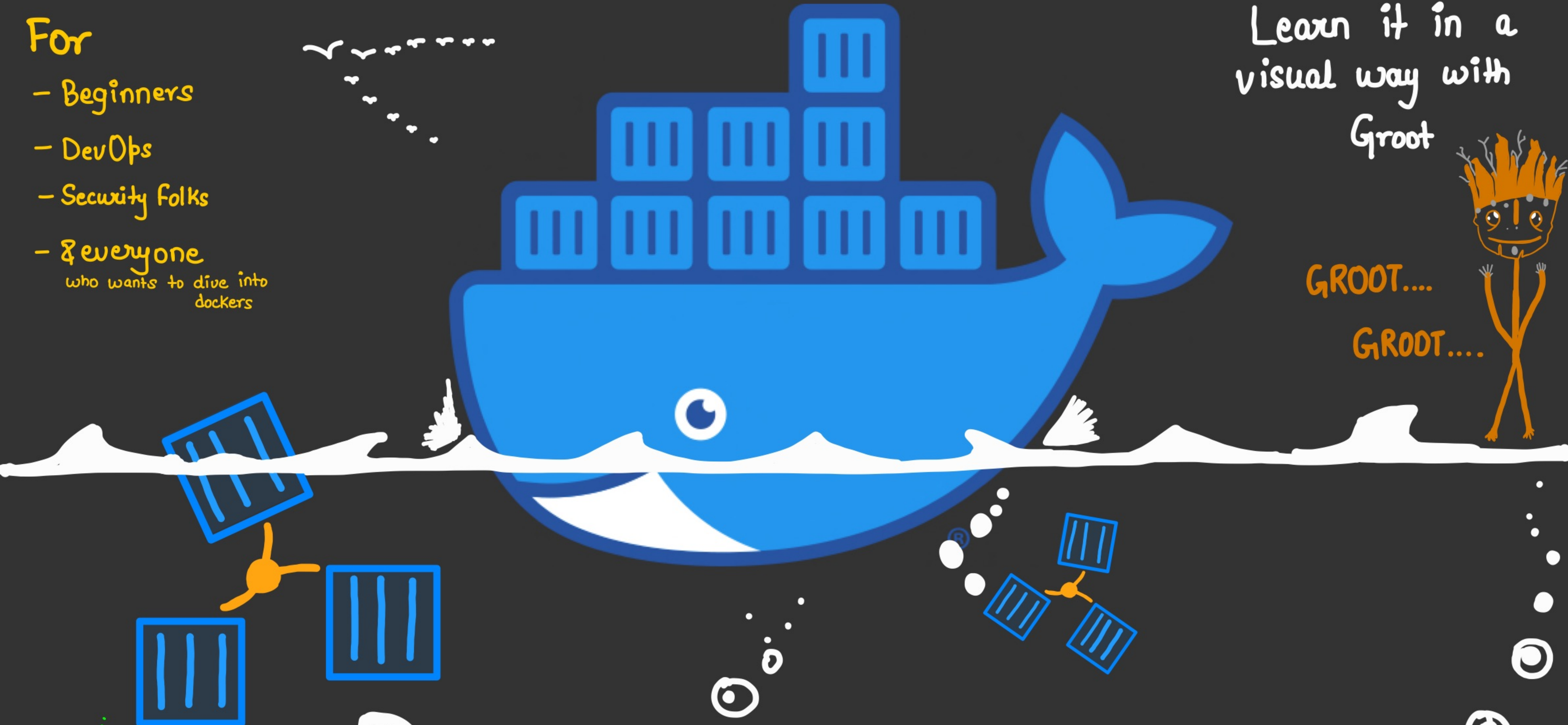
For

- Beginners
- DevOps
- Security folks
- & everyone
  who wants to dive into dockers

Learn it in a visual way with Groot



Docker Containers

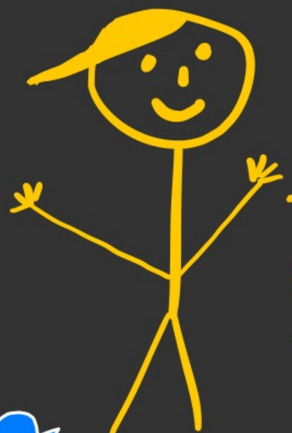
Application Isolation

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 containes
   - → 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

sec_r0



Let's talk about dockers today

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

sec_r0

I want to learn it.
I am groot !!

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

Let me start with very 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

  ↳ scale: through Orchestration
  ① docker-Compose
  ② kubernetes

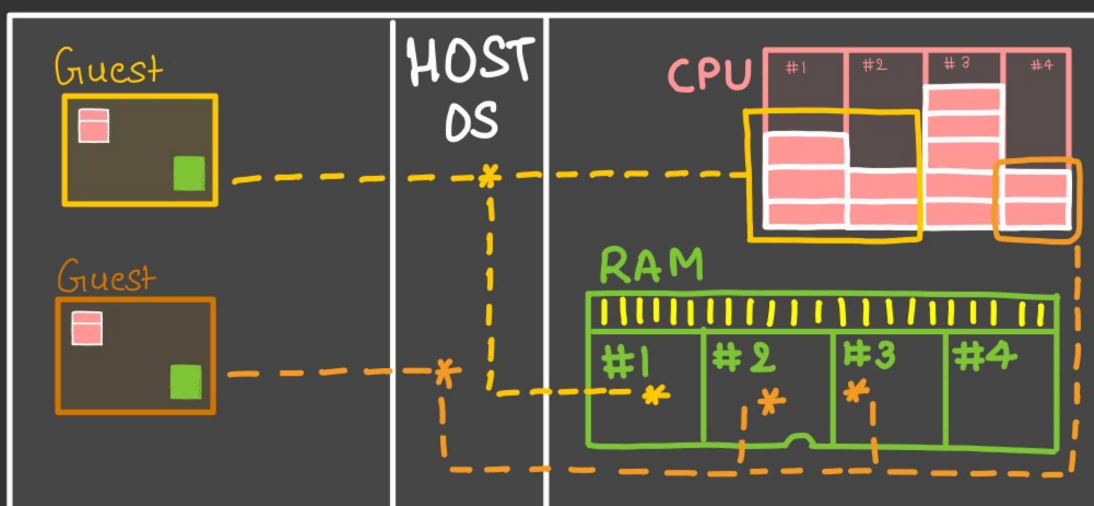
  ↳ OS independent: By packaging it

  ↳ secure: 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 hardware isolation
* Multiple VM running on One Single host donot share hardwares, not even with host on which the VMs are running



* It's the responsibility of HOST OS to perform Hardware isolation

sec_r0

* 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 window as they use Linux techniques.
So behind the scene, windows create Linux VM and runs docker in that VM.

