Get the best out of Live Sessions

HOW?

e!

Check your Internet Connection

Log in 10 mins before, and check your internet connection to avoid any network issues during the LIVE session.

Speak with the Instructor

By default, you will be on mute to avoid any background noise. However, if required you will be **unmuted by instructor**.

Clear Your Doubts

Feel free to clear your doubts. Use the "**Questions**" tab on your webinar tool to interact with the instructor at any point during the class.

Let us know if you liked our content

Please share feedback after each class. It will help us to enhance your learning experience.

edureka!

edureka!

DevOps Certification Training

COURSE OUTLINE
MODULE 06

1. Introduction to DevOps
2. Version Control with Git
3. Git and Jenkins
4. Continuous Integration with Jenkins
5. Configuration Management using Ansible
6. Containerization using Docker Part - I
7. Containerization using Docker Part - II
8. Container Orchestration Using Kubernetes Part-I
9. Container Orchestration Using Kubernetes Part-II
10. Monitoring Using Prometheus and Grafana
11. Provisioning Infrastructure using Terraform Part - I
12. Provisioning Infrastructure using Terraform Part - II

edureka!

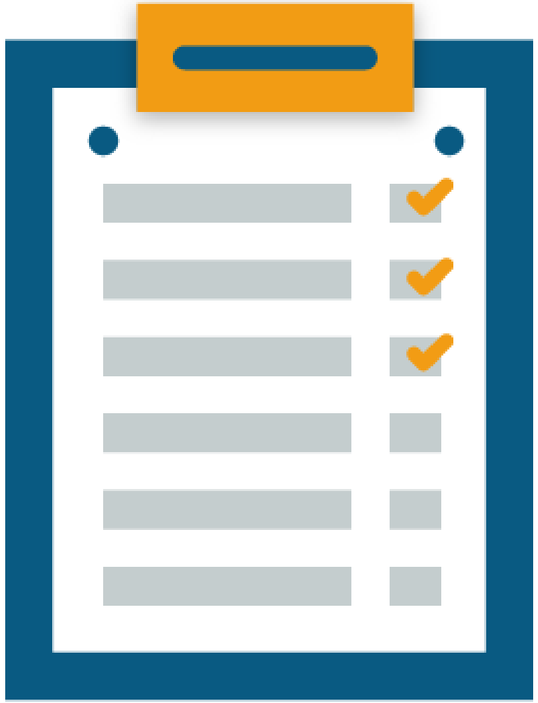
Module 6 – Containerization using Docker Part - I

edureka!

Topics

Following are the topics covered in this module:

- Containerization
- Namsepaces
- Docker
- Docker Architecture
- Container Lifecycle
- Docker CLI
- Port Binding
- Detached and Foreground Mode
- Dockerfile
- Dockerfile Instructions
- Docker Image



edureka!

Objectives

After completing this module, you should be able to:

- Understand Containerization
- Learn the evolution of virtualization to containers
- Understand the Docker Architecture
- Perform Various actions using Docker CLI
- Bind container ports to the Machine ports
- Run containers in different modes
- Write and build a Dockerfile to create a Docker Image

Yelp: Consistency

edureka!

Yelp

Yelp is a reviews and recommendations company based out of America

1

The company as of 2019 had hosted over 192 million user reviews

2

It hosts over 178 million active monthly users across different platforms

3

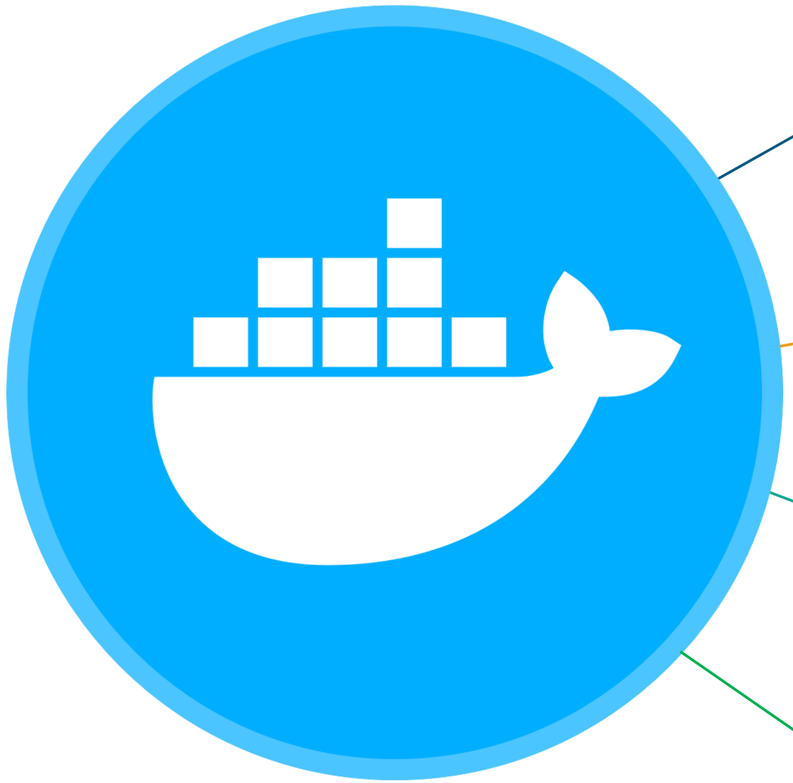
edureka!

Yelp: Monolithic Architecture

- Yelp originally was developed in monolithic architecture
- This disabled the application from bringing about major changes for the ever-growing user base
- In order to adapt, Yelp has been trying to move its newer services to AWS

But a huge base of the application's critical services still runs in-house which cannot be readily moved to cloud

Docker Bridges the Gap



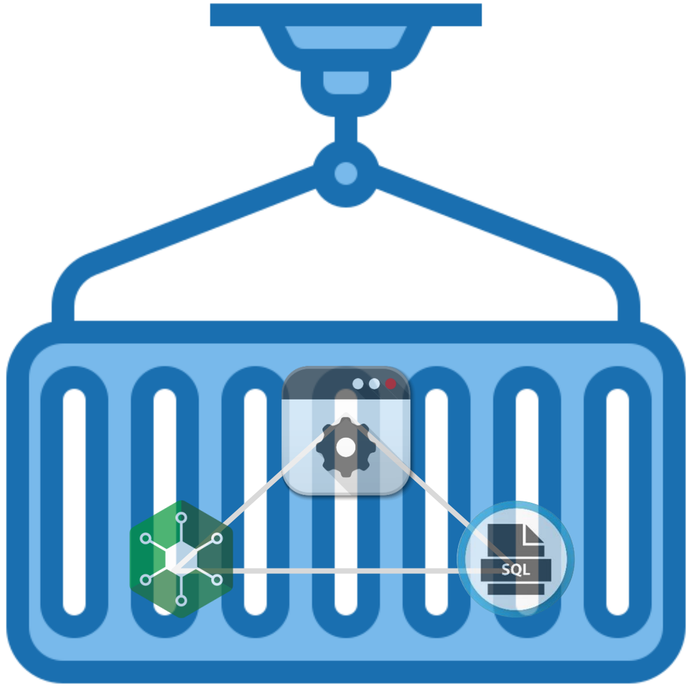
1. Docker helped Yelp to smooth out the differences between its traditional Data Centers and AWS Instances
2. The easy-to-use nature of Docker enabled Yelp to containerize its critical services and then move them to AWS
3. Docker has now become a very integral part of Yelp's infrastructure
4. It has enabled developers to be more independent, thus making the deployments that much faster

Containerization

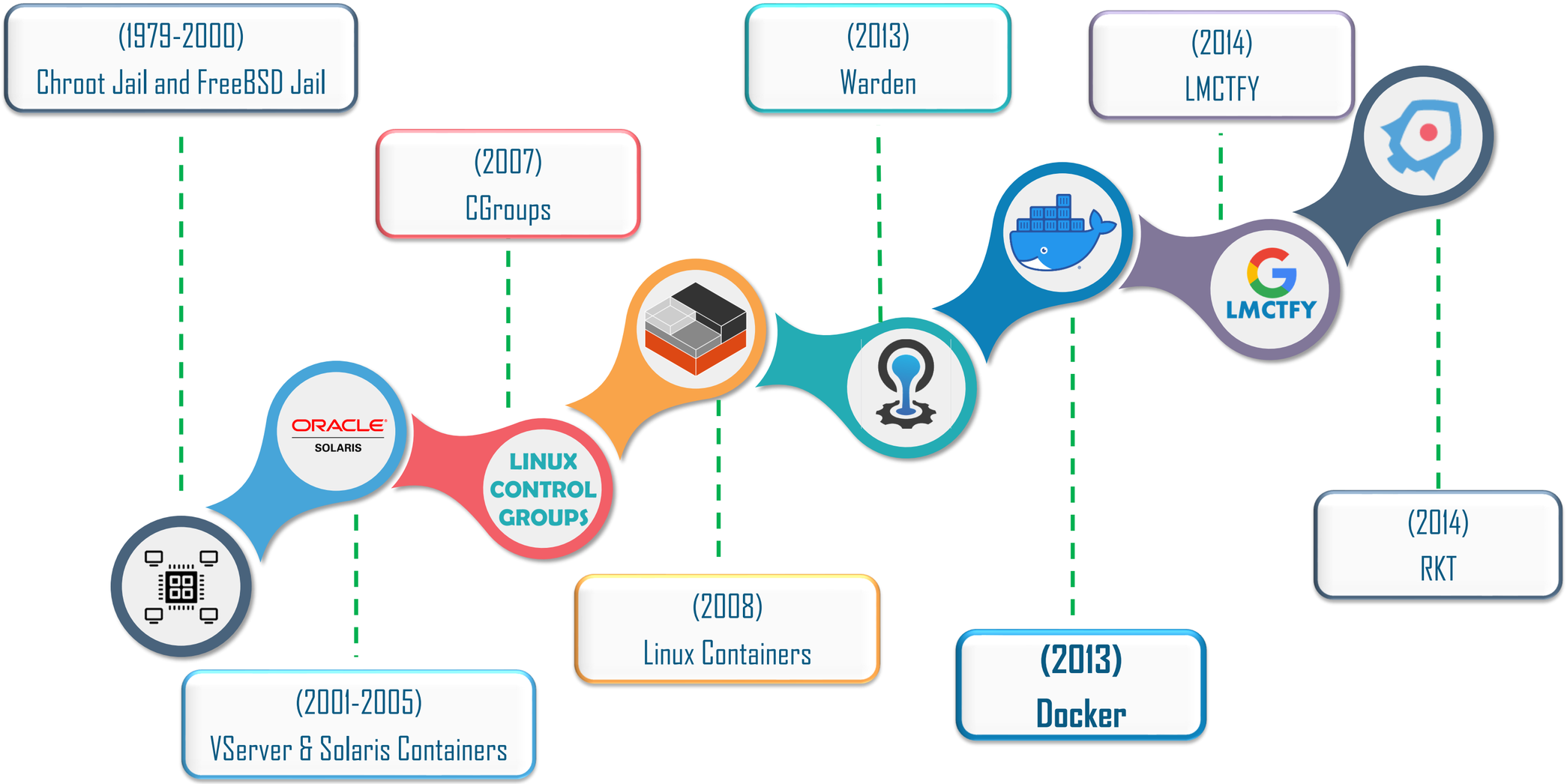
edureka!

