Introduction to Docker

Docker is a containerization engine. It helps in containerizing your applications, along with the environments.

What are Containers?

- A way to package your application and environments.
- A way to ship your application with ease.
- A way to isolate resources in a shared system.

Containers have been around much before Docker. OpenVZ and LXC projects came along about half a decade before Docker. So all Docker containers and containers but not all containers are Docker containers.

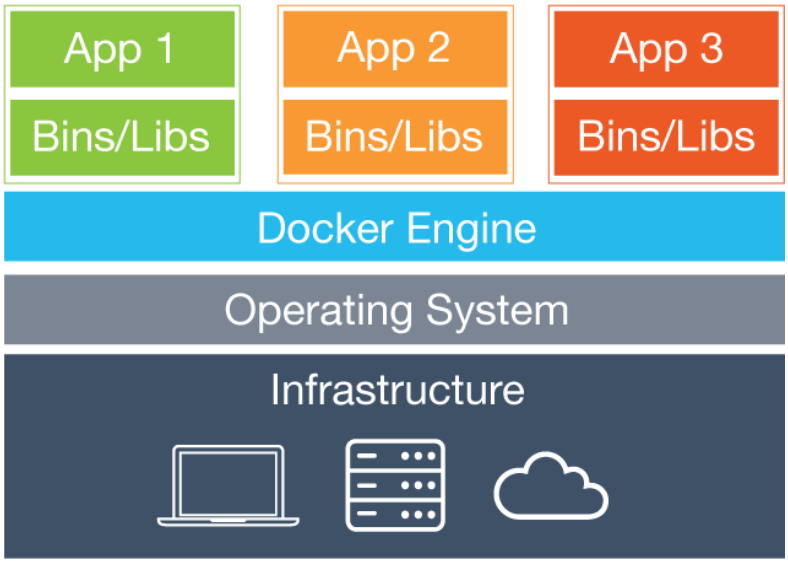
What is Docker?

Official definition of Docker

Docker allows you to package an application with all of its dependencies into a standardized unit for software development.

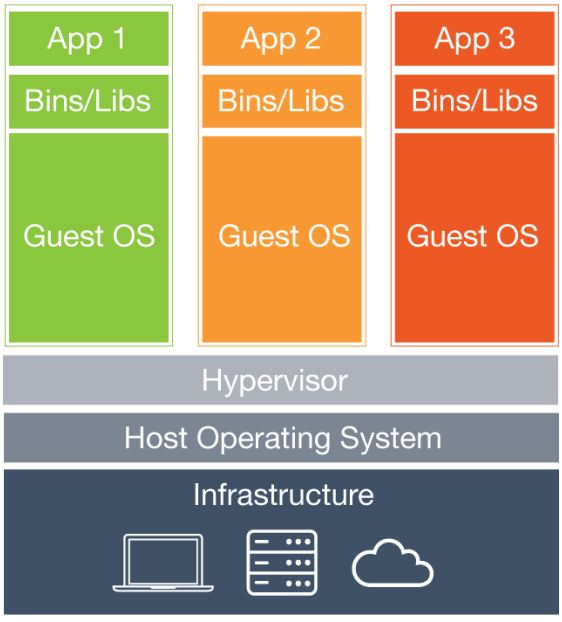
How does a Docker containers looks like?

- Docker runs on host operating system.
- The kernel is shared among all the containers but the resources are in appropriate namespaces and cgroups.
- Typical container consists of operating system libraries and the application code, much of which is shared.
- No emulation of hardware.



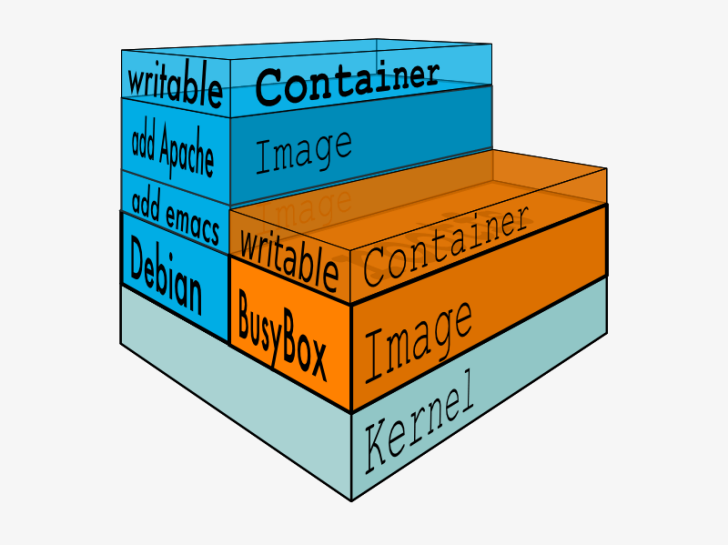
How is this different that a Virtual machine?

