Docker - Administration

Hi There !!

Instructor : Vishal Saini

RedHat Certified Architect

AWS certified professional

Corporate Trainer and Consultant with 12+ years of rich experience in Linux, Cloud computing, Devops.

Microsoft CERTIFIED AZURE ADMINISTRATOR ASSOCIATE

RED HAT CERTIFIED ENGINEER Red Hat

aws certified Solutions Architect Associate

CERTIFIED kubernetes ADMINISTRATOR

About the Course: Outline

Module 1: Getting Started with Containers

Module 2: Fundamentals of Dockers

Module 3: Managing Docker images

Module 4: Understanding Docker file

Module 5: Docker storage and networking

Module 6: Docker swarm

Module 7: Docker compose

Module 1: Getting Started

Training Materials

The training materials which includes slides, lab modules and additional learning is available on

https://www.github.com/vsaini44/Dockeradmin

P 2 V 2 C

Any application, web or otherwise, will need to be hosted on a system for it to function. These systems are often called as servers. for example, web applications have to be hosted on a powerful web server and made available over internet to serve web pages.

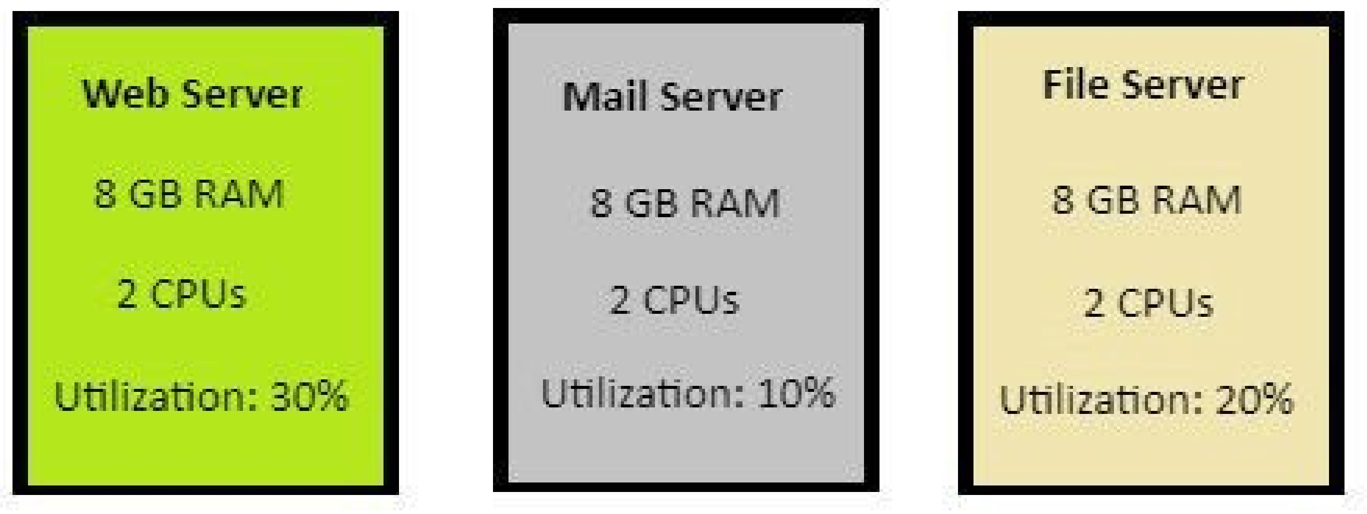
How these servers/systems have evolved over time and what physical systems, virtual machines and containers bring into the table?

Physical Systems / Servers:

Earlier days, Individual physical systems were used to host individual applications. Say, you need a web server to cater your web application, you will have to buy a dedicated physical machine with so and so configuration to power it up.

Again if you need a mail server to cater your mailboxes, you have to buy a new physical system. Your physical infrastructure keeps growing as your need grows.

