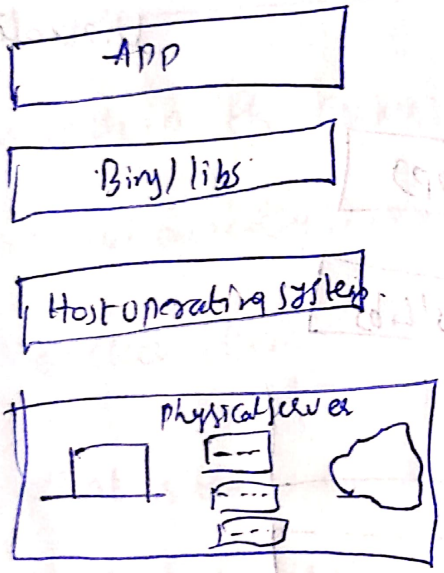
Docker

virtualization technology

Docker is one implementation of container based virtualization technologies

Pre-virtualization World:



Pre virtualization

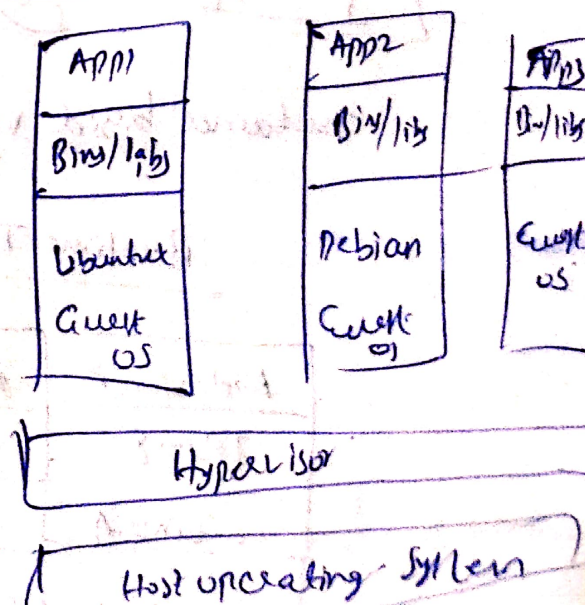
problems

- Huge cost
- slow deployment
- Hard to migrate

Hypervisor - based virtualization:-

Hypervisor providers

Vmware, virtualBox.



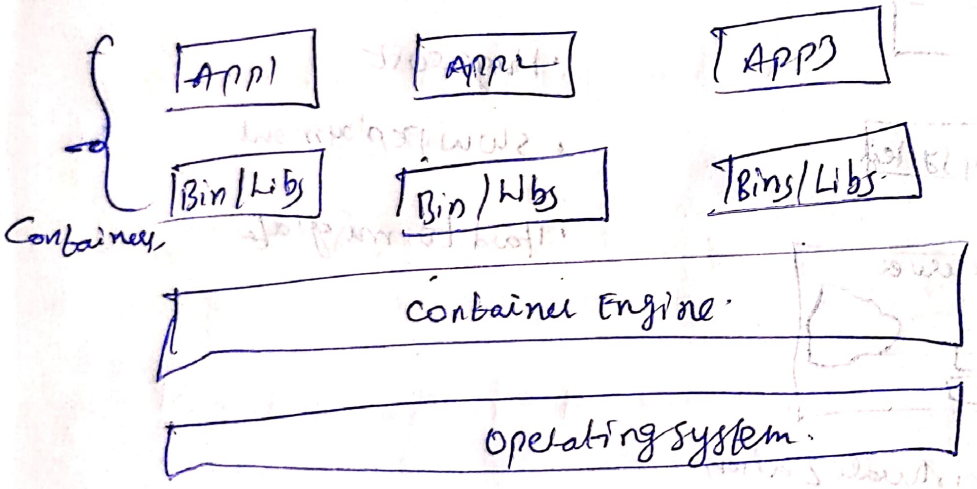
Benifits:

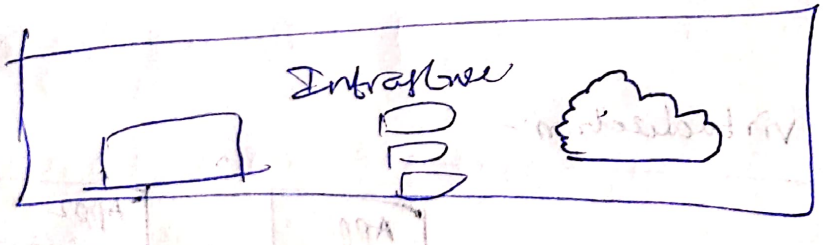
- cost Efficient
- Easy to scale

Limitations:-

- Kernel Resource Duplication
- Application Portability Issue

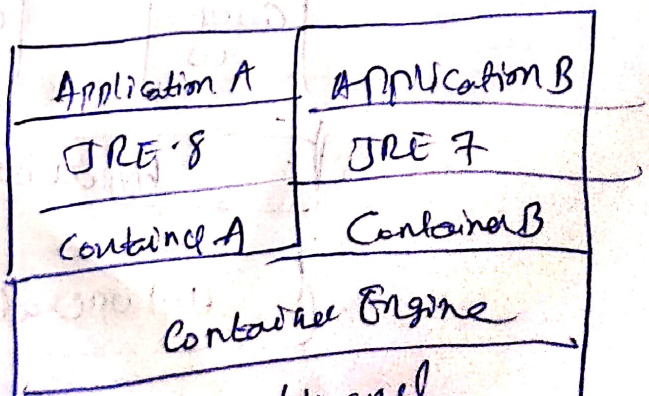
Docker is one implementation of one containe based Virtualization





container based Virtualization.

Runtime Isolation.



Container Virtualization

Benefits:

- Cost-Efficient
- Fast Deployment
- Guaranteed portability

Docker

Client-Server Architecture

Daemon with in the server:

User do not directly interact with daemon but interact with docker client.

Docker client is the primary user interface to docker it accepts commands from user and communicate with docker daemon.

There are two types of docker client

1. Command Line.
2. Automatic Docker client with UI

Images

- Images are read only templates used to create containers

- Images are composed of layers of other images.
- Images are stored in a Docker registry.

'Images are stored in a Docker registry

Containers

- If an image is a class, then a container is an instance of a class - a runtime object.
- Containers are light weight and portable encapsulations of an environment in which to run applications
- Containers are created from images.

Registries and Repositories.

1. A registry is where we store our images.
2. You can host own registry or you can use Docker's public registry which is called Docker Hub
3. Inside a registry, images are stored in repository
4. Docker repository is a collection of different docker images with same name.

why using official images.

- clear Documentation
- Dedicated Team for Reviewing Image content.
- Security Update in a Timely manner.

$ docker images

To run container

$ docker run busybox:1.24 echo "helloworld".

$ docker images

docker run busybox:1.24 ls /

The -i flag starts an interactive container.

The -t flag creates a pseudo-TTY that attaches stdin and stdout.

$ docker run -i -t busybox:1.24.

↳ moves inside container

/ # ls.

/ # touch a.txt

$ docker run -d busybox:1.24 sleep 1000

$ docker ps.

$ docker ps -a

$ docker run --rm busybox:1.24 sleep 1

$ docker ps -a → to list all containers

$ docker run --name hello_word busybox:1.24.

$ docker ps -a → to list all container.

docker inspect displays the low level information about a container or image

$ docker run -d busybox:1.24 sleep 100.

$ docker inspect container_id.

Docker port mapping and Docker logs command.

$ docker run -it -p 8888:8080 tomcat:8.0

$ docker logs Container_id

Docker image layers :- Docker Image made up of list of read only layers.

Image layers.

