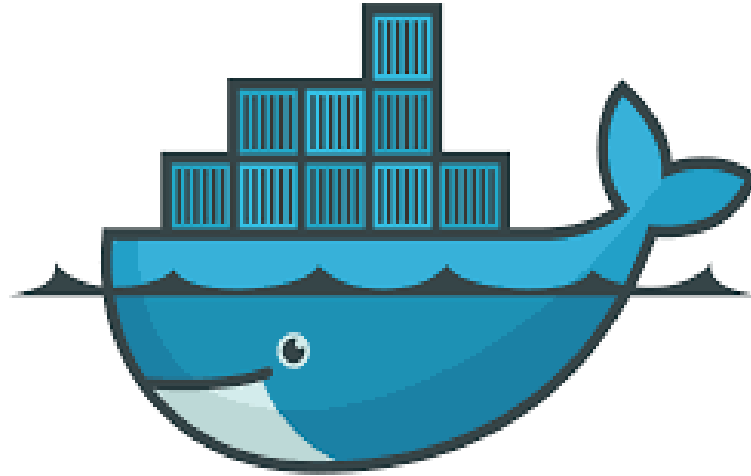


# Mastering Docker: Practical & Hands-On



docker

# Introduction

- Introduction to Docker and Containers
- Docker Architecture
- Installation Docker
- Deep Dive – Docker Containers
- Deep Dive – Docker Images, Registry
- Deep Dive – Container Networks
- Deep Dive – Docker Volumes
- Deep Dive – Docker Compose
- Docker commands/ Lab Environment



# Session: 1

## Introduction Docker and Containers



# What is Docker ?



# Introduction Docker

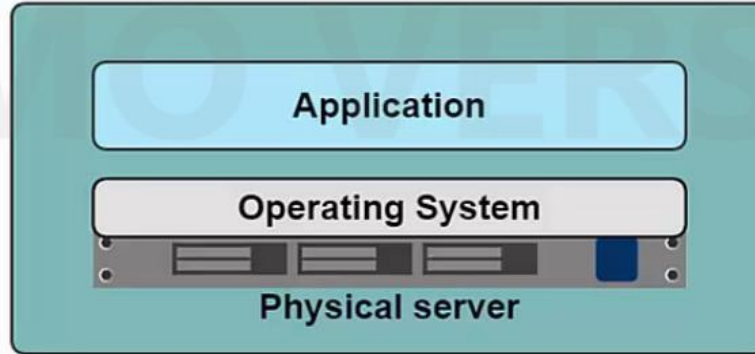
- Docker is an “**Open Source Container Platform**” which is designed to **Build, Ship** and **Run** applications as containers.
- It's a leading container technology in this emerging container world.
- Its provided by Docker Inc.
- Technology is moving towards “**Containerization**”.
- Host centric infrastructure to Application or Containers centric infrastructure.
- Before proceeding with Docker, lets have a look on the history/traditional approaches used for the application since ages .



# Traditional approach / Bare Metal

- In the traditional ages of development and deployment of applications.

## One Application on one physical server



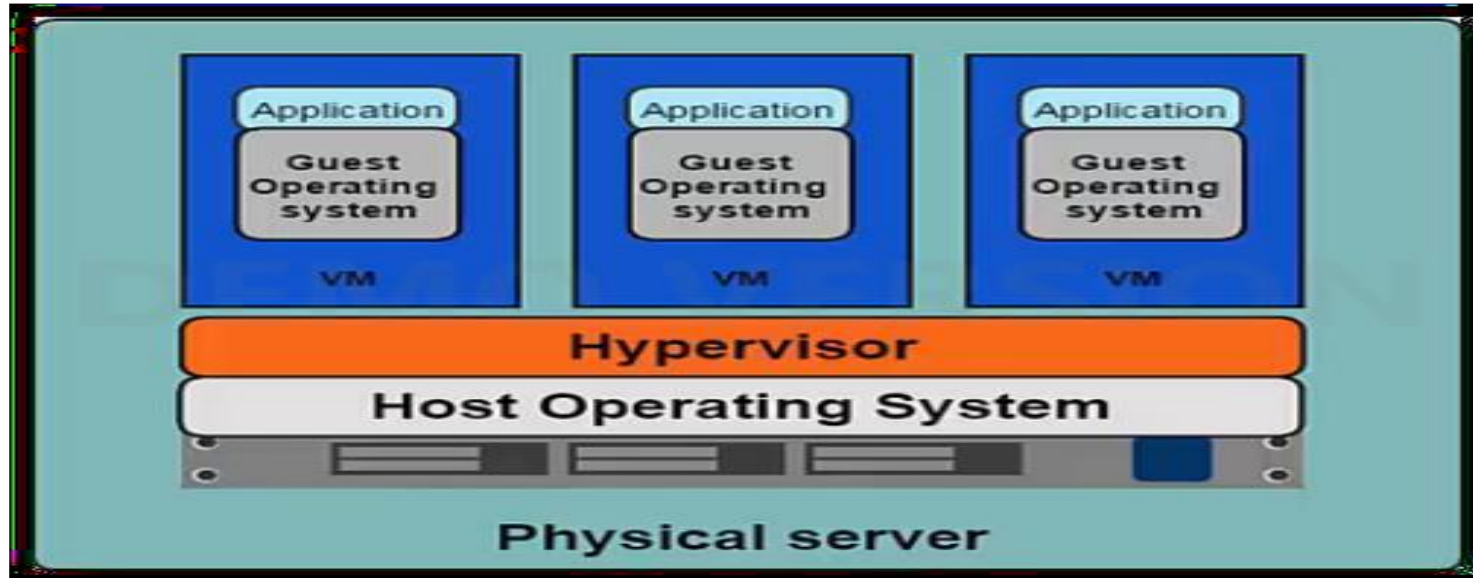
# Problems in the Past

- Slowness in Deployment
  - Difficult to Scale
  - Huge cost in Infrastructure
  - Unused Resources
  - Vendor dependency
- 
- The problem got the solution by a technology called “Hypervisor-Based Virtualization”.  
Next slide....



