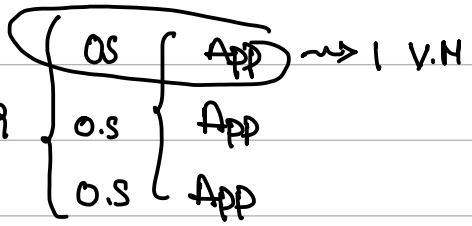
* Docker *

→ Containers are advancement to VM.

VM structure & workflow.

to use the resources completely ⇝ VM ⇝ hypervisor



→ Containers – do not have full OS.
– light weight

→ Containers → on top of physical machines.
→ on top of VM

⇝ shares resources from the Host operating systems.

" Base Image → System dependencies + Application libraries "

* Life cycle of Docker :

Docker file —build→ image —run→ Container

in docker platform ⇝ Commands → Docker Engine.

* Alternate approach to Docker can be "Buildah"

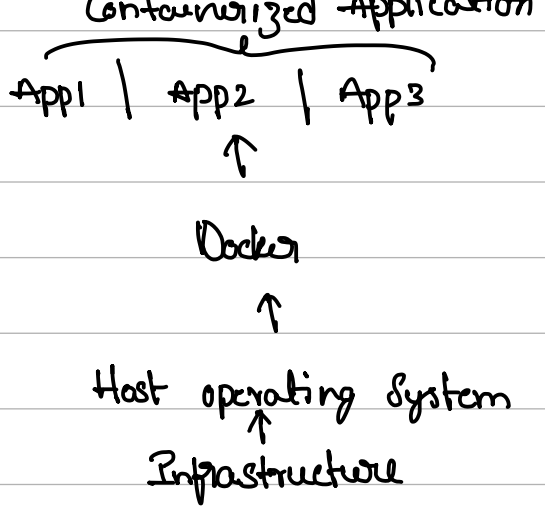
why different approach?

In Docker ⟶ Docker Engine ⟶ Containers.

Single point of failure (SPOF)

- if Docker Engine goes down all the docker Containers will go down.

* Docker Architecture



* Files & folders of the Base image:

1. /bin : Contains binary executable files. ex: ls, cp, etc.

2. /sbin : system binary executable files, ex: init, etc.

3. /etc : configuration files for various system services.

4. /lib : Contains library files that are used by binary executables

5. /usr : contains user related files.

6. /var : contains variable data, such as logs, spool files, etc.

7. /root : home directory of the root user.

* files & folders that containers use from host O.S :

1. Network stack : to provide network connectivity to the container.

2. System calls : which is how the containers accesses the host resources such as CPU, memory and I/O.

3. Namespaces : Docker uses linux namespaces to create isolated environment.

4. Control groups (C groups) : it uses c groups to limit and control the amount of resources, such as CPU, memory and I/O, that a containers can access.

* Install Docker :

- Sudo apt install docker.io

to check the docker is running or not

- sudo systemctl status docker

* Docker daemon runs only with the root user.

→ adding user to a group.

sudo usermod -aG docker ubuntu

Example: docker run hello-world

* Structure of Docker file:

```
From ubuntu : latest

# Setup the working directory
WORKDIR /app

# Copy files from the host system to image file system
COPY . /app

# Install the necessary packages.
Run apt-get update && apt-get install -y python3 python3-pip

# Setup environment variables.
ENV Name world.

# Run a command to start the application
CMD ["python 3", "app.py"]
```

* To build the docker image:

docker build -t username/name : lecbest [

tag

* Docker Containerization for Django Application:

- Difference btw 'Entrypoint' & 'CMD'
  - Both can be the starting point
  - CMD can change.
  - Entrypoint can't change.

→ once we have the docker file same as the one in the example in the Django application.

1. docker build .
2. docker images
   - to check the images that are build & copy the id.
3. docker run -it "id that copied"

   ↳ interactive mode.

~> the application will not run because the port was not exposed. In order to expose the port we should forward the port.

