



BITS Pilani

Cloud Computing

SEWP ZG527

Agenda



- ❖ Virtualization Recap
- ❖ Containers
 - ❖ Introduction & Motivation for Containers
 - ❖ Linux Container –LXC and LXD
 - ❖ Container Architecture
 - ❖ Orchestration technologies
 - ❖ Docker Container and Components
 - ❖ Hands on with Docker



BITS Pilani

Pilani | Dubai | Goa | Hyderabad



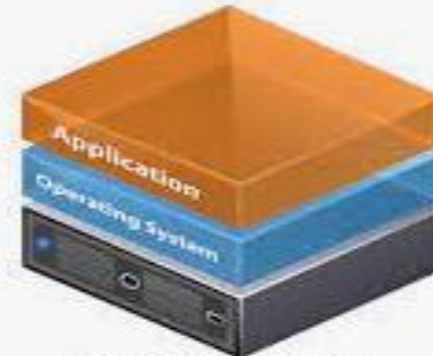
Recap



Virtualization

What Is Virtualization ?

- Virtualization is a Technology that transforms hardware into software.
- Virtualization allows to run multiple operating systems as virtual machines.
 - Each copy of an operating system is installed in to a virtual machine.

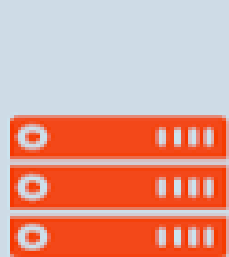


Traditional Architecture

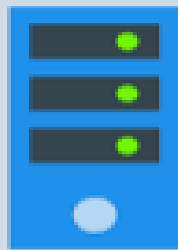


Virtual Architecture

Virtualization



physical Hardware

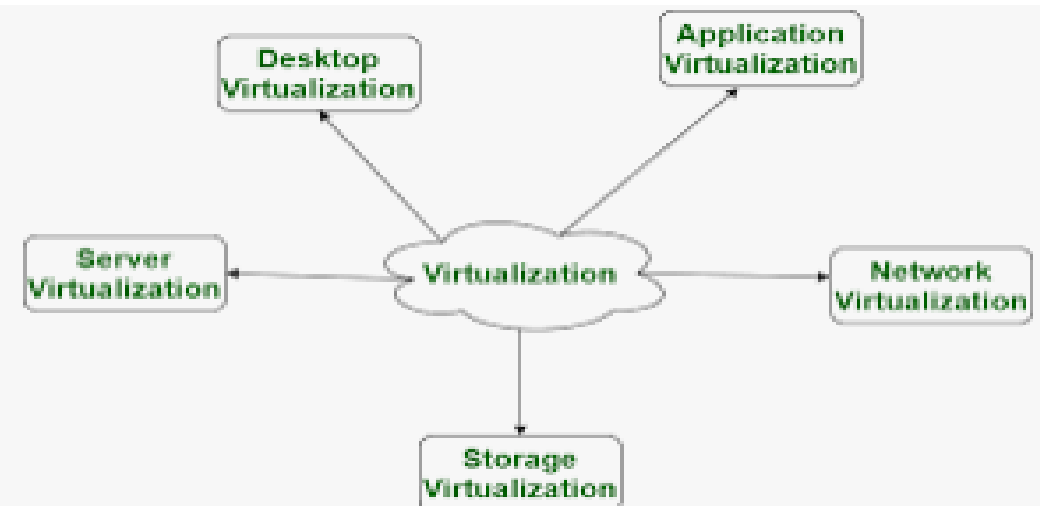


Hypervisor



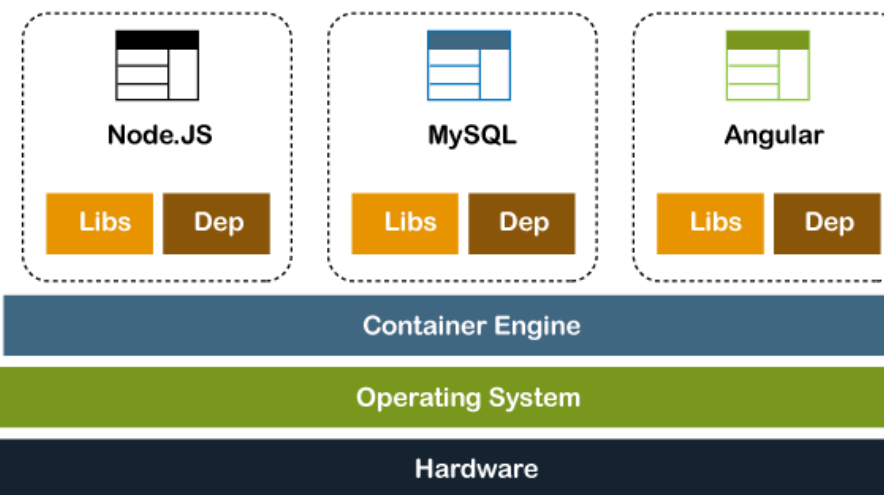
VM

VM

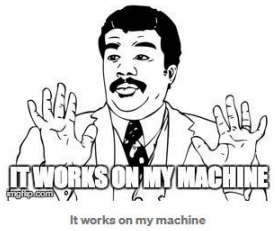




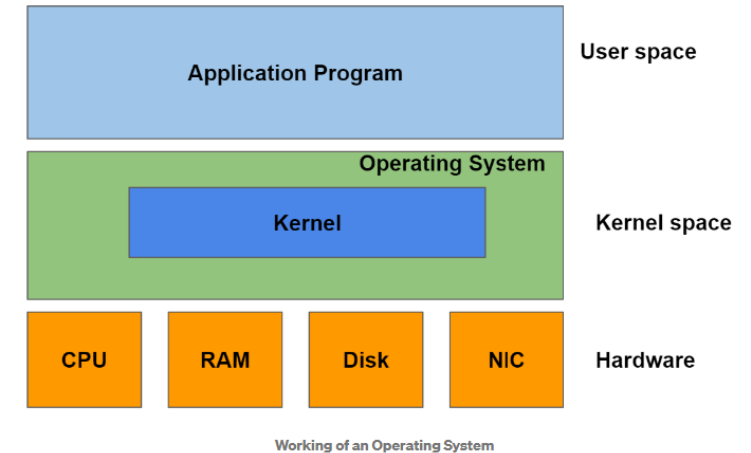
Containers



Motivation towards Containers



- Let's assume that you are building a web service on an Ubuntu machine. Your code works fine on your local machine. You have a remote server in your data centre that can run your application.
- You copy your local binaries on the remote server and try to run your code. The next thing that you see is your code doesn't work there. The above problems result in portability issues. The developer has to spend a lot of time debugging the environment-specific issues.



- The computer has different hardware resources such as RAM, hard disk, Network Interface Card, IO devices, etc. The operating system is the software that manages this hardware.
- OS consists of a system program known as the kernel, which is loaded in the memory when OS starts. The kernel is responsible for process management, CPU scheduling, file system & IO.
- User programs interact with the hardware through the means of the kernel. For eg:- Let's say your application wants to open a file and write content in it. The application will invoke system calls like *fopen()* and *fwrite()* to perform its functions.
- The kernel performs the function on behalf of the user program and gives the output back to it. The following diagram shows the different layers involved in the functioning of an application program.

What are Containers?

A **container** is a standard unit of **software** that packages up **code** and all its **dependencies** so the **application** **runs** quickly and **reliably** from **one computing** environment to **another**. The way which containerized applications operate is shown →

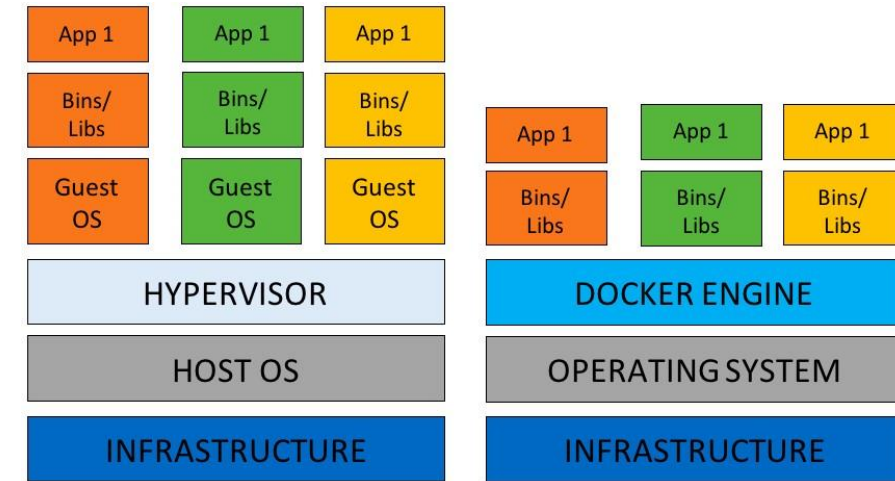
Containers are an **operating system virtualization** technology used to package **applications** and their **dependencies** and run them **in isolated environments**.

They provide a lightweight method of packaging and deploying applications in a standardized way across many **different types of infrastructure**

Containers **run consistently** on any **container-capable host**, so developers can test the same software locally that they will later deploy to full production environments.

The container format also ensures that the application dependencies are baked into the image itself, simplifying the hand off and release processes.

Because the hosts and platforms that run containers are generic, infrastructure management for container-based systems can be standardized.



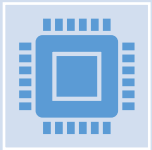
As can be seen from this diagram, a container includes an application plus any binaries or libraries that the application requires in order for it to run.

The container runs under the control of the container engine (such as Docker or CRI-O), which in turn runs on top of the operating system (which can be Windows 10, Windows Server 2016, or Linux depending on the container engine being used).