# Hypervisor-Based Virtualization

- One physical server can contain multiple running applications.





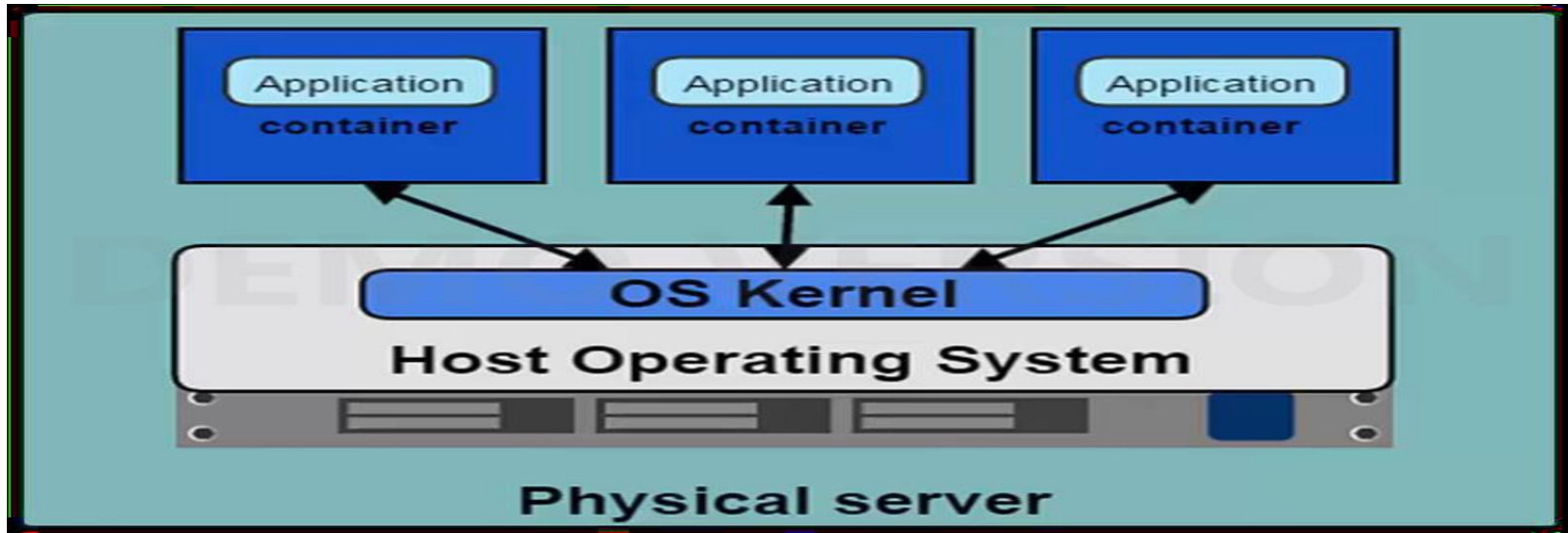
# Benefits/Limitations of Hypervisor

- Effective use of System Resources like memory/CPU/Disk/Network( Resource optimization)
- Cost saving
- More useful for Cloud business
- **Limitations**
- Every VM still requires (CPU, Memory, Disk, Guest OS).
- The more VM's you run, the more resources of system you need.
- Hard to maintain VMs
- Slow in start/stop
- The problem got the solution by a technology called “Container-Based Virtualization”.  
Next slide....



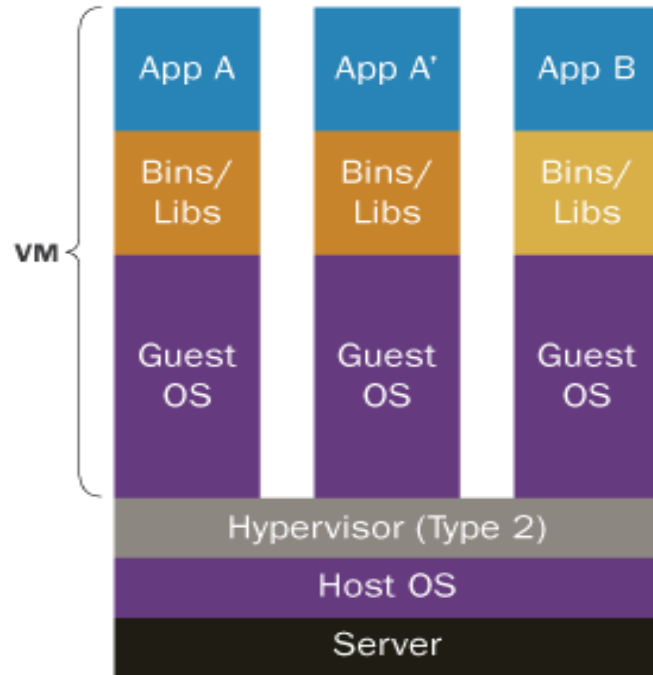
# Container-Based Virtualization

- Container based virtualization uses the kernel on the host's operating system to run multiple guest instances called “containers”.
- OS Kernel Virtualization.

