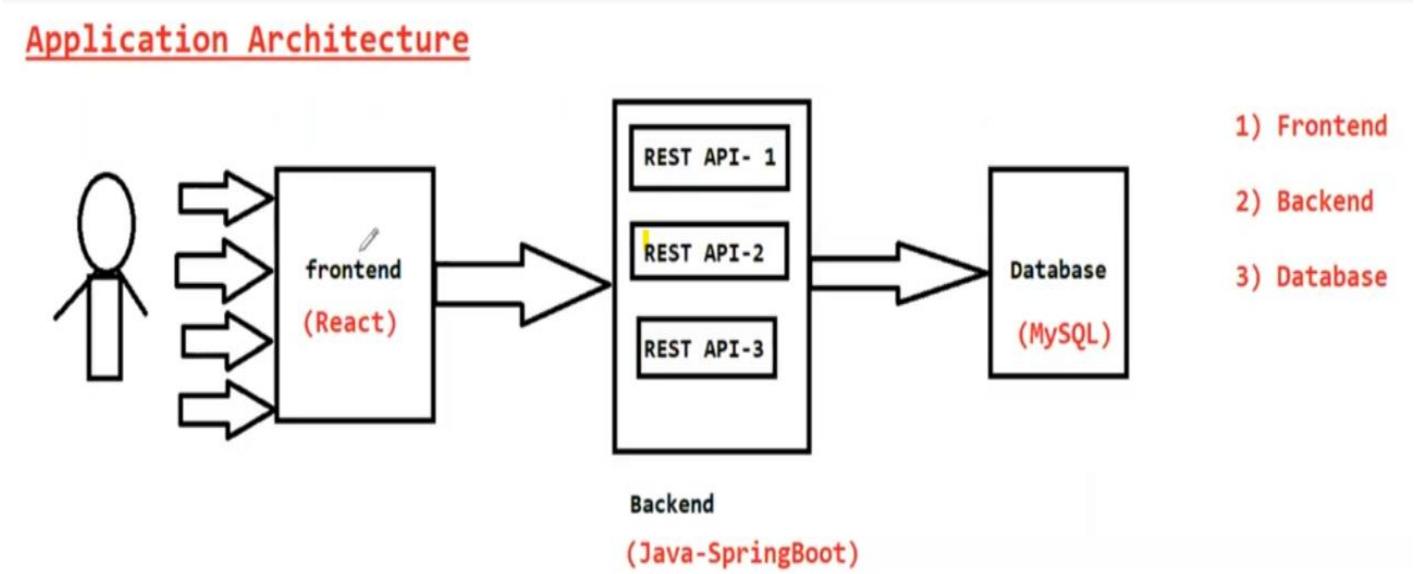
==========

Docker & Kubernetes

==========

Application Tech Stack : It represents technologies used in the application

1) Frontend Stack : HTML, CSS, JS, BS & Angular / React JS
2) Backend Stack : Java / .Net / Python / Node JS
3) Database : Oracle / MySQL / PostgresSQL / Mongo DB



========

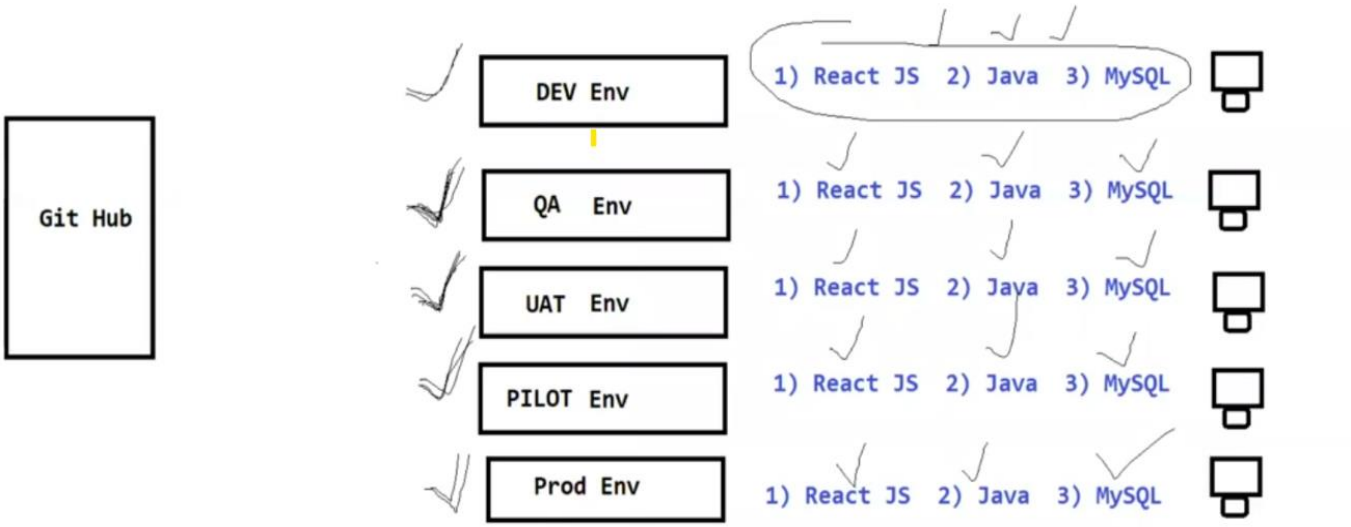
Docker

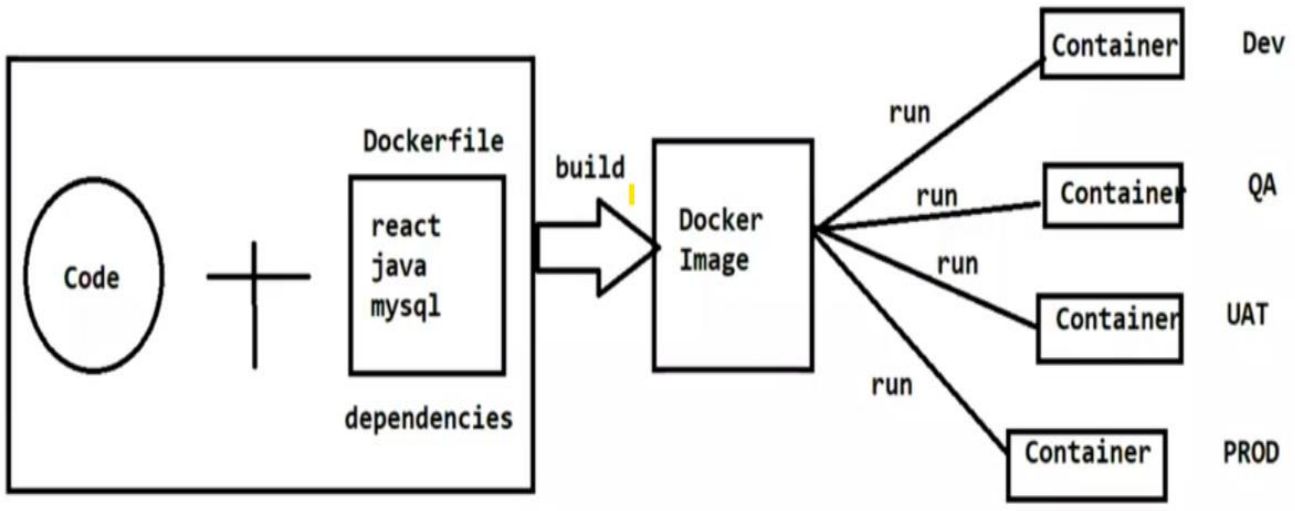
========

-> Docker is a containizeration platform
-> Docker is used to simplify application deployment process in Multiple Environments
(DEV, SIT, UAT, PILOT and PROD)

