

For

- Beginners
- DevOps
- Security folks
- & everyone

who wants to dive into  
dockers

Learn it in a  
visual way with  
Groot



# Docker Containers

## Application Isolation

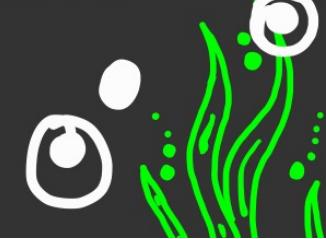
# Docker Containers

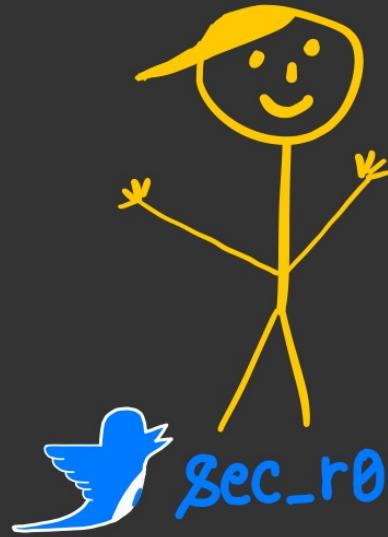


What do we have today ?

- ① What makes docker different from VMs ? ..... P4
- ② Internals of Dockers
  - ↳ Control groups
  - ↳ namespaces ..... P5
- ③ Basic overview of
  - Installation
  - Running containers
  - Docker Images
  - Dockerfile..... P7
- ④ Dockerfile ..... P8

- ⑤ Detour on UFS ..... P9
- ⑥ Docker Volumes ..... P10
- ⑦ Docker Networks
  - Types
  - Publishing ports ..... P11
- ⑧ Dockerfile cheatsheet ..... P15
- ⑨ Docker command cheatsheet ..... P17
- ⑩ Staged builds and when to use them ..... P20
- ⑪ Security Best Practices
  - Dockerfile
  - Containers ..... P22
- ⑫ docker-compose ..... P24
- ⑬ docker swarm ..... P27
- ⑭ docker-machine ..... P28





Let's talk about dockers today

I am Rohit, I use dockers almost daily.  
They are fun to learn and use.

I want to learn it.  
I am groot !!

But I don't know even  
'D' for Docker. Groot, Groot



Let me start with the basics  
and gradually discuss it in detail.

- \* Docker is a magical concept which can package application and its dependencies into a single unit Known as containers
- \* This can help scale your application , makes it OS independent and Secure
  - ↳ through Orchestration
    - ① docker-Compose
    - ② Kubernetes
  - ↳ By packaging it
    - ↳ By isolating app logic from rest of operating system



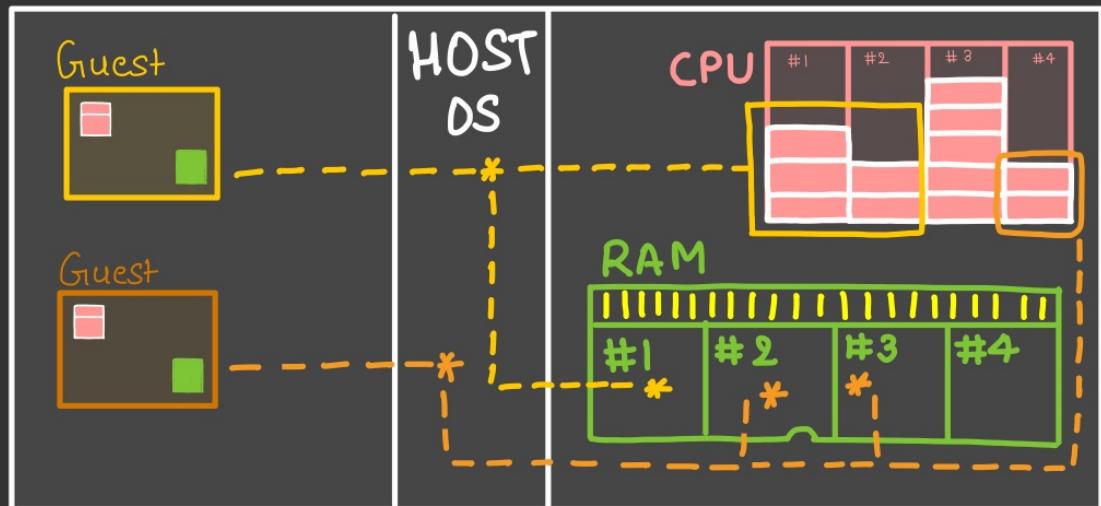
That sounds similar to  
Virtual machines.....

I am Groot !!



\* Yes it Similar to Virtual machines  
But it works differently than VMs

- \* VMs are more like hardware isolation
- \* They donot isolate hardware but creates simulation similar to that.



- \* It's the responsibility of HOST OS to simulate Hardware isolation



\* Docker Containers are Software virtualization which use Linux's namespace and control groups to create virtually isolated environment for apps.



Note ....

Linux's namespace and Control groups concepts



Does this means dockers containers donot work on Windows ?

But I have seen docker running on windows too !!

\* Yes, docker containers can not directly run on windows as they use Linux techniques.  
So behind the scene, windows create Linux VM and runs container in that VM

- \* Lets see those namespaces in detail.

## Namespaces

\* NS partitions kernel resources virtually

\* for 7 different resources you have 7 different NS in Linux

1. mount: virtual File System partition
2. Process: Partitions process tree
3. IPC: Communication between Processes
4. network: make network visible
5. user: virtual root inside NS
6. UTS: virtual DNS and hostname
7. Cgroup: allocates hardware resources to NS.

A control group is a namespace too !!



Yes.  
That namespace limits hardware resources

## Control Groups :- Cgroups

\* Introduced by Google and implemented in Linux Kernel.

\* A framework to put limits on processes for hardware resources

\* Resources like :- CPU, memory, network and I/O

### Open your terminal

```
$ sudo mkdir /sys/fs/cgroup/memory/foo  
$ echo 40000000 | sudo tee /sys/fs/cgroup/memory/foo/memory.limit_in_bytes  
$ echo 1337 > /sys/fs/cgroup/memory/foo/cgroup.procs  
$ ps -o cgroup 1337
```

↳ Create a memory cgroup named foo  
↳ limit foo cgroup memory to 40 MB  
↳ Add PID 1337 to memory cgroup named foo  
↳ List Cgroup settings for 1337

\* If process 1337 takes more Space than 40 MB, the Kernel will kill it 😊



Open your terminal

\$ sudo lsns  
↳ List all the namespaces present in your system

\$ sudo nsenter -t 1337 --mount --pid  
↳ move current user inside namespace with id 1337

Can I create custom namespace?



Yes,  
Exactly. That's what docker Command does.....

By default each docker that you create, forks all the namespaces. All NS apart from cgroup NS, allocates virtual resources and cgroup virtually allot hardware resource.

So, docker containers  
are isolated environment ,  
isolation created using Linux  
Kernel namespaces



That's why they are not  
virtual machines.

Just a light weight virtual  
isolation.

\* Virtual machines allocate hardware  
resources.

\* whereas containers shares the same  
hardware resources and isolation is  
done by kernel on software level.