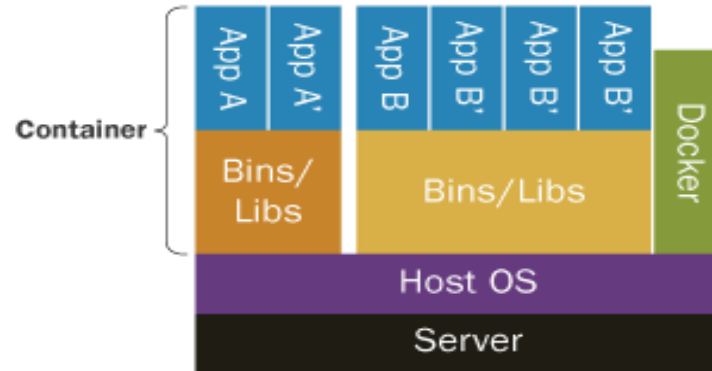


# Virtualized vs. Containerized Architecture

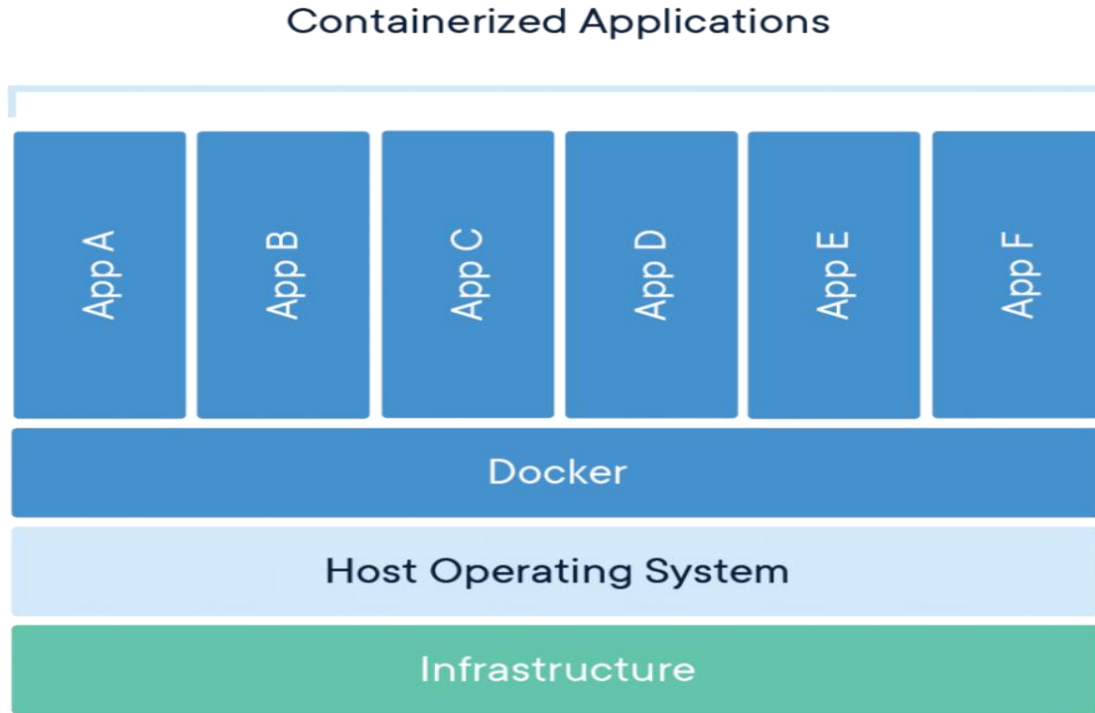
## Containers vs. VMs



Containers are isolated, but share OS and, where appropriate, bins/libraries



# Containerized Applications



# Features of Containers

- Containerization is increasingly popular because containers are:
- **Flexible:** Even the most complex applications can be containerized.
- **Lightweight:** Containers leverage and share the host kernel.
- **Interchangeable:** You can deploy updates and upgrades on-the-fly.
- **Portable:** You can build locally, deploy to the cloud, and run anywhere.
- **Scalable:** You can increase and automatically distribute container replicas.
- **Stackable:** You can stack services vertically and on-the-fly.



# Containers vs VM's

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



# Origins of Docker Project

- Company: 'dotCloud' Inc. in 2008 ( Docker Inc. now).
- Solomon Hykes started Docker as an internal project within dotCloud.
- The codename for this project is "Docker." Name came from **Dock Worker**.
- Docker was released as open source project in March 2013.
- Docker Inc. is the primary sponsor and contributor to the Docker Project.
- HQ in San Francisco.
- Docker is written in Go and takes advantage of several features of the Linux kernel to deliver its functionality.



# Docker becomes a container platform

- The container engine is now known as "**Docker Engine**".
- Software's by Docker Inc:
  - Docker Compose**
  - Docker Machine**
  - Docker Swarm**
  - Docker Datacenter
  - Docker Cloud (formerly "Tutum")
- Docker Inc. launches commercial offers. (Enterprise Edition)