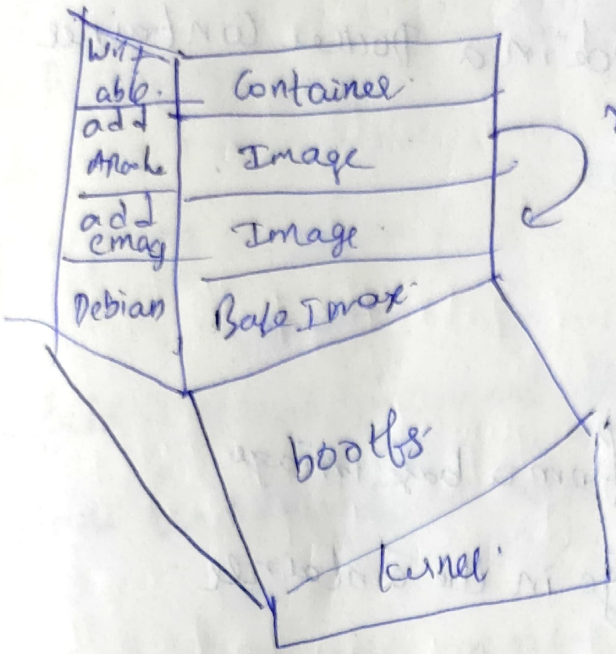


Image layers are stacked on top of each other to form a base for the container file system.

Docker is pulling the image layer by layer

$ docker history busybox:1.24

↳ Image.

1. All changes made in to the running container will be written in to the Writable layer

2. When the container is deleted, the Writable layer is also deleted, but the underlying image remains unchanged

3. multiple containers can share access to the same underlying image

Build Docker images

Two ways:-

1. Commit changes made in a Docker container
2. Write a Dockerfile

Steps

1. Spin up a container from a base image
2. Install Git package in the container
3. Commit changes made in the container

$ docker run -it debian:jessie

$ apt-get update && apt-get install -y git

Docker commit

Docker commit command would save the changes we made to the Docker Containers file system to a new image

docker Commit container-ID repository_name

$ docker commit container ID usa/debian:1.00

$ docker run -it usa/debian:1.0

Build Docker Images

Using Writing a Docker file

- A dockerfile is a text document that contains all the instructions user provide to assemble an image
- Each instruction will create a new image layer to the image
- Instructions specify what to do when building the image

$ touch Dockerfile → creating Dockerfile

```
From debian:jessie
```

↓ base image

→ Instructions

→ Instruction to execute

```
RUN apt-get update
RUN apt-get install -y git
RUN apt-get install -y vim
```

$ docker build -t ~~name~~ bul/debian .

(.) building docker.

d

new image

Dockerbuild Context:

- Docker build command takes the path to the build context as an argument
- When build starts, docker client would pull all the files in the build context in to tarball, then transfer the tarball file to the daemon
- By default, docker would search for the Dockerfile in the build context path.

chain Run instructions:

- Each run command will execute the command on the top writable layer of the container, then commit the container as a new image.
- The new image is used for the next step in the dockerfile. So each Run instruction will create a new image layer.

- It is recommended to chain the RUN instructions in the Dockerfile to reduce the number of images layers it creates.

Sort multi-line arguments alphanumerically.

- avoid duplication of packages and make the list much easier to update

CMD instruction:

- CMD instruction specifies what command you want to run when the container starts up.
- If we don't specify CMD instruction in Dockerfile, Docker will use the default command defined in the base image.
- The CMD instruction doesn't run when building the image, it only runs when the container starts up

```
CMD ["echo", "helloWorld"]
```

```
$ docker build -t user/debian .
```

docker run container-ID> echo "hello docker"

Docker cache

- Each time Docker executes an instruction it builds a new image layer.
- The next time, if the instruction doesn't change, Docker will simply reuse the existing layer.

