# DOCKER

It's all about apps

* IN VM’S

# We Isolate Services

* To host our apps we need Infrastructure.
* We Use VM’s/Cloud Computing to setup Infra
* We Isolate our service in OS of VM
* Because of Isolation we end up setting up multiple VM’s/Instances.
* VM’s/Instances will be overprovisioned.
* Results in High CapEx and OpEx

# VM’s are expensive

●Every VM has OS

● OS needs nurturing

● OS Needs Licensing

● OS takes time to boot

● VM’s are Portable but Bulky.

● VM needs Resources for its OS

● All this to Isolate services

# Point to be Noted.

● Isolating services are IMPORTANT (Need OS)

● High availablity achived by multiple instances/vm’s

● Portablity Matters or Eases the Deployment.

● All this raises CapEx and OpEx

Isolation without OS? (Like hollow vm’s)

Imagine Multiple Services running in same OS but isolated and it has own file system/lib (so no interferences 🡪 container

# Containers

Process running in a Directory.

isolated in a directory. So all the libraries, binaries, configuration

it needs is available in that directory from where it is running. And imagine other things like giving IP addresses to the directory, like we have IP address to the virtual machine. So the container is really a kernel trick.

● A Process[Isolated]

● A Directory[Namespace, cgroup]

● Necessary bin/lib in the Directory

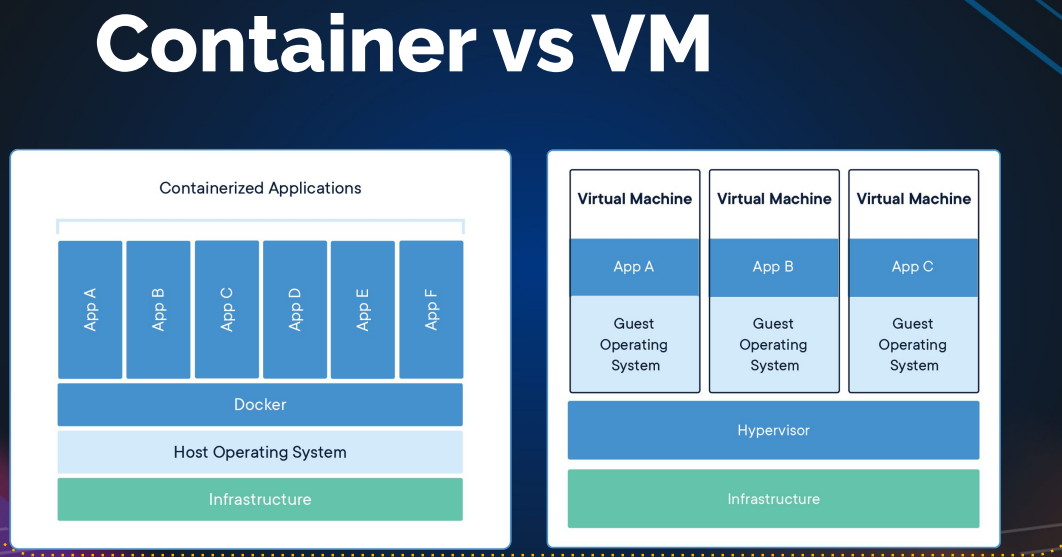
● A directory with IP address to connect.



## Note:

* Containers share the machine’s OS system kernel and therefore do not require an OS per application.
* A container is a standard unit of software that packages up

1. Code
2. Dependencies



## Link

[What is a Container? | Docker](https://www.docker.com/resources/what-container/)

<https://www.docker.com/resources/what-container/>

# VM vs Container

● Containers offer Isolation not Virtualization

● Containers are OS virtualization

● VM’s are Hardware virtualization

● VM needs OS

● Containers don’t need OS.

● Containers uses Host OS for Compute Resource

For a process to run, you need operating system but as we already established, containers don't have operating system but process needs to run on operating system so containers or kernel does some trick and bluffs your process which is running in a directory that this directory is an operating system which in realityis not VM needs operating system and containers don't need operating system that's the biggest difference and for computer resource, container uses host operating system resources

# Docker

Manages your Containers

## Docker History

● Formerly Known as DotCloud Inc

● Into PAAS Business

● Used LXC (Linux Containers)

● Saved CapEx by using Containers instead of VM’s

● Developed TOOLS to manage containers.

● Business Failed.

● Made their tools OpenSource project knows as Docker.

● Got Funding

● Changed name to Docker Inc

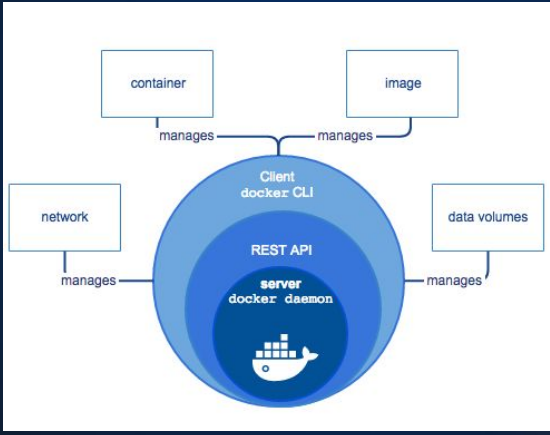
## So What’s Docker?

● Docker Inc

● Docker Engine

● Docker Project (OpenSource)

# Docker Engine



Docker engineer is a Daemon(it is service running in the operating system) and we can connect IT by using Rest API and Client docker CLI

Power house of docker is images

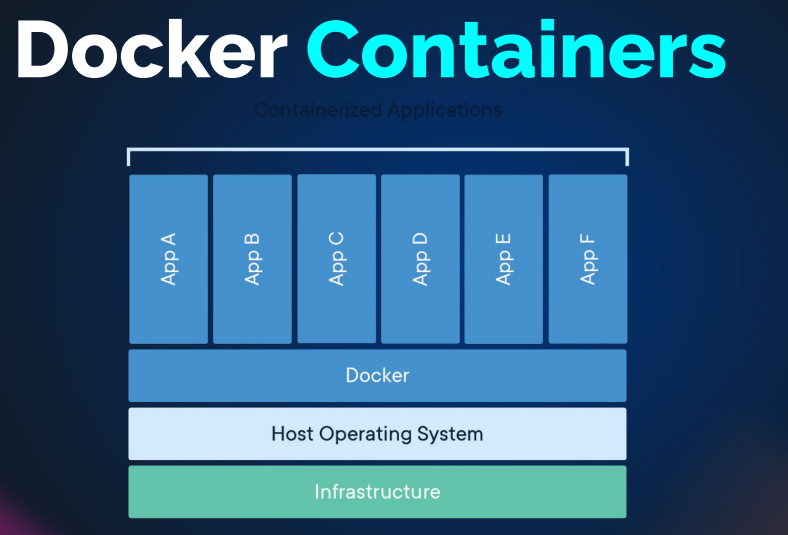
# Docker Containers

Docker containers that run on Docker Engine:

• Standard: Docker created the industry standard for containers, so they could be portable anywhere

• Lightweight: Containers share the machine’s OS system kernel and therefore do not require an OS per application, driving higher server efficiencies and reducing server and licensing costs

• Secure: Applications are safer in containers and Docker provides the strongest default isolation capabilities in the industry



# Docker Installation

* Linux or Windows
* Windows Containers runs on Windows OS
* Linux Containers runs on Linux OS
* Docker Desktop 🡪 it used to run the Linux container on windows OS (But it internal it will create a Linux VM on top of that Linux container will create)

# Docker setup

🡪Create a VM (ubuntu or anything you want)

Docker doc 🡪 <https://www.docker.com/>

🡪 <https://docs.docker.com/engine/install/ubuntu/>

*# Add Docker's official GPG key:*

sudo apt-get update

sudo apt-get install ca-certificates curl gnupg

sudo install -m **0755** -d /etc/apt/keyrings

curl -fsSL https://download.docker.com/linux/ubuntu/gpg **|** sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

sudo chmod a+r /etc/apt/keyrings/docker.gpg

*# Add the repository to Apt sources:*

echo \

"deb [arch="**$(**dpkg --print-architecture**)**" signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \

"**$(**. /etc/os-release **&&** echo "$VERSION\_CODENAME"**)**" stable" **|** \

sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update

install docker

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin

Some commands:

🡪Sudo systemctl status docker

To add the user to group

1. Vim etc/group 🡪 docker:x:998: ubuntu add the ubuntu
2. Use sudo usermod -aG docker ubuntu

And logout and login

* Docker images 🡪 to show the images
* Docker ps 🡪 to show the running images
* Docker ps -a 🡪 to show the all the images
* Docker run Hello-world 🡪 it is basic images, just used to check the whether it is working or not

# Docker commands & concepts

<https://hub.docker.com/> 🡪 it is the backbone of docker because they is a lot of images present(official, non-official etc)

## DockerHub

Registry for Docker Images

## Docker Image

● A stopped Container like vm Image.

● Consist of multiple layers.

● An app will be bundled in an Image.

● Containers runs from Images

● Images are called as Repositories in Registries.

### From chatgpt:

1. Multiple layers will be involved. Like first create image, they you create directory like that they are layers
2. Each layer has its unique file system format.
3. These layers will operate in read-only mode.
4. Our application is mainly packaged within the image.
5. Containers are executed directly from these images.
6. Containers are tightly connected to the images, unlike VMs.
7. VMs created from images, like Vagrant boxes, result in separate entities, whereas containers run directly from images.
8. Containers prevent image removal while running.
9. In registries, images are referred to as repositories, such as the "Tomcat" repository on Docker Hub.
10. Docker images become containers when they run on the Docker engine.
11. Docker Hub isn't the sole registry; others include GCR (Google Container Registry) and Amazon ECR.
12. You can even create your private registry for image storage, like using Nexus Three, JFrog Artifactory, or Docker Trusted Registry (DTR), which is container-based.
13. Containers are derived from images.
14. Ubuntu image serves as the foundation for running containers, and you can create as many containers as needed.
15. Containers maintain a thin read-write layer, with the data primarily sourced from the image, which remains in read-only mode.

