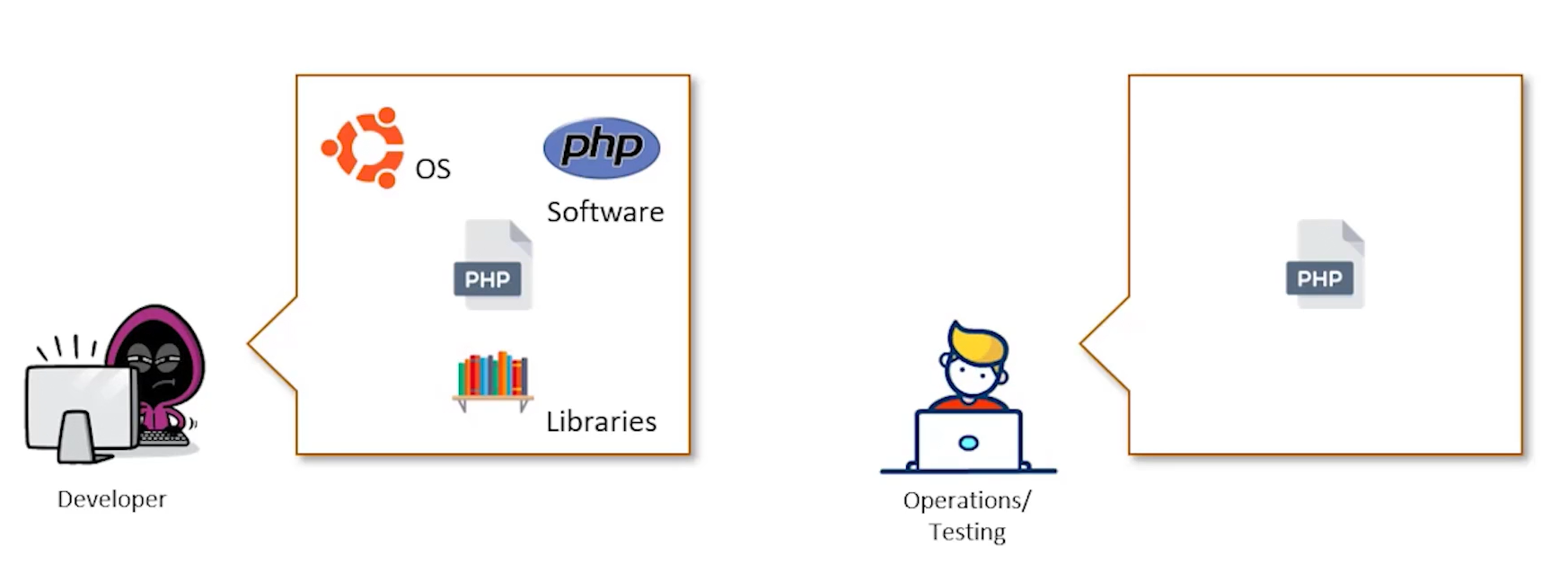
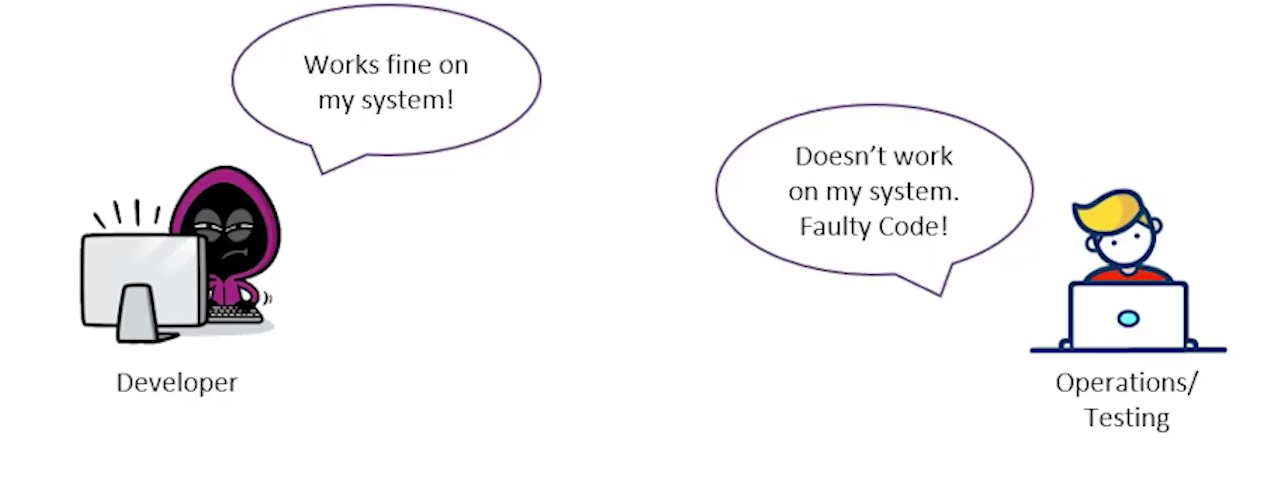
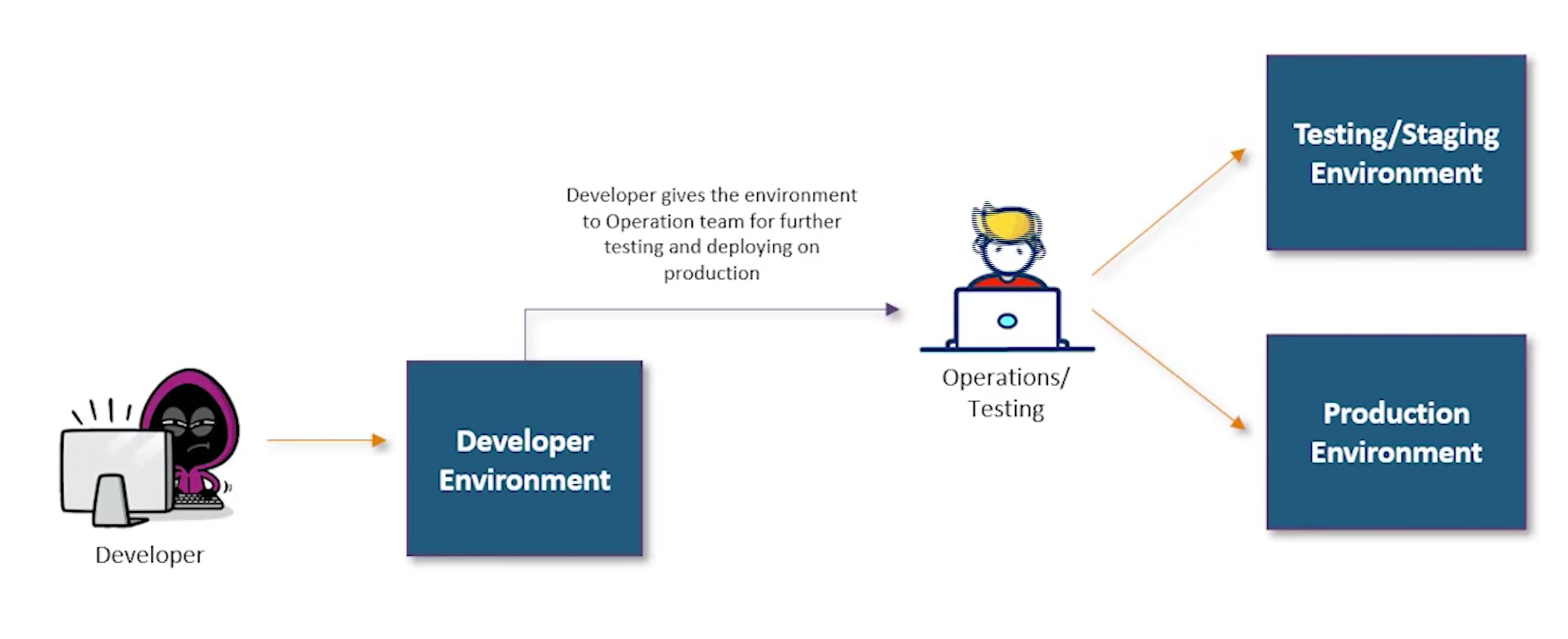
**What is Docker :**

