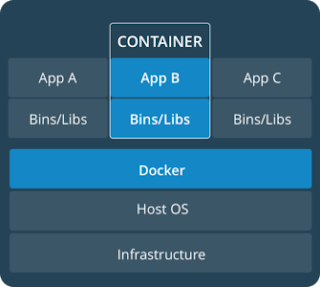
**DOCKER**

**Introduction:**

Before going to the concept of Docker, let’s discuss about containerization.

**What is Containerization?**

Containerization also called Container. Containerization is a lightweight virtualization technology alternative to hypervisor virtualization. Any application can be bundled in a container can run without any worries about dependencies, libraries and binaries. Because container creates the isolated environment with all the required dependencies, libraries and binaries to run your application without any issue. Hence we can build the packages, ship the application to any environment and run it.



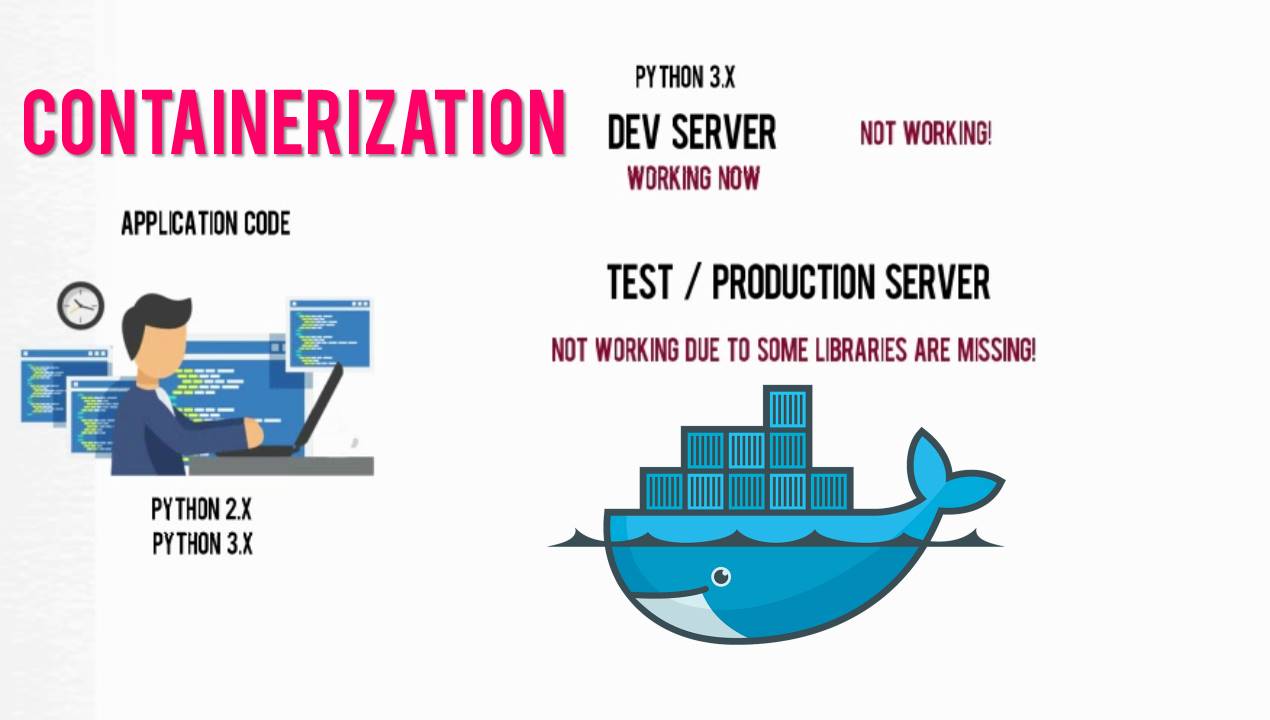
So Containers are designed to run on physical servers, virtual machines and any cloud instances.

**Why do we need containerization (Container)?**

Containers are a solution to the problem of how to run the applications efficiently when moved from one environment to another environment. This is what the biggest problem we had before containerization.  
  
Let’s say, a developer has developed the Application code in his own machine, and it was working fine in his machine. But when he moved the same code to development server, application will not work properly. Because, His own machine and development environment server configuration and versions are not similar. He developed the application code in python 2 and Development server has python 3. So again he need to work on the application code to make it work based on development server version i.e. Python 3. Now he fixed the issue and its working now on development server.  
  
This time, He is moving the application code to testing or production environment. But still not working because of some libraries are missing to run the application.

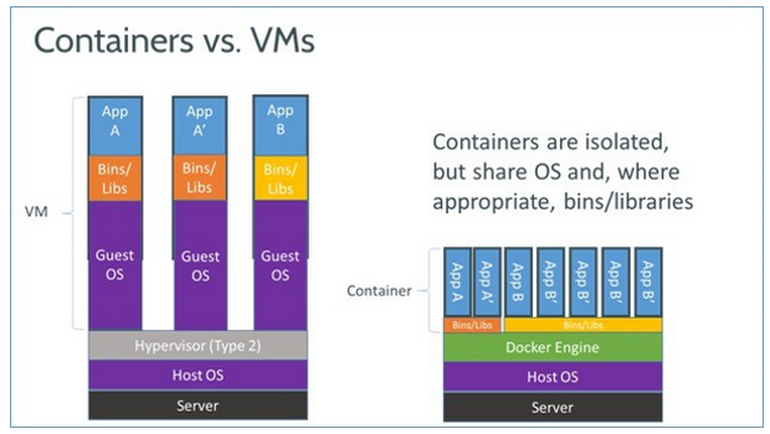
It’s a big problem. So we needed a solution to make the applications able to run anywhere, on any machine.

That is why we have a technology called containerization (Container).



**Difference of containerization and Virtualization:**

Virtualization technology that allow us to have multiple operating systems to share a single hardware processor.  
  
It is mainly used to utilize the maximum hardware resources efficiently by deploying the application into “virtual machines” (VMs) with their own operating systems, so an application can run independently on top of a server’s operating system which is a dedicated virtual machine for an application. Operation system / Guest OS lies on top of the hypervisor software’s. That might be VMware esx, Microsoft Hyper V, Kvm, Xen and so on.  
  
But Container runs on a single operating system i.e. host os, and each container shares the Host operating system kernel with the other containers. Here you would have only one operating system, on top of it you will have applications within container, so it doesn’t require an additional operating system for each application as we have in virtualization technology.  
  
So I would say Containerization is an application-specific virtualization, because it provides application with dedicated environments in the form of container to run on, which can be deployed and run anywhere without a dedicated virtual machine with Operating system for each application.  
  
Also Container was designed to solve modern problems and application management issues. So it’s not a replacement for virtualization, but it’s complementary to it.



**Advantages of Container**  
  
Containers are **isolated**, doesn’t require operating system and it shares a host kernel. So containers run on the same server and use the same resources, they do not interact with each other because it’s isolated. If one application crashes, other containers with the same application will keep running without any issues.  
  
It’s a **Portable** and **light weight** operating system and it contains only the required binaries, dependencies and libraries to run the application. So it can be move anywhere easily and can run without worrying about compatibility, dependencies kind of issues.  
  
**Faster** and **Resource Efficiency**, It’s very fast to boot, because containers are lightweight and start in less than a second since they do not require an operating system boot.  
  
Resource efficiency since containers do not require a separate operating system, they use less resources.  
  
Improves **Scalability** and **lowers costs** - By allowing more containers in the environment without the need for more servers, containerization increases scalability anywhere from 10 to 100 times that of traditional VM environments.

**What is Docker?**

Docker is an open source platform tool designed to manage the containers, which allow us to build the application in a container with required libraries, binaries, dependencies to run the application, ship the container and run anywhere.

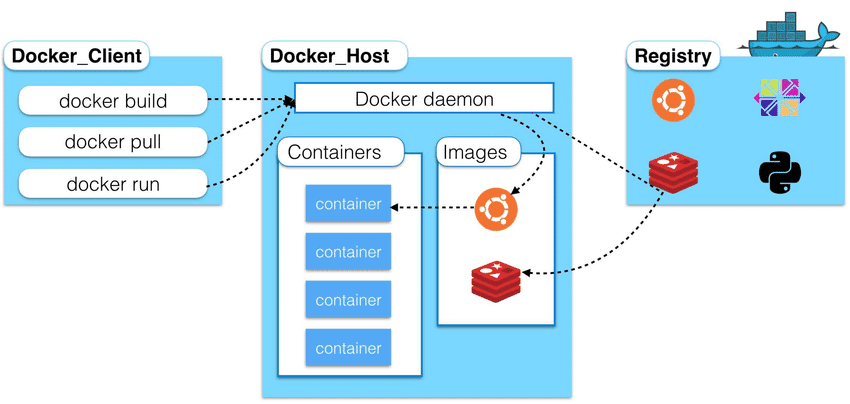
**Why do we use Docker?**  
1. *Portability*: An applications can be bundled in to a single unit and same unit can be deployed to various environments such as dev server, testing server and production server without making any changes to the container.  
2. *Light Weight*: Docker containers are pretty lightweight so it provides a smaller footprint of the operating system via containers.  
3. *Fast Delivery* and *Scalable*: Since Docker containers are pretty lightweight, so they can be deployed faster and they are very easily scalable.

4. Docker used for *Continuous Deployment* and *Testing*. So With the help of containers, it becomes easier for teams across different units, such as development, QA and Operations to work seamlessly across applications.  
5. Docker also provides you, ability to run multiple *isolated* OS on single host.  
6. *Resource Optimization:* Docker enable you to utilize the maximum resources and reduce the resource wastages of your hardware.

**What is Docker on container?**  
Docker is a tool designed to manage the containers. Using Docker you can create, deploy, and run the applications containers.

**Docker Architecture and Docker Components**

The below image shows you the Docker Architecture with Docker Components where you have docker client, docker host, docker hub, docker images, docker containers.  
  
So Docker Architecture depends on three main components:  
  
1. Docker Client  
2. Docker Host (Docker Engine)   
3. Docker Registry  
  
Also we have other components  
4. Docker Images  
5. Docker Containers  
6. Docker Network  
7. Docker Storages



**Docker Client**  
Docker client is used to manage the docker components such as containers, images, networks, storage volumes by giving the specific instructions to perform some operations like build, pull, create, run, delete, stop, restart,.  
  
So Docker client interacts with the docker host where docker daemon (dockerd) is installed, i.e. Docker Host.  
  
It’s not necessary that Docker Client must be available at the Docker host, it can also be used from the remote machine to manage your docker host. The Docker client and Docker daemon can be communicated each other using a REST API, over UNIX sockets or a network interfaces.

**Docker Host (Docker Engine)**  
Docker host is the server where the Docker Daemon is running and it manages the docker containers and other resources. It listens and accepts the requests from Docker Client.  
  
**Docker Registry - Docker Hub**Docker registry or Docker Hub is a repository to manage the Docker images. So this will allow us to push or pull the Docker images which means you can download or upload the images..  
  
There are two types of docker registry available,   
1. Docker hub - It is available in internet, Maintaining by Docker Company in cloud. We can download (pull) the publicly available images or also we can upload (push) the images to Docker Hub online. We have two services available with docker hub i.e. public and private. If you want your images to available to everyone, set it to public. If you want your images only to you, set it to private.

2. Local Docker Hub - Its available locally within the organization. By default if we pull the images, it checks the local docker hub regsitry. If not finds the requested images locally, then it goes to download from online docker hub.  
  
**Docker Images**  
Docker Images are a just templates, it is very similar to snapshot image.  
Docker images can be used to create the containers. It is a read only layer.  
Docker Images can be created from the container once you have done the changes on the container or you can create the Docker Images using a Docker file with specific instructions. So these Docker Images are very lightweight, small, and fast to deploy the containers.  
  
**Docker Containers**Docker Containers are actually a executable run-time light weight operating system.  
Docker Container is a read/write layer of a Docker Images.  
Docker containers can be created using a docker images also we can create a docker images from a docker container once the required modification is done.

**Docker Network**  
Docker network enables the network to communicate with other containers. By default, we have three networks created when you have installed docker.  
  
1*. Bridge*: This is the default network driver attached to the container during the container creation if you didnt specify other network. So any containers on the same network can be communicated with other containers.  
2. *Host*: It allows a container to directly attached to the docker host network. It can be used to have a standalone container, so this will not have isolate the network between docker host and docker containers.  
3. *None*: This can be used to disable the networking for the containers. So containers will not have ip address when you have used this network.  
  
Also we have other network drivers can be used for other features.  
  
4. *Overlay:* also ingress network, it can be used to connect different containers hosted on different docker host or many. Mostly this network automatically used in the docker swarm clustering environment.  
5. *Macvlan*: This network driver used to assign a MAC address to the container, so the container attached with a Macvlan network driver will looks like physical network attched to it. Because there are some application would work only on physical network not on virtual network. On that cases, we can use this network driver.

**Docker Storages**  
Any data written to the docker container will not be available when docker container is removed. So we must use a docker storage to have the data permanently if need.

There are three types of storage available with docker.

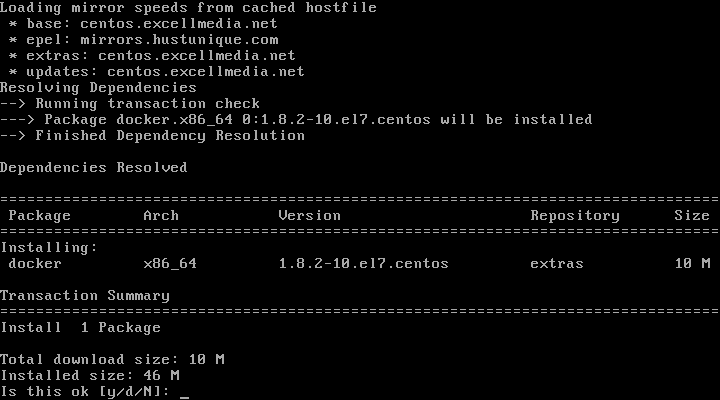
1. **Volumes** are stored in a part of the host filesystem which is managed by Docker (/var/lib/docker/volumes/ on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.
2. **Bind mounts** may be stored anywhere on the host system. They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time.
3. **tmpfs mounts**are stored in the host system’s memory only, and are never written to the host system’s filesystem.

**How to install Docker on Cent OS:**

1. Docker binaries are incorporated into RHEL/CentOS 7 extras repositories, the installation process being pretty simple. Install Docker package by issuing the following command with root privileges:

Install Docker on CentOS 7:

**#** **yum install docker**



Install Docker on CentOS 6:

To install Docker, the [Epel repositories](https://www.tecmint.com/how-to-enable-epel-repository-for-rhel-centos-6-5/) must be enabled on your system by issuing the following command:

# yum install epel-release

# yum install docker-io



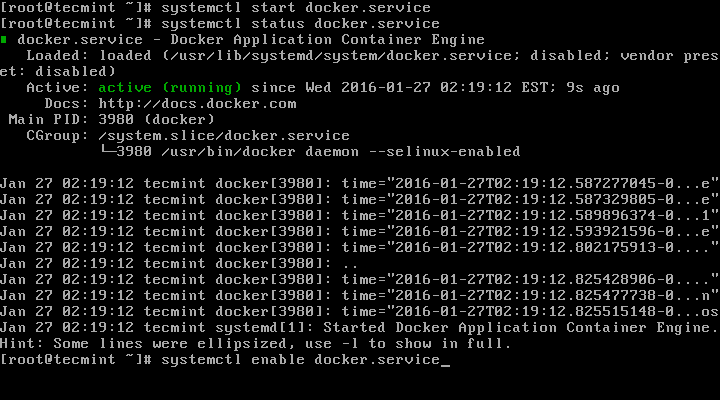
2. After, Docker package has been installed, start the daemon, check its status and enable it system wide using the below commands:

On CentOS 7:

# **systemctl start docker**

# **systemctl status docker**

# **systemctl enable docker**

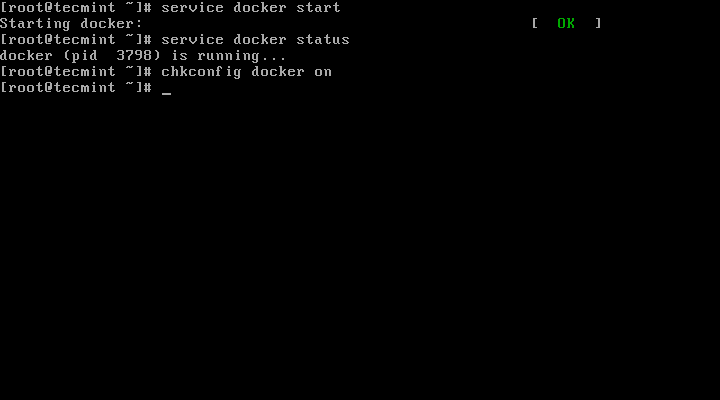


On CentOS 6:

# service docker start

# service docker status

# chkconfig docker on



3.  Finally, run a container test image to verify if Docker works properly, by issuing the following command:

# **docker run hello-world**

If you can see the below message, then everything is in the right place.

