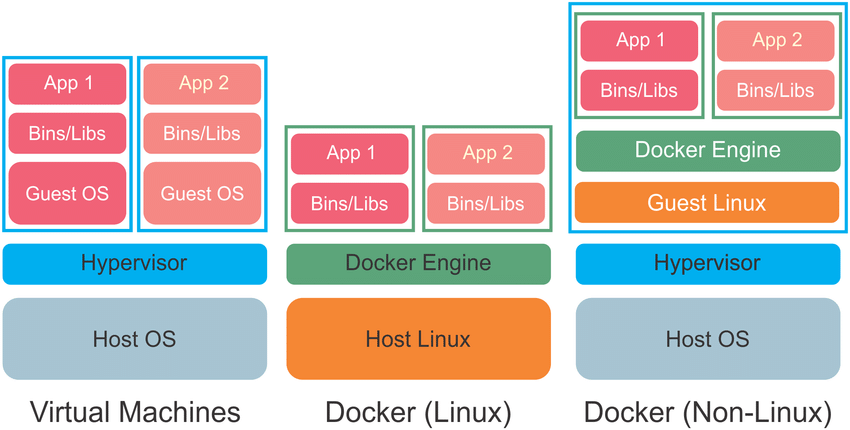
**Docker** is a tool used to create virtual machines called "containers".  
is a tool that performs operating-system-level virtualization, also known as containerization".



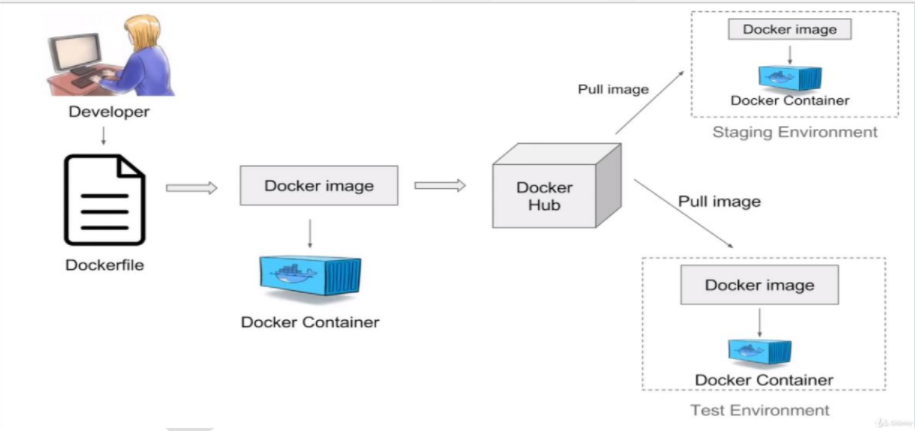
* Docker is a tool designed to make containers in which we can deploy any type of applications easily.
* Docker uses union file system (layered)
* Docker performs os-level virtualization  
    
  **Container** : A container like a virtual machine  
  **Docker** : Docker is a tool to create those virtual machines

**Docker benefits**

* Containerization (OS level virtualization)(No need guest OS)
* No pre-allocation of RAM
* Can replicate same environment
* Less cost
* Less weight (MB’s in size)
* Fast to fire up
* Can run on physical/virtual/cloud
* Can re-use(same image)
* Can create machines in less time.

**Docker components**

**Docker image**: Contains OS(very small)(almost negligible) + softwares  
 **Docker Container**: Container like a machine which is created from Docker image.  
 **Docker file**: Describes steps to create a docker image.  
 **Docker hub/registry**: Stores all docker images publicly.  
 **Docker daemon**: Docker service



**Ways to create Docker Images:**

* Take image from Docker hub
* Create image from existing docker containers
* Create image from docker file

**Dockerfile** : A text file with instructions to build image  
Automation of Docker Image Creation

**o FROM**

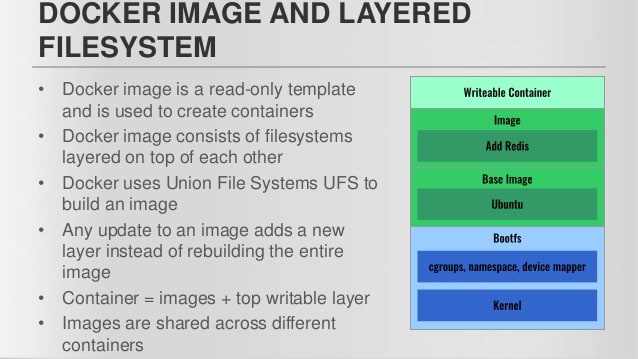
**o RUN**

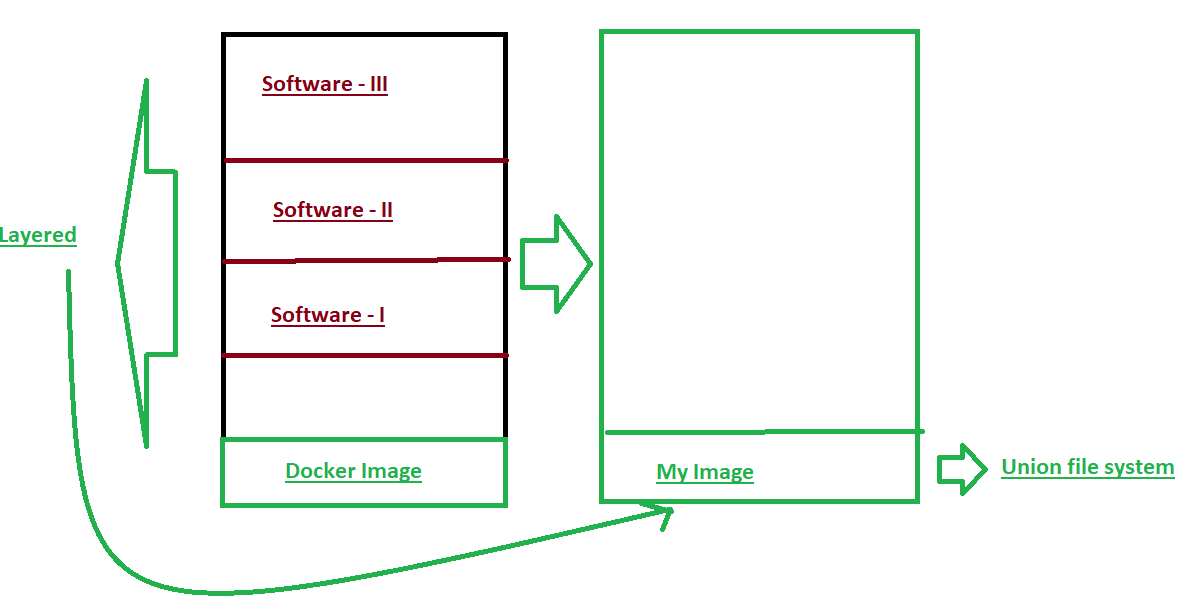
**o CMD**

* Step 1 : Create a file named Dockerfile
* Step 2 : Add instructions in Dockerfile
* Step 3 : Build dockerfile to create image
* Step 4 : Run image to create container