Developers used to run the code on their system , it would run perfectly . But the same code did not run on the testing/operation team system. The issue may be because of the php version or any dependent software/binaries/libraries.

We need an entity which can contain all the software dependencies and can be ported on to other computers as plug and play





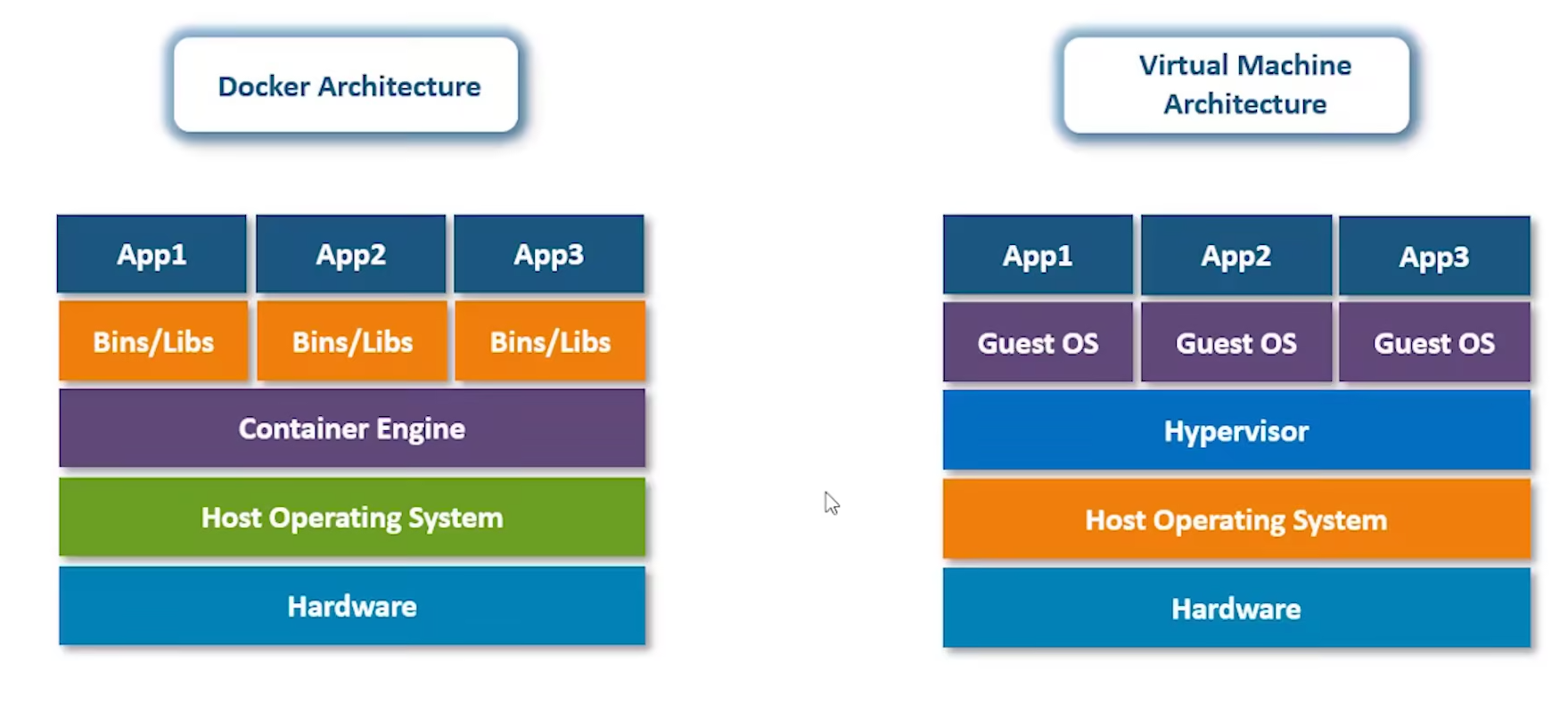


Docker is a computer program that performs operating system level virtualization, also known as containerization. It was first released in 2013 and developed by docker inc. Docker is used to run software packages called containers.

Application containerization is an OS level virtualization method used to deploy and run distributed application without launching entire virtual machine VM for each app.

Docker architecture :

Hardware 🡺 Operating System 🡺 Container Engine 🡺 Bins/libs 🡺 application



VM architecture :

Hardware 🡺 HOST OS 🡺 Hypervisor 🡺 Guest OS 🡺 Application

Advantages of Docker container:

* Take less space as compared to VM as it does not contain Guest OS.
* Runs reliably on any OS.
* Better resource utilization. It requires less hardware resulting reduction in cost.
* App isolation. Although all application deployed in one container all applications are isolated with each other.
* Container orchestration is solved.

Docker vs VM

* one example, let say you are creating an image for cent os which is Linux os.
* you have Linux machine and above that you are using docker to create centos image.
* so docker will create only binaries or libraries (bin/lib) to run internal or external commands.
* this way docker is not copying the whole OS.
* it is just using some of its resources.

Your app 🡺 Docker file (dockerize app) 🡺 Docker image 🡺 Repository 🡺 deploy into container

Docker container life cycle :

Docker hub 🡸 🡺 pull/push 🡸 🡺 docker engine 🡺 docker images 🡺 run/stop/delete

Docker hub : Docker hub the repo like git hub.

Docker engine : Docker engine is installed in your OS

Docker images : Docker image you download from docker hub to docker engine.

Container stages : If you run the image, it will be container.

Install Docker 🡺

On aws instance

sudo yum update

sudo yum install docker

#checking the version of docker 🡺

docker – - version

**#**Pull the image from docker hub 🡺

**docker pull <image-name>**

**ex : docker pull ubuntu**

**#**list out all docker images downloaded into your system 🡺

**docker images**

**#**running containers from the image name🡺

**docker container run <image-name>**

**docker container run -it -d <imagename>**

**#**listing all the containers which are running in the system **=>**

**docker container ls**

**#**list out all container including stopped 🡺

**docker container ls --all**

**#** for logging into or accessing the container🡺

**docker exec -it <containerid> bash**

**#** for stopping a running container🡺

**docker container stop <container id>**

**#**kill the container🡺

**docker container kill <container id>**

* this command kills the container by stopping the execution immediately.
* the difference between docker kill and docker stop is that docker stop gives the container

time to shutdown gracefully, in situations when it is taking too much time for getting the container to stop, one can opt to kill.

