**Understand Docker Features like**

**Docker Build**

The Docker File can be built with the following command −

-docker build

**Syntax**

docker build -t ImageName:TagName dir

**Options**

* -t − is to mention a tag to the image
* ImageName − This is the name you want to give to your image.
* TagName − This is the tag you want to give to your image.
* Dir − The directory where the Docker File is present.

**Example**

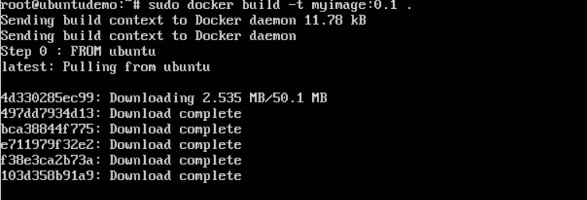
sudo docker build –t myimage:0.1.

Here, myimage is the name we are giving to the Image and 0.1 is the tag number we are giving to our image.

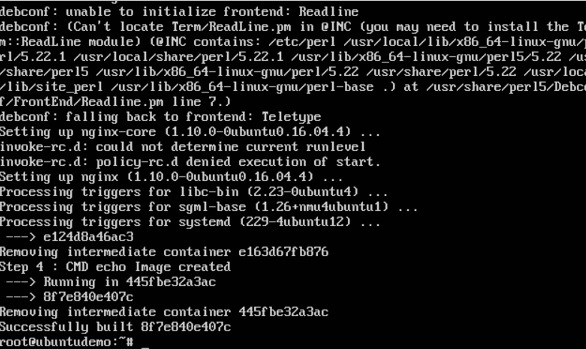
Since the Docker File is in the present working directory, we used "." at the end of the command to signify the present working directory.

**Output**

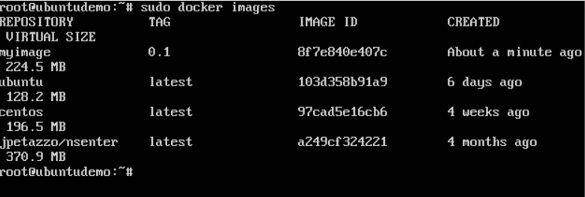
From the output, you will first see that the Ubuntu Image will be downloaded from Docker Hub, because there is no image available locally on the machine.



Finally, when the build is complete, all the necessary commands would have run on the image.



You will then see the successfully built message and the ID of the new Image. When you run the Docker images command, you would then be able to see your new image.



You can now build containers from your new Image.

**Docker Tag:**

Docker tags convey useful information about a specific image version/variant. They are aliases to the ID of your image which often look like this: f1477ec11d12. It’s just a way of referring to your image.

The two most common cases where tags come into play are:

1.When building an image, we use the following command:

**docker build -t username/image\_name:tag\_name .**

We tell the Docker daemon to fetch the Docker file present in the current directory (that’s what the . at the end does). Next, we tell the Docker daemon to build the image and give it the specified tag. If you run docker images, you should see an image whose repository is username/image\_name and tag is tag\_name.

For the public Docker registry, you’re restricted to a two level hierarchy while naming images. For example, your image cannot have the name a/b/c:1. This restriction usually doesn’t exist in private registries.

2. Explicitly tagging an image through the tag command.

**docker tag SOURCE\_IMAGE[:TAG] TARGET\_IMAGE[:TAG]**

This command just creates an alias (a reference) by the name of the TARGET\_IMAGE that refers to the SOURCE\_IMAGE. That’s all it does. It’s like assigning an existing image another name to refer to it. Notice how the tag is specified as optional here as well, by the [:TAG] .

**What happens when you don’t specify a tag?**

This is where the latest tag comes into the picture. Whenever an image is tagged without an explicit tag, it’s given the latest tag by default.

**Scenario 1:**

Suppose the following statement is present in our Dockerfile:

**FROM debian**

Since we didn’t specify any tag, Docker will add the latest tag and try to pull the image debian:latest .

**Scenario 2:**

FROM debian:9.3

Since the tag is explicitly mentioned here, Docker will pull the Debian image tagged 9.3

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

**Docker run, with different modes like**

There are different modes in docker run they are

**Interactive mode**

Popular distributions such as Ubuntu, Fedora, and Debian all have official Docker images available in the hub. Programming languages such as python, php, go or run-times like node and deno all have their official images.

These images do not just run some pre-configured program. These are instead configured to run a shell by default. In case of the operating system images it can be something like sh or bash and in case of the programming languages or run-times, it is usually their default language shell.

As you may have already learned from your previous experiences with computers, shells are interactive programs. An image configured to run such a program is an interactive image. These images require a special **-it** option to be passed in the **container run** command.

As an example, if you run a container using the ubuntu image by executing **docker container run ubuntu** you'll see nothing happens. But if you execute the same command with the -it option, you should land directly on bash inside the Ubuntu container.

docker container run --rm -it ubuntu

# root@dbb1f56b9563:/# **cat /etc/os-release**

# NAME="Ubuntu"

# VERSION="20.04.1 LTS (Focal Fossa)"

# ID=ubuntu

# ID\_LIKE=debian

# PRETTY\_NAME="Ubuntu 20.04.1 LTS"

# VERSION\_ID="20.04"

# HOME\_URL="https://www.ubuntu.com/"

# SUPPORT\_URL="https://help.ubuntu.com/"

# BUG\_REPORT\_URL="https://bugs.launchpad.net/ubuntu/"

# PRIVACY\_POLICY\_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"

As you can see from the output of the cat /etc/os-release command, I am indeed interacting with the bash running inside the Ubuntu container.

The -it option sets the stage for you to interact with any interactive program inside a container. This option is actually two separate options mashed together.