Physical System



Individual Physical Servers

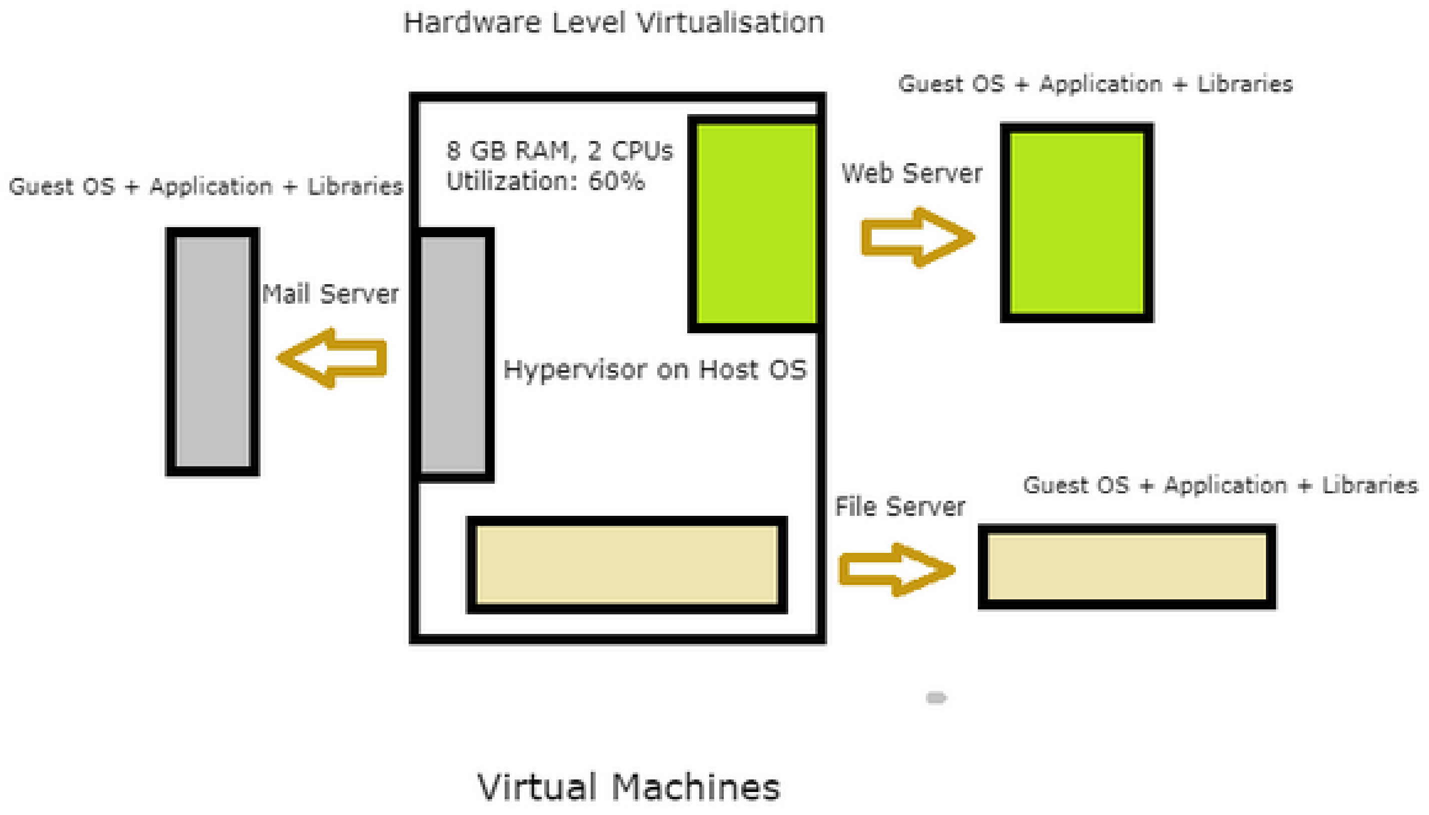
Virtual Machine ?

A Virtual machine provides a way to make a single physical system work as multiple isolated systems, resulting in higher infrastructure usage and reduced physical hardware infrastructure overhead.

How is it achieved?

Using hypervisor that runs on your host system, you can effectively split your underlying physical infrastructure into multiple smaller units that can run multiple isolated systems.