**#**to remove a stopped container from system 🡺

**docker rm <container id>**

**#** to remove an image from the system🡺

**docker rmi <image id >**

**#** login to docker hub 🡺

**docker login**

**#** pushing you container on docker hub🡺

**docker push <docker hub- username>/<image-name>**

**#**you can verify the push on dockerhub🡺

**docker exec -it <container-id> bash**

**saving changes to a docker container. You can save the container as image and reuse image there on**

* run the container : **docker container run -it -d <image-name-example:ubuntu>**
* access the container : **docker exec -it <containerid> bash**
* do any operation
* exit from the container
* save that container as image so that next time you can run the image and use the container
* docker commit <containerid> <username>/<container name>

**Installing Apache in docker 🡺**

* Run a container
* docker container run -it -d <image-name>
* go to the container

docker exec -it <containerid> bash

* **install Apache**

apt-get update

apt-get install apache2

* map the port of your machine(the server on which docker is running) to the docker container
* exit from the container
* run the container as below

sudo docker run -it -p 82:80 -d <imagename>

* you can access the Apache page outside by browsing ipaddess-of-server:82

**#Pushing a container to dockerhub 🡺**

ubuntu container 🡺any changes on container 🡺commit the changes 🡺new\_image\_with\_cmmitted\_saved\_operations 🡺push 🡺docker HUB

sudo docker login

**Dockerfile**

* Docker can build images automatically by reading the instructions from a Dockerfile
* A Dockerfile is a text file of instructions which are used to automate installation and configuration of a Docker image . Dockerfiles make it easy to deploy multiple Docker containers without having to maintain the same image across multiple virtual machines.
* Using docker build users can create an automated build that executes several command line instructions in succession.
* Dockerfile 🡺 docker image 🡺 docker container 🡺 docker hub 🡺 staging server 🡺 production server

|  |  |
| --- | --- |
| FROM | First line of a docker file is always from  The FROM keyword is used to define the base image on which we will be building  FROM ubuntu |
| ADD | The ADD keyword is used to add files to the container being built. Similar to COPY but can also fetch files from URLs and unpack compressed files.  ADD https://example.com/file.tar.gz /app/  ADD <source > <destination>  ADD . /var/www/html |
| RUN | the RUN keyword is used to add layers to the base image, by installing components. Each RUN statement , adds a new layer to the docker image.  EX : RUN apt-get -y install apache2 |
| CMD | The CMD keyword is used to run commands on the start of the container. These commands run only when there is no argument specified while running the container.  CMD apachectl -D FOREGROUND |
| ENTRYPOINT | ENTRYPOINT keyword is used strictly run commands when the container initializes.  ENTRYPOINT apachectl -D FOREGROUND |
| ENV | The ENV keyword is used to define environment variables in the container run time.  ENV MY\_VARIABLE=value |
| WORKDIR | WORKDIR /path |
| EXPOSE | Informs Docker that the container listens on specified network ports. EXPOSE 80/tcp |
| MAINTAINER |  |
| USER |  |
| VOLUME | Creates a mount point and marks it as holding externally mounted volumes. |

**Creating a docker file 🡺**

* mkdir docker 🡺 First create a folder docker
* sudo nano Dockerfile 🡺 create the docker file and add the content

FROM ubuntu

RUN apt-get update

RUN apt-get-y install apache2

ADD . /var/www/html

ENTRYPOINT apachectl -D FOREGROUND

ENV name Devops Intellipaat

* sudo nano index.html 🡺 create one more file index.html to test
* docker build <directory of docker file > -t <name of the container> 🡺 execute the docker file

docker build /home/nihar -t container-name

* docker run -it -p81:90 -d <container-name> 🡺 Run the built image
* echo $name 🡺 login to container and check variable name

**ADD /home/source /home/destination**

RUN apt-get update

RUN apt-get -y install apache2 >>> used to execute any command

CMD apachectl -D FOREGROUND > cmd is used to run a command without any argument specified and runs when container starts

ENTRYPOINT : same as cmd but entrypoint will run irrespective of argument

ENV :

ENV name name-of-valriable value-of-variable

**Ex :**

**FROM ubuntu**

**RUN apt-get update**

**RUN apt-get -y install apache2**

**ADD . /var/www/html**

**ENTRYPOINT apachectl -D FOREGROUND**

**ENV name Devops Intellipaat**

**Certainly! Here's an example Dockerfile that incorporates the `FROM`, `ADD`, `CMD`, `RUN`, `ENTRYPOINT`, `EXPOSE`, `ENV`, `WORKDIR`, and `USER` instructions:**

Dockerfile

# Use an official Ubuntu as a base image

FROM ubuntu:20.04

# Set environment variables

ENV APP\_HOME=/app

ENV USER\_NAME=myuser

ENV USER\_HOME=/home/${USER\_NAME}

# Set working directory

WORKDIR ${APP\_HOME}

# Create a non-root user

RUN useradd -m -s /bin/bash ${USER\_NAME}

# Add files to the image

ADD . ${APP\_HOME}

# Install necessary dependencies

RUN apt-get update && \

apt-get install -y python3 && \

apt-get clean && \

rm -rf /var/lib/apt/lists/\*

# Expose a port

EXPOSE 8080

# Switch to the non-root user

USER ${USER\_NAME}

# Specify the default command to run on container start

CMD ["./myapp.sh"]

# Specify an entrypoint

ENTRYPOINT ["/bin/bash", "-c", "./entrypoint.sh"]

In this example:

- We use the official Ubuntu 20.04 image as the base.

- Set some environment variables with `ENV`.

- Set the working directory to `/app` using `WORKDIR`.

- Create a non-root user named `myuser` with a home directory.

- Add the current directory's content to `/app` in the image.

- Install Python 3 as an example dependency.

- Expose port 8080 with `EXPOSE`.

- Switch to the non-root user using `USER`.

- Specify the default command to run when the container starts using `CMD`.

- Specify an entrypoint script using `ENTRYPOINT`.

You can customize this Dockerfile based on your specific application and requirements. Remember to adjust the package installations, dependencies, and scripts as needed.

**Docker storage :**

by default all the data of a container is stored on a writable container layer. This layer has the following properties

* data only exists while container is active, if the container no longer exists, the data is also deleted along with the container.
* The writable container layer is tightly coupled with the host machine, hence not portable.
* The data on the writable layer in the container is written using a storage driver.