Linux Containers



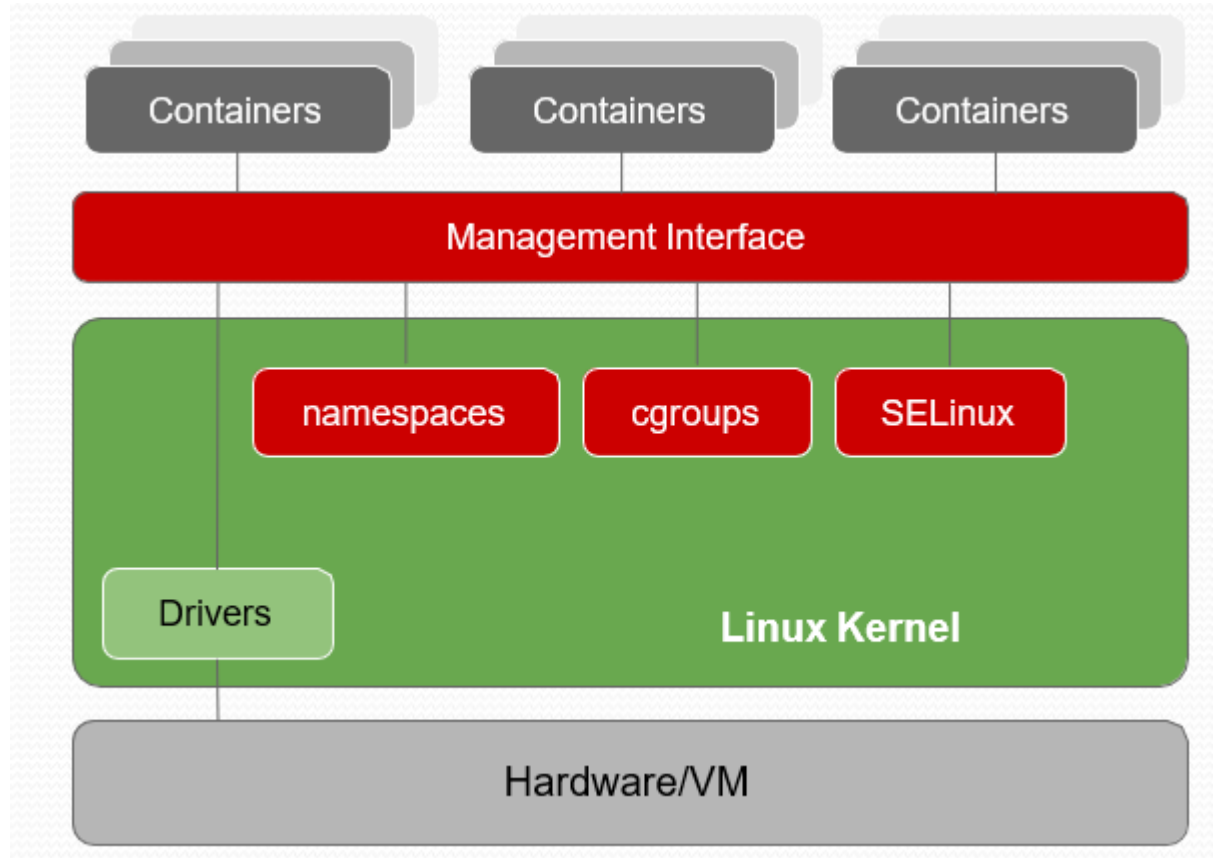
Linux Containers(LXC) allow running multiple isolated Linux instances (containers) on the same host.



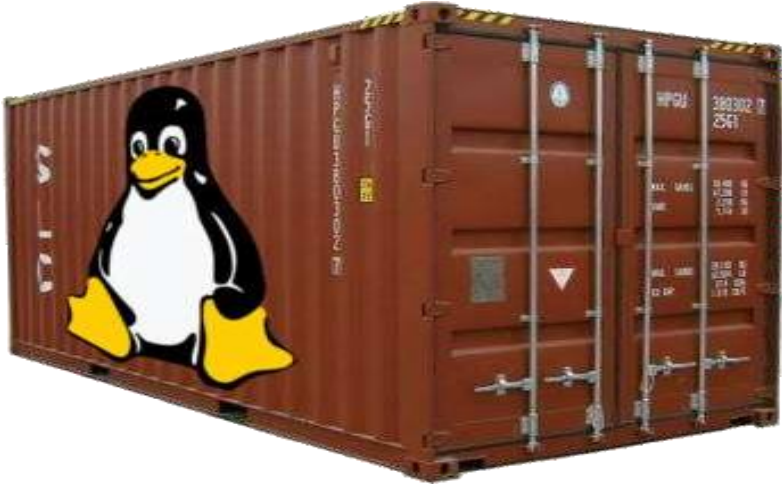
Containers share the same kernel with anything else that is running on it, but can be constrained to only use a defined amount of resources such as CPU, memory or I/O.



A container is a way to isolate a group of processes from the others on a running Linux system.



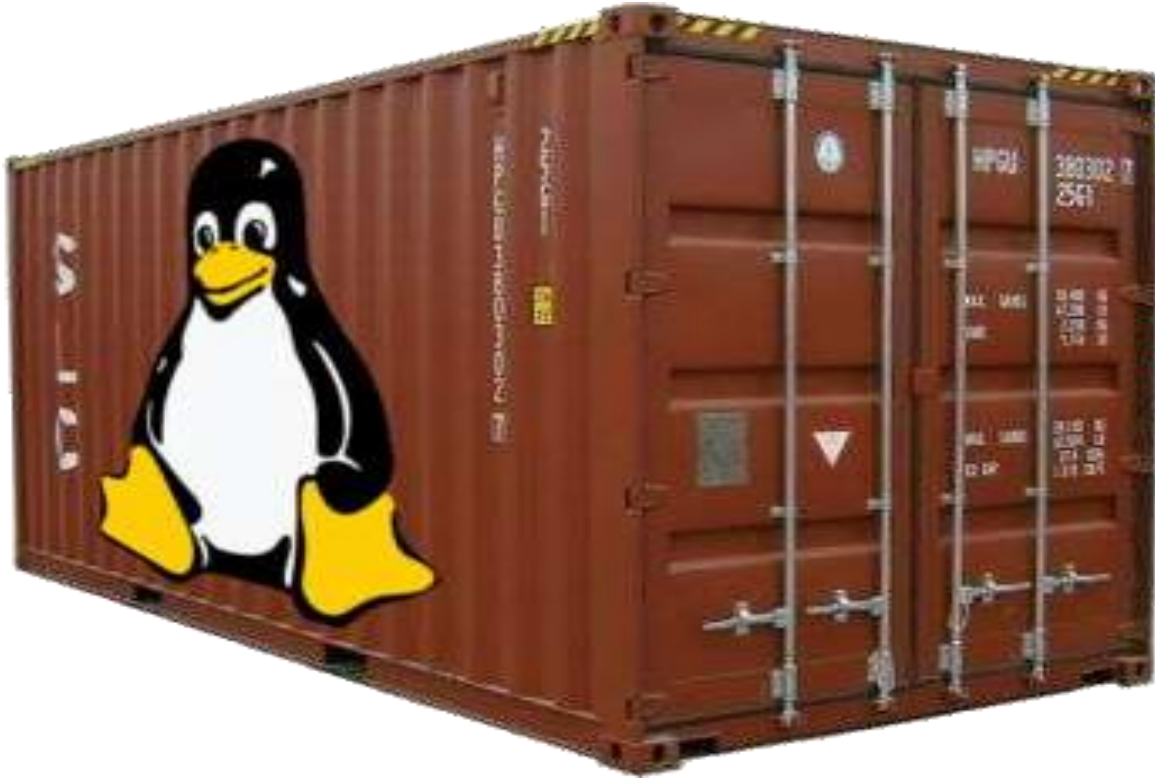
Linux Containers LXC



- LXC is an abbreviation used for Linux Containers which is an operating system that is used for running multiple Linux systems virtually on a controlled host via a single Linux kernel.
- LXC bundles with the kernel's Cgroups to provide the functionality for the process and network space instead of creating a full virtual machine and provides an isolated environment for the applications.

- Features provided by LXC :
 - It provides Kernel namespaces such as IPC, mount, PID, network, and user.
 - It provides Kernel capabilities.
 - Control groups (Cgroups).
 - Seccomp profiles

Linux Containers LXC



- Lightweight virtualization.
- OS-level virtualization
- Allow single host to operate multiple **isolated & resource-controlled** Linux Instances.
- included in the Linux kernel called LXC (Linux Container)

Containers are not a new technology: the earliest iterations of containers have been around in open source Linux code for decades.

Control Groups

Control Groups or **cgroups** was introduced by Google in 2006 which was based on Linux Kernel Feature.

By using cgroups, system administrators gain fine-grained control over allocating, prioritizing, denying, managing, and monitoring system resources.

All processes on a Linux system are child processes of a common parent: the **init process**, which is executed by the kernel at boot time and starts other processes (which may in turn start child processes of their own). Because all processes descend from a single parent, the Linux process model is a single hierarchy, or tree.

Cgroups are similar to processes in that:

- **they are hierarchical**, and
- child cgroups **inherit certain attributes** from their parent cgroup.

The fundamental difference is that many different hierarchies of cgroups can exist simultaneously on a system. If the Linux process model is a single tree of processes, then the cgroup model is one or more separate, unconnected trees of tasks (i.e. processes).

Multiple separate hierarchies of cgroups are necessary because each **hierarchy is attached to *one or more subsystems***. A subsystem^[2] represents a single resource, such as CPU time or memory. Red Hat Enterprise Linux 6 provides ten cgroup subsystems, listed below by name and function.

Control Groups

- `blkio` — this subsystem sets limits on input/output access to and from block devices such as physical drives (disk, solid state, or USB).
- `cpu` — this subsystem uses the scheduler to provide cgroup tasks access to the CPU.
- `cpuacct` — this subsystem generates automatic reports on CPU resources used by tasks in a cgroup.
- `cpuset` — this subsystem assigns individual CPUs (on a multicore system) and memory nodes to tasks in a cgroup.
- `devices` — this subsystem allows or denies access to devices by tasks in a cgroup.
- `freezer` — this subsystem suspends or resumes tasks in a cgroup.
- `memory` — this subsystem sets limits on memory use by tasks in a cgroup and generates automatic reports on memory resources used by those tasks.
- `net_cls` — this subsystem tags network packets with a class identifier (classid) that allows the Linux traffic controller (`tc`) to identify packets originating from a particular cgroup task.
- `net_prio` — this subsystem provides a way to dynamically set the priority of network traffic per network interface.
- `ns` — the *namespace* subsystem.
- `perf_event` — this subsystem identifies cgroup membership of tasks and can be used for performance analysis.

