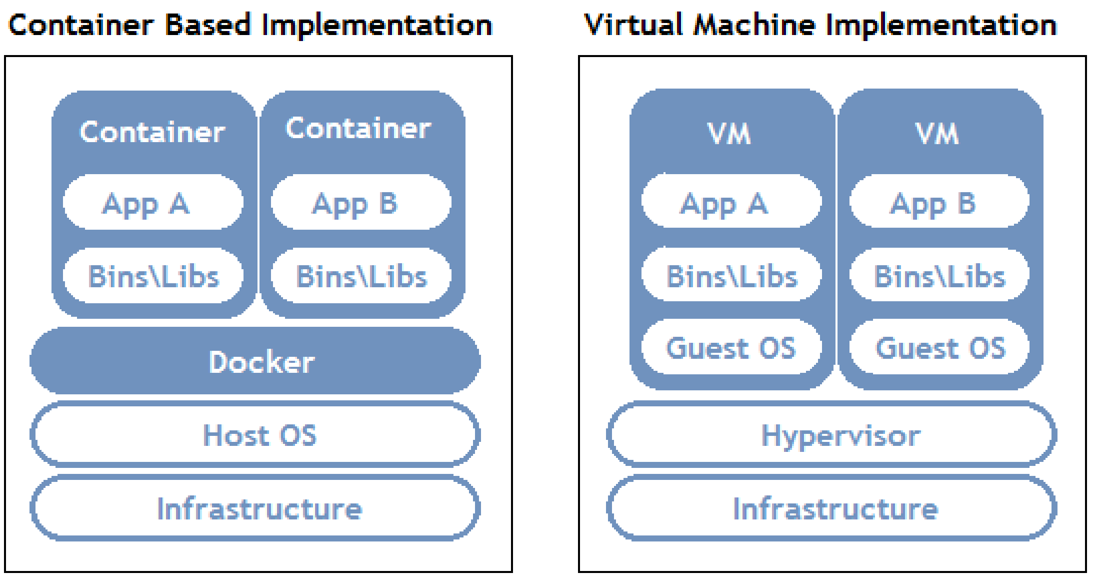
[redacted]

To master Docker you need to start with a clear understanding of its architecture, and how each component of the Docker system interacts with the others. Let's look at Docker and its architecture and its various components in detail. Let us first compare containers to their closest cousin – Virtual Machines.

Containers vs. Virtual Machines

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 much faster and more lightweight instances .



Advantages

The main advantages of Docker are:

Resource Efficiency :

Process level isolation and usage of the container host's kernel is more efficient when compared to virtualizing an entire hardware server.

[redacted]

[redacted]

Portability :

All the dependencies for an application are bundled in the container. This means they can be easily moved between development, test, and production environments.

Continuous Deployment and Testing :

The ability to have consistent environments and flexibility with patching has made Docker a great choice for teams that want to move from waterfall to the modern DevOps approach to software delivery.

The Docker Engine

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:

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.

Docker Engine REST API :

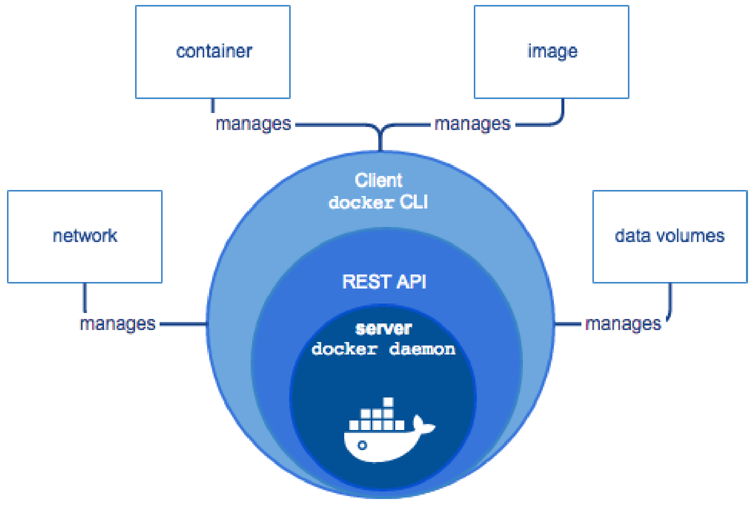
An API used by applications to interact with the Docker daemon; it can be accessed by an HTTP client.

Docker CLI :

A command line interface client for interacting with the Docker daemon. It greatly simplifies how you manage container instances and is one of the key reasons why developers love using Docker.

[redacted]

[redacted]



We will see how the different components of the Docker Engine are used, let us dive a little deeper into the architecture.

Implementation
Docker is available for implementation across a wide range of platforms:

Desktop :

Mac OS, Windows 10.

Server :

Various Linux distributions and Windows Server 2016.

