd)",target=/src ubuntu bash

The --***mount*** option tells Docker to create a bind mount

src is the current working directory on your host machine (getting-started/app)

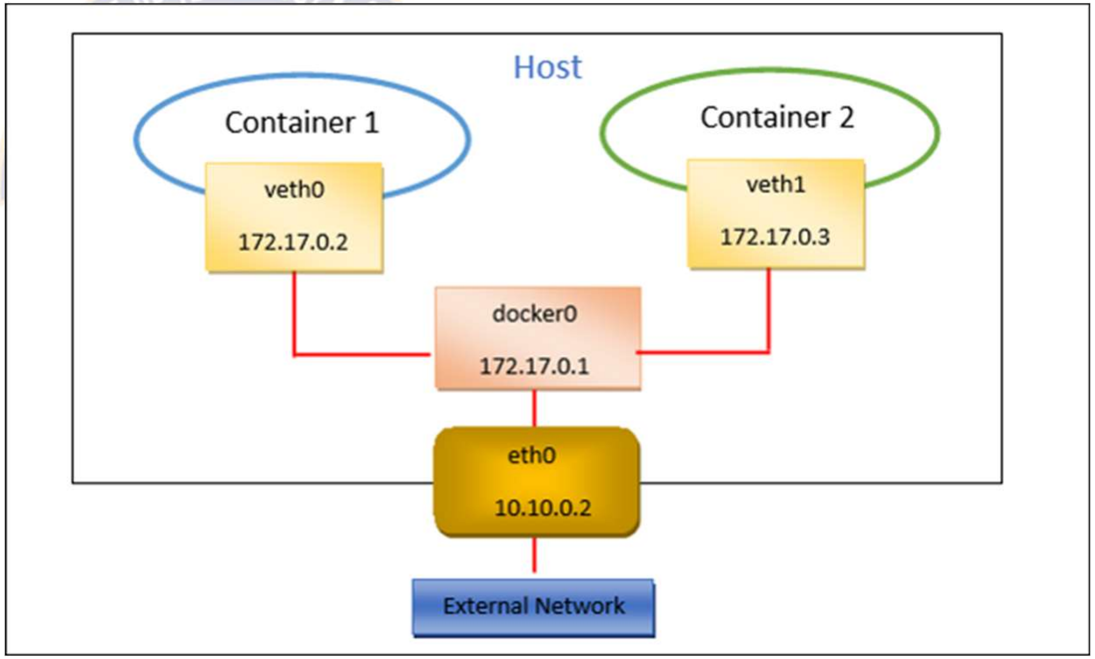
target is where that directory should appear inside the container (/src)

Container Networking

- Docker containers and services are powerful
  - Connect them together, or connect them to non-Docker workloads.
  - Docker containers and services do not even need to be aware that they are deployed on Docker, or whether their peers are also Docker workloads or not.
  - Whether your Docker hosts run Linux, Windows, or a mix of the two, you can use Docker to manage them in a platform-agnostic way.

Network drivers

- Docker’s networking subsystem is pluggable, using drivers.
- Several drivers exist by default, and provide core networking functionality:
  - Bridge
  - Host
  - Overlay
  - Ipvlan
  - Macvlan
  - none



Network drivers: bridge

- The default network driver.
- If you don’t specify a driver, this is the type of network you are creating.
- **Bridge networks are usually used when your applications run in standalone containers that need to communicate.**

```
docker network ls                                #command to list all the networks(drivers) in Docker
docker run -dit --rm --name alpine1 alpine ash   #command to run a container
docker run -dit --rm --name alpine2 alpine ash
docker container ls                              #list all the containers
docker network inspect bridge            #details of bridge network driver
docker attach alpine1                            #attach to the container (get console)
$ip addr show
$ping -c 2 google.com
$ping -c 2 alpine2
docker stop alpine1 alpine2                      #to stop the containers
```

Network drivers: bridge

- User Defined bridge network

```
docker network create --driver bridge mynet
docker network ls
docker network inspect mynet
---
docker run -dit --rm --name alpine1 --network mynet alpine ash
docker run -dit --rm --name alpine2 --network mynet alpine ash
docker run -dit --rm --name alpine3 alpine ash
docker run -dit --rm --name alpine4 --network mynet alpine ash
docker network connect bridge alpine4
---
docker network inspect bridge
docker network inspect mynet
---
```