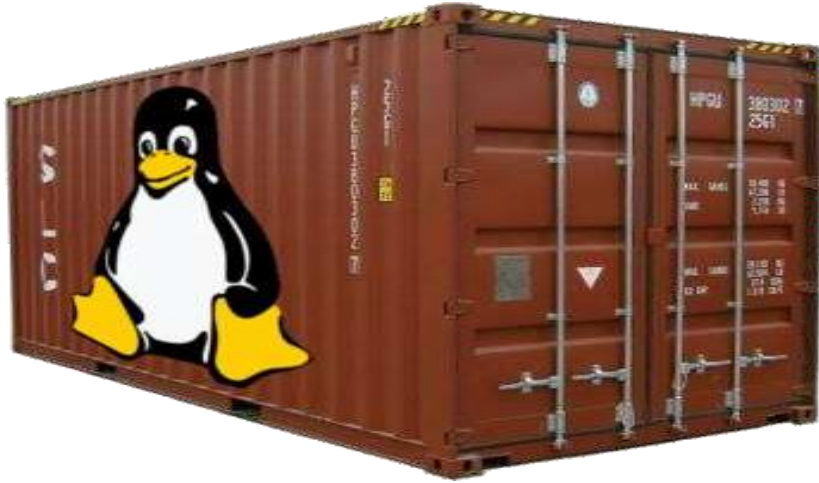
cgroup: Control Groups provide a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behaviour.

• **namespace**: wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource.

In short:

- Cgroups = limits how much you can use;
- namespaces = limits what you can see (and therefore use)

Linux Containers LXD



- The simplest way to define LXD is to say it's an extension of LXC. LXD also happens to be LXC's main claim to fame, now that LXC has ceased to be important for Docker and CoreOS.
- The more technical way to define LXD is to describe it as a REST API that connects to libxlc, the LXC software library. LXD, which is written in Go, creates a system daemon that apps can access locally using a Unix socket, or over the network via HTTPS.

A host **can run many LXC containers using only a single system daemon**, which simplifies management and reduces overhead. With pure-play LXC, you'd need separate processes for each container.

The LXD daemon can take advantage of host-level security features to make containers more secure. On plain LXC, container security is more problematic.

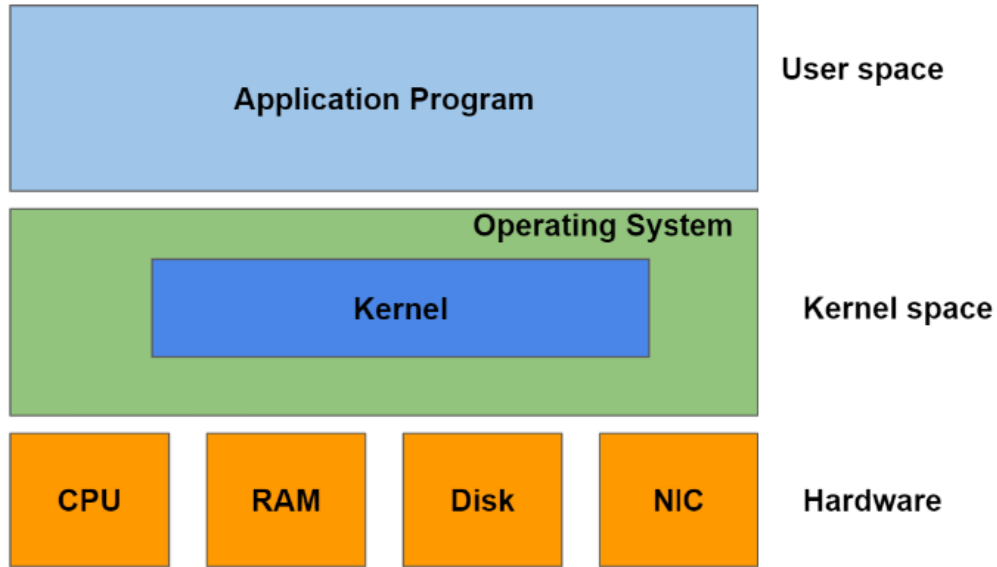
Because the LXD daemon handles networking and data storage, and users can control these things from the LXD CLI interface, it simplifies the process of sharing these resources with containers.

LXD offers advanced features not available from LXC, including live container migration and the ability to snapshot a running container.

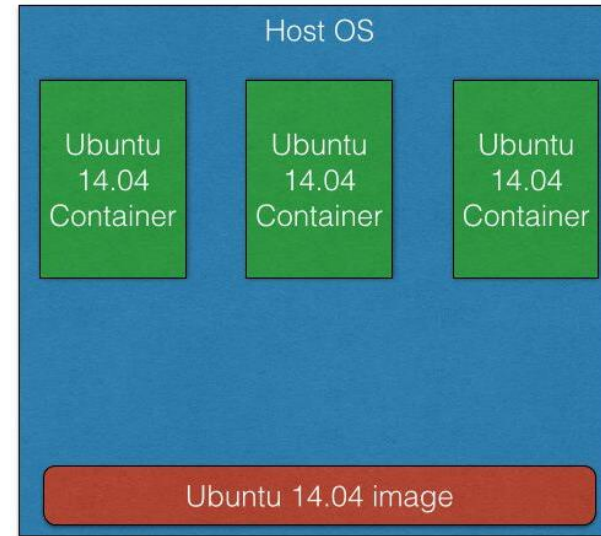
Comparing LXC & LXD

LXC is a virtual environment creation tool, it was built by Google, IBM etc.	LXD is an add on for the LXC to provide advanced features and functionalities.
Multiple processes are needed for multiple containers and hence it is not flexible.	LXD makes it flexible by providing a single process for multiple containers.
Snapshots, Live Migration etc are some of the features which are not supported by LXC.	LXD supports snapshots and lives migration features.
Scalability functionality is not provided by LXC and hence users shift to other virtual solutions.	With the use of LXD, scalability is achieved in LXC.
Management capabilities are poor, especially in the case of network and storage.	It has better management capabilities like storage pooling.
It is not user friendly and needs the expertise to handle the processes.	It provides a user-friendly interface.
After data processing, the data cannot be retrieved.	Data retrieval functionality after data processing is provided in LXD.
C API is used by the LXC.	LXD uses REST API.

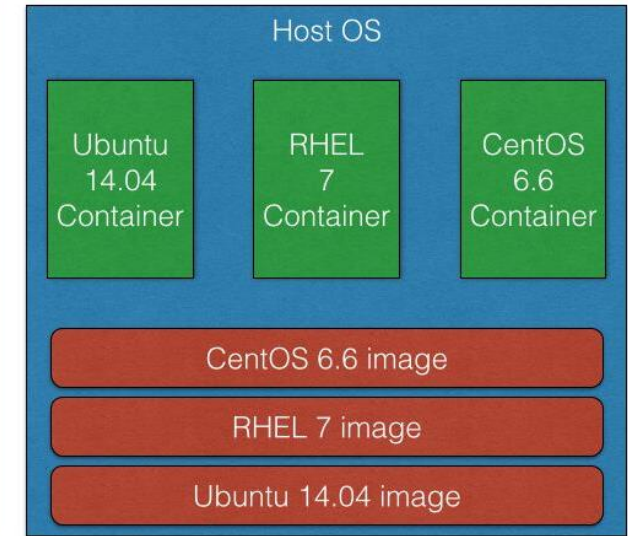
Container Types – OS Containers



Working of an Operating System



Identical OS containers



Different flavoured OS containers

- **OS containers** are virtual environments that **share the kernel** of the **host operating system** but provide **user space isolation**.
- For all practical purposes, you can think of **OS containers as VMs**. You can install, configure and run different applications, libraries, etc., just as you would on any OS.
- Just as a VM, **anything running inside a container** can only see **resources that have been assigned to that container**.
- OS containers are useful when you want to run a fleet of identical or different flavors of distros.
- Most of the times containers are created from templates or images that determine the structure and contents of the container.
- It thus allows you to create containers that have identical environments with the same package versions and configurations across all containers.

Minimalistic OS

A common set of ideas:

A minimal operating system is one that has been stripped of all unnecessary components and provides only the functionality needed for a specific purpose.

- Stability is enhanced through transactional upgrade/rollback semantics.
- Traditional package managers are absent and may be replaced by new packaging systems (Snappy), or custom image builds (Atomic).
- Security is enhanced through various isolation mechanisms

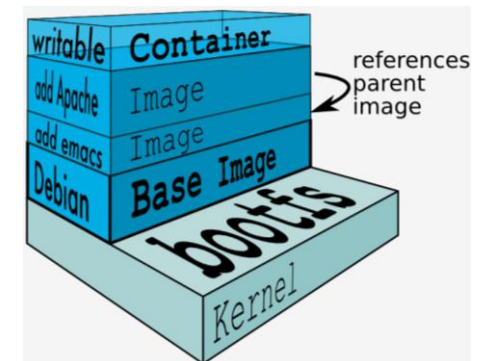
	CoreOS (647.0.0)	RancherOS (0.23.0)	Atomic (F 22)	Photon	Snappy (edge – 145)
Size	164MB	20MB	151/333MB	251MB	111MB
Kernel version	3.19.3	3.19.2	4.0.0	3.19.2	3.18.0
Docker version	1.5.0	1.6.0	1.6.0	1.5.0	1.5.0
Init system	systemd	Docker	systemd	systemd	systemd
Package manager	None (Docker/Rocket)	None (Docker)	Atomic	tdnf (tyum)	Snappy
Filesystem	ext4	ext4	xfs	ext4	ext4
Tools	Fleet, etcd	–	Cockpit (Anaconda, kickstart), atomic	–	

Container Types – Application Containers

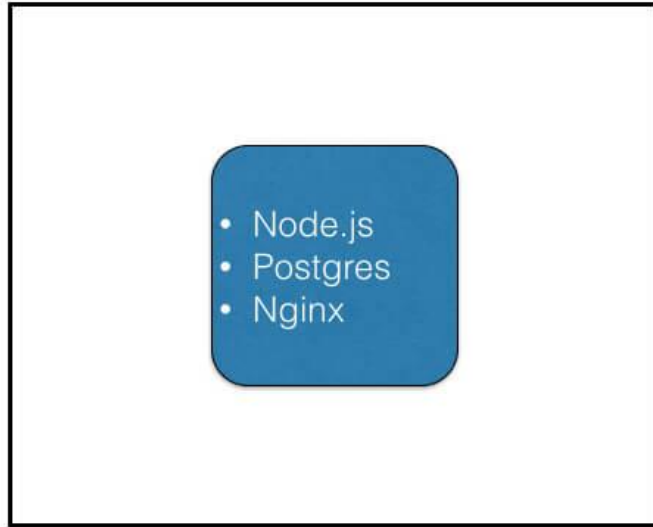
APPLICATION CONTAINERS

- Designed to package and run a single service
- All containers share host kernel
- Subtle differences from operating system containers
- Examples: Docker, Rocket
- Docker: runs a single process on creation
- OS containers: run many OS services, for an entire OS
- Create application containers for each component of an app
- Supports a micro-services architecture
- DevOPS: developers can package their own components in application containers
- Supports horizontal and vertical scaling

