DOCKER BEGINNER TUTORIALS

1. Docker Basics:

Needs for docker:

1. Matrix from Hell (Managing all the versions of your libraries on your OS.

2. Setting up a new environment for new employees is very difficult as each developer had to set their own environment up and it took running hundreds of commands with correct configurations and versions.

3. Some developers are comfortable with some OS and the system running smoothly on various OS systems like (Linux, Windows etc) was a big task to handle.

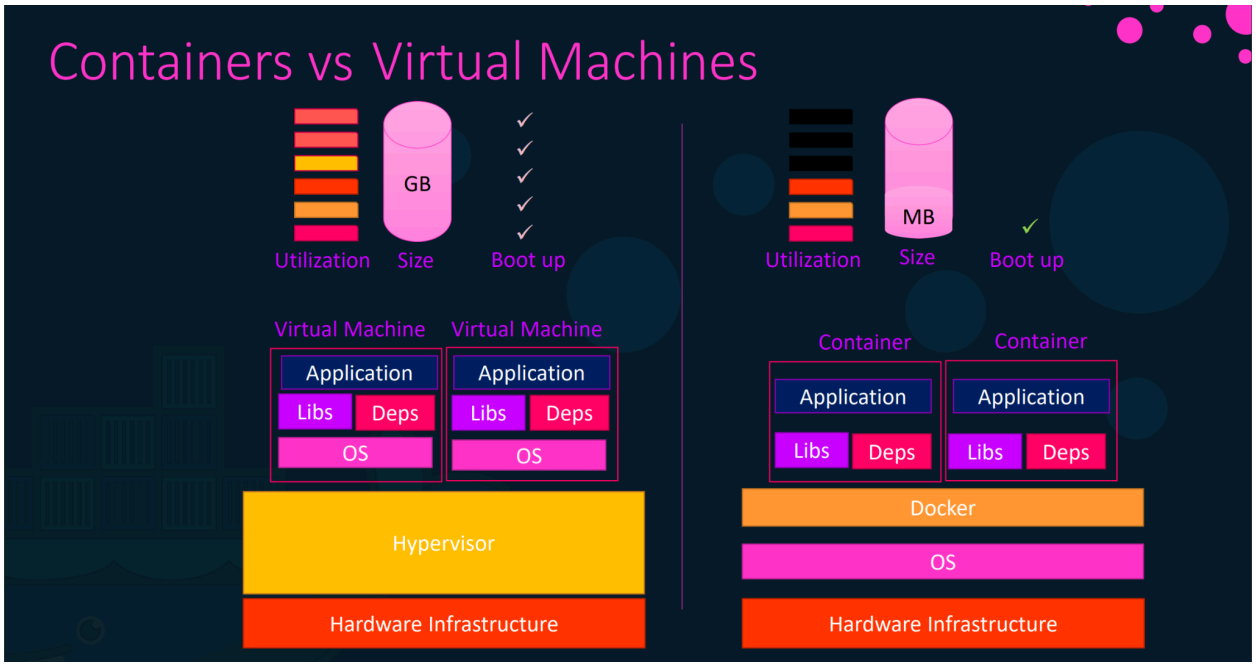
Docker containers run on the underlying OS so if you're running Docker on a Linux System you can create containers running Ubuntu, Debian, Fedora, Arch etc but not a distribution of Windows on the container that is running on a Linux System.

Docker is a LCX type of Container.

Docker is not meant to virtualize and run different operating systems and kernels on the same hardware. The main purpose of Docker is to package and containerize applications and to ship them and to run them anywhere, anytime as many times as you want.

Benefits of Docker against VM:

1. Docker utilizes less resources, it uses less space and it takes a lot less time to boot up.
2. It has less isolation between resources.



DOCKER COMMANDS:

```
$ docker version
```

After installing docker you can get the docker version using this command.

```
$ docker run {image_name}
```

It will run the docker container from an image and it will find the image on the host and if it cannot find it on the host it will go to the docker hub and pull it down. But it is only done one time, for the subsequent executions the same image will be reused.

For example:

```
$ docker run nginx
```

```
$ docker run --name {container_name} {image_name}
```

This command will create the container based on the provided image and name the container what we have provided.

For example:

```
$ docker run --name database1 mysql
```

This command will create a container named database1 with mysql image.

```
$ docker ps
```

Docker PS command lists all the running containers and some basic information such as the container ID, the name of the image we use to run the container, the current status and the name of the container. Each container gets a random ID and a name created for it.

To see all the containers running or not use the -a option. This outputs all running as well as previously stopped or exited containers.

For example:

```
$ docker ps -a
```

```
$ docker stop {container_name}
```

To stop a running container use the docker stop command, but you must provide either the container ID or the container name in the stop command. If not sure use the 'docker ps' command to get it. On success, you will see the name printed out and running 'docker ps' will not show the container running anymore. However running 'docker ps -a' will show the container status and that is in exited state a few seconds ago.

For example:

```
$ docker stop container_a
container_a
```

```
$ docker rm {container_name}
```

Now what if we don't want this container lying around consuming space? We can get rid of it using 'docker rm {container_name}'. After running it the container will be removed and you can check it using 'docker ps -a' command.

For example:

```
$ docker rm container_a
container_a
```

```
$ docker images
```

You can list all the docker images in the system using this command.

```
$ docker rmi {image_name}
```

You can remove the image from your system using this command. But be careful to delete all the dependent containers on this image before deleting this image.

```
$ docker pull {image_name}
```

Earlier we used 'docker run {image_name} command to run the container from an image. If the image was not present in the host then it was pulled from docker hub, but what if you just wanted to pull the image and not run a container on it directly? You can do it using the 'docker pull {image_name}' command.

For example:

```
$ docker pull nginx
```

*** Note:**

If you run 'docker run ubuntu' command it will execute and create a container and directly go into exited state as unlike virtual machines the purpose of the containers is to run a process inside them (an application, database, configuration). They are not supposed to run an OS and hence as soon as the container is created it exits and you will find it in the exited state using the 'docker ps -a' command.

You can use ubuntu to create the base image of your container and then run applications on it.

```
$ docker run ubuntu sleep 5
```

This will run a container and make it sleep for 5 seconds so it won't get exited for 5 seconds while it's asleep. It is called appending a command, we gave the ubuntu system a command to sleep for 5 seconds after running and so it executes that before exiting.

```
$ docker exec {container_name}
{command_to_be_executed_on_container}
```

This command can be used to run a command on a running container.

For example:

```
$ docker run ubuntu sleep 100
```

```
root@fkyounvidia:/home/gaurav# docker ps
CONTAINER ID   IMAGE     COMMAND       CREATED          STATUS          PORTS     NAMES
4499884644f3   ubuntu    "sleep 100"   15 seconds ago   Up 14 seconds             cranky_swartz
root@fkyounvidia:/home/gaurav# docker exec cranky_swartz cat /etc/hosts
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2      4499884644f3
```

The above command just executed the 'cat /etc/hosts' command inside the running ubuntu container.

```
$ docker run myspringbootapp
...
...
...
...
Running the SpringBoot Application {MyDemoApplication.java}
on port 8080
```

This command is to show that when you run an application inside the container you are able to see the output of the service you are running. You won't be able to do anything else on this console other than view the output until the Docker Container stops. It won't respond to your inputs. Press Ctrl+C to stop the container and the application to get back to your terminal.

```
$ docker run -d myspringbootapp
```

This command will run the service in DETACH (-d) mode and you can use the terminal and check the running container using 'docker ps' command.

```
$ docker attach {container_id}
```

This command will let you attach back to your service.

For example:

```
$ docker run -d myspringbootapp
```

```
$ docker ps
```

You can use the ps command to see that the container is running in the detached mode.

```
$ docker attach 0342dfnnf
...
...
...
...
Running the SpringBoot Application {MyDemoApplication.java}
on port 8080
```

You can also run

```
$ docker attach 034
...
...
...
...
Running the SpringBoot Application {MyDemoApplication.java}
on port 8080
```

As the container_id starting with 034 is unique, docker will identify the container and you don't have to specify the whole name.