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
```
↳ Create a memory cgroup named foo

```
$ echo 40000000 | sudo tee /sys/fs/cgroup/memory/foo/memory.limit_in_bytes
```
↳ limit foo cgroup memory to 40MB

```
$ echo 1337 > /sys/fs/cgroup/memory/foo/cgroup.procs
```
→ Add PID 1337 to memory cgroup named foo

```
$ ps -o cgroup 1337
CGROUP
8:memory:/foo,1:name=systemd:/user.slice/user-0.slice/session-4.scope
```
→ List Cgroup settings for 1337

* If process 1337 takes more Space than 40MB, Kernel will Kill it ☺

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: vitual DNS and hostname
7. cgroup: allocates hardware resources to NS.

A control group is a namespace too !!

Yes.
That namespace limits hardware resources

Open your terminal

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

```
$ sudo nsenter -t 1337 --mount --pid
```
↳ move current user inside namepace 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 allots virtual resources and cgroup virtually allots hardware resource.

@sec_r0

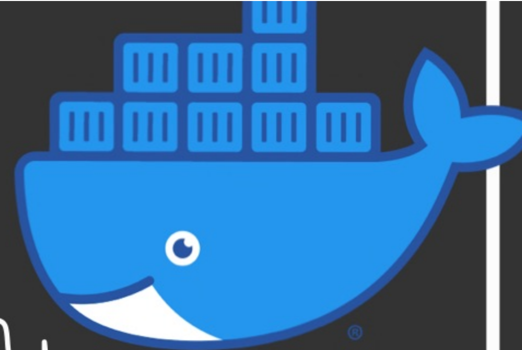
So, docker containers are isolated environment, isolation created using Linux Kernel namespaces

That's why they are not virtual machines.
Just a light weight virtual isolation.

* Virtual machines allocate hardware resources.
* where as containers shares the same hardware resources and isolation is done by kernel on software level.
* Dockers do this magic automatically

@sec_r0

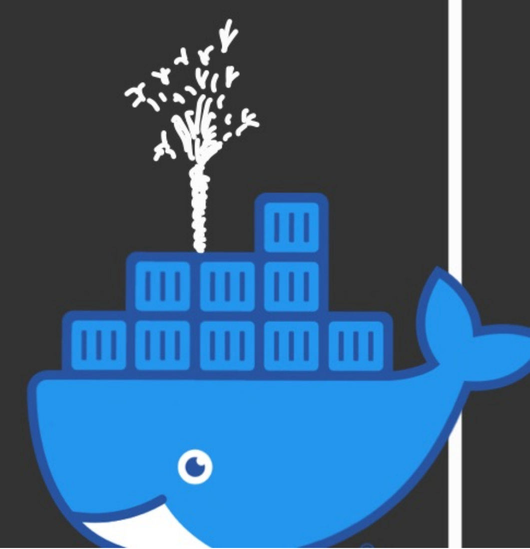
And my magic is well battle tested.



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

Testing Installation

* Run hello-world container

$ docker run hello-world

docker → docker command

run → 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]

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

Dockerfile

* You create docker image from Dockerfile
* Dockerfile contains set of instructions about how to build docker image.

$ docker build -t <name> .

given here

build the docker image with name from filename Dockerfile in current directory

* 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

@sec_r0

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