Top of Form

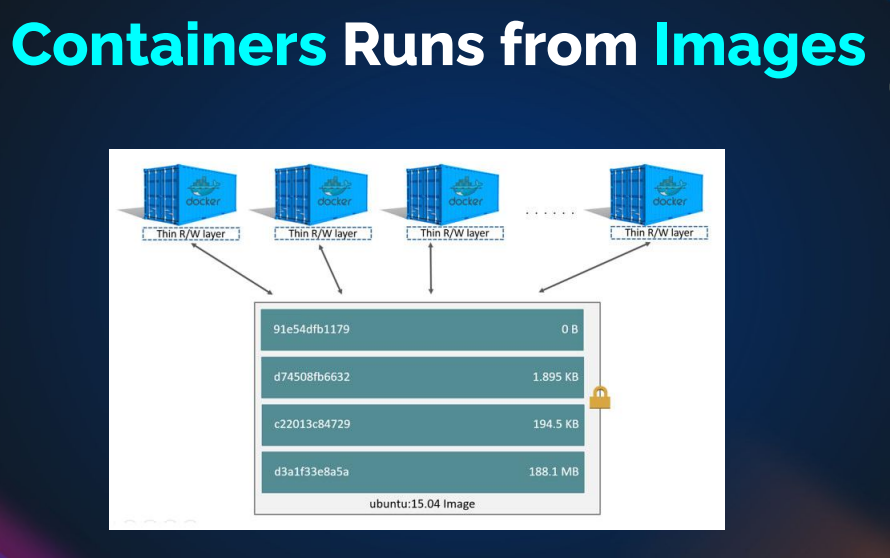
# Docker Images

Images become containers when they run on Docker Engine.

## Docker Registries

* Storage for Docker Images.
* Dockerhub is default registry
* Cloud based Registries.
  1. Dockerhub
  2. GCR (Google Container Registry)
  3. Amazon ECR
* Inhouse or Local Registries

1. Nexus 3 +
2. Jfrog Artifactory
3. DTR (Docker trusted Registry)



1. When working with containers, you are essentially running them from a base image.
2. Containers operate with a thin read-write layer, saving space by not duplicating the entire image content.
3. Image layers are set in read-only mode, and containers access the data from the image.
4. If there is a directory like "/opt/data" in a layer, containers reference it.
5. To create a container, you use the "docker run" command, specifying the desired image, such as "docker run hello world."
6. Docker offers a variety of commands which we will explore shortly.
7. To start, let's visit Docker Hub and select a suitable image. For example, we can choose NGINX, a web service that typically runs on port 80.

## Creating Container

# docker run

Run the docker image ex: nginx

* Docker run –name myweb -d -p 7980:80 nginx

-d 🡪 detech(running from background )

-p 🡪 port

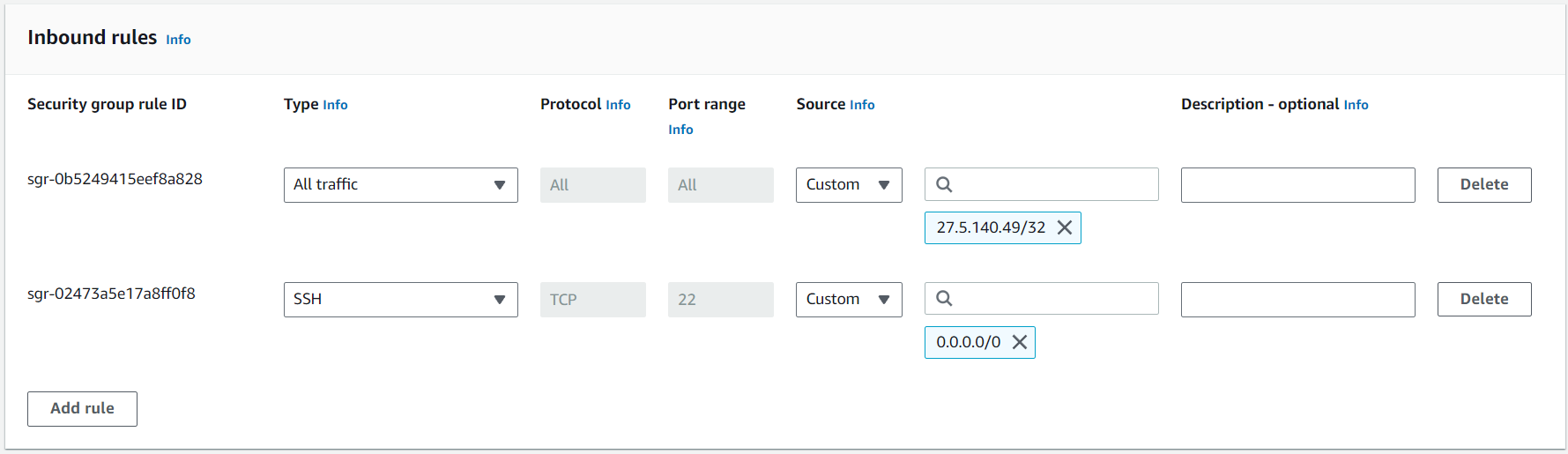
7980:80

7980 🡪 host port

80 🡪 container port

This is also called as port mapping or port forwording

If nginx is not running via bower 🡪 check the security group 🡪 allow all traffic to my ip



### Tag

Docker run –name myweb -d -p 7980:80 nginx:mainline-bookworm-perl

Run the 🡪 docker images to see the tag name

### Note

We can user docker id or docker name to preform actions

ubuntu@ip-172-31-46-217:~$ docker stop 9375eba59de9

9375eba59de9

ubuntu@ip-172-31-46-217:~$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

ubuntu@ip-172-31-46-217:~$ docker start myweb

myweb

ubuntu@ip-172-31-46-217:~$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

9375eba59de9 nginx "/docker-entrypoint.…" 28 minutes ago Up 5 seconds 0.0.0.0:7980->80/tcp, :::7980->80/tcp myweb

ubuntu@ip-172-31-46-217:~$

### History

1 sudo apt update

2 sudo apt-get update

3 sudo apt-get install ca-certificates curl gnupg

4 sudo install -m 0755 -d /etc/apt/keyrings

5 curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

6 sudo chmod a+r /etc/apt/keyrings/docker.gpg

7 echo "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \

8 "$(. /etc/os-release && echo "$VERSION\_CODENAME")" stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

9 sudo apt-get update

10 sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin

11 sudo docker run hello-world

12 sudo vim /etc/group

13 sudo usermod -aG docker

14 sudo usermod -aG docker ubuntu

15 sudo vim /etc/group

16 docker images

17 exit

18 docker images

19 docker ps

20 docker ps -a

21 Sudo systemctl status docker

22 docker images

23 systemctl status docker

24 docker run --name myweb -d -p 7980:80 nginx

25 docker ps

26 docker run --name myweb -d -p 7980:80 nginx:mainline-bookworm-perl

27 docker ps

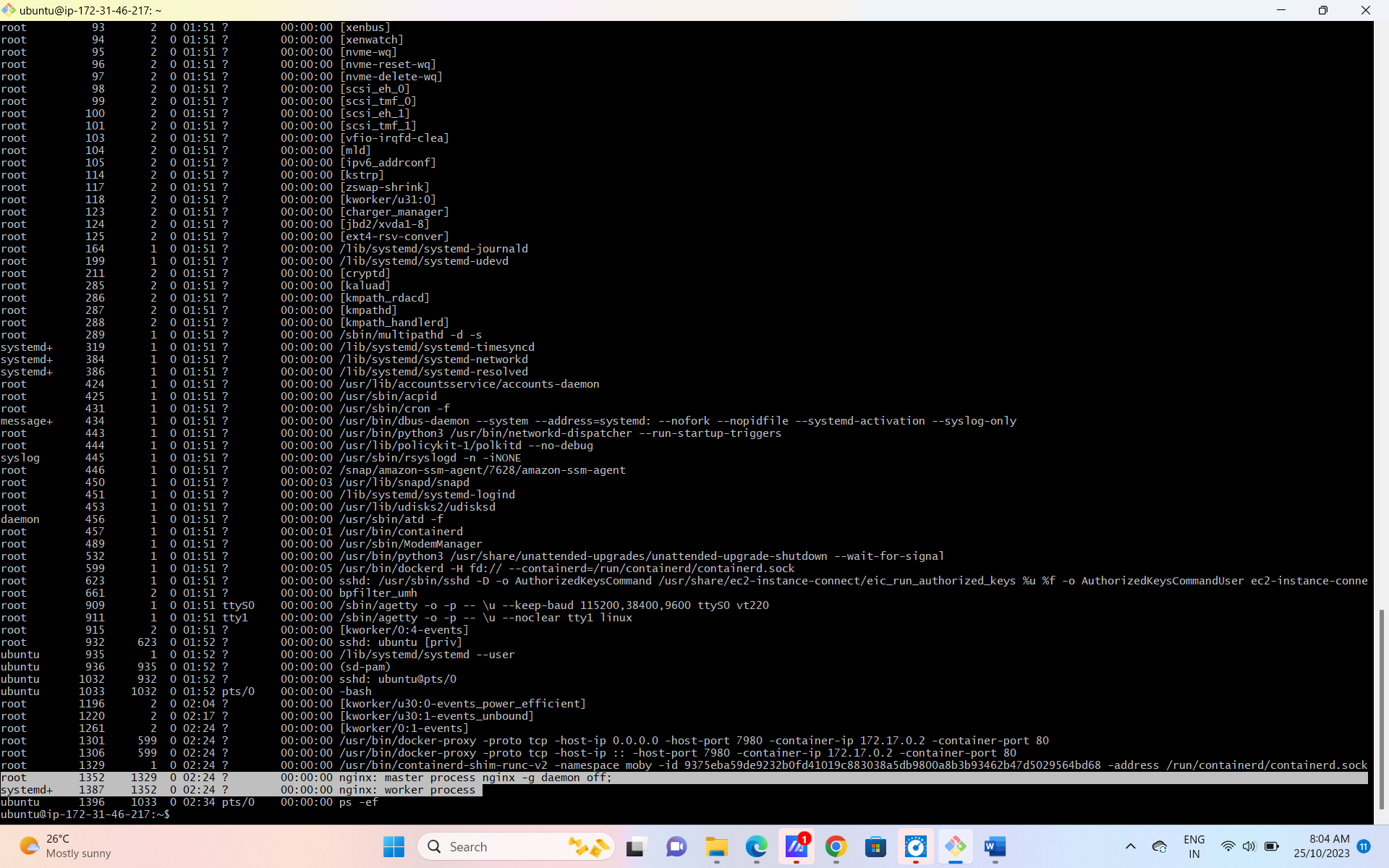
28 docker ps -a

29 docker images

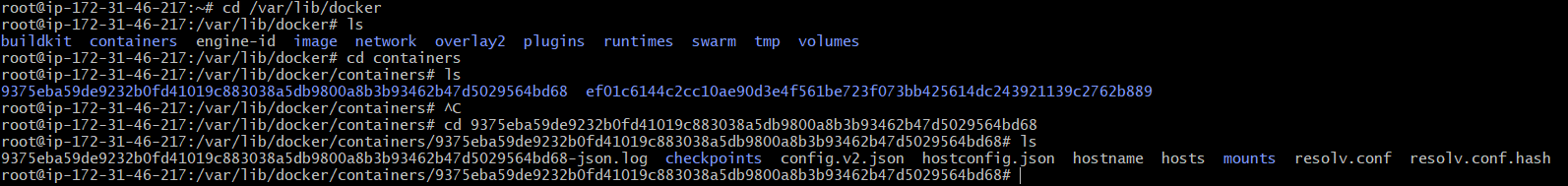
30 history

PROVE

* + 1. Container is just process running from directory
* ps -ef 🡪 check the docker pocess



* /var/lib/docker/container 🡪 open or cat the docker directory then you see the process



root@ip-172-31-46-217:/var/lib/docker/containers/9375eba59de9232b0fd41019c883038a5db9800a8b3b93462b47d5029564bd68# du -sh

44K

## Docker Commands

● # docker images => Lists Images locally

● # docker run => command creates a new container.