Network drivers: bridge

```
---
docker container attach alpine1
ping -c 2 alpine2
ping -c 2 alpine3 (not pingable)
---
```

Detach from alpine1 using detach sequence, CTRL + p CTRL + q (hold down CTRL and type p followed by q).

```
ping -c 2 google.com
docker container stop alpine1 alpine2 alpine3 alpine4     #stop and remove containers
docker container rm alpine1 alpine2 alpine3 alpine4
docker network rm mynet                                   #remove user-defined bridge network
```

Network drivers: Host

Networking using the host network

- The goal of this tutorial is to start a container(nginx) which binds directly to port 80 on the Docker host.
- From a networking point of view, this is the same level of isolation as if the nginx process were running directly on the Docker host and not in a container.
- However, in all other ways, such as storage, process namespace, and user namespace, the nginx process is isolated from the host.

- This procedure requires port 80 to be available on the Docker host.
- The host networking driver only works on Linux hosts, and is not supported on Docker Desktop for Mac, Docker Desktop for Windows, or Docker EE for Windows Server.

Network drivers: Host

Networking using the host network

- The goal of this tutorial is to start a container(nginx) which binds directly to port 80 on the Docker host.
- From a networking point of view, this is the same level of isolation as if the nginx process were running directly on the Docker host and not in a container.
- However, in all other ways, such as storage, process namespace, and user namespace, the nginx process is isolated from the host.

- This procedure requires port 80 to be available on the Docker host.
- **The host networking driver only works on Linux hosts**, and is not supported on Docker Desktop for Mac, Docker Desktop for Windows, or Docker EE for Windows Server.

```
docker run --rm -d --network host --name my_nginx nginx

sudo netstat -tulpn | grep :80                          #Verify which process is bound to port 80

docker container stop my_nginx
```

Network drivers: Overlay

- Overlay networks connect multiple Docker daemons running in multiple hosts together and enable swarm services to communicate with each other.
- You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons.
- This strategy removes the need to do OS-level routing between these containers.

Network drivers: Overlay

Use the default overlay network

- Create the swarm
  - On manager. initialize the swarm.

```
docker swarm init
docker swarm join --token <TOKEN> \
  --advertise-addr <IP-ADDRESS-OF-WORKER-1> \     #optional
  <IP-ADDRESS-OF-MANAGER>:2377

docker node ls                              #on manager
docker network ls
```

The docker_gwbridge connects the ingress network to the Docker host’s network interface so that traffic can flow to and from swarm managers and workers.

Network drivers: Overlay

Use the default overlay network

```
sudo docker network create -d overlay nginx-net
docker service create \
  --name my-nginx \
  --publish target=80,published=80 \
  --replicas=5 \
  --network nginx-net \
  nginx

sudo docker service ls
sudo docker service ps my-nginx
sudo docker service rm my-nginx
```

Network drivers: Overlay

Use an overlay network for standalone containers

- Set up the swarm

```
docker swarm init
docker swarm join --token <your_token> <your_ip_address>:2377
```

On host1, create an attachable overlay network called test-net:

```
docker network create --driver=overlay --attachable test-net
docker run -it --name alpine1 --network test-net alpine
docker network ls                  #on host2
docker run -dit --name alpine2 --network test-net alpine
docker network ls
ping -c 2 alpine2                  #from host1
docker container stop alpine2
docker network ls
docker container rm alpine2
docker container rm alpine1
docker network rm test-net
```

Network drivers

- Ipvlan
  - IPvlan networks give users total control over both IPv4 and IPv6 addressing.
  - The VLAN driver builds on top of that in giving operators complete control of layer 2 VLAN tagging and even IPvlan L3 routing for users interested in underlay network integration.
- Macvlan
  - Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network.
  - The Docker daemon routes traffic to containers by their MAC addresses
- None
  - For this container, disable all networking.
  - Usually used in conjunction with a custom network driver.