Docker

Introduction to Docker

Docker is an operating system container management tool that allows you to easily manage and deploy applications by making it easy to package them within operating system containers. Docker's **portability and lightweight** also make it easy to dynamically manage workloads, scaling up or tearing down applications, in near real time. One of the main benefits of using Docker, and container technology, is the portability of applications. It's possible to spin up an application on-prem or in a public cloud environment in a matter of minutes.

Containers vs Virtual Machines

Terms "Containers" and "Virtual Machines" are often used interchangeably, however, this is often a misunderstanding. But, both are just different methods to provide Operating System Virtualization.

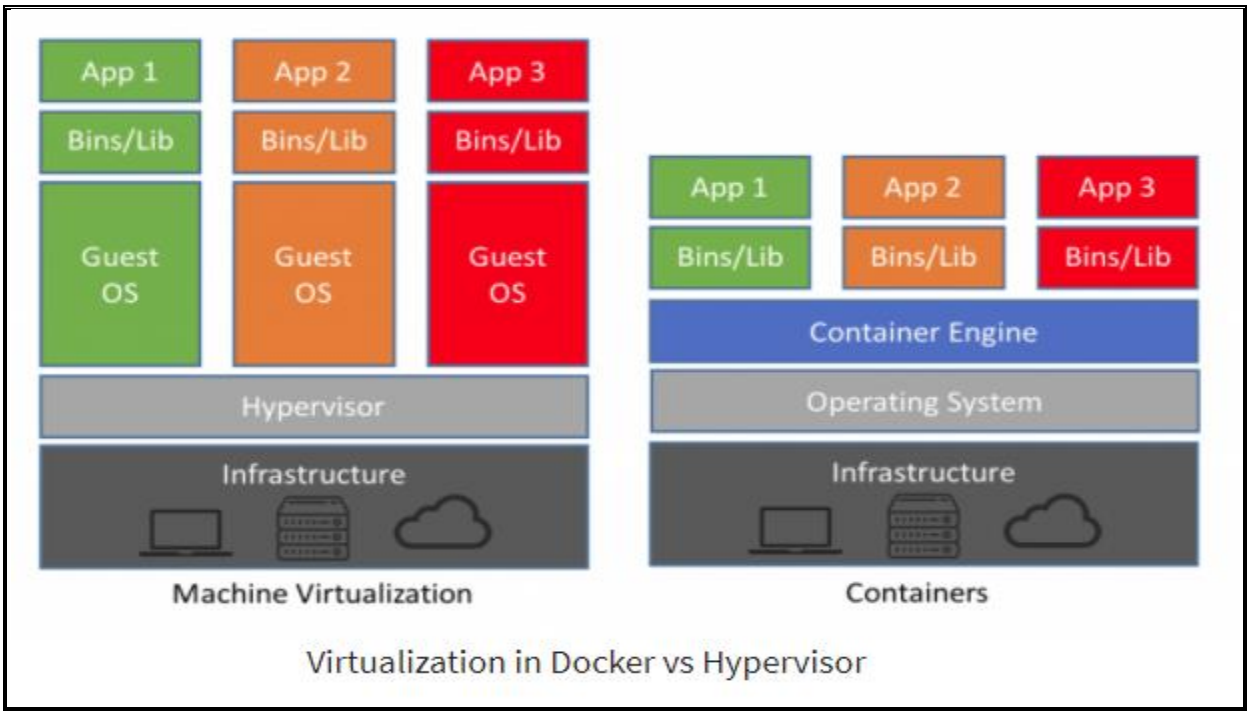
Standard virtual machines generally include a full Operating System, OS Packages and if required, few applications. This is made possible by a Hypervisor which provides hardware virtualization to the virtual machine. This allows for a single server to run many standalone operating systems as virtual guests.

Containers are similar to virtual machines except that Containers are not full operating systems. Containers generally only include the necessary OS Packages and Applications. They do not generally contain a full operating system or hardware virtualization, that's why these are "lightweight".

Virtual Machines are a way to take a physical server and provide a fully functional operating environment that shares those physical resources with other virtual machines.

Whereas, a **Container** is generally used to isolate a running process within a single host to ensure that the isolated processes cannot interact with other processes within that same system. Containers sandbox processes from each other. For now, you can think of a container as a lightweight equivalent of a virtual machine.

Docker enables creating and working with Containers as easy as possible.



What is Docker?