Aggrisive caching

Chain instructions

FROM ubuntu:14.04

~~RUN apt-get update~~

RUN apt-get update && apt-get install
-y \
git \

CMD

Solution 2: Specify --no-cache option

docker build -t user/debian . --no-cache
=true

Copy Instruction:

The copy instruction copies new files or directory

from build context and adds them to the file system of the container.

$ touch abc.txt

Copy abc.txt /src/abc.txt

↳ copying file to container.

$ docker build -t usa/debian .

$ docker run -it container-ID

/# ls

/# cd src

/src# ls

ADD Instruction

ADD instruction can not only copy files but also allow you to down load the file from internet and copy to the container.

ADD instruction also has the ability to automatically unpack compressed files

Push Images to docker hub

$ docker images

$ docker tag IMAGE-ID Repository name

latest tag:

- Docker will use latest as a default tag when no tag is provided.
- Image which are tagged latest will not be updated automatically when a newer version of the image is pushed to the repository

- Avoid using latest tag

$ docker images

$ docker login --username=tutorial

$ docker push tutorial/debian:1.01

↓

Repository name

$ docker images

$ docker run -d -P 5000:5000 ImageID

$ docker-machine ls

$ docker exec -it → interactive mode
container_Id

↓

allows you to run a command in a running Container

$ docker exec -it container_Id bash

$ ps axw

↓

all running processes.

Implement a simple key-value lookup Service.

$ git stash && git checkout V0.2

$ docker build -t dockerapp:V0.2

$ docker run -d -p 5000:5000 dockerapp:V0.2

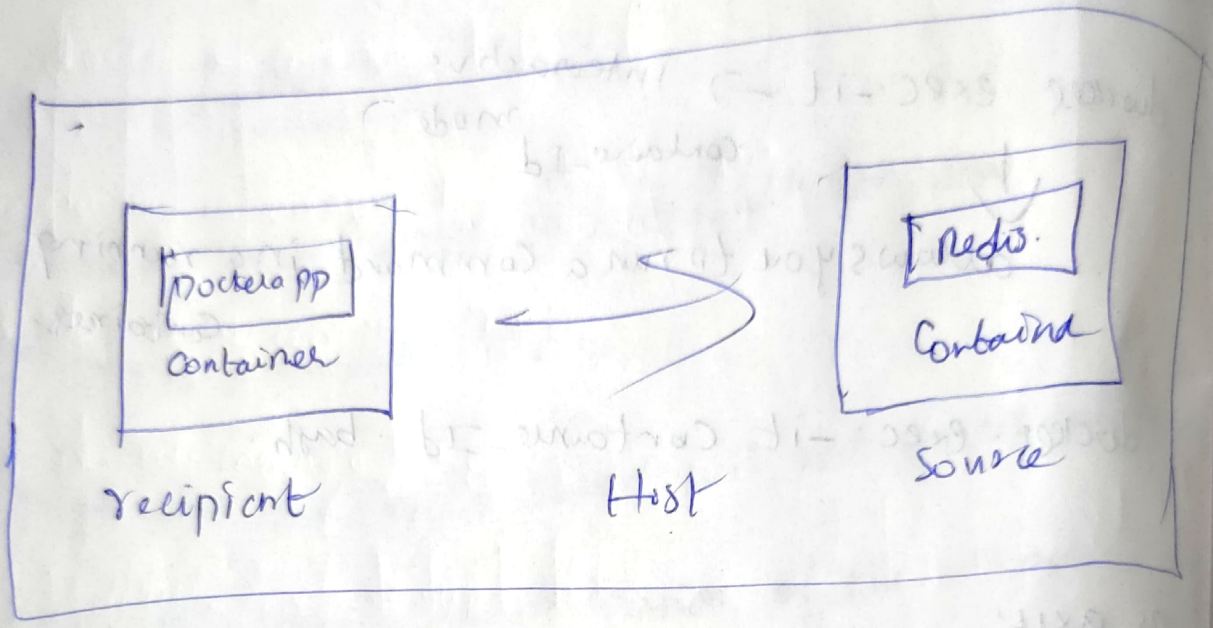
redis

*In memory data structure store, used as database, Cache and message broker.

