Docker Tutorial

Docker Tutorial provides basic and advanced concepts of Docker. Our Docker Tutorial is designed for both beginners as well as professionals.

Docker is a centralized platform for packaging, deploying, and running applications. Before Docker, many users face the problem that a particular code is running in the developer's system but not in the user's system. So, the main reason to develop docker is to help developers to develop applications easily, ship them into containers, and can be deployed anywhere.

Docker was firstly released in March 2013. It is used in the Deployment stage of the software development life cycle that's why it can efficiently resolve issues related to the application deployment.

What is Docker?

Docker is an **open-source centralized platform designed** to create, deploy, and run applications. Docker uses **container** on the host's operating system to run applications. It allows applications to use the same **Linux kernel** as a system on the host computer, rather than creating a whole virtual operating system. Containers ensure that our application works in any environment like development, test, or production.

Docker includes components such as **Docker client, Docker server, Docker machine, Docker hub, Docker composes,** etc.

Let's understand the Docker containers and virtual machine.

Docker Containers

Docker containers are the **lightweight** alternatives of the virtual machine. It allows developers to package up the application with all its libraries and dependencies, and ship it as a single package. The advantage of using a docker container is that you don't need to allocate any RAM and disk space for the applications. It automatically generates storage and space according to the application requirement.

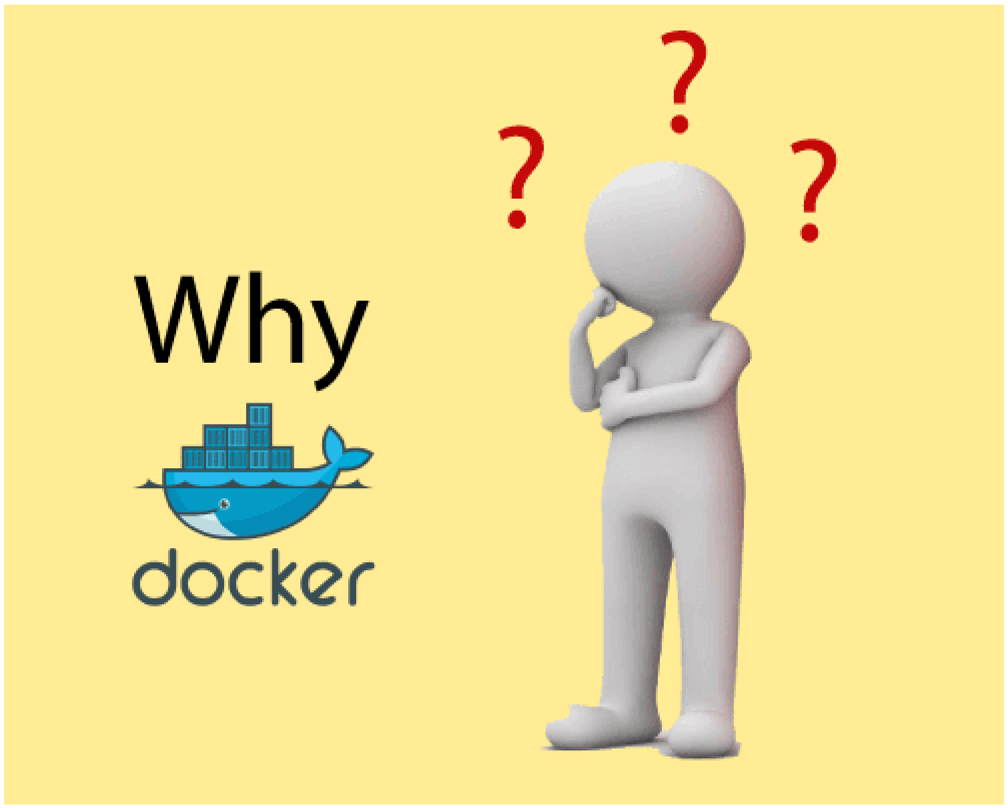
Virtual Machine

A virtual machine is a software that allows us to install and use other operating systems (Windows, Linux, and Debian) simultaneously on our machine. The operating system in which virtual machine runs are called virtualized operating systems. These virtualized operating systems can run programs and preforms tasks that we perform in a real operating system.

Containers Vs. Virtual Machine

Containers	**Virtual Machine**
Integration in a container is faster and cheap.	Integration in virtual is slow and costly.
No wastage of memory.	Wastage of memory.
It uses the same kernel, but different distribution.	It uses multiple independent operating systems.

Why Docker?



Docker is designed to benefit both the Developer and System Administrator. There are the following reasons to use Docker -

- Docker allows us to easily install and run software without worrying about setup or dependencies.
- Developers use Docker to eliminate machine problems, i.e. "**but code is worked on my laptop**." when working on code together with co-workers.
- Operators use Docker to run and manage apps in isolated containers for better compute density.
- Enterprises use Docker to securely built agile software delivery pipelines to ship new application features faster and more securely.
- Since docker is not only used for the deployment, but it is also a great platform for development, that's why we can efficiently increase our customer's satisfaction.

Advantages of Docker

There are the following advantages of Docker -

- It runs the container in seconds instead of minutes.
- It uses less memory.
- It provides lightweight virtualization.
- It does not a require full operating system to run applications.
- It uses application dependencies to reduce the risk.
- Docker allows you to use a remote repository to share your container with others.
- It provides continuous deployment and testing environment.

Disadvantages of Docker

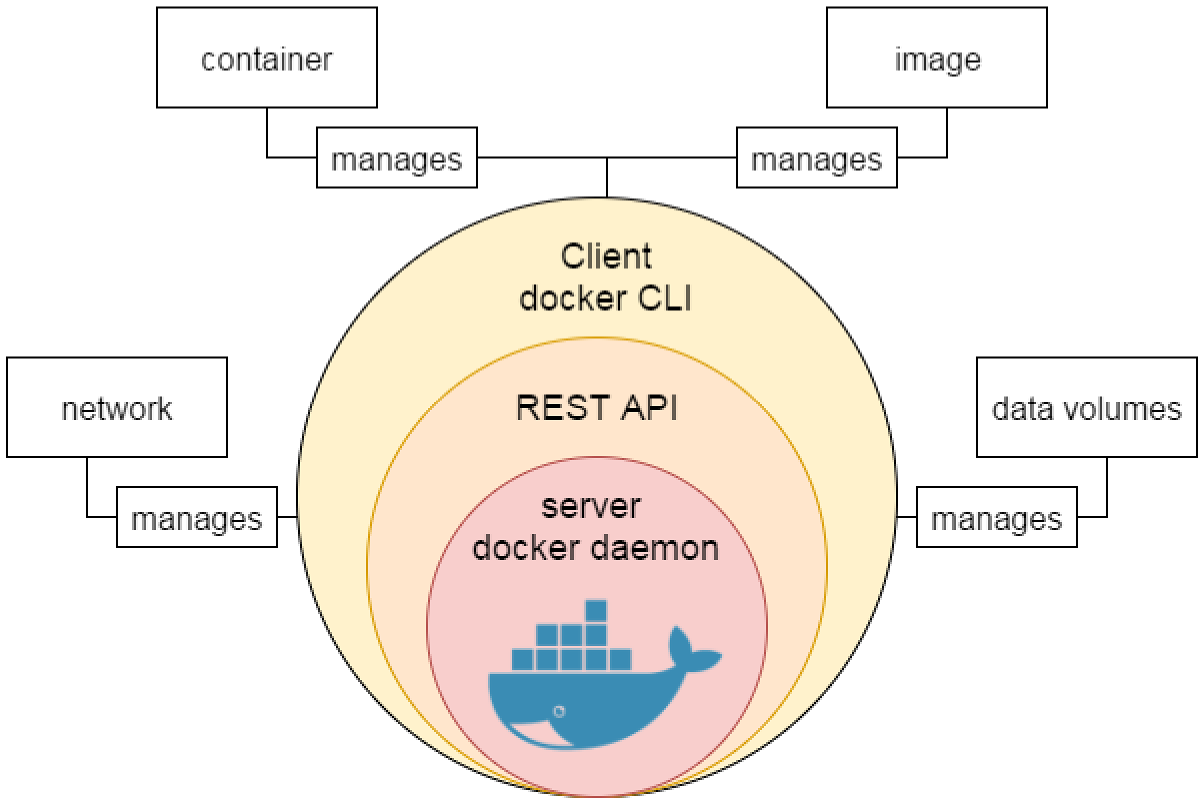
There are the following disadvantages of Docker -

- It increases complexity due to an additional layer.
- In Docker, it is difficult to manage large amount of containers.
- Some features such as container self -registration, containers self-inspects, copying files form host to the container, and more are missing in the Docker.
- Docker is not a good solution for applications that require rich graphical interface.
- Docker provides cross-platform compatibility means if an application is designed to run in a Docker container on Windows, then it can't run on Linux or vice versa.

Docker Engine

It is a client server application that contains the following major components.

- A server which is a type of long-running program called a daemon process.
- The REST API is used to specify interfaces that programs can use to talk to the daemon and instruct it what to do.
- A command line interface client.



Prerequisite

Before learning Docker, you must have the fundamental knowledge of Linux and programming languages such as java, php, python, ruby, etc.