**Volumes:**

* Volume is a directory inside your container
* First declare directory as a volume and then share volume
* Even if we stop container, still we can access volume
* Volume will be created in one container
* You can declare a directory as volume only while creating
* container
* You can't create volume from existing container
* You can share one volume across any no of containers
* Volume will not be included when you update an image
* Map volumes in two ways
* Share host – container
* Share container – container

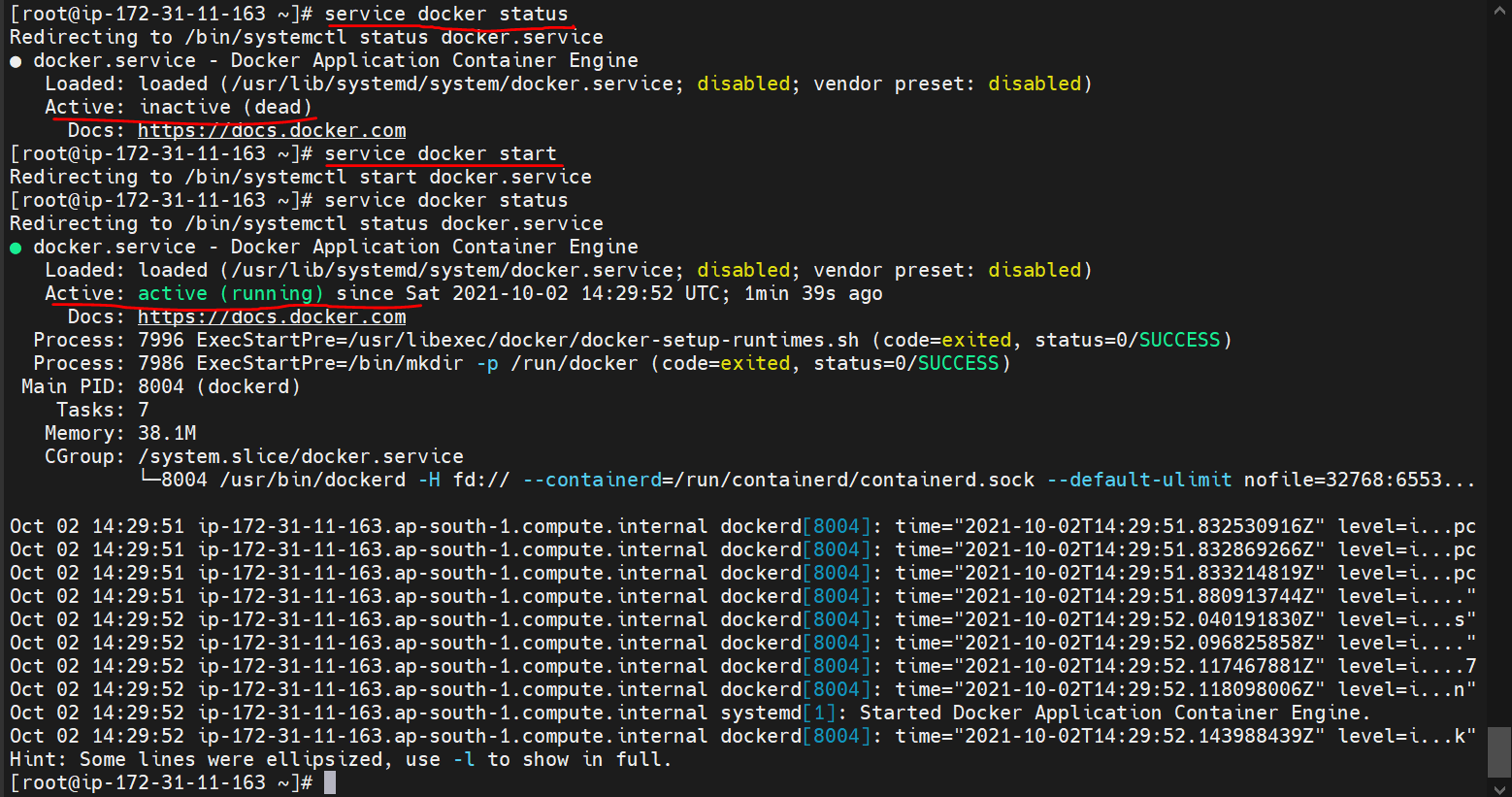




To install docker **yum install docker –y**

To verify the service status **service docker status**

To start the service **service docker start**



**Docker Commands:**To know the images created docker images

To know the current running containers docker ps (process)