2. Docker Run Commands:

```
$ docker run redis
```

This will run the container based on the latest redis image.
Docker considers latest to be the default tag.
If you want to specify a particular version of the redis image you can do so by using:

```
$ docker run redis:4.0
```

This is called TAG.

```
$ ./app.sh

Please enter your name: Gaurav

Hello and Welcome Gaurav!
```

If you dockerize this application and run it as a docker container, it wouldn't wait for the prompt. It just prints whatever the application is supposed to print on standard out. That is because by default, the Docker container does not listen to standard input.

```
$ docker run simpleapp

Hello and Welcome  !
```

Even though you're attached to its console, it is not able to read any input from you. It doesn't have a terminal to read inputs from. it runs in non-interactive mode. If you'd like to provide your input, you must map the standard input of your host to the Docker container using the '-i' parameter. The '-i' parameter is for interactive mode.

```
$ docker run -i simpleapp
Gaurav

Hello and Welcome Gaurav!
```

And when I input my name, it prints the expected output. But there is something still missing from this. The prompt, when we run the app, at first, it asked us for our name. But when dockerized, that prompt is missing, even though it seems to

have accepted my input. That is because the application prompt is not on the terminal, and we have not attached to the container's terminal. For this use '-t' parameter as well. The '-t' stands for psuedo terminal.

```
$ docker run -it simpleapp
Please enter your name: Gaurav

Hello and Welcome Gaurav!
```

So with the combination of '-i' and '-t' we're now attached to the terminal, as well as the interactive mode on the container.

```
$ docker run -p 5000:8080 springbootapp
...
...
...
...
Running the SpringBoot Application {MyDemoApplication.java}
on port 8080
```

We will now look at port mapping or port publishing on containers. The springbootapp is running on the 8080 port inside the container but to be able to access it on your localhost (192.18.0.1 for example) and on port 5000 so basically 192.18.0.1:5000 we have to use the above command.

And similarly we can keep doing this for different containers as well.

For example:

```
$ docker run -p 3306:3306 mysql
```

Accessible on 192.18.0.1:3306

```
$ docker run -p 5000:8080 springbootapp
```

Accessible on 192.18.0.1:5000

```
$ docker run mysql
```

Add tons of data in the database.

```
$ docker stop mysql
$ docker rm mysql
```

Now all the data is lost with the container. To avoid this you can use the '-v' parameter so that the data is stored inside a mysql database that you mention.

```
$ docker run -v /opt/datadir:/var/lib/mysql mysql
```

It will mount the database /var/lib/mysql (DOCKER CONTAINER DIRECTORY) -$ /opt/datadir (LOCAL SYSTEM DIRECTORY).

```
$ docker inspect {container_name}
```

If you need more information than the information provided by the 'docker ps' command, you can use 'docker inspect {container_name} command to get detailed information about the container in the JSON format such as state, mount, configuration data, network settings, etc.

```
$ docker logs {container_name}
```

If you run a container in detached mode using 'docker run -d {image_name}' then you can view the logs of the container using the logs command.

3. Docker Image Commands:

You have to create a Dockefile with step by step instructions on it in the similar way you would to build and run your application on your system.

Dockerfile Format:

```
INSTRUCTION ARGUMENTS
```

Dockefile Example:

```
FROM Ubuntu:latest

RUN sleep 100
```

How to create an image?

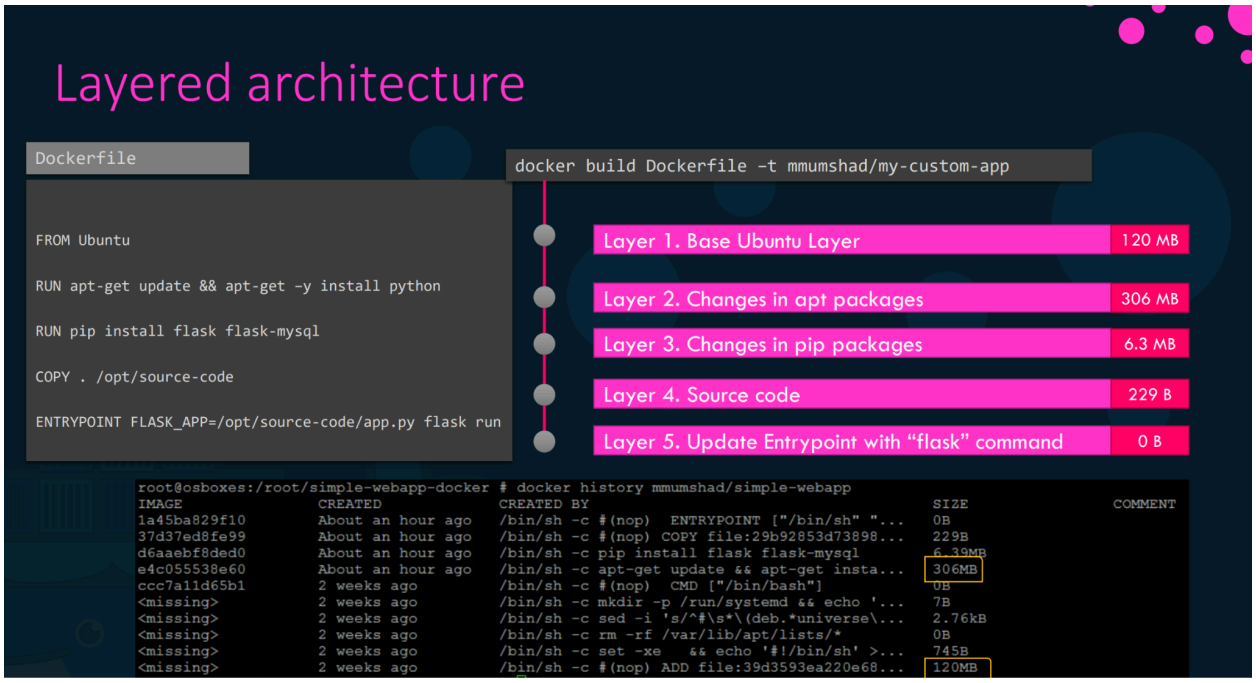
Step 1: Select an OS (Ubuntu)
Step 2: Update APT
Step 3: Install dependencies using APT
Step 4: Install Python dependencies using Pip
Step 5: Copy source code to /opt folder
Step 6: Run the web server using "flash" command

The following is the Dockerfile:

```
FROM Ubuntu:latest

RUN apt-get update -y
RUN apt-get install python -y

RUN pip install flask -y
RUN pip install flask-mysql -y

COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask running
```

```
$ docker build Dockerfile -t my-custom-app
```

When Docker builds images, it builds these in a layered architecture. Each line of instruction creates a new layer in the Docker image with just the changes from the previous layer.



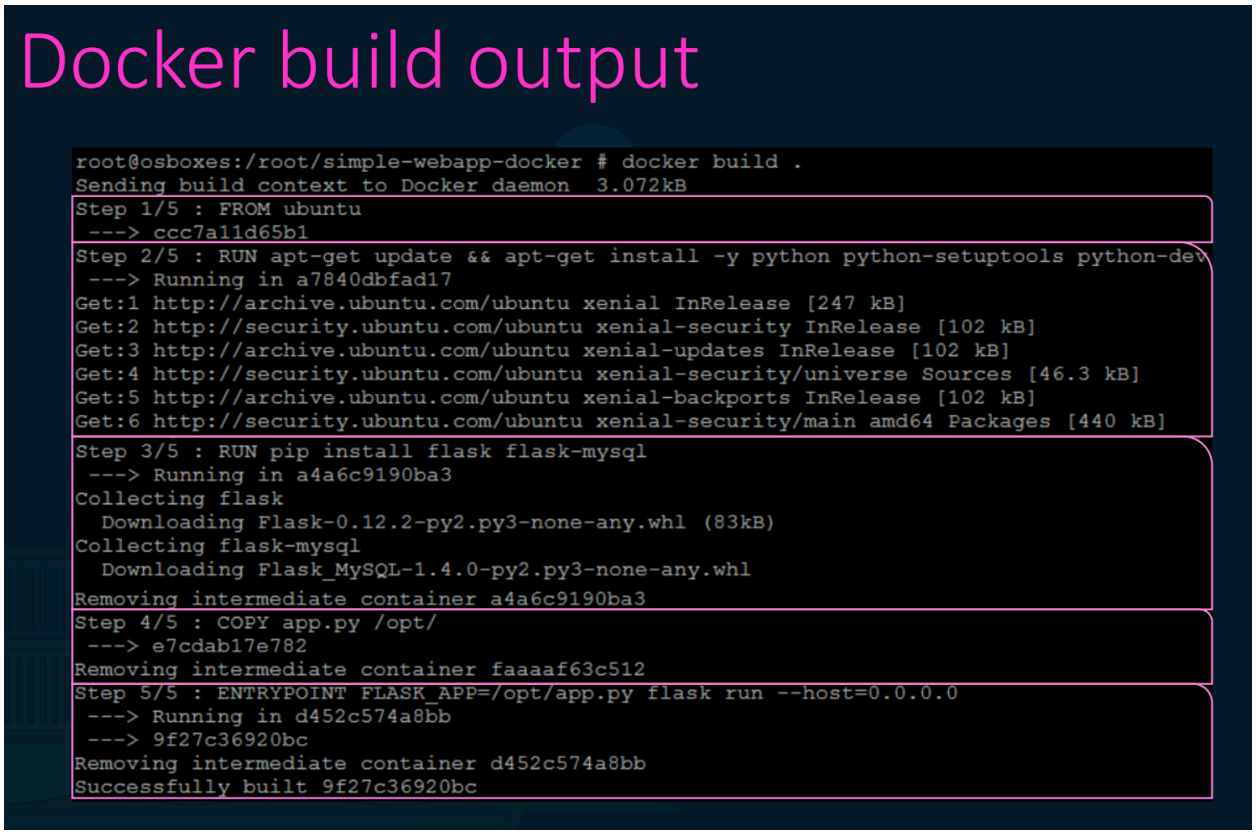
For example, the first layer is a base Ubuntu OS, followed by the second instruction that creates the second layer, which installs all the APT packages. And then the third instruction creates a third layer with the Python packages followed by the fourth layer that copies the source code over and the final layer that updates the ENTRYPOINT of the image.