Cloud :

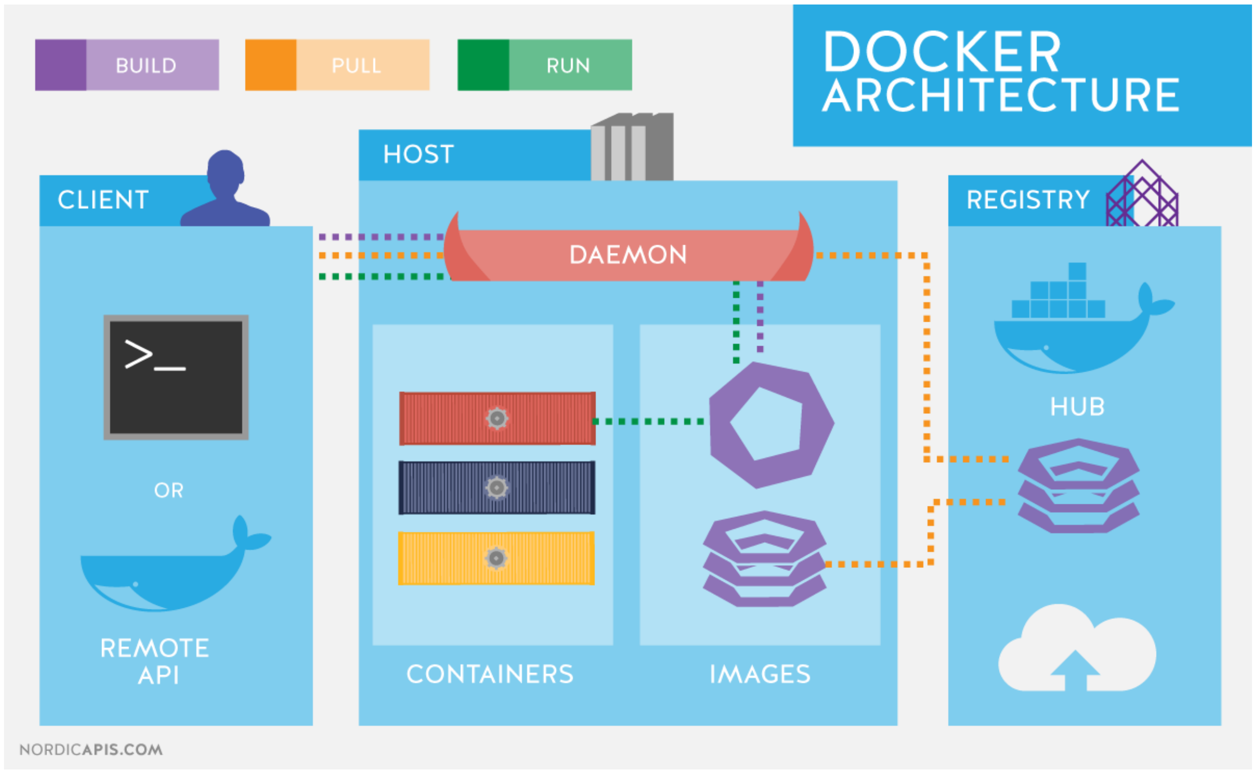
Amazon Web Services, Google Compute Platform, Microsoft Azure, IBM Cloud, and more.

Docker Architecture :

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

[redacted]

[redacted]



Docker Client :

The Docker client enables users to interact with Docker. 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 Docker client provides a command line interface (CLI) that allows you to issue build, run, and stop application commands to a Docker 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. Common commands issued by a client are:

docker build
docker pull
docker run

[redacted]

[redacted]

DockerHost :

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. 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.

Docker Objects :

Various objects are used in the assembling of your application. The main requisite Docker objects are:

Images :

Images are a read-only binary template used to build containers. Images 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. Container images can be shared across teams within an enterprise using a private container registry, or shared with the world using a public registry like Docker Hub. Images are a core part of the Docker experience as they enable collaboration between developers in a way that was not possible before.

Containers :

Containers are encapsulated environments in which you run applications. The 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

[redacted]

[redacted]

much smaller than VMs, they can be spun up in a matter of seconds, and result in much better server density.

#docker login **(Login to www.hub.docker.com)**

#docker images **(Lists all the images from the local cache)**

Pulling the Image :

Go to www.hub.docker.com → Repositories → Explore Repositories → Search → ubuntu → Select Ubuntu → Details →docker pull ubuntu (Copy)

#docker pull ubuntu **(Execute this command on docker host to pull ubuntu image to local cache)**

#docker images

#docker images –q **(Lists docker images with Image IDS)**

#docker rmi   < Image ID > **(Deleting a specific Image)**

#docker ps **(Lists all the active containers)**

#docker ps –a **(Lists all the active & Inactive Containers)**

#docker run –it ubuntu **(Creating ubuntu container with interactive Shell)**

#docker start <container Name / Container ID > **(To start a Container)**

#docker stop <Container Name / Container ID > (**To stop a container)**

#docker attach < Containername / Container ID> **(Brings the container from background to Foreground)**

#docker stats **(Displays information about Running Containers like CPU, Memory Utilization and along with Network / Disk IO statistics)**

#docker system df **(Displays disk usage of Docker)**

[redacted]

[redacted]

#docker system prune –a **(This command removes all the stopped containers, Deletes all the networks not associated to any container, Deletes all the dangling Images)**

#docker pull ubuntu:18.04 **(Downloading docker image of a specific tag)**

#docker images

#docker run –name myubuntu –it ubuntu bash **(Creating container with custom name)**

#docker inspect <image name / Image ID> **(Displays detailed information of an image)**

#docker stop <Contianer Name / Container ID> **(Stopping a container)**

#docker rm < Container Name / Container ID > **(Deleting Container)**

[redacted]