## Session: 2

### Docker Architecture



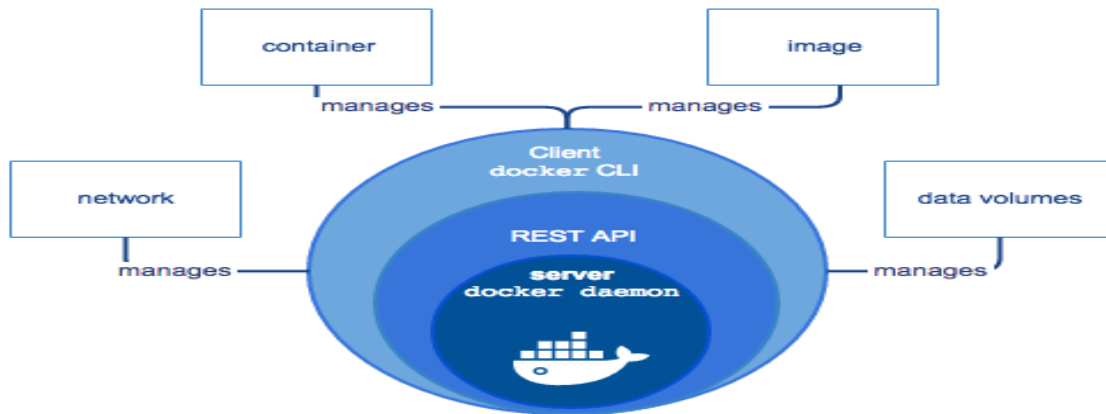
# Docker Components

- **Docker Engine** ( Build, Ship and Run applications)
- **Docker Images** ( Build component)
- **Docker Container** ( Run component)
- **Docker Registry** ( Distribution component) Docker Hub
- **Docker Orchestration** ( Native Docker Clustering and Orchestrator) by Docker Machine, Docker Swarm, Docker Compose



# Docker Engine

- Docker Engine is a client-server application.
- The client uses the Docker REST API to control or interact with the Docker daemon through CLI commands.
- The daemon creates and manages Docker objects, such as images, containers, networks, and data volumes.

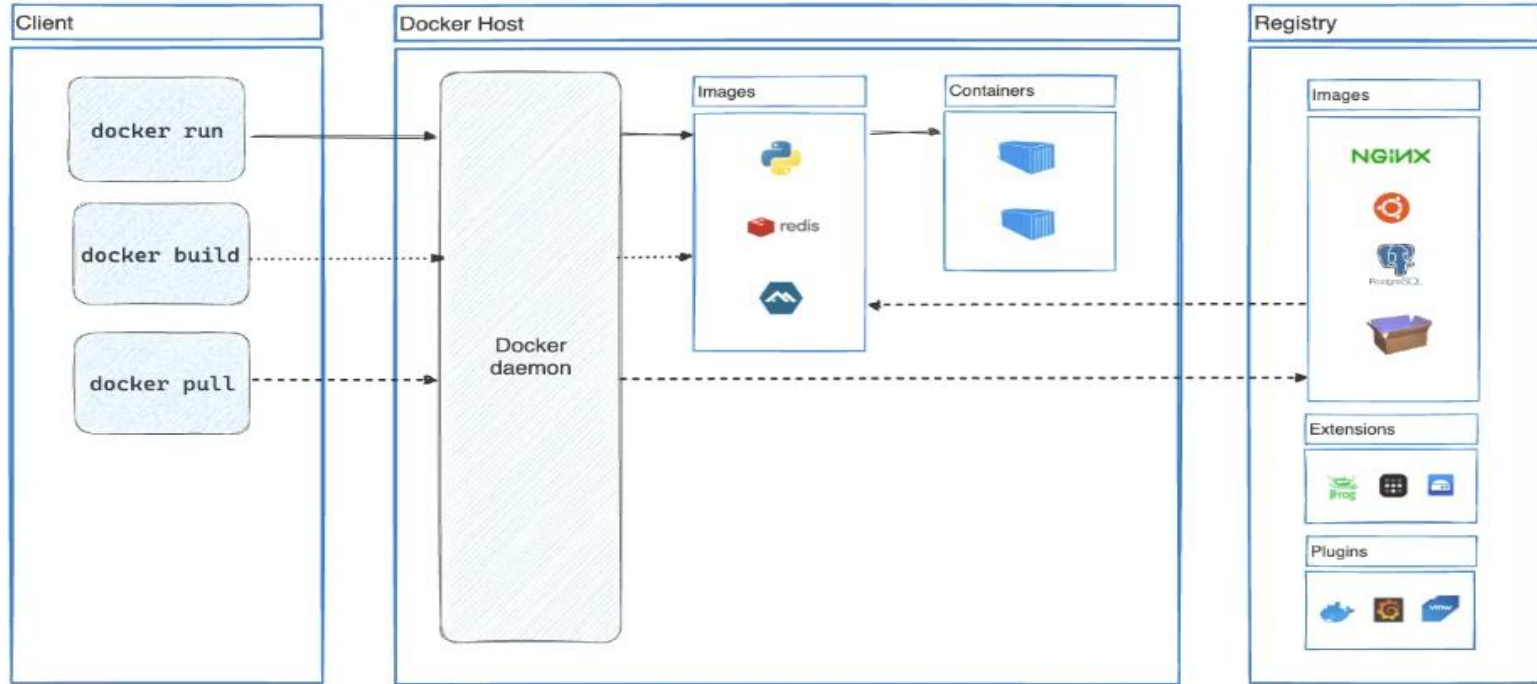


# Docker Architecture

- 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



# Docker Images

- Docker images are the **build component** of Docker.
- A Docker image is a read-only template with instructions for creating a Docker container.
- For example, an image might contain an centos operating system with Apache web server and your 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.



# Docker Containers

- Docker containers are the **run component** of Docker.
- A Docker container is a “**running instance**” of a Docker image.
- You can run, start, stop or delete a container using Docker CLI commands.



# Docker Registry

- Docker registries are the **distribution component** of Docker.
- 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 or on a totally separate server.
- “**Docker Hub**” is known as Default and global registry.





# Docker Orchestration

- 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 applications.