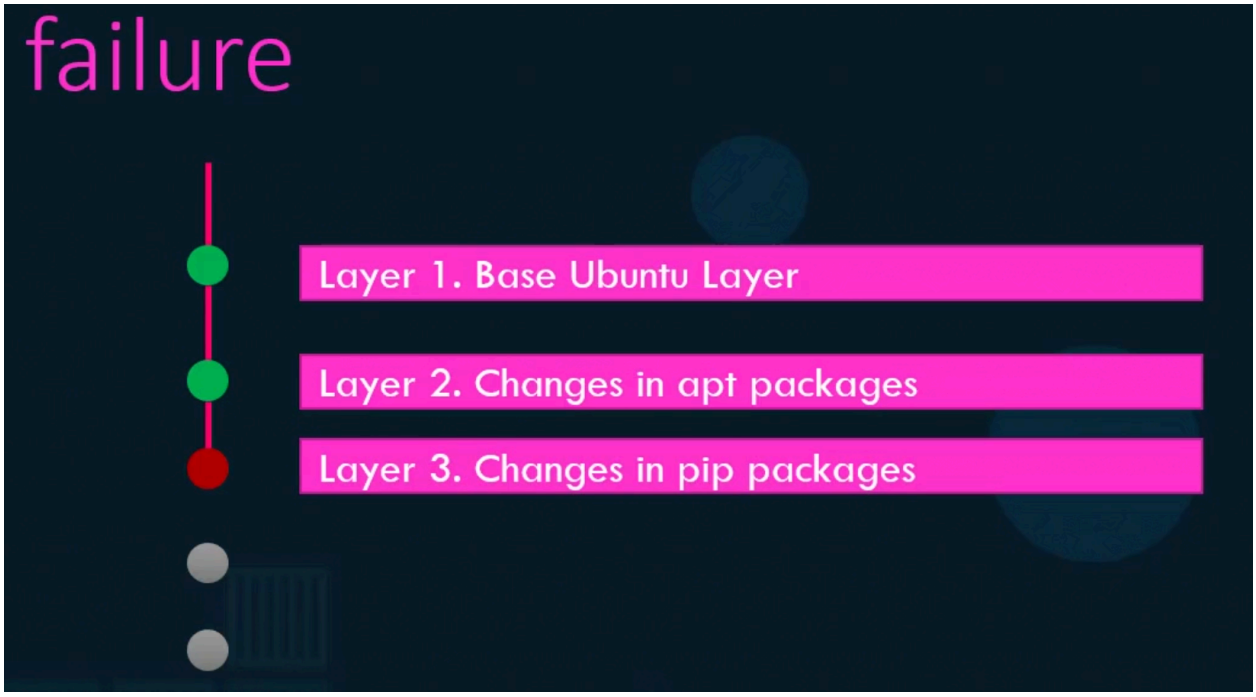
Since each layer only stores the changes from the previous layers, it is reflected in the size as well.

If you look at the base Ubuntu image it is around 120 MB in size, the APT packages are around 300 MB, and the remaining layers are small. You can get this information by running:

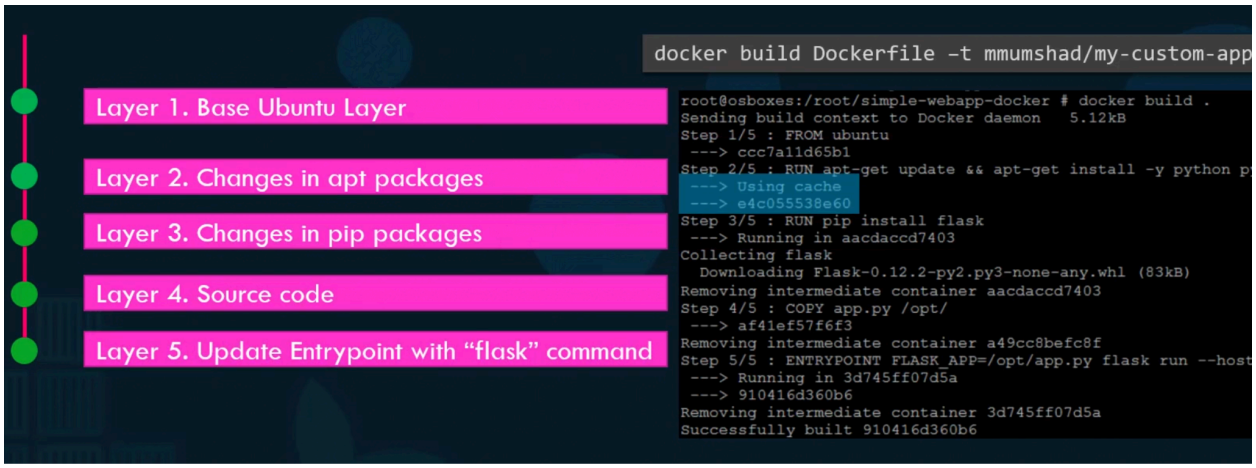
```
$ docker history simple-web-app
```

When you run the Docker build command you can see the various steps involved and the results of each task. All the layers built are cached. So the layered architecture helps you restart docker build from that particular step in case it fails. Or if you were to add new steps in the build process, you wouldn't have to start all over again.





All the layers are cached by Docker. So in case a particular build failed, for example, in the above case step 3 failed, and you were to fix the issue and rerun the Docker build, it will reuse the previous layers from the cache and continue to build the remaining layers as you can see below.



The same is true if you were to add additional steps in the Dockerfile.

This way, rebuilding your image is faster, and you don't have to wait for Docker to rebuild the entire image each time. This is helpful especially when you update the source code of your application, as it may change more frequently.

Only the layers above the updated layers need to be rebuilt.

You can containerize everything. You can run any application there and stop it directly on the container. This helps in cleanup if you don't want it anymore and also when you want to set up some software, instead of downloading and setting up the whole environment you can just run an image and use the application.

How to set environment variables?

The following is an example of a basic python application:

```
import os
from flask import Flask

app = Flask(__name__)

# Multiple lines of code

color = blue

@app.route("/")
def main():
    print(color)
    return render_template('hello.html', color=color)


if __name__ == "__main__":
    app.run(host="0.0.0.0", port="8080")
```

So instead of assigning the value for the variable color directly you can do it in the following way:

```
import os
from flask import Flask

app = Flask(__name__)

# Multiple lines of code

color = os.environ.get('APP_COLOR')

@app.route("/")
def main():
    print(color)
    return render_template('hello.html', color=color)


if __name__ == "__main__":
    app.run(host="0.0.0.0", port="8080")
```

And you can run this application using the '-e' parameter option with the docker run command:

```
$ docker run -e APP_COLOR=blue simple-webapp-color
```

```
$ docker run -e APP_COLOR=red simple-webapp-color
```

```
$ docker run -e APP_COLOR=green simple-webapp-color
```

You can inspect the environment variables set for a container in the following way:

Commands vs Arguments vs Entrypoint in Docker:

You can create an entrypoint and command inside the Dockerfile, you can decide which commands to run while running the container using these.

Let's understand the scenarios to use each of them using the following example:

```
$ docker run ubuntu
```

The above command will just run the container with ubuntu as the OS but stop right away as a container needs an application to run on the container and just the OS not running any program will stop automatically (because the container thinks that the purpose of the container which is running a program has been completed).

