DOCKER

Physical servers:-

In previous days we used physical servers to deploy applications

Drawbacks:-

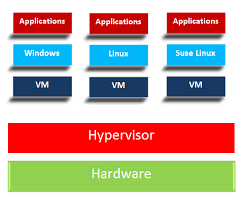
1. Unused resources(ram, cpu, hdd)
2. Costly to buy servers
3. Space consuming

Virtual technology:-

Later came virtual technology. Virtualization makes one physical computer acts and perform like many computers. Vmware is hardware level virtualization

In a server we install host o.s (linux/windows) and then install hypervisor software. Now hypervisor creates many virtual machines. In each vm we can install guest o.s and on top applications will be installed

* A vm is nothing but a file (bin/lib files) which acts like a physical server
* Using hypervisor we can create 15-20 vms



Benefits:-

1. Can easily move vm’s to other data centers
2. Low cost
3. Scalability (can scale up and scale down servers quantity [increase])
4. Can create multiple vm’s so we can run multiple o.s
5. Conserve less power
6. Less space consumption

Drawbacks:-

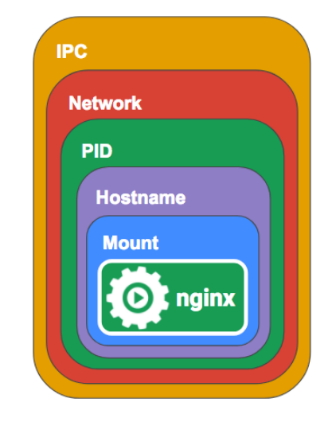
1. Each vm will have copy of o.s files, which makes booting slow
2. vm hard disk size is too large and difficult to move between environments
3. Performance down
4. We need to provide motherboard, cpu, ram o.s to each vm

Containers:-

Container is o.s level virtualization which abstracts user space. Containers don’t require ram, CPU, hdd, o.s instead they share host kernel & resources

Containers are lightweight since it doesn’t have o.s. It is better to run one service in a container.

Inside container we have code, libraries, package manager, application, and data. Outside the container we have logging, monitoring, remote access, network configuration

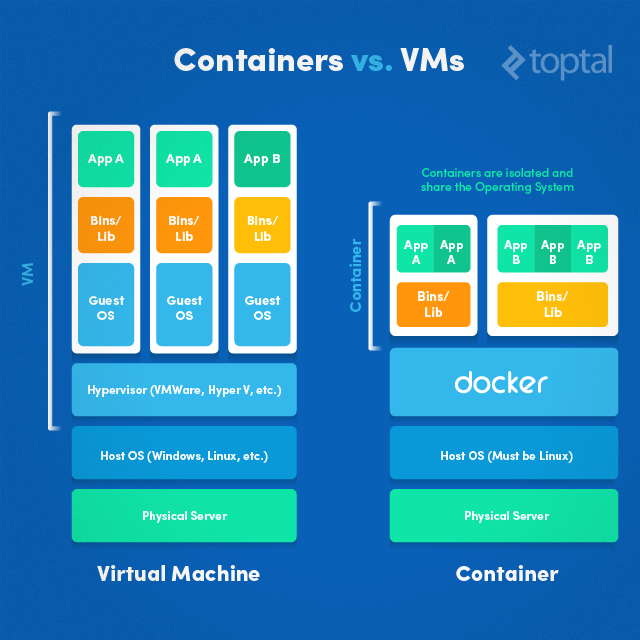


Benefits:-

1. Portability
2. Stability
3. Light weight virtualization
4. Boot time is fast
5. Fast deployment
6. Isolation = every app works in its own container n don’t interfere other aps

cons:-

1. Managing huge amount of containers is challenging
2. Increased complexity due to addition layers
3. We cant create windows containers on linux host o.s



Stateful vs. Stateless:-

Stateless means your application doesn’t store data on local file system (are generally micro services or containerized applications) (in calculator if you do 2+2=4, it displays result but doesn’t store the result).

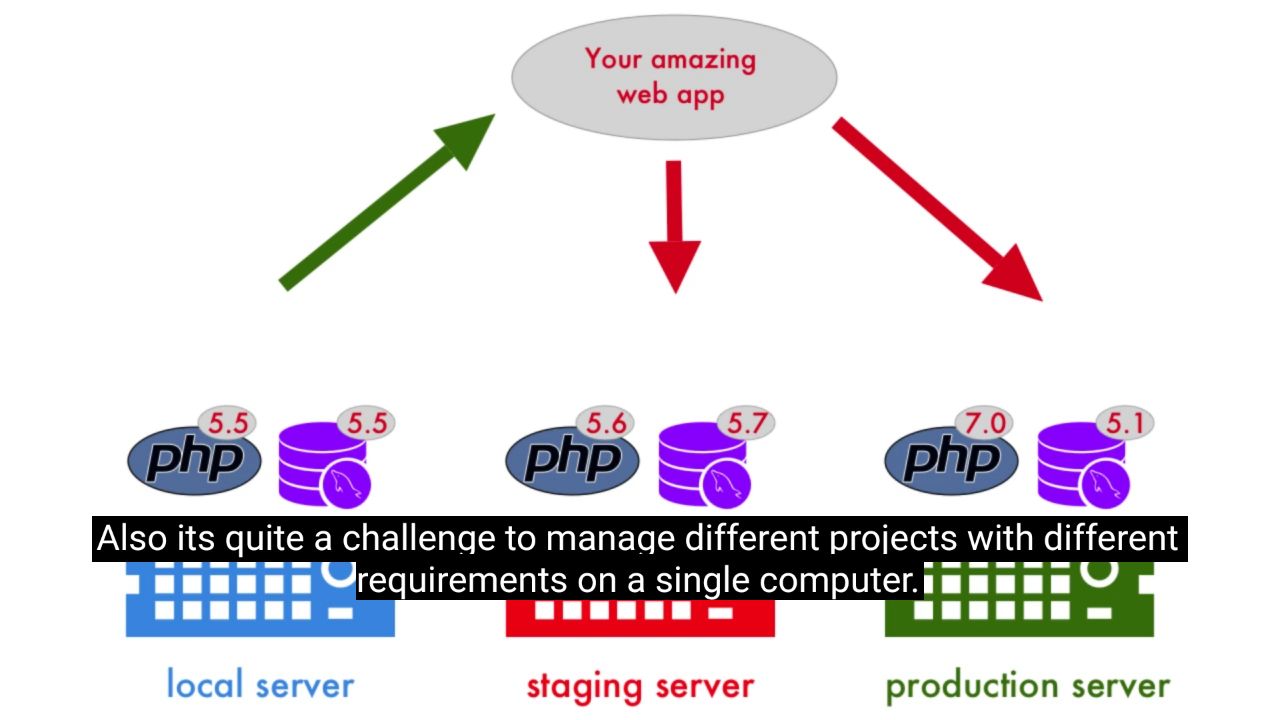
Ex:- web services (frontend ui) such as HTTP

Stateful applications needs volumes

Docker can handle both Stateful and stateless applications.