\* Docker containers do this magic  
automatically



sec\_r0

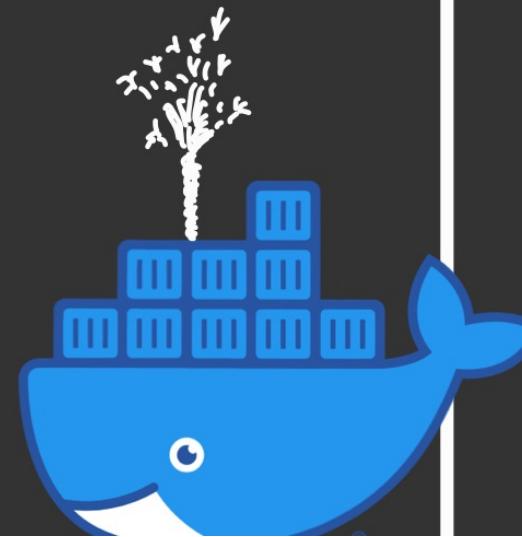
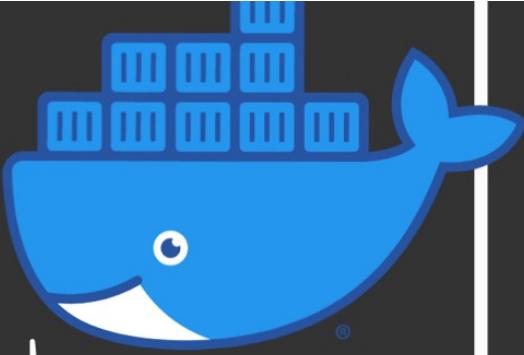
And my magic is  
well battle tested.

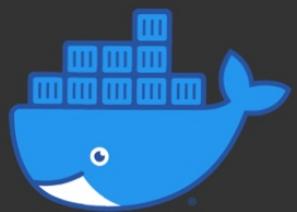
And is used in  
production grade  
systems .....



I want to learn your magic.  
But do I need to create  
cgroups and NS before talking  
to you Mr. Docker ??

You Just need  
to run a couple  
of commands  
to get everything  
running





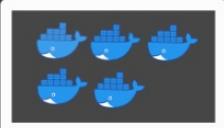
## Installing Docker

- \* Installation of docker engine is specific to OS

- \* let me skip that step here.

- \* Once you install docker engine you can run docker containers

## Docker Hub



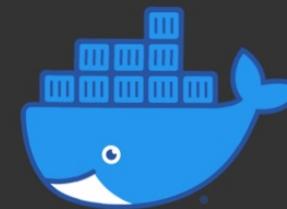
- \* managed by Docker Inc

- \* public repository server

- \* even you can push your own images

- \* Create account before pushing & login in command line

**\$ docker login**



## Testing Installation

- \* Run hello-world container

**\$ docker run hello-world**

↓      ↓  
docker command   telling docker engine to  
create container from this image

# Docker Images

- \* Pre-existing set of instruction telling docker engine what to do & how to create container.

- \* These images are stored in local system or in online repositories

- \* If you run image; docker will search locally if not found then it will search in docker-hub OR DTR [docker trusted repository]

## Dockerfile

### Dockerfile

- \* You create docker image from Dockerfile

- \* Dockerfile contains set of instructions about how to build docker image.

**\$ docker build -t <name> .**

build the docker image with name  
from filename Dockerfile in current directory  
\*\* Tag can also be included in Image name

- \* Once Image is built, check the list of local images

**\$ docker images**

- \* For pushing built image to hub name should be like  
<account\_name>/<image\_name>

**\$ docker build -t r0hi7/foo .**

**\$ docker push r0hi7/foo**



Can you tell me how to Create Dockerfile

That's simple again.....



Dockerfile contains set of instructions one after the other in each line & having specific meaning

Each line is of the form

INSTRUCTION argument

Case insensitive

Run command  
mkdir after using  
python base image

change  
work directory to  
/foo and run subsequent  
commands under /foo  
directory

build docker image from existing  
docker image name python

Copy all python  
files from host current  
directory to inside docker

For running command  
inside docker

Tell  
docker that Container listens on  
specific port. This instruction donot  
publish port to host system

Entrypoint Command  
tells what the Container  
will run. Here it will run  
python command.

## Dockerfile example

```
FROM python
RUN mkdir /foo
COPY *.py /foo/
WORKDIR /foo
RUN apt update
EXPOSE 80
ENTRYPOINT ["python"]
CMD ["server.py"]
```

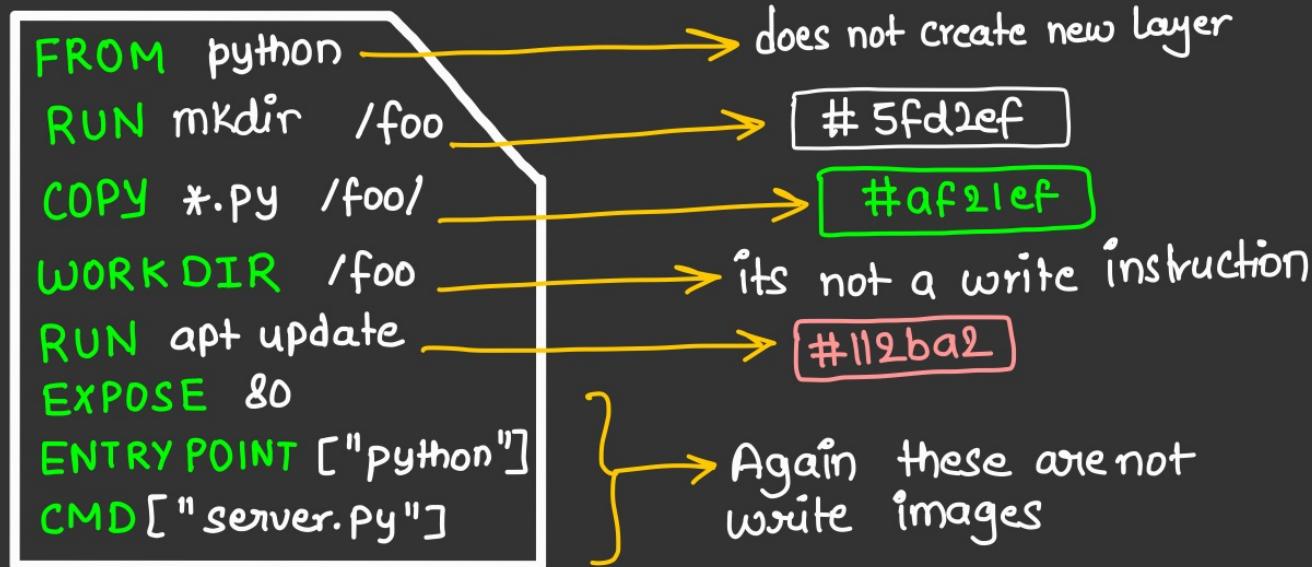
Cmd provides arguments to  
Command specified in Entrypoint Instruction  
These arguments can be override while  
running image in docker run.

# UFS

## Union File System

- \* union file system is layer based approach where each write instruction creates copy of entire file system
- \* Making each layer Read Only and Reusable
- \* Each layer has unique "Hash" as Layer identifier

