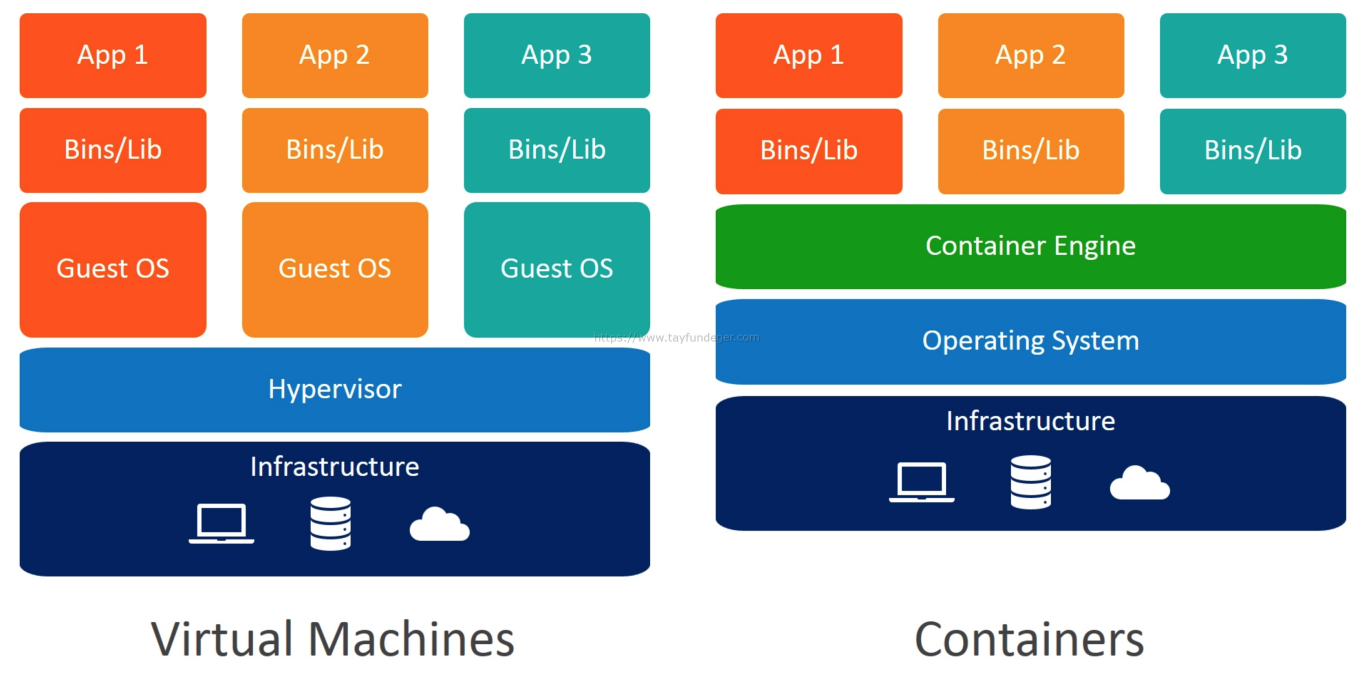
**DOCKER**

Nitro hypervisor – the hypervisor we are using in aws



**Docker Image/image:** Application, code configurations, and dependencies are packaged into a single file is called docker image.

Docker images are created by Dockerfile

**Dockerfile:** the set of instructions used to create an image.

**Docker:** Docker is a PaaS that allows you to build, test, and deploy applications quickly.

**Can we install docker on Windows?**

No, we cannot install docker as a core software or a program, but docker has provided a way to install it on Windows called docker desktop.

When we installed docker desktop, it will automatically spin a Linux virtual machine on Windows (oracle virtual machines)

**Container:** It is a group of isolated processes. It is an instance or server for that application.

**How does docker or container work internally?**

**What are cgroups and namespaces and how it is related to docker?**

**What is a Linux container?**

In a normal virtualized environment, one or more VM run on top of a physical machine using a hypervisor like xen hypervisor

Container or docker used Linux kernel features like namespace control group (Cgroup), and chroot to create a container on top of an OS and automate application deployment on the container.

Containers are isolated in a host using three Linux kernel features.

Namespace, cgroup ad chroot.

1. namespace: process isolation requires separation from other processes which is achieved from the default Linux feature called as a namespace.

namespace helps us to separate and group some processes from other processes.

Some namespaces are PID, UFS (isolated kernel & version identifier), GID, cgroup , net(network interfaces), IPC, Mount, User

1. cgroup (resource limiter): Cgroup will help us to control the access, creation, and deletion of files.

The resources used by the container are managed by Linux cgroup

We can decide on how much CPU and memory resources a container should use using cgroup ex: CPU, Memory (RAM), sockets (Port), and devices (USB).

1. chroot: it will create the root directory.

**Have you ever created or changed cgroup in a container?**

In our company, we have not changed any default characteristic, but I have worked with cgroup which are created default by docker.

**How docker will limit resources to the container using cgroup?**

CPU, Memory (RAM), sockets (Port), and devices (USB).

