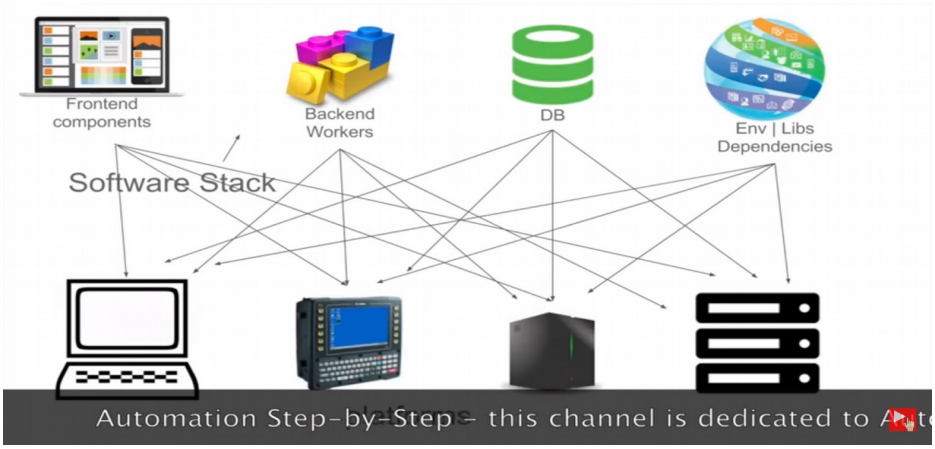
Note:- This reference guide I have made from youtube channel named Automation Step by Step by Raghav Pal.
Practical project portion decribed in page 6-8 is from youtube channel named TechWorldwithNana.

What is docker & why we use docker?

Docker is the worlds leading container platform. It is a tool to deploy and run application using containers.

Containers help developers to package an application with all of its dependency and libraries and ship it out as one package.

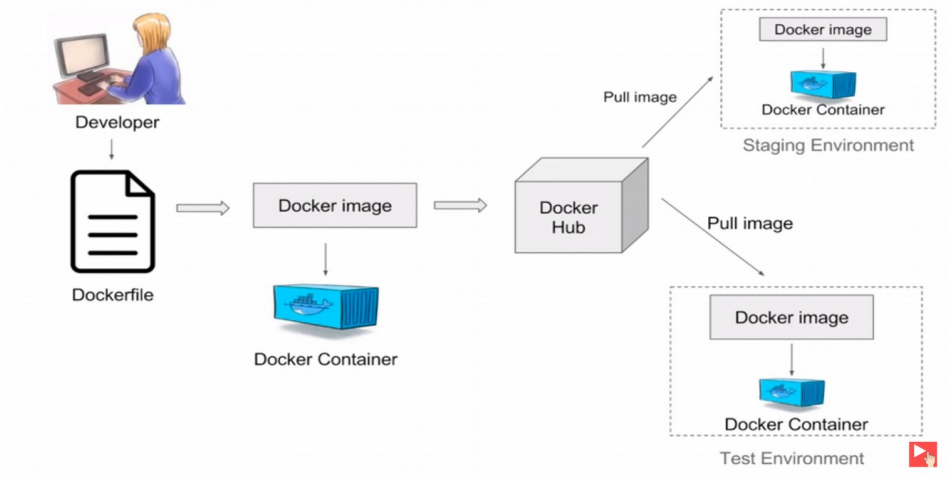
Currently a typical application contain several layer in it stack like frontend, backend , databases and we also like to deploy it in various different kind of system and environment, we expect all of the stacklayer work seamlessly on deployment, but in reality some software work in one environment and some not. To fix this we use docker to ship all our application stack and its dependencies in a conatiner & we doply these containers in those system or environment.