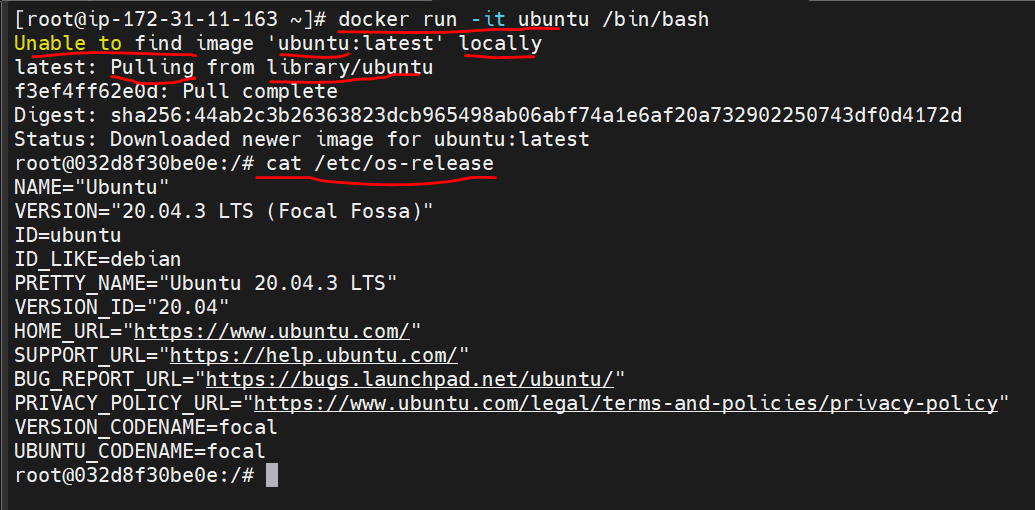
docker ps –a to know all created images

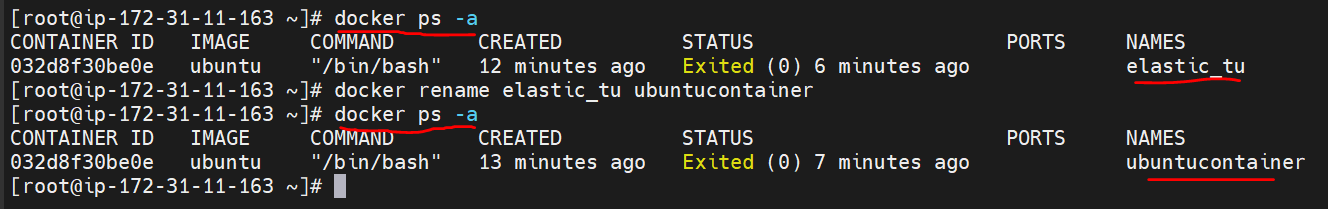
**To create a container:  
docker run –it <dockerImage> /bin/bash  
i – interactive  
t – terminal  
/bin/bash - bash is a shell(cmd window) taken from bin dir under root dir of ubuntu  
Ex:** docker run –it ubuntu /bin/bash  
 docker run –it centos /bin/bash

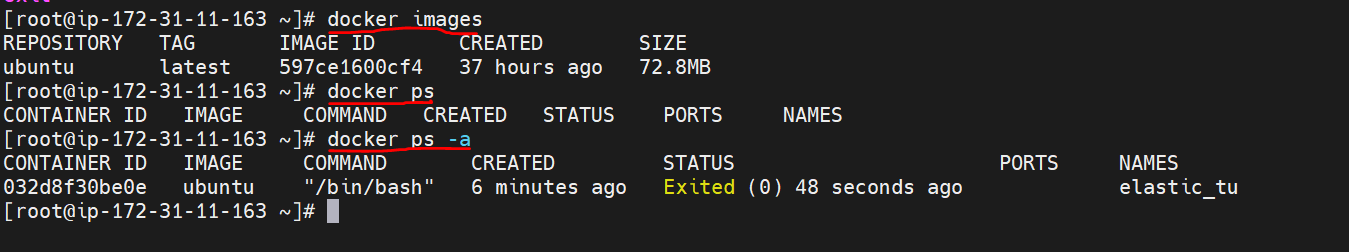
**To search docker images in web, goto dockerhub at** [**docker hub**](https://hub.docker.com/)

**To search docker images in docker hub via linux : docker search <keyword>  
 docker search ubuntu**

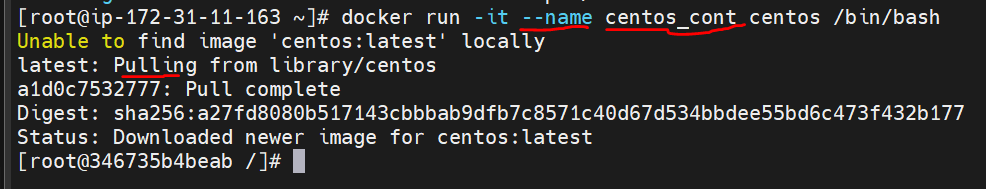
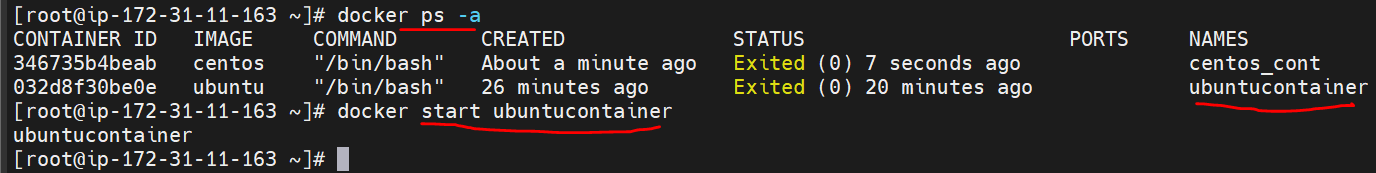
**When we run above command docker will search for the Ubuntu command in local if not exists , it will pull automatically from the docker hub then user will be inside container**

To verify whether we are in container or not --> check the os : cat /etc/os-release  


To rename the docker images : docker rename <old\_name> <new\_name>  
Ex: docker rename elastic\_tu myContainer  




To give a name to container while creating :  
docker run –it - - name <cont\_name> <docker\_image> /bin/bash

EX: docker run –it –name my\_container centos /bin/bash  
 To start a container : docker start <container\_name>

**In UBUNTU replace commands with apt instead of yum :**   
yum update, yum install  
apt update, apt install

**To create a image from the existing container** (mycont)  
docker commit <cont\_name> <image\_name>  
docker commit mycont myimage  
**To create a new container from above image**  
docker run –it –name myNewContainer myimage /bin/bash

**Creating a container by using DockerFile:**

A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.

**\*\*\*\*\*\*\*File name should be Dockerfile**

FROM ubuntu

RUN echo “Hello User” > /tmp/newfile1

Each instruction creates one layer:

* FROM creates a layer from the ubunt Docker image.
* RUN creates a new file under tmp dir

When you run an image and generate a container, you add a new *writable layer* (the “container layer”) on top of the underlying layers. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this writable container layer.

