**Docker**

Many of us believe that Docker is an integral part of [*DevOps*](https://bit.ly/2mFJsSU). So behind this incredible tool, there has to be an amazing architecture. In this blog, I will be covering everything that you must know about the Docker architecture. These are the points that I will be discussing here

1. Traditional Virtualization vs Docker
2. Docker’s Workflow
3. Docker Architecture:
   * Docker’s Client
   * Docker Host
   * Docker Objects
   * Docker’s Registry

**Traditional Virtualization Vs Docker**

**What is a VM (Virtual Machine)?**

A VM is a virtual server that emulates a hardware server. A virtual machine relies on the system’s physical hardware to emulate the exact same environment in which you install your applications. Depending on your use case, you can use a system virtual machine (that runs an entire OS as a process, allowing you to substitute a real machine for a virtual machine), or process virtual machines that let you execute computer applications alone in the virtual environment.

Earlier, we used to create virtual machines, and each VM had an OS which took a lot of space and made it heavy.

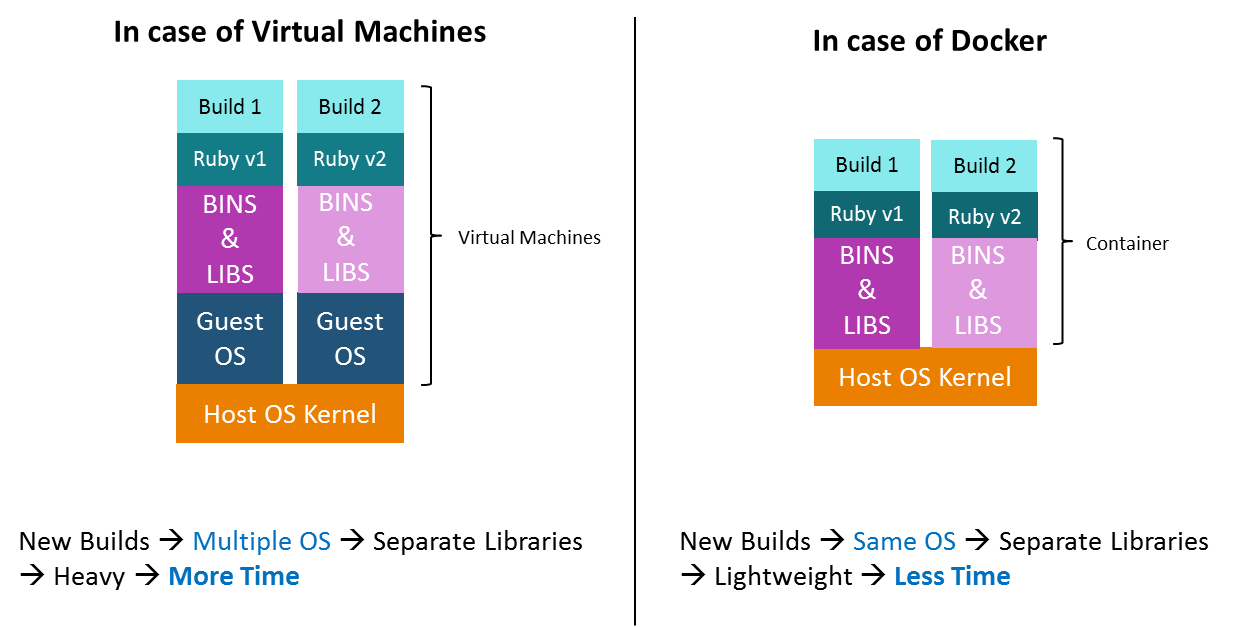
**What is Docker?**

Docker is an open-source project that offers a software development solution known as containers. To understand Docker, you need to know what containers are. According to [*Docker*](https://www.docker.com/)*,* a container is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it.

Containers are platform-independent and hence Docker can run across both Windows and Linux-based platforms. In fact, Docker can also be run within a virtual machine if there arises a need to do so. The main purpose of Docker is that it lets you run microservice applications in a distributed architecture.