*Build in replication and different levels of persistence.

Docker container links

Allows containers to communicate with other containers



$ docker run -d --name redis:3.2.0

$ docker run -d -p 5000:5000 --link redis docker-app:v0.3

$ docker exec -it container-ID bash

$: app $ more /etc/hosts

$ docker inspect redis container-ID | grep IP

$: /app $ ping redis

Benifits of Docker Container links:

The main use for docker container links is when we build an application with a microservice architecture, we are able to run many independent components in different containers.

Docker creates a secure tunnel b/w the containers that don't need to expose any ports externally on the container.

Docker Compose:

Manual linking containers and configuring services become impractical when the number of containers grows.

Docker compose is a tool for defining and running multi-container Docker applications.

$ docker-compose version

$ touch docker-compose.yml

$ sudo docker-compose.yml

$ docker ps

$ docker stop Container_ID

$ docker-compose up -d

↓

to run compose in background

$ docker-compose ps

$ docker-compose logs -f

$ docker-compose logs dockerapp

$ docker-compose rm

$ docker compose build

$ docker-compose up -d

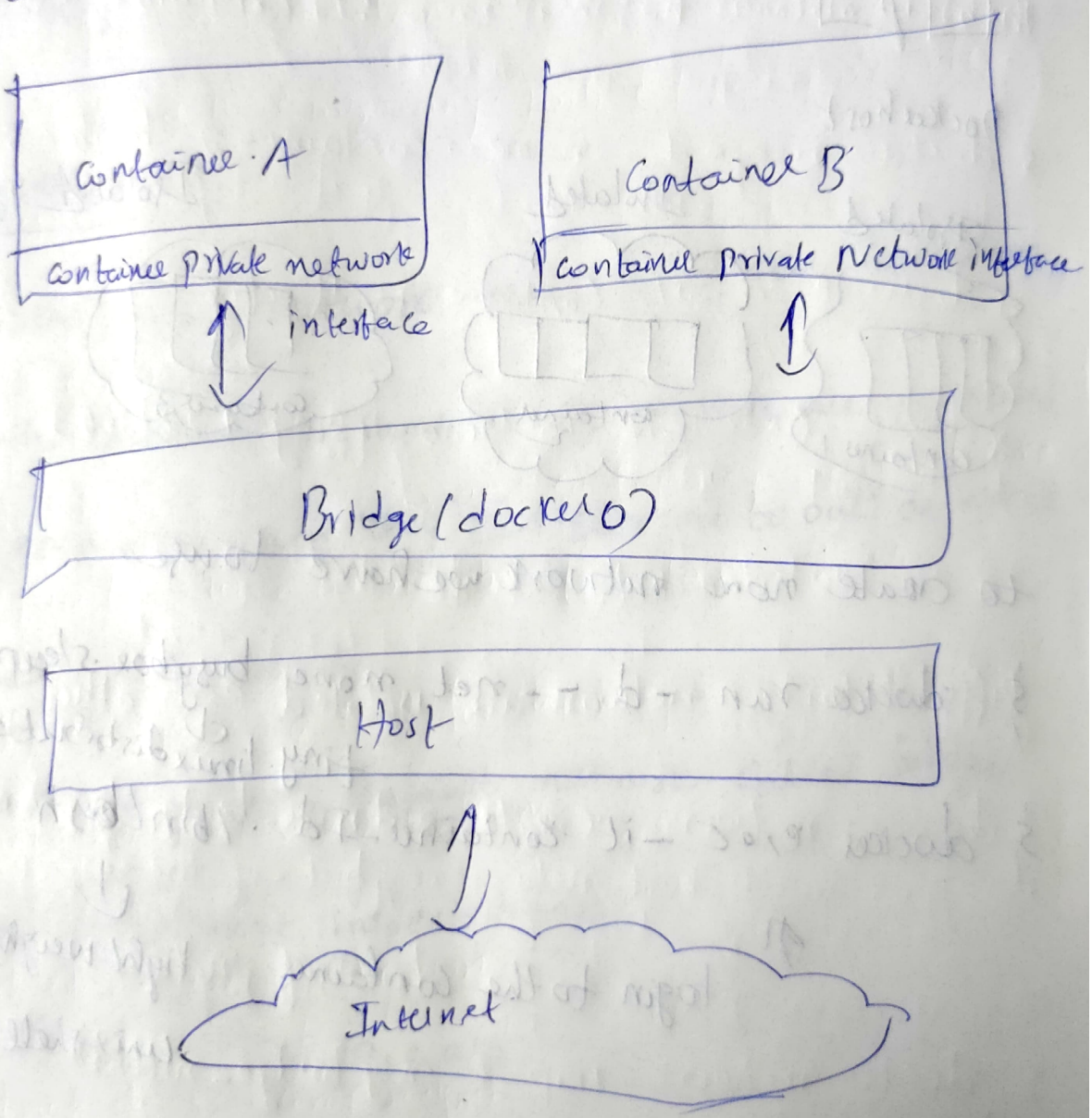
$ docker ps

- docker compose up starts up all the containers
- docker compose ps checks the status of the containers managed by docker compose.
- docker compose logs with dash f option outputs appended log when the log grows
- docker compose logs with container name in the end outputs the logs of a specific container.
- docker compose stop stops all running containers

- docker compose rm removes all the containers.
- docker compose build rebuilds all the images

Docker Networking:

Default Docker Network model



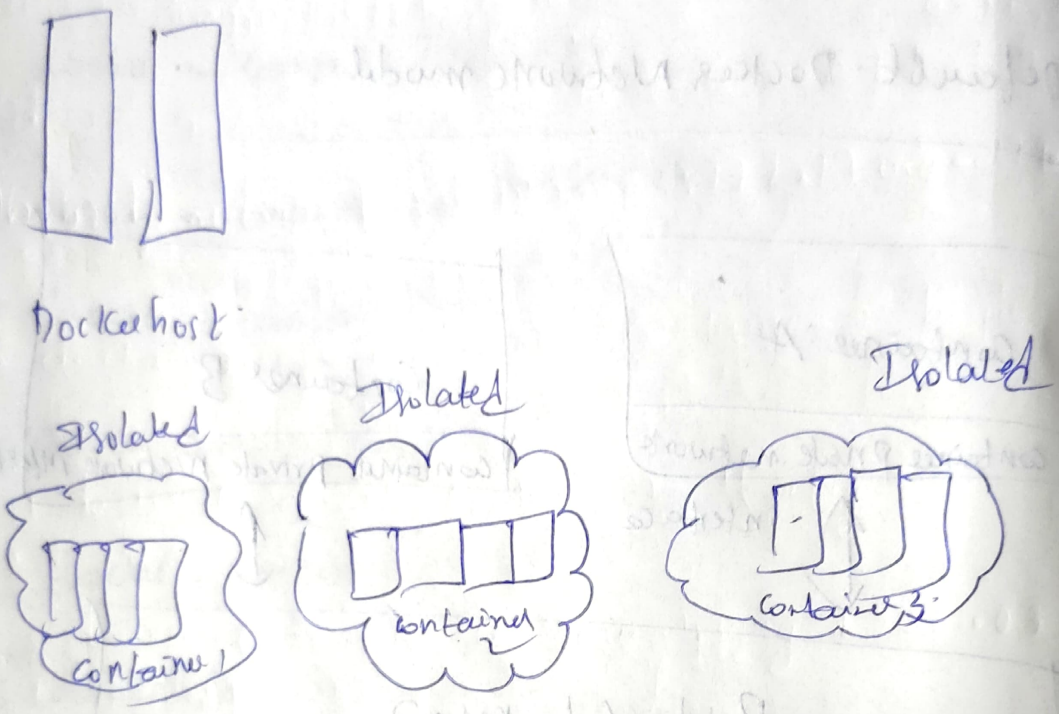
Docker network types:

1. closed network / None network
4. overlay network

None network:

This network does not have any access to outside world

The none network adds a container to container stack



to create none network we have to use

$ docker run -d --net none busybox sleep 1000
(tiny linux distribution)

$ docker exec -it container_Id /bin/bash
(login to the container) (light weight unix shell)

$ ping 8.8.8.8

/# ifconfig

loopback interface can not connect to any other network

. provides the maximum level of network protection

. Not a good choice if network or internet connection is required

. suited well where the container require the maximum level of network security and network acces is not necessary

Bridge network :-

This is the default network model of docker containers.

All the containey in same bridge network are connected to each other and they can connect to outside world

$ docker network ls.

docker creates default network called bridge

$ docker network inspect bridge.

Subnet : "172.17.0.0/16"

IP range : 172.17.0.0 - 172.17.255.255

$ docker run -d --name container-1 busybox sleep 1000

$ docker exec -it container-1 ifconfig

there are two interfaces.

Privatenetwork interface → connected to bridge network.

loopback interface → used for internal applications

The bridge network has the ip range : 172.17.0.0 – 172.17.255.255

$ docker run -d --name container-2 busybox sleep 1000

$ docker exec -it container-2 ifconfig

$ docker exec -it container-1 ping 172.17.0.3

↓

To connect to outside.

Different bridge networks are isolated each other.

$ docker network create --driver bridge my_bridge_network

↓

driver of network

$ docker network inspect my_bridge_network

$ docker run -d --name container_3 my_bridge busybox sleep 1000

$ docker exec -it container-3 ifconfig

$ docker exec -it container_3 ping 172.17.0.2

$ docker network connect bridge container_3

$ docker exec -it container_3 ifconfig

$ docker network disconnect bridge container_3

$ docker exec -it container_3 ifconfig

- In bridge network, containers have access to two network interfaces.
  - A loopback interface - no network access to outside
  - private interface - connected to bridge network of host.
- All containers in the same bridge network can communicate with each other.
- Containers from different bridge networks can't connect with each other by default.
- Reduces the level of network isolation.
- small network on single host.

Host and overlay network:

Host Network:

- The least protected network model, it adds a

Container on the host's network stack

• container deployed on host stack have full access to the host's interface

• This kind of containers are usually called open containers

$ docker run -d --name container-4 --net host busybox sleep 1000

↓

Integrate over network

$ docker exec -it container-4 ifconfig

• Host Network:

1. minimum network security level.

2. No isolation on this two type of open containers

Overlay network:

• supports multi-host networking out-of-the-box.

• Require some pre-existing conditions before it can be created.

— Running Docker engine in swarm mode.

— A key-value store such as consul

Define container networks with docker compose

git stash && git checkout

$ docker-compose up -d

↓

to fire up redis

$ docker network ls

$ docker-compose down

↓

to stop all containers

```
networks:
  my_net:
    driver: bridge
```

creating own network.

Unit tests in containers:

Unit test should test some basic functionality of our docker app code

$ docker-compose up -d

↓

building all containers

docker-compose run

Continuous Integration:

Continuous integration is a software engineering practise in which isolated changes are immediately tested and reported when they are added to a large code base

CI server develop a image and push to docker registry.

- SSH keys are way to identify trusted computer without involving passwords.
- Generate a SSH key pair and save the private SSH key in your local box and add the public key to your Github account.