Virtual Machine

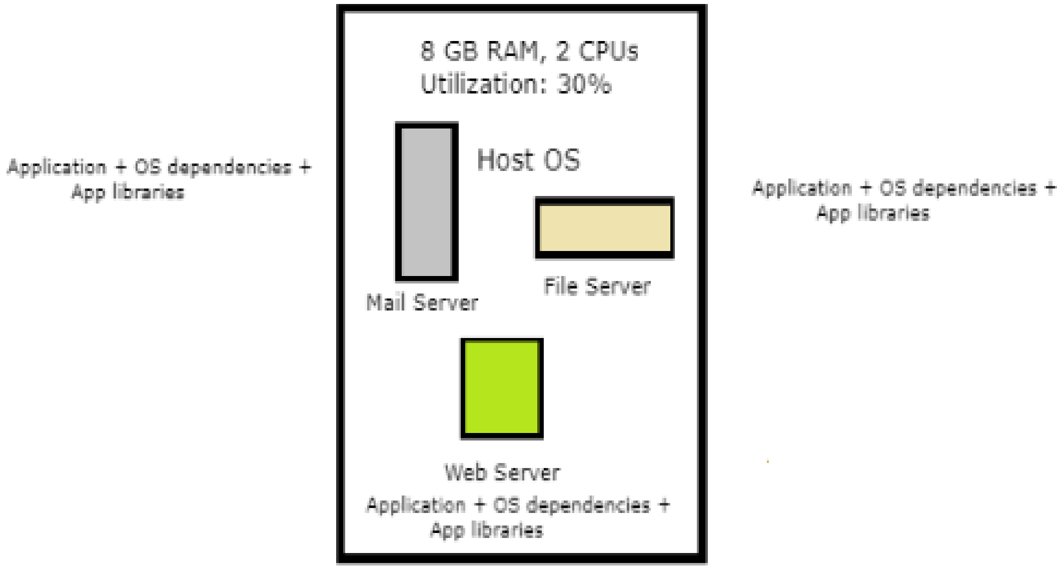

Hardware Level Virtualisation
Guest OS + Application + Libraries
8 GB RAM, 2 CPUs
Utilization: 60%
Web Server
Guest OS + Application + Libraries
Mail Server
Hypervisor on Host OS
File Server
Guest OS + Application + Libraries
Virtual Machines

Containers?

Containers enable virtualization in the sub-system level rather than hardware/OS level.

While hypervisors achieve virtualization by splitting hardware resources and running a separate set of OS on the virtualized hardware, containers utilize isolation at the subsystem level using namespaces and cgroups.

Subsystem level Virtualization


Application + OS dependencies + App libraries
8 GB RAM, 2 CPUs
Utilization: 30%
Host OS
Application + OS dependencies + App libraries
File Server
Mail Server
Web Server
Application + OS dependencies + App libraries

Containers

Advantages of Containers ?

> Portable

> Extremely small footprint

> Reduced IT management resources

> Quicker spinning of apps

What is Docker ?

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers.

Docker is currently the most popular container platform. Docker appeared on the market at the right time, and was open source from the beginning, which likely led to its current market domination.

The Need for Orchestration Systems

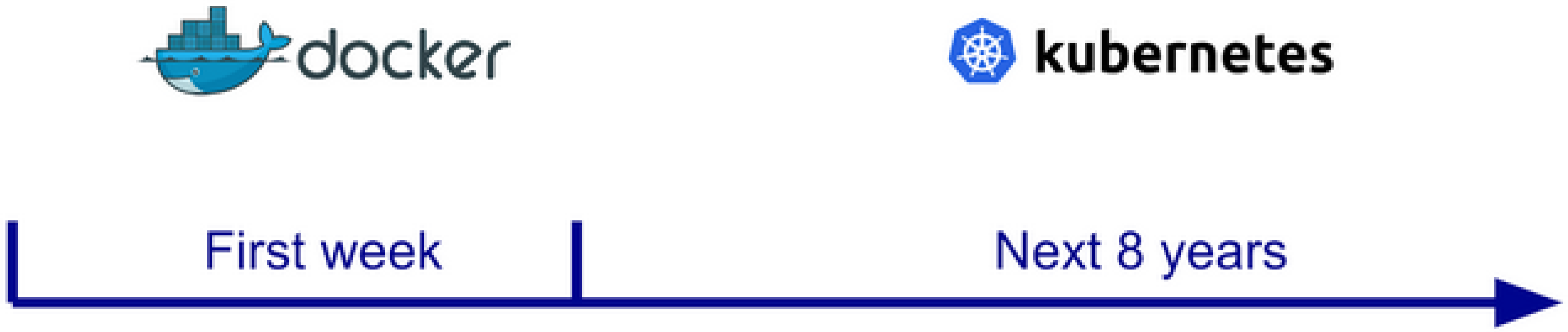
While Docker provided an open standard for packaging and distributing containerized applications, there arose a new problem.

> How would all of these containers be coordinated and scheduled?

>How do you seamlessly upgrade an application without any interruption of service?

> How do you monitor the health of an application, know when something goes wrong and seamlessly restart it?

Life of an application



Container Orchestration tools ?

docker SWARM

Apache MESOS™

kubernetes

Kubernetes ?

Kubernetes is the container orchestrator that was developed at Google which has been donated to the CNCF and is now open source.

It has the advantage of leveraging Google’s years of expertise in container management.

It is a comprehensive system for automating deployment, scheduling and scaling of containerized applications, and supports many containerization tools such as Docker.

Can you use Docker without Kubernetes?

Docker is commonly used without Kubernetes, in fact this is the norm. While Kubernetes offers many benefits, it is notoriously complex and there are many scenarios where the overhead of spinning up Kubernetes is unnecessary or unwanted.

In development environments it is common to use Docker without a container orchestrator like Kubernetes.

Module 2: Docker Fundamentals

Docker ??