- While OS containers are designed to run multiple processes and services, application containers are designed to package and run a single service.
- Container technologies like Docker and Rocket are examples of application containers.
- So even though they share the same kernel of the host there are subtle differences make them different, which I would like to talk about using the example of a Docker container:

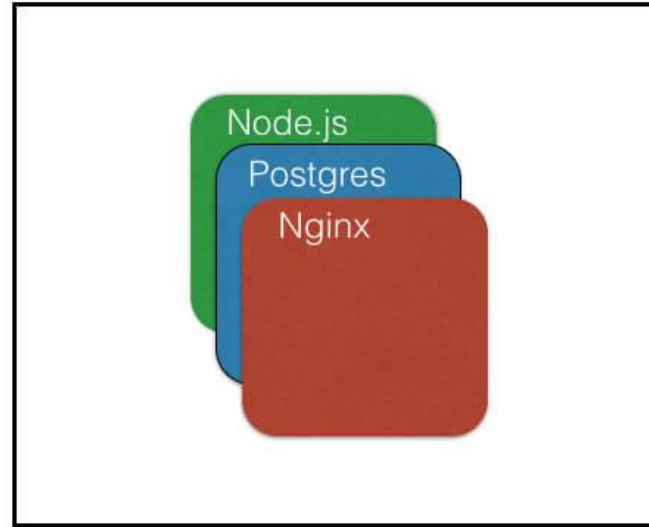


Comparison OS vs Application Containers



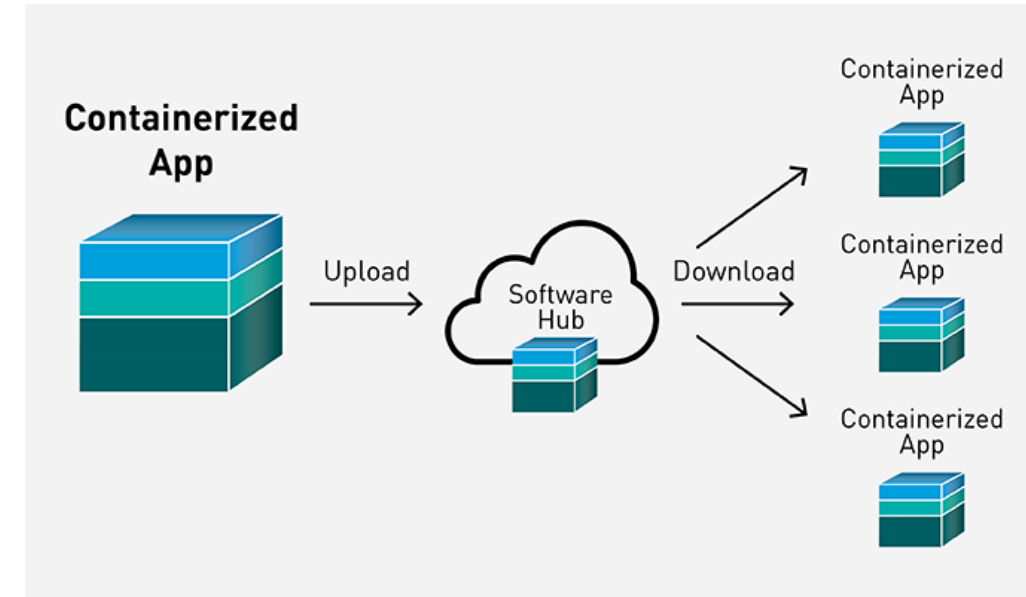
OS containers

- Meant to be 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

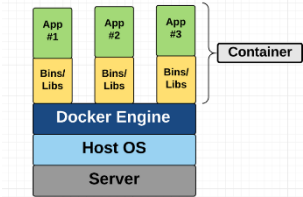
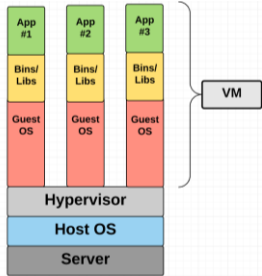


App containers

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



Difference between VM & Container



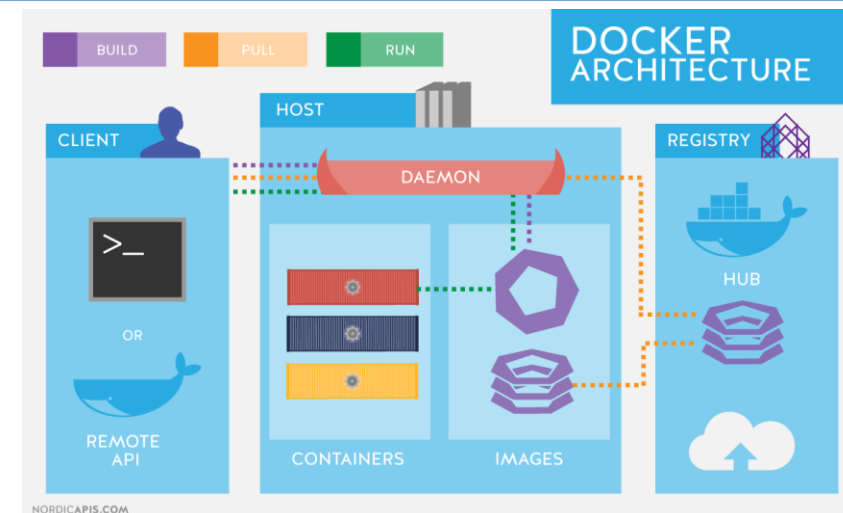
VM	Container
Virtual machines , or VMs, are a hardware virtualization technology that allows you to fully virtualize the hardware and resources of a computer .	Containers take a different approach. Rather than virtualizing the entire computer, containers virtualize the operating system directly .
A separate guest operating system manages the virtual machine , separate from the OS running on the host system.	They run as specialized processes managed by the host operating system's kernel , but with a constrained and heavily manipulated view of the system's processes, resources, and environment.
On the host system , a piece of software called a hypervisor is responsible for starting, stopping, and managing the virtual machines.	Containers are unaware that they exist on a shared system and operate as if they were in full control of the computer .
Because VMs are operated as completely distinct computers that, under normal operating conditions, cannot affect the host system or other VMs, virtual machines offer great isolation and security.	it is more common to manage containers more similarly to applications.
In general, virtual machines let you subdivide a machine's resources into smaller, individual computers, but the result doesn't differ significantly from managing a fleet of physical computers.	containers occupy a space that sits somewhere in between the strong isolation of virtual machines and the native management of conventional processes.



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

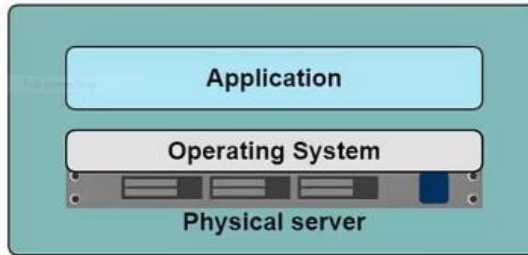
Dockers



Dockers - Motivation

Problems in the Past

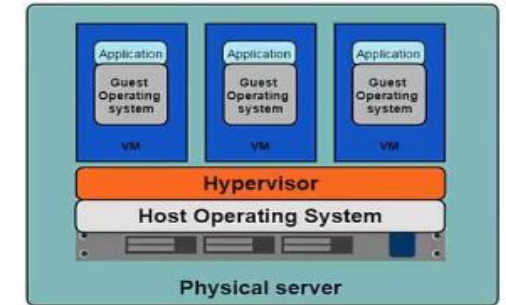
- Slow deployment times
- Huge costs
- Wasted resources
- Difficult to scale
- Difficult to migrate
- Vendor lock in



Virtualization

Hypervisor-based Virtualization

- One physical server can contain multiple applications
- Each application runs in a virtual machine (VM)



Limitations of VMs

- Each VM still requires
 - CPU allocation
 - Storage
 - RAM
 - An entire guest operating system
- The more VM's you run, the more resources you need
- Guest OS means wasted resources
- Application portability not guaranteed



Containers



Dockers - Introduction

What Is Docker?



- Lightweight, open, secure platform
- Simplify building, shipping, running apps
- Runs natively on Linux or Windows Server
- Runs on Windows or Mac Development machines (with a virtual machine)
- Relies on "images" and "containers"



- Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. At a high level, Docker is a utility that can efficiently create, ship, and run containers.
- Docker containers wrap a piece of software in a complete file system that contains everything needed to run: **code, runtime, system tools, system libraries**
- Docker enables you to quickly, reliably, and consistently deploy applications regardless of environment.
- Docker enables you to **achieve compute density** by running several **isolated containers** on the **same hardware**.
- Namespaces provides the isolation

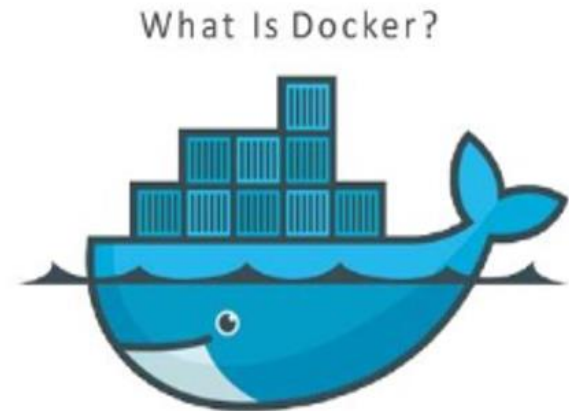
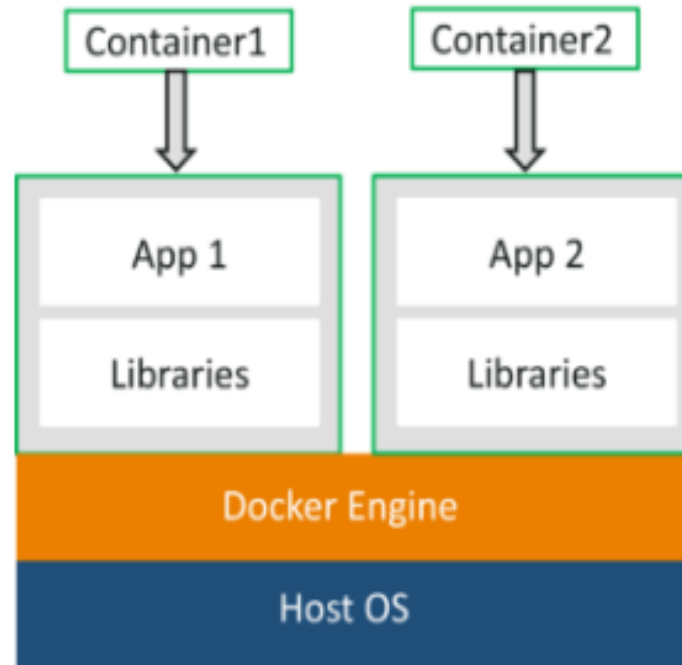
Dockers - Introduction

What is Docker?