What is Containerization?

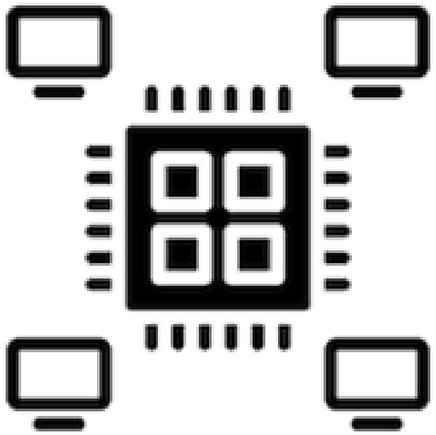
It is the process of packaging the application and all its dependencies to execute them in an efficient and hassle-free way across different environments. A single unit of this bundled package is known as a **Container**



A Short History of Containers (HoC)



HoC: Chroot Jail



1979

- The Chroot system was introduced in Version 7 of the Unix OS in 1979
- It stands for Change Root
- It allowed simple isolation of a process and its sub-processes from the rest of the OS
- This method could not provide complete isolation as root processes could exit the Chroot system

Note: FreeBSD Jail was released in the FreeBSD OS in 2000. FreeBSD introduced the concept of isolating all the process related activities from the view of the filesystem

HoC: Linux VServer & Solaris Containers

2001

ORACLE

SOLARIS

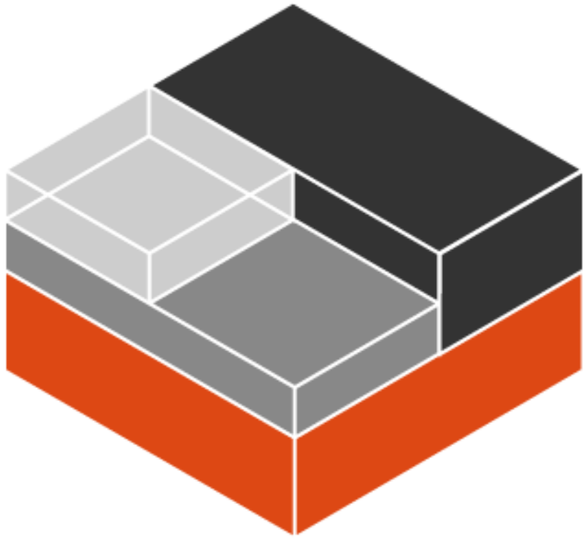
2004

- VServer introduced a system which utilized chroot with “security contexts” and OS-level virtualization
- This enabled users to run multiple Linux distros on a single distribution
- Solaris Containers provide an implementation of Linux VServer
- It utilized system resource controls and implemented a separation mechanism called “zone”

HoC: CGroups

- Google introduced CGroups in 2007
- CGroups allow users to allocate resources to the isolated processes
- The cgconfig service can be configured to make the cgroups persist over reboots

HoC: Linux Containers (LXC)



2008

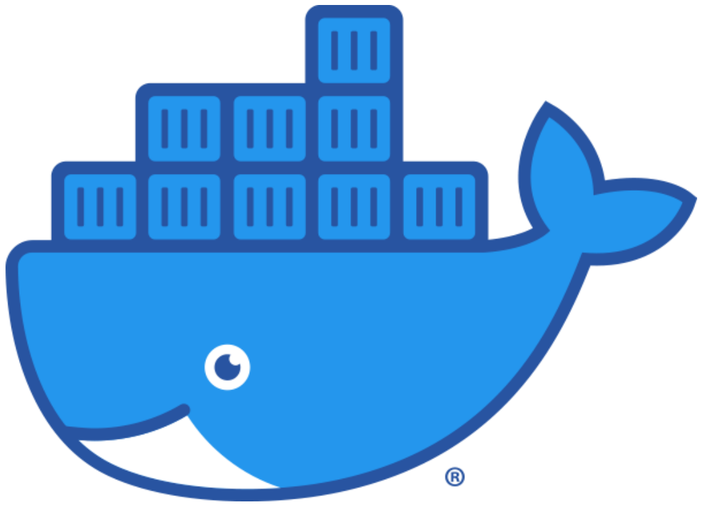
- LXC works similar in implementation to VServer and Solaris Containers
- However, it adds CGroups to the container implementation
- It aims to provide system-level containers without the overhead of Virtual Machines

HoC: Warden

2013

- CloudFoundary introduced Warden in 2013
- It is an API that manages isolated, resource controlled environments
- The first implementation of Warden used LXC as base

HoC: Docker



2013

- Docker was made public as an open-source containerization technology in 2013
- It initially used LXC execution engine
- A year later, with the release of Docker v0.9, docker introduced their own component replacing LXC

HoC: LMCTFY by Google

- LMCTFY stands for Let Me Contain That For You
- It was Google's version of Linux Container Stack
- Google engineers collaborated with Docker to work on libcontainer

HoC: rkt by CoreOS

2014

- CoreOS released their own version of Container platform called “Rocket” in 2014
- It was released with the goal of being the most secure and composable container engine
- Though, initially launched as an alternative to Docker, it has since added support for docker images

Why use Containers?

Performance Overhead

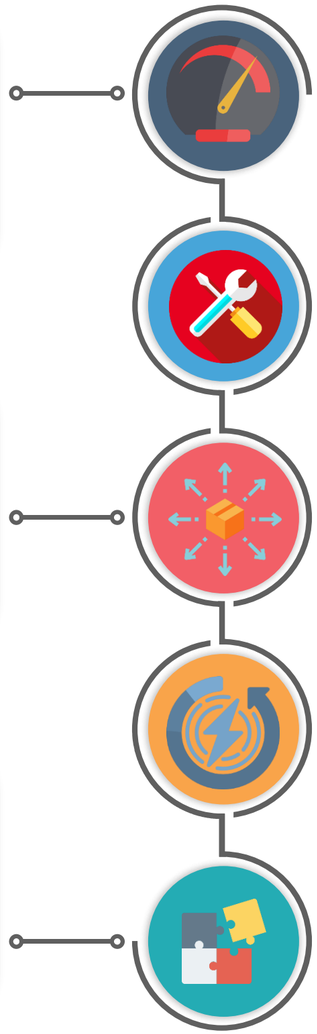
Containers work on top Host OS's Kernel, therefore, there is little to no performance overhead

Platform Independent

Containers can be deployed to platforms with different network topologies and security policies without any hassle

Modularity

Depending upon the approach, containers work seamlessly in a monolithic as well as the microservice environment



Easily Manageable

Since all the dependencies run inside an isolated instance, it is easy to manipulate and make changes to the application

Instant Boot

Containerized application has zero boot time making them available instantaneously

Containers: Working

edureka!

What is a Container?

- It is an Operating System Level Virtualization technology
- Detaches the application and its dependencies from the rest of the system
- utilizes **namespaces(Na)** and **cgroups(CG)** feature of Linux Kernel to isolate processes

Container: Working

Containers utilize two of the Linux Kernel features:

CG

CGROUPS

- Control Groups is a Linux Kernel feature
- Allows segregating the processes and the required resources
- Manages the isolated resources and processes as a single module

Na

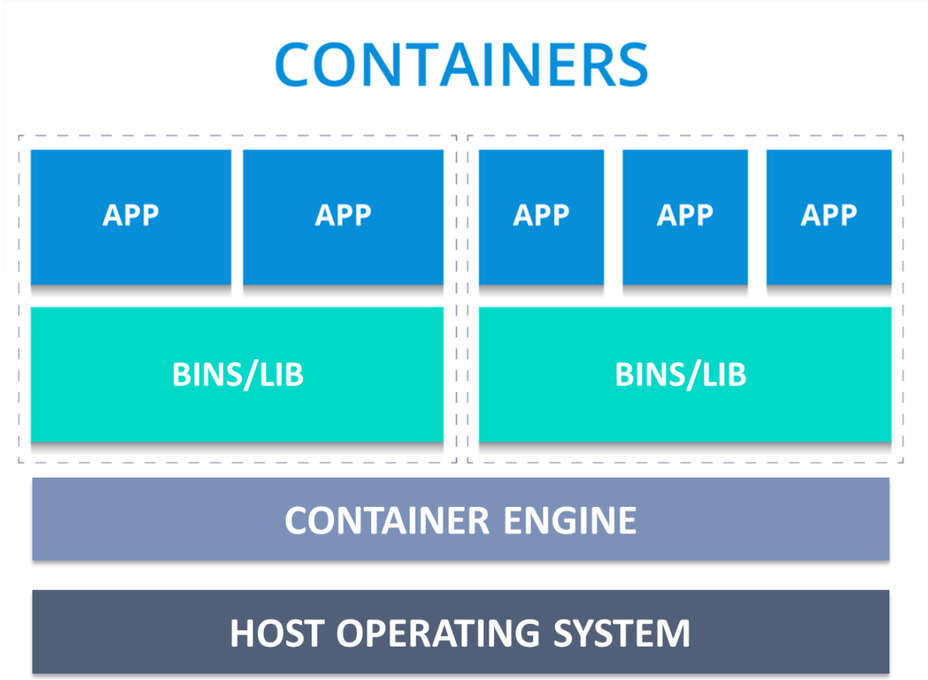
{...}

NAMESPACES

- Used to limit process's access to view the system
- Process is unaware of everything running outside of its own namespace

Container: Working

- A container is just another process for the host operating system
- Works in contained environment under the OS
- Gets restricted view and access to other system processes, resources, and environments