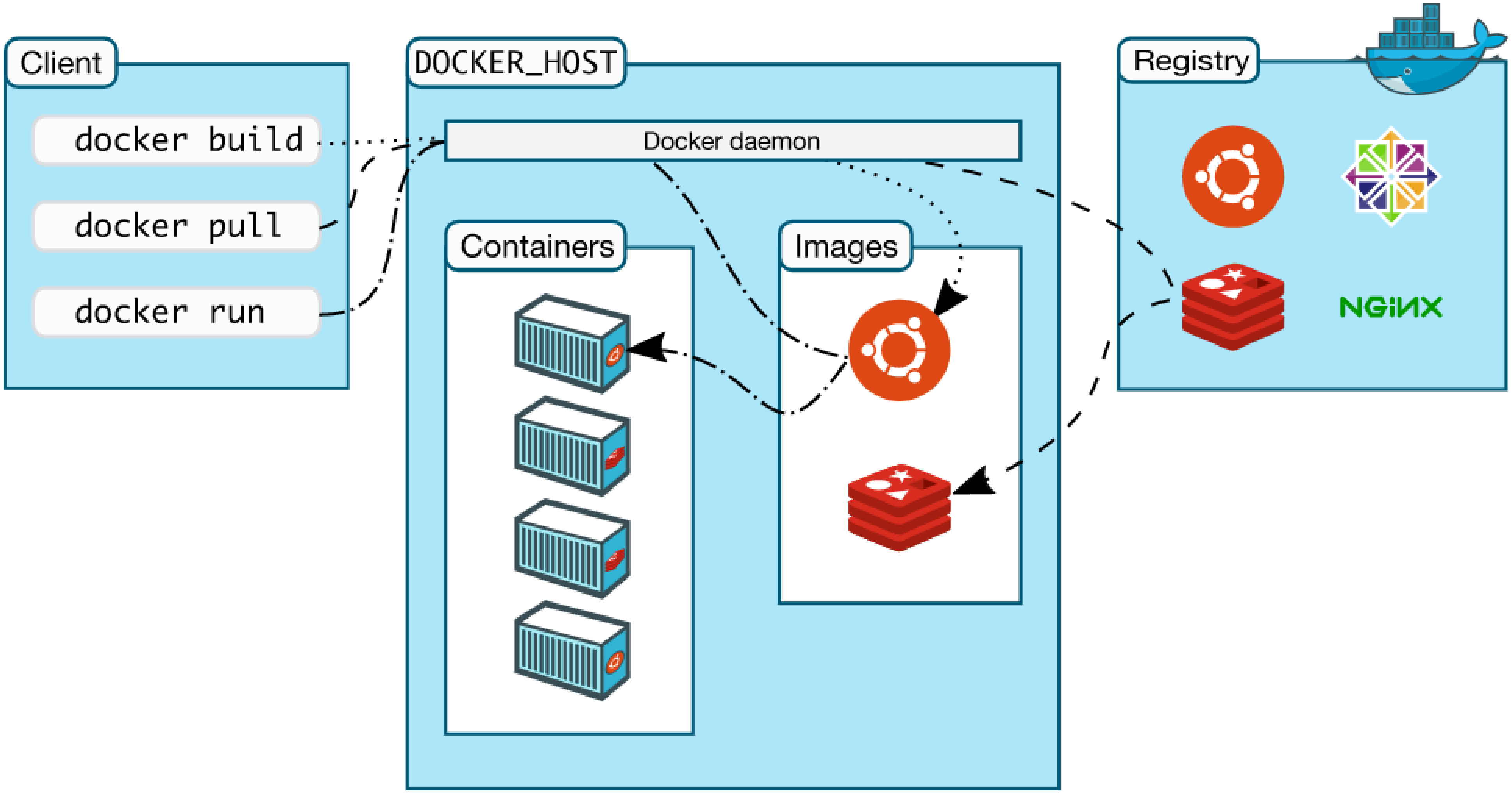
Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers.

Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels

Why Docker ??

- Fast, consistent delivery of your applications
- Responsive deployment and scaling
- Running more workloads on the same hardware

