Problem: **How to build a custom web-app image and link to mongodb container using docker-compose.**

Solution:

Before we will start solving this problem let's understand some important points.

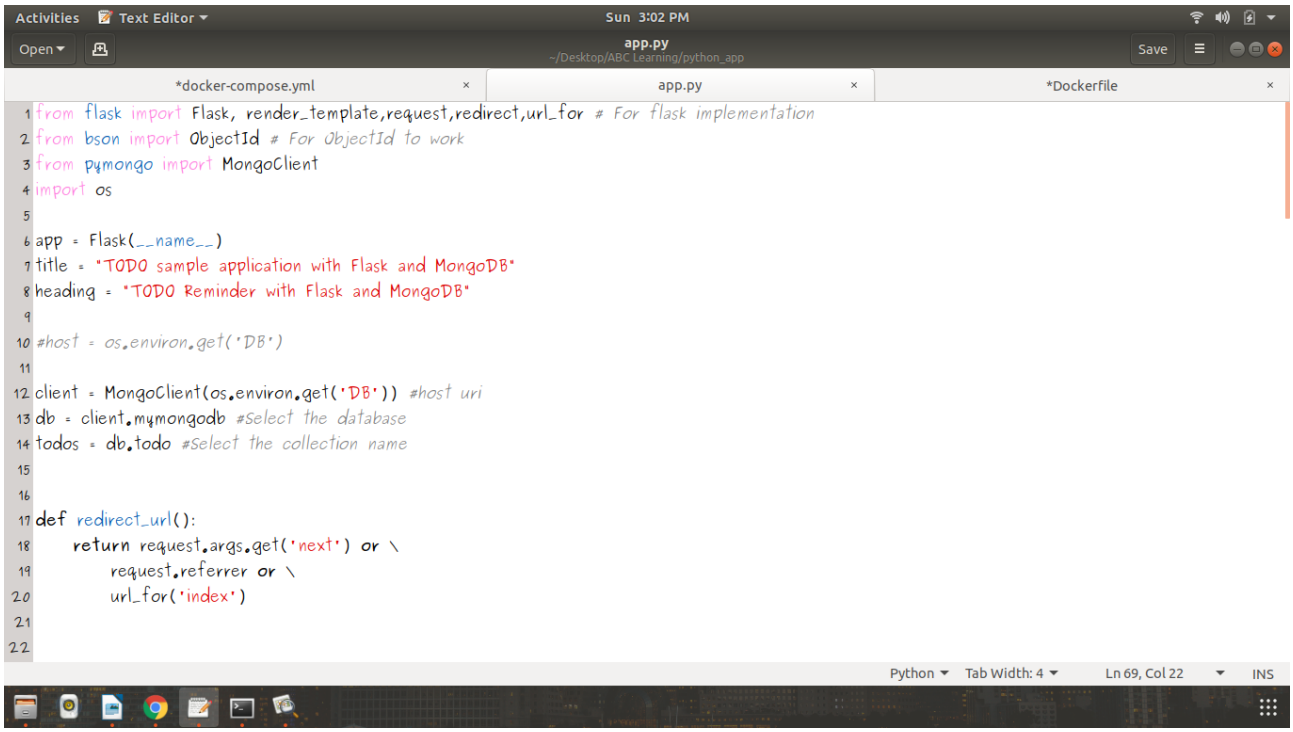
What is docker-compose.yml file?

A docker-compose.yml is a yaml file which defines how Docker container will behave when you will exectue this file.

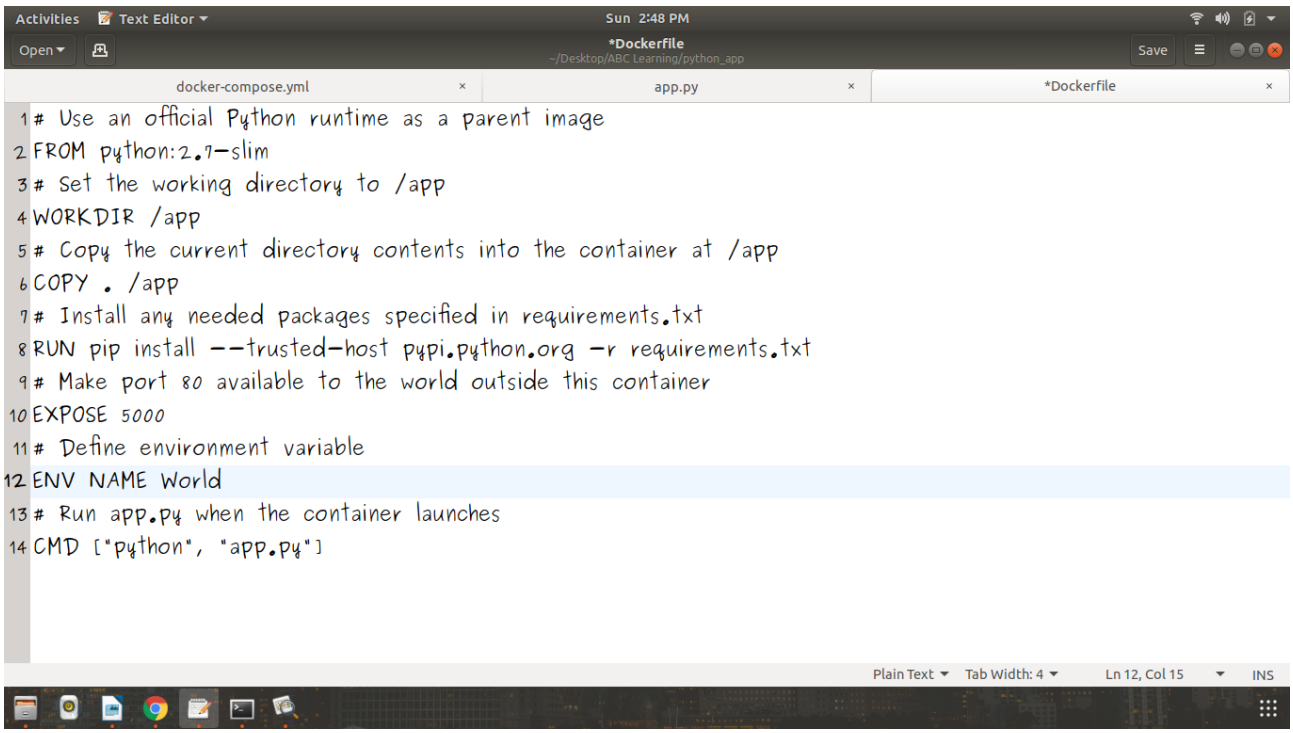
In this problem we will run a python web application(flask) which is linked with mongodb nosql database.

Let's start:

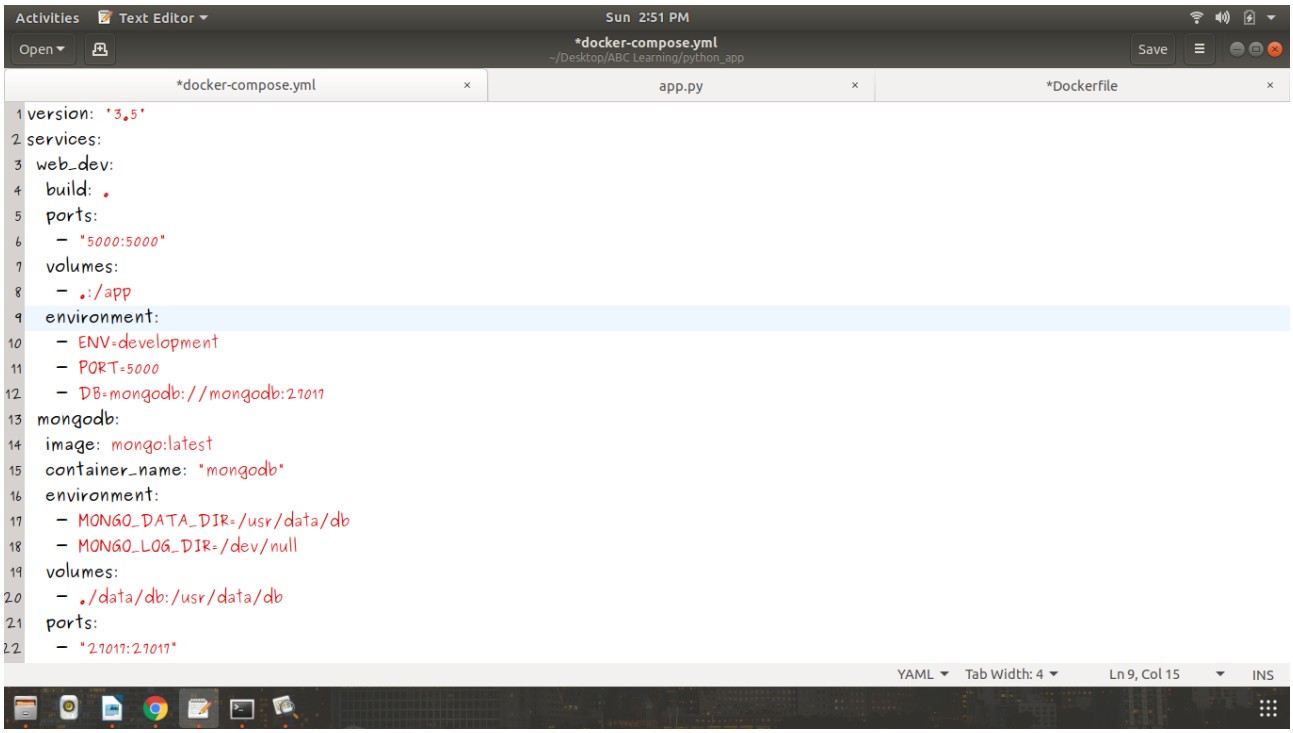
Step1: First create a python web application and name that file as 'app.py'



Step2: Create a Dockerfile which will create a custom image that will execute your application.



Step3: Create a file named ‘docker-compose.yml’ where we will write a script which will tell docker to do what is mentioed in ‘docker-compose.yml’.

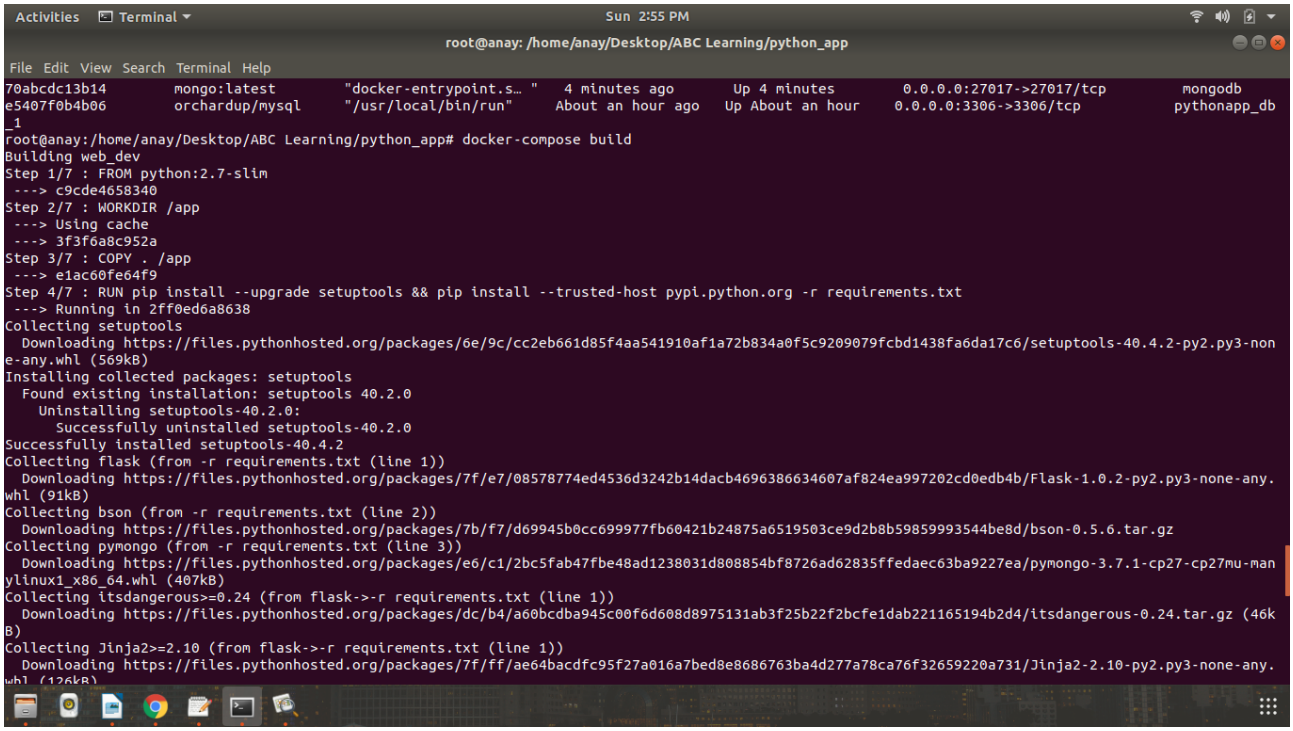


Step4: To run this docker-compose.yml file we need to have docker-compose service in our system . If you do not have this, you can install by this command ‘apt-get install docker-compose’.

To run this docker-compose.yml file, we need to execute a command ‘docker-compose up -d’ or to build your application we can run a commad ‘docker-compose build’.