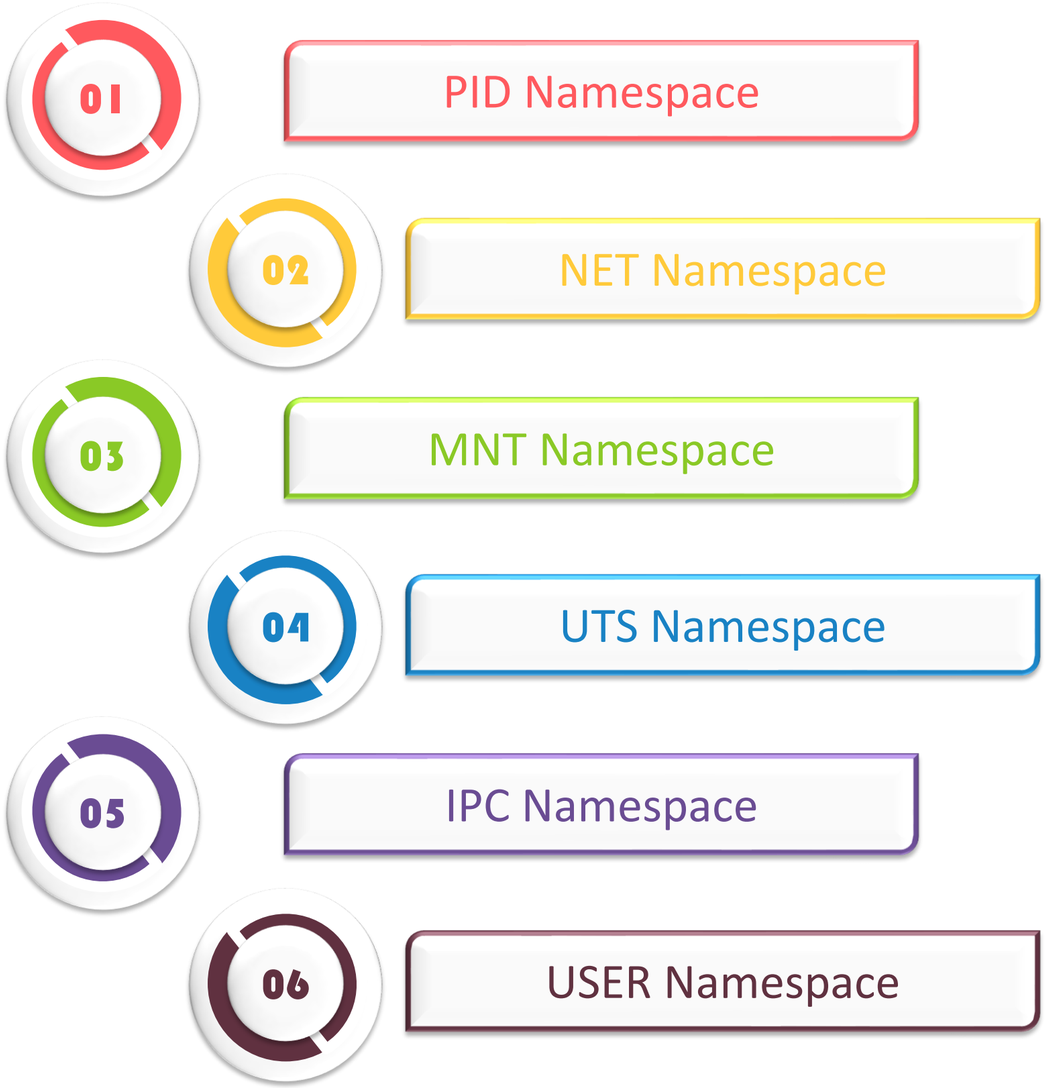
edureka!

Introduction to Namespaces

edureka!

Namespaces

- Provides a system resource, a layer of abstraction
- Application presumes it has an isolated instance of the resource
- There are six types of namespaces



edureka!

Types of Namespaces

PID Namespace

- Processes within PID namespaces can access/view other processes within that namespace only
- Each PID namespace starts from PID1
- If PID1 is killed, all the child processes are terminated

01

02

NET Namespace

- Segregates the processes by providing them their own private network
- Private network can include its own routing table, iptables, sockets and so on

MNT Namespace

- Enables processes to have their own root file system
- Mounts can be private or shared depending upon application requirements

03

04

UTS Namespace

- Provides the processes a separate copy of hostname to work with
- This isolates kernel for the processes and system identifiers

IPC Namespace

- Used to provide access to IPC resources
- Isolates the inter-process communication resource
- Resources such as IPC semaphores, IPC message queues, IPC shared memory can be accessed.

05

06

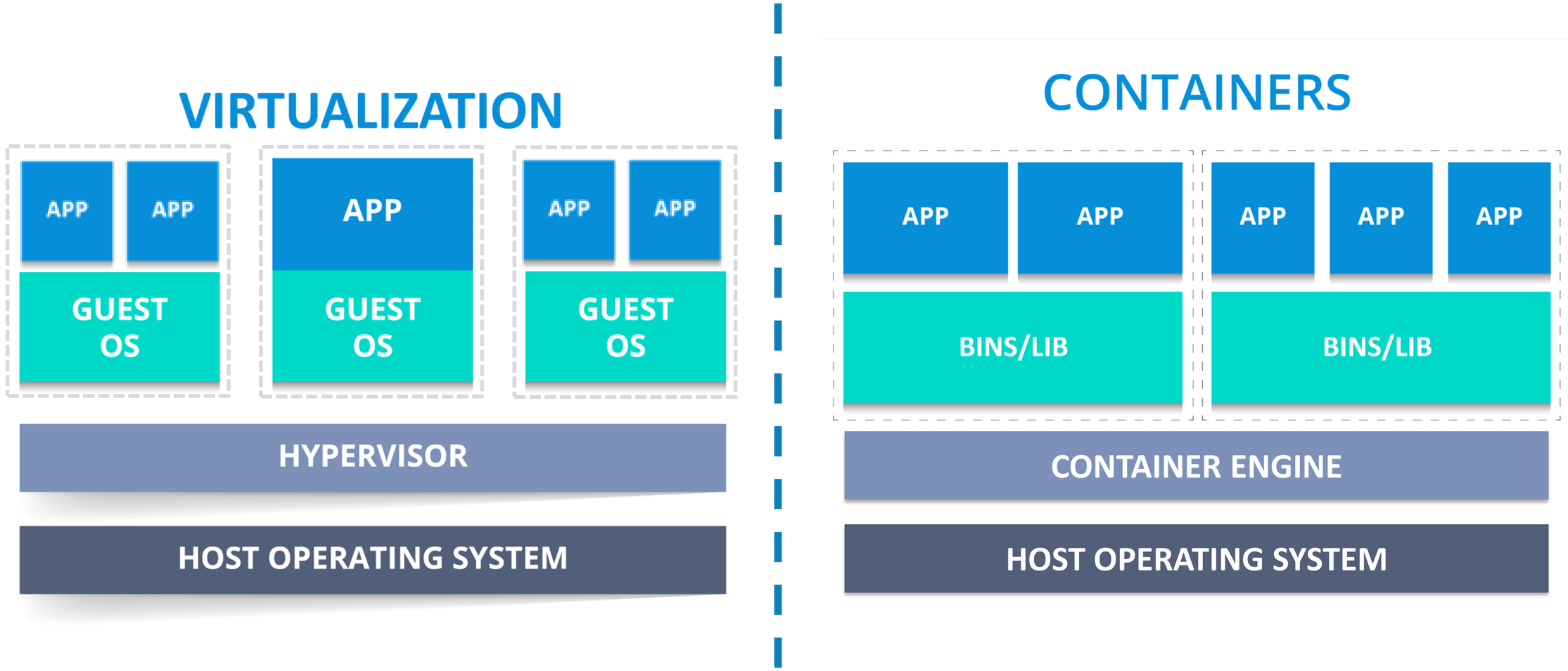
USER Namespace

- Provides identification and isolation to a set of processes
- Different UIDs and GIDs can be mapped to the processes

Containers vs Virtual Machines

edureka!

Virtual Machine vs Container



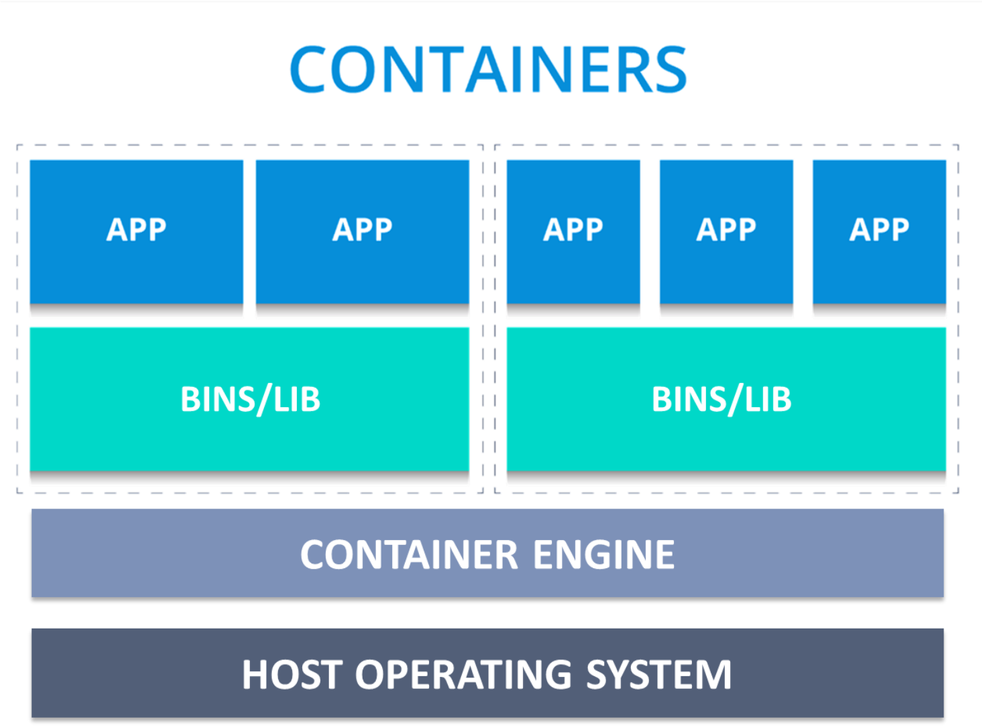
edureka!

