**Docker Conatiner**

**Image:** Docker images are the foundations of containers. Images are an immutable, ordered collection of root filesystem changes and the corresponding execution parameters for use

**Volume:**A volume is a specially-designated directory within one or more containers that bypasses the Union File System. Volumes are designed to persist data, independent of the container’s life cycle.

**Registry:** A storage and content delivery system used for distributing Docker images.

**Repository:** A collection of related Docker images, often different versions of the same application.

A registry has many repositories which also means a store for images of different types and a repository has many different versions of the same image which are individually versioned with tags.

Containers are **designed for running specific tasks and processes**, not for hosting operating systems.

Process to start in Ubuntu:

1. Install the docker environment- docker.io
2. Check docker version – docker --version
3. Create a Dockerfile and spin-off container from it.

Dockerfile: A Dockerfile defines the process of spinning-off containers. It is a text documet or script made up of instructions/commands on how to build a Docker image. Apart from the script, there are **two types of instructions** that can define the process running in the container:

The [docker build](https://docs.docker.com/engine/reference/commandline/build/) command builds an image from a Dockerfile and a context. The build’s context is the set of files at a specified location PATH or URL. The PATH is a directory on your local filesystem. The URL is a Git repository location.

A context is processed recursively. So, a PATH includes any subdirectories and the URL includes the repository and its submodules. This example shows a build command that uses the MyDockerimages/Dockerfile as context:

$ docker build -t webserver **.**

Sending build context to Docker daemon 3.072kB

Step 1/6 : FROM ubuntu

latest: Pulling from library/ubuntu

The build is run by the Docker daemon, not by the CLI. The first thing a build process does is send the entire context (recursively) to the daemon. In most cases, it’s best to start with an empty directory as context and keep your Dockerfile in that directory. Add only the files needed for building the Dockerfile.

Dockerfile should ideally be at the root of the context this is why we use:

$ docker build **.**

and add –t parameter to name the image.

$ docker build –t [Image Name] **.**

Dockerfile can be placed anywhere and build process uses–f flag to search the path.

$ docker build -f /path/to/a/Dockerfile **.**

**DOCKERFILE FORMAT**

Docker runs instructions in the Dockerfile order and # is for comment.

**FROM Ubuntu:latest**

**MAINTAINER Omeshwar**

**EXPOSE 80 8080**

**ENV TZ=Asia/Calcutta**

**RUN apt-get update && apt-get install -y tzdata && apt-get install apache2 –y&& apt-get install curl**

**RUN mkdir /myvolume**

**VOLUME /myvolume**

**HEALTHCHECK CMD curl --fail http://localhost:80/ || exit 1**

**ENTRYPOINT service apache2 restart && bash**

1. **ARG:** are also known as build-time variables. They are only available from the moment they are 'announced' in the **Dockerfile** till the Image Build which means the running container does not see/access the ARG variables. ARG is the only instruction that can precede FROM.
2. **FROM: The FROM** instruction specifies the [Parent Image](https://docs.docker.com/glossary/#parent_image) from which you are building the image e.g. FROM Ubuntu and it also helps in pulling a valid base image from the public repository like Github.
3. **ENV:** The **ENV** instruction sets the environment variable <key> to the value <value>. The value **will** be in the environment of all “descendant” **Dockerfile** commands and **can** be replaced inline as well. The environment variables set using **ENV will** persist when a container is run from the resulting image.

e.g. **ENV TZ=Asia/Calcutta** where Key (TZ) has value IST Time-zone **Asia/Calcutta**.

1. **RUN:** The RUN instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile.

e.g. apt-update will be re-used during the next build of an Image unless --no-cahe flag.

1. **CMD**: CMD is an instruction that is best to use if you need a default command which users can easily override. If a Dockerfile has multiple CMDs, it only applies the instructions from the last one.
2. **ENTRYPOINT:** is preferred when you want to define a container with a specific executable.

The ENTRYPOINT instruction works very similarly to CMD in that it is used to specify the command executed when the container is started. However, where it differs is that ENTRYPOINT doesn't allow you to override the command.

You can only override an ENTRYPOINT when starting a container unless you add the **--entrypoint** flag.

**CMD** will be overridden when running the container with alternative arguments

**ENTRYPOINT** sets the concrete default application that is used every time a container is created/started using the image so it can be seen as command to run when container starts.

**CMD vs ENTRYPOINT:**

**Lets say we are creating a container using our image “webserver” by passing an argument /bin/bash “docker run –dit webserver /bin/bash”**

**If Dockerfile has CMD Instruction only then arg /bin/bash will override CMD.**

**If Dockerfile has ENTRYPOINT Instruction then arg /bin/bash can not override ENTRYPOINT.**

**Yes “docker run –dit --entrypoint /bin/bash webserver /bin/bash” will help /bin/bash help override ENTRYPOINT.**

1. **LABEL:** The LABEL instruction adds metadata to an image. A LABEL is a key-value pair. Labels included in base or parent images (images in the FROM line) are inherited by your image. If a label already exists but with a different value, the most-recently-applied value overrides any previously-set value.
2. **MAINTAINER:** The MAINTAINER instruction sets the Author field of the generated images. Its deprecated as Label is better and flexible approach.

LABEL <key>=<value> <key>=<value> <key>=<value> ...