Docker is a platform which packages an application and all its dependencies together in the form of containers. This containerization aspect ensures that the application works in any environment.

As you can see in the diagram, each and every application runs on separate containers and has its own set of dependencies & libraries. This makes sure that each application is independent of other applications, giving developers surety that they can build applications that will not interfere with one another.

So a developer can build a container having different applications installed on it and give it to the QA team. Then the QA team would only need to run the container to replicate the developer's environment.



Dockers - Architecture

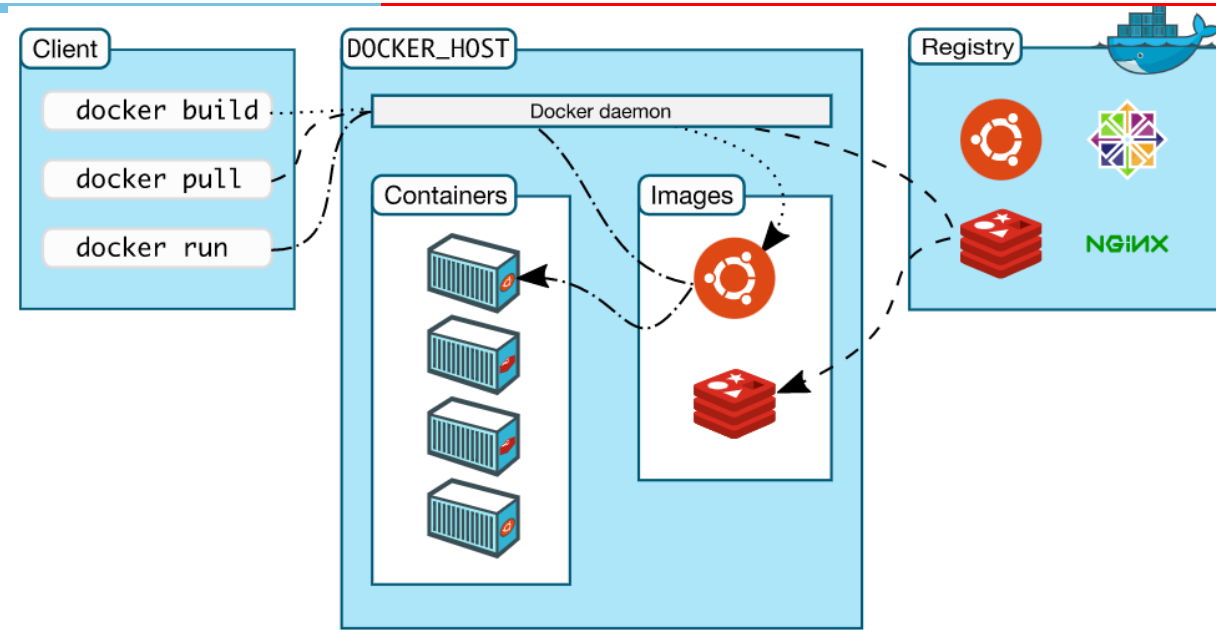
Docker uses a client-server architecture.

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.

Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.



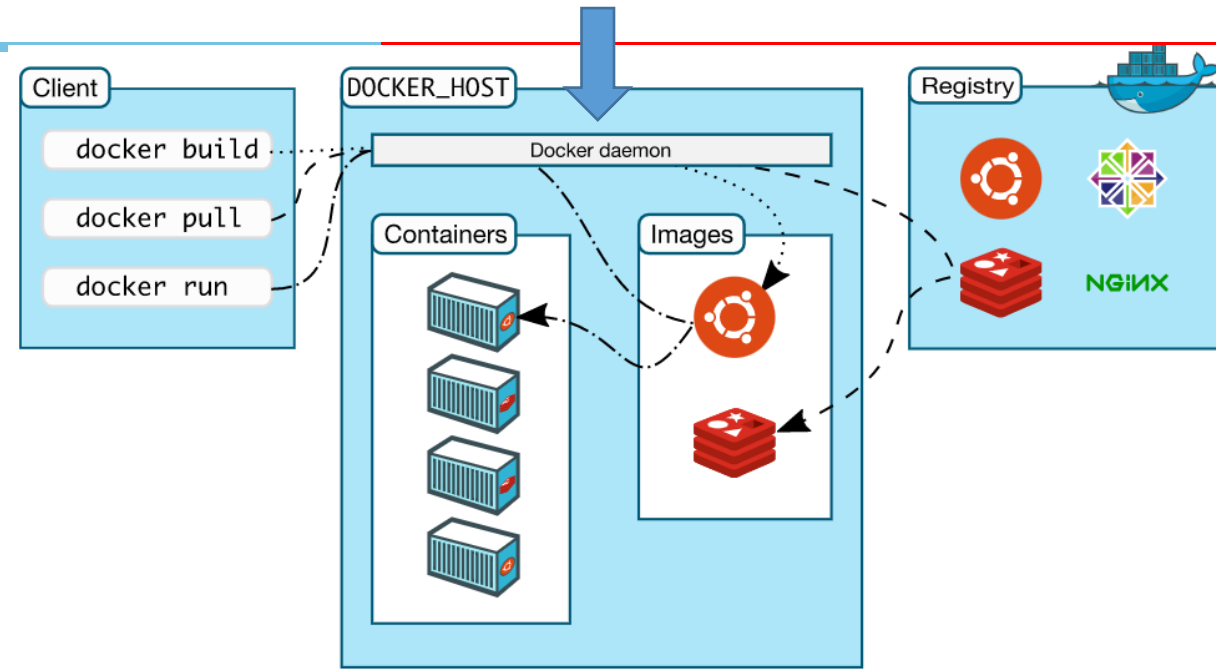
Dockers – Components - Daemon

The Docker daemon, also known as 'dockerd', consistently listens to the requests put forward by the Docker API.

It is used to carry out all the heavy tasks such as creating and managing Docker objects including containers, volumes, images, and networks.

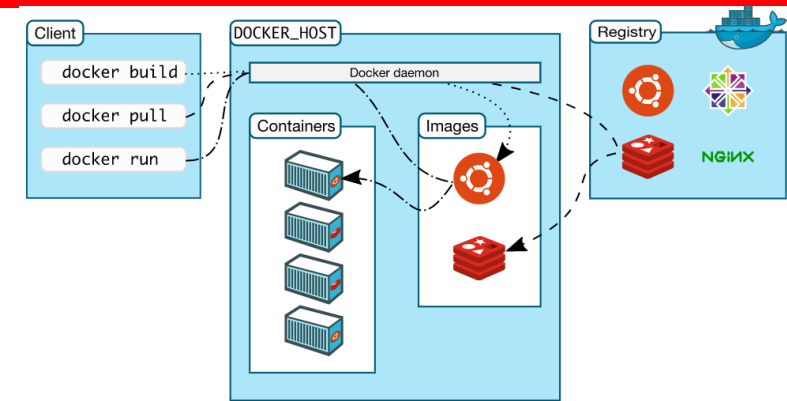
A Docker daemon is also capable of communicating with other daemons in the same or different host machines.

For example, in the case of a swarm cluster, the host machine's daemon can communicate with daemons on other nodes to carry out tasks.



Dockers – Components – Client

The **Docker client** (docker) is the primary way that many Docker users interact with Docker. The Docker users can leverage simple HTTP clients like Command line to interact with Docker. When a user executes a Docker command such as Docker run, the CLI will send this request to the dockerd via the REST API. The Docker CLI can also communicate with over one daemon.



Docker Desktop

Docker Desktop is an easy-to-install application for your Mac or Windows environment that enables you to build and share containerized applications and microservices. Docker Desktop includes the Docker daemon ([dockerd](#)), the Docker client ([docker](#)), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper. For more information, see [Docker Desktop](#).

Docker registries

A Docker *registry* stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

When you use the [docker pull](#) or [docker run](#) commands, the required images are pulled from your configured registry. When you use the [docker push](#) command, your image is pushed to your configured registry.

Dockers – Components – Registries

Docker registries

A Docker *registry* stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

When you use the `docker pull` or `docker run` commands, the required images are pulled from your configured registry. When you use the `docker push` command, your image is pushed to your configured registry.

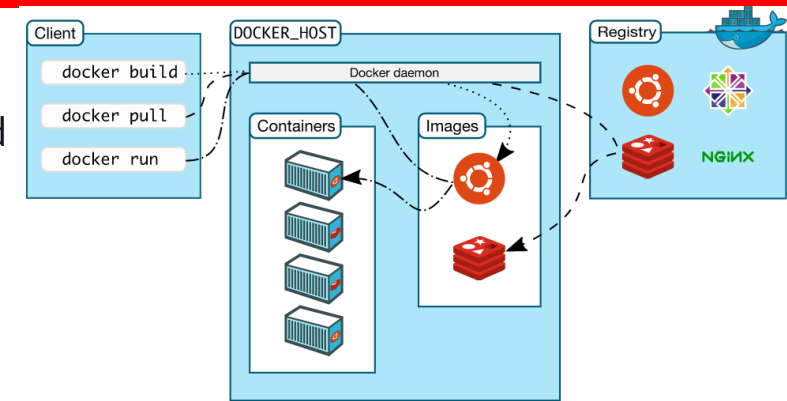
The official [Docker registry](https://docs.docker.com/docker-hub/) called Dockerhub contains several official image repositories.

A repository contains a set of similar Docker images that are uniquely identified by Docker tags.

Dockerhub provides tons of useful official and vendor-specific images to its users.

Some of them include Nginx, Apache, Python, Java, Mongo, Node, MySQL, Ubuntu, Fedora, Centos, etc.

You can even create your private repository inside Dockerhub and store your custom Docker images using the Docker push command.