INSRUCTION argument

CaSe insensitive

Dockerfile example

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

FROM: build docker image from existing docker image name python

RUN: Run command mkdir after using python base image

COPY: Copy all python files from host current directory to inside docker

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

RUN: For running command inside docker

EXPOSE: Tell docker that Container listens on Specific port. This instruction donot publish port to host system

ENTRYPOINT: Entrypoint Command tells what the Container will run. Here it will run Python command.

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

@sec_r0

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
* Various instructions in docker file creates a new layer

Docker file

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

- FROM python → does not create new layer
- RUN mkdir /foo → # 5fd2ef
- COPY *.py /foo/ → #af21ef
- WORKDIR /foo → its not a write instruction
- RUN apt update → #112ba2
- EXPOSE 80, ENTRYPOINT ["python"], CMD ["server.py"] → Again these are not write images

docker image

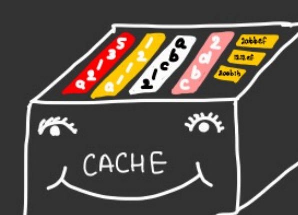
5fd2ef
#af21ef
#112ba2

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

Base image	Image 1	Image 2	Final Image
Python	Python	Python	Python
	# 5fd2ef	# 5fd2ef	# 5fd2ef
		#af21ef	#af21ef
			#112ba2

Base image —RUN→ Image 1 —COPY→ Image 2 —RUN→ Final Image

Image 1 ←—→ Image 2: Intermidiate images are deleted after build. But Layers exist in cache



$docker build -t <name> .

If layer exists pull it. ①

If its a new layer cache it. ②

You can only Read from layer if you write I will create a new Layer

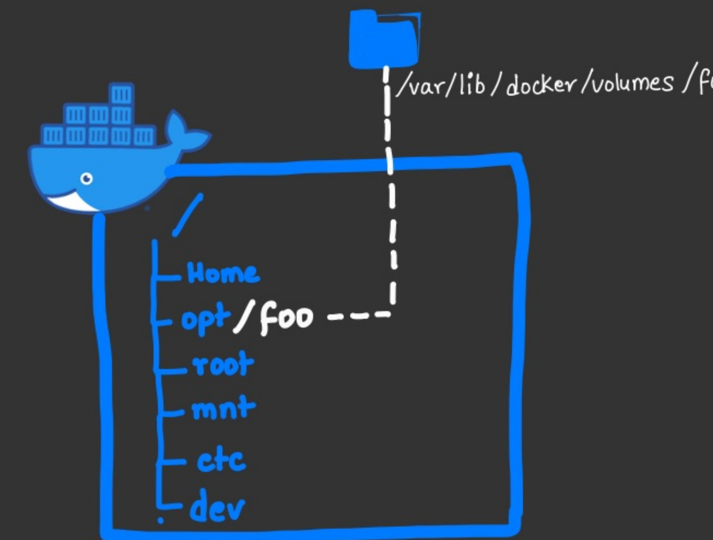
sec_r0

Docker Volumes

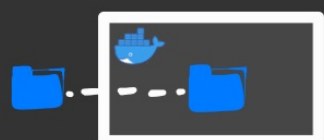
Important point to note here is that, on running docker image and create docker instance, any data that you generate/add to docker instance, that data is lost as docker instance exits.

So how that data can be preserved.

Docker volumes......



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

Now if your docker container exits, volume will keep data stored.