So you can use ARGUMENT here and make the container run for a while longer executing a command:

```
$ docker run ubuntu sleep 10
```

It will run for 10 seconds and then stop.

If you don't want to always pass this as an argument then you can use the CMD (Command) option inside the Dockerfile instead.

You can use CMD in two ways:

```
CMD "command1 param1"
CMD "sleep 10"

CMD ["command1", "param1", "command2", "param2"]
CMD ["sleep", "10"]
```

But the problem with this is that you have to hardcode the parameters, so if you want to run it for let's say 100 seconds you will have to run this command:

```
$ docker run ubuntu-sleeper sleep 100
```

This looks silly as you have already created a separate version of ubuntu container just for sleeping and you would like it to take the parameter instead of the whole command:

```
$ docker run ubuntu-sleeper 100
```

You can achieve this using ENTRYPOINT inside the Dockerfile

```
ENTRYPOINT ["sleep"]
```

Whatever you write inside ENTRYPOINT will get appended ahead of the command passed from the user. So the actual command running would be as follows, Where sleep was appended from the ENTRYPOINT from Dockerfile:

```
$ docker run ubuntu-sleeper sleep 100
```

Now what if you just want to run the ubuntu-sleep container and not pass any argument and just use a default argument?

A combination of ENTRYPOINT and CMD is perfect for that.

```
ENTRYPOINT ["sleep"]

CMD ["10"]
```

So all the following commands will now work just fine:

```
$ docker run ubuntu-sleeper
# docker run ubuntu-sleeper sleep 10
# sleep from ENTRYPOINT and 10 from CMD
```

```
$ docker run ubuntu-sleeper 100
# docker run ubuntu-sleeper sleep 100
# sleep from ENTRYPOINT
```

Now if you want your container to do something other than sleep you can replace the sleep command you passed inside the ENTRYPOINT using:

```
$ docker run --entrypoint sleep2.0 ubuntu-sleeper 20
# docker run ubuntu-sleeper sleep2.0 20
# sleep2.0 will replace the original sleep passed in
ENTRYPOINT
```

4. Docker Compose:

The purpose of Docker Compose is to provide us with the capability to run multiple containers that use resources from each other or are dependent on one another. We do that using docker-compose.yml file.

So instead of running the following commands:

```
$ docker run my-simple-webapp
```

```
$ docker run mongodb
```

```
$ docker run redis:alpine
```

```
$ docker run ansible
```

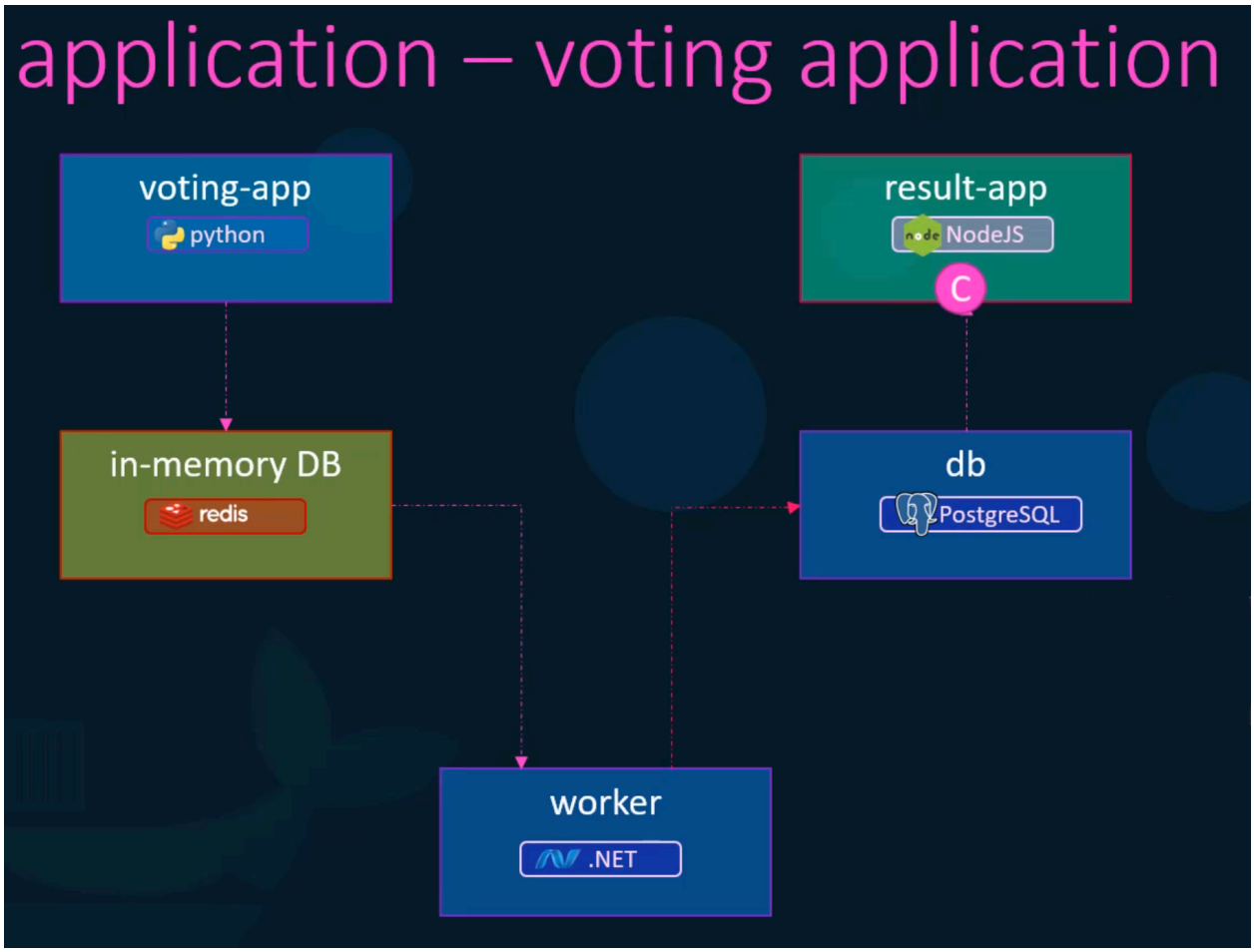
We would just run the docker-compose.yml file:

```
services:
    web:
        image: "my-simple-webapp"
    database:
        image: "mongodb"
    messaging:
        image: "redis:alpine"
    orchestration:
        image: "ansible"
```

```
$ docker-compose up # to run it
```

This was just a very basic example, generally it is very complex as containers depend on other containers and Docker Compose makes it easier for us to deal with such complexities.

Let us take the following example to understand how Docker Compose will help us using a voting application example:



This system will consist of a voting-app (Python) which is a web application to provide users with an interface to choose between two options: a cat and a dog. When the user selects an option it will get stored in the in memory database (redis). That will trigger the worker (.NET) to take the vote and update in the persistent database (PostgreSQL). User can view the result of the voting on the result-app (NodeJS)

This is how you will run this voting application without using docker-compose.yml:

```
$ docker run -d --name=redis redis
```

```
$ docker run -d --name=db postgresql:9.4
```

```
$ docker run -d --name=vote -p 5000:80 --link redis:redis
voting-app
```

```
$ docker run -d --name=result -p 5001:80 --link db:db
result-app
```

```
$ docker run -d --name=worker --link redis:redis --link
db:db worker-app
```

The docker-compose.yml file:

```
redis:
     image: redis
db:
     image: postgresql:9.4
vote:
     image: voting-app
     ports:
     - 5000:80
     links:
     - redis
result:
     image: result-app
     ports:
     - 5001:80
     links:
     - db
worker:
     image: worker-app
     links:
     - redis
     - db
```

To run the docker-compose.yml file:

```
$ docker-compose up
```

And just in case instead of running the container based on the image you want to build the image based on the Dockerfile first and then run it you have to make these changes in your docker-compose.yml file:

```
redis:
    image: redis
db:
    image: postgresql:9.4
vote:
    build: ./vote
    ports:
    - 5000:80
    links:
    - redis
result:
    build: ./result
    ports:
    - 5001:80
    links:
    - db
worker:
    build: ./worker
    links:
    - redis
    - db
```

It will first build the images using the Dockerfile in the directories- /vote, /result and /worker respectively and after building an image it will run the containers on those images.

