## **Module 1: Working with Docker Images**

Every Docker container is based on an image.

Till now we have been using images that were created by others and available in Docker Hub.

Docker can build images automatically by reading the instructions from a Dockerfile

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



## **Module 2: Overview of Dockerfile**

Thee format of Dockerfile is similar to the below syntax:

# Comment

INSTRUCTION arguments

A Dockerfile must start with a FROM` instruction.

The FROM instruction specifies the Base Image from which you are building.

There are multiple INSTRUCTIONS that are available in Dockerfile, some of these include:

* FROM
* RUN
* CMD
* LABEL
* EXPOSE

## **Module 3: COPY vs ADD Instruction**

COPY and ADD are both Dockerfile instructions that serve similar purposes.

They let you copy files from a specific location into a Docker image.

3.1 Difference between COPY and ADD

COPY takes in an src and destination. It only lets you copy in a local file or directory from your host

ADD lets you do that too, but it also supports 2 other sources.

First, you can use a URL instead of a local file/directory.

Secondly, you can extract a tar file from the source directly into the destination.

3.2 Use WGET/CURL wherever possible

Using ADD to fetch packages from remote URLs is strongly discouraged; you should use curl or wget instead.

ADD http://example.com/big.tar.xz /usr/src/things/

RUN tar -xJf /usr/src/things/big.tar.xz -C /usr/src/things

RUN make -C /usr/src/things all

RUN mkdir -p /usr/src/things \

&& curl -SL http://example.com/big.tar.xz \

| tar -xJC /usr/src/things \

&& make -C /usr/src/things all

## 

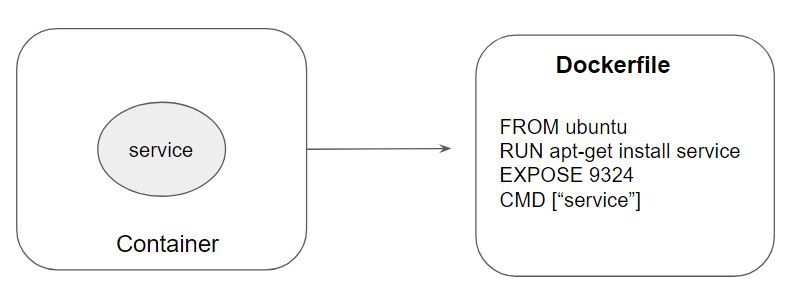
.

## **Module 4: EXPOSE Instruction**

The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime.

The EXPOSE instruction does not actually publish the port.

It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published.



## **Module 5: HEALTHCHECK Instruction**

HEALTHCHECK instruction Docker allows us to tell the platform on how to test that our application is healthy.

When Docker starts a container, it monitors the process that the container runs. If the process ends, the container exits.

That's just a basic check and does not necessarily tell the detail about the application.

We can specify certain options before the CMD operation, these includes:

HEALTHCHECK --interval=5s CMD ping -c 1 172.17.0.2

* --interval=DURATION (default: 30s)
* --timeout=DURATION (default: 30s)
* --start-period=DURATION (default: 0s)
* --retries=N (default: 3)

## **Module 6: ENTRYPOINT Instruction**

The best use for ENTRYPOINT is to set the image’s main command

ENTRYPOINT doesn’t allow you to override the command.

It is important to understand the distinction between CMD and ENTRYPOINT.

Sample Code Snippet:

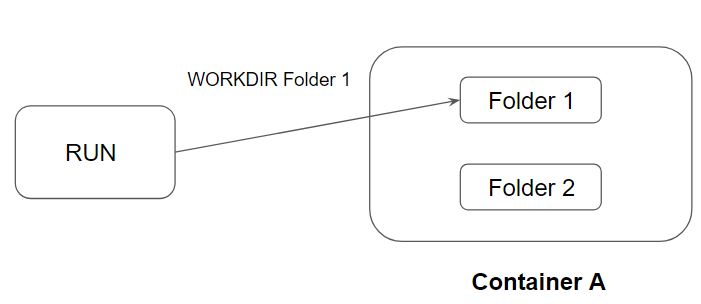
FROM ubuntu

ENTRYPOINT ["top", "-b"]

CMD ["-c"]

## **Module 7: WORKDIR Instruction**

The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile



The WORKDIR instruction can be used multiple times in a Dockerfile

Sample Snippet:

* WORKDIR /a
* WORKDIR b
* WORKDIR c
* RUN pwd

Output = /a/b/c

## **Module 8: ENV Instruction**

The ENV instruction sets the environment variable <key> to the value <value>.