DOCKER INTRO

Sometimes application runs on developer system, but don’t run on testing/production/staging environment or other developers machine due to dependency issues & docker solve this problem



Docker is an open source tool that allows you to build, deploy & run any applications

Build/create---- it allows you to create application

Deploy/ship---- you can deploy app in testing/staging/production

Run --- you can run app regardless kernel version, host disto,

Docker packages your application with its libraries and dependencies

We have development lifecycle as

1. Plan

2. Develop

3. Deploy

4. Test

Docker is mainly used at deployment stage

Advantages:-

1. Using docker we can run application on any platform
2. Portability
3. It avoids you to configure application multiple times on different environment
4. Isolation=every app work in their own container and don’t interfere other apps

Disadvantages:-

1. Limited to linux based o.s applications
2. Can’t create windows container on linux host o.s

TERMINOLOGY:-

Build=process of creating image from docker file

Run=process of creating container to run applications

Pull=downloading image from docker hub

Node=a server (V/P) where docker is installed

Persistent storage=it is a data storage which lives even if container/machine s crashed (docker data volume, links/port forwarding)

Vertical scaling=increasing system resources like memory, CPU, hdd… to a machine

Horizontal scaling=adding more machines into a pool

Monolithic =a single large application which is tightly coupled and should be pushed all at once (or) all different business service are packed into single service which is tightly coupled

Monolithic application has 3 layers

1. Presentation layer = frontend, ui (html, css, javascript...)
2. Application layer = business logic (java, python, c...)
3. Data layer = database (mysql/oracle/mongodb)

\*presentation layer will communicate with application layer as well as database layer with api calls & application layer will communicate with database layer with data layer

DisAdvantage:-

1. Application is very large
2. Scaling application is difficult
3. Frequent deployments are not easy
4. If one service fails entire application goes down
5. Have to use single technology stack (only java or only python)

Micro-services=creating a small service for specific concern according to the business requirement (or) a smaller component of application which can independently run. Each service can be deployed into a container. These services are loosely coupled, Ex: - based on the business function we break into smaller components like login service, cache service, user management, user roles, search engine, e-commerce cart.

Use case:-

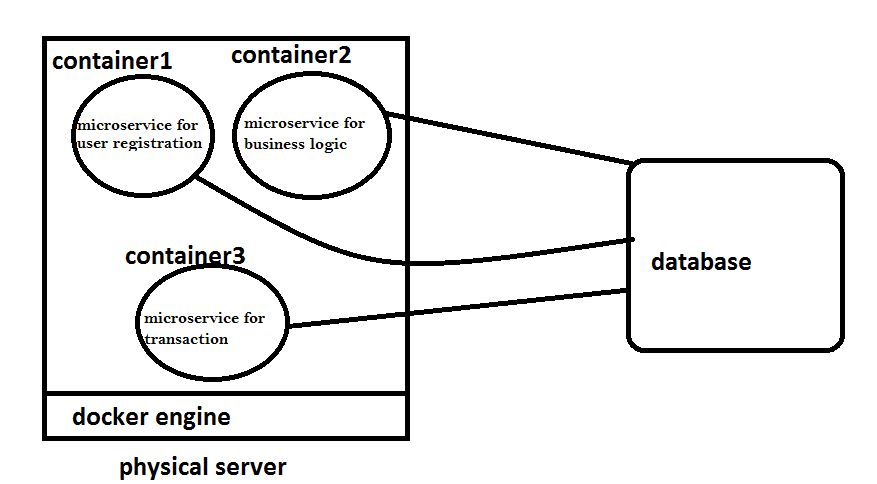
Assume during big billion day I need to scale my entire application, architecture. But if we use micro services we can scale only the services we want ex:- every user tries to login on that day so I will scale only login services, since each service is deployed into a container/pod I can only scale login service containers/pods. Thus micro services are advantageous

ADVANTAGES OF MICROSERVICE:-

1. Language independent
2. Easy development
3. Works well with containers
4. If one micro service fails the application doesn’t fail
5. Scalable

DIS ADVANTAGES OF MICROSERVICE:-

1. If network is slow then communication between different micro services is difficult
2. Number of microservice may spawn means one service has v1.0, v1.1, v1.2



Cloud native applications = micro services + devops practice + continuous delivery

Dockerizing application = the process of converting an application to run within a Docker container

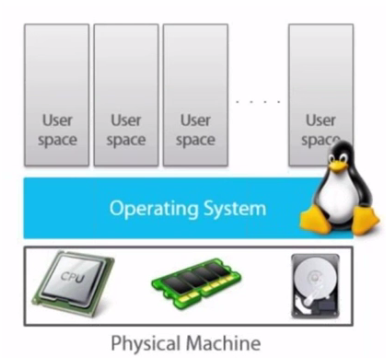
Namespace= the containers isolation is done by namespace. Namespace isolates system resources and virtualizes system resources. It is kernel feature.

