WEEK 12 – Session 5

CONTAINERS

Introduction

Let's assume that you are building a web service on a Ubuntu machine. Your code works fine on your local machine. You have a remote server in your data centre that can run your application.

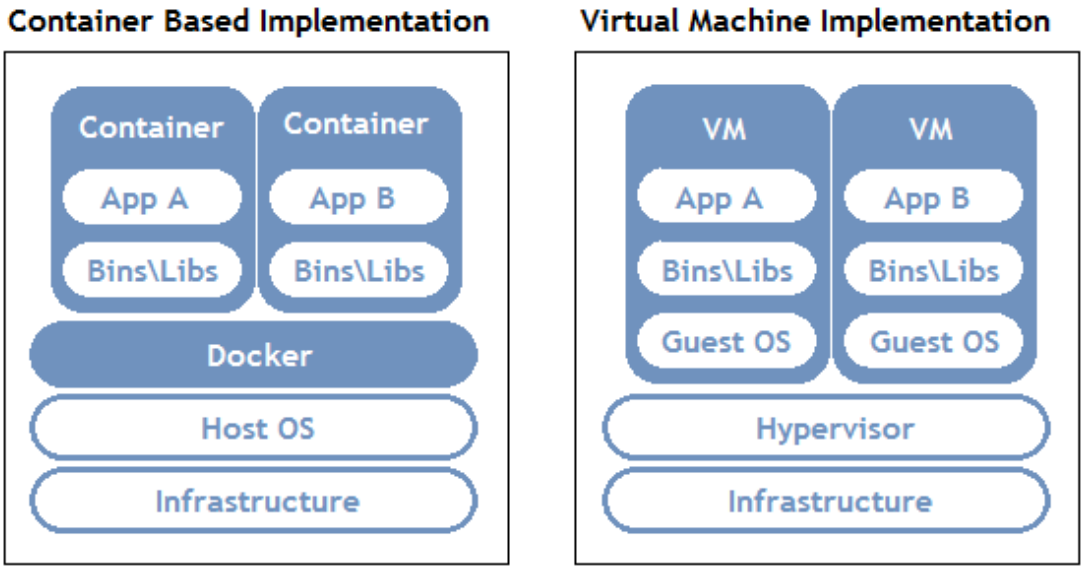
You copy your local binaries on the remote server and try to run your code. The next thing that you see is your code doesn't work there.

There might be multiple reasons for the above failure. The OS running on the server may be different from your local computer. As a result, it may not have the binaries and files required to run your application. Another reason may be incompatible software installed on the remote server. For eg: a different version of python or java interpreter used to launch the application.

The above problems result in portability issues. The developer has to spend a lot of time debugging the environment-specific issues. Fixing these issues may be time-consuming and it's often hated by most developers. Hence, it's essential to find a solution to these problems.

.

The fundamental difference between containers and Virtual Machines is that containers don't contain a hardware hypervisor.



Why containers?

Containers are packages of software that contain all of the necessary elements to run in any environment.

Containerization is a process of software deployment that bundles an application's code with all the files and libraries it needs to run on any infrastructure.

Reasons to Adopt Containers

Portability – Ability to Run Anywhere

Resource Efficiency and Density

Container Isolation and Resource Sharing

Speed: Start, Create, Replicate or Destroy Containers in Seconds

High Scalability

Improved Developer Productivity

What is a Docker?