● # docker ps => Lists running container

● # docker ps –a => Lists all the containers

● # docker exec => executes commands on containers.

● # docker start/stop/restart/rm

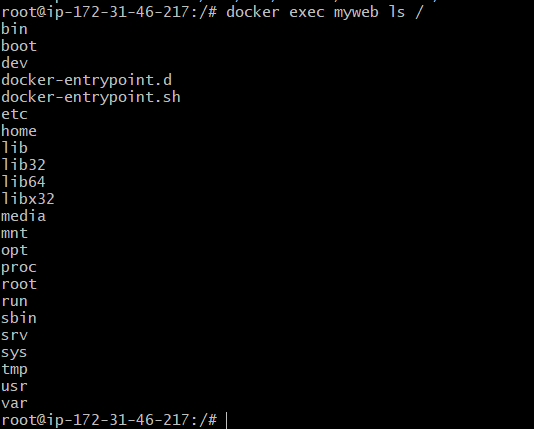
● # docker rmi => Remove docker images.

● # docker inspect => Detail of container & Image

<https://docs.docker.com/engine/reference/commandline/cli/>

# exec commands docker

docker exec myweb ls / 🡪 it will list the file in docker

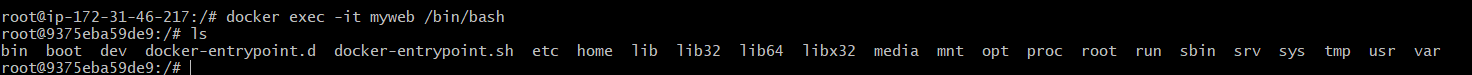


## To get in docker container

docker exec -it myweb /bin/bash

-i is interactive

-t is tty



Note : default we can’t run some commands like (ps -ef , ipconfig etc) , you need to install

1 ls

2 cat etc/os-release

3 apt install procps -y 🡪 we can run the process ps -ef

4 apt update

5 apt install procps -y

root@ip-172-31-46-217:/# docker rm myweb

Error response from daemon: You cannot remove a running container 9375eba59de9232b0fd41019c883038a5db9800a8b3b93462b47d5029564bd68. Stop the container before attempting removal or force remove

root@ip-172-31-46-217:/#

to remove the container first you need to stop the process

stop the process 🡪 remove the container 🡪 remove the image

docker exec myweb ls /

13 Docker exec myweb /bin/bash

14 Docker exec -it myweb /bin/bash

15 docker exec -it myweb /bin/bash

17 docker rm myweb

18 docker stop myweb

19 docker rm myweb

20 docker rmi nginx

21 history

# DOCKER LOGS

The process that’s start and the process output is logs, if something goes wrongs while process starts, we can check with docker logs

* Docker logs container-name

Normally when we run the docker container, it run’s 🡪 if sometime went wrong we can check with docker logs

Ex:

Pull & run the nginx 🡪 if use -d (it will run background ) 🡪 we can check the logs

* If don’t use -d (it takes shell) 🡪 logs is appear in the screen)🡪 if you kill it(control c) 🡪 the process will stop

docker images

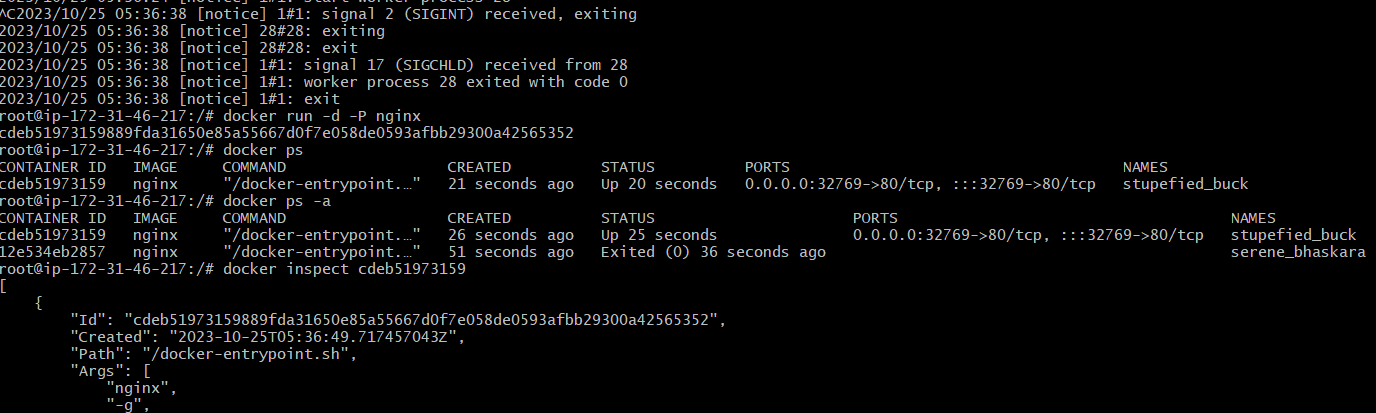
31 docker pull nginx

32 docker run -P nginx

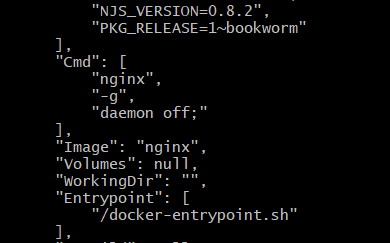
33 docker run -d -P nginx

34 docker ps

35 docker ps -a



36 docker inspect cdeb51973159 🡪 it used to see the meta data of container in JSON format



Note when you run the “docker run” 🡪 first it go for entrypoint 🡪 then CMD and run the commands

Both the command will return output

Note:

This used for troubleshooting, when you build your own images, which we will be doing and when you run container from own custom build images, you might make a mistakes and container won’t start, in that case docker logs is useful

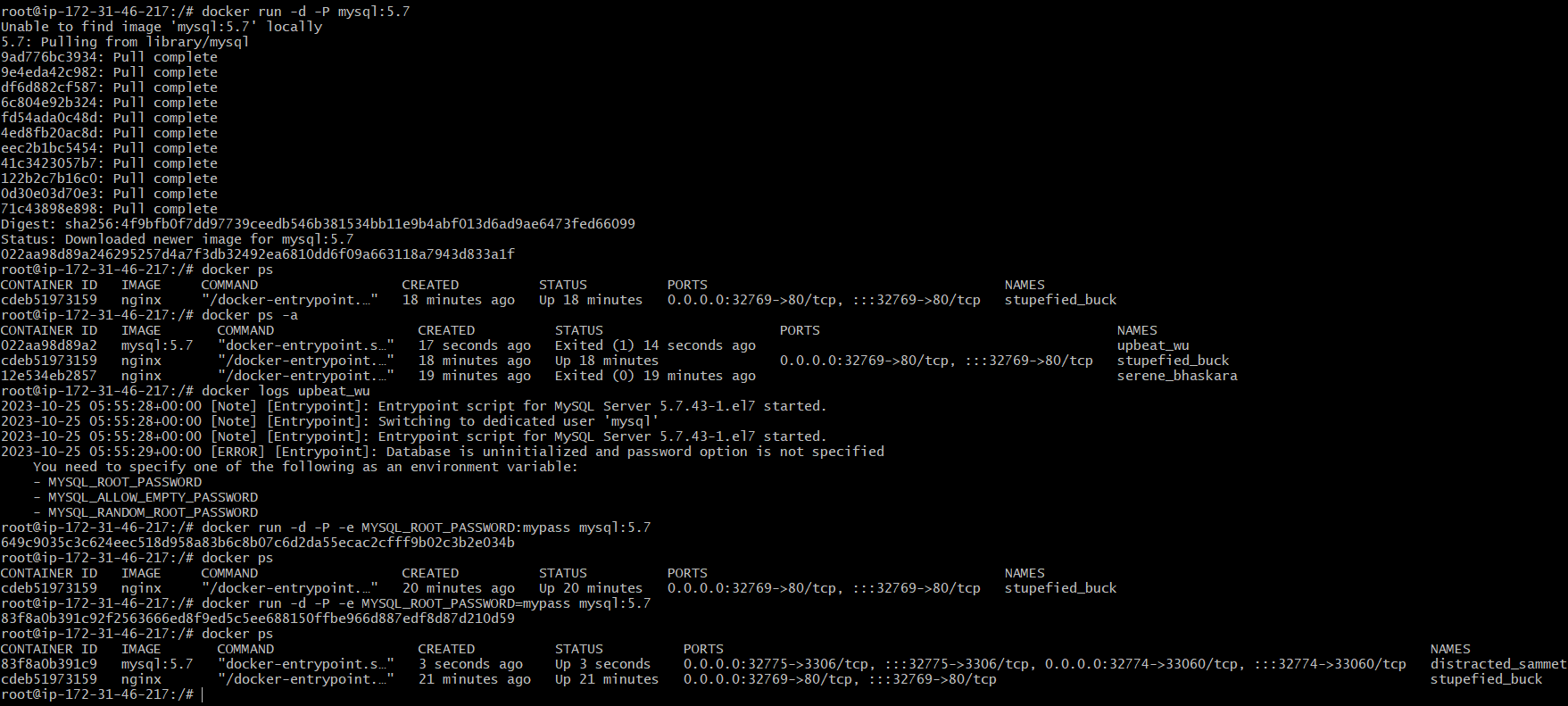
Ex :

Docker run -d -P mysql:5.7

It didn’t not start 🡪 docker ps

If you see the 🡪 docker logs container-name

Then you can understand where the error occurs



docker run -d -P mysql:5.7 🡪 -P it automatically map the port to host : container

41 docker ps

42 docker ps -a

43 docker logs upbeat\_wu

44 docker run -d -P -e MYSQL\_ROOT\_PASSWORD:mypass mysql:5.7

45 docker ps

46 docker run -d -P -e MYSQL\_ROOT\_PASSWORD=mypass mysql:5.7 🡪 -e export variable

47 docker ps

# Container Volumes

# Persistent storage for volatile containers

Container Data

● The data doesn’t persist when that container no longer exists, and it can be difficult to get the data out of the container if another process needs it.