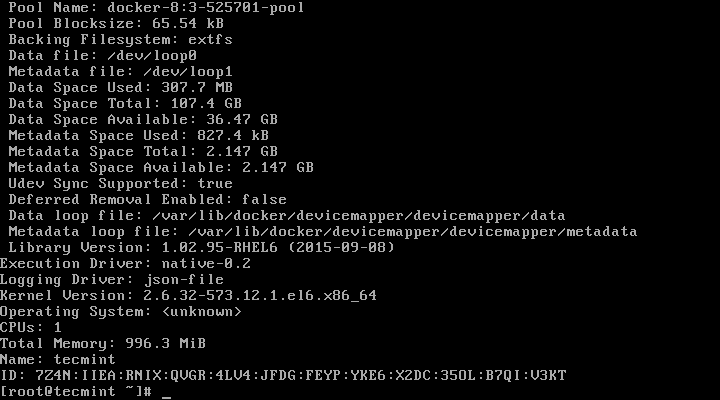
**Hello from Docker!**

**This message shows that your installation appears to be working correctly.**

4. Now, you can run a few basic Docker commands to get some info about Docker:

For system-wide information on docker:

# **docker info**



For Docker version:

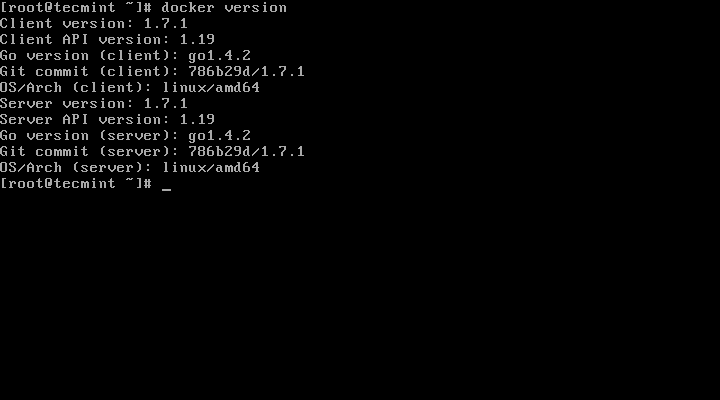
Use below docker command with "-v" option to know the docker package version with build release information.

[root@localhost /]# **docker -v**

Docker version 1.13.1, build dded712/1.13.1

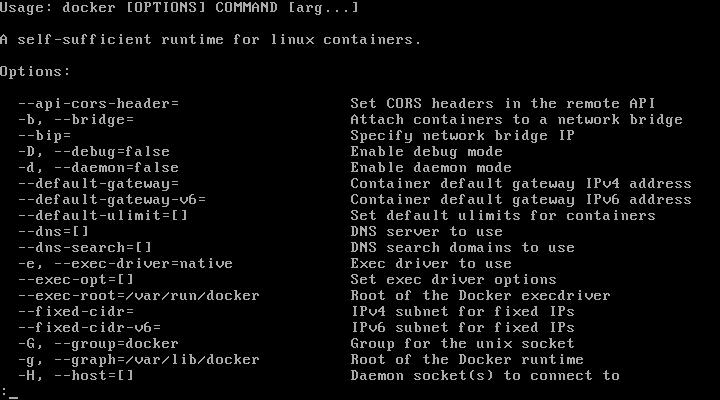
Also use "version" argument along with docker command to know the relevant information about the docker package.

[root@localhost /]# **docker -version**



5. To get list of all available docker commands on the console:

# **docker**



**What is a snapshot image?**

A snapshot image is a logical, read-only copy of volume content, captured at a particular point in time. You can use snapshots to protect against data loss.

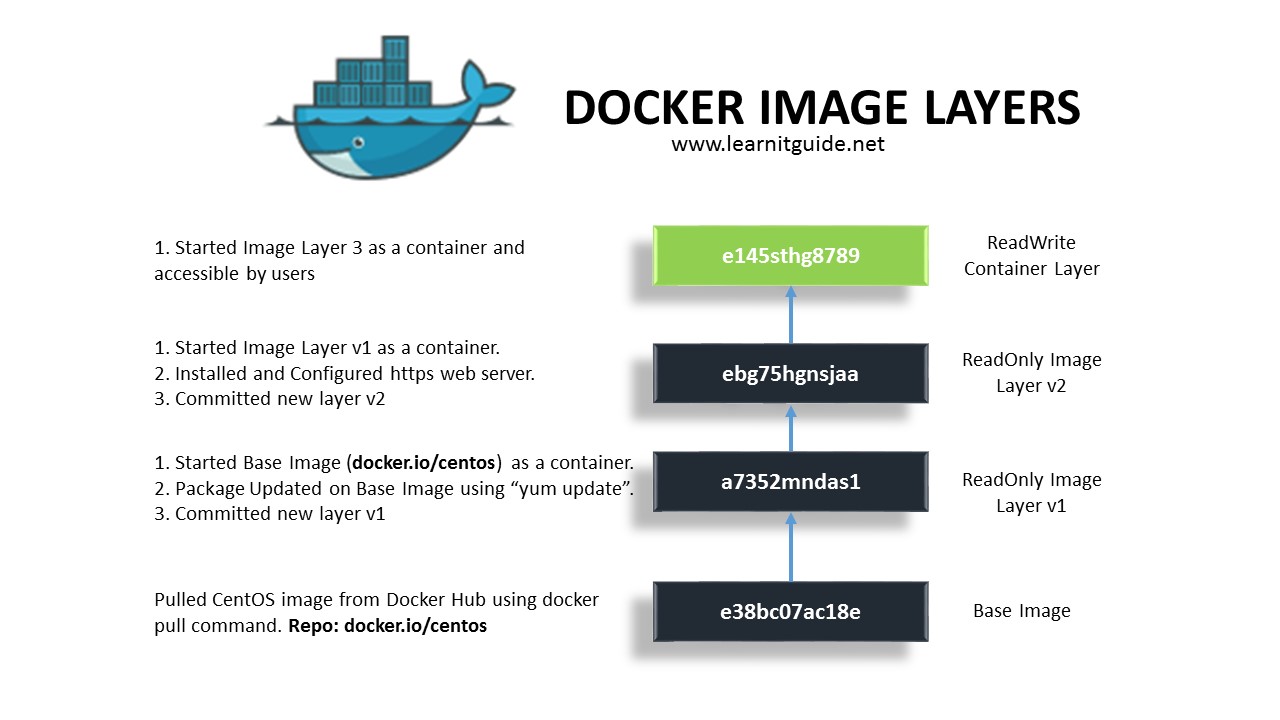
Snapshot images also are useful for test environments. By creating a virtual copy of data, you can test data using the snapshot without altering the actual volume itself. In addition, hosts do not have write access to snapshot images, so your snapshots are always a secure backup resource.

(The Volumes page is where you create the storage containers for your applications, databases, and file systems. A volume is the logical component created for the host to access storage on the array.)

**What is a docker image?**

Docker images are just a templates of a Docker Containers and it is very similar to snapshot image with smaller in size. Singe Docker Image can be used to create multiple containers for different environment like development, UAT and Production.

Docker Images are very lightweight, small, and fast to deploy the containers.



Docker Images are consists of many layers with unique Image ID (e.g. : e34fs4553) from Base Images. Each Layers may have some changes committed on top of a existing layers.  
  
Docker Images are read only layer of Docker Containers and Docker Containers are read write layer of Docker Images.

**Images:**

The file system and configuration of our application which are used to create containers.

**Containers:**

Running instances of Docker images - containers run the actual applications.

A container includes an application and all of its dependencies.

It shares the kernel with other containers, and runs as an isolated process in user space on the host OS.

**Docker daemon:**

The background service running on the host that manages building, running and distributing Docker containers.

**Docker client:**

The command line tool that allows the user to interact with the Docker daemon.

**Docker Store:**

Store is, among other things, a registry of Docker images. You can think of the registry as a directory of all available Docker images.

For example,

Image is like a file. Image usually contains software like os, MySQL, notepad , jdk etc and data like .Java, .class, .txt etc . By using image we create container which is a place/box where we can perform action (run Java code, maintain database, ui code) by making use softwares, data present in the image.

**How to Create or Customize a Docker Images?**  
Docker Images can be created from the Container itself, when some changes are done on the container or you can create the [Docker Images using a Dockerfile with specific instructions](http://www.learnitguide.net/2018/06/write-dockerfile-to-build-own-images.html). But remember, we must need an existing docker images for both the cases, either to create a container or to customize a new docker images. Let’s see how to get these docker existing base images.

**How to get Docker Images?**  
Docker Images are available in Docker Registry hub (<https://hub.docker.com/>) over the internet, it is a repository to manage the docker images. So Docker Images can be pulled (download) and pushed (Upload) to the repository at any time.  
  
But Account registration is required for uploading your Docker images not for downloading. Account Registration is free and pretty straight forward.

**How to pull / download docker images?**  
Docker is the tool used to manage the entire docker environment.

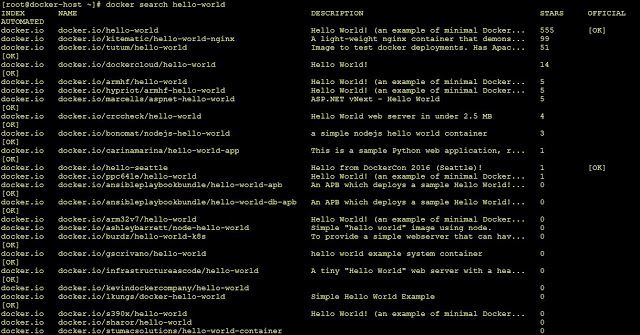
Let’s take an example that, I want "Hello-world"

We must know the exact repository name to pull corresponding docker images. So use "docker search" command to find the correct repository name to download the particular docker images.

For Hello-World:

[root@docker-host ~]# docker search hello-world

Above commands will give the list of available docker images from docker hub registry with name, description, number of stars awarded, whether the image is official, and whether it is automated as shown in the below image.



Get repository path from the name column to download the particular docker images. Similarly you can try for other docker images too.  
  
For example,  
Hello-World has "docker.io/hello-world"  
centos has "docker.io/centos"

Use "docker pull" command to download the particular docker images. By default, it will check the local docker hub regsitry. If it didnt find the requested images locally, then it goes to download from online docker hub as below. This will download each layer of docker images.

For Hello-World Docker Image:

[root@docker-host ~]# docker pull docker.io/hello-world  
Using default tag: latest  
Trying to pull repository docker.io/library/hello-world ...  
latest: Pulling from docker.io/library/hello-world  
Digest: sha256:f5233545e43561214ca4891fd1157e1c3c563316ed8e237750d59bde73361e77  
Status: Image is up to date for docker.io/hello-world:latest  
[root@docker-host ~]#

**How to list available docker images?**  
Use "docker images" command to know the list of docker images available in your local docker host.

[root@docker-host ~]# docker images  
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE  
docker.io/centos        latest              49f7960eb7e4        2 weeks ago         200 MB  
docker.io/hello-world    latest              e38bc07ac18e        2 months ago        1.85 kB  
[root@docker-host ~]#

Above "docker images" command displays the list of docker images with following information.  
  
Repository - Name of the Repository  
Tag - Version of the Docker Images  
Image ID - Unique Docker Images ID to manage the Docker Images  
Created - When Docker Images is created  
Size - Size of the Docker Images

[root@localhost /]# **docker images**

REPOSITORY TAG IMAGE ID CREATED SIZE

docker.io/hello-world latest 2cb0d9787c4d 7 weeks ago

**How to know detailed information about a Docker Images?**  
Use "docker inspect" command to know detailed information about a Docker Images available locally using Docker Image ID.  
This will give lot of information about Docker Images as below.

[root@localhost /]# **docker inspect 2cb0d9787c4d**

[

{

"Id": "sha256:2cb0d9787c4dd17ef9eb03e512923bc4db10add190d3f84af63b744e353a9b34",

"RepoTags": [

"docker.io/hello-world:latest"

],

"RepoDigests": [

"docker.io/hello-world@sha256:4b8ff392a12ed9ea17784bd3c9a8b1fa3299cac44aca35a85c90c5e3c7afacdc"

],

"Parent": "",

"Comment": "",

"Created": "2018-07-11T00:32:08.432822465Z",

"Container": "6b6326f6afc81f7850b74670aad2bf550c7f2f07cd63282160e5eb564876087f",

"ContainerConfig": {

"Hostname": "6b6326f6afc8",

"Domainname": "",

"User": "",

"AttachStdin": false,

"AttachStdout": false,

"AttachStderr": false,

"Tty": false,

"OpenStdin": false,

"StdinOnce": false,

"Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"

],

"Cmd": [

"/bin/sh",

"-c",

"#(nop) ",

"CMD [\"/hello\"]"

],

"ArgsEscaped": true,

"Image": "sha256:6bc48d210ad4c6bbb74e02e6196a9133b57107033c09e92cac12616cad30ebcf",

"Volumes": null,

"WorkingDir": "",

"Entrypoint": null,

"OnBuild": null,

"Labels": {}

},

"DockerVersion": "17.06.2-ce",

"Author": "",

"Config": {

"Hostname": "",

"Domainname": "",

"User": "",

"AttachStdin": false,

"AttachStdout": false,

"AttachStderr": false,

"Tty": false,

"OpenStdin": false,

"StdinOnce": false,

"Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"

],

"Cmd": [

"/hello"

],

"ArgsEscaped": true,

"Image": "sha256:6bc48d210ad4c6bbb74e02e6196a9133b57107033c09e92cac12616cad30ebcf",

"Volumes": null,

"WorkingDir": "",

"Entrypoint": null,

"OnBuild": null,

"Labels": null

},

"Architecture": "amd64",

"Os": "linux",

"Size": 1848,

"VirtualSize": 1848,

"GraphDriver": {

"Name": "devicemapper",

"Data": {

"DeviceId": "2",

"DeviceName": "docker-8:3-269536477-3d406d917b8bbaee1799f1af0481f735306151438a12dc2dd127ceed8545746f",

"DeviceSize": "10737418240"

}

},

"RootFS": {

"Type": "layers",

"Layers": [

"sha256:ee83fc5847cb872324b8a1f5dbfd754255367f4280122b4e2d5aee17818e31f5"

]

}

}

]

**How to get history of a Docker Images?**  
Use "docker history" command along with Docker Image ID to get a history of a Docker Images.

[root@localhost /]# **docker history 2cb0d9787c4d**

**IMAGE CREATED CREATED BY SIZE COMMENT**

2cb0d9787c4d 7 weeks ago /bin/sh -c #(nop) CMD ["/hello"] 0 B

<missing> 7 weeks ago /bin/sh -c #(nop) COPY file:3c3ca82dfdb40d... 1.85 kB

where **2cb0d9787c4d** is the docker id of “hello-world” docker image.

**Scenario - 1: *How to Save or Backup a Docker Image?***Use "docker save" command along with Docker Image ID to save or backup Docker Images in archive format. It can be used to restore the Docker Images in any Docker Host, when it is required.

[root@localhost /]# **docker save 2cb0d9787c4d > backup\_hello-world.tar**

List the folder to confirm the docker Image ID is saved or not.

[root@localhost /]# **ll**

total 72

-rw-r--r--. 1 root root 11776 Aug 29 06:08 backup\_hello-world.tar

lrwxrwxrwx. 1 root root 7 Dec 12 2017 bin -> usr/bin

dr-xr-xr-x. 4 root root 4096 Mar 5 02:54 boot

drwxr-xr-x. 2 manasa manasa 6 Mar 26 05:57 data

drwxr-xr-x. 19 root root 3280 Aug 29 06:08 dev

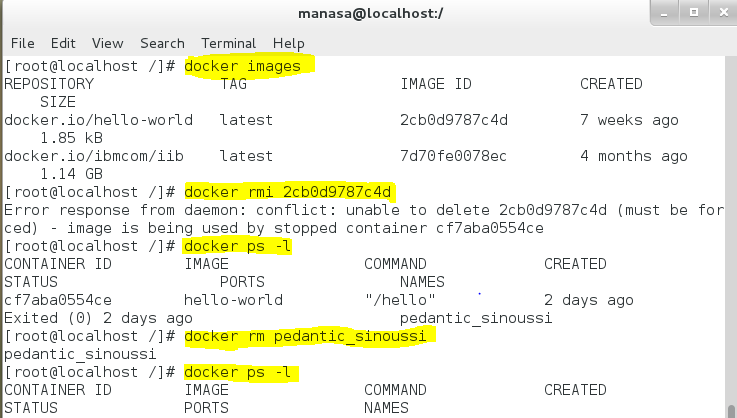
Or can check in this way,

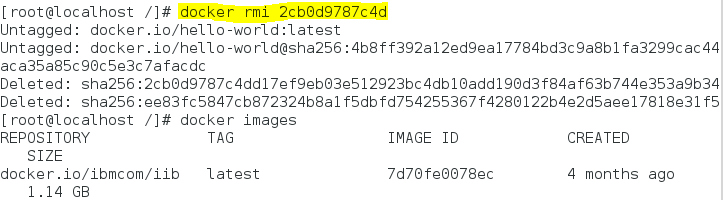
[root@localhost /]# **ll backup\_hello-world.tar**

-rw-r--r--. 1 root root 11776 Aug 29 06:08 backup\_hello-world.tar

**How to remove or delete Docker Images?**  
Use "docker images" to find the Docker Image ID or Image Name and use "docker rmi" command to remove or delete docker images.

Removed the docker image “hello-world”

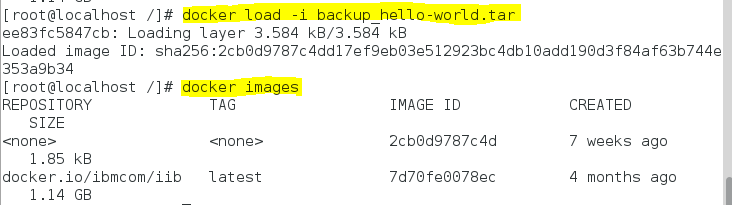




To remove multiple docker images, mention the docker images ID separated by spaces as below.