Crate a Dockerfile with all instructions

vi Dockerfile  
add instructions like  
FROM ubuntu

RUN echo “Hello User” > /tmp/newfile1

Then build the Dockerfile

docker build –t <image\_name> <path to Dockerfile>  
here t – test

(Dockerfile is default file, so docker can recognize even without giving the name)

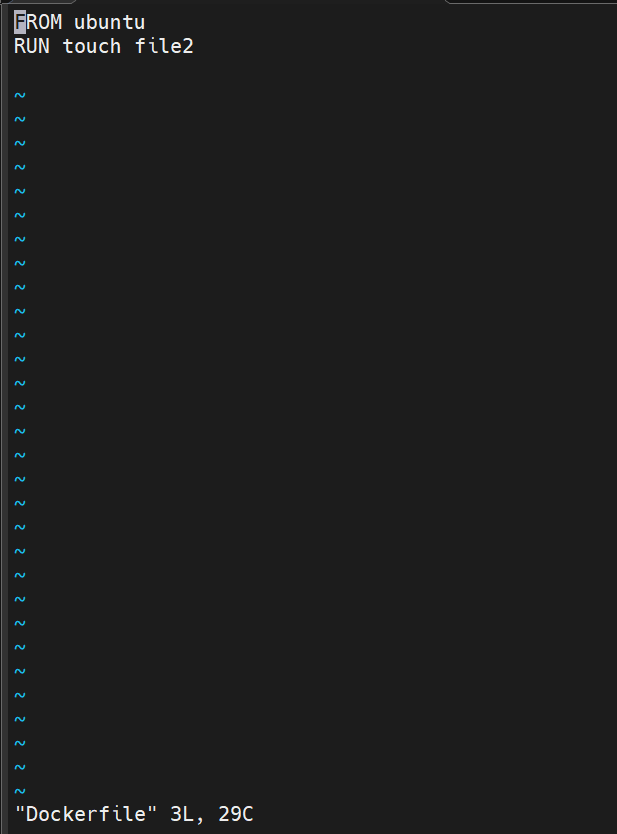
Docker build –t myImage .

Here . means current working dir  
here Docker will recognize the Dockerfile in current working dir

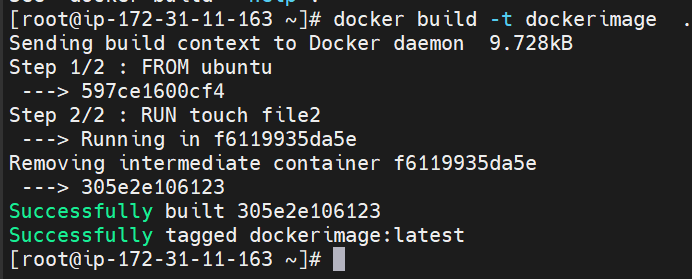
To create a docker image from a otherfile instead of file names Dockerfile

docker build –t <image\_name> <path to the file> -f <filename>

Ex: docker build –t mydockerimage /etc/. mydockerfile

**Dockerfile**:  


**Creating docker image from Dockerfile:**



To know more about Dockerfile keywords refer [Dockerfile keywords](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/)

# Keywords Explained:

By convention, Docker uses all capital letters for Docker keywords, as you have seen in the file above. I highly recommend that you stick to that.

## Comments

In the Dockerfile, the **#**(hash) character represents a line comment. See an example comment in the first line of the Dockerfile above.

## 1. FROM

The FROM keyword specifies the base image on which we want our image to be built on. As you will find out, most Docker images begin with the FROM keyword. The core concept here is to not reinvent the wheel. There are a lot officially supported base images out there, so build on them.

FROM node:10.21.0-alpine as builder

In our Dockerfile, we have specified two base images. The FROM keyword above simply says we will use Node-alpine as the base image on which our image is built on. We tag it with the name builder because we will mainly use it for building our application.

Later in the Dockerfile, we also specify another base image that we will use to serve our application (tagged as server). It is based on the popular NGINX server.

FROM nginx:1.16.0-alpine as server

## 2. WORKDIR

WORKDIR sets the working directory or context inside the image we are building.

In this example, we have set our working directory to the app folder in our image with this statement:

WORKDIR /app

Once we set our context, all subsequent commands are run in the root of the specified context unless otherwise specified. In this case, it is the appdirectory.

## 3. COPY

The COPY keyword is pretty much self-explanatory. It simply copies content from one folder and places it into another. For this, you can either specify the source and destination or specify the files to copy and the destination folder in the image.

From the Dockerfile above, we notice that the COPY keyword appears several times.

The first time we use the COPY keyword is as follows:

COPY package.json package-lock.json ./

Here, we explicitly specify the files to be copied into the image. We copy the package.json and package-lock.json files from our application to the working directory of our image, which is app.

The second time, we use the COPY keyword to copy all source code from our applications root folder into the Docker images’ working directory (app):

COPY . .

Towards the end of the Dockerfile, we also have a COPY keyword. Here, we use it to copy all content from the dist folder that was created from the build stage and dump it into the NGINX server HTML directory:

COPY --from=builder /app/dist/angular-docker /usr/share/nginx/html

In addition to the COPY keyword, docker also provides the ADD keyword. It can be used for copying zip files or content from an HTTP URL.

## 4. RUN

If you want to run a specific Linux command, then the RUN keyword is what you are looking for. In the Dockerfile above, the RUN keyword is used twice.

First to restore npm dependencies in our Docker images’ working context by running the npm install command:

RUN npm install

Later, the RUN keyword is used to make a production build inside the applications image:

RUN npm run build --prod

## 5. EXPOSE

The EXPOSE keyword is used to expose a port inside of the Docker image to the outside world:

EXPOSE 80

In this case, we expose the Docker image on port 80, which we can then use when running the container.

## 6. CMD

The CMD keyword is a bit similar to the RUN keyword discussed above. It is also used in running a Linux command. But unlike the RUN keyword that runs commands for building the image, the CMD keyword is used to run Linux commands when the container is started (i.e. in a live image).

In addition, Docker also provides the ENTRYPOINT keyword that can be used in place of the CMD keyword or together with it.