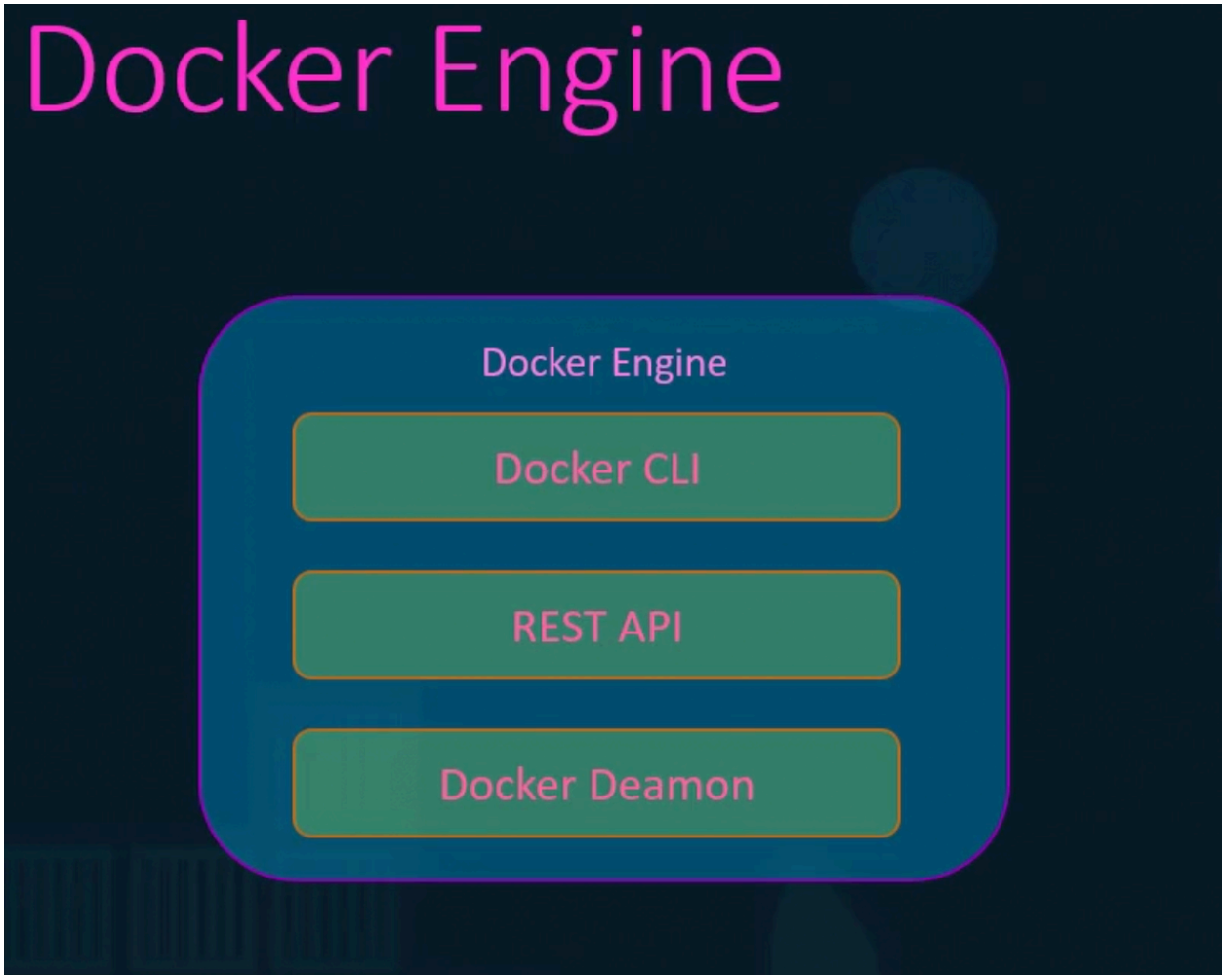
Docker Compose Versions (You have to specify on top of the file which version you are using apart from version 1):

Version 1 was used in the above example, there was no way that you could specify if a container depended on another container (voting container needs redis container to be up and running so that it can store the data inside the redis database), version 2 solves this issue by providing **"depends_on"** parameter which will ensure that redis container is always up and running for voting container to function properly.

Another feature of version 2 is that you can add several containers under a group (named services above) and docker will create a bus driver for all these containers to communicate with each other so you don't have to use the link parameter anymore.

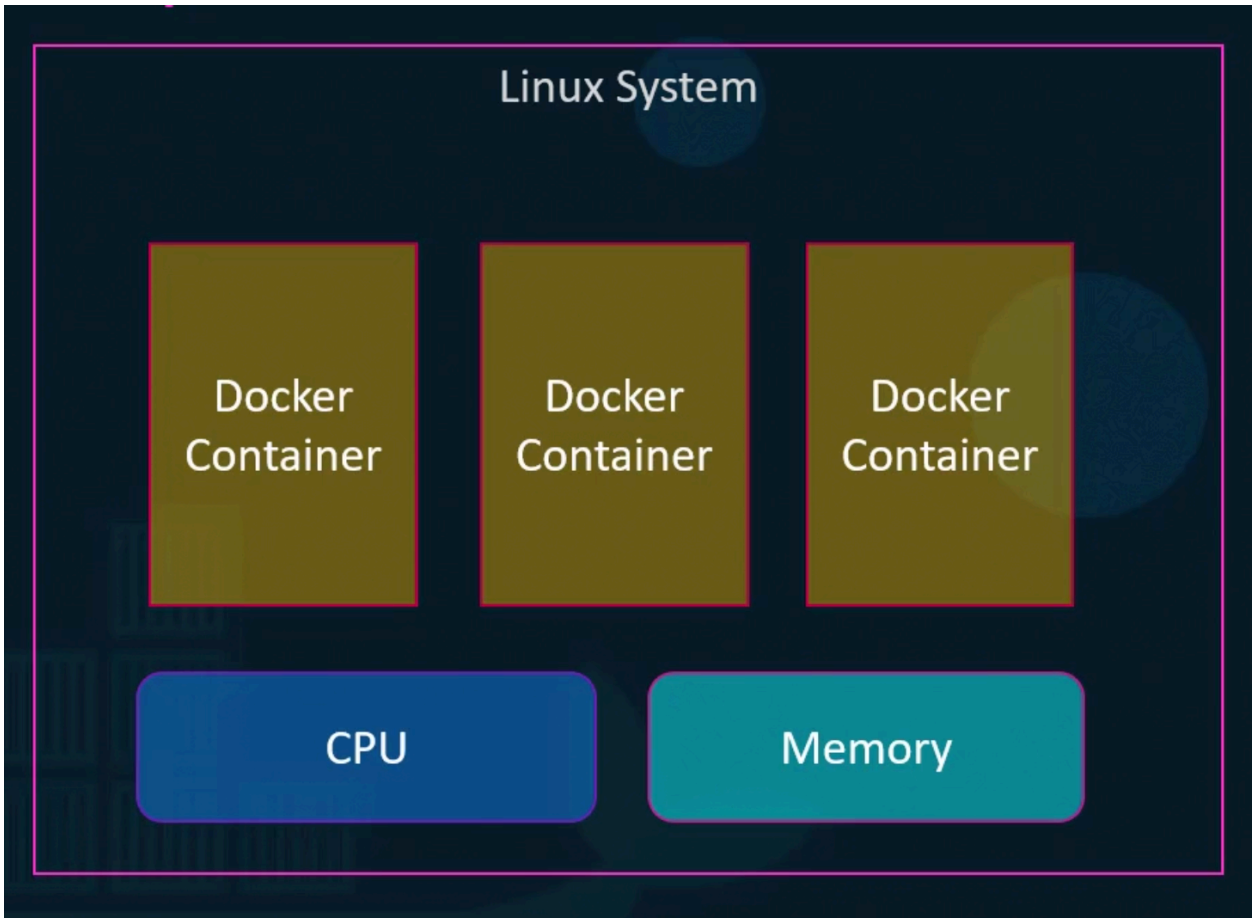
More information will be provided on version 3 later.