# Advanced concepts of Docker

- **Namespaces** : ( mnt, pid, net, ipc, uts/hostname, user/ids )
- When you run a container, Docker creates a set of namespaces for that container. These namespaces provide a layer of isolation.
- **Control groups(cgroups)**: ( cpu, memory, disk, i/o-resource management )
- cgroups is a Linux kernel feature. Control groups allow Docker Engine to share available hardware resources to containers and optionally enforce limits and constraints.
- **Open Container Initiative (OCI)**:
  - Formed in June 2015, the Open Container Initiative (OCI) aims to establish common standards for software containers.
  - It contains two specifications:
    - runtime-spec: The runtime specification
    - image-spec: The image specification



# Advanced concepts of Docker

- **Docker Daemon (dockerd)**
- Path: /usr/bin/dockerd
- Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers, Managing images, Network, Volumes.
- Prior to 1.11, Daemon manages container's runtime. Later runtime shifted to containerd.
- **containerd**
- only deals with containers — it takes the role of starting, stopping, pausing, and destroying containers. Since the container runtime is isolated from the engine, Engine ultimately will be able to be restarted or upgraded without having to restart the containers.



# Advanced concepts of Docker

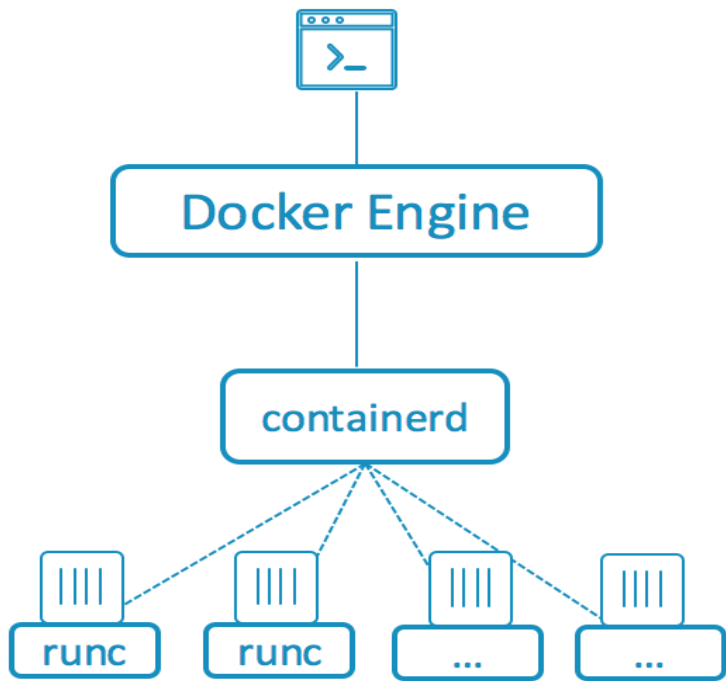
- Docker, LXC and Rocket use the technologies available in Linux kernel to manage the lifecycle of the Container.
- Before Docker version 0.9, Docker was using LXC to interact with Linux kernel. From Docker version 0.9, Docker directly interacts with Linux kernel using libcontainer interface that they developed.

```
29542 ?    Ssl  1:25 /usr/bin/dockerd
```

- ```
29545 ?    Ssl  3:06 docker-containerd --config /var/run/docker/containerd/containerd.toml
```
- ```
29668 ?    Sl   0:00 docker-containerd-shim -namespace moby -workdir  
/var/lib/docker/containerd/daemon/io.containerd.runtime.v1.linux/moby/881efc2c78caa12041cfcafa85015e157e4f478e8ba5182d5c4989f35c  
17a2eb -address /var/run/docker/containerd/docker-containerd.sock -containerd-binary /usr/bin/docker-containerd -runtime-root  
/var/run/docker/runtime-runc
```



# Advanced concepts of Docker



Same Docker UI and commands

User interacts with the Docker Engine

Engine communicates with containerd

containerd spins up runc or other OCI compliant runtime to run containers



## Session: 3

### Installation Docker



# Install Docker CE

- Official guide to install Docker
- <https://docs.docker.com/engine/install/>
- Docker can be installed via:
- [set up Docker's repositories](#) and install from them. This is the recommended approach.
- Download the RPM package and [install it manually](#).
- In testing and development environments, some users choose to use automated [convenience scripts](#) to install Docker.



Docker-Lab.xml



Docker-Lab-Logs.txt



# Install Docker CE

- Docker UP and Running
- Verification
- `$ sudo systemctl status docker`
- `$ docker version`
- Verify that Docker is installed correctly by running the hello-world image.
- `$ sudo docker run hello-world`
- `$ sudo docker ps -a`





# Configure and troubleshoot the Docker daemon

- 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, use below:
  - `$ sudo groupadd docker`
  - `$ sudo usermod -aG docker $USER`
- Docker **Daemon logs**:
  - `$ journalctl -u docker.service` on Linux systems
  - `$ less /var/log/messages` and search string `/docker`



# Display system-wide information

- `$ docker info`
- This command displays system wide information regarding the Docker installation. Information displayed includes the kernel version, number of containers and images.



## Session: 4

### Deep Dive – Docker Containers



# Containers

- A Docker container is a “running instance” of a Docker image.
- Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), and start almost instantly.