- Additional hypervisor layer.
- Each guest OS comes with its own Kernel and libraries.
- Hardware is emulated.
- Much more resource intensive.
- Difficult to share a virtual machine.



Basic Docker Terminologies

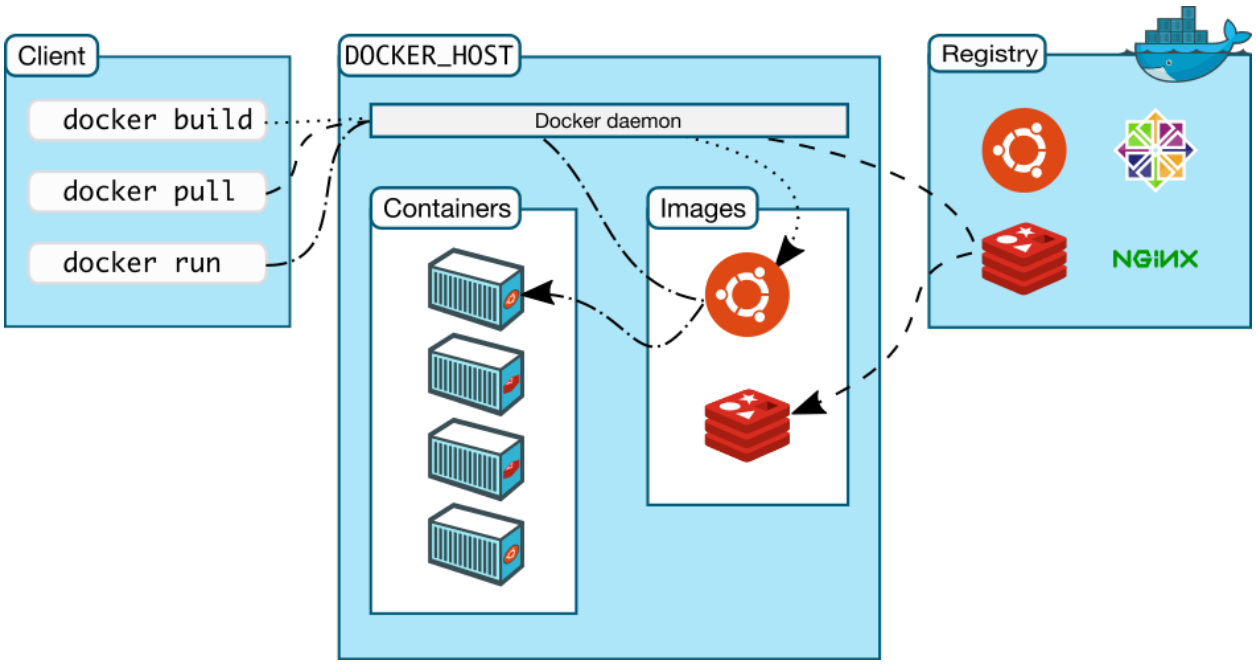
- Docker image Docker images are *read-only* frozen templates which may consists of system libraries, application code and dependencies.
- Docker container A *stateful and running instance* of a Docker image is called a Docker container. Another way to look at it is that a running Docker container is a Docker image with a writable layer on top of it.



- Docker registry A Docker registry is a collection or repository of Docker images. They can be hosted publicly or privately in on-premise setup.
- Docker Hub The default Docker registry operated by Docker Inc. is called Docker Hub.

How this all works together?

A typical Docker workflow looks like this.



Docker client gives a call to Docker host to run a container from a particular image. If Docker host has that image, it run a container from it. If it does not have the image, it looks for it in the registry and pulls it on the host and then runs a container from it. Note that any of these three components can be on the same machine or on different machines.

How to install Docker?

