St. Cloud State University
theRepository at St. Cloud State

Culminating Projects in Information Assurance | Department of Information Systems

5-2018

Virtualization Using Docker Containers: For Reproducible Environments and Containerized Applications

Srinath Reddy Meadusani
St. Cloud State University, srinathreddy45.sr@gmail.com

Follow this and additional works at: https://repository.stcloudstate.edu/msia_etds

Recommended Citation
Meadusani, Srinath Reddy, "Virtualization Using Docker Containers: For Reproducible Environments and Containerized Applications" (2018). *Culminating Projects in Information Assurance*. 50.
https://repository.stcloudstate.edu/msia_etds/50

This Starred Paper is brought to you for free and open access by the Department of Information Systems at theRepository at St. Cloud State. It has been accepted for inclusion in Culminating Projects in Information Assurance by an authorized administrator of theRepository at St. Cloud State. For more information, please contact rswexelbaum@stcloudstate.edu.

Virtualization Using Docker Containers: For Reproducible Environments and Containerized Applications

by

Srinath Reddy Meadusani

A Starred Paper

Submitted to the Graduate Faculty of

St. Cloud State University

In Partial Fulfillment of the Requirements

For the Degree

Master of Science

In Information Assurance

May, 2018

Starred Paper Committee:
Susantha Herath, Chair person
Ezzat Kirmani
Balasubramanian Kasi

Abstract

Software Development practices which evolved in recent years have fundamentally changed the development and management of applications in environment. This evolution includes the Microservices architecture where the applications with large monolithic code has transformed into collections of many small services which are loosely coupled together. The evolution of microservices has changed the requirements of underlying infrastructure, technologies, and tools which were once used to manage the applications. These services improved the agility of delivering software which are portable across all the platforms and infrastructures. Previously large workloads have been processed in large servers which are provisioned by Virtual Machines. But in today's application development environment these large applications have been divided into small applications which collectively run across a collection of commodity hardware. Containers have become handful in running these applications on the same OS as they share the same kernel and hardware. In this paper, I will be discussing about new container technology which is Docker and I will be presenting you how this technology has overcome the previous issues which includes building and deploying large applications. This paper also discusses about the security features of Docker which provides an additional layer of isolation and security for application services.

Acknowledgements

I would like to thank all the committee members Dr. Herath, Dr. Kirmani, and Dr. Kasi for their time, encouragement and valuable suggestions. This paper would not have been possible without their guidance.

Table of Contents

Page

Lists of Tables 6

List of Figures 7

Chapter

I. Introduction 10

Introduction 10

Problem Statement 11

Nature and Significance of the Problem 12

Objective of the Study 12

Study Questions and/or Hypotheses 12

Definition of Terms 13

Summary 13

II. Background and Review of Literature 15

Introduction 15

Background Related to the Problem 15

Literature Related to the Problem 16

Literature Related to the Methodology 23

Summary 25

III. Methodology 26

Introduction 26

Design of the Study 26

Chapter Page

Data Collection 31
Tools and Techniques 31
Hardware and Software Environment 31
IV. Implementation 33
Introduction 33
Docker Networking 33
Docker Storage 39
Dockerfile 41
Docker Swarm 44
Docker Compose 46
Setting Up the Environment and Running Containerized Applications 47
Spinning Up an Apache-based Web Container and Exposing It to Outside World 49
Scaling the Website Using Docker Swarm 57
Docker Security 69
V. Conclusion 72
Future Work 72
References 73
Appendix 74

List of Tables

Table Page

1. Hardware Requirement 31
2. Software Requirement 32
3. Manager and Worker Nodes 57

List of Figures

Figure		Page
1.	Virtualization Using Virtual Machines	17
2.	Cgroups in Linux Containers	19
3.	Containerization Using Docker	26
4.	Containers in VMs Using Docker	27
5.	Docker Architecture	28
6.	Web Interface for DTR	29
7.	Bridge Network	34
8.	Overlay Network	36
9.	Host Network	37
10.	Macvlan Network	38
11.	Containers Storage	40
12.	Dockerfile Sample	42
13.	Image Creation from Dockerfile	42
14.	Image Layers 1	43
15.	Image Layers 2	43
16.	Containerization in Swarm Mode	45
17.	Installing Docker Engine 1	48
18.	Installing Docker Engine 2	48
19.	Docker Engine Status	49
20.	Dockerfile for Apache Webserver	50

Figure Page

21. Building the Apache Image from Dockerfile 1 51
22. Building the Apache Image from Dockerfile 2 51
23. Running the Container Out of Apache Image 52
24. Accessing the Website Using Hostname 53
25. Pushing the Image to Docker Hub 54
26. Verifying the Push on Docker Hub 54
27. Updated Web Content 55
28. Pushing the Updated Image Content to Docker Hub 56
29. Verifying the Updated Push on Docker Hub 56
30. Retrieving the Manager IP Address 58
31. Initializing the Swarm on Manager Node 58
32. Joining Worker1 to Swarm 58
33. Joining Worker2 to Swarm 59
34. Verifying Nodes Status 59
35. Creating the Service on Our Swarm 60
36. Verifying Service Status on Manager Node 61
37. Verifying Service Status on Worker1 Node 61
38. Verifying Service Status on Worker2 Node 61
39. Accessing the Website Using Manager Node Hostname 62
40. Accessing the Website Using Worker1 Node Hostname 62
41. Inspecting the Service 63

Figure Page

42. Building the Updated Image 63

43. Pushing the Updated Image to Docker Hub 64

44. Verifying the Updated Image on Docker Hub 64

45. Rolling Update of Image on all the Containers 65

46. Accessing the Updated Website Using Manager Node Hostname 66

47. Accessing the Updated Website Using Worker2 Node Hostname 66

48. Scaling the Number of Containers Hosting Our Website 67

49. Verifying the Number of Containers Running o Worker1 67

50. Draining a Node 68

51. Verifying Nodes Status 68

52. Verifying Capacity on Worker 1 69

53. Verifying Capacity on Worker2 69

Chapter I: Introduction

Introduction

In the history of computing, Containers have unique recognition because of its importance in virtualization of infrastructure. Unlike traditional Hypervisor virtualization where one or more independent machines run virtually on physical hardware via an intermediate layer, containers run the user space on top of the operating system kernel. Containers provide the isolation between multiple user work space instances. Because of this unique feature container virtualization is often referred to as operating system level virtualization. Instead of starting a complete operating system on the host operating system containers shares the kernel with the operating system which eliminates the overheads and it also provide isolation between the applications. These features of the containers make it possible to ship the small container which acts as a complete operating system which encapsulates only those files which are needed to run our desired applications.

This paper exclusively discusses about one of the containers technology which is currently being used in many production environments to package their applications in isolated environment. This newly evolved containers are none other than Docker which has changed the perspective of deploying the applications in production environment. Docker is an open-source engine which was introduced by Docker Inc in 2013 under apache 2.0 license. The primary goal of the Docker is to provide fast and lightweight environment in which to run the developers code as well as the efficient workflow to get that from the Dev environment to test environment and then into

production Environment. Docker containers are built from application images which are stored and managed in Docker hub. Users can also create their own Docker registries to store their customized images which are created from a Docker file or from an existing container. These flexible functionality features of Docker have made it popular with in no time.

Problem Statement

In today's competitive environment using the available resources efficiently has become imperative for IT organizations. The traditional virtualization techniques which are being used to create virtual machines from few past years, has shown some degradation in the performance of applications which are deployed on those virtual machines. Data center size of these organizations have also been increasing because of these obsolete virtualizations techniques which in turn increasing the cost of infrastructure. Even though the public clouds like Amazon Web Services (AWS), Microsoft Azure have been evolved in past few years to solve these increasing size of private data centers but unfortunately using large number of servers on these public clouds has become an overhead for the organizations because of the monetary constraints. So, it has become a highly important concern for any organization to solve this problem to sustain in today's competitive environment.

There is also been a huge issue in developing and deploying applications in development and production environments. Applications working perfectly fine in Development environment have been showing glitches when they are deployed in production environment. This environment issues have widened the gap between

Development and Operation Teams. Further, this issue has led to the slower delivery of the software with increasing the cost of maintenance. This shows that there is an urgent need of a technology which can deliver the reproducible environments to avoid difference development and production environments.

Nature and Significance of the Problem

Above problem statement has stated the issues facing by the organizations which are using traditional virtualization techniques. The significance of improving the performance of organizations operations by solving this problem has become an inevitable task. There is an urgent need for these organizations to adapt new virtualization techniques to avoid the unnecessary overheads which are specified in the above problem statement.

Objective of the Study

The primary objective of this study is to discuss the available alternative solutions for the specified problem statement. This paper introduces a new virtualization technique using Docker containers which is as an alternative solution for the traditional Virtual Machines for reproducible environments. This paper also compares the security and performance of the applications running in the Containers and Virtual Machines and their resource utilization.

Study Questions and/or Hypotheses

- Why there is a need for virtualization of infrastructure in an organization?
- What are the problems with existing virtualization techniques?

- Why there is a need to use containerized virtualization?
- How did these containers solve the overheads of an organization?
- How secure and compatible are these Containers?

Definition of Terms

Virtualization: Virtualization is a technology in which an application, data storage or guest operating system is abstracted from the truly underlying software or hardware.

Virtual machines: A virtual machine or VM is a virtual computer within the physical computer that runs with an operating system and can used to run applications.

Containers: Containers uses Operating system level virtualization for deploying applications instead of creating an entire VM.

Docker: Docker is an open source tool which is developed to create light weight containers.

Amazon Web Services (AWS): Amazon web services provides public cloud service developed by Amazon where we can host our applications on their servers.

Summary

In this chapter, we discussed about the problems and overheads faced by the organizations who are using existing and obsolete virtualization techniques to virtualize the infrastructure using virtual machines. This chapter introduced the new virtualization technique using Docker Containers which is an alternative approach for the virtual

machines. Further chapters of this paper discuss more about the Operation, Networking, and Security of the Docker containers.

Chapter II: Background and Review of Literature

Introduction

Virtualization is the process of migrating physical environment into virtual environment. This virtual environment can include anything from virtual operating systems to virtual servers. Many companies have already adopted virtualization because they reduce the overheads like maintaining the hardware which is included in large rooms or data centers occupied with large number of devices and cables. Although Virtualization did not completely solve the problem of using bulky hardware but it got succeeded in reducing the usage of unnecessary bulky and costly hardware which was a burden to most of the organizations. This chapter focuses more on the existing and newly evolved virtualization technologies.

Background Related to the Problem

With virtualization, a company can have limitless access to the computing resources which improves operations speed and the business capabilities. There are many ways to do virtualization where creating Virtual Machines using Hypervisors is one of them. Most of the organizations used this method for the virtualization of their operations but the disadvantages of using this method have been widespread recently. VM is a large-weight computer resource and an average VM is a copy of an operating system running on a top of a hypervisor which is running on top of a physical hardware which our application is run on top of. This presents some challenges in for speed and performance and also some other problems in agile sort of environment.

Linux Container technology (LXC) has been evolved to solve this problem but they haven't completely succeeded in overcoming this problem. This chapter discusses more about the problems with the Virtual Machines in production and non-production environment. This paper gives an insight about solve problem of producing a more lightweight, more agile computer resource. Further a brief over view of existing Linux Containers (LXC) is given and their operation is compared with the newly evolved Docker container.

Literature Related to the Problem

In "Using Docker to support reproducible Research" R. Chamberlain and J. Schommer [1] stated that reproducibility and sharing of an environment is imperative factor for an organization to make faster operations. A brute-force approach to achieve this reproducibility and sharing of an environment is through virtual machines. Virtual machines are safe and predictable way to share a complete computational environment. However, there are serious drawbacks in using Virtual machines for reproducibility and sharing of resources. Firstly, it is very hard for a user to do this reproducibility without the very high-level knowledge of Systems administration. Secondly, Virtual Machines consume lots of storage space irrespective of the applications or processes running on them. The below figure depicts how virtualization has been achieved using Virtual Machines and the problem with this method has been discussed with respect to the figure.

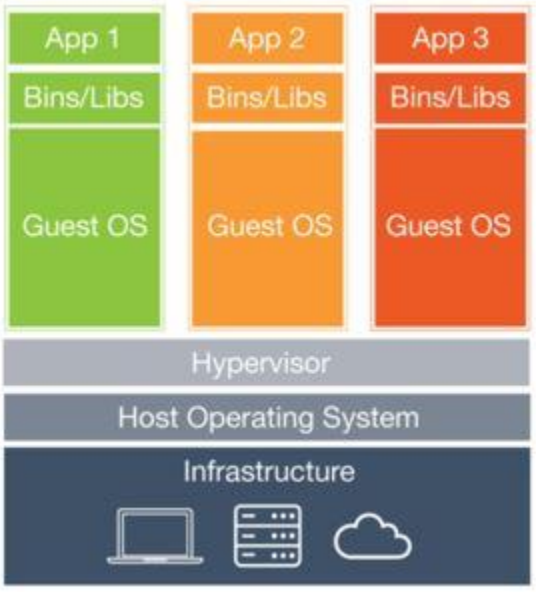

Figure 1. Virtualization Using Virtual Machines [2]

The above figure clearly portrays that a Hypervisor which is an intermediate layer is used to create virtual machines with different operating systems. This hypervisor is installed on the host operating system which distributes the resources to virtual machines as per the configurations specified by the user. Running the virtual machines on these hypervisors consume a lot of CPU memory which degrades the performance of the Host machines if it got bumped up with more virtual machines. Each virtual machine creates a new guest server with GUI, dedicated hardware and library files which means they create a complete replica of the operating systems. A user wants only particular binary files and software in a machine to host and run his applications. But Virtual Machines provides unnecessary binary files which are not required by the user. These unnecessary files consume lots of storage space leading to the ineffective use of infrastructure resources.

Linux Containers (LXC) have been introduced to solve the problem of resource utilization which was created by virtual machines. These containers do not need a separate hypervisor to create an isolated environment. In a large-scale environment using Virtual Machines would mean you are probably running many duplicate instances of the same OS and many redundant boot files which are not required [2]. Containers are lightweight compared to VM's since they contain only the bootable files specific to that application. Since Containers decouple the applications from operating systems users can have a clean and minimal Linux operating system and they can run in one or more isolated containers.