Docker Architecture

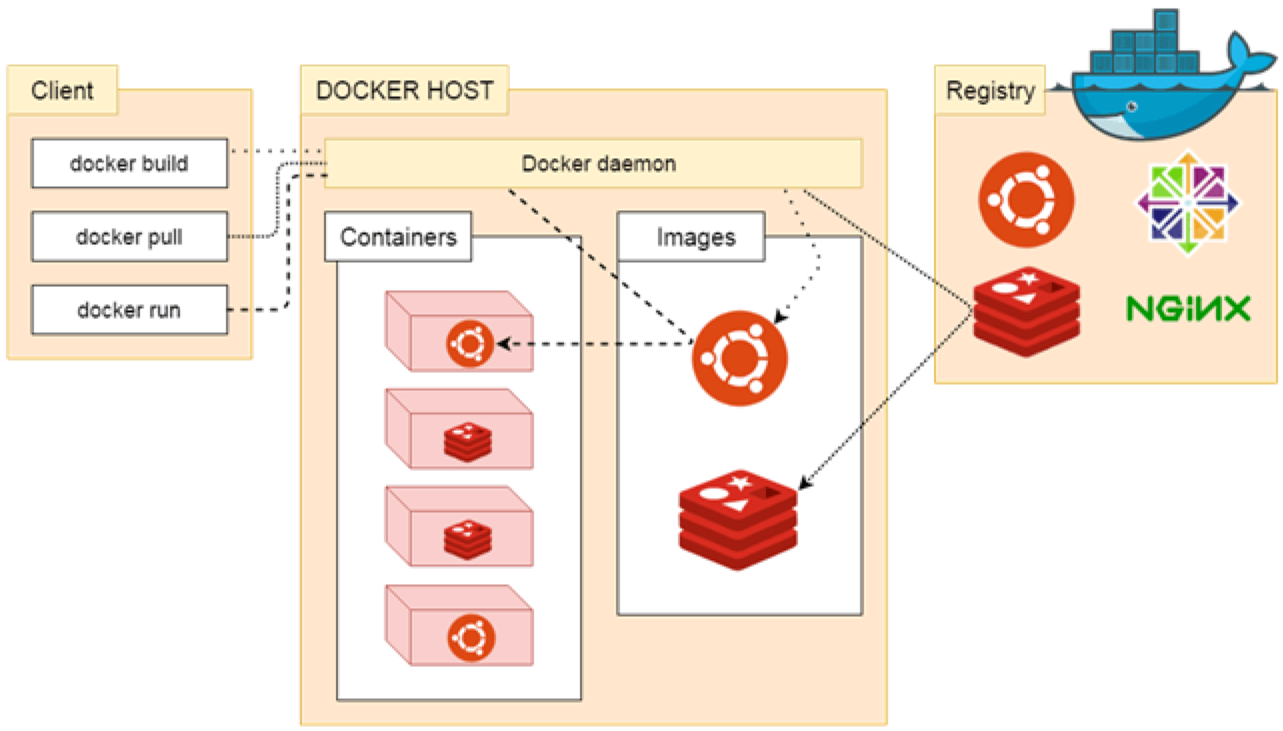
Before learning the Docker architecture, first, you should know about the Docker Daemon.

What is Docker daemon?

Docker daemon runs on the host operating system. It is responsible for running containers to manage docker services. Docker daemon communicates with other daemons. It offers various Docker objects such as images, containers, networking, and storage. s

Docker architecture

Docker follows Client-Server architecture, which includes the three main components that are **Docker Client**, **Docker Host**, and **Docker Registry**.



1. Docker Client

Docker client uses **commands** and **REST APIs** to communicate with the Docker Daemon (Server). When a client runs any docker command on the docker client terminal, the client terminal sends these docker commands to the Docker daemon. Docker daemon receives these commands from the docker client in the form of command and REST API's request.

Note: Docker Client has an ability to communicate with more than one docker daemon.

Docker Client uses Command Line Interface (CLI) to run the following commands -

docker build

docker pull

docker run

2. Docker Host

Docker Host is used to provide an environment to execute and run applications. It contains the docker daemon, images, containers, networks, and storage.

3. Docker Registry

Docker Registry manages and stores the Docker images.

There are two types of registries in the Docker -

Pubic Registry - Public Registry is also called as **Docker hub**.

Private Registry - It is used to share images within the enterprise.

Docker Objects

There are the following Docker Objects -

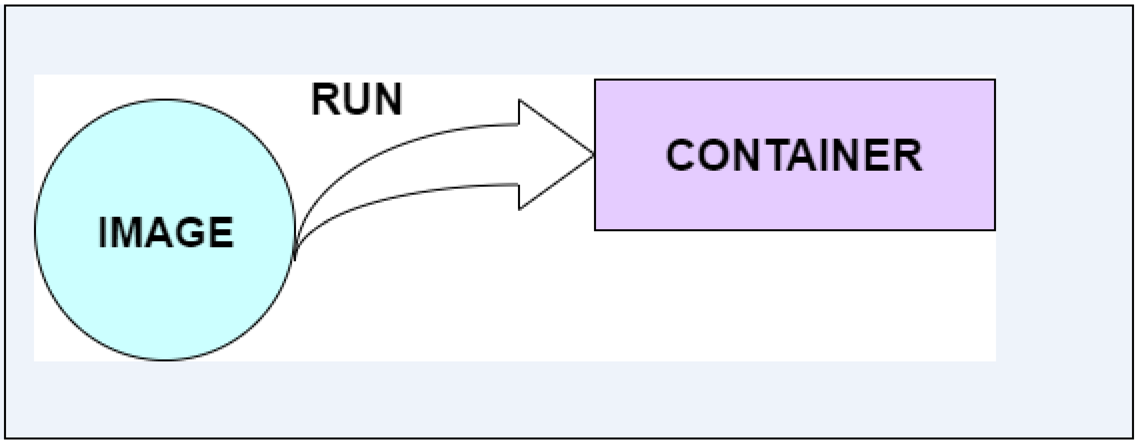
Docker Images

Docker images are the **read-only binary templates** used to create Docker Containers. It uses a private container registry to share container images within the enterprise and also uses public container registry to share container images within the whole world. Metadata is also used by docket images to describe the container's abilities.

Docker Containers

Containers are the structural units of Docker, which is used to hold the entire package that is needed to run the application. The advantage of containers is that it requires very less resources.

In other words, we can say that the image is a template, and the container is a copy of that template.



Docker Networking

Using Docker Networking, an isolated package can be communicated. Docker contains the following network drivers -

- **Bridge -** Bridge is a default network driver for the container. It is used when multiple docker communicates with the same docker host.
- **Host -** It is used when we don't need for network isolation between the container and the host.
- **None -** It disables all the networking.
- **Overlay -** Overlay offers Swarm services to communicate with each other. It enables containers to run on the different docker host.
- **Macvlan -** Macvlan is used when we want to assign MAC addresses to the containers.

Docker Storage

Docker Storage is used to store data on the container. Docker offers the following options for the Storage -

- **Data Volume -** Data Volume provides the ability to create persistence storage. It also allows us to name volumes, list volumes, and containers associates with the volumes.
- **Directory Mounts -** It is one of the best options for docker storage. It mounts a host's directory into a container.
- **Storage Plugins -** It provides an ability to connect to external storage platforms.

Docker Dockerfile

A Dockerfile is a text document that contains commands that are used to assemble an image. We can use any command that call on the command line. Docker builds images automatically by reading the instructions from the Dockerfile.

The docker build command is used to build an image from the Dockerfile. You can use the -f flag with docker build to point to a Dockerfile anywhere in your file system.

```
1. $ docker build -f /path/to/a/Dockerfile .
```

Dockerfile Instructions

The instructions are not case-sensitive but you must follow conventions which recommend to use uppercase.

Docker runs instructions of Dockerfile in top to bottom order. The first instruction must be **FROM** in order to specify the Base Image.

A statement begin with # treated as a comment. You can use RUN, CMD, FROM, EXPOSE, ENV etc instructions in your Dockerfile.

Here, we are listing some commonly used instructions.

FROM

This instruction is used to set the Base Image for the subsequent instructions. A valid Dockerfile must have FROM as its first instruction.

Ex.

1. FROM ubuntu

LABEL

We can add labels to an image to organize images of our project. We need to use LABEL instruction to set label for the image.

Ex.

1. LABEL vendorl = "Example"

RUN

This instruction is used to execute any command of the current image.

Ex.

1. RUN /bin/bash -c 'source $HOME/.bashrc; echo $HOME'

CMD

This is used to execute application by the image. We should use CMD always in the following form

1. CMD ["executable", "param1", "param2"?]

This is preferred way to use CMD. There can be only one CMD in a Dockerfile. If we use more than one CMD, only last one will execute.

COPY

This instruction is used to copy new files or directories from source to the filesystem of the container at the destination.

Ex.

1. COPY abc/ /xyz

Rules

- The source path must be inside the context of the build. We cannot COPY ../something /something because the first step of a docker build is to send the context directory (and subdirectories) to the docker daemon.
- If source is a directory, the entire contents of the directory are copied including filesystem metadata.

WORKDIR

The WORKDIR is used to set the working directory for any RUN, CMD and COPY instruction that follows it in the Dockerfile. If work directory does not exist, it will be created by default.

We can use WORKDIR multiple times in a Dockerfile.

Ex.

1. WORKDIR /var/www/html

Docker Java Application Example

As, we have mentioned earlier that docker can execute any application.

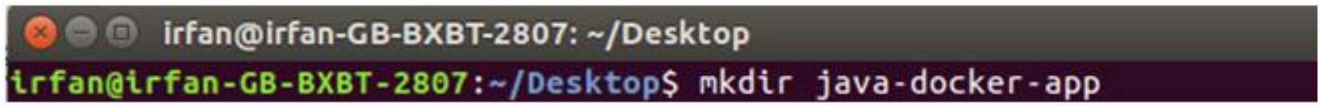
Here, we are creating a Java application and running by using the docker. This example includes the following steps.

