MODULE-V

Containerization with Docker

What is Docker?

Docker is an open-source containerization platform. It enables developers to package applications into containers—standardized executable components combining application source code with the operating system (OS) libraries and dependencies required to run that code in any environment.

Docker is an OS virtualized software platform that allows IT organizations to easily create, deploy, and run applications in Docker containers, which have all the dependencies within them.

Features of Docker:

1. **Faster and Easier configuration:**
   It is one of the key features of Docker that helps you in configuring the system in a faster and easier manner. Due to this feature, codes can be deployed in less time and with fewer efforts. The infrastructure is not linked with the environment of the application as Docker is used with a wide variety of environments.

2. **Increase in productivity:**
   It helps in increasing productivity by easing up the technical configuration and rapidly deploying applications. Moreover, it not only provides an isolated environment to execute applications, but it reduces the resources as well.

3. **Application isolation:**
   Docker provides containers that are used to run applications in an isolated environment. Since each container is independent, Docker can execute any kind of application.

4. **Swarm:**
   Swarm is a clustering and scheduling tool for Docker containers. At the front end, it uses the Docker API, which helps us to use various tools to control it. It is a self-organizing group of engines that enables pluggable backends.

5. **Services:**
   Services is a list of tasks that specifies the state of a container inside a cluster. Each task in the Services lists one instance of a container that should be running, while Swarm schedules them across the nodes.

6. **Routing Mesh**
   It routes the incoming requests for published ports on available nodes to an active container. This feature enables the connection even if there is no task is running on the node.

Why we need Docker

1. **Portability**
   Once you have tested your containerized application you can deploy it to any other system where Docker is running and you can be sure that your application will perform exactly as it did when you tested it.

2. **Performance**
   Although virtual machines are an alternative to containers, the fact that containers do not contain an operating system (whereas virtual machines do) means that containers have much smaller footprints than virtual machines, are faster to create, and quicker to start.

3. **Scalable and Cost effective**
   Running your service on hardware that is much cheaper than standard servers. You can quickly create new containers if demand for your applications requires them. When using multiple containers, you can take advantage of a range of container management options.

What is Virtualization?

Virtualization is the technique of importing a Guest operating system on top of a Host operating system. This technique was a revelation at the beginning because it allowed developers to run multiple operating systems in different virtual machines all running on the same host. This eliminated the need for extra hardware resource.

The advantages of Virtual Machines or Virtualization are:

1. **Multiple operating systems can run on the same machine.**
2. **Maintenance and Recovery were easy in case of failure conditions.**
3. **Total cost of ownership was also less due to the reduced need for infrastructure.**

In the diagram, you can see there is a host operating system on which there are 3 guest operating systems running which is nothing but the virtual machines.

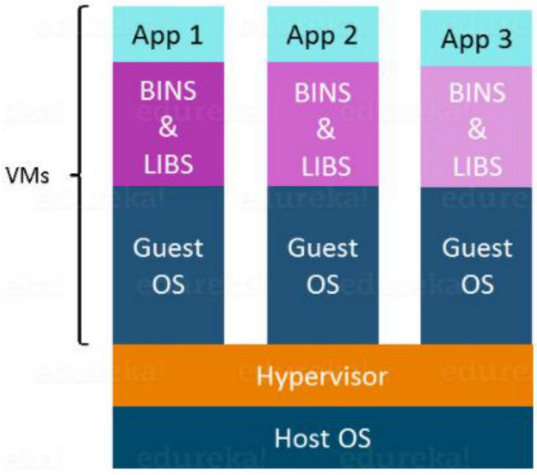


Fig: Virtualization

Running multiple Virtual Machines in the same host operating system leads to performance degradation. This is because of the guest OS running on top of the host OS, which will

have its own kernel and set of libraries and dependencies. This takes up a large chunk of system resources, i.e. hard disk, processor and especially RAM.

Another problem with Virtual Machines which uses virtualization is that it takes almost a minute to boot-up. This is very critical in case of real-time applications.