To persists data inside the container, even after it is deleted , we have two options

* Docker volumes
* Bind Mounts

**Docker volumes :**

A docker volumes is a mountable entity which can be used to store data in docker file system .

it creates a docker volume which can be attached or detached from docker

* + Volume is simple a directory inside our container
  + First we t declare the directory as volume and the share volume
  + Even if we stop the container, we can still access the volume.
  + Volume will be created in side container.
  + You can declare a directory as a volume only while creating container.
  + You can not create volume from existing container
  + You can share one volume across any number of container
  + Volume will not be included when you update an image
  + You can mapped volume in two ways

Benefits of volume :

* + Decoupling containe from stoiarge
  + Share volume among different containers
  + Attach volume to container
  + On deleting container volume does not delete

Container <- -------- -> container

Host <- --------- -> contiainer

**Create docker volume using a docker file :**

* + Create a file vi Dockerfile
  + Write below in docker file. First line to pull the image ubuntu and second line to create a volume called myvolume1

FROM ubuntu

VOLUME ["myvolume1"]

* + Build the docker file, it will create an image as per the data mentioned in the dockerfile.

**docker build -t myimage .**

myimage is the image name, you can give any image name as per your choice.

* + **docker run -it --name container1 myimage /bin/bash**
  + it will create a container from the image you have created using dockerfile.
  + **docker run -it --name container2 --privileged=true --volumes-from container1 ubuntu /bin/bash**
  + this will create another container 2 and --volumes-from container1 means, volume1 will be shared between container1 and container2
  + start the container by using **docker attach container1**
  + login to both container ,you can see files are in sync in the volume created.

Create a docker volume using command :

* + create a volume while creating a container

**docker run -it --name conatiner3 -v /volume2 ubuntu /bin/bash**

* + create another container and use the volume from old container

**docker run -it --name container4 --privileged=true --volumes-from conatiner3 ubuntu /bin/bash**

* + login to the container **docker attach conatiner3**

A black screen with blue and white text

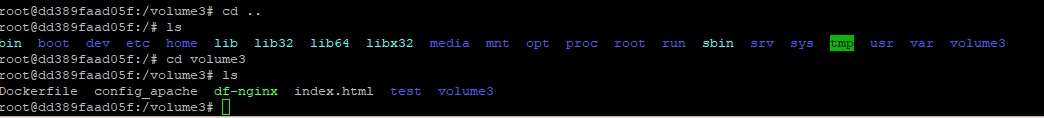
Description automatically generated

Volume (host - container) :

* + verify the directory in your host machine /home/nihar/docker
  + docker run -it --name hostcont -v /home/nihar/docker:/volume3 --privileged=true ubuntu /bin/bash
  + this will create the cnatiner named hostcont and will attach the volume of host machine

A screen shot of a computer

Description automatically generated



Docker volume commands :

### 1. Create a Volume:

docker volume create [OPTIONS] [VOLUME]

\*\*Example:\*\*

docker volume create my\_volume

### 2. List Volumes: To list all volumes on your system, you can use the `docker volume ls` command:

docker volume ls

### 3. Inspect a Volume:To get detailed information about a specific volume, you can use the `docker volume inspect` command:

docker volume inspect [VOLUME]

\*\*Example:\*\*

docker volume inspect my\_volume

### 4. Remove a Volume:To remove a volume, you can use the `docker volume rm` command:

docker volume rm [VOLUME]

\*\*Example:\*\*

docker volume rm my\_volume

### 5. Remove Unused Volumes:To remove all unused volumes (volumes not attached to any containers), you can use the `docker volume prune` command:

docker volume prune

### 6. Mount a Volume to a Container:

When running a container, you can mount a volume to it using the `-v` or `--mount` option. This is not a separate command but a flag used with `docker run`.

\*\*Example:\*\*

docker run -v my\_volume:/path/in/container my\_image

### 7. Remove Volumes Used by a Container:

If you want to remove volumes associated with a specific container when the container is removed, you can use the `--rm` option with `docker run`. This is not a separate command.

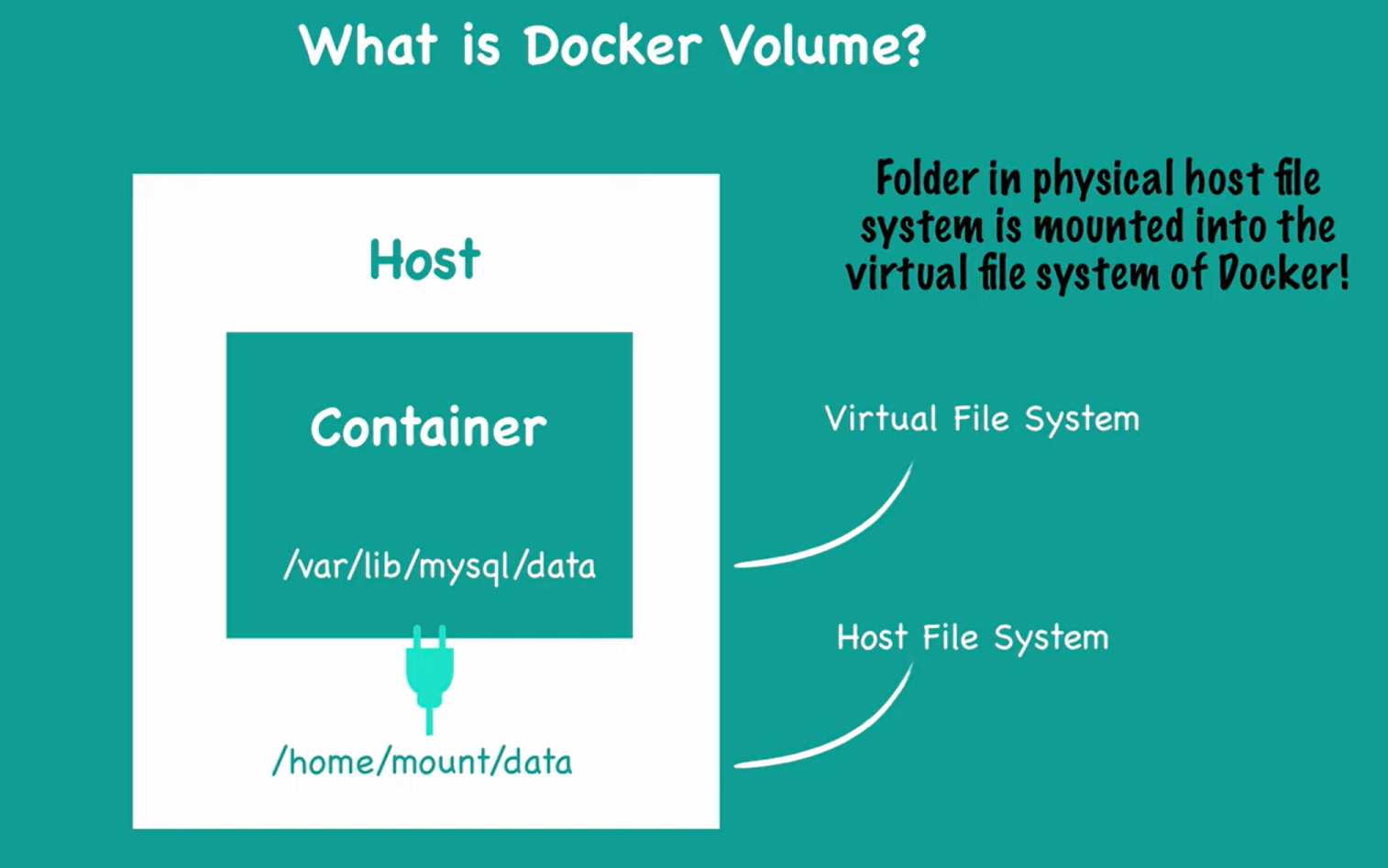
\*\*Example:\*\*

docker run --rm -v my\_volume:/path/in/container my\_image

**Bind mount**

Bind mounts mount a directory of the host machine to the docker container

**Docker run -v /home/host/file/system:/container/file/system**

****

**Linking to Docker container**

* linking is a legacy feature of docker, which is used to connect multiple containers
* with linking containers can communicate among each other