1. **Create a directory**

   Directory is required to organize files. Create a director by using the following command.

```
$ mkdir  java-docker-app
```

   See, screen shot for the above command.



```
irfan@irfan-GB-BXBT-2807: ~/Desktop
irfan@irfan-GB-BXBT-2807:~/Desktop$ mkdir  java-docker-app
```

2. **Create a Java File**

   Now create a Java file. Save this file as **Hello.java** file.

   // Hello.java

```java
class Hello{
public static void main(String[] args){
System.out.println("This is java app \n by using Docker");
}
}
```

   Save it inside the directory **java-docker-app** as Hello.java.

3. **Create a Dockerfile**

   After creating a Java file, we need to create a Dockerfile which contains instructions for the Docker. Dockerfile does not contain any file extension. So, save it simple with **Dockerfile** name.

   // Dockerfile

```
FROM java:8
COPY . /var/www/java
WORKDIR /var/www/java
RUN javac Hello.java
CMD ["java", "Hello"]
```

Write all instructions in uppercase because it is convention. Put this file inside **java-docker-app** directory. Now we have Dockerfile parallel to Hello.java inside the **java-docker-app** directory.

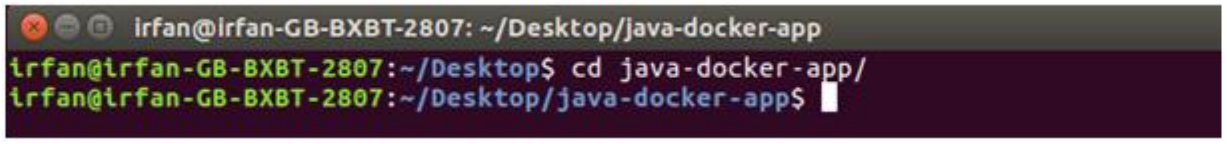
See, your folder inside must look like the below.

4. **Build Docker Image**

After creating Dockerfile, we are changing working directory.

1. $ cd   java-docker-app

See, the screen shot.

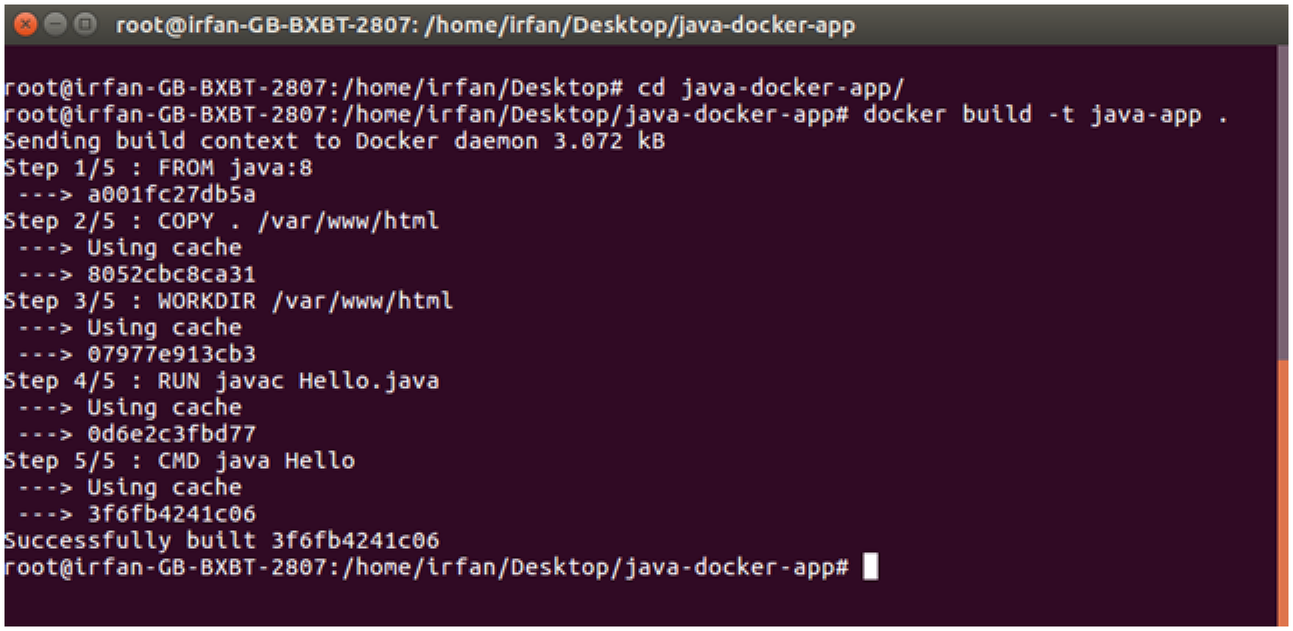


```
irfan@irfan-GB-BXBT-2807: ~/Desktop/java-docker-app
irfan@irfan-GB-BXBT-2807:~/Desktop$ cd java-docker-app/
irfan@irfan-GB-BXBT-2807:~/Desktop/java-docker-app$ 
```

Now, create an image by following the below command. we must login as root in order to create an image. In this example, we have switched to as a root user. In the following command, **java-app**is name of the image. We can have any name for our docker image.

1. $ docker build -t java-app .

See, the screen shot of the above command.



```
root@irfan-GB-BXBT-2807: /home/irfan/Desktop/java-docker-app

root@irfan-GB-BXBT-2807:/home/irfan/Desktop# cd java-docker-app/
root@irfan-GB-BXBT-2807:/home/irfan/Desktop/java-docker-app# docker build -t java-app .
Sending build context to Docker daemon 3.072 kB
Step 1/5 : FROM java:8
 ---> a001fc27db5a
Step 2/5 : COPY . /var/www/html
 ---> Using cache
 ---> 8052cbc8ca31
Step 3/5 : WORKDIR /var/www/html
 ---> Using cache
 ---> 07977e913cb3
Step 4/5 : RUN javac Hello.java
 ---> Using cache
 ---> 0d6e2c3fbd77
Step 5/5 : CMD java Hello
 ---> Using cache
 ---> 3f6fb4241c06
Successfully built 3f6fb4241c06
root@irfan-GB-BXBT-2807:/home/irfan/Desktop/java-docker-app#
```

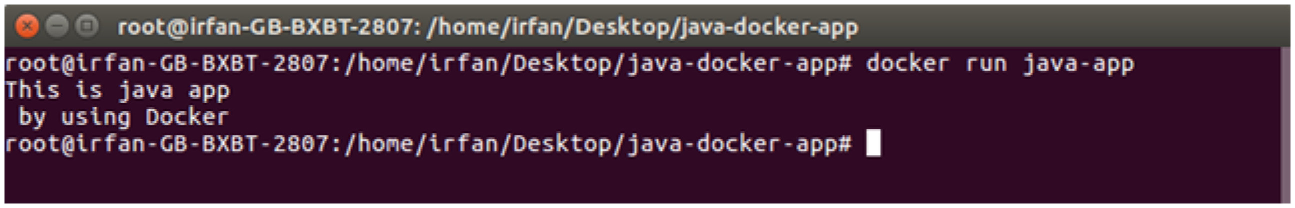
After successfully building the image. Now, we can run our docker image.

5. **Run Docker Image**

After creating image successfully. Now we can run docker by using run command. The following command is used to run java-app.

1. $ docker run java-app

See, the screen shot of the above command.



```
root@irfan-GB-BXBT-2807: /home/irfan/Desktop/java-docker-app
root@irfan-GB-BXBT-2807:/home/irfan/Desktop/java-docker-app# docker run java-app
This is java app
 by using Docker
root@irfan-GB-BXBT-2807:/home/irfan/Desktop/java-docker-app#
```

Here, we can see that after running the java-app it produced an output.

Now, we have run docker image successfully on your system. Apart from all these you can also use other commands as well.

Docker Useful Commands

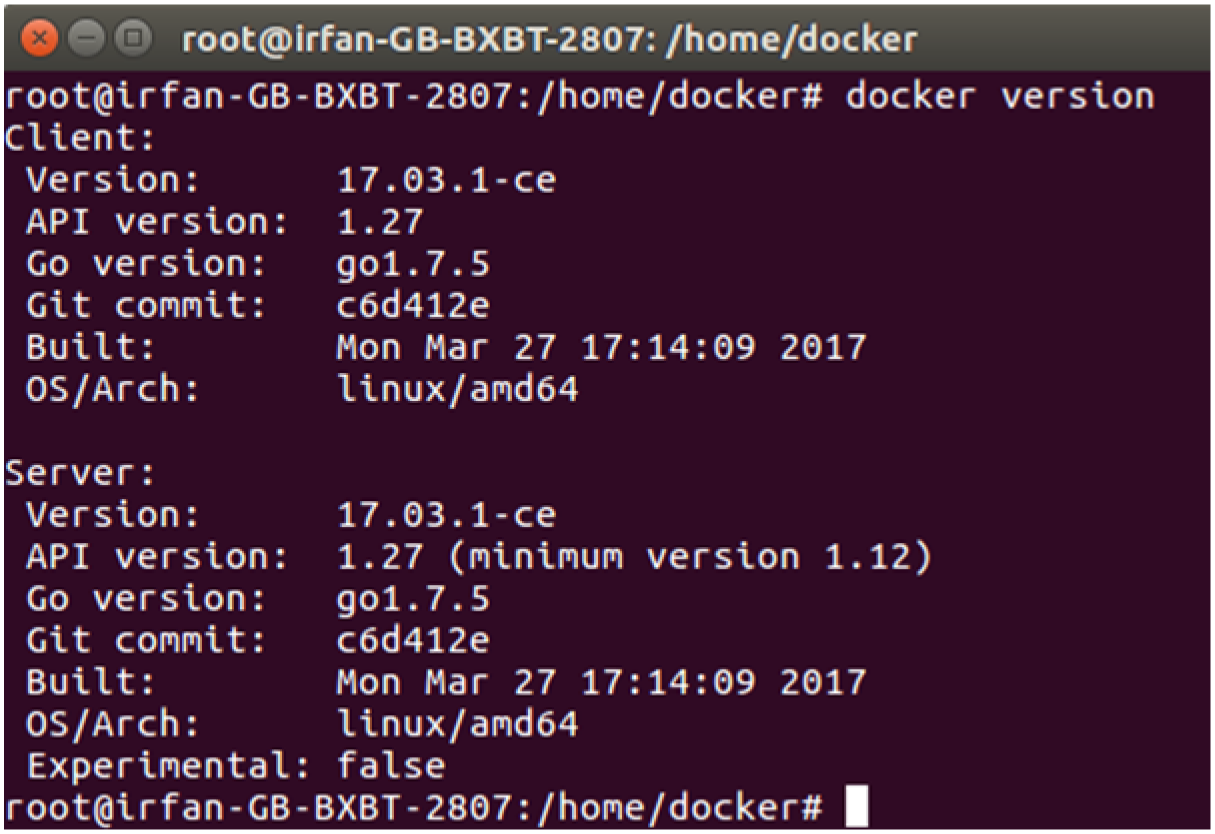
Docker is natively Linux based software so that it provides commands to interact and work in the client-server environment.