Docker Architecture



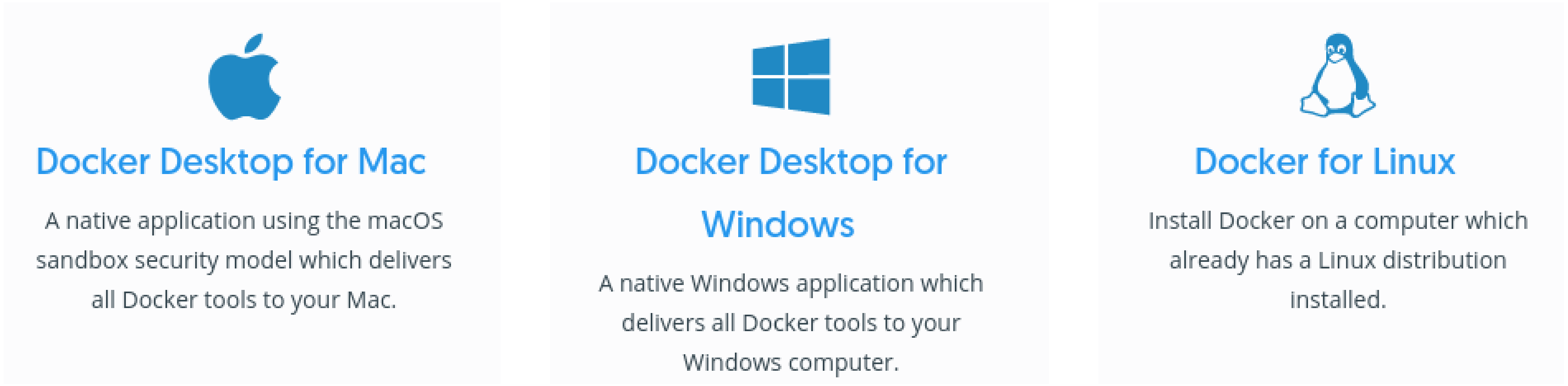
The underlying Technology

Docker is written in the Go programming language and takes advantage of several features of the Linux kernel to deliver its functionality.

Docker uses a technology called namespaces to provide the isolated workspace called the container.

Get Docker

You can download and install Docker on multiple platforms.



Docker Desktop for Mac

A native application using the macOS sandbox security model which delivers all Docker tools to your Mac.

Docker Desktop for Windows

A native Windows application which delivers all Docker tools to your Windows computer.

Docker for Linux

Install Docker on a computer which already has a Linux distribution installed.

Lab 1: Installing Docker

Estimated Time: 20 min

Follow the lab Module to get Docker installed on your server.

Docker fundamental commands

```
# docker images
# docker ps
# docker inspect <resource_name>
# docker run
# docker start
# docker stop
# docker rm
# docker rmi
# docker ps -a
```

Docker fundamental commands ...

```
# docker attach
# docker exec
# docker pause
# docker unpause
```

Lab 2: Performing docker basics

Estimated Time: 40 min

Follow the lab Module to get the hands on docker basic commands

Module 3: Docker Images

Docker images

Docker images are a read-only template which is a base foundation to create a container from. We need an image to start the container.

There are a lot of pre-built images out there on the docker hub. You can also have your own custom image built with the help of Dockerfile and the command “docker build”.

Docker search

To search an image on a Docker registry, run the following command.

```
# docker search [search term]
```

Fields :

NAME : Is the name of the docker image.

DESCRIPTION : A short description on what the image is about.

STARS : How many people have liked the image.

OFFICIAL : Specifies whether the image is built from a trusted source.

AUTOMATED : Tells whether the images is built automatically with a push in GitHub or Bitbucket repositories.

Docker pull

To pull an image from the Docker registry, run the following command:

```
# docker pull NAME[:TAG]
```

Fields:

NAME : The main group of images with similar role. For Example centos.

TAG : Image with a specific tag such as centos7.

Docker list

Run the below command to list all the images available locally on the system:

# docker images

Fields:

Repository : the name of the repository the image downlaoded from

Tag : tag (version) of the image repository

Image id : image id of repository

Created : When the image was created

Size : the size of the image

Image remove

Run the below command to remove the image from your local system:

```
# docker rmi repository:tagname
```

Note: Make sure the image is not been used by any container.

Docker commit

Create a new image from a container’s changes

Usage:

```
# docker commit <container> <repository:tag>
```

Docker save

Save one or more images to a tar archive (streamed to STDOUT by default)

Usage:

```
# docker save [OPTIONS] IMAGE [IMAGE...]
```

Docker load

Load an image from a tar archive or STDIN

Usage:

```
# docker load -i <tarfile>
```

Docker export

Export a container’s filesystem as a tar archive

Usage :

```
# docker export -o <tarfile> <CONTAINER>
```

Docker import

Import the contents from a tarball to create a filesystem image

Usage:

```
# docker import [OPTIONS] file|URL|- [REPOSITORY[:TAG]]
```

Docker push

Push an image or a repository to a registry

Usage:

```
# docker push [OPTIONS] NAME[:TAG]
```

Example

```
$ docker image tag myimage:latest
vsaini44/newimage:latest

$ docker image push vsaini44/newimage:latest
```

Lab 3: Image Operations

Estimated Time: 1 Hour

Follow the lab Module to get the hands on docker image commands

Module 4: Docker file

Dockerfile

Docker builds images automatically by reading the instructions from a Dockerfile -- a text file that contains all commands, in order, needed to build a given image.

A Docker image consists of read-only layers each of which represents a Dockerfile instruction. The layers are stacked and each one is a delta of the changes from the previous layer

Dockerfile Example

```
FROM ubuntu:18.04

COPY . /app

RUN make /app

CMD python /app/app.py
```

Each instruction creates one layer:

- **FROM creates a layer from the ubuntu:18.04 Docker image.**
- **COPY adds files from your Docker client’s current directory.**
- **RUN builds your application with make.**
- **CMD specifies what command to run within the container.**

Instruction

# From : used to specify the valid docker base image.

# Maintainer: used to specify the author who creates the new docker image.

# Label: used to specify the metadata information to an image

# Expose: used to inform about the network ports that the container listens on runtime.

# Add: instruction used to copy files/directories inside the container

Instruction

# Run : used to execute any command on top of the current image

# Cmd: used to set a command to be executed when running a container

# Entrypoint: used to configure and run a container as an executable.

Build

Once the Dockerfile is created you can build the image using the following command

```
# docker build -t myimg:latest .
```

Lab 4: Docker file

Estimated Time: 30 Hour

Follow the lab Module to get the hands on docker file.