These Linux Containers (LXC) are designed for operating system level virtualization method for running multiple Linux containers on the single Linux Host machine. Cgroups and Namespaces are the two primary features that make this Linux containers possible. Linux Cgroups are developed by google, which governs the isolation and usage of system resources like CPU and memory usage for a group of processes. Consider an example application which takes up a lot of CPU cycles and memory we can put this application in a Cgroup to limit the usage of CPU and memory by that application. Below figure clearly portrays the functionality of cgroups:

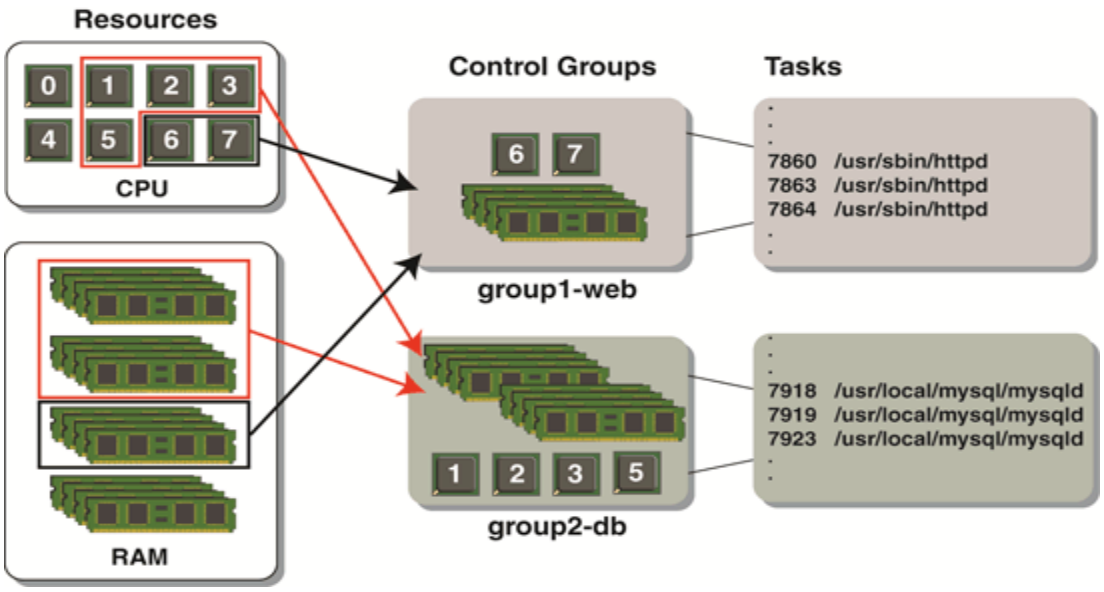

Figure 2. Cgroups in Linux Containers [3]

From the above diagram, we can say that resources allocated to group 2 are twice as that of group 1. In the above example web applications are hosted on Apache web server and they are using MySQL database as a backend server to store the users' data. Since the backend servers like MySQL uses more memory and CPU cycles resources have been allocated to them twice that of front end servers like Apache. In above Figure 2, Apache web server processes have been allocated with 6, 7 CPUs and 1 block of RAM whereas 1,2,3,5 CPUs and two blocks of RAM have been allocated to MySQL servers since they consume more resources. Although these Cgroups are created using the Host Resources these Cgroups control the allocation of resources to processes which is important in isolating the applications from Host operating system. This isolating functionality of Cgroups plays a vital role in creating Linux containers.

Another important feature in creating Linux containers is Namespaces. While Cgroups provides isolation for group processes Namespaces deals with the isolation of resources for a single process. Namespaces isolate the set of system resources and dedicate them to a single process. There are six Namespaces currently, which are implemented Linux. The purpose of each Namespace is to provide an isolated environment for the processes and to implement lightweight containers in Linux distributed systems. In the following paragraphs, we will be discussing about these namespaces one by one:

Mount namespaces. This Namespace isolate the set of file systems seen by the group of processes. Thus, the processes in each namespace will see distinct single directory hierarchies. The mount( ) and unmount( ) calls in mount namespace ceases the set of mount points which are visible to all the global processes and then these set of mount points are dedicated to a single process which is associated with the mount namespace. Mount propagation is another advantage in mount namespaces where mount event in one mount object propagates to another mount object and vice versa if the two mount objects have shared relationship. The mount object which propagates the mount event is called “shared mount” and the mount object which receives the mount event is called “slave mount”. Mount object which neither propagates mount event nor receives an event is called “private mount”.

UTS namespaces. The term UTS is derived from the name of the structure passed to the uname( ) system call. By using this namespace, we can give a separate domain name and host name to the processes. Sethostname( ) and setdomainname( )

systems calls are used to set these separate names for a process. This UTS namespace allows the containers to have its own domain name and host name. We can initialize the scripts in containers to automate the tasks based on these separate domain and host name of the containers.

IPC namespaces. IPC stands for inter process communications which isolates the certain inter process communication resources namely System V IPC and POSIX message queues. By using this namespace, we can isolate the communication between two processes and we can also share data between processes in the form of messages.

PID namespace. Process ID namespace isolates the process ID numbers, which means that processes in different namespaces can have same PID. The main benefit of this namespace is we can migrate the containers between the hosts without changing the PIDs of processes running inside the containers. Migration of containers would have been failed without this namespace because the PIDs would have been same in the destination host which creates the conflicts between when addressing the tasks using their respective PIDs. This namespace creates its own init(PID1) for the containers, which is the ancestor of all processes responsible for various system initialization tasks.

Network namespace. Network Namespace isolates the system resources associated with networking. With this namespace, each network can have its own IP addresses, IP route tables, Network devices and port numbers etc. Containers can have their own virtual network devices and its own applications that bind to isolated port numbers, which is possible by changing the routing rules in the host system such that

the network traffic can be directed to the network devices associated with a specific container. By leveraging this namespace, we can host multiple webservers (running on different containers) on a single host with network traffic routed to port number 80.

User namespaces. So far, this namespace is the most complex namespace added to the Linux kernel. This namespace allows the per-namespace mapping of user and group IDs. In containers, this namespace allows the users and groups to have certain privileges only inside that container. For example, a user can have root privileges inside the container but he/she is a guest user on the host system. Each process user IDs and group IDs have two different values one inside the container and the one outside the container which is host system. This duality can be achieved by mapping the user IDs on host system to the user IDs inside the container. For example, a user ID 1500 host system might be mapped to the user ID 10 inside a container where user ID 1500 would be a normal user on the host system and user ID 10 would have root privileges inside the container.

Although there are so many complex operations in Linux Containers which can solve our above-mentioned problem, they are not portable as they do not completely abstract the applications from lower level resources like networking, OS, and storage. To address this issue Docker Inc. has come with a solution by introducing their new software in 2013 which is called Docker. Further sections of this paper discuss more about how Docker solved the above-mentioned problems.

Literature Related to the Methodology

Docker is open source platform based on Linux Containers (LXC) which completely packages software applications. It is backed by a private company that focuses on providing a platform which is easy and scalable for hosting web applications. As we discussed in the above sections LXCs provide a completely operating system level virtualization which creates a sandboxed virtual environment in Linux that eliminates the overhead without creating a complete virtual machine. Docker extends this terminology of LXCs to make it user friendly and provides easy versioning, distribution, and deployment.

These Docker containers can be launched in a sub second, and then you can have a hypervisor that sits directly on top of the operating system. By this we can pack a lot of the containers on a single Physical or Virtual Machine. This gives an added advantage of effective usage of available resources. Docker, allows there to be just one host operating system, and provides a layer of software at the top of the operating system that isolates multiple applications and their required supporting stacks of software from each other, and from the operating system.

Docker are created to provide lightweight and fast environment in which to run users code with efficient workflow and get that code from user's laptop to test environment and then into production environment [4]. Docker is very simply because one can run it on simple host which has nothing but a compatible Linux kernel and binary files. The mission of Docker is to provide following features:

Easy and lightweight way to model reality. Docker are so fast such that one can easily containerize their applications within minutes. Users can modify their applications and dockerize their applications within no time. When a change is applied to an application a new container will be created to run these modified applications. Unlike Virtual machines which uses hypervisor Docker containers takes only seconds to launch. Then the modified applications are packaged into the newly created containers.

Logical segregation of duties. With Docker, it has become easy for an organization to segregate the duties between Development and Operations teams. Development focuses on developing the applications inside the containers while operations team focuses on managing these containers. Docker enhances the consistency by providing the same environments in which Developers write the code and operations team deploy the code. This methodology removes the conflicts between Dev and Ops teams by resolving “worked in Dev, failed in Ops” problem [4].

Fast and efficient application lifecycle development. The downtime in the production environment can be drastically reduced by using Docker. They reduce the cycle time between code being written by developers and code being tested, deployed by the operations team into the production environment.

With the above features, Docker have resolved many challenges like Dependency Hell, imprecise documentations, tackling code-rot with image versions and barriers to adoption and research [5]. Docker containers also enhances the security features of application in two ways. One is by providing isolation between application and another is providing isolation between application and host system. They also

reduce the host surface area to protect both the host and co-located containers by restricting access to the host [6].

Docker Containers also enhances security by providing Process restrictions, Device and file restrictions, application image security and open-source security and other Linux kernel security features. Inside the containers unprivileged users cannot be added to root group so that they won't have privileges like sudo. This improves the overall security of the applications and makes running applications inside the containers more secure.

Summary

In this chapter, we have discussed about the literature related to the problems created by using Virtual Machines. A brief theory about Linux containers has been discussed which are the major contributors to the above-mentioned Docker methodology. Introduction to Docker features and their security enhancements have been provided in this chapter. In the next chapter, this paper discusses more about Docker methodology, architecture, design of study and work in progress.

Chapter III: Methodology

Introduction

This chapter discusses more about architecture of Docker methodology which has been introduced in the above chapter. It also gives more information about how Docker containers are used in real time environments. By the end of this chapter audience will gain a high-level idea about how to create containers using Docker and how to deploy them in production environment.

Design of the Study

The proposed study uses the mix of Qualitative and Quantitative approach as we are comparing the security features and resource utilization of containers and virtual machines. Docker containers have been used to support the study and detailed explanation of these technology has been provided in the following paragraphs.

Before moving on to discussing how Docker works let us first understand how Docker are used in creating the containers with in no time and how its functionality is different from traditional Virtual machines.

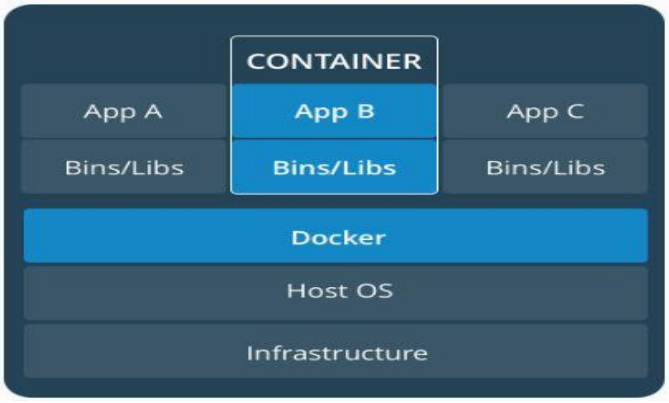

Figure 3. Containerization Using Docker [6]

From above figure, it is evident that by using Docker we do not need to use a hypervisor to create a new environment for our applications. Containers created using Docker are vanilla which means the containers are created only with the bootable files which are necessary to start up the system and it does not contain all the unnecessary binary files or libraries. These containers have only those files which are specific to that application running inside the container. Since these containers are vanilla flavored its easy and fast to create them. One can also create containers on virtual machines using Docker for to achieve more flexibility in deploying the applications. The following figure gives the architecture of how containers are created on virtual machines:

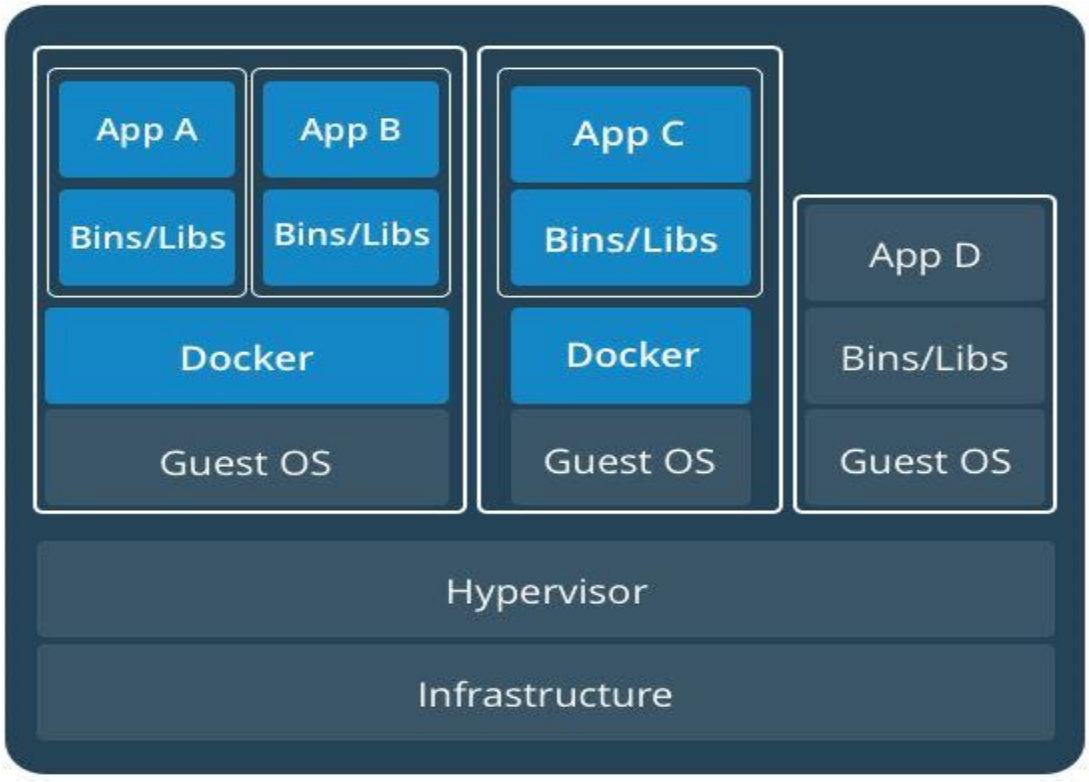

Figure 4. Containers in VMs Using Docker [6]

From the above figure, it is evident that it is very easy to create containers on Virtual Machines using Docker. The only thing we need to do is to install Docker engine on the required virtual machines where containers are needed to be created.

Docker architecture. The working operation of Docker can be understood by taking a deep dive at its architecture. The following figure depicts the underlying architecture of Docker:

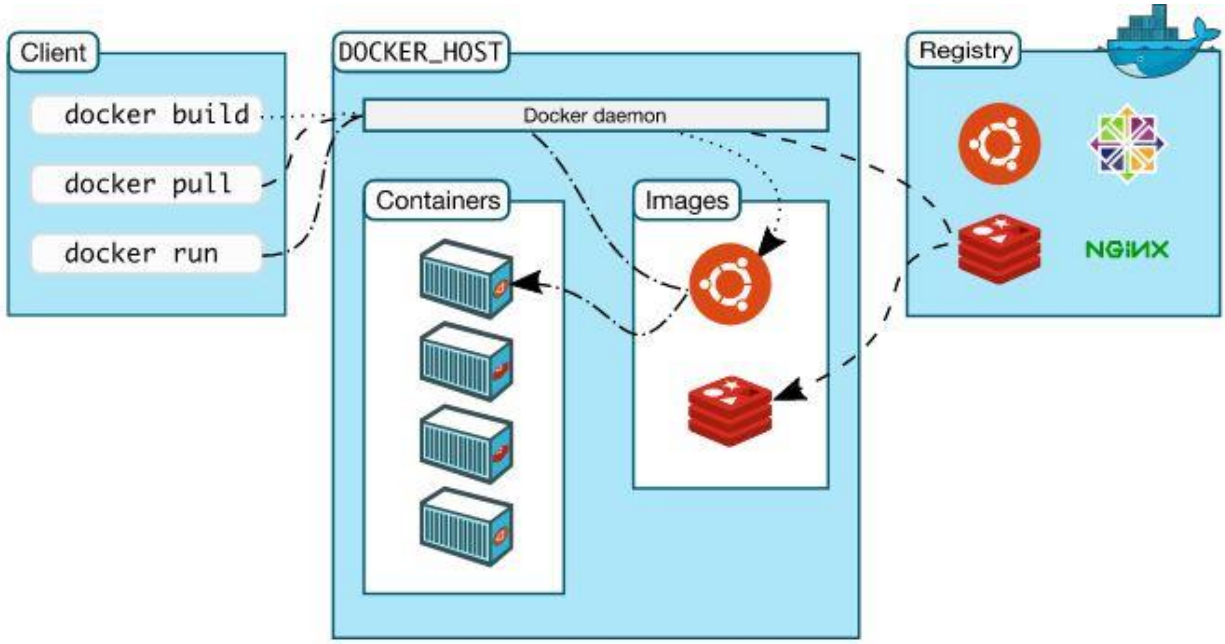

Figure 5. Docker Architecture [7]

Docker client, Docker Daemon and Registry are the three main components of the Docker architecture. More about these components are discussed below:

Docker client. Docker is a Client-server application, where Docker client talks with the Docker server or daemon which in turn does all the work. Docker ships with a command line client library Docker and full restful API [4]. One can run Docker client

and daemon on the same host or can connect local Docker client to remote Docker Daemon which is running on the remote host like AWS Server.