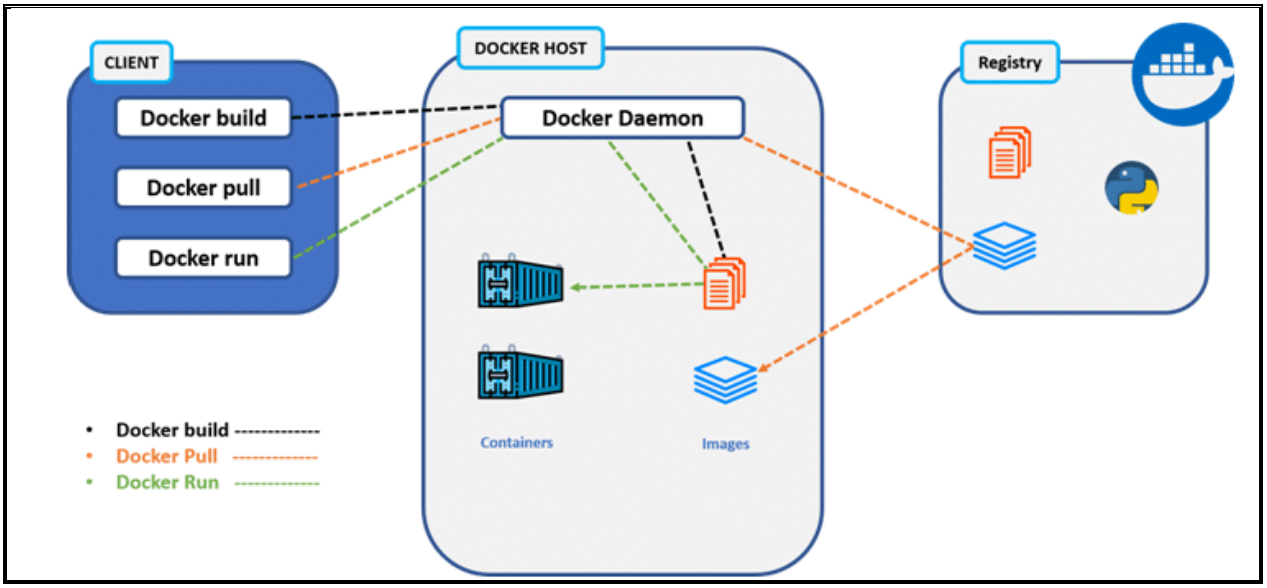
Docker is a software development platform for virtualization with multiple Operating systems running on the same host. It helps to separate infrastructure and applications in order to deliver software quickly. Unlike Hypervisors, which are used for creating VM (Virtual machines), virtualization in Docker is performed on system-level, also called Docker containers.
As you can see the difference in the image below, Docker containers run on top of the host's Operation system. This helps you to improves efficiency and security. Moreover, we can run more containers on the same infrastructure than we can run Virtual machines because containers use fewer resources.

Why use Docker?

- Docker is computer software used for Virtualization in order to have multiple Operating systems running on the same host
- Docker is the client-server type of application which means we have clients who relay to the server
- Docker images are the "source code" for our containers; we use them to build
- Dockerfile has two types of registries 1.) public and 2)private registries
- Containers are the organizational units of Docker volume. In simple terms, an image is a template, and a container is a copy of that template. You can have multiple containers (copies) of the same image.

Docker Architecture

Docker uses client–server architecture. The Docker client consists of Docker build, Docker pull, and Docker run. The client approaches the Docker daemon that further helps in building, running, and distributing Docker containers. Docker client and Docker daemon can be operated on the same system; otherwise, we can connect the Docker client to the remote Docker daemon. Both communicate with each other using the REST API, over UNIX sockets or a network.



The basic architecture in Docker consists of three parts:

- Docker Client
- Docker Host
- Docker Registry

Docker Client

- It is the primary way for many Docker users to interact with Docker.
- It uses command-line utility or other tools that use Docker API to communicate with the Docker daemon.
- A Docker client can communicate with more than one Docker daemon.

Docker Host

In Docker host, we have Docker daemon and Docker objects such as containers and images. First, let's understand the objects on the Docker host, and then we will proceed toward the functioning of the Docker daemon.

- **Docker Objects:**
    - **What is a Docker image?** A Docker image is a type of recipe/template that can be used for creating Docker containers. It includes steps for creating the necessary software.
    - **What is a Docker container?** A type of virtual machine created from the instructions found within the Docker image. It is a running instance of a Docker image that consists of the entire package required to run an application.

- **Docker Daemon:**
    - Docker daemon helps in listening requests for the Docker API and in managing Docker objects such as images, containers, volumes, etc. Daemon issues to build an image based on a user's input and then saves it in the registry.
    - In case we don't want to create an image, then we can simply pull an image from the Docker hub (which might be built by some other user). In case we want to create a running instance of our Docker image, then we need to issue a run command that would create a Docker container.

A Docker daemon can communicate with other daemons to manage Docker services.

Docker Registry

- Docker registry is a repository for Docker images which is used for creating Docker containers.
- We can use a local/private registry or the Docker hub, which is the most popular social example of a Docker repository.

Docker Installation

Step1: Install docker package using yum command

```
yum install docker -y
```

Step 2: Start docker Service and check status

```
Systemctl start docker
```

```
Systemctl status docker
```

```
Systemctl enable docker
```

Step 3: Make configuration changes in /etc/containers/registries.conf and make sure its pointing to docker.io

vi /etc/containers/registries.conf
registries = ['docker.io']

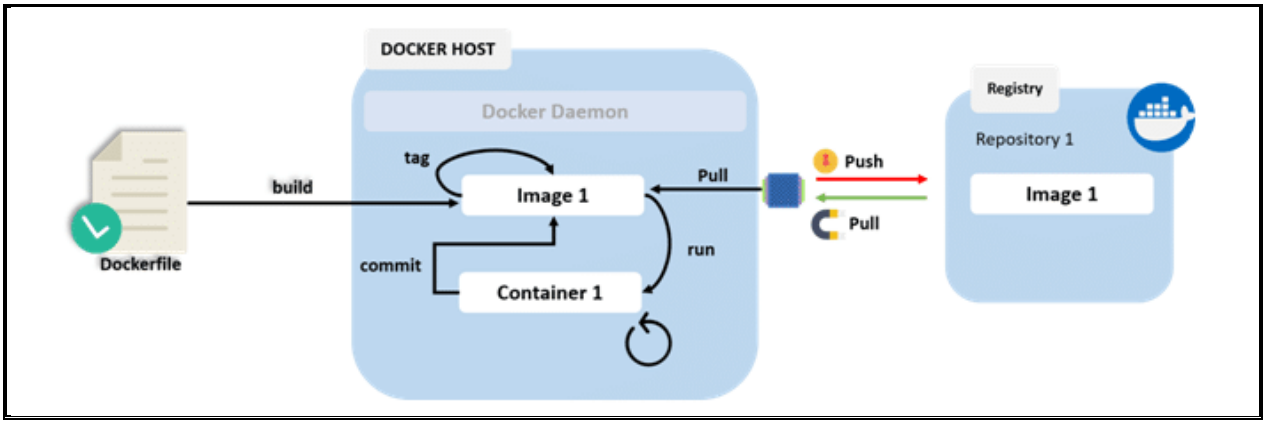
Step 4: Restart docker Service and check status

```
systemctl restart docker
```

```
systemctl status docker
```

Typical Local Workflow

Docker's typical local workflow allows users to create images, pull images, publish images, and run containers.
Let's understand this typical local workflow from the diagram below:



Dockerfile here consists of the configuration and the name of the image pulled from a Docker registry, like a Docker hub. This file basically helps in building an image from it which includes the instructions about container configuration or it can be image pulling from a Docker registry like Docker hub.