Installing Docker is easy on most of the popular Linux based operating systems. Docker itself provides deb and rpm repositories to do so. A full list of supported Linux flavors and respective repository settings can be found at Docker documentation. However, if you are a Mac OS X or Windows user, things will become a little tricky for you since Docker needs some of the Linux Kernel's functionalities to run. A simple way to run Docker on your Mac or Windows it to enable virtualization and then using Docker Toolbox.

For the training session, we would use CentOS Linux to run Docker engine. CentOS is one of the market leaders for providing enterprise grade Linux distribution and is an ideal choice to run on production.

Step 1: Add the repository

```
$ sudo vim /etc/yum.repos.d/dockerrepo.repo
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/$releasever/
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
```

Step 2: Install Docker engine

```
$ sudo yum install docker-engine
```

Step 3: Start the Docker service

```
$ sudo systemctl start docker
```

In three simple steps, we installed Docker on our machine. Let us verify that our Docker setup is actually running

```
$ sudo docker ps
```

Since we have just installed Docker, the above command will show that there are no containers running at the moment. If we do not see any error, then we can assume that our setup was successful.

Basic Docker commands

Here is a list of ten basic and easiest Docker commands. This will help you in getting started with Docker

- List all running containers

```
$ sudo docker ps
```

- List all running and stopped containers

```
$ sudo docker ps -a
```

- List all Docker images

```
$ sudo docker images
```

- Pull a Docker image

```
$ sudo docker pull alpine
```

- Run a Docker container

```
$ sudo docker run -i -t alpine /bin/sh
```

- Check logs of a Docker container

```
$ sudo docker logs <container_id>
```

- Run a command in a running Docker container

```
$ sudo docker exec -i -t <container_id> /bin/sh
```

- Stop a Docker container

```
$ sudo docker stop <container_id>
```

- Delete a Docker container

```
$ sudo docker rm <container_id>
```

- Delete a Docker image

```
$ sudo docker rmi <image_id>
```

Bonus command

```
$ sudo docker --help
```

Docker Images

Docker images are the read-only templates from which a container is created. In previous sections, we have seen how to pull and list the docker images. Let us learn how to build them.

Building Docker images by using docker commit

A lot of Docker's command line sub-commands are inspired by Git, including this one. Docker provides a very simple way to create images. The workflow goes like this:

- Create a container from a Docker image
- Make required changes, like installing a webserver or adding a user
- On command line execute the following

```
$ sudo docker commit <container_id> <optional_tag>
```

While this presents a very simple way to create images, this is not a good way to do so. Creating images this way is not very reproducible and hence not recommended.

Building Docker images by using a Dockerfile

Docker provides an easy way to build images from a description file. This file is known as Dockerfile. A simple Dockerfile will look like this:

```
FROM centos

RUN yum -y update && yum clean all
```

```
RUN yum -y install httpd
EXPOSE 80
CMD ["/usr/sbin/httpd", "-D", "FOREGROUND"]
```

Dockerfile supports various commands. I have used a few of them and I am going to describe them below:

- FROM: This defines the base image for the image that would be created
- MAINTAINER: This defines the name and contact of the person that has created the image.
- RUN: This will run the command inside the container.
- EXPOSE: This informs that the specified port is to be bind inside the container.
- CMD: Command to be executed when the container starts.

A comprehensive list and description of all the supported commands can be found in thedocumentation. Let us create a file with the above lines and build an image from this.

```
$ sudo docker build -f <path_of_dockerfile> .
```

Images created this way document the steps involved clearly and hence using Dockerfile is a good way to build reproducible images. It is also worth nothing that each line in Dockerfile create a new layer in Docker image. So often, we will club statements using and operator (&&) like this:

```
RUN yum -y update && yum clean all
```

Docker Registry

Docker registry is a repository of Docker images. It enables users to share images publicly and privately. Docker registry can be hosted on-premise, and can be password protected which makes it a great tool for any corporate environment where security and privacy are important.

How to setup Docker Registry?

