=========
Docker
========

1) What is an application ?

2) Application Tech stack ?

3) Application Architecture

4) Life without Docker

5) Life with Docker

6) What is Docker

7) What is Virtualization ?

8) What is Containerization ?

9) Docker Architecture ?

=> Application is nothing but collection of programs

=> Every Application contains 3 layers

1) Frontend (UI)

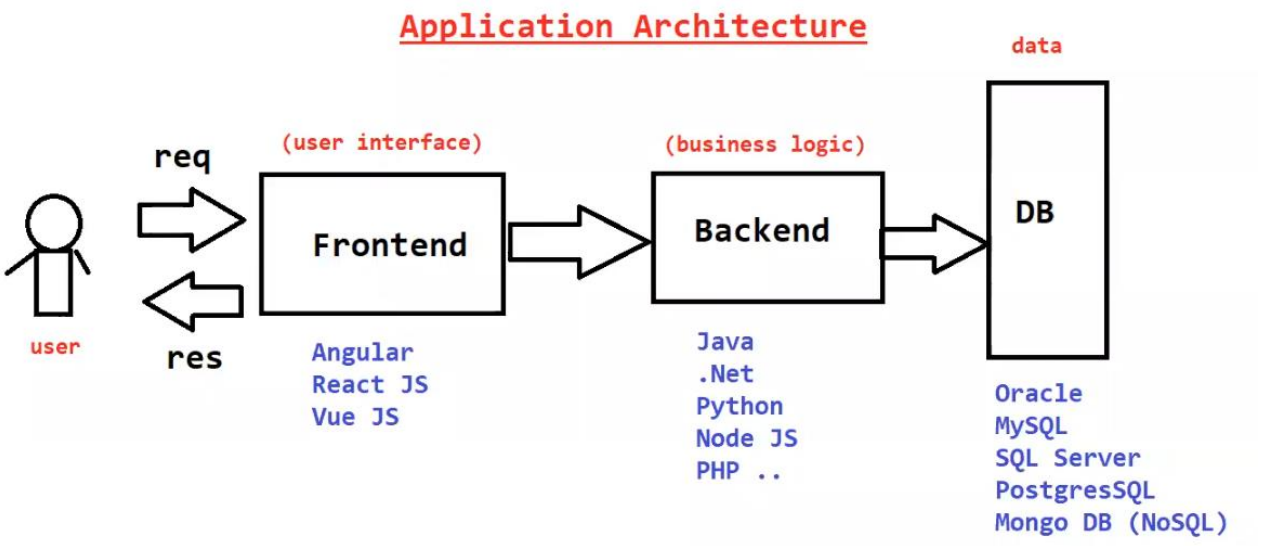
- Angular
- React JS
- Vue JS

2) Backend (Business Logic)

- Java
- .Net
- Python
- Node JS
- PHP

3) Database

- Oracle
- MySQL
- Postgres
- SQL Server
- Mongo DB



==========
Application Environments
==========

=> In Realtime, our application will be deployed into Multiple Environments for Testing Purpose

1) DEV Env ---> Developers Testing

2) SIT Env ---> Testing Team (QA) - System Integration Testing

3) UAT Env ---> Client Side Testing - User Acceptance Testing

4) PILOT Env (Pre-Production) ---> Testing with Live Data

=> Once testing completed in all above environments then it will be deployed into PRODUCTION Env.

=> Production env means live environment.

=> End users will access application from Production env

==========
Life without Docker
==========

=> We need to install all the required softwares in all environments to run our application.

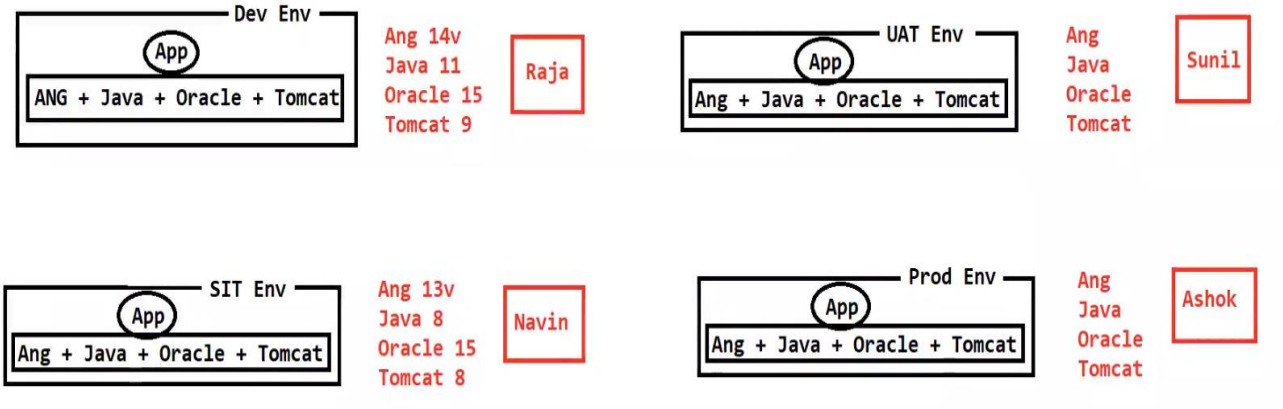
=> We need to make sure we are using same versions of softwares in all machines.

=> If any software version is not matched then application execution may fail

Ex : Raja installed Java 11 v in Dev Env

Sunil installed Java 8 v in SIT Env

=> If we want to run our application in multiple machines then we have to install required softwares in all those machines which is hectic task.