Let's understand this process in a little detailed way:

- It basically involves building an image from a Dockerfile that consists of instructions about container configuration or image pulling from a Docker registry, like Docker hub.
- When this image is built in our Docker environment, we should be able to run the image which further creates a container.
- In our container, we can do any operations such as:
    - Stopping the container
    - Starting the container
    - Restarting the container

- These runnable containers can be started, stopped, or restarted just like how we operate a virtual machine or a computer.
- Whatever manual changes are made such as configurations or software installations, these changes in a container can be committed to making a new image, which can further be used for creating a container from it later.
- At last, when we want to share our image with our team or to the world, we can easily push our image into a Docker registry.

One can easily pull this image from the Docker registry using the pull command.

Pulling an Image from the Docker Registry

The easiest way to obtain an image, to build a container from, is to find an already prepared image from Docker's official website.

We can choose from various common software such as MySQL, Node.js, Java, Nginx, or WordPress on the Docker hub as well as from the hundreds of open-source images made by common people across the globe.

For example, if we want to download the image for Nginx, then we can use the pull command:

```
[root@ip-172-31-7-117 ~]# docker pull nginx
Using default tag: latest
Trying to pull repository docker.io/library/nginx ...
latest: Pulling from docker.io/library/nginx
f7ec5a41d630: Pull complete
aa1efa14b3bf: Pull complete
b78b95af9b17: Pull complete
c7d6bca2b8dc: Pull complete
cf16cd8e71e0: Pull complete
0241c68333ef: Pull complete
Digest: sha256:75a55d33ecc73c2a242450a9f1cc858499d468f077ea942867e662c247b5e412
Status: Downloaded newer image for docker.io/nginx:latest
[root@ip-172-31-7-117 ~]#
```

Running an Image

In order to run a Docker image, all we need to do is use the run command followed by our local image name or the one we retrieved from the Docker hub.

Usually, a Docker image requires some added environment variables, which can be specified with the **-e** option. For long-running processes like daemons, we also need to use the **–d** option.

```
[root@ip-172-31-7-117 ~]# docker run -itd --name nginx-continer nginx
68ac4c1dc73b757bc8285b12ce226c6480ef2655c63f68707c45ad1a8ab4d102
[root@ip-172-31-7-117 ~]#
```

To check the container running, use the command below:

docker ps

This command lists all of our running processes, image, the name they are created from, the command that is run, ports that software are listening on, and the name of the container.

```
[root@ip-172-31-7-117 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
68ac4c1dc73b        nginx               "/docker-entrypoin..."   11 seconds ago      Up 11 seconds       80/tcp              nginx-continer
[root@ip-172-31-7-117 ~]#
```

Stopping and Starting Containers

Once we have our Docker container up and running, we can use it by typing the docker stop command with the container name as shown below

```
[root@ip-172-31-7-117 ~]# docker stop nginx-continer
nginx-continer
[root@ip-172-31-7-117 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
[root@ip-172-31-7-117 ~]#
```

As our entire container is written on a disk, in case we want to run our container again from the state in which we shut it down, we can use the start command:

```
[root@ip-172-31-7-117 ~]# docker start nginx-continer
nginx-continer
[root@ip-172-31-7-117 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
68ac4c1dc73b        nginx               "/docker-entrypoin..."   3 minutes ago       Up 2 seconds        80/tcp              nginx-continer
[root@ip-172-31-7-117 ~]#
```

Now, let's see how we can tag an image.

Tagging an Image

Once we have our image up and running, we can tag it with a username, image name, and the version number before we push it into the repository using the docker tag command:

```
[root@ip-172-31-7-117 ~]# docker image ls docker.io/nginx
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
docker.io/nginx     latest              62d49f9bab67        2 weeks ago         133 MB
[root@ip-172-31-7-117 ~]# docker tag docker.io/nginx:latest docker.io/nginx:5.0
[root@ip-172-31-7-117 ~]# docker image ls docker.io/nginx
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
docker.io/nginx     5.0                 62d49f9bab67        2 weeks ago         133 MB
docker.io/nginx     latest              62d49f9bab67        2 weeks ago         133 MB
[root@ip-172-31-7-117 ~]#
```

Now, in this Docker tutorial, let's see how we can push an image to the registry.

Pushing an Image into the Repository

Now, we are ready to push our image into the Docker hub for anyone to use it via a private repository.