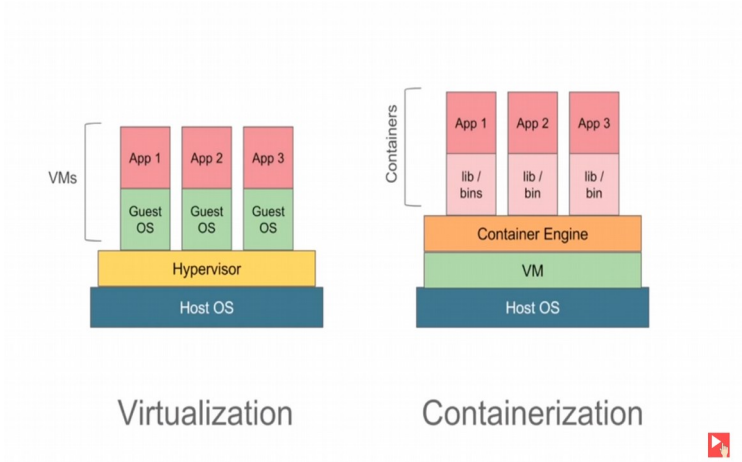
How docker works?

Docker workflow:-

1. A developer will document all of its applicaton depenedencies in a single file called Dockerfile.
2. Dockerfile is then used to create docker images. This docker image will have all the application requirements and its depnedencies.
3. When this image is run we get a docker conatiner. So docker container is nothing but the run time instances of docker image.
4. These images can also be stored in Docker hub.(It is a online cloud repositories to store images)



Virtualization vs Containerization:-

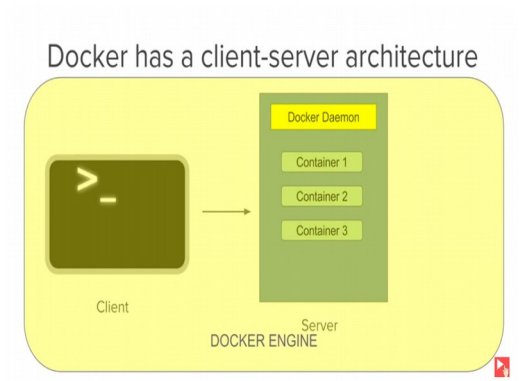


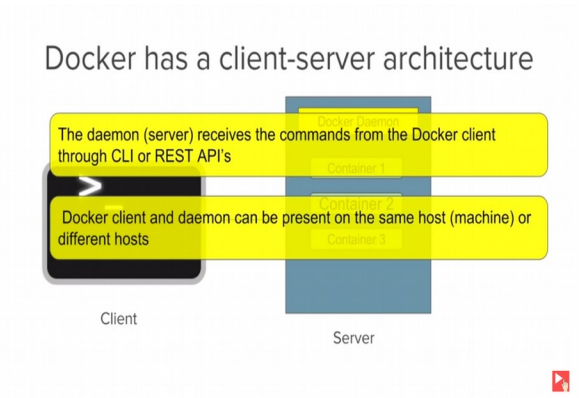
Virtualization:-
In virtualization we have a software called hypervisor which is used to create and run virtual machine. These virtual machine have there own operating system, it do not use host operation system on which hypervisor is running.

These VM create an overhead on host as they have their own OS, also we have to provide fix resources to these VM which doesn't change on runtime as per the need of application, thus there is alot of wastage of resources.

Containerization:-
In containerization we have container engine in place of hypervisor, also it doesnt have separate OS it is using the host OS only, but we have conatiners which contains application and all of it dependencies. Here resources are not fixed it will used as per the need of application. It is very light weight and fast.

Docker architecture:-





CLI or Rest api is the client and Docker server or Docker daemon is the server which containes all the container. Docker client and Docker daemon can be present in same machine or different machine.

Practice & Learn more advanced feature about docker:-

https://labs.play-with-docker.com/
https://training.play-with-docker.com/

One of the best cheatsheet for docker:-
https://github.com/wsargent/docker-cheat-sheet

export PS1="\u$ " -> this will shorten the string it displays in shell every line. Here "\u" means username.

Docker commands:-

Basic commands:-

Basic

: **docker version** -> This commads gives basic information about the docker client and docker server.
: **docker -v** -> This command gives the docker version installed in system.
: **docker info** -> This will give the detailed info and stats about the docker running in your system.
: **docker --help**
: **docker login** -> Login to dockerhub, need to create account first at docker hub. [https://hub.docker.com].