Attaching Volumes

Volume that we created

$ docker run -v foo:/app nginx

location where we want to mount volume inside docker

OR

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

--mount is more verbose

Volume Lifecycle

Create — $ docker volume create foo

↓

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

↓

Docker Instance Removed — $ docker rm nginx-docker

↓

Remove — $ docker volume rm foo

sec_r0

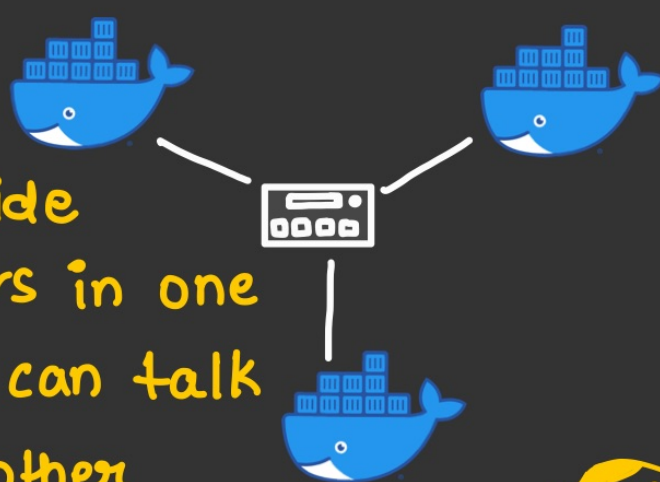
Docker Networks

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

Yes thats when docker networks comes handy

* Couple of points to remember

① apps inside two dockers in one network can talk to each other



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

- Bridge
- Host
- Overlays *
- Macvlan *
- None
  → Disable all networking

$docker run --network none foo

Let's see what network drivers mean

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

sec_r0

Network Drivers And Working with N/Ws

Bridges

* defaut network driver used when docker network is created.

* only containters in same bridge can talk to each other

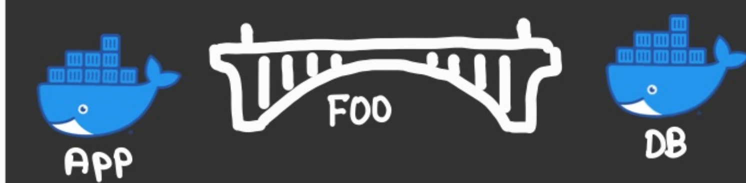
* Containers on same bridge expose all ports to each other

$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

Already running container name

hostname entry to create in side app container

@sec_r0

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

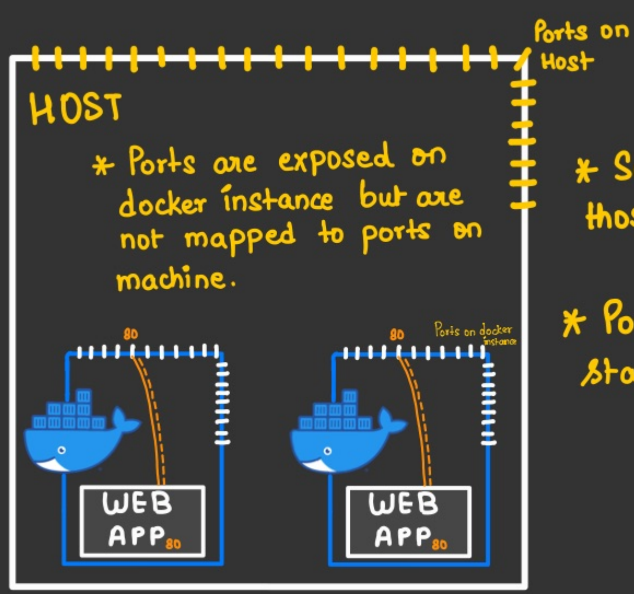
So how can that be done, GROOT....

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

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

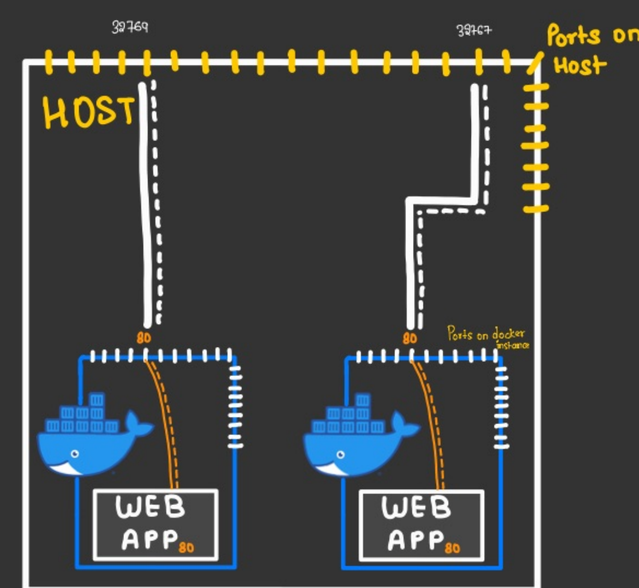
Port on host

docker exposed port

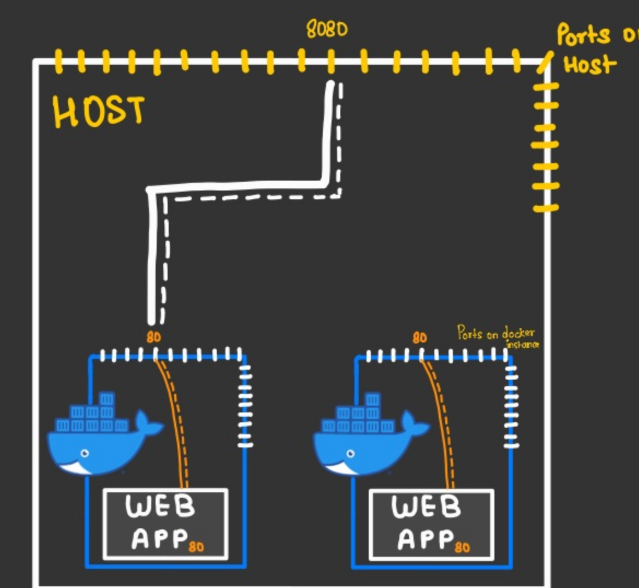


* So we need to publish those exposed ports

* Ports the published while starting a container



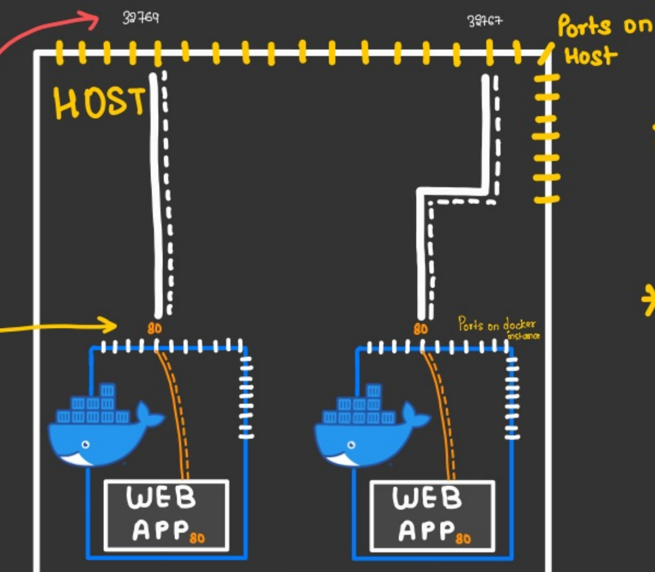
* This approach maps all exposed ports to some random ports



* Only maps the ports which are mentioned in command

@sec_r0

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



* This approach maps all exposed ports to some random ports

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