Here, we have listed some important and useful Docker commands.

Check Docker version

1. $ docker version

It shows docker version for both client and server. As given in the following image.



```
root@irfan-GB-BXBT-2807: /home/docker
root@irfan-GB-BXBT-2807:/home/docker# docker version
Client:
 Version:      17.03.1-ce
 API version:  1.27
 Go version:   go1.7.5
 Git commit:   c6d412e
 Built:        Mon Mar 27 17:14:09 2017
 OS/Arch:      linux/amd64

Server:
 Version:      17.03.1-ce
 API version:  1.27 (minimum version 1.12)
 Go version:   go1.7.5
 Git commit:   c6d412e
 Built:        Mon Mar 27 17:14:09 2017
 OS/Arch:      linux/amd64
 Experimental: false
root@irfan-GB-BXBT-2807:/home/docker#
```

Build Docker Image from a Dockerfile

1. $ docker build -t image-name docker-file-location

-t : it is used to tag Docker image with the provided name.

Run Docker Image

1. $ docker run -d image-name

-d : It is used to create a daemon process.

Check available Docker images

1. $ docker images

Check for latest running container

1. $ docker ps -l

-l : it is used to show latest available container.

Check all running containers

1. $ docker ps -a

-a : It is used to show all available containers.

Stop running container

1. $ docker stop container_id

container_id : It is an Id assigned by the Docker to the container.

Delete an image

1. $ docker rmi image-name

Delete all images

1. $ docker rmi $(docker images -q)

Delete all images forcefully

1. $ docker rmi -r $(docker images -q)

-r : It is used to delete image forcefully.

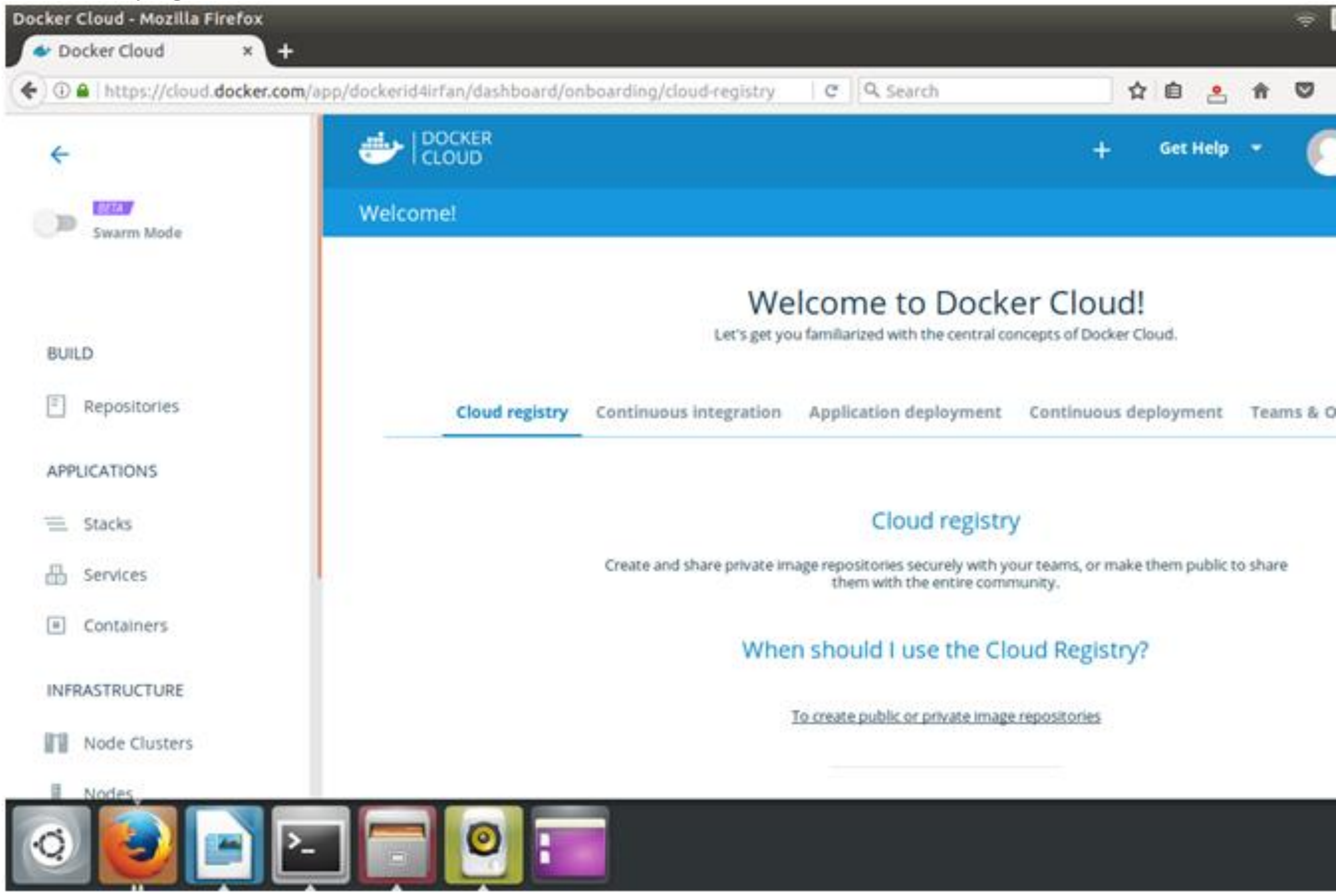
Delete all containers

1. $ docker rm $(docker ps -a -q)

Enter into Docker container

1. $ docker exec -it container-id bash

2. Docker Cloud

3. Docker provides us the facility to store and fetch docker images on the cloud registry. We can store dockerized images either privately or publicly. It is a full GUI interface that allows us to manage builds, images, swarms, nodes and apps.
4. We need to have Docker ID to access and control images. If we don't have, create it first.
5. Here, in the following screenshot, we have logged in to Docker cloud. It shows a welcome page.
6. 

Welcome to Docker Cloud!

Let's get you familiarized with the central concepts of Docker Cloud.

Cloud registry

Create and share private image repositories securely with your teams, or make them public to share them with the entire community.

When should I use the Cloud Registry?