- First, go to https://hub.docker.com/ and create a free account
- Next, login to the account using the login command:

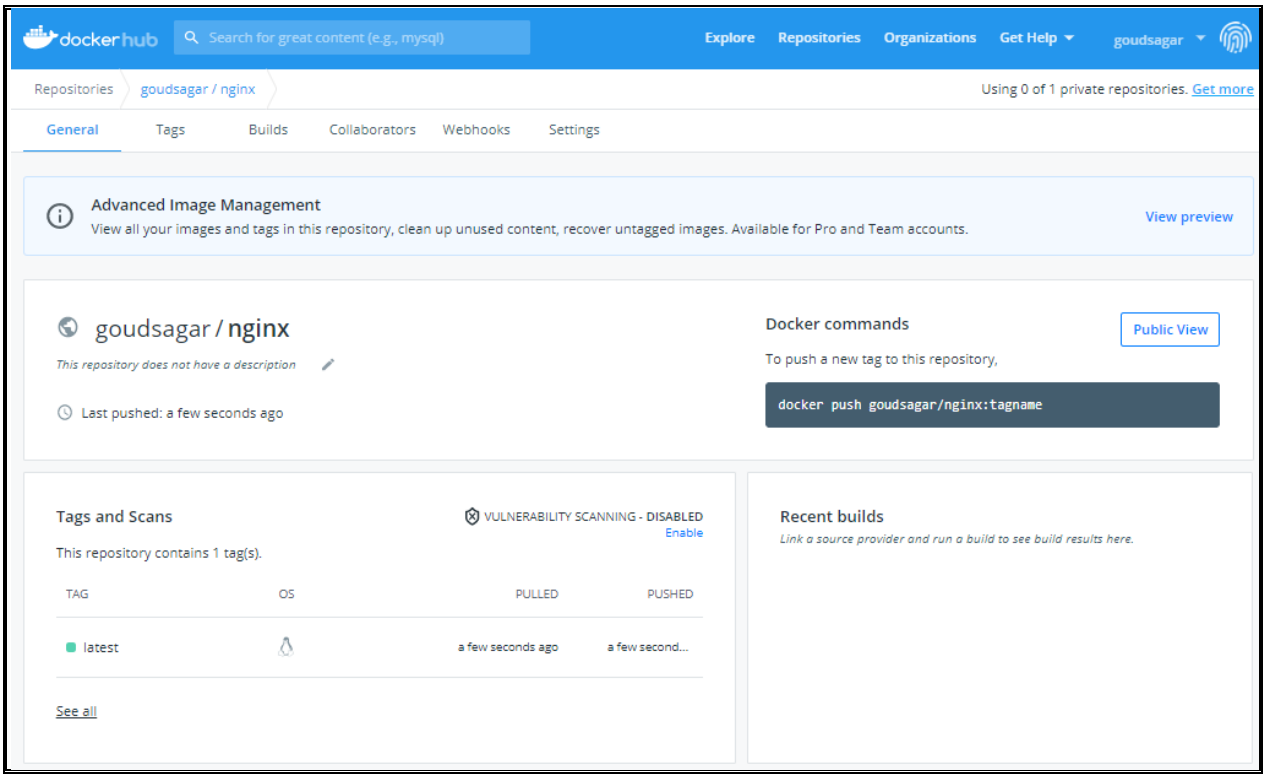
```
[root@ip-172-31-7-117 ~]# docker login docker.io
Login with your Docker ID to push and pull images from Docker Hub.
Username (goudsagar): goudsagar
Password:
Login Succeeded
[root@ip-172-31-7-117 ~]#
[root@ip-172-31-7-117 ~]#
[root@ip-172-31-7-117 ~]#
```

- Input username, password, and email address we are registered with
- Push our image using the push command, with our username, image, and the version name

Within a few minutes, we will receive a message about our repository stating that our repository has been successfully pushed.

```
[root@ip-172-31-7-117 ~]# docker image ls docker.io/goudsagar/nginx
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE
docker.io/goudsagar/nginx   latest              62d49f9bab67        2 weeks ago         133 MB
[root@ip-172-31-7-117 ~]# docker push docker.io/goudsagar/nginx
The push refers to a repository [docker.io/goudsagar/nginx]
64ee8c6d0de0: Mounted from library/nginx
974e9faf62f1: Mounted from library/nginx
15aac1be5f02: Mounted from library/nginx
23c959acc3d0: Mounted from library/nginx
4dc529e519c4: Mounted from library/nginx
7e718b9c0c8c: Mounted from library/nginx
latest: digest: sha256:42bba58a1c5a6e2039af02302ba06ee66c446e9547cbfb0da33f4267638cdb53 size: 1570
[root@ip-172-31-7-117 ~]#
```

When we go back to our Docker hub account, we will see that there is a new repository as shown below:



Docker Commands

Listing Containers

We have already seen, in this Docker tutorial, how to list the running containers using the ps command, but now what we want is to list all the containers, regardless of their state. Well, to do that, all we have to do is add the **-a** option as shown below:

```
[root@ip-172-31-7-117 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
68ac4c1dc73b        nginx               "/docker-entrypoin..."   15 minutes ago      Up 11 minutes       80/tcp              nginx-continer
[root@ip-172-31-7-117 ~]#
```

```
[root@ip-172-31-7-117 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS               NAMES
68ac4c1dc73b        nginx               "/docker-entrypoin..."   14 minutes ago      Up 11 minutes               80/tcp              nginx-continer
37cd43eb7944        webapp-tomcat-1     "/opt/tomcat/bin/c..."   3 weeks ago         Exited (143) 3 weeks ago                        tomcatcontainer1
99501e3b3965        965464b9b49f        "/bin/sh -c 'tar x..."   3 weeks ago         Exited (2) 3 weeks ago                          compassionate_kirch
[root@ip-172-31-7-117 ~]#
```

Now, we can easily distinguish between which container we want to start with and which one to remove.

Removing Containers

After using a container, we would usually want to remove it rather than having it lying around to consume the disk space.