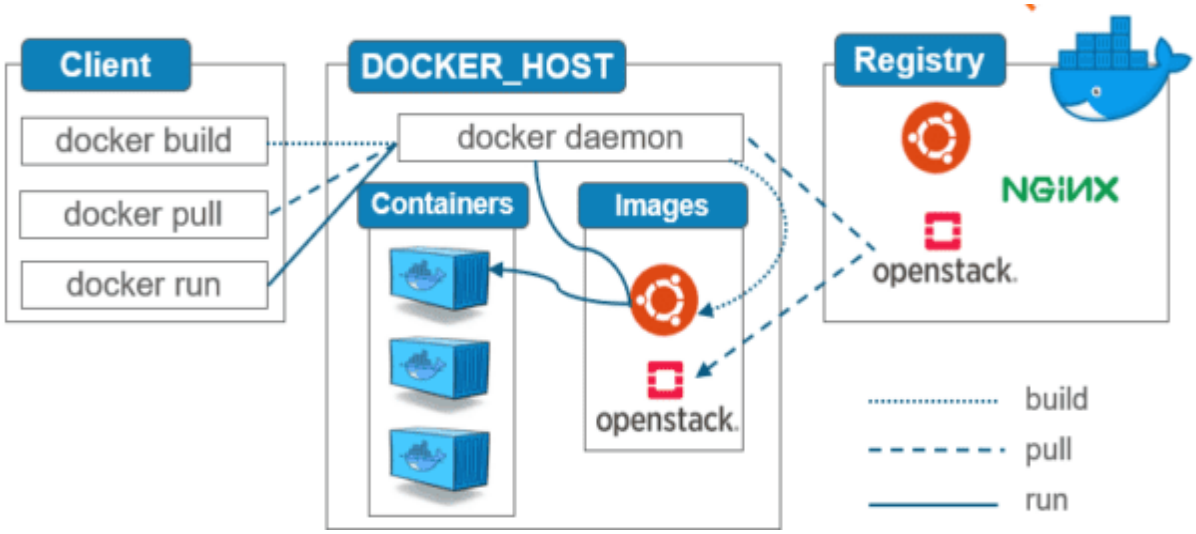
Docker is a technology that allows you to incorporate and store your code and its dependencies into a neat little package – an image. This image can then be used to spawn an instance of your application – a container.

Docker Architecture

To allow for an application to be self-contained the Docker approach moves up the abstraction of resources from the hardware level to the Operating System level.

To further understand Docker, let us look at its architecture. It uses a client-server model and comprises of the following **components**:

- **Docker client**
- **Docker daemon**
- **Docker image**
- **Docker container**
- **Docker registry**



Docker architecture

How Docker works?

We have seen the core building blocks of Docker.

Now let's understand the Docker workflow using the Docker components.

- **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. Docker client uses commands and REST APIs to communicate with the Docker Daemon.

- **Docker daemon**

Docker has a client-server architecture. Docker Daemon (dockerd) or server is responsible for all the actions related to containers.

The daemon receives the commands from the Docker client through CLI or REST API. Docker client can be on the same host as a daemon or present on any other host.

- **Docker image**

  Images are the basic building blocks of Docker. It contains the OS libraries, dependencies, and tools to run an application.

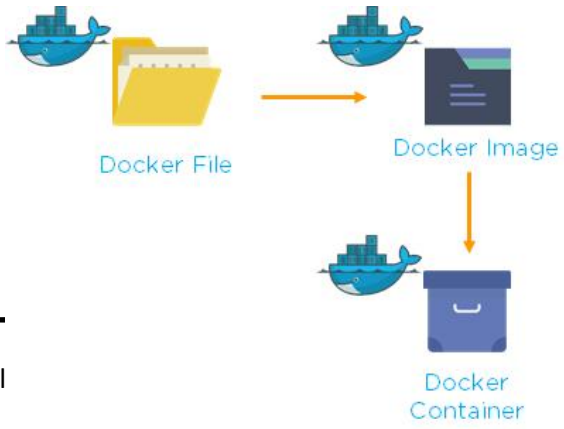
  Docker has a concept of Dockerfile that is used for building the image. A Dockerfile is a text file that contains one command (instructions) per line.

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

- **Docker container**

  After you run a docker image, it creates a docker container. All the applications and their environment run inside this container.

  You can use Docker API or CLI to start, stop, delete a docker container.



- **Docker registry**

  It is a repository (storage) for Docker images. A registry can be public or private.

  Docker Hub is hosting registry service on the cloud that allows you to upload and download images from a central location.