● A container’s writable layer is tightly coupled to the host machine where the container is running. You can’t easily move the data somewhere else. Docker has two options for containers to store files in the host machine

● Volumes

○ Managed by Docker (/var/lib/docker/volumes/ on Linux )

● Bind Mounts

○ Stored anywhere on the host system

Benefits of official image we can trust & document will be they

Ex:

<https://hub.docker.com/_/mysql> 🡪

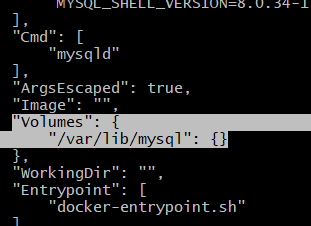
docker run --name some-mysql -v /my/own/datadir:/var/lib/mysql -e MYSQL\_ROOT\_PASSWORD=my-secret-pw -d mysql:tag

but we can find for official image but if it is un-official image we can run inspect command to find the where it going to store.

docker pull mysql:5.7

docker images

docker inspect mysql:5.7



From enteypoint have higher priority, it going run the script then it run the cmd

docker images

ls

mkdir vprodbdata

docker run –name vprodb -e MQSQL\_ROOT\_PASSWORD=mypass -v /home/ubuntu/vprodbdata:/var/lib/mysql mysql:5.7

docker ps

ls vprodbdata/

(inside the folders is same as container folder in /var/lib/mqsql)

docker exec -it vprodb /bin/bash

cd /var/lib/mqsql

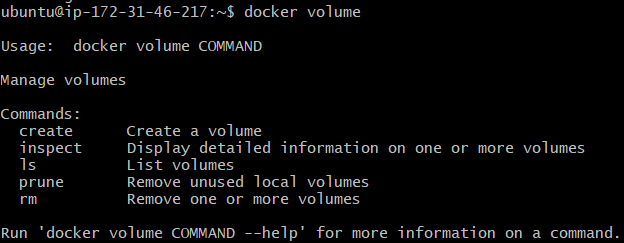
(vprodbdata/ both the folder or content are same)

This is bind mount

Bind mount is mostly used to inject data from host machine to container like developer code reflect in container

But preserving data better option is volume

docker volume



docker run –name vprodb -e MQSQL\_ROOT\_PASSWORD=mypass -v vprodbdata:/var/lib/mysql mysql:5.7

docker ps

sudo -i

ls /var/lib/docker/volume/mydbdata/\_data/

1. "Web vprodb is our Docker container inspector."
2. "A container provides similar information in JSON format, including its creation time, running script, MySQL status, bind(/var/lib/mysql), port bind and more "
3. "You can find the container's ID, the path of its running script, and its arguments, such as 'MySQL D status is running running true.'"
4. "Other details include the process ID on your host machine and the image from which it was created."
5. "The container's log is stored in 'log bar,' allowing you to retrieve it with the 'docker logs' command followed by the container's name."
6. "These logs are generated by the running process and contain its output."
7. "Using the 'docker logs' command with the container name is valuable for troubleshooting."

When run the docker run first entrypoint run the script and then cmd pass as argument to it.

You can connect mysql in side the host machine

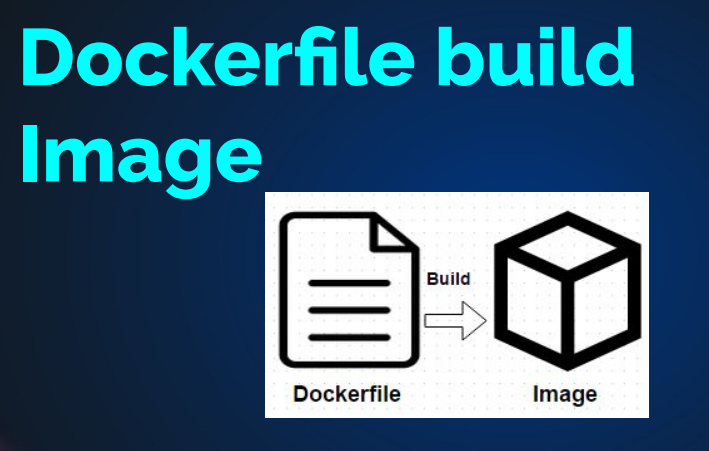
mysql -h 172.17.0.2 -u root -pmypass

show databases;

* Volumes are stored in a part of the host filesystem which is managed by Docker (/var/lib/docker/volumes/ on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.
* Bind mounts may be stored anywhere on the host system. They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time.
* tmpfs mounts are stored in the host system’s memory only, and are never written to the host system’s filesystem.

# DOCKER IMAGES

Dockerfile contains information to build Images



## Dockerfile Instructions

● FROM => Base Image

● LABEL => Adds metadata to an image

● RUN => execute commands in a new layer and commit the results.

● ADD/COPY => Adds files and folders into image.

The difference between add and copy is copy

will just take the file and dump it.

Add instruction you can provide a link, an archive so it

can download from a link and then put it in your

image, or it can even unarchive an archive file.

● CMD => Runs binaries/commands on docker run

● ENTRYPOINT => Allows you to configure a container that will run as an executable.

● VOLUME => Creates a mount point and marks it as holding externally mounted volumes.

● EXPOSE => Container listens on the specified network ports at runtime

● ENV =>  Sets the environment variable

● USER => Sets the user name (or UID)

● WORKDIR => Sets the working directory

● ARG => Defines a variable that users can pass at build-time

● ONBUILD => Adds to the image a trigger instruction to be executed at a later time

On build instruction is useful when you are going

to use this image as the base image.

When you are building some other images, you can specify

any instruction, any command on that, and it will run

that whenever this image is used as the base image.

Refer Documentation <https://docs.docker.com/engine/reference/builder/>

mkdir images

cd images/

mkdir nano

wget <https://github.com/startbootstrap/startbootstrap-grayscale/archive/gh-pages.zip>

or <https://www.tooplate.com/zip-templates/2110_character.zip>

sudo apt update

sudo apt install unzip -y

ls

unzip 2110\_character.zip

cd nano

tar czvf nano.tar.gz

mv nano.tar.gz ../

ls

cd ..

rm rf 2110\_character.zip 2110\_character

mv czvf nano.tar.gz nano/

cd nano/

ls

vim Dockerfile

FROM ubuntu:latest

LABEL “Author”=”Purandhatr”

LABEL “Project”=”nano”

ENV DEBIAN\_FRONTEND=noninteractive

RUN apt update && apt install git -y

RUN apt install apache2 -y

CMD [“/usr/sbin/apache2ctl”,”-D”,”FOREGOUND”]

EXPOSE 80

WORKDIR /var/www/html

VOLUME /var/log/apache2

ADD nano.tar.gz /var/www/html

#COPY nano.tar.gz /var/www/html

docker build -t nanoimg .

docker images

docker run -d –name nanowebsite -p 9080:80 nanoimg:v2

IN Brower ip address:port🡪 3.52.0.192:9080 🡪 check whether it is working or not

<https://hub.docker.com/> 🡪 Signup🡪 fill the information and sign-in

docker run -d –name nanowebsite -p 9080:80 purandhar/nanoimg :v2

Purandhar is username of docker hub

It not going to create image , it going to tag the image 🡪 we check with docker images (both images id should be same)

Docker login

Username: Purandhar

Password:

Login succeeded

Docker push Purandhar/nanoimg:v2

Remove the docker container

docker ps -a

docker stop d2141ae8709d

docker rm d2141ae8709d

docker rmi purandhar/nanoimg :v2 nanoimg :v2

docker run -d -–name nanowebsite -p 9080:80 purandhar/nanoimg : v2

Now anybody in the world runs this command? Anybody?

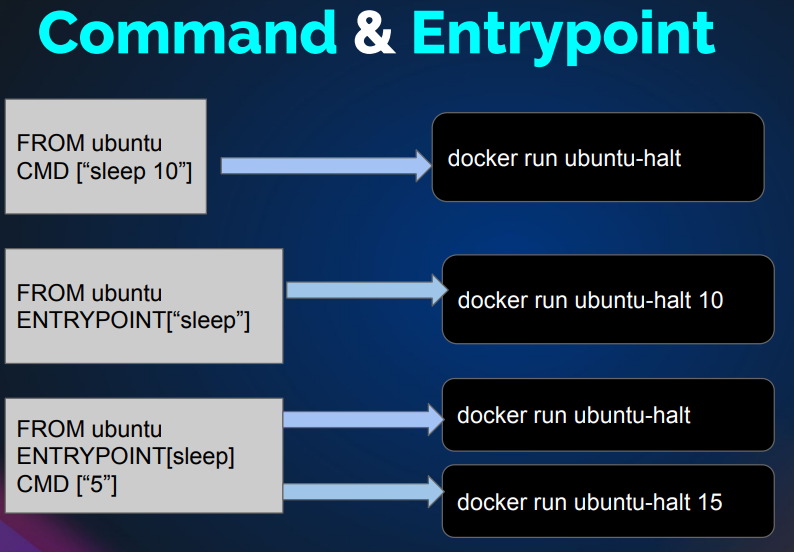
They will have exactly the same image as I am having.

They'll have exactly same container as I'm running.

It's a public image.

docker ps

# Entrypoint and CMD



mkdir EntryCMD

cd EntryCMD

mkdir cmd entry entrycmd

vim cmd/Dockerfile

FROM ubuntu:latest

CMD [“echo”,”hello”]

docker build -t printer:v1 cmd/ 🡪 it should create a image

docker images

docker run printer:v1 🡪 it gives hello 🡪 because cmd[“echo”,”hello”]

vim entry/Dockerfile

FROM ubuntu:latest

ENTRYPOINT[“echo”]

docker build -t printer:v2 entry/ 🡪 it should create the images

docker images

docker run printer:v2 🡪 it won’t give anything because 🡪 ENTRYPOINT[“echo”] 🡪 just echo

but you have command in the ENTRYPOINT and you do not have argument the user has to pass argument

docker run printer:v2 hello

* hello

vim entrycmd/Dockerfile

FROM utuntu:latest

ENTRYPOINT [“echo”]

Cmd [“hello”]

* entrypoint have command and cmd have argument

docker build -t printer:v3 entrycmd/

docker run printer:v3

* hello

if entrypoint and cmd used together then cmd will have default argument, which user can override

docker run printer:v3 hello world

* hello world

## note:

entrypoint have higher priority, and then comes CMD

# DOCKER COMPOSE

Manage containers from docker-compose.yml file

<https://docs.docker.com/compose/>

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

Compose works in all environments; production, staging, development, testing, as well as CI workflows. It also has commands for managing the whole lifecycle of your application:

* Start, stop, and rebuild services
* View the status of running services
* Stream the log output of running services
* Run a one-off command on a service

[Try Docker Compose | Docker Docs](https://docs.docker.com/compose/gettingstarted/)