-> Docker is used to package application code + application dependencies for easy execution

-> Using Docker we will create Docker images
-> Docker Image contains App code + App dependencies
-> We can run docker image in any machine. It will take care of dependencies & execution
-> When we run Docker image, it will create Docker container
-> Docker Container will run our application





==========

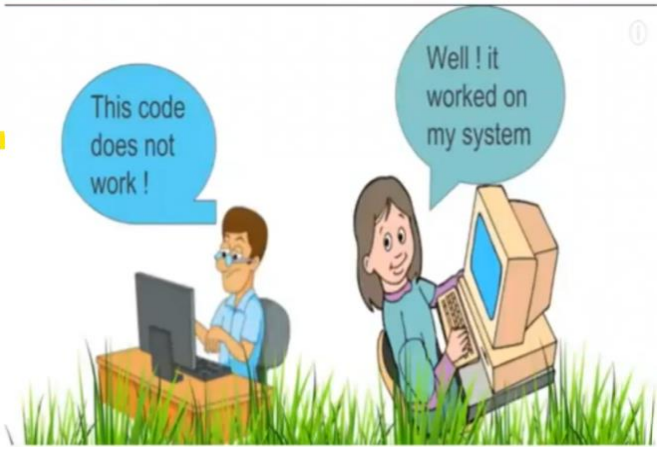
Virtualization

==========

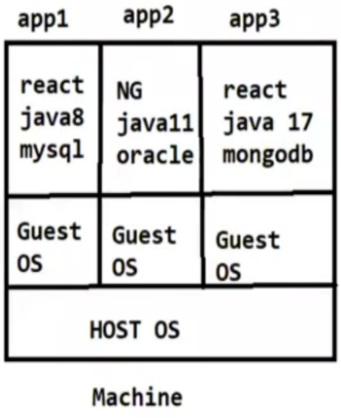
-> Installing Multiple Guest Operating Systems in one Host Operating System
-> Hypervisior S/w will be used to achieve this.
-> We need to install all the required softwares in HOST OS to run our application
-> It is old technique to run the applications
-> System performance will become slow in this process
-> To overcome the problems of Virtualization we are going for Containerization concept

Issues with Development Stack Setup

- You use a Mac OS , they use Windows
- You developed the app using Java 1.8v, they have Java 11v installed
- You used MongoDB v3.6, they're using v4.2



Virtualization



-> Virtualization means using multiple Operating Systems in same machine

-> Hypervisior s/w will be used to create virtualization

-> It is old technique to run multiple apps on same machine

-> System performance will become very slow with Virtualization

-> To overcome Virtualization problems we are going for Containerzation

==========

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

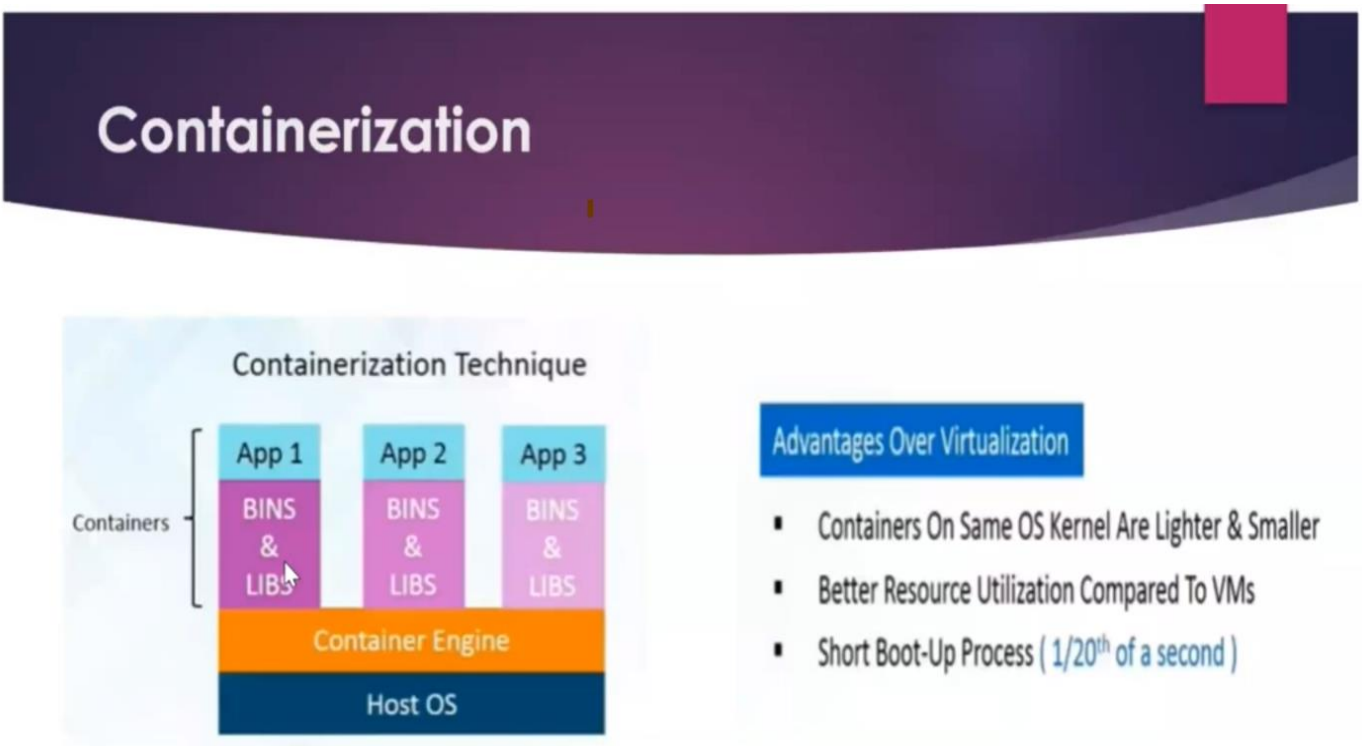
-> Docker is a containerization software

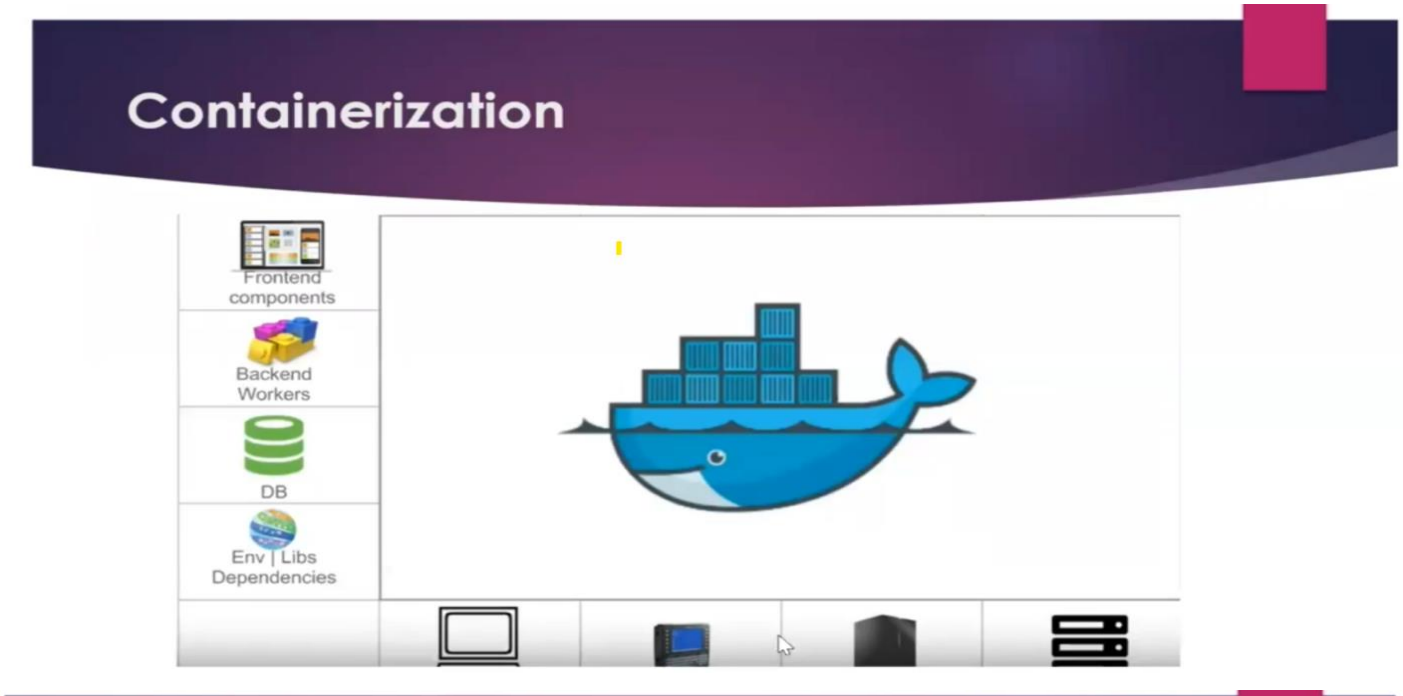
-> Using Docker we will create container for our application