Containers vs Virtual Machines

	Containers	Virtual Machines
Number of Apps on Server	Can run a limited number of applications, depending on VM Configuration	Can run multiple instances of different applications together
Resource Overhead	Light-weight with little to no overhead	Highly resource-intensive with performance overhead of managing multiple Oss
Speed of Deployment	Very fast deployment. Container images require seconds to deploy new instances	Very slow Deployment. Requires setting up OS, app dependencies, etc.
Security	Users usually require root level permissions to execute container related tasks	Provides better security
Portability	Highly portable with emphasis on consistency across environments	Very limited portability

Why use Containers?

- Containers are light compared to a VM, as they directly utilize the Host Operating System's Kernel
- For a single VM instance, the machine hardware deals with 3 separate Kernels

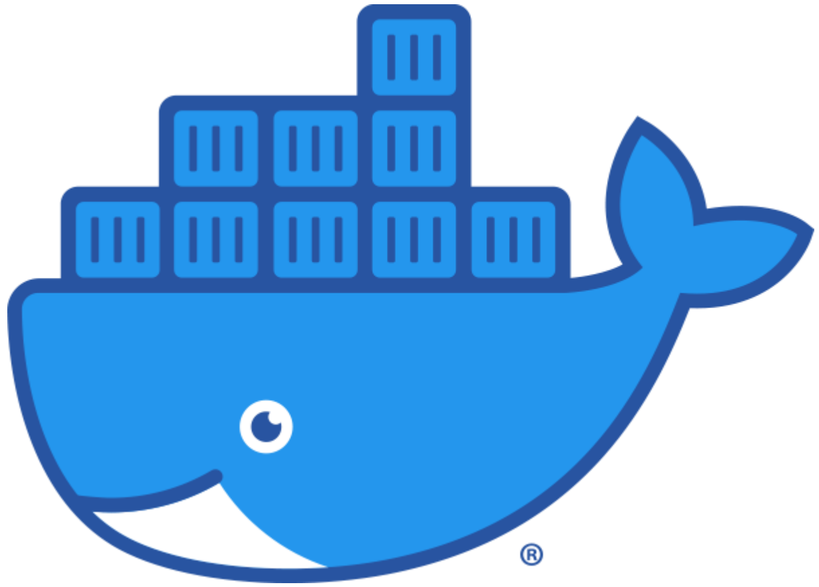


Containerization using Docker

edureka!

Introduction to Docker

Docker is one of the most popular Container engines today because of the way it handles containers



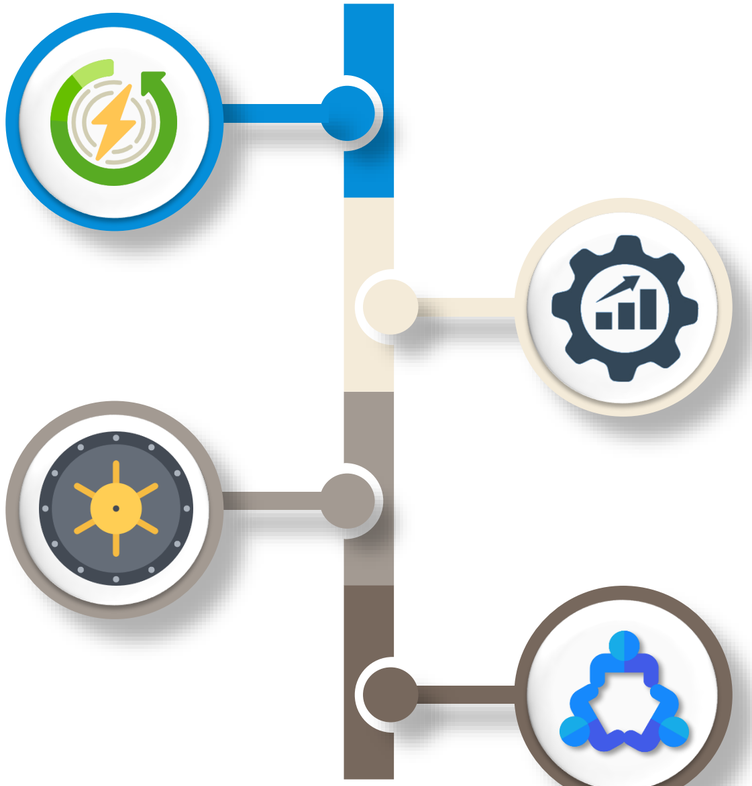
Docker: Features

Fast Configuration

- The set up process is very quick and easy
- It separates the requirements of the infra from the requirements of the application

Application Isolation

- Provides applications/service isolation
- Applications inside containers execute independent of the rest of the system



Productivity

- Rapid deployment, in turn, increases the productivity
- Reduced resource utilisation is another factor increasing the productivity

Swarm

- It helps clustering and scheduling docker containers
- It enables controlling cluster of docker hosts from a single point

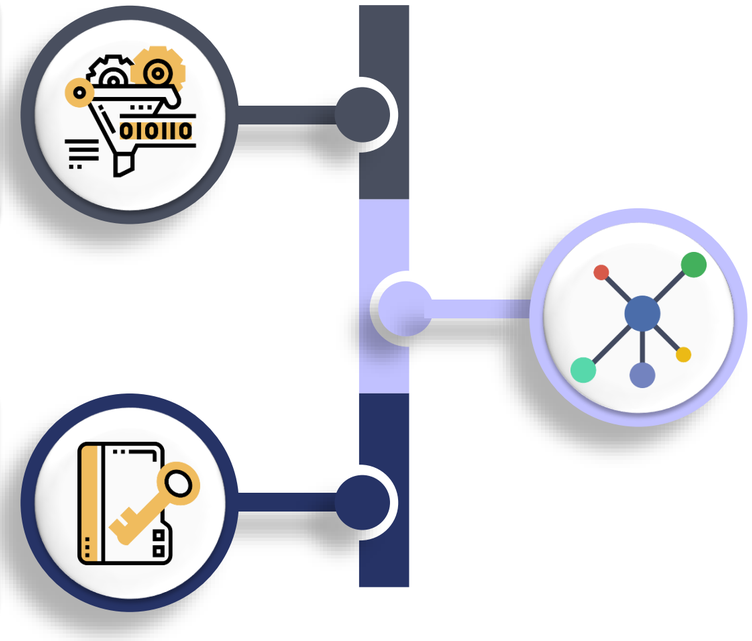
Docker: Features (Contd.)

Services

- Services use a set of tasks to define the state of a container inside a cluster
- Swarm manager takes the defined service as input and schedules it on the nodes

Security

- Allows saving secrets inside the swarm cluster
- These secrets can then be accessed by the required service



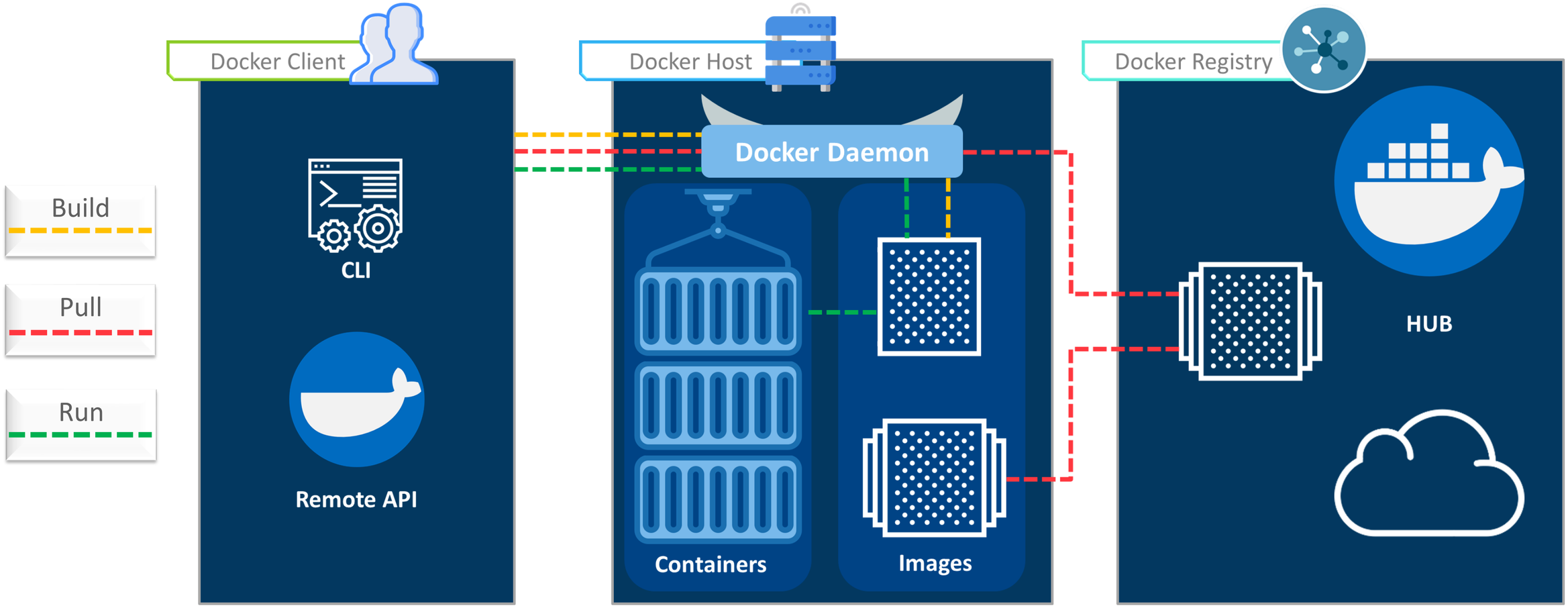
Service Discovery

- Mesh allows service discovery through the same port on all the nodes in swarm
- It is possible to reach the node even if the service is not deployed on it