<https://docs.docker.com/compose/gettingstarted/> 🡪 please follow the document. They give very clearly

#!/bin/bash

# Install docker on Ubuntu

sudo apt-get update

sudo apt-get install \

ca-certificates \

curl \

gnupg \

lsb-release -y

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

echo \

"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu \

$(lsb\_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# Install docker-compose

sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io -y

sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose

# Add ubuntu user into docker group

sudo usermod -a -G docker ubuntu

# Multi Stage Dockerfile

## Problem

FROM tomcat:8-jre11

LABEL "Author"="purandhar"

LABEL "project"="vprofile"

RUN apt update && apt install openjdk-8-jdk -y && apt install maven -y

RUN git clone -b vp-rem https://github.com/devopshydclub/vprofile-project.git && cd vprofile-project && mvn install

RUN rm -rf /usr/local/tomcat/webapps/\*

COPY target/vprofile-v2.war /usr/local/tomcat/webapps/ROOT.war

EXPOSE 8080

CMD ["cataline.sh","run"]

WORKDIR /usr/local/tomcat

VOLUME /usr/local/tomcat/webapps

1. The issue with this is that the Tomcat image includes a repository, adding to the storage.
2. Additionally, running "MVN install" will download many dependencies into the image.
3. The "target" directory contains more than just the artifact, including unnecessary elements.
4. Many unnecessary things are present in this image; we are only interested in the artifact.
5. If we build within the same image, it will become very large in size (in gigabytes), making it difficult to ship.
6. While it's possible to ship, a smaller Docker image size is preferred for efficiency.
7. Some might suggest reverting to the previous method of building the artifact separately on the host machine and then copying it.

But we can’t goto previous method and we can’t do manually so overcome this we are using Multistage Dockerfile

# Multistage Dockerfile

FROM openjdk:8 AS BUILD\_IMAGE

RUN apt update && apt install maven -y

RUN git clone -b vp-docker https://github.com/imranvisualpath/vprofile-repo.git

RUN cd vprofile-repo && mvn install

FROM tomcat:8-jre11

RUN rm -rf /usr/local/tomcat/webapps/\*

COPY --from=BUILD\_IMAGE vprofile-repo/target/vprofile-v2.war /usr/local/tomcat/webapps/ROOT.war

EXPOSE 8080

CMD ["catalina.sh", "run"]

cd git clone -b docker https://github.com/devopshydclub/vprofile-project.git

cd vprofile-project/Docker-files/app/

ls

cd multistage/

ls

cat Dockerfile

docker build -t appimg:v1 .

docker images

# Containerization

## Project of containerization of application

### SCENARIO

* Multi Tier Application Stack
* Running on VM’S
* Regular Deployment
* Continuous Changes

### PROBLEM

* High CapEX & OpEx
* Human Error in Deployment
* Not Compatible with microservice architecture
* Resource Wastage
* Not Portable, Env not in Syncs

### SOLUTION

* Deployment via Images
* Same container images across environment
* Reusable & Repeatable

In containers, deployments are performed using images. If you package your images properly with all the necessary dependencies, binaries, and libraries, and it runs well on your laptop, it will also work in a QA environment. The same holds true for the production environment because we use the same container image across all our environments. This makes our stack reusable, allowing us to use the same stack in multiple environments. It's repeatable, so we can quickly replicate our stack from production to QA or QA to production. So, these are the benefits you gain from containerizing your application. Now, let's look at some statistics.

## STATISTICS - 2020

Containerization Statistics

Half of their application 🡪 50 %

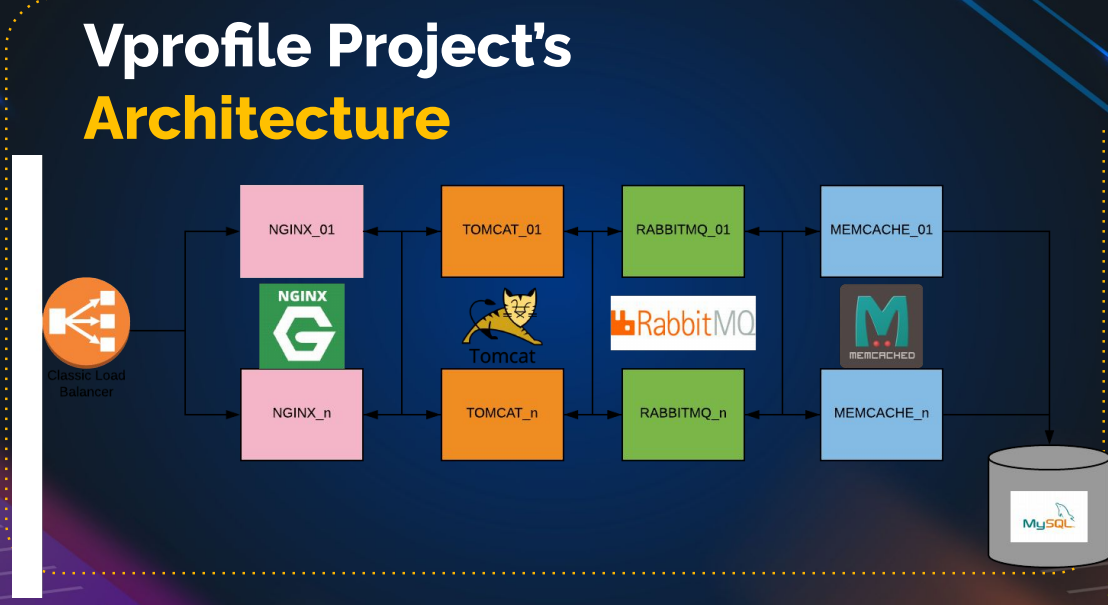
Of them running in production 🡪 29%

AWS Platform 🡪 78%

Managed by DevOps 🡪 81%

## TOOLS

Docker 🡪 container runtime environment



## Steps to setup vprofile stack

* Steps to setup our stack services
* Find right base image from dockerhub
* Write Dockerfile to customize images
* Write docker-compose.yml file to run multicontainers
* Test it & host images on dockerhub

# Architectural design

Docker push to dockerhub(customized docker images)

Docker-compose

docker build command

Dockerfile

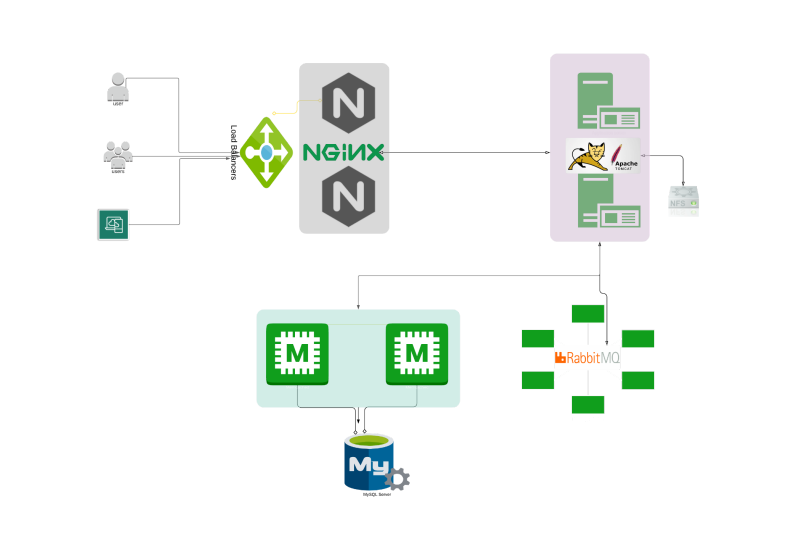
Git repository

Pulled the base images like apache tomcat, mysql,nginx from dockerhub

# Overview of Base Image

[VprofileProjectSetupWindowsAndMacIntel.pdf](file:///C:\Users\puran\OneDrive\Desktop\devOps%20learning\vprofile-project\vagrant\Manual_provisioning_WinMacIntel\VprofileProjectSetupWindowsAndMacIntel.pdf)

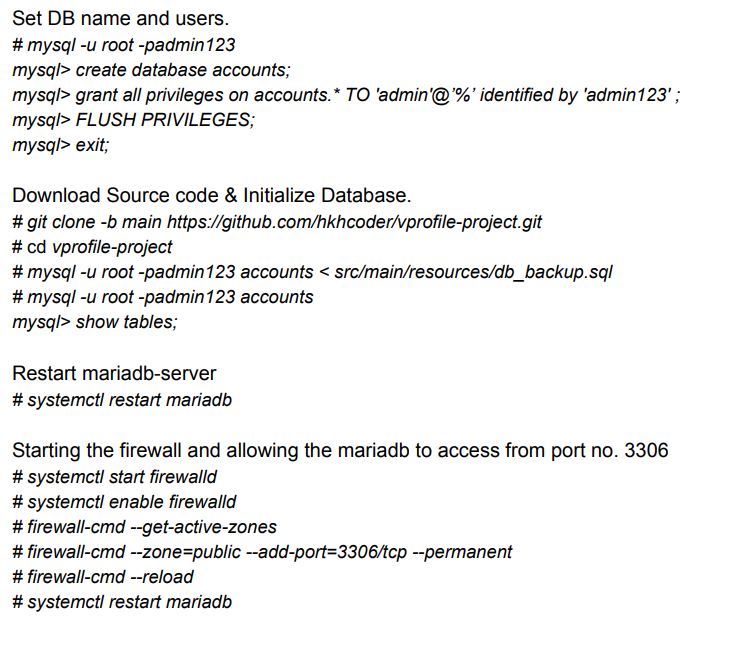
We already setup the vprofile project locally🡪 now is the same setup project stack to build the project using docker containers



### Mysql

<https://hub.docker.com/_/mysql> 🡪 go though the document what are requirement you need check if you need to customize the image or not , in this case we need to customize because

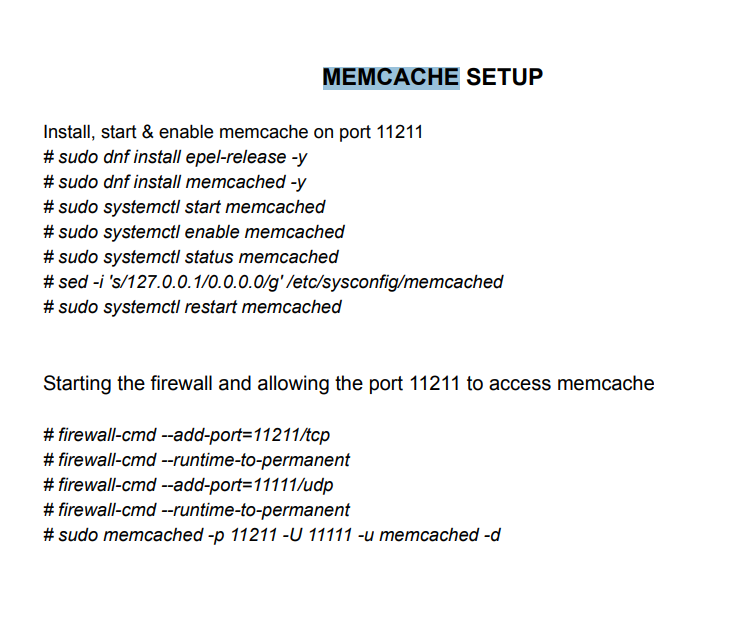
we need to write sql queries from our side to create schema