## Docker file

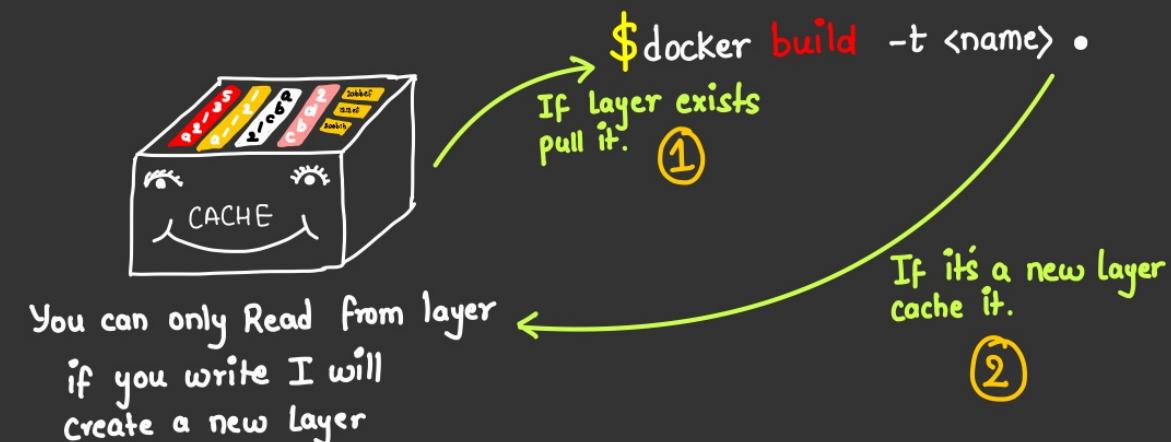
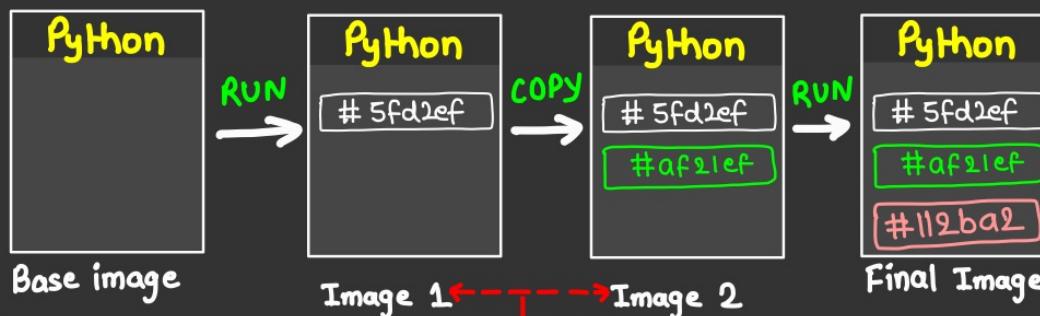


- \* Various instructions in docker file creates a new layer

## docker image



- \* Each layer creation creates a docker image too, called as intermediate docker image





Important point to note here is that, on running docker image and create docker instance, any data that you generate/add to docker instance, that data is lost as docker instance exits.

Docker volumes.....

So how that data can be preserved.



## Volumes

\* Directories managed by docker

\$ docker volume create foo  
will create volume under  
/var/lib/docker/volumes/foo  
\* In Linux Systems

\$ docker volume ls  
for listing volumes present

\$ docker run -v /foo/bar:/app nginx  
If this is exact path on host then  
directory will be mounted not  
volume

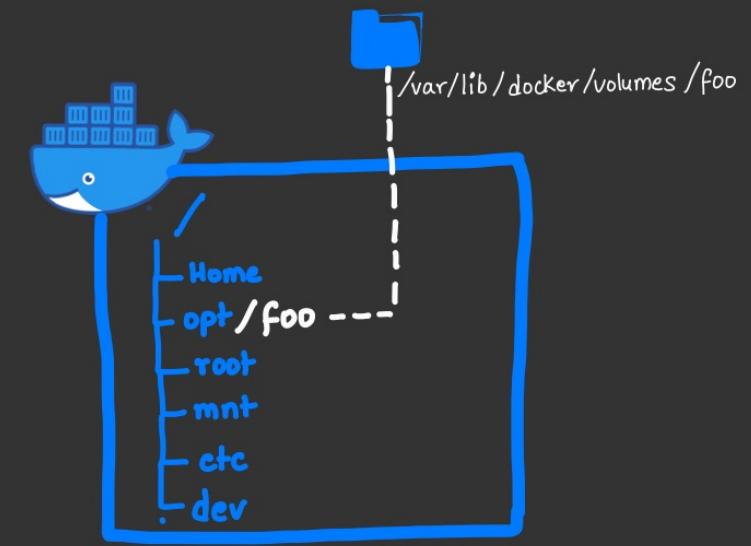
## Attaching Volumes

\$ docker run -v foo:/app nginx  
Volume that we created  
location where we want to mount Volume inside docker

OR

\$ docker run --name nginx-docker  
--mount source=foo,target=/app  
nginx:latest

--mount is more verbose



## Volume Lifecycle

Create

\$ docker volume create foo

Attach

\$ docker run --name nginx-docker  
--mount source=foo,target=/app  
nginx:latest

Docker Instance Removed

\$ docker rm nginx-docker

Remove

\$ docker volume rm foo

# Docker Networks



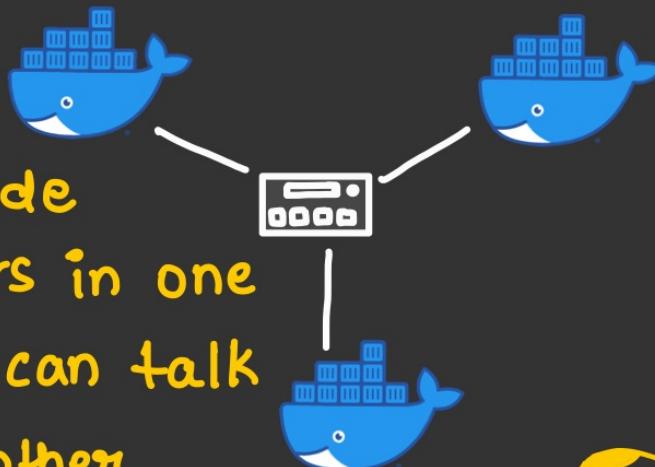
If our applications inside two different docker instances wants to talk through each other ??



Yes that's when docker networks comes handy

\* Couple of points to remember

①

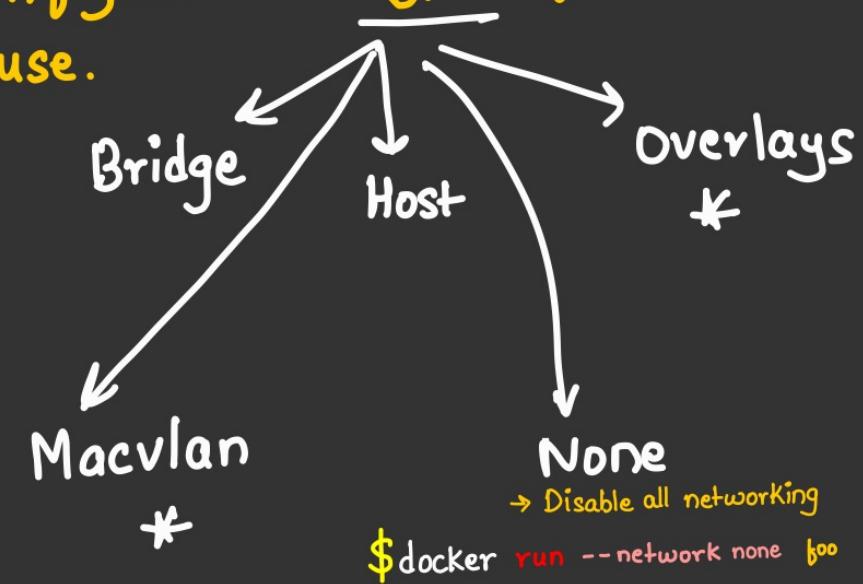


apps inside two dockers in one network can talk to each other



Let's see what network drivers mean

② Docker networking system is pluggable, you can configure the type of network drivers to use.



\* Lets mark them out of scope as they are advanced network drivers useful in swarms

# Network Drivers



## Bridges

\* default network driver used when docker network is created.

\* only containers in same bridge can talk to each other



\* Containers on same bridge expose all ports to each other

```
$ docker network create foo
```



## Attaching with Bridges

```
$ docker run --network foo  
--name app my-app
```

```
$ docker run --network foo  
--name db mysql
```



\* Already running containers to network

```
$ docker network connect foo <running-container-name>
```

\* attaching containers to bridges creates local DNS entry with container name inside all containers participating in bridge....

meaning

From inside app container, hostname reference to db will resolve to local IP address of db and vice versa

## HOST

\* With this driver configuration container joins host network and is isolated from other containers

\* Since it uses host network, you cannot create a new host network.