[root@localhost /]# docker rmi 49f7960eb7e4 47fjay738290 987tr78rt5g5

Once removed image can be restored if backup is taken,



To remove all docker images in a single command,

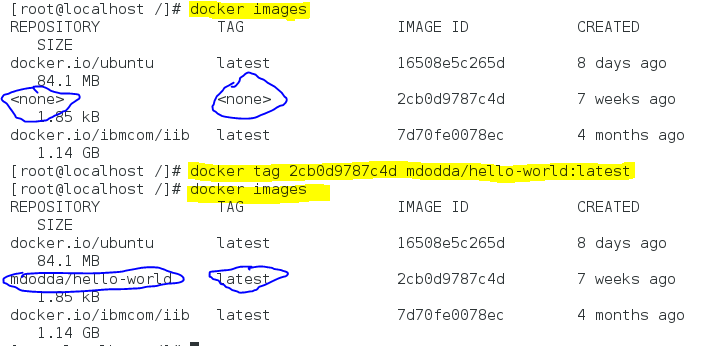
[root@localhost /]# docker rmi $(docker images -a -q)

**Scenario - 2: *Adding tag name to the restored docker image***

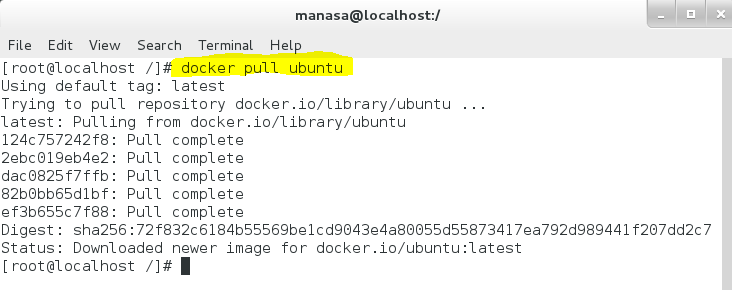
1. Pulled docker image hello-world
2. Took back up of the image
3. Now, removed image and again reloaded the image
4. Adding tag to the uploaded image as it is reloaded without any tag name and repo name.

 Tag the ‘**hello-world**’ image using the ‘**docker tag**’ command and the **image ID**.

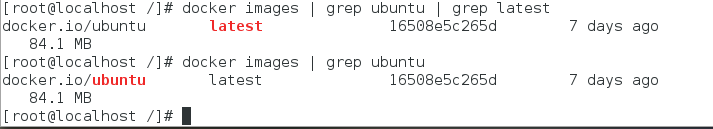
# **docker tag <image-id of hello-world> <your docker hub username>/hello-world:latest**



**Pull an ubuntu image from docker hub:**



To search for the image you want from the list of images use **grep** command.



**How to create docker container?**

Containers is a runnable instance of an image

It is a standardized unit of a software

Docker containers are created from docker image

To run a container:

# **docker run image\_name**

# **docker run ubuntu**

To see running containers

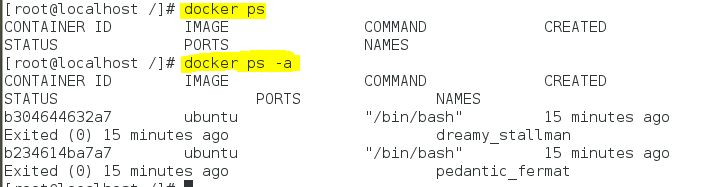
# **docker ps :** shows running containers

# **docker ps –a :** shows both the running and stopped containers.

When we use the command # **docker run image\_name** command, the newly created container ran and exited/terminated

# **docker ps** displays nothing(container exited), So use # **docker ps -a** to see all the past containers too.

Use # **docker ps -l** to look most recently ran container



To make the docker containers running in the background as a daemon

--------------------need to work with--------------------------

docker run –ti <image-name> -> t : tty, i : stdin

docker run –dti <image-name> -> d : docker

-------------------------------------------------------------------------

**What is a Dockerfile?**

* A Dockerfile is a text configuration file written in a popular, human-readable Markup Language called YAML.
* It is a step-by-step script of all the commands you need to run to assemble a Docker Image.
* The docker build command processes this file generating a Docker Image in your Local Image Cache, which you can then start-up using the docker run command, or push to a permanent Image Repository.

Dockerfile is a file used to build our own docker images by giving some instructions to customize an existing docker images based on our requirement in an automated way without running a docker container.

**Why Dockerfile is required?**

We have lot of pre-existing docker images available in docker hub registry. It can be pulled directly and we can use it for our need.

But still we may need to customize the existing docker images based on our requirement. So dockerfile helps us to build our own docker images automatically as per the instructions given.

**How to write a dockerfile?**

Use any editor to create a dockerfile with filenames either "dockerfile" or "Dockerfile" and add the instruction based on the requirement. Check the docker instructions available for writing a dockerfile and examples below.

**Create a Dockerfile**

Creating a Dockerfile is as easy as creating a new file named “Dockerfile” with your text editor of choice and defining some instructions.

# **Dockerfile Instructions Explained with Examples**

Syntax to write instruction and its arguments within a dockerfile is,

# Comment  
INSTRUCTION arguments

Instructions can be given in lowercase or uppercase letters. But to differentiate from the instructions and arguments, we use uppercase letters.

**Dockerfile example:**

FROM docker.io/centos  
MAINTAINER Devops Engineer  
RUN yum update && yum -y install httpd  
RUN mkdir -p /data/myscript  
WORKDIR /data/myscript  
CMD python app.py

We have listed all dockerfile instructions and its functions with examples below,

**FROM**  
FROM instruction used to specify the valid docker image name. So specified Docker Image will be downloaded from docker hub registry if it is not exists locally.  
  
Examples:

FROM docker.io/centos:latest  
FROM docker.io/centos:6

If tag "6" is not specfied, FROM instruction will use the latest tag (version).  
This is a mandatory instruction in dockerfile, rest all are optional and those can be used based on the requirement.  
  
**MAINTAINER**  
MAINTAINER instruction is used to specify about the author who creates this new docker image for the support.  
  
Example:

MAINTAINER Administrator

**LABEL**  
LABEL instruction is used to specify metadata informations to an image. A LABEL is a key-value pair.  
  
Example:

LABEL "Application\_Environment"="Development"

**EXPOSE**  
EXPOSE instruction is used to inform about the network ports that the container listens on runtime. Docker uses this information to interconnect containers using links and to set up port redirection on docker host system.  
  
Examples:

EXPOSE 80 443  
EXPOSE 80/tcp 8080/udp

**ADD**  
ADD instruction is used to copy files, directories and remote URL files to the destination (docker container) within the filesystem of the Docker Images. Add instruction also has two forms - Shell Form and Executable Form.  
  
Examples:   
Shell Form - ADD src dest

ADD /root/testfile /data/

Executable Form - ADD ["src","dest"]

ADD /root/testfile /data/

If the "src" argument is a compressed file (tar, gzip, bzip2, etc) then it will extract at the specified "dest" in the container's filesystem.  
  
**COPY**  
COPY instruction is used to copy files, directories and remote URL files to the destination within the filesystem of the Docker Images. COPY instruction also has two forms - Shell Form and Executable Form.

Examples:   
Shell Form

COPY src dest  
COPY /root/testfile /data/

Executable Form

COPY ["src","dest"]  
COPY /root/testfile /data/

If the "src" argument is a compressed file (tar, gzip, bzip2, etc), then it will copy exactly as a compressed file and will not extract.

**RUN**  
RUN instruction is used to execute any commands on top of the current image and this will create a new layer. RUN instruction has two forms - Shell Form and Executable Form.  
  
Examples:   
Shell form:

RUN yum update  
RUN systemctl start httpd

Executable form:

RUN ["yum","update"]  
RUN ["systemctl","start","httpd"]

**CMD**  
CMD instruction is used to set a command to be executed when running a container. There must be only one CMD in a Dockerfile. If more than one CMD is listed, only the last CMD takes effect.  
CMD instruction has two forms - Shell Form and Executable Form.  
  
Example:   
Shell form:

CMD ping google.com  
CMD python myapplication.py

Executable form:

CMD ["ping","google.com"]  
CMD ["python","myapplication.py"]

**VOLUME**  
VOLUME instruction is used to create or mount a volume to the docker container from the docker host filesystem.  
  
Examples:

VOLUME /data  
VOLUME /appdata:/appdata

**USER**  
USER instruction is used to set the username,group name, UID and GID for running subsequent commands. Else root user will be used.  
  
Examples:

USER webadmin  
USER webadmin:webgroup  
USER 1008  
USER 1008:1200

**WORKDIR**  
WORKDIR instruction is used to set the working directory.  
  
Examples:

WORKDIR /app/  
WORKDIR /java\_dst/

**ENV**  
ENV instruction is used to set environment variables with key and value. Lets say, we want to set variables APP\_DIR and app\_version with the values /data and 2.0 respectively. These variables will be set during the image build also available after the container launched.  
  
Examples:

ENV APP\_DIR /data/  
ENV app\_version 2.0

**ARG**  
ARG instruction is also used to set environment variables with key and value, but this variables will set only during the image build not on the container.  
  
Examples:

ARG TMP\_NAME mycustom\_image  
ARG TMP\_VER 2.0

**ONBUILD**  
ONBUILD instruction is used to specify a command that runs when the image in the Dockerfile is used as a base image for another image.  
  
Examples:

ONBUILD ADD . /app/data  
ONBUILD RUN yum install httpd

## **Simple Dockerfile for NGINX:**

#

# Each instruction in this file generates a new layer that gets pushed to your local image cache

#

#

# Lines preceeded by # are regarded as comments and ignored

#

#

# The line below states we will base our new image on the Latest Official Ubuntu

FROM ubuntu:latest

#

# Identify the maintainer of an image

MAINTAINER My Name "myname@somecompany.com"

#

# Update the image to the latest packages

RUN apt-get update && apt-get upgrade -y

#

# Install NGINX to test.

RUN apt-get install nginx -y

#

# Expose port 80

EXPOSE 80

#

# Last is the actual command to start up NGINX within our Container

CMD ["nginx", "-g", "daemon off;"]

EXPLAINATION:

**FROM** – Select the base image to build the new image on top of

FROM ubuntu:latest

**MAINTAINER** – Optional field to let you identify yourself as the maintainer of this image

MAINTAINER Some One "someone@xyz.xyz"

**RUN** – Specify commands to make changes to your Image and subsequently the Containers started from this Image. This includes updating packages, installing software, adding users, creating an initial database, setting up certificates, etc. These are the commands you would run at the command line to install and configure your application

RUN apt-get update && apt-get upgrade -y && apt-get install -y nginx && rm-rf /var/lib/apt/lists/\*

**EXPOSE** – Define which Container ports to expose

EXPOSE 80

**CMD** – This is the command that will run when the Container starts

CMD ["nginx", "-g", "daemon off;"]

**Scenario – 3:** ***To create a docker image using dockerfile.***

**Step-1:** Create a Dockerfile or create a dockerfile under /root directory

[root@localhost /]# pwd

/

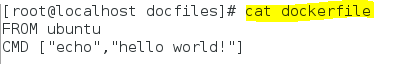
[root@localhost /]# mkdir docfiles

[root@localhost /]# cd docfiles

[root@localhost docfiles]# touch dockerfile

**Step-2:** Define the instructions to customize the existing docker image as per our requirement

Edited the created “dockerfile” and specify some instructions to print “hello world!” message.

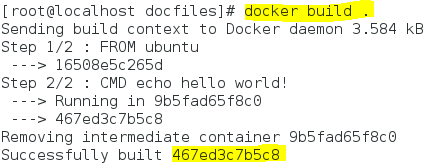


**Explanation:**

Used,

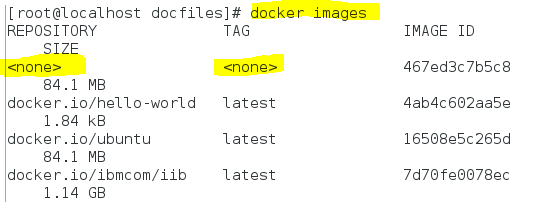
* FROM instruction to specify the image
* CMD sets default command and/or parameters, which can be overwritten from command line when docker container runs.

**Step-3:** Build a docker image from the dockerfile

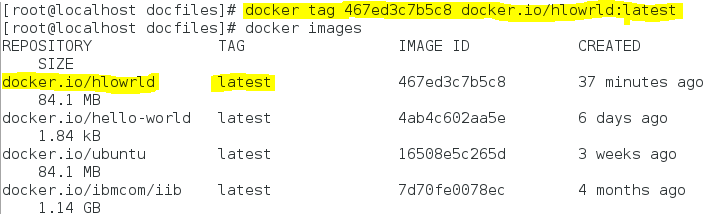


Use “docker build .” command to build the docker image if dockerfile exists in current directory. If dockerfile exists in different directory, use “docker build /path/dir”

New docker image is built with image id “467ed3c7b5c8”



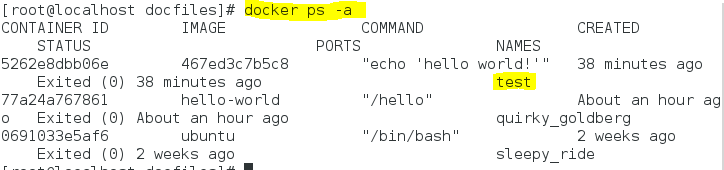
Tag the newly created image now using “docker tag” command.



**Step-4:** Run and test the customized docker image by launching docker container.

Use “docker run” command to run the docker container using the docker image created newly.



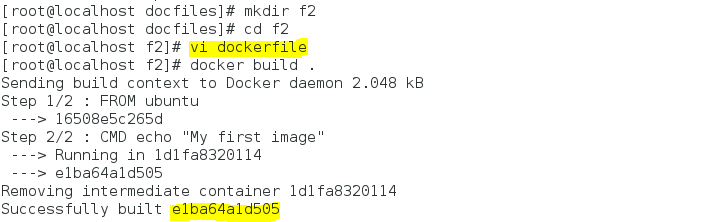


--name: to specify a unique container name

**Scenario – 4: *Create a new image and push into the docker hub.***

1. Created a new directory and a new dockerfile in it.

2. Build the new dockerfile using docker build command



3. Add a tag to the new image



4. Ran and tested my customized docker image by launching a docker container:

[root@localhost f2]# docker run --name sandbox e1ba64a1d505

My first image

[root@localhost f2]# docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

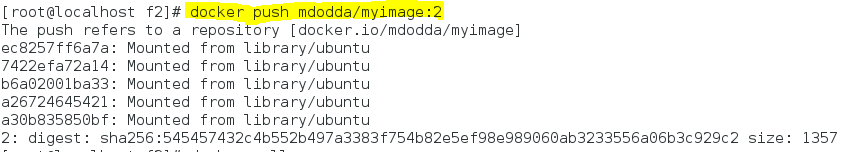
26d1d793d8e3 e1ba64a1d505 "/bin/sh -c 'echo ..." 21 seconds ago Exited (0) 7 seconds ago sandbox

5262e8dbb06e 467ed3c7b5c8 "echo 'hello world!'" 2 hours ago Exited (0) 2 hours ago test

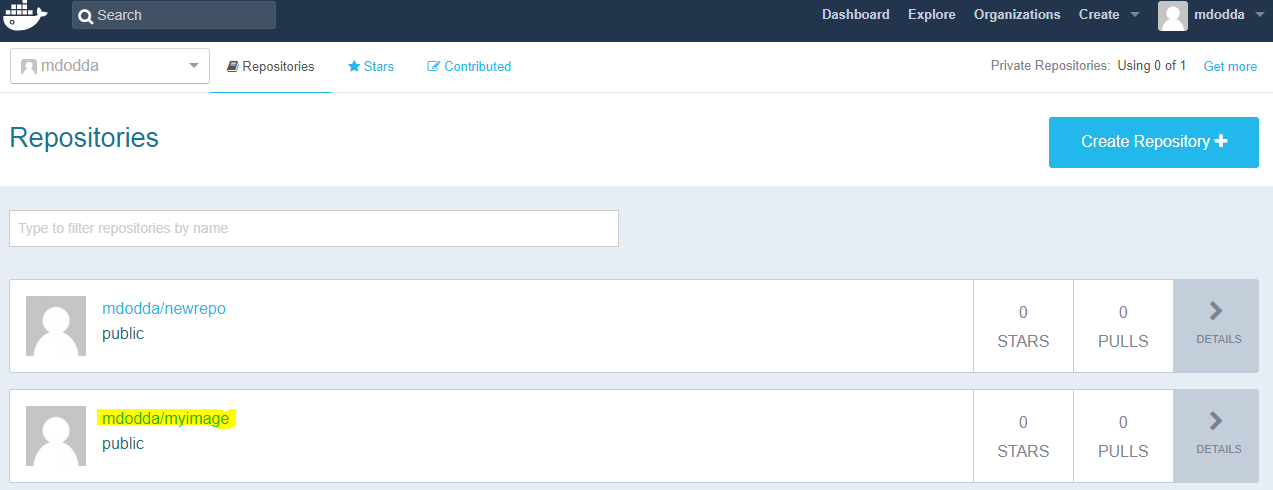
77a24a767861 hello-world "/hello" 3 hours ago Exited (0) 3 hours ago quirky\_goldberg

0691033e5af6 ubuntu "/bin/bash" 2 weeks ago Exited (0) 2 weeks ago sleepy\_ride

5. Push the image to the docker hub. Before pushing the image to the hub, provide login credentials of the hub.



6. Login to the docker hub and check whether the image is pushed or not.



**Issue#** Another application is holding yum lock........

Open a terminal.  
Become superuser by entering 'su' and then the password.  
Then,  
# **cd /var/run/**# dir   
Look for the 'yum.pid' file.  
# **rm -f yum.pid**# dir  
Confirm that the 'yum.pid' file has been deleted.  
# **yum update**This should re-start your update process.