5. Docker Engine:



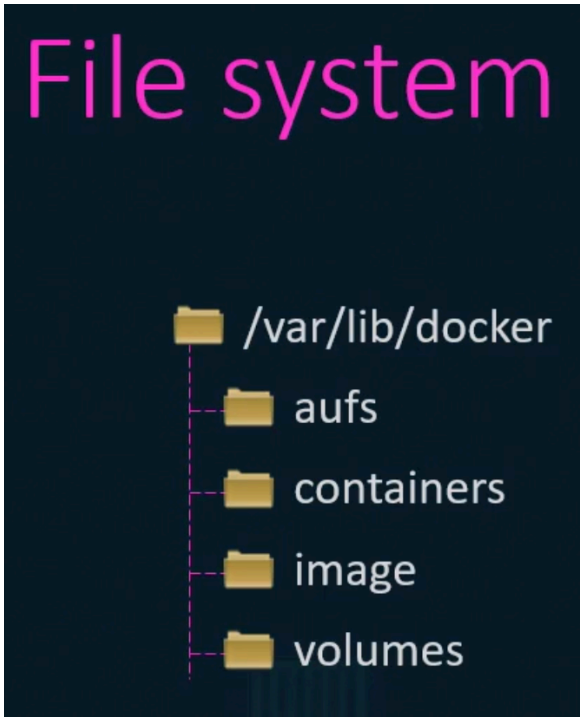
Docker Engine consists of three components: Docker daemon, REST API server, and Docker CLI.
Docker Daemon is a background process that manages Docker objects such as images, containers, volumes and networks.
Docker REST API server is the API interface that programs can use to talk to the daemon and provide instructions and Docker CLI is nothing but the command line interface we have been using until now the CLI uses the REST API server to interact with Daemon to perform these tasks in the background.

The underlying Docker host as well as the containers share the system resources such as CPU and memory.
How much of the resources are dedicated to the host and the containers? And how does Docker manage and share the resources between the containers? By default, there is no restriction as to how much of a resource a container can use. And hence, a container may end up utilizing all of the system resources on the underlying host. But there is a way to restrict the amount of CPU or RAM a container can use.
Docker uses **“cgroups”** or **“control groups”** to restrict the amount of hardware resources allocated to each container.




```
$ docker run --cpus=0.5 ubuntu # MAX 50% CPU
$ docker run --memory-100m ubuntu # MAX 100 MB RAM
```

6. Docker Storage:

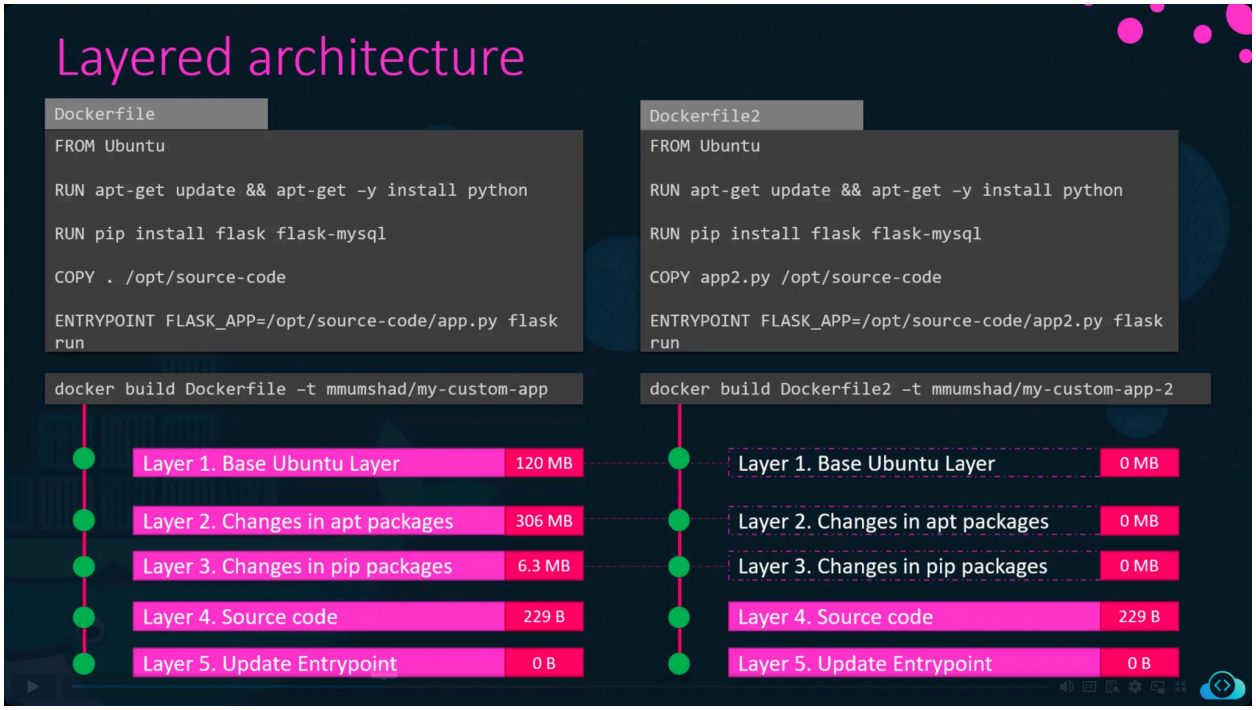


This is where Docker stores all the files related to containers, images, volumes etc respectively inside the specific folders.

To understand the internal storage system of Docker we need to recap the Layered Architecture concept that is explained under the Docker Image Commands topic and Docker Build sub topic.

Docker caches the output of each instruction of the Dockerfile and reuses that cache instead of executing the same instruction again and again in case of failure or updates.

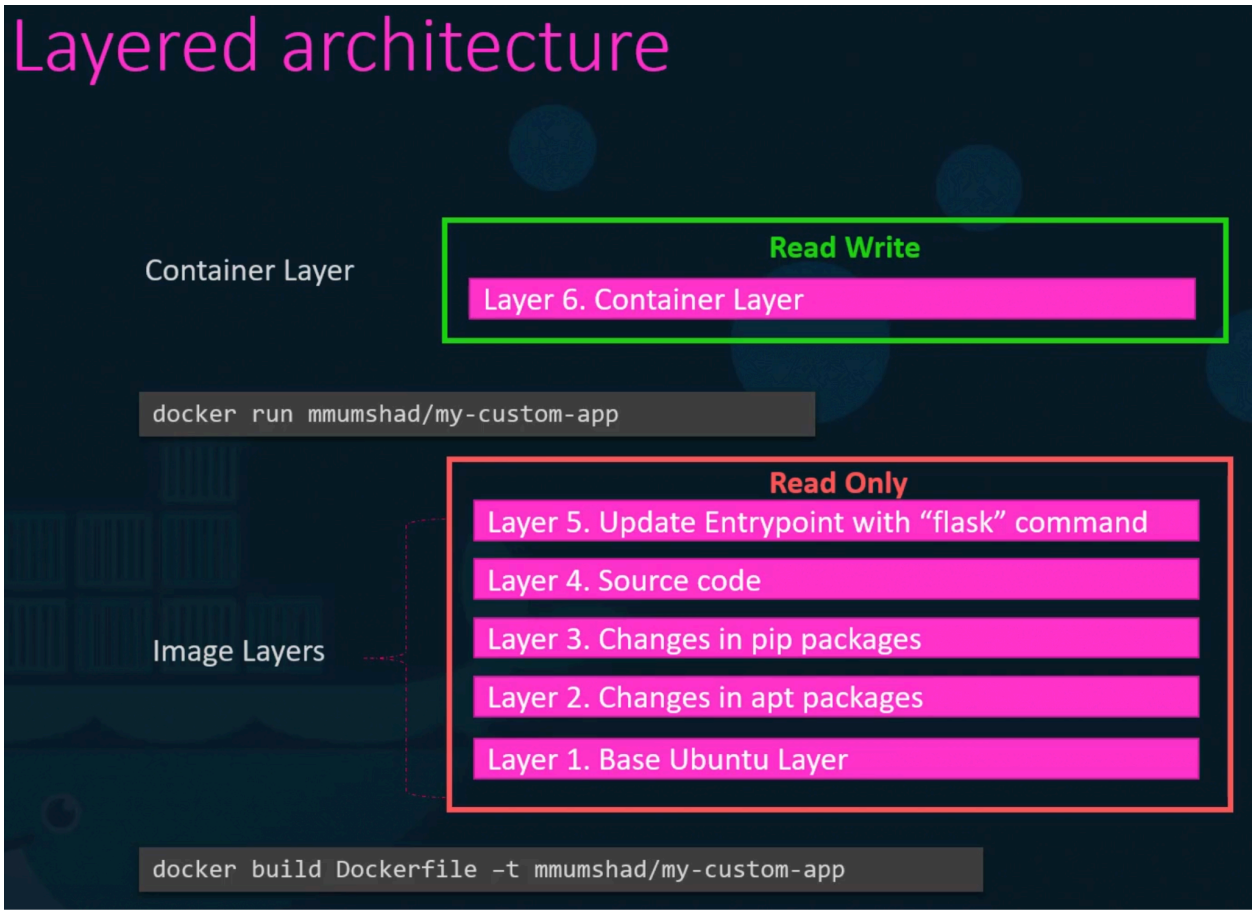
Similarly Docker also reuses those caches when we are using the same image for different applications as well, which really improves the speed for executing a new application running on the same base of already existing applications.