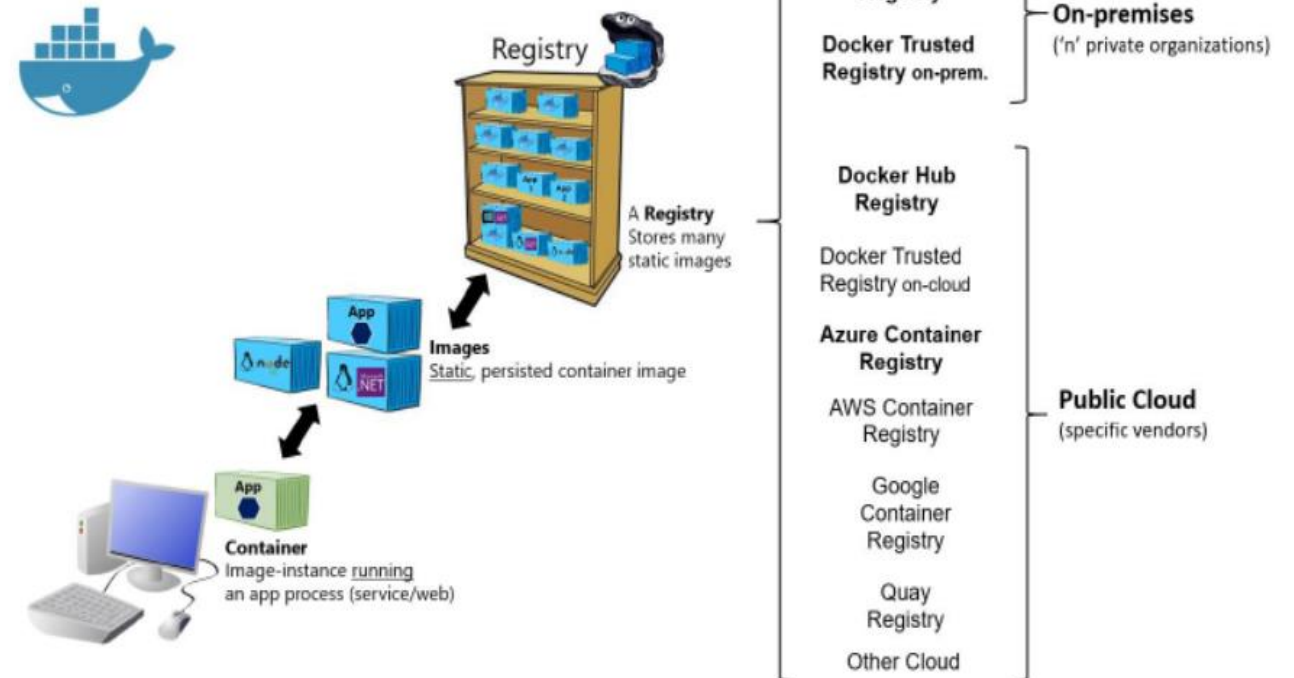
Docker Objects

A Docker user frequently interacts with

Docker objects such as

- Images
- Containers
- Volumes
- Plugins
- Networks and so on.

Basic taxonomy in Docker



Docker Objects - Images

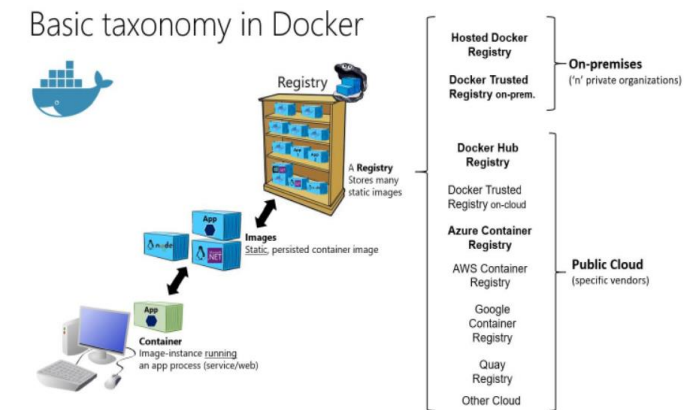
Docker Images are read-only templates that are built using multi-layers of file.

You can build Docker images using a simple text file called Dockerfile which contains instructions to build Docker images.

The first instruction is a FROM instruction which can pull a base image from any Docker registry. Once this base image layer is created, several instructions are then used to create the container environment. Each instruction adds a new layer on top of the previous one.

A Docker image is simply a **blueprint of the container environment**. Once you create a container, it creates a writable layer on top of the image, and then, you can make changes. The images all the metadata that describes the container environment. You can either directly pull a Docker image from Dockerhub or create your customized image over a base image using a Dockerfile.

Once you have created a Docker image, you can push it on **Dockerhub** or any other registry and share it with the outside world.

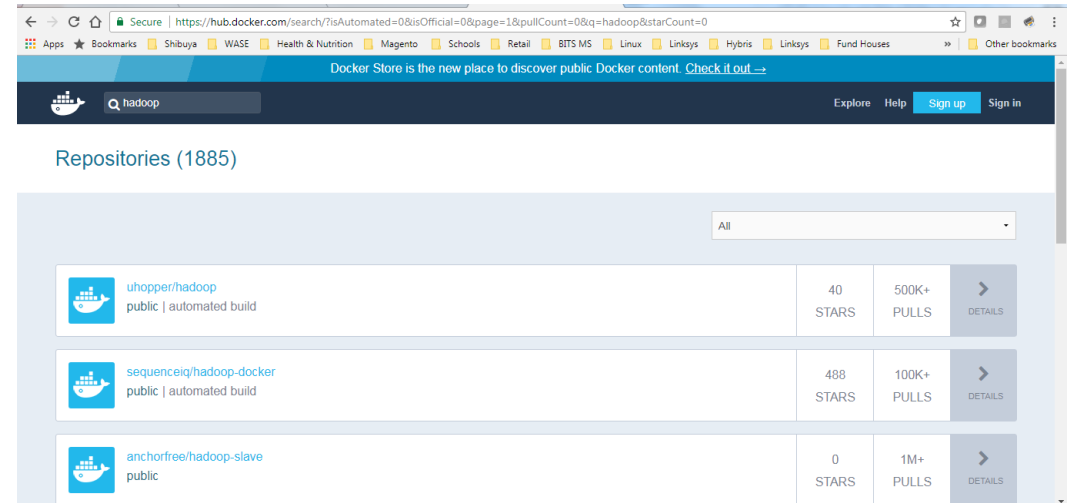


Docker Objects - Images

docker image index: a repository (public or private) for docker images

- Docker images constitute the base of docker containers *from* which everything starts to *form*. They are very similar to default operating-system disk images which are used to run applications on servers or desktop computers.
- Having these images (e.g. Ubuntu base) allow seamless portability across systems. They make a solid, consistent and dependable base with everything that is needed to run the applications. When everything is self-contained and the risk of system-level updates or modifications are eliminated, the container becomes immune to external exposures which could put it out of order - *preventing the dependency hell*.

Note: Repositories contain images from un verified sources. Exercise caution. Always use the official images

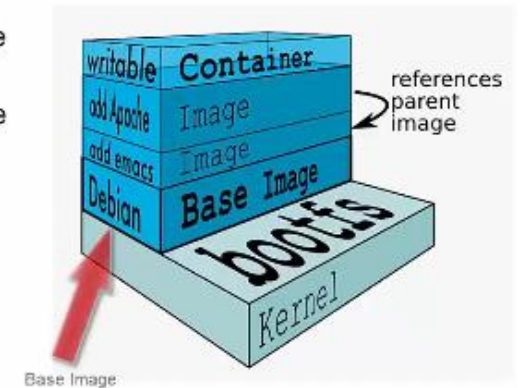


Docker Objects - Images

- As more layers (tools, applications etc.) are added on top of the base, new images can be formed by *committing* these changes. When a new container gets created from a saved (i.e. committed) image, things continue from where they left off. And the [union file system](#), brings all the layers together as a single entity when you work with a container.
- These base images can be explicitly stated when working with the **docker CLI** to directly create a new container or they might be specified inside a **Dockerfile** for automated image building.

Image Layers

- Images are comprised of multiple layers
- A layer is also just another image
- Every image contains a base layer
- Docker uses a copy on write system
- Layers are read only

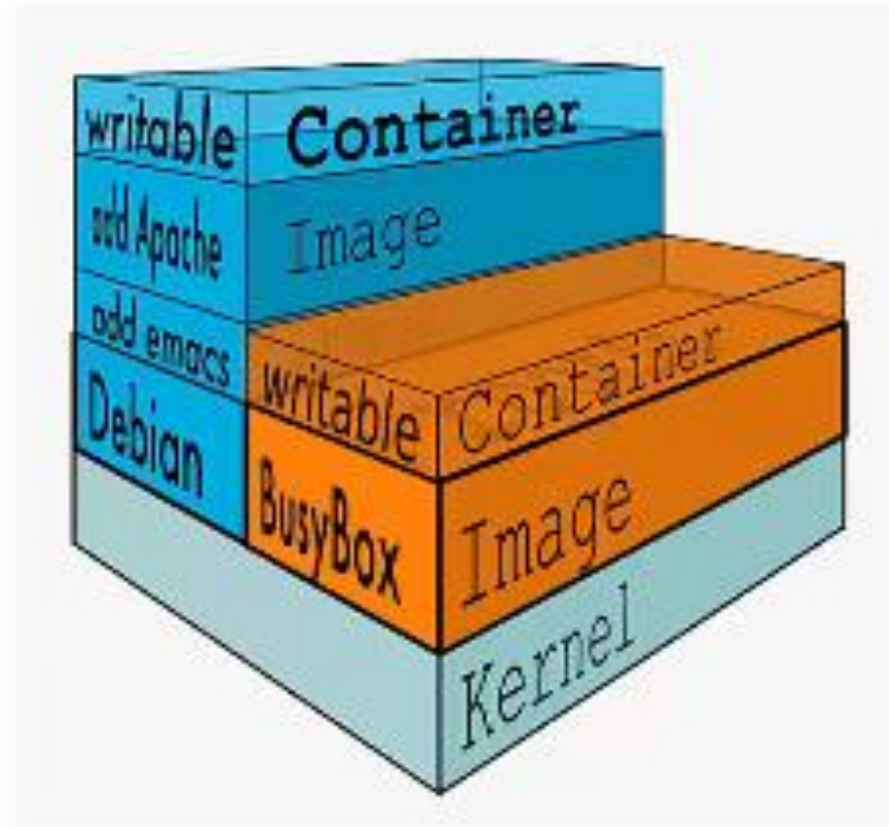


Note: Repositories contain images from unverified sources. Exercise caution. Always use the official images

Docker Objects - Images

The Container Writable Layer

- Docker creates a top writable layer for containers
- Parent images are read only
- All changes are made at the writeable layer



Docker Objects - Containers

Docker containers are isolated, encapsulated, packaged, and secured application environments that contain all the packages, libraries, and dependencies required to run an application.

For example, if you create a container associated with the Ubuntu image, you will have access to an isolated Ubuntu environment. You can also access the bash of this Ubuntu environment and execute commands.