We can use the rm command to remove a container as shown below:

```
[root@ip-172-31-7-117 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
68ac4c1dc73b        nginx               "/docker-entrypoin..."   16 minutes ago      Up 13 minutes       80/tcp              nginx-continer
[root@ip-172-31-7-117 ~]# docker rm -f nginx-continer
nginx-continer
[root@ip-172-31-7-117 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
[root@ip-172-31-7-117 ~]#
```

Removing Images

We already know how to list all the locally cached images using the images command. These cached images can occupy a significant amount of space, so in order to free up some space by removing unwanted images, we can use the rmi command as shown below:

```
[root@ip-172-31-7-117 ~]# docker image ls docker.io/nginx
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
docker.io/nginx     5.0                 62d49f9bab67        2 weeks ago         133 MB
docker.io/nginx     latest              62d49f9bab67        2 weeks ago         133 MB
[root@ip-172-31-7-117 ~]# docker rmi docker.io/nginx:5.0
Untagged: docker.io/nginx:5.0
[root@ip-172-31-7-117 ~]# docker image ls docker.io/nginx
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
docker.io/nginx     latest              62d49f9bab67        2 weeks ago         133 MB
[root@ip-172-31-7-117 ~]#
```

Listing Ports

Knowing which ports are exposed by a container beforehand makes our work a lot easier and faster, e.g., Port 3306 is for accessing a MySQL database and Port 80 is for accessing a web server. Using the port command, as shown below, we can display all the exposed ports:

```
[root@ip-172-31-7-117 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                   NAMES
0ad3b0c96663        nginx               "/docker-entrypoin..."   About a minute ago   Up 59 seconds       0.0.0.0:8080->80/tcp   nginx-container
[root@ip-172-31-7-117 ~]# docker port nginx-container
80/tcp -> 0.0.0.0:8080
[root@ip-172-31-7-117 ~]#
```

Listing Processes

To display processing in a container, we can use the top command in Docker, which is much similar to the top command in Linux.

```
[root@ip-172-31-7-117 ~]# docker top nginx-container
UID                 PID                 PPID                C                   STIME               TTY                 TIME                CMD
root                2738                2724                0                   11:10               pts/1               00:00:00            nginx: master process nginx -g da
emon off;
101                 2787                2738                0                   11:10               pts/1               00:00:00            nginx: worker process
[root@ip-172-31-7-117 ~]#
```

Executing Commands

To execute commands in a running container, we can use the exec command.

For example, if we want to list the contents of the root of the hard drive, we can use the exec command as shown below:

```
[root@ip-172-31-7-117 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                   NAMES
0ad3b0c96663        nginx               "/docker-entrypoin..."   2 minutes ago       Up 2 minutes        0.0.0.0:8080->80/tcp    nginx-container
[root@ip-172-31-7-117 ~]# docker exec nginx-container ls /
bin
boot
dev
docker-entrypoint.d
docker-entrypoint.sh
etc
home
lib
lib64
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
[root@ip-172-31-7-117 ~]#
```

We can gain access to the bash shell if we wish to ssh as root into the container we can use the following command:

```
[root@ip-172-31-7-117 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                   NAMES
0ad3b0c96663        nginx               "/docker-entrypoin..."   4 minutes ago       Up 4 minutes        0.0.0.0:8080->80/tcp    nginx-container
[root@ip-172-31-7-117 ~]# docker exec -it  nginx-container /bin/bash
root@0ad3b0c96663:/# ls
bin  boot  dev  docker-entrypoint.d  docker-entrypoint.sh  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@0ad3b0c96663:/#
```

Dockerfile

A Dockerfile contains all the instructions, e.g., the Linux commands to install and configure the software. Dockerfile creation, as we already know, is the primary way of generating a Docker image. When we use the build command to create an image, it can refer to a Dockerfile available on our path.

Dockerfile Structure

A Dockerfile consists of a set of instructions. Each instruction consists of a command followed by arguments to that command, similar to command line executables. Here is a simple example of a Dockerfile:

```
# The base image
FROM ubuntu:latest

# More instructions here that install software and copy files into the image.
```

```
COPY    /myapp/target/myapp.jar    /myapp/myapp.jar

# The command executed when running a Docker container based on this image.
CMD echo Starting Docker Container
```

Docker Base Image

A Docker image consists of *layers*. Each layer adds something to the final Docker image. Each layer is actually a separate Docker image. Thus, your Docker image consists of one or more underlying Docker images, on top of which you add your own layers.

When you specify your own Docker image via a Dockerfile you typically start with a *Docker base image*. This is another Docker image on top of which you want your own Docker image to be built. The Docker base image you are using may itself consist of multiple layers, and can itself be based on another base image etc., until you get down to the most basic Docker image you can create - a raw Linux container image with no special settings applied.

Dockerfile Commands

- **FROM**

  FROM - specifies the base (parent) image. Alpine version is the minimal docker image based on Alpine Linux which is only 5mb in size.

  The Dockerfile FROM command specifies the base image of your Docker images. If you want to start with a bare Linux image, you can use this FROM command:

```
# The base image
FROM ubuntu:latest
```

- **MAINTAINER**

The Dockerfile MAINTAINER command is simply used to tell who is maintaining this Dockerfile. Here is an example:

```
MAINTAINER   Joe Blocks <joe@blocks.com>
```

The MAINTAINER instruction is not often used though, since that kind of information is also often available in GIT repositories and elsewhere.

- ## RUN

  `RUN` - runs a Linux command. Used to install packages into container, create folders, etc

  The Dockerfile RUN command can execute command line executables within the Docker image. The RUN command is executed during build time of the Docker image, so RUN commands are only executed once.

  The RUN command can be used to install applications within the Docker image, or extract files, or other command line activities which are necessary to run once to prepare the Docker image for execution.

```
RUN yum install httpd -y
```

- ## ENV

  `ENV` - sets environment variable. We can have multiple variables in a single dockerfile.

  The Dockerfile ENV command can set an environment variable inside the Docker image. This environment variable is available for the application that is started inside the Docker image with the CMD command. Here is an example:

```
ENV    MY_VAR   123
```

This example sets the environment variable MY_VAR to the value 123 .