Docker daemon. Docker Daemon listens for Docker API requests from Client and manages Docker images, containers, networks, and volumes. When a request from client has been received to create a container, Docker daemon pulls the specified image from the Docker registry or local image registry and then creates the container from that image. A Daemon can also communicate with other Daemons to manage Docker services.

Docker registry. Images which are used to create containers by Docker are stored in Docker registry. Docker Hub and Docker cloud are two official public registries maintained by Docker Inc, and is supported by the Docker community by uploading thousands of images regularly into those registries. When a user use Docker pull or run command the required images are pulled from their docker hub registries.

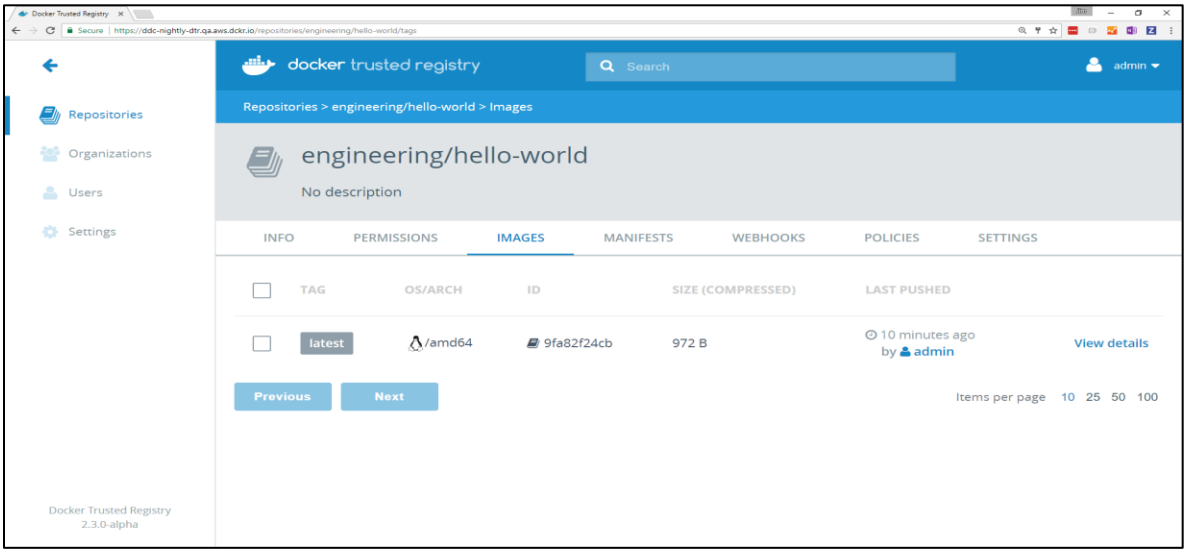

Figure 6. Web Interface for DTR [8]

For enterprise users Docker Trusted Registry (DTR) is a secured repository to store their images. Users can install it behind their firewall so that images can be stored and accessed securely. DTR can be installed on on-premises virtual machines or a public cloud depending up on the organizations requirement and it can also be accessed easily because of its user-friendly web interface.

Docker commands. In this section, we will be discussing few basics commands used by Docker to create and manage containers:

The following command is used to create a container:

"*docker run -it --name <name of the container> <image> /bin/bash*"

Here it is optional to give a name for the container, but it is best practice to give a name to avoid confusions between other containers. Image can be any Linux operating system flavor such as Ubuntu, centos, openSUSE etc. If the image is not available locally then the above command will pull the required image from the docker hub or DTR and then run it inside the container. If we want to explicitly pull or get an image from the docker hub use the below command:

"docker pull <image_name>"

To start a stopped container, we need to run:

"*docker start <container name or id>* "

"*docker attach <container name or id>"*

To run the container in background or in daemonized mode we need to give parameter "-d" while creating the container.

For this research, Docker engine will be installed on the local server and containers will be created. Subsequently Applications will be deployed on the Containers which are created using Docker and on Virtual Machines. These applications and their resource utilization will be monitored, and their data will be analyzed using various tools.

Data Collection

As of now no data has been collected for this paper. Subsequently no tools or techniques have been used to analyze the data. In the future, performance and resource utilization of the Applications will be analyzed using Nagios tool and Splunk data analyzing Tool and applications security will be tested using AppDynamics**.**

Tools and Techniques

For this study, Nagios Monitoring tool will be installed on Local server and on virtual machines to monitor the resource utilization by this machine. AppDynamics will be used for security analysis of the applications and data will be analyzed using Splunk.

Hardware and Software Environment

Table 1. Hardware Requirement

Resource	**Minimum Required**
Processor	Intel/AMD
Processor Type	32/64 bit
Speed	2 GHz
Disk Space	100 GB

Table 2. Software Requirement

Tools	Docker Engine
Hypervisor	VMware work station, Virtual Box
Operating System	Linux (Ubuntu, Centos or RHEL)
Database	MySQL
Languages	HTML, CSS, Shell Scripting
Web / Application Server	Apache
Web browser	Google chrome
Cloud Provider	Amazon Web Services(AWS)

Chapter IV: Implementation

Introduction

This chapter provides more information about how to implement docker containers in developing and testing environments by leveraging its out of box features and techniques. As the containerization using docker is a vast topic one should have knowledge about its core features such as Docker Networks, Docker Storage, Dockerfile, Docker Swarm, Docker Compose and Security. Before we implement anything using docker, understanding these core features is an imperative task which would make it easy to understand the further topics while we are implementing the containers using docker.

Docker Networking

The way networking has been designed for docker containers is one of the primary reasons for making it as the one of the most modern day powerful tool for containerization. Docker networking is much sophisticated that the containers and services can be run together on same hosts or a different host and a container running on a Linux machine can connect with a container running on a windows machine. These features can be implemented by using network drivers such as bridge, host, overlay and macvlan drivers which are provided by docker engine itself. Depending on the application requirements we will be using the below container networks for our project.

Bridge network driver. When a Docker engine is installed, and a container is spanned up on the host machine, bridge network is the default network that our container is created on. It is the most simple and easy to create networks on docker

engine, which restricts its capability to single host. This creates a private internal network on the host and containers created within this network can communicate to each other and external access to this container is granted by exposing its ports [9]. Docker engine takes care of behind the scenes such as iptables, network interfaces and host routes to make this communication and connections possible. Below figure gives a clear idea about the functionality of docker's bridge network:

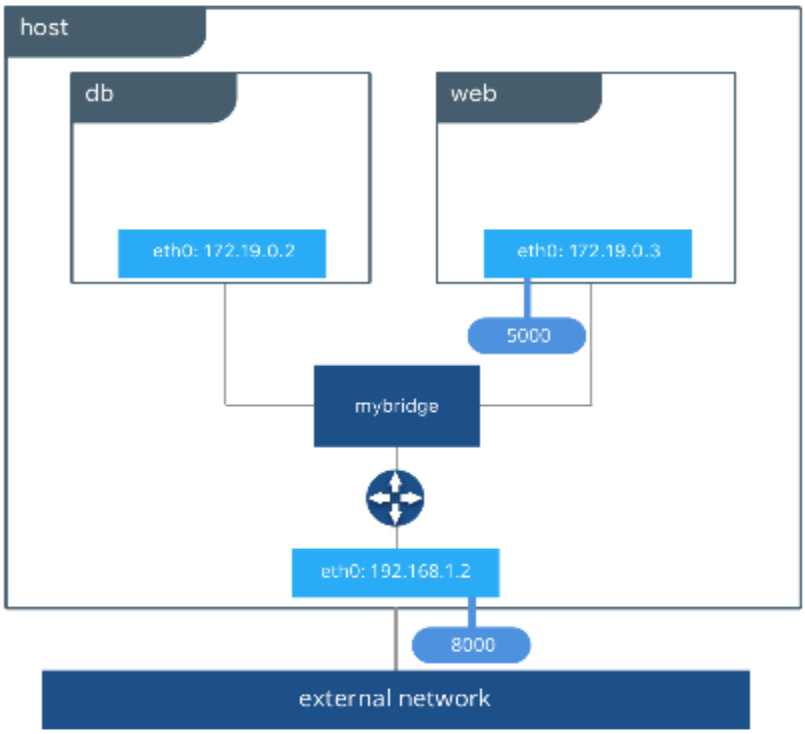

Figure 7. Bridge Network

The above figure depicts that a bridge network called **mybridge** has been created on our host and the containers **db** and **web** are created within this bridge network. Here **web** container can directly communicate with **db** container for any selects, inserts or updates without any networking hassles as they are created within

the same network. To access the contents of the website which has been deployed in **web** container, we need to expose the ports of **web** container to the host machine. In our above example, website has been served in web container on port 5000 and this port has been mapped with port 8000 on host machine. So that user can access this website by using the host's IP address and the associated mapped port.

The bridge network is easy to understand, simple to create and troubleshoot but its capability is limited to only single host. If we want to deploy our website and databases on different hosts bridge network cannot provide us the essential networking features for the communication between these two. Overlay networks overcome these difficulties of hosting on single host which has been discussed in the next section.

Overlay network driver. Overlay network is a built-in network driver which simplifies the complexity of hosting the containers on multiple hosts without any external provision or components. Load balancing between the containers, service discovery and multi-host connectivity are built right in this driver, which makes it one of the most efficient container networking drivers. The overlay driver utilizes an industry-standard VXLAN data plane that decouples the container network from the underlying physical network (the underlay). This has the advantage of providing maximum portability across various cloud and on-premises networks [9]. Below figure depicts the powerful features of this driver:

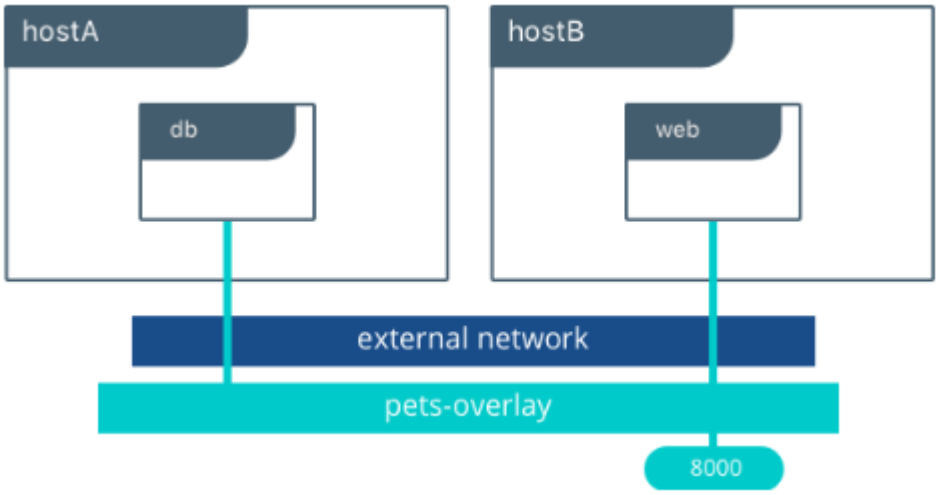

Figure 8. Overlay Network

Here **db** container and **web** containers are created on different hosts hostA and hostB respectively but they are connected to the same network which is pets-overlay network. This network can be created on universal control pane (UCP) or on docker swarm manager and this network should be attached to the containers while spinning up them. Based upon the traffic one can scale up the number of **web** containers to reduce the down time where UCP and Docker swarm will take care of load balancing the traffic between these containers. When services are deployed in multiple containers VIP based load balancing will be distributing the traffic across all the containers. Overlay networks provides an outstanding solution for many networking challenges to host the applications on multi host containers.

Host network driver. Host networks is the simplest form of network drivers which does not isolate the container network from docker host network. For example, if we bind the port 9090 on our container and if we use host network that container

application which is hosted on that port will be accessible on the same port (9090) of the docker host.

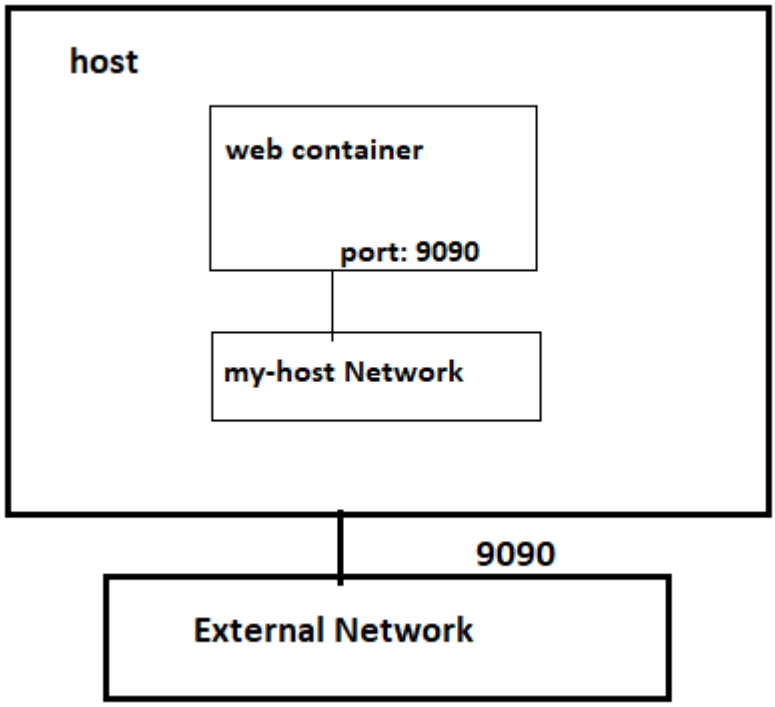

Figure 9. Host Network

Above figure depicts that a web container has been hosted on the my-host network which is a host network with port 9090 exposed and this port can be accessed on the same port number of the host such as <host_ip>:9090. Host network has limited capabilities and if your container doesn't use or publish ports then host network is a no go.