\* You can only attach to host network while starting container

\* Since network is shared with host, two containers cannot expose services on same ports

```
$ docker run --network host foo
```

docker image name

There is also legacy of connecting two

dockers to default bridge network using --link while creating docker

```
$ docker run --link db:db  
--name app my-app
```

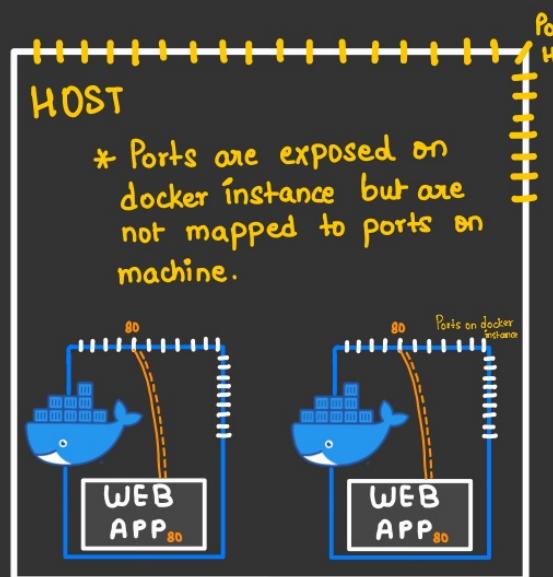


# Publishing Ports

Above methods will enable communication between two running containers but services running inside docker will not be accessible from outside docker.



Let visualize the current scenario, it looks something like this



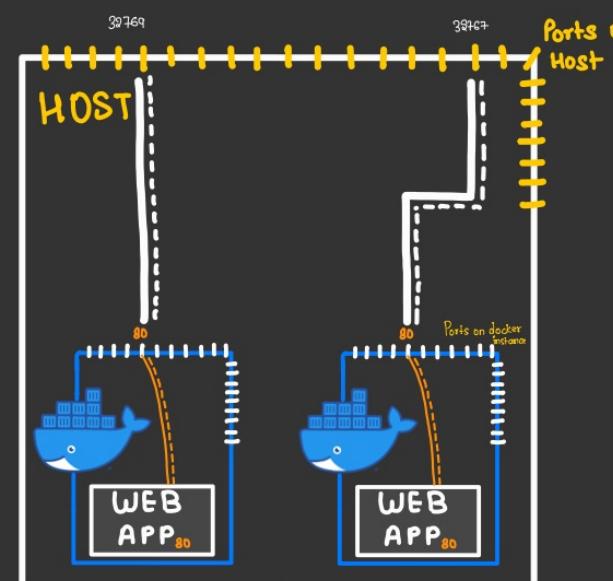
- \* So we need to publish those exposed ports
- \* Ports are published while starting a container



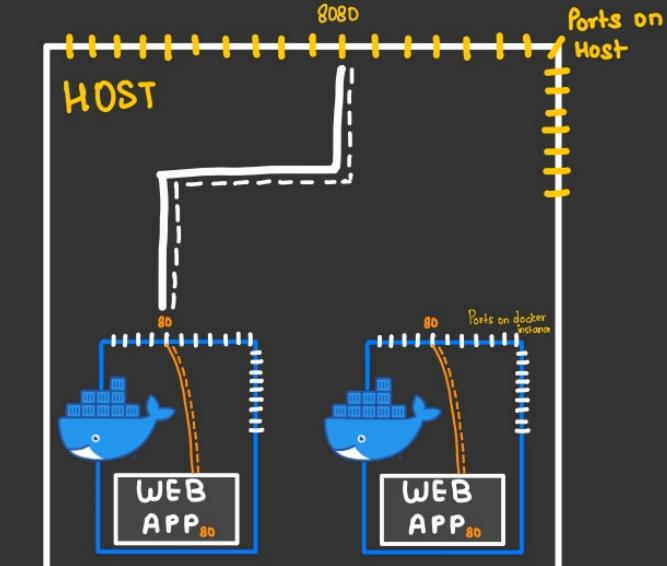
So how can that be done, GROOT....

```
$ docker run -P --name app-name web-app
```

```
$ docker run -p 8080:80 --name app-name web-app
```



\* This approach maps all exposed ports to some random ports



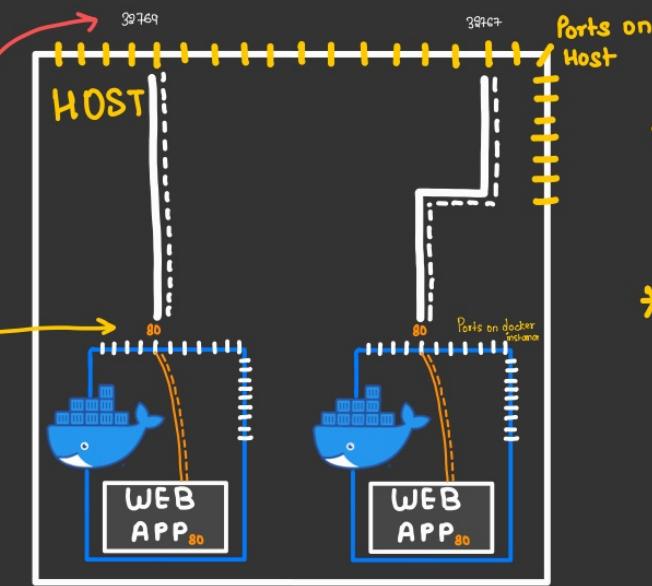
\* Only maps the ports which are mentioned in command

```

FROM python
RUN mkdir /foo
COPY *.py /foo/
WORKDIR /foo
RUN apt update
EXPOSE 80
ENTRYPOINT ["python"]
CMD ["server.py"]

```

\$ docker run -P --name app-name web-app



\* This approach maps all exposed ports to some random ports

- \* EXPOSE instruction in Dockerfile opens the port at docker Level
- \* --publish or -p or -P maps those exposed ports to ports in host machine

I will remember  
this ..... GROOT



\$ docker run

-p 8080:80

-p 7070:8080

--name app-name web-app

\* For multiple ports

--name app-name  
web-app

\$ docker run -p 127.0.0.1:8080:80  
 ↓  
 Host Interface  
 publish on local host only

Host port  
 docker port

\$ docker run -p 0.0.0.0:8080:80  
 ↓  
 Publish on all interfaces on host

All above commands have to run in single line in shell,  
 Just to manage real estate properly I have done this way.

# Dockerfile Cheatsheet



**FROM** <image>:<tag>

optional if not specified latest image is used



**MAINTAINER** <name>

Add author name to Image



**RUN** <command>

Run Command inside docker inside shell



**LABEL** <key>=<value> [<key>=<value>...]

Add metadata to docker image



**EXPOSE** <PORT-number>



writing docker file can be  
tricky let me share a  
cheat sheet of all the  
Instructions

Groot Groot !!



Key Val

**ENV** <key>=<value>

Sets environment variable

**+ ADD** <src> [<src>...] <dest>

copy files/directories from source to destination



**COPY** <src> <dest>

similar to ADD, just that ADD has more functionalities like  
uncompressing while adding compressed source , ADDing from URLs etc



**ENTRYPOINT** ["cmd", "c" ...]

Entrypoint Command tells what the Container will run.

