Docker & Kubernetes

1. What is Docker?

↳ used to create and manage containers

#Container is a standardized unit of software (code + Environment)

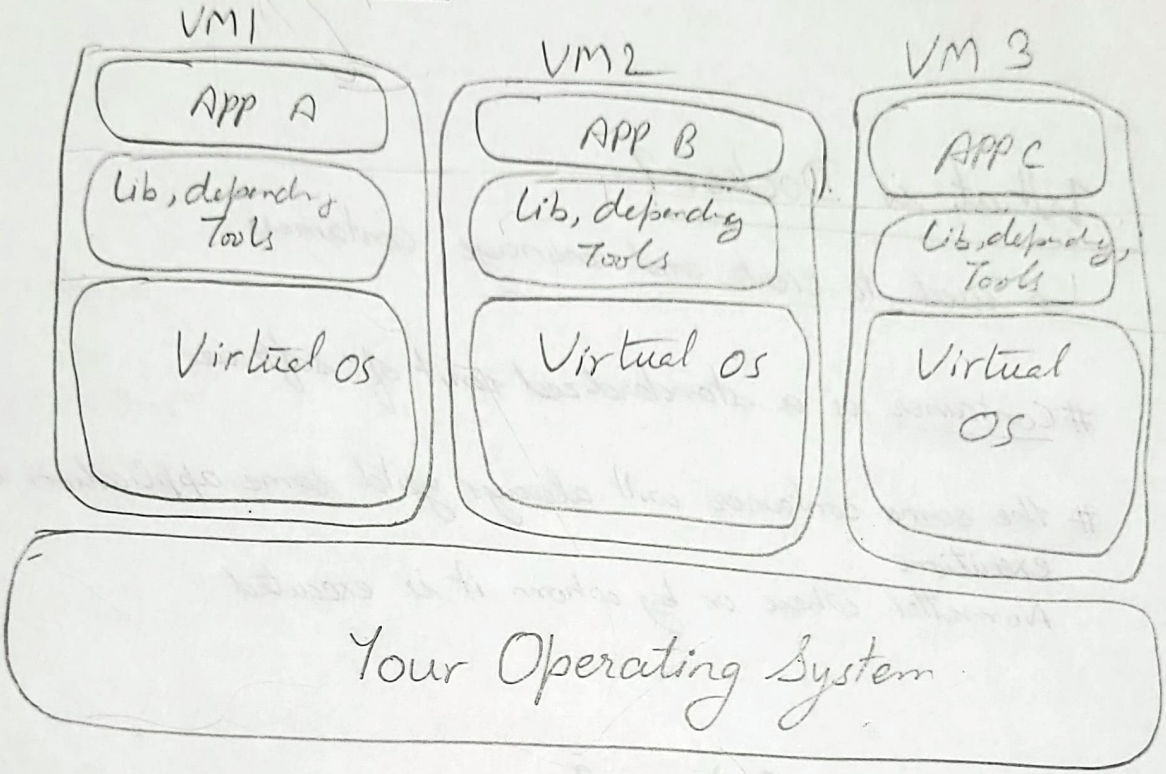
The same container will always yeild same application and execution
Nomatter where or by whom it is excauted

Why Docker Containers?

1) we want the same environment in Dev & Prod
2) it should be easy to share development env b/w developers
3) Dependency management