* 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

* Other variants of -p can be....

$docker run -p 127.0.0.1:8080:80 --name app-name web-app

- 127.0.0.1 → Host Interface publish on local host only
- 8080 → Host port
- 80 → docker port

$docker run -p 0.0.0.0:8080:80 --name app-name web-app

- 0.0.0.0 → Publish on all interfaces on host

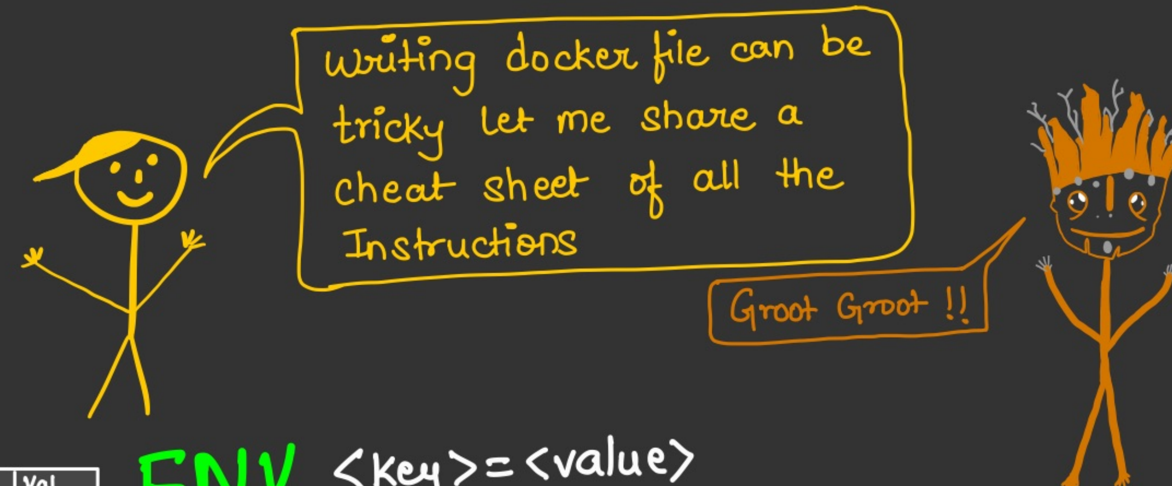
* For multiple ports

$docker run -p 8080:80 -p 7070:8080 --name app-name web-app

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

sec_r0

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>

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 over written while running docker

sec_r0

Dockerfile Cheatsheet .... contd

VOLUME <path> [<path>...] This doesnot mounts external volumes, rather it just creates a mount point for holding volumes

USER <username | uid> Sets user for subsequent dockerfile Instructions, if not specif ROOT user is used. This is not user in HOST, this is user inside docker.

WORKDIR <path> Sets working directory for subsequent docker Instⁿ

ARG <name>[=defaut] defines a variable may be with default value that can be passed at build time.

ONBUILD <Instⁿ> 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

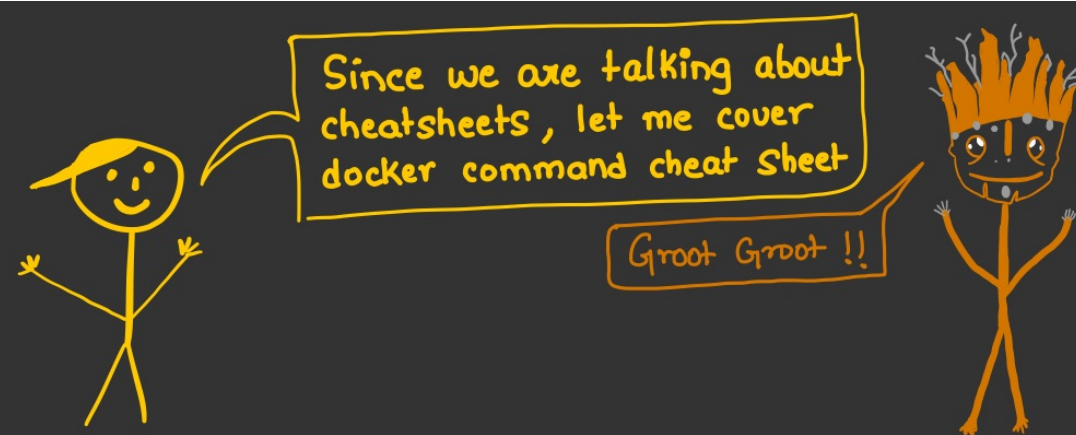
<value> : value in angle brackets needs to be replaced   [ ] : Denotes optional

sec_r0

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

• ⇒ any thing with this color is optional

sec_r0



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

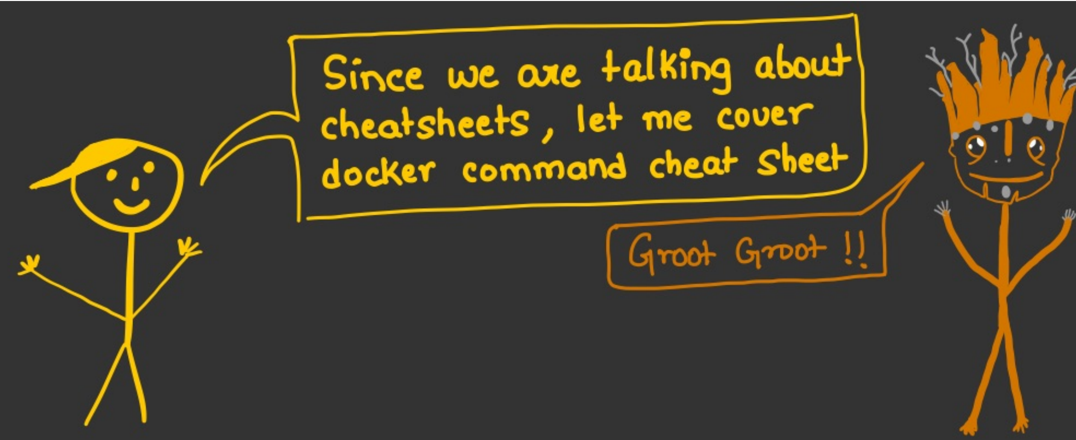
$docker history <image-name>
⇒ Show image history

$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 .....



Talk to Container

$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

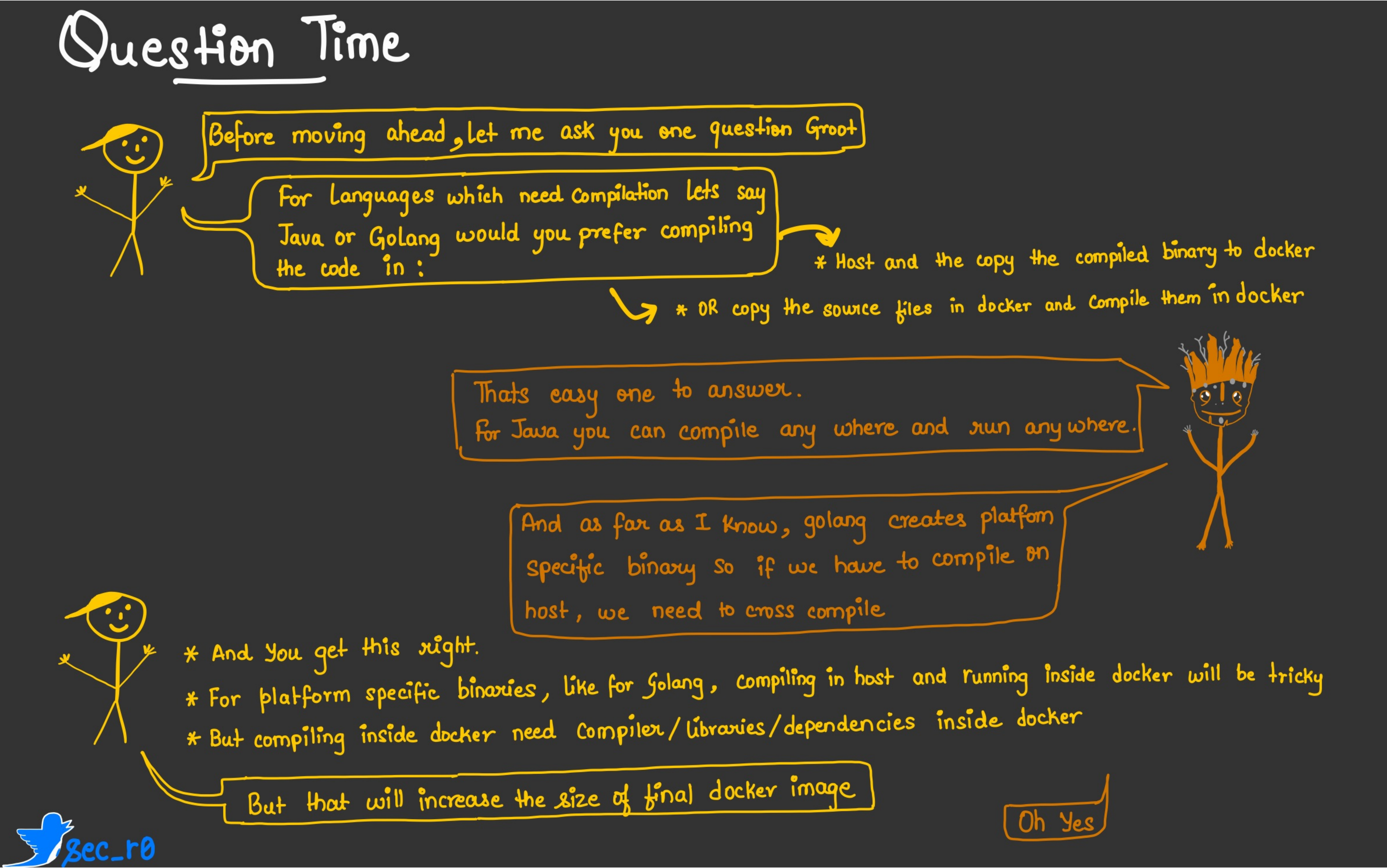
$docker wait <container-name>
→ wait till container terminates

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

* You can also create account on hub.docker.com 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

sec_r0

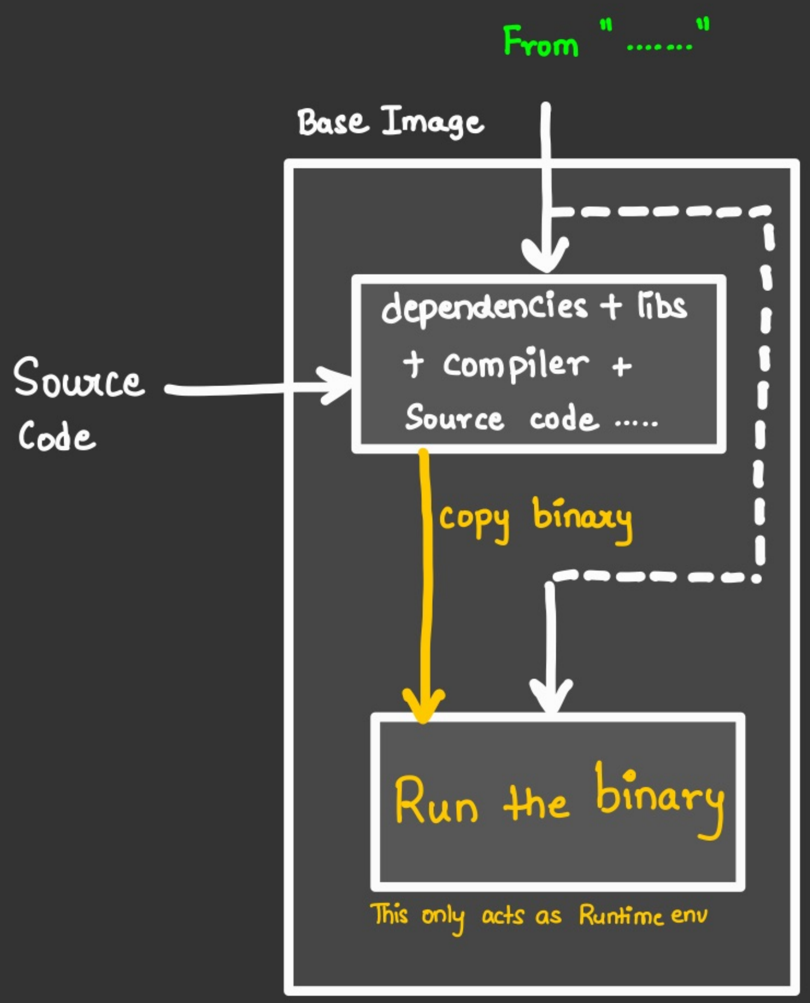

Question Time
Before moving ahead, let me ask you one question Groot