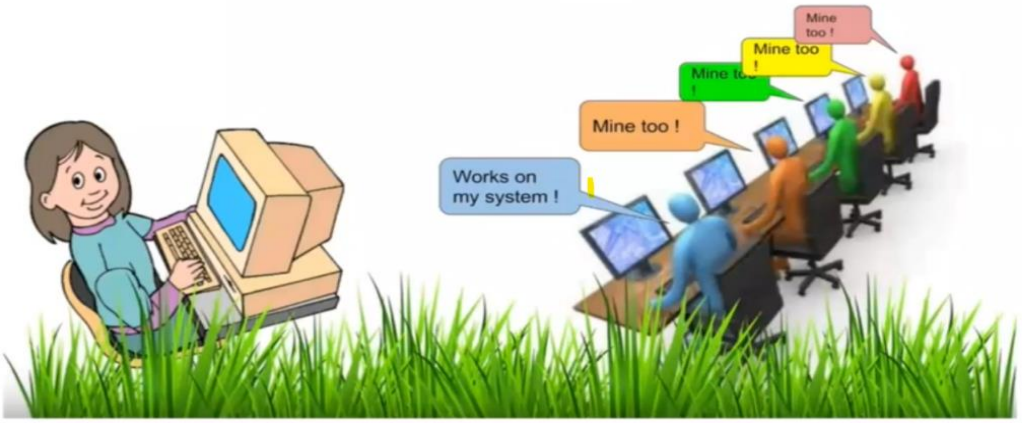
-> Using Docker we will create image for our application

-> 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

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

Container = Application Code + Application Libraries + Application Dependencies

-> Docker is a containerization software
-> Using Docker we will create Docker image

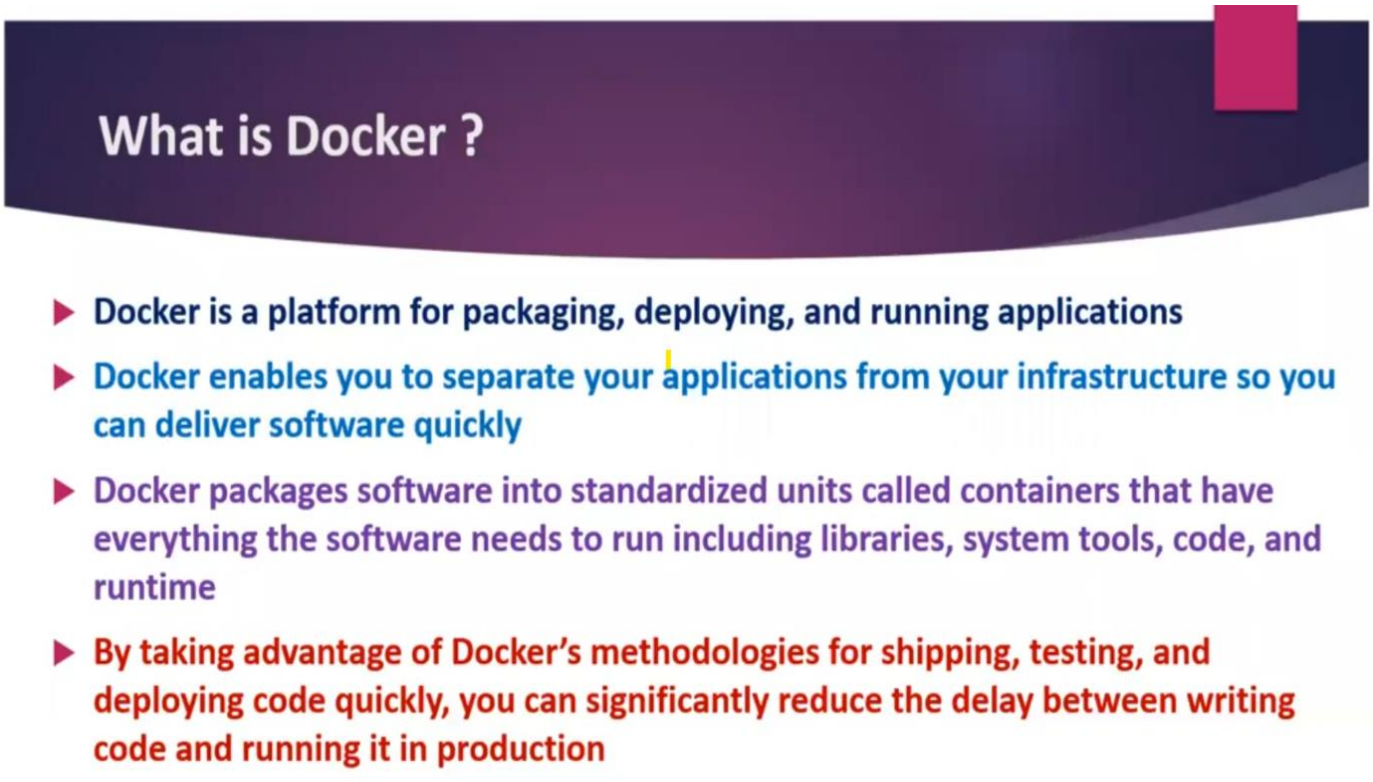
Docker Image = Application code + Application libs (maven dependencies) +
Application Dependencies(java, tomact, mysql etc...)