**CMD** ["cmd", "cmd" ...]

Provides argument to ENTRYPOINT, can be  
overwritten while running docker

# Dockerfile CheatSheet . . . contd

- 📁 **VOLUME** <path> [<path>...]  
This does not mounts external volumes,  
rather it just creates a mount point for holding volumes
- 👤 **USER** <username|uid>  
Sets user for subsequent dockerfile Instructions, if not specified  
ROOT user is used. This is not user in HOST, this is user inside docker.
- WORKDIR <path>  
Sets working directory for Subsequent docker Inst'
- 👤 **ARG** <name>[=default]  
defines a variable may be with default value that can be passed at build time.
- 👤 **ONBUILD** <Inst'n>  
Adds a trigger instruction to image which is called when current image is used as base.
- ⌚ **HEALTH CHECK** <cmd>  
check health of container by running command inside container

<value>: value in angle brackets needs to be replaced

[ ] : Denotes optional



<something> ⇒ any thing in <> needs to be replaced with actual values.

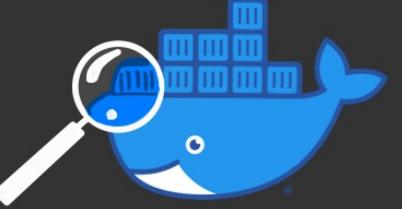
- ⇒ any thing with this color is optional



## Container Lifecycle

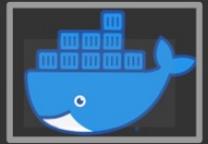
```
$ docker build -t <name> .  
  
$ docker run --name <container-name>  
-P -p<host-port>:<Container-port>  
-v<volume>:<directory-in-Container>  
<image-name>  
  
$ docker stop <container-name>  
  
$ docker start <container-name>  
  
$ docker restart <container-name>  
= Stop + Start  
  
$ docker rm [-f] <container-name>  
= destroy the container
```

### Inspecting Containers



- \$ docker ps  
⇒ List running Containers
- \$ docker ps -a  
⇒ list all containers
- \$ docker logs <container-name>  
⇒ Show stdout /stderr
- \$ docker logs -f <container-name>  
⇒ Follow the stdout and stderr
- \$ docker top <container-name>  
⇒ list processes running inside containers
- \$ docker inspect <container-name>  
⇒ show low-level info in json fmt

### Image Management



- \$ docker images  
⇒ List all local images
- \$ docker history <image-name>  
⇒ Show image history
- \$ docker tag <image-name> <tag>  
⇒ tag an image
- \$ docker import <tar-location> <tag>  
⇒ Create image from previously exported snapshot
- \$ docker rmi <image-name>  
⇒ destroy image from local system

# Contd.....



\$ docker **attach** <container-name>  
→ attach host terminal stdIn/stdOut/stdErr to container

\$ docker **cp** <host-path>:<container-path>  
→ copy files from host to running container

\$ docker **cp** <container-path>:<host-path>  
→ copy files from running container to host

\$ docker **exec** <container-name> --<cmd> <args>  
→ execute a command inside container

\$ docker **export** <container-name>  
→ export container content to tar archive

\$ docker **wait** <container-name>  
→ wait till container terminates

\$ docker **commit** <container-name> <image>  
→ create docker image from container snapshot.

\* You can also create account on [hub.docker.com](https://hub.docker.com)  
and push your local images online

- ① \$ docker **login** \* login account on command line for pushing image
- ② \$ docker **build** -t <username>/<image-name>  
\* For pushing image to hub image name must be tagged with username
- ③ \$ docker **push** <username>/<image-name>
- ④ \$ docker **pull** <username>/<image-name>  
Pull that image anywhere
- ⑤ \$ docker **run** <username>/<image-name>  
Run that image



# Question Time



Before moving ahead, let me ask you one question Groot

For Languages which need compilation lets say Java or Golang would you prefer compiling the code in :

\* Host and then copy the compiled binary to docker

→ \* OR copy the source files in docker and compile them in docker



That's easy one to answer.

For Java you can compile anywhere and run anywhere.

And as far as I know, golang creates platform specific binary So if we have to compile on host, we need to cross compile



\* And You get this right.

\* For platform specific binaries, like for Golang, compiling in host and running inside docker will be tricky

\* But compiling inside docker need Compiler/libraries/dependencies inside docker

But that will increase the size of final docker image

Oh Yes



One thing to note here is that after compilation compiler/deps are not needed

And that's why we use

## Multi Staged builds

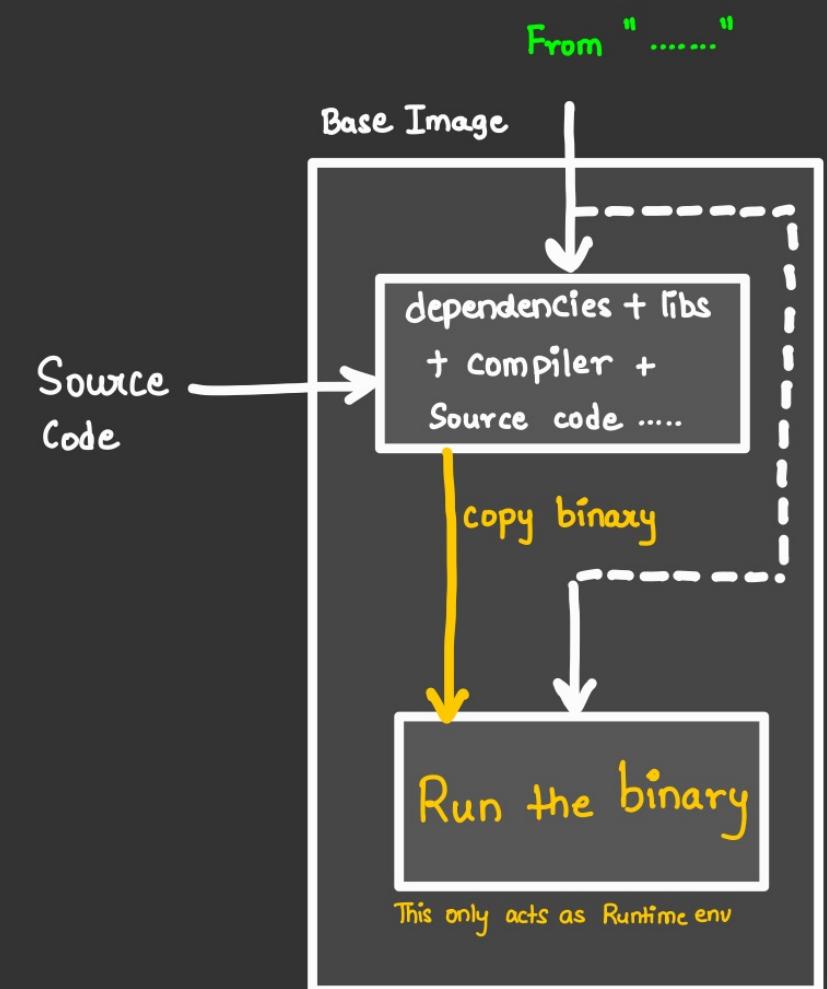
The idea is fairly simple

- \* Build the binary file inside one docker image , this image will have all dependencies & libraries necessary for compilation - Called build stage image
- \* Copy the output file of previous step and copy it in new docker image . So current image will only contain binary



Do I need to create two different docker files and perform copy manually ....?

You can, but docker make it a step easier, through staged build.





lets now create a sample staged build file for Golang binary as an example

Dockerfile

```
FROM golang:1.16 AS Builder
WORKDIR /opt/go-app/
RUN go get golang.org/foo
COPY app.go .
RUN CGO_ENABLED=0 \
    GOOS=linux \
    go build -a -o app .

FROM alpine:latest
WORKDIR /opt/
COPY --from=Builder /opt/go-app/app
CMD ["./app"]
```

name of this stage

Copy source file from host

Compiling it for any linux OS.

Now copy file from Builder stage

Docker image at this stage will be discarded automatically.

to current location

From this location of Builder Stage

# Docker Security Best Practices

- For Dockerfiles
- for running docker containers

Insecure Dockerfiles can cause security issues  
And since this zinc is from hands of security guy  
How can I miss talking about security concepts.