As you can see in the above image, docker built the cache for Layer 1, 2 and 3 and reused them while building a different application but using the same layers.

Hence just running the source code becomes much faster when working with Docker.

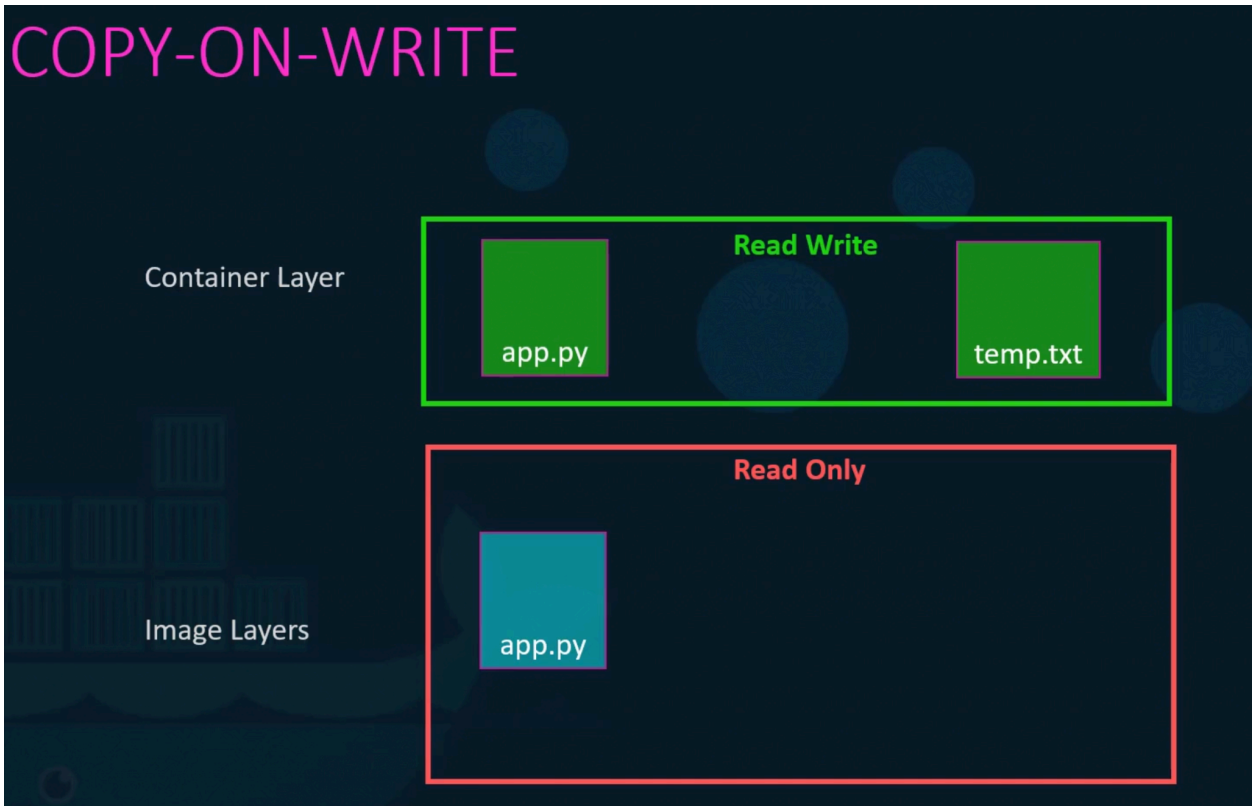
Now let's look at the scenario of running the containers:



This shows that a container layer is created when we run the container based on the image layers and the container layer is read and write unlike the image layer that is only read as we need to update it and rerun it to write in it, as the container is running on these layers they need to be stable.

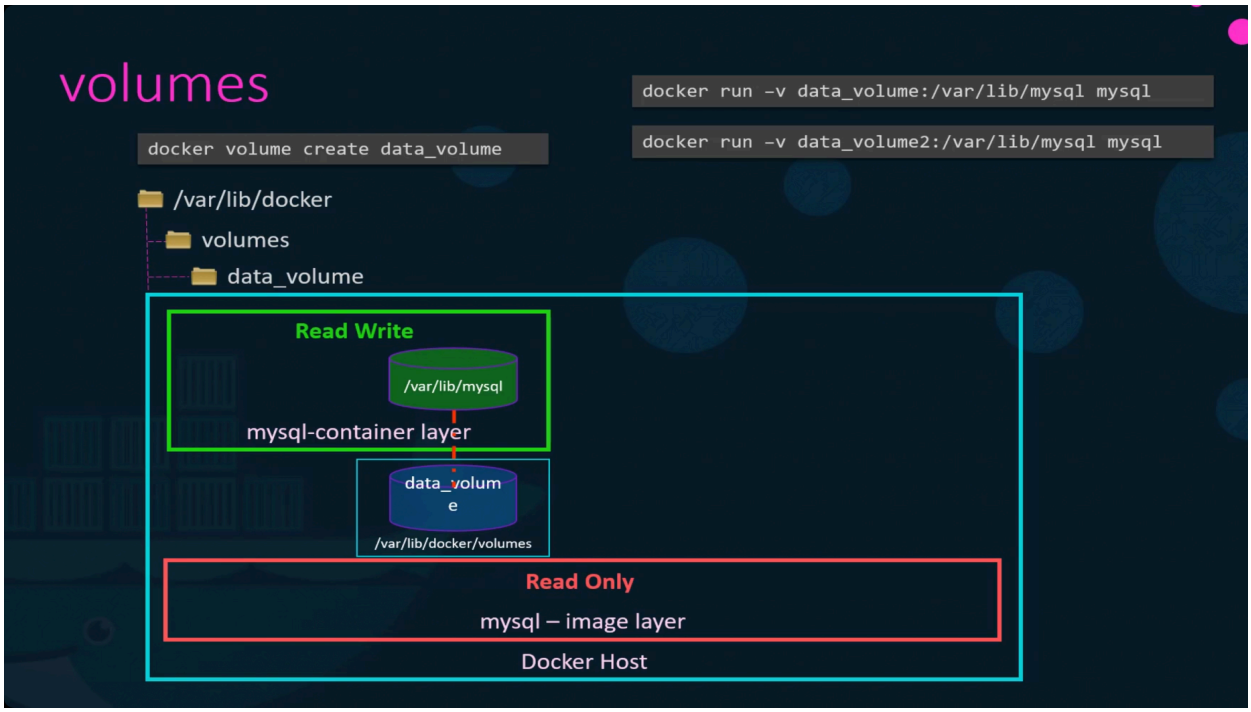
Container layer on the other hand will only exist only till the container is up and running, as soon as the container stops all the files and information of the container layer will be deleted along with the container stopping.

For example, if there is a app.py file inside the image layers and we update that file, it will make a copy of that file on the container layer and keep it there until the container is up and running, as soon as the container stops the app.py will be deleted automatically along with all the other files like temp.txt.



So if we want to add persistence to our data we need to add it using the volume parameter.