————————————

Images

: **docker images** -> Gives the list of all the images we have in system.
: **docker images -f "dangling=false"** -> This will fetch all those images which are not associated with a container.
Dangling images are those image which are associated with a container.

: **docker pull <imagename>** -> Pull the docker image from docker hub.
: **docker pull <imagename:tag> ->** Pull the particular tag of image, like **docker pull ubuntu:18.04.**

: **docker inspect <imagename>** -> This will provide the different details about the image.
: **docker history <imagename/id>** -> This will give some history of the image.

: **docker rmi <imageid>** -> This will remove the docker image. But it wont remove a image which is associated with any container. To forcefully remove use -f option **docker rmi -f <imageid>**

————————————

Containers

: **docker ps** -> List all the running containers (Remember container is the running instance of the image)
Helpful command:- (**docker ps -a**) ->List all the containers either in running or non running state

: **docker container ls** -> list all the running containers
: **docker container ls -a** -> list all the running and stopped same like (docker ps -a) containers

: **docker run <imgname/imgid>** -> This command will create a container out of the image and run the conatiner. If not present locally it will try to fetch image from dockerhub.
Helpful command:- docker run -it <imgname/imgid>

: **docker start <container-id>** -> Start the container.
: **docker pause <container-id>** -> Pause the conatiner.
: **docker unpause <container-id>** -> Unpause the conatiner.
: **docker stop <container-id>** -> Stop the conatiner.

: **docker rm <container-id>** -> Remove the container (It will remove only the stopped container).
: **docker container rm <container-id>** -> Remove the container (It will remove only the stopped container)

: **docker attach <container-id/container-name>** -> Attach the current shell to the running terminal. For this command to work container must be running.
: **docker kill <container-id/container-name>** -> Kill a running container.For this command to work container must be running.

: **docker top <container-id/container-name>** -> Works like the top command in linux.
: **docker stats <container-id/container-name>** -> Kill Give the stat of the given conatiner like its memory/cpu uses etc.

————————————

System

: **docker stats** -> Give the different stats of running container, like its memory/cpu uses etc.
: **docker system df** -> Give all the information about docker disk usage.
: **docker system prune** -> It will delete all the stopped conatiner and all the dangling images.
If you want to remove all the images also, then give command
: **docker system prune -a**

Docker volumes:-

Before going in details with volumes we need to understand that docker container run in an isolated environment meaning whatever is happening inside container it is not going to affect the host system or any other container runing on the host. Therefore any data that is generated by container only exist inside container and as soon as container is stopped it will be removed from the system.

But this will create problem for Jenkins or Database container where user create Jenkins job or Database create records which needed to be restored for later use. To handle this problem we use docker volumes which is independent of container, and it will persist even after the container is stopped or deleted.

Run command to persist data generated by jenkins container on host system path we can use the below command:-

```
: docker run --name <container-name> \
-p <hostport>:<containerport> \
-p <hostport>:<containerport> \
-v <host path>:<jenkins-home path on conatiner> \
<jenkins-imagename/imageid>
```

Example:-

```
: docker run --name MyJenkins \
-p 8080:8080 \
-p 50000:50000 \
-v /Users/ritesh/Desktop/Jenkins_Home:/var/jenkins_home \
jenkins
```

Volume Commands:-

: docker volume ls -> List the volumes present in the system.
: docker volume create <volume-name> -> Create volume.
: docker volume inspect <volume-name> -> Inspect volume.
: docker volume rm <volume-name> -> Remove volume.

Run command to persist data generated by jenkins container on docker volume we can use the below command:-

```
: docker run --name <container-name> \
-p <hostport>:<containerport> \
-p <hostport>:<containerport> \
-v <volume name/volumeid>:<jenkins-home path on conatiner> \
<jenkins-imagename/imageid>
```

Example:-

Step1 :- Create a docker volume

```
: docker volume create  myjenkinsvolume
```

Step2 :- Run docker container using the volume created in step 1.

```
: docker run --name MyJenkins \
-p 9090:8080 \
-p 50000:50000 \
-v myjenkinsvolume:/var/jenkins_home \
jenkins
```

Dockerfile:-

It is a simple file with instructions to build a image. We can say that dockerfile is automation of dockerimage creation.

