STARRED PAPER DEFENSE

VIRTUALIZATION USING DOCKER CONTAINERS: FOR CREATING REPRODUCIBLE ENVIRONMENTS AND CONTAINERIZED APPLICATIONS

****

by

Srinath Reddy Meadusani

Information Assurance M.S, St. Cloud State University, St. Cloud, 2018

March 16th, 2018

Committee members:

1. Dr. Susantha Herath, Chair person

2. Dr. Ezzat Kirmani

3. Dr. Kasi Balasubramanian

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.

TABLE OF CONTENTS

Page

LIST OF TABLES........................................................................................... v

LIST OF FIGURES.......................................................................................... vi

Chapter

I. INTRODUCTION………………………………………………………………… 1

Introduction.......................................................................................... 1

Problem Statement……........................................................................ 2

Nature and Significance of the Problem............................................... 3

Objective of the Research…………………………………………………. 3

Research Questions and/or Hypotheses………………………………… 3

Definition of Terms................................................................................ 4

II. BACKGROUND AND REVIEW OF LITERATURE…………………………. 5

Introduction .......................................................................................... 5

Background Related to the Problem..................................................... 5

Literature Related to the Problem………….......................................... 6

Literature Related to the Methodology …………………………………. 13

III. METHODOLOGY ………………………………………………………….......... 16

Introduction ............................................................................................. 16

Design of the Study……………………………………………………...……16

Data Collection …….................................................................................21

Tools and Techniques………………………………………………………...21

Hardware and Software Requirements……….…………………………….22

WORK IN PROGRESS……………………………………………………………….23

TIMELINE………………………………………………………………………………24

REFERENCES....................................................................................................25

LIST OF TABLES

Table Page

1. Hardware Requirement………………………............................................ 21

2. Software Requirement…...….................................................................... 22

3. Timeline………………………………………………………………………… 23

LIST OF FIGURES

Figure Page

1. Virtualization using Virtual Machines............................................ 7

2. Cgroups in Linux Conatiners......................................................... 9

3. Containerization using Docker.................................................... 17

4. Containers in VMs using Docker…………………………………. 18

5. Docker Architecture…………………………………………………. 19

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/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” Ryan Chamberlain, 2014 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’s 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.

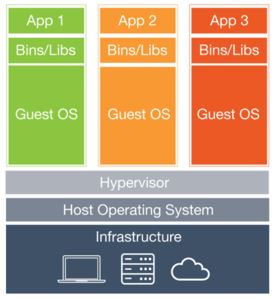


Figure 1. Virtualization Using Virtual Machines (Wang, 2106)

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 don’t 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 (Chenxi Wang, 2016). 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:

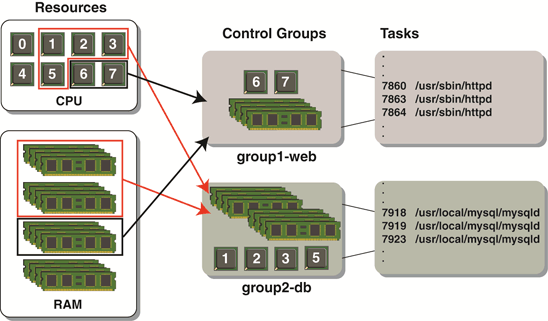


Figure 2. Cgroups in Linux Containers (Henningsen, 2012)

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 figure2, 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 process 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, an 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 (James Turnbull, 2014). 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 (Turnbull, 2014).

**Fast and efficient 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 (Boettiger, 2014). 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 (Docker Community, 2015).

Docker Containers also enhances security by providing Process restrictions, Device & 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.

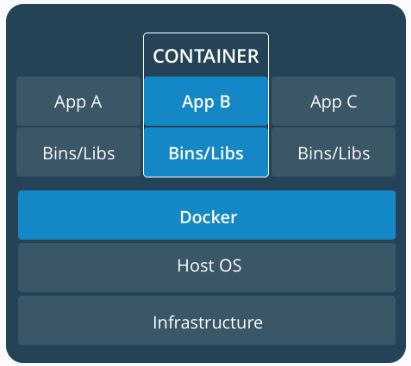


Figure 3 Containerization using Docker (Retrieved from: <https://www.docker.com/what-container>)

From above figure, it is evident that by using Docker we don’t 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 doesn’t 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:

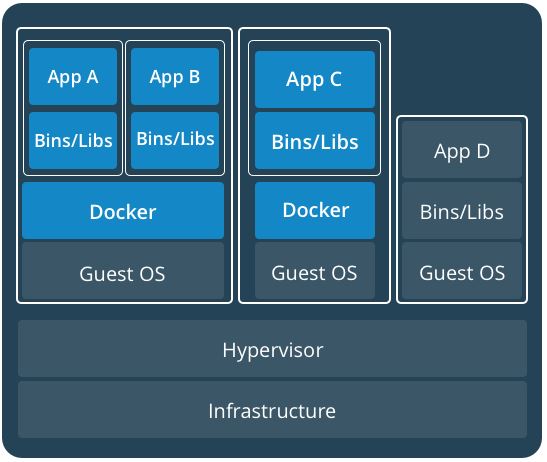


Figure 4 Containers in VMs using Docker (Retrieved from: <https://www.docker.com/what-container>)

From the above figure, it is evident that it’s 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:

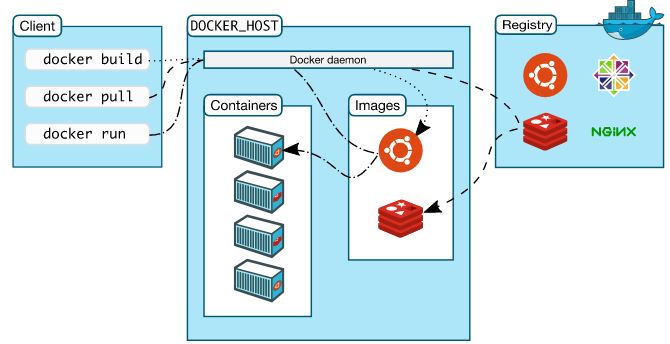


Figure 5 Docker Architecture (Retrieved from: <https://docs.docker.com/engine/docker-overview/#docker-architecture>)

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 (Turnbull, 2014). 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.

