School of Information Sciences

MANIPAL UNIVERSITY, MANIPAL

Seminar Report

Submitted by

Anand D R

171041004

Computing Technologies and Virtualization

Submitted on

17 September 2017

Table of Content

1. Introduction ………………………………………….. 4
2. Docker in Detail ………………………………………….. 5
3. Practical / Tools related ……………………………………………. 12
4. Future Scope …………………………………………… 14
5. References ……………………………………………. 15

Table of Figures

1. Dockers Model ……………………………………………………. 4

2. Docker Engine ……………………………………………………. 7

3. Docker Architecture ……………………………………………………. 8

4. ngix server page ……………………………………………………. 13

Introduction

Dockers are a container-virtualization technology; it is a lightweight virtual machine. It is also an open platform for building, shipping, and running applications. It enables you to separate applications from infrastructure so that it can be delivered easily and quickly.

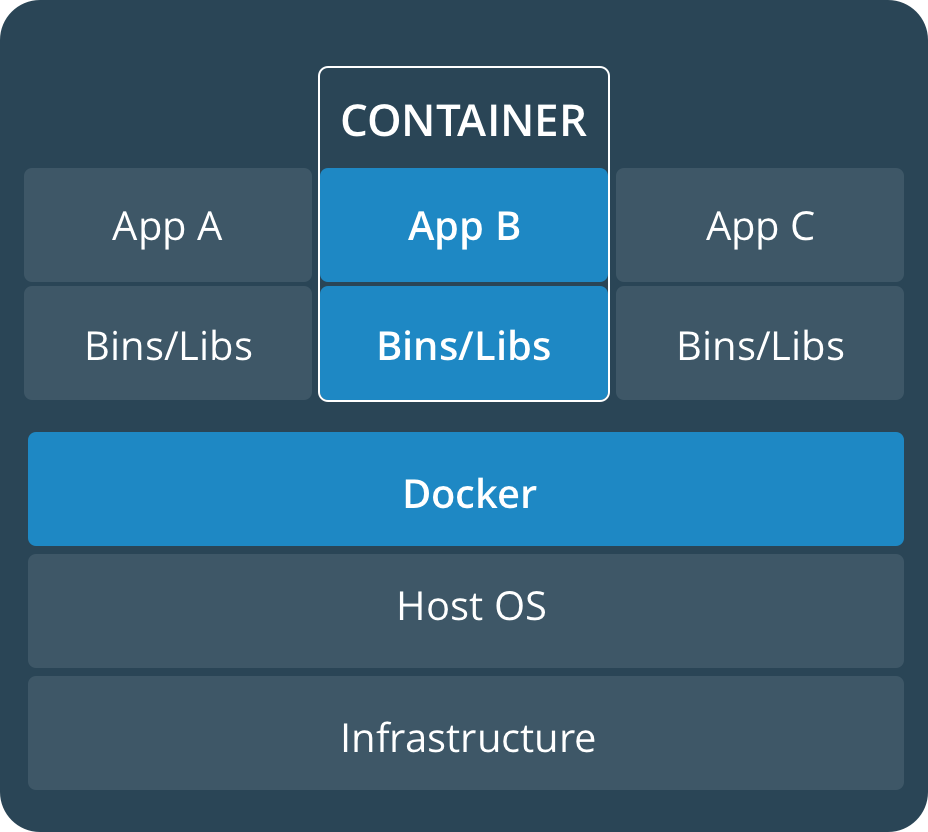


Figure 1: Dockers Model

Dockers are lightweight as they don't require an extra load of the hypervisor as in virtual machines, they run directly within the host machines kernel. And we can run Dockers inside a virtual machine as well. Dockers are loaded with required resources and can be ported all the way to develop test and then be able to be instantiated in production also.

Dockers are great for continuous integration and continuous development work flows. It can be used for running more workloads on the same hardware. And these Dockers can be built much faster and simpler when compared to virtual machines. There is another advantage of Dockers when we take capacity into consideration; Dockers have low capacity when compared to a virtual machine.