Most usable command or instruction in dockerfile :-

1. **FROM:-** Instruction to pick base image from which our image will be created.
   Ex:- FROM ubuntu

2. **MAINTAINER:-** Optional step to provide the maintainer details.
   Ex:- MAINTAINER fname lastname <abcd@gmail.com>

3. **RUN:-** This get excuted during the building of the image.
   Ex:- RUN apt-get update

4. **CMD:-** Command to run after container creation.
   Ex:- CMD [“echo”, “Hello World...”]

Example contents of Dockerfile

```
# getting base image . This is how we write comments with a hash
FROM ubuntu

MAINTAINER fname lastname <abcd@gmail.com>

RUN apt-get update
# Image build out of this will have python 3.6 installed
RUN apt-get install python3.6

CMD [“echo”, “Hello World...”]
```

Build this image with command:-

: docker build -t <imagename>:<tag> <directory of Dockerfile>

Ex:- **docker build -t ubuntu-python:1.0 /*home*/ritesh/dockerdir**

Dockercompose:-

It is a tool for defining and running multicontainer docker application.

Commands:-

: docker-compose up -> Can start all services with a single command.
: docker-compose down -> Can stop all services with a single command.
: docker-compose up -d --scale <servicename>=<number of intances>
Note:- **-d** is for detached mode
: docker-compose config -> Check the validity of the docker compose file.

Example contents of Docker compose yaml file

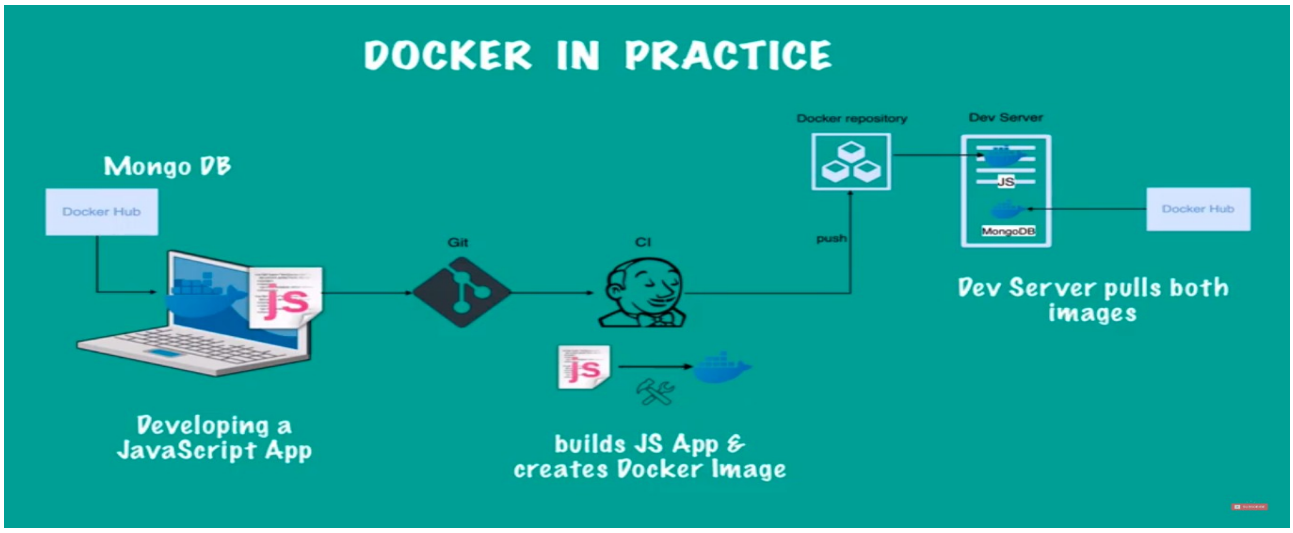
```
version: ‘3’

services:
  web:
    image: nginx
    ports:
    - 9090:80

  database:
    image: redis
```

Docker in practice:-

Project descrption:-
We are developing a node application which will connect to mongodb and update database records.