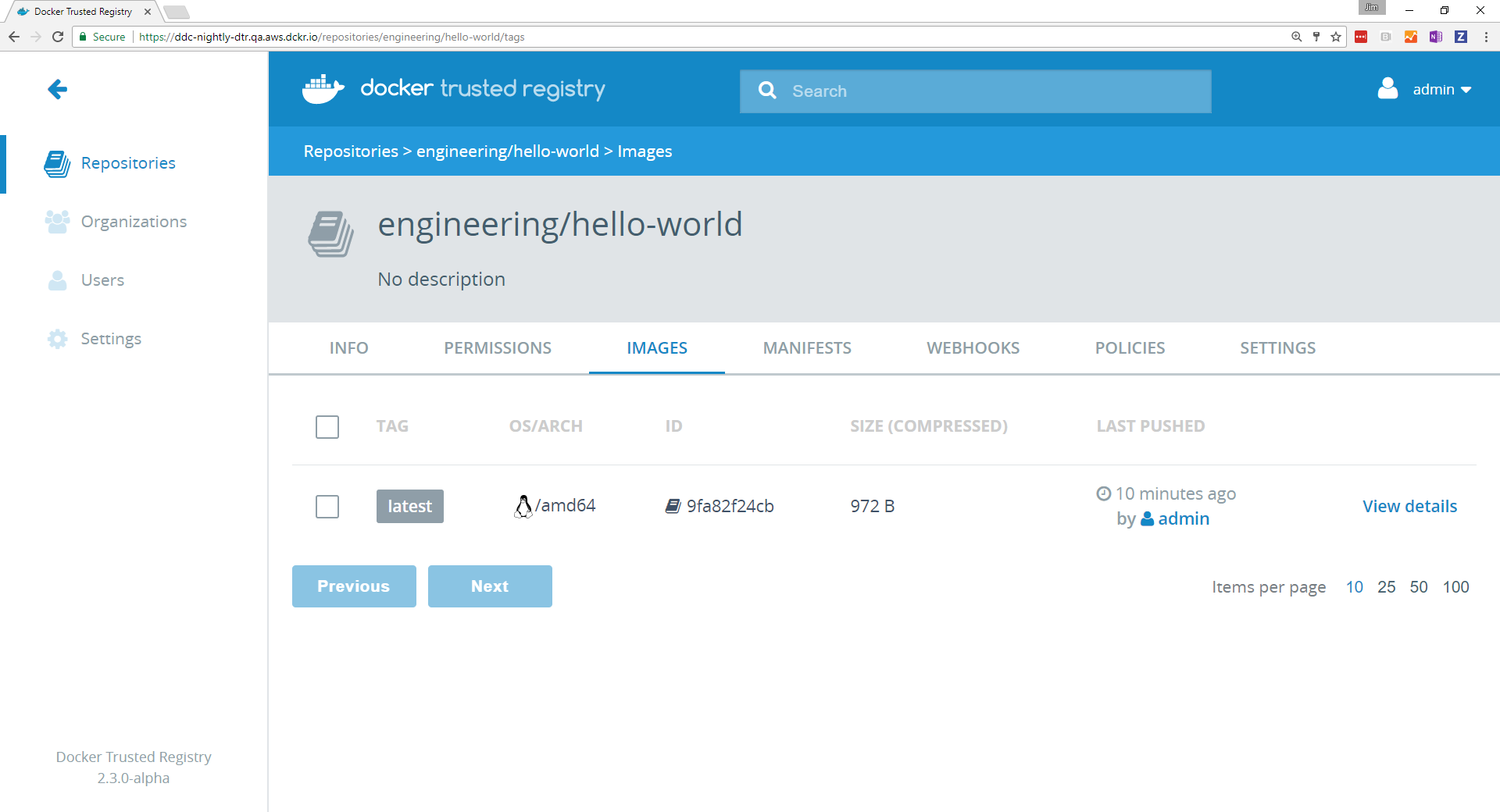


Figure 6 Web Interface for DTR ( Retrieved from https://docs.Docker. com/ datacenter/ucp/2.2/ guides/admin/configure/integrate-with-dtr/#2-test-your-local-setup)

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 Nagiostool 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

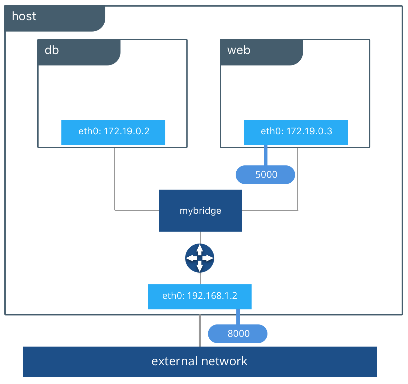
Introduction

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**

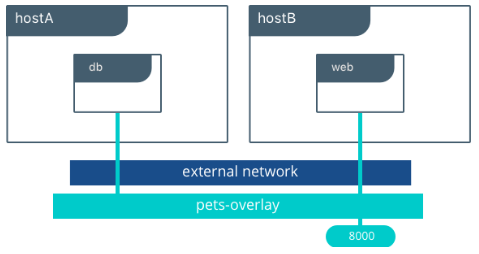
When a Docker engine is installed, and a container is spanned up on the host machine, bridge network is the default network that our container is created on. It is the most simple and easy to create networks on docker engine, which restricts its capability to single host. This creates a private internal 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. 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: 

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**

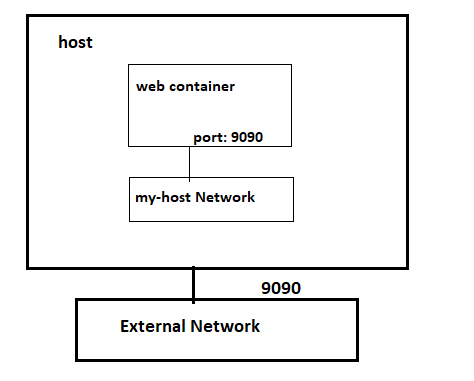
Overlay network is a built-in network driver which simplifies the complexity of hosting the containers on multiple hosts without any external provision or components. Load balancing between the containers, service discovery and multi-host connectivity are built right in this driver, which makes it one 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. Below figure depicts the powerful features of this driver:



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**

Hostnetworks is the simplest form of network drivers which doesn’t 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.



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.

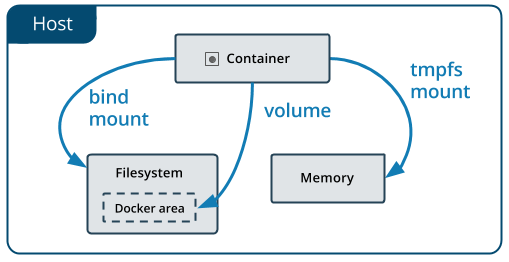


The macvlan driver can be configured in different ways to achieve different results. In the below example we create two MACVLAN networks joined to different sub interfaces. This type of configuration can be used to extend multiple L2 VLANs through the host interface directly to containers. The VLAN default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

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.



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. 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**

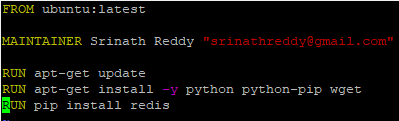
Bind mounts are similar to volumes where a file or directory is mounted into the containers when we use the bind mount. But this file or directory is referenced by its full path on the host machine and it doesn’t need to be already exist on the docker host. When the data of these files or directory changes 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.