- ## COPY

  `COPY` - copies files and directories to the container.

```
COPY    /myapp/target/myapp.jar    /myapp/myapp.jar
```

This example copies a single file from the Docker host at /myapp/target/myapp.jar to the Docker image at /myapp/myapp.jar . The first argument is the Docker host path (where to copy from) and the second argument is the Docker image path (where to copy to).
You can also copy a directory from the Docker host to the Docker image. Here is an example:

```
COPY    /myapp/config/prod    /myapp/config
```

This example copies the directory /myapp/config/prod from the Docker host to the /myapp/config directory in the Docker image.
You can also copy multiple files into a single directory in the Docker image using the COPY command. Here is an example:

```
COPY    /myapp/config/prod/conf1.cfg   /myapp/config/prod/conf2.cfg
/myapp/config/
```

This example copies the two
files /myapp/config/prod/conf1.cfg and /myapp/conig/prod/conf2.cfg into the Docker image directory /myapp/config/ . Notice how the destination directory has to end with a / (slash) for this to work.

- # EXPOSE

  `EXPOSE` - expose ports

  The Dockerfile EXPOSE instruction opens up network ports in the Docker container to the outside world. For instance, if your Docker container runs a web server, that web server will probably need port 80 open for any client to be able to connect to it.

  Here is an example of opening a network port using the EXPOSE command:

```
EXPOSE   8080
```

You can also set which protocol is allowed to communicate on the opened port. For instance, UDP or TCP. Here is an example of setting the allowed protocol also:

```
EXPOSE   8080/tcp 9999/udp
```

If no protocol is set (after the /) , then the protocol is assumed to be TCP.

- # ENTRPOINT

  `ENTRYPOINT` - provides command and arguments for an executing container.

  The Dockerfile ENTRYPOINT instruction provides an *entrypoint* for Docker containers started from this Docker image.

An *entrypoint* is an application or command that is executed when the Docker container is started up. In that way, ENTRYPOINT works similarly to CMD, with the difference being that using ENTRYPOINT the Docker container is shut down when the application executed by ENTRYPOINT finishes.

Thus, ENTRYPOINT kind of makes your Docker image an executable command itself, which can be started up and which shut down automatically when finished. Here is a Dockerfile ENTRYPOINT example:

```
ENTRYPOINT java -cp /apps/myapp/myapp.jar com.jenkov.myapp.Main
```

This example will execute the Java application main class com.jenkov.myapp.Main when the Docker container is started up, and when the application shuts down, so does the Docker container.

- ## CMD

  `CMD` - provides a command and arguments for an executing container. There can be only one CMD.

  The CMD command specifies the command line command to execute when a Docker container is started up which is based on the Docker image built from this Dockerfile. Here are a few Dockerfile CMD examples:

```
CMD echo Docker container started.
```

This example just prints the text Docker container started when the Docker container is started. The next CMD example runs a Java application:

```
CMD java -cp /myapp/myapp.jar com.jenkov.myapp.MainClass arg1 arg2 arg3
```

- ## VOLUME

  `VOLUME` - create a directory mount point to access and store persistent data.

  The Dockerfile VOLUME instruction creates a directory inside the Docker image which you can later mount a volume (directory) to from the Docker host.
  In other words, you can create a directory inside the docker image, e.g.
  called /data which can later be mounted to a directory, e.g. called /container-

data/container1 in the Docker host. The mounting is done when the container is started up.
Here is an example of defining a volume (mountable directory) in a Dockerfile using the VOLUME instruction:

```
VOLUME    /data
```

- **WORKDIR**

  `WORKDIR` - sets the working directory for the instructions that follow.

  The WORKDIR instruction specifies what the working directory should be inside the Docker image. The working directory will be in effect for all commands following the WORKDIR instruction. Here is an example:

```
WORKDIR    /java/jdk/bin
```

- `LABEL` - provides metadata like maintainer.

The Dockerfile needs metadata, and the LABEL command is what you'd use to create them. After building an image and running a container off it, you can use the **docker inspect** command to find information on the container.

- `ADD` - Copies files and directories to the container. Can unpack compressed files.

Here are a few Dockerfile ADD examples:

```
ADD    myapp.tar    /myapp/
```

This example will extract the given TAR file from the Docker host into the /myapp/ directory inside the Docker image.

Here is another example:

```
ADD    http://jenkov.com/myapp.jar    /myapp/
```

COPY vs. ADD

Both commands serve a similar purpose, to copy files into the image.

- `COPY` - let you copy files and directories from the host.
- `ADD` - does the same. Additionally it lets you use URL location and unzip files into image.

Docker documentation recommends to use COPY command.

ENTRYPOINT vs. CMD

- `CMD` - allows you to set a default command which will be executed only when you run a container without specifying a command. If a Docker container runs with a command, the default command will be ignored.
- `ENTRYPOINT` - allows you to configure a container that will run as an executable. ENTRYPOINT command and parameters are not ignored when Docker container runs with command line parameters.

VOLUME

You declare `VOLUME` in your Dockerfile to denote where your container will write application data. When you run your container using `-v` you can specify its mounting point.

Real time Scenario deploying Application war into tomcat.

Step1: Creating Docker Tomcat Image – Example

First Let us start with Creating a New Directory (workspace) in which we are going to create our Dockerfile and Copy the Web Applications and other configuration files which needs to be shared with the Container.

You can also use the Existing directory but creating a new and Separate directory for your all your container projects are recommended for Clean infrastructure

Creating a Work Space Directory (or) Use the Existing one.