Macvlan network driver. Macvlan driver is the newest driver in this driver stack of docker which connects the containers interfaces directly to the host interfaces. Some of the legacy applications which monitors the network traffic expect to be directly

connected to the physical network. In this case we can use the macvlan network driver to assign a MAC address to each container's virtual network interface, making it appear to be a physical network interface directly connected to the physical network. Containers on this network are addressed with the routable IP addresses which are on the subnet of external network.

The macvlan driver can be configured in different ways to achieve different results. In the below example we create two MACVLAN networks joined to different sub interfaces. This type of configuration can be used to extend multiple L2 VLANs through the host interface directly to containers. The VLAN default gateway exists in the external network.

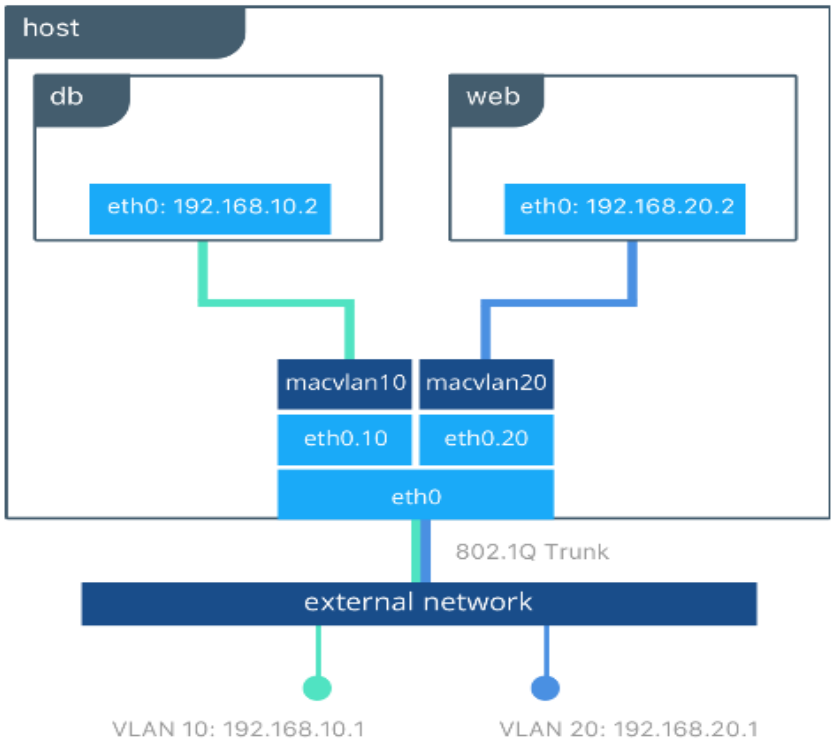

Figure 10. Macvlan Network

In the above figure the **db** and **web** containers are connected to different MACVLAN networks in this example. Each container resides on its respective external network with an external IP provided from that network. Using this design an operator can control network policy outside of the host and segment containers at L2. The containers could have also been placed in the same VLAN by configuring them on the same MACVLAN network.

Docker Storage

One of the biggest challenges in using containerization is about storing the data of the applications which are running inside the container. Data persistence can be lost when a container has been longer running and another container needs the data from this stopped container. Data can also be completely if a container has been removed or crashed due to the internal glitches. Docker provides us a mechanism to persist the data irrespective of stopping or removing the containers. It offers three different approaches such as volumes, bind mounts and tmpfs to mount the data into the container from the host where this container is up and running.

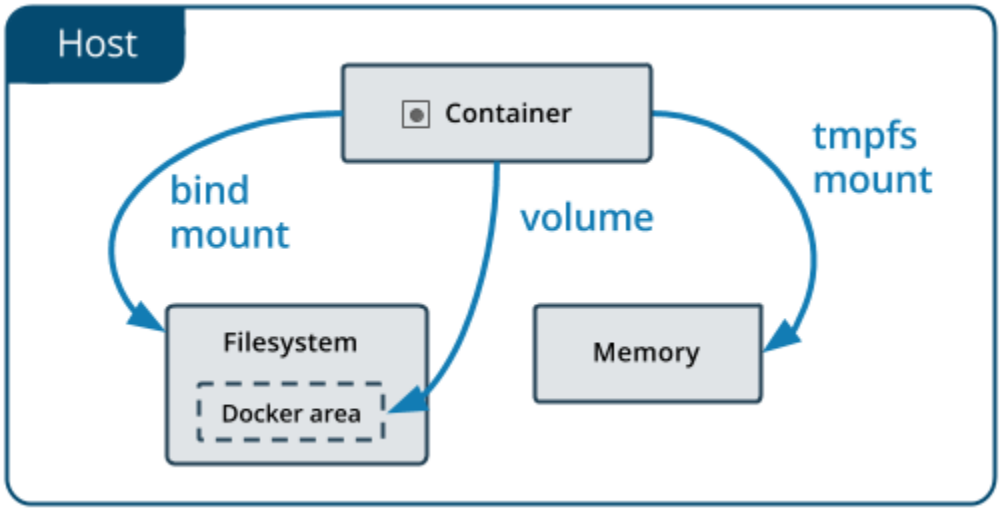

Figure 11. Containers Storage

The above figure gives a clear idea about where the container's data is stored on the docker host using those different approaches to mount the data. Upcoming section describes about these three approaches and the differences between them in storing and persisting the data.

Volumes. Volumes are created and managed by docker and are stored as part of the host file system, where as non-docker processes should not modify this file system. One can create a volume by explicitly using *docker volume create* command or docker will be automatically creating a volume for you when you create a container. When we create a volume, it is stored inside the directory on the docker host and this directory will be mounted into the container when we mount volume into that container [10]. A Volume can be mounted into more than one container and the data will be persisted on the docker host even though when that container is removed or stopped.

Volumes also support the use of volume drivers which would allow to store your data on cloud providers or remote host which would make the data more persistent.

Bind mounts. Bind mounts are similar to volumes where a file or directory is mounted into the containers when we use the bind mount. But this file or directory is referenced by its full path on the host machine and it doesn't need to be already exist on the docker host. When the data of these files or directory changes on docker host these changes are automatically reflected inside the containers and vice versa where it has been mounted. One of the disadvantages of using bind mounts is it completely relies on host filesystem having a specific directory structure available and it also has the capability of modifying the important data on the host file system by the processes running inside the container which is a major security concern.

Tmpfs mounts. Tmpfs mounts can be best used in the scenario where you do not want the data to persistent either on the host or inside the container. This can be for security reasons or the performance of your container where your application writes a large amount of non-persistent data. Tmpfs mount store the data on the host memory which makes it volatile and this data cannot be shared by multiple containers.

Dockerfile

Dockerfile is best described as **infrastructure as a code** where one can create the images which are necessary for building an QA or Production environments by scripting the Dockerfile and these images can be used to spin up multiple number of container for reproducible environments. Dockerfile simply consists of a bunch of command which will be executed in the order they were scripted and finally gives us the

image where these instructions have been assembled. Below figures give us a clear idea about how this can be achieved:

```
FROM ubuntu:latest

MAINTAINER Srinath Reddy "srinathreddy@gmail.com"

RUN apt-get update
RUN apt-get install -y python python-pip wget
RUN pip install redis
```

Figure 12. Dockerfile Sample

```
[user@srinathreddy452 ~]$ docker build -t python_image .
Sending build context to Docker daemon  58.22MB
Step 1/5 : FROM ubuntu:latest
latest: Pulling from library/ubuntu
22dc81ace0ea: Pull complete
1a8b3c87dba3: Pull complete
91390a1c435a: Pull complete
07844b14977e: Pull complete
b78396653dae: Pull complete
Digest: sha256:8b9e8a1fdc77c5042a1151788f1e4f5fd5d45528833fd8cf22ff6bbf8992cf7b
Status: Downloaded newer image for ubuntu:latest
 ---> f975c5035748
Step 2/5 : MAINTAINER Srinath Reddy "srinathreddy@gmail.com"
 ---> Running in 83ad3119a782
Removing intermediate container 83ad3119a782
 ---> 44e31af722c1
Step 3/5 : RUN apt-get update
 ---> Running in d941585d12a6
Get:1 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
Get:2 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
Get:3 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]
Get:4 http://archive.ubuntu.com/ubuntu xenial-backports InRelease [102 kB]
Get:5 http://security.ubuntu.com/ubuntu xenial-security/universe Sources [73.2 kB]
```

Figure 13. Image Creation from Dockerfile

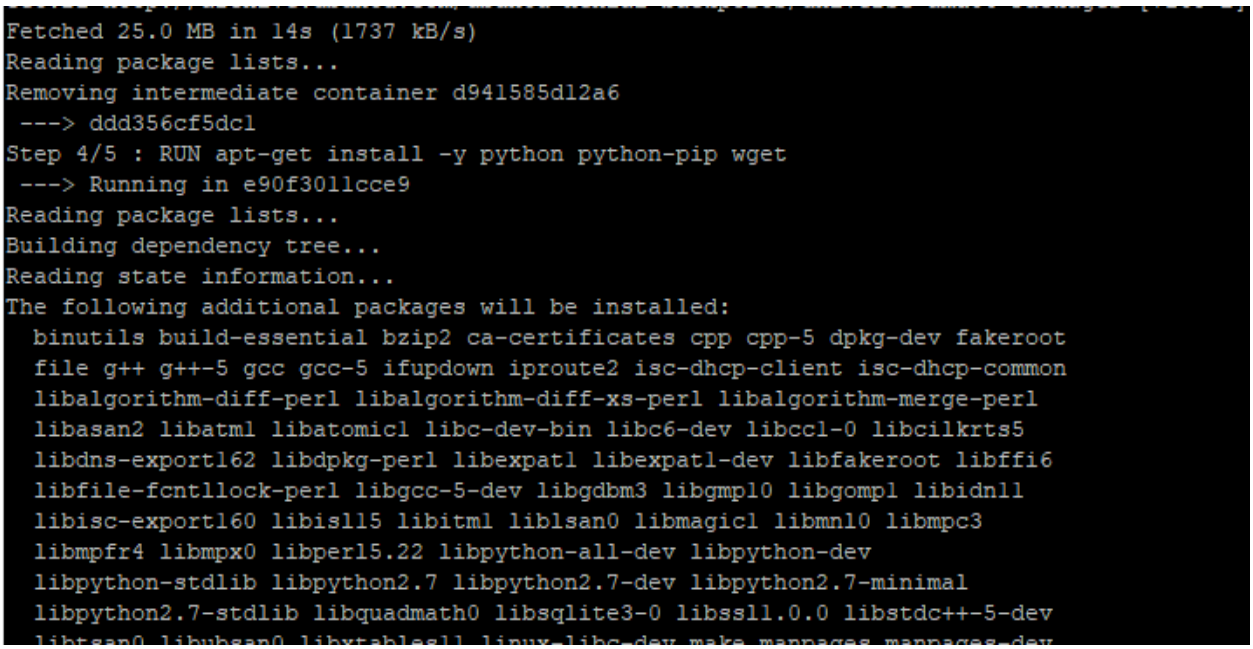

Figure 14. Image Layers 1

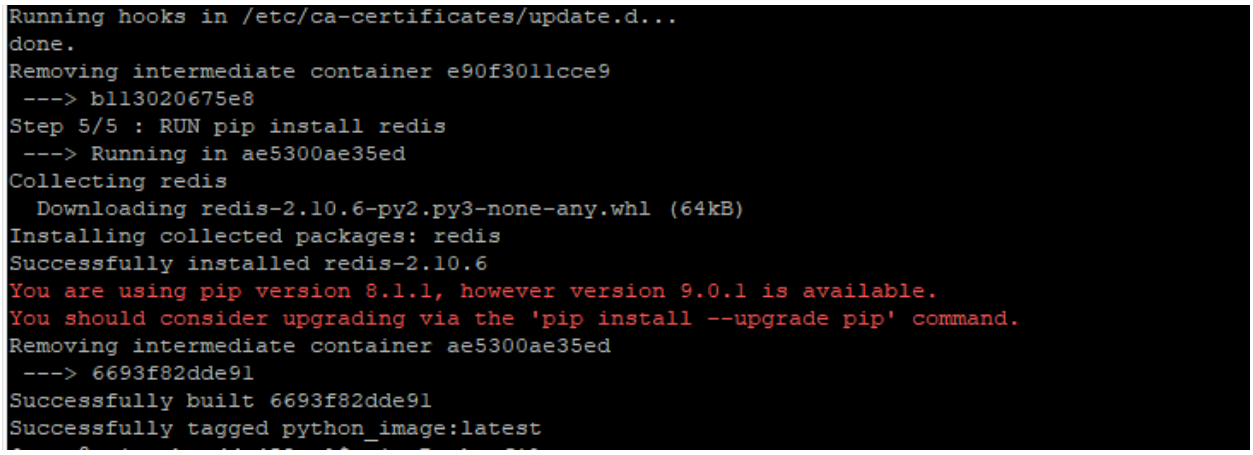

Figure 15. Image Layers 2

In the figure depicts a Dockerfile which creates a ubuntu image, updates the packages, installs python and wget and then with help of python's pip library installs redis. Here we can see that for every step in Dockerfile it creates a new container and creates a new image out of that container and merges them all together at end and

removes all those intermediate containers. One can spin up as many containers as he/she can with this newly created image and the dependency between these containers is not at all a mandatory task.

As discussed in the previous sections one can also store these images on Docker hub or DTR so that it would be easy to share them to other members of the organization. If we want to delete an image on your local docker host all the associated containers of that image must be stopped and removed first and then we would be able to delete our image.

Docker Swarm

For any application to be up and running with zero percentage of downtime, the underlying infrastructure should be able to scale up or down based upon the traffic to that application. Container orchestration system takes care of this hassles by deploying the applications on multiple cluster of nodes or virtual machines based upon that application's traffic which are running inside the container. This system should also be able to perform the health checks on the nodes where the containers and running and should be capable of routing away the traffic from the node where the health checks have failed. Apart from this it should also be able to load balance the traffic and should be capable of doing the rolling updates on the applications deployed in the cluster. Docker Swarm is an orchestration tool which comes within the docker engine which performs the all the above tasks and makes sure that down time to be at zero percentage of the applications which are deployed in the swarm cluster.