There are six types of namespaces

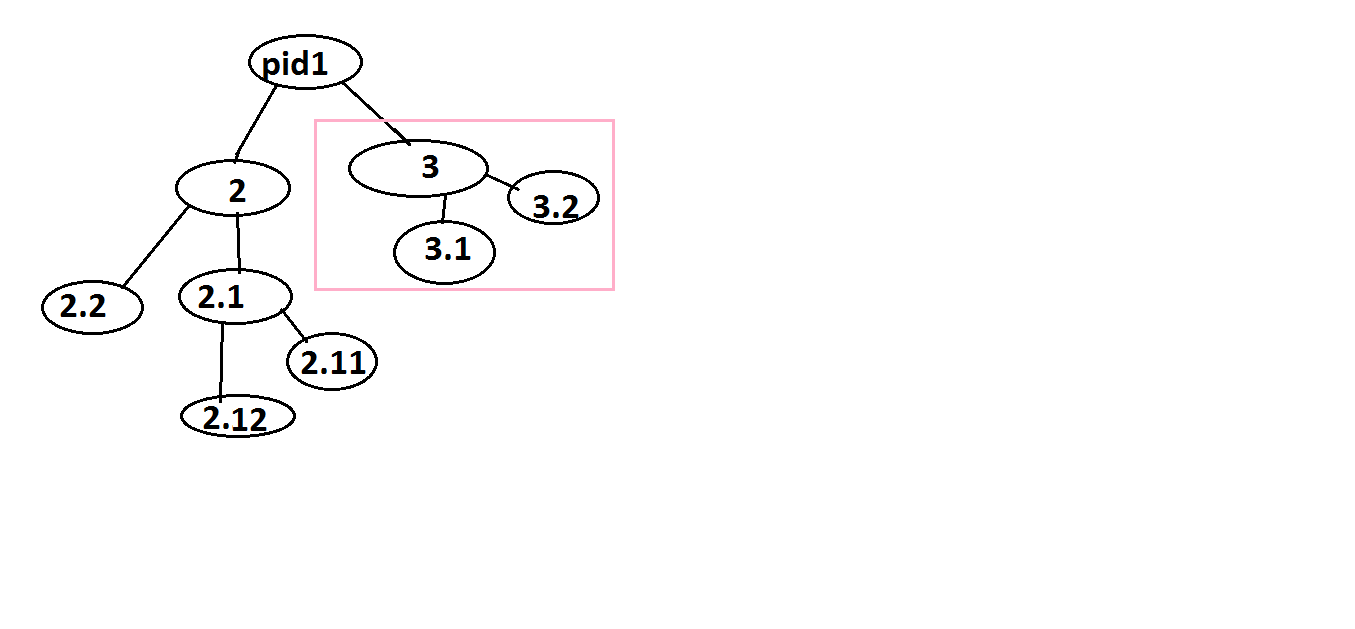
1. pid-- process name space (it isolates process)
2. net --- network name space (it isolates network)
3. mnt --- mount namespace (it isolates mountpoints)
4. uts --- hostname and nis domain
5. ipc--- inter process communication
6. user---user namespace (it isolates users)

Userspace= in a Linux O.S, for each user, there is some space allocated and that space has memory, cpu, networking, etc…

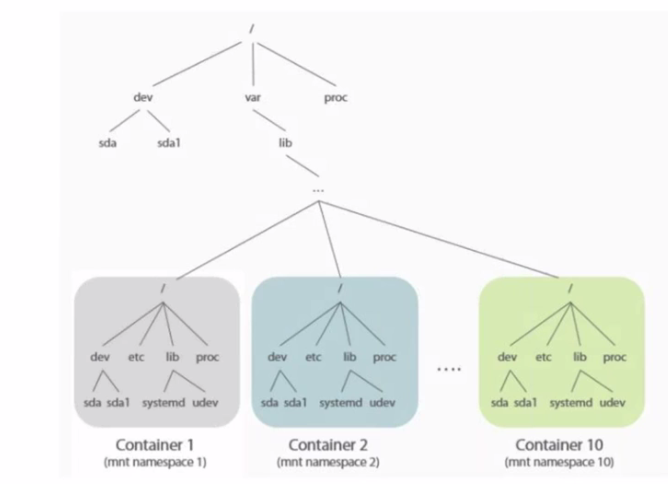
Each userspace is called container. This userspace is isolated.



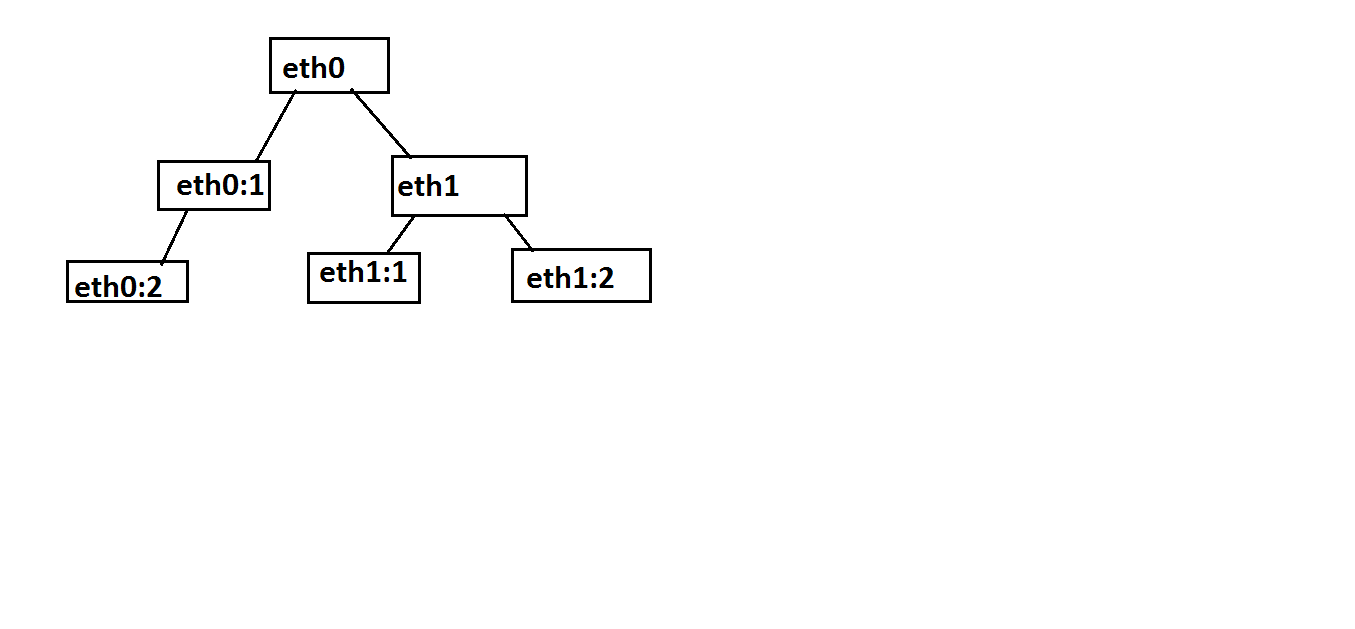
Process tree=



Filesystem tree=



Network tree=



Cgroups= if you want to increase/decrease system resources like system memory/cpu time/network bandwidth/disk bandwidth to a container. We can do using cgroups