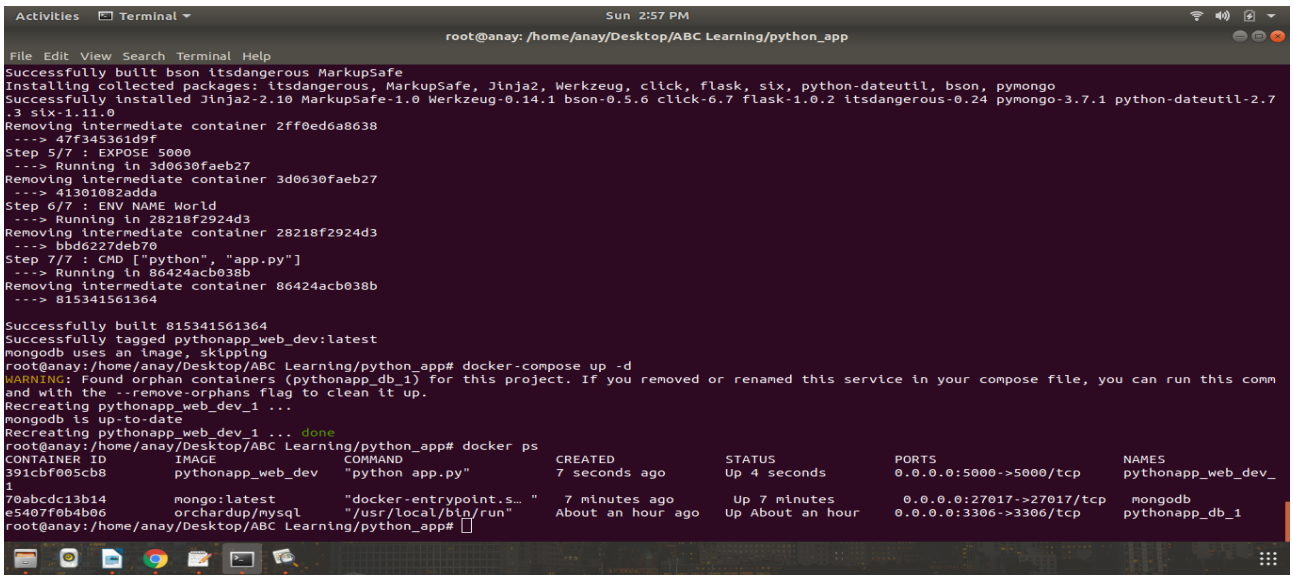
First we will build this docker-compose file.

```
$ docker-compose build
```



After buidling , we can run this as well through a command ‘docker-compose up -d’.

```
$ docker-compose up -d
```



Now, we can see that two containers are running named pythonapp_web_dev_1 and mongodb. Now open your browser and enter the url 'http://localhost:5000'.

Problem: Docker example to show the differnene between RUN , CMD and ENTRYPOINT.

Docker build images automatically by reading the instruction that we have written in our Dockerfile. And one can say that Dockerfile is a text document that contains all the commands or instruction a user could call on the command line to create an Image.
This scenario will describes you about some of those commands which are RUN, CMD, Entry Point.

1. RUN:

Run has two froms:
---> RUN<command>

The RUN command is the central executing directive for Dockerfiles. It takes a command as its argument and runs it to form the image. Unlike CMD, it actually is used to build the image (forming another layer on top of the previous one which is committed).

Example:

Lets take a base imgae of ubuntu:latest and we will instaall jdk in it. So for this task our Dockerfilw would be like this as given below.

```
FROM ubuntu:latest

RUN apt-get upade -y && apt-get upgrade y && aptget install default-jdk -y
```

After building this Dockerfile we will get to know that JDK is installed in ths image.

```
root@anay:/home/anay/Desktop/ABC Learning/RUN# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
jdk_app             latest              1958d4e6b915        12 minutes ago      632MB
```

We will run ths image and run a command in java -version.

```
root@b047807c2138:/# java -version
openjdk version "10.0.2" 2018-07-17
OpenJDK Runtime Environment (build 10.0.2+13-Ubuntu-1ubuntu0.18.04.2)
OpenJDK 64-Bit Server VM (build 10.0.2+13-Ubuntu-1ubuntu0.18.04.2, mixed mode)
```

2. CMD

The command CMD, similarly to RUN, can be used for executing a specific command. However, unlike RUN it is not executed during build, but when a container is instantiated using the image being built. Therefore, it should be considered as an initial, default command that gets executed (i.e. run) with the creation of containers based on the image.

an example for CMD would be running an application upon creation of a container which is already installed using RUN (e.g. RUN apt-get install …) inside the image.

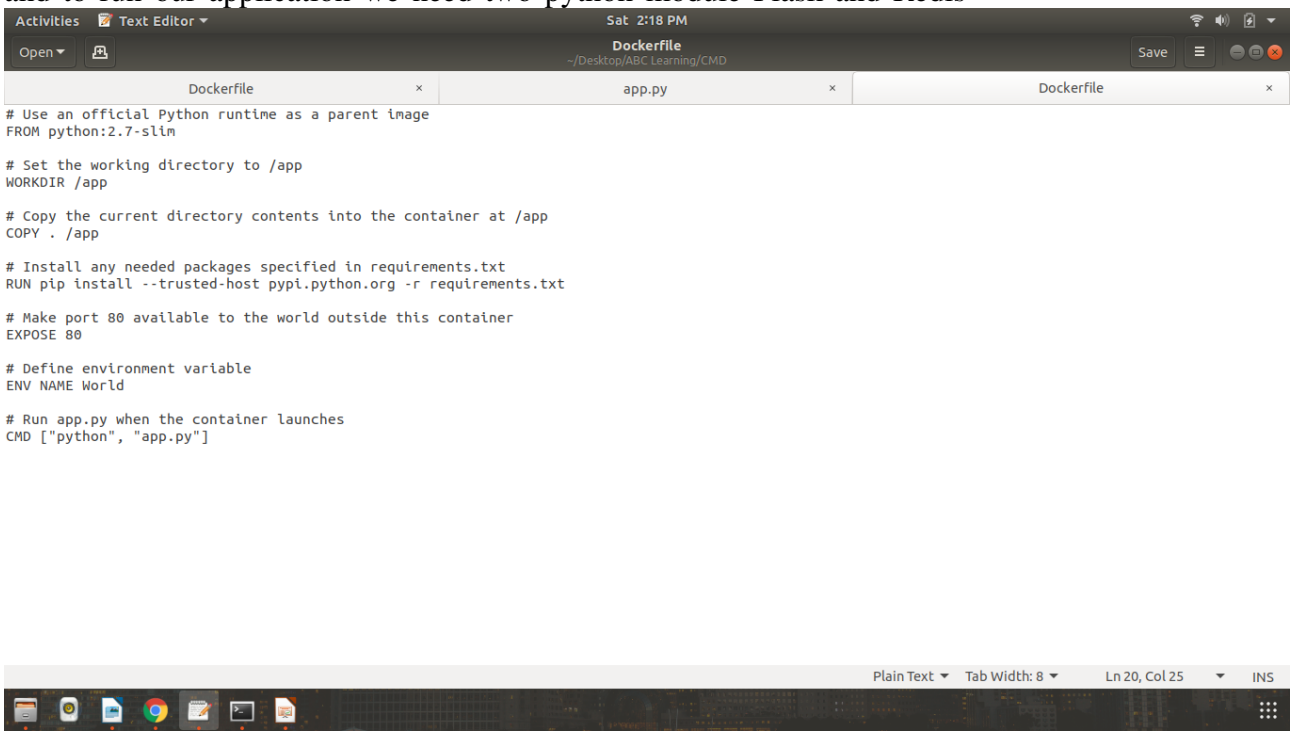
Usage 1: CMD application "argument", "argument", ..

CMD "echo" "Hello docker!"

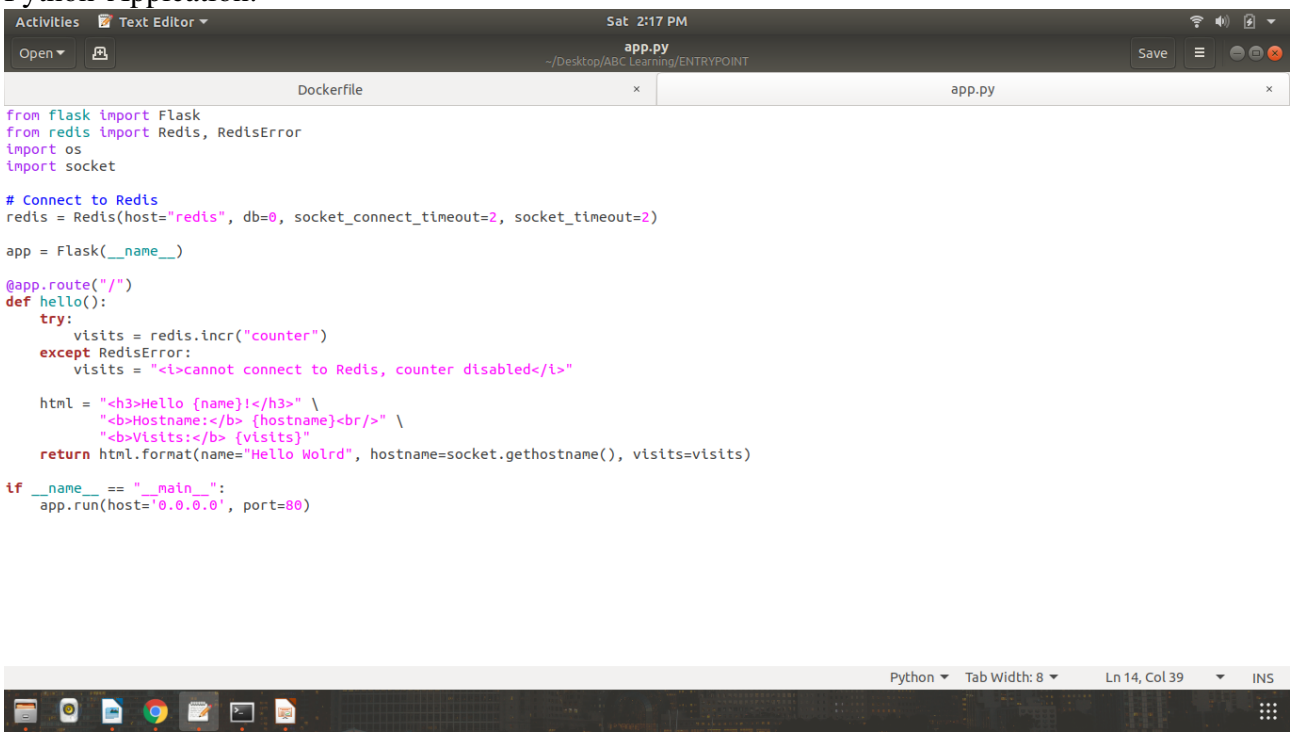
Example: We will deploy a simple python application in docker conatiner.

We have a dockerfile for that. We will be using python2.7-slim

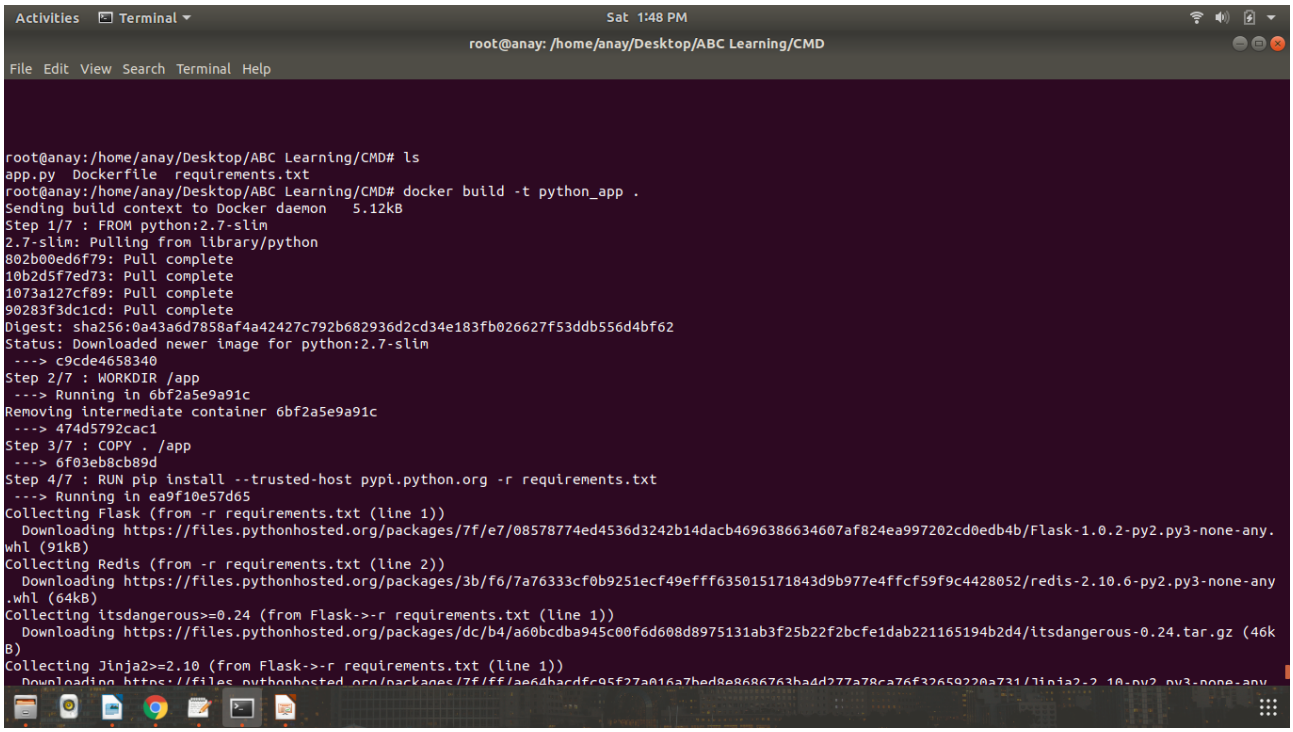
and to run our application we need two python module Flask and Redis