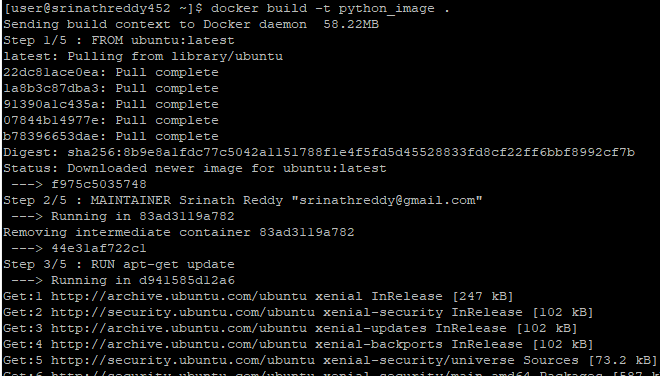
**Tmpfs Mounts**

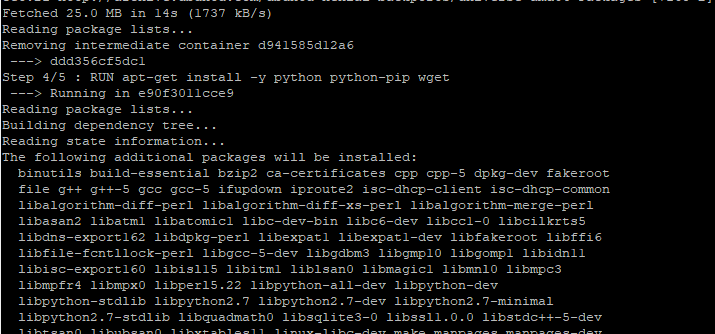
Tmpfs mounts can be best used in the scenario where you do not want the data to persistent either on the host or inside the container. This can be for security reasons or the performance of your container where your application writes a large amount of non-persistent data. Tmpfs mount store the data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

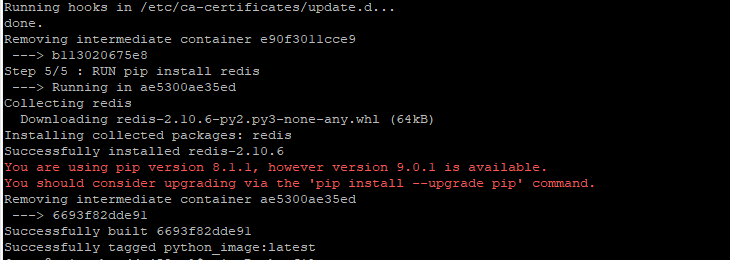
Dockerfile

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:









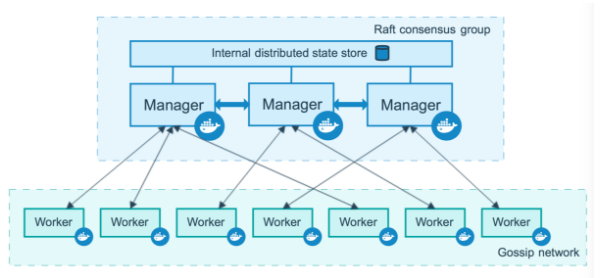
In the figure depicts a Dockerfile which creates a ubuntu image, updates the packages, installs python and wget and then with help of python’s pip library installs redis. Here we can see that for every step in Dockerfile it creates a new container and creates a new image out of that container and merges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



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 its 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. Its 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.

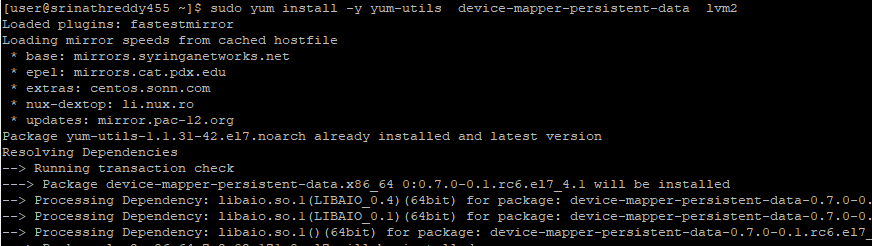
Docker Compose

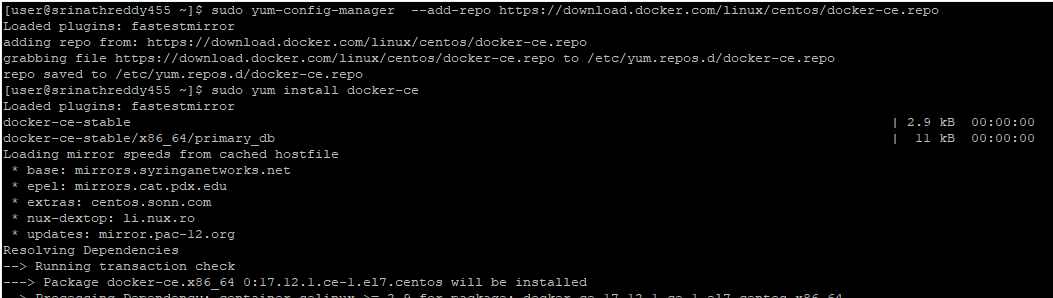
When your application needs more than one container which are isolated Docker compose would come in handy in building, running, and connecting those containers and entire setup can be done on single host. Docker compose is very useful in Dev and QA environments which reduces the overhead of maintaining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

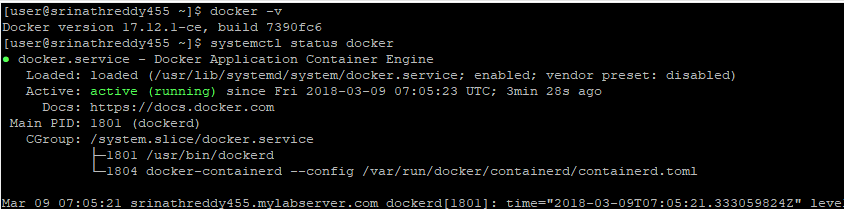
Setting Up the Environment and running containerized applications

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:





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”.*

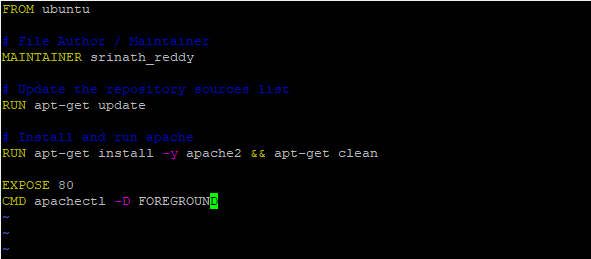


As carrying the activities on Linux machines without using the root user is considered as industry’s best practice, one need not to be a root or sudo user to run docker commands and to achieve this add user name to docker group using the following command ‘*usermod -aG docker <user\_name>’.*

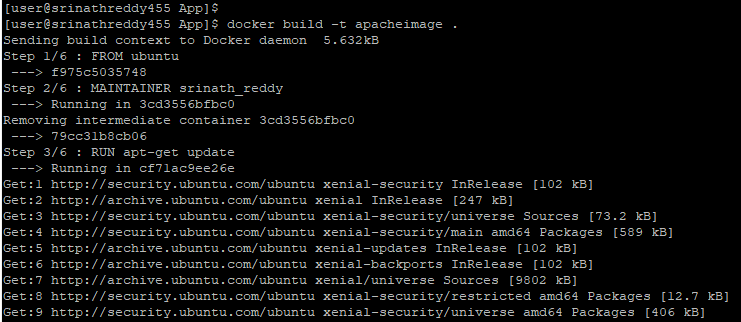
Spinning up an apache-based web container and exposing it to outside world