Virtual Machine vs Docker Containers

1. Virtual Machines



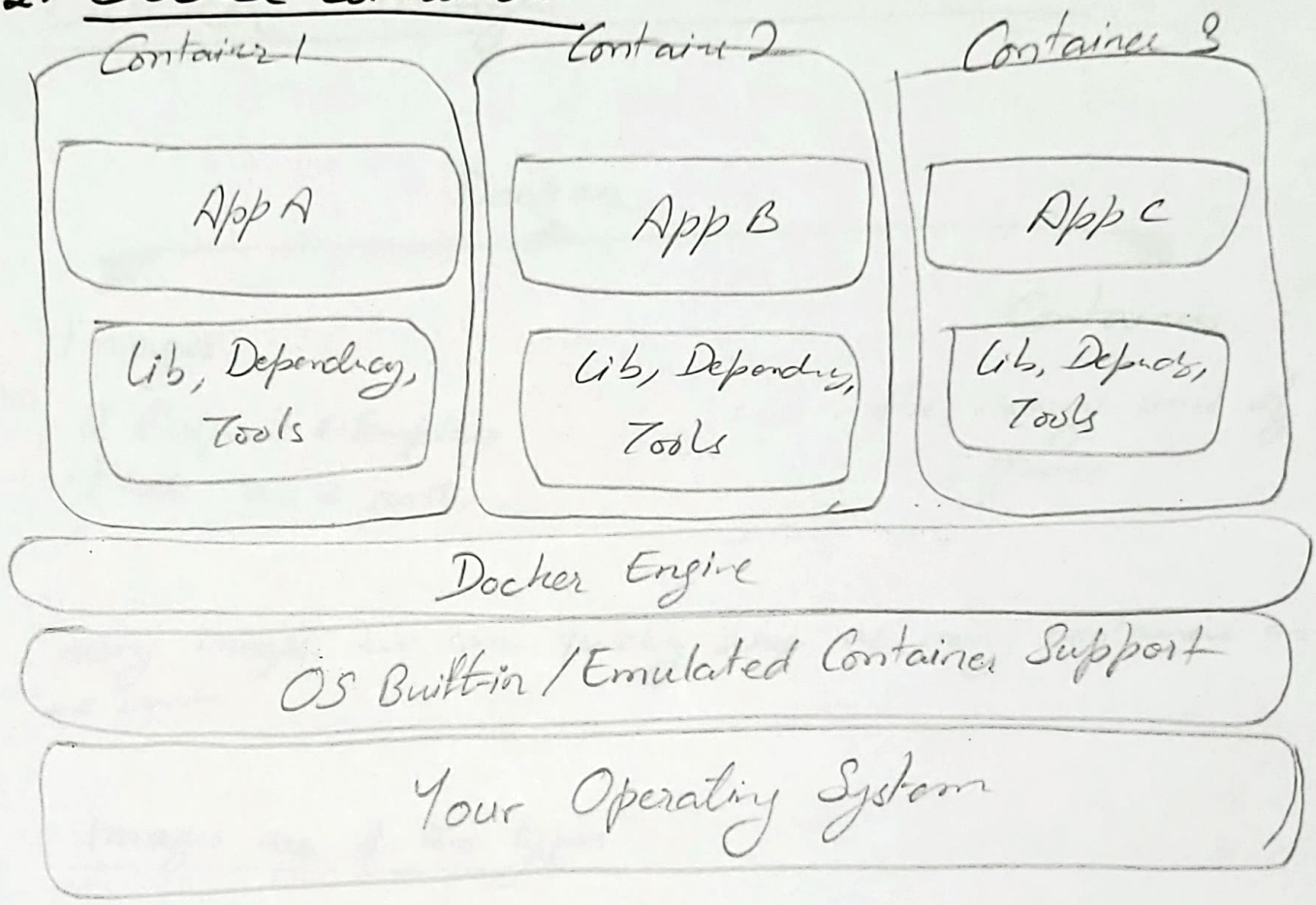
Pros

1. Separated Environments
2. Environment Specific Config are Possible
3. Environment config can be shared and reproduced reliably

Cons

1. Redundant Duplication, waste of Space
2. Performance can be slow, boot time can be long
3. Reproducing on other Computer/ Server is possible, but may be tricky

2. Docker Containers



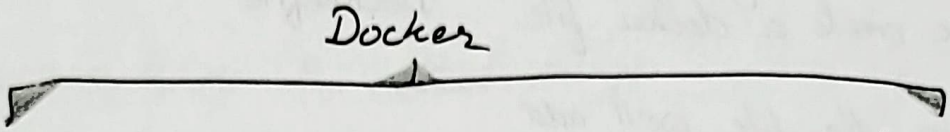
Docker Containers

1. Low Impact on OS, very fast, minimal disk space wastage
2. Sharing, rebuilding & distribution is easy
3. Encapsulates apps and Environment instead of whole machine