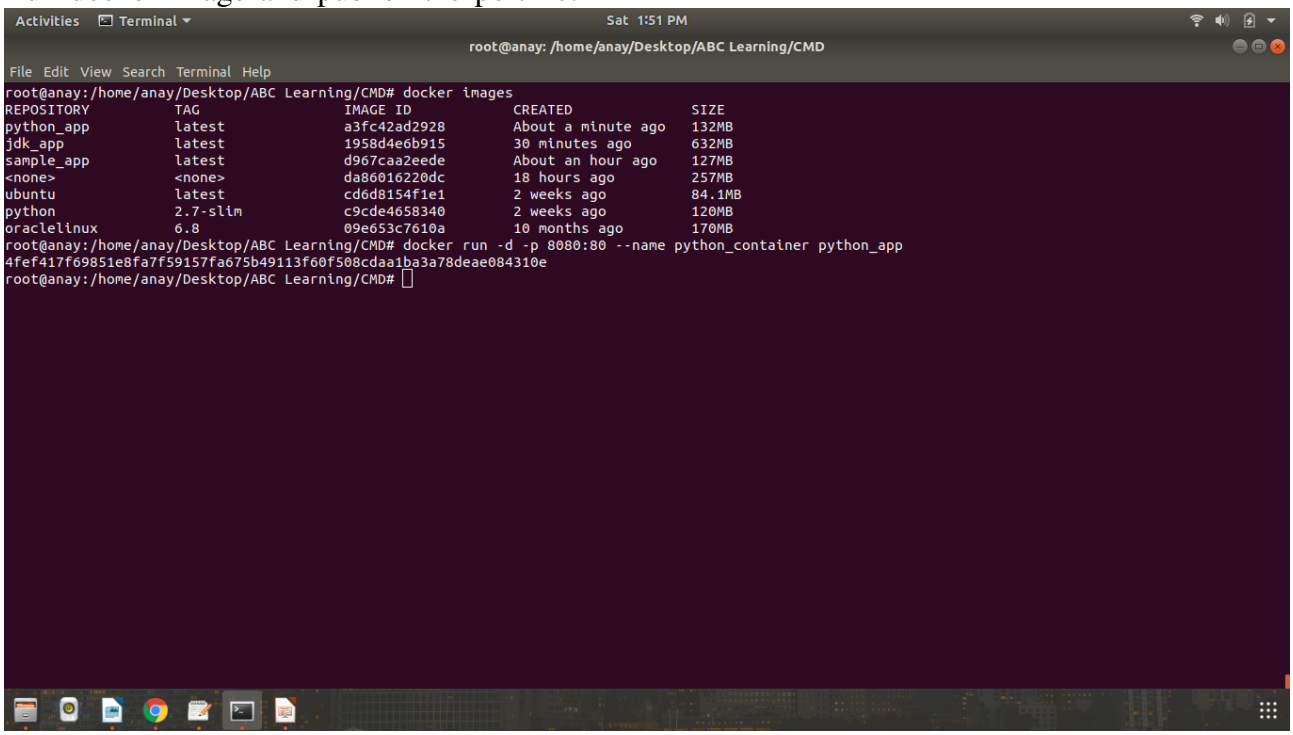
Python Applcation:



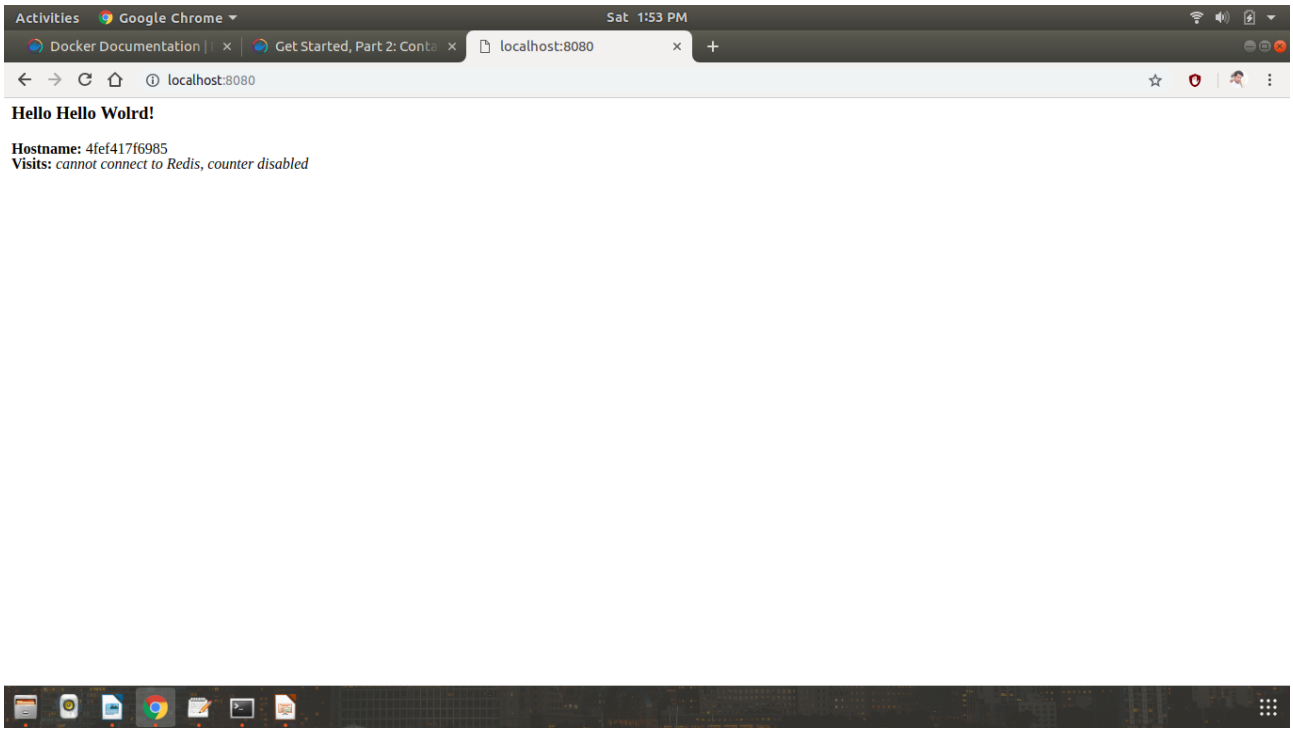
Building Dockerfile:



Run docker image and publish the port no:



Now open to test if app is running or not open the browser and open the url http://localhost:8080.



3.ENTRYPOINT: This argument sets the concrete default application that is used every time a container is created using the image. For example, if you have installed a specific application inside an image and you will use this image to only run that application, you can state it with ENTRYPOINT and whenever a container is created from that image, your application will be the target.

Usage: ENTRYPOINT application "argument", "argument", ..

Remember: arguments are optional. They can be provided by CMD or during the creation of a container.

Usage example with CMD:

Arguments set with CMD can be overridden during *run*

CMD "Hello docker!"

ENTRYPOINT echo

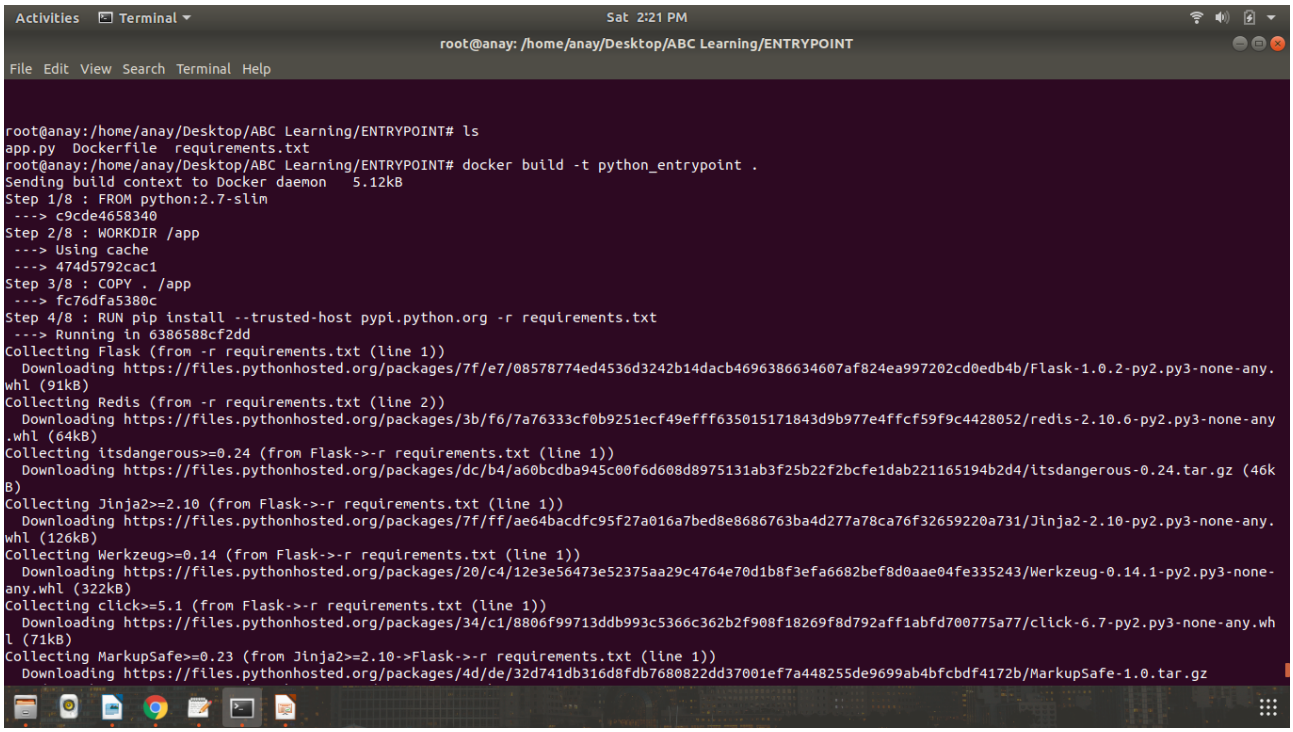
NOTE: Entrypoint is use to set the command and parameters that will be exectued very first time when a container start running.

The syntax of Entrypoint is :

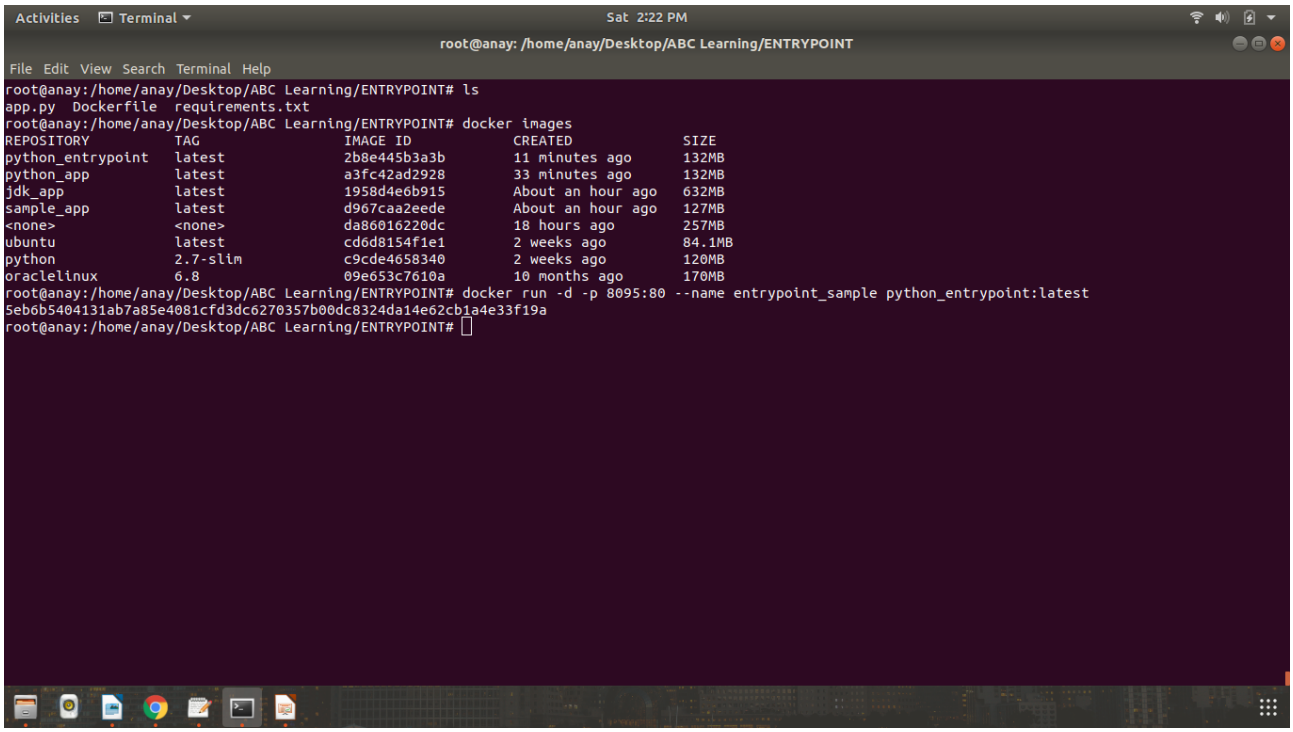
ENTRYPOINT [“executable”,”parameter1”]

Dockerfile:

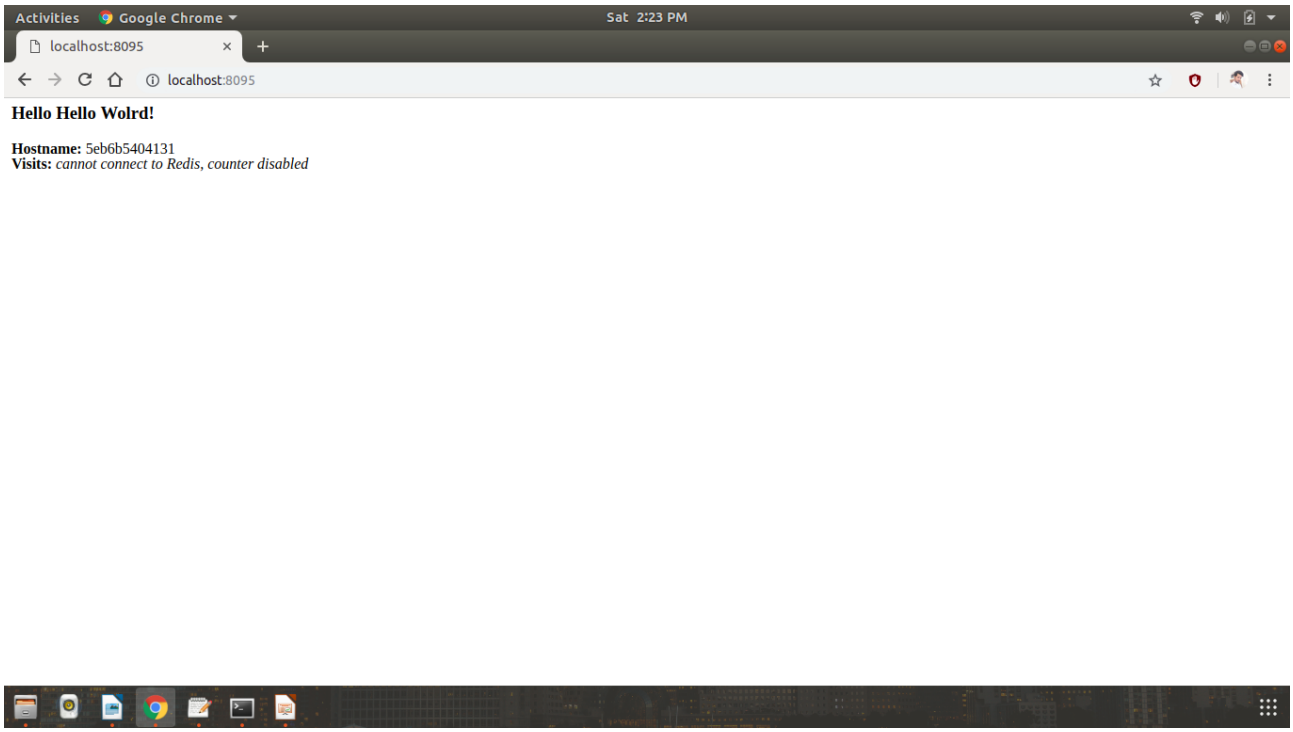
Building the docker file:



Runing docker image:



Testing the application:



Docker Volume:

Let's imagine that you run a mongodb/mysql container. Now you are storing your data in mongodb/mysql db. Now if you need to take a backup of this or you want your data to be persists then docker volume would be the best option for you.

Docker volumes are the preferred method for presisting data which is generated and used by Docker containers. These volumes are completely managed by Docker itself.

There are some advantages of Docker volumes over bind mounts:

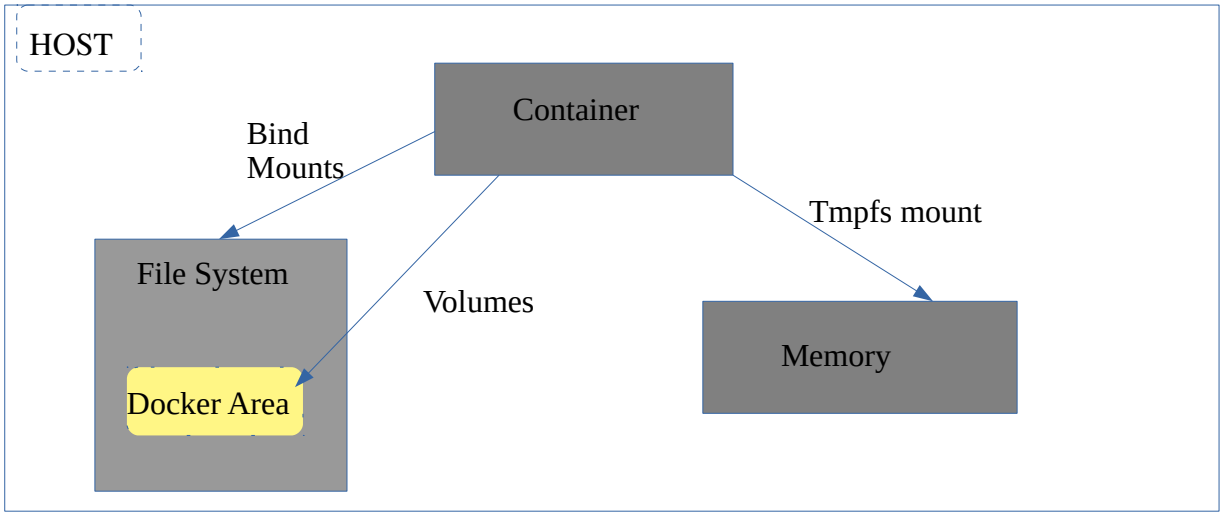
1. We can easily take back up of container or migrate than bind mounts.

2. We can manage Docker volumes by using Docker rest api i.e. Docker CLI etc.

3. We can safley share volumes among multiple container.

4. Volumes does't increase the size of container.

NOTE: The contant of volumes exist outside the lifecycle of a container.



Originally, for volumes we use -v or - - volume flag for standalone containers and - - mount flag for swarm services.

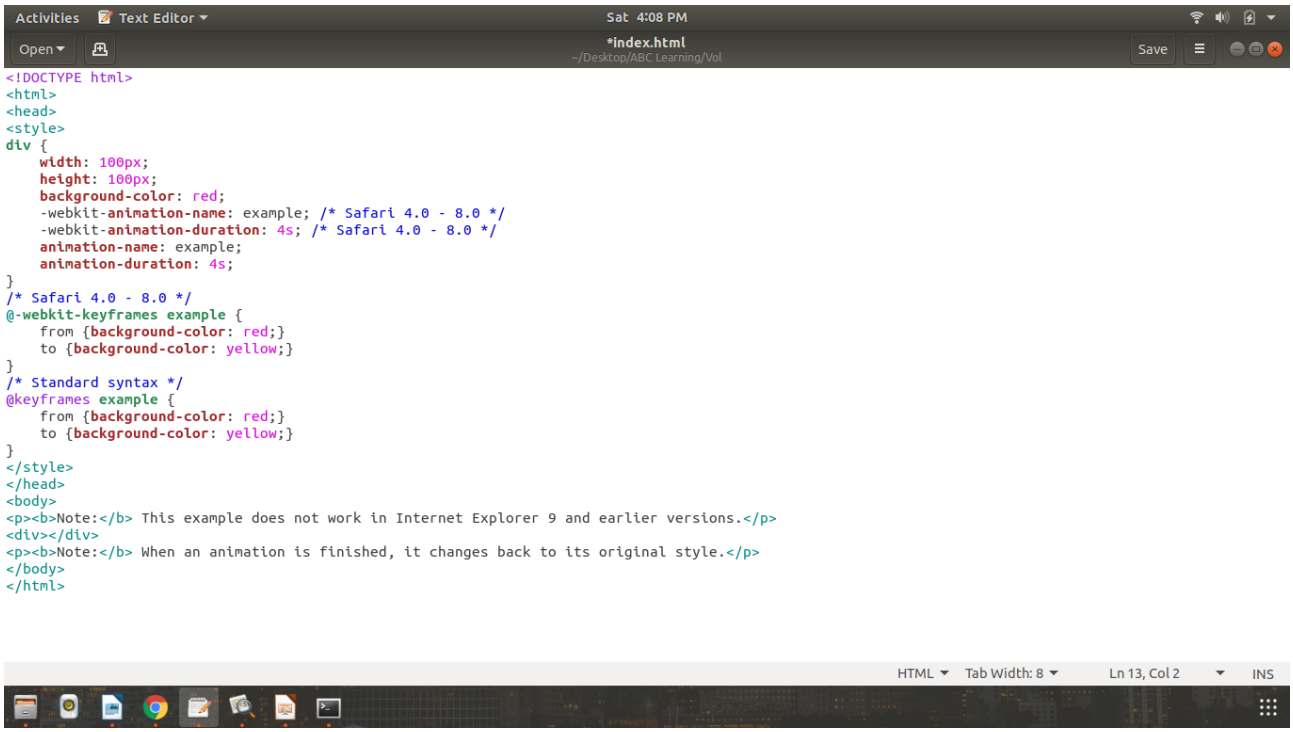
The difference between -v syntax and - - mount syntax is that -v synatx combines all the options together in one field as - - mount syntax separates them.

Synatx of - - mount is :

```
- - mount 'type=volume,src=<VOLUME-NAME>,dst=<CONTAINER-PATH>,volume-driver=local,volume-opt=type=nfs,volume-opt=device=<nfs-server>:<nfs-path>,"volume-opt=o=addr=<nfs-address>,vers=4,soft,timeo=180,bg,tcp,rw'
```

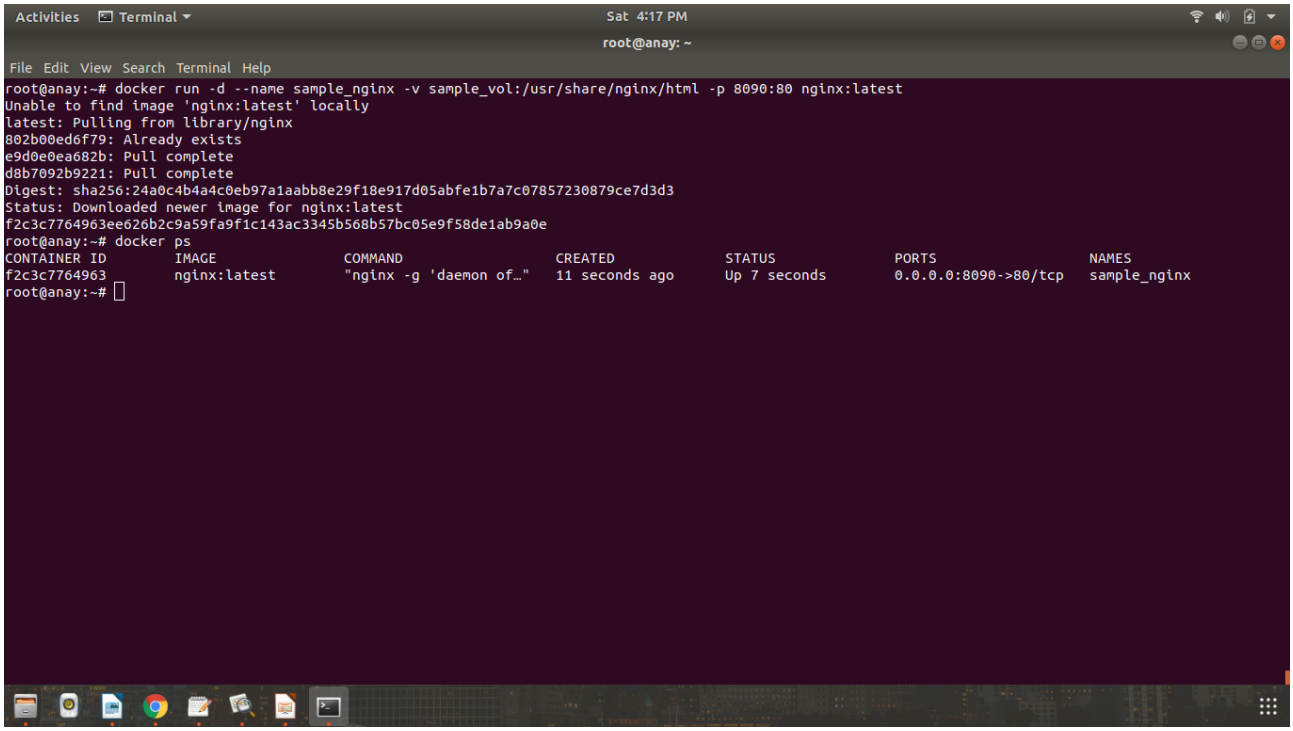
Now, let's take an example, we will create a sample web application and will host at nginx server.

Step1: craete a html page.



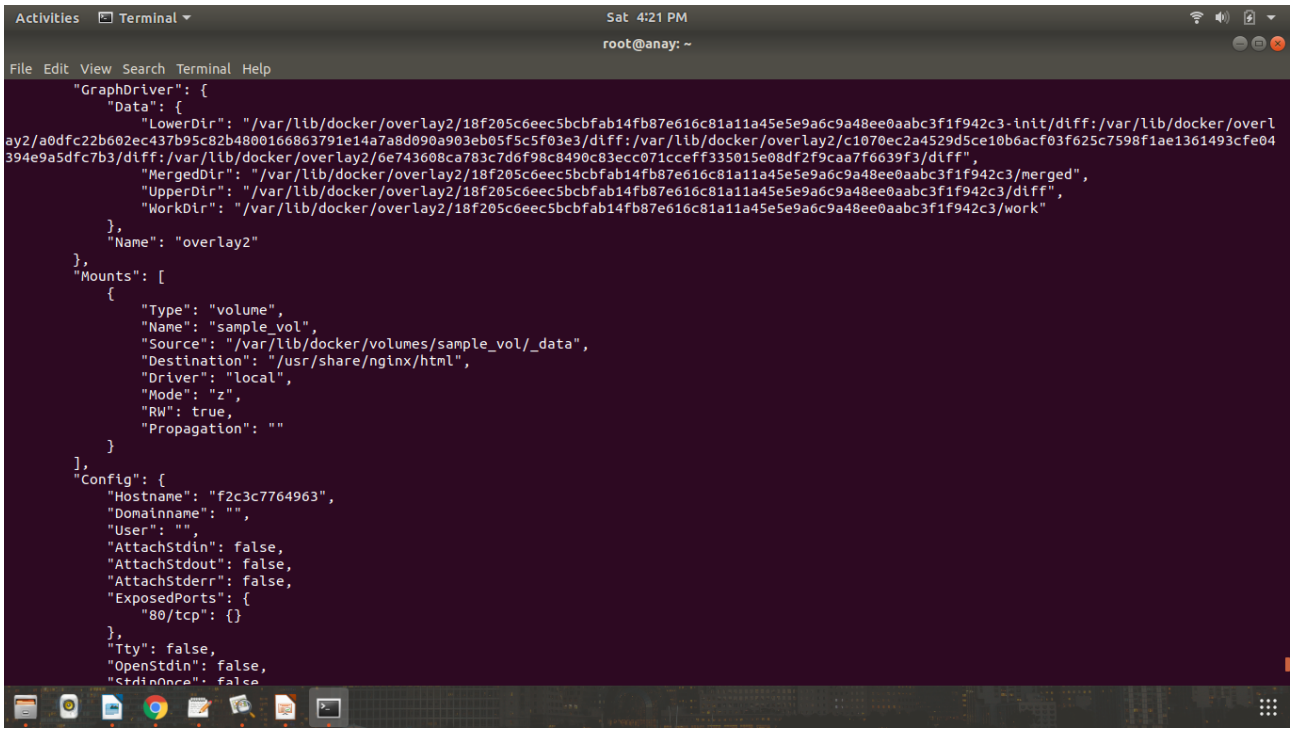
Step2: Run an image nginx:latest with volumes.