# How Container works

- When you use the “docker run” CLI command, the Docker Engine client instructs the Docker daemon to run a container.
- When you run this command, Docker Engine does the following:

Pulls the centos image

Creates a new container

Allocates a filesystem and mounts a read-write layer

Allocates a network / bridge interface

Sets up an IP address

Executes a process that you specify

Captures and provides application output



# Lab

- Creating containers
- Pull an image or a repository from a registry
- `$ docker pull centos`
- By default, Docker will run the latest version available.
- lets start creating containers using docker run
- `$ docker run hello-world` ( this is **non-interactive** mode)
- `$ docker run -it --name centos_server centos /bin/bash` ( **Interactive** mode)

Now exit here

List running containers

`$ docker ps`

List all containers

`$ docker ps -a`



# Lab

- Lets start containers in detached mode
- `$ docker run -itd --name centos_server centos /bin/bash`
- ( Its not working because we did not remove CID, `$ docker rm <CID>` )
- It will always run now, its time to login into container now.
- `$ docker ps`
  
- Login into container
- `$ docker exec -it centos_server /bin/bash`



# Lab

- Stop/Kill/Start/Restart Containers

- `$ docker stop centos_server`

(The main process inside the container will receive SIGTERM, and after a grace period, SIGKILL. `--time , -t 10` Seconds to wait for stop before killing it)

`$ docker start centos_server` or `<CID>`

- Send a KILL signal to a container (SIGKILL it will kill directly)

- `$ docker kill centos_server`

- Remove container

- `$ docker rm <CID>`





# Lab

- Logs containers
- `$ docker logs centos_server`
- `$ docker logs -f centos_server`
- `$ docker logs --tail=100 centos_server`
- Ports containers

## Publish on ports

```
$ docker run -itd --name centos_container1 -p 40001:8080 centos /bin/bash
```

```
PORTS: 0.0.0.0:40001->8080/tcp
```

```
$ docker run -itd --name centos_container2 -p 8080 centos /bin/bash
```

```
0.0.0.0:32768->8080/tcp
```



# Deep dive view on containers

- **Detailed view** on containers
- Docker inspect provides detailed information(Json) on constructs controlled by Docker.
- `$ docker inspect centos_server` or `<CID>`
- **Runtime metrics** of containers
- You can use the docker stats command to live stream a container's runtime metrics. The command supports CPU, memory usage, memory limit, and network IO metrics.
- `$ docker stats`



# Deep dive view on containers

- **Limit** a container's resources
- ([https://docs.docker.com/engine/containers/resource\\_constraints/](https://docs.docker.com/engine/containers/resource_constraints/))
- By default, a container has no resource constraints and can use as much of a given resource as the host's kernel scheduler allows.
- Docker provides ways to control how much memory, CPU, or block IO a container can use, setting runtime configuration flags of the docker run command.
- `$ docker run -itd --cpus=".5" ubuntu /bin/bash`



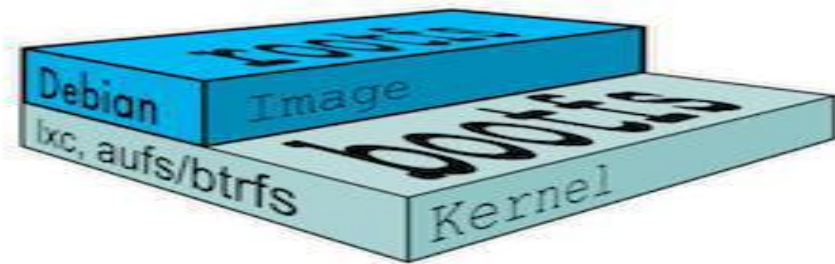
## Session: 5

### Deep Dive – Docker Images, Registry



# Docker Images

- A Docker image is a read-only template. For example, an image could contain an ubuntu operating system with apache and your web application installed.
- Images are used to create Docker containers. Docker Images are the build component of Docker.
- Having everything within the image allows you to migrate images between different environments and be confident that if it works in one environment, then it will work in another.



# 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



# Docker Registry

- Images can be stored:
  - Docker host( local host)
- Docker registry ( Docker Hub, GitHub, Self Hosted Registry, Nexus, etc..)
- The public Docker registry is provided with the Docker Hub ([hub.docker.com](https://hub.docker.com))
- You can use the Docker client to download (pull) or upload (push) images.



# Lab

- Pull/Push Images to DockerHub
- By default, docker pull pulls images from Docker Hub.
- `$ docker pull ubuntu` or `$ docker pull ubuntu:14.04`
- `$ docker images`
- Push an image or a repository to a registry
- `$ docker tag ubuntu:14.04 vstechops/ubuntu:v1`
- `$ docker login`
- `$ docker push vstechops/ubuntu:v1`
- `$ docker rmi <ImageID>`
- `$ docker pull vstechops/ubuntu:v1`
- Search Images on Dockerhub
- `$ docker search redis`





# Lab

- Pull/Push Images to different registry
- Setup local Registry
- `$ docker run -d --name my-docker-registry -p 5000:5000 registry`
- `$ docker images`
- `$ docker ps`
- `$ docker tag ubuntu:latest localhost:5000/ubuntu`
- `$ docker push localhost:5000/ubuntu`
- `$ docker rmi -f <ImageID>`
- `$ docker pull localhost:5000/Ubuntu`



# Building Images

- Docker images can be build by many ways:
  - Docker Commit
  - Docker Export/Import
  - Docker Build ( Based on Dockerfile)



# Docker Commit

- Let's start from base image "centos":
- `$ docker ps` ( see if any centos container is running, if not please run it and login it)
- `$ yum -y update`
- `$ yum install httpd`
- `exit`
- Inspect the changes:
- `$ docker diff <yourContainerId>`
- Commit the changes:
- `$ docker commit <yourContainerId>`
- `$ docker tag <newImageId> httpd_v1`



# Docker Export/Import

- Export / Import Containers
- If we wanted to move the centos to another machine then we can export it to a .tar file.
- `$ docker export CID > centosContainer.tar`
- `$ docker import centosContainer.tar`
- `$ docker images`



# Docker Build

## Dockerfile:

- Docker can build the images automatically by reading the instructions from a Dockerfile.
- A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.
- Using docker build users can create an automated build that executes several command-line instructions in succession.
- The docker build command builds an image from a Dockerfile and context



# Docker Build

- **RUN** <command> allows you to execute any command as you would at a command prompt,
- **COPY** <src> <dest> allows you to copy files from the directory containing the Dockerfile to the container's image. This is extremely useful for source code and assets that you want to be deployed inside your container.
- **EXPOSE** <port> command you tell Docker which ports should be open and can be bound too.
- **CMD** line in a Dockerfile defines the default command to run when a container is launched.