==========
Life with Docker
==========

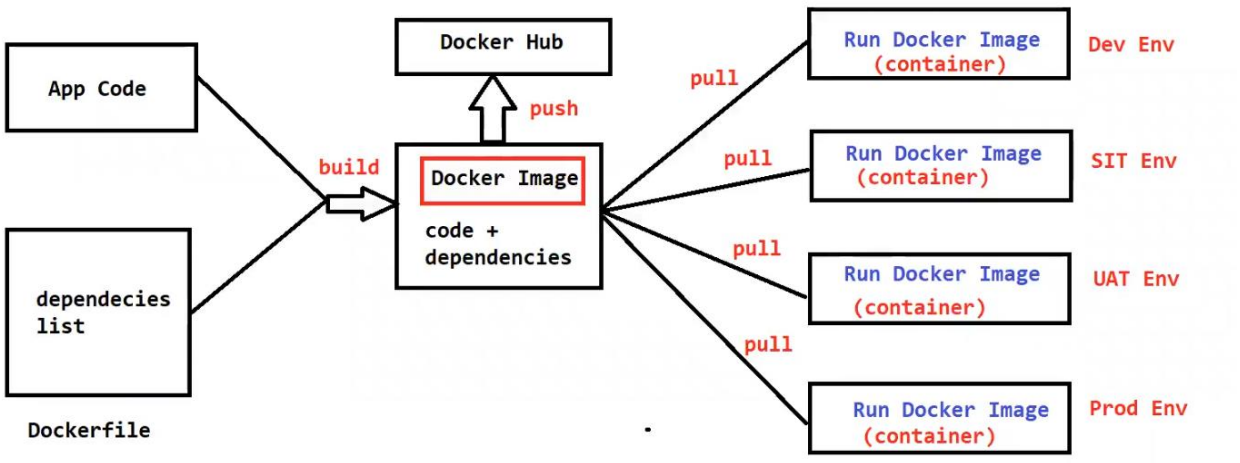
=> Docker is a containerization platform

=> Docker is used to build and deploy our application into any machine without bothering about dependencies.

=> Dependencies means the softwares which are required to run our application.

Dependencies = OS / Angular / React / Java / DB / Tomcat etc...

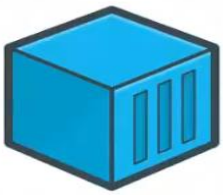
=> Docker will reduce the gap between Development and Deployment

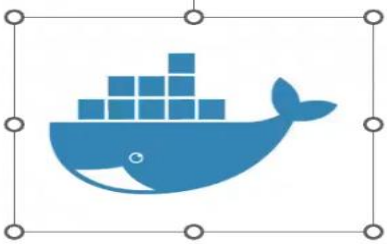


What is Docker

ASHOK IT
Learn Here.. Lead Anywhere..!!

- **Docker is a containerization platform**
- **The whole idea of Docker is to easily develop applications, ship them into containers which can then be deployed anywhere.**





==========
Docker Architecture
==========

1) Dockerfile : It contains instructions to build docker image

2) Docker Image : It is a package which contains code + dependencies

3) Docker Registry : It is Arepository to store docker images

4) Docker Container : It is a runtime process which runs our application

Note: Once Docker image is created then we can pull that image and we can run that image in any machine.

==========
Virtualization
==========

-> Installing Multiple Guest Operating Systems in one Host Operating System

-> Hypervisior S/w will be used to achieve this

-> We need to install all the required softwares in Guest Operating Systems to run our application

-> It is old technique to run the applications

-> System performance will become slow in this process

-> To overcome the problems of Virtualization we are going for Containerization concept.

==========
Containerization
==========

-> It is used to package all the softwares and application code in one container for execution

-> Container will take care of everything which is required to run our application

-> We can run the containers in Multiple Machines easily

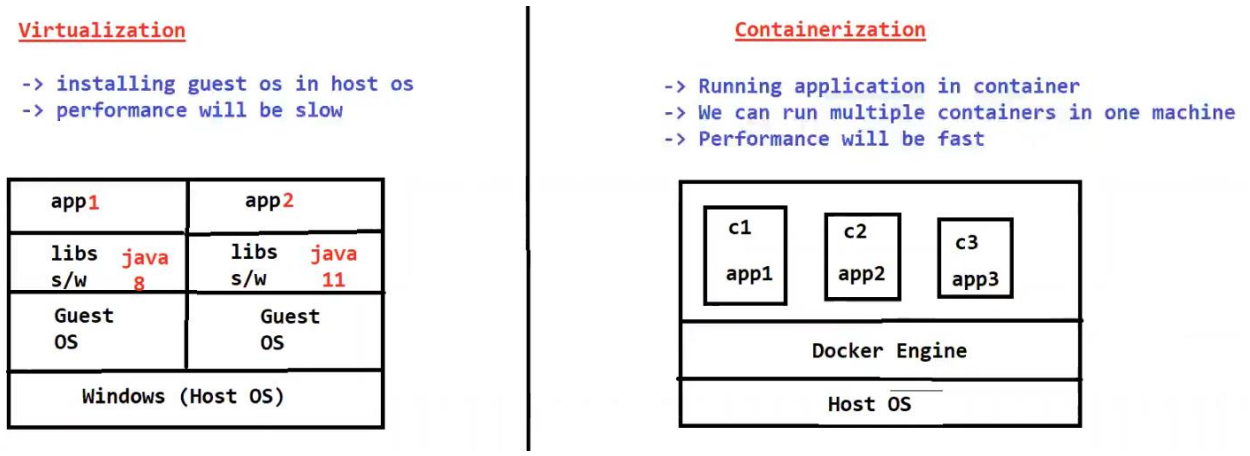
-> Docker is a containerization software

-> Using Docker we will create container for our application

-> Using Docker we will create image for our application

-> Docker images we can share easily to mulitple machines

-> Using Docker image we can create docker container and we can execute it



==========
Conclusion
==========

-> Docker is a containerization software

-> Docker will take care of application and application dependencies for execution

-> Application Deployments into multiple environments will become easy if we use Docker containers concept.