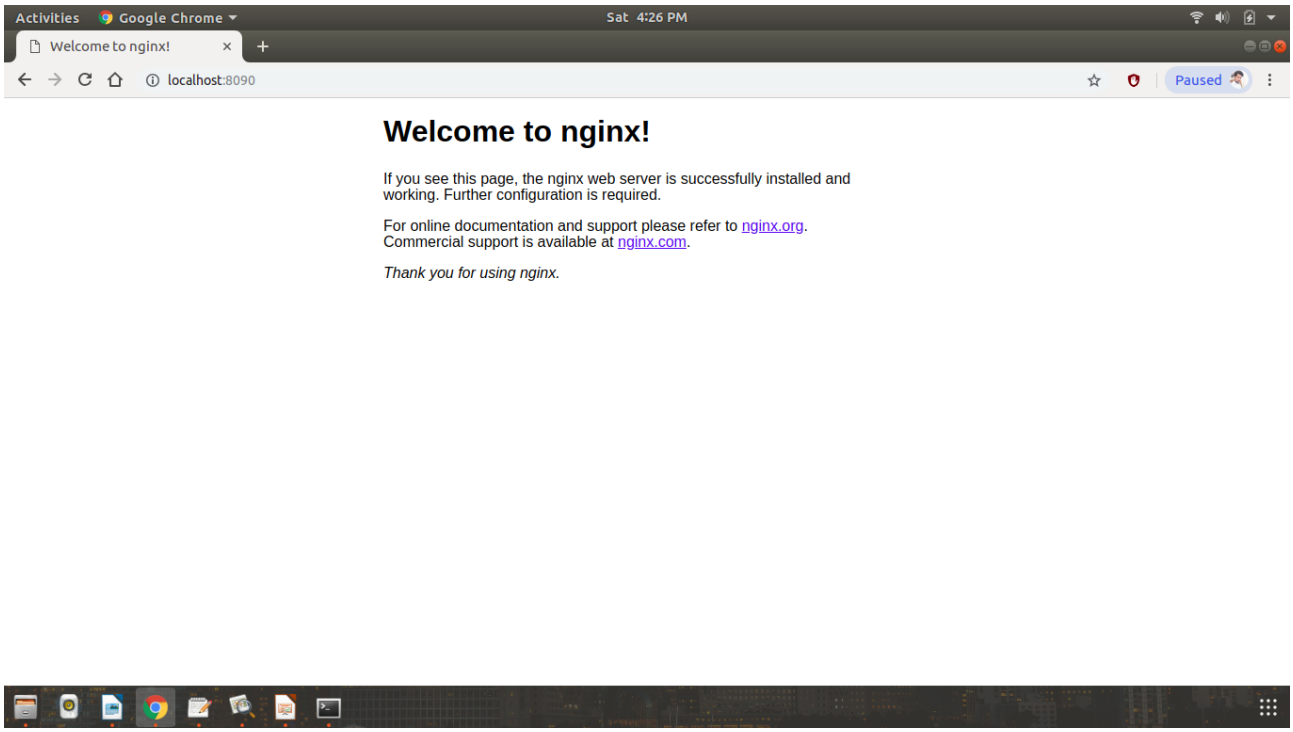
If we start with a volumes that does not exist, docker will create the volume for us. Now we will mount the volume ‘sample_vol’ into ‘usr/share/nginxhtml/’ in the contaner.



Step3: Now we will use docker inspect command to verfiy that the volue was created and mounted correctly at time when we execute the pervious command. We have to concentrate the

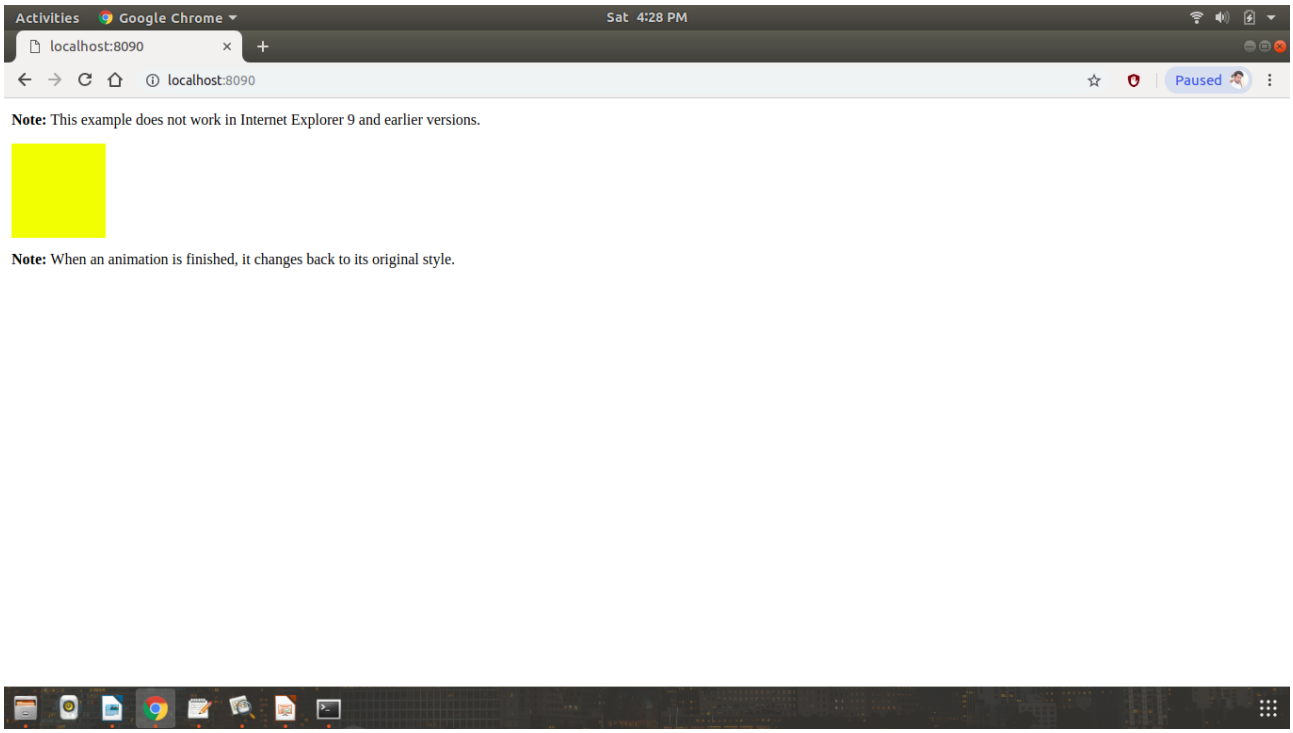
Mounts section:





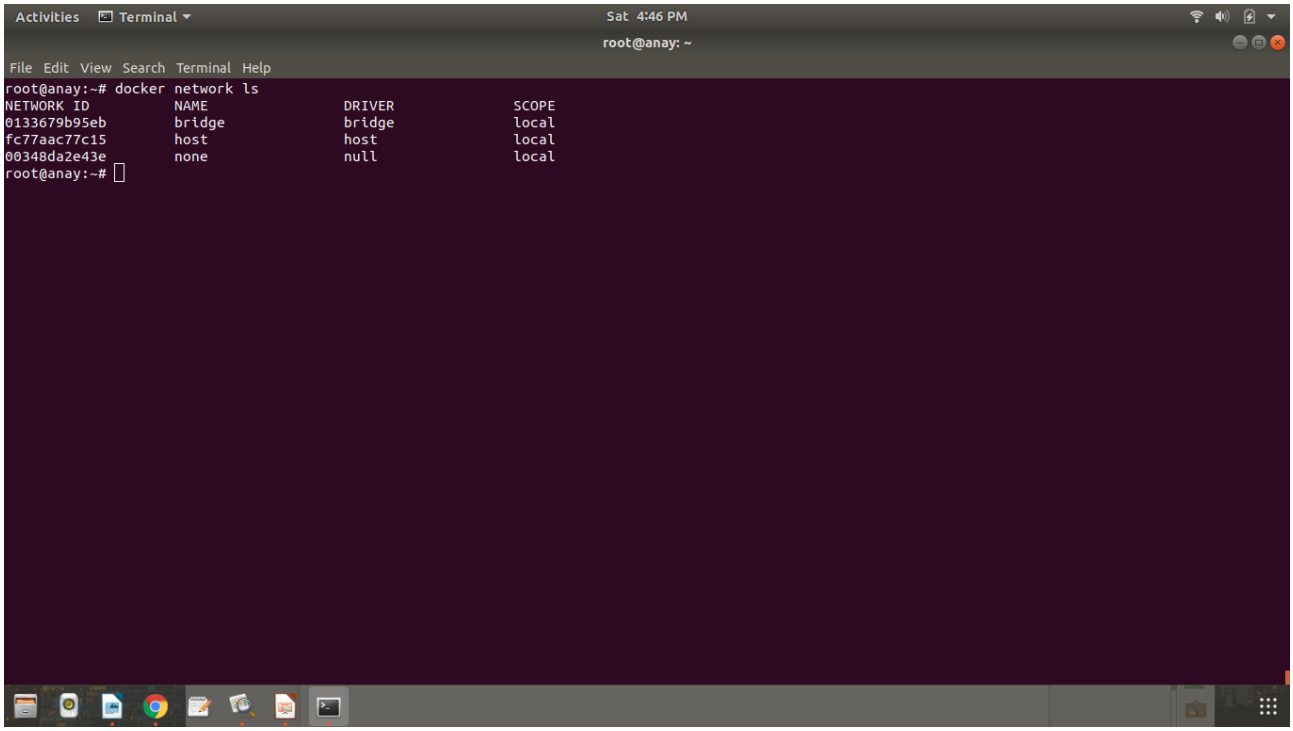
Step4: Next, we will navigate to ‘cd /var/lib/docker/volumes/sample_vol/_data/’ and will paste our html page here.

Step5: Open your brwser and we will see our new html page.



Docker Networking

When we install Docker, it creates three network automatically, which can be listed using docker network command 'docker network ls'



Docker network types:

Docker Network	
Bridge	The bridge network represents the docker0 network present in all Docker installations. Docker daemon connects containers to this network by default.
Host	Host network adds a conatiner on the host's network stack. We will find that the network configuration inside the container is identical to the host.
None	The None network adds a container to a container –specific network stack. This container lacks a network interface.

When to use Bridge Network:

if you have a set of containers and each providing micro-services to the other and should not be exposed to the external world.

User – defined Bridge Network:

In terms of networking, a bridge network is a Link layer device which forwards traffic b/w network segments.

We can create our own bridge network.

1) User-defined bridges provides better isolation and interoperability between containerized applications.

2) User-defined bridges provides automatic DNS resolution between containers.

3) Containers can be attached and detached from user-defined network on the fly

NOTE: linked containers on the default bridge network share env variables.

We can manage a user-defined bridge, to create user defined network run the following command.

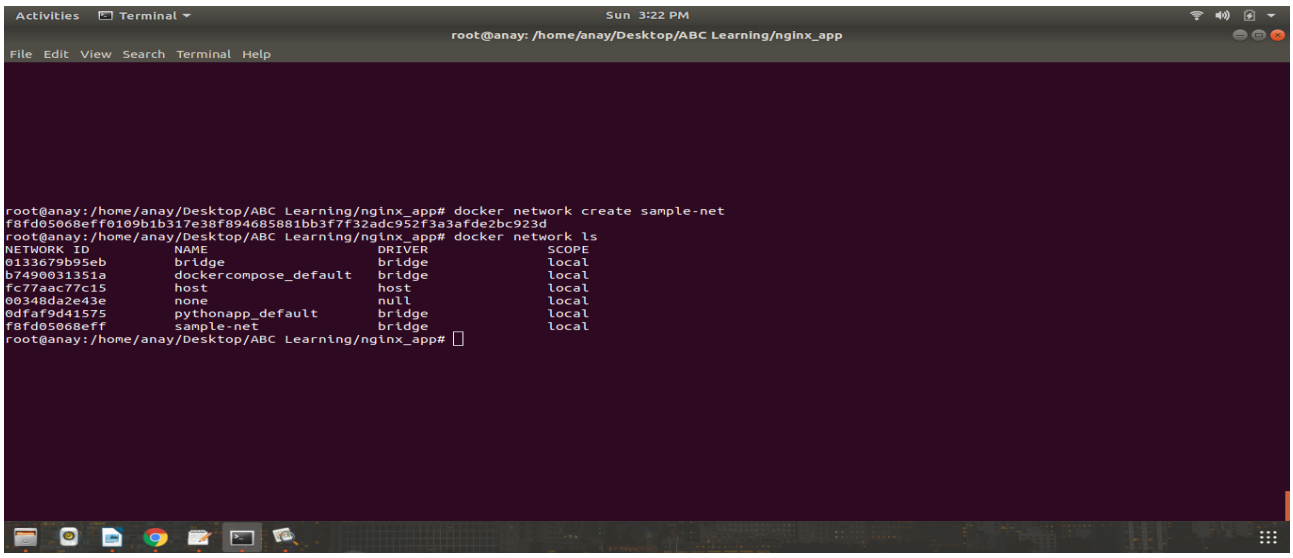
```
$ docker network create sample-network
```

And if you wanted to remove this bridge network run the following command. But before we remove it first disconnect all the container connected to this network.

```
$ docker network rm sample-network
```

Example: We will create a static web app and deploy in nginx serve and will connect with a user-defined bridge network ‘sample_net’.