For a more detailed discussion on the CMD and ENTRYPOINT keywords, check out [this article](https://medium.com/bb-tutorials-and-thoughts/docker-entrypoint-vs-cmd-8934cfdf78f3) by

[Bhargav Bachina](https://medium.com/u/c4c28b790cfe?source=post_page-----57f25d7e5fa6--------------------------------)

.

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

The command above will start the NGINX server and serve the static files from the production build when the Docker image is run.

# Build Image

Finally, we can build our Docker image based on the Dockerfile above. We will simply run the following command:

$ docker build -t mwizak/angular-docker -f Dockerfile .

In this case, we have tagged the Docker image with the name mwizak/angular-docker. In addition, we specify the location of the Dockerfile instructions.

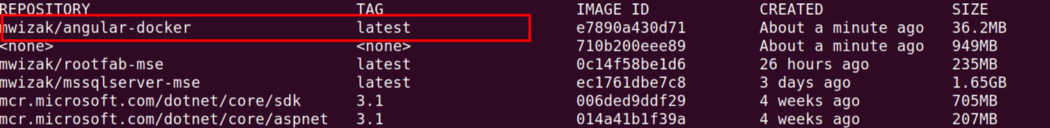
Note: mwizak is the name of my docker registry. You can use it or replace it with any name of your liking.

After the image is finished building, you can check if it is listed in your local Docker images with the command:

$ docker images

You should get output similar to the one below. The newly created image is highlighted in red:

https://miro.medium.com/max/60/1*iRb-S3fTHWTCiKF92X9p1Q.png?q=20



# Run the Image

To run the new image, we will use its image name. Simply run the command below:

$ docker run -ti --rm -p 8080:80 mwizak/angular-docker

We have specified that the Docker image application should run on the HTTP port 8080, which maps to port 80 inside the image.

Go to [http:localhost:8080/](https://betterprogramming.pub/localhost:8080/) in your browser and you will find your Angular application running as a Docker container.

# Push the Image to Registry

This process is not necessary for creating Docker images, but in case you want to easily share your newly created Docker image, you can push it to a Docker registry for people to pull and run on the local PCs.

In my case, I have a registry on Docker Hub, but it could be anywhere (Azure, AWS, etc.).

To push the image to Docker Hub, you first have to log into your account. Run the command:

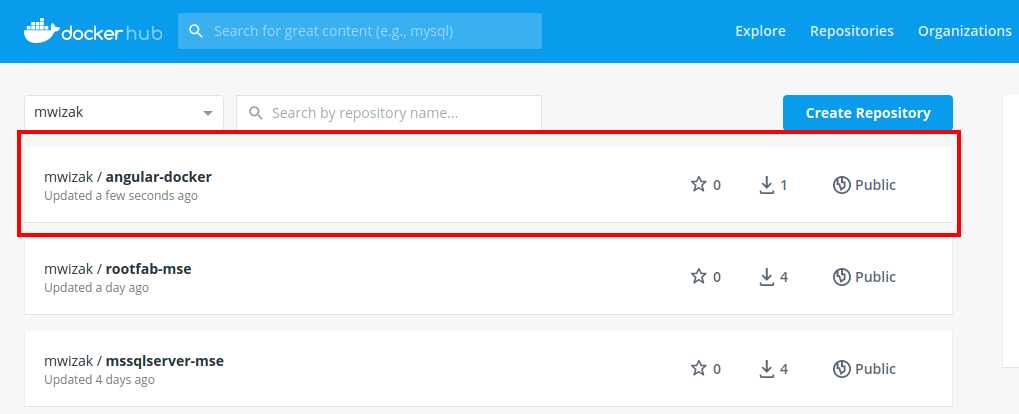
$ docker login

Once you are authenticated, you can push the image to the registry with the following command:

$ docker push mwizak/angular-docker

Where mwizak is the name of your Docker Hub registry account.

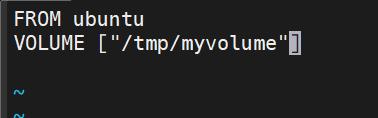
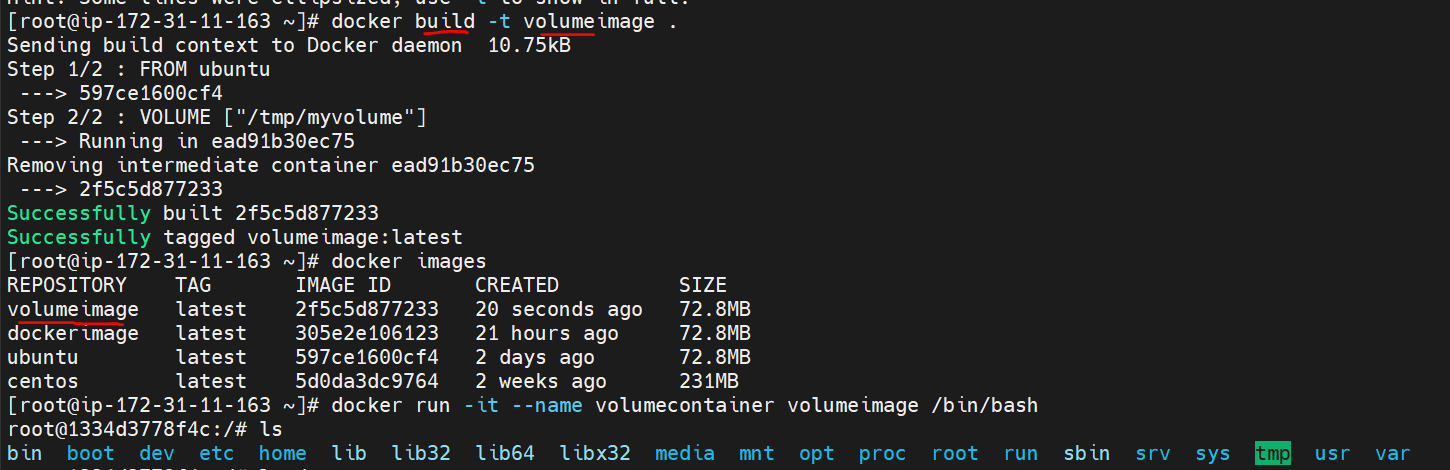
Log into your Docker hub and the image should be listed as below:



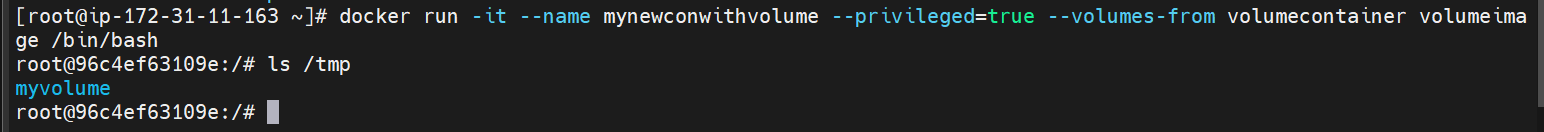
**Volumes:**

* Volume is a directory inside your container
* First declare directory as volume and then share volume
* Even if we stop container, still we can access volume
* Volume will be created in one container
* You can declare a directory as volume only while creating container
* You can’t create volume in an existing container
* You can share one volume across any no. of containers
* Map volumes in two ways  
   1.Share host – container  
   2.Share container – container

To declare a Volume in a container:

In Dockerfile VOLUME [“<dir\_name>”]  
  
Then create docker image by running the docker command:  
docker build –t <image\_name>  


To map the volume from “volumeimage” to a new container:

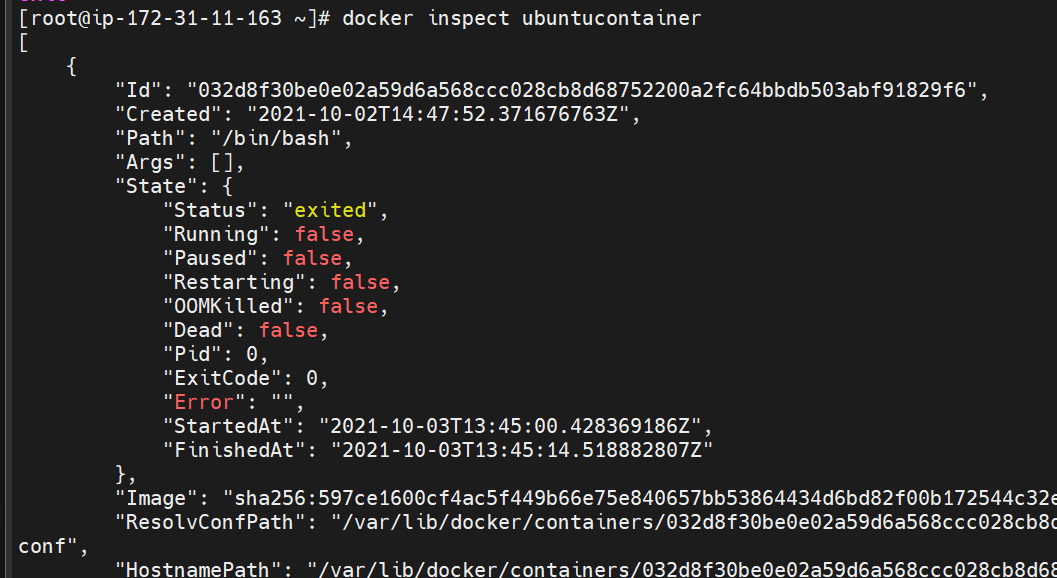
docker run –it - - name volumecont1 - - privileged=true - - volumes-from volumecontainer volumeimage /bin/bash   


To go inside a non running container: docker attach <container\_name>

To create a volume while creating a container :  
docker run –it - - name <new\_container\_name> -v <path to the volume dir> <imagename> /bin/bash  
Ex: docker run –it - - name mynewcon –v /tmp/vlolumedir Ubuntu /bin/bash

**Volume mapping from host t o container:**

docker run –it - - name <container\_name> -v <path to host dir> : <path to con dir> /bin/bash   
Ex: docker run –it - - name myconainer –v /root:/mydir /bin/bash  
\*In this case root dir in host and mydir in container will act like hardlink

To verify the volume in a host or container: docker inspect <container\_name>  
  


Docker uses the following types of volumes and bind mounts to persist data. For this setup, I’m using macOS.

1. Anonymous volumes
2. Named volumes
3. Bind mounts

For this post, we will run a MySQL server and execute some commands. By default, MySQL will store its data files inside /var/lib/mysql directory in the container, and Docker volumes will help us to persist that data.

We have three docker-compose.yml files to demonstrate volumes and bind mounts. To start these files, you will need to use the following command.

docker compose up

Once our container is running, we can use the following commands to create a table inside our container for testing purposes.

# Access the container  
docker exec -it mysql\_db\_1 bash# Connect to MySQL server  
mysql -uroot -proot# Run MySQL commands  
USE test\_db;  
SHOW TABLES;  
CREATE TABLE users (  
 user\_id int NOT NULL AUTO\_INCREMENT,  
 name VARCHAR(20),  
 PRIMARY KEY (user\_id)  
);  
SHOW TABLES;

## 1. Anonymous volumes

If we run the following docker-compose.yml file, an anonymous volume will be created. If we restart our container, the data will be visible, but not after we remove the container. Also, it’s not accessible by other containers. It is helpful if we want to persist data temporarily. These volumes are created inside /var/lib/docker/volume local host directory.

version: '3.8'  
services:  
 db:  
 image: mysql  
 restart: always  
 environment:  
 MYSQL\_ROOT\_PASSWORD: root  
 MYSQL\_DATABASE: test\_db  
 ports:  
 - "3306:3306"  
 volumes:  
 - /var/lib/mysql

As we can see, we don’t have to specify the host directory. We just need to specify the directory inside the container.

If we remove the volume instruction from the docker-compose.yml file, the container will create an anonymous volume by default because it’s specified inside the MySQL [Dockerfile](https://github.com/docker-library/mysql/blob/master/8.0/Dockerfile.debian" \t "_blank). So, the MySQL image ensures that we can still access the data if we don’t provide any volume information.