Following are the disadvantages of Virtualization:

- Running multiple Virtual Machines leads to unstable performance.
- Hypervisors are not as efficient as the host operating system.
- Boot up process is long and takes time.
- These drawbacks led to the emergence of a new technique called Containerization. Now let me tell you about Containerization.

What is Containerization?

Containerization is the technique of bringing virtualization to the operating system level. While Virtualization brings abstraction to the hardware, Containerization brings abstraction to the operating system. Do note that Containerization is also a type of Virtualization.

Containerization is however more efficient because there is no guest OS here and utilizes a host's operating system, share relevant libraries & resources as and when needed unlike virtual machines. Application specific binaries and libraries of containers run on the host kernel, which makes processing and execution very fast. Even booting-up a container takes only a fraction of a second. Because all the containers share, host operating system and holds only the application related binaries & libraries. They are lightweight and faster than Virtual Machines.

Advantages of Containerization over Virtualization:

- Containers on the same OS kernel are lighter and smaller
- Better resource utilization compared to VMs
- Boot-up process is short and takes few seconds

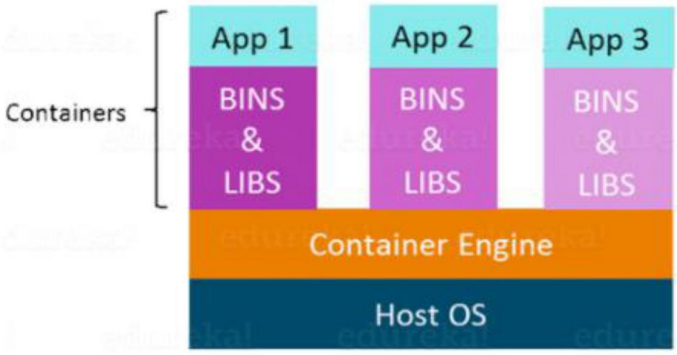
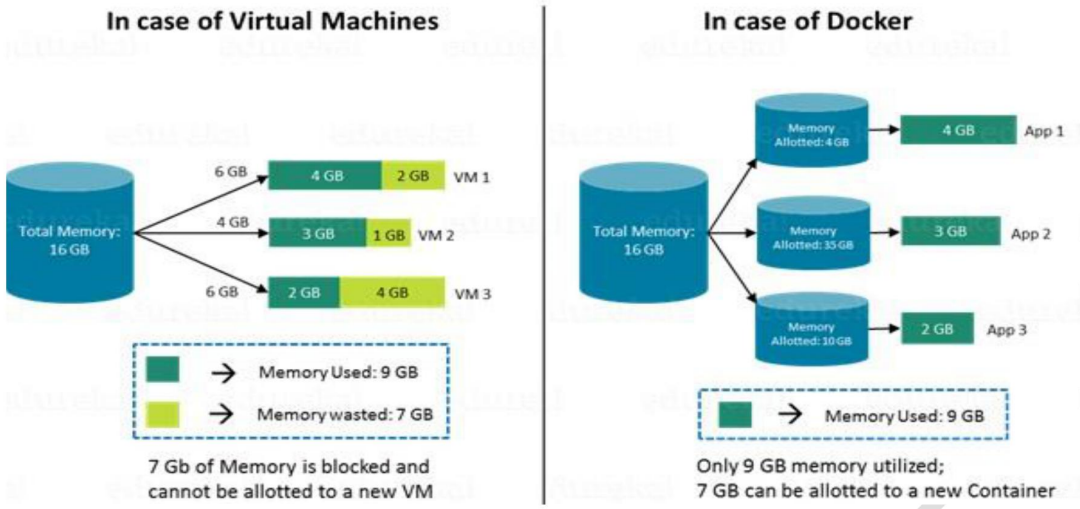


Fig: Containerization

In the diagram on the right, you can see that there is a host operating system which is shared by all the containers. Containers only contain application specific libraries which are separate for each container and they are faster and do not waste any resources.