# Docker Build

- CMD defines a default command to run when none is given.
- The EXPOSE instruction tells Docker what ports are to be published in this image.
- EXPOSE 8080
- EXPOSE 80 443
- EXPOSE 53/tcp 53/udp
- All ports are private by default. A private port is not reachable from outside.
- The MAINTAINER instruction tells you who wrote the Dockerfile.
- It is possible to execute multiple commands in a single step:
- RUN apt-get update && apt-get install -y wget && apt-get clean
- The COPY instruction adds files and content from your host into the image.
- COPY . /src



# Lab

- Deploy Static HTML Website as Container
- Create index.html and Dockerfile
- `$ cat index.html`

```
<h1>Hello NGINX server Demo</h1>
```

- `$ cat Dockerfile`

```
FROM nginx:1.11-alpine
```

```
COPY index.html /usr/share/nginx/html/index.html
```

```
EXPOSE 80
```

```
CMD ["nginx", "-g", "daemon off;"]
```





# Lab

- `$ docker build -t my-nginx-image:latest .`
- `$ docker images`
- `$ docker run -itd -p 80:80 my-nginx-image:latest`
- `$ docker ps`
- The history command lists all the layers composing an image.
- `$ docker history my-nginx-image`



# Lab

- Another example with details

- `$ cat Dockerfile`

```
FROM centos
```

```
MAINTAINER abc <abc@company.com>
```

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

```
RUN echo "Website is hosted inside a container" > /var/www/html/index.html
```

```
EXPOSE 80
```

```
VOLUME /mnt/docker_vol /data
```

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

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



## Session: 6

### Deep Dive – Container Networks

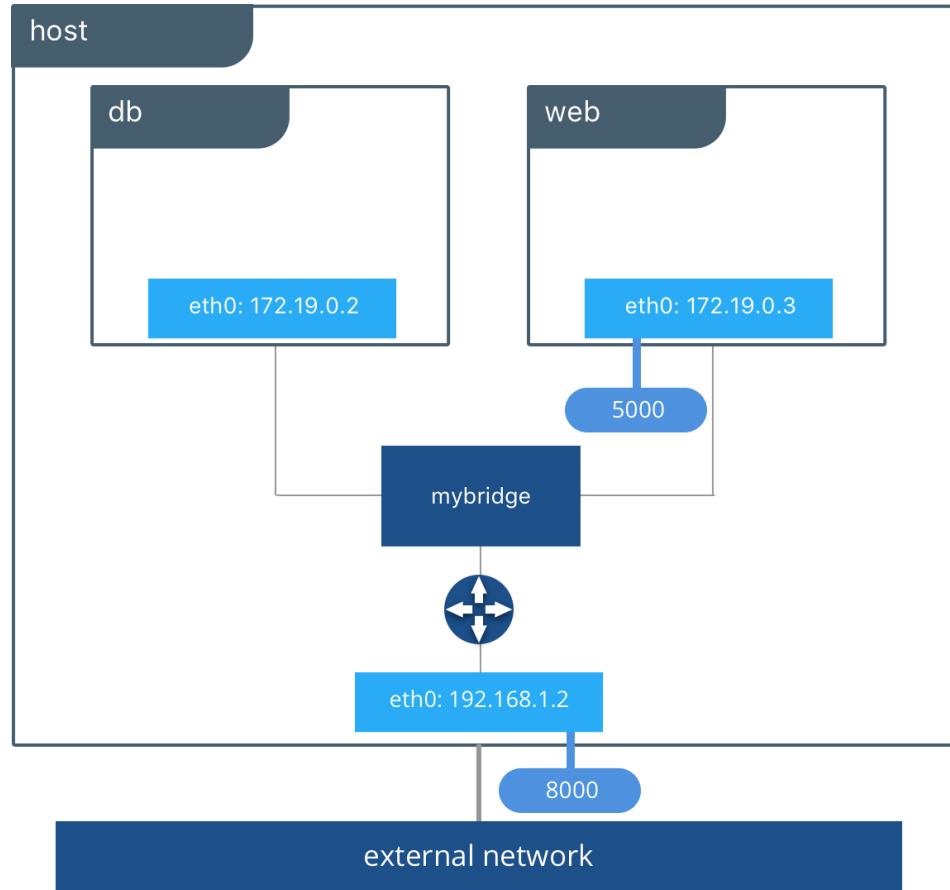


# Container Network Model (CNM)

- The CNM was introduced in Engine 1.9.0 (November 2015).
- Conceptually, a network is a virtual switch.
- A network has an IP subnet associated to it.
- A network is managed by a *driver*.
- `$ docker network ls`



# Container Network Model



# Container Networks

- **Ports** concepts

Publish on ports

```
$ docker run -itd --name centos_container1 -p 40001:8080 centos /bin/bash
```

PORTS: 0.0.0.0:40001->8080/tcp

```
$ docker run -itd --name centos_container2 -p 8080 centos /bin/bash
```

0.0.0.0:32768->8080/tcp

```
$ docker run -it --name centos_container3 centos /bin/bash
```