LABEL maintainer=[SvenDowideit@home.org.au](mailto:SvenDowideit@home.org.au)

1. **EXPOSE:** EXPOSE <port> [<port>/<protocol>...] The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime and its just a documentation.

e.g. EXPOSE 80/TCP

To actually publish the port when running the container, use the -p flag on docker run to publish and map one or more ports

e.g. docker run –dit –p 80:80 “Image Name” (Host Node Port : Container Port)

1. **VOLUME:** The VOLUME instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.

The VOLUME instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers

The host directory (the mountpoint) is host-dependent to preserve image portability so you can’t mount a host directory from within the Dockerfile. The VOLUME instruction does not support specifying a host-dir parameter. You must specify the mountpoint when you create or run the container.

# Port 80 and 8080 open will help launch webpage on two ports.

# service apache2 restart ensures apache2 services started on image creation itself

# bash is equivalent to CMD [“apache2ctl”, “-d”, “foreground”] to help run docker on the foreground, however if we run as ENTRYPOINT [“apache2ctl”, “-d”, “foreground”] then docker run –dit will run a container in exited mode.

# clean helps remove unnecessary apps installed

A Comprehensive Command:

docker run -it -p abc : xyz -p def : uvw -v ~/myvolume : /test [PUBLISHER]/[IMAGE REPO]:[IMAGE TAG]

-it: This parameter creates an interactive terminal you can use to interact with your container

-p abc : xyz: Expose Port [HOST:CONATINER] for application A

-p def : uvw: Expose Port [HOST:CONATINER] for application B

-v ~/myvolume : /test: Volume tag, This shares the folder ~/mywork on your host machine to /test folder inside your container

Bash: This will create a new Bash session in the container

**Webserver Using Docker Container:**

1. **sudo apt install docker.io**

**sudo usermod -aG docker < user name >**

**exit and login**

**docker info**

**docker run hello-world**

# In case : Got permission denied while trying to connect to the Docker daemon socket , the try below.

**sudo groupadd docker**

**sudo systemctl restart docker**

**sudo chmod 666 /var/run/docker.sock**

1. Create a directory for images e.g. MyDockerimage

omesh@ubuntu:~/MyDockerImages$

1. Crate a Docker File in Image Directory named Dockerfile with the below settings to create a customized webserver using Apache on Ubuntu OS.

FROM ubuntu

EXPOSE 80 8080

ENV TZ=Asia/Calcutta

RUN apt-get update && apt-get install -y tzdata && apt-get install apache2 -y && apt-get clean && apt-get install curl

HEALTHCHECK CMD curl --fail http://localhost:80/ || exit 1

ENTRYPOINT service apache2 restart && bash

# Port 80 and 8080 open will help launch webpage on two ports.

# service apache2 restart ensures apache2 services started on image creation itself

# bash is equivalent to CMD [“apache2ctl”, “-d”, “foreground”] to help run docker on the foreground.

# clean helps remove unnecessary apps installed

# Curl installation helps in running health checkup to make container healthy.

# HEALTHCHECKUP will help check health status of the docker by using default standards.

# ENTRYPOINT will help run apache2 service automatically

1. Create server with name webserver using Dockerfile

**docker build -t webserver .**

1. omesh@ubuntu:~/MyDockerImages$ **docker images**

REPOSITORY TAG IMAGE ID CREATED SIZE

webserver latest fc43895b0c03 7 seconds ago 214MB

ubuntu latest f643c72bc252 5 weeks ago 72.9MB

1. Create container from custom image and check the status with health

**docker run -dit -p 80:80 webserver**

**docker ps -a**

74b823bb5b38 webserver "/bin/sh -c 'service…" 3 seconds ago **Up 2 seconds (health: starting)** 0.0.0.0:80->80/tcp, 8080/tcp romantic\_cartwright

1. Status should be heathy e.g. after 30 secs or retry time ( 30 \* retry numbers)

omesh@ubuntu:~/MyDockerImages$ **docker ps -a**

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

74b823bb5b38 webserver "/bin/sh -c 'service…" About a minute ago **Up About a minute (healthy)** 0.0.0.0:80->80/tcp, 8080/tcp romantic\_cartwright

1. Everything is working fine so lets create our custom image using this healthy docker after chaning the status stop.

**docker stop 74b**

**docker commit -a "Omeshwar" -m "Custom Apache Webserver with Health Check" romantic\_cartwright custom-webserver**

1. Check Images

omesh@ubuntu:~/MyDockerImages$ **docker images**

REPOSITORY TAG IMAGE ID CREATED SIZE

**custom-webserver** latest c3d8bb512fc4 9 seconds ago 216MB

webserver latest fc43895b0c03 11 minutes ago 214MB

ubuntu latest f643c72bc252 5 weeks ago 72.9MB

1. Test the image by creating container and it should turn heathy because curl is installed.

**docker run -dit -p 8080:80 custom-webserver**

**docker ps -a**

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

7909cae1d52a custom-webserver "/bin/sh -c 'service…" 4 seconds **ago Up 3 seconds (health: starting)**  8080/tcp, 0.0.0.0:8080->80/tcp trusting\_pascal

**docker ps -a**

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

7909cae1d52a custom-webserver "/bin/sh -c 'service…" 52 seconds ago **Up 51 seconds (healthy)**  8080/tcp, 0.0.0.0:8080->80/tcp trusting\_pascal

1. We can run this image by pulling from docker repo in future.

omesh@ubuntu:~$ **docker pull omeshwar/omjyokan:customwebserver**