Docker Architecture

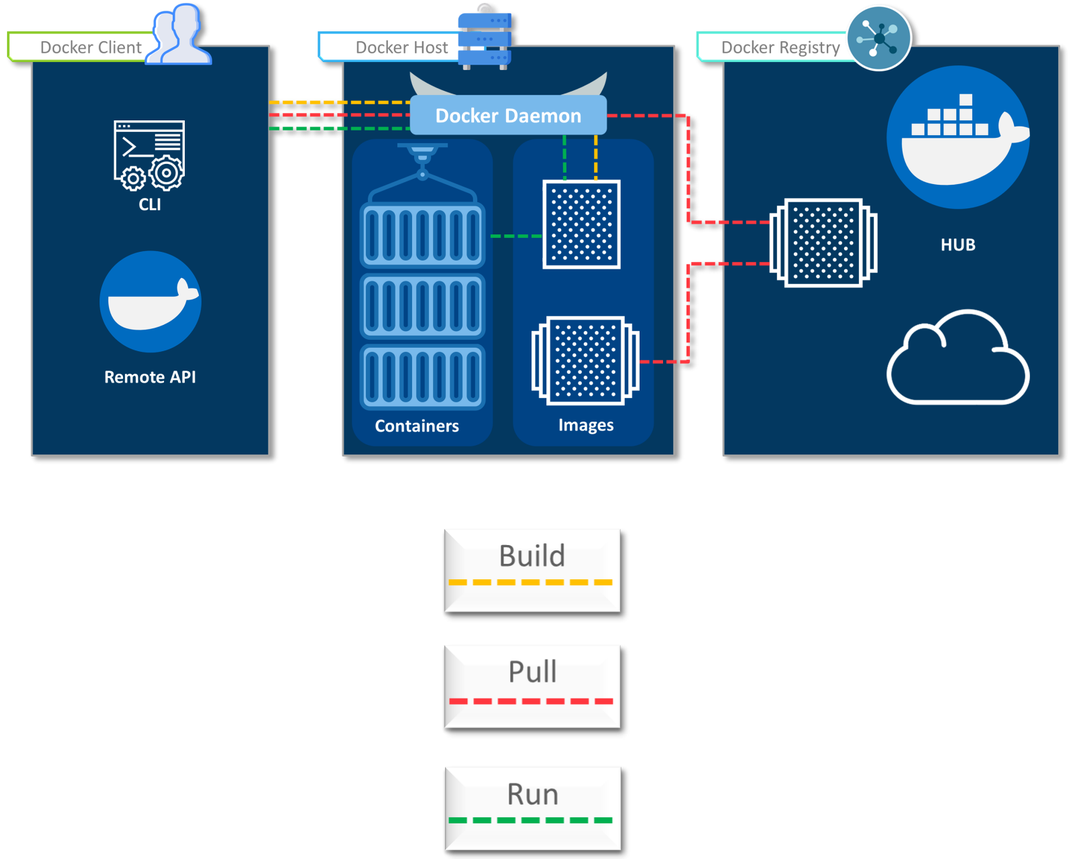
edureka!

Docker Architecture



Docker Architecture (Contd.)

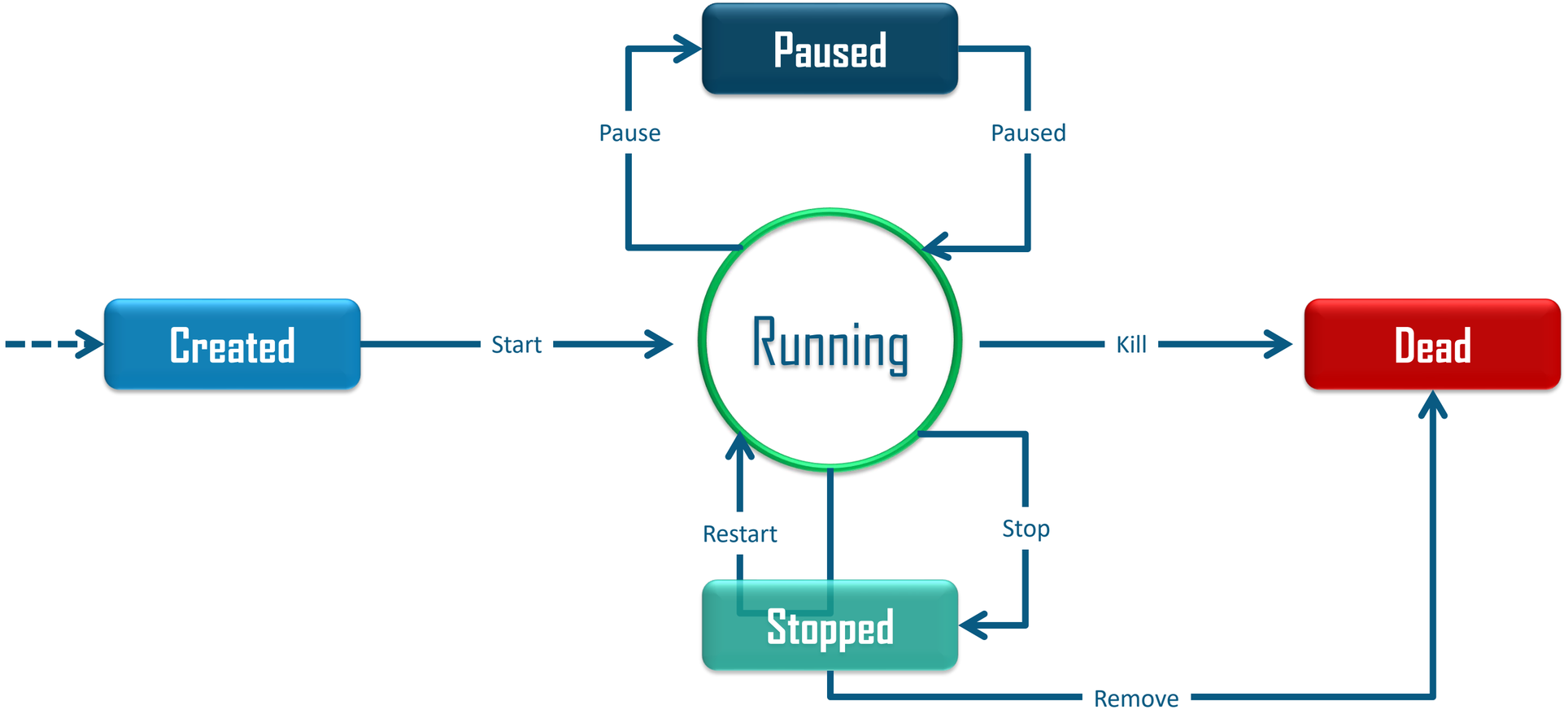
1. Docker works in a client-server architecture
2. The client sends the control commands to the Docker Daemon through a REST API
3. Docker Daemon is responsible for building, running and distributing the containers
4. The Client and Daemon can be on the same system or the client can connect to a remote Daemon



Container Lifecycle

edureka!

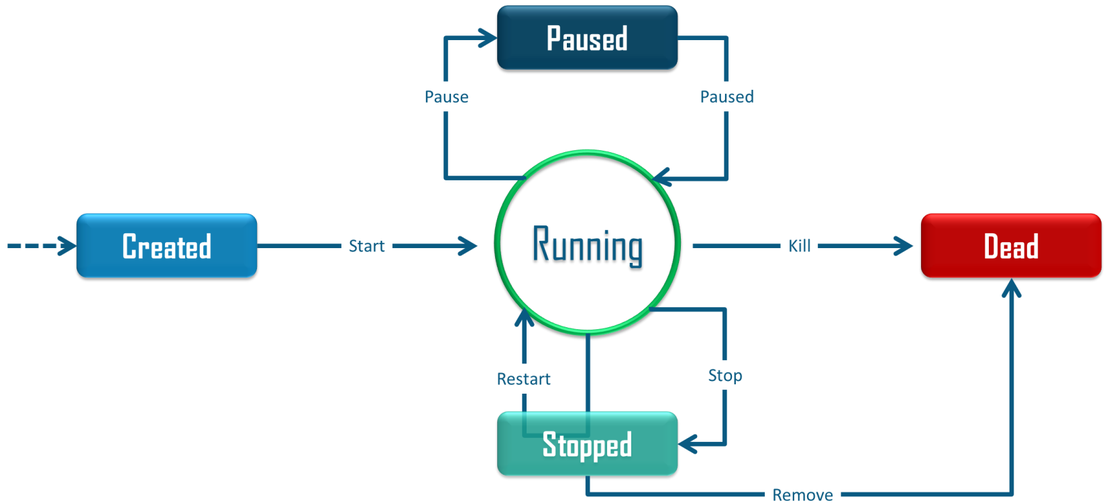
Container Lifecycle



Container Lifecycle

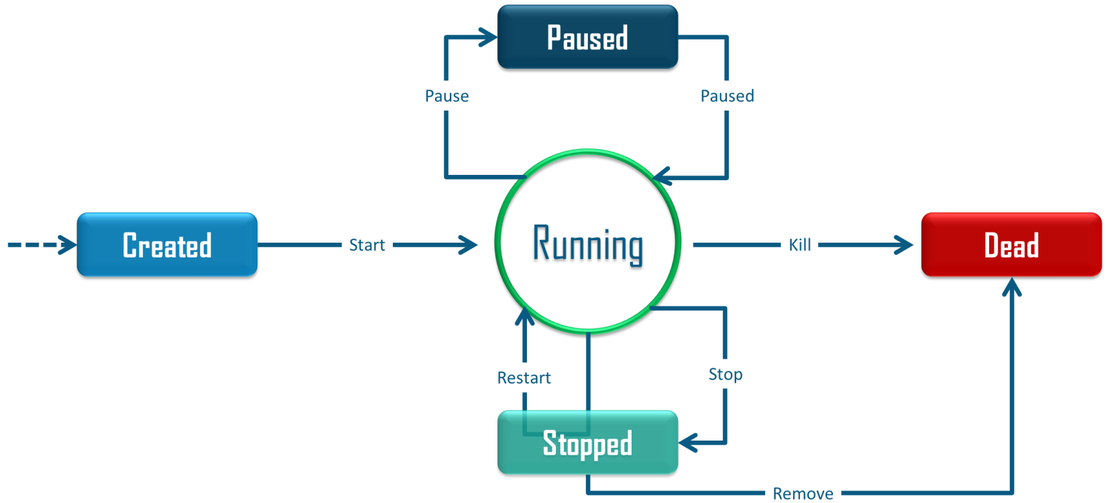
A newly created container can be in one of six states:

- Created
- Running
- Paused
- Stopped
- Restarted
- Dead



Container Lifecycle

State	Description
Created	A container that has been created but not yet started
Running	A container in its normal working state
Paused	Container whose processes have been paused at the moment
Stopped	A container that is no longer working. Also known as Exited
Restarting	A previously stopped container which is now being restarted
Dead	A container which is no longer in use and is discarded



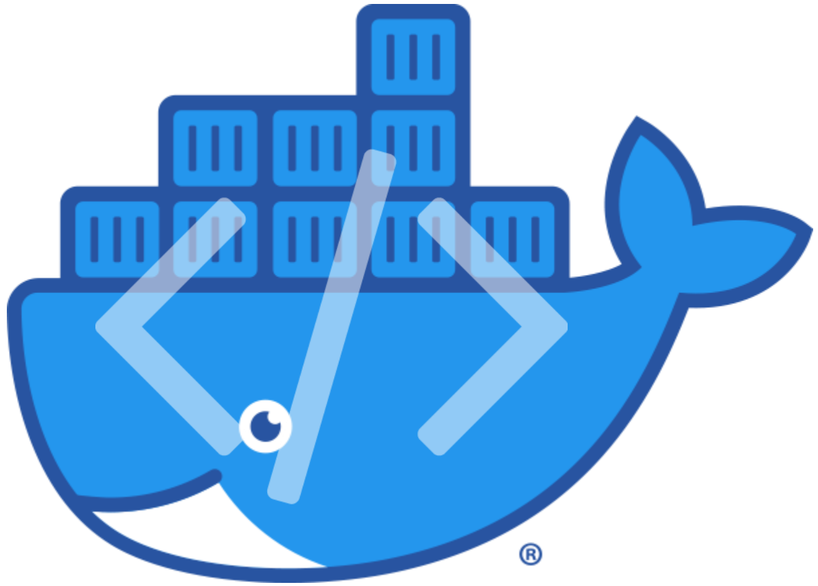
edureka!