I have created a new directory named "`/apps/docker/DockerTomcat`" Along the way till the end of this article, I would refer this directory as `WORKSPACE` directory

Creating a DockerFile – Docker Tomcat Image

Inside the workspace, we are going to create a Dockerfile with the following content

Note*: Dockerfile must start with 'D' as upper case.

```
FROM centos

RUN mkdir /opt/tomcat/

WORKDIR /opt/tomcat
RUN curl -O http://archive.apache.org/dist/tomcat/tomcat-8/v8.5.40/bin/apache-tomcat-8.5.40.tar.gz
RUN tar xvfz apache*.tar.gz
RUN mv apache-tomcat-8.5.40/* /opt/tomcat/.
RUN yum -y install java
RUN java -version

WORKDIR /opt/tomcat/webapps
RUN curl -O -L https://github.com/AKSarav/SampleWebApp/raw/master/dist/SampleWebApp.war

EXPOSE 8080

CMD ["/opt/tomcat/bin/catalina.sh", "run"]
```

Here

- `MAINTAINER` – Who Crate and manage this container image
- `FROM` – What is the base image, we are going to use to host our container. you can either use a minimal OS image like CentOS, Alpine or you can create your own from the scratch by mentioning SCRATCH as a value to this.
- `RUN` – Commands to Run to make the image(the future container) in the way you want

- `EXPOSE` Do you want your image or application in the image to expose any port to the external world or at least to the host. For example if you are building Apache HTTP server image you can EXPOSE port 80, In our case it is 8080

- `CMD` The Default Command of the container which gets created using this image. Every Container must have a Default Command. the Container would run as long as the Default Command is running.

- `ADD` or `COPY` The files you want to copy into the container from your host.

- `WORKDIR` Define a workspace where the upcoming (or) following set of commands/instructions should be executed in

- Here you can see we have used WORKDIR twice, One is to execute set of commands on the TOMCAT_HOME/CATALINA_HOME another WORKDIR is to download the Sample Application WAR file and deploy the war into Docker Tomcat Container.

Step2: Build the Docker Tomcat Image

On the same WorkSpace Directory where our Dockerfile is residing. Run the following command to build the image.

The Syntax of the Docker Image command is

```
docker build -t [Name Of the Image]  .
```

Here the PERIOD. (DOT) represents the Current working directory which is also part of the syntax

```
[root@ip-172-31-7-117 DockerTomcat]# pwd
/apps/docker/DockerTomcat
[root@ip-172-31-7-117 DockerTomcat]# docker build -t goudsagar/webapp-application-tomcat .
Sending build context to Docker daemon  2.56 kB
Step 1/12 : FROM centos
 ---> 300e315adb2f
Step 2/12 : RUN mkdir /opt/tomcat/
 ---> Using cache
 ---> a51c00b8085d
Step 3/12 : WORKDIR /opt/tomcat
 ---> Using cache
 ---> c349dca5ddba
Step 4/12 : RUN curl -O http://archive.apache.org/dist/tomcat/tomcat-8/v8.5.40/bin/apache-tomcat-8.5.40.tar.gz
 ---> Using cache
 ---> 70fbe54d3b55
Step 5/12 : RUN tar xvfz apache*.tar.gz
 ---> Using cache
 ---> dc52e20ab586
Step 6/12 : RUN mv apache-tomcat-8.5.40/* /opt/tomcat/.
 ---> Using cache
 ---> 7be65870a119
Step 7/12 : RUN yum -y install java
 ---> Using cache
 ---> f3a67fc4db2a
Step 8/12 : RUN java -version
 ---> Using cache
 ---> 50de81905cf0
Step 9/12 : WORKDIR /opt/tomcat/webapps
 ---> Using cache
 ---> cbf9db2739a7
Step 10/12 : RUN curl -O -L https://github.com/AKSarav/SampleWebApp/raw/master/dist/SampleWebApp.war
 ---> Using cache
 ---> c147ce7f64ed
Step 11/12 : EXPOSE 8080
 ---> Using cache
 ---> 8a6656e829b2
Step 12/12 : CMD /opt/tomcat/bin/catalina.sh run
 ---> Using cache
 ---> 85b315e538c5
Successfully built 85b315e538c5
[root@ip-172-31-7-117 DockerTomcat]#
```

But why did I name my image as “goudsagar/webapp-application-tomcat” instead of just webapp-application-tomcat

Here `goudsagar` is my Docker Login Name / Docker User Name.

It is always recommended to Name the image you are creating with your Docker User Name, So that when you are publishing the Image to Docker hub. (A Central Shared Repository) it would be easy and People across the world can just download your image by specifying the same name that you have set.

For Example, If you want to download this image, you can simply use `docker pull goudsagar/webapp-application-tomcat` That’s all.

So when you create your images, Please make sure you use your Docker Login Name.

Step3: Publish or Push the image to DockerHub

Before you start publishing the images to DockerHub. It is necessary that you have to create your Docker Hub (or) Docker Account. Visit hub.docker.com

Once you have set up your username in Docker Hub.

Log in to Docker Hub from the Docker CLI

```
[root@ip-172-31-7-117 ~]# docker login docker.io
Login with your Docker ID to push and pull images from Docker Hub.
Username (goudsagar): goudsagar
Password:
Login Succeeded
[root@ip-172-31-7-117 ~]#
[root@ip-172-31-7-117 ~]#
[root@ip-172-31-7-117 ~]#
```

Once the Login succeeded.