### MEMCACHE

<https://hub.docker.com/_/memcached>

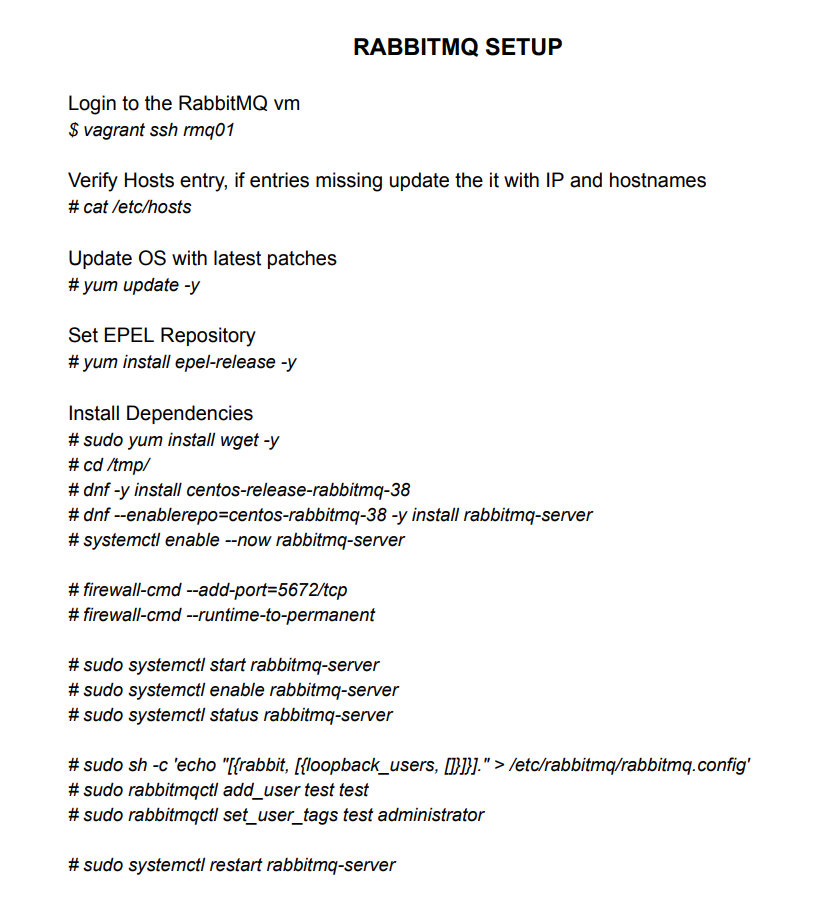
from document no need to customize the image



### RABBITMQ

<https://hub.docker.com/_/rabbitmq>

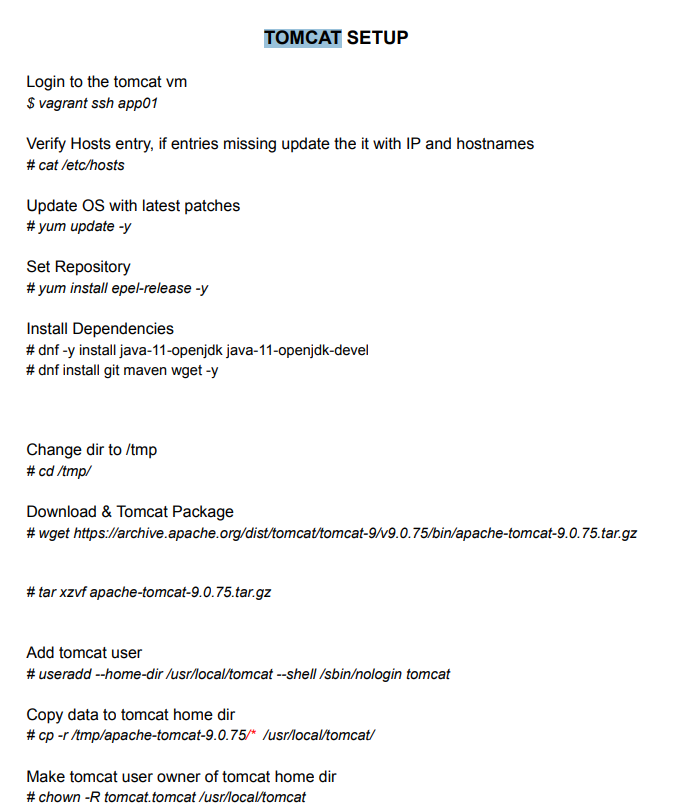
from document no need to customize the image

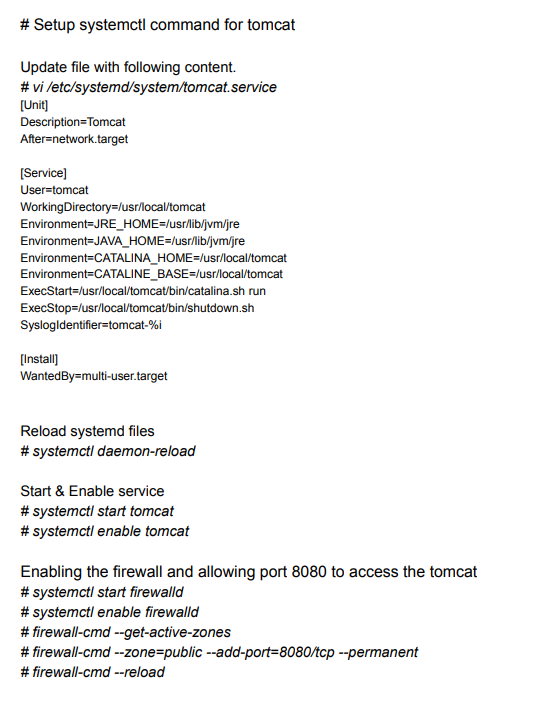


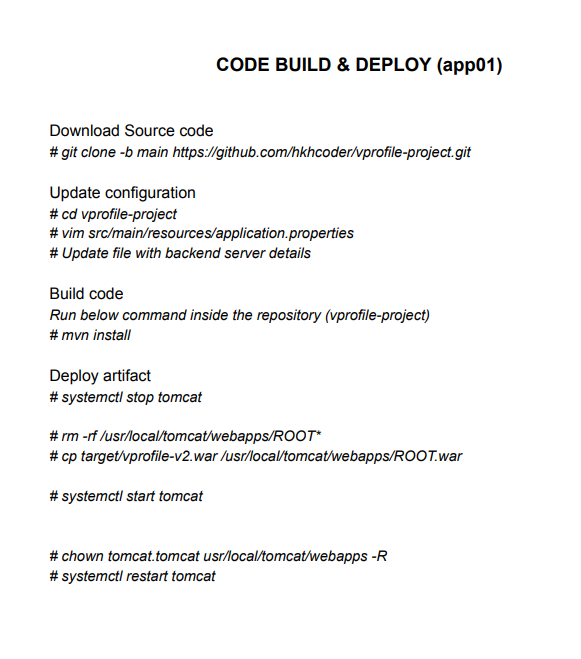
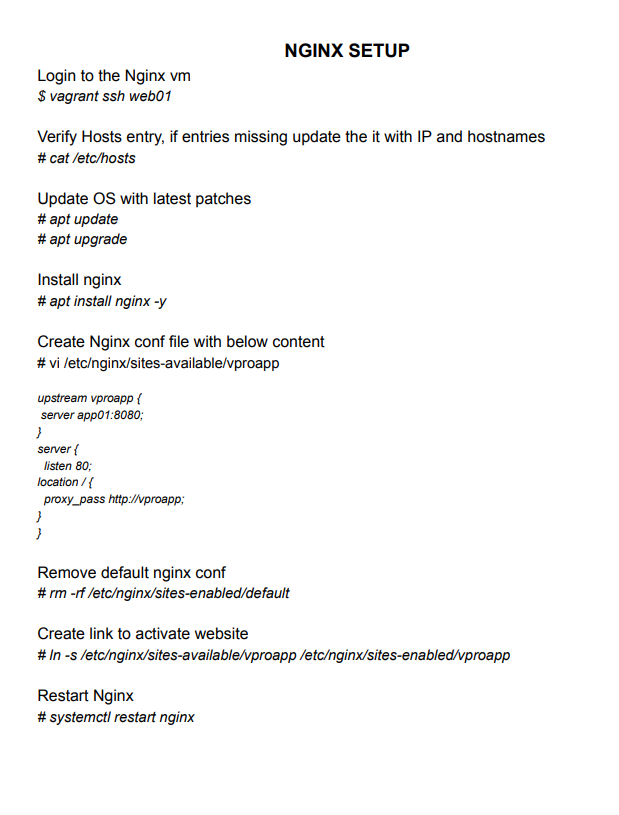
### TOMCAT

<https://hub.docker.com/_/tomcat>

we need to customize the image, because we need to host the project in it





Certainly, here are simplified and clearer sentences:

1. We should start the Nginx service using this configuration.
2. Building may be necessary, or it may not be required.
3. There are two methods to achieve this.
4. When running a container, you can use your configuration as a volume.
5. Alternatively, you can create an image with this configuration to have it consistently available in the image.
6. Choose the method that best suits your needs.
7. We will also create an Nginx image during this process.

# Dockerhub setup – Organization

* Please don’t waste time to read (if you have time you can you,

In short it use to manage containers , alart , mail trigger 🡪 it is not free now (company always maintain this ) to create private repository – access though organizations include collaborative work.

Okay, now that we've decided which images to create, we'll be hosting them on Docker Hub. If you don't have a Docker Hub account, sign up, create one, and log in. Once you're logged in, you can explore the concept of organizations. Creating an organization is not mandatory, but it used to be free when I recorded this lecture, which is no longer the case. However, you can simply watch the tutorial about creating an organization to understand its benefits. In real-life scenarios, companies use organizations to collaborate on multiple development and DevOps tasks for Docker image building. Take a look at how organizations can be created and their advantages, but it's not necessary to do it yourself. Feel free to skip this part.

We're going to create an organization so that we can work together as a team. I'll choose the free team option for our project. Now, for the organization namespace, we'll use 'vprofile,' and for the company name, I'll enter 'visualpath' and select 'create organization.'