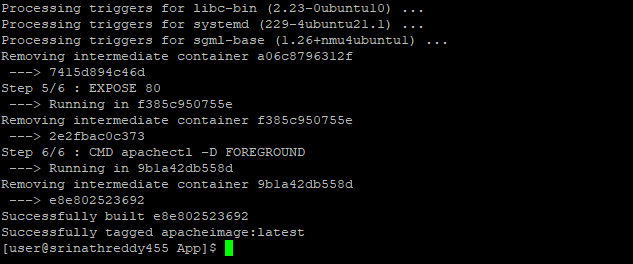
After installing the docker engine successfully we will be spinning up an apache-based web container and deploy our website in that container which can be accessed by the outside world. Here we will be creating a Dockerfile which consists of all the instructions which are needed to host our website on 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:

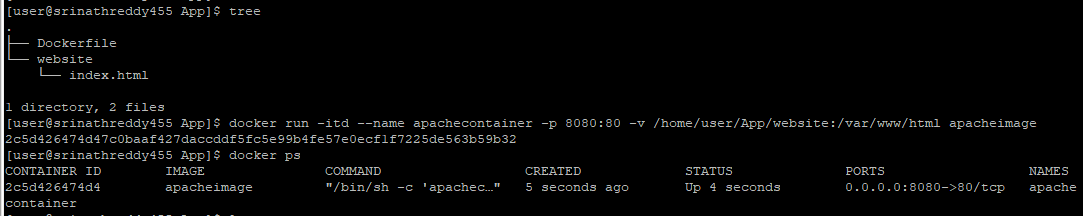


Here we are using ubuntu as our container OS and running the update command inside the conatiner followed by installing the apache webserver using **RUN** command and then exposing the port of the conatiner to outside host on which our apache server runs using **EXPOSE** command and then starting the 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:





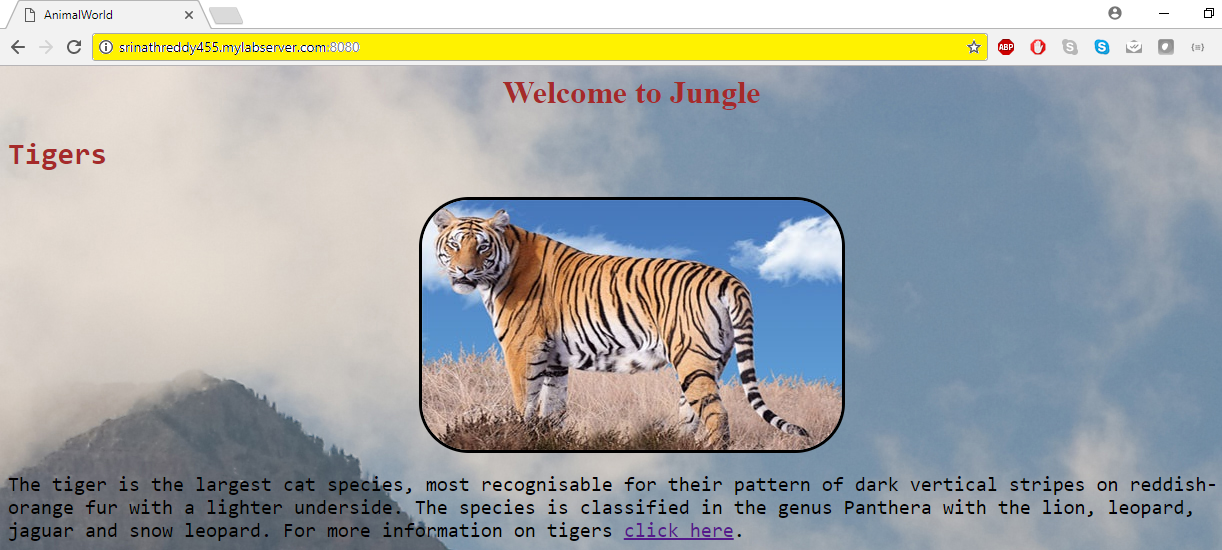
Since our image has been ready to use, now we should be able to spin our container to host our website. Before that I have create a directory called website where our website contents reside, and this can be viewed in below figures where I have used **tree** command for this purpose.



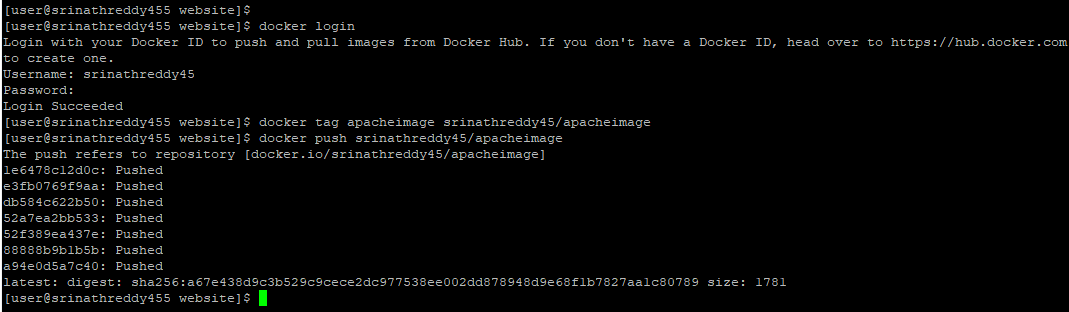
After creating the directory, we will be using **docker run** command and few other parameters to spin our container which has been shown in above figure. We have named our container as **apacheconatiner** using the parameter **--name** and this container runs in the interactive mode, allocates a pseudo-tty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