Cgroups allocate system resources to an application

\*[[namespace, cgroups, chroot are kernel feature which divide userspace and forms a container]]

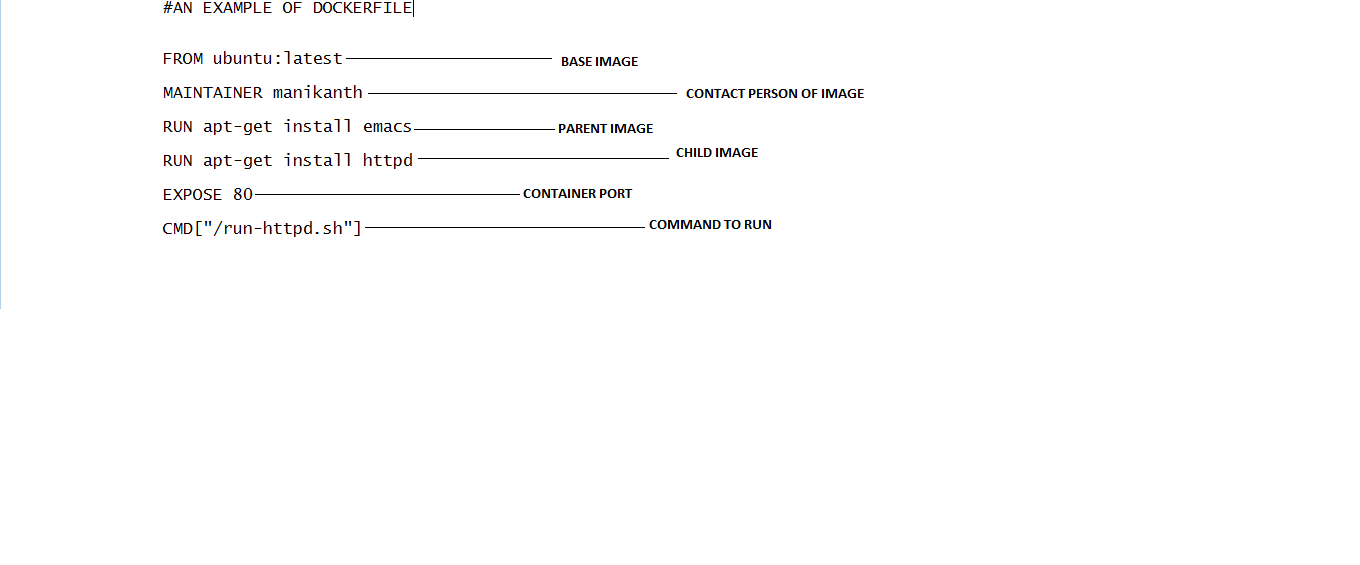
Ufs= collection of different file systems [and directories (called branches)] into single logical file system. REF=https://www.linkedin.com/pulse/drooling-over-docker-2-understanding-union-file-systems-rahul-singh-1/

#docker has built special driver on top of lxc container (o.s level containers)

DOCKER FILE:-

Docker file is nothing but a text file that has series of instructions to build image

* the name of a docker file should always be Dockerfile ex:- vi Dockerfile



* FROM = base image on which you are going to build image (ex:- if you want build tomcat image on java then base image will be java)

FROM <image>:<tag>

Ex:- FROM ubuntu:latest

\*Parent image = which depends on base image

* LABEL=similar to maintainer

Ex:-

LABEL maintainer:‘java home cloud’ \

employee\_name:‘manikanth’ \

version:1.0

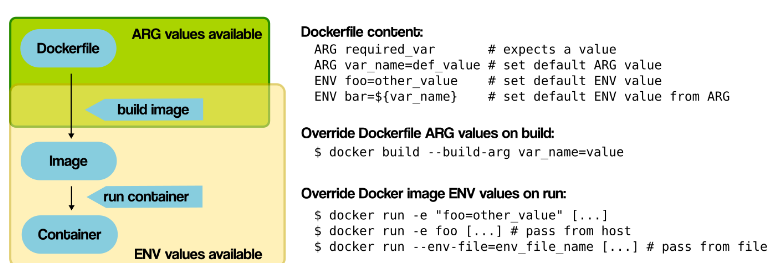
* MAINTAINER = author of docker file, image, container

MAINTAINER <Name>

Ex:- MAINTAINER mani [mani@gmail.com](mailto:mani@gmail.com)

* EXPOSE = expose or map container port to host port EXPOSE <port>

Ex: - EXPOSE 8080



* ARG = used same like environment variable but the image will set only at building image. For variable reusability we also use arg instruction

ARG <key> <value1>

Ex: - VAR=JAVA\_HOME

ENV JAVA\_HOME=/usr/lib/java

* ENV = if you want to set environment variable, used at run time ENV <key>=<value1> <value2> <value3> (or) ENV <key> <value> Ex: - JAVA\_HOME=/usr/lib/java
* COPY = copies file from host machine to container

COPY <src>... <dest>

Ex: - COPY /john.mp3 /usr/home/john

* ADD = similar to copy but can handle both tar files and remote url

ADD <src>... <dest>

Ex: - ADD f1.tar.gz /data

Ex: - ADD http container

* VOLUME = invoking data volume inside a container

VOLUME ["/data"]

Ex: - VOLUME [“/dir”]

so docker creates dir volume inside container

* USER = by default root user will be created in a new container. If you want to add some new user in container, you use this command.

🡪user can be overridden by “u” flag

#docker run –it –u xxx /Dockerfile

* WORKDIR = by default docker runs everything in / (root) inside a container but if you want to run in particular directory then you have to set workdir

WORKDIR /path/to/workdir

Ex: - WORKDIR /usr/home/mani

WORKDIR can be changed at runtime ex:- docker run -w dir-path

It is just like “cd” used for navigating from one directory to other

* CMD = command that executes as soon as container is launched, takes only one command (latest cmd) for docker file although you specify many

CMD ["executable/command","param1","param2"]

Ex: - CMD [“/bin/ping”, “localhost”]

CMD ["sleep", "5"]

If you add one more command in next statement. Then the last/latest cmd will be executed & if you specify cmd while docker run then also the latest cmd will be executed

\*cmd is fed to entrypoint

\*Docker cmd will be used only once while running image to container

Whereas run command used to execute while building images from docker files

* ENTRYPOINT = command that runs when container starts, just like shell when system started,

.