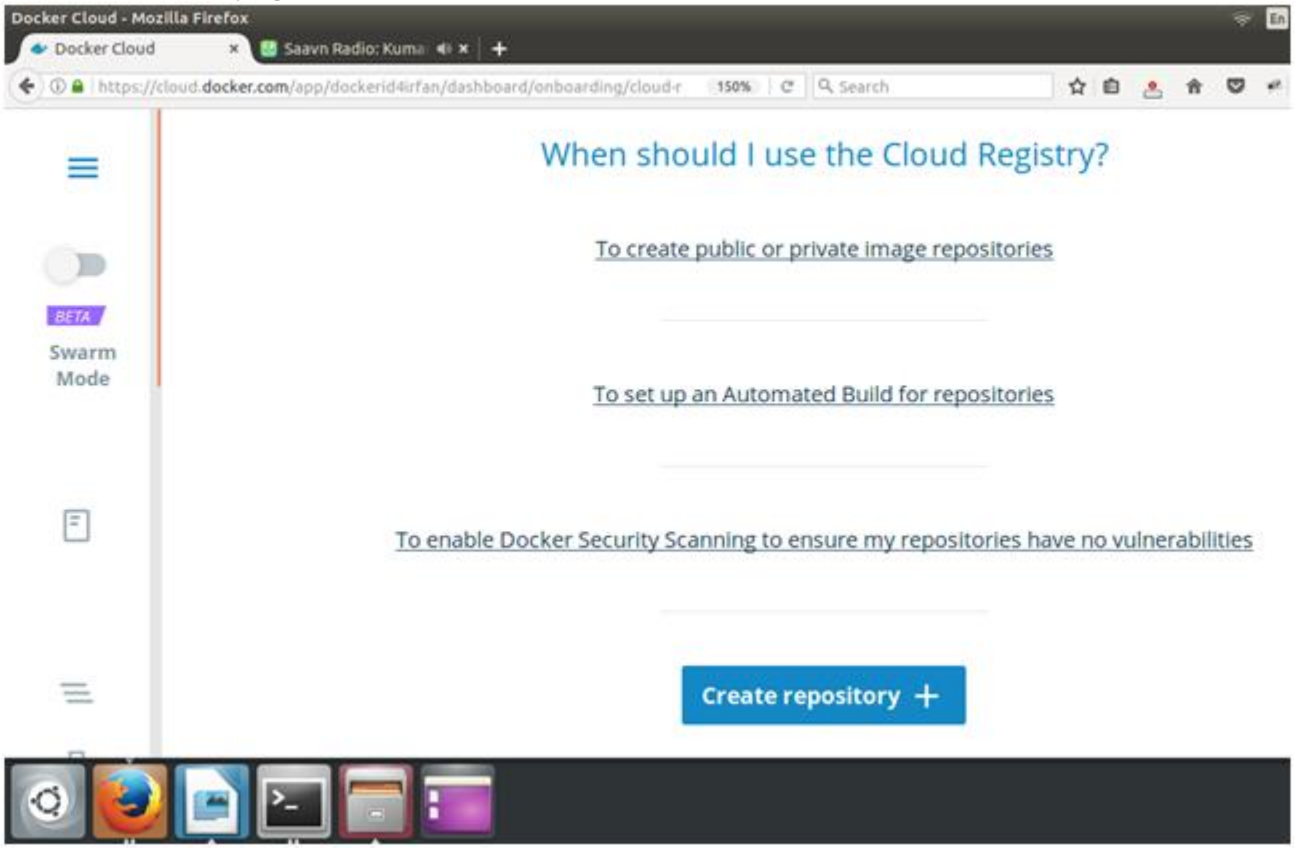
To create public or private image repositories

7. In the left panel, we can see that it provides lots of functionalities that we use on the cloud. Apart from all these, let's create a repository first.
8.

---

9. Creating Repository

10. To create Docker cloud repository, click on the create repository +button available on the welcome page at the bottom.


Docker Cloud - Mozilla Firefox
Docker Cloud
Saavn Radio: Kuma
https://cloud.docker.com/app/dockerid4irfan/dashboard/onboarding/cloud-r
150%
Search
BETA
Swarm Mode
Create repository +

When should I use the Cloud Registry?

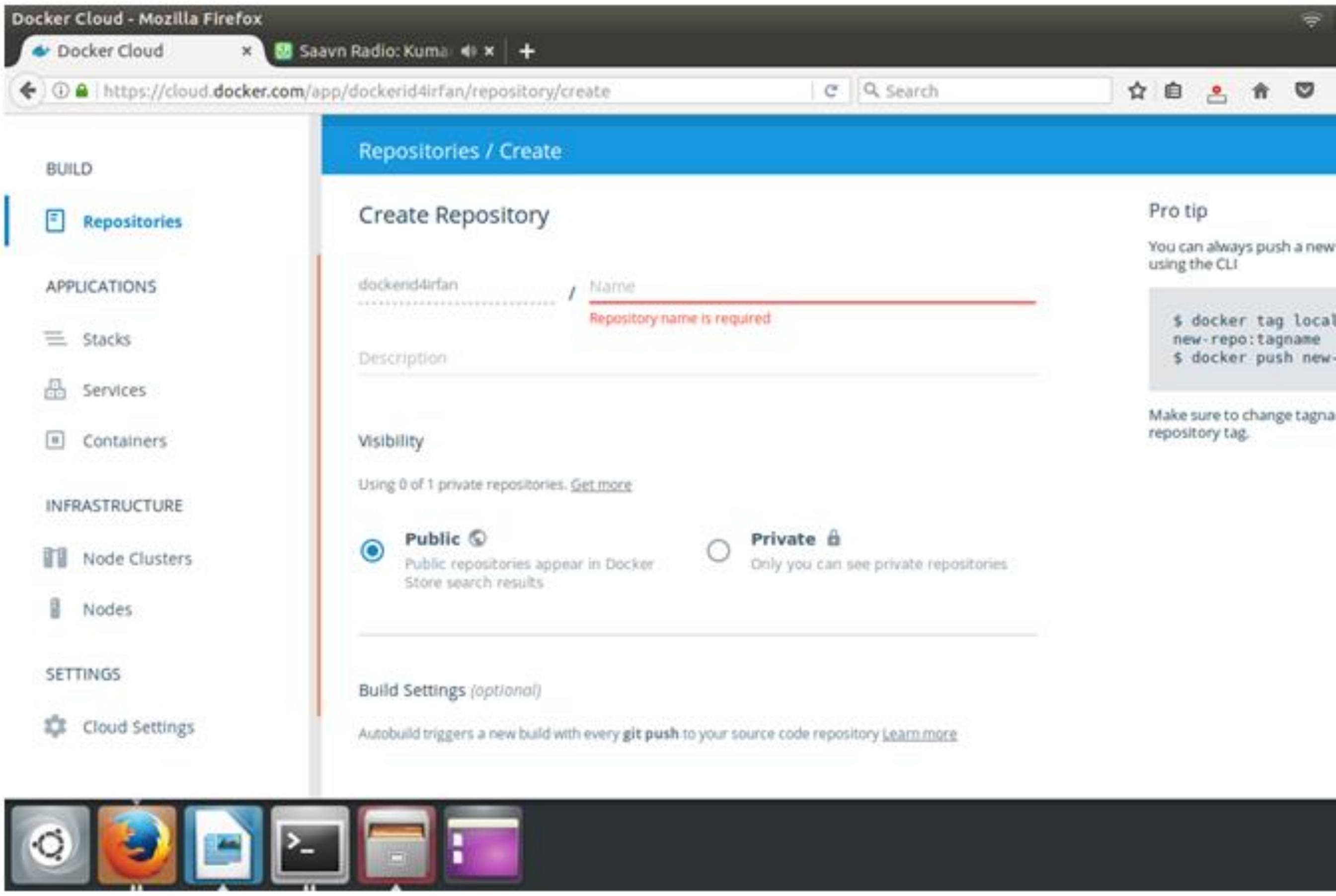
To create public or private image repositories

To set up an Automated Build for repositories

To enable Docker Security Scanning to ensure my repositories have no vulnerabilities

11.

12. After clicking, it displays a form to enter the name of the repository. The page looks looks like the following.



Docker Cloud - Mozilla Firefox

https://cloud.docker.com/app/dockerid4irfan/repository/create

Repositories / Create

Create Repository

dockerid4irfan / Name

Repository name is required

Description

Visibility

Using 0 of 1 private repositories. Get more

Public — Public repositories appear in Docker Store search results

Private — Only you can see private repositories

Build Settings (optional)

Autobuild triggers a new build with every **git push** to your source code repository Learn more

Pro tip

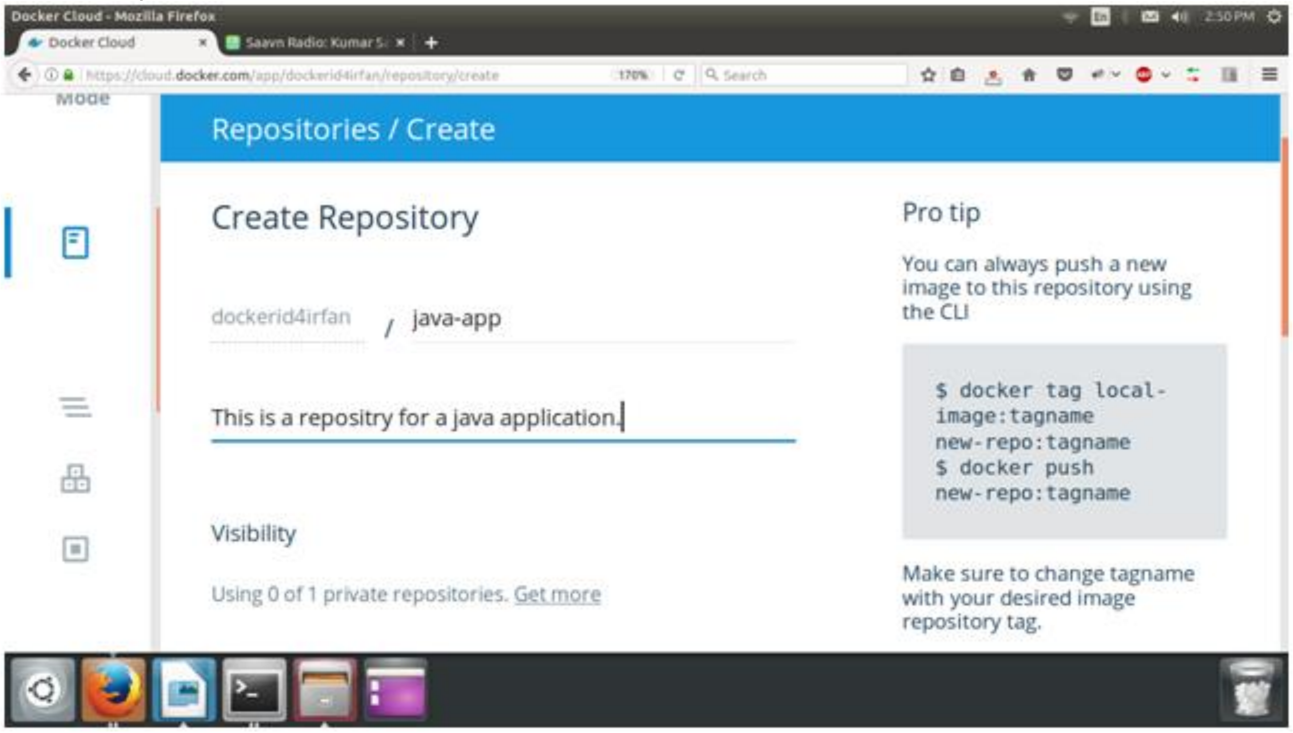
You can always push a new
using the CLI

```
$ docker tag local
new-repo:tagname
$ docker push new-
```

Make sure to change tagna
repository tag.