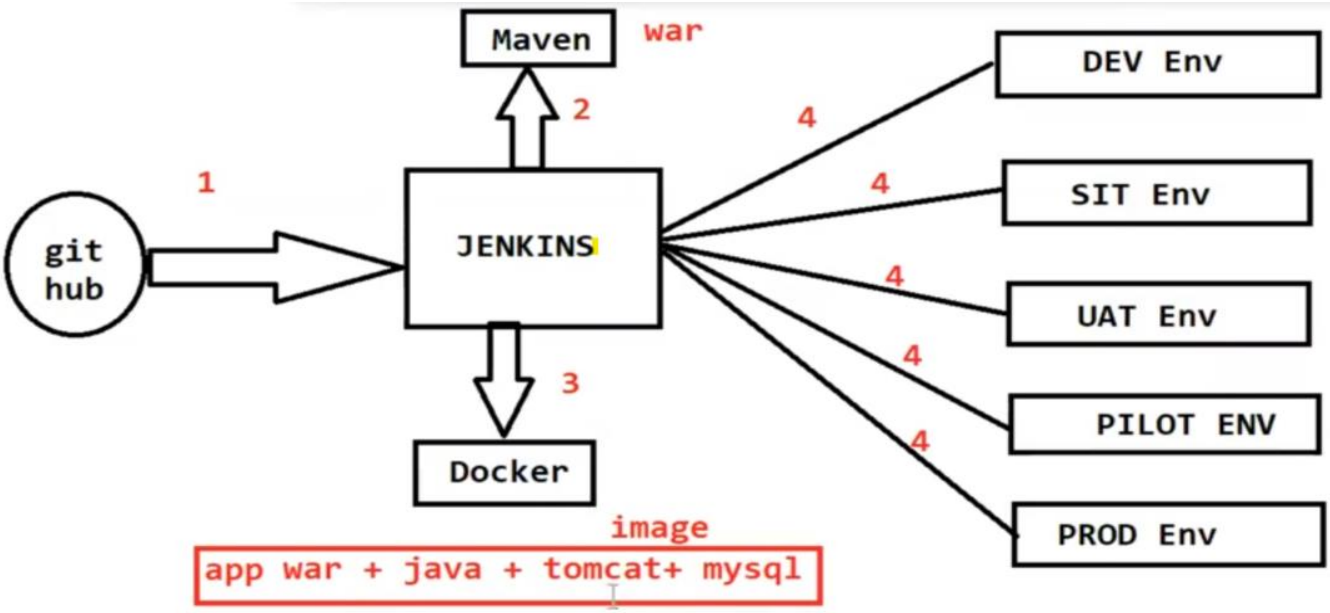
-> Once docker image is created then we can use jenkins to run docker image in multiple machines

-> Jenkins is just deployment software. We will use jenkins to run docker images in all environments

-> When we run Docker image it will create Docker container

-> Docker Container means Runtime instance of our application



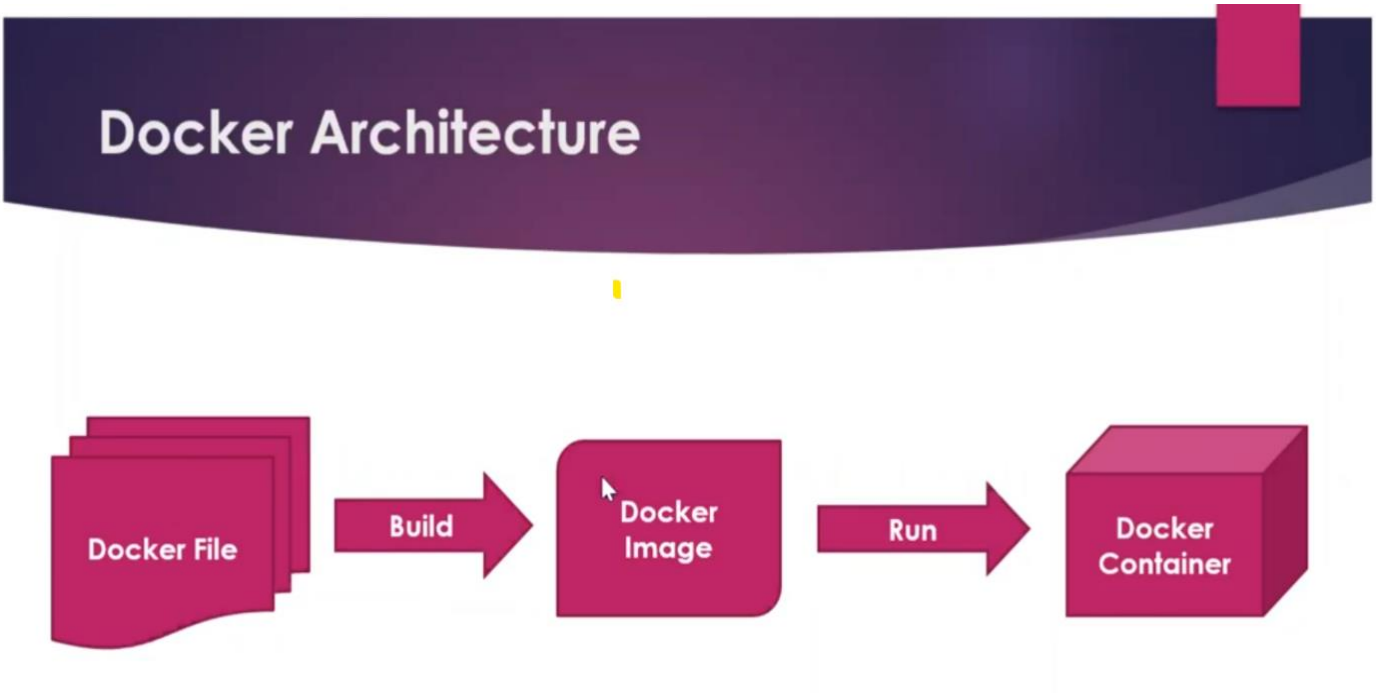


==========

Docker Architecture ?

==========

1) Dockerfile
2) Docker Imag
3) Docker Registry
4) Docker Container



-> Dockerfile contains set of instructions to build docker image. In dockerfile we will specify which softwares are required to run our code/application.

-> Docker image means a package which contains application code + app libs + app dependencies

-> Docker Registry is a place which is used to store docker images for future purpose

Ex: Docker Hub, Amazon ECR etc....

-> Docker container is runtime instance of our application. When we run Docker Image it will create Docker Container. Inside Container our application and application dependencies will be available.

1) What is Docker
2) What is Virtualization
3) What is Containerization
4) What is Docker Architecture
5) Dockerfile
6) Docker Image
7) DockerHub
8) Docker Container

==========

Install Docker in Linux VM

==========

-> Loging into AWS account

-> Create Linux Virtual Machine using Amazon Linux AMI

-> Connect to Linux VM using MobaXterm

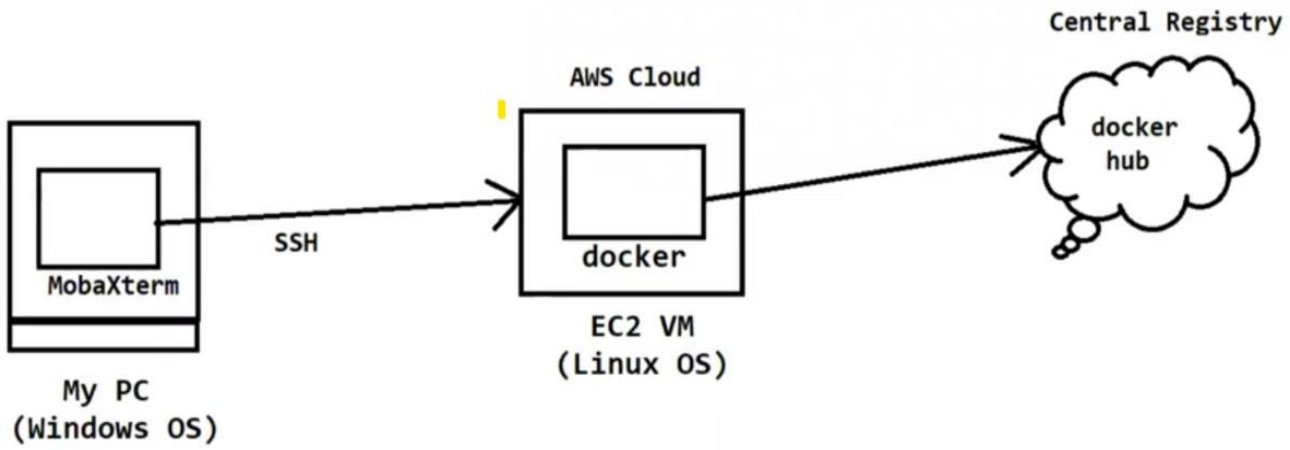
-> Execute below commands to install Docker s/w