**$ docker run –it --name <name-of-container> --link <container-name> -d ubuntu**

**docker run -it --link <name of conatiner > -d <imagename>**

**ex :**

**docker run -it --name cont2 --link cont1 -d ubuntu**

here while running cont2 from image ubuntu it is being linked to container 1

**Microservices**

**Monolithic applications and its disadvantages:**

* A Monolithic application is a single tiered software application in which different components are combined into a single program which resides in a single platform.
* basically monolithic application means, all present in single source code.
* application is large and complex to understand
* entire application must redeployed on an application update
* bug in any module, can bring down entire applications
* has a barrier to adopting new technologies

**Microservices :**

* Microservices are a software development architectural style that structures an application as a collection of loosely coupled services .
* application is distributed, hence is easy to understand
* the code of only the microservice which is supposed to be updated is changed
* bug in one service does not affect other services
* no barrier to any specific technologies

**Docker Compose**

* to resolve the problem of monolithic application, we came up with microservices. And microservices can be deployed without any manual intervention by using docker compose.
* Compose is a tool for defining and running multi-container docker application.
* With compose you use a YAML file to configure your applications services. Then with a single command you create and start all the services from your configuration.

**install :**

sudo curl -L https://github.com/docker/compose/releases/download/1.29.2/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose

**YAML file :**

YAML is a superset of a json file. There are

maps : maps are key value pair

<key>:<value>

when we map a key to a value in YAML files they are termed MAPS.

Name:Intellipaat

course:Devops

**Lists : YAML Lists are a sequence of objects**

args

-arg1

-arg2

-arg3

ex

args

- sleep

- "1000"

- message

**runnig a sample docker file 🡺**

* create a folder called docker
* sudo mkdir docker
* cd docker
* create a yml file named docker-compose.yml

version: '2'

services:

sample1:

image:'httpd'

ports:

- "80:80"

sample2:

image: 'nginx'

* **build the docker compose file**

docker-compose up -d

docker ps

**to down all containers created by docker compose file 🡺**

docker-compose down

**Example 2: setting up wordpress project**

version: '3.3'

services:

db:

image: mysql:5.7

volumes:

- db\_data:/var/lib/mysql

restart: always

environment:

MYSQL\_ROOT\_PASSWORD: somewordpress

MYSQL\_DATABASE: wordpress

MYSQL\_USER: wordpress

MYSQL\_PASSWORD: wordpress

wordpress:

depends\_on:

- db

image: wordpress:latest

ports:

- "8000:80"

restart: always

environment:

WORDPRESS\_DB\_HOST: db:3306

WORDPRESS\_DB\_USER: wordpress

WORDPRESS\_DB\_PASSWORD: wordpress

WORDPRESS\_DB\_NAME: wordpress

volumes:

db\_data: {}

Docker Compose is a tool for defining and running multi-container Docker applications. It allows you to define an entire application stack, including services, networks, and volumes, in a single `docker-compose.yml` file. Docker Compose simplifies the process of managing complex multi-container Docker setups, making it easier to define, run, and scale applications.

Let's go through the main concepts and commands related to Docker Compose:

### Docker Compose Basics:

#### 1. \*\*`docker-compose.yml` File:\*\*

- Docker Compose configurations are defined in a YAML file named `docker-compose.yml`.

- This file describes services, networks, and volumes for your application.

#### 2. \*\*Services:\*\*

- Each service in Docker Compose corresponds to a container.

- Services are defined under the `services` key in the `docker-compose.yml` file.

- Example service definition:

yaml

services:

web:

image: nginx:latest

ports:

- "8080:80"

#### 3. \*\*Networks:\*\*

- Networks facilitate communication between services.

- They are defined under the `networks` key in the `docker-compose.yml` file.

- Example network definition:

yaml

networks:

mynetwork:

#### 4. \*\*Volumes:\*\*

- Volumes allow persisting data between containers.

- They are defined under the `volumes` key in the `docker-compose.yml` file.

- Example volume definition:

yaml

volumes:

myvolume:

### Docker Compose Commands:

#### 1. \*\*`docker-compose up`:\*\*

- Build and start containers defined in the `docker-compose.yml` file.

- Example:

docker-compose up

#### 2. \*\*`docker-compose down`:\*\*

- Stop and remove containers, networks, and volumes defined in the `docker-compose.yml` file.

- Example:

docker-compose down

#### 3. \*\*`docker-compose ps`:\*\*

- List containers started by `docker-compose`.

- Example:

docker-compose ps

#### 4. \*\*`docker-compose logs`:\*\*

- View output logs from services.

- Example:

docker-compose logs

#### 5. \*\*`docker-compose exec`:\*\*

- Execute a command in a running container.

- Example:

docker-compose exec web sh

#### 6. \*\*`docker-compose build`:\*\*

- Build or rebuild services.

- Example:

docker-compose build

#### 7. \*\*`docker-compose restart`:\*\*

- Restart services.

- Example:

docker-compose restart

### Advanced Concepts:

#### 1. \*\*Environment Variables:\*\*

- You can define environment variables for services in the `docker-compose.yml` file.

- Example:

yaml

services:

web:

image: nginx:latest

environment:

- MY\_VARIABLE=value

#### 2. \*\*Dependencies:\*\*

- Specify dependencies between services.

- Example:

yaml

services:

web:

image: nginx:latest

depends\_on:

- db

db:

image: postgres:latest

#### 3. \*\*Scaling Services:\*\*

- You can scale services using the `docker-compose up --scale` option.

- Example:

docker-compose up --scale web=3

#### 4. \*\*Override Compose Files:\*\*

- You can use multiple `docker-compose.yml` files and override specific settings.

- Example:

docker-compose -f docker-compose.yml -f docker-compose.prod.yml up

#### 5. \*\*Docker Compose for Production:\*\*

- For production, consider using features like health checks, resource constraints, and orchestration tools like Docker Swarm or Kubernetes.

### Example Docker Compose File:

Here's a simple example of a `docker-compose.yml` file for a web application with a database:

yaml

version: '3'

services:

web:

image: nginx:latest

ports:

- "8080:80"

networks:

- mynetwork

db:

image: postgres:latest

environment:

POSTGRES\_PASSWORD: mysecretpassword

networks:

- mynetwork

volumes:

- myvolume:/var/lib/postgresql/data

networks:

mynetwork:

volumes:

myvolume:

In this example, we have two services (`web` and `db`), a custom network (`mynetwork`), and a volume (`myvolume`). The `docker-compose up` command will start both services, create the network and volume if they don't exist, and bind the necessary components together.