The SSH public key file usually sit under ~/.ssh/ directory

```
- deploy:
    name: push application Docker Image
    Command:
        docker login -e $ Docker Hub Email
```

Tag the Docker images with two Tags:

1. commit hash of the source code.
2. latest

$ docker-compose up -d

↓

to start all containers

Running Docker in production:-

- Distributed web app can be deployed at scale using Docker

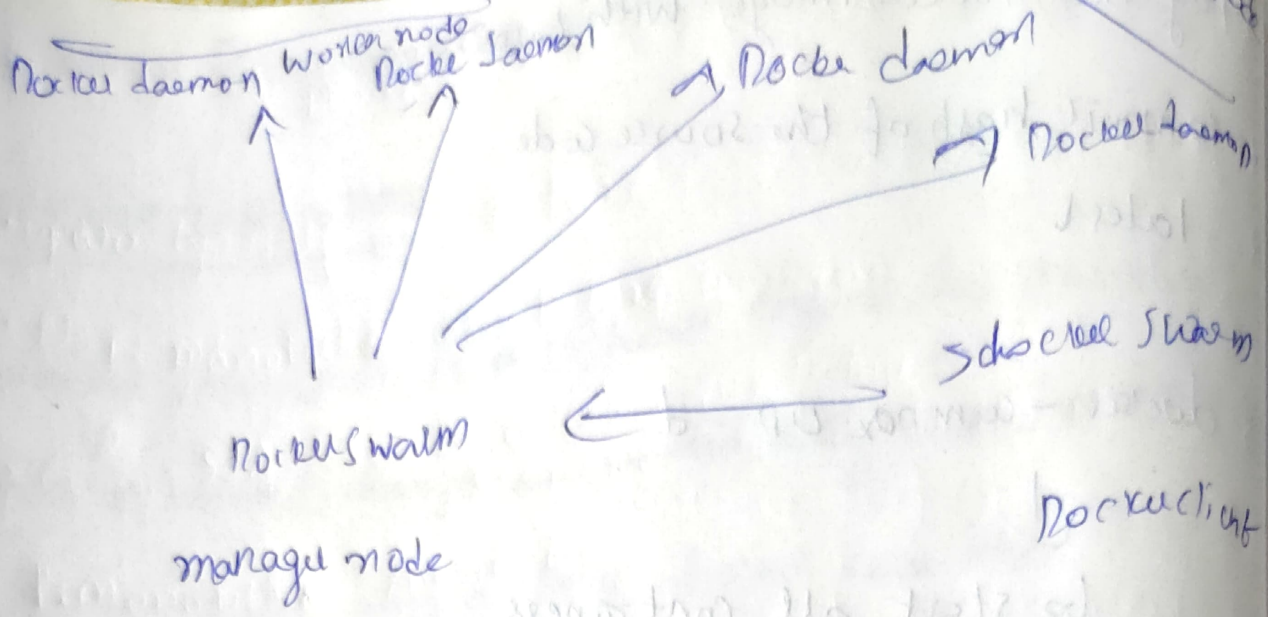
Digital ocean is a cloud infrastructure provider.

$ docker-machine ls

$ docker-machine create --driver digitalocean --digitalocean-access-token token-id docker-app-machine

- Docker Swarm is a tool that clusters many Docker Engines and schedule containers.

- Docker swarm decides which host to run the container based on your scheduling methods



- To deploy your application to a swarm, you submit your service to a manager node.
- The manager node dispatches units of work called tasks to worker nodes.
- manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm.
- Worker nodes receive and execute tasks dispatched from manager nodes.

$ docker-machine create --driver digitalocean --
digitalocean-access-token tokenId
swarm-manager

Docker Swarm commands:

- docker swarm init
  - Initialize a swarm. The docker engine targeted by this command becomes a manager in the newly created single-node swarm.
- docker swarm join
  - Join a swarm as a swarm node
- docker swarm leave
  - leave the swarm