Docker swarm uses the concept of Manager nodes and worker nodes where the worker nodes are registered with one of the manager nodes and sends the health checks to the manager so that manager could schedule the tasks on these worker nodes. Swarm mode is composed of multiple docker hosts where any host can perform as manager, worker or both based upon the application requirement. When a worker node is unavailable the manager would automatically assign that task to the healthy and available node. There can be more than one manager node but however only one manager node will be the primary while the rest are used as standby manager nodes which only participates in the election to elect the primary manager when it is down. Orchestration using Docker Swarm can be easily understood by looking at the below figure:

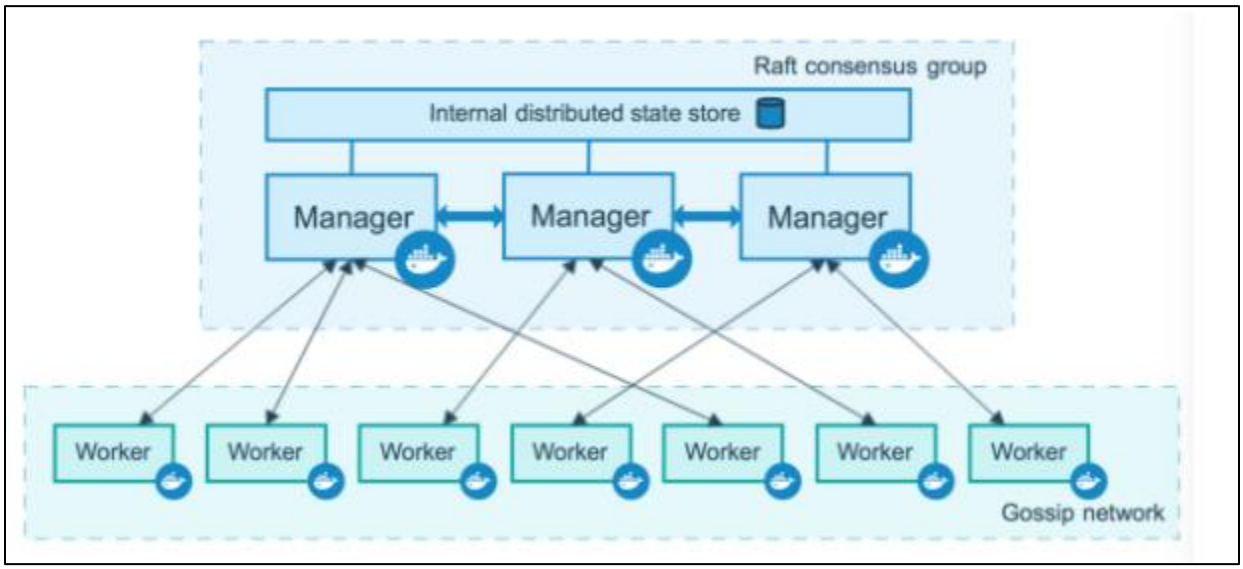

Figure 16. Containerization in Swarm Mode

Docker swarm uses Raft consensus algorithm to store the state of all the nodes so that scheduling the tasks on worker nodes would make easy for manager node. Whenever a task or service is scheduled on the nodes the task status and swarm state get updated on all the manager nodes to maintain the synchronization between them. Whenever a primary manager goes down the newly elected manager could easily continue it tasks as it would have already received the status of the swarm. For better fault tolerance it is always a best practice to have at least three managers but increasing the number of managers might decrease the performance of the cluster as the data synchronization would take more time between the multiple number of manager nodes.

It is also a best practice to not to assign any tasks on manager nodes so that we can always make sure that load on the manager node is low and this can be achieved by making the availability of manager nodes to drained state. We can also promote worker node to manager node whenever a manager node is taken down for maintenance. All these docker swarm features are implemented practically in the upcoming sections which would give a better idea of clustering and orchestration using docker swarm.

Docker Compose

When your application needs more than one container which are isolated Docker compose would come in handy in building, running, and connecting those containers and entire setup can be done on single host. Docker compose is very useful in Dev and QA environments which reduces the overhead of maintaining and monitoring the

infrastructure. One can easily spin up a development environment on his/her local desktop by using docker-compose.yml file and this can be shared on the source control repositories with the other team members contributing to it and leveraging the rapid creation of an environment without installing any tools locally.

Docker-compose.yml contains the instructions which are written in YAML (Yet Another Markup Language or YAML Ain't Markup Language) to spin up our Dev / QA environments. Compose tool is such powerful that it can manage the whole application lifecycle such as starting and stopping the services, building the services, running a command against your service, and monitoring your service logs. Implementation of the docker compose has been explained in the upcoming sections which would explicitly describes its core features.

Setting Up the Environment and Running Containerized Applications

For implementing this project, we will be using Linux based virtual machines which are hosted on Amazon web services (AWS) with 1 core, 2GB RAM and 20GB disk space. First of all, we need to install the docker engine on all these host machines using CLI which has been depicted in the following figures:

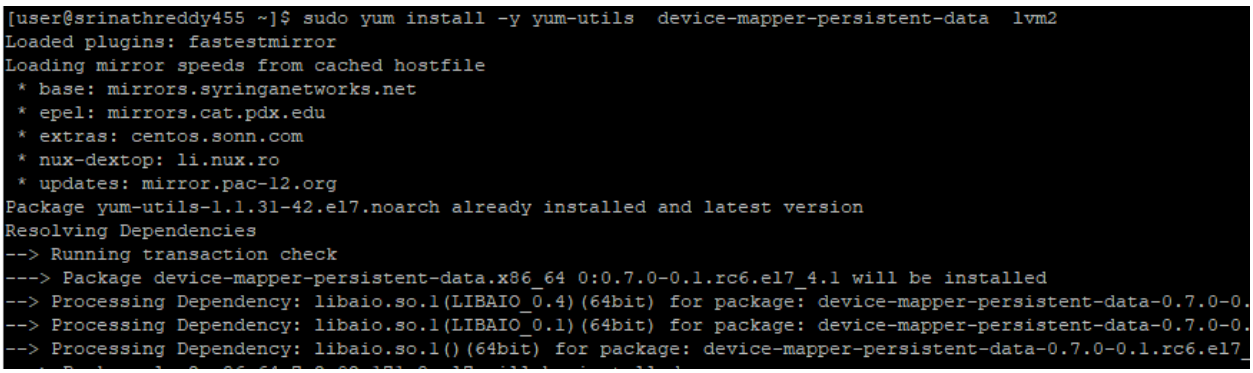

Figure 17. Installing Docker Engine 1

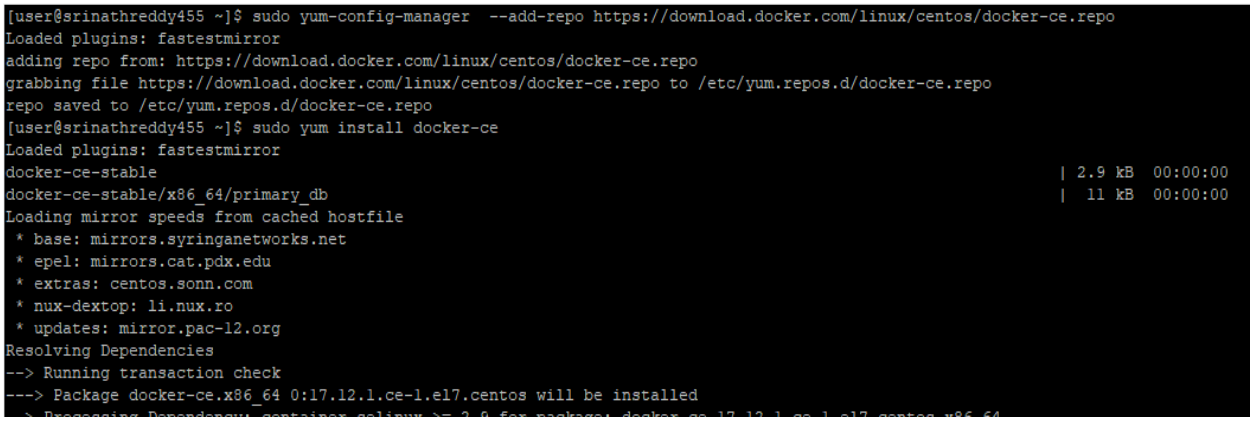

Figure 18. Installing Docker Engine 2

Here we are following three steps in installing the docker engine, first we need to install few packages such as yum-utils, device-mapper-persistent-data and lvm2. Second, we need to setup stable repository by yum-config-manager which comes from yum-utils package and then add the docker repo. Finally install the docker engine on the host machine using yum install where yum is the package manager for Redhat based Linux machines and start the docker engine using systemctl command. To check the version of docker engine installed on your host machine use "*docker -v*" and to check if docker engine has started on your host machine use "*systemctl status docker".*

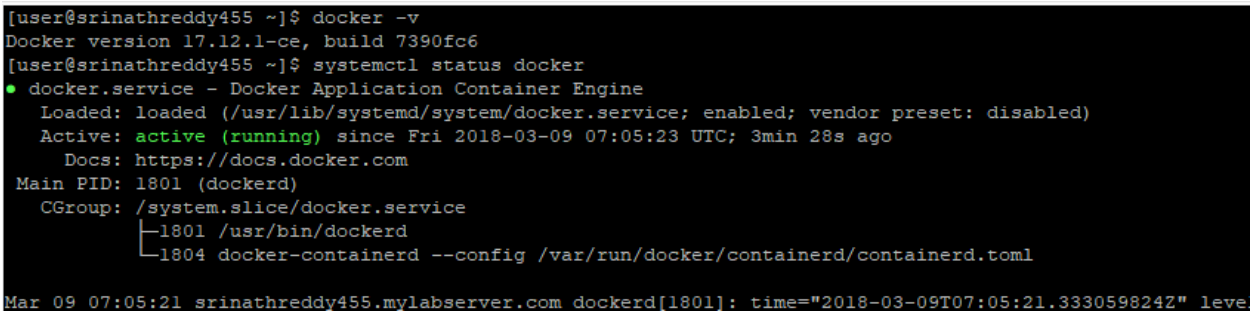

Figure 19. Docker Engine Status

As carrying the activities on Linux machines without using the root user is considered as industry's best practice, one need not to be a root or sudo user to run docker commands and to achieve this add user name to docker group using the following command '*usermod -aG docker <user_name>'.*

Spinning Up an Apache-based Web Container and Exposing It to Outside World

After installing the docker engine successfully we will be spinning up an apache-based web container and deploy our website in that container which can be accessed by the outside world. Here we will be creating a Dockerfile which consists of all the instructions which are needed to host our website on apache web server which is running inside our container. Before that we need to build our website and it should be ready to get deployed in our container.

For this project the website for "animals" was build using HTML and CSS and the following figures describes how this website was deployed into the container:

```
FROM ubuntu

# File Author / Maintainer
MAINTAINER srinath_reddy

# Update the repository sources list
RUN apt-get update

# Install and run apache
RUN apt-get install -y apache2 && apt-get clean

EXPOSE 80
CMD apachectl -D FOREGROUND
~
~
~
```

Figure 20. Dockerfile for Apache Webserver

Here we are using ubuntu as our container OS and running the update command inside the conatiner followed by installing the apache webserver using **RUN** command and then exposing the port of the conatiner to outside host on which our apache server runs using **EXPOSE** command and then starting the webserver using **CMD** command. After editing our Dockerfile we will be using **docker build** command to build our image as shown in below figures. Once this command is being exceuted all the instructions in our Dockerfile will be executed layer by layer image and at the end all these layers are formed into one single image which is used to create our container. These execution steps are depicted in the figures below:

```
[user@srinathreddy455 App]$
[user@srinathreddy455 App]$ docker build -t apacheimage .
Sending build context to Docker daemon  5.632kB
Step 1/6 : FROM ubuntu
 ---> f975c5035748
Step 2/6 : MAINTAINER srinath_reddy
 ---> Running in 3cd3556bfbc0
Removing intermediate container 3cd3556bfbc0
 ---> 79cc31b8cb06
Step 3/6 : RUN apt-get update
 ---> Running in cf71ac9ee26e
Get:1 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
Get:2 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
Get:3 http://security.ubuntu.com/ubuntu xenial-security/universe Sources [73.2 kB]
Get:4 http://security.ubuntu.com/ubuntu xenial-security/main amd64 Packages [589 kB]
Get:5 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]
Get:6 http://archive.ubuntu.com/ubuntu xenial-backports InRelease [102 kB]
Get:7 http://archive.ubuntu.com/ubuntu xenial/universe Sources [9802 kB]
Get:8 http://security.ubuntu.com/ubuntu xenial-security/restricted amd64 Packages [12.7 kB]
Get:9 http://security.ubuntu.com/ubuntu xenial-security/universe amd64 Packages [406 kB]
```

Figure 21. Building the Apache Image from Dockerfile 1

```
Processing triggers for libc-bin (2.23-0ubuntu10) ...
Processing triggers for systemd (229-4ubuntu21.1) ...
Processing triggers for sgml-base (1.26+nmu4ubuntu1) ...
Removing intermediate container a06c8796312f
 ---> 7415d894c46d
Step 5/6 : EXPOSE 80
 ---> Running in f385c950755e
Removing intermediate container f385c950755e
 ---> 2e2fbac0c373
Step 6/6 : CMD apachectl -D FOREGROUND
 ---> Running in 9b1a42db558d
Removing intermediate container 9b1a42db558d
 ---> e8e802523692
Successfully built e8e802523692
Successfully tagged apacheimage:latest
[user@srinathreddy455 App]$
```

Figure 22. Building the Apache Image from Dockerfile 2

Since our image has been ready to use, now we should be able to spin our container to host our website. Before that I have create a directory called website where our website contents reside, and this can be viewed in below figures where I have used **tree** command for this purpose.

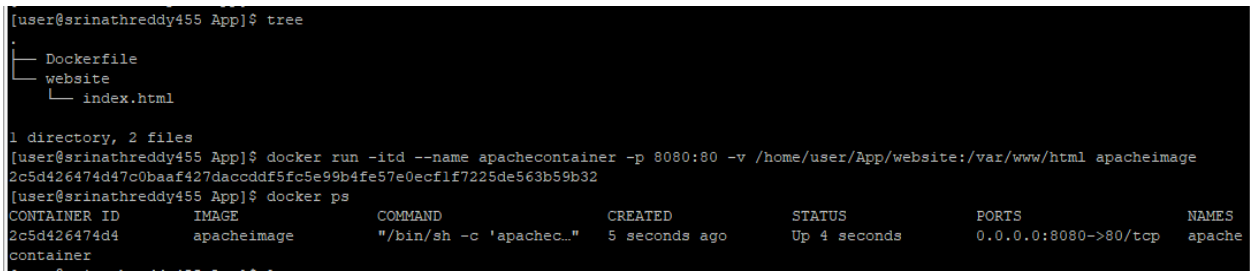

Figure 23. Running the Container Out of Apache Image

After creating the directory, we will be using **docker run** command and few other parameters to spin our container which has been shown in above figure. We have named our container as **apacheconatiner** using the parameter **--name** and this container runs in the interactive mode, allocates a pseudo-tty and runs in the detached mode by using **-i, -t** and **-d** parameters respectively.

Since apache runs on the default port 80 it has been mapped to port 8080 of the docker host using **-p** parameter so that it can accessed by the outside world. Here we are mounting the contents of website directory onto the containers /var/www/html directory which is the path of apache webservers website content using volumes **-v** parameter. Since both the contents are binded together changes made in one paths index.html file reflects in another path and also on the website content. At the end we will be using the image name **apacheimage** which has been created using Dockerfile and finally executes our command.