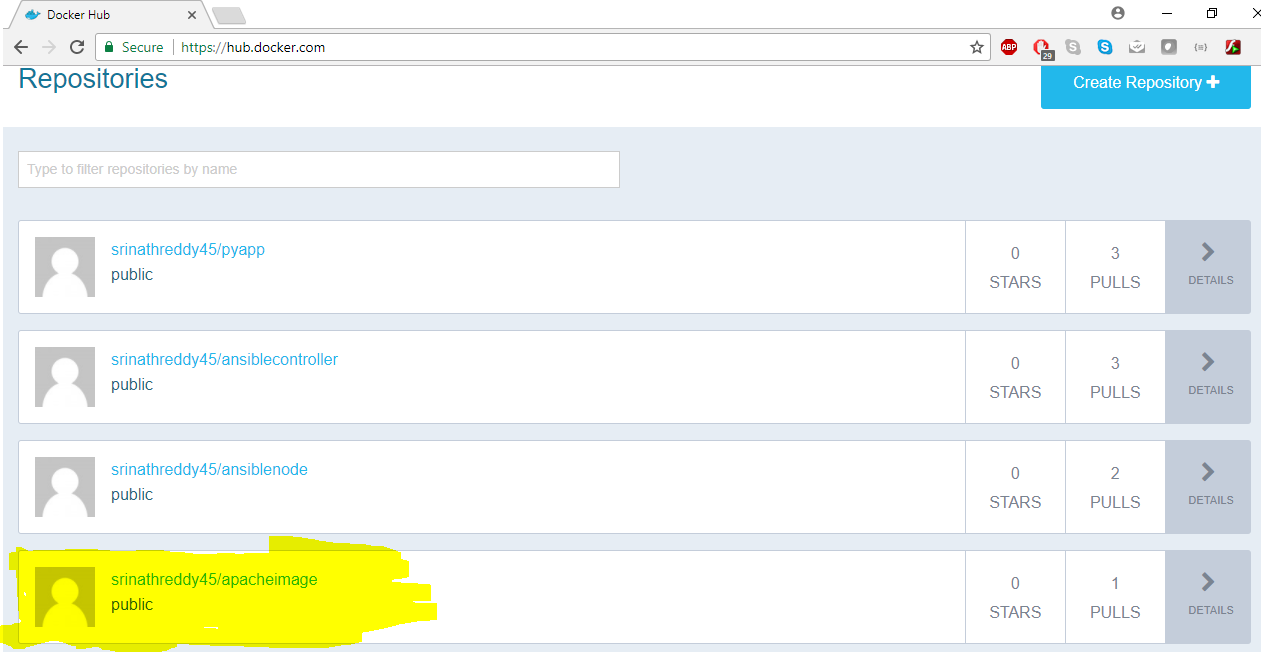


After verifying the web content has been served as expected we can push our **apacheimage** to our dockerhub repository to reproduce the similar environment on other Docker hosts. Before that we need to login to our dockerhub repository using **docker login** command and provide username and password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

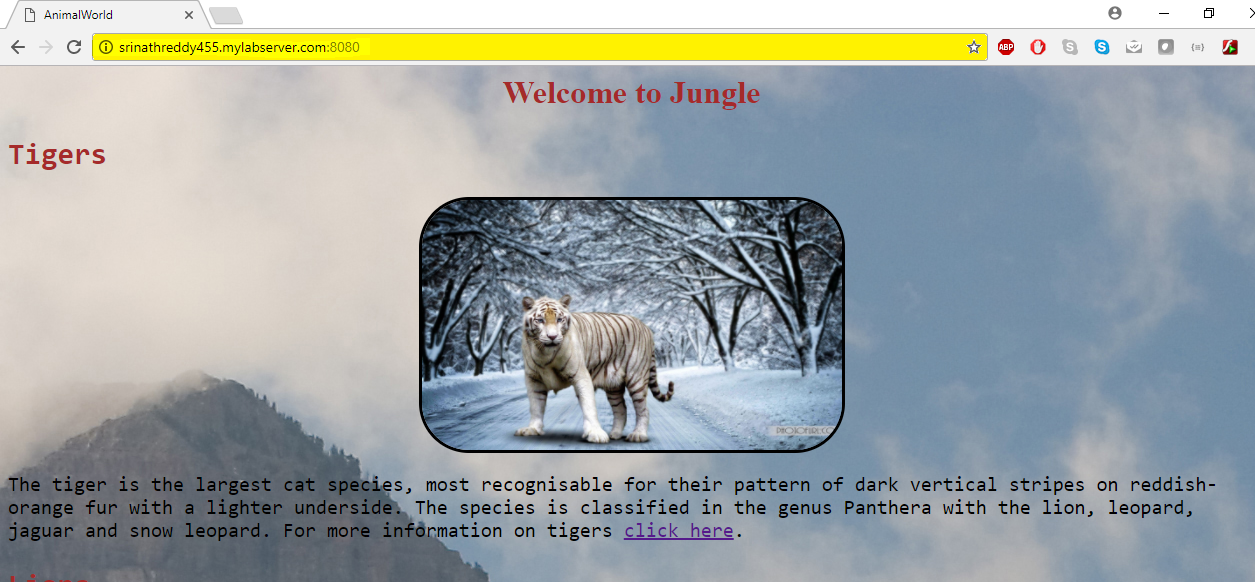


Also tag the **apacheimage** name with our dockerhub login id using **docker tag** command and then push your image to the repository using **docker push** command.

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:

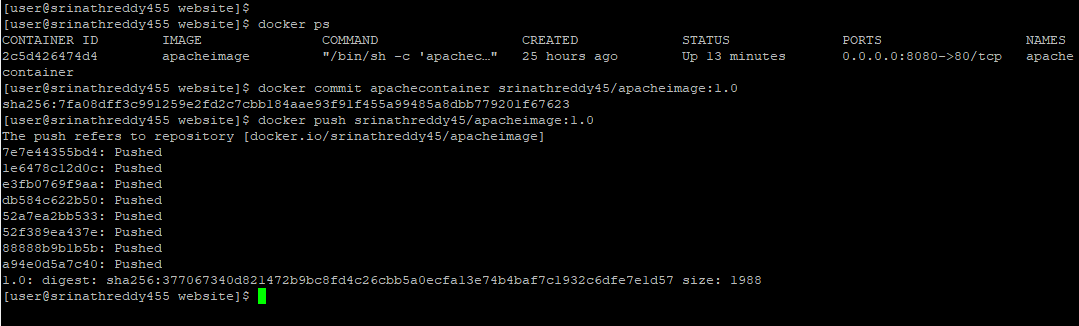


If we want to change the content of our website, we can simply change the code in index.html file on our docker host. Since it is mapped to the container our website content will be changed automatically once we refresh our webpage. Here I have changed the image of tiger from Siberian tiger to Bengal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

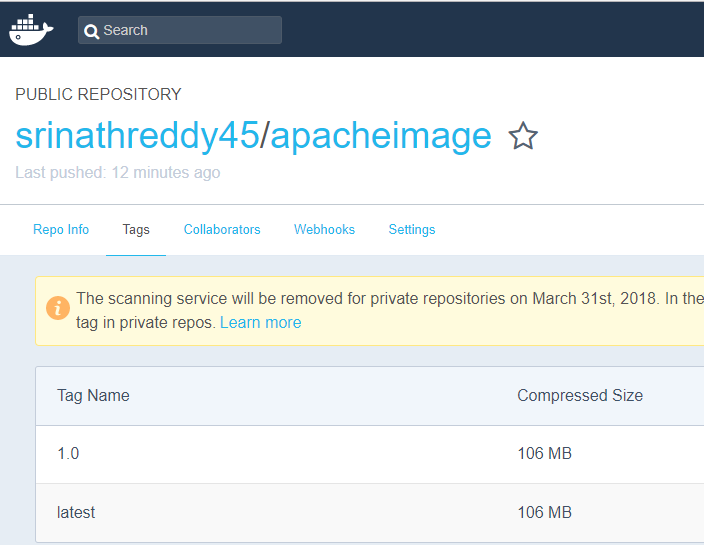


By leveraging the volumes, we could deliver the changes to our website in a fast-paced environment which has been evident in the above figure.