Publish/Push the image you have created to Docker Hub. Based on your Internet speed you can see the Upload gets completed in a couple of minutes.

```
[root@ip-172-31-7-117 DockerTomcat]# docker push goudsagar/webapp-application-tomcat
The push refers to a repository [docker.io/goudsagar/webapp-application-tomcat]
5678910cf837: Mounted from goudsagar/webapp-tomcat-1
7a0183e38e8c: Mounted from goudsagar/webapp-tomcat-1
de6fec936279: Mounted from goudsagar/webapp-tomcat-1
b9fe6a2413cc: Mounted from goudsagar/webapp-tomcat-1
f64134134570: Mounted from goudsagar/webapp-tomcat-1
41936be387a6: Mounted from goudsagar/webapp-tomcat-1
ea3da3c6996a: Mounted from goudsagar/webapp-tomcat-1
2653d992f4ef: Mounted from goudsagar/webapp-tomcat-1
latest: digest: sha256:13fdbca0eeadaec1db4ff8a0f4695d0189183c560e37198a98d61a7b3a7d8c17 size: 1996
[root@ip-172-31-7-117 DockerTomcat]#
```

Once the image is uploaded (or) Pushed, The entire world can reuse your image as we have mentioned earlier using **docker pull**

Step4: Run Docker Tomcat image as a container.

Now the Image is ready and available in Docker Hub. Irrespective of, If the image is available locally or not. You can start the image as a container.

As mentioned earlier, Docker would search for the image in DockerHub if it is not available in local.
So the command given below can even be run at your Host from anywhere in the world. as long as you are connected to the internet. Docker would do the rest.

Docker container run –it –d –name container_name –p container_port: tomcat_port user/image_name

```
[root@ip-172-31-7-117 DockerTomcat]# docker container run -it -d --name webapp-application-tomcat-container  -p 8081:8080 goudsagar/webapp-application-tomcat
013a03ff3fbd464882795ab473ac724db21eb06cd9d5f0397a5c58fc7e22ad42
[root@ip-172-31-7-117 DockerTomcat]#
```

Here

`-it:` to enable Interactive Session/SSH to get into the container at a later point in time

`-d:` Run the container in the background (always recommended)

`--name:` name your container

`goudsagar/webapp-application-tomcat:` the Image used to create this container. the Image instantiated as a container

`-p 8081:8080:` Forwarding the Container port 8080 to Host 8081

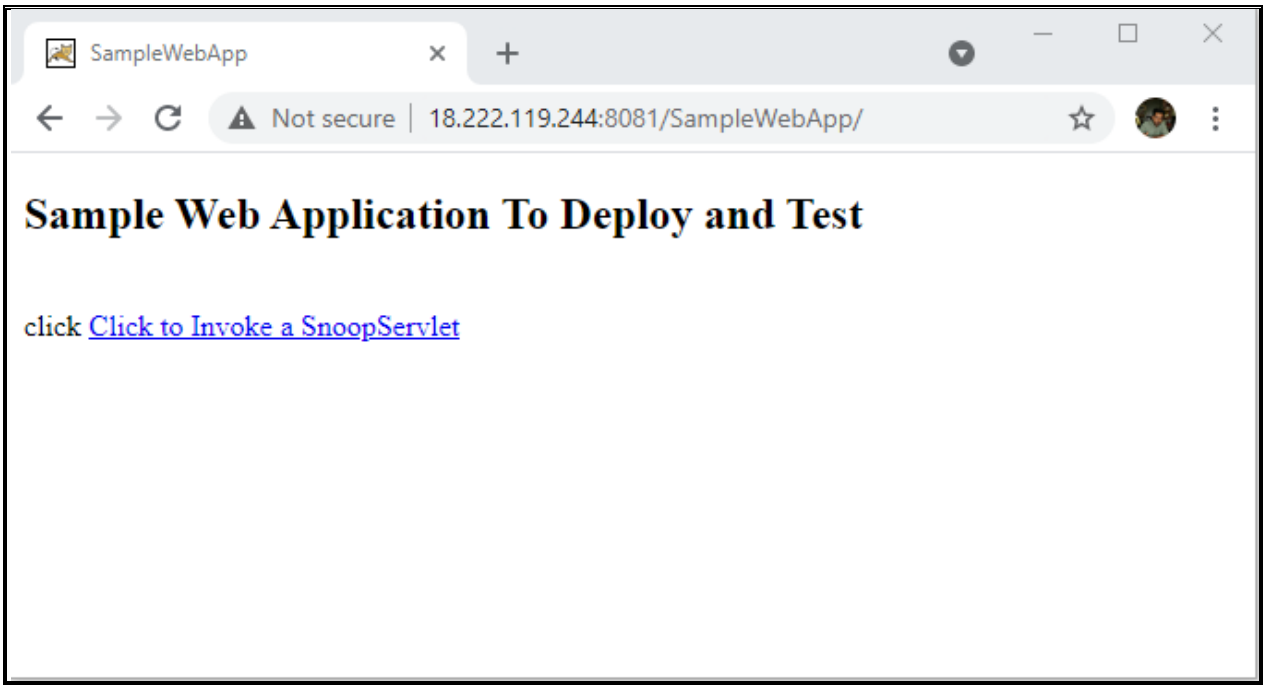
To quickly verify, if your container use `docker ps` command. Then at the end of this post, we will share more commands to manage your container

```
[root@ip-172-31-7-117 DockerTomcat]# docker ps
CONTAINER ID        IMAGE                                COMMAND                  CREATED             STATUS              PORTS                    NAMES
013a03ff3fbd        goudsagar/webapp-application-tomcat  "/opt/tomcat/bin/c..."   2 minutes ago       Up 2 minutes        0.0.0.0:8081->8080/tcp   webapp-application-tomcat-container
[root@ip-172-31-7-117 DockerTomcat]#
```

Step5: Access the Deployed/Built-In SampleWebApp

Since we have port forwarding and redirecting the container 8080 to the Host (mac/windows/Linux) 8081. We can access the Sample Web Application installed inside the tomcat container at the following URL

`http://IP-ADDRESS:8081/SampleWebApp`



Step5: Checking docker logs

```
docker logs -f container_id
```