When compared to Virtual machines, the Docker platform moves up the abstraction of resources from the hardware level to the Operating System level. This allows for the realization of the various benefits of Containers e.g. application portability, infrastructure separation, and self-contained microservices.

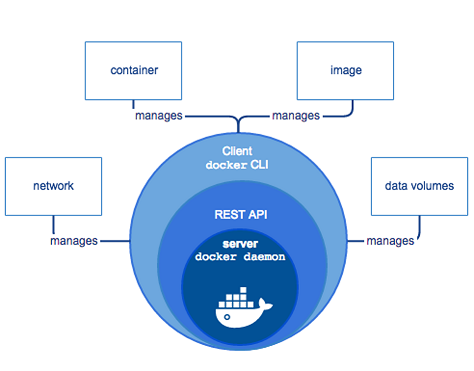
In other words, while Virtual Machines abstract the entire hardware server, Containers abstract the Operating System kernel. This is a whole different approach to virtualization and results in a much faster and more lightweight instances.



**Docker’s Workflow**

First, let us look take a look at Docker Engine and its components so we have a basic idea of how the system works. Docker Engine allows you to develop, assemble, ship, and run applications using the following components:

1. **Docker Daemon**: A persistent background process that manages Docker images, containers, networks, and storage volumes. The Docker daemon constantly listens for Docker API requests and processes them.
2. **Docker Engine REST API**: An API is used by applications to interact with the Docker daemon. It can be accessed by an HTTP client.
3. **Docker CLI**: A command-line interface client for interacting with the Docker daemon. It significantly simplifies how you manage container instances and is one of the key reasons why developers love using Docker.

At first, Docker client talks to the Docker daemon, which performs the heavy lifting of the building, running, as well as distributing our Docker containers. Fundamentally, both the Docker client and daemon can run on the same system. We can also connect a Docker client to a remote Docker daemon. In addition, by using a REST API, the Docker client and daemon, communicate, over UNIX sockets or a network interface.

**Docker Architecture**

The architecture of Docker uses a client-server model and consists of the Docker’s Client, Docker Host, Network and Storage components, and the Docker Registry/Hub. Let’s look at each of these in some detail.

**Docker’s Client**

Docker users can interact with Docker through a client. When any docker commands runs, the client sends them to dockerd daemon, which carries them out. Docker API is used by Docker commands. It is possible for Docker client to communicate with more than one daemon.

**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. It can also communicate with other daemons to manage its services.

**Docker Objects**

**1. Images**

Images are nothing but a read-only binary template that can build containers. They also contain metadata that describe the container’s capabilities and needs. Images are used to store and ship applications. An image can be used on its own to build a container or customized to add additional elements to extend the current configuration.

You can share the container images across teams within an enterprise with the help of a private container registry, or share it with the world using a public registry like Docker Hub. Images are the core element of the Docker experience as they enable collaboration between developers in a way that was not possible before

**2. Containers**

Containers are sort of encapsulated environments in which you run applications. Container is defined by the image and any additional configuration options provided on starting the container, including and not limited to the network connections and storage options. Containers only have access to resources that are defined in the image, unless additional access is defined when building the image into a container.

You can also create a new image based on the current state of a container. Since containers are much smaller than VMs, they can be spun in a matter of seconds, and result in much better server density

**3. 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.

**4. 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.

**Docker’s Registry**

Docker registries are services that provide locations from where you can store and download images. In other words, a Docker registry contains Docker repositories that host one or more Docker Images. Public Registries include two components namely the Docker Hub and Docker Cloud. You can also use Private Registries. The most common commands when working with registries include: docker push, docker pull, docker run

**Docker Command**

Following are the commands which are being covered:

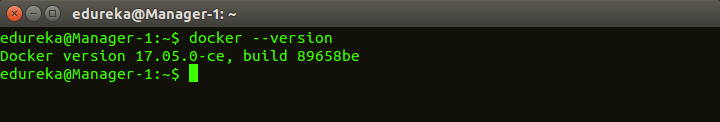
* + docker -version
  + docker pull
  + docker run
  + docker ps
  + docker ps -a
  + docker exec
  + docker stop
  + docker kill
  + docker commit
  + docker login
  + docker push
  + docker images
  + docker rm
  + docker rmi
  + docker build

So, let’s get started:

**Docker Commands**

1.**docker –version**

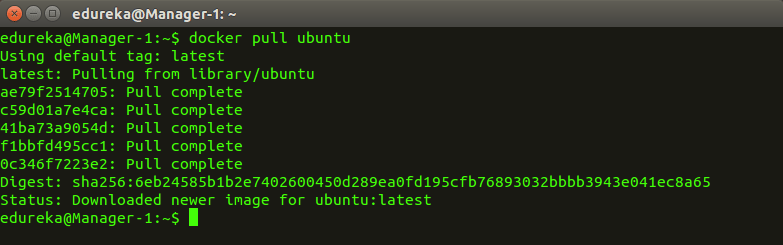
This command is used to get the currently installed version of docker



2.**docker pull**

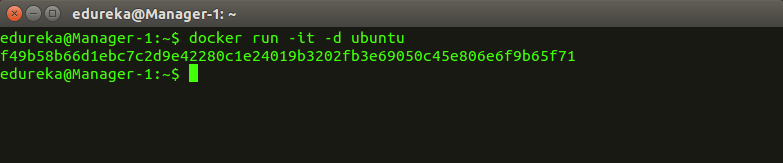
**Usage: docker pull <image name>**

This command is used to pull images from the **docker repository**(hub.docker.com)

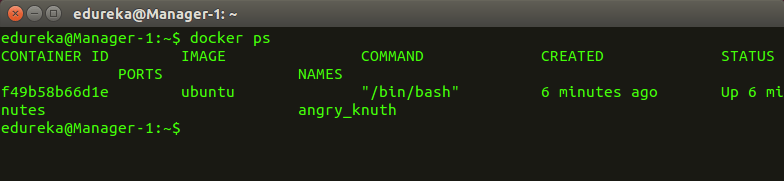
  
  
3. **docker run**

**Usage: docker run -it -d <image name>**

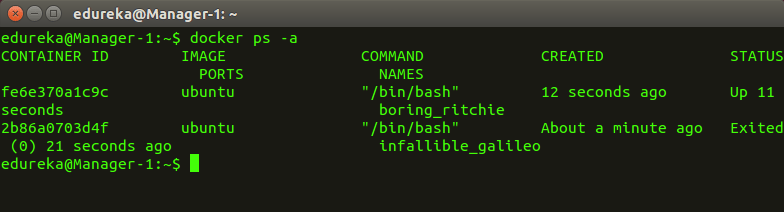
This command is used to create a container from an image

  
  
4. **docker ps**

This command is used to list the running containers

  
  
5. **docker ps -a**

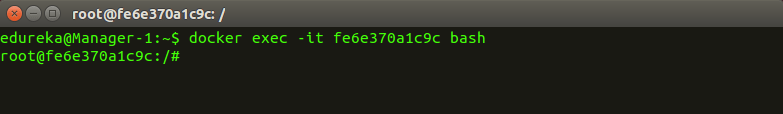
This command is used to show all the running and exited containers



6. **docker exec**

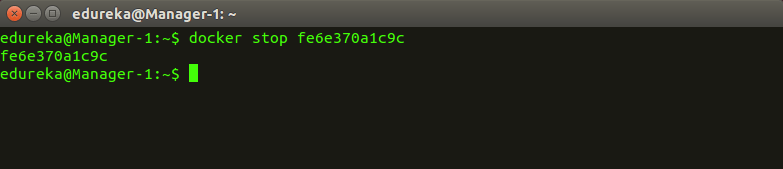
**Usage: docker exec -it <container id> bash**

This command is used to access the running container

  
  