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

 # add ec2-user to docker group by executing below command (to give docker permissions to ec2-user accnt)
$ sudo usermod -aG docker ec2-user

 # Close the terminal
$ exit
Then press 'R' to restart the session (This is in MobaXterm)

#execution below command to see docker status
$ docker info



==========
Basic Docker Commands
==========

 # display docker images available in our machine
$ docker images

 # download docker image
$ docker pull <image-name / image-id>

 # Run docker image
$ docker run <image-name / image-id>

 # Delete docker image
$ docker rmi <image-name / image-id>

 # Display all running docker containers
$ docker ps

 # display all running and stopped containers
$ docker ps -a

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

 # Delete docker image forcefully
$ docker rmi -f <image-id>

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

 # Delete all stopped containers and un-used images and un-used networks
$ docker system prune -a

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

-> Dockerfile contains instructions to build docker image
-> In Dockerfile we will use DSL (Domain Specific Language) keywords
-> Docker engine will process Dockerfile instructions from top to bottom
-> Below are the Dockerfile Keywords

FROM
MAINTAINER
COPY
ADD
RUN
CMD
ENTRYPOINT
ENV
LABEL
ARG
WORKDIR

EXPOSE
VOLUME

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

-> FROM keyword is used represent base image to create our own image
-> On Top of base image our image will be created

Syntax:

FROM java:jdk-1.8
FROM tomcat:9.5
FROM mysql:6.8
FROM python:3.3

==========
MAINTAINER
==========

-> MAINTAINER keyword is used to specify Dockerfile author information

Syntax:

MAINTAINER Ashok <ashok.b@oracle.com>

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

-> COPY command is used to copy the files from source to destination while creating docker image

Syntax:

COPY <source-location> <destination-location>

Ex:
COPY target/sbi-app.war /app/tomcat/webapps/sbi-app.war

=======
ADD
=======

-> ADD command is also used to copy files from source to destination while creating docker image

Syntax:

ADD <source-location> <destination-location>

ADD <url> <destination-location>

Ex:
ADD <URL> /app/tomcat/webapps/sbi-app.war

Q) What is the difference between COPY and ADD commands ?

-> Using COPY command we can just copy the files from one path to another path with in the machine

-> Using ADD command we can copy files from one path to another path and it supports source location as URL also.

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

-> RUN instructions will execute while creating the image
-> Using RUN we can give instructions to docker to execute commands
-> We can write multiple RUN instructions, docker will process all the RUN instructions from top to bottom

Example

RUN yum install maven
RUN yum install git
RUN git clone repo-url
RUN mvn clean package

=======
CMD
=======

-> CMD instructions will execute while creating the container
-> Using CMD command we can run our application package file jar / war file

Example

CMD sudo start tomcat

Note: If we write multiple CMD instructions also docker will process only last CMD instruction. There is no use of writing multiple CMD instructions in one Dockerfile.

Q) What is the difference between RUN and CMD in Dockerfile ?

-> RUN is used to execute instructions while creating image
-> CMD is used to execute instruction while creating Container

-> We can write multiple RUN instructions in Dockerfile, docker will process all those instructions one by one.
-> If we write multiple CMD instructions in Dockerfile, docker willl process only last CMD instruction.

==========

Sample Dockerfile

==========

```
FROM ubuntu
MAINTAINER Ashok<ashokit@gmail.com>
RUN echo "Hi, i am RUN-1"
RUN echo "Hi, i am RUN-2"
CMD echo "Hi, I am CMD-1"
RUN echo "Hi, i am RUN-3"
CMD echo "Hi, i am CMD-2"
```

-> Save the above content in docker file
filename : Dockerfile

 # Command to create docker image using dockerfile
Syntax : $ docker build -t <image-name> .

Ex : $ docker build -t myfirstimage .

 # Command to run docker image
$ docker run myfirstimage

 # Command to login with dockerhub account
$ docker login

Note: We need to enter our dockerhub account credentials correctly (it will ask only first time)

 # Command to tag our docker image
$ docker tag <image-name> <tag-name>

Ex: $ docker tag myfirstimage ashokit/myfirstimage

 # command to push docker image to docker hub account
$ docker push <tag-name>

Note: Delete all unused images and stopped containers
$ docker system prune -a

 # Pull the image from docker hub
$ docker pull ashokit/myfirstimage

 # Run the image
$ docker run ashokit/myfirstimage

Note: We can use customized name also for the dockerfile. When we change dockerfile name we need to pass filename as input for docker build command using -f option like below.

$ docker build -f <dockerfile-name> -t <image-name> .

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

-> ENTRYPOINT instructions will execute while creating container

Syntax

ENTRYPOINT ["echo" , "Welcome to Ashok IT"]

ENTRYPOINT ["java" , "-jar" , "target/boot-app.jar"]

Q) What is the difference between CMD and ENTRYPOINT ?

-> We can override CMD instructions in runtime while creating container
-> We can't override ENTRYPOINT instructions

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

-> It is used to set working directory for an image / container

Ex:
WORKDIR /app/

Note: The Dockerfile instructions which are available after WORKDIR those instructions will be processed from given working directory.

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

-> ENV is used to set Environment Variables

Ex:
ENV <key> <value>
ENV java /etc/softwares/java

====
ARG
====

-> It is used to remove hard coded values
-> Using ARG we can pass values in the runtime like below

Ex:
ARG branch
RUN git clone -b $branch <repo-url>
$ docker build -t imageone --build-arg branch=master

=====
USER
=====

-> We can set user for creating image / container

Note: After USER instruction all the remaining commands will execute with given user permissions

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

-> It is used to specify our container running PORT

Ex:
EXPOSE 8080

Note: It is just like a documentation command to provide container running port number.

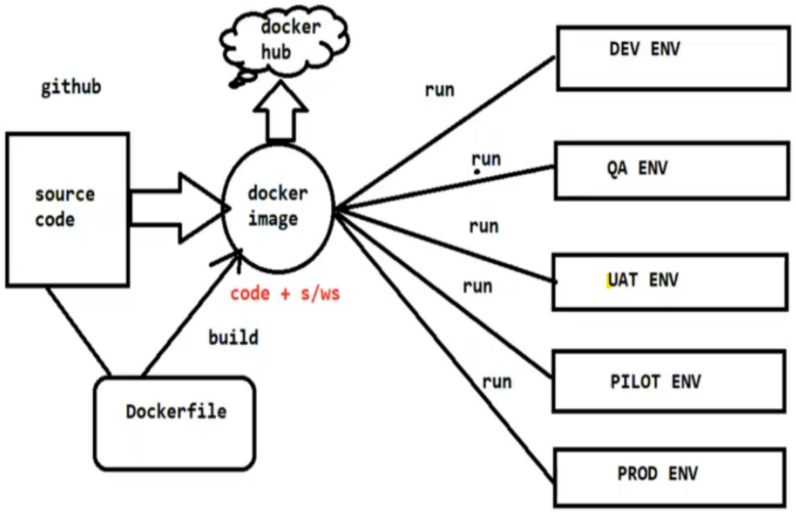
========
VOLUME
========

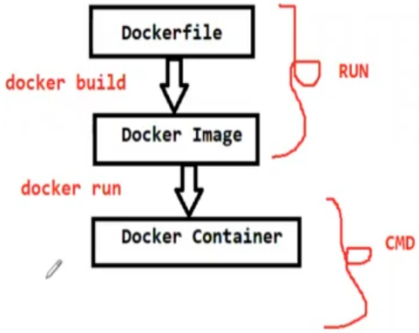
-> VOLUME is used to specify docker container data storage location.