**Issue#** E: Unable to locate package nginx while building a dockerfile.

For the above issue, check the argument of FROM instruction, whether the correct image is mentioned or not.

# **Scenario - 5: *Steps to Build Apache Web Server Docker Image using a dockerfile***

**DOCKER**

Docker is an open-source lightweight virtualization tool. It is containerizing platform in which user can run and deploy application and its dependencies and form containers to run over any linux infrastructure.

**APACHE WEB SERVER**

Apache web server is popular open source http web server tool which is widely used for deployment of webpages. It can be installed in any operating system.

**Advantages of using Docker:**

1) Application that runs on docker require lesser size.

2) Better utilization CPU.

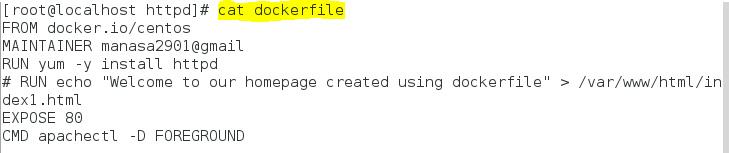
3) Short boot-up process.

4) Docker can be integrated with other tools.

To create apache webserver image using DOCKER FILE, work with the following steps:

**Step - 1:** Created a new directory with any name like I did “httpd”

**Step - 2:** Create a new file in the httpd directory namely “dockerfile”



“#” represents comment

**Explanation of the docker instructions used in the above dockerfile:**

**FROM:** To specify the image.

**MAINTAINER:**  This command is used to give the information about the author or manager who is managing this image.

**RUN****:**  Before building an image if want some configuration that needs to be present in the image. Inside the image we need to install Apache web server image the command to install that image is

**COPY:** This command is used to copy a file from host os to docker container

**EXPOSE**: This command is used to specify the port number in which the container is running its process. Anybody can come from outside and connect to this port. Apache webserver is launched at port 80 by default that is why we need to expose container at port 80.

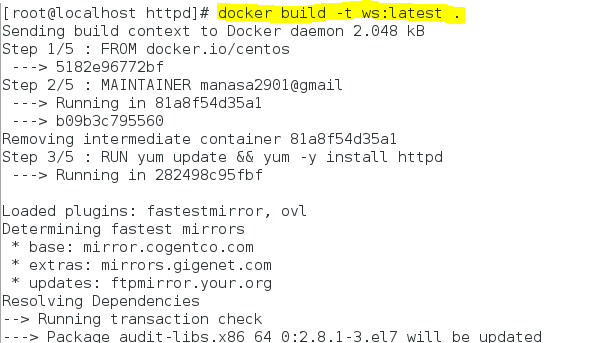
**CMD**: To run a command as soon as container is launched. CMD command is different from RUN because RUN is used at the time of building an image and CMD used to run command when container is started.

**-DFOREGROUND :** This is not a docker command this is http server argument which is used to run webserver in background. If we do not use this argument the server will start and then it will stop.

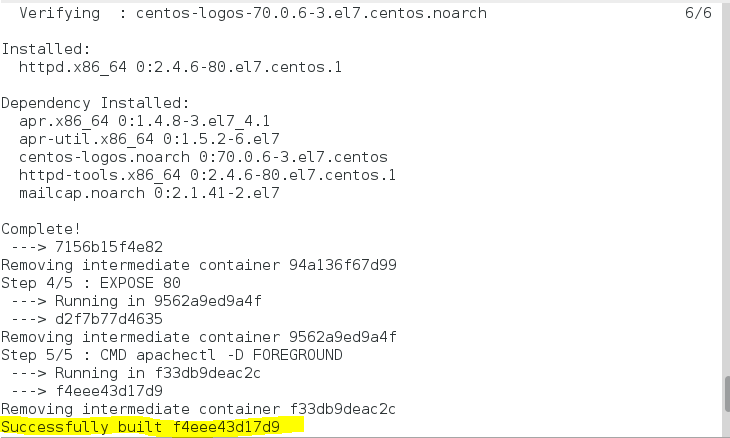
**Step - 3:** Use docker build command to build the docker image from the dockerfile

# docker build -t ws:latest .

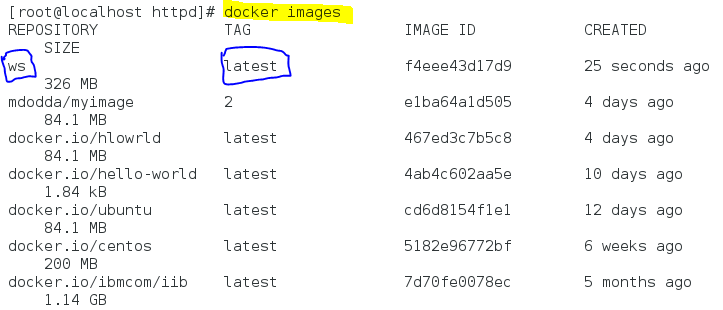
-t : tag name



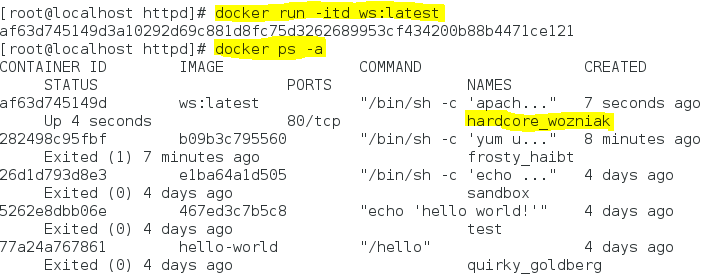
Once image was build we can see below screenshot with an image id.



Check the list of docker images,



**Step - 4:** Run the docker image using “docker run” command and use “docker ps –a” command to see the status of the launched docker container. In the below command, Port mapping is not done.



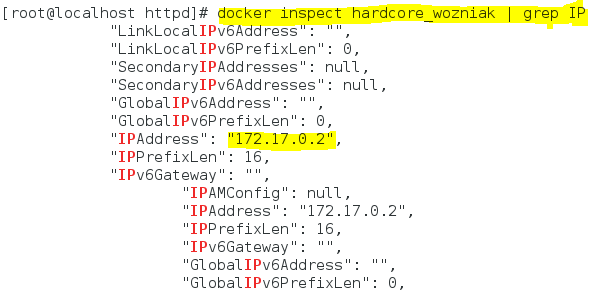
run : used to run the container

-i : interactive session

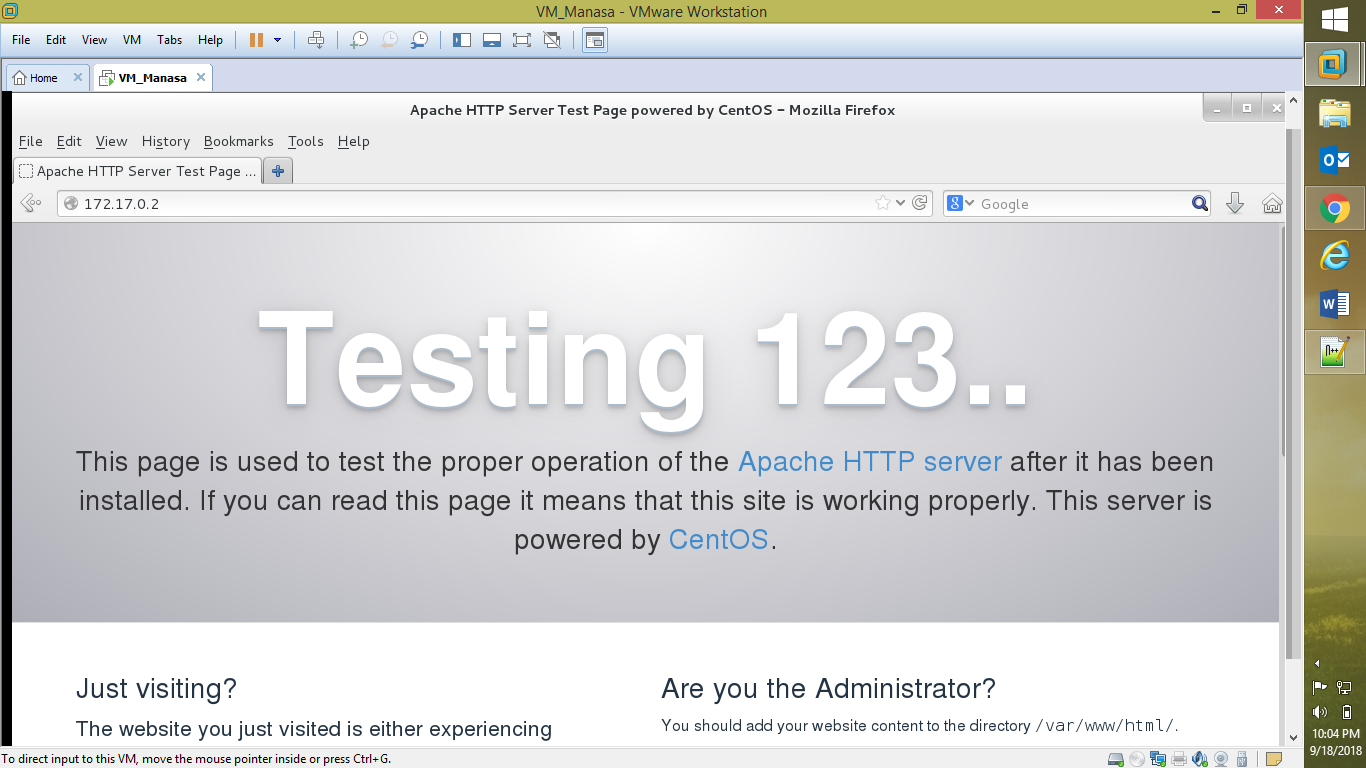
-t : to allocate a pseudo terminal for container

-d : daemonize (continuously)

**Step - 5:** We can now access our web page using the ip which is retrieved by using the inspect command shown in the below screenshot by providing the name of the the particular container (check in above screenshot for container name)



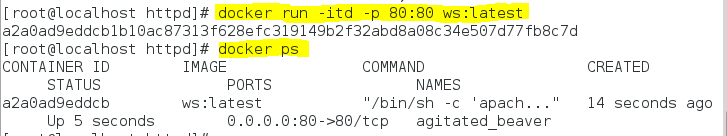
Thus our docker image (apache webserver) is created from the dockerfile.



**(or)**

Maps port 80 with the port 80 of the container (Port mapping is done)

You can run with the -P flag to auto map these ports to ports on your host. Alternatively you can use -p to expose and map any ports of your choice.

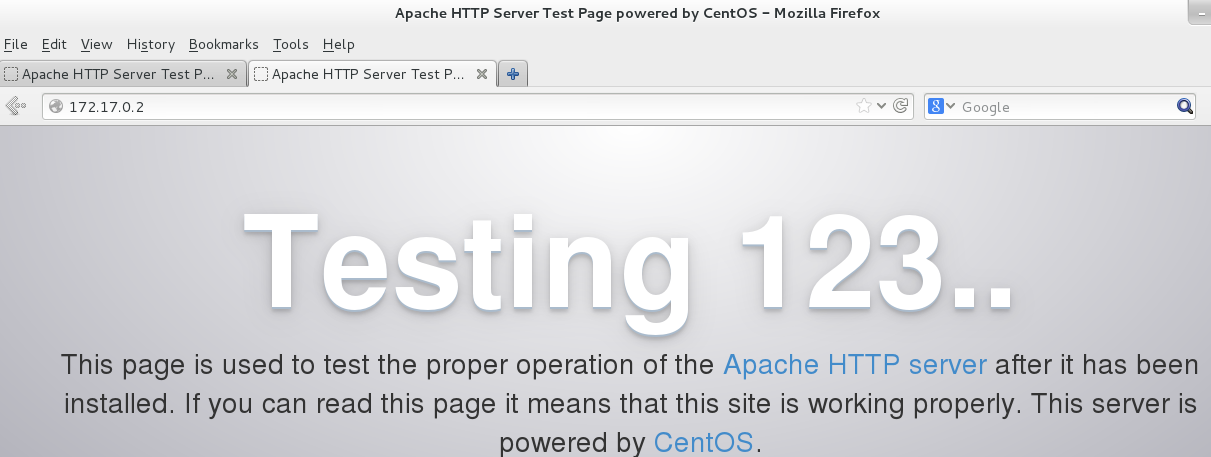




Also it can be tested using IP



Run it in the browser and check it works,



Just for verification the port mapping,



**NOTE: *Differences between Docker CMD & ENTRYPOINT explained in detailed***

Docker CMD and Docker ENTRYPOINT command instructions are used to set a command to be executed when running a container as an executable. Both instructions are used for same purpose but not similar in the functionality. So must be aware of which instruction to be used for our requirement.

Both the declarations can be used in shell form or executable form. Also both has the same syntax as below.

**Example of CMD instruction:**

Shell form:

CMD ping google.com  
CMD python myapplication.py

Executable form:

CMD ["ping","google.com"]  
CMD ["python","myapplication.py"]

**Example of ENTRYPOINT:**  
Shell form:

ENTRYPOINT ping google.com  
ENTRYPOINT python myapplication.py

Executable form:

ENTRYPOINT ["ping","google.com"]  
ENTRYPOINT ["python","myapplication.py"]

Let's see one by one with an example to explain this better.  
  
For this, I have created two dockerfiles, one with CMD instruction and another one with ENTRYPOINT instruction to run a same simple script which executes "ping google.com" as an executable.  
  
**Dockerfile1: With CMD instruction.**  
Here is a dockerfile which has CMD instruction runs our simple script in a container as an executable.

FROM docker.io/centos:latest  
MAINTAINER Devops Engineer  
WORKDIR /data  
RUN echo 'ping google.com' > runapp.sh  
CMD sh runapp.sh

Image has already built using command and created an image tag " docker run -itd cmdchk:latest .”  
  
**Dockerfile2: With ENTRYPOINT instruction.**  
Here is a dockerfile which has ENTRYPOINT instruction.

FROM docker.io/centos:latest  
MAINTAINER Devops Engineer  
WORKDIR /data  
RUN echo 'ping google.com' > runapp.sh  
ENTRYPOINT sh runapp.sh

Image has already built using command and created an image tag " docker run -itd epchk:latest .”

**Testing the Containers:**

Let’s run the containers with the newly created images (cmdchk and epchk).

[root@localhost /]# docker run -dit --name cmdtest cmdchk:latest  
0eb048ff4218472871c99492c57e5369badfeeb7771ea73ba8d3d44f653ca839  
[root@localhost /]# docker run -dit --name eptest epchk:latest  
ea5a97405ee67e2fc5b3cfe9914ac3059ea542844a2ff312d7ebac6f7b613c70  
[root@localhost /]#

Check the logs of newly created containers (cmdtest and eptest) to ensure the application / script has run or not.

[root@localhost /]# docker logs cmdtest  
PING google.com (172.217.24.206) 56(84) bytes of data.  
64 bytes from 2a00:1450:400f:809::200e: icmp\_seq=1 ttl=54 time=29.5 ms  
64 bytes from 2a00:1450:400f:809::200e: icmp\_seq=2 ttl=54 time=29.2 ms  
[root@localhost /]# docker logs eptest  
PING google.com (172.217.24.206) 56(84) bytes of data.  
64 bytes from 2a00:1450:400f:809::200e: icmp\_seq=1 ttl=54 time=28.9 ms  
64 bytes from 2a00:1450:400f:809::200e: icmp\_seq=2 ttl=54 time=29.0 ms  
[root@ localhost /]#

Both containers runs the application / script without any issues.  
  
**What happens if we use any arguments at the end of "docker run" command?**

For example, let’s run the new containers with "date" command.

[root@localhost /]# docker run -dit --name cmdtest2 cmdtest:latest date  
38ab7707591c7a2e0640180aedf66d3bebbd881f746d073aaad72fb78f509145  
[root@localhost /]# docker run -dit --name eptest2 eptest date  
4e244d8a6f682d6a0b086454fec71575f9bc64a5f9a75f3598eb068fcc2a1a3e

Again check the logs of newly created containers (cmdtest2 and eptest2) to ensure the application / script has run or not.

[root@localhost /]# docker logs cmdtest2  
Wed Jun 27 17:23:51 UTC 2018  
[root@localhost /]# docker logs eptest2  
PING google.com (216.58.220.206) 56(84) bytes of data.  
64 bytes from 2a00:1450:400f:809::200e: icmp\_seq=1 ttl=54 time=29.5 ms  
64 bytes from 2a00:1450:400f:809::200e: icmp\_seq=2 ttl=54 time=29.3 ms

Above logs shows differently than before when we use an extra argument at the end of the “docker run” command. This is where the actual problem occurs, our script is not running in our “cmdtest2” container but it works on “eptest2” container**.**

**WHY?**

Because when a user runs a container with any arguments (commands) at the end of “docker run” command, the specified commands override the default argument in CMD instruction, so the container will run the argument given at the end of the docker run command.