This example is just a starting point, and you can customize it based on the requirements of your application.

Feel free to ask if you have specific questions or if there's anything more you'd like to learn about Docker Compose!

**Container Orchestration**

* Applications are typically made up of individually containerized components, often called microservices.
* That must be organized at the networking level for the application to run as intended.
* The process of organizing multiple containers in this manner is known as container orchestration.

**Docker swarm :**

* Docker swarm is clustering and scheduling tool for docker containers. With swarm, IT administrators and developers can establish and manage a cluster of docker nodes as single virtual system.

**Diagram

Description automatically generated**

**Creating a Docker Swarm Cluster :**

* docker swarm comes pre installed with docker .

docker swarm init --advertise-addr=<ip-address-of-leader>

🡺it will convert the node into master and generate one connection link which we need to execute on slave node

docker node ls 🡺 this will show the master slave nodes

sudo docker leave --force 🡺 to leave the swarm, if execute in the node, it will be disconnected

**Services in docker swarm :**

* containers on the cluster are deployed using services on docker swarm.
* A service is a long running docker container that can be deployed to any node worker

**Diagram

Description automatically generated with medium confidence**

docker service create --name <name-of-service> --replicas <number-of-replicas> <image-name>

docker service create -name <name-of-service> --replicas <no-of-replicas> --publish <port-mapping> <imagename>

docker service ls  **🡺 list out all services**

docker service rm <service-name> 🡺**to delete any service**

**example :**

* **create a service for nginx webserver**
* **there should be 3 replicas of this service running on the swarm cluster**
* **try accessing the service from master ip and slave ip**

**docker service create --name nginx --replicas 3 -p 80:80 nginx**

**docker service ls**

**Docker Network**

* **in order to have interactions between docker containers we need docker networks.**
* **One of the reasons Docker containers and services are so powerful is that you can**

**connect them together or connect them to non-Docker workloads. And this can be accomplished using Docker Networks**

**website container 🡸🡺 database container**

**Docker network type 🡺**

* **bridge :**
* **host :**
* **overlay:**
* **macvlan:**
* **none:**

**<html>**

**<head>**

**<title>Docker Sample App</title>**

**<?php**

**if($\_SERVER['REQUEST\_METHOD'] == "POST")**

**{**

**$servername = "db";**

**$username = "root";**

**$password = "intelli";**

**$dbname = "docker";**

**$name=$\_POST["name"];**

**$phone=$\_POST["phone"];**

**// Create connection**

**$conn = new mysqli($servername, $username, $password, $dbname);**

**// Check connection**

**if ($conn->connect\_error) {**

**die("Connection failed: " . $conn->connect\_error);**

**}**

**$sql = "INSERT INTO emp (name, phone)**

**VALUES ('".$name."', '".$phone."')";**

**if ($conn->query($sql) === TRUE) {**

**echo "New record created successfully";**

**} else {**

**echo "Error: " . $sql . "<br>" . $conn->error;**

**}**

**$conn->close();**

**}**

**?>**

**</head>**

**<body>**

**<form action="index.php" method="POST">**

**<input type="text" name="name">**

**<input type="text" name="phone">**

**<input type="submit" name="submit">**

**</form>**

**</body>**

**</html>**

**How you can save changes to a container :**

|  |  |
| --- | --- |
| Pull the docker image | docker pull ubuntu |
| run the container | docker run -it -d ubuntu |
| Access the container using the command | docker exec -it <container-id> bash |
| install apache 2 on this container | sudo apt-get update  sudo apt-get install apache2 |
| Exit the container and save the container. The saved container will be converted to an image | docker commit container\_id username/imagename |
|  |  |

**Pushing a container to Docker Hub :**

|  |  |
| --- | --- |
| sudo docker login |  |
| sudo docker push user-name/image-name |  |

**Docker File :**

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

Using docker build users can create an automated build that executes several command line instructions in succession.

Dockerfile 🡺 docker image 🡺 docker container 🡺 docker hub 🡺 staging server 🡺 production server

|  |  |
| --- | --- |
| FROM | First line of a docker file is always from  The FROM keyword is used to define the base image on which we will be building  FROM ubuntu |
| ADD | The ADD keyword is used to add files to the container being built.  ADD <source > <destination>  ADD . /var/www/html |
| RUN | the RUN keyword is used to add layers to the base image, by installing components. Each RUN statement , adds a new layer to the docker image.  EX : RUN apt-get -y install apache2 |
| CMD | The CMD keyword is used to run commands on the start of the container. These commands run only when there is no argument specified while running the container.  CMD apachectl -D FOREGROUND |
| ENTRYPOINT | ENTRYPOINT keyword is used strictly run commands when the container initializes.  ENTRYPOINT apachectl -D FOREGROUND |
| ENV | The ENV keyword is used to define environment variables in the container run time.  ENV name <variable-name> <value-of-variable> |
| WORKDIR | WORKDIR /path |
| EXPOSE | you can specify the port number |
| MAINTAINER |  |
| USER |  |
| VOLUME |  |

EX:

FROM ubuntu

RUN apt-get update

RUN apt-get -y install apache2

ADD . /var/www/html

ENTRYPOINT apachectl -D FOREGROUND

Difference between CMD and ENTRYPOINT :

The difference between CMD and ENTRYPOINT is , NETRYPOINT will run irrespective of the argument is specified or not.

**Creating a docker file :**

|  |  |
| --- | --- |
| First create a folder docker | mkdir docker |
| create the docker file and add the content | sudo nano Dockerfile  FROM ubuntu  RUN apt-get update  RUN apt-get-y install apache2  ADD . /var/www/html  ENTRYPOINT apachectl -D FOREGROUND  ENV name Devops Intellipaat |
| create one more file index.html to test | sudo nano index.html |
| execute the docker file | docker build <directory of docker file > -t <name of the container>  docker build /home/nihar -t container-name |
| Run the built image | docker run -it -p81:90 -d <container-name> |
| login to container and check variable name | echo $name |
|  |  |