Docker CLI

edureka!

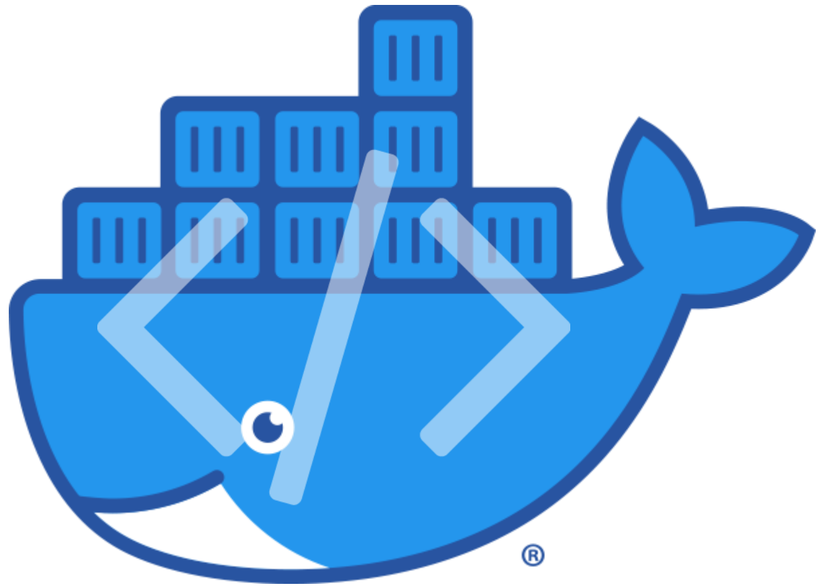
Docker Command Line Interface (CLI)

Docker offers a Command Line Interface (CLI) to manage and interact with containers. The CLI can also be used to manage remote server operations and the Docker Hub repository operations



Docker CLI Configuration

- Docker commands can be prefaced to run with or without `sudo` permissions
- The command line config files are stored in `.docker` directory inside `$HOME` directory
- The default docker command behavior can be altered using the `config.json` file



Common Docker Commands

Command	Description
`docker run`	Creates a container from an image
`docker start`	Starts an already stopped container(s)
`docker stop`	Stops an active container
`docker build`	Builds a docker image from a dockerfile
`docker pull`	Pulls pre-created images from a specified repository
`docker push`	Push images to the specified repository
`docker export`	Exports containers filesystem to a .tar archive file

Common Docker Commands

Command	Description
`docker images`	Lists the docker images currently on the local system
`docker search`	Searches repository for the specified image
`docker ps`	Lists all active containers running on the system
`docker kill`	Kills an active container without any grace period to shut down its processes
`docker commit`	Creates a new image out of an already active container
`docker login`	Command to login to docker hub repository

Note: The container exec command is taught in a separate section

Demo: Docker CLI Commands

edureka!

Port Binding

edureka!

Why we need Port Binding?

- By default docker containers can connect to the public internet without requiring any configurations
- But the public internet does not know how to connect with the containerized service

What is Port Binding?

The process of exposing the required Docker container port and binding it to a port on the system is known as Port Binding. It is also known as port forwarding or port mapping

Port Binding

- The container port can be exposed using the `-p` flag while starting the container
- This exposes the port on the default System IP i.e. `0.0.0.0`

```
docker run -p <system_port>:<container_port> -d <containerName> <imageName>
```

- In order to expose the port on a specific IP, use the following syntax

```
docker run -p <IP>:<system_port>:<container_port> -d <containerName> <imageName>
```

Demo: Port Binding

edureka!

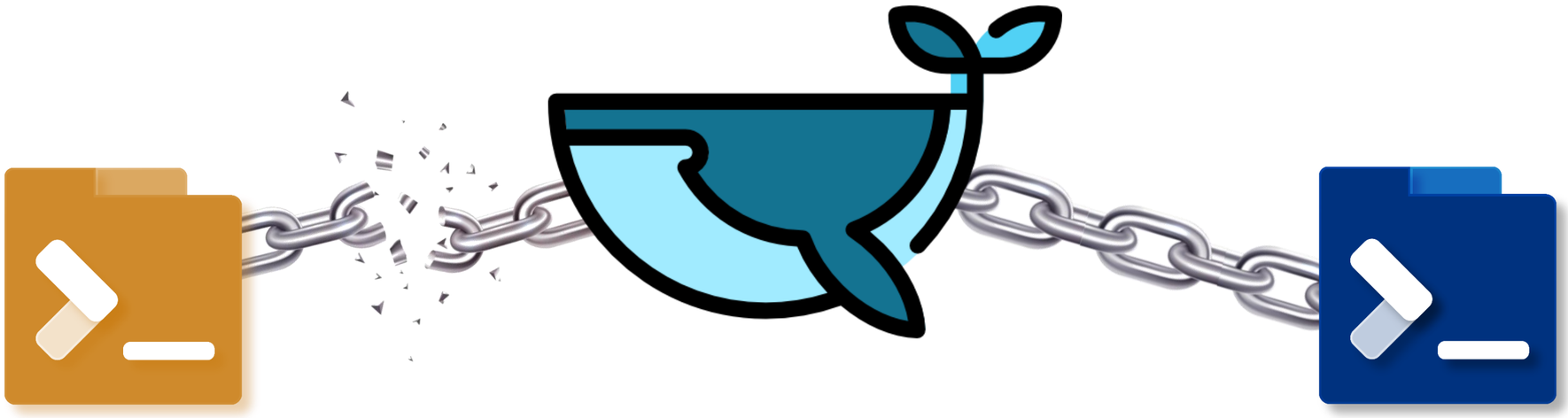
Attached vs Foreground Mode

edureka!

Docker Container Running Mode

Docker provides two modes to run the containers:

- Detached: The container runs in the background
- Foreground or attached: The container attaches itself to the terminal



Detached vs Foreground

Detached

In the detached mode, the container process executes in the background

The container exits when the root process starting the container ends

The '`-d`' flag is used to start the container in detached mode

Example:

```
docker run -d -p 80:80  imageName
```

Foreground

In the attached mode, the container application attaches itself to the console

The process can be connected to any of STDIN, STDOUT, and/or STDERR streams

All containers execute in foreground mode if the '`-d`' flag is not specified

Example:

```
docker run --rm -p 80:80 imageName
```

Detached Mode

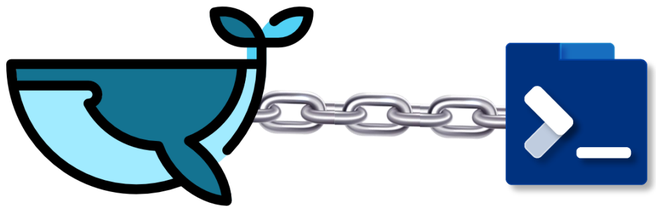
- Using the `--rm` flag with the detached container exits it when the daemon stops
- Input/output actions on detached containers can be done by connecting through network
- A detached container can be reattached to the terminal by using the `attach` command

Foreground Mode

- An attached container is usually started in TTY (TeleType) mode, which is done using the `-t` flag
- By default, docker attaches the container to STDOUT and STDERR streams
- The `-t` flag must not be used when providing input through a pipe '|'

Configurable flags

```
-a  : attach to a 'STDIN', 'STDOUT', and/or 'STDERR'
-t  : allocates a terminal to the container
-i  : keeps the STDIN open, even if not attached
```



Demo: Starting Containers in different Modes

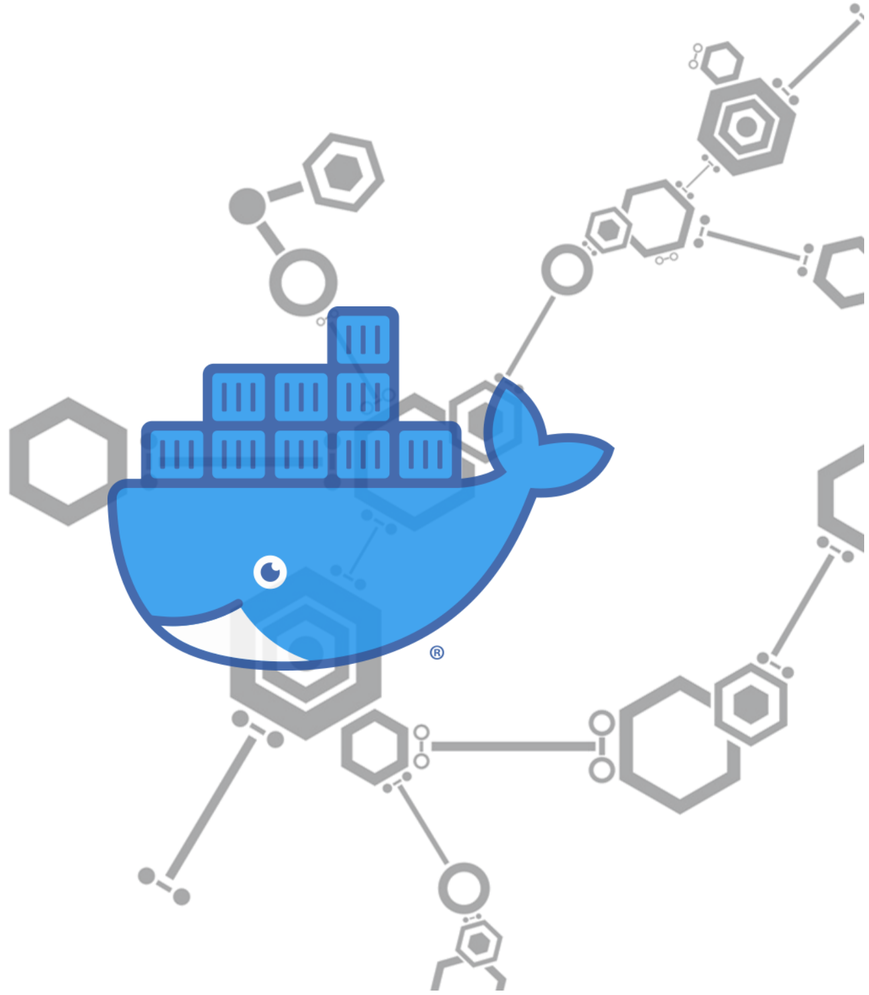
edureka!