All these containers are handled by the containerization layer which is not native to the host operating system. Hence a software is needed, which can enable you to create & run containers on your host operating system.

Docker vs Virtualization



Virtualization	Containerization
Virtualizes hardware resources	Virtualizes only OS resources
Takes more time for Booting	Takes less time for Booting
Each VM runs in its own OS.	All containers share the host OS
Requires more Memory	Requires less memory
Requires the complete OS installation for every VM	Installs the container only on a host OS
Heavyweight	Lightweight
It is not portable	It is very portable. We can build, ship and run anywhere
Once the memory is allocated for virtual machine, the unused memory can't be shared/reallocate which leads to Wastage of Memory	The unused memory can be shared across other Containers.
Use Hardware level virtualization	Uses OS Virtualization
Limited performance	Native performance
Fully isolated	Process-level isolation
More Secure	Less Secure
Ex: Docker, LXC, LXCD , CG Manager	VMware, vSphere, Virtual Box, Hyper-V

Docker Architecture

Docker uses a client-server architecture. The Docker client consists of Docker build, Docker pull, and Docker run. The client approaches the Docker daemon that further helps in building, running, and distributing Docker containers. Docker client and Docker daemon can be operated on the same system; otherwise, we can connect the Docker client to the remote Docker daemon. Both communicate with each other by using the REST API, over UNIX sockets or a network.

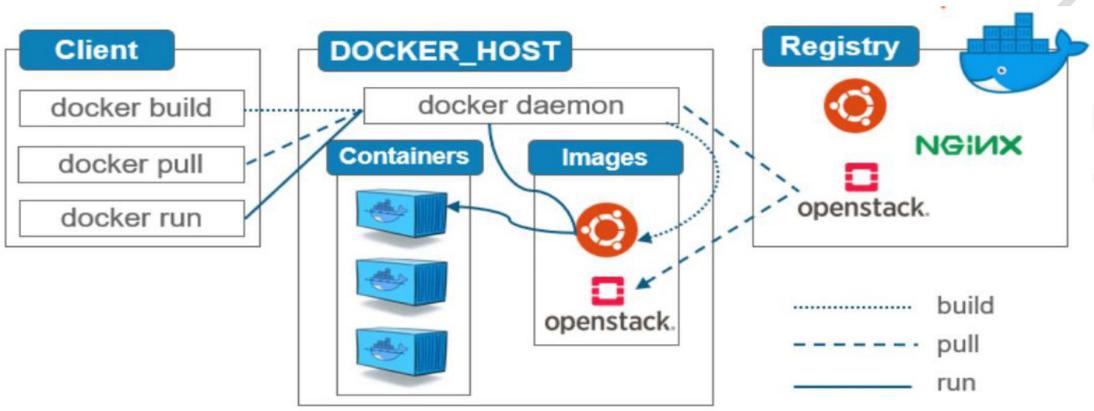


Fig: Docker Architecture

Docker Engine

It is the core part of the whole Docker system. Docker Engine is an application which follows client-server architecture. It is installed on the host machine. There are three components in the Docker Engine:

Server: It is the docker daemon called dockerd. The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker daemon pulls and builds container images as requested by the client. Once it pulls a requested image, it builds a working model for the container by utilizing a set of instructions known as a build file. The build file can also include instructions for the daemon to pre-load other components prior to running the container, or instructions to be sent to the local command line once the container is built.

Rest API: An API used by applications to interact with the Docker daemon; it can be accessed by an HTTP client. It is used to instruct docker daemon what to do.

Command Line Interface (CLI): It is a client which is used to enter docker commands.

Docker Client

Docker client uses commands and REST APIs to communicate with the Docker Daemon (Server). When a client runs any docker command on the docker client terminal, the client terminal sends these docker commands to the Docker daemon. Docker daemon receives these commands from the docker client in the form of command and REST API's request.

The Docker client can reside on the same host as the daemon or connect to a daemon on a remote host. A docker client can communicate with more than one daemon.