13.

14. It asks for the repository name to create a new one. The following screen-shot show the description.



Docker Cloud - Mozilla Firefox

https://cloud.docker.com/app/dockerid4irfan/repository/create

Repositories / Create

Create Repository

dockerid4irfan / java-app

This is a repositry for a java application.

Visibility

Using 0 of 1 private repositories. Get more

Pro tip

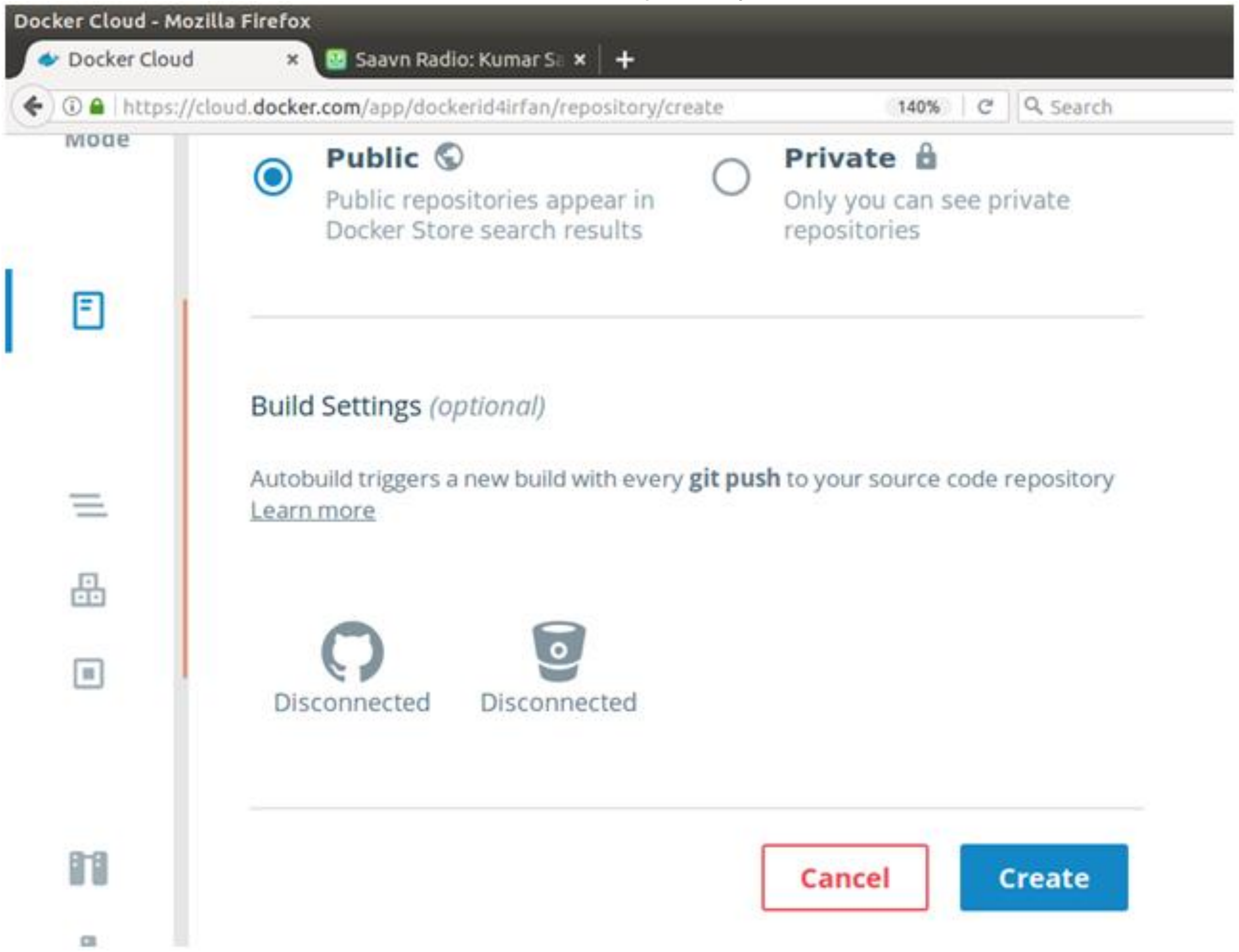
You can always push a new image to this repository using the CLI

```
$ docker tag local-
image:tagname
new-repo:tagname
$ docker push
new-repo:tagname
```

Make sure to change tagname with your desired image repository tag.

15.

16. After filling the details, we should make this repository public. Now, just click on the create button at the bottom. It will create repository for us.



Docker Cloud - Mozilla Firefox

https://cloud.docker.com/app/dockerid4irfan/repository/create

Mode

[✓ Public] Public repositories appear in Docker Store search results

Private — Only you can see private repositories

Build Settings *(optional)*

Autobuild triggers a new build with every **git push** to your source code repository
Learn more

Disconnected   Disconnected

17.
18. So, we can see that it provides the other tools also to manage and control Docker cloud.

Docker Compose

It is a tool which is used to create and start Docker application by using a single command. We can use it to file to configure our application's services.

It is a great tool for development, testing, and staging environments.

It provides the following commands for managing the whole lifecycle of our application.

- Start, stop and rebuild services
- View the status of running services

- Stream the log output of running services
- Run a one-off command on a service

To implement compose, it consists the following steps.

1. Put Application environment variables inside the Dockerfile to access publicly.
2. Provide services name in the docker-compose.yml file so they can be run together in an isolated environment.
3. run docker-compose up and Compose will start and run your entire app.

A typical **docker-compose.yml** file has the following format and arguments.

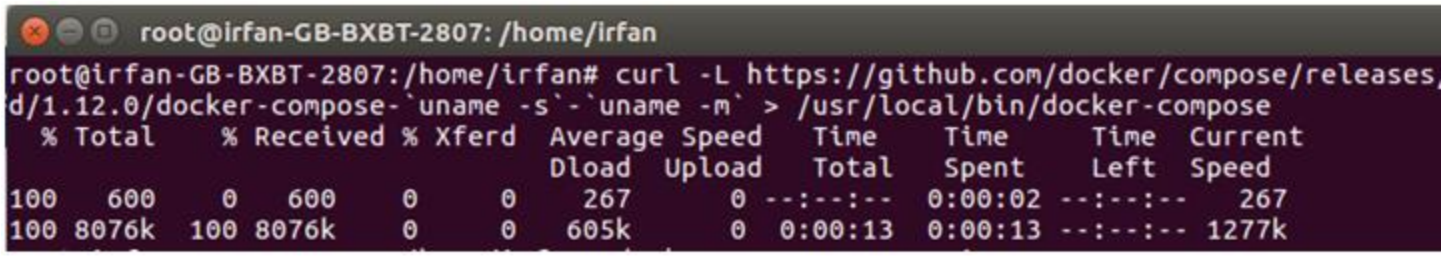
// docker-compose.yml

```
version: '3'
services:
web:
build: .
ports:
- "5000:5000"
volumes:
- .:/code
- logvolume01:/var/log
links:
- redis
redis:
image: redis
volumes:
logvolume01: {}
```

Installing Docker Compose

Following are the instructions to install Docker Compose in Linux Ubuntu.

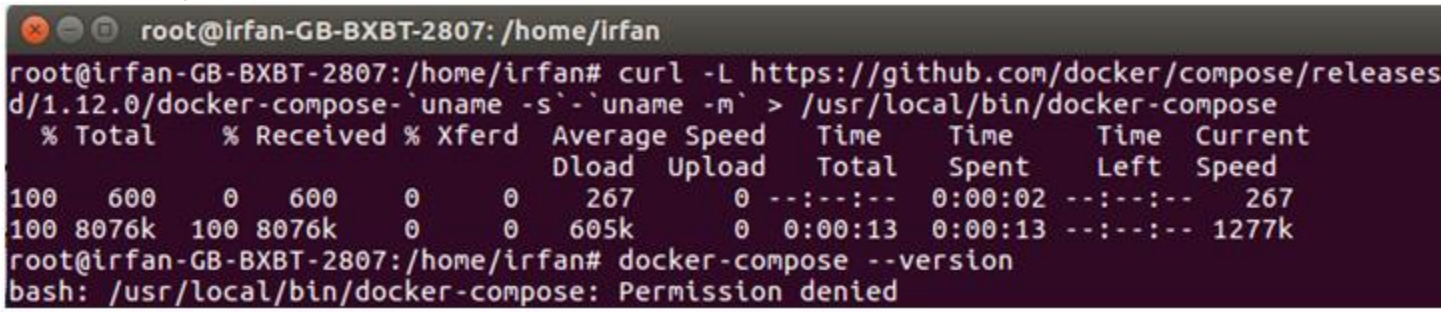
1. curl -L https://github.com/docker/compose/releases/download/1.12.0/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose



```
root@irfan-GB-BXBT-2807: /home/irfan
root@irfan-GB-BXBT-2807:/home/irfan# curl -L https://github.com/docker/compose/releases
d/1.12.0/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   600    0   600    0     0    267      0 --:--:--  0:00:02 --:--:--   267
100 8076k  100 8076k    0     0   605k      0  0:00:13  0:00:13 --:--:-- 1277k
```