|  |  |
| --- | --- |
| Common Docker Commands | |
| docker build |  |
| docker –version | Display the docker version |
| docker pull <image\_name> | Pull images from central docker repository  Ex : docker pull ubuntu |
| docker images | List out all installed docker images in the system |
| docker run <image\_name> | it helps running containers from the image name  Ex: docker run ubuntu |
| docker run -it -d <image\_name> | -it option is for interactive |
| docker ps | List out containers which are running in the system |
| docker ps -a | list out container which are running and stopped in the system |
| docker rm <container id> | remove the stopped container |
| docker rmi <image id> | remove the image from the system |
| docker run -p HOST\_PORT:CONTAINER\_PORT IMAGE[:TAG]  docker run -p 8080:80 nginx:latest | Expose Container Ports:\*\* |
| docker run -it IMAGE[:TAG] [COMMAND]  docker run -it --rm ubuntu:20.04 /bin/bash | Run in Interactive Mode:\*\* |
| docker exec CONTAINER\_ID [COMMAND]  docker exec abc123 ls /app | Run a Command in a Running Container:\*\* |
| docker run -e KEY=VALUE IMAGE[:TAG]  docker run -e SQL\_ROOT\_PASSWORD=mysecretpassword mysql:latest | Set Environment Variables:\*\* |
|  |  |
| docker kill <container\_id> | This command kill the container by stopping its execution immediately. |
| docker container prune | Remove All Stopped Containers |
| docker exec <container id> | logging into the container . container id can be get from ps command  ex: docker exec 1236h438 |
| Docker start container-name | Starting the container |
| Docker attach container | it will login into started container |
| docker stop <container id> | for stopping a running container. It gives time to container to stop gracefully. |
| **docker hub** | |
| docker login | login to docker hub account |
| docker push <username>/<containerid> | pushing container on docker hub  Ex: docker push nihar/123gh56k |
| docker exec -it <containerid>bash | verify the push on docker hub |
| docker pull <username>/<container> | pull the container from hub |
| **Docker File** | |
| FROM: | FROM ubuntu:20.04 |
| COPY: | COPY ./app /app |
| ADD: | ADD ./archive.tar.gz /data/ |
| RUN: | RUN apt-get update && apt-get install -y nginx |
| WORKDIR: | WORKDIR /app |
| ENV: | ENV MY\_VARIABLE=my\_value |
| EXPOSE: | EXPOSE 80 |
| CMD: | CMD ["nginx", "-g", "daemon off;"] |
| ENTRYPOINT: | ENTRYPOINT ["nginx", "-g", "daemon off;"] |
| VOLUME: | VOLUME /data |
| USER: | SET THE USERNAME OR UID  USER appuser |
| Arg: | Defines variables that users can pass at build-time to the builder with the docker build command  ARG version=latest |
| LABEL: | Adds metadata to an image in the form of key-value pairs.  LABEL maintainer="your-email@example.com" |
| RUN(SHELLFORM) | The shell form of the **RUN** instruction executes the command using **/bin/sh -c**.  RUN echo "Hello, Docker!" |
| SHELL | Allows the default shell used for the shell form of commands to be overridden.  SHELL ["/bin/bash", "-c"] |
| HEALTHCHECK | Tells Docker how to test a container to check that it is still working.  HEALTHCHECK CMD curl --fail http://localhost || exit 1 |
| STOPSIGNAL | Sets the system call signal that will be sent to the container to exit.  STOPSIGNAL SIGTERM |
| ONBUILD: | Adds a trigger instruction to the image that will be executed at a later time, when the image is used as the base for another build  ONBUILD ADD . /app/src |
| RUN(multiline) | RUN apt-get update && \  apt-get install -y \  package1 \  package2 |
| COPY/ADD(MULTI-STAGE) | Use multiple **FROM** statements to create intermediate images and copy artifacts between them.  FROM builder as intermediate  COPY . /app  RUN make  FROM alpine  COPY --from=intermediate /app/output /app |
| DOCKER VOLUME |  |
| docker volume create VOLUME\_NAME | Create a Volume |
| docker volume ls | List Volumes: |
| docker volume inspect VOLUME\_NAME | Inspect a Volume |
| docker volume prune | Remove Unused Volumes: |
| docker volume rm my\_volume | Remove a Volume: |
| docker run -v VOLUME\_NAME:/path/in/container IMAGE[:TAG]  docker run -v my\_volume:/app/data my\_image | Mount a Volume to a Container: |
| docker run --rm -v VOLUME\_NAME:/path/in/container IMAGE[:TAG]  docker run --rm -v my\_volume:/app/data my\_image | Remove Volumes Used by a Container:\*\* |
| Swarm Commands: |  |
| docker swarm init | Initialize Swarm:\*\* |
| docker swarm join --token <TOKEN> <MANAGER-IP>:2377 | Join a Node to Swarm: |
| docker service create --replicas 3 --name my-web-app -p 8080:80 my-web-app-image | Create a Service:\*\* |
| docker service ls | List Services:\*\* |
| docker service scale my-web-app=5 | Scale a Service:\*\* |
| docker service inspect my-web-app | Inspect Service:\*\* |
|  |  |
| docker run -p HOST\_PORT:CONTAINER\_PORT IMAGE[:TAG]  docker run -p 8080:80 nginx:latest | Expose Container Ports:\*\* |
| docker run -it IMAGE[:TAG] [COMMAND]  docker run -it --rm ubuntu:20.04 /bin/bash | Run in Interactive Mode:\*\* |
| docker exec CONTAINER\_ID [COMMAND]  docker exec abc123 ls /app | Run a Command in a Running Container:\*\* |
| docker run -e KEY=VALUE IMAGE[:TAG]  docker run -e SQL\_ROOT\_PASSWORD=mysecretpassword mysql:latest | Set Environment Variables:\*\* |
| docker run -v HOST\_PATH:CONTAINER\_PATH IMAGE[:TAG]  docker run -v /host/data:/container/data ubuntu:20.04 | Volume Mounting:\*\* |

Docker stop gives time to container shutdown gracefully. In situation where docker stop is taking time to stop, once can opt to kill it.

Docker Image 🡺

used to run code in a docker container

docker images are like templates

an image is similar to a snapshot.

an image is a collection of files/layer that contain all of the components required to setup a fully functional container.

Why should we use docker image 🡺

* deployed in a docker environment , it can be used to run asa a docker container,
* docker images are reusable assets.
* does not have to recrate an image from scratch.

Container 🡺

* unit of software that packages up code and all its dependencies.
* container is a runnable docker image instance.

docker images vs container 🡺

* snapshots of containers that are used to create new containers.
* run the image, it turns into a container
* images can be created only once
* containers can be created unlimited number of times
* we can share docker images but not docker containers.

**Docker important questions**

**what is the difference between docker stop and docker kill ?**

Ans : Docker stop command gives time to stop the container however docker kill does not give any time. It immediately stop the container.