==========

 Launching EC2 Instance in AWS : https://youtu.be/uI2iDk8iTps

 Connect to Ec2 using Putty : https://youtu.be/GXc_bxmP0AA

==========

==========
Environment Setup
==========

1) Create Account in AWS Cloud
2) Create Linux Machine using AWS EC2 service (Image : Amazon Linux)
3) Connect to Linux Machine using MobaXterm / Putty
4) Install Docker software in Linux VM using below commands

++++++++ Install Docker in Amazon Linux ++++++++++++

$ sudo yum update -y
$ sudo yum install docker -y
$ sudo service docker start

# add ec2-user to docker group by executing below command
$ sudo usermod -aG docker ec2-user

#Restart the session
$ exit

$ docker info
Then press 'R' to restart the session (This is in MobaXterm)

++++++++++++++++++++Docker Commands++++++++++++++++++++

# see docker info
$ docker info

# To see docker images
$ docker images

# Pulling hello-world docker image
$ docker pull hello-world

# see docker image
$ docker images

# Running hello-world docker image
$ docker run hello-world

```
[ec2-user@ip-172-31-32-129 ~]$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.
```

# Display Running Docker containers
$ docker ps

# Displaying Running + stopped containers
$ docker ps -a

# Inspect docker image
$ docker inspect <image-id>

# Remove Docker image
$ docker rmi <image-name / image-id>

# Remove docker image forcefully
$ docker rmi -f <image-name / image-id>

# Stop the container
$ docker stop <container-id>

# Remove docker container
$ docker rm <container-id>

# Remove all stopped containers + un-used images + un-used networks
$ docker system prune -a

############## Note: Create account in Docker Hub (https://hub.docker.com/) ##############

============
Dockerfile
===========

=> Dockerfile contains set of instructions to build docker image

=> In Dockerfile we will use DSL (Domain Specific Language)

=> Docker Engine will read Dockerfile instructions from top to bottom to process

=> In Dockerfile we will use below keywords

1) FROM
2) MAINTAINER
3) COPY
4) ADD
5) RUN
6) CMD
7) ENTRYPOINT
8) ENV
9) ARG
10) WORKDIR
11) EXPOSE
12) VOLUME
13) USER
14) LABEL

=========
FROM
=========

=> It represents base image to create our docker image

Syntax:

FROM java:1.8
FROM python:1.2
FROM mysql:8.5
FROM tomcat:9.5

==========
Maintainer
==========

=> It is used to specify docker file author information

Syntax:

MAINTAINER Ashok <ashok.b@oracle.com>

======
COPY
======

=> It is used to copy the files from source to destination while creating docker image

Syntax:

COPY <SRC> <DESTINATION>

Ex: COPY target/sb-api.war /app/tomcat/webapps/sb-api.war

=====
ADD
=====

=> It is used to copy the files from source to destination while creating docker image

Syntax:

ADD <SRC> <DESTINATION>

ADD <HTTP-URL> <DESTINATION>

Ex: ADD <url> /app/tomcat/webapps/sb-api.war

==========
Q) What is the difference between COPY and ADD command ?
==========

COPY : It can copy from one path to another path with in the same machine.

ADD : It can copy from one path to another path & it supports URL also as source.

======
RUN
======

-> RUN instructions will execute while creating docker image

Syntax:

RUN yum install git
RUN yum install maven
RUN git clone <repo-url>
RUN cd <repo-name>
RUN mvn clean package

Note: We can write multiple RUN instructions, docker engine will process from top to bottom.

====
CMD
====

=> CMD instructions will execute while creating docker container

=> Using CMD command we can run our application in container

Syntax:

CMD sudo start tomcat

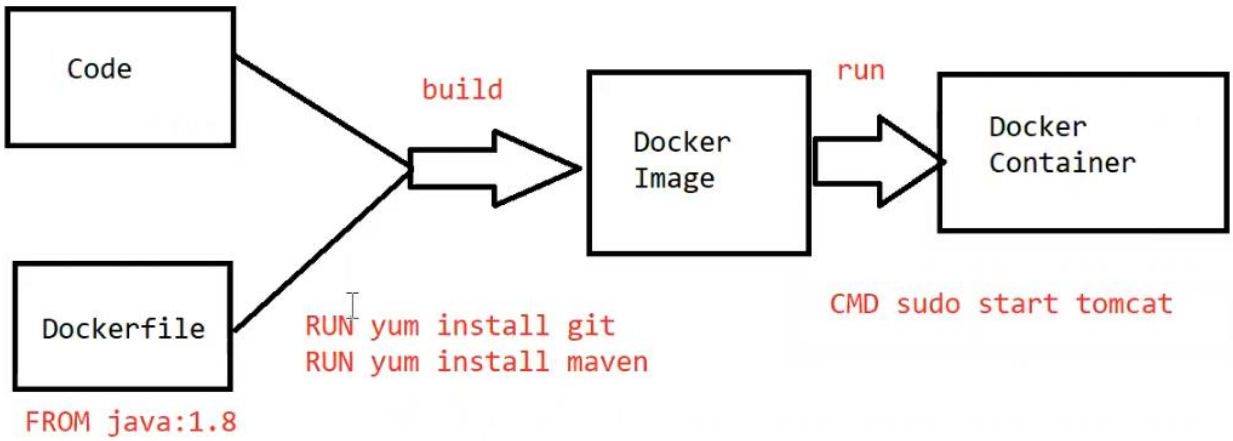
CMD java -jar <jar-file>

Note: We can write multiple CMD instructions but docker engine will process only last CMD instruction.

==========
Q) What is the difference between RUN and CMD ?
==========

-> RUN instructions will execute while creating docker image
=> CMD instructions will execute while creating docker container

=> We can write multiple RUN instructions, docker engine will process from top to bottom.
=> We can write multiple CMD instructions but docker engine will process only last CMD instruction.
Note: There is no use of writing multiple CMD instructions.