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:

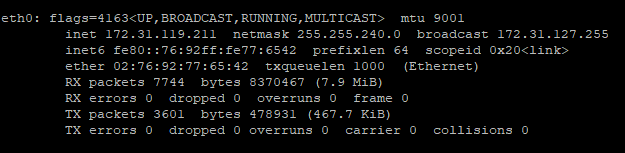


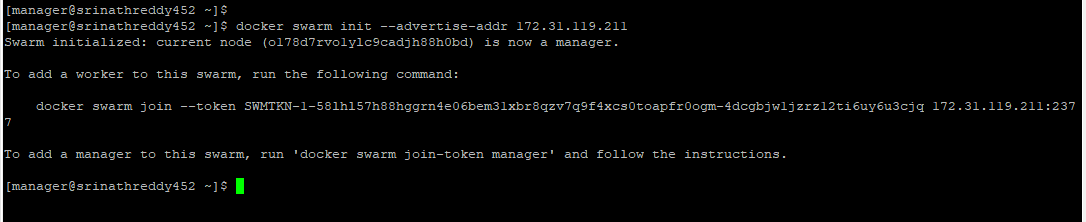
Here 1.0 denotes the version of our image which is used to identify the changes to our web content. Once the image has been created we will be pushing the image to our dockerhub repo using **docker push.**  We can verify the push by visiting dockerhub as shown in below figure:

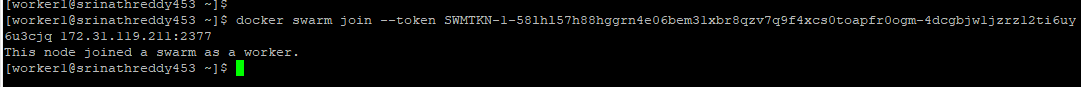


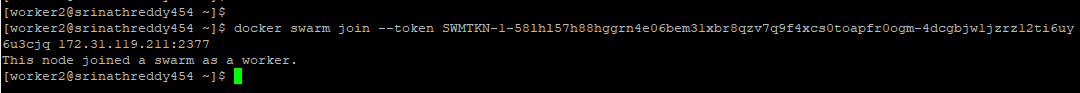
Scaling the website using Docker Swarm

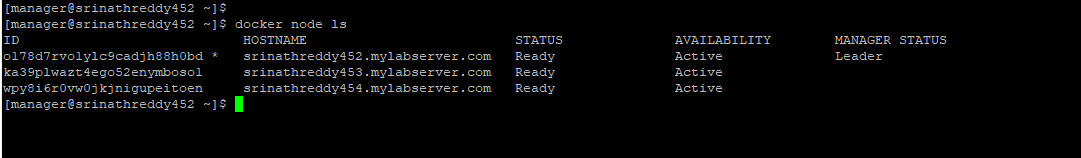
|  |  |
| --- | --- |
| Hostname | Role |
| srinathreddy452.mylabserver.com | Manager |
| srinathreddy453.mylabserver.com | Worker1 |
| srinathreddy454.mylabserver.com | Worker2 |