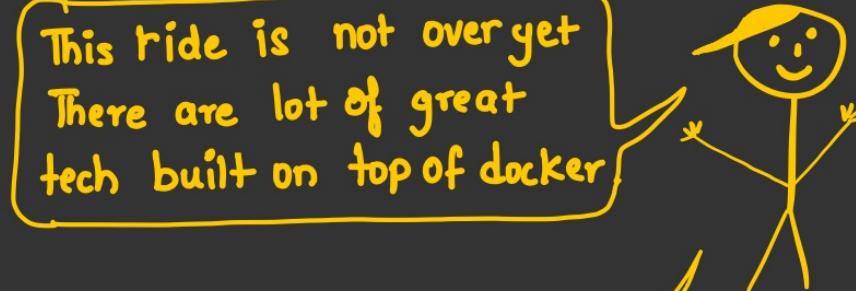
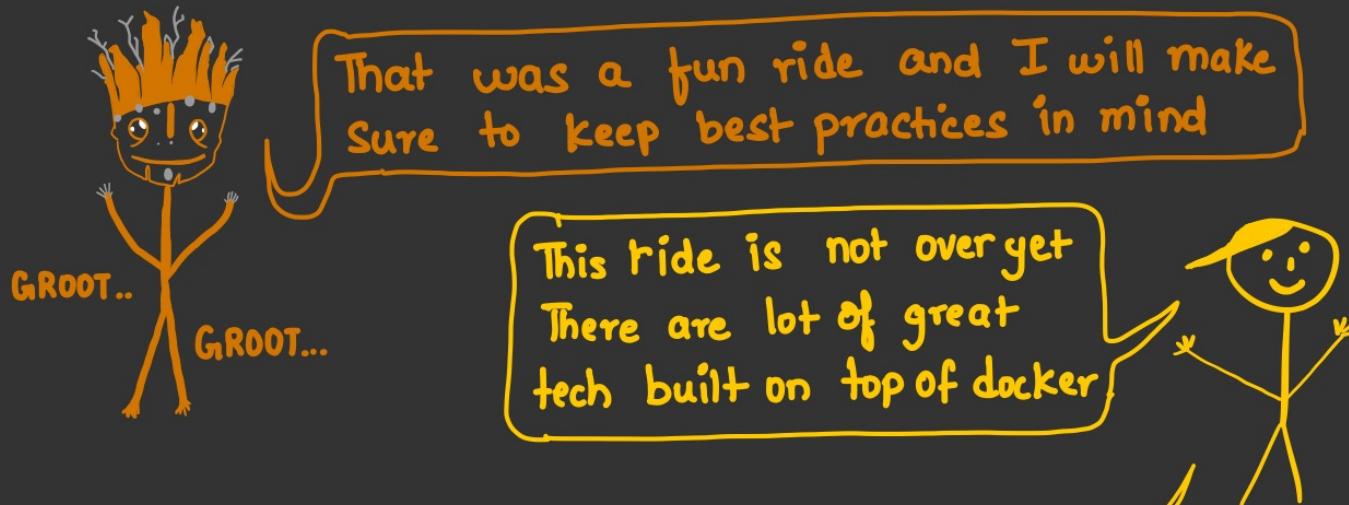
## Dockerfile Security Best Practices

- ✗ Avoid copying everything `COPY . .` \* This can copy eg .git folder (if any) and incase if app is compromised or git history can be exposed
- ✓ Use non root user and group \* By default application will run as root inside docker
- ✓ Use only trusted/official images in Production \* Non official images may have backdoor  
\* Create/manage your trusted repository
- ✓ Keep docker image minimal, dont add unnecessary libraries/dependencies
- ✗ Open the ports that your application needs
- ✗ Don't put any secrets inside dockerfile
- ✓ Sanity check :- Each instruction inside docker file creates a new layer  
So try to club instructions together

# Container Security Best Practices

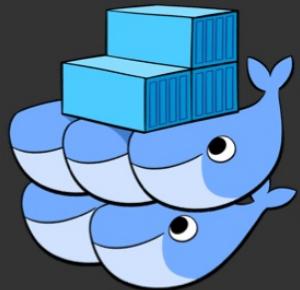
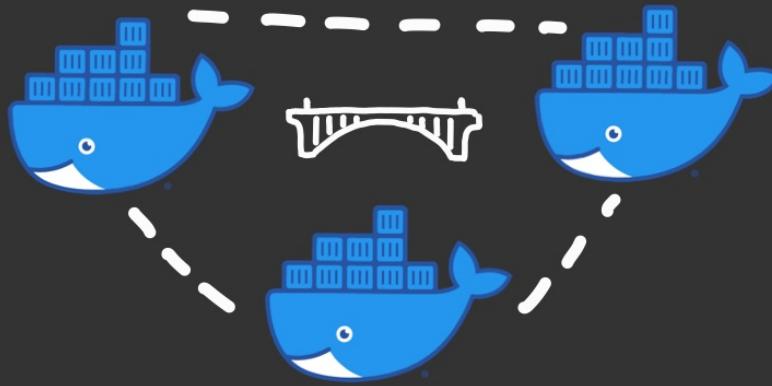


- Keep docker host and docker upto date.
- Docker engine talks to docker containers through docker Socket. Make sure never expose this socket to docker container with volume mounts
- You can run docker contained application as other user from command line as well
  - \$ docker run -u 4000 alpine
  - ⇒ useful in cases when you dont control docker image
- Drop the capabilities that docker donot need. Here we are talking of Linux capabilities  
Each docker runs with set of 'CAP', so give only those which are needed  
Best Approach - Remove all, give one by one
  - \$ docker run --cap-drop all --cap-add CHOWN alpine
- Never run docker with --privileged flag set.
- Running docker with --security-opt=no-new-privileges will prevent privilege Escalation
- Specify access modes with Volumes : RO,RW eg with --read-only flag
  -  sec\_r0

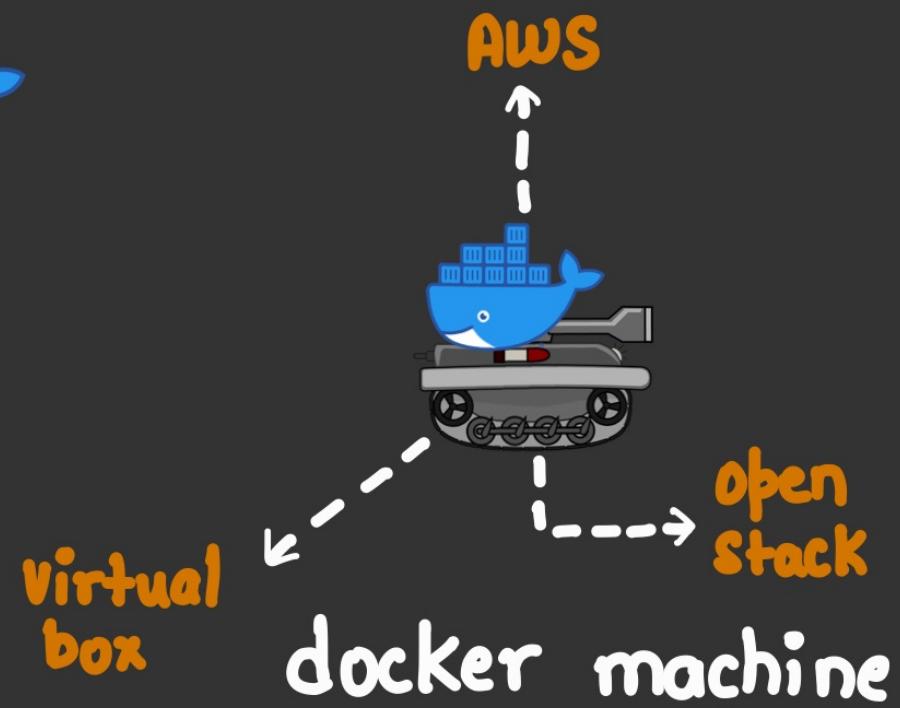


Let me introduce few of them here

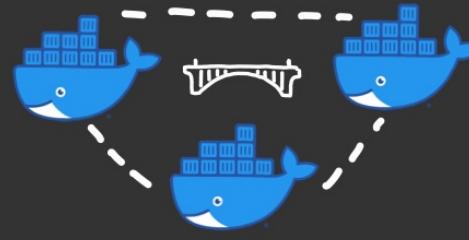
## docker Compose



docker Swarm



# docker Compose



Hey Groot



Consider a case where you always want to spin multiple containers in a single network and with volume every time you run it.