Containers have all the access to the resources that you define while using the **Dockerfile** while creating an image. Such configurations include *build context, network connections, storage, CPU, memory, ports, etc.*

For example, if you want access to a container with libraries of Java installed, you can use the Java image from the Dockerhub and run a container associated with this image using the Docker run command.

You can also create containers associated with the custom images that you create for your application using the Dockerfiles. Containers are very light and can be spun within a matter of seconds.

Creating a Container

- Use **docker run** command
- Syntax
`sudo docker run [options] [image] [command] [args]`
- Image is specified with `repository:tag`

Examples

```
docker run ubuntu:14.04 echo "Hello World"
docker run ubuntu ps ax
```

Docker Objects - DockerFiles

An alternative to the previous image build method using docker commit

Dockerfiles are scripts containing a successive series of instructions, directions, and commands which are to be executed to form a new docker image.

Each command executed translates to a new layer of the onion, forming the end product.

They basically replace the process of doing everything manually and repeatedly.

When a **Dockerfile** is finished executing, you end up having formed an image, which then you use to start (i.e. create) a new container.



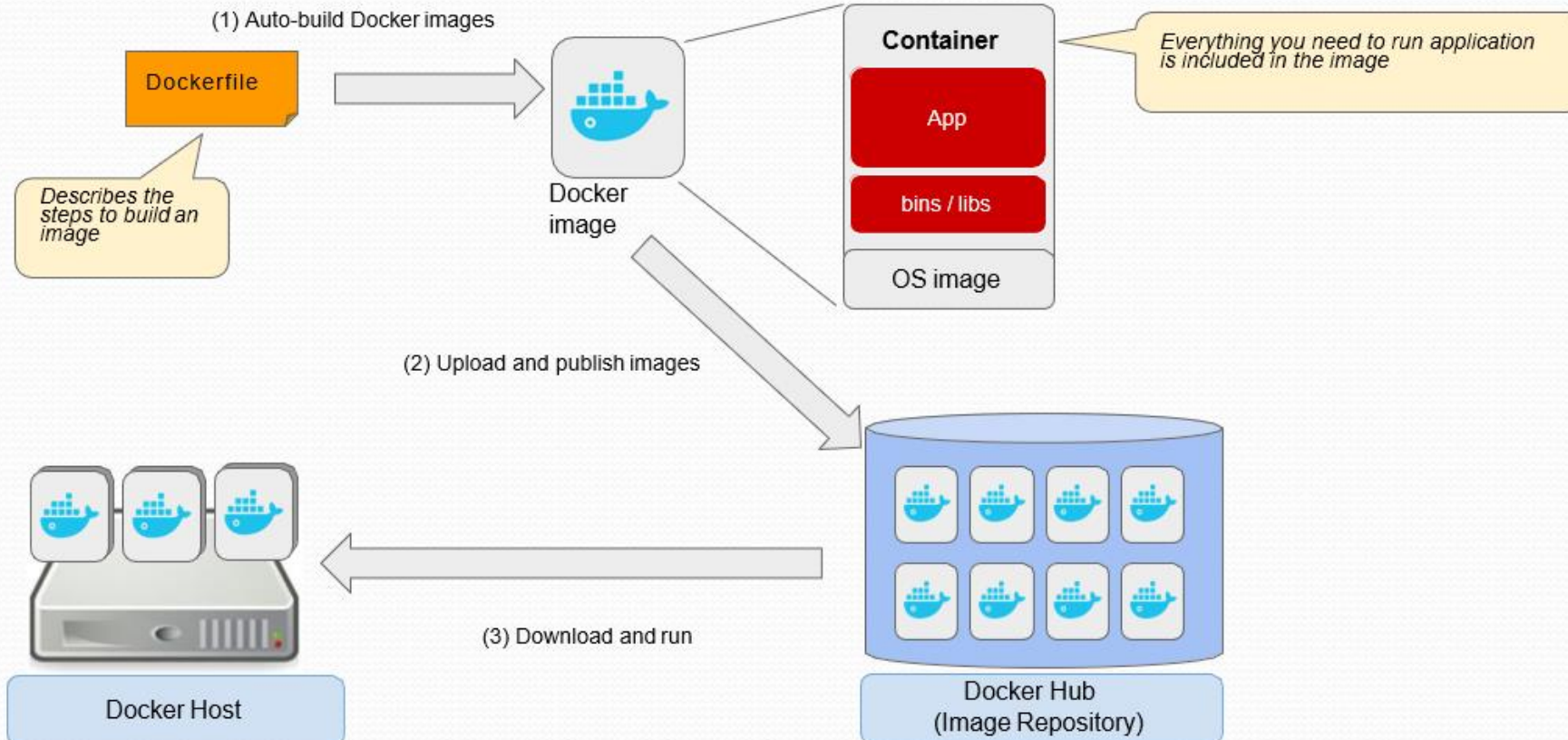
Docker Objects - DockerFiles

Dockerfile Instructions

- Instructions specify what to do when building the image
- **FROM** instruction specifies what the base image should be
- **RUN** instruction specifies a command to execute

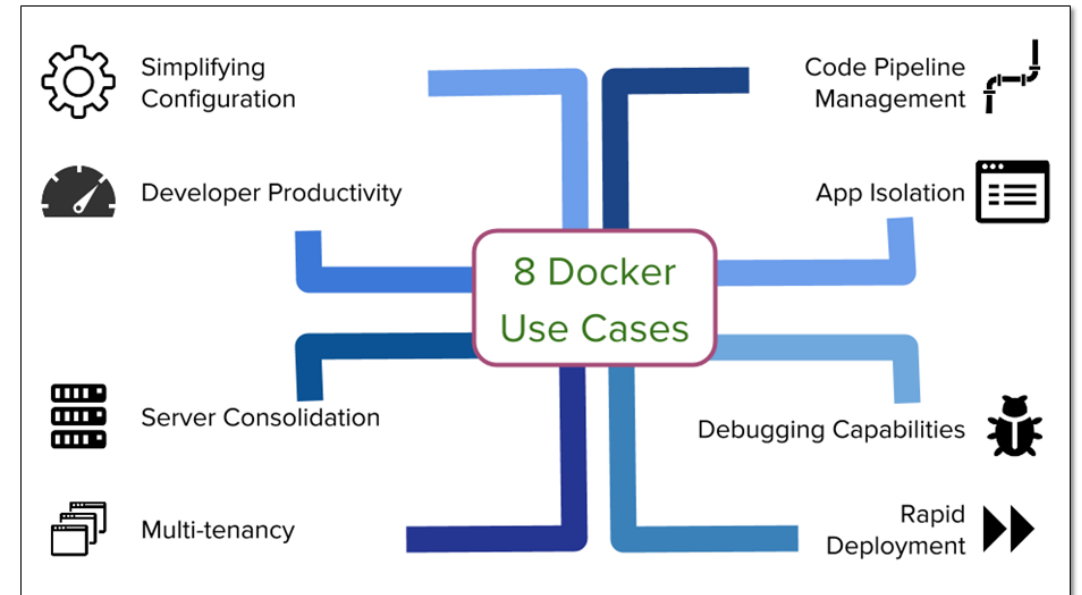
```
#Example of a comment  
FROM ubuntu:14.04  
RUN apt-get install vim  
RUN apt-get install curl
```

Docker Life Cycle



Docker Summary

- A **container** is one or more runtime instances of a Docker image that usually will contain a single app/service. The container is **considered the live artifact being executed in a development machine or the cloud or server**.
- An **image** is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. An image typically contains a union of layered filesystems (deltas) stacked on top of each other. An image does not have state and it never changes.
- A **registry** is a service containing repositories of images from one or more development teams. Multiple development teams may also instance multiple registries. The default registry for Docker is the public "Docker Hub" but you will likely have your own private registry network close to your orchestrator to manage and secure your images, and reduce network latency when deploying images.
- The beauty of the images and the registry resides on the possibility for you to store static and immutable application bits including all their dependencies at OS and frameworks level so they can be versioned and deployed in multiple environments providing a consistent deployment unit.

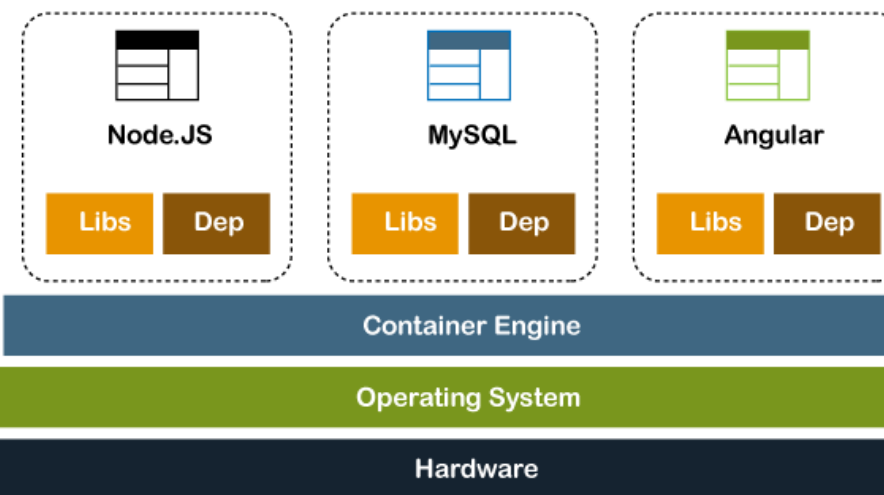


Docker – Video





Container Orchestration

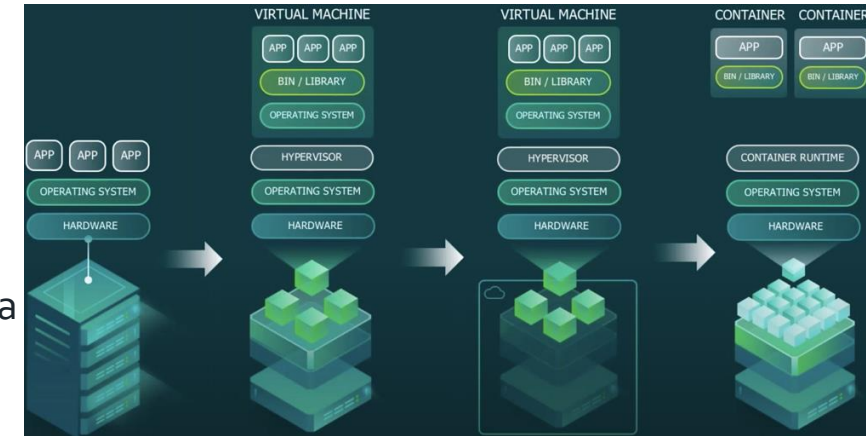


Container Orchestration

Container orchestration is all **about managing the lifecycles of containers**, especially in large, **dynamic environments**.