Docker-compose version

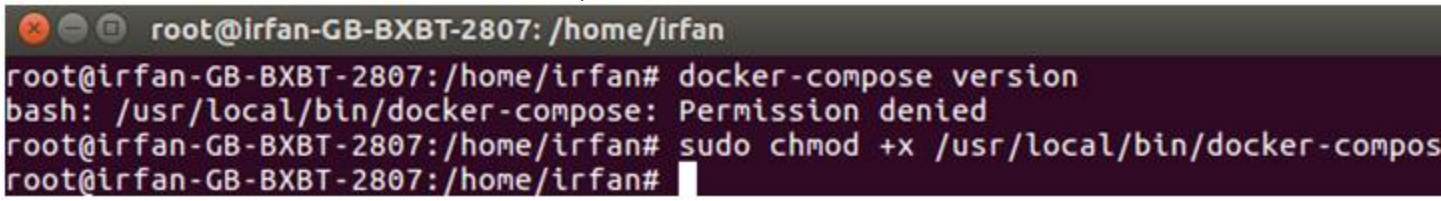
1. $ docker-compose --version



```
root@irfan-GB-BXBT-2807: /home/irfan
root@irfan-GB-BXBT-2807:/home/irfan# curl -L https://github.com/docker/compose/releases
d/1.12.0/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   600    0   600    0     0    267      0 --:--:--  0:00:02 --:--:--   267
100 8076k  100 8076k    0     0   605k      0  0:00:13  0:00:13 --:--:-- 1277k
root@irfan-GB-BXBT-2807:/home/irfan# docker-compose --version
bash: /usr/local/bin/docker-compose: Permission denied
```

It says, permission denied. So, make file executable.

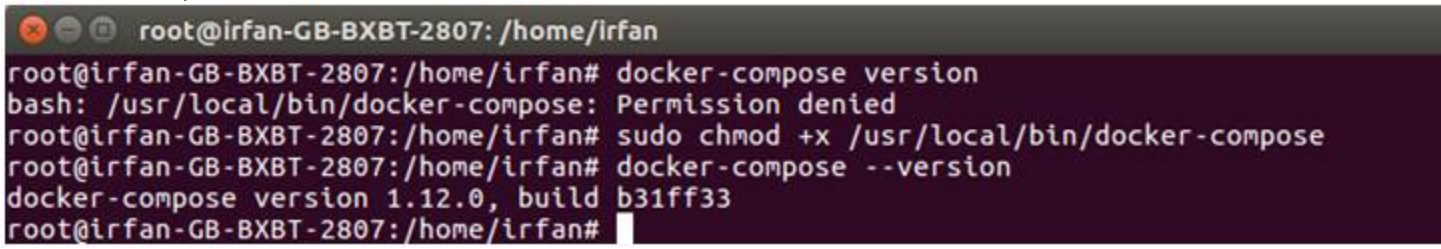
1. $ sudo chmod +x /usr/local/bin/docker-compose



```
root@irfan-GB-BXBT-2807: /home/irfan
root@irfan-GB-BXBT-2807:/home/irfan# docker-compose version
bash: /usr/local/bin/docker-compose: Permission denied
root@irfan-GB-BXBT-2807:/home/irfan# sudo chmod +x /usr/local/bin/docker-compos
root@irfan-GB-BXBT-2807:/home/irfan#
```

Now, check version again.

1. $ docker-compose ?version



```
root@irfan-GB-BXBT-2807: /home/irfan
root@irfan-GB-BXBT-2807:/home/irfan# docker-compose version
bash: /usr/local/bin/docker-compose: Permission denied
root@irfan-GB-BXBT-2807:/home/irfan# sudo chmod +x /usr/local/bin/docker-compose
root@irfan-GB-BXBT-2807:/home/irfan# docker-compose --version
docker-compose version 1.12.0, build b31ff33
root@irfan-GB-BXBT-2807:/home/irfan#
```

Running Application using Docker Compose

Example

Follow the following example

1) Create a Directory

```
$ mkdir docker-compose-example
$ cd docker-composer-example
```

2) Create a file **app.py.**

// app.py

```
from flask import Flask
from redis import Redis
app = Flask(__name__)
redis = Redis(host='redis', port=6379)
@app.route('/')
def hello():
count = redis.incr('hits')
return 'Hello World! I have been seen {} times.\n'.format(count)
if __name__ == "__main__":
app.run(host="0.0.0.0", debug=True)
```

3) Create a file **requirements.txt.**

// requirements.txt

```
flask
redis
```

4) Create a Dockerfile.

// Dockerfile

```
FROM python:3.4-alpine
ADD . /code
```

```
WORKDIR /code
RUN pip install -r requirements.txt
CMD ["python", "app.py"]
```

5) Create a Compose File.

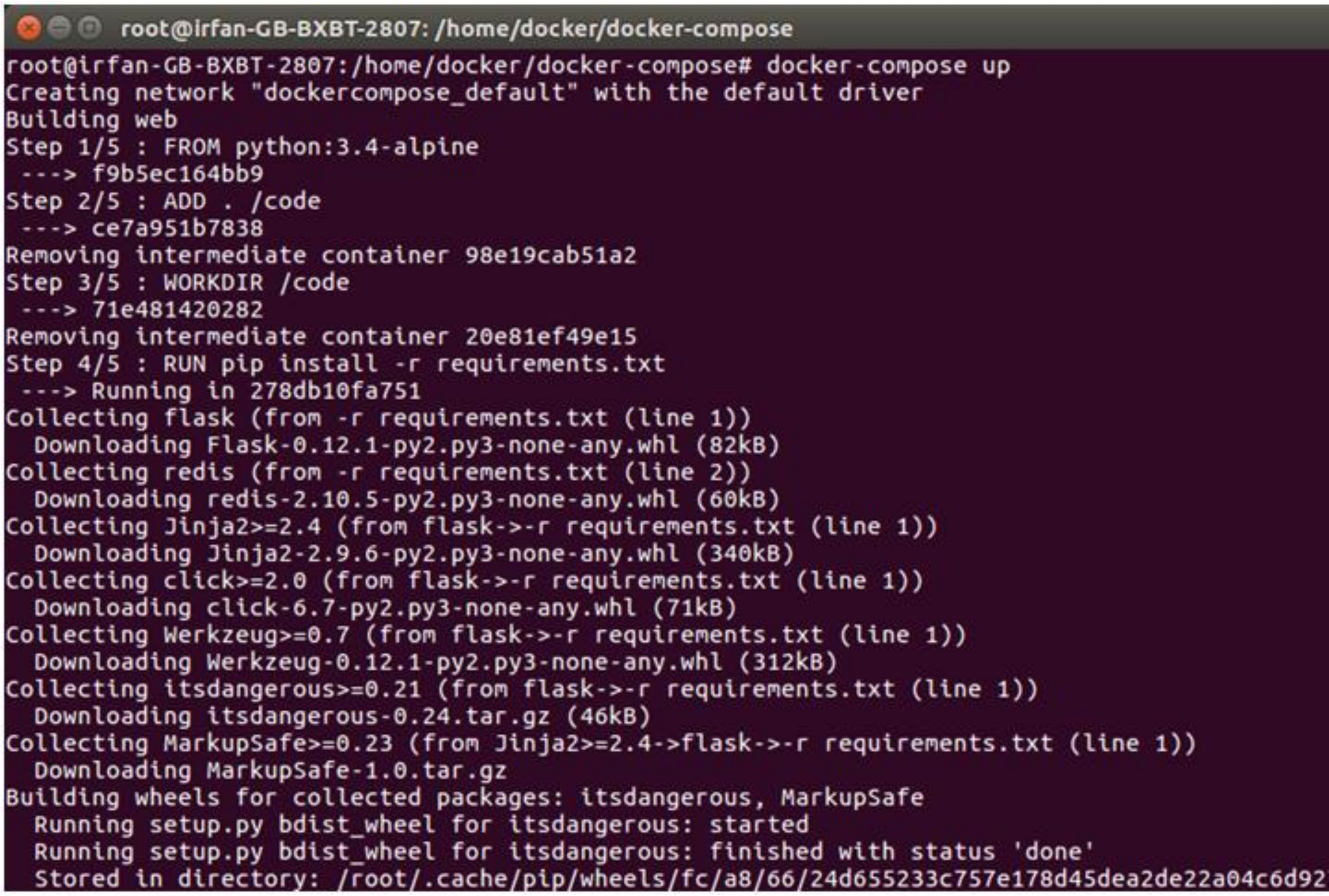
// docker-compose.yml

```
version: '2'
services:
web:
build: .
ports:
- "5000:5000"
volumes:
- .:/code
redis:
image: "redis:alpine"
```