Note: Volumes are used for storage.

FROM
MAINTAINER
COPY
ADD
RUN
CMD
ENTRYPOINT
WORKDIR
USER

ENV
ARG
EXPOSE
VOLUME





==========

Dockerize Spring Boot Application

==========

-> Spring Boot is one ready made java based framework available in the market to develop java based applications quickly

-> Spring Boot is providing emedded server (internal server will be available, we no need to configure server for execution)

-> Spring Boot application will be packaged as jar file (mvn clean package goal will do that package)

Note: When we do maven package, project jar will be created in project target folder

-> To run spring boot applications we just need to run jar file like below

$ java -jar <boot-app-jar-file>

Dockerizing Spring Boot Application

-> In springboot app, embedded server will be available. We no need to configure server seperatley

-> boot apps will be packaged as jar file

-> We will just run jar file of project (server will start automatically)

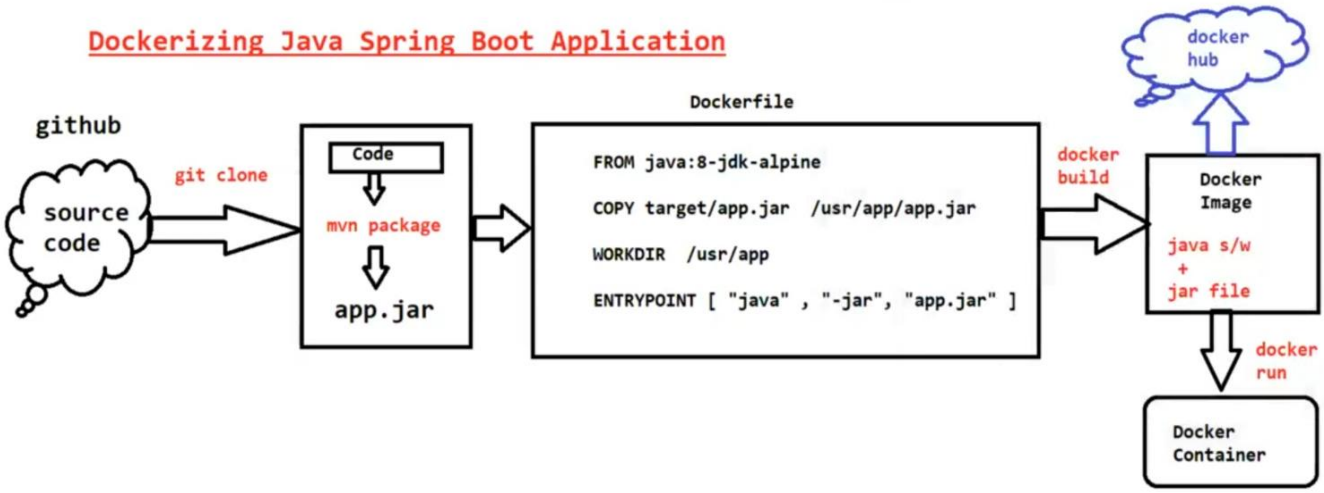
Dockerizing Java Web Application

-> To run normal java web app we need tomcat server

-> Normal java web app will be packaged as war file

-> we will deploy war file for this kind of projects in the servers

Dockerizing Java Spring Boot Application



----------Dockerfile----------

```
FROM openjdk:11
COPY target/spring-boot-docker-app.jar  /usr/app/
WORKDIR /usr/app/
ENTRYPOINT ["java", "-jar", "spring-boot-docker-app.jar"]
```

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

 # install git client s/w
$ sudo yum install git

 # Clone Git Repo
$ git clone https://github.com/ashokitschool/spring-boot-docker-app.git

 # Navigating to project folder
$ cd spring-boot-docker-app

 # install maven s/w
$ sudo yum install maven

 # execute maven goals
$ mvn clean package

Note: After package got success, we can see project jar file in target folder.

 # create docker image
$ docker build -t sb-app .

 # run docker image with port mapping
$ docker run -p 8080:8080 sb-app

Note: Enable 8080 port number in EC2 VM security group

URL To Access Application : http://ec2-vm-public-ip:8080/welcome/Ashok

==========

How to run Docker container in Detached Mode

==========

=> With below command our terminal will be blocked, we can't execute any other command. To execute other command we need to type CTRL+C then terminal will open for commands execution but our container gets stopped.

$ docker run -p 8080:8080 ashokit/sb-app

Note: To overcome above problem we can pass '-d' to run container in detached mode. When we execute below command it will run the container in detached mode and it will open terminal for commands execution.

$ docker run -d -p 8080:8080 ashokit/sb-app

=> Once above command is executed, we can see running containers using below command

$ docker ps

=> We can check logs of the container using below command

$ docker logs <container-id>

==========

Dockerizing Java Web Application (Without SpringBoot)

==========

-> Java web applictaions will be packaged as war file
-> WAR (Web Archive) contains application code
-> To run the war file we need webserver (Ex: Apache Tomcat)
-> We need to deploy war file in Tomcat Server for Execution
-> In Tomcat server we will have "webapps" folder for deployment

Note: To run normal java web applications we need "java & tomcat" as dependencies

----------Dockerfile----------

```
FROM tomcat:8.0.20-jre8

COPY target/01-maven-web-app.war  /usr/local/tomcat/webapps/maven-web-app.war
```

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

$ git clone https://github.com/ashokitschool/maven-web-app.git
$ cd maven-web-app
$ mvn clean package
$ docker build -t maven-web-app .
$ docker images
$ docker run -d -p 8080:8080 maven-web-app
$ docker ps
$ docker logs <container-id>

URL To access The Application : http://ec2-vm-public-ip:host-port/maven-web-app/

Note: In the above url "maven-web-app" is called as context path (name of the war file will become context path)

==========
Dockerize Python Application
==========

-> Python is a general purpose scripting language
-> Python programs will have .py as extension
-> Compilation is not required for python programs

----------Dockerfile----------

```
FROM python:3.6
MAINTAINER Ashok Bollepalli "ashokitschool@gmail.com"
COPY . /app
WORKDIR /app
EXPOSE 5000
RUN pip install -r requirements.txt
ENTRYPOINT ["python", "app.py"]
```

Python Flask app Git hub Repo: https://github.com/ashokitschool/python-flask-docker-app.git

$ git clone https://github.com/ashokitschool/python-flask-docker-app.git
$ cd python-flask-docker-app
$ docker build -t python-flask-app .
$ docker images
$ docker run -d -p 5000:5000 python-flask-app
$ docker ps
$ docker logs <container-id>

==========

 # Command to enter into docker container
$ docker exec -it <container-id> /bin/bash

Note: Execute 'exit' command to come out from docker container vm.