For languages which need Compilation lets say Java or Golang would you prefer compiling the code in :
* Host and the copy the compiled binary to docker
* OR copy the source files in docker and Compile them in docker
Thats easy one to answer.
For Java you can compile any where and run any where.
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
sec_r0

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

And that's why we use **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

Docker image at this stage will be discarded automatically.

Copy source file from host

Compiling it for any linux OS.

to current location

From this location of Builder Stage

Now copy file from Builder stage

sec_r0

Docker Security Best Practices

→ For Dockerfiles

→ For running docker containers

Insecure Dockerfiles can cause security issues
And since this zine 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

@sec_r0

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

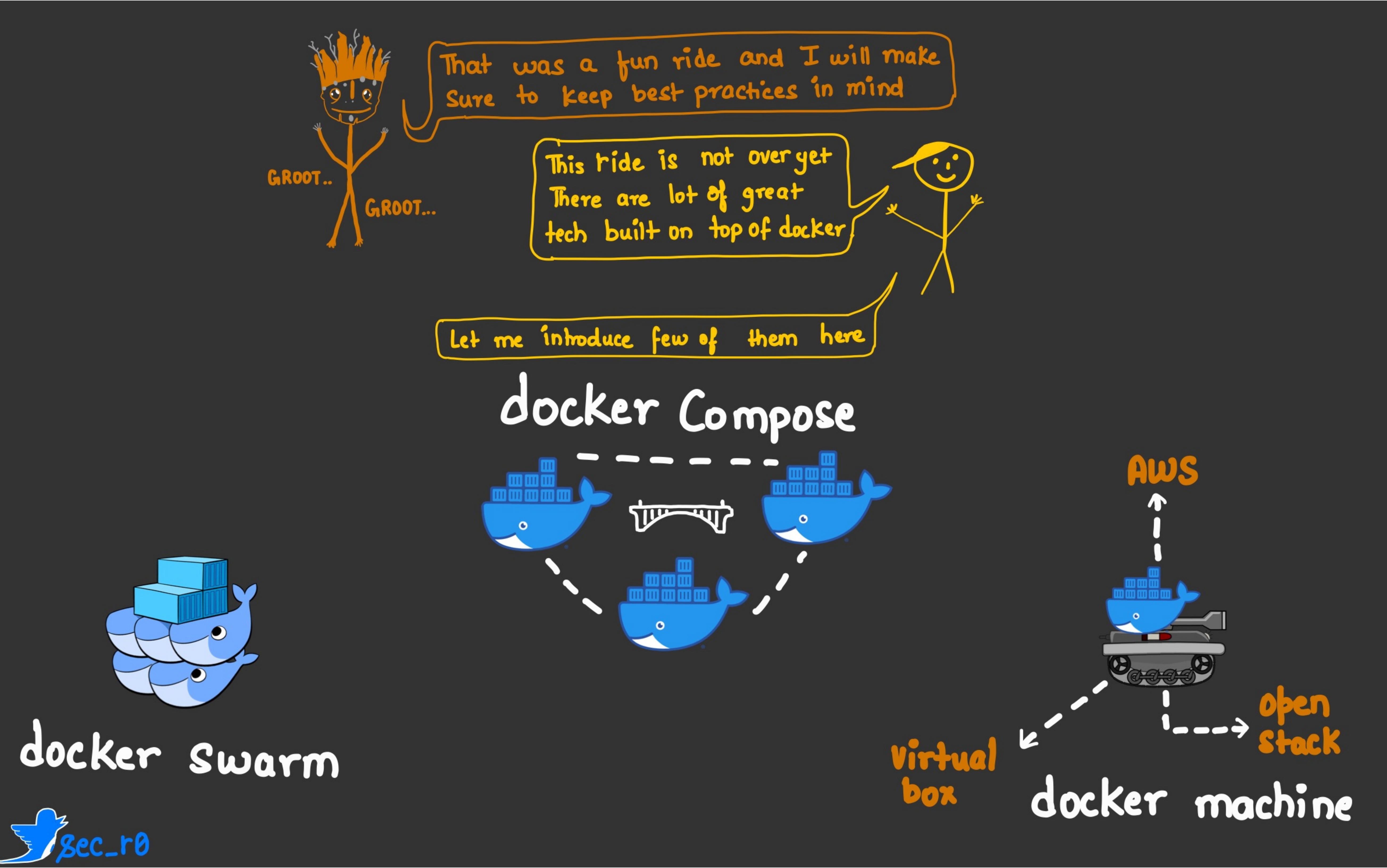
  $ docker run -u 4000 alpine

  ⇒ useful in cases when you dont Control dockerimage
- 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

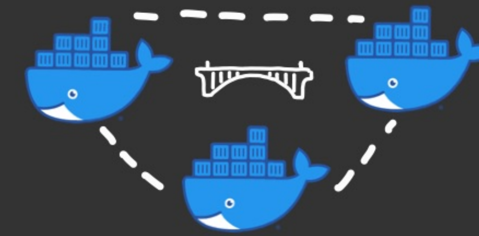
  Best Approach - Remove all, give one by one

  $ docker run --cap-drop all --cap-add CHOWN alpine
- use --log-level = INFO From exposing debug logs of application.
- 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


That was a fun ride and I will make sure to keep best practices in mind
GROOT..
GROOT...
This ride is not over yet There are lot of great tech built on top of docker
Let me introduce few of them here
docker Compose
docker swarm
AWS
open stack
Virtual box
docker machine
sec_r0

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......

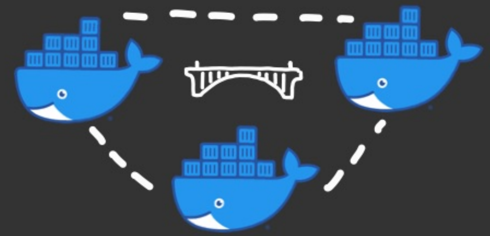
But what if I ask you clean every thing after Sometime ??

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.....

sec_r0

docker Compose



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

① → You'll be defining the entire configuration inside a 'spec' file → YAML format

② → By telling docker-compose to perform operation based on spec

docker-compose.yml