Virtual Machines

1. Bigger Impact on OS, slower, higher disk space usage
2. Sharing, rebuilding & distribution can be challenging
3. Encapsulates whole machine

Docker : Images & Containers



Images

i) Blueprint & Template
ii) Code, tools & runtime

Containers

the running unit of Software

using images, we can quickly setup as many containers as we want

Images are of two types

i) PreBuilt images
ii) Our own images

Prebuilt Images

→ can be found on docker hub
→ can be pulled using `docker pull _____` command
→ can be run through `docker run _____` command
→ can be run interactively through `docker run -it _____` command
→ To check what processes are running `docker ps`
→ To check all process that are running `ps -a`
→ Multiple containers can run using same image

To build Custom Image

1. In the directory of the app we're creating an image of, we create a docker file : "Dockerfile"

2. in the file we'll add:

```
FROM node
```
→ pre existing image on either Local or dockerhub
executing this contains will locally cache this image if not done already

```
COPY . /app
```
. → path of app we want to copy
. ⇒ same directory as docker file
/app → path inside container,
a container's file system is isolated from your file system

3. now we'll have to install dependencies

```
FROM node
WORKDIR /app
COPY . /app
RUN npm install
```
WORKDIR → tells docker that use this path as working directory for all cmds
COPY . /app → Absolute path, Relative path will be ./
RUN npm install → install dependency command for Node

4. now will add our execution cmd:

```
CMD ["node", "server.js"]
```

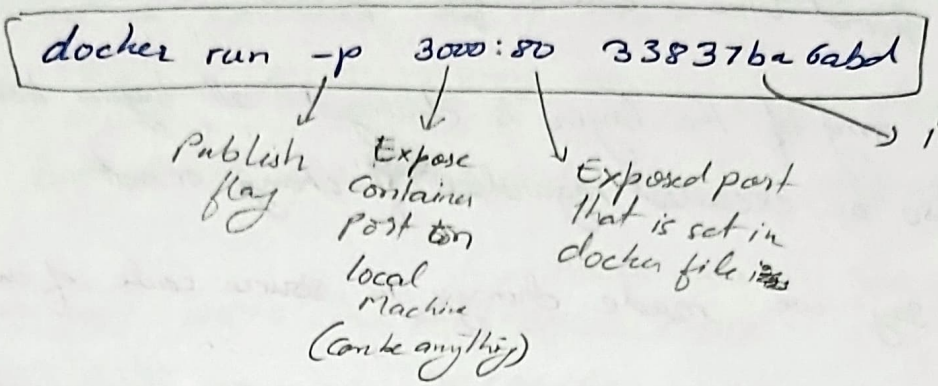
5. We'll add expose endpoints } optional

```
EXPOSE 80
CMD ----
```

6. now to actually create an image:

`docker build .` → relative path of dockerfile

7. to execute/run the actual image:



8. To Stop a running Container,

i) `docker ps -a` : to find running processes to identify
ii) `docker stop <container_name>` : to stop container execution

note: Images are Read Only
i.e. updates to code won't automatically be reflected.
a new image has to be created for new code base

Images works in layers

→ Every instruction in the dockerfile is a cachable layer

→ if ~~one~~ layers are unchanged, the during image creation,
first time : all instructions will be executed
Second time : cache from previous build will be used

→ if any of the layer is changed, all layers below it will also be executed regardless of chang or not

eg: say we made changes to source code of our app.

dockerfile 1

```
FROM node
WORKDIR /app
COPY . /app
RUN npm install
CMD ["node", "server.js"]
```

dockerfile 2

```
FROM node
WORKDIR /app
COPY package.json /app
RUN npm install
COPY . /app
CMD ["node", "server.js"]
```