==========

Assignment -1 : Dockerize React JS Application

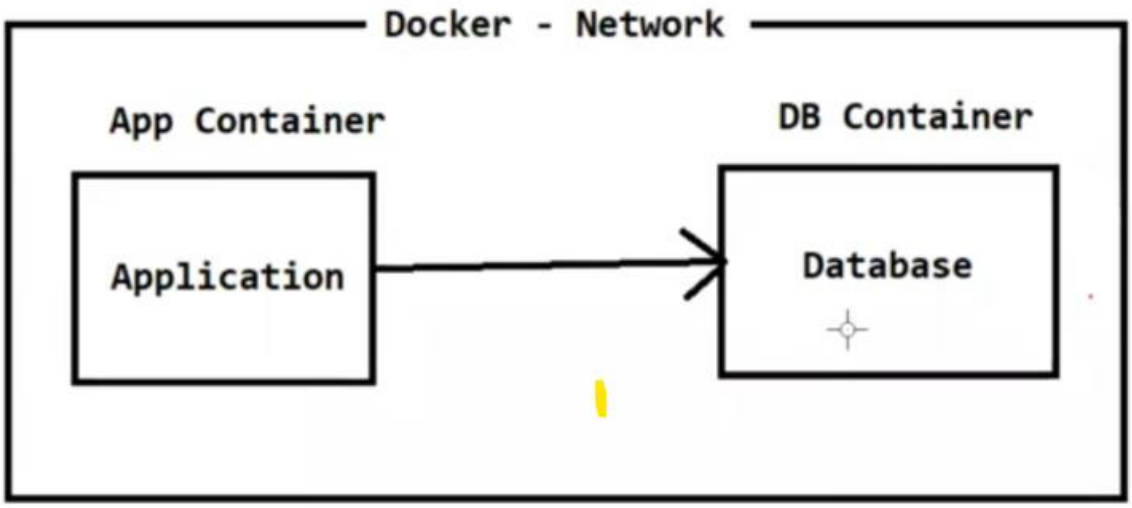
Assignment-2 : Dockerize Node JS Application

==========

==========

Docker Network

==========



-> Network is all about communication

-> Docker network is used to provide isolated network for Docker Containers

-> In Docker we will have below 3 default networks

1) none
2) host
3) bridge

-> In Docker we have below 5 network drivers

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

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

-> Host Driver is also used for standalone container. IP will not be assigned for container

-> None means no network will be provided by our Docker containers

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

-> Macvlan driver will assign MAC address for a container. It makes our container as Physical.

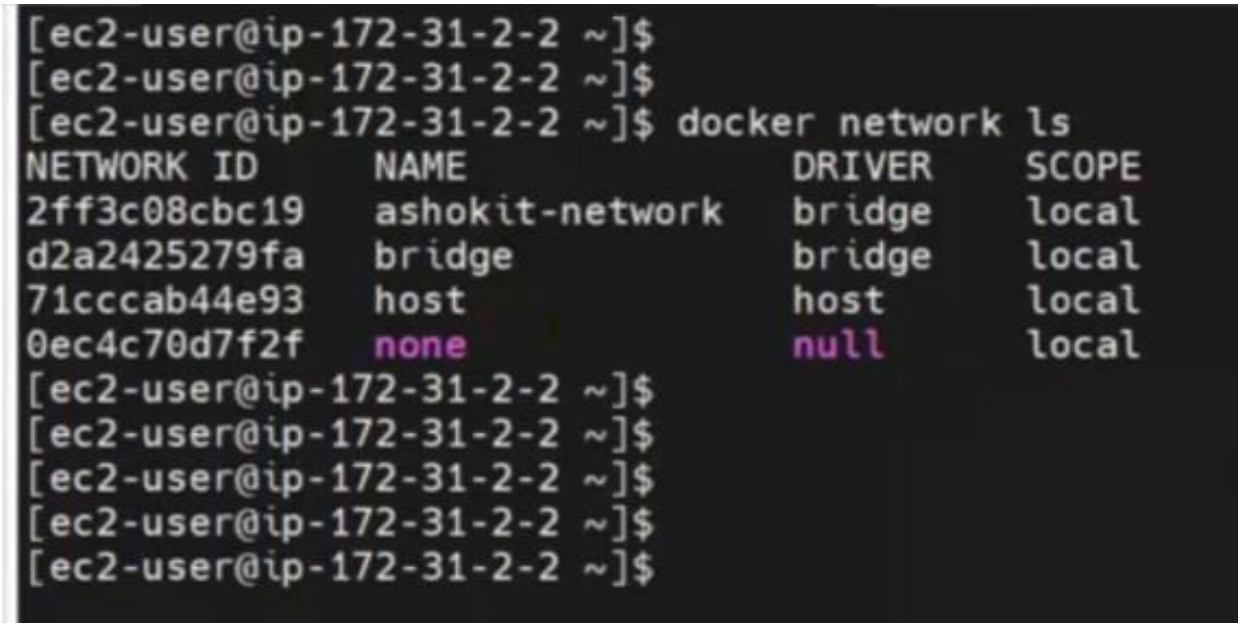
 # display docker networks available
$ docker network ls

 # Create docker network
$ docker network create ashokit-network

 # Inspect network
$ docker network inspect <network-name>

 # delete docker network
$ docker network rm <network-id>

 # Run a container with given network
$ docker run -d -p hport:cport --network ashokit-network <imagename>



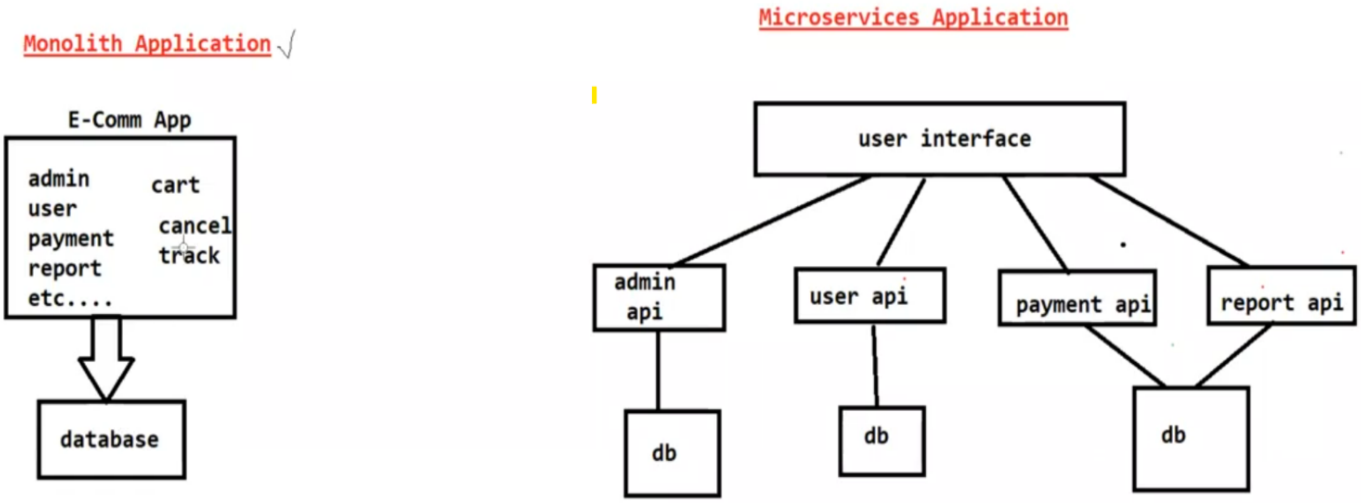
```
[ec2-user@ip-172-31-2-2 ~]$
[ec2-user@ip-172-31-2-2 ~]$ docker network inspect ashokit-network
[
    {
        "Name": "ashokit-network",
        "Id": "2ff3c08cbc1925b769859b231577df44ad475647a6b557d7f94332bfba6df148",
        "Created": "2022-12-01T02:03:57.693633094Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
[ec2-user@ip-172-31-2-2 ~]$
```

```
[ec2-user@ip-172-31-2-2 ~]$
[ec2-user@ip-172-31-2-2 ~]$
[ec2-user@ip-172-31-2-2 ~]$
[ec2-user@ip-172-31-2-2 ~]$ docker images
REPOSITORY                TAG       IMAGE ID       CREATED         SIZE
react-app                 latest    15cef1468ca8   21 hours ago    1.39GB
ashokit/python-flask-app  latest    82781d5652bc   24 hours ago    913MB
node                      latest    5f352850ed59   25 hours ago    995MB
[ec2-user@ip-172-31-2-2 ~]$
[ec2-user@ip-172-31-2-2 ~]$
[ec2-user@ip-172-31-2-2 ~]$
[ec2-user@ip-172-31-2-2 ~]$ docker run -d -p 3000:3000 --network ashokit-network react-app
8367192785f30df5f816629a177421223192ec28467aaecdd3a4efb516e59b22
[ec2-user@ip-172-31-2-2 ~]$
```

==========

Docker Compose

==========



Monolith Application : One application which contains all the functionalities is called as Monlith App.