Module 5: Docker Volume and Network

Volume

In order to understand Docker volumes, it is important to first understand how the Docker file system works.

A Docker image is a collection of read-only layers. When you launch a container from an image, Docker adds a read-write layer to the top of that stack of read-only layers. Docker calls this the Union File System. Any time a file is changed, Docker makes a copy of the file from the read-only layers up into the top read-write layer. This leaves the original (read-only) file unchanged. When a container is deleted, that top read-write layer is lost. This means that any changes made after the container was launched are now gone.

Volume

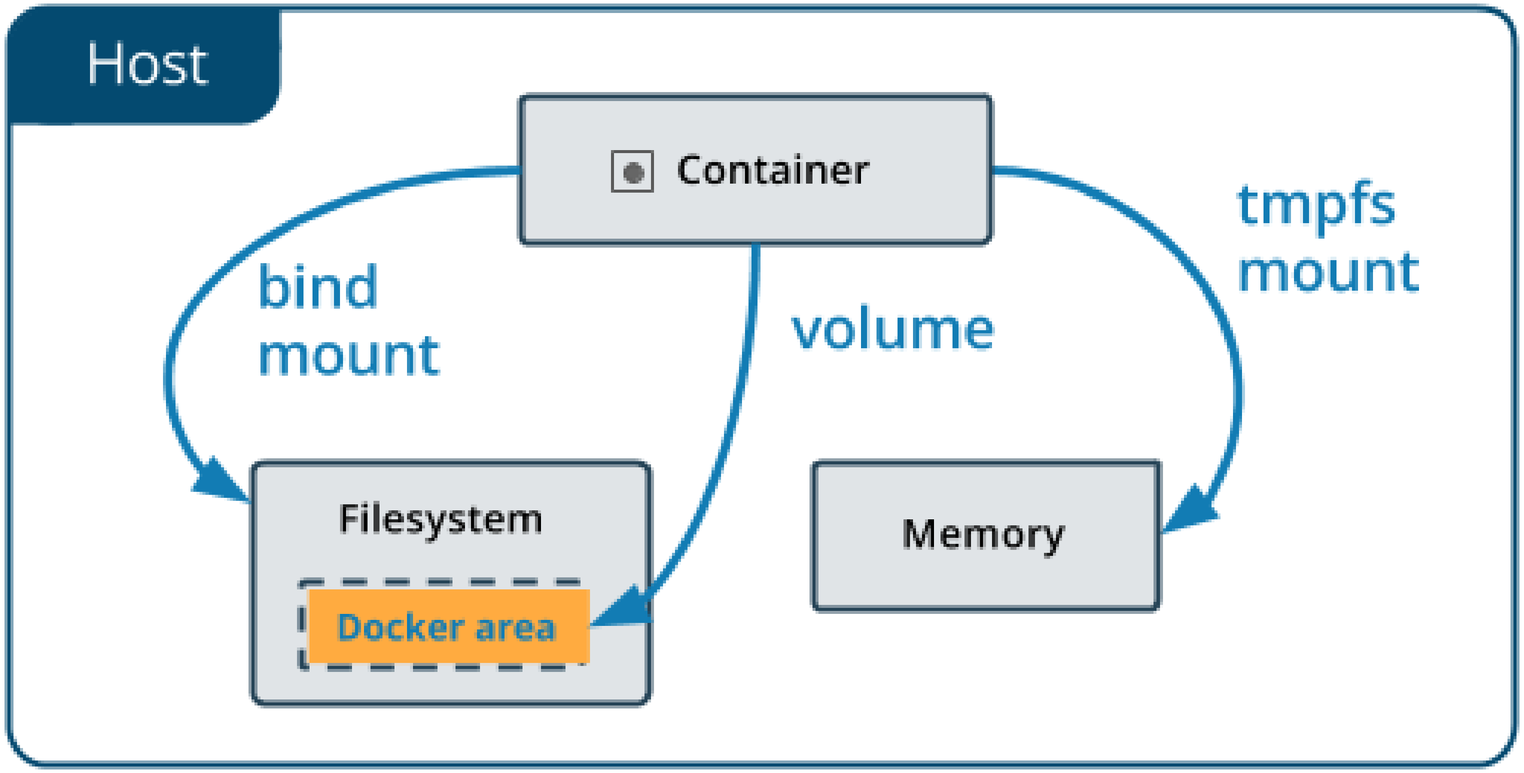
Volumes are the preferred mechanism for persisting data generated by and used by Docker containers.

```
# docker volume ls
```