But if the same argument is given along with ENTRYPOINT instruction in dockerfile, even when a user gives any argument at the end of the docker run command, that will not override ENTRYPOINT instruction. So instruction will run as it is.

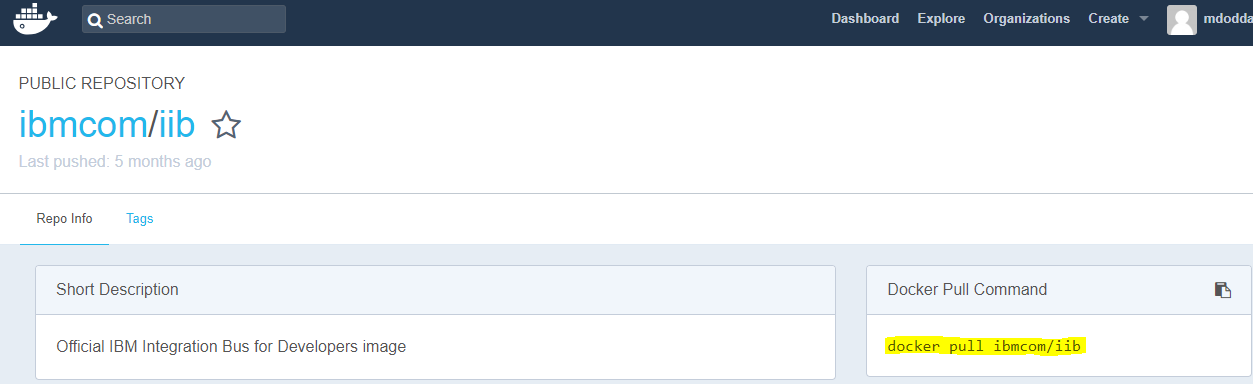
**Scenario - 6: *Docker and IBM Integration Bus (IIB) integration (on CentOS 7)***

To integrate IIB and Docker follow the below steps,

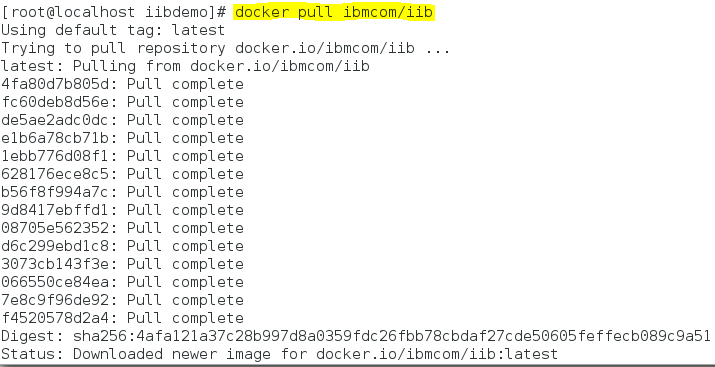
**Step - 1:** Pull the latest IIB image from Docker hub - <https://hub.docker.com/r/ibmcom/iib/>

Firstly, login to the docker hub account. Create one if you don’t have it.

Click on **Explore** > search for an iib image > copy the docker pull command as shown in the screenshot.



**Step - 2:** Go to the terminal and paste the docker command that you have copied



**Step - 3:** Check whether the image is pulled from the docker hub or not by issuing docker command

# **docker images** (which shows list of docker images)

**What the image contains?**

The built image contains a full runtime installation of [IBM Integration Bus for Developers Edition V10.0](https://ibm.biz/iibdevedn). If you wish to include the toolkit in your installation then you should build your own version of our Dockerfile but with the --exclude iib-10.0.0.11/tools part of the tar command removed.



**Step - 4:** Run a container such that it creates a new container of the image.

(Execute the container using # docker exec command only if you want to run iib commands manually such as creating broker and server; if you enter this command then you will enter into the container)

After building a Docker image from the supplied files, you can [run a container](https://docs.docker.com/userguide/usingdocker/) which will create and start an Integration Node to which you can [deploy](http://www-01.ibm.com/support/knowledgecenter/SSMKHH_10.0.0/com.ibm.etools.mft.doc/af03890_.htm) integration solutions.

**Running with the default configuration:**

In order to run a container from this image, it is necessary to accept the terms of the IBM Integration Bus for Developers license. This is achieved by specifying the environment variable LICENSE equal to accept when running the image.

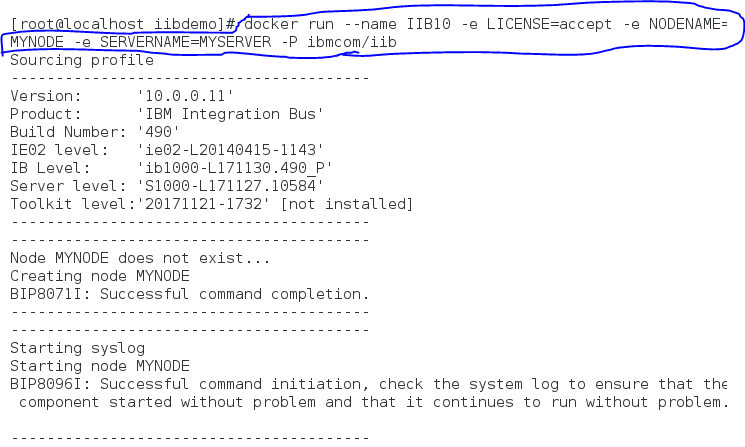
Starting from this Docker image, you can [run a container](https://docs.docker.com/userguide/usingdocker/) which will create and start an Integration Node to which you can [deploy](http://www-01.ibm.com/support/knowledgecenter/SSMKHH_10.0.0/com.ibm.etools.mft.doc/af03890_.htm) integration solutions.

In addition to accepting the license, you can optionally specify an Integration Node name using the NODENAME environment variable and an Integration Server name using the SERVERNAME environment variable.

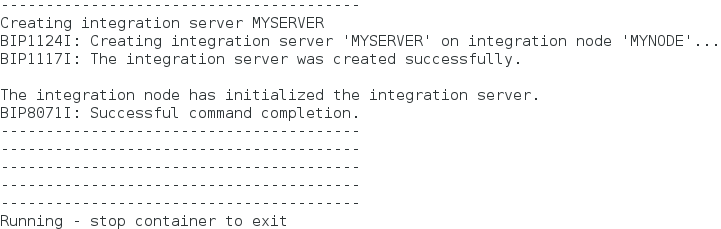
The last important point of configuration when running a container from this image, is port mapping. The image exposes ports 4414 and 7800 by default, for Integration Node administration and Integration Server HTTP traffic respectively. This means you can run with the -P flag to auto map these ports to ports on your host. Alternatively you can use -p to expose and map any ports of your choice.

# **docker run --name IIB10 -e LICENSE=accept -e NODENAME=MYNODE -e SERVERNAME=MYSERVER -P ibmcom/iib**

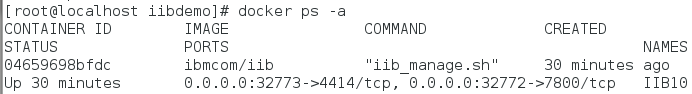
This will run a container that creates and starts an Integration Node, also default integration server is also created.



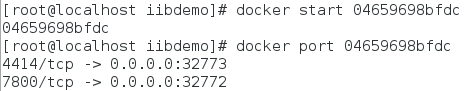
contd…



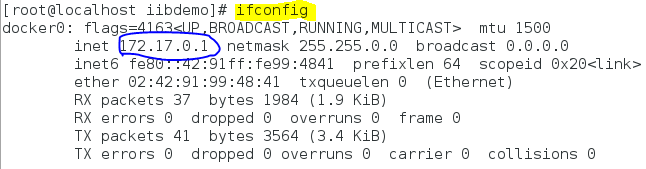
**Step - 5:** Check the list of docker containers whether it is created with the name “IIB10” or not using docker command # **docker ps –a**



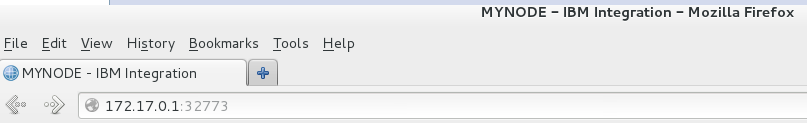
**Step - 6:** Once created, start the container using container id to check the ports have been mapped then connect to the Node's web user interface as normal.



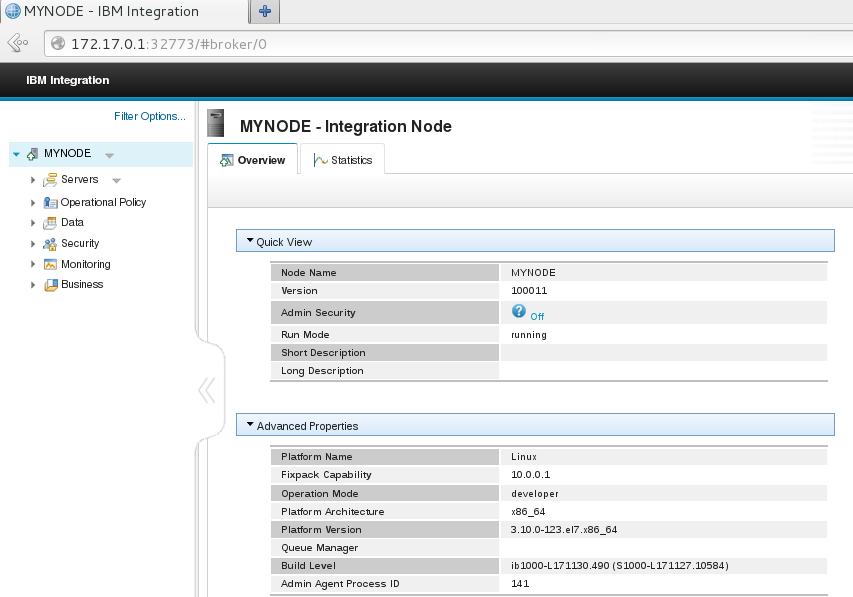
**Step - 7:** Check the Docker’s host IP by issuing the command # **ifconfig**



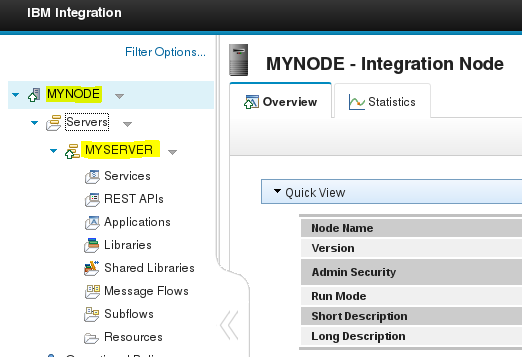
**Step - 8:** Open the browser and provide Docker’s host IP and port as shown below



Overview of full runtime installation of [IBM Integration Bus](https://ibm.biz/iibdevedn) is shown below



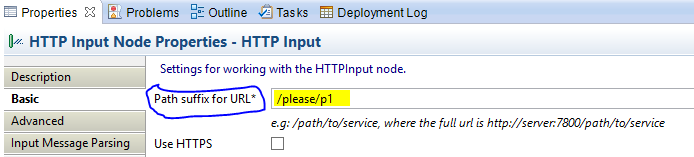
You can see that the integration node and integration server are created with the names respectively “MYNODE” and “MYSERVER”



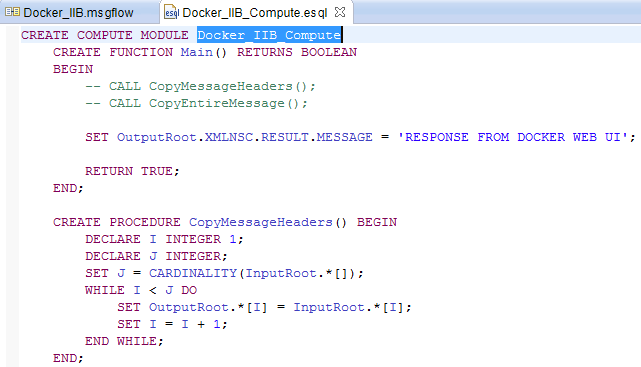
**Step - 9:** Create a new Application in IBM integration toolkit 10.0.0.11 and develop a message flow in the application containing HTTP nodes (HTTP Input, compute and HTTP Reply)

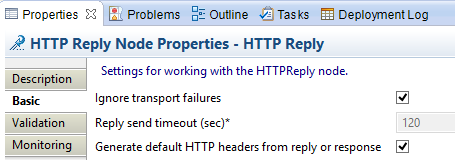


Configure the nodes present in the message flow.

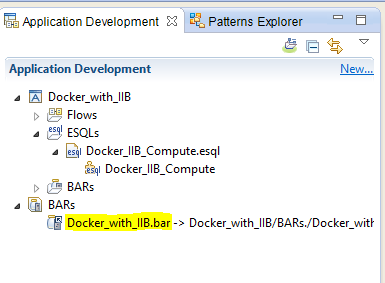


Esql code written in compute node:

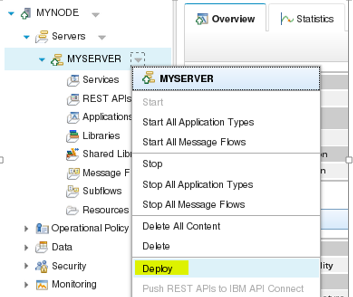




**Step - 10:** Create a new bar file for the respective application/project.

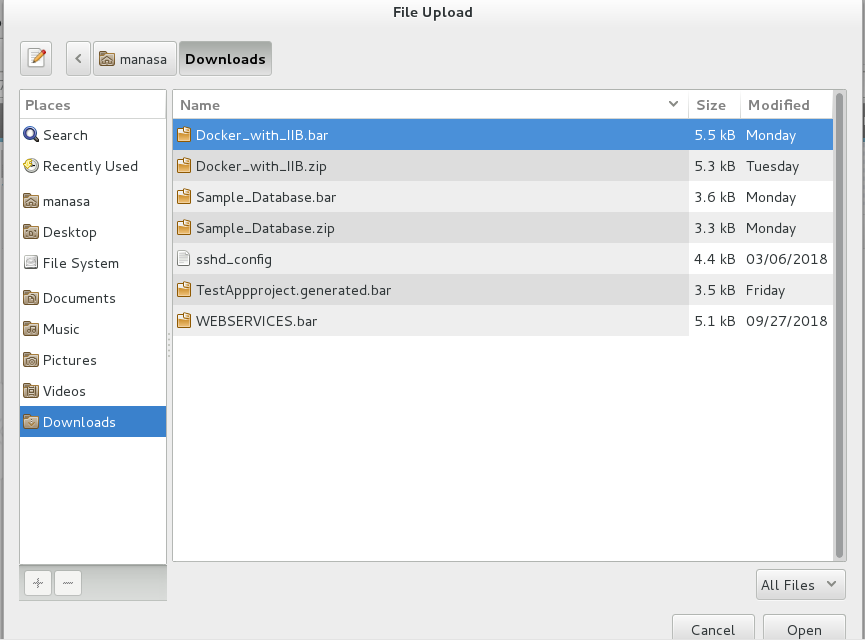


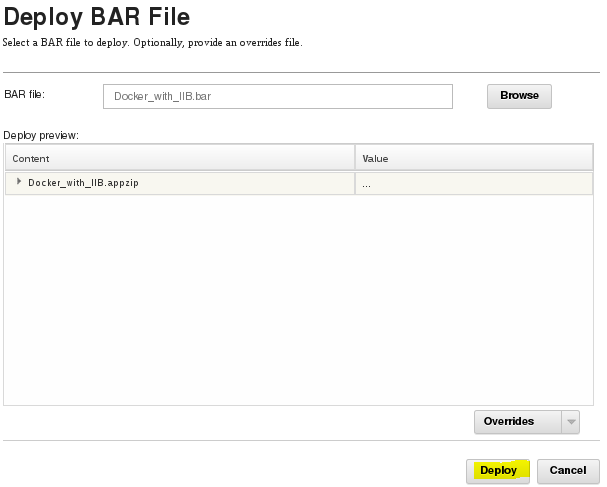
**Step - 11:** Now open the run time environment of IIB WEB UI.



Click on **deploy** and browse the bar file.

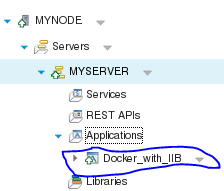


Now, select the required bar file and click **open**.

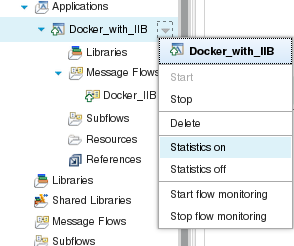
Click Deploy and you can see the content of the message flow there itself.  


Once deployed successfully, we can see as “The deployment of the bar file was successful” in green color on the top of WEB UI.