A typical docker development flow in practice:-

1. In local system we will be developing a node app.
2. We will pull mongodb image.
3. After node development is done we will create Dockerfile to create a node app image.
4. After this we will create a Dockercompose file create and run it. Alternately we can run the node app image and mongo image independently and connect them within a docker network , after creating a docker network.
5. Then we will test our code. If test is successfull we will push the code in github.
6. Jenkins will then pull the code & create a image out of it using Dockerfile.Might also run different step of continuous integration on it. If everything looks good then it will push the image to registry.
7. In this step we will deploy our application. Jenkins will use the Dockercompose file to pull all the image(Node app image and mongo image) and run it.
8.Thus deployment is complete!!!

Gitlab link of Node app:- https://gitlab.com/nanuchi/techworld-js-docker-demo-app

Create a docker network and connect mongodb and mongo-express container with that network.

: docker network create <network-name>
docker network create mongo-network

Docker run command to connect with a network:-

```
docker run -d \
-p 27017:27017 \
-e MONGO_INITDB_ROOT_USERNAME=admin \
-e MONGO_INITDB_ROOT_PASSWORD=password \
--name mongodb \
--net mongo-network \
mongo # image name
```

```
docker run -d \
-p 8081:8081 \
-e ME_CONFIG_MONGODB_ADMINUSERNAME=admin \
-e ME_CONFIG_MONGODB_ADMINPASSWORD=password \
-e ME_CONFIG_MONGODB_SERVER= mongodb \    # mongo container name defined above
--name mongo-express \
--net mongo-network \
mongo-express # image name
```

Docker compose for development:-

```
version: '3'

services:
  mongodb:  # container name
    image: mongo
    ports:
      - 27017:27017
    environments:
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=password

  mongo-express:
    image:  mongo-express
    ports:
      -  8081:8081
    environments:
      - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
      - ME_CONFIG_MONGODB_ADMINPASSWORD=password
      - ME_CONFIG_MONGODB_SERVER= mongodb
```

Dockerfile for Node app:-

```
FROM node:13-alpine

ENV MONGO_DB_USERNAME=admin \
        MONGO_DB_PWD=password

RUN mkdir -p /home/app

COPY . /home/app  # copy code from local to container

CMD ["node", "/home/app/server.js"]
```

Steps to push your image into ECR:-

1. Get the ECR login command from AWS ECR service and run from commandline. (This is docker login).
2. Build your image.
3. Get the image tagging command from AWS ECR service and tag your image.
4. Push the image to ECR.

Deploy the application:-

Update the docker-compose for deployment. Now pull the node app image from ECR to run the application.

Updated docker compose file:-

```
version: '3'

services:
  my-node-app:
    image: <ecr registry uri>/my-app:1.0
    ports:
      - 3000:3000

  mongodb:  # container name
    image: mongo
    ports:
      - 27017:27017
    environments:
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=password

  mongo-express:
    image:  mongo-express
    ports:
      -  8081:8081
    environments:
      - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
      - ME_CONFIG_MONGODB_ADMINPASSWORD=password
      - ME_CONFIG_MONGODB_SERVER= mongodb
```

Updated docker compose file with volume to use:-

```
version: '3'

services:
  my-node-app:
    image: <ecr registry uri>/my-app:1.0
    ports:
      - 3000:3000

  mongodb:
    image: mongo
    ports:
      - 27017:27017
    environments:
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=password
    volumes:
      -  mongo-volume:/data/db

  mongo-express:
    image:  mongo-express
    ports:
      -  8081:8081
    environments:
      - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
      - ME_CONFIG_MONGODB_ADMINPASSWORD=password
      - ME_CONFIG_MONGODB_SERVER= mongodb

volumes:
  mongo-volume:
    driver: local    # Create or search volume in local filesystem
```