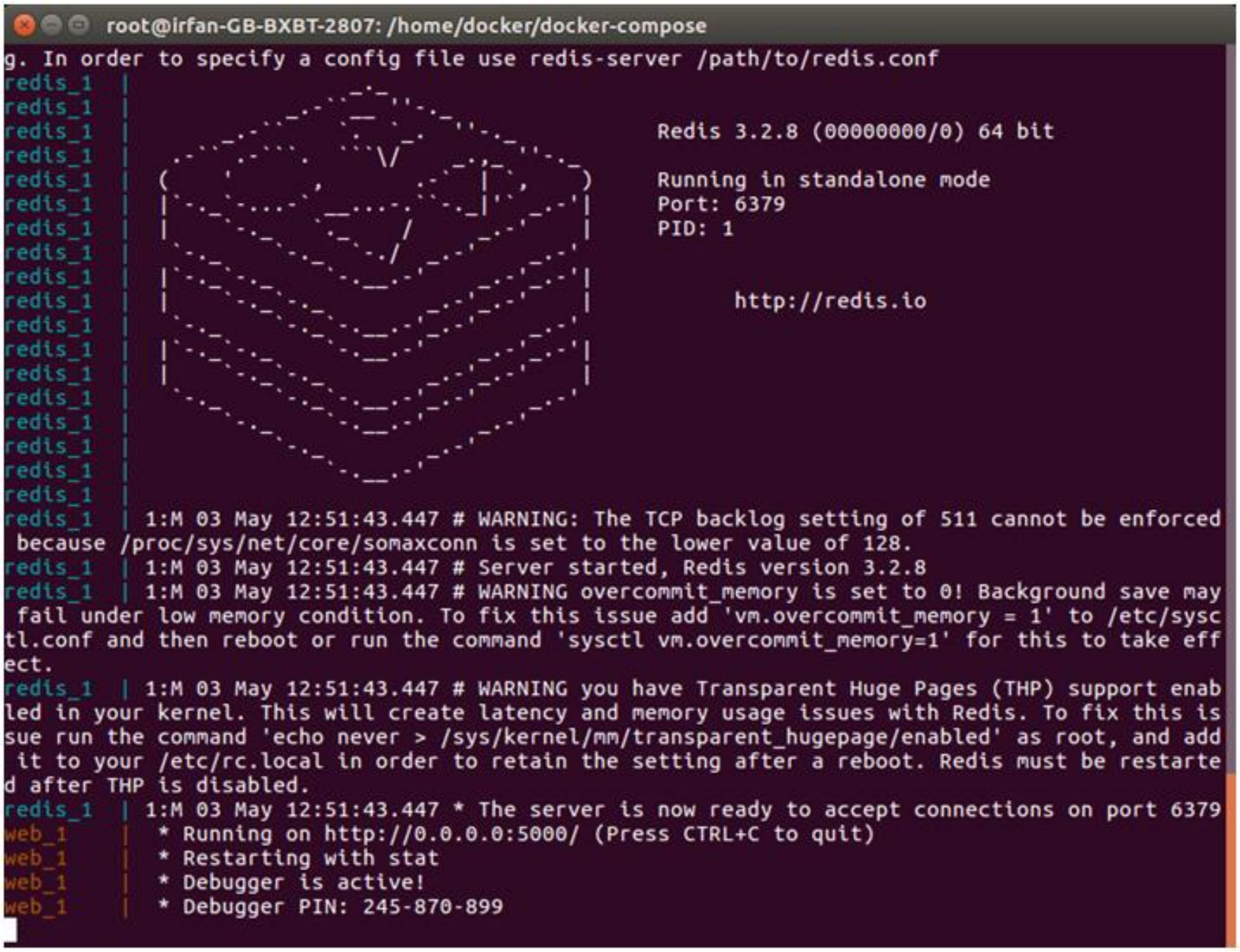
6) Build and Run Docker App with Compose

```
$ docker-compose up
```

After running the above command, it shows the following output.



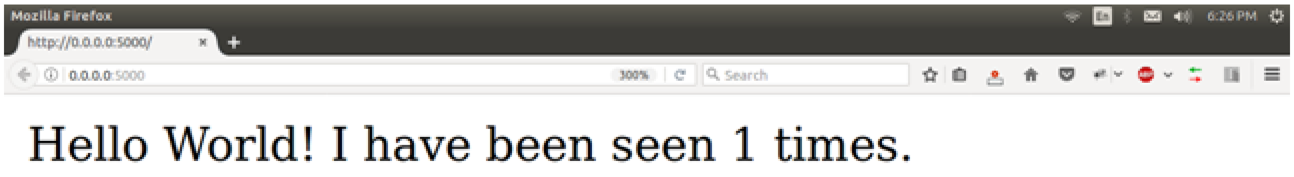
```
root@irfan-GB-BXBT-2807: /home/docker/docker-compose
root@irfan-GB-BXBT-2807:/home/docker/docker-compose# docker-compose up
Creating network "dockercompose_default" with the default driver
Building web
Step 1/5 : FROM python:3.4-alpine
 ---> f9b5ec164bb9
Step 2/5 : ADD . /code
 ---> ce7a951b7838
Removing intermediate container 98e19cab51a2
Step 3/5 : WORKDIR /code
 ---> 71e481420282
Removing intermediate container 20e81ef49e15
Step 4/5 : RUN pip install -r requirements.txt
 ---> Running in 278db10fa751
Collecting flask (from -r requirements.txt (line 1))
  Downloading Flask-0.12.1-py2.py3-none-any.whl (82kB)
Collecting redis (from -r requirements.txt (line 2))
  Downloading redis-2.10.5-py2.py3-none-any.whl (60kB)
Collecting Jinja2>=2.4 (from flask->-r requirements.txt (line 1))
  Downloading Jinja2-2.9.6-py2.py3-none-any.whl (340kB)
Collecting click>=2.0 (from flask->-r requirements.txt (line 1))
  Downloading click-6.7-py2.py3-none-any.whl (71kB)
Collecting Werkzeug>=0.7 (from flask->-r requirements.txt (line 1))
  Downloading Werkzeug-0.12.1-py2.py3-none-any.whl (312kB)
Collecting itsdangerous>=0.21 (from flask->-r requirements.txt (line 1))
  Downloading itsdangerous-0.24.tar.gz (46kB)
Collecting MarkupSafe>=0.23 (from Jinja2>=2.4->flask->-r requirements.txt (line 1))
  Downloading MarkupSafe-1.0.tar.gz
Building wheels for collected packages: itsdangerous, MarkupSafe
  Running setup.py bdist_wheel for itsdangerous: started
  Running setup.py bdist_wheel for itsdangerous: finished with status 'done'
  Stored in directory: /root/.cache/pip/wheels/fc/a8/66/24d655233c757e178d45dea2de22a04c6d92
```



```
root@irfan-GB-BXBT-2807: /home/docker/docker-compose
g. In order to specify a config file use redis-server /path/to/redis.conf
redis_1  |
redis_1  |                 _._
redis_1  |            _.-``__ ''-._
redis_1  |       _.-``    `.  `_.  ''-._           Redis 3.2.8 (00000000/0) 64 bit
redis_1  |   .-`` .-```.  ```\/    _.,_ ''-._
redis_1  |  (    '      ,       .-`  | `,    )     Running in standalone mode
redis_1  |  |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
redis_1  |  |    `-._   `._    /     _.-'    |     PID: 1
redis_1  |   `-._    `-._  `-./  _.-'    _.-'
redis_1  |  |`-._`-._    `-.__.-'    _.-'_.-'|
redis_1  |  |    `-._`-._        _.-'_.-'    |           http://redis.io
redis_1  |   `-._    `-._`-.__.-'_.-'    _.-'
redis_1  |  |`-._`-._    `-.__.-'    _.-'_.-'|
redis_1  |  |    `-._`-._        _.-'_.-'    |
redis_1  |   `-._    `-._`-.__.-'_.-'    _.-'
redis_1  |       `-._    `-.__.-'    _.-'
redis_1  |           `-._        _.-'
redis_1  |               `-.__.-'
redis_1  |
redis_1  | 1:M 03 May 12:51:43.447 # WARNING: The TCP backlog setting of 511 cannot be enforced
 because /proc/sys/net/core/somaxconn is set to the lower value of 128.
redis_1  | 1:M 03 May 12:51:43.447 # Server started, Redis version 3.2.8
redis_1  | 1:M 03 May 12:51:43.447 # WARNING overcommit_memory is set to 0! Background save may
 fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysc
tl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take eff
ect.
redis_1  | 1:M 03 May 12:51:43.447 # WARNING you have Transparent Huge Pages (THP) support enab
led in your kernel. This will create latency and memory usage issues with Redis. To fix this is
sue run the command 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add
 it to your /etc/rc.local in order to retain the setting after a reboot. Redis must be restarte
d after THP is disabled.
redis_1  | 1:M 03 May 12:51:43.447 * The server is now ready to accept connections on port 6379
web_1    |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
web_1    |  * Restarting with stat
web_1    |  * Debugger is active!
web_1    |  * Debugger PIN: 245-870-899
```

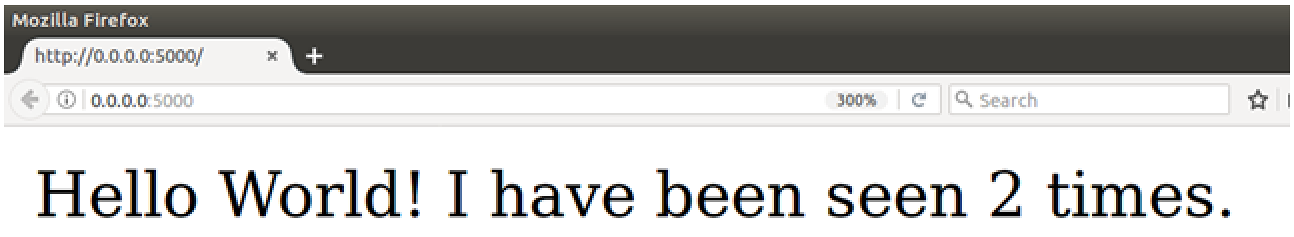
Now, we can see the output by following the running http url.

Output:



Hello World! I have been seen 1 times.

Each time, when we refresh the page. It shows counter incremented by 1.



```
Hello World! I have been seen 2 times.
```