1. We specify entry point in docker file. If you run docker with some parameter (docker run /bin/bash) then entrypoint will append those parameter

#we can append parameters to entrypoint not to cmd other than this both are same

ENTRYPOINT [“cmd/script”]

Ex:- FROM Ubuntu

ENTRYPOINT ["sleep"]

CMD ["5"]

* RUN = generally used for installing & updating packages, adding directories/files/users... For every run statement there will be a new layer created

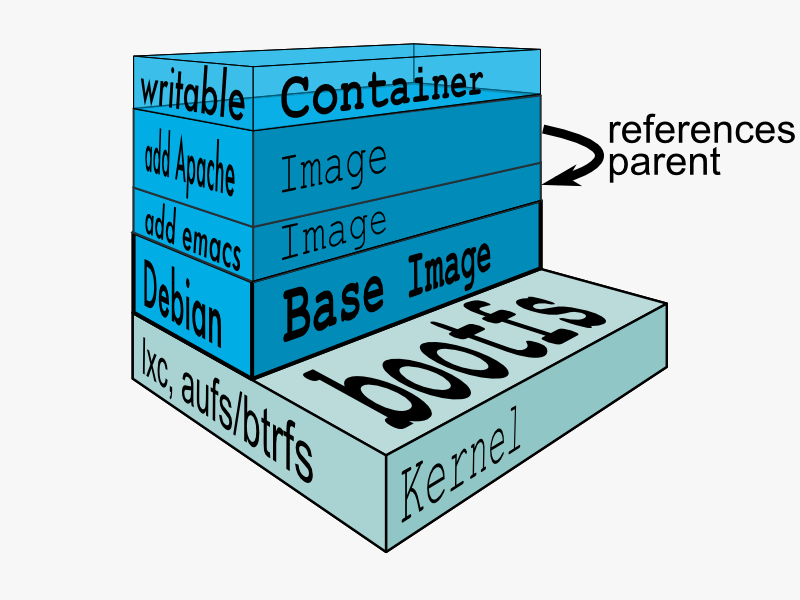
RUN <command>

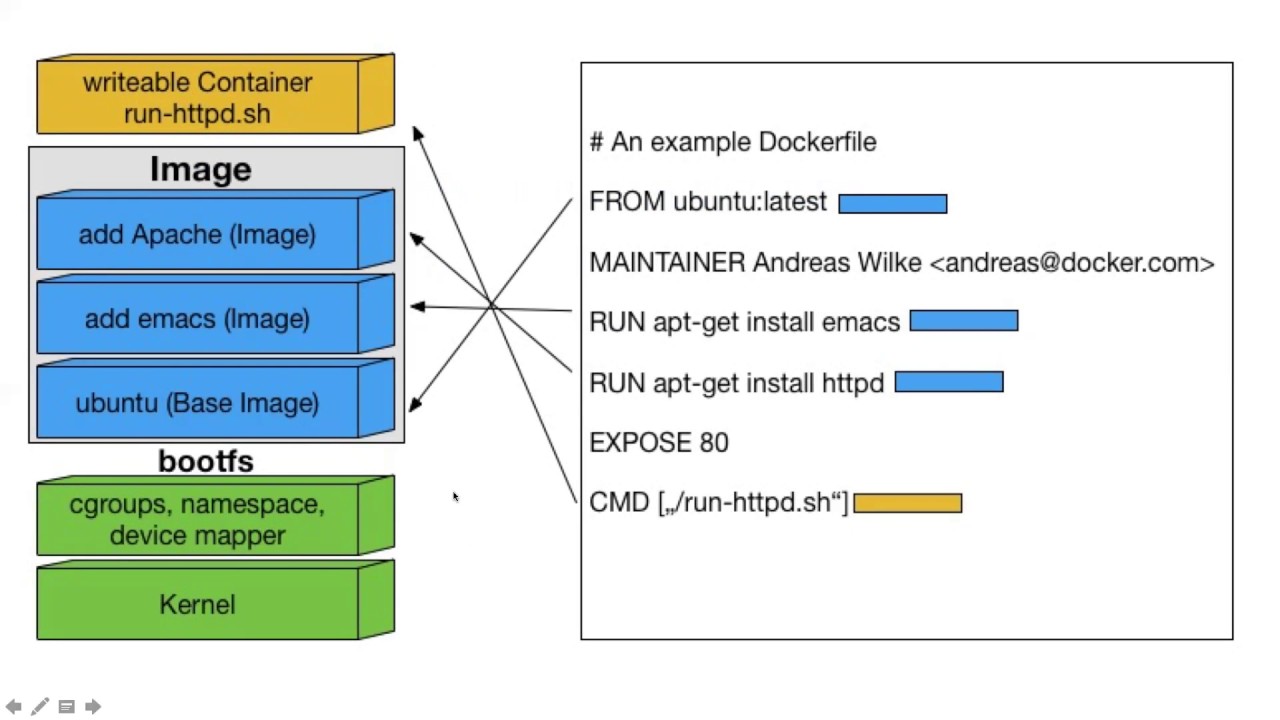
Ex:- RUN yum install httpd && yum clean

\* for every instruction a new image will be created inside docker image

DOCKER IMAGE:-

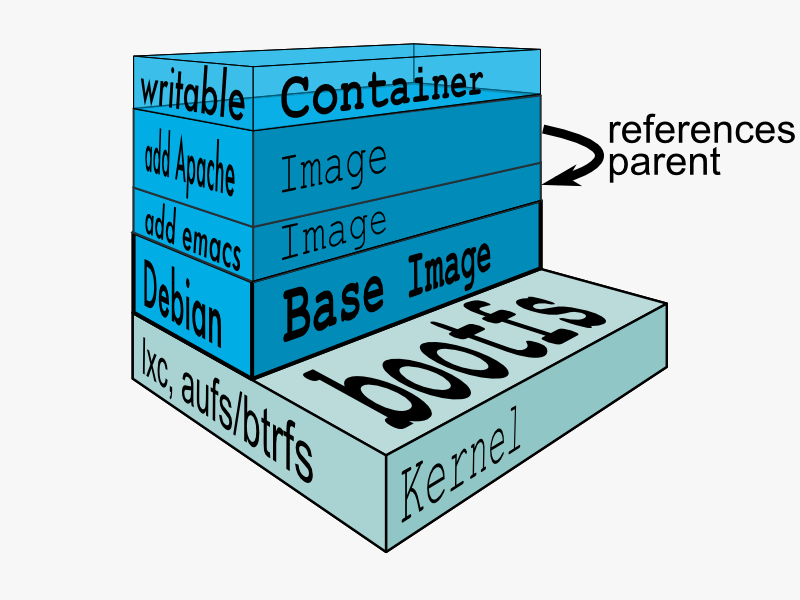
* When you build docker file, docker daemon creates docker image
* Each instruction in docker file is treated as an image
* Docker image consists multiple layers, each layer have an image and Each image refers to other image & each layer is independent of other
* Image is snapshot of filesystem (or) collections of files and some meta data
* Docker image is read only mode and immutable (can’t be modified)
* All images are stored in docker hub/local repo
* With one image we can create multiple containers