Dockerfile Instruction

edureka!

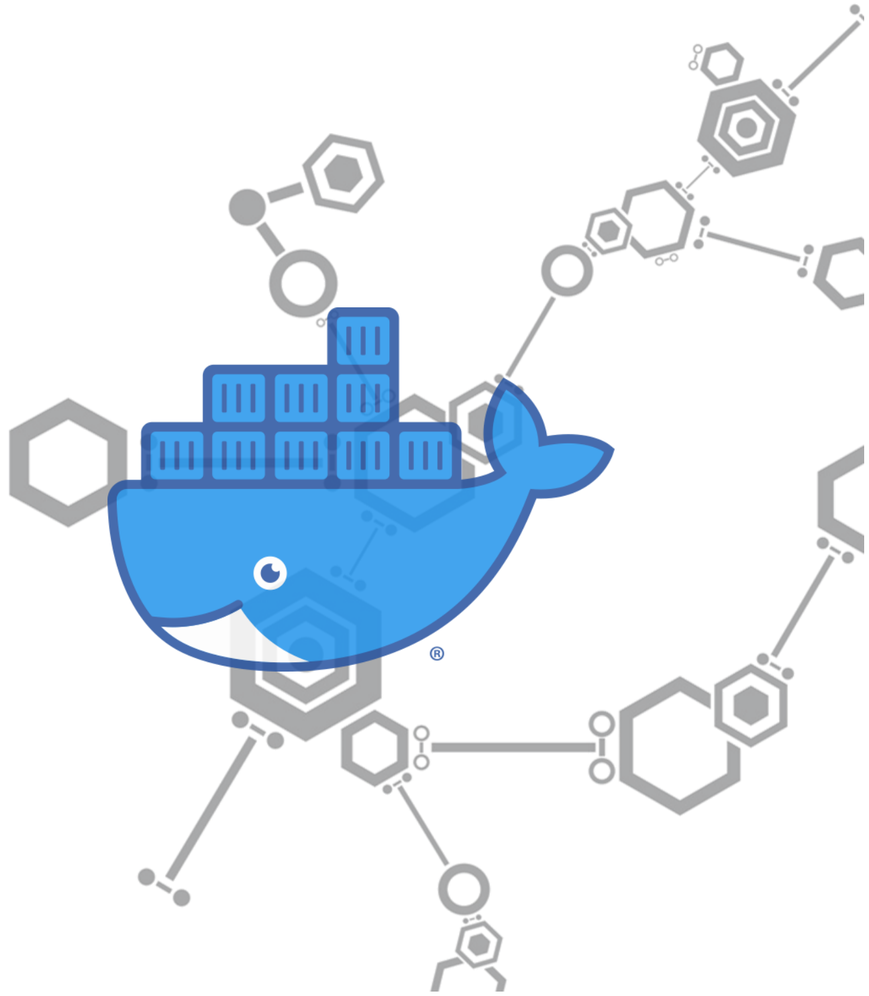
Dockerfile Instruction

- The docker image is made of read-only layers
- Each of the layers corresponds to an instruction in the Dockerfile
- These layers, when stacked together, represent an image
- Executing the image and generating a container adds a writable layer on top



Dockerfile Instruction: Example

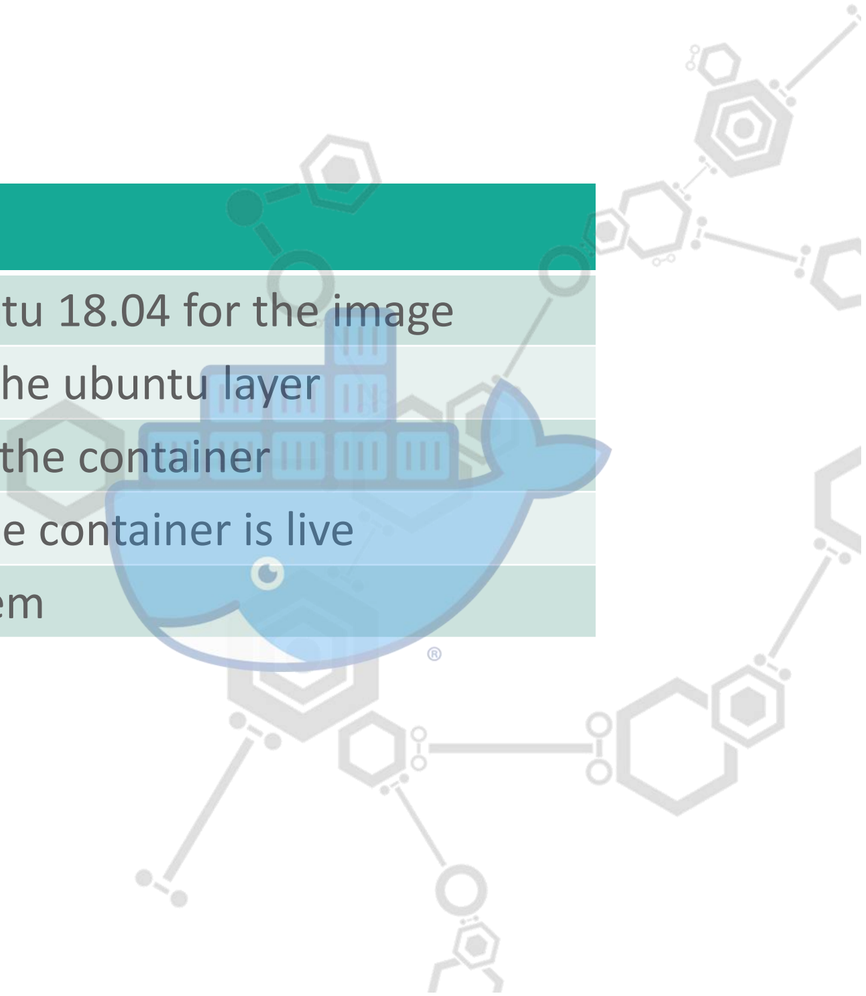
```
FROM ubuntu:18.04
RUN  apt install -y apache2
COPY index.html /var/www/html/
CMD  ["/usr/sbin/httpd", "-D", "FOREGROUND"]
EXPOSE 80
```



edureka!

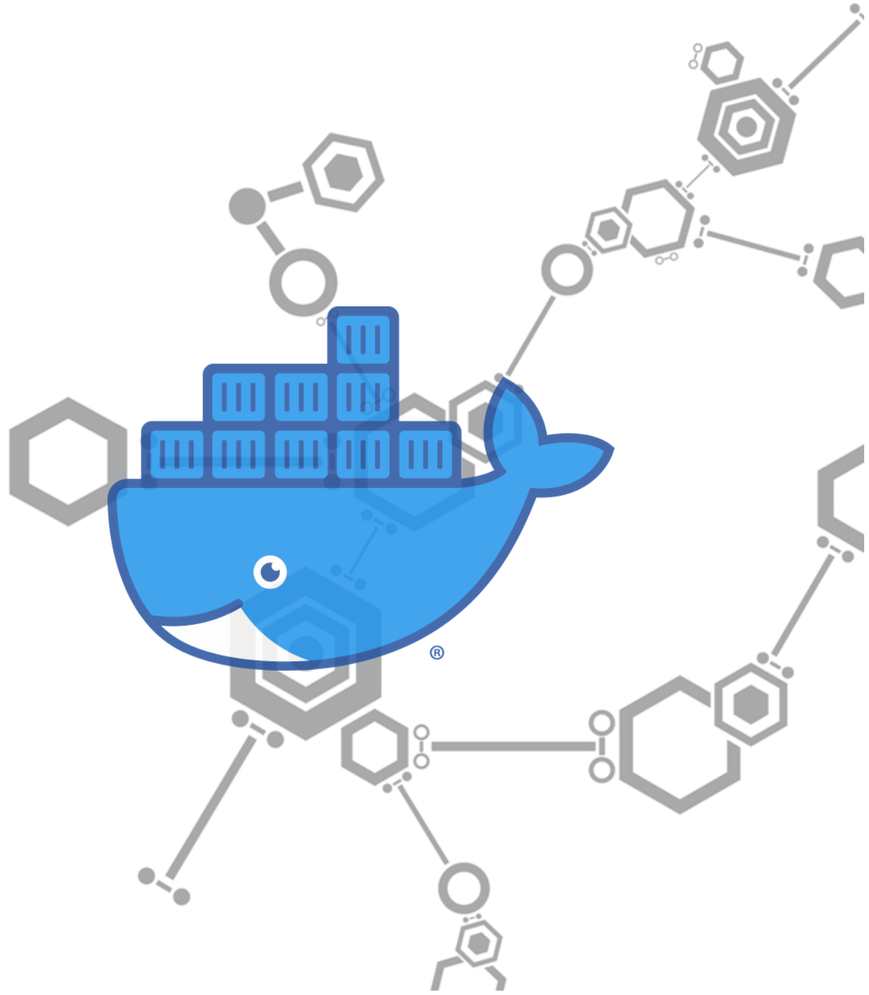
Dockerfile: The Instruction and its Layer

Commands	Description
FROM	Creates the bottom most layer of ubuntu 18.04 for the image
RUN	Installs apache httpd server on top of the ubuntu layer
COPY	Copies files from the local directory to the container
CMD	Specifies the command to run when the container is live
EXPOSE	Exposes the container port to the system