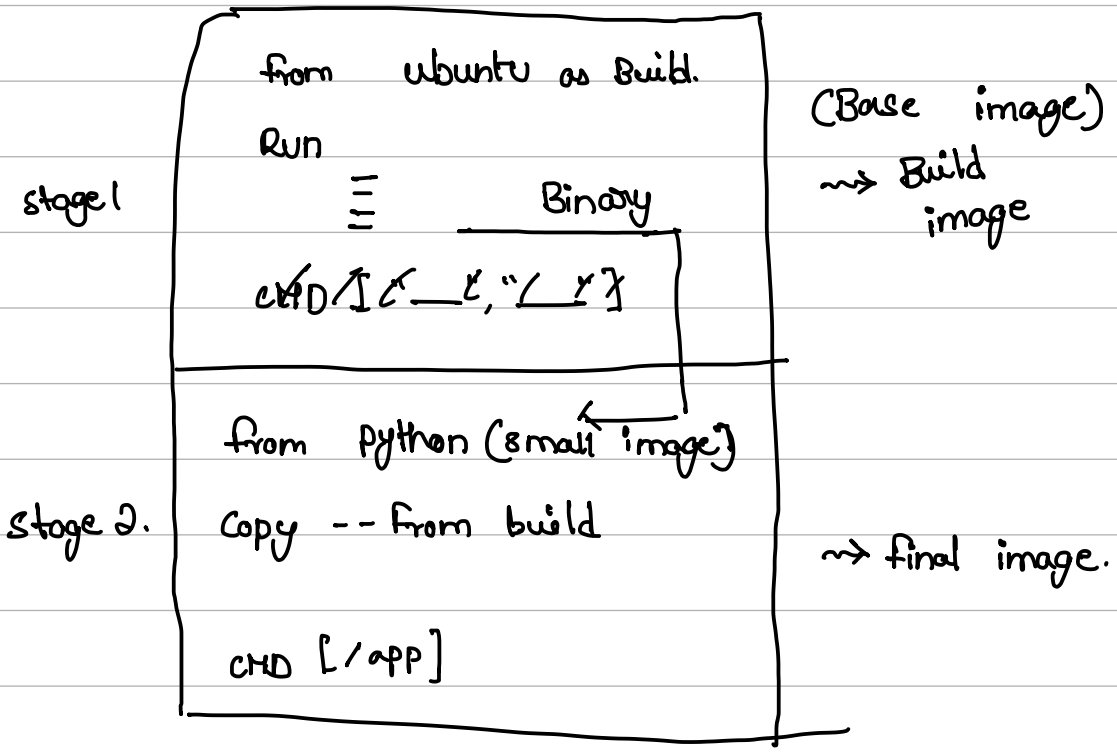
4. docker run -p 8000:8000 -it "id of the image".

* Multi Stage Builds:

why? - for example, to run a calculator application we may require only python run-time not all the other 'apt' and ubuntu packages to build the application. So for that we use multi-stage builds.

Structure:

multi-stage Build.



* same way we can do multiple & 'n' number of stages.

* Distroless image → very minimalistic.
  - light weight image that will only have the run time.
  - highest security., very safe bcz they will not have basic commonds.

Example: Docker file for Golang Caluculator.

→ Dockerfile without multistage.

```
from ubuntu as Build
Run apt-get update && apt-get install -y golang-go

ENV GO111 Module = off

Copy . .
Run CGO_Enable=0 go build -o /app .

Entrypoint ["/app"]
```

* To build the above docker file.

```
docker build -t simplecalculator
```

& we can choose our own tag.

* for multistage dockerfile, continution for base image.

```
from scratch

# copy the compiled binary from the build stage.
COPY --from = build /app /app

# set the entrypoint for the container to run binary
Entrypoint ["/app"]
```

→ In this simple caluculator app itself the image size has been reduced from 800mb to 1mb i.e it was reduced by 800%.

→ we can search for distroless images directly and can find the github repo for various widely used one.

* Docker Bind mounts & Volumes:

→ Containers are ephemeral in nature. mean they are short lived.
→ If the container goes down files it has will be deleted.
→ Another problem is we can directly read the files at specific directory.

Solution → 1. Bind mounts
→ 2. Volume.

* Bind mounts: we attach the specific directory of the container to the specific directory of the host file.
- so that even the container goes down files & data will be saved on the host file.

* Volume: - logical partition on the host
- we can create & destroy
- we can attach in multiple ways to different containers
- we can control through docker cli

→ In Binding we can attach only to host, but in volumes we can attach to host, EC2, S3, etc. in simple terms we can attach external source.

→ Easy to share and high performance.

→ volumes are more preferable to use.

docker -v <——> } both are same just
docker --mount } syntax is different.

Example:

- docker volume create "name"
- docker volumes ls
- docker volume inspect "name" ~> to get the details.
- docker volume rm "name" ~> to delete.

* To mount:

docker run -d --mount source = "name_of_volume", target = /app imagename

→ to delete volume we should stop the container where the mount was attached.

* Docker Networking:

↓

allows to communicate.

Case 1: container 1 talks to container 2

Case 2: Container 1 needs isolation from container 2.

①

→ Bridge networking:

- default by docker.
- used to communicate b/w container and host via bridge.
- it creates 'veth' ~ virtual ethernet.

② Host networking:

→ when both are in same network same ip can make it possible.

→ But this was less secure.

③ Overlay Networking: { Complicated }

- when there multiple hosts it creates a network between them.

Case 2

→ when using the above networks the host have 'eth 0' port and container will have 'veth' through docker.

→ But when there is need isolation it is not possible as they will use the same 'veth' network.

→ To solve this we have "custom Bridge network."

Finally,

Case 1 → Bridge network

Case 2 → Custom Bridge network.

* Command to create a container and assign different networks.

docker run -d --name host-demo --network = host nginx:latest

name of your docker container

you can select the network on your own either it can be host or any custom network of your choice.

* Command to create custom network.

docker network create secure-network.

* Docker Compose:

→ Manage multi container applications.

→ Using this multiple image can be runned & managed at a time.

→ Simple commands like:

```
docker compose up
docker compose down.
```

* Docker multi Arch / platforms builds.

→ To create the & build the docker image for multiple architure & platforms like amd64, arm64, etc.

~> for it in the docker desktop we have a built in option called " buildx ".

Syntax :

```
docker buildx build --platforms "platform name1", "platformname2"
```

* In case we did not have the docker desktop and buildx we can install it.

```
Sudo apt install docker-buildx
```

→ to add different architechure to it

```
docker buildx create --name multiarch --platform linux/amd64,
linux/arm64 --driver docker container --bootstrap --use.
```