Volume

- **Volumes are easier to back up or migrate than bind mounts.**
- **You can manage volumes using Docker CLI commands or the Docker API.**
- **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, to encrypt the contents of volumes, or to add other functionality.**
- **New volumes can have their content pre-populated by a container.**
- **Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts.**

Volume


Host
Container
bind mount
volume
tmpfs mount
Filesystem
Docker area
Memory

Volume

Let us begin first with the most basic operation i.e. mounting a data volume in one of our containers

```
$ docker run -it -v /data --name container1 nginx
```

Docker Network

One of the reasons Docker containers and services are so powerful is that you can 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 Driver

Bridge

Host

Overlay

Macvlan

None

Network plugins

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.

Host

For standalone containers, remove network isolation between the container and the Docker host, and use the host’s networking directly

Overlay

Overlay networks connect multiple Docker daemons 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.

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. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network

Bridge network creation

# docker network create my-net

# docker network rm my-net

To create the container using the docker network you just created

#docker create --name my-nginx   --network my-net nginx:latest

Lab 5: Docker Volume and Network

Estimated Time: 45 Hour

Follow the lab Module to get the hands on docker volume and network.

Module 6: Docker Swarm

What is Docker Swarm

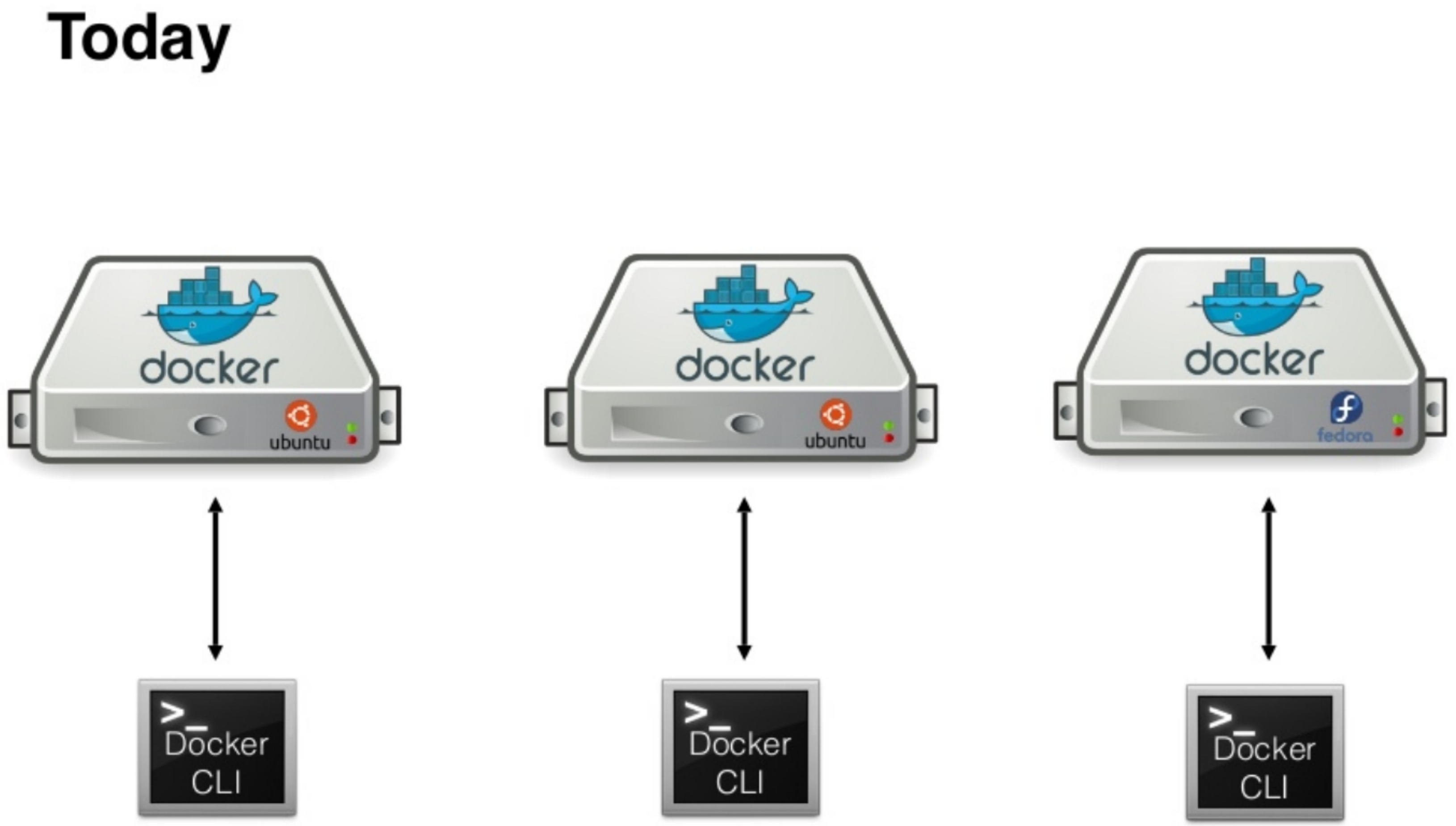
A Docker Swarm is a group of either physical or virtual machines that are running the Docker application and that have been configured to join together in a cluster.