Example Snippet:

ENV NGINX 1.2

RUN curl -SL http://example.com/web-$NGINX.tar.xz

RUN tar -xzvf web-$NGINX.tar.xz

You can use -e, --env, and --env-file flags to set simple environment variables in the container you’re running or overwrite variables that are defined in the Dockerfile of the image you’re running.

Example Snippet:

docker run --env VAR1=value1 --env VAR2=value2 ubuntu env | grep VAR

## 

## **Module 9: Tagging Docker Images**

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: 8f5487c8b942



## **Module 10: Docker Commit**

Whenever you make changes inside the container, it can be useful to commit a container’s file changes or settings into a new image.

By default, the container being committed and its processes will be paused while the image is committed.

Syntax:

docker container commit CONTAINER-ID myimage01

The --change option will apply Dockerfile instructions to the image that is created.

Supported Dockerfile instructions:

CMD | ENTRYPOINT | ENV | EXPOSE

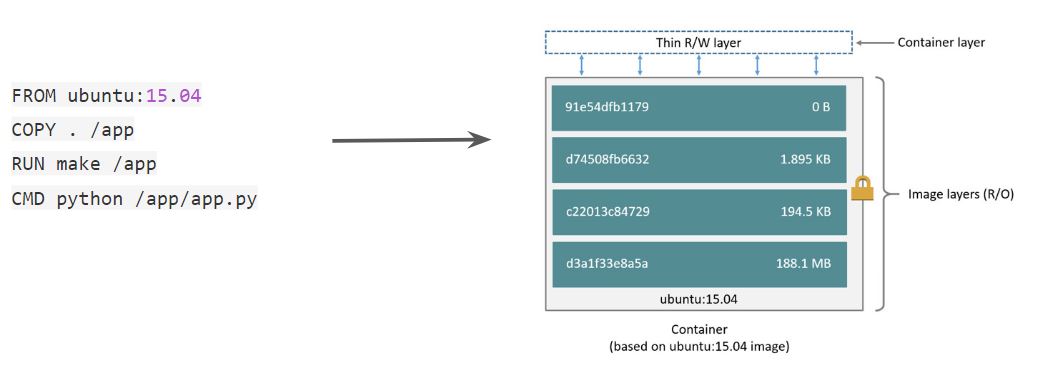
LABEL | ONBUILD | USER | VOLUME | WORKDIR

## 

## **Module 11: Docker Image Layers**

A Docker image is built up from a series of layers.

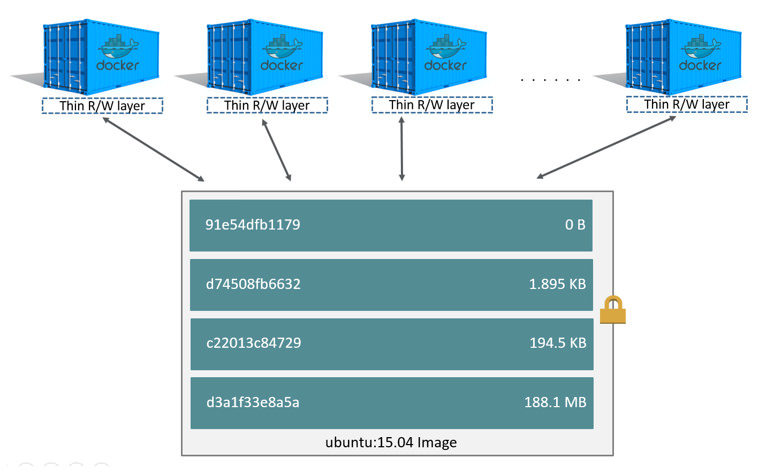
Each layer represents an instruction in the image’s Dockerfile.