```yaml
version : "3.9"
services :
  web :
    build : .
    ports :
      - "8080 : 8080"
    volumes :
      - . : /code
      - logvol : /var/log
  redis :
    image : redis
volumes :
  logvol : {}
```

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

Configurations for how to run a docker Container

* 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

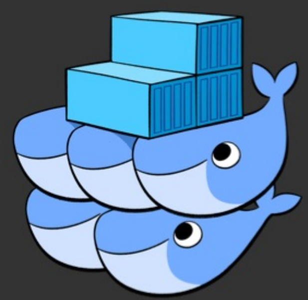
$ docker-compose build → build docker image if any

$ docker-compose up -d → daemon mode .... optional; → Bring containers to life

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

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

sec_r0



docker Swarm

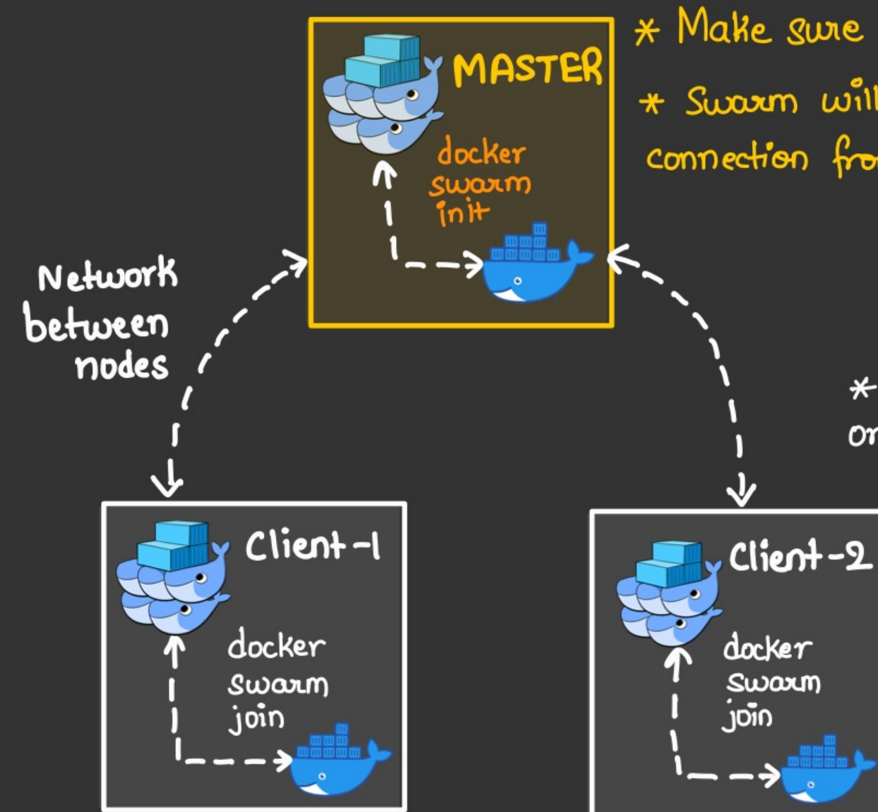
Docker swarm is an orchestration built by docker Inc.

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

* 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



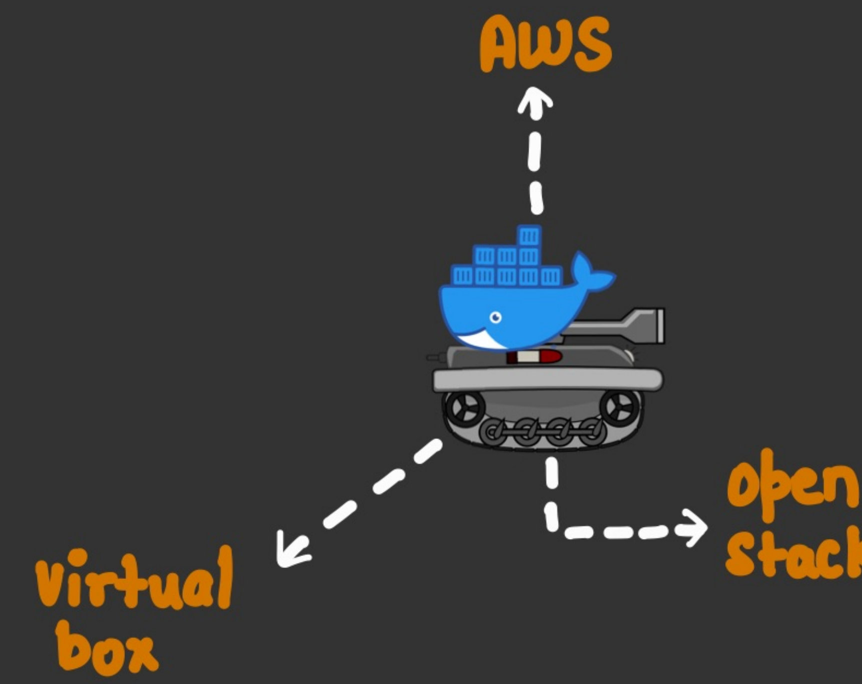
* Make sure nodes are reachable
* Swarm will take care of connection from node to container

* K8s is advanced orchestration than swarm. So, I will create a zine on K8s than to create one on Swarm.