The main purpose of the Docker Client is to provide a means to direct the pull of images from a registry and to have it run on a Docker host. Docker Client uses Command Line Interface (CLI) to run the following commands -**docker build, docker pull and docker run.**

Docker host

The Docker host provides a complete environment to execute and run applications. It comprises of the Docker daemon, Images, Containers, Networks, and Storage. As previously mentioned, the daemon is responsible for all container-related actions and receives commands via the CLI or the REST API.

Docker Registry

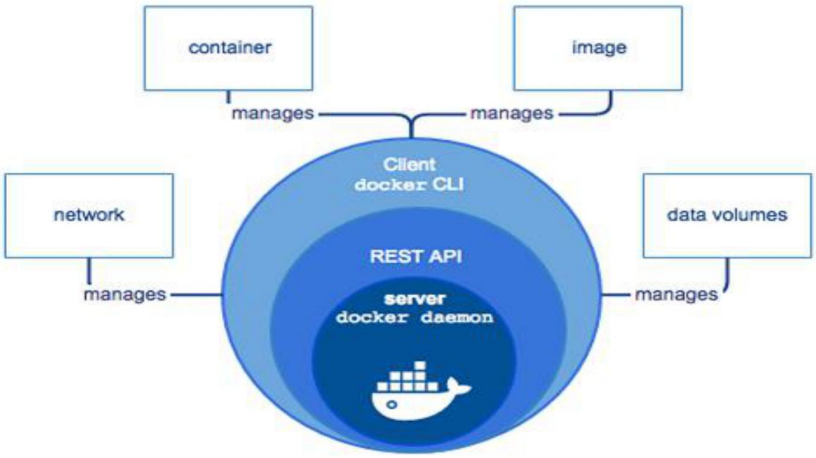
Docker registry is a repository which manages and stores the Docker images that are used for creating Docker containers. There are two types of registries in the Docker -
Pubic Registry - Public Registry is also called as Docker hub.
Private Registry - It is used to share images within the enterprise.

The Registry can be either a user's local repository or a public repository like a Docker Hub allowing multiple users to collaborate in building an application. Even with multiple teams within the same organization can exchange or share containers by uploading them to the Docker Hub, which is a cloud repository similar to GitHub.

Components Docker

The four key components that make up the entire Docker architecture are:

1. The Docker Daemon or the server.
2. The Docker Command Line Interface or the client
3. Docker Registries
4. Docker Objects –
    i. Images
    ii. Containers
    iii. Network
    iv. Storage



1. Docker daemon/Server

The docker daemon called dockerd. The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.
Rest API: An API used by applications to interact with the Docker daemon; it can be accessed by an HTTP client. It is used to instruct docker daemon what to do.

2. Docker Client

Docker client uses commands and REST APIs to communicate with the Docker Daemon (Server). When a client runs any docker command on the docker client terminal, the client terminal sends these docker commands to the Docker daemon. Docker daemon receives these commands from the docker client in the form of command and REST API's request.

The main purpose of the Docker Client is to provide a means to direct the pull of images from a registry and to have it run on a Docker host. Docker Client uses **Command Line Interface (CLI)** to run the following commands -**docker build, docker pull and docker run.**

3. Docker Registry

Docker registry is a repository which manages and stores the Docker images that are used for creating Docker containers. There are two types of registries in the Docker -
Pubic Registry - Public Registry is also called as Docker hub.
Private Registry - It is used to share images within the enterprise.

You can even create your private repository inside Dockerhub and store your custom Docker images using the Docker push command. Docker allows you to create your own private Docker registry in your local machine using an image called 'registry'. Once you run a container associated with the registry image, you can use the Docker push command to push images to this private registry.