* The -i or --interactive option connects you to the input stream of the container, so that you can send inputs to bash.
* The -t or --tty option makes sure that you get some good formatting and a native terminal-like experience by allocating a pseudo-tty.

**Detach mode**

Detached mode, shown by the option --detach or -d, means that a Docker container runs in the background of your terminal. It does not receive input or display output.

**docker run -d IMAGE**

If you run containers in the background, you can find out their details using docker ps and then reattach your terminal to its input and output.

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

**Docker Volume**

Docker volumes are widely used and useful tools for ensuring data persistence while working in containers.

Docker volumes are file systems mounted on Docker containers to preserve data generated by the running container.

The volumes are stored on the host, independent of the container life cycle. This allows users to back up data and share file systems between containers easily.

**Getting Started With Docker Volumes**

There are different ways to mount a Docker volume while launching a container. Users can decide between the -v and the --mount flags, which are added to the docker run command.

**How to Create a Docker Volume**

To create a Docker Volume use the command:

docker volume create [volume\_name]

Docker automatically creates a directory for the volume on the host under the /var/lib/docker/volume/ path.

For example, to create a volume under the name data, you would run the command:

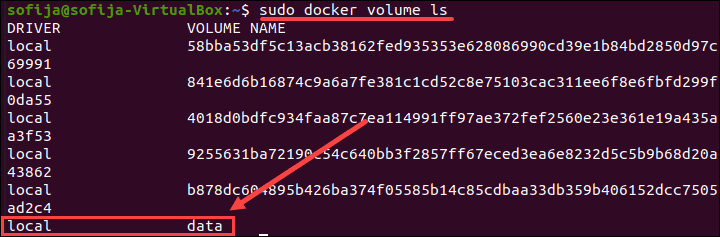
docker volume create data

**List Docker Volumes**

To verify you have successfully created a Docker volume, prompt Docker to list all available volumes with:

docker volume list

The output displays a list of volumes, specifying their location (DRIVER) and their VOLUME NAME. In the image below, you can see the volume data created in the previous section.



**Mounting a Data Volume**

To mount a data volume to a container add the --mount flag to the docker run command. It adds the volume to the specified container, where it stores the data produced inside the virtual environment.

To run a container and mount a data volume to it, follow the basic syntax:

docker run --mount source=[volume\_name],destination=[path\_in\_container] [docker\_image]

Replace [path\_in\_container] with the path where you want to place the data volume in the container. Everything stored in that directory automatically gets saved on the data volume on the host as well.

For example, to launch an Ubuntu container and mount the data volume to it, run:

docker run -it --name=example1 --mount source=data,destination=/data ubuntu

The command instructs Docker to run a container in interactive mode (-it) from the Ubuntu image, under the name example1, while mounting the volume data in the /data directory inside the container.

Then, check to verify the volume was successfully mounted by listing the content of the container:

ls

Find the Docker volume under the name defined while launching the container. In this example, it is data

Find Docker volume in container.

**How to Delete Docker Volumes**

To delete a Docker volume, you need to specify its name.

The basic syntax for removing a Docker volume in the command line is:

docker volume rm [volume\_name]

Docker removes volumes only if they are not in use at the moment. If there is a container with the specified volume, it responds with an error. To proceed, stop and remove the container and then rerun the docker volume rm command.

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

**Docker Expose**

At times, you may need to set out some networking rules to enable smooth interaction between containers in multi-container applications or make your Docker ports accessible by services in the outside world.

You can do this in the following ways:

* Add an EXPOSE instruction in the Dockerfile
* Use the –expose flag at runtime to expose a port
* Use the -p flag or -P flag in the Docker run string to publish a port

**Exposing Docker ports via EXPOSE or –expose**

There are two ways of exposing ports in Docker:

* Including an EXPOSE instruction in the Dockerfile
* Using the –expose flag at runtime

While the two commands are equivalent, they differ in how they work.

Let’s talk about each of them.

**a) Using EXPOSE**

With the EXPOSE rule, you can tell Docker that the container listens on the stated network ports during runtime

Here is an example of how to expose a port in Dockerfile:

EXPOSE 8080

The above line will instruct Docker that the container’s service can be connected to via port 8080.

You can also expose multiple ports:

EXPOSE 80

EXPOSE 8080

EXPOSE 3306

EXPOSE 27018

By default, the EXPOSE keyword specifies that the port listens on TCP protocol.

Here is how to expose on UDP:

EXPOSE 8080/udp

**b) Using –expose**

On the other hand, –expose is a runtime flag that lets you expose a specific port or a range of ports inside the container.

Using the flag is additive, which means that it will expose additional ports together with those stated by the EXPOSE keyword.

Here is an example of how to use the flag in a Docker run string:

Docker run --expose=8080 test

You can also provide a range of ports as an argument:

--expose 2000-3000

**Publishing Docker ports via -P or -p**

There are two ways of publishing ports in Docker:

* Using the -P flag
* Using the -p flag

**a) Using the -P flag**

Using the -P (upper case) flag at runtime lets you publish all exposed ports to random ports on the host interfaces. It’s short for –publish-all.

As earlier mentioned, EXPOSE is usually used as a documentation mechanism; that is, hinting to the container operator about the port(s) providing services.

Docker allows you to add -P at runtime and convert the EXPOSE instructions in the Dockerfile to specific port mapping rules.

**b) Using the -p flag**

Using the -p (lower case) flag at runtime lets you publish a container’s specific port(s) to the Docker host. It’s short for –publish.

It allows you to map a container’s port or a range of ports to the host explicitly—instead of exposing all Docker ports.

Note that irrespective of the EXPOSE instructions in the Dockerfile, using the -p flag at runtime allows you to override them.

Docker run -p <host port> :<container port>imageName

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