I can do it.....

- ① Create a network
- ② Create a volume
- ③ Run containers with bind to network & volume



Yes, you get it right.....

But what if I ask you clean every thing after Sometime ??

GROOT..

GROOT..

GROOT..

GROOT..



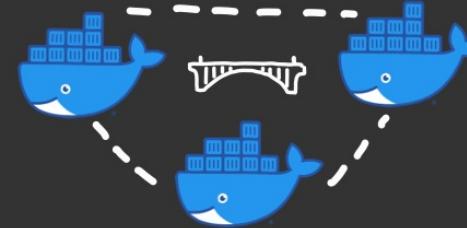
I would need to perform cleanup one by one from reverse of ③ → ② → ①

If you have say do it 10 times and each time doing it for 4-5 Containers ?? Then docker-compose comes to rescue....

# docker Compose



docker-compose is a utility from docker that can automate running multiple containers with network, volume or other constraints



YAML format

- ① You'll be defining the entire configuration inside a 'spec' file
- ② By telling docker-compose to perform operation based on spec

```
version of compose file format, optional
each Service represent a docker container.....  
this will be the name of running container  
In compose you can either build the Container from Dockerfile or use an image  
Volume creation within Compose

YAML format
version : "3.9"
Services :
  web:
    build : .
    ports :
      - "8080 : 8080"
    volumes :
      - .:/Code
      - logvol:/var/log
  redis:
    image: redis
  volumes:
    logvol: {}

$ docker-compose build
$ docker-compose up -d
$ docker-compose down
```

\* By default all containers defined under services will belong to one single network.

\* network name would be <folder\_name>-default

Folder where you have kept your docker-compose.yaml

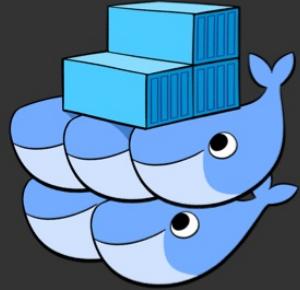
→ Configurations for how to run a docker container

\$ docker-compose build → build docker image if any

\$ docker-compose up -d → daemon mode ... optional  
Bringing containers to life

\$ docker-compose down → Stop all containers ..... do not remove images

\* all commands above uses file docker-compose.yaml in current directory.



## docker Swarm

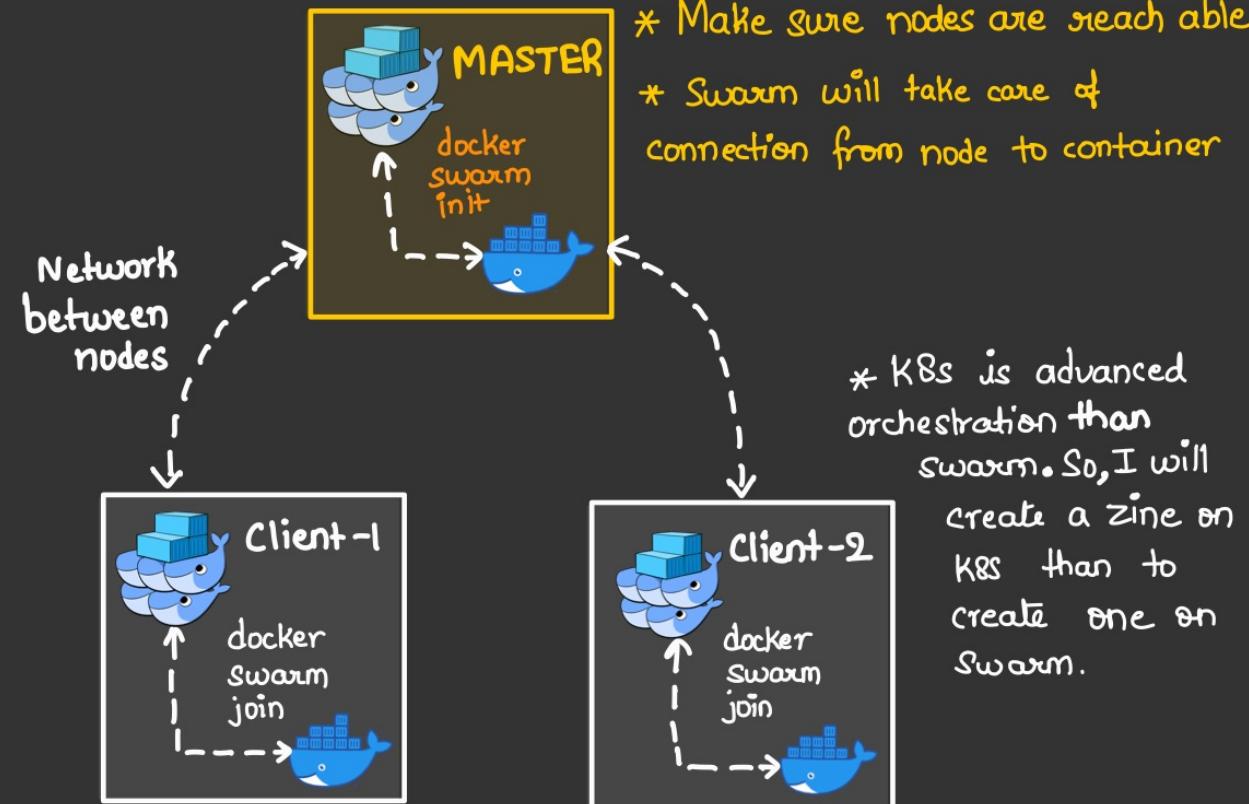
- \* docker containers are not fault tolerant  
if a container goes down, it will not come online automatically.
- \* compose can take care of network and volume, but it can not
  - create replicas
  - run on distributed networks
  - recreate a container in case it goes down
  - Rollback to previous stable release easily
  - handle other orchestration duties
- \* docker-swarm is multi node docker orchestration
  - ↖ Easy to Setup
  - ↘ Super easy to use



Docker Swarm is an orchestration built by docker Inc.

But why do we need orchestration for docker,  
we already have docker-compose?

Swarm is orchestration, that can run container in cluster of docker nodes



# docker machine



Before telling you about docker machine  
I want to discuss about how docker engine  
works.