**Docker file instructions:**

1. FROM: to pull the image
2. WORKDIR: it is similar to CD, to define a working directory we use WORKDIR ex: WORKDIR /home, WORKDIR /lib
3. RUN: it is used to update the image and commit and also used to run commands and commit it.
4. CMD: it is the default command that runs when a new container starts.
5. ENTRY POINT: it is a default command that runs when the new container starts.
6. EXPOSE: it is used to make a port available for all containers in the same cluster.
7. PUBLISH: Publish is used to map the port inside the container to the port of the host machine, where the same container is in the run.
8. MAINTAINER: use the name of the user who maintains the image. Usually given after FROM
9. COPY: Instructions will copy the new file from the source path to the container file system path.
10. ADD: ADD will do the same thing as copy, but add allows us to do URLs, tar, or zip file as the source file while executing it extracts the data and

Arg: docker build -t imagename --build -arg TEST=/test

Env: docker run -d -e TEST=”5”

Difference between ENV & ARG?

ARG- Values are available in the docker file during building the docker image.

values are not available after the image is built or the running container won’t have access.

ENV- It is mainly meant to provide default values for your future environment variables.

Running docker applications can access environment variables.

It is a great way to pass configuration values to your project.

-e – environmental variable for a container which can be accessed anywhere inside the container.

**Difference between CMD and ENTRY POINT**

When both commands are present CMD becomes an argument to ENTRY POINT. We can override CMD at run time, but we cannot override ENTRY POINT.

**Can we have two or more CMD or ENTRY POINT commands?**

Yes, but it will consider the last CMD or ENTRY POINT only, the remaining will be ignored.

The Docker engine is called a Docker daemon.

**Docker images:** it is a command used to list all the images that exist in the local machine.

**Docker ps**: it is used to list running containers in the local machine.

**docker ps -a**: it is used to list all the containers in the local machine.

**Docker build:** it is a command used to build the images

Syntax: docker build -t <image name>. or docker build -t <image name>:<tag> .

Or docker build -t <image name> -f Dockerfile

**Docker run:** it is a command used to create the container from the image

Syntax: docker run -it <image name>:<tag>

-it: interactive mode

-d: detached mode

**Docker rmi -f imageid:** to delete the docker images

**Docker rm -f containerid:** to delete the containers

**How do I know which image uses which port?**

By using docker hub documentation I will come to know which image is using which port.

**Port mapping:**

It will inform the docker daemon to the associated port of the container to the host machine.

**There are 2 ways:**

1. -p: It maps the individual port of the container to the port of the host machine.

Example: -p <hostport>:<containerport>

-p 8888:8080

1. -P: All ports of the container are used to the same port of the host machine.

Example: docker run -it <imageid> -P

**Docker exec:** it is used to get inside the running container. Syntax: docker. exec -it <container id> /bin/bash (without /bin/bash I can give just “bash” as well)

Docker exec -it containerid bash

**Without a base image how to create an image? Or how to create your own image or base image?**

FROM scratch, it doesn't pull any images the images which are empty.

**Volumes and binds/ type of layers/Types of volumes:**

Docker containers doesn't store persistent data, any data returned to the persistent layer will no longer be available once the container stops running.

To solve the problem of persistent data from a container docker has two options.

1. Volumes
2. bind mount

**Bind mount:** It is a file, or a folder stored anywhere on the container host file system, mounted into a running file system.

It exists in the host machine so the process outsider of docker can also modify it.

Note: (Disadvantage)

Non-docker process on the docker host or docker container can modify data at anytime

Syn: -v hostmachinefile/folderpath:containerfile/folderpath

**Volumes:** single volume can be attached to multiple containers. Volumes are maintained by a docker hub no one can access the volume outside the container, this is a mountable entity that can be used to store persistent data.

Volumes are stored in the host file system, but which is managed by docker, non-docker processes should not modify this part of the file system.

Create volume syntax: docker volume create volumename

**Docker inspects:** if you want to get more information or details of a container or volume, we use docker inspect.

Syn: docker inspects container id or volume id.

**List the volumes:** docker volume ls

Inspect the volume: docker inspect volume ls

**Mount the volume:** -v volume:containerpath (container path : /usr/local/tomact/webapps)

**Docker tag**: it is used to tag the image with a different name, or it is used to rename the image.

Syntax: docker tag <name of the existing image>:<tag> <new or existing name of the image>:new tag

**Docker registry:** the place where docker images are saved.

The default docker registry is docker hub other registries are Amazon ECR, nexus registry etc…

**Docker login:** command to communicate with docker hub/docker registry.

We need to login to the docker registry before pushing or pulling the images.

Syntax: docker login -u <username> -p <password> -h <docker registry name>

**How do you list all containers for a particular image?**

Docker ps -a --filter ancestor=<image name>

**How to check whatever execution is going on in a container or how to check the logs of the container?**

Sudo docker logs <container id>

We can check only the running containers.

**How to list the id of the running container?**

Docker ps -q

Docker ps -a -q: to list all the stopped containers.