VOLUME /var/lib/mysql

Now, we have an anonymous volume with a random identifier.

docker volume ls  
DRIVER VOLUME NAME  
local 4e679725b7179e63e8658bc157a1980f320948ab819f271fd5a44fe94c16bf23

Let’s inspect our Docker conatiner.

docker inspect mysql\_db\_1  
.  
.  
."Mounts": [  
 {  
 "Type": "volume",  
 "Name": "4e679725b7179e63e8658bc157a1980f320948ab819f271fd5a44fe94c16bf23",  
 "Source": "/var/lib/docker/volumes/4e679725b7179e63e8658bc157a1980f320948ab819f271fd5a44fe94c16bf23/\_data",  
 "Destination": "/var/lib/mysql",  
 "Driver": "local",  
 "Mode": "",  
 "RW": true,  
 "Propagation": ""  
 }  
 ],  
.  
.  
.

We can use the following [command](https://docs.docker.com/engine/reference/commandline/rm/) to remove the container and its associated anonymous volume.

docker rm -v mysql\_db\_1

If we don’t remove the anonymous volume and the container together, it becomes a dangling volume.

docker rm mysql\_db\_1

We can list and remove all the dangling volumes using the following commands.

docker volume ls -qf dangling=true  
docker volume rm $(docker volume ls -qf dangling=true)

## 2. Named volumes

Named volumes can persist data after we restart or remove a container. Also, it’s accessible by other containers. These volumes are created inside /var/lib/docker/volume local host directory.

version: '3.8'  
services:  
 db:  
 image: mysql  
 restart: always  
 environment:  
 MYSQL\_ROOT\_PASSWORD: root  
 MYSQL\_DATABASE: test\_db  
 ports:  
 - "3306:3306"  
 volumes:  
 - db\_data:/var/lib/mysqlvolumes:  
 db\_data:

Here, the first field is a unique name of the volume on a host machine. The second part is the path in the container.

Moreover, if we remove the container using the following command, we will still have the volume, unlike anonymous volumes.

docker rm -v mysql\_db\_1

## 3. Bind mounts

Bind mounts can persist data after we restart or remove a container. As we can see, named volumes and bind mounts are the same, except the named volumes can be found under a specific host directory, and bind mounts can be any host directory.

version: '3.8'  
services:  
 db:  
 image: mysql  
 restart: always  
 environment:  
 MYSQL\_ROOT\_PASSWORD: root  
 MYSQL\_DATABASE: test\_db  
 ports:  
 - "3306:3306"  
 volumes:  
 - $PWD/data:/var/lib/mysql

Here, we are mounting a host folder. The first part is the path in the host machine. The second part is the path in the container.

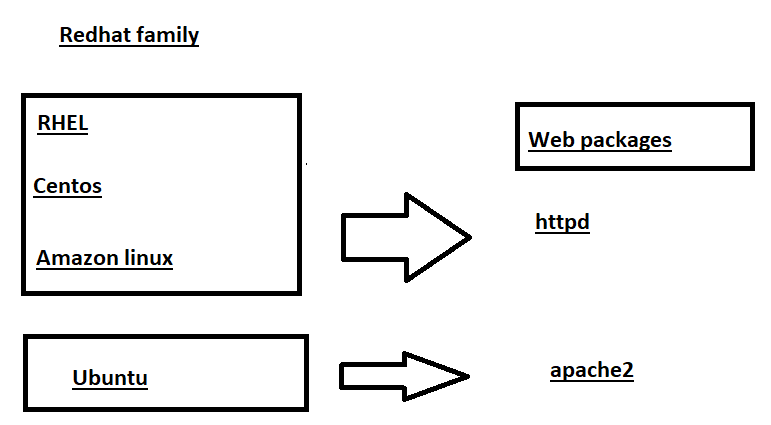
# Commands

Now, let’s list all the available commands for the volume instruction.

docker volume --help  
  
Commands:  
 create Create a volume  
 inspect Display detailed information on one or more volumes  
 ls List volumes  
 prune Remove all unused local volumes  
 rm Remove one or more volumes

We can use these commands to manage anonymous volumes and named volumes.

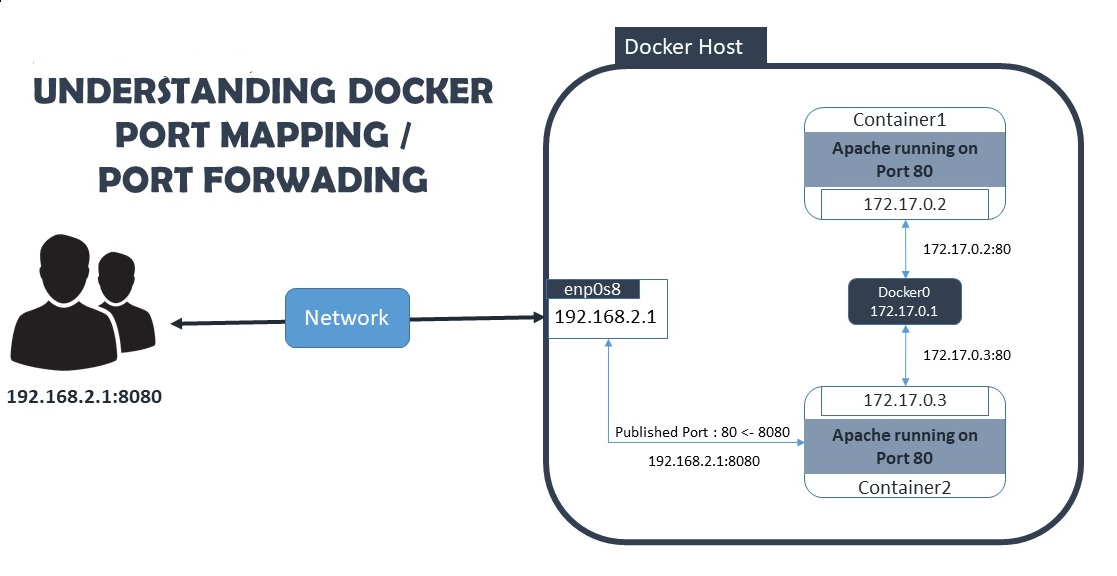
# Creat a volume  
docker volume create test-vol  
# test-vol# Inspect a volume  
docker inspect test-vol  
# [  
# {  
# "CreatedAt": "2021-07-17T07:23:25Z",  
# "Driver": "local",  
# "Labels": {},  
# "Mountpoint": "/var/lib/docker/volumes/test-vol/\_data",  
# "Name": "test-vol",  
# "Options": {},  
# "Scope": "local"  
# }  
# ]# List all volumes  
docker volume create test-vol-2  
docker volume ls  
# DRIVER VOLUME NAME  
# local test-vol  
# local test-vol-2# Remove all volumes  
docker volume prune  
# WARNING! This will remove all local volumes not used by at least one container.  
# Are you sure you want to continue? [y/N] y  
# Deleted Volumes:  
# test-vol  
# test-vol-2# Remove volumes  
docker volume create test-vol-3  
docker volume rm test-vol-3  
# test-vol-3docker volume create test-vol-4  
docker volume create test-vol-5  
docker volume rm test-vol-4 test-vol-5  
# test-vol-4  
# test-vol-5