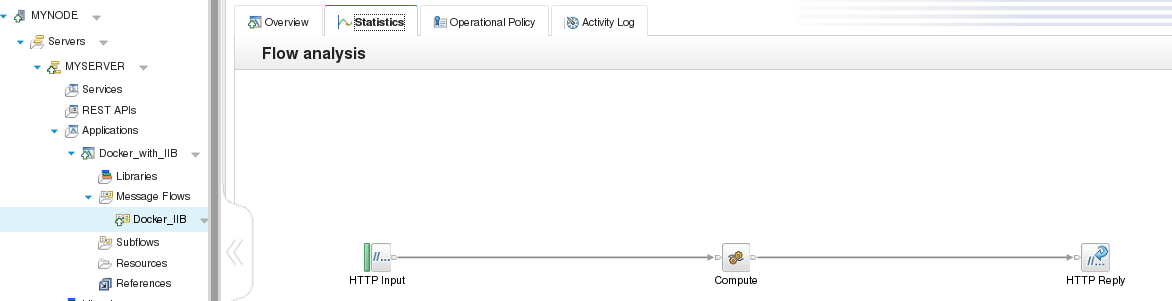
Check below, deployed application can be viewed.



Turn on the statistics on in order to see the flow in UI.



Click on statistics tab and check the message flow.



**NOTE** (Resolved the following issues):

**Issue#** Deployed the bar file, but deployment was failed due to the error BIP2087

The entire internal configuration message failed to be processed successfully.

(Before I deployed bar file of version 9, later I deployed the bar file from IBM Integration toolkit 10)

**Issue#** Error response from daemon: devmapper: Error mounting '/dev/mapper/docker-8:3-269536477-cd6732d0edad34d1b5b48234d7c2f839c69f1c735f7994226d50a70c2d4be008' on '/var/lib/docker/devicemapper/mnt/cd6732d0edad34d1b5b48234d7c2f839c69f1c735f7994226d50a70c2d4be008'. fstype=xfs options=nouuid,context="system\_u:object\_r:svirt\_sandbox\_file\_t:s0:c352,c373": structure needs cleaning

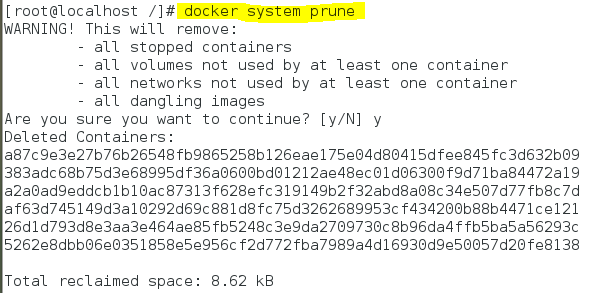
<1>[ 254.847093] XFS (dm-1): metadata I/O error: block 0x521580 ("xlog\_recover\_do..(read#2)") error 117 numblks 16

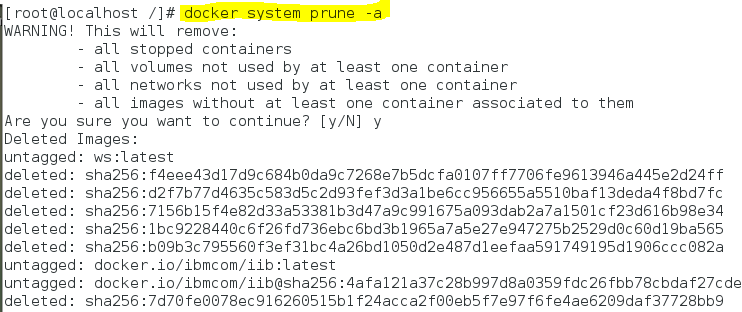
<4>[ 254.847568] XFS (dm-1): log mount/recovery failed: error 117

<4>[ 254.847744] XFS (dm-1): log mount failed

Error: failed to start containers: 70f5574d8b11

Solved the above issue by cleaning the memory as shown below.





Thus below amount of space has been cleaned.

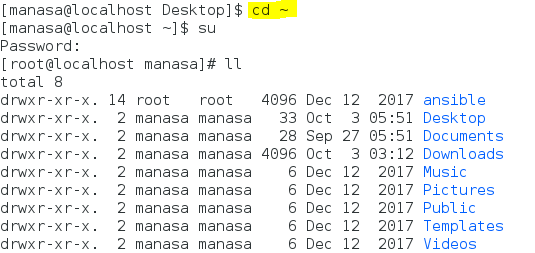


Development of the flow in IBM integration toolkit and deployment of bar file in the runtime environment was completed. Let’s test the whole scenario from SoapUI.

Firstly, need to install the SoapUI.

**Downloading and installation steps for SoapUI via shell script on CentOS 7:**

**Step - 1:** Go to the root directory.



**Step - 2:** Provide the below link to download SoapUI

“wget https://s3.amazonaws.com/downloads.eviware/soapuios/5.4.0/SoapUI-x64-5.4.0.sh”



**Step - 3:** Change the permissions of file or directory.



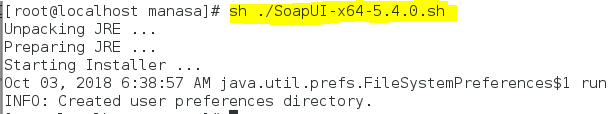
Before changing the permissions



Type command # **ll** (to list the files present in the directory)



**Step - 4:** Install the SoapUI using the below command.



**Step - 5:** Post installation steps to launch the Soap UI

1. A setup wizard is opened, click next
2. Accept the license agreement & click next to continue
3. Choose the installation directory or leave the default installation directory as is
4. Choose the components that you wish to install.
5. Soap UI is checked by default and NOT user configurable.
6. Source – Enable, if you would like to get access to the source code of SOAP-UI. We have not selected it.
7. Hermes JS – Enable, if the application requires JMS testing.
8. Tutorial – Enable, if you want to access SOAP-UI tutorials Post installation.
9. The installation wizard asks the user to download and install 'Load UI'. Since Load[Testing](https://www.guru99.com/software-testing.html)is not the context of the discussion, we can proceed without selecting it. Click ‘Next’
10. If**'Hermes JMS'**is selectedin step#4, then the license agreement for 'Hermes JMS' pops up. Accept the license agreement and click 'Next'.
11. Choose the folder location for tutorials or else leave the default location as is and click 'Next'.
12. Choose the start menu folder location or else leave the default location as is and click 'Next'.
13. Enable the checkbox 'create a desktop icon' and click 'Next'.
14. The Installation starts and upon completing the same, the wizard shows the status **completed the setup wizard**. Click 'Finish'.
15. Upon clicking the 'Finish' button, SOAP UI is launched.

**Related paths:**

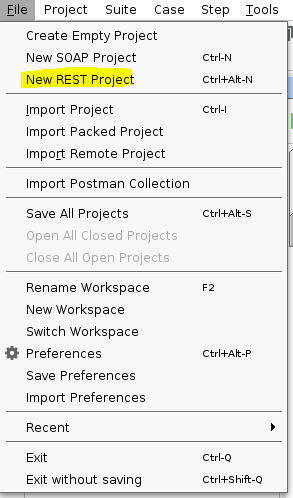
*Path to open SoapUI*: /opt/SmartBear/SoapUI-5.4.0

*Tutorials:* /root/SoapUI

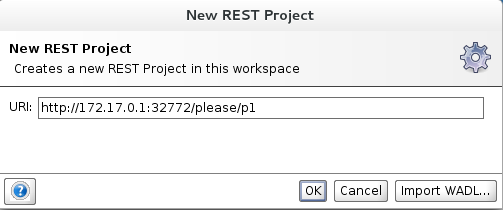
*Like to create symlinks:* /usr/local/bin

Open Soap UI from the path /opt/SmartBear/SoapUI-5.4.0

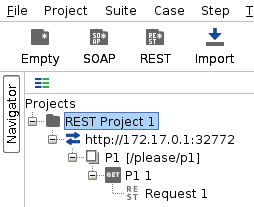
Create a New REST Project



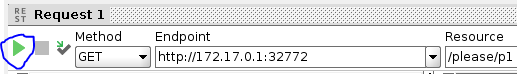
The below pop-up is opened and provide the URI containing the Docker’s host IP, port number and also provide the URL given in the message flow to hit the flow and Click OK.



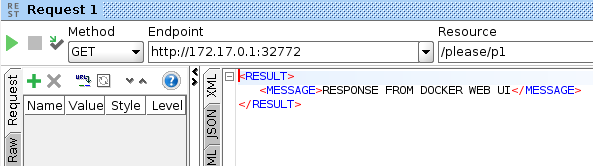
Project is created as shown below.



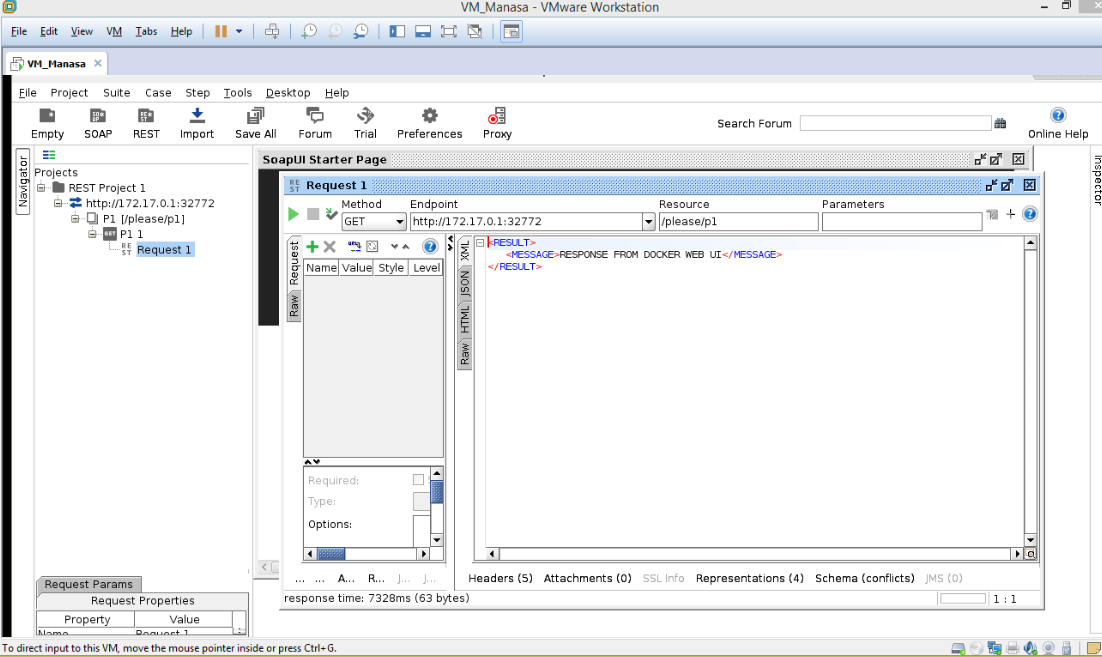
Click on the symbol that is highlighted



And you get the response from the Docker (i.e., message flow present in the run time environment of IIB)



Overview of SoapUI main window can be clearly seen that on left hand side projects are present and output can be seen on the right hand side when we hit the required URL (of the required project).



**Save the projects in Soap UI:**

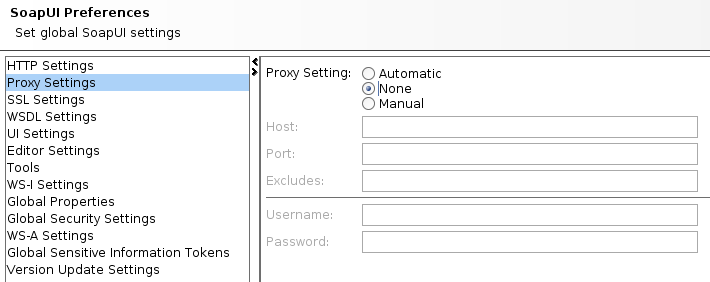
File > Save all projects (all the projects are saved in the current workspace)

Hence, Docker and IBM Integration Bus (IIB) integration wasdone successfully.

**Resolved the following issue I faced when working with the above scenario:**

**Issue#** Error getting response: java.net.SocketTimeoutException: Read timed out

Solved this issue by changing the proxy settings to none in the Preferences

File > Preferences > Proxy settings > Select “None” > Click OK

(or) Directly click on the icon ‘proxy’ to turn on and off in the icons shortcut toolbar.



**Issue#** Error getting response; org.apache.http.conn.HttpHostConnectException to given url is refused

Open command prompt (cmd) > Type the command - ping url (mention the URL you need to test)

Eg: ping <http://172.17.0.1:32768/please/p1>

Click **Enter**

I got the response as “Ping request could not find host http://172.17.0.1:32768/please/p1. Please check the name and try again”

If you do not get any response that means you don’t have access to that URL (Call your system administrator and request access to that URL).

**Scenario - 7: *Setting up Docker and NGINX webserver (Deploying nginx in Docker container)***

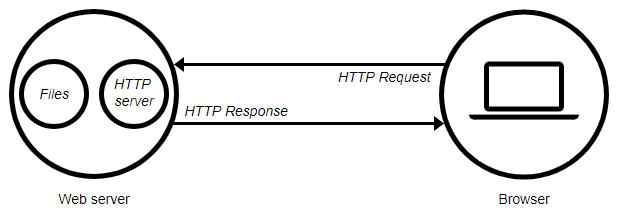
**What is a Web server?**

A Web [server](https://whatis.techtarget.com/definition/server) is a program that uses [HTTP](https://searchwindevelopment.techtarget.com/definition/HTTP) (Hypertext Transfer Protocol) to serve the files that form Web pages to users, in response to their requests, which are forwarded by their computers' HTTP clients.

"Web server" can refer to hardware or software, or both of them working together.

1. On the hardware side, a web server is a computer that stores web server software and a website's component files (e.g. HTML documents, images, CSS stylesheets, and JavaScript files). It is connected to the Internet and supports physical data interchange with other devices connected to the web.
2. On the software side, a web server includes several parts that control how web users access hosted files, at minimum an *HTTP server*. An HTTP server is a piece of software that understands [URLs](https://developer.mozilla.org/en-US/docs/Glossary/URL) (web addresses) and [HTTP](https://developer.mozilla.org/en-US/docs/Glossary/HTTP) (the protocol your browser uses to view webpages). It can be accessed through the domain names (like mozilla.org) of websites it stores, and delivers their content to the end-user's device.

At the most basic level, whenever a browser needs a file which is hosted on a web server, the browser requests the file via HTTP. When the request reaches the correct web server (hardware), the *HTTP server* (software) accepts request, finds the requested document (if it doesn't then a [404](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) response is returned), and sends it back to the browser, also through HTTP.



To publish a website, you need either a static or a dynamic web server.

A **static web server**, or stack, consists of a computer (hardware) with an HTTP server (software). We call it "static" because the server sends its hosted files "as-is" to your browser.

A **dynamic web server**consists of a static web server plus extra software, most commonly an application server and a database. We call it "dynamic" because the application server updates the hosted files before sending them to your browser via the HTTP server.

For example, to produce the final webpages you see in the browser, the application server might fill an HTML template with contents from a database. Sites like MDN or Wikipedia have many thousands of webpages, but they aren't real HTML documents, only a few HTML templates and a giant database. This setup makes it easier and quicker to maintain and deliver the content.

"Static" means "served as-is". Static websites are the easiest to set up, so we suggest you make your first site a static site.

"Dynamic" means that the server processes the content or even generates it on the fly from a database. This solution provides more flexibility, but the technical stack becomes more difficult to handle, making it dramatically more complex to build the website.

**What is Ngnix?**

**Nginx** ( [/ˌɛndʒɪnˈɛks/](https://en.wikipedia.org/wiki/Help:IPA/English) [EN-jin-EKS](https://en.wikipedia.org/wiki/Help:Pronunciation_respelling_key)) (stylized as NGINX, NGiИX or nginx) is a [web server](https://en.wikipedia.org/wiki/Web_server) which can also be used as a  [reverse proxy](https://en.wikipedia.org/wiki/Reverse_proxy), [load balancer](https://en.wikipedia.org/wiki/Load_balancer), [mail proxy](https://en.wikipedia.org/w/index.php?title=Mail_proxy&action=edit&redlink=1) and [HTTP cache](https://en.wikipedia.org/wiki/HTTP_cache).

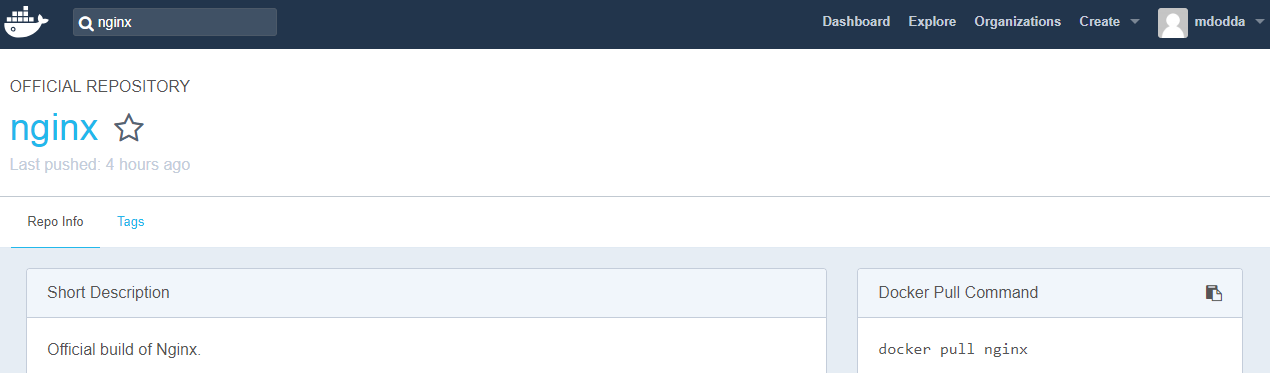
**Nginx** is a popular lightweight web application that is used for developing server-side applications.