Docker in detail

Dockers are an open platform for developing, shipping along with other dependencies and running application. It separates your application from your infrastructure so that you can deliver your product as quickly as possible. By using Dockers you can significantly reduce the delay between writing code and running application in production.

Dockers provide the ability to package and run an application in a loosely isolated environment called container. The features like isolation and security allow you to run many applications simultaneously. Containers don't have an extra load of hypervisor hence they are lightweight. These Dockers can be deployed inside virtual machines also.

Dockers fall under OS-level virtualization in which host operating system will be at the base (that may be windows or Linux) and the virtualization layers are on top of it and run as an application within the operating system. This virtualization layer offers a file system and kernel service abstraction layer which isolates resources from these containers and it ensures that these containers appear as a standalone server. These containers or Dockers doesn't have overhead in running a completely installed operating system; the advantage of this is there is no need to duplicate functionality like hardware calls as there is only one operating system to take care of all hardware access functionalities.

**BENEFITS OF DOCKER-CONTAINERS**

1. **Application Portability**

Dockers add all dependencies that the application needs along with it inside the containers so that it can use on different platforms. Application developers, administrators all of them can use same application in their laptops, VMs or cloud.

1. **Lightweight and fast**

When compared to virtual machines, containers are lightweight and fast since there is no operating system. Starting a container is same as starting a process.

1. **Resource utilization is optimal**

Dockers use Linux's control group which allows allocating memory, CPU, memory, network and disk resources. It makes sure that no process will consume all the resources so that other process waits in the queue for resource allocation.

1. **Best fit for Micro services architecture**

Each micro service can be deployed into containers without interfering with other micro services since containers support micro service architecture.

1. **Resource Utilization**

Since containers are lightweight, portable, efficient so that we can run more containers in a server when compared to virtual machines which result in higher resource utilization. Containers contain application code along with dependencies which run isolated and share the kernel with other containers in user space on the host operating system.

**DOCKER ENGINE**

Docker engine is basically a client application with these components.

* Daemon process is a server which is a long-running program (the *dockerd* command).These creates and manages Docker objects, such as images, containers, volumes and networks.
* Above that is the REST API which provides the interface for the programs that can used to instruct daemon what to do.
* A command line interface (CLI) client, i.e., *docker* command.

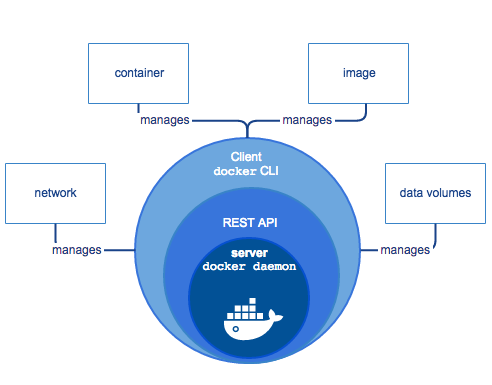
****

Figure 2: Docker Engine.

**DOCKER ARCHITECTURE**

Dockers leverage Linux containers which has certain features like *cgroups* and *namespaces* for resource control and process isolation. Linux containers (LXC) are OS-level virtualization which runs multiple isolated Linux systems on single Linux control host.

Mainly it’s client-server architecture. The Docker client and daemon communicate using RESR API, or UNIX sockets or a network interface. Docker daemon is the component which does building, running and distributing of docker containers. Both docker daemon and client can run on same system, or can connect a client to a remote Docker daemon.