Step1: create a network ‘sample-net



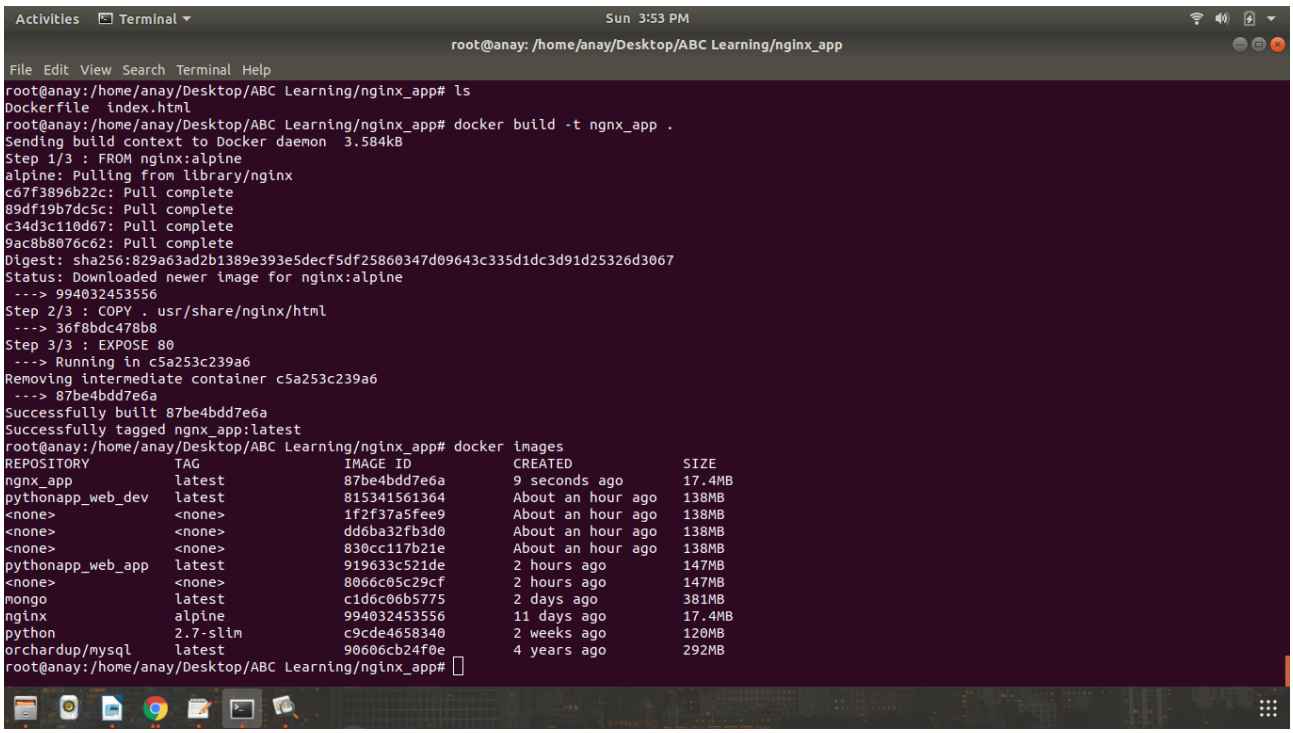
Step2: Create a Dockerfile to create a custom image;

```
FROM nginx:alpine

COPY . usr/share/nginx/html

EXPOSE 80
```

Next, build this Dockerfile.

```
$ docker build -t nginx_app .
```

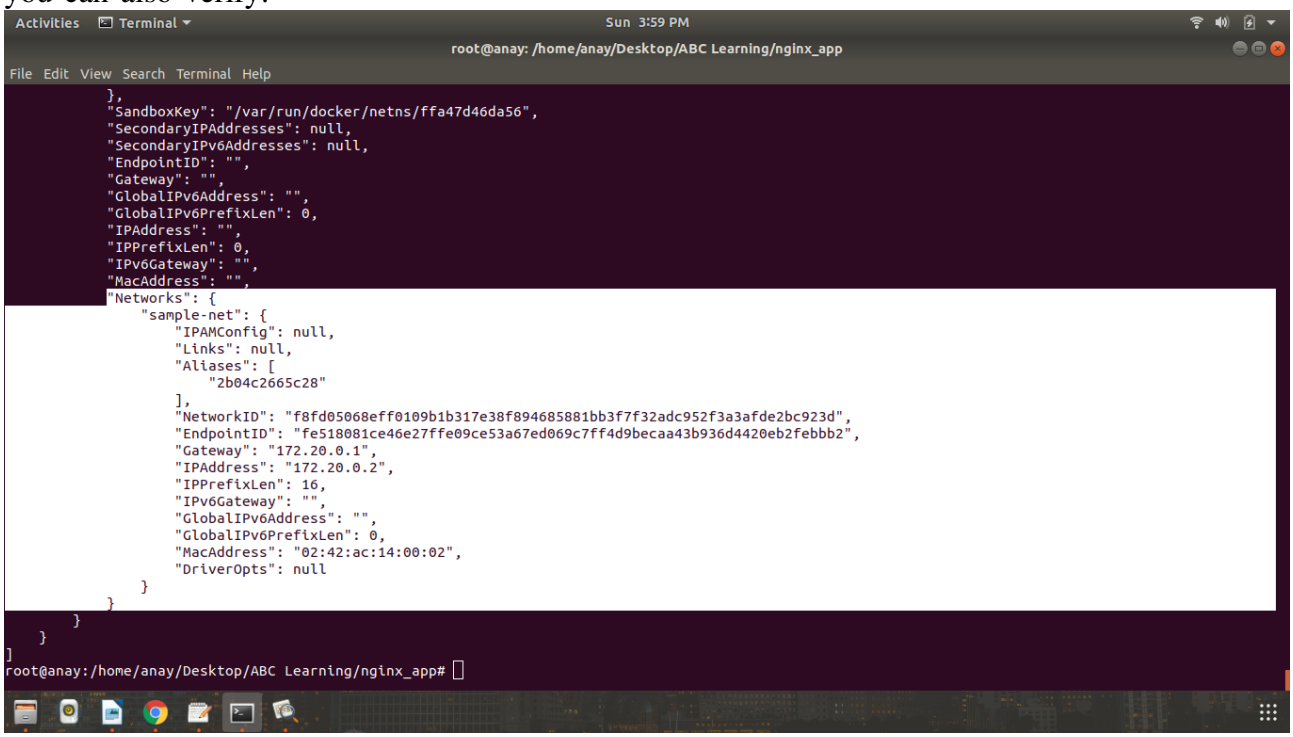
Step3: Now we will start this image ‘nginx_app’ in a container and connect with the user defined network ‘sample-net’.

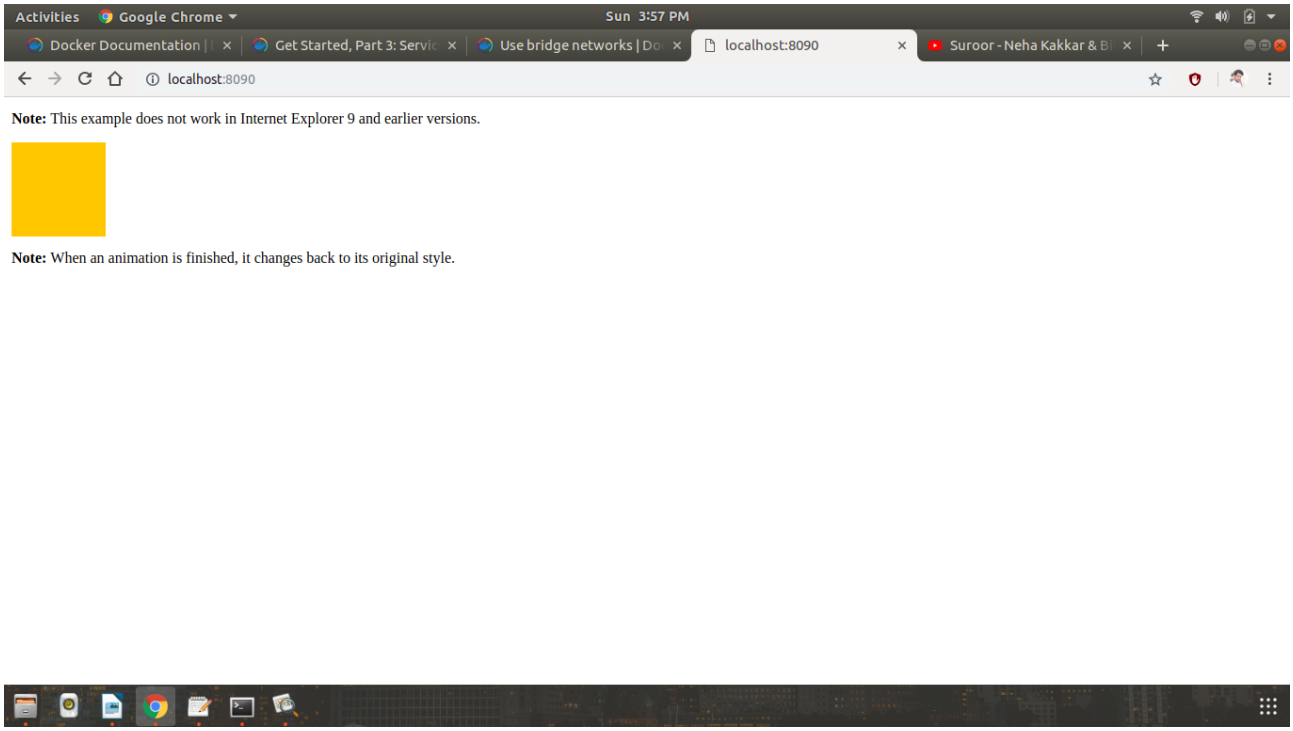



```
root@anay:/home/anay/Desktop/ABC Learning/nginx_app# docker run -d -p 8090:80 --name sampl_nginx_container --network sample-net ngnx_app

2b04c2665c28d2ccdde14071fc3672a1b1f78c5d6af22acb39c88010e600c6d4
```

Next , open your browser and enter your url http://localhost:8090

you can also verify:





index.html page:

```
<!DOCTYPE html>
<html>
<head>
<style>
div {
    width: 100px;
    height: 100px;
    background-color: red;
    -webkit-animation-name: example; /* Safari 4.0 - 8.0 */
    -webkit-animation-duration: 4s; /* Safari 4.0 - 8.0 */
    animation-name: example;
    animation-duration: 4s;
}
/* Safari 4.0 - 8.0 */
@-webkit-keyframes example {
```

```
    from {background-color: red;}
    to {background-color: yellow;}
}

/* Standard syntax */
@keyframes example {
    from {background-color: red;}
    to {background-color: yellow;}
}
</style>
</head>
<body>
<p><b>Note:</b> This example does not work in Internet Explorer 9 and earlier versions.</p>
<div></div>
<p><b>Note:</b> When an animation is finished, it changes back to its original style.</p>
</body>
</html>
```

Problem:Docker swarm implementation and load balancer example.

Before we start, lets understand some concepts.

Swarm cluster:

A swarm is a group of machines that are running Docker and joined into a cluster. After we execute swarm command , we can continue to run the docker commands but the only difference will be that after that the container will start running on a cluster by a swarm manager.

The machines in a swarm can be physical or virtual. And after joining a swarm, they are referred to as nodes.

Note: Enabling swarm mode instantly makes the current machine a swarm manager.

How to enable swarm?

Basically a swarm is made up of multiple nodes, which can be anything physical or virtual machines. To enable swarm we have to execute the following command as given below:

```
$ docker swarm init
```

We can join another machines in swarm as workers by runing the command as given below:

```
$ docker swarm join
```

Docker Machine:

The Docker machine can create and tear down whole fleets of Docker enabled hosts in a second.

This Docker machine ships with a numbers of drivers which integrates Docker machine with a different virtual machines technology or cloud based virtual computing providers.

Step1: We will create three docker hosts by using a driver virtualbox.