**DOCKER PLURALSIGHT**

**Installing Docker 🡺**

**Architecture 🡺**

**Container : Isolated area on an OS with resources usage limits applied.**

**Kernel Internals :**

**there are two building blocks for creating a container in linux.**

* + **namespaces (isolation)**
  + **control groups (controlling groups )**

**-docker container is an organized collection of name spaces. There are different namespaces we used in docker container engine.**

* **process id**
* **network**
* **filesystem/mount**
* **inter-proc comms (ipc)**
* **uts**
* **user**

**-Control group controls the resources by different containers.**

**Graphical user interface, application

Description automatically generated**

**in Linux : client 🡸 🡺 daemon 🡸🡺container 🡺 run c**

**Diagram

Description automatically generated**

**Images 🡺**

* **an image is a read only template for creating application containers.**
* **an image is combination of os files & objects,app files,manifest files.**
* **image is a bunch of layers communicated by manifest file.each layer is combination of files and objects related to os.**
* **manifest files are the json files.**
* **when you pull any image from any docker hub, two thigs happen. 1. get manifest, 2. pull the layers.**
* **manifest file are the config files.**

**Registry 🡺**

**Registry is the repository where we keep or save our images.**

**Diagram

Description automatically generated**

**A picture containing text, document, screenshot, receipt

Description automatically generated**

**Docker file 🡺**

* **Dockerfile is used to automate the build.**
* **it is file where we write instructions to pull specific image from registry and execute operations.**
* **we can keep the dockerfile in some of the directories in host os and execute build command to pick the dockerfile**
* **Or we can keep the dockerfile in github and use the build command to pick it from github**

**docker image build -t psweb <github link >**

**docker image build -t psweb <directoryname >**

**ex: docker image build -t psweb https://github.com/niharswain/psweb.git**

docker image pull <image-name> 🡺 pulling docker image from docker hub

docker system info 🡺 shows the system information, json detail

docker image ls 🡺 list out the images downloaded

docker image ls --digests 🡺 list out the images along with

Container 🡺

* container is the smallest object in docker.
* container is a writable layer above an image.
* it is a run time execution environment.

Diagram

Description automatically generated

logging 🡺

journalctl -u docker.service

try /var/log/messages

Swarm 🡺

swarm has two parts

* secure cluster
* orchestrator part

swarm is a secure cluster of docker nodes including master and worker.

docker swarm is pre installed.

initialize the swarm.

docker swarm init 🡺 activate the docker swarm

docker node ls 🡺 list out docker swarm nodes

docker swarm join-token manager 🡺 it will make the current node as manager node and create a token. by executing that token in any node will convert that to manager node as well.

you can create multiple manage node but one will be the leader.

docker swarm join-token worker 🡺 it will generate the token for worker. you can execute this command on any manager to create this token. once you get the token , execute that token in any node to convert that to worker node

docker swarm join-token --rotate worker 🡺 it will rotate worker node

if you lock your swarm, it stops restarted managers from automatically rejoining the swarm.

when you lock the swarm, you would get a passkey. by using that passkey later it will unlock the swarm.

docker swarm init --autolock 🡺 auto lock new swarm

docker swarm update --autolock=true 🡺 auto lock for existing swarm

docker swarm unlock 🡺 unlock the swarm, it will ask the key

docker swarm update --cert-expiry -2d 🡺 update the certificate expiry to 2 days

Container Networking 🡺

Network type :

Bridge network: docker 0.

overlay

macvlan

docker network ls 🡺 list of the docker networks

docker network create -o encrypted

Graphical user interface, diagram

Description automatically generated

docker history <imagename> 🡺docker history ubuntu , it will show the history how the image was built in the host

docker image inspect <imagename> 🡺 docker image inspect ubuntu, it will show the content of manifest file of the image. ex layers and

**docker image rm <imagename> 🡺 docker image rm ubuntu, it will delete the image ubuntu**

**docker image build -t psweb <github link > 🡺 execute the Dockerfile from github**

**docker image build -t psweb <directoryname > 🡺 execute the dockerfile from the mentioned directory**

**docker container ls 🡺 list out the container**

**docker container run -it <imagename> <shellname>**

**ex: docker container run -it ubuntu bash 🡺 run the container in interactive mode and can be accessible through bash shell**

**docker container run -d <imagename> sleep 1d 🡺 sleep container for 1 day**

**docker container stop**

**docker container start**

**DOCKER INTERVIEW QUESTIONS AND CHEATSHEET**

### Common Commands:

#### 1. \*\*Pull an Image:\*\*

docker pull IMAGE[:TAG]

Example:

docker pull ubuntu:20.04

#### 2. \*\*Run a Container:\*\*

docker run [OPTIONS] IMAGE[:TAG] [COMMAND] [ARG...]

Example:

docker run -it --rm ubuntu:20.04 /bin/bash

#### 3. \*\*List Running Containers:\*\*

docker ps

#### 4. \*\*List All Containers (including stopped ones):\*\*

docker ps -a

#### 5. \*\*Stop a Running Container:\*\*

docker stop CONTAINER\_ID

Example:

docker stop abc123

#### 6. \*\*Remove a Container:\*\*

docker rm CONTAINER\_ID

Example:

docker rm abc123

#### 7. \*\*Remove All Stopped Containers:\*\*

docker container prune

#### 8. \*\*Inspect Container Details:\*\*

docker inspect CONTAINER\_ID

Example:

docker inspect abc123

### Management Commands:

#### 9. \*\*Manage Images:\*\*

- \*\*List Images:\*\*

docker images

- \*\*Remove an Image:\*\*

docker rmi IMAGE\_ID

Example:

docker rmi ubuntu:20.04

#### 10. \*\*Manage Volumes:\*\*

- \*\*Create a Volume:\*\*

docker volume create VOLUME\_NAME

Example:

docker volume create my\_volume

- \*\*List Volumes:\*\*

docker volume ls

- \*\*Remove a Volume:\*\*

docker volume rm VOLUME\_NAME

Example:

docker volume rm my\_volume

#### 11. \*\*Manage Networks:\*\*

- \*\*List Networks:\*\*

docker network ls

- \*\*Remove a Network:\*\*

docker network rm NETWORK\_NAME

Example:

docker network rm mynetwork

### Swarm Commands:

#### 12. \*\*Initialize Swarm:\*\*

docker swarm init

#### 13. \*\*Join a Node to Swarm:\*\*

docker swarm join --token <TOKEN> <MANAGER-IP>:2377

#### 14. \*\*Create a Service:\*\*

docker service create --replicas 3 --name my-web-app -p 8080:80 my-web-app-image

#### 15. \*\*List Services:\*\*

docker service ls

#### 16. \*\*Scale a Service:\*\*

docker service scale my-web-app=5

#### 17. \*\*Inspect Service:\*\*

docker service inspect my-web-app

### General Commands with Options:

#### 18. \*\*Expose Container Ports:\*\*

docker run -p HOST\_PORT:CONTAINER\_PORT IMAGE[:TAG]

Example:

docker run -p 8080:80 nginx:latest

#### 19. \*\*Run in Interactive Mode:\*\*

docker run -it IMAGE[:TAG] [COMMAND]

Example:

docker run -it --rm ubuntu:20.04 /bin/bash

#### 20. \*\*Run a Command in a Running Container:\*\*

docker exec CONTAINER\_ID [COMMAND]

Example:

docker exec abc123 ls /app

#### 21. \*\*Set Environment Variables:\*\*

docker run -e KEY=VALUE IMAGE[:TAG]

Example:

docker run -e MYSQL\_ROOT\_PASSWORD=mysecretpassword mysql:latest

#### 22. \*\*Volume Mounting:\*\*

docker run -v HOST\_PATH:CONTAINER\_PATH IMAGE[:TAG]

Example:

docker run -v /host/data:/container/data ubuntu:20.04

These are just a selection of commonly used Docker commands across various aspects of container management. The options and use cases can be extensive, so be sure to refer to the [official Docker documentation](https://docs.docker.com/) for more details on each command and its options**.**