The benefits of organizations include collaborative work, with up to three members in a free account. You can have repositories with the organization's name, create different teams, and link your GitHub or Bitbucket accounts. Docker Hub also provides integration to automatically fetch your Docker file and build the image when there's a comment, saving you the trouble of setting up your own continuous integration pipeline. You can set the privacy of your repositories to public or private, but note that with a free account, you can't have multiple private repositories. Additionally, you can receive notifications of build failures through email or Slack. That's it! I've created an organization, and you can do the same with your preferred name."

Certainly! Here's a breakdown of the text point by point:

1. **Decisions on Image Building**: The first point mentions that some decisions have been made about which images to build.
2. **Hosting on Docker Hub**: After making these decisions, the images will be hosted on Docker Hub. Docker Hub is a platform for storing and sharing containerized applications.
3. **Docker Hub Account**: If you don't have a Docker Hub account yet, you need to sign up and create one. This account will be used to manage your images.
4. **Logging In**: Once you have your Docker Hub account, you should log in to it.
5. **Exploring Organizations**: Within Docker Hub, there is a feature called "organizations." The text suggests exploring this feature.
6. **Creating an Organization (Optional)**: The text explains that creating an organization is not mandatory. It used to be free, but now there might be a cost associated with it.
7. **Benefits of Organizations**: The benefits of creating an organization are then outlined:
   * Collaboration: Organizations facilitate collaboration among multiple team members for tasks like Docker image building.
   * Members: A free account can have up to three members.
   * Repositories: You can have repositories named after the organization, not your individual account.
   * Teams: You can create different teams within the organization.
   * Linking Accounts: You can link your GitHub or Bitbucket accounts with your Docker Hub organization.
   * Automation: Docker Hub can automatically build images when a comment is made, which can save you from setting up your own continuous integration pipeline.
   * Privacy: You can set the privacy of your repositories to public or private. Private repositories are limited for free accounts.
   * Notifications: You can receive notifications of build failures via email or Slack.
8. **Organization Creation**: The text goes on to describe how to create an organization. In this case, it creates an organization with the name "vprofile."
9. **Encouragement to Create an Organization**: The text suggests that you should create an organization, but it's ultimately your choice.
10. **Summary**: The text wraps up by emphasizing that an organization has been created, and you're encouraged to create one with your preferred name if you want to explore its benefits.

These points explain the main ideas and steps discussed in the text, making it easier to understand and follow the process of working with Docker Hub and organizations.

# Setup Docker Engine

Create a vagrant vm

mkdir docker-engine

cd docker-engine

vagrant init bento/ubuntu-22.04

vim Vagrantfile

change config file :

config.vm.network “private\_network”, ip: “192.168.56.38”

config.vm.network “public\_network”

config.vm.provider “virtualbox” do |vb|

vb.memory=”2048”

end

* vagrant up
* install the docker engine from the document (it will change the step if any new release is came
* <https://docs.docker.com/engine/install/ubuntu/> 🡪 it is for ubuntu

if you want to run the docker command with vagrant user then add vagrant in docker group

usermod -aG docker Vagrant 🡪 prefer this run from root user

or

vim etc/group 🡪 add the vagrant in docker group

logout and login again

run the docker image in vagrant user

# Dockerhub & Dockerfile references

For the Dockerfile , fork the irom-vprofile-project repo (so that we can make changes)

----------------------------------------------------------------------------------------------------------------------------

## Fork – not important

In GitHub, a "fork" is a feature that allows you to create a copy of someone else's repository (a project) into your own GitHub account. Forking is a fundamental concept in collaborative software development and has several important use cases:

1. **Contributing to Open Source Projects:** When you find an open-source project on GitHub that you want to contribute to, you typically start by forking the repository. This creates a copy of the project in your GitHub account that you have full control over. You can make changes to your forked repository without affecting the original project.
2. **Making Changes:** After forking, you can make changes to the code in your forked repository. This is often done to fix bugs, add new features, or customize the project to your needs.
3. **Pull Requests:** Once you've made changes in your forked repository and want to contribute those changes back to the original project, you create a pull request. A pull request is a request to the original project's maintainers to consider your changes and merge them into the original repository. The maintainers can review your code and decide whether to accept your changes or suggest improvements.
4. **Collaboration:** Forking also allows multiple contributors to work on the same project simultaneously. Each contributor can have their own fork, make changes independently, and submit pull requests to the original project. This enables a collaborative and distributed approach to software development.
5. **Backups and Experimentation:** Forking is not just for contributing to other projects. You can also use it to create a backup of your own repositories or experiment with changes to a project without altering the original codebase.
6. **Version Control and Tracking:** Forking maintains a clear separation between the original project and your copy. It helps you keep track of your changes and contributions separately from the original codebase.

Remember that when you fork a repository, you are essentially cloning it to your account, so any changes you make in your fork won't directly affect the original project until you submit a pull request and the maintainers of the original project choose to merge your changes.

In summary, forking in GitHub is a way to create a copy of a repository for the purpose of contributing to or collaborating on a project, making changes, and maintaining a clear separation between the original and your own version.

Get the repo to vs code 🡪 you can create/make changes to dockerfile

Create a docker hub account and create a repository , in any companies it on private repository

<https://hub.docker.com/repositories/gpurandhar>

vprofileweb , vprofiledb and vprofileapp

# Dockerfile references

<https://docs.docker.com/engine/reference/builder/>

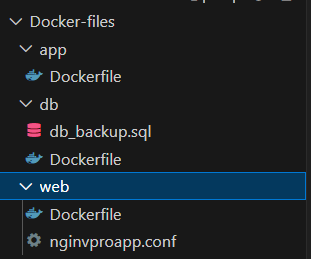
create the all the content clearly

1. A Dockerfile consists of instructions.
2. We choose a base image from these instructions.
3. Our base images are Tomcat, MySQL, and nGenx.
4. We just need to determine the right tag for the base image.
5. In a Dockerfile, we copy files from our machine to the container's directory.
6. "Run" executes commands in a Dockerfile; we use Windows commands for Windows containers and Linux commands for Linux containers.
7. Docker instructions include "Add," "Copy," "ENV," "Expose," "From," "Label," "Stop Signal," "User," "Volume," and "Work Directory."
8. We use these instructions to build Docker images.
9. The "CMD" instruction specifies the command to run in the container.
10. "Label" in a Dockerfile is like tags in AWS, using key-value pairs.
11. "Expose" is used to define ports for running services.
12. "ENV" sets environment variables.
13. "Add" and "Copy" are used to move files into the container, with "Add" offering more options.
14. "Entry Point" is a command that runs when the container is started and can be overridden.
15. Practical examples can help you understand Dockerfile concepts better.

For instance, a common example is creating a Dockerfile to fetch a Debian image, install Apache, expose ports, define volumes, and set an entry point with a specific command.

# App image Dockerfiles

Always try to maintain proper structure



In the app images , we need to build it. We use MVN , it will install lot of dependences then image size become huge(open-jdk, maven and it dependences)

To avoid this , we have a standard practice 🡪 multi-stage dockerfile

* One image to building the build artifact
* Other image will copy the artifact from build artifact to own image
* FROM openjdk:11 AS Build\_image
* FROM tomcat:9-jre11

Openjdk image to build the artifact 🡪 tomcat to have the artifact

## Note:

Before selecting the base images , you need to do lot of research/experiment in <https://hub.docker.com/search?q=>

to find the right image for you. 🡪 you have spent a lot of time here

## dockerfile

FROM openjdk:11 AS Build\_image

RUN apt update && apt install maven -y

RUN git clone https://github.com/devopshydclub/vprofile-project

RUN cd vprofile-project && git checkout docker && mvn install

FROM tomcat:9-jre11

RUN rm -rf /usr/local/tomcat/webapp/\*

COPY --from=Build\_image vprofile-project/target/vprofile-v2.war /usr/local/tomcat/webapp/ROOT.war

CMD ["catalina.sh", "run"]

EXPOSE 8080

# this enough --> you can give work directory volume etc

# DB Image Dockerfile

## db\_backup.sql

-- MySQL dump 10.13  Distrib 5.7.18, for Linux (x86\_64)

--

-- Host: localhost    Database: accounts

-- ------------------------------------------------------

-- Server version 5.7.18-0ubuntu0.16.10.1

/\*!40101 SET @OLD\_CHARACTER\_SET\_CLIENT=@@CHARACTER\_SET\_CLIENT \*/;

/\*!40101 SET @OLD\_CHARACTER\_SET\_RESULTS=@@CHARACTER\_SET\_RESULTS \*/;

/\*!40101 SET @OLD\_COLLATION\_CONNECTION=@@COLLATION\_CONNECTION \*/;

/\*!40101 SET NAMES utf8 \*/;

/\*!40103 SET @OLD\_TIME\_ZONE=@@TIME\_ZONE \*/;

/\*!40103 SET TIME\_ZONE='+00:00' \*/;

/\*!40014 SET @OLD\_UNIQUE\_CHECKS=@@UNIQUE\_CHECKS, UNIQUE\_CHECKS=0 \*/;

/\*!40014 SET @OLD\_FOREIGN\_KEY\_CHECKS=@@FOREIGN\_KEY\_CHECKS, FOREIGN\_KEY\_CHECKS=0 \*/;

/\*!40101 SET @OLD\_SQL\_MODE=@@SQL\_MODE, SQL\_MODE='NO\_AUTO\_VALUE\_ON\_ZERO' \*/;

/\*!40111 SET @OLD\_SQL\_NOTES=@@SQL\_NOTES, SQL\_NOTES=0 \*/;

--

-- Table structure for table `role`

--

DROP TABLE IF EXISTS `role`;

/\*!40101 SET @saved\_cs\_client     = @@character\_set\_client \*/;

/\*!40101 SET character\_set\_client = utf8 \*/;

CREATE TABLE `role` (

  `id` int(11) NOT NULL AUTO\_INCREMENT,

  `name` varchar(45) DEFAULT NULL,

  PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=2 DEFAULT CHARSET=utf8;

/\*!40101 SET character\_set\_client = @saved\_cs\_client \*/;

--

-- Dumping data for table `role`

--

LOCK TABLES `role` WRITE;

/\*!40000 ALTER TABLE `role` DISABLE KEYS \*/;

INSERT INTO `role` VALUES (1,'ROLE\_USER');

/\*!40000 ALTER TABLE `role` ENABLE KEYS \*/;

UNLOCK TABLES;

--

-- Table structure for table `user`

--

DROP TABLE IF EXISTS `user`;

/\*!40101 SET @saved\_cs\_client     = @@character\_set\_client \*/;

/\*!40101 SET character\_set\_client = utf8 \*/;