==========
Sample Docker
==========

FROM ubuntu

MAINTAINER Ashok<ashok.b@oracle.com>

RUN echo "Hi, i am run - 1"
RUN echo "Hi, i am run - 2"
RUN echo "Hi, i am run - 3"

CMD echo "Hi, i am CMD-1"
CMD echo "Hi, i am CMD-2"
CMD echo "Hi, i am CMD-3"

=> Create a file (filename: Dockerfile)
$ vi Dockerfile

=> Copy above sample docker file content and keep in docker file (Esc + :wq)

Create docker image using Dockerfile

$ docker build -t <image-name> .

[t >Tag Name ; . -> Current working directory is where my dockerfile is present]

# login into docker hub account from docker machine
$ docker login

Note: Enter your docker hub account credentials.

# tag docker image
$ docker tag <image-name> <tagname>

Ex : $ docker tag myimg1 ashokit/myimg1

Push Docker image

$ docker push <Tag-name>

==========
Q) Can we use user defined name for Dockerfile ?

Ans) Yes, we can do it.

$ docker build -f <filename> -t <imagename> .
==========

==========
ENTRYPOINT
==========

=> It is used to execute instructions while creating container

Synatx:

ENTRYPOINT ["echo" , "Container Created Successfully"]

ENTRYPOINT ["java", "-jar", "target/springboot.jar"]

==========
Q) What is the difference between CMD and ENTRYPOINT ?
==========

=> CMD instructions we can override while creating container

=> ENTRYPOINT instructions we can't override

=========
WORKDIR
=========

=> It is used to specify working directory for image and container

Syntax:

WORKDIR /app/usr/

======
ENV
======

=> ENV is used to set Environment Variables

Ex:

ENV java /etc/softwares/jdk

=========
EXPOSE
=========

=> It is used to specify on which port number our docker container will run

EX:

EXPOSE 8080

====
ARG
====

=> By using ARG we can take dynamic values from CLI

=> It is used to remove hard coded values in Dockerfile

Ex:

ARG branch

RUN git clone -b $branch <repo-url>

$ docker build -t <imagename> --build-arg branch=master .

======
USER
======

=> It is used to specify username for creating image / container

Ex:

USER dockeruser

========
VOLUME
========

=> It is used to specify docker volume storage location

=> Volumes are used for storage purpose

=========
LABEL
=======
=> It is used to add METADATA to docker objects in key-value format

Ex:

LABEL name="sbi_image"

1) What is an application ?

2) Application Tech stack ?

3) Application Architecture

4) Life without Docker

5) Life with Docker

6) What is Docker

7) What is Virtualization ?

8) What is Containerization ?

9) Docker Architecture ?

10) Docker Installation in Linux

11) Docker Image Creation

12) Docker Container Creation

13) Dockerfile creation

14) Docker Registry (push image to registry)

15) Dockerfile Keywords

==========
Java applications Dockerization
==========

=> We can see two types of java applications in the companies

1) Normal Java Application without Spring Boot

2) Java Application with Spring Boot

Note: Spring Boot is ready made framework to make java application development simple.

=> Normal Java Web Applications will be packaged as war file and war file will be deployed in webserver
(Ex: tomcat)

=> Spring Boot applications will be packaged as jar file and we need to run the jar file. It will take care of server internally (Embedded Server).

Java Maven Web App Git Repo : https://github.com/ashokitschool/maven-web-app.git

==========Dockerfile For Java Web Application==========

```
FROM tomcat:8.0.20-jre8

COPY target/app.war  /usr/local/tomcat/webapps/app.war

EXPOSE 8080
```

==========Working Procedure==========

Connect to Docker Machine

```
$ sudo service docker start
$ sudo yum install git
$ sudo yum install maven
$ git clone https://github.com/ashokitschool/maven-web-app.git
$ cd maven-web-app
$ mvn clean package
$ ls -l target
$ docker build -t maven-web-app .
$ docker images
$ docker run -d -p 8080:8080 maven-web-app                    [Host portNo: Container portNo]
$ docker ps
```

Note: Enable 8080 Port in Security Group Inbound Rules (Custom TCP - 8080) which is attached to docker machine.

Type : Custom TCP
Port Range : 8080
Source : Anywhere IPv4

=> Access our application in browser

URL : http://ec2-vm-publicip:8080/maven-web-app/

Note: if we use this command "$ docker run -p 8080:8080 maven-web-app" the terminal will not be open for other commands. So we must run the container in the detach mode by using this commands "$ docker run -d -p 8080:8080 maven-web-app"
-d -> To Run the container in detach mode
-p -> To perform the port mapping

==========
Dockerizing Spring Boot Application
==========

Spring Boot App Git Repo : https://github.com/ashokitschool/spring-boot-docker-app.git

=> Spring Boot app will be packaged as jar file (even if it is web app)

=> Spring Boot will have embedded tomcat server to run

=> We no need to deploy Spring Boot app in server manually.

=> We just need to run spring boot app jar file, it will care of server and deployment.

======Spring Boot Application Dockerfile=====

```
FROM openjdk:11

COPY target/app.jar  /usr/app/app.jar

WORKDIR /usr/app/

EXPOSE 8080

ENTRYPOINT [ "java", "-jar", "app.jar" ]
```

==========Working Procedure=====

```
$ git clone https://github.com/ashokitschool/spring-boot-docker-app.git
$ cd spring-boot-docker-app
$ ls -l
$ mvn clean package
$ ls -l target
$ docker images
$ docker build -t sbapp .
$ docker images
$ docker run -d -p 9090:8080 sbapp
$ docker ps
```

Note: Enable 9090 port in security group of docker machine

=> Access the application in browser

URL : http://ec2-public-ip:9090/

==========

==========
Python App with Docker
==========

=> Python is a general purpose scritping language

=> Python programs will have .py extension

=> Compilation is not required for Python programs

========== Python App Dockerfile ==========

```
FROM python:3.6

MAINTAINER Ashok <ashok.b@oracle.com>

COPY . /app/

WORKDIR /app/

EXPOSE 5000

RUN pip install -r requirements.txt

ENTRYPOINT [ "python", "app.py" ]
```

==========
Working Procedure
==========

```
$ docker system prune -a

$ git clone https://github.com/ashokitschool/python-flask-docker-app.git

$ cd python-flask-docker-app

$ ls -l

$ cat Dockerfile

$ docker build -t python-flask-app .

$ docker images

$ docker run -d -p 5000:5000 python-flask-app

$ docker ps
```

Note: As we have mapped container port 5000 to host port 5000 we need to enable 5000 port in security group.