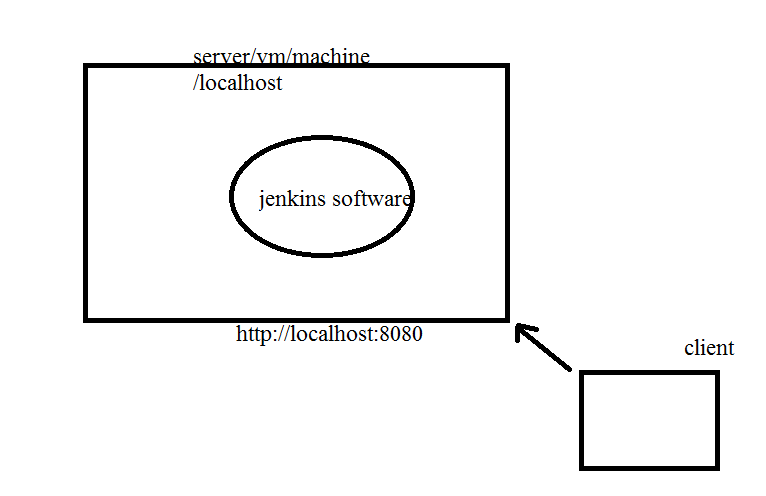
DOCKER CONTAINER:-

When we run docker image, docker daemon creates container. It is similar to docker image except read write filesystem on the top layer

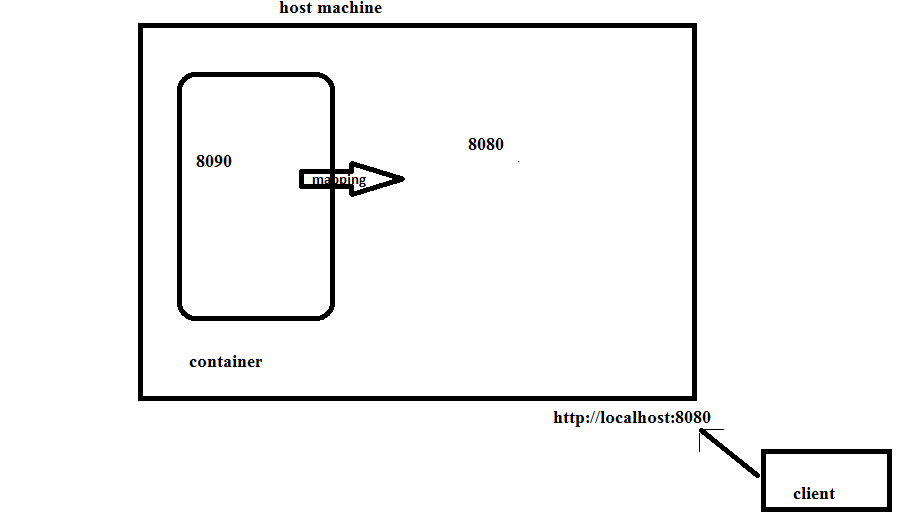


Container has unique id,

We usually install Jenkins in a host machine and access through 8080 port



But assume we installed Jenkins in docker container and this docker container is in host machine. Now you can’t access through 8080 since it is host port. So we map/expose container port to host port. So when client access 8080 host port, it will redirect to container port



syntax:-

docker run -d -p hostport:containerport image:latest /bin/bash

ex:- docker run -d -p 8090:8080 ngnix:latest /bin/bash

here -p is called port binding

#docker port containerid ----- to check container port

DOCKER REGISTRY:-

It is a centralized repository maintained by docker where docker images are stored. So docker images can be pulled or pushed

1. Public registry is free of cost and anyone can push/pull any images
2. Private registry is paid and private repository