It is an open-source web server that is developed to run on a variety of operating systems.

Since nginx is a popular web server for development, Docker has ensured that it has support for nginx.

**Setting up Docker - Nginx web server:**

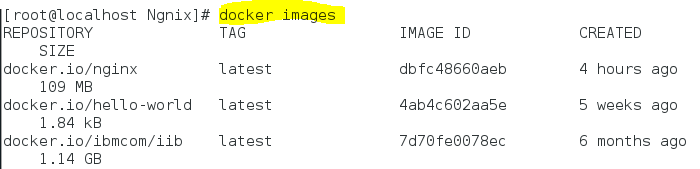
**Step - 1:** The first step is to pull the image from Docker Hub. When you log into Docker Hub, you will be able to search and see the image for nginx as shown below. Just type in nginx in the search box and click on the nginx (official) link which comes up in the search results.



**Step - 2:** You will see that the Docker **pull** command for **nginx** in the details of the repository in Docker Hub.

**Step - 3:** On the Docker Host, use the Docker pull command as shown above to download the latest nginx image from Docker Hub.



**Step - 4:** Verify the list of docker images and check for the nginx image.

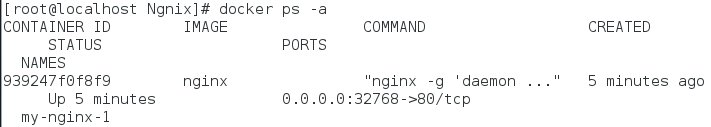
**Step - 5:** Use the docker run command to launch an instance of Nginx running in a container by using the default configuration:



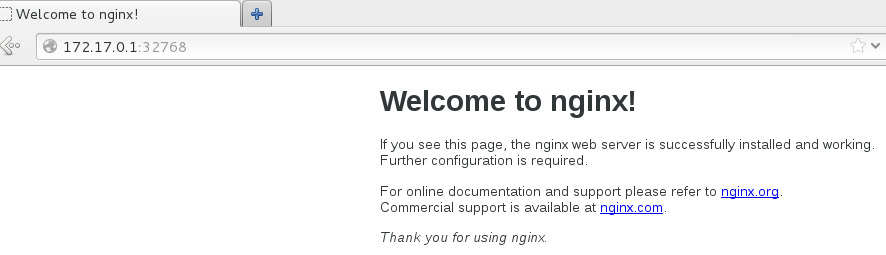
The above command creates a container named my-nginx-1 based on the Nginx image and runs it. The Nginx image exposes ports 80 and 443 in the container and the -P option tells Docker to map those ports to ports on the Docker host that are randomly selected from the range between 49153 and 65535.

We do this because if we create multiple Nginx containers on the same Docker host, we may induce conflicts on ports 80 and 443. The port mappings are dynamic and are set each time the container is started or restarted. If we want the port mappings to be static, set them manually with the -p option. The long form of the "Container Id" will be returned.

**Step - 6:** Run the command # docker ps -a, to verify that the container was created and use the command # docker ps, to verify that the container running, and to see the port mappings.



**Step - 7:** Verify that Nginx is running by making an HTTP request to port 32768 (reported in the output from the preceding command as the port on the Docker host that is mapped to port 80 in the container), the default Nginx welcome page appears as shown below:

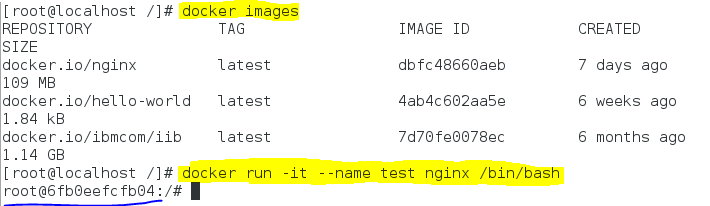


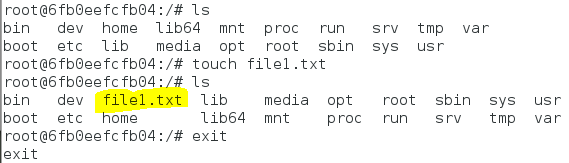
**Scenario - 7.1: *Copying files from a Docker container/containers file system to host machine/local machine***

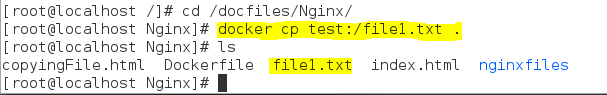
Copying files from a container to Docker host machine.

**Step - 1:** Verify the Docker images for nginx whether its installation is done or not, if not pull the respective image from docker hub.

**Step - 2:** Running the docker container for nginx in interactive shell mode.



**Step - 3:** List the files present in the container and created a new file namely **“file1.txt”**

**Step - 4:** Now check the files you need to copy from the container and exit from it as shown in the above screenshot.

The **docker cp** command serves for copying files and folders between Docker container and a host machine. The docker cp utility copies the contents of **SRC\_PATH** to the **DEST\_PATH.**

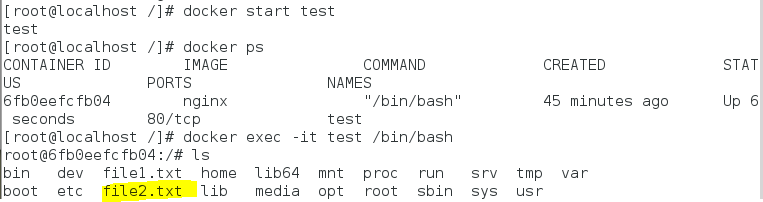
So copy the files into the respective path using docker cp command as shown above and check them using the **ls** command to check the list of files in the destination; where **test** is the container of nginx image and **file1.txt** is a file from the container.

**Scenario - 7.2: *Copying files from host machine to the Docker container***

**Step - 1:** Go to the location where the file is present and then issue the below shown command in the screenshot as I would like to copy “**file2.txt”** into the “**test”** container.



**Step - 2:** Now start the respective container where the file is copied and execute the container in interactive shell mode, then list the files. You can check the copied file in the list.



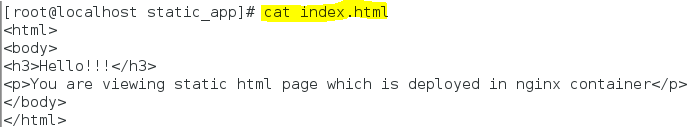
Thus files are copied from Docker container to the host machine and vice versa.

We can follow the above method if we need to take the backups from the container’s fil system to the local machine.

**Scenario - 7.3: *Deploy static website as container& Run Static Website Using Docker (or) hosting a simple web page into the ngnix container.***

**Step - 1:** Created a new directory “**static\_app**” > moved to that directory > created an “**index.html”** file which contained some static data.



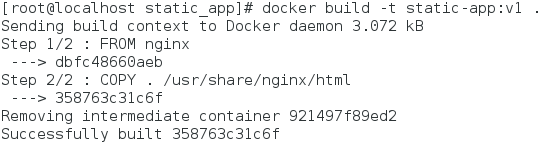


**Step - 2:** Created a Dockerfile in order to build a Docker image. The docker file consists of docker instruction to copy a file which has a static content to the html folder of nginx web server.





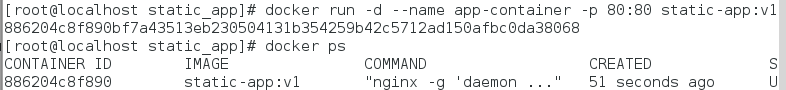
**Step - 3:** Built the docker image with the image name “**static-app**” and the tag name “**v1**”from the above shown dockerfile.



**Step - 4:** Check the list of docker images.

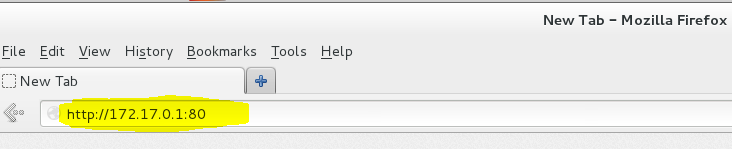


**Step - 5:** Now created the nginx instance using docker run command by providing a new name for the container as “**app-container**”.

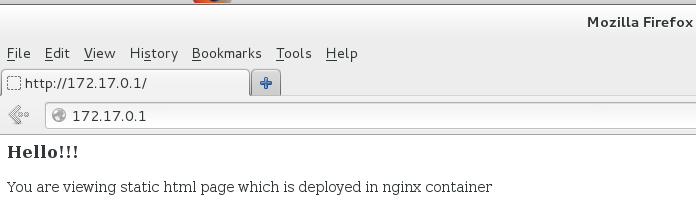


**Step - 6:** Verify the containers list using docker command # **docker ps** whether the container is running or not

**Step - 7:** Open the browser and issue the docker host IP and the port that is assigned to the nginx container as shown below,



Press Enter



Display logs of docker by providing its container name



**Scenario - 8: *Linking the containers***

**Use of linking the containers:**

 By linking containers, we can provide a secure channel via which Docker containers can communicate to each other.

**What is Redis?**

Redis is an open source, BSD licensed, advanced key-value store. It is often referred to as a data structure server, since the keys can contain strings, hashes, lists, sets and sorted sets. Redis is written in C.

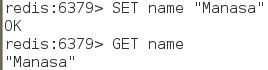
Redis supports 5 types of data types, I have used one of the data type as shown below.

**Usage of data types in redis:**

## **Strings**

Redis string is a sequence of bytes. Strings in Redis are binary safe, meaning they have a known length not determined by any special terminating characters. Thus, you can store anything up to 512 megabytes in one string.

### **Example**



In the above example, **SET** and **GET** are Redis commands, **name** is the key used in Redis and **Manasa** is the string value that is stored in Redis.

**Note**: A string value can be at max 512 megabytes in length.

Thinking of a sample application. You might have a Server and a client. When we talk about linking Docker Containers, what we are talking about here is the following:

1. We can launch one Docker container that will be running the Server.
2. We will launch the second Docker container (Client) with a link flag to the container launched in **Step 1**. This way, it will be able to talk to the Server via the link name.

This is a generic and portable way of linking the containers together via the --link flag rather than via the networking. A new tool [Docker Compose](https://docs.docker.com/compose/) is the recommended way moving forward but here I used the --link flag only.

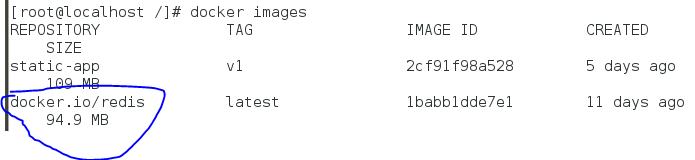
**Steps to be followed to link the container:**

**Step - 1:** Firstly, I have launched the popular NoSQL Data Structure Server Redis. Like other software, Redis too has its official Docker image available in the Docker Hub.

Go to the docker hub > search for redis image > pull down the Redis image using the following command

# **docker pull redis**

**Step - 2:** Verify the list of docker images for the pulled image “redis”



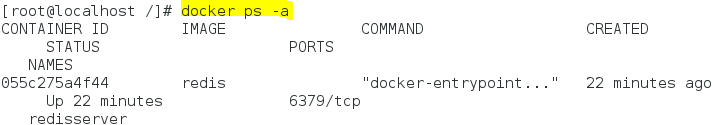
**Step - 3:** Next launch a Redis container (named **redisserver**) in detached mode as follows,

# **docker run -d --name redisserver redis**

We can check that **redisserver** container has started via the following command:



Verify the containers list for the container **redisserver**, which is launched by the default port number.



Notice that it has started on port 6379.

**Step - 4:** Start the redis server container as shown below,

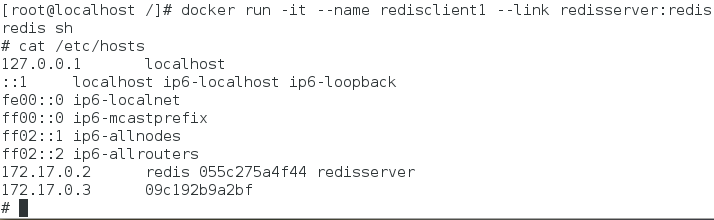


**Step - 5:** Now, let us run another container, a **redisclient1** container as shown below

# **docker run -it --name redisclient1 --link redisserver:redis redis sh**

Notice the --**link** flag. The value provided to the **-- link** flag is **sourcecontainername:containeraliasname**. We have chosen the value **redisserver** in the **sourcecontainername** since that was the name that was given to our first container that we launched earlier. The **containeraliasname** has been selected as **redis** and it could be any name of one’s choice.

The above launch of container (**redisclient1**) will lead you to the shell prompt.



Notice an entry at the end, where the container **redisserver** has got associated with the **redis** name.

If you print out the environment variables you will see the following and we can see that various environment variables were auto-created for us to help reach out to the **redisserver** from the **redisclient1**.



**Step - 6:** We have a redis client container running in the shell prompt so that all the redis client tools are ready for us. Note that the container redis server (**redisserver)** that we launched is still running.

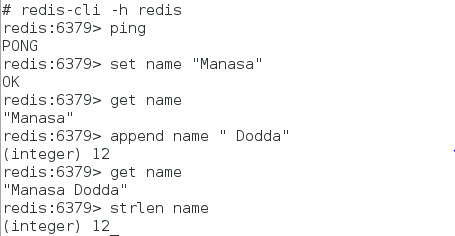
So, now launch the redis client (**redis-cli**) and connect to our redis server (running in another container and to which we have linked) as given below:



I have already created a container redisclient, so I have used “docker exec” command to enter into the container.

You can see that we have been able to successfully able to connect to the redis server via the alias name that we specified in the --**link** flag while launching the container. Of course if we were running the Redis server on another port (other than the standard 6379) we could have provided the **-p** parameter to the **redis-cli** command and used the value of the environment variable over here (**REDIS\_PORT\_6379\_TCP\_PORT**).

Now, let us execute some standard redis commands:



**Step - 7:** Opened another terminal and created (or run) other client container (**redisclient2**). Now connect to the server by linking redis**client2** container to the redis server container (rediseserver),

# **docker run -it --name redisclient2 --link redisserver:redis redis sh**

**Step - 8:** Connect to the redis server via alias name that is specified in the --**link** flag while launching the container.



**Step - 9:** Now we should be able to retrieve the data in the current client container (redisclient2) from the first client container (**redisclient1**) that is stored in string key / value pair format.

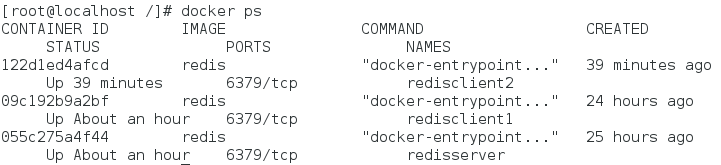
Once again, we execute few redis commands to validate things:





Retrieved the required value from the given key, also retrieved the length of the given key successfully as we linked two client containers to the server container.

Just verify that our three containers must be running,



**Scenario - 9: *Creating volumes and accessing them in other containers & sharing the data between the containers.***

**(Or)**

***Sharing directories/folders/volumes between containers and accessing volume from one container to the other container (for this, create a volume in one container and access it in another container)]***

Docker containers themselves are non-persistent by nature. If you want some persistency for the data on the Docker container, volumes are the way to go. Besides containers sometime need data from the host so mounting (a backing, setting, or support for something.) the host data as a volume on the container is the perfect solution.

Some of the practical examples of the volume usage are: Running any database containers like mysql, cassandra, mongo db and mounting a data folder from the host to store the database. Or mounting a bunch of config files while running your application container.

Here in this scenario, I have created a volume in one container and accessed it from another container.

**Use of volumes:**

In order to be able to save (persist) data and also to share data between containers, Docker came up with the concept of volumes. Quite simply, volumes are directories (or files) that are outside of the default Union File System and exist as normal directories and files on the host filesystem.

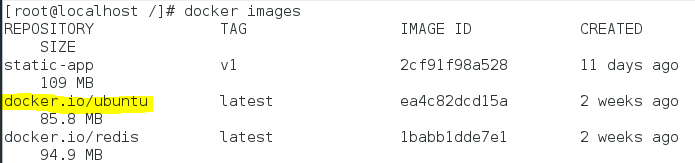
There are several ways to initialize volumes, with some subtle differences that are important to understand. The most direct way is declare a volume at run-time with the -v flag (use -v to map volumes between containers).

Volumes have several advantages:

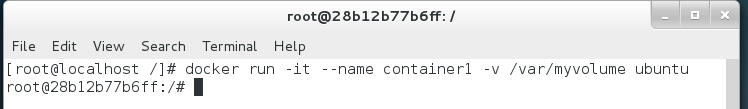
* Volumes are easier to back up or migrate.
* You can manage volumes using Docker CLI commands or the Docker API.
* Volumes work on both Linux and Windows containers.
* Volumes can be more safely shared among multiple containers.
* Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
* New volumes can have their content pre-populated by a container.

In addition, volumes are often a better choice than persisting data in a container’s writable layer, because a volume does not increase the size of the containers using it, and the volume’s contents exist outside the lifecycle of a given container.

**Step - 1:** Search for and pull the Ubuntu image from docker hub account and use docker command to check for the pulled image.