We can create a host using the command given below:

```
$ docker-machine create –driver virtualbox host1
```

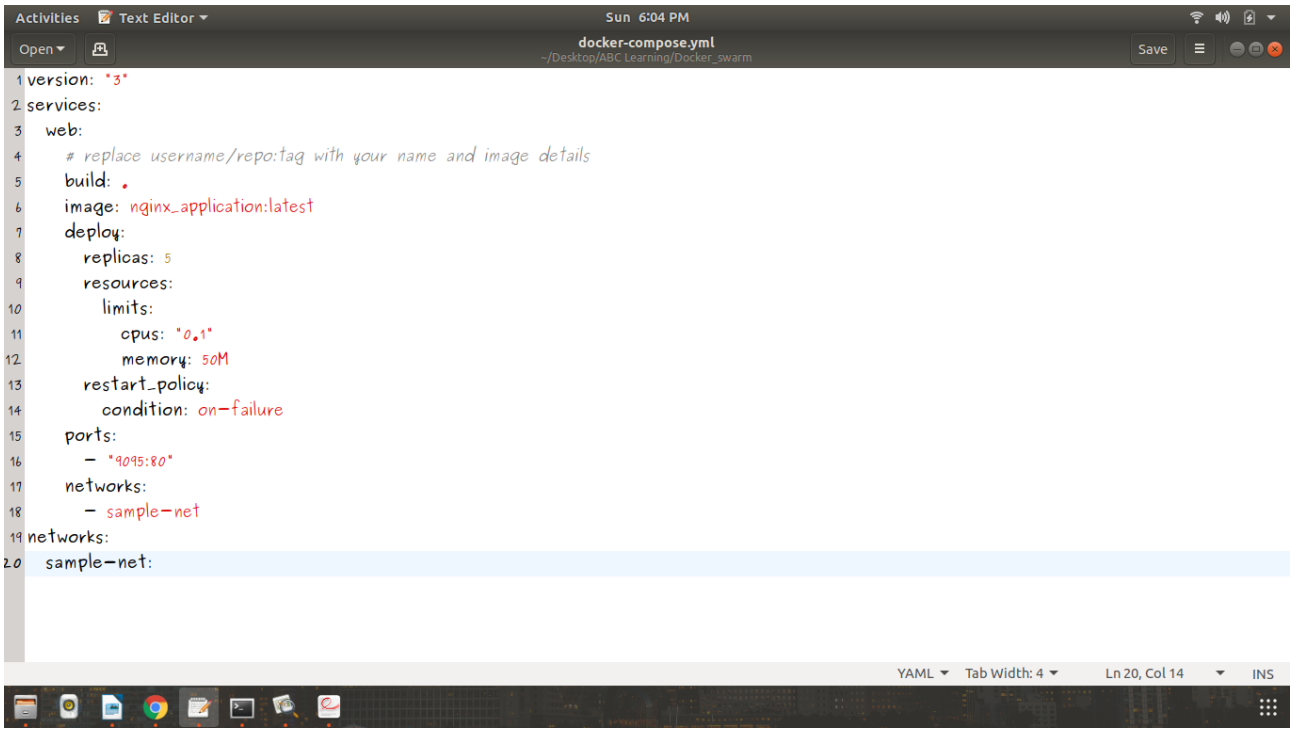
To install Docker-machine run this command:

```
root@anay:~#  base=https://github.com/docker/machine/releases/download/v0.14.0 &&
>    curl -L $base/docker-machine-$(uname -s)-$(uname -m) >/tmp/docker-machine &&
>    sudo install /tmp/docker-machine /usr/local/bin/docker-machine
```

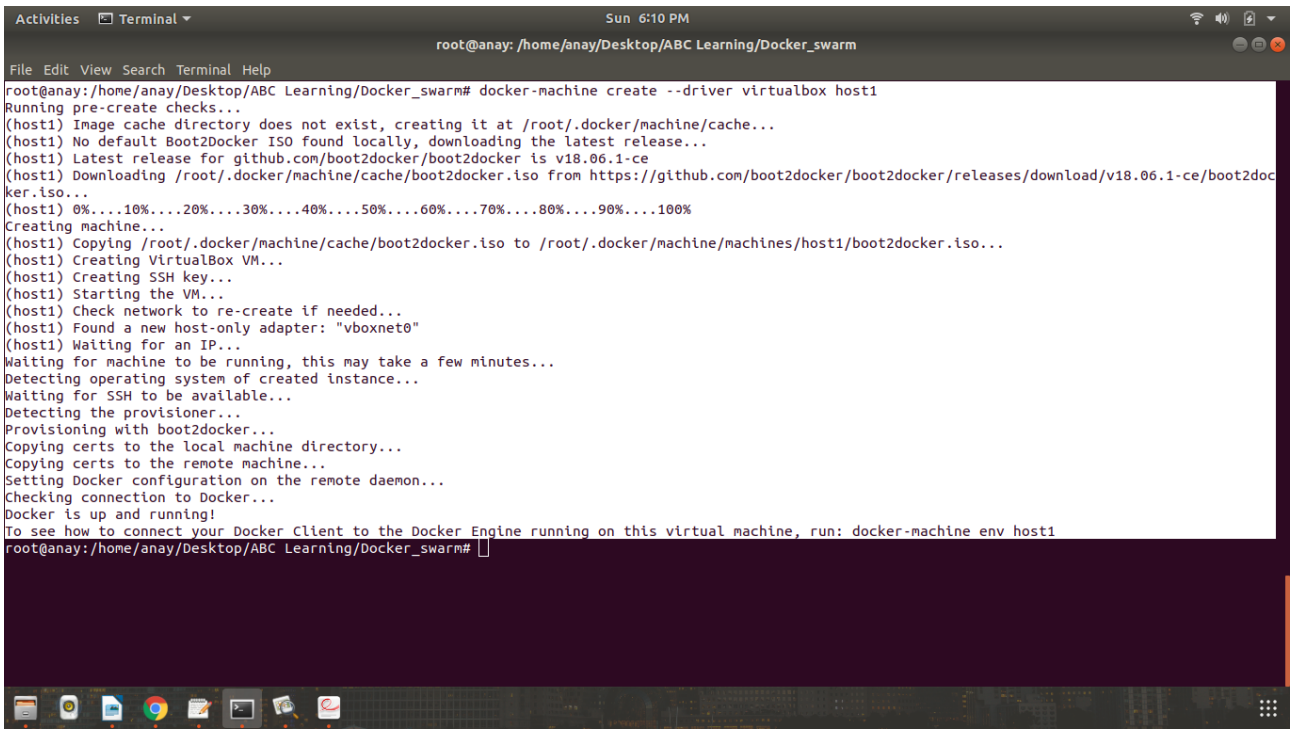
Now Let's take an example:

We will deploy a simple static web app which is running on nginx.

Step1: create a docker-compose.yml file and Dockerfile , index.html is same as we used in pervious examples.



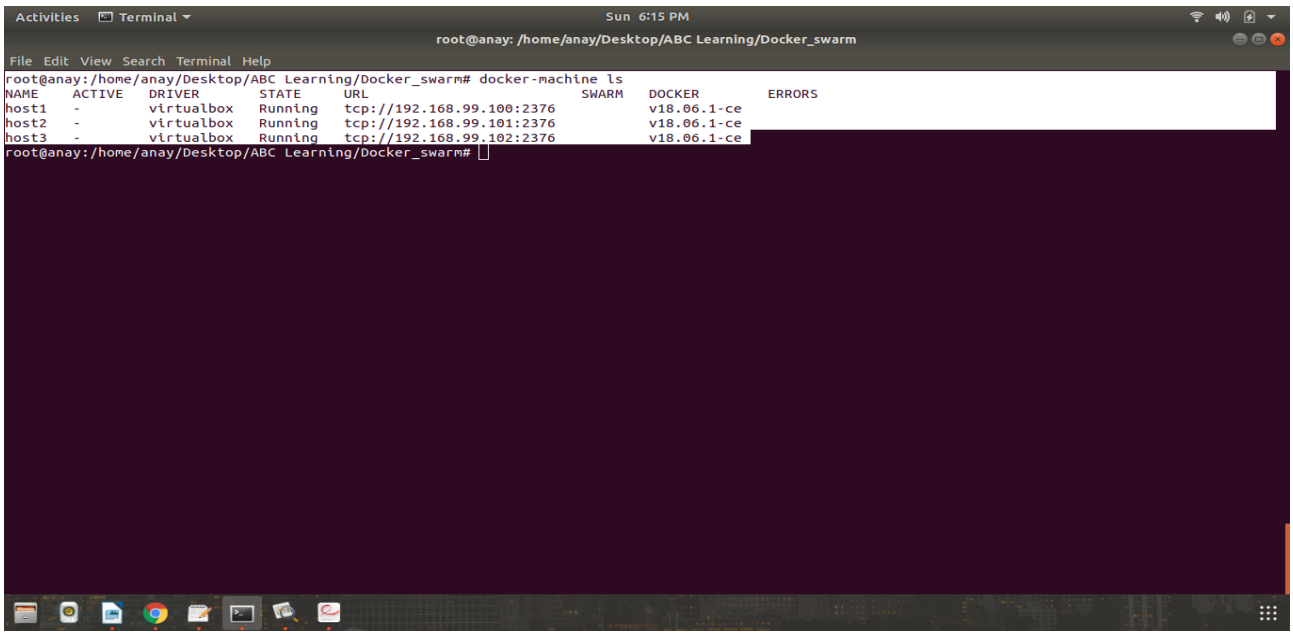
Now, let's create three physical virtual machine using virtualbox driver.



In the same way create two more machnes.

And you can verfiy that they have created or not using a command given below:

```
$ docker-machine ls
```

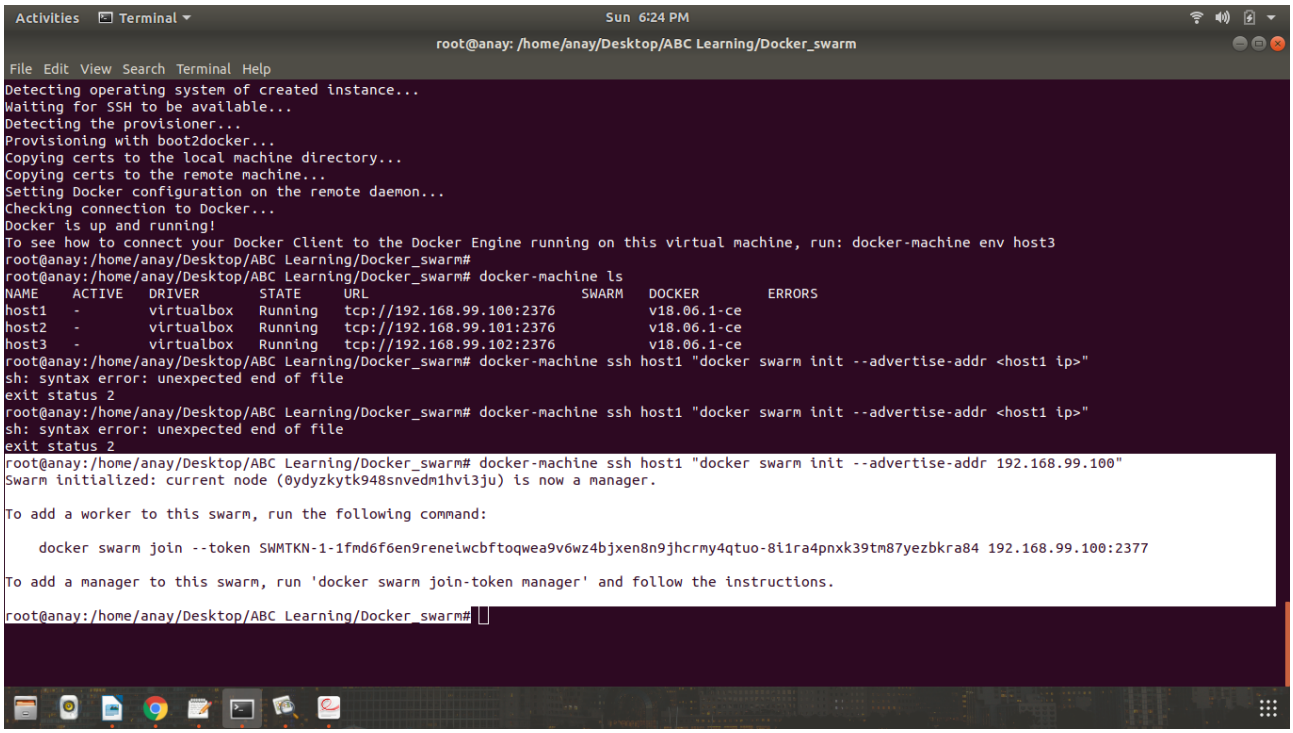


Setp2: Now initialize the swarm and add nodes.

The first mchine acts as the manager, which will exectue management commands and authenticates workers to join the swarm.

Here we will send a command to a machine using a command given below:

```
$ docker-machine ssh <machinename> “<commands>”
```



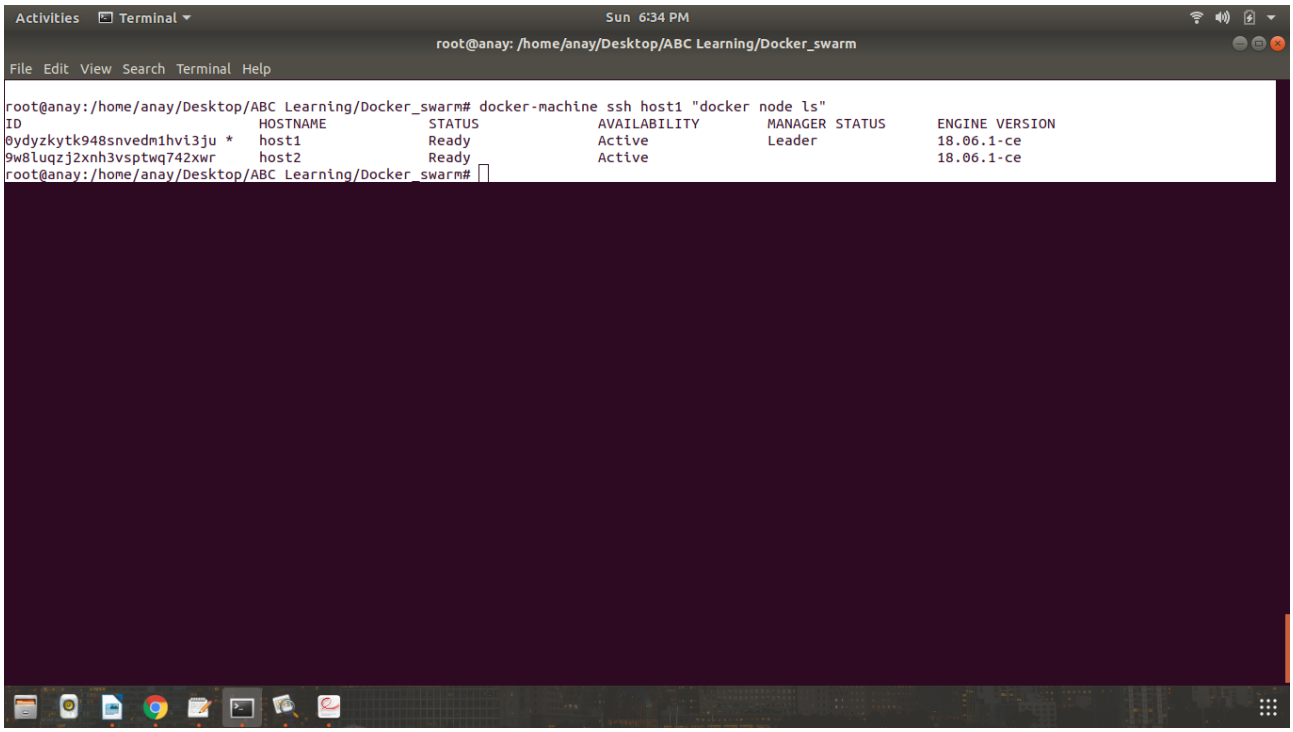
In this step, we have started swarm and add host1 as manager.