**Port mapping:**

**\*\*\*In Ubuntu : we use apt or apt-get in place of yum**

**apt-get install apache2 or apt install apache2**

**Public ip is for inter communications  
Private ip is for intra communications**



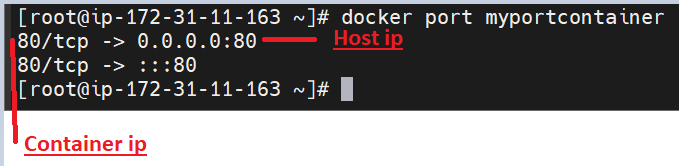
**To map the port**

**docker run –id - - name mywebcontainer –p 80:80 ubuntu /bin/bash**

**-I interactive  
-d detached  
-p port**

We should not change container port num, but we can change host port num if required

To verify which ports we mapped  
docker port <container\_name>



To go inside a container which is created in detached mode (docker run –id - - name …)

docker exec –it <container\_name> /bin/bash  
Ex: docker exec –it mywebcontainer /bin/bash  
After going inside container

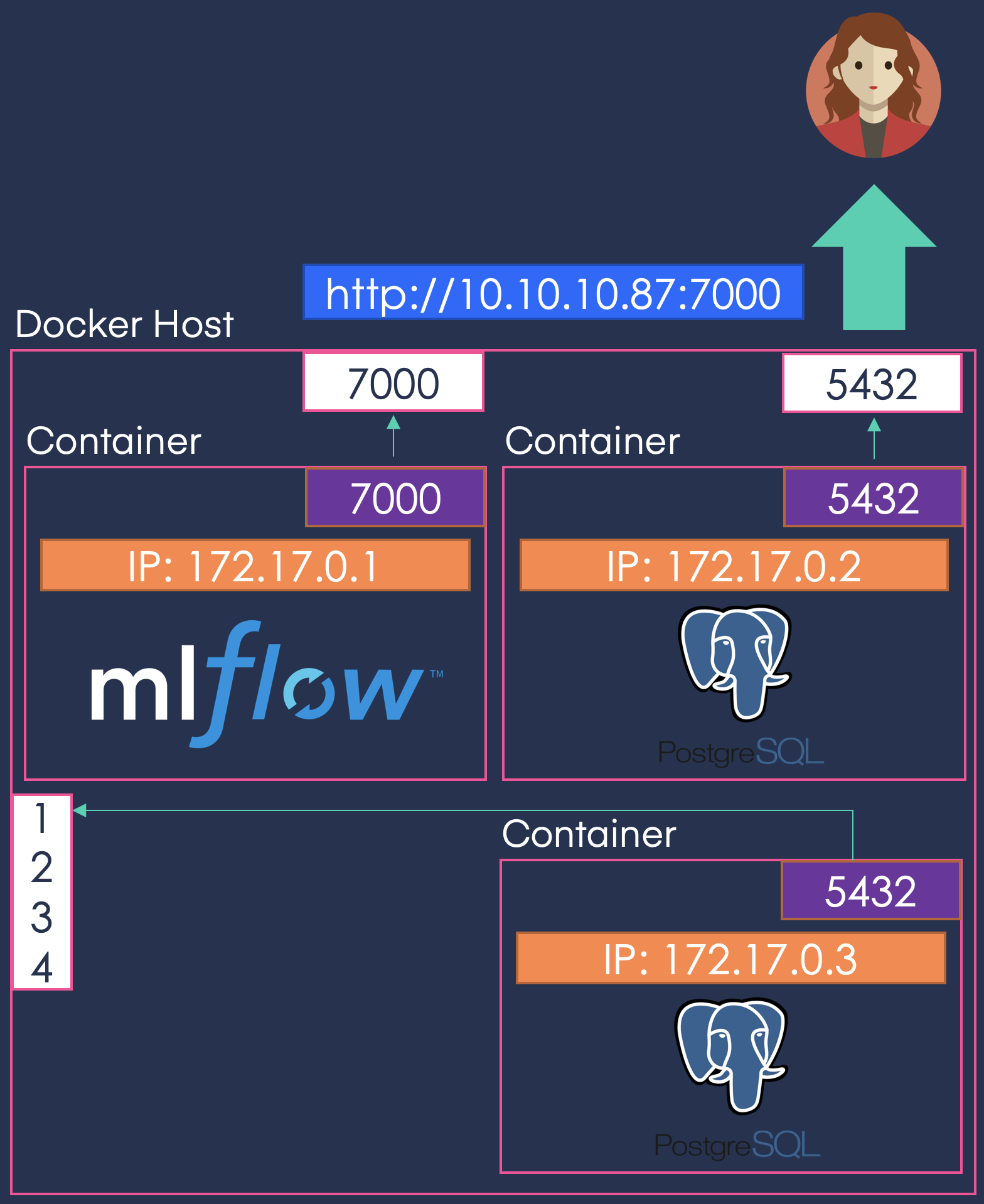
install apache2 in Ubuntu container or httpd web package in other than Ubuntu container

In container create a html file at path /var/www/html

Then service apache2 start

Then you can access the web content of html file by the publice ip/port  
\*\*to access the web content default port is 80  
default html file is index.html

0 – 65535 ports are available in a machine

Make sure that same port is open in host machine  


To enable ports In amazon EC2 instance  