- dockerfile 2 Runs faster than dockerfile 1 since, when we updated source file, in dockerfile1, ~~npm~~ copy . /app will be executed again, subsequently npm install will also run again. But in dockerfile 2, npm install runs from cache ∴ faster

optimized

Container Management

1. Stopping and Restarting a Container

→ instead of typing `docker run` everytime we coont to start a container, as it creates a entirely new contain

→ we can use `docker start container name` to simply start the container, but it opens the container in a different mode

in `docker run` we're in ATTACHED MODE → Process runs in foreground

in `docker start` we're in DETACHED MODE → Process runs in Background

→ In attached mode, our Terminal is attached to the container and actively listens for console logs, this doesn't happen in Detached mode

→ We can switch modes

Case 1 : we're using run command, but we don't want to attach

⇒ `docker run -d` → detach flag

Case 2 : if we're in detached mode, and we want to go into attach mode

⇒ for a running container, use the command

`docker attach container name -a` → gives logs from now on

→ Alternatively we can use :

a) `docker logs ____` gives all logs so far

b) `docker logs -f ____` → follow

give all logs so far and future logs

Interactive Mode

Python file

```
a = int(input("Enter num1:"))
b = int(input("Enter num2:"))
print("a: " + str(a) + str(b))
```

→ Say you have a python program that expects 2 inputs from user

→ Neither Attached nor detached mode allows you to interact with the running instance vs console itself

→ To do this we need INTERACTIVE MODE

→ there are two flags we use for this

-i and -t OR -it # combined

→ -i : interactive

-t : getting a psudo-TTY (terminal)

- use -i flag when we want to give predefined input

  eg: 1) echo -e "123\n456" | docker run -i ubuntu cat

  → pipe operator takes output from left gives it as input to right

  2) docker run -i xyz < input.txt

  this file contains inputs

  eg 123 / 456 ⇒ input.txt

- use -t flag when you want to see output in terminal window only

- use -it flag when you want to interact Dynamically with the program.
  - -t gives you a terminal
  - -i give you permission to interact with this terminal

Deleting Images and Containers

→ To remove a container, use command `docker rm Container name`

note: you CANNOT remove a RUNNING container. it must be stopped first

→ To stop multiple containers at once pass them with space

eg: `docker rm container_1 container_2 ----`

note

Docker ps lists all containers
Docker images lists all images

→ To remove Image, use `docker rmi image id`

note: to remove an Image, all containers using that image must be removed first, stopped containers also count

this is because containers, (start/stopped) are running instances of Images

→ multiple images can be removed at once

→ To remove all images that are ~~unsue~~ unused use `docker image prune`

Similarly, use `docker contain prune` to remove all stopped containers

Removing Stopped Containers Automatically

→ when running a container, we can use the --rm flag to automatically remove the container once the execution is completed

eg: `docker run --rm container_name`

note: 1) this can't be used for a container that is already built i.e can't work with the `start` command

2) to stop a container that's already built automatically, you may use:

`docker start -a <container_id> && docker rm <container_id>`

Inspecting Images

→ use `docker image inspect ____` to get metadata about the image

Copying file from and into a Container

→ to copy files into a container, use

```
docker cp <Path of local file> <container_name>:<path of file in container>
```

eg docker cp dummy/. testContainer:/testpath

this will copy files inside dummy folder to the /testpath folder inside container "testContainer".

- useful when uploading config files

→ To copy files out of a container into local machine

```
docker cp <container_name>:<Path of file inside container>  <Path to file in local machine>
```

eg: docker testContainer:/testpath dummy

Naming and Tagging a Container & Images

1) `docker run --name name <container_id>`

use this command to name containers

2) `docker build -t <name>:<tag> .`

use this command to name & tag an image

name:tag can be used to create specific version

eg: node:14
node:latest
:

note

- you can't rename an already built image
- you can rename a built container

`docker rename <oldname> <newName>`