-> If we make any small code change then we need to re-deploy entire application
-> If we make any code change in one functionality there may be a impact on another functionality
-> Maintenence will become very difficult when we go for Monolith Based Application

Note: To overcome the problems of Monlith we are using Microservices Architecture.

Microservices Application : Application functionality will be developed as micro services (rest apis)

-> Every functionality will be developed as individual project (individual API)

ADMIN_API
REPORT_API
PAYMENT_API
CART_API

TRACKING_API
PRODUCT_API
CANCEL_API

=> Every API should run in a seperate container

=> Running Multiple containers manully for all the apis is difficult job

******* To solve this problem Docker-Compose came into picture *********

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

=> Using Docker Compose we can easily setup & deploy multi container based applications

=> We will give containers information to Docker Compose using YML file (docker-compose.yml)

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

==========

Docker Compose YML File

==========

version:

services:

network:

volumes:

==========

=> Docker Compose default file name is "docker-compose.yml"

 # Create Containers using Docker Compose
$ docker-compose up

 # Create Containers using Docker Compose with custom file name
$ docker-compose -f <filename> up

 # Display Containers created by Docker Compose
$ docker-compose ps

 # Display docker compose images
$ docker-compose images

 # Stop & remove the containers created by docker compose
$ docker-compose down

==========

Docker Compose Setup

==========

 # 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

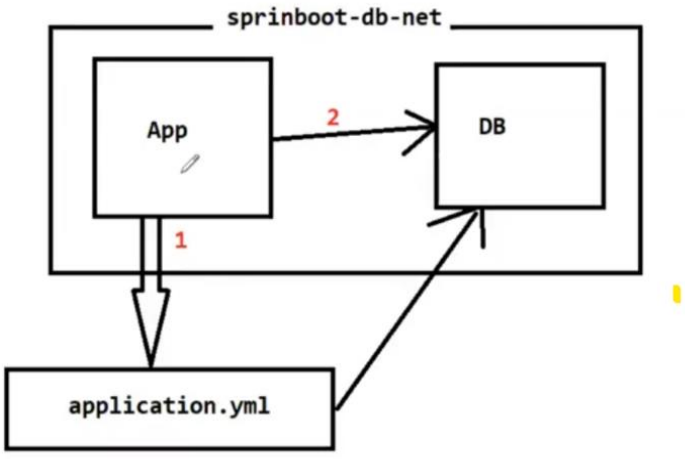
==========

Spring Boot with MySQL using Docker Compose

==========

```
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
```

Spring Boot with MySQL Using Docker Compose




```
version: "3"
services:
  application:
    image: spring-boot-mysql-app
    networks:
      - springboot-db-net
    ports:
      - "8080:8080"
    depends_on: mysqldb
  mysqldb:
    image: mysql
    networks:
      - springboot-db-net
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=sbms
networks:
    springboot-db-net:
```

=> Spring Boot App with MySQL DB Git repo URL

URL : https://github.com/ashokitschool/spring-boot-mysql-docker-compose.git

=> Below is the docker-compose file

```
---
version: "3"
services:
  application:
    image: spring-boot-mysql-app
    networks:
      - springboot-db-net
    ports:
      - "8080:8080"
    depends_on:
      - mysqldb
  mysqldb:
    image: mysql:5.7
    networks:
      - springboot-db-net
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=sbms
networks:
  - springboot-db-net:
...
```

==========

Steps to Run Spring Boot application with MySQL DB using Docker Compose

==========

 # Clone git repo url

$ git clone https://github.com/ashokitschool/spring-boot-mysql-docker-compose.git

 # Get into project directory

$ cd spring-boot-mysql-docker-compose

 # Build Maven Project

$ mvn clean package

 # Create Docker Image (Image Name : spring-boot-mysql-app)
$ docker build -t spring-boot-mysql-app .

 # Check docker image created or not
$ docker images

 # Run containers using docker compose (docker-compose.yml available)
$ docker-compose up -d

 # Check containers which are created
$ docker-compose ps

 # Check logs of application contianer
$ docker logs -f <container-name>

Note: Access the application in browser

URL : http://ec2-vm-public-ip:host-port/

 # Get into App container to check jar file
$ docker exec -it <container-name> /bin/bash/

 # Check DB tables by entering into container
$ docker exec -it <db-container-name> /bin/bash

$ mysql -u root -p
$ show databases;
$ use <db-name> (our db name is sbms)
$ show tables;
$ select * from table-name (our table name is book)

==========

Stateful Conatiners Vs Stateless Containers

==========

-> Stateless Container means container will not remember the data which got generated by that container. When we re-create the new container we will loose old data.

Note: By default docker containers are stateless containers.

-> In above springboot application when we recreate the containers we lost our old data database which we inserted through application. (This is not accepted in the realtime).

Note: Even if we deploy latest code or if we re-create the containers we should not loose our old data. Our data should remain in the database.

-> If we don't want to loose the data even if we re-create the container then we need to make our Docker Container as Statefull Container.

-> To make Docker Containers as statefull, we need to use Docker Volumes.

==========

Docker Volumes

==========

-> Applications we are executing as Docker Containers

-> Docker containers are by default stateless

-> Once container is removed then we will loose the data that got stored in the container

-> In realtime we shouldn't loose the data even if container got removed
For Example : Database container

-> Application will store data in database, even if we delete application container or db container data should be available.

-> To make sure data is available even after the container is deleted then we need to use Docker Volumes concept

****** Docker Volumes are used to store container data permanently *******

=> Volumes are the preferred mechanism for persisting data generated by and used by Docker containers.

=> We have 3 types of volumes in Docker

1) Anonymous Volumes (without name)
2) Named Volumes (Will have a name) ----> Recommended
3) Bind Mounts (Storing on Host Machine)

Q) What is Dangling volume in Docker ?

-> The volumes which are created but not associated to any container are called as Dangling Volumes

 # Delete all dangling volumes
$ docker volume rm $(docker volume ls -q -f dangling=true);

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

 # Display all docker volumes
$ docker volume ls

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

 # Delete docker volume
$ docker volume rm <vol-name>

 # Delete all docker volumes
$ docker system prune --volumes

---------- **Docker Compose with Docker Named Volumne** ----------

```
version: "3"
services:
  application:
    image: spring-boot-mysql-app
    ports:
      - "8080:8080"
    networks:
      - springboot-db-net
    depends_on:
      - mysqldb
  mysqldb:
    image: mysql:5.7
    networks:
      - springboot-db-net
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=sbms
    volumes:
      - /var/lib/mysql
networks:
  springboot-db-net:
```

==========

Docker Swarm

==========

Docker : It is a containerization platform. It is used to deploy the applications as containers.

Docker Swarm : It is an Orchestration Platform. It is used to manage docker containers.

-> Managing docker containers nothing but creating / updating / scale up / scale down / remove containers

Note: In market we have docker swarm, kubernetes, open shfit as Orachestration platforms.

-> 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>

=========
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>


# To check os version
$ cat /etc/os-release

# To check kernal version
$ uname -r
```