Ports empty



# Container Networks

- Create Network
- The first step is to create a network using the CLI. This network will allow us to attach multiple containers which will be able to discover each other.
- `$ docker network ls`
- `$ docker network create backend-network`



# Communicating Between Containers

- Communication between containers can be done by 3 ways:
  - Hostname:IP
  - Links
  - Docker Networks





# Communication on Hostname:IP

- using Hostname:IP
- We need to set below config in as an example.

environment:

- `spring.cloud.config.uri: http://IP:Port`



# Communication using Links

- To connect to a source container you use the `--link <container-name|id>:<alias>` option when launching a new container.
- `$ docker run -d --name redis-server redis`
- `$ docker ps`
- Lets launch alpine container and use link to connect it to redis
- `$ docker run --link redis-server:redis alpine env`
- `$ docker run --link redis-server:redis alpine cat /etc/hosts`
- `$ docker run --link redis-server:redis alpine ping -c 1 redis`



# Communication using Docker Networks

- Docker no longer assigns environment variables or updates the hosts file of containers.
- This network will allow us to attach multiple containers which will be able to discover each other. Embedded DNS Server in Docker.
- `$ docker network ls`
- `$ docker network create backend-network-demo`
- `$ docker run -itd --name redis-server-net --net=backend-network-demo redis`
- Verify if it is having entry for these, its should not be there.
- `$ docker run --net=backend-network-demo alpine env`
- `$ docker run --net=backend-network-demo alpine cat /etc/hosts`



# Communication using Docker Networks

- When containers attempt to access other containers via a well-known name, such as Redis, the DNS server will return the IP address of the correct Container. In this case, the fully qualified name of Redis will be redis-server.backend-network.
- `$ docker run --net=backend-network-demo alpine cat /etc/resolv.conf`
- `$ docker run --net=backend-network-demo alpine ping -c1 redis-server`
- `$ docker run -it --net=backend-network alpine sh`
- `/ # ping redis-server`
- Inspect the network details
- `$ docker network inspect backend-network-demo`



## Session: 7

### Deep Dive – Docker Volumes



# Docker Volumes

- Docker volumes can be used to achieve many things, including:
  - – Bypassing the copy-on-write system to obtain native disk I/O performance.
  - –Sharing a directory between multiple containers.
  - –Sharing a directory between the host and a container.
  - –Sharing a single file between the host and a container.
- Docker Volumes allows you to upgrade containers, restart machines and share data without data loss. This is essential when updating database or application versions.



# Persisting Data Using Volumes

- Docker Volumes allows you to upgrade containers, restart machines and share data without data loss. This is essential when updating database or application versions.
- This mapping is bi-directional. It allows data stored on the host to be accessed from within the container. It also means data saved by the process inside the container is persisted on the host.
- Volumes can be declared in two different ways.
  - Dockerfile, with a VOLUME instruction.

VOLUME /config

-v flag for “docker run”.

```
$ docker run -itd --name redis-server -v /docker/redis-data:/data redis
```



# Lab

- Create Volume
- `$ docker run -itd --name redis-server -v /docker/redis-data:/data redis --appendonly yes`
- Verify it
- `$ docker exec -it <CID> sh` and `cd /data`
- This same directory can be mounted to a second container.
- Test it : `$ docker run -v /docker/redis-data:/backup ubuntu ls /backup`
- **Sharing Volumes between containers**

`$ docker run -itd --name ubuntu_server --volumes-from redis-server ubuntu /bin/bash`

`$ docker exec <CID> ls /data`

## **Read-only Volumes**

`$ docker run -v /docker/redis-data:/data:ro -it ubuntu rm -rf /data`





## Session: 8

### Deep Dive – Docker Compose



# Docker Compose

- Docker Compose is a “tool for defining and running your multi-container Docker applications”.
- Just need to download Docker Compose binary.
- Docker Compose is based on a `compose.yaml` file. This file defines all of the containers and settings you need to launch.
- Docker Compose supports all of the properties which can be defined using `docker run`.



# Installation of Docker Compose

- `$ curl -SL https://github.com/docker/compose/releases/download/v2.39.2/compose-linux-x86_64 -o /usr/local/bin/compose`
- `$ sudo chmod +x /usr/local/bin/compose`
- `$ compose.exe version`



# Lab

- Create Dockerfile and compose.yaml

- `$ cat Dockerfile`
- FROM alpine
- EXPOSE 3000
- `$ cat compose.yaml`
- web:
  - build: .
  - links:
    - - redis
  - ports:
    - - "3000"
    - - "8000"
- redis:
  - image: redis:alpine
  - volumes:
    - - /var/redis/data:/data



# Lab

- With the created compose.yml file in place, you can launch all the applications with a single command of up. If you wanted to bring up a single container, then you can use up <name>.
- `$ docker compose up`
- `$ docker compose ps`
- `$ docker compose logs`
- Docker Scale
- `$ docker compose scale web=2`  
`$ docker compose ps`
- To stop a set of containers you can use the command `$ docker compose stop`
- To remove all the containers use the command `$ docker compose rm`
- `$ docker compose down`



## Session: 9

### Docker commands/ Lab Environment



# Links/References

- Docker Homepage: <https://www.docker.com/>
- Install Docker CE: <https://docs.docker.com/install/linux/docker-ce/centos/>
- Docker Commands: <https://docs.docker.com/engine/reference/commandline/cli/>
- Docker Compose: <https://docs.docker.com/compose/>
- Docker Compose File: <https://docs.docker.com/compose/compose-file/>
- Docker Forum: <https://forums.docker.com/c/general-discussions/23>
- Docker Slack: <https://communityinviter.com/apps/dockercommunity/docker-community>



**Thank You**