sec_r0

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

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

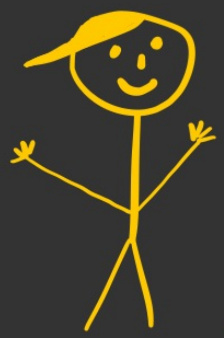
Server [REST]

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

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

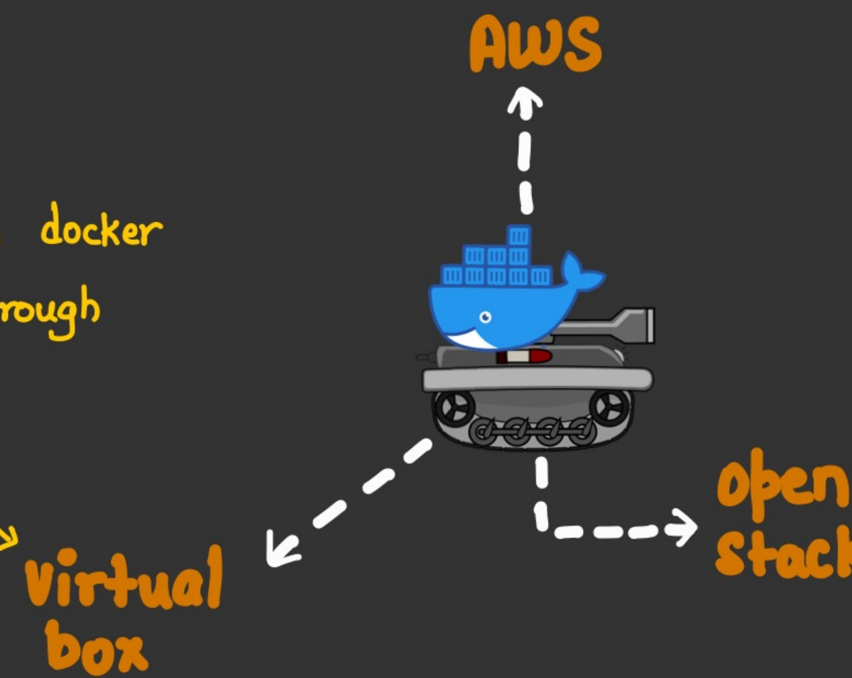
sec_r0

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

name of docker machine

$ docker-machine create --driver virtualbox foo] Single command will setup everything

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

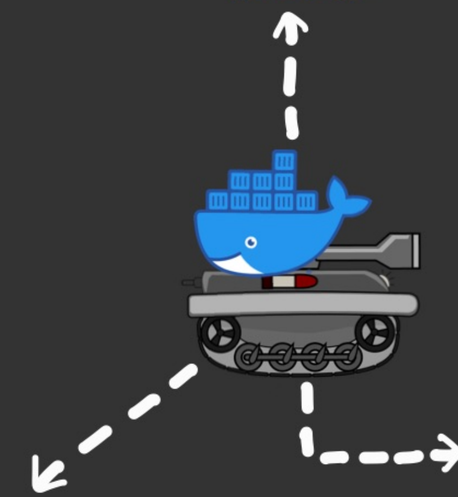
sec_r0

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



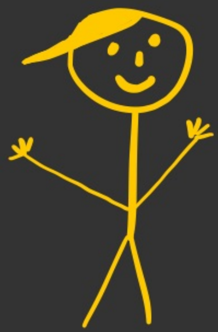
* After using this setup you won't realise if docker is running on local or remote system

* This makes your personal system clean.

GROOT.. GROOT...

I will use machines.....

@sec_r0



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

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

GROOT..
GROOT...

Thank You ......

Read more zines @ SecurityZines.com

Join for more updates sec_r0