Netx, we will add host2 in swarm,

```
root@anay:/home/anay/Desktop/ABC Learning/Docker_swarm# docker-machine ssh host2 "docker swarm join --token SWMTKN-1-1fmd6f6en9reneiwcbftoqwea9v6wz4bjxen8n9jhcrmy4qtuo-8i1ra4pnxk39tm87yezbkra84 192.168.99.100:2377"

This node joined a swarm as a worker.

root@anay:/home/anay/Desktop/ABC Learning/Docker_swarm#
```

Step3: Now we will verify that host2 acts as a node for host1 or not. To do this we need to run a command as given below:

$docker-machine ssh <machine_name> “docker node ls”



Step4: we will deploy our application.

Now hard part is over and we will deploy our application in swarm.

So far , we have been using docker commands in docker-machine ssh to talk to the virtual machines. There is one more option to run docker-machine env <machine_name> to get and run a command that configures our current shell to talk to Docker daemon on the VM.

root@anay:/home/anay/Desktop/ABC Learning/Docker_swarm# docker-machine env host1

export DOCKER_TLS_VERIFY="1"

export DOCKER_HOST="tcp://192.168.99.100:2376"

export DOCKER_CERT_PATH="/root/.docker/machine/machines/host1"

export DOCKER_MACHINE_NAME="host1"

Run this command to configure your shell:

eval $(docker-machine env host1)

So as it is given that we need to run eval $(docker-machine env host1) to configure our shell to talk to host1.

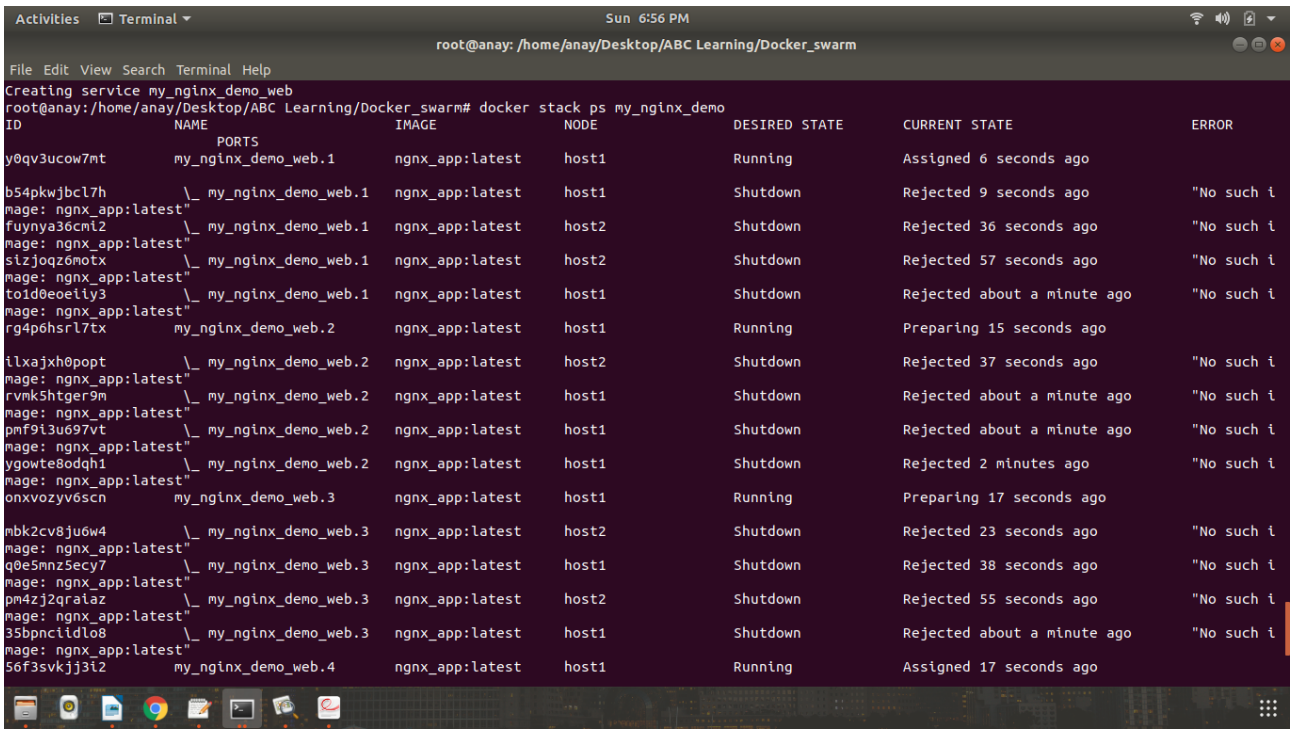
```
root@anay:/home/anay/Desktop/ABC Learning/Docker_swarm# eval $(docker-machine env host1)

root@anay:/home/anay/Desktop/ABC Learning/Docker_swarm# docker-machine ls

NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS

host1 * virtualbox Running tcp://192.168.99.100:2376 v18.06.1-ce

host2 - virtualbox Running tcp://192.168.99.101:2376 v18.06.1-ce

host3 - virtualbox Running tcp://192.168.99.102:2376 v18.06.1-ce
```

As we can see * to left of host1 which indicates that host1 is active.

Netx, we will deploy ur application.

```
root@anay:/home/anay/Desktop/ABC Learning/Docker_swarm# docker stack deploy -c docker-compose.yml my_nginx_demo

Creating network my_nginx_demo_sample-net

Creating service my_nginx_demo_web

root@anay:/home/anay/Desktop/ABC Learning/Docker_swarm#
```

Now, to check what are the services running as given below:



Now, you can see this application. Open your browser and enter the IP of host1 and we can see the same page as we have seen in pervious example.

Problem: Execute a script or application using docker exec commad

The docker exec command provides the needed help to users, who are trying to deploy their application which is running in background.

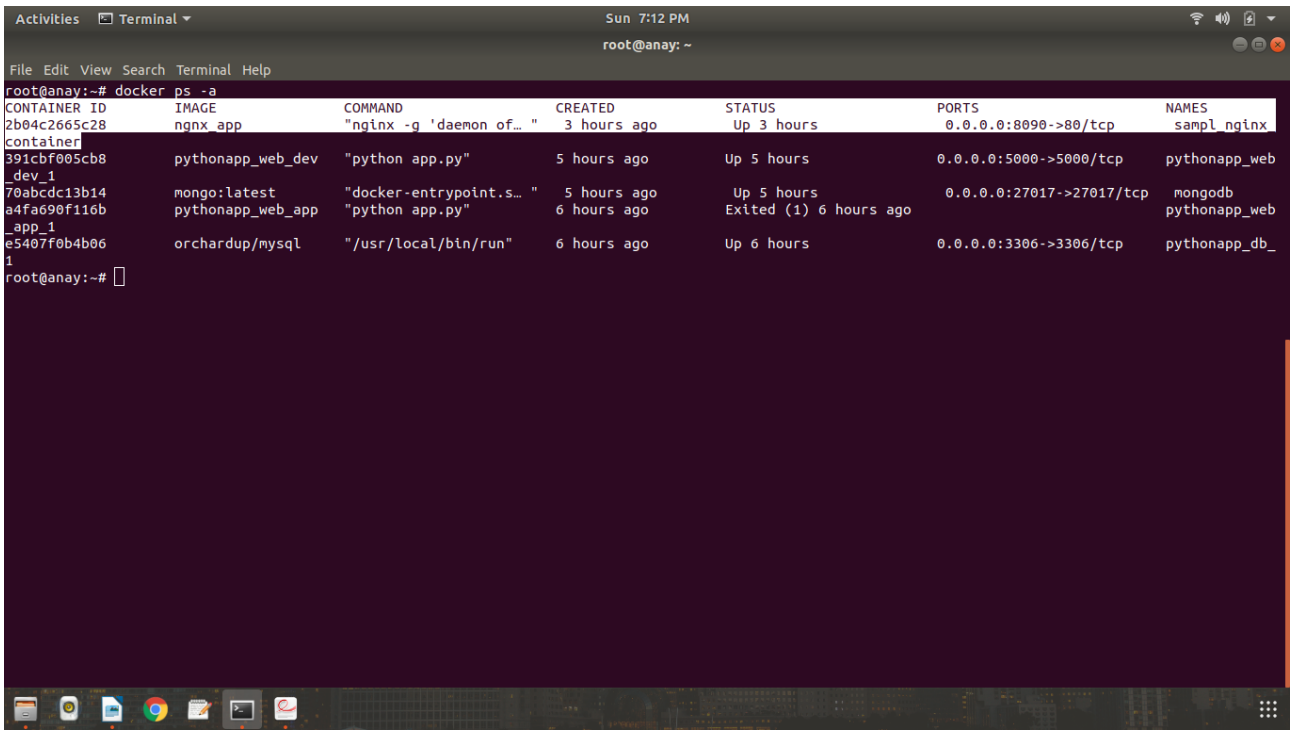
Because of this command, it is not necessary to log in to runn the SSH daemon in the container.

NOTE: docker exec command can only access the running containers. This command spawns a new process in the target container using Docker API and CLI. To check that we can run pe -aef command inside the target container.

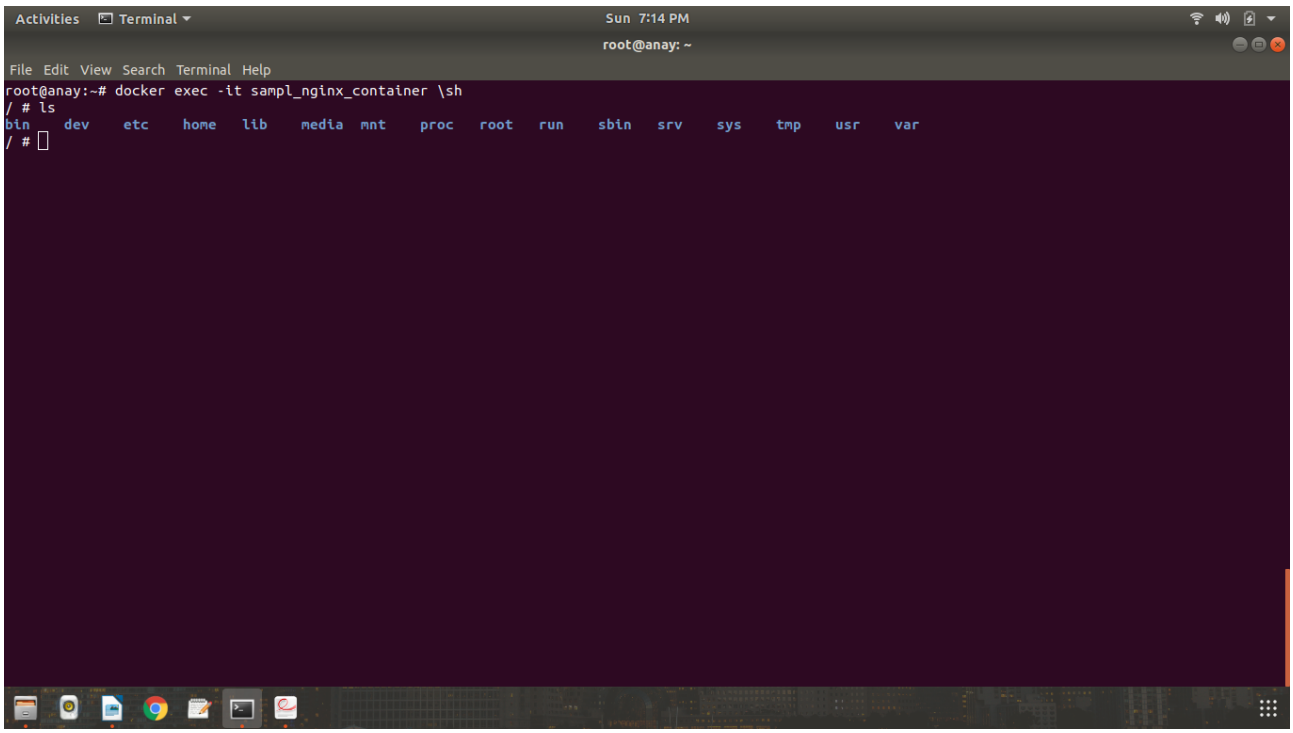
Example:

We have some container like nginx_app which we have created in VOLUME section.

Type docker ps -a.



Now, we will run docker exec command and will enter inside the container.



As you can see we entered inside the container.