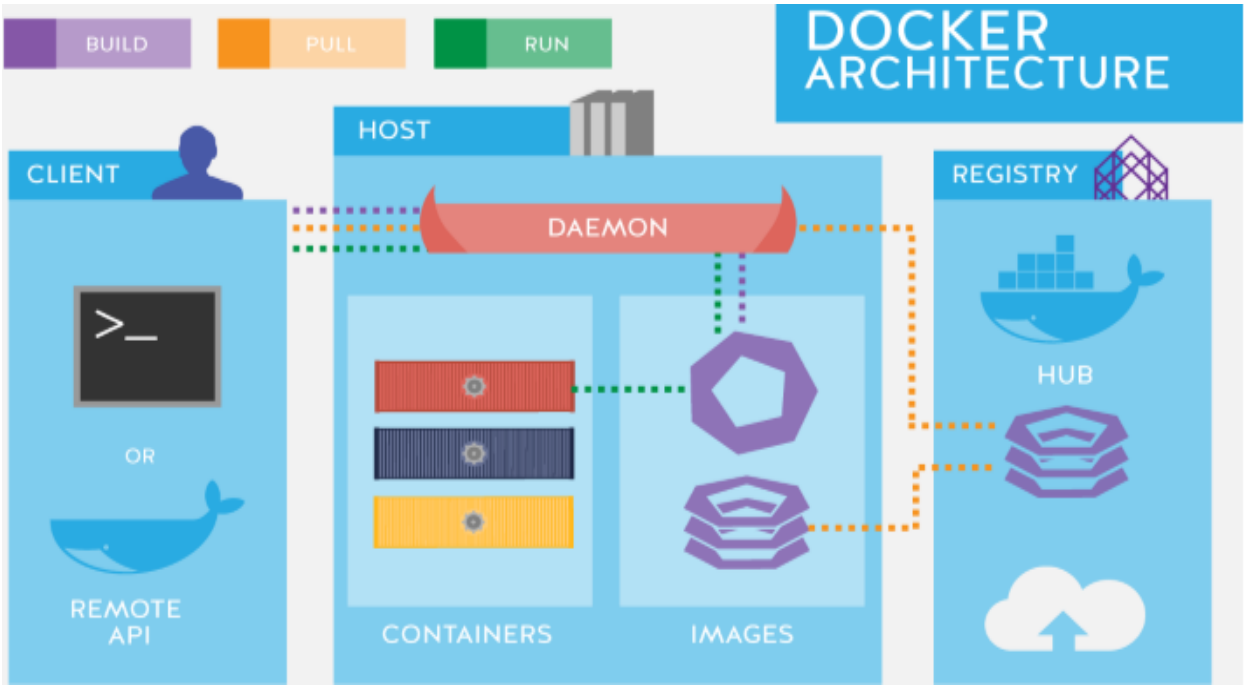
Basic Introduction of Docker:

Introduction Containers allow the packaging of your application (and everything that you need to run it) in a "container image". Inside a container you can include a base operational system, libraries, files and folders, environment variables, volumes mount-points, and the application binaries.

A "container image" is a template for the execution of a container. It means that you can have multiple containers running from the same image, all sharing the same behavior, which promotes the scaling and distribution of the application. These images can be stored in a remote registry to ease the distribution.

Once a container is created, the execution is managed by the "Docker Engine" aka "Docker Daemon". You can interact with the the Docker Engine through the "docker" command. These three primary components of Docker (client, engine and registry) are diagrammed below:



Container related commands

Create & Run a Container in interactive mode

$ docker run --name <name> -it <Image> <FirstCMD>

- Download the Image
- create a new container (ID) using the Image
- start the container
- attach to the container

Start Container

$ docker start <ContainerID|Name>

Stop Container

$ docker stop <ContainerID|Name

Attach to a Running Container

$ docker attach <ContainerID|Name>

Delete Container

$ docker rm <ContainerID|Name>

Create & Run a Container in detached mode

$ docker run --name <name> -d <Image> <FirstCMD>

Print Logs of a Container

$ docker logs <ContainerID|Name>

$ docker logs –f <ContainerID|Name> # Live feed of the containers output

List Containers

$ docker ps #List of only Active containers

$ docker ps –a #List all containers

Execute a cmd/process inside a running Container without attaching to the terminal

$ docker exec <ContainerID|Name> <CMD>

$ docker exec –it <ContainerID|Name> /bin/bash #Executes and access bash

Inspecting the container

$ docker inspect <ContainerID|Name>

Inspecting the container's process

$ docker top <ContainerID|Name>

Binding Ports:

Docker containers can connect to the outside world without further configuration, but the outside world (access over browser) cannot connect to Docker containers by default.

- In Docker, the containers themselves can have applications running on ports.
- When you run a container, if you want to access the application in the container via a port number, you need to map the port number of the container to the port number of the Docker host.

Create a Container & map port to host machine

$ docker run --name <name> -d –p <HostPort>:<ContainerPort> <Image> <FirstCMD>

Create a Container & map all the ports to host machine

$ docker run --name <name> -d –P <Image> <FirstCMD>

Volumes:

A volume is a specially designated directory within one or more containers that bypasses the Union File System

- Volumes can be shared and reused between containers.
- A container doesn't have to be running to share its volumes.
- Changes to a volume are made directly.
- Changes to a volume will not be included when you update an image
- Volumes persist until no containers use them

Create a Container with Volume

$ docker run --name <name> -it –v <ContainerDir> <Image>

Create a Container with Volume & map to host machine

$ docker run --name <name> -it –v <HostDir>:<ContainerDir> <Image>

Create a Container and access volumes from another Container

$ docker run --name <name> -it --volumes-from <ContainerID> --privileged=true <Image>

Images related commands

Download Image

$ docker pull <Image>

Upload Image

$ docker push <Image>

List Images

$ docker images

Delete Image

$ docker rmi <ImageID|Name>

Dockerfile

A Docker File is a simple text file with list of instructions on how to build your images automatically.

Step 1 − Create a file called Docker File and edit it using vim. Please note that the name of the file must be "Dockerfile" with "D" as capital.

Step 2 – Build your Docker File using the following instructions.

Instructions	**Arguments**
FROM	Sets the Base image for subsequent instructions
MAINTAINER	Sets the author field of the generated images
RUN	Executes commands inside the container
CMD	Default first command to execute
ENV	Sets an environment variable
ENTRYPOINT	Allows you to configure a container that will run as an exec
COPY	Copies new files or directories into the filesystem of the container
ADD	Copies new files, directories or remote file URLs into the filesystem of the container
USER	Sets the username or UID to use when running an image
WORKDIR	Sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD commands
EXPOSE	Informs Docker that the container listens on the specified network port at runtime.
VOLUME	Creates a mount point and marks it as holding externally mounted volumes from native host or other containers

Step 3 - Build Image using Dockerfile

$ docker build –t <ImageName> # pick a different file other than default Dockerfile

$ docker build –t <ImageName> -f <Dockerfilepath>

Example Dockerfile we had in our class,

```
FROM centos:latest
MAINTAINER Vignesh
RUN yum install httpd -y
COPY application.html /var/www/html
CMD ["/usr/sbin/httpd", "-D", "FOREGROUND"]
EXPOSE 80
```

Docker-compose

Docker compose installation

- curl -L "https://github.com/docker/compose/releases/download/1.26.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
- chmod +x /usr/local/bin/docker-compose
- ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose

Docker-compose syntax

Docker-compose will be written in yml file

```
version
        services:
          service 1
                DB installation
                Varibales
          service 2
                wordpress installation
                to consume the database
```