=> Access Python Application in Browser

URL : http://ec2-public-ip:5000/

==========Troubleshooting==========

# Print container logs
$ docker logs <container-id>

# Get into docker container
$ docker exec -it <container-id> /bin/bash

To come out from container use 'exit' command

==========

```
[ec2-user@ip-172-31-13-66 python-flask-docker-app]$ sudo docker exec -it 451e78b9de19 /bin/bash
root@451e78b9de19:/app# ls -l
total 16
-rw-r--r--. 1 root root 181 Jan  2 08:51 Dockerfile
-rw-r--r--. 1 root root 323 Jan  2 08:51 README.md
-rw-r--r--. 1 root root 270 Jan  2 08:51 app.py
-rw-r--r--. 1 root root   6 Jan  2 08:51 requirements.txt
root@451e78b9de19:/app# cd ..
root@451e78b9de19:/# ls -l
total 0
drwxr-xr-x.   3 root root  91 Jan  2 09:06 app
drwxr-xr-x.   1 root root 179 Dec 21  2021 bin
drwxr-xr-x.   2 root root   6 Dec 11  2021 boot
drwxr-xr-x.   5 root root 340 Jan  2 09:17 dev
drwxr-xr-x.   1 root root  66 Jan  2 09:17 etc
drwxr-xr-x.   2 root root   6 Dec 11  2021 home
drwxr-xr-x.   1 root root  41 Dec 21  2021 lib
drwxr-xr-x.   2 root root  34 Dec 20  2021 lib64
drwxr-xr-x.   2 root root   6 Dec 20  2021 media
drwxr-xr-x.   2 root root   6 Dec 20  2021 mnt
drwxr-xr-x.   2 root root   6 Dec 20  2021 opt
dr-xr-xr-x. 172 root root   0 Jan  2 09:17 proc
drwx------.   1 root root  20 Jan  2 09:06 root
drwxr-xr-x.   3 root root  30 Dec 20  2021 run
drwxr-xr-x.   1 root root  20 Dec 21  2021 sbin
drwxr-xr-x.   2 root root   6 Dec 20  2021 srv
dr-xr-xr-x.  13 root root   0 Jan  2 08:47 sys
drwxrwxrwt.   1 root root   6 Jan  2 09:06 tmp
drwxr-xr-x.   1 root root  19 Dec 20  2021 usr
drwxr-xr-x.   1 root root  19 Dec 20  2021 var
root@451e78b9de19:/# exit
exit
[ec2-user@ip-172-31-13-66 python-flask-docker-app]$
```

==========
React JS with Docker
==========

=> React JS is a java script library

=> React JS is used to develop Front end of the application (user interface)

=> React JS will use Node Package Manager to install required sotwares

React App Git Repo : https://github.com/ashokitschool/React_App.git

========== React JS App Dockerfile ==========

```
FROM node:latest
WORKDIR /app
COPY package.json ./
RUN npm install
COPY . .
CMD ["npm", "start"]
```

==========

==========
Docker Network
==========

=> Network is all about communication

=> Docker Network is used to provided isolated network for Docker Container

=> In Docker we have below 3 default networks

1) bridge
2) host
3) none

=> In Docker we have network drivers

1) Bridge ---> This is default network driver
2) Host
3) None
4) Overlay ----> Docker Swarm
5) Macvlan

-> Bridge driver is recommended driver when we are using standalone container. It will assign one IP for for our docker container.

-> Host driver is also used to run standalone container but it will not assign any IP for container.

-> None means no network will be available for docker container.

-> Overlay network driver is used for Orchestration. Docker swarm will use this overlay driver

-> Macvlan network driver provide physical IP for container.

# Display Docker Networks
$ docker network ls

# Create docker network
$ docker network create ashokit-nw

# Inspect Docker Network
$ docker network inspect ashokit-nw

# Run Docker container with our network
$ docker run -d -p 9090:9090 --network ashokit-nw ashokit/spring-boot-rest-api

# Delete Docker network
$ docker network rm ashokit-nw

#Stop the Running Docker Containerization
$ docker stop <Container-id>

Note: After stopping the container only we can delete the network. We cannot delete the network if our container is in running state.

#Clear Everything
$ docker system prune -a

==========
Docker Compose
==========

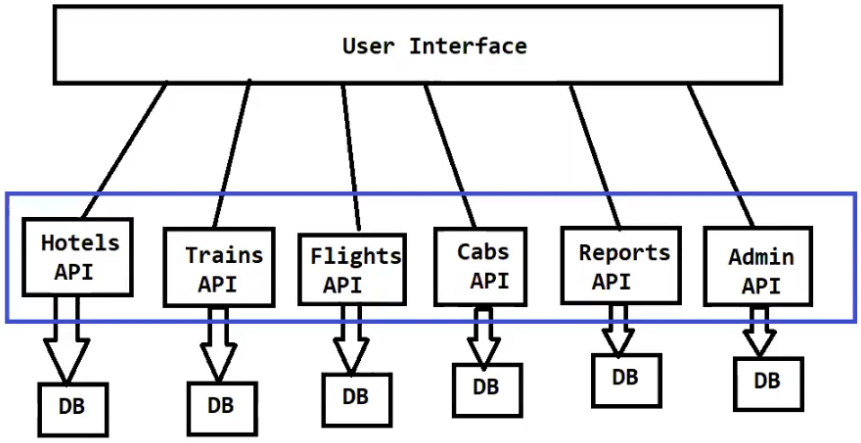
=> Now a days projects are developing based on Microservices Architecture

=> Our application requires multiple containers for execution

a) Frontend app container
b) Backend app containers (microservices)
c) DB containers

-> Creating multiple containers manually is very difficult and time taking process.