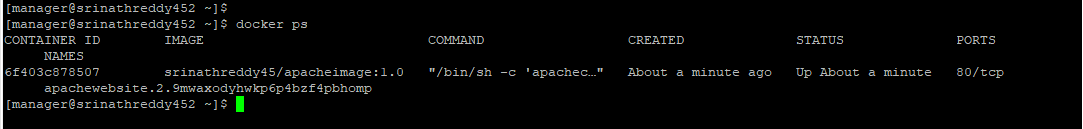


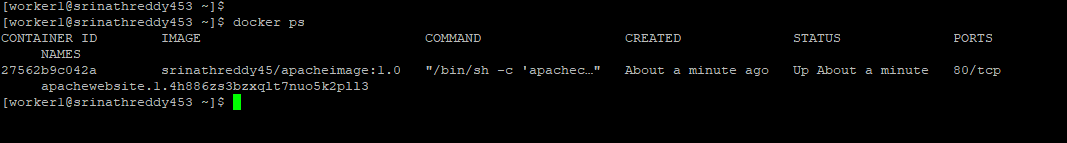


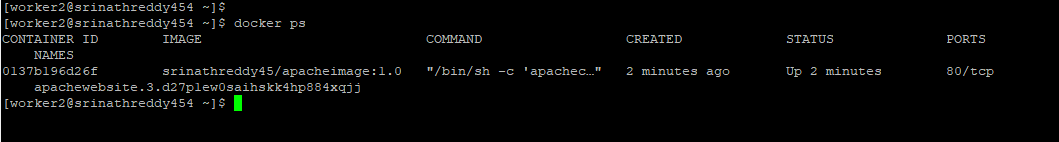






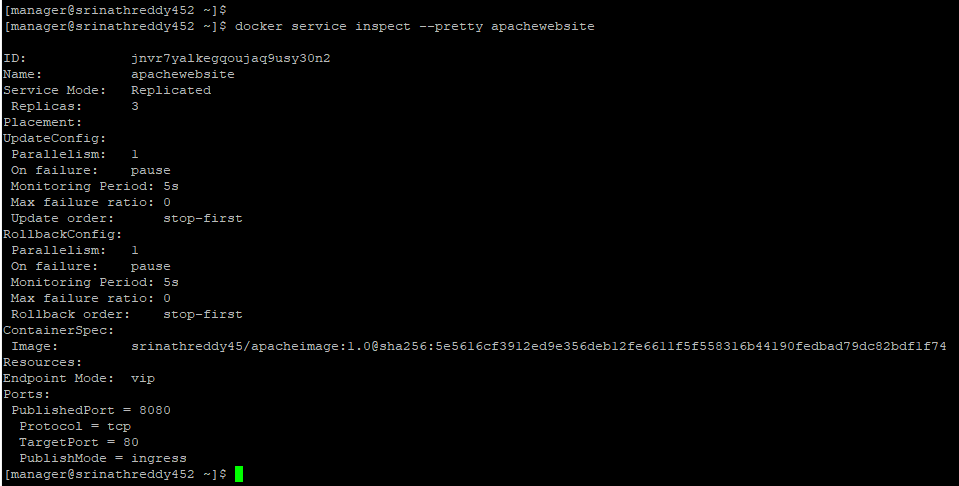


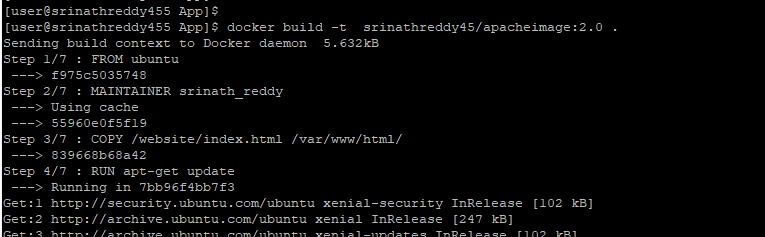


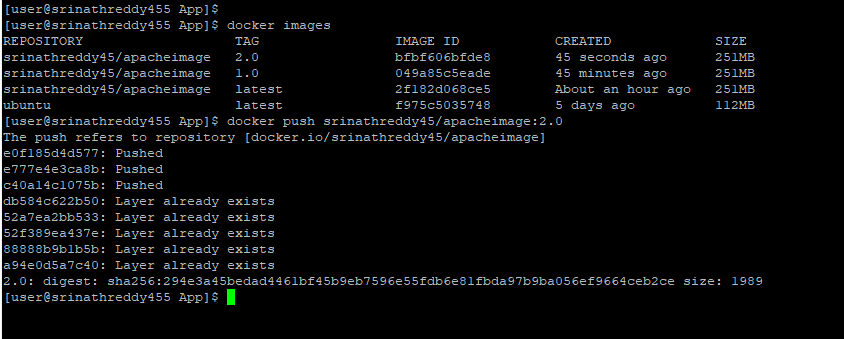


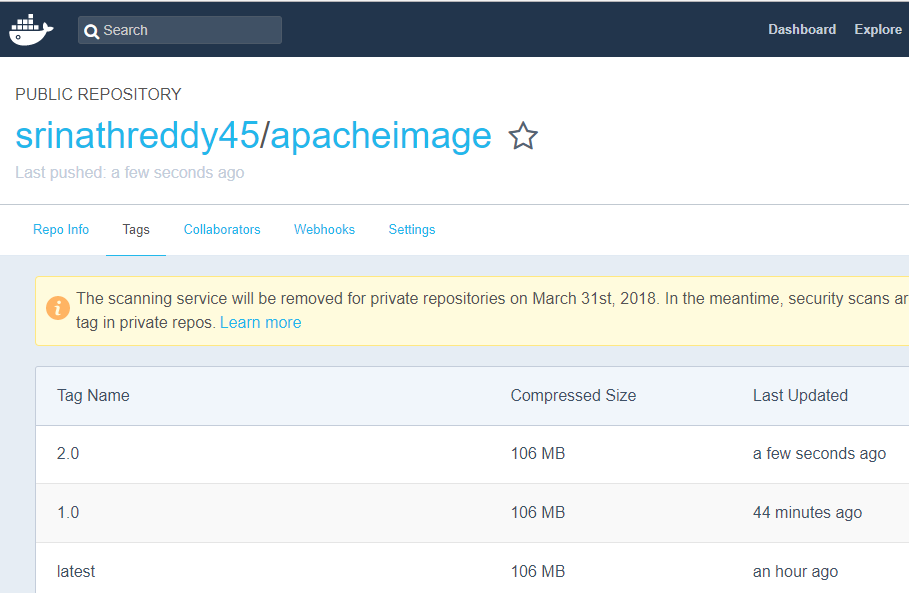


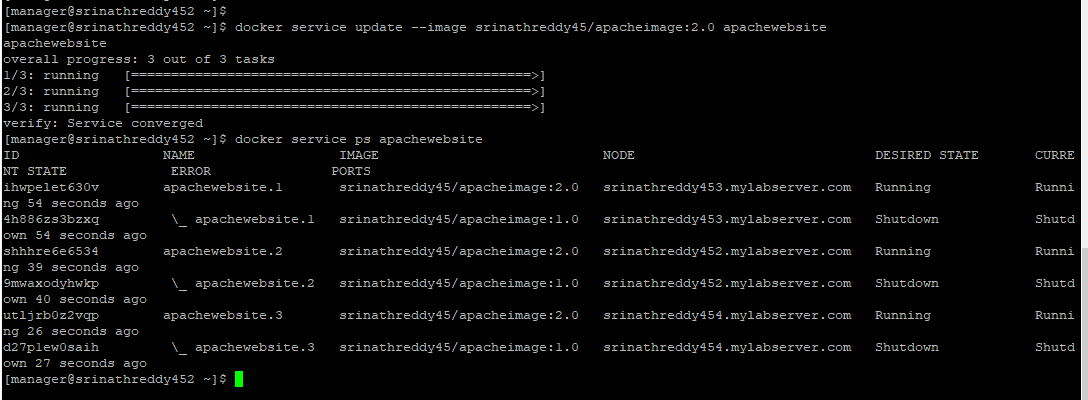


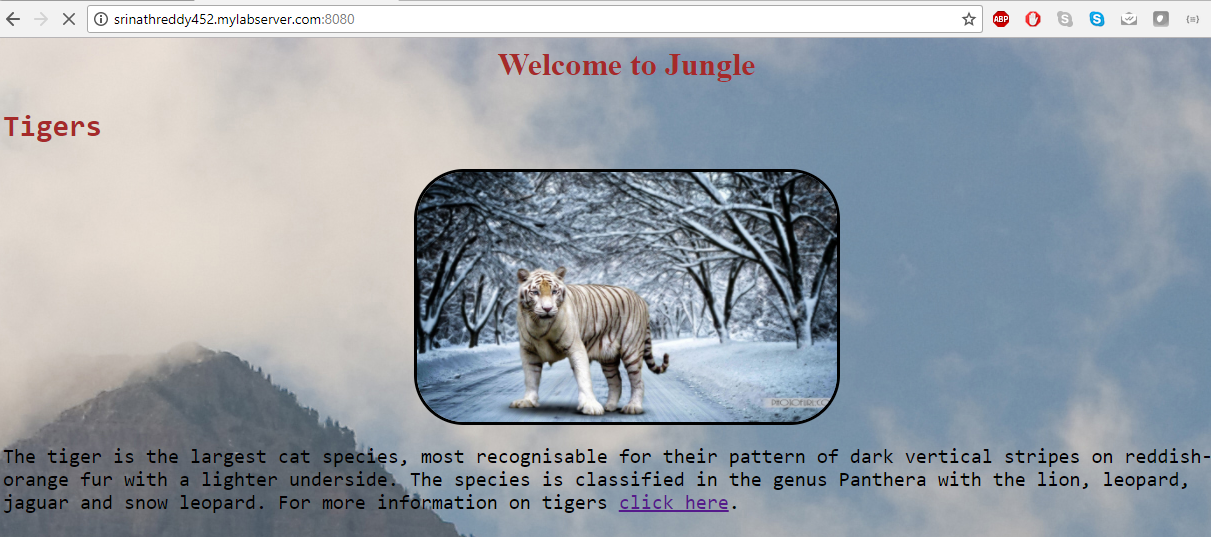


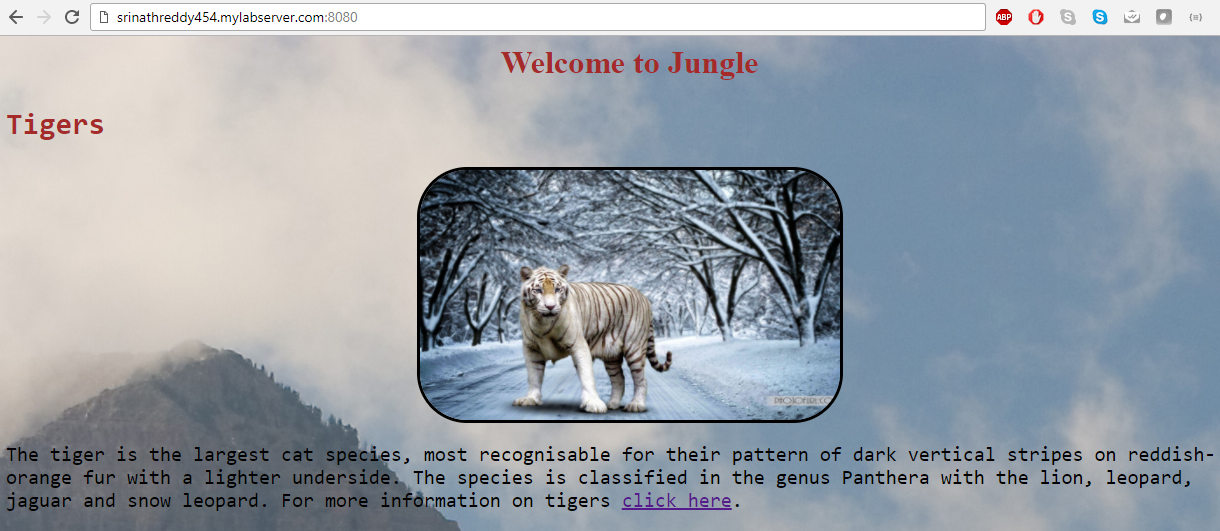


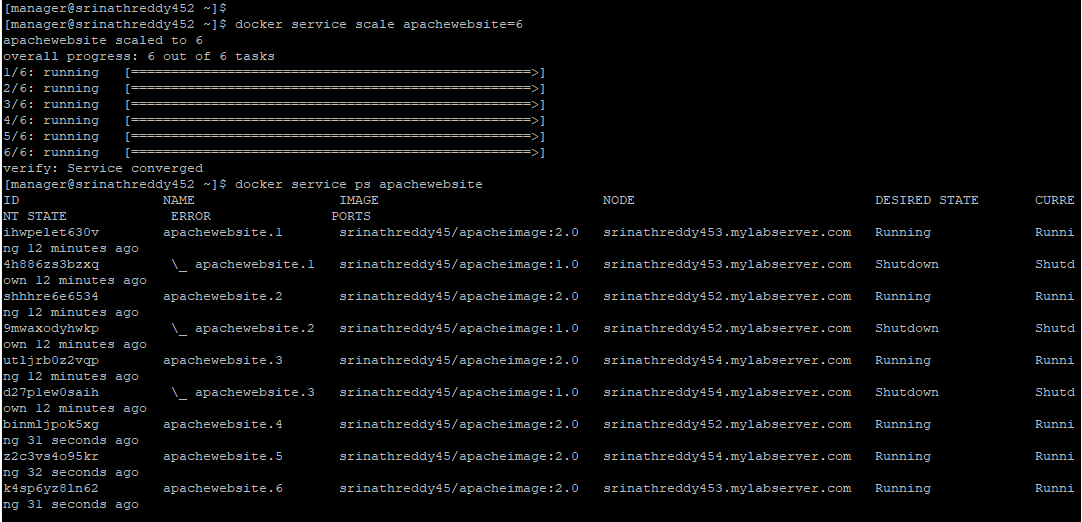


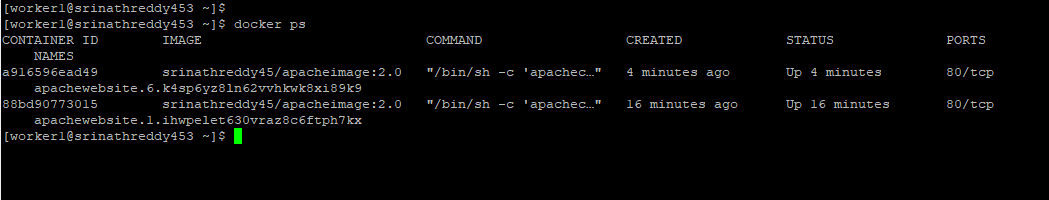




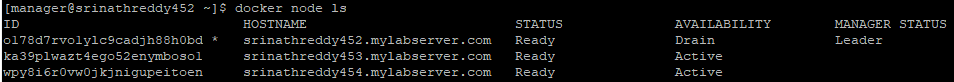


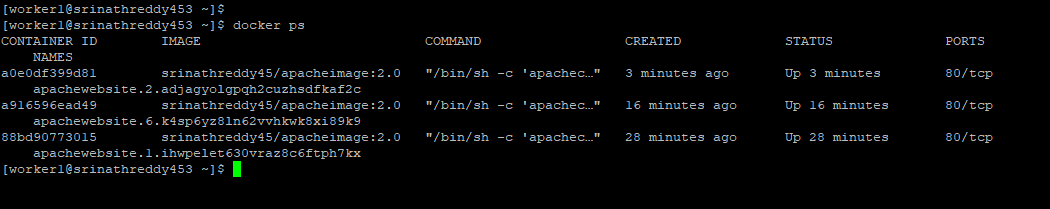


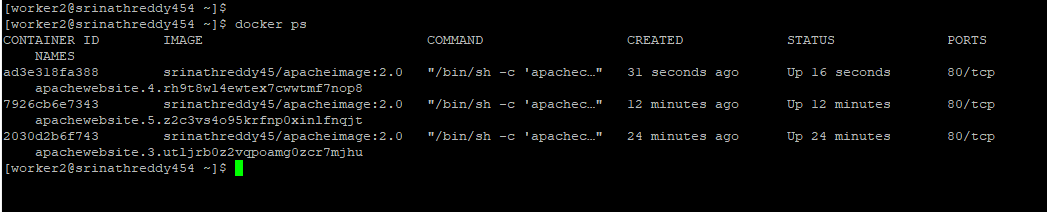












REFERENCES

Ryan Chamberlain, Jennifer Schommer: “Using Docker to Support Reproducible

Research”, 2014.

Chao Zheng, Douglas Thain: “Integrating containers into workflows: A case study

using Makeflow, Work Queue and Docker”, 2015.

Carl Boettiger: “An introduction to Docker Reproducible Research with Examples

from R Environment”, 2014.

Charles Anderson: “Docker in Software Engineering”, 2015.

Salman Basnet, Stefan Berger: “IBM Research Report on Docker and Container

security”, 2016.

James Turnbull: “The Docker Book”, 2014.

Shrikrishna Holla: “Orchestrating Docker: Manage and Deploy Docker services to

containerize applications efficiently”, 2015.

Guo P. CDE: “A Tool for Creating Portable Experimental Software Packages

using Docker”, 2012.

LeVeque R, Mitchell I: “Reproducible Research for Scientific Computing: Tools

and Strategies for Changing the Culture”, 2012.

Altintas, C. Berkley: “An extensible system for design and execution of

scientific workflows in Scientific and Statistical Database Management”, 2004.

Wang, Chenxi. (2016). Containers 101: Linux containers and Docker explained.

Retrieved from <http://www.infoworld.com/article/3072929/linux/containers-101-linux-containers-and-docker-explained.html>

What are containers. (2015). Retrieved from

<https://www.sdxcentral.com/cloud/containers/definitions/what-are-containers-like-docker-linux-containers/>

M. Rosenblum, T. Garnkel: “Virtual machine monitors: Current technology and

future trends”, Computer, 2005.

Dockeroverview. (2016). Retrieved from Official docs: https://docs.docker.com/engine/docker-overview/

Cgroup.(2012). Retrieved from

<http://www.oracle.com/technetwork/cn/articles/servers-storage-admin/resource-controllers-linux-1506602-zhs.html>