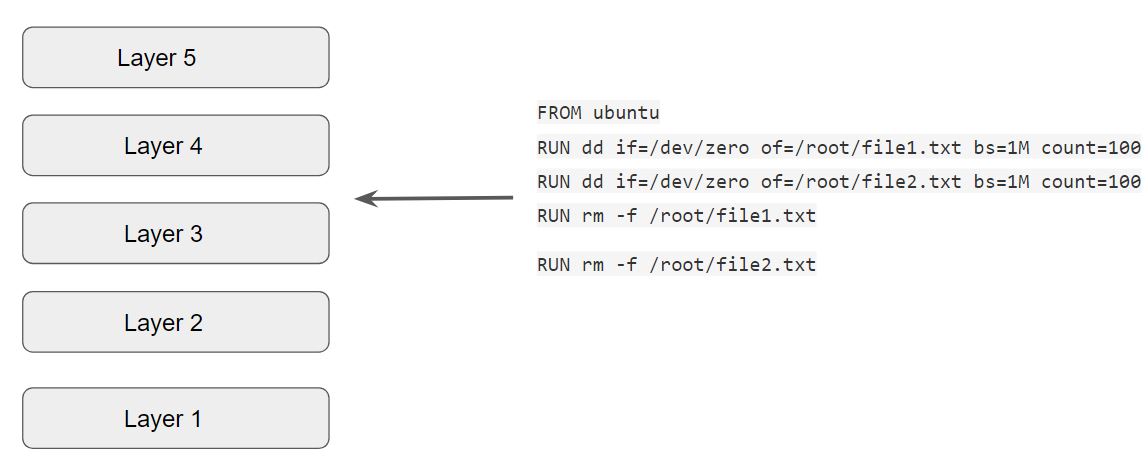
11.1 Difference Between Docker Containers and Docker Image

The major difference between a container and an image is the top writable layer.

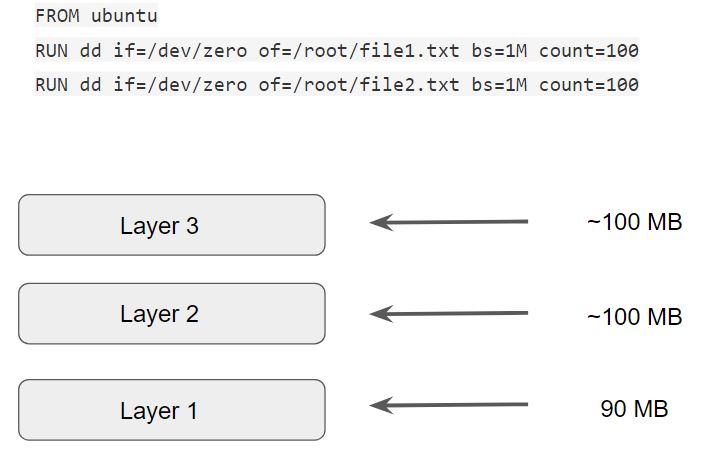
All writes to the container that adds new or modifies existing data are stored in this writable layer.



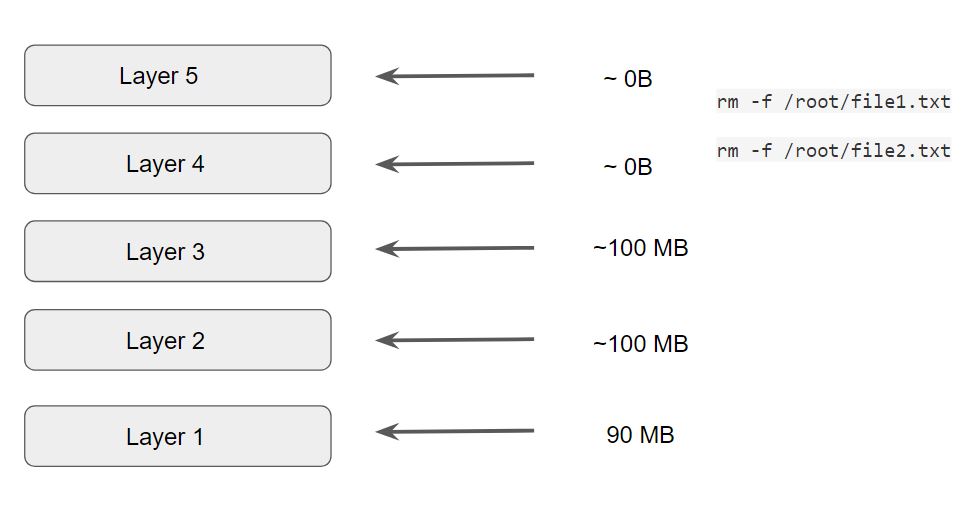
11.2 Understanding Image Layers:



Analyzing the first three image layers:



Analyzing the size of all the layers



## 

## 

## **Module 12: Managing Images with CLI**

Docker CLI can be used to manage various aspects related to Docker Images which include building, removing, saving, tagging, and others.

We should be familiar with the docker image child-commands

Here are some of the sample commands that are available.