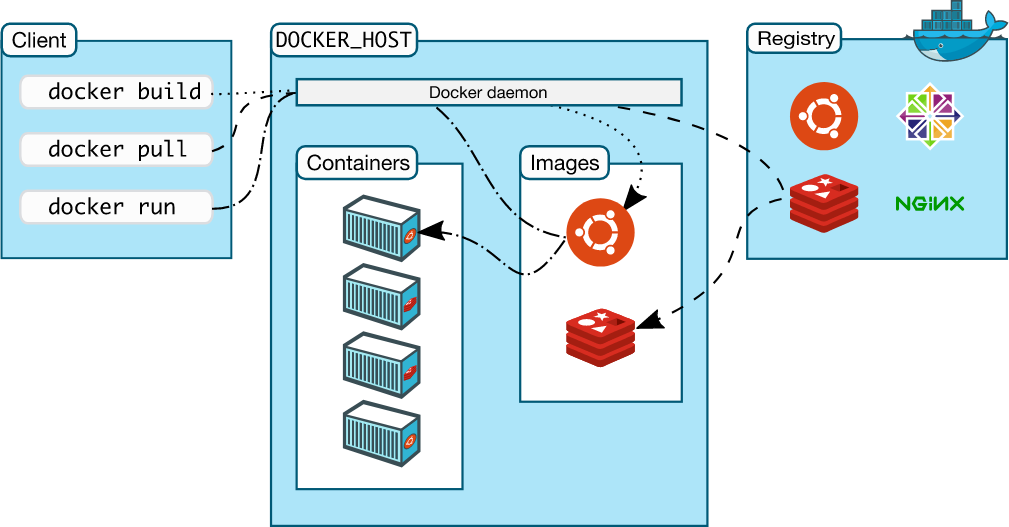


Figure : Docker Architecture

**The Docker Daemon**

This creates and manages Docker objects such as images, containers, networks and volumes. It listens to Docker API and performs the above-specified operations. These daemons communicate with each other to manage Docker services.

**The Docker Client**

This is the primary way that Docker users interact with Docker. When we execute commands such as *docker run*, the client sends these commands to daemon i.e., which carries them out. The docker command uses Docker API. These clients can communicate with more than one daemon.

**Docker registers**

These registers store Docker images. There are many public registers like Docker Hub and Docker Cloud, and Docker is configured to check for images on Docker Hub by default.

When we use *docker pull* or *docker run* command, the required images are pulled from configured registery. When we use *docker push* command, the image is pushed to configured registry.

**Docker Objects**

In Dockers, we create and use images, containers, networks, volumes, plugins and other objects. This section is brief overview of some of those objects.

**IMAGES**

It's a read-only template with instructions for creating a Docker container. For example we may build an image which installs Linux along with Apache web server and your application and configuration details required for running our application.

We can create our own image or use images that are created by others and published in a registry. To build our own image we need create Dockerfile with a simple syntax for defining steps the steps needed to create an image and run it. Each instruction in dockerfile creates a layer in the container, when we change the dockerfile and rebuild it then only those layers which have been changed are rebuilt. This functionality is which makes Dockers lightweight, small and fast.

**CONTAINERS**

It's a runnable instance of an image. We can create, run, stop, move or delete a container using the Docker API or CLI. We can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, containers are well isolated from other containers and also from its host machine. We can control the all above aspects.

**Example *docker run* command**

*$ docker run –i –t Ubuntu /bin/bash*

When we run this command, the following happens.

1. If an Ubuntu image is not present locally, Docker pulls it out from our configured registry, as though we had run *docker pull* Ubuntu manually.
2. Docker creates a new container, as though we had executed *docker create* command manually.
3. Docker allocates a read-write file system as its final layer, this allows the running container to create, delete, copy, move and truncate files and directories in its local file system.
4. Docker creates a network interface to connect a container to the network. This includes assigning the IP to container. By default these containers can connect to network with the help of host’s network connection.
5. Docker starts the container and executes */bin/bash*. Because the container is attached to terminal we can provide input using keyboard and output is logged to our terminal.
6. When we pass exit command to terminate /bin/bash, container stops but it’s not removed.