Note: Managing "Multi - Container" based applications is difficult task.



Frontend : 1 container

Backend : 6 containers

Database : 6 containers

Total : 13 containers

################ Docker Compose is used for Managing Multiple - Containers ################

=> Docker compose is a tool which is used to manage multi container based applications

=> Using Docker compose we can easily setup & deploy mulitple containers

=> We will use "docker-compose.yml" file to provide containers information to Docker Compose tool

=> Docker Compose YML should contain all the information related to containers creation.

==========
Docker Compose YML File
==========

version :

services :

network:

volumes:

Note: Docker Compose Default file name is "docker-compose.yml" (we can change it also)

=> Docker Compose file we will keep in source code repository.

Note: Difference between Dockerfile and docker-compose.yml filename ?
Dockerfile is used to create docker image whereas docker-compose.yml is used to manage multiple containers.

==========
Installing Docker Compose
==========

download docker compose
$ sudo curl -L "https://github.com/docker/compose/releases/download/1.24.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

# Give permission
$ sudo chmod +x /usr/local/bin/docker-compose

# How to check docker compose is installed or not
$ docker-compose --version

==========
Deploy Spring Boot + MySQL with Docker Compose
==========

$ git clone https://github.com/ashokitschool/spring-boot-mysql-docker-compose.git
$ cd spring-boot-mysql-docker-compose
$ mvn clean package
$ docker build -t spring-boot-mysql-app .
$ docker images
$ docker-compose up -d
$ docker-compose ps

=> Access the application

URL : http://ec2-public-ip:8080/

# check app container logs
$ docker logs <app-container-name>

# Connect to DB Container
$ docker exec -it <db-container-name> /bin/bash

# connect with mysql db using mysql client
$ mysql -u root -p

# display databases available in mysql
$ show databases

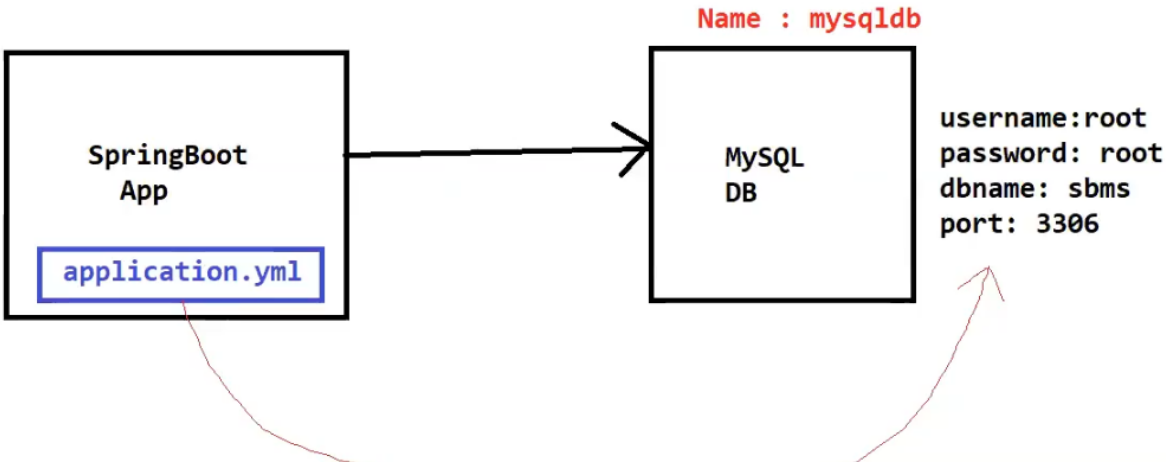
# select db name (sbms is our db name)
$ use sbms

# display tables created in database
$ show tables

# Display table data(book is our tablename)
$ select * from book;

```
# exit from database
$ exit

# exit from container
$ exit


========================
Docker Compose Commands
========================

# Create Containers using Docker Compose
$ docker-compose up

# Create Containers using different file name
$ docker-compose -f <filename> up

# Run docker containers in detached mode
$ docker-compose up -d

# Display containers created by docker compose
$ docker-compose ps

# Display docker images
$ docker-compose images

# Check container logs
$ docker logs -f <container-name>

# Stop docker containers
$ docker-compose stop

# Start docker containers
$ docker-compose start

# Stop & remove docker containers
$ docker-compose down

=====================================================

version: "3"
services:
  application:
    image: springboot-mysql-app
    ports:
      - 8080:8080
    networks:
      - springboot-db-net
    depends_on:
      - mysqldb
    volumes:
```

```
    - /data/springboot-app
  mysqldb:
    image: mysql:5.7
    networks:
      - springboot-db-net
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=sbms
    volumes:
      - /data/mysql
networks:
 springboot-db-net:
==========================================
```



main spring-boot-mysql-docker-compose / src / main / resources / application.yml

ashokitschool Update application.yml

1 contributor

10 lines (10 sloc) | 212 Bytes

```yaml
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://mysqldb:3306/sbms
    username: root
    password: root
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
```

==========
Common problems occurs while executing docker-compose.yml files
==========

[Configured wrong database Name]

Consider in our project “spring-boot-mysql-docker-compose/src/main/resources/application.yml” you have configured different database name “://mydb:3306/sbms” then the name that you have configured in your docker-compose.yml “mysqldb:” then it will create only one container i.e mysqldb(DATABASE) container and the container in which our project is their will be failed to create.

$ mvn clean package
$ docker build -t spring-boot-mysql-app .
$ docker images
$ docker-compose up -d
$ docker-compose ps

```
[ec2-user@ip-172-31-34-24 spring-boot-mysql-docker-compose]$ docker-compose ps
                  Name                                Command             State          Ports
-------------------------------------------------------------------------------------------------------
spring-boot-mysql-docker-compose_application_1   java -jar /spring-boot-mys ...   Exit 1
spring-boot-mysql-docker-compose_mysqldb_1       docker-entrypoint.sh mysqld      Up       3306/tcp, 33060/tcp
[ec2-user@ip-172-31-34-24 spring-boot-mysql-docker-compose]$
```

Here we can see that application container is failed to start.