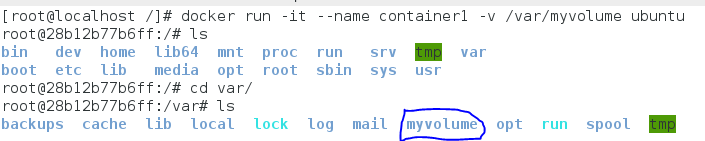


**Step - 2:** Run a container (container1) using the docker image Ubuntu as containers are instances of Docker images that can be run using the Docker run command and create new volume /myvolume in the container under /var folder.

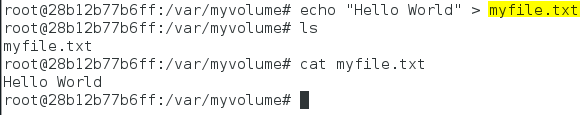


Used -v flag to map the volumes between the containers (used for standalone containers).

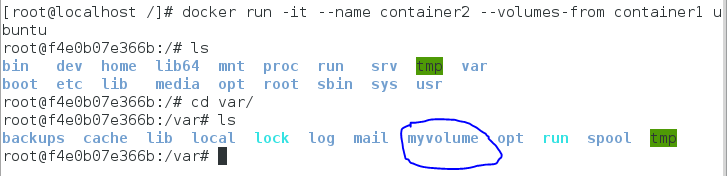
**Step - 3:** Once the above container is run, you will enter into the container and check for the folder/volume that is created in the given path.

docker creates “myvolume” directory inside the ubuntu and is accessible from outside

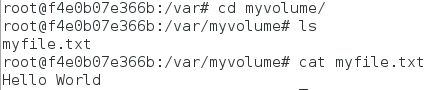
**Step - 4:** Create a new file “myfile.txt” and added some content to the file.



**Step - 5:** Open a new terminal and run another container with the name “container2” using the flag --volumes-from in order to share the volumes from one container (container1) to the other container (container2).

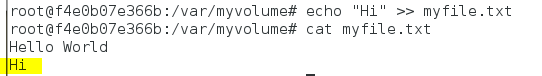


We can find the volume “myvolume” is shared in the container (container2) as shown in the above screenshot. Also we can find that the files under the volume (myvolume) directory are also shared as shown in the below screenshot. We are able to the data present in the files.



**Step - 6:** Now modify the file in the volume (myvolume) container2 and automatically the file in the container1 is updated.

Added another line or message (Hi) to the file as shown below,



**Step - 7:** Now open other terminal where the container1 is running and check whether the file is updated or not.



This is how the data is shared between the containers successfully by creating the volume in one container and accessed it from the other container (using volumes).

**Scenario - 10: *Build a simple Python web application running on Docker Compose. The application uses the Flask framework and maintains a hit counter in Redis.***

**Overview of docker compose:**

Compose is a tool for defining and running multi-container Docker applications.

With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration.

**Features:**

The features of Compose that make it effective are:

* [Multiple isolated environments on a single host](https://docs.docker.com/compose/overview/#Multiple-isolated-environments-on-a-single-host)
* [Preserve volume data when containers are created](https://docs.docker.com/compose/overview/#preserve-volume-data-when-containers-are-created)
* [Only recreate containers that have changed](https://docs.docker.com/compose/overview/#only-recreate-containers-that-have-changed)
* [Variables and moving a composition between environments](https://docs.docker.com/compose/overview/#variables-and-moving-a-composition-between-environments)

**Multiple isolated environments on a single host:**

Compose uses a project name to isolate environments from each other. You can make use of this project name in several different contexts:

* on a dev host, to create multiple copies of a single environment, such as when you want to run a stable copy for each feature branch of a project
* on a CI server, to keep builds from interfering with each other, you can set the project name to a unique build number
* on a shared host or dev host, to prevent different projects, which may use the same service names, from interfering with each other

**Preserve volume data when containers are created:**

Compose preserves all volumes used by your services. When docker-compose up runs, if it finds any containers from previous runs, it copies the volumes from the old container to the new container. This process ensures that any data you’ve created in volumes isn’t lost.

### **Only recreate containers that have changed:**

Compose caches the configuration used to create a container. When you restart a service that has not changed, Compose re-uses the existing containers. Re-using containers means that you can make changes to your environment very quickly.

### **Variables and moving a composition between environments:**

Compose supports variables in the Compose file. You can use these variables to customize your composition for different environments, or different users.

You can extend a Compose file using the **extends** field or by creating multiple Compose files.

## **Common use cases:**

Compose can be used in many different ways. Some common use cases are outlined below.

### **Development environments:**

When you’re developing software, the ability to run an application in an isolated environment and interact with it is crucial. The Compose command line tool can be used to create the environment and interact with it.

The [Compose file](https://docs.docker.com/compose/compose-file/) provides a way to document and configure all of the application’s service dependencies (databases, queues, caches, web service APIs, etc). Using the Compose command line tool you can create and start one or more containers for each dependency with a single command (docker-compose up).

Together, these features provide a convenient way for developers to get started on a project. Compose can reduce a multi-page “developer getting started guide” to a single machine readable Compose file and a few commands.

### **Automated testing environments:**

An important part of any Continuous Deployment or Continuous Integration process is the automated test suite. Automated end-to-end testing requires an environment in which to run tests. Compose provides a convenient way to create and destroy isolated testing environments for your test suite. By defining the full environment in a [Compose file](https://docs.docker.com/compose/compose-file/), you can create and destroy these environments in just a few commands:

$ docker-compose up -d

$ ./run\_tests

$ docker-compose down

### **Single host deployments:**

Compose has traditionally been focused on development and testing workflows, but with each release we’re making progress on more production-oriented features. You can use Compose to deploy to a remote Docker Engine.

**Install Docker Compose:**

You can run Compose on macOS, Windows, and 64-bit Linux.

**Prerequisites:**

Docker Compose relies on Docker Engine so make sure you have Docker Engine installed either locally or remote, depending on your setup.

* On desktop systems like Docker for Mac and Windows, Docker Compose is included as part of those desktop installs.
* On Linux systems, first install the [Docker](https://docs.docker.com/install/#server)  Compose on Linux systems.

**Steps to install and test Docker Compose on a CentOS 7 server, and used the Compose file in the YAML format:**

**Install Docker Compose:**

Once Docker has been installed, install Docker Compose. First of all, install the EPEL repository by executing the command:

# **yum install epel-release**

Next, install python-pip:

# **yum install -y python-pip**

At this point, it is possible to install Docker Compose by executing a pip command:

# **pip install docker-compose**

Upgrade also all the Python packages on CentOS 7:

# **yum upgrade python\***

Check Docker Compose version with the following command:

$ **docker-compose -v**

The output should be something like this:

docker-compose version 1.18.0, build 8dd22a9

**Testing Docker Compose:**

The Docker Hub includes a Hello World image for demonstration purposes, illustrating the configuration required to run a container with Docker Compose.

**Create a new directory and move into it:**

# mkdir hello-world

# cd hello-world

**Create a new YAML file:**

# vim docker-compose.yml

In this file paste the following content:

unixmen-compose-test:

image: hello-world

**Note:***The first line is used as part of the container name.*

Save and exit.   
Run the container

**Next, execute the following command in the hello-world directory:**

# docker-compose up

If everything is correct, this should be the output shown by Compose

Creating dctest\_unixmen-compose-test\_1 ... done

Attaching to dctest\_unixmen-compose-test\_1

unixmen-compose-test\_1 |

unixmen-compose-test\_1 | Hello from Docker!

unixmen-compose-test\_1 | This message shows that your installation appears to be working correctly.

unixmen-compose-test\_1 |

unixmen-compose-test\_1 | To generate this message, Docker took the following steps:

unixmen-compose-test\_1 | 1. The Docker client contacted the Docker daemon.

unixmen-compose-test\_1 | 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.

unixmen-compose-test\_1 | (amd64)

unixmen-compose-test\_1 | 3. The Docker daemon created a new container from that image which runs the

unixmen-compose-test\_1 | executable that produces the output you are currently reading.

unixmen-compose-test\_1 | 4. The Docker daemon streamed that output to the Docker client, which sent it

unixmen-compose-test\_1 | to your terminal.

unixmen-compose-test\_1 |

unixmen-compose-test\_1 | To try something more ambitious, you can run an Ubuntu container with:

unixmen-compose-test\_1 | $ docker run -it ubuntu bash

unixmen-compose-test\_1 |

unixmen-compose-test\_1 | Share images, automate workflows, and more with a free Docker ID:

unixmen-compose-test\_1 | https://hub.docker.com/

unixmen-compose-test\_1 |

unixmen-compose-test\_1 | For more examples and ideas, visit:

unixmen-compose-test\_1 | https://docs.docker.com/get-started/

unixmen-compose-test\_1 |

dctest\_unixmen-compose-test\_1 exited with code 0

# **Get started with Docker Compose real time application:**

Building a simple Python web application running on Docker Compose. The application uses the Flask framework and maintains a hit counter in Redis.

## **Prerequisites:**

Make sure that we need to have already installed both [Docker Engine](https://docs.docker.com/install/) and [Docker Compose](https://docs.docker.com/compose/install/). No need to install Python or Redis, as both are provided by Docker images.

**Step 1:** Setup

Define the application dependencies.

1. Create a directory for the project:

#mkdir testcompose

#cd testcompose

2. Create a file called **app.py** in the project directory with following code,

import time

import redis

from flask import Flask

app = Flask(\_\_name\_\_)

cache = redis.Redis(host='redis', port=6379)

def get\_hit\_count():

retries = 5

while True:

try:

return cache.incr('hits')

except redis.exceptions.ConnectionError as exc:

if retries == 0:

raise exc

retries -= 1

time.sleep(0.5)

@app.route('/')

def hello():

count = get\_hit\_count()

return 'Hello World! I have been seen {} times.\n'.format(count)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host="0.0.0.0", debug=True)

In this example, redis is the hostname of the redis container on the application’s network. We use the default port for Redis, 6379.

**Note:** The way the get\_hit\_count function is written. This basic retry loop lets us attempt our request multiple times if the redis service is not available. This is useful at startup while the application comes online, but also makes our application more resilient if the Redis service needs to be restarted anytime during the app’s lifetime. In a cluster, this also helps handling momentary connection drops between nodes.

3. Created another file called **requirements.txt** in the project directory,

flask

redis

**Step 2:** Create a **Dockerfile**

Created a Dockerfile that builds a Docker image. The image contains all the dependencies the Python application requires, including Python itself.

Created a file named Dockerfile with the following code:

1. FROM python:3.4-alpine
2. WORKDIR /code
3. RUN pip install -r requirements.txt
4. CMD ["python", "app.py"]

**Note:** Do not include the numbers in the dockerfile, it is just for illustrating the example.

This tells Docker to:

1. Build an image starting with the Python 3.4 image.
2. Add the current directory **.** into the path /code in the image.
3. Set the working directory to /code.
4. Install the Python dependencies.
5. Set the default command for the container to python app.py.

**Step 3:** Define services in the compose file

Created a file called **docker-compose.yml** in the project directory with the following code,

version: '3'

services:

web:

build: .

ports:

- "5000:5000"

redis:

image: "redis:alpine"

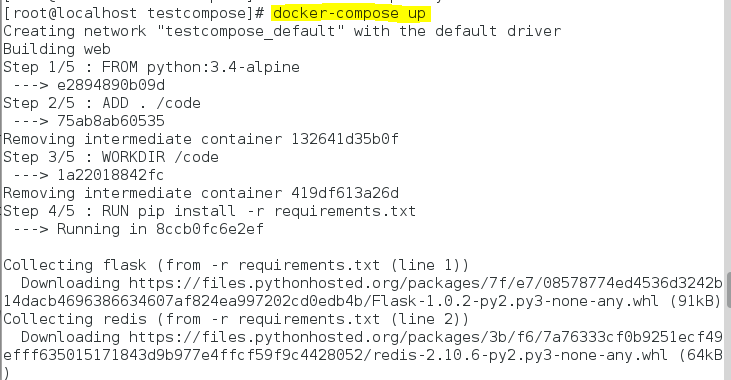
This compose file defines two services, **web** and **redis**. The web service:

* Uses an image that’s built from the Dockerfile in the current directory.
* Forwards the exposed port 5000 on the container to port 5000 on the host machine. We use the default port for the Flask web server, 5000.

The redis service uses a public Redis image pulled from the Docker Hub registry.

**Step 4:** Build and run app with compose.

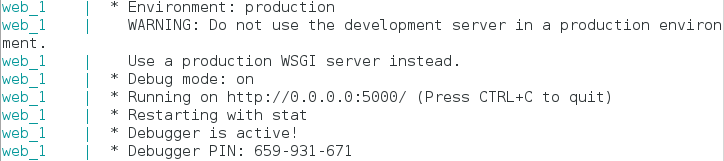
1. From the project directory, start up the application by running #**docker-compose up** command.



contd…….



contd……



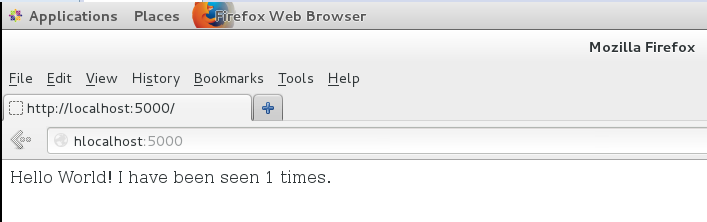
Compose pulls a Redis image, builds an image for your code, and starts the services defined. In this case, the code is statically copied into the image at build time.

1. Enter http://0.0.0.0:5000/ in a browser to see the application running.

Web app should now be listening on port 5000 on our Docker daemon host. Now, point web browser to http://localhost:5000 to find the Hello World message. If this doesn’t resolve, we can also try http://0.0.0.0:5000.

The following message should be seen in the browser saying:

Hello World! I have been seen 1 times.



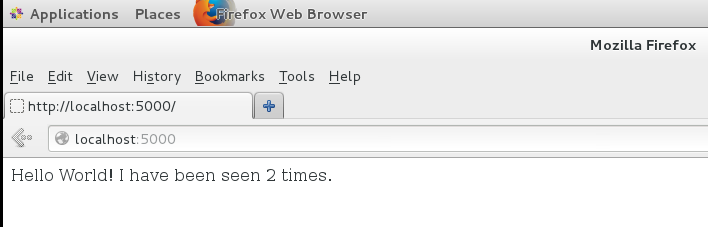
You can find the following screen in the docker terminal.



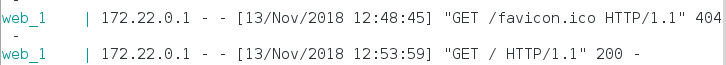
1. Refresh the page.

The number should increment.

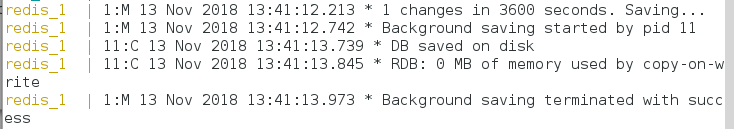
Hello World! I have been seen 2 times.



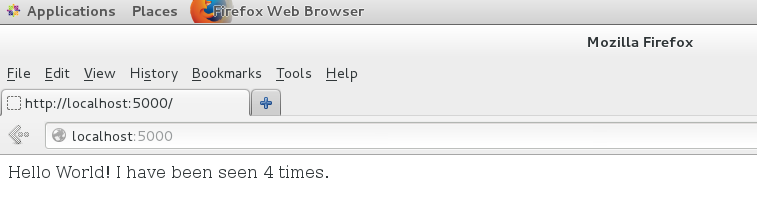
Same, you can find the following screen in the docker terminal after a hit to the url in the browser.



If terminal is not in use, below message is displayed.

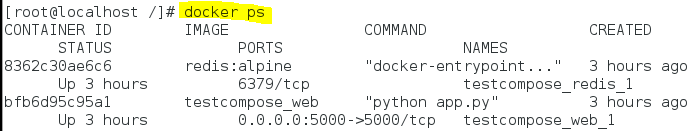


After two other hits to the url,



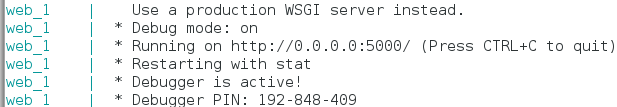


1. Switch to another terminal window, and type # **docker ps** to list the docker containers that are running, also same with the command # **docker images** at this point should return redis and web.

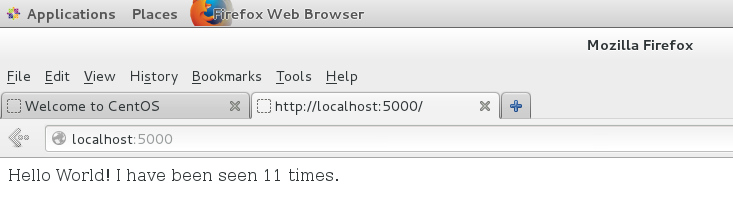


1. Stop the application, either by running # **docker-compose down** from within the project directory in the second terminal, or by hitting CTRL+C in the original terminal where you started the app (same terminal).

When app is started the following will be in the screenshot,



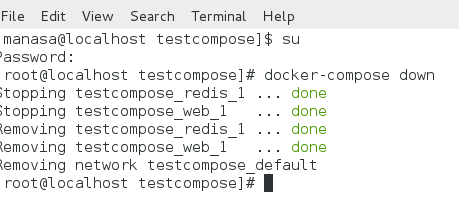
When we hit the url,



Following is displayed in the terminal,



Switched to the other terminal and issued # **docker-compose down** command to stop the application.



You can find the below in the terminal.