Docker has containerize the registry which makes the installation extremely easy. At the moment, registry:2 is the latest registry. So let us pull the image first

```
$ sudo docker pull registry:2
```

As and when we push images to the registry, registry will store the data. We want to ensure that our data is safe, even if the container running registry dies. The easiest way to achieve that is to mount a directory from the host to the container which will store the data. So let us create a directory.

```
$ sudo mkdir /opt/registry-data
```

Now will mount this directory inside the registry container.

```
$ sudo docker run -d -p 5000:5000 -v /opt/registry-data:/var/lib/registry
registry:2
```

Let us verify that the container is actually running

```
$ sudo docker ps
```

Once we confirm that the container is running, we should tag an image and try to push it.

```
$ sudo docker tag <image_id> localhost:5000/myimage
$ sudo docker push localhost:5000/myimage
```

Docker Networking

Connecting containers is very important for most of the applications out there. A classic web application consists of at least a web server, an application server and a database server. The web server needs to talk to the application server and the application server would need to talk to database server. A legacy, but popular way to do so is via passing `--link` flag to `docker run` command.

```
$ sudo docker run -d --name mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw mysql
$ sudo docker run --link mysql -i -t centos /bin/bash
```

Currently, Docker has a very elaborated network suite to cater to a wide variety of use-cases. By default it comes with three network pre-configured:

- **none:** Containers in this network have only local interface. They have no external connectivity. So this is ideal for applications that do not need network but may contain sensitive info
- **host:** This network copies the host network. So the containers running in this network basically have identical network as the host.
- **bridge:** This the default network in which Docker will boot up containers. This creates a bridge (docker0 interface) with a gateway and other required network components. Containers running in this network can talk to the other containers in the same network but they cannot talk to to any other containers in any other network.

Please take moment to run a container in all these network and run `ifconfig` or `ip addr`.

```
$ sudo docker run --net=<type_of_the_network> -i -t centos /bin/bash
```

Docker does not limit us to these three networks only. We can create additional networks as and when required. This helps in maintaining isolation between networks and improves security. Docker lets us create two kinds of user-defined networks:

- **Bridge:** It is similar to default bridge network with minor differences, like --link flag is not available for user-defined networks. Since bridge networks are defined per host, all the containers in this network must reside on the same host. Multiple networks on the same host are isolated from each other, however, a container can be a part of multiple network and facilitate inter-network communication. Let us create and run a container in user-defined bridge network.
  - `$ sudo docker network create --driver bridge isolated_nw`

```
$ sudo docker run --net=isolated_nw -i -t centos /bin/bash
```

Try pinging containers running in other networks.

- **Overlay:** Overlay network is Docker's way to simplify multi-host networking. An overlay network can span to several hosts and maintain network level isolation. It uses libnetwork and libkv. To run libkv, we need to install one of the key value stores supported by Docker which are Consul, Etcd, and ZooKeeper. For this session, we will go ahead with etcd. So let us start by installing etcd on all the servers that are going to run the Docker daemon.

```
$ sudo yum -y install etcd
```

Designate one of the servers as etcd master and configure it to listen to the non-local IP address.

```
ETCD_LISTEN_CLIENT_URLS="http://<ip-addr>:2379"
ETCD_ADVERTISE_CLIENT_URLS="http://<ip-addr>:2379"
```

Now restart the etcd service.

```
$ sudo systemctl restart etcd
```

Once our etcd service is up, we can start the Docker daemon with some additional parameters.

```
$ nohup sudo docker daemon  --cluster-store=etcd://<ip-addr>:2379 --
cluster-advertise=eth0:2376 &
```

Now let us create an overlay network, named `overlay-net1` on one of the machines.

```
$ sudo docker network create --driver overlay overlay-net1
```

And then let us check this on the other machine.

```
$ sudo docker network ls
```

This is it! Now any container in the overlay network will be able to communicate to each other regardless of what host it is running on.

Docker Remote Api

Docker provides a rich set of APIs which can be used to do manage containers. This opens up a huge opportunity of creating automation to improve user experience and scalability. For this lab, we will bind the Docker daemon to a TCP port.

```
$ sudo docker daemon -H :1234
```

Let us try doing some basic operations using the Remote API.

- Listing the running containers

```
$ curl http://localhost:1234/containers/json
```

- Pulling a Docker image

```
$ curl -d '' http://localhost:1234/images/create?fromImage=alpine
```

- Creating a container

```
$ curl -H "Content-Type: application/json" -d '{"Image":"alpine",
"Cmd":["sleep", "10000"]}' http://localhost:1234/containers/create
```

- Running a container

```
$ curl -d '' http://localhost:1234/containers/<container_id>/start
```