**How do you delete all running container**

Docker stop $( sudo docker ps -q )

Docker rm $( sudo docker ps -a -q )

**How to check whether command execution is successful or not**

After execution, if it is successful, it will display the container id otherwise it will display errors

**How does docker pull images internally or stores image internally?**

It just stores the difference between the images.

**Dangling images:** images without name and tag and which are not used by any container are called dangling images.

**How to list dangling images only?**

Docker images --filter dangling=true

If dangling=false it will list all images other than dangling images

To delete only dangling images

Docker rmi $( sudo docker images -q --filter dangling=true)

**Difference between kill and stop?**

Stop: it will have a grace period to stop the process inside the container (10sec)

Kill: it will forcefully stop the container without thinking of any process running.

Pause: it will pause all processes within one or more containers.

Docker objects image, container, volume, networks & bridges.

**How do you check which are all ports mapped on to container?**

Docker ps port containerid

Prune: remove all stopped containers, images, and volume.

Syntax: docker image prune

Remove all dangling images or unused images.

Docker system prune: it removes all unused docker objects.

**Data-only container:**

It is used for persistent data storage example: databases in docker. The data-only container does nothing else except exposing a data volume. It is used to add volume to other containers.

**Postgres-** used to save and pull the data from the database. Postgres in one of the databases, we can use it as a container.

**Docker Networks/networking:**

For docker containers to communicate with each other and the outside world via the host machine, there must be a layer of networking involved which is docker networking.

**Different types of docker networks:**

* + - 1. Bridge/host networks
      2. Overlay networks
      3. Macvlan network

**Docker default networking (docker0):**

When docker is installed a default bridge network named docker0 is created, and each new docker container is automatically attached to this network unless a custom network is specified.

Besides docker0 two other networks get created automatically by docker.

Host (no isolation between host and containers on this network, to the outside world when they are on the same network)

None (attached containers run on container-specific network stack) or disabling all networks for a container.

**Bridge/host networks:**

Bridge networking is the most common network type. It is limited to a container within a single host running on the docker engine. Bridge networks are easy to create, manage and troubleshoot.

Bridge networks are usually used when your application runs in standalone containers that need to communicate.

Host: for standalone containers, remove the network isolation between the container and the docker host, and uses the host networking directly.

**Overlay networks:**

Overlay networks connect multiple docker demons together and enable warm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different docker demons.

This strategy removes the need to do OS-level routing between this container.

**Macvlan networks:**

Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. docker demon routes the traffic to containers by their MAC addresses.

**Docker composes / docker-compose file**: it is a tool for defining and running multiple container docker applications, with compose we use a YAML file to configure your application services. Then with a single command, you create and start all services from your configuration.

It is a YAML file defining service networks and volumes, it is used to run multiple containers as a single service.

We can start and stop all services using a single command.

Docker-compose up or docker-compose up -d (detached mode)

Docker-compose down

**Docker swarm:** Ihave not used docker swarm, but I have an idea of this, and I use k8(Kubernetes) for container deployment.

It is a container orchestration tool, meaning that it allows users to manage multiple containers deployed acrossmultiple host machines.

One of the key benefits associated with the operation of a docker swarm is the high level of availability offered for application.

**Install docker-compose:**

[**https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-compose-on-ubuntu-20-04**](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-compose-on-ubuntu-20-04)

Ass: Write a Jenkin file it has 3 stages

1. Need to check out the source from github
2. Build stage: docker build to build a new image
3. Deploy stage: docker run

And

Deploy to a different server

Publish stage: push the new image to the docker hub

**Advantages of Multistage build:**

The multistage build is used to harden the docker images

Multistage build helps in reducing the docker image size

In Multistage build we use lightweight images such as Alpine and Dabian.net images

Each FROM instruction can use a different base image.

**Note:** Alpine is lightweight images and is about 5 MB.

In order to build the docker images we need small-size images (advantages).

**\*\*\*CICD Pipeline (end-to-end process) \*\*\***

Once the developer completes his work, he will raise a PR (pull request) as soon as the developer raises the PR local build job will get triggered. Once the build is successful, the reviewer or approver will merge the code to the central repo by clicking on the merge button (the merge button will not be enabled in the pull request if the local build fails). As soon as merges with the central repo Jenkins’s pipeline job will get triggered.

* + - 1. **Build stage:** we build respective docker images once the docker images complete build stage is completed.
      2. **Publish and Test:** here we run some basic test cases written by the dev and QA team, once the test cases are completed, we send reports to the QA team and push the successful artifacts to the docker registry.
      3. **Deliver:** we do the delivery to the lower environments once the QA test is successful, we do a deployment to production environments

**Docker security:**

These are four major areas to consider when reviewing docker security.

Intrinsic security of the kernel and its support for namespaces and Cgroup

The attached surface of the docker demon itself

Loopholes in the container configuration profile either by default or customized by the customer.

(We should use always official docker images instead of customized)

The hardening security feature of the kernel and how they interact with containers.