Check the logs of the container
$ docker logs –f spring-boot-mysql-docker-compose_application_1

```
.22]
        at com.mysql.cj.jdbc.ConnectionImpl.createNewIO(ConnectionImpl.java:826) ~[mysql-c
        ... 58 common frames omitted
Caused by: java.net.UnknownHostException: mydb: Name or service not known
        at java.net.Inet4AddressImpl.lookupAllHostAddr(Native Method) ~[na:1.8.0_342]
        at java.net.InetAddress$2.lookupAllHostAddr(InetAddress.java:929) ~[na:1.8.0_342]
        at java.net.InetAddress.getAddressesFromNameService(InetAddress.java:1330) ~[na:1.
        at java.net.InetAddress.getAllByName0(InetAddress.java:1283) ~[na:1.8.0_342]
        at java.net.InetAddress.getAllByName(InetAddress.java:1199) ~[na:1.8.0_342]
        at java.net.InetAddress.getAllByName(InetAddress.java:1127) ~[na:1.8.0_342]
        at com.mysql.cj.protocol.StandardSocketFactory.connect(StandardSocketFactory.java:
ar!/:8.0.22]
```

Now You have make the necessary changes like you have updated the correct Database Name in “application.yml” file. After that it will create only application container bcz our Database container is up and running.

```
[ec2-user@ip-172-31-34-24 spring-boot-mysql-docker-compose]$ docker-compose up -d
spring-boot-mysql-docker-compose_mysqldb_1 is up-to-date
Recreating spring-boot-mysql-docker-compose_application_1 ... done
[ec2-user@ip-172-31-34-24 spring-boot-mysql-docker-compose]$
```

```
[ec2-user@ip-172-31-34-24 spring-boot-mysql-docker-compose]$ docker-compose ps
                 Name                               Command             State                    Ports
-------------------------------------------------------------------------------------------------------------------------
spring-boot-mysql-docker-                  java -jar /spring-boot-mys ...   Up      0.0.0.0:8080->8080/tcp,:::8080->8080/tcp
compose_application_1
spring-boot-mysql-docker-compose_mysqldb_1   docker-entrypoint.sh mysqld      Up      3306/tcp, 33060/tcp
[ec2-user@ip-172-31-34-24 spring-boot-mysql-docker-compose]$
[ec2-user@ip-172-31-34-24 spring-boot-mysql-docker-compose]$
[ec2-user@ip-172-31-34-24 spring-boot-mysql-docker-compose]$
```

==========
Stateful Vs Stateless Containers
==========

Stateless container : Data will be deleted after container got deleted

Stateful Container : Data will be maintained permanently

Note: Docker Containers are stateless container (by default)

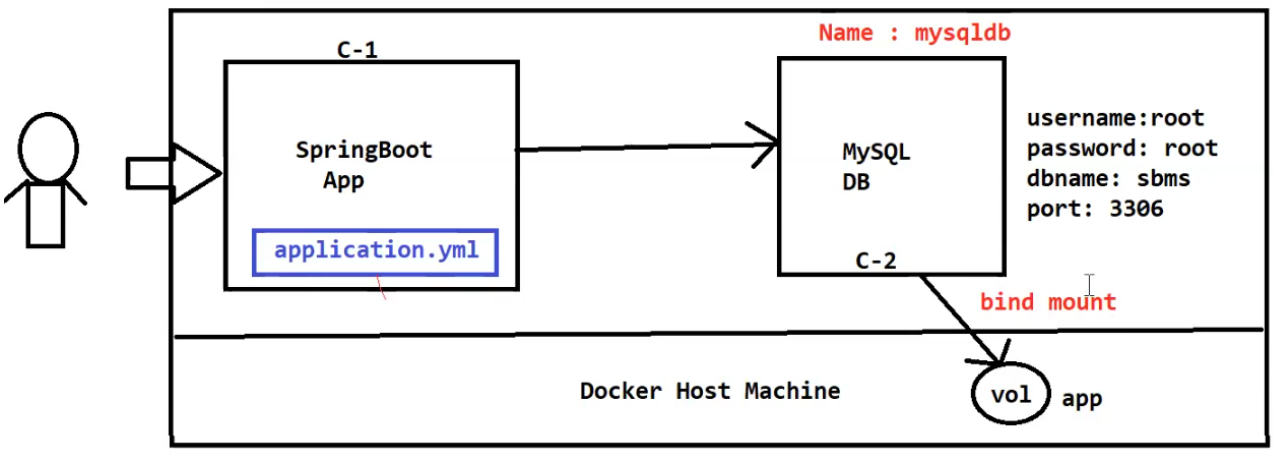
Note: In above springboot application we are using mysql db to store the data. When we re-create containers we lost our data (This is not accepted in realtime).

=> Even if we deploy latest code or if we re-create containers we should not loose our data.

=> To maintain data permenently we need to make our container as Stateful Container.

=> To make container as stateful, we need to use Docker Volumes concept.