\*docker registry is just like github

Ex:-dockerhub, ecr, gcr, jfrog, nexus

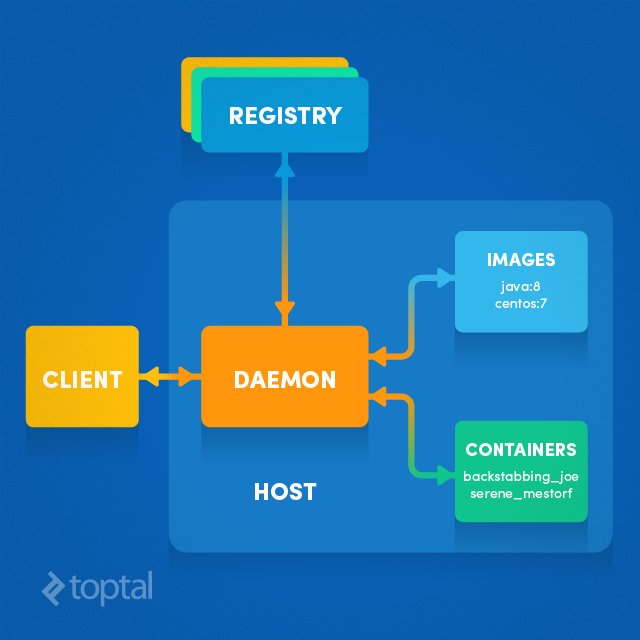
DOCKER REPOSITRY:-

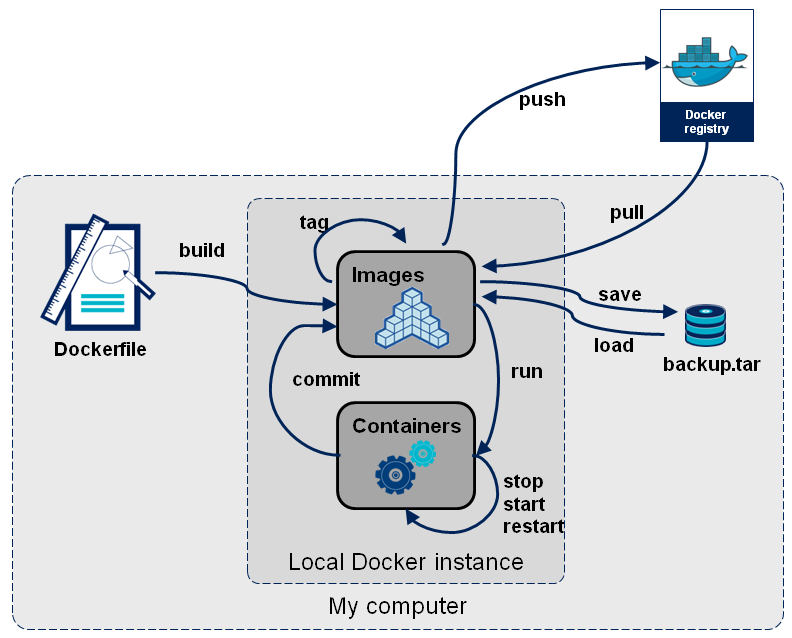
Docker repository is a repository inside docker registry just like we create a repository inside a github. A collection of same docker images with same name and different tags

ex:-kvvmanikanth/demo-repo

DOCKER WORKFLOW:-

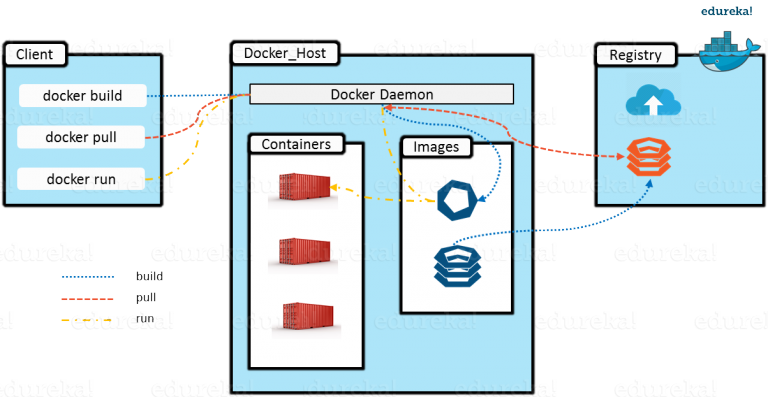
**Dockerfile** > (Build) > **Image** > (Run) > **Container**.





1. We create a docker file and build it, now docker image will be created
2. Either we can push/pull docker image from docker hub (or) we can run docker image to start container
3. Now we can install application in container & the container can be stopped/started/restarted
4. We can create container into image using commit

ARCHITECTURE:-



* In a server we install host o.s (Linux) and then docker software now we can create as many containers as possible.
* Docker client is the CLI where we issue commands, docker client interacts with docker daemon. Docker client & docker daemon can be on same host/different host
* Docker host is the system where docker software is installed.
* Docker engine (or) daemon which runs, monitors & orchestrate

CONFIG FILES:-

1. etc/docker---
2. /bin/docker --- docker location
3. /var/lib/docker---docker volume, aufs, tmp, vfs, repos.. are located here

DOCKER COMMANDS:-

1. docker run:-

if you run docker image docker container will be created and started (but immediately exit, since nothing is running. Exit means coming out from container to host machine)

#docker run imagename shell

Ex:- docker run ubuntu /bin/bash

#you need shell because all commands you want to execute will be in shell

-t = attach terminal to container (after login into computer we can see shell environment to execute some task, in the similar way inorder to do something you should have an environment(shell))

#docker run -t imagename environment

#docker run -t ubuntu /bin/bash

-i = interactive mode #docker run -it ubuntu /bin/bash

-d = daemonized mode (container will start and run in background, if we attach everytime we have to come out so container will be stopped. So we use daemonized mode) #docker run -d ubuntu /bin/bash

1. docker rename:-

Since we can’t remember container id every time we rename container. Although docker gives default container name

#docker rename old\_container\_name new\_container\_name

#docker rename peny\_cont1 ubuntulatest1

1. docker remove:-

Deleting a container (first stop container and delete container)

#docker rm container1id/name container2id/name #docker rm cont1

1. docker inspect:-

Shows all the information of container like ipaddress, mac address gateway, drivers, mounts, paths

#docker inspect containerid/name

1. docker list:-

* showing all running containers

#docker ps

* showing all running and stopped containers

#docker ps -a

1. docker start:-

Starts a container (just like a computer starts till you see login screen)

#docker start containerid/name

1. docker stop:-

To stop a container (just like switching a computer)

#docker stop containerid/name

Docker kill will send a message and immediately kills the container, but stop will wait for few seconds and then terminates the container

1. docker logs:-

To see what is running inside a container. just like /var/logs in linux

#docker logs -f containerid/name

If you want to see any output of a container then execute

#docker logs container-id (or) docker logs --tail 5 container-id

1. docker stats:-

If we want to see resource utilization like cpu, mem… of container

#docker stats containerid/name

ctrl + c to escape

1. docker copy:-

If you want to copy files/folders from host machine to docker container or vice versa or container to container

#docker cp src\_path dest\_path

#docker cp focused\_payne:/mani /home/docker----cont to host

#docker cp /manikanth newubuntu1:/home------host to cont

src=container or host

dest=container or host

1. docker attach:-

Connects to a running container, To enter into container from host terminal (just like logging into computer so you can see c,d,e and do some actions)

#docker attach containerid/name

Ctrl+p+q to come out of the container (runs in background)

\*docker attach command allows you to switch from host to container, now you can see what is inside a container

1. docker exec:-

From host, we can execute commands in already running/started container, without attaching/going into the container

#docker exec containerid/name command

ex:- [root@host1.com]docker exec 122eue34 touch f1

we are in host and we want to create a file in container. So we use docker exec command for that. Now whether a file is created or not just go inside the container by docker attach command

1. docker remove image:-

Deleting an docker image

#docker rmi imageid

1. docker login:-

Login to the docker hub repository, give your credentials

#docker login

1. docker push:-

Pushing docker image to docker hub

#docker push Username/RepositryName

1. docker search:-

Searching docker image in docker hub

#docker search Username/\*

1. docker logout:-

Logout from docker hub

#docker logout

1. docker pause:-

Docker pause will pause the process running inside a container

#docker pause container-name

1. docker prune:-

Docker prune will remove all unused containers, volumes, network and images (dangling and unreferenced)

#docker container prune------🡪remove all stopped containers

#docker image prune ------🡪remove all unused images

#docker volume prune------🡪remove all unused volumes

1. docker commit:-

Docker commit is used to convert a running container into an image. So you can launch container out of a docker image.