Once the command has been executed Docker engine will spinning up a container with a random container id. To check the status of our container run **docker ps** command which shows the details of our container such as container id, name of the

container, image used, and ports exposed. Website deployed inside our container can be accessed using docker host's hostname followed by the port mapped on the host on your favorite web browser, in this case hostname and port are **srinathreddy455.mylabserver.com** and **8080** respectively. Below figure shows our website content on google chrome's web browser where our docker host's hostname and port are highlighted in the yellow color:

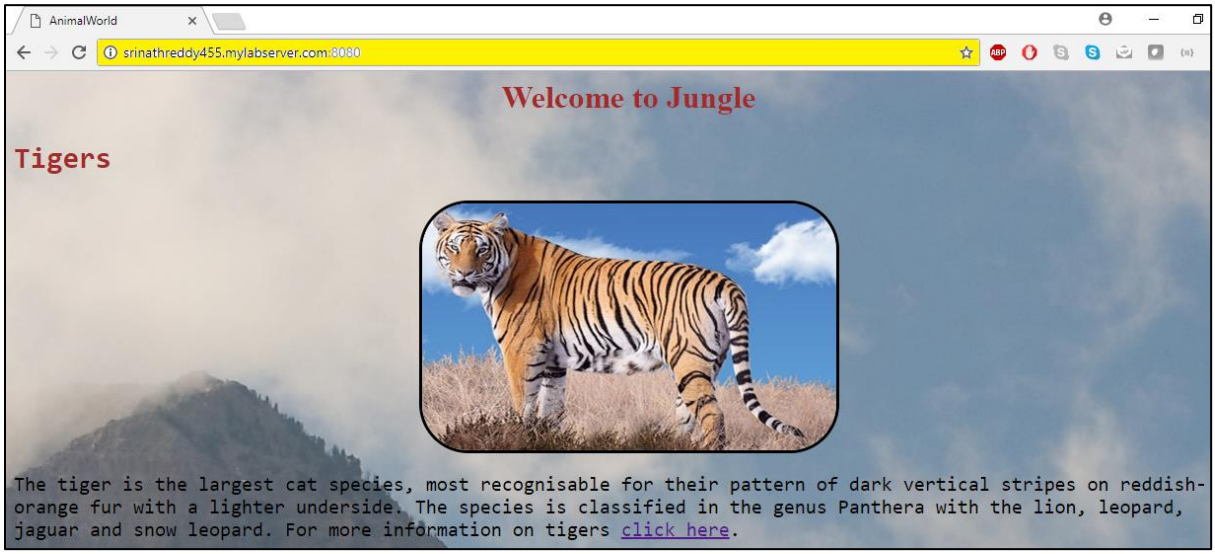

Figure 24. Accessing the Website Using Hostname

After verifying the web content has been served as expected we can push our **apacheimage** to our dockerhub repository to reproduce the similar environment on other Docker hosts. Before that we need to login to our dockerhub repository using **docker login** command and provide username and password of our repository which has been shown in below figure:

```
[user@srinathreddy455 website]$
[user@srinathreddy455 website]$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com
to create one.
Username: srinathreddy45
Password:
Login Succeeded
[user@srinathreddy455 website]$ docker tag apacheimage srinathreddy45/apacheimage
[user@srinathreddy455 website]$ docker push srinathreddy45/apacheimage
The push refers to repository [docker.io/srinathreddy45/apacheimage]
1e6478c12d0c: Pushed
e3fb0769f9aa: Pushed
db584c622b50: Pushed
52a7ea2bb533: Pushed
52f389ea437e: Pushed
88888b9b1b5b: Pushed
a94e0d5a7c40: Pushed
latest: digest: sha256:a67e438d9c3b529c9cece2dc977538ee002dd878948d9e68f1b7827aa1c80789 size: 1781
[user@srinathreddy455 website]$
```

Figure 25. Pushing the Image to Docker Hub

Also tag the **apacheimage** name with our dockerhub login id using **docker tag** command and then push your image to the repository using **docker push** command. To verify if the image has been pushed to your repository visit dockerhub website and then look for your image name which has been shown in below figure highlighted in yellow:

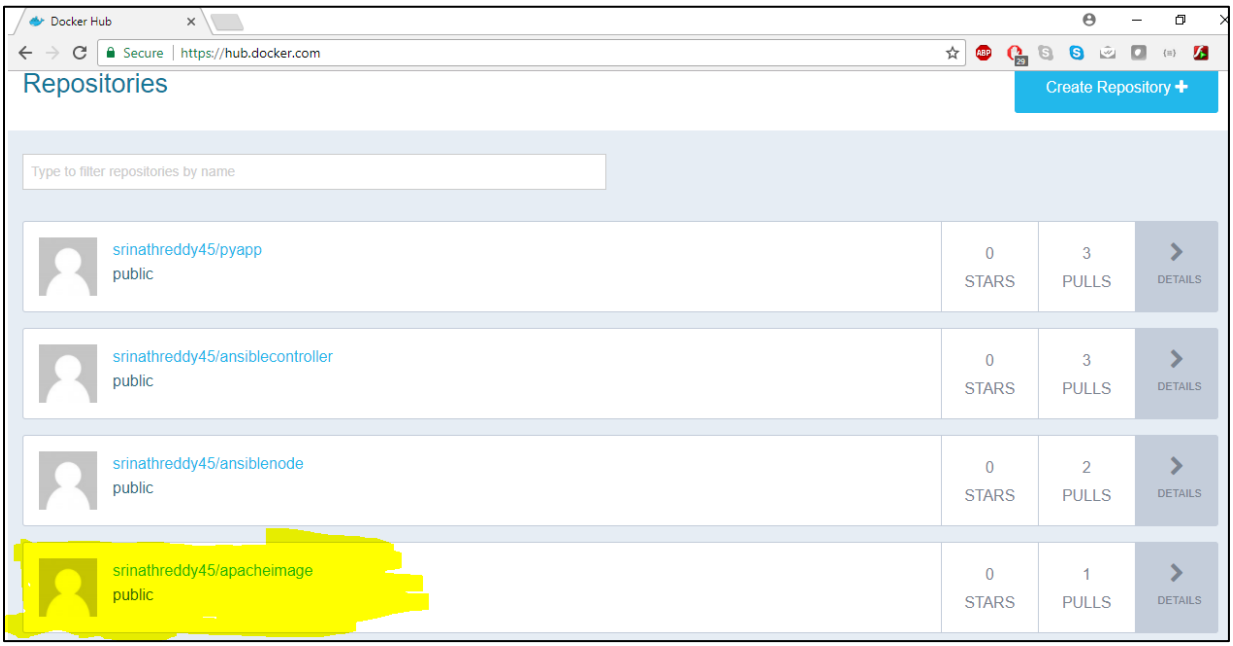

Figure 26. Verifying the Push on Docker Hub

If we want to change the content of our website, we can simply change the code in index.html file on our docker host. Since it is mapped to the container our website content will be changed automatically once we refresh our webpage. Here I have changed the image of tiger from Siberian tiger to Bengal tiger and this change has been immediately affected once we refresh our webpage which has been shown in below figure:

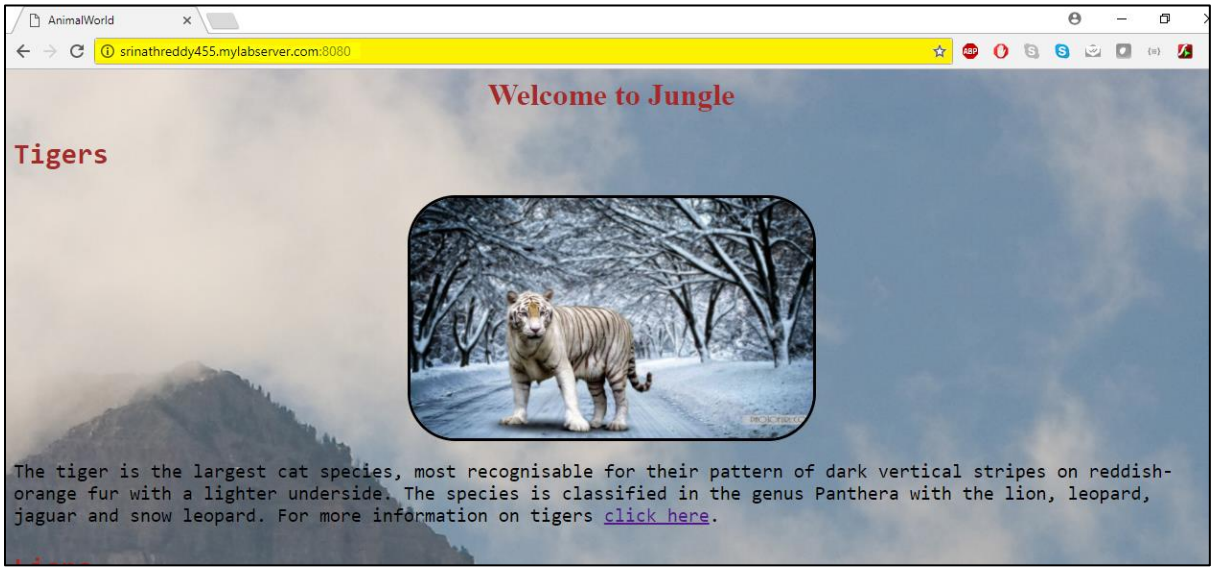

Figure 27. Updated Web Content

By leveraging the volumes, we could deliver the changes to our website in a fast-paced environment which has been evident in the above figure.

Previously we have only pushed the plain apacheimage where our website has been not deployed yet. After verifying the content, we can create an image out of our container using **docker commit** command followed by our container name and desired image name which has been shown below figure:

```
[user@srinathreddy455 website]$
[user@srinathreddy455 website]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                  NAMES
2c5d426474d4        apacheimage         "/bin/sh -c 'apachec…"   25 hours ago        Up 13 minutes       0.0.0.0:8080->80/tcp   apache
container
[user@srinathreddy455 website]$ docker commit apachecontainer srinathreddy45/apacheimage:1.0
sha256:7fa08dff3c991259e2fd2c7cbb184aae93f91f455a99485a8dbb779201f67623
[user@srinathreddy455 website]$ docker push srinathreddy45/apacheimage:1.0
The push refers to repository [docker.io/srinathreddy45/apacheimage]
7e7e44355bd4: Pushed
1e6478c12d0c: Pushed
e3fb0769f9aa: Pushed
db584c622b50: Pushed
52a7ea2bb533: Pushed
52f389ea437e: Pushed
88888b9b1b5b: Pushed
a94e0d5a7c40: Pushed
1.0: digest: sha256:377067340d821472b9bc8fd4c26cbb5a0ecfa13e74b4baf7c1932c6dfe7e1d57 size: 1988
[user@srinathreddy455 website]$
```

Figure 28. Pushing the Updated Image Content to Docker Hub

Here 1.0 denotes the version of our image which is used to identify the changes to our web content. Once the image has been created we will be pushing the image to our dockerhub repo using **docker push.** We can verify the push by visiting dockerhub as shown in below figure:

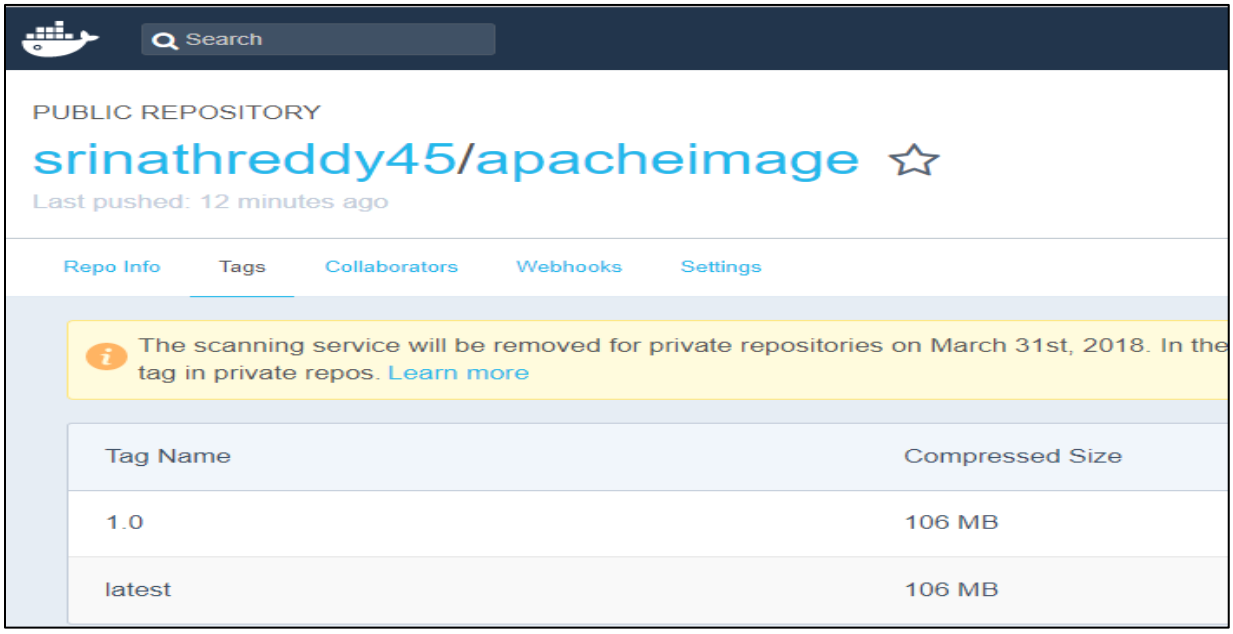

Figure 29. Verifying the Updated Push on Docker Hub

Scaling the Website Using Docker Swarm

Suppose if the server where we deployed our website has been crashed because of the traffic overload or internal system issue, our website will be down until we fix the issue. But in today's competitive world down time is considered as a major setback for an organization which could lose the credibility of the customers. So, it is always best practice to deploy your application in more than one server.

We will be leveraging docker swarm mode to deploy our application in more than one server with docker engine installed and running on each of the server. The group of these servers can be referred to as cluster which consists of at least one manager and worker. For our project I have used three servers with docker engine installed on them where one of the servers acts as manager while rest two act as workers. Details of the servers and their roles are listed in the below table:

Table 3. Manager and Worker Nodes

Hostname	Role
srinathreddy452.mylabserver.com	Manager
srinathreddy453.mylabserver.com	Worker1
srinathreddy454.mylabserver.com	Worker2

Initializing the swarm. First, we need to initialize the swarm mode in our manager node, before that we need to know the IP address of our manager server which can be used while using **init** command. To get the IP address we will be using **ifconfig** command which has been shown in below figure:

```
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 9001
        inet 172.31.119.211  netmask 255.255.240.0  broadcast 172.31.127.255
        inet6 fe80::76:92ff:fe77:6542  prefixlen 64  scopeid 0x20<link>
        ether 02:76:92:77:65:42  txqueuelen 1000  (Ethernet)
        RX packets 7744  bytes 8370467 (7.9 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3601  bytes 478931 (467.7 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 30. Retrieving the Manger IP Address

After retrieving the IP address of the manager node use **docker swarm init --advertise <ip_address>** command on our manager server to initialize the swarm mode which has been shown in below figure:

```
[manager@srinathreddy452 ~]$
[manager@srinathreddy452 ~]$ docker swarm init --advertise-addr 172.31.119.211
Swarm initialized: current node (ol78d7rvolylc9cadjh88h0bd) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-581hl57h88hggrn4e06bem31xbr8qzv7q9f4xcs0toapfr0ogm-4dcgbjwljzrzl2ti6uy6u3cjq 172.31.119.211:237
7

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

[manager@srinathreddy452 ~]$
```

Figure 31. Initializing the Swarm on Manager Node

A token gets generated from the manager after initializing the swarm mode, which can be used by the workers to get registered with the manager. Use **docker swarm join --token <token>** command on each of our workers as shown in below figures:

```
[worker1@srinathreddy453 ~]$
[worker1@srinathreddy453 ~]$ docker swarm join --token SWMTKN-1-581hl57h88hggrn4e06bem31xbr8qzv7q9f4xcs0toapfr0ogm-4dcgbjwljzrzl2ti6uy
6u3cjq 172.31.119.211:2377
This node joined a swarm as a worker.
[worker1@srinathreddy453 ~]$
```

Figure 32. Joining Worker1 to Swarm

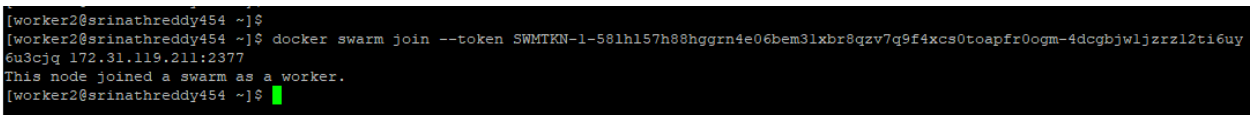

Figure 33. Joining Worker2 to Swarm

We will be getting a confirmation message from the node like “this node joined a swarm as a worker” after joining the swarm. One can check the swarm status and the nodes details of the cluster using **docker node ls** command where **srinathreddy452.mylabserver.com** manager status is shown as leader since the swarm mode has been initialized in this server.

Figure 34. Verifying Nodes Status

Deploying the website in swarm. As our swarm is setup now, we are ready to deploy our website in this cluster for achieving high availability and reliability. Each task deployed in the cluster is considered as service and service has a name which can be further used for debugging purposes. We will be **docker service create** command to deploy our website in the cluster and this has been depicted in below figure:

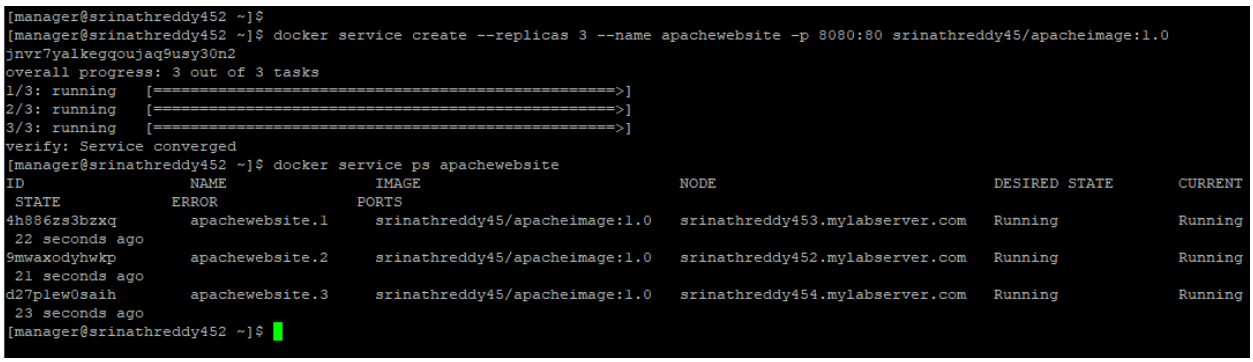

Figure 35. Creating the Service on Our Swarm

Here we are deploying our website in three container which is denoted by **--replicas** parameter with each container running on our three nodes of the swarm and **--name** parameter can initialize our service name, in our case which is **apachewebsite.** Containers ports are being exposed to docker engine host using **-p** parameter so that website deployed can be accessed by outside world.

Finally, we will be giving the image name where our application is packaged, here it is **srinathreddy45/apacheimage:1.0** which has been created in the previous section. Once we execute the command it would take few minutes to deploy our website on all the hosts inside the containers. To check the status of our service use **docker service ps <service_name>** command so that it could display the heath of the server and the container IDs where this service has been deployed. To verify if our service has been deployed run **docker ps** command on the manager and worker nodes and check the containers as shown in below figures:

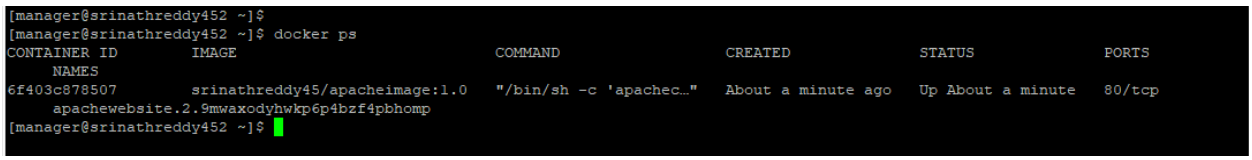

Figure 36. Verifying Service Status on Manager Node

Figure 37. Verifying Service Status on Worker1 Node

Figure 38. Verifying Service Status on Worker2 Node

To access the website deployed in our swarm we can go a web browser and we can access it by using the hostname name or IP address of any node in the cluster followed by the port mapped which is 8080 in our case. The following figures denotes the same where the website accessed using the IP addresses of our manger node and one of the worker node.

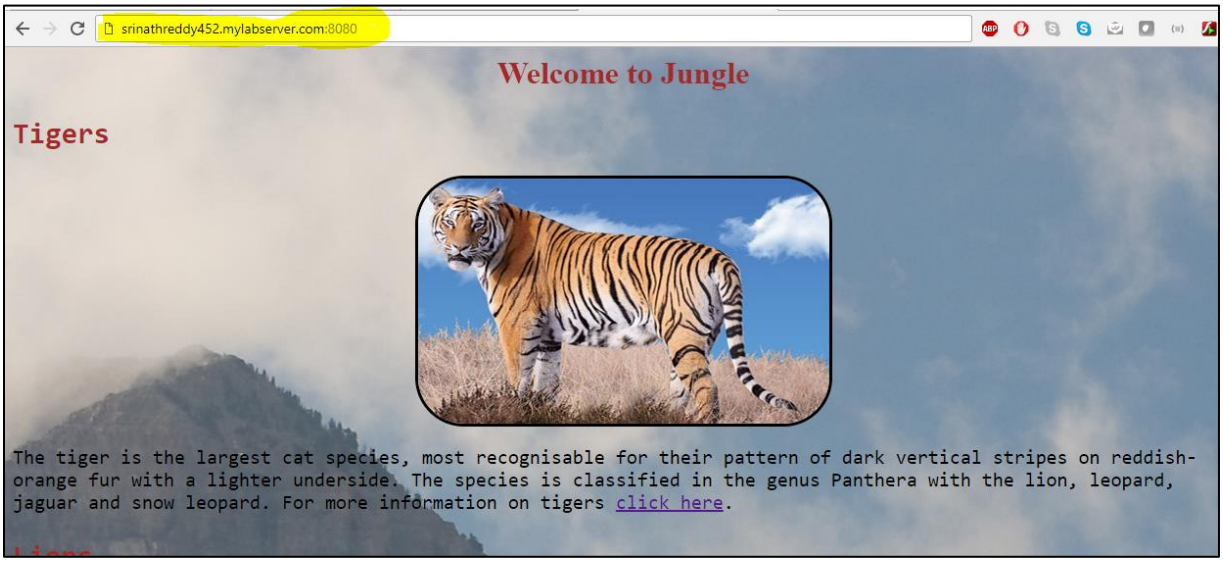

Figure 39. Accessing the Website Using Manger Node Hostname

Figure 40. Accessing the Website Using Worker1 Node Hostname

The configurations of our service can be retrieved using **docker service inspect** command followed by our service name.

```
[manager@srinathreddy452 ~]$
[manager@srinathreddy452 ~]$ docker service inspect --pretty apachewebsite

ID:             jnvr7yalkegqoujaq9usy30n2
Name:           apachewebsite
Service Mode:   Replicated
 Replicas:      3
Placement:
UpdateConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Update order:      stop-first
RollbackConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Rollback order:    stop-first
ContainerSpec:
 Image:         srinathreddy45/apacheimage:1.0@sha256:5e5616cf3912ed9e356deb12fe6611f5f558316b44190fedbad79dc82bdf1f74
Resources:
Endpoint Mode:  vip
Ports:
 PublishedPort = 8080
  Protocol = tcp
  TargetPort = 80
  PublishMode = ingress
[manager@srinathreddy452 ~]$
```

Figure 41. Inspecting the Service

Updating the contents of the website. Once our website has been deployed and serving the traffic in swarm mode successfully we might need to make some changes to our web content to attract more users. In our case I have changed the image of tiger from **Siberian** to **Bengal** tiger in our website and then created a new docker image from Dockerfile after making the changes to index.html file.

```
[user@srinathreddy455 App]$
[user@srinathreddy455 App]$ docker build -t  srinathreddy45/apacheimage:2.0 .
Sending build context to Docker daemon  5.632kB
Step 1/7 : FROM ubuntu
 ---> f975c5035748
Step 2/7 : MAINTAINER srinath_reddy
 ---> Using cache
 ---> 55960e0f5f19
Step 3/7 : COPY /website/index.html /var/www/html/
 ---> 839668b68a42
Step 4/7 : RUN apt-get update
 ---> Running in 7bb96f4bb7f3
Get:1 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
Get:2 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
Get:3 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]
```

Figure 42. Building the Updated Image

Once the image has been built it has been pushed to our docker hub repository so that it can accessed by our swarm to update all the containers which are hosting our website.

```
[user@srinathreddy455 App]$
[user@srinathreddy455 App]$ docker images
REPOSITORY                    TAG                 IMAGE ID            CREATED             SIZE
srinathreddy45/apacheimage    2.0                 bfbf606bfde8        45 seconds ago      251MB
srinathreddy45/apacheimage    1.0                 049a85c5eade        45 minutes ago      251MB
srinathreddy45/apacheimage    latest              2f182d068ce5        About an hour ago   251MB
ubuntu                        latest              f975c5035748        5 days ago          112MB
[user@srinathreddy455 App]$ docker push srinathreddy45/apacheimage:2.0
The push refers to repository [docker.io/srinathreddy45/apacheimage]
e0f185d4d577: Pushed
e777e4e3ca8b: Pushed
c40a14c1075b: Pushed
db584c622b50: Layer already exists
52a7ea2bb533: Layer already exists
52f389ea437e: Layer already exists
88888b9b1b5b: Layer already exists
a94e0d5a7c40: Layer already exists
2.0: digest: sha256:294e3a45bedad4461bf45b9eb7596e55fdb6e81fbda97b9ba056ef9664ceb2ce size: 1989
[user@srinathreddy455 App]$
```

Figure 43. Pushing the Updated Image to Docker Hub

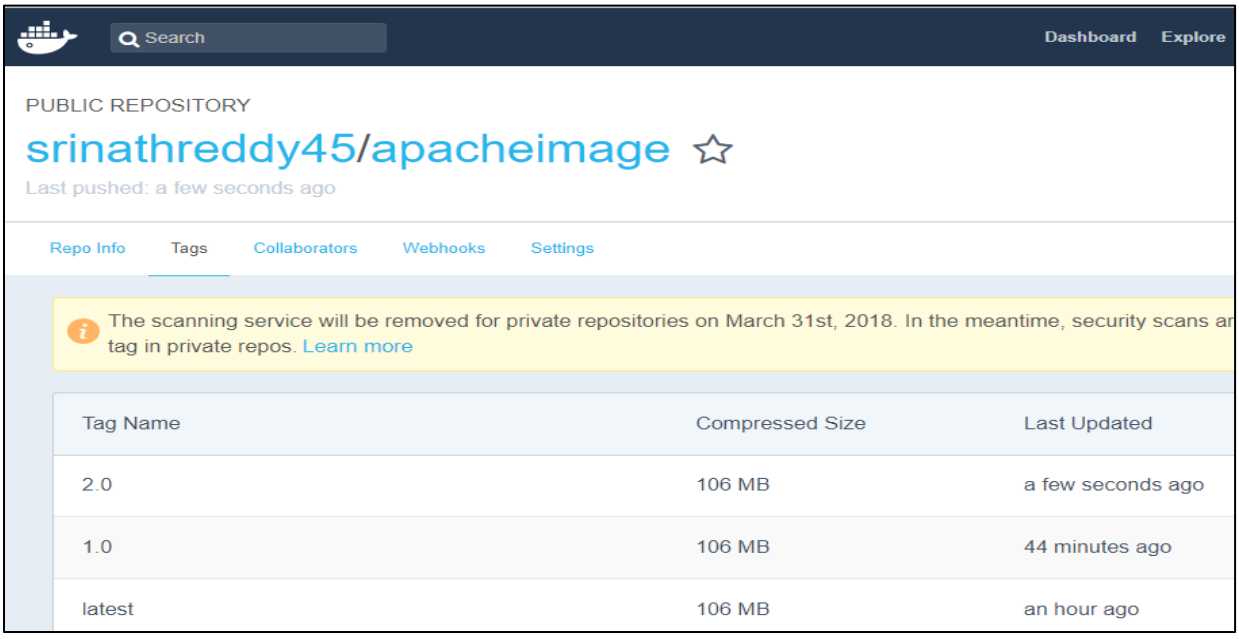

Figure 44. Verifying the Updated Image on Docker Hub

After pushing the image to docker hub we will be doing rolling update on our containers using docker service update command followed by image name and service name which has been shown in below figure:

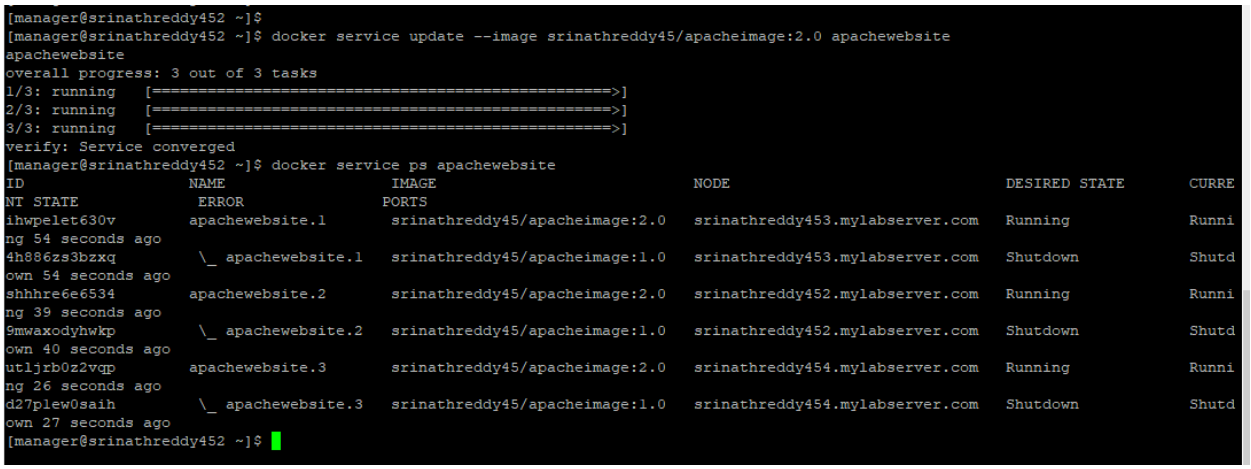

Figure 45. Rolling Update of Image on all the Containers

Here we can see that updates have been affected on the containers one at a time to ensure zero percent down time. One more thing to observe is the previous containers which have old image have been shutdown and the new containers have been spanned up to run the latest image. We can see in the below figure that our website content has been now changed with the **Bengal** tiger.

Figure 46. Accessing the Updated Website Using Manger Node Hostname

Figure 47. Accessing the Updated Website Using Worker2 Node Hostname

Scaling up the number of containers. Suppose after making above changes to our web content if the number of users has been increased there be might chance that our containers hosting our website might go down because of the increase in load. To avoid this, we can increase the number of containers hosting our website using **docker**

service scale command by number of containers we want to be deployed in our swarm as shown in below figure.