Volumes:



We will create a volume using:

```
$ docker volume create data_volume
```

And then we run the container and tell docker to store the data from data_volume to /var/lib/mysql database in our system:

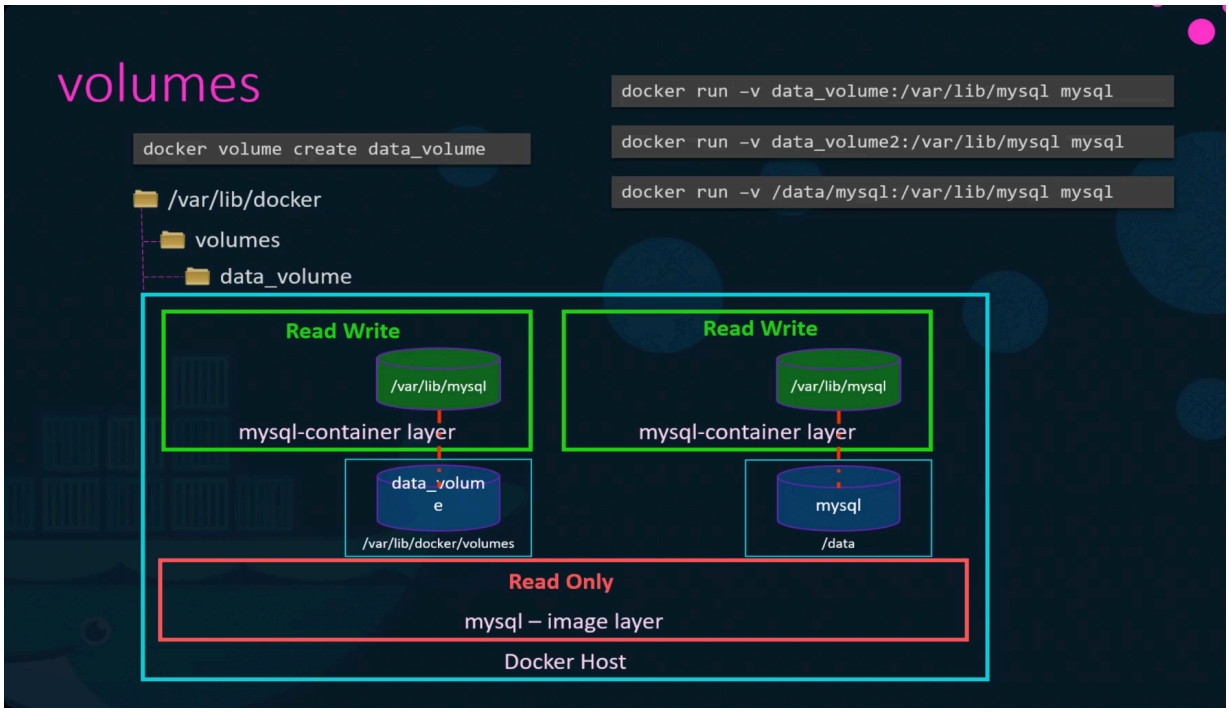
```
$ docker run -v data_volume:/var/lib/mysql mysql
```

Also, we don't have to specifically create a volume, if we run:

```
$ docker run -v data_volume2:/var/lib/mysql mysql
```

Docker will create a volume "data_volume2" for us.

What we did above is called **data mounting**. But what if we had our data already at another location? For example, let's say, we have some external storage on Docker host at /data, and we would like to store database data on that volume and not in the default /var/lib/docker/volumes folder.

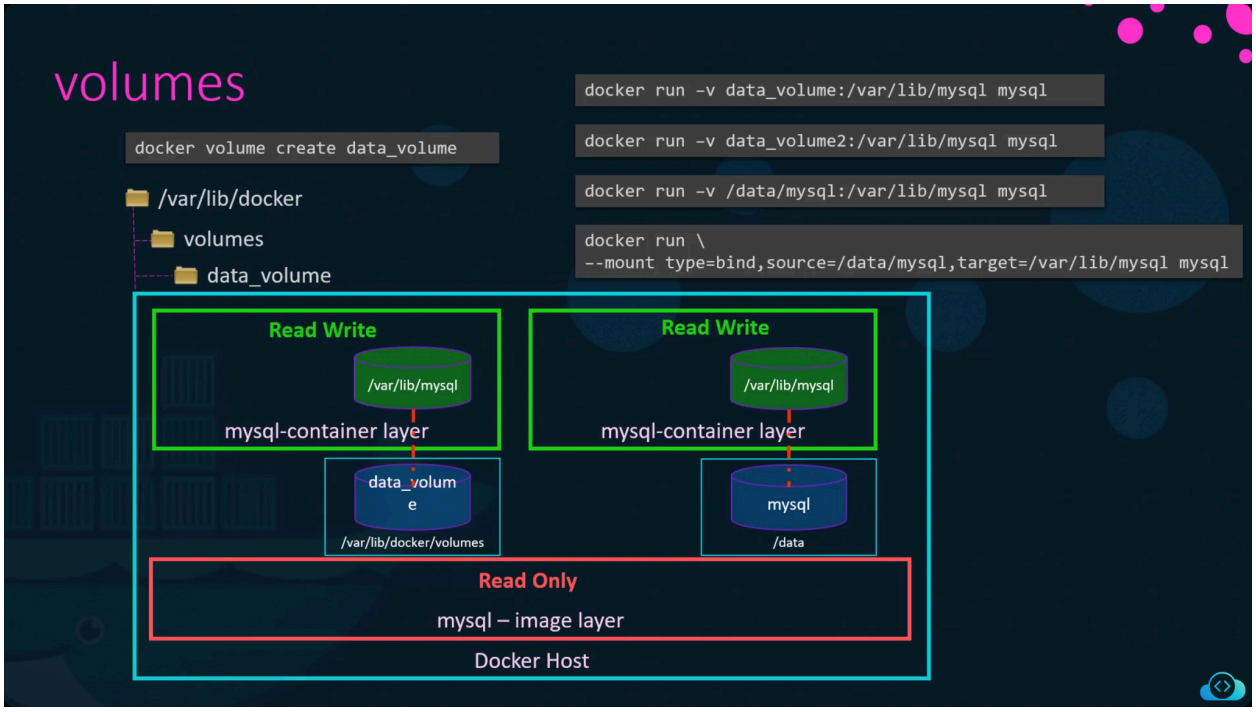


In that case we would run a container using the following command, but in this case we would provide the complete part to the folder we would like to mount, that is, /data/mysql and so, it will create a container and mount the folder to the container:

```
$ docker run -v /data/mysql:/var/lib/mysql mysql
```

And this is called **bind mounting**.

So there are two types of mounts: a volume mounting and a bind mounting. Volume mount mounts a volume from the volume directory and bind mount mounts a directory from any location to the Docker host.



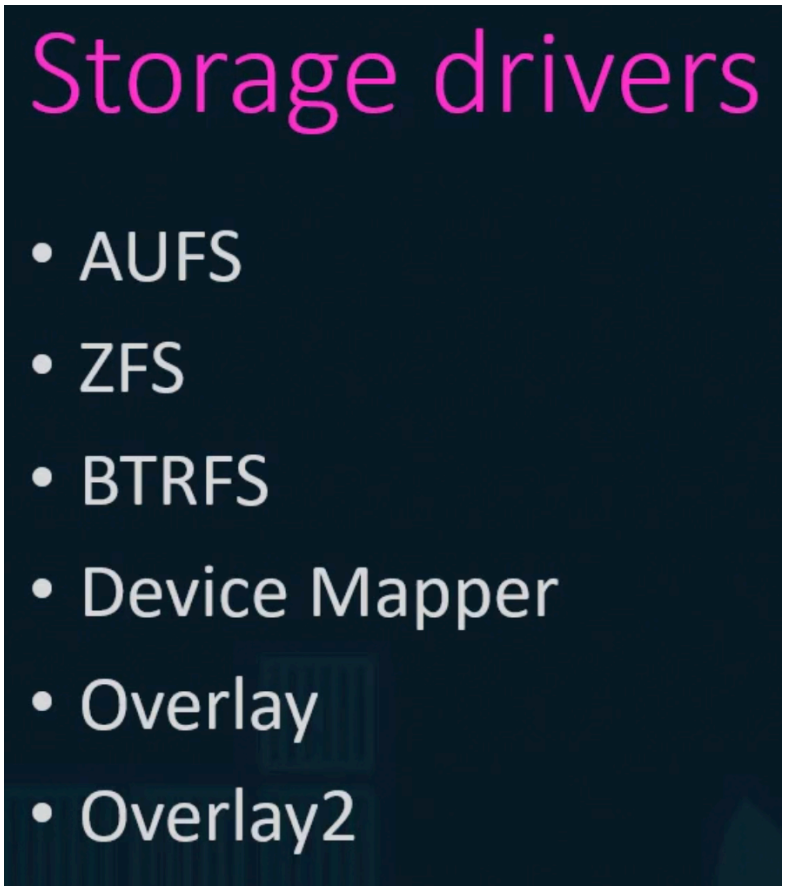
One final point is to note that using -v is old style and new way to use the mount option is the --mount, it is preferred and is more verbose as you have to specify each parameter in key value pair.

```
$ docker run --mount type=bind, source=/data/mysql,
target=/var/lib/mysql mysql
```

So who is responsible for doing all these operations? Maintaining the layered architecture. Creating a writable layer moving files across layers to enable copy and write etc?

It is the **Storage drivers.**

Docker uses Storage drivers to enable layered architecture.



The selection of the Storage Driver depends on the underlying OS, for example, with Ubuntu, the default storage driver is UFS.

Docker will choose the best storage driver automatically based on the Docker Host System.