Software teams use container orchestration to control and automate many tasks:

- Provisioning and deployment of containers
- Scaling up or removing containers to spread application load evenly across host infrastructure
- Movement of containers from one host to another if there is a shortage of resources in a host, or if a host dies
- External exposure of services running in a container with the outside world
- Load balancing of service discovery between containers
- Health monitoring of containers and hosts
- Configuration of an application in relation to the containers running it



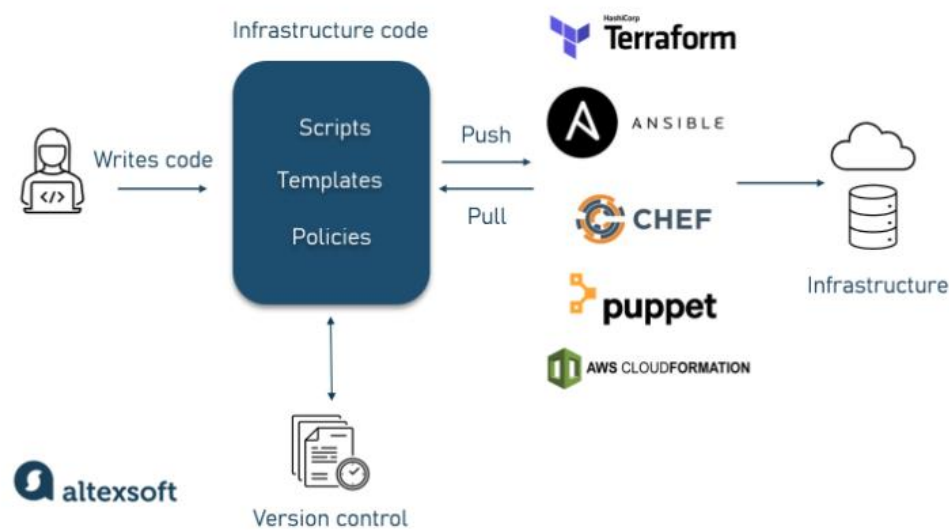
Container Orchestration

KEY ORCHESTRATION FEATURES

- Management of container hosts
- Launching set of containers
- Rescheduling failed containers
- Linking containers to support workflows
- Providing connectivity to clients outside the container cluster
- Firewall: control network/port accessibility
- Dynamic scaling of containers: horizontal scaling
 - Scale in/out, add/remove containers
- Load balancing over groups of containers
- Rolling upgrades of containers for application

IAC






HOW INFRASTRUCTURE AS CODE WORKS



Infrastructure as Code implementation

- 1.The team writes infrastructure configurations in a required programming language.
- 2.The code files are sent to a code repository.
- 3.An IaC tool runs the code and performs the required activities.

COMPARING INFRASTRUCTURE AS CODE TOOLS

Name	Languages	Function	Approach	Infrastructure type
 Terraform	HCL + Typescript, Python, Java, C#, Go with CDK	Provisioning	Declarative	Immutable
 AWS CLOUDFORMATION	JSON, YAML + TypeScript, Python, Java, .NET, and Go with CDK	Provisioning	Declarative	Both
 ANSIBLE	Python, Ruby, YAML	Configuration management	Imperative	Mutable
 puppet	PuppetDSL, YAML	Configuration management	Declarative	Mutable
 CHEF	Ruby	Configuration management	Declarative	Mutable

Imperative vs Declarative Model in IAC

The terms [imperative and declarative](#) come up frequently in IAC discussions.

Both terms refer to how the user provides direction to the automation platform.

With an imperative tool, you define the steps to execute in order to reach the desired solution.

With a declarative tool, you define the desired state of the final solution, and the automation platform determines how to achieve that state.

Imperative systems are often initially easier.

You could say that an imperative system is organized more like how a human thinks.

Imperative systems allow you to continue to think about configuration as a series of actions or steps, each bringing you closer to the goal.

Another benefit of imperative language is that it allows you to automate very detailed and complex configurations by building up multiple layers of commands. And, because an imperative system gives the user more control over how to accomplish a task, it is often more efficient and easier to optimize for a specific purpose than a declarative system.

Imperative vs Declarative Model in IAC

With all the advantages of imperative languages, you might be wondering why declarative languages are so much in vogue.

Declarative tools have been gaining popularity for several years now and arguably are the dominant format for IaC automation.

One reason for the popularity of declarative tools is that they require less knowledge on the user's part, at least once you understand how they work.

With an imperative tool, the user must have enough knowledge to tell the automation platform what to do.

With a declarative system, the user only needs to define the state of the final configuration, and the platform determines how to get there. The complexity of the step-by-step execution is hidden from the user.

Another benefit of a declarative language is that it is more *idempotent*. The concept of idempotence refers to a process that can be executed multiple times with the same result. Because a declarative language just defines the final state, you always end up in the same place no matter where you start. On the other hand, an imperative language envisions a task as a series of predefined steps that could take you to a different endpoint depending on the starting point.

Imperative vs Declarative Model in IAC

Think about **imperative configuration** like a remote control for a television.

You use the buttons on it to change things until it's how you want it to be. Make the sound louder, change the channel, turn on recording, etc.

Simply put, **imperative** is the "**how**" of configuration paradigms.

In an edge API Gateway, you can go to an interface or a REST API to specifically enter the timeout for routing rules.

First, it's 8 seconds. Then later on, you change it to 5 seconds. You embed how you want it to work into the control flow of your API or the program calling it.

In our analogy, the **declarative model** is like a thermostat. You know that you'll be cold until it's 70F so you set the temperature to 70F and you go about your day.

The thermostat detects its environment and controls the temperature accordingly.

Declarative is **expressing intent** of "**how**" you want your **system to work**.

This model uses a GitOps style approach, because you can write down in your source file that the timeout is 10 seconds and that is committed to source control. Your continuous integration system reads that source file and deploys it to your Kubernetes cluster. So, Git becomes your source of truth.

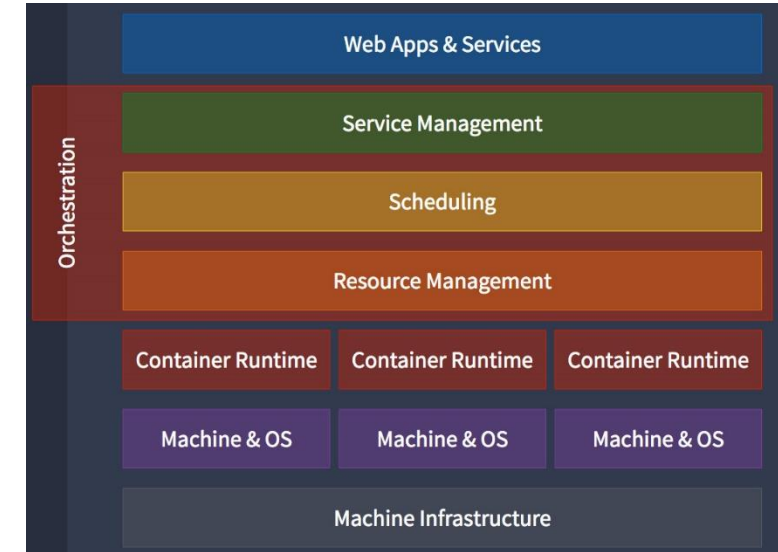
Container Orchestration

Most container orchestration tools support a **declarative configuration model**:

A developer writes a **configuration file** (in YAML or JSON depending on the tool) that defines a desired configuration state, and the orchestration tool runs the file uses its own intelligence to achieve that state. The configuration file typically

- Defines which container images make up the application, and where they are located (in what registry)
- Provisions the containers with storage and other resources
- Defines and secures the network connections between containers
- Specifies versioning (for phased or canary rollouts)

The orchestration tool schedules deployment of the containers (and replicas of the containers, for resiliency) to a host, choosing the best host based on available CPU capacity, memory, or other requirements or constraints specified in the configuration file.



Container orchestration mediates between the apps or services above and the container runtimes below. Three main functional aspects of what they do include:*

- **Service Management:** Labels, groups, namespaces, dependencies, load balancing, readiness checks.
- **Scheduling:** Allocation, replication, resurrection, rescheduling, rolling deployment, upgrades, downgrades.
- **Resource Management:** Memory, CPU, GPU, volumes, ports, IPs.

Kubernetes – Introduction Video



Kubernetes is a self-service Platform-as-a-Service (PaaS) that creates a hardware layer abstraction for development teams. Kubernetes is also extremely portable. It runs on Amazon Web Services (AWS), Microsoft Azure, the Google Cloud Platform (GCP), or in on-premise installations. we can move workloads without having to redesign your applications or completely rethink your infrastructure—which helps you to standardize on a platform and avoid vendor lock-in.

Container Orchestration - Tools

Tools of Container Orchestration



Amazon ECS
FROM AMAZON



Azure Container Services
FROM MICROSOFT



Docker Swarm
DOCKER OPENSOURCE TOOLS



Google Container Engine
FROM GOOGLE CLOUD PLATFORM



Kubernetes
DOCKER OPENSOURCE TOOLS



CoreOS Fleet
FROM COREOS



Mesosphere Marathon
FROM MARATHON



Cloud Foundry's Diego
FROM CLOUD FOUNDRY

- **Docker Swarm:** Provides native clustering functionality for Docker containers, which turns a group of Docker engines into a single, virtual Docker engine.
- **Google Container Engine:** Google Container Engine, built on Kubernetes, lets you run Docker containers on the Google Cloud.
- **Kubernetes:** An orchestration system for Docker containers. It handles scheduling and manages workloads based on user-defined parameters.
- **Mesosphere Marathon:** Marathon is a container orchestration framework for Apache Mesos that is designed to launch long-running applications.
- **Amazon ECS:** The ECS supports Docker containers and lets you run applications on a managed cluster of Amazon EC2 instances.
- **Azure Container Service (ACS):** ACS lets you create a cluster of virtual machines that act as container hosts along with master machines that are used to manage your application containers.
- **Cloud Foundry's Diego:** Container management system that combines a scheduler, runner, and health manager.
- **CoreOS Fleet:** Container management tool that lets you deploy Docker containers on hosts in a cluster as well as distribute services across a cluster.

Q & A.....





BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Credits

*Hwang, Kai; Dongarra, Jack; Fox, Geoffrey C.. Distributed and Cloud Computing: From Parallel Processing to the Internet of Things (Kindle Locations 3532-3533). Elsevier Science. Kindle Edition.