#docker commit -m "yourmessage" -a "maintainer" container\_name maintainer/tagname

ex:- docker commit -m "first docker comit" -a "manikanth" ubuntu.ngnix manikanth/ubungnix.1.0

#docker commit container\_id

DOCKER VOLUME:-

Every time we make changes inside a container and rebuild the image which is a headache. So we go for volumes in that case Volume is a special directory which is used to share data between two containers or host to container (just like nfs or hard link for clear understanding)

container1 container2

Volume

sharing data volumes are three types:-

1. Sharing data between two containers :-
2. Assume there are two containers as cited in figure. There is a data volume (directory) in container1
3. Even if the container1 is not running you can still share the data volume to other container and container 2 can access the contents
4. Suppose you shared data volume from container 1 to container2, you have made some changes in container1 and those will be reflected in container2 data volume
5. Data volume in container exists only if container exists irrespective container1 is stopped/running. If container1 is removed then you cant see the data volume in container2

syntax:-

#docker run --name container\_name -it -v containerpath image shell (-v = volume or mount)

ex:-docker run -it --name volcont1 -it -v /usr/data ubuntu /bin/bash

this will create a new ubuntu container with data volume volcont1 and the path will be /usr/data

so if you want to access /usr/data directory from volcont1 you need to create one more container

#docker run --name new\_container\_name -it --volumes-from container\_that\_u\_want\_to\_access --priviliged=true image shell

Ex:-docker run --name volcont2 -it –volumes-from volcont1 --priviliged=true ubuntu /bin/bash

Privileged=true means we have write permission in container1

Now go to usr🡪data🡪you shuld see f21 file

1. Sharing data between host and container:-

we will map host and container to share the data between them

syntax:-

docker run -it -v hostpath:containerpath image shell

ex:- docker run -it -v /media/mani/:/usr/data/ ubuntu /bin/bash

here -v is also called as bind mounts

Since we have mapped host path to container path. Whatever the contents in mani will also be present in ubuntu. If anything modified in host file, will also be reflected in container and vice-versa

#docker run --name test00 -it --volumes-from volcont --privileged=true ubuntu /bin/bash

\*\*if privileged is not given true then we can’t modify but just access

1. Creating independent volume:-

To create independent data volumes without relating to any container

#docker volume create --name datavolume\_name

#docker volume create --name datavol22

1. docker volume list:-

To see all data volumes

#docker volume ls

1. docker volume remove:-

To remove a particular data volume

#docker volume rm datavolume\_name

#docker volume rm --name datavol22

DOCKER NETWORKING:-

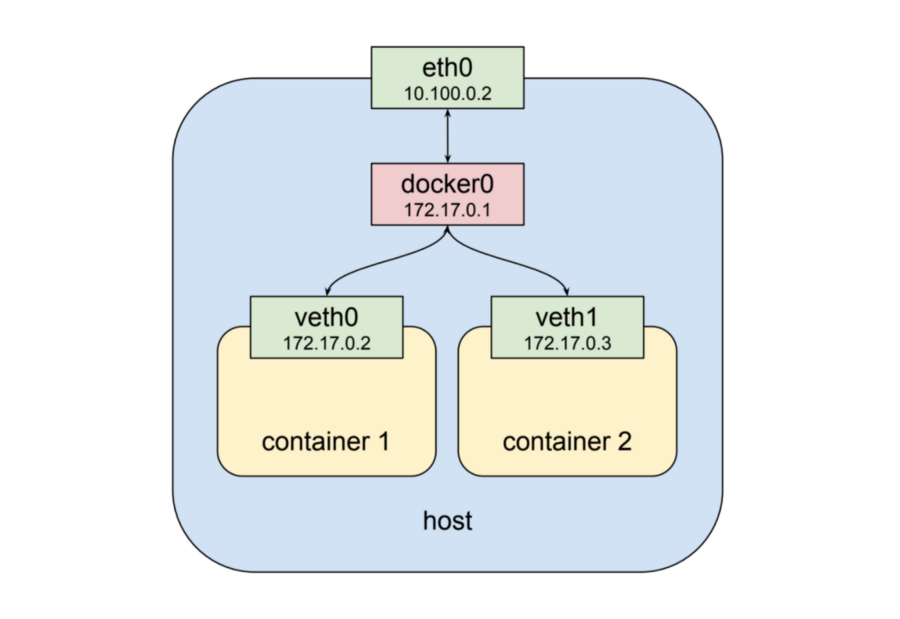
Docker is using CNI (container network interface). There are 2 types of networking techniques

1. single host which uses underlay
2. multi host which uses overlay

Docker has its own network interface, docker uses Tcp connection

When we install docker, by default 3 networks will be created

1. None---we choose this when we don’t need network or go for custom networking
2. Host---the container have same network that of host, it exposes container directly to public network. This mode is fast
3. Bridge ---it is default networking mode where containers port is mapped/exposed to host port



Whenever docker is created a bridge network is created there will be a pool of ip address. Whenever you launch a container a virtual ethernet is created, where bridge driver assigns an ip to container. Physical Ethernet is attached to bridge and bridge network is attached to virtual Ethernet. Both docker network and virtual Ethernet are on the same network

1. docker network ls:-

To list all docker networks

#docker network ls

1. docker network inspect:-

To see more about docker networking

#docker network inspect NetworkName

1. create your own docker network:-

To create new network in docker before launching containers. This network is different from bridge network

#docker network create --driver drivername name

#docker network create –driver bridge New\_Net

\*Where driver name = host/none/bridge

* Running containers in new network

#docker run -dt –name c1 --network=New\_Net ubuntu:14 sleep 1d

#docker run -dt –name c2 --network=New\_Net ubuntu:14 sleep 1d

new exec into containers and ping

# docker exec -it c1 /bin/bash

#ping c2

Similarly do in other container

* One container attached to a network can’t communicate with other container with another network.

DOCKER BEST PRACTICES:-

1. One application in one container
2. Run the application in foreground
3. Keep the data out of container
4. Avoid manual configuration

REF : https://github.com/wsargent/docker-cheat-sheet