Docker Engine



has two components



Client

\$ docker run alpine

↓ translates to REST request

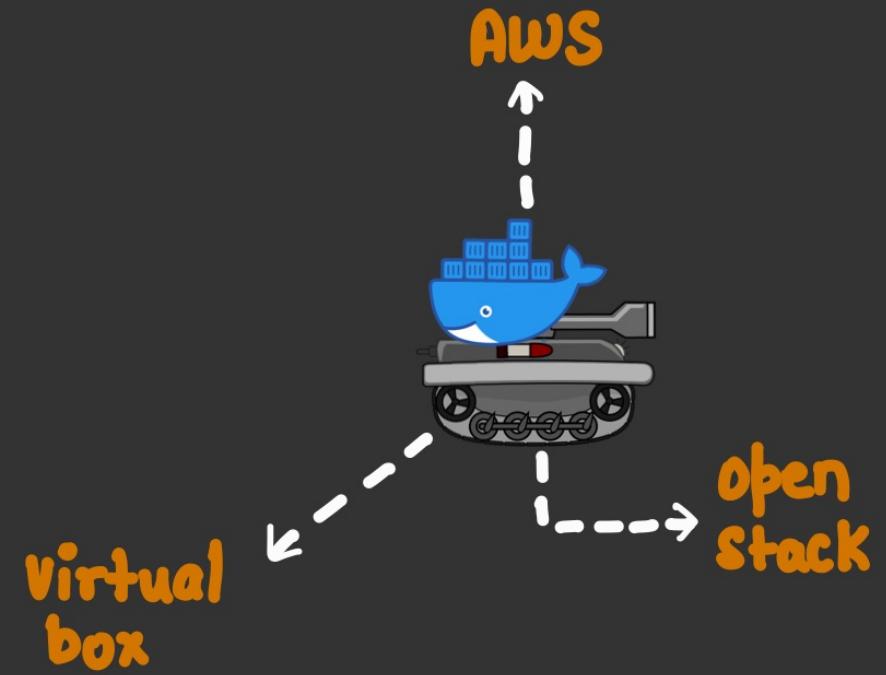


Server [REST]

\* Server can run on any  
machine, just that it  
should be reachable by client

In local host setup APIs are  
available at unix socket @ unix:///var/run/docker.sock

\* docker commands that we were running for those commands client and server  
were in same machine.



# docker machine



\* Now docker machines performs, the necessary setup to run docker Server on some other machine which are connected through network.

\* By default, it performs the setup on virtual box

Lets create a machine

\$ docker-machine create --driver virtualbox foo ]→ Single command will setup everything  
name of docker machine

Next Step is to pull the details of newly created machine

\$ docker-machine env foo

Inform docker to use new server

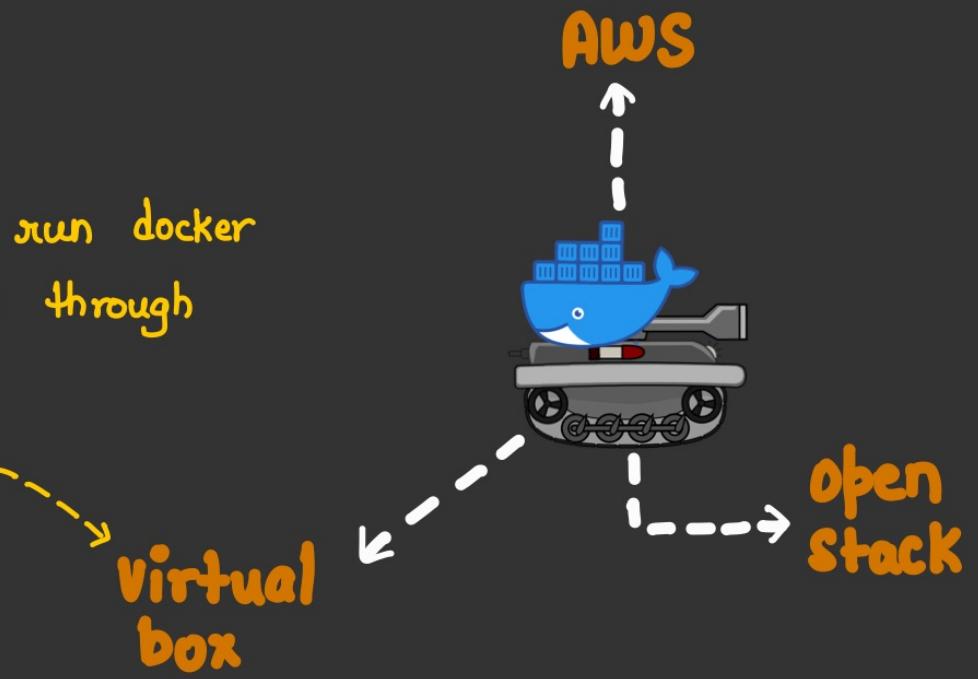
\$ eval "(\$ docker-machine env foo)"

\$ docker-machine ls

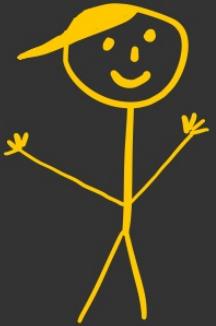
\* list docker machines

\$ docker-machine rm foo

\* Remove machine foo



# docker machine

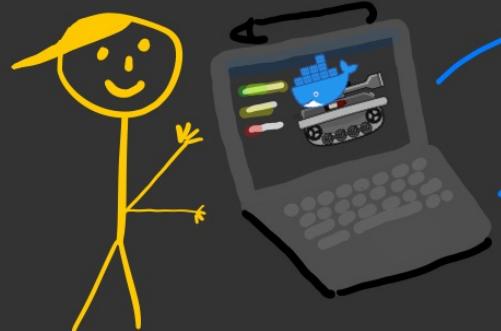
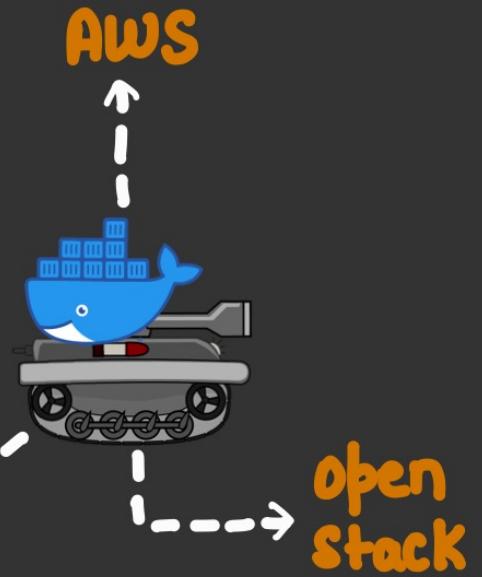


\* Once you connect with docker server with below command.

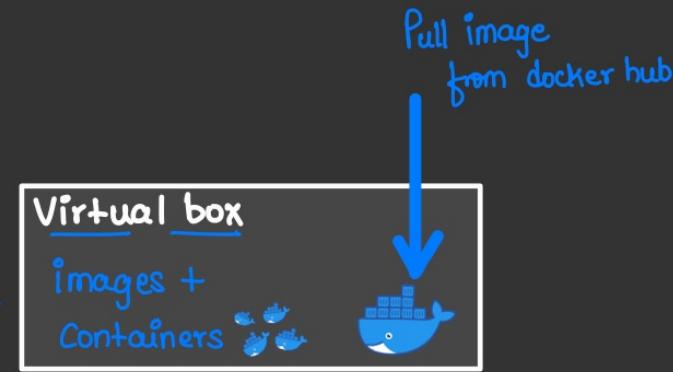
```
$ eval "$(docker-machine env foo)"
```

You will get connected to docker machine running on

Virtual  
box



GET ....



\* After using this setup you won't realise if docker is running on local or remote System

\* This makes your personal system clean.



I will use machines....



So Groot,  
that's it for today  
I hope you like it.....

I enjoyed it.....  
I will wait for K8s  
now.....



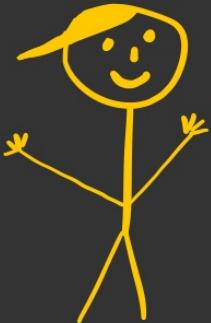
Thank You .....

Read more Zines @ SecurityZines.com

Join for more updates



A big thanks to awesome reviewers, without your reviews this would not have been possible



@ aurelievache

@ ianmiel

@ sanjogpanda

@ i\_shivamsoni

@ KathanP19

@ m4giKtrick



Sneha Anand

Tilak Lodha

Akash Jhadav