Once a group of machines have been clustered together, you can still run the Docker commands that you're used to, but they will now be carried out by the machines in your cluster.

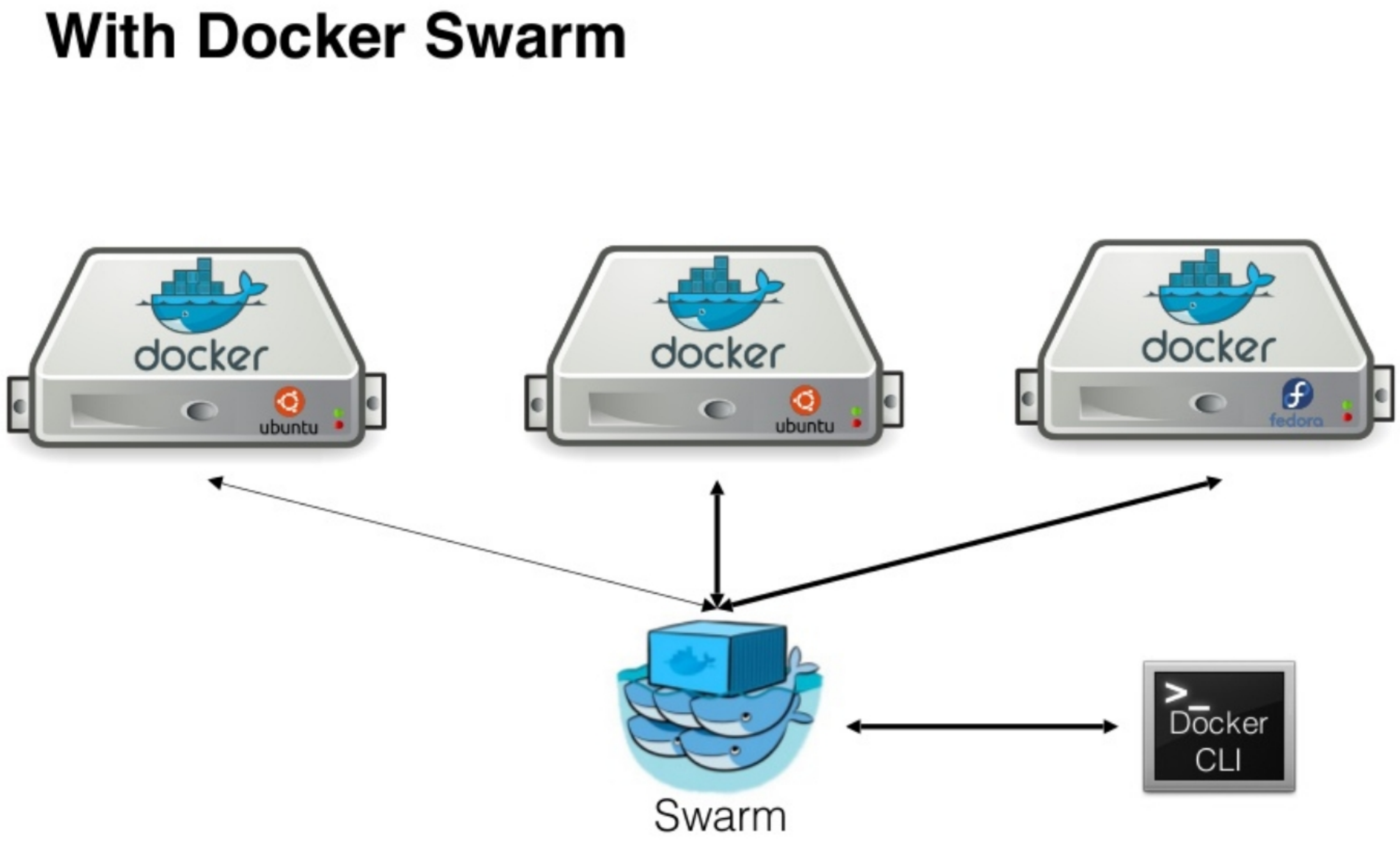
What is Docker Swarm

The activities of the cluster are controlled by a swarm manager, and machines that have joined the cluster are referred to as nodes

What is Docker Swarm



What is Docker Swarm



Swarm in a nutshell

Exposed server docker engines as a single virtual engine

Serves the standard docker api

Extremely easy to get started