4. Docker Objects

When you are working with Docker, you use images, containers, volumes, networks; all these are Docker objects.

i.Docker Images

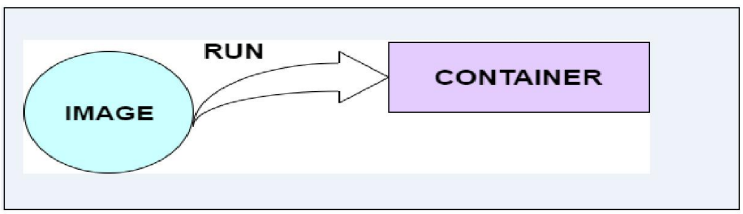
An image is a read-only template with instructions for creating a Docker container. Docker image can be pulled from a Docker hub and used as it is, or you can add additional instructions to the base image and create a new and modified docker image. You can create your own docker images also using a dockerfile. Create a dockerfile with all the instructions to create a container and run it; it will create your custom docker image.

It uses a private container registry to share container images within the enterprise and also uses public container registry to share container images within the whole world.

ii.Docker Containers

In simple terms, an image is a template, and a container is a copy of that template. You can have multiple containers (copies) of the same image.

Let's understand about containers in a different way that, if an image is a class, then a container is an instance of a class—a runtime object. Or you can say they are running instances of the Images and they hold the entire package and dependencies needed to run the application.



The container is also inherently portable. Another benefit is that it runs completely in isolation. Even if you are running a container, it's guaranteed not to be impacted by any host OS securities or unique setups, unlike with a virtual machine or a non-containerized environment. The memory for a Docker environment can be shared across multiple containers, which is really useful, especially when you have a virtual machine that has a defined amount of memory for each environment.

By using the Docker API or CLI, it is possible to ship, create, start, stop or delete a container.

iii.Networks

Docker networking is a passage through which all the isolated container communicates and share data or information. There are mainly five network drivers in docker: **Bridge Driver, Host Driver, Overlay Driver, Macvlan and None.**

iv.Storage

As soon as you exit a container, all your progress and data inside the container are lost. To avoid this, you need a solution for persistent storage. Docker provides several options for persistent storage using which you can share, store, and backup your valuable data. These are - **Data Volumes, Volume Container, Directory Mounts and Storage Plugins.**

APPENDIX

Docker Networks

Docker networking is a passage through which all the isolated container communicate. There are mainly five network drivers in docker:

1. **Bridge:** It is the default network driver for a container. You use this network when your application is running on standalone containers, i.e. multiple containers communicating with the same docker host.

2. **Host:** This driver removes the network isolation between docker containers and docker host. You can use it when you don't need any network isolation between host and container.

3. **Overlay:** This network enables swarm services to communicate with each other. You use it when you want the containers to run on different Docker hosts or when you want to form swarm services by multiple applications.

4. **None:** This driver disables all the networking.

5. **Macvlan:** This driver assigns mac address to containers to make them look like physical devices. It routes the traffic between containers through their mac addresses. You use this network when you want the containers to look like a physical device, for example, while migrating a VM setup.

Docker Storage

You can store data within the writable layer of a container but it requires a storage driver. Being non-persistent, it perishes whenever the container is not running. Moreover, it is not easy to transfer this data. With respect to persistent storage, Docker offers four options:

1. **Data Volumes:** They provide the ability to create persistent storage, with the ability to rename volumes, list volumes, and also list the container that is associated with the volume. Data Volumes are placed on the host file system, outside the containers copy on write mechanism and are fairly efficient.

2. **Volume Container:** It is an alternative approach wherein a dedicated container hosts a volume and to mount that volume to other containers. In this case, the volume container is independent of the application container and therefore you can share it across more than one container.

3. **Directory Mounts:** Another option is to mount a host's local directory into a container. In the previously mentioned cases, the volumes would have to be within the Docker volumes folder, whereas when it comes to Directory Mounts any directory on the Host machine can be used as a source for the volume.

4. **Storage Plugins:** Storage Plugins provide the ability to connect to external storage platforms. These plugins map storage from the host to an external source like a storage array or an appliance. You can see a list of storage plugins on Docker's Plugin page.