* docker image build
* docker image history
* docker image import
* docker image inspect
* docker image load
* docker image ls
* docker image prune
* docker image pull
* docker image push

## 

## **Module 13: Inspecting Docker Images**

A Docker Image contains lots of information, some of these include:

* Creation Date
* Command
* Environment Variables
* Architecture
* OS
* Size

**docker image inspect** command allows us to see all the information associated with a docker image.

## **Module 14: Pruning Images**

**docker image prune** command allows us to clean up unused images.

By default, the above command will only clean up dangling images.

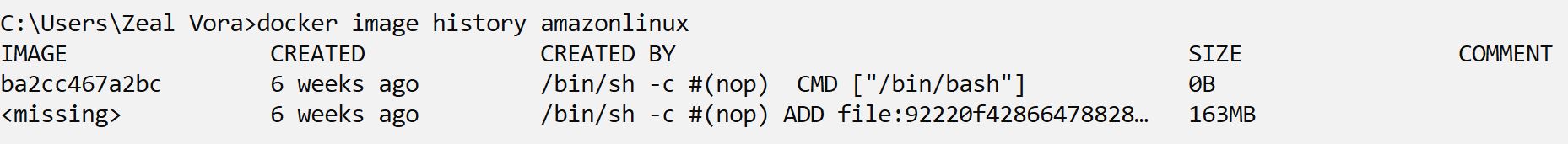
Dangling Images = Image without Tags and Image not referenced by any container

## **Module 15: Flattening Docker Images**

In a generic scenario, the more the layers an image has, the more the size of the image.

Some of the image sizes go from 5GB to 10GB.

Flattening an image to a single layer can help reduce the overall size of the image.



## **Module 16: Docker Registry**

A Registry a stateless, highly scalable server-side application that stores and lets you distribute Docker images.

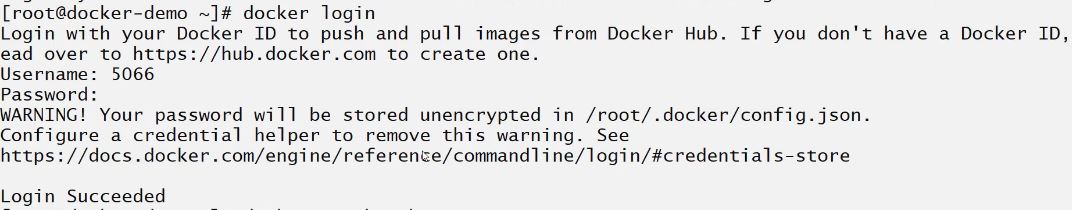
Docker Hub is the simplest example that all of us must have used.

There are various types of registry available, which includes:

* Docker Registry
* Docker Trusted Registry
* Private Repository (AWS ECR)
* Docker Hub

To push the image to a central registry like DockerHub, there are three steps:

1. Authenticate your Docker client to the Docker Registry



1. Tag Docker Image with Registry Repository and optional image tag.

docker tag busybox 5066/mydemo:v1

1. Push Image using docker push command:

docker push 5066/mydemo:v1

## 

## **Module 17: Applying Filters for Docker Images**

|  |  |
| --- | --- |
| **Description** | **Command** |
| Search for Busybox image | docker search busybox |
| Search for Busybox image with Max Result of 5 | docker search busybox --limit 5 |
| Filter only official images | docker search --filter is-official=true nginx |

## **Module 18: Moving Images Across Hosts**

Example Use-Case:

James has created an application based on Docker. He has the image file in his laptop.

He wants to send the image to Matthew over email.



The **docker save** command will save one or more images to a tar archive

Example Snippet:

docker save busybox > busybox.tar

The **docker load** command will load an image from a tar archive

Example Snippet:

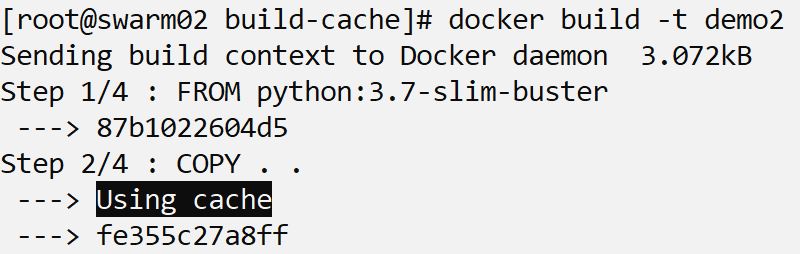
docker load < busybox.tar

## **Module 19: Build Cache**

Docker creates container images using layers.

Each command that is found in a Dockerfile creates a new layer.

Docker uses a layer cache to optimize the process of building Docker images and make it faster.



If the cache can’t be used for a particular layer, all subsequent layers won’t be loaded from the cache.