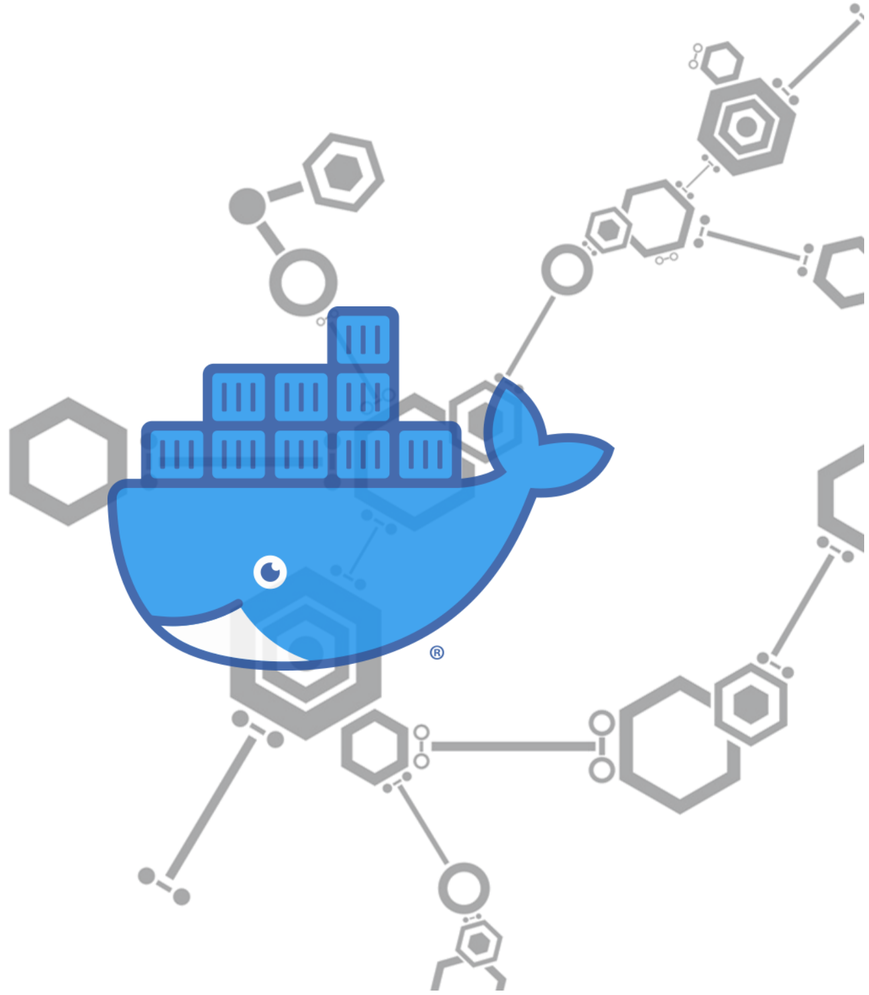
Dockerfile: Build Context

While issuing the build command for a Dockerfile, the current working directory is considered as the build context. The Docker daemon collects all the files from the build context to build the image



Dockerfile: The Build Command

- The build command is handled by the daemon and not the CLI
- To exclude files while executing COPY instruction, use .dockerignore file in the context directory
- Use the `-t` flag to specify the repository and tag the image
- The `-t` flag can also be used to store the image in multiple repositories

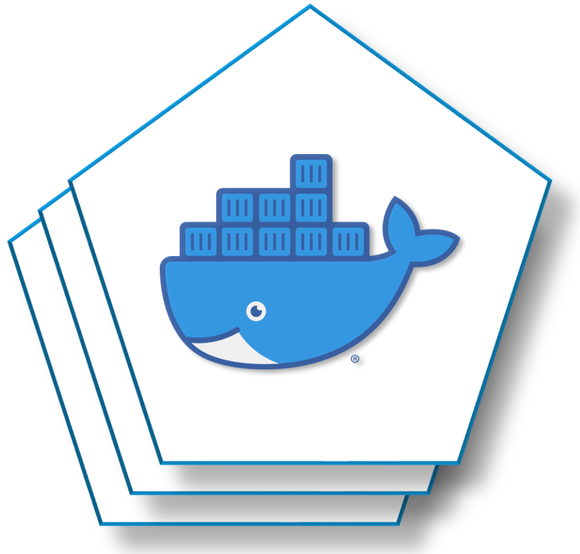


Docker Image

edureka!

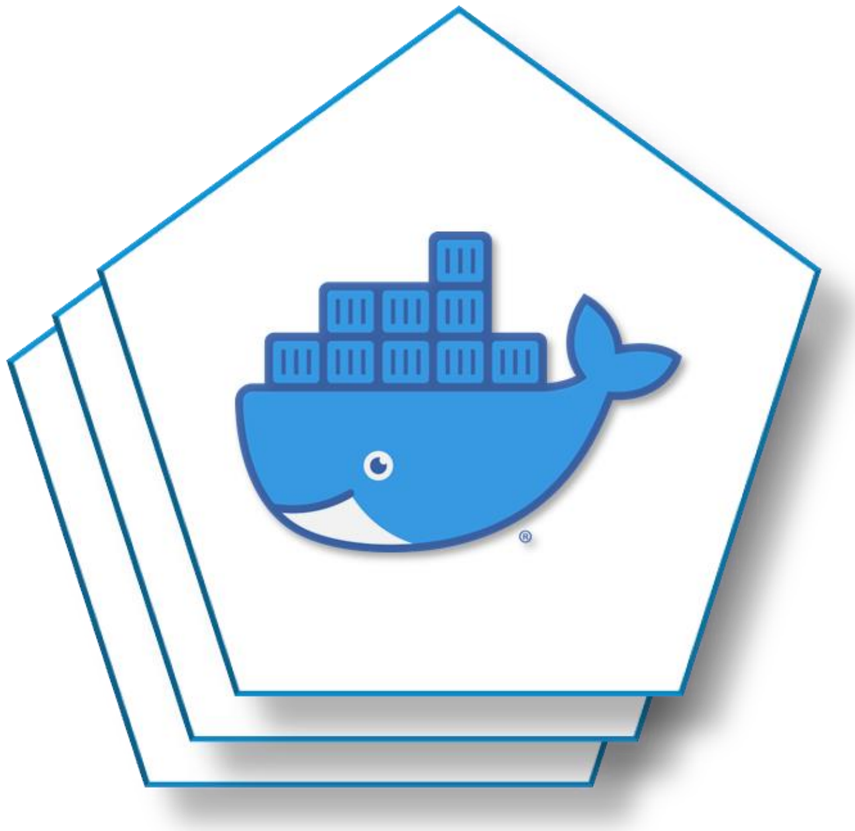
Docker Image

- Docker image is a file, consisting of multiple layers, that is executed to create a container
- The image is built from the instructions in the Dockerfile
- Containers use images to construct a run-time environment



Docker Base Image

- Docker base image is the basic image on top of which layers are added
- Docker tracks changes by adding a new image layer over the base image by using the Union File System (UFS)
- Example: To run the LAMP Stack, the user will require a base image of Linux OS and then subsequent layers of Apache, MySQL, and PHP are added on top



Building a Docker Image: Creating Dockerfile

```
root@test01:~# touch Dockerfile
root@test01:~# gedit Dockerfile
```

Dockerfile

```
1  FROM ubuntu
2  CMD ["echo", "hello world"]
```

FROM specifies the base image. In this scenario, it is the ubuntu image on the Docker hub repository

The layer above the base image. Here it executes the given command

Building a Docker Image: Building the Dockerfile

```
root@test01:~# gedit Dockerfile
root@test01:~# docker build .
Sending build context to Docker daemon   513 kB
Step 1 : FROM ubuntu
 ---> 4ca3a192ff2a
Step 2 : CMD echo hello world
 ---> Running in 876177664b4f
 ---> 7c226dc91bb2
Removing intermediate container 876177664b4f
Successfully built 7c226dc91bb2
root@test01:~#
```

Running hello-world in intermediate container

Final image ID

```
root@test01:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              7c226dc91bb2        8 minutes ago       128.2 MB
hello-world         latest              48b5124b2768        3 months ago        1.84 kB
ubuntu              latest              4ca3a192ff2a        4 months ago        128.2 MB
root@test01:~#
```

edureka!

Building a Docker Image: Running the File

```
root@test01:~# docker run --name test 7c226dc91bb2
hello world
root@test01:~#
```

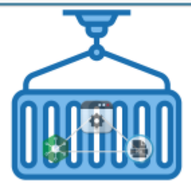
Demo: Writing a Dockerfile to Create an Image

edureka!

Summary

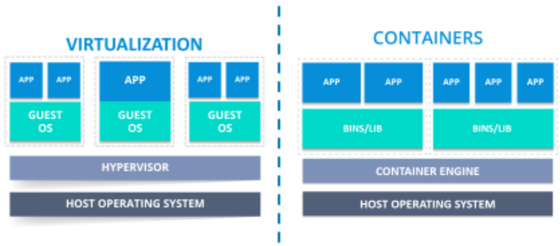
What is Containerization?

It is the process of packaging the application and all its dependencies to execute them in an efficient and hassle-free way across different environments. A single unit of this bundled package is known as a **Container**



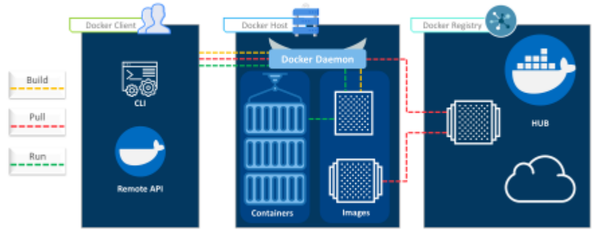
edureka!

Virtual Machine vs Container



edureka!

Docker Architecture



edureka!

Docker Command Line Interface (CLI)

Docker offers a Command Line Interface (CLI) to manage and interact with containers. The CLI can also be used to manage remote server operations and the Docker Hub repository operations

edureka!

Dockerfile Instruction

- The docker image is made of read-only layers
- Each of the layers corresponds to an instruction in the Dockerfile
- These layers, when stacked together, represent an image
- Executing the image and generating a container adds a writable layer on top

edureka!

Docker Image

- Docker image is a file, consisting of multiple layers, that is executed to create a container
- The image is built from the instructions in the Dockerfile
- Containers use images to construct a run-time environment

edureka!

edureka!

edureka!

Thank You

For more information please visit our website
www.edureka.co