==========
Docker Volumes
==========



=> Volumes are used to persist the data which is generated by Docker container

=> Volumes are used to avoid data loss

=> Using Volumes we can make container as stateful container

=> We have 3 types of volumes in Docker

1) Anonymous Volume (No Name)
2) Named Volume
3) Bind Mounts

# Display docker volumes
$ docker volume ls

# Create Docker Volume
$ docker volume create <vol-name>

# Inspect Docker Volume
$ docker volume inspect <vol-name>

# Remove Docker Volume
$ docker volume rm <vol-name>

# Remove all volumes
$ docker system prune --volumes

==========
Making Docker Container State-full using Bind Mount
==========

=> Create a directory on host machine

$ mkdir app

=> Map 'app' directory to container in docker-compose.yml file like below

```
version: "3"
services:
  application:
    image: spring-boot-mysql-app
    ports:
      - "8080:8080"
    networks:
      - springboot-db-net
    depends_on:
      - mysqldb
    volumes:
      - /data/springboot-app

  mysqldb:
    image: mysql:5.7
    networks:
      - springboot-db-net
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=sbms
```

```
    volumes:
      - ./app: /var/lib/mysql
networks:
  springboot-db-net:
```

=> Start Docker Compose Service

```
$ docker-compose up -d
```

=> Access the application and insert data

=> Delete Docker Compose service using below command

```
$ docker-compose down
```

=> Again start Docker Compose service

```
$ docker-compose up -d
```

=> Access application and see data (it should be available)

==========

==========
Docker Swarm
==========

-> It is a container orchestration sofware

-> Orchestration means managing processes

-> Docker Swarm is used to setup Docker Cluster

-> Cluster means group of servers

-> Docker swarm is embedded in Docker engine (No need to install Docker Swarm Seperatley)

-> We will setup Master and Worker nodes using Docker Swarm cluster

-> Master Node will schedule the tasks (containers) and manage the nodes and node failures

-> Worker nodes will perform the action (containers will run here) based on master node instructions

==========
Swarm Features
==========
1) Cluster Management
2) Decentralize design
3) Declarative service model
4) Scaling

5) Multi Host Network
6) Service Discovery
7) Load Balancing
8) Secure by default
9) Rolling Updates

==========
Docker Swarm Cluster Setup
==========

-> Create 3 EC2 instances (ubuntu) & install docker in all 3 instances using below 2 commands

$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sudo sh get-docker.sh

Note: Enable 2377 port in security group for Swarm Cluster Communications

1 - Master Node
2 - Worker Nodes

-> Connect to Master Machine and execute below command

# Initialize docker swarm cluster
$ sudo docker swarm init --advertise-addr <private-ip-of-master-node>

Ex : $ sudo docker swarm init --advertise-addr 172.31.37.100

# Get Join token from master (this token is used by workers to join with master)
$ sudo docker swarm join-token worker

Note: Copy the token and execute in all worker nodes with sudo permission

Ex: sudo docker swarm join --token SWMTKN-1-
4pkn4fiwm09haue0v633s6snitq693p1h7d1774c8y0hfl9yz9-8l7vptikm0x29shtkhn0ki8wz
172.31.37.100:2377

Q) what is docker swarm manager quarm?

Ans) If we run only 2 masters then we can't get High Availability

Formula : (n-1)/2

If we take 2 servers

2-1/2 => 0.5 (It can't become master)

3-1/2 => 1 (it can be leader when the main leader is down)

Note: Always use odd number for Master machines

-> In Docker swarm we need to deploy our application as a service.

==========
Docker Swarm Service
==========

-> Service is collection of one or more containers of same image

-> There are 2 types of services in docker swarm

1) Replica (default mode)
2) global

$ sudo docker service create --name <serviceName> -p <hostPort>:<containerPort> <imageName>

Ex : $ sudo docker service create --name java-web-app -p 8080:8080 ashokit/javawebapp

Note: By default 1 replica will be created

Note: We can access our application using below URL pattern

URL : http://master-node-public-ip:8080/java-web-app/

# check the services created
$ sudo docker service ls

# we can scale docker service
$ docker service scale <serviceName>=<no.of.replicas>

# inspect docker service
$ sudo docker service inspect --pretty <service-name>

# see service details
$ sudo docker service ps <service-name>

# Remove one node from swarm cluster
$ sudo docker swarm leave

# remove docker service
$ sudo docker service rm <service-name>
==========

==========
Docker Compose v/s Docker swarm
==========
Docker Compose is used to create multiple container for our application. It is used to manage multi-container based application. Whereas Docker swarm is called as orchestrator platform, it is used to balance the load, it is used to manage the container i.e if we want to scale up/scale down the containers.

=========
Summary
=========

1) What is Application Stack
2) Life without Docker
3) Life with Docker
4) Docker introduction
5) Virtualization vs Containerization
6) Docker Installation in Linux
7) Docker Architecture
8) Docker Terminology
9) Dockerfile & Dockerfile Keywords
10) Writing Dockerfiles
11) Docker image commands
12) Docker container commands
13) Dockerizing Java Spring Boot Application
14) Dockerizing Java Web Application with External Tomcat
15) Dockerizing Python Flask Application
16) Docker Network
17) Monlolith Vs Microservices
18) Docker Compose
19) Docker Compose File Creation
20) Docker Volumes
21) Spring Boot with MySQL DB Dockerization using Docker Compose
22) What is Orchestration ?
23) Docker Swarm
24) Docker Swarm Cluster Setup
25) Deployed java web app as docker container using swarm cluster

$ docker info
$ docker images
$ docker rmi <imagename>
$ docker pull <imagename>
$ docker run <imagename>
$ docker run -d -p host-port : container-port <image-name>
$ docker tag <image-name> <image-tag-name>
$ docker login
$ docker push <image-tag-name>

$ docker ps
$ docker ps -a
$ docker stop <container-id>

```
$ docker rm <container-id>
$ docker rm -f <container-id>
$ docker system prune -a
$ docker logs <container-id>
$ docker exec -it <container-id> /bin/bash

$ docker network ls
$ docker network create <network-name>
$ docker network rm <network-name>
$ docker network inspect <network-name>

$ docker-compose up -d
$ docker-compose down
$ docker-compose ps
$ docker-compose images
$ docker-compose stop
$ docker-compose start

$ docker volume ls
$ docker volume create <vol-name>
$ docker volume inspect <vol-name>
$ docker volume rm <vol-name>
$ docker system prune --volumes

$ sudo docker service --name <service-name> -p 8080:8080 <img-name>
$ sudo docker service scale <service-name = replicas
$ sudo docker service ls
$ sudo docker service rm <service-name>
```