7. **docker stop**

**Usage: docker stop <container id>**

This command stops a running container

  
  
8. **docker kill**

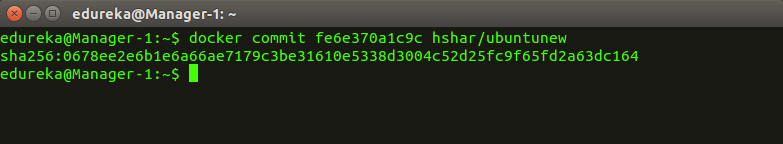
**Usage: docker kill <container id>**

This command kills the container by stopping its execution immediately. The difference between ‘docker kill’ and ‘docker stop’ is that ‘docker stop’ gives the container time to shutdown gracefully, in situations when it is taking too much time for getting the container to stop, one can opt to kill it

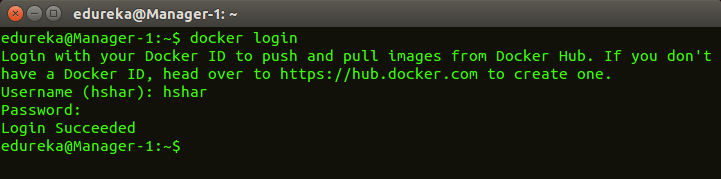
  
  
9. **docker commit**

**Usage: docker commit <conatainer id> <username/imagename>**

This command creates a new image of an edited container on the local system



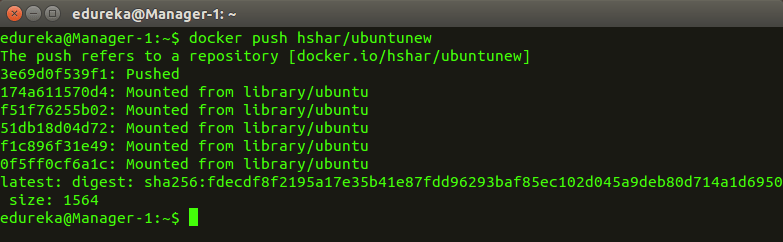
10. **docker login**



11.**docker push**

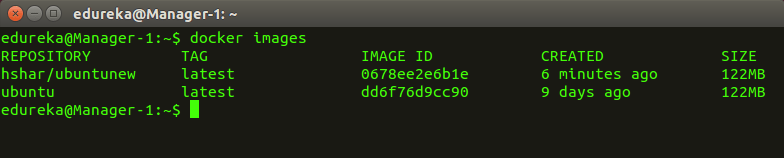
**Usage: docker push <username/image name>**

This command is used to push an image to the docker hub repository



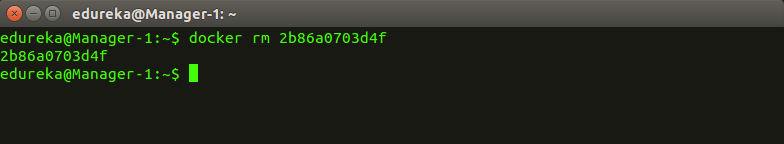
12. **docker images**

This command lists all the locally stored docker images

  
  
13. **docker rm**

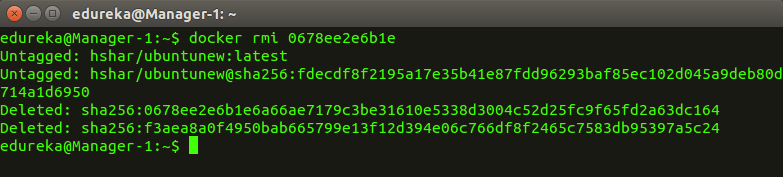
**Usage: docker rm <container id>**

This command is used to delete a stopped container

  
  
14. **docker rmi**

**Usage: docker rmi <image-id>**

This command is used to delete an image from local storage

  
  
15. **docker build**

**Usage: docker build <path to docker file>**

This command is used to build an image from a specified docker file