```
[manager@srinathreddy452 ~]$
[manager@srinathreddy452 ~]$ docker service scale apachewebsite=6
apachewebsite scaled to 6
overall progress: 6 out of 6 tasks
1/6: running   [==================================================>]
2/6: running   [==================================================>]
3/6: running   [==================================================>]
4/6: running   [==================================================>]
5/6: running   [==================================================>]
6/6: running   [==================================================>]
verify: Service converged
[manager@srinathreddy452 ~]$ docker service ps apachewebsite
ID                  NAME                IMAGE                                NODE                              DESIRED STATE       CURRE
NT STATE            ERROR               PORTS
ihwpelet630v        apachewebsite.1     srinathreddy45/apacheimage:2.0       srinathreddy453.mylabserver.com   Running             Runni
ng 12 minutes ago
4h886zs3bzxq         \_ apachewebsite.1 srinathreddy45/apacheimage:1.0       srinathreddy453.mylabserver.com   Shutdown            Shutd
own 12 minutes ago
shhhre6e6534        apachewebsite.2     srinathreddy45/apacheimage:2.0       srinathreddy452.mylabserver.com   Running             Runni
ng 12 minutes ago
9mwaxodyhwkp         \_ apachewebsite.2 srinathreddy45/apacheimage:1.0       srinathreddy452.mylabserver.com   Shutdown            Shutd
own 12 minutes ago
utljrb0z2vqp        apachewebsite.3     srinathreddy45/apacheimage:2.0       srinathreddy454.mylabserver.com   Running             Runni
ng 12 minutes ago
d27plew0saih         \_ apachewebsite.3 srinathreddy45/apacheimage:1.0       srinathreddy454.mylabserver.com   Shutdown            Shutd
own 12 minutes ago
binmljpok5xg        apachewebsite.4     srinathreddy45/apacheimage:2.0       srinathreddy452.mylabserver.com   Running             Runni
ng 31 seconds ago
z2c3vs4o95kr        apachewebsite.5     srinathreddy45/apacheimage:2.0       srinathreddy454.mylabserver.com   Running             Runni
ng 32 seconds ago
k4sp6yz8ln62        apachewebsite.6     srinathreddy45/apacheimage:2.0       srinathreddy453.mylabserver.com   Running             Runni
ng 31 seconds ago
```

Figure 48. Scaling the Number of Containers Hosting Our Website

Here I have scaled up to six containers and we can see that a new container has been spanned up in each of the nodes in swarm with each node now hosting two containers on them. To verify, run **docker ps** command has been ran on each node as shown in below figure:

```
[worker1@srinathreddy453 ~]$
[worker1@srinathreddy453 ~]$ docker ps
CONTAINER ID        IMAGE                           COMMAND                  CREATED             STATUS              PORTS
    NAMES
a916596ead49        srinathreddy45/apacheimage:2.0   "/bin/sh -c 'apachec…"   4 minutes ago       Up 4 minutes        80/tcp
    apachewebsite.6.k4sp6yz8ln62vvhkwk8xi89k9
88bd90773015        srinathreddy45/apacheimage:2.0   "/bin/sh -c 'apachec…"   16 minutes ago      Up 16 minutes       80/tcp
    apachewebsite.1.ihwpelet630vraz8c6ftph7kx
[worker1@srinathreddy453 ~]$
```

Figure 49. Verifying the Number of Containers Running on Worker1

Draining a node in the swarm. It is always a best practice to keep load as low as possible on our manager node since performance issues on our manager node

would make our entire website to go down. So, we can keep our manager in drain mode so that it won't take any tasks which could increase the load. To achieve this, we will be using **docker node update** command followed by the hostname of our manager as shown in the below figure:

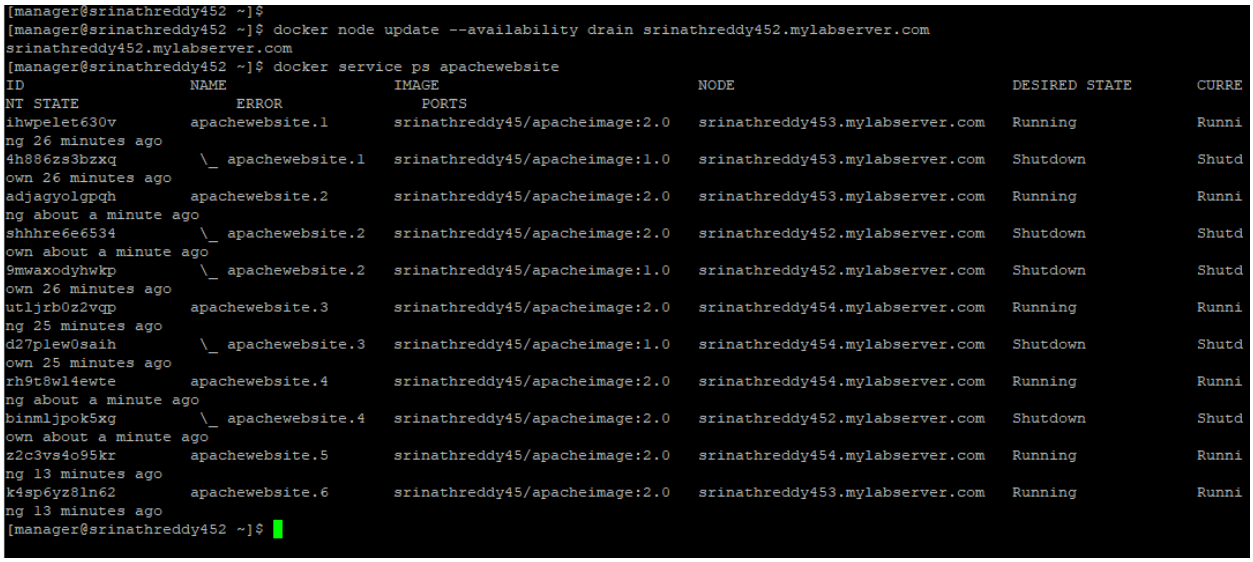

Figure 50. Draining a Node

We can also see that containers running on our manager node has been shutdown and the new containers have been spanned on our worker nodes. We can verify the status of our nodes using **docker node ls** command which would show the availability of our manager node as drain.

Figure 51. Verifying Nodes Status

We can verify the capacity of number of containers hosting our application by running **docker ps** command on each of our worker nodes as shown in below figures:

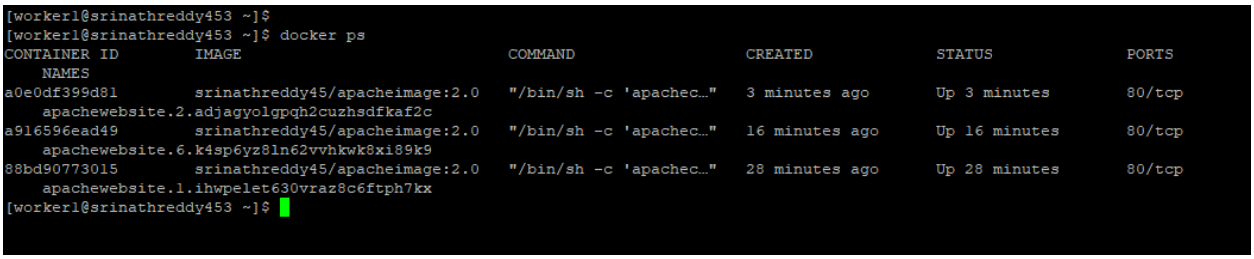

Figure 52. Verifying Capacity on Worker1

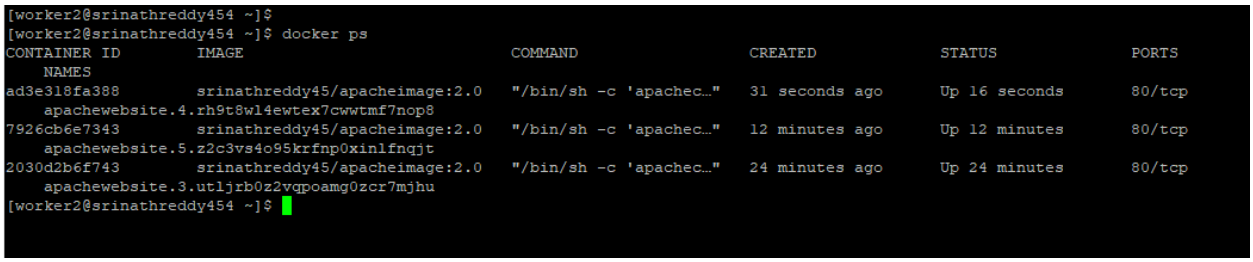

Figure 53. Verifying Capacity on Worker2

From the above project it is evident that it is easy to deploy an application using docker containers and it can be scaled up within no time to decrease the down time of an application. It is also evident that using docker containers an application can be packaged, shipped, and delivered across different environments within no time using the out of box features of docker.

Docker Security

Although there are certain rich features from Docker which we have discussed in above sections, there are few downsides in using docker containers when considering

its security. However, these security vulnerabilities can be avoided by following some of the industry's best practices for securing the containerized applications.

Risk of privilege escalation is very high when using containers, for example if an attacker can become a root user inside the container it would be a cake walk for him/her to gain the root access of host system which could be disastrous for the entire Dev/Prod environment. To avoid this, it is always a best practice to run the docker containers with **-u** flag so that they run as ordinary user instead of root user. Spanning up the containers on traditional hypervisors such as KVM and Hyper-V would mitigate the risk of privilege escalation as virtual environment is strictly abstracted from the host system [11].

One needs to careful while using images from public repositories, these unsecure images would become absolute threat which would compromise the infrastructure of our application. It always a best practice to use official images which are approved by the docker itself or one can build their own images by leveraging Dockerfile.

Another thing which needs to be considered while using containers is Denial of Service(DoS) attack, where one compromised container can seize the operability of other containers by denying the host system resources to other containers. Using Cgroups and namespaces can avoid this vulnerability, since cgroups sets the limitations on resources that a container can use while name spaces isolates one container from the other [11].

While container security is no longer a concern in an application development lifecycle but it is always a best practice to be more vigilant while spanning up the containers. We can also consider using third party tools which would scan our docker servers and detect the vulnerabilities. By taking proactive approach and by creating strict security polices one can easily prevent the vulnerabilities in a containerized environment.

Chapter V: Conclusion

From this paper we can concluded that Docker containers have made application lifecycle development easy in all the environments such as develop, test and production. It also evident that Docker can be used to reproduce the environments on our local desktops or remote servers within no time to test and deploy our application without any additional installation of tools which could consume the resources of the machines. This paper also concludes that application downtime can be decreased by scaling up the number of containers within no time where the application is hosted in those containers.

Future Work

In this paper I have primarily focused on Docker containers and it is out of box features used for faster delivery and containerization of our applications. In the future I would like to work on automating the creation of containers using scripts and several other automation tools such as Jenkins where we can achieve Continuous Integration and Continuous Delivery while building an application. Apart from this I would also like to work on several other third-party cluster management tools such as Apache Mesos and Kubernetes which are much more sophisticated and advanced than the Docker's own swarm mode which has been discussed in this paper.

References

[1] R. Chamberlain and J. Schommer, “Using Docker to Support Reproducible Research,” 2014.

[2] C. Wang, Chenxi, “Containers 101: Linux containers and Docker explained,” 2016, http://www.infoworld.com/article/3072929/linux/containers-101-linux-containers-and-docker-explained.html.

[3] G. Henningsen, “How I used cgroups to manage system resources in Oracle Linux 6,” 2012, http://www.oracle.com/technetwork/articles/servers-storage-admin/resource-controllers-linux-1506602.html.

[4] J. Turnbull, “The Docker Book,” 2014, https://dockerbook.com/#toc.

[5] C. Boettiger, “An introduction to Docker Reproducible Research with Examples from R Environment,” 2014.

[6] “What are containers,” 2015, https://www.sdxcentral.com/cloud/containers/definitions/what-are-containers-like-docker-linux-containers/.

[7] “Docker overview,” 2016, Official docs: https://docs.docker.com/engine/docker-overview/.

[8] https://docs.Docker. com/ datacenter/ucp/2.2/ guides/admin/configure/integrate-with-dtr/#2-test-your-local-setup).

[9] M. Church, “Understanding Docker networking drivers and their use cases,” 2016, https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/.

[10] E. Mavungu, “Docker storage: An introduction,” 2017, https://blog.codeship.com/docker-storage-introduction/.

[11] C. Tozzi, “Securing Docker containers,” 216, https://www.sumologic.com/blog/security/securing-docker-containers/.

Appendix

Following is the sample code which has been used to build our animal world website and this code has been written in simple HTML and CSS:

```
<html>
<style>
.fonting-conf{
font-family:monospace;
font-size:20px;
}
 p{
color:black;
}
h1{
font-family:"times",fantasy;
color:brown;
}
h2{
font-family:monospace;
color:brown;
font-style:solid;
font-size:30;
}
```

```
body{
background-image:url("https://upload.wikimedia.org/wikipedia/commons/c/c7/Uinta-
national-forest-banner02.jpg");
background-color:gray;
}
.border-image{
border-style:solid;
border-color:black;
border-radius:50px;
}
</style>
<head>
<title>AnimalWorld</title>
</head>
<body>
<center>
<h1>
This webpage is for animals
</h1>
</center>
<h2>Tigers</h2>
<center>
```

```
<img class="border-image"
src="http://www.irctctourism.com/ttrs/railtourism/images/CDR03.jpg" alt="tiger is
sleeping"/>
</center>
<p class="fonting-conf">
The tiger is the largest cat species, most recognisable for their pattern of dark vertical
stripes on reddish-orange fur with a lighter underside. The species
 is classified in the genus Panthera with the lion, leopard, jaguar and snow leopard.
 For more information on tigers <a href="https://en.wikipedia.org/wiki/Tiger"
target="_blank">click here</a>.
</p>
<h2>Lions</h2>
<center>
<img class="border-image"
src="https://cbs.umn.edu/sites/cbs.umn.edu/files/public/african_lion_king-wide.jpg"
alt="lion" style="width:420px;height:250px;"/>
</center>
<p class="fonting-conf">
The lion (Panthera leo) is one of the big cats in the genus Panthera and a member of
the family Felidae.
The commonly used term African lion collectively denotes the several subspecies in
Africa. With some males
```

exceeding 250 kg (550 lb) in weight,[5] it is the second-largest living cat after the tiger, barring hybrids

like the liger.[6][7] Wild lions currently exist in sub-Saharan Africa and in India (where an endangered

remnant population resides in and around Gir Forest National Park). In ancient historic times, their

range was in most of Africa, including North Africa, and across Eurasia from Greece and southeastern

Europe to India. For more information on Lions <a href="https://en.wikipedia.org/wiki/Lion" target="_blank">click here</a>.

</p>

</body>

</html>

For creating our containers following code has been scripted in Dockerfile:

```
FROM ubuntu

# File Author / Maintainer

MAINTAINER srinath_reddy
```

```
# Update the repository sources list

RUN apt-get update


# Install and run apache

RUN apt-get install -y apache2 && apt-get clean


EXPOSE 80

CMD apachectl -D FOREGROUND
```