**SERVICES**

This allows you to scale containers across multiple Docker daemons, which all work together as swarm with multiple *managers* and *workers*. Since each member of swarm is daemons, all of them can communicate using Docker API. Service allows you to define the states, such as number of copies of the service that must be available at the given point of time. By default its load balanced across all worker nodes.

**The underlying technology**

Docker platform is written is Go and it takes advantage of several features of Linux kernel for its functionality.

**Namespaces**

Docker uses technology called namespaces to provide the isolated workspaces called containers. When we run a container, Docker creates a couple of namespaces for that container.

Each container runs in separate namespaces and its access is limited to that namespace.

Some of the namespaces are:

* pid namespace : Process isolation.
* net namespace: Managing network interfaces.
* ipc namespace: Managing access to IPC resources.(ipc: inter-process communication).
* mnt namespace: Managing file system mount points.
* uts namespace: Isolating kernel and version identifiers.

**Control groups**

Docker engine relies on another technology called control groups (*cgroups*).It confines the application to set of resources. For example; you can limit the memory available for specific container.

**Union File systems**

These file system operate by creating layers, making them light weight and fast. These file systems are used by dockers to provide building blocks for containers. Some of the variants of UnionFS are AUFS, btrfs, vfs and DeviceMapper.

**Container format**

Namespaces, control groups and UnionFS are combined into a wrapper called a container format. The default container format used by Docker is *libcontainer*. Docker may support other containers formats in the future by integrating with technologies such as BSD jails or Solaries Zones.

Practical’s /Tools related

Installing the Docker CE

1. Update the apt package index.

*sudo apt-get update*

1. Install the latest version of Docker CE.

*suod spt-get install docker-ce*

In this example we are building the nginx proxy server.

FROM Ubuntu

MAINTAINER Anand Rao "dranandrao@yahoo.com"

RUN apt-get update

RUN apt-get install -y nginx

COPY index.html /usr/share/nginx/html/

ENTRYPOINT ["/usr/share/nginx","-g","daemon off;"]

EXPOSE 80

**Set of commands used in Dockerfile**

1. **FROM**

This command should be mandatorily should be the first command in in the dockerfile. This specifies the base image to use in the build.

1. **MAINTAINER**

An optional value for maintainer of the script.

1. **RUN**

This instruction is used to execute commands. In the above example we are using to update and then installing nginx.

1. **ENTRYPOINT**

This is used for running the nginx executable.

1. **EXPOSE**

This is used to inform what port the container will be listening on.

1. **COPY**

This command will copy the files/folders from the source to the destination in the container.

If we build the image and run the container as follows

*docker build .*

*docker run –d –p 80:80 -- name webserver myimage*

You will find that nginx is started on port 80.And if we visit the host IP, you will see the following.

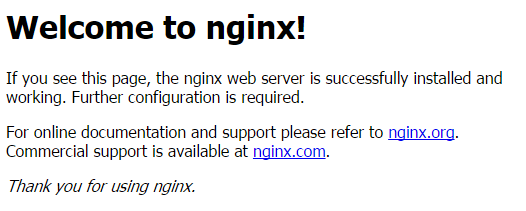


Figure 4:nginx server index page.

Future Scope

Docker basically provides many advantages for DevOps such as sharing containers. It provides DockerHub which provides many pre-build images. It is very positive that most third party developers share dockers idea of openness and provide pluggable modules that can be combined.

It offers an easy versioning of images that we use which can be upgraded, downgraded switched to another when necessary. Containerization gains more and more attention since companies may develop application faster, cheaper and better. It’s a great tool and supports continuous deployment. It can be integrated in existing configuration management tool. Since its ecosystem is big and still growing it offers many use cases.

References

[1]. C. Anderson, "Docker [Software engineering]," in IEEE Software, May-June 2015, vol. 32, no. 3, pp. 102-c3.

[2]. T. Inagaki, Y. Ueda and M. Ohara, "Container management as emerging workload for operating systems," 2016 IEEE International Symposium on Workload Characterization (IISWC), Providence, RI, 2016, pp. 1-10.