CREATE TABLE `user` (

  `id` int(11) NOT NULL AUTO\_INCREMENT,

  `username` varchar(255) DEFAULT NULL,

  `userEmail` varchar(255) DEFAULT NULL,

  `profileImg` varchar(255) DEFAULT NULL,

  `profileImgPath` varchar(255) DEFAULT NULL,

  `dateOfBirth` varchar(255) DEFAULT NULL,

  `fatherName` varchar(255) DEFAULT NULL,

  `motherName` varchar(255) DEFAULT NULL,

  `gender` varchar(255) DEFAULT NULL,

  `maritalStatus` varchar(255) DEFAULT NULL,

  `permanentAddress` varchar(255) DEFAULT NULL,

  `tempAddress` varchar(255) DEFAULT NULL,

  `primaryOccupation` varchar(255) DEFAULT NULL,

  `secondaryOccupation` varchar(255) DEFAULT NULL,

  `skills` varchar(255) DEFAULT NULL,

  `phoneNumber` varchar(255) DEFAULT NULL,

  `secondaryPhoneNumber` varchar(255) DEFAULT NULL,

  `nationality` varchar(255) DEFAULT NULL,

  `language` varchar(255) DEFAULT NULL,

  `workingExperience` varchar(255) DEFAULT NULL,

  `password` varchar(255) DEFAULT NULL,

  PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=14 DEFAULT CHARSET=utf8;

/\*!40101 SET character\_set\_client = @saved\_cs\_client \*/;

--

-- Dumping data for table `user`

--

LOCK TABLES `user` WRITE;

/\*!40000 ALTER TABLE `user` DISABLE KEYS \*/;

INSERT INTO `user` VALUES (7,'admin\_vp','admin@visualpathit.com',NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,'$2a$11$0a7VdTr4rfCQqtsvpng6GuJnzUmQ7gZiHXgzGPgm5hkRa3avXgBLK'),(8,'WahidKhan','wahid.khan74@gmail.com',NULL,NULL,'28/03/1994','M Khan','R Khan','male','unMarried','Ameerpet,Hyderabad','Ameerpet,Hyderabad','Software Engineer','Software Engineer','Java HTML CSS ','8888888888','8888888888','Indian','english','2 ','$2a$11$UgG9TkHcgl02LxlqxRHYhOf7Xv4CxFmFEgS0FpUdk42OeslI.6JAW'),(9,'Gayatri','gayatri@gmail.com',NULL,NULL,'20/06/1993','K','L','male','unMarried','Ameerpet,Hyderabad','Ameerpet,Hyderabad','Software Engineer','Software Engineer','Java HTML CSS ','9999999999','9999999999','India','english','5','$2a$11$gwvsvUrFU.YirMM1Yb7NweFudLUM91AzH5BDFnhkNzfzpjG.FplYO'),(10,'WahidKhan2','wahid.khan741@gmail.com',NULL,NULL,'28/03/1994','M Khan','R Khan','male','unMarried','Ameerpet,Hyderabad','Ameerpet,Hyderabad','Software Engineer','Software Engineer','Java HTML CSS ','7777777777','777777777','India','english','7','$2a$11$6oZEgfGGQAH23EaXLVZ2WOSKxcEJFnBSw2N2aghab0s2kcxSQwjhC'),(11,'KiranKumar','kiran@gmail.com',NULL,NULL,'8/12/1993','K K','RK','male','unMarried','California','James Street','Software Engineer','Software Engineer','Java HTML CSS ','1010101010','1010101010','India','english','10','$2a$11$EXwpna1MlFFlKW5ut1iVi.AoeIulkPPmcOHFO8pOoQt1IYU9COU0m'),(12,'Saikumar','sai@gmail.com',NULL,NULL,'20/06/1993','Sai RK','Sai AK','male','unMarried','California','US','Software Engineer','Software Engineer','Java HTML CSS AWS','8888888111','8888888111','India','english','8','$2a$11$pzWNzzR.HUkHzz2zhAgqOeCl0WaTgY33NxxJ7n0l.rnEqjB9JO7vy'),(13,'RamSai','ram@gmail.com',NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,'$2a$11$6BSmYPrT8I8b9yHmx.uTRu/QxnQM2vhZYQa8mR33aReWA4WFihyGK');

/\*!40000 ALTER TABLE `user` ENABLE KEYS \*/;

UNLOCK TABLES;

--

-- Table structure for table `user\_role`

--

DROP TABLE IF EXISTS `user\_role`;

/\*!40101 SET @saved\_cs\_client     = @@character\_set\_client \*/;

/\*!40101 SET character\_set\_client = utf8 \*/;

CREATE TABLE `user\_role` (

  `user\_id` int(11) NOT NULL,

  `role\_id` int(11) NOT NULL,

  PRIMARY KEY (`user\_id`,`role\_id`),

  KEY `fk\_user\_role\_roleid\_idx` (`role\_id`),

  CONSTRAINT `fk\_user\_role\_roleid` FOREIGN KEY (`role\_id`) REFERENCES `role` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,

  CONSTRAINT `fk\_user\_role\_userid` FOREIGN KEY (`user\_id`) REFERENCES `user` (`id`) ON DELETE CASCADE ON UPDATE CASCADE

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

/\*!40101 SET character\_set\_client = @saved\_cs\_client \*/;

--

-- Dumping data for table `user\_role`

--

LOCK TABLES `user\_role` WRITE;

/\*!40000 ALTER TABLE `user\_role` DISABLE KEYS \*/;

INSERT INTO `user\_role` VALUES (4,1),(5,1),(6,1),(7,1),(8,1),(9,1),(10,1),(11,1),(12,1),(13,1);

/\*!40000 ALTER TABLE `user\_role` ENABLE KEYS \*/;

UNLOCK TABLES;

/\*!40103 SET TIME\_ZONE=@OLD\_TIME\_ZONE \*/;

/\*!40101 SET SQL\_MODE=@OLD\_SQL\_MODE \*/;

/\*!40014 SET FOREIGN\_KEY\_CHECKS=@OLD\_FOREIGN\_KEY\_CHECKS \*/;

/\*!40014 SET UNIQUE\_CHECKS=@OLD\_UNIQUE\_CHECKS \*/;

/\*!40101 SET CHARACTER\_SET\_CLIENT=@OLD\_CHARACTER\_SET\_CLIENT \*/;

/\*!40101 SET CHARACTER\_SET\_RESULTS=@OLD\_CHARACTER\_SET\_RESULTS \*/;

/\*!40101 SET COLLATION\_CONNECTION=@OLD\_COLLATION\_CONNECTION \*/;

/\*!40111 SET SQL\_NOTES=@OLD\_SQL\_NOTES \*/;

-- Dump completed on 2017-12-07 16:32:31

## dockerfile

FROM MYSQL:8.0.33

LABEL "PROJECT"="vprofile"

LABEL "Author"="Imron"

ENV MYSQL\_ROOT\_PASSWORD="vprodbpass"

ENV MYSQL\_DATABASE="account"

ADD db\_backup.sql /docker-entrypoint-initdb.d/db\_backup.sql

# Web image dockerfile

## DOCKERFILE

FROM nginx

LABEL "PROJECT"="vprofile"

LABEL "Author"="Imron"

RUN rm -rf /etc/nginx/conf.d/default.conf

COPY nginxvproapp /etc/nginx/conf.d/default.conf/vproapp.conf

## Nginxvproapp.conf

upstream vproapp {

 server vproapp:8080;

}

server {

  listen 80;

location / {

  proxy\_pass http://vproapp;

}

}

# Docker-compose

<https://docs.docker.com/compose/features-uses/>

Instructor: We now have all three images ready, and we can proceed to build and test them. To simplify these tasks, we'll use Docker Compose. First, ensure you have a docker-compose.yml file. While learning Docker Compose, it's best to dive into it hands-on, but let's start by understanding the basics.

Docker Compose is like Vagrant for VMs, but for Docker containers. Just as you can manage multiple VMs with Vagrant, Docker Compose allows you to work with multiple containers, even building and running them together. Let's explore a quick example.

To begin, create a folder and add a Python file, though Python isn't directly related to Compose. You'll also need a requirements.txt file and a Dockerfile, specific to your project. In the Compose file, we define services, which represent our containers. For example, we have a container named "web" and another named "redis."

When you run "Docker Compose build," it finds the Dockerfile and builds it. To run a container, use "Docker Compose up." This offers flexibility in specifying port numbers, volumes, and environment variables. To run everything in the background, use "Docker Compose up -d."

Now, let's return to VS Code and start writing our Docker Compose file. We'll reference the example we just discussed.

Specify the version as "3.8," create services (containers), and configure each one. For instance, the "vprodb" container builds from the Dockerfile located in the "./Dockerfiles/db" directory, uses port 3306, and sets up a volume. We need to declare this volume within the Compose file.

For environment variables like "MYSQL\_ROOT\_PASSWORD," reference your application's requirements. It should match the application's configuration, ensuring it connects properly. Create containers for other services like "memcache" and "rabbitmq" in a similar manner.

For your main app image, replicate the structure for the "vprodb" container. This approach streamlines the process and simplifies writing the Compose file. Finally, don't forget to create the necessary volumes for data persistence, such as "vproappdata" and "vprodbdata."

These volumes will be created when you execute "Docker Compose up." Verify the correctness of names, spacing, and indentation to avoid typographical errors. Save your work before testing. If everything works, you can commit your changes and consider pushing them to your repository.

version: '3.8' # version

services: # containers

  vprodb: # container

    build: # build . (dot means it is expecting # Dockerfile in current working directory

      context: ./Docker-files/db #

    image: gpurandhar/vprofiledb # it should be same name in dockerhub # account

    container\_name: vprodb #

    ports:

      - "3306:3306" # it running on the port 3306 expose o # on 3306

    volumes:

      - vprodbdata:/var/lib/mysql # need to mount the volume(real world)🡪 this folder :/var/lib/mysql should) mount to vprodbdata (we can declare the volume in compose file itself

    environment:

      - MYSQL\_ROOT\_PASSWORD=vprodbpass

  vprocache01: # database service Memcached # rabbitmq . mention in image and port

    image: memcached

    ports:

      - "11211:15672"

  vpromq01:

    image: rabbitmq

    ports:

      - "15672:15672"

    environment:

      - RABBITMQ\_DEFAULT\_USER=guest

      - RABBITMQ\_DEFAULT\_PASS=guest

  vproapp: # Application container

    build:

      context: ./Docker-files/app

    image: gpurandhar/vprofileapp

    container\_name: vproapp

    ports:

      - "8080:8080"

    volumes:

      - vprodbdata:/var/local/tomcat/webapps

  vproweb: # web container

    build:

      context: ./Docker-files/web

    image: gpurandhar/vprofileweb

    container\_name: vproweb

    ports:

      - "80:80"

volumes: # volume mount declaration

  vprodbdata: {} #it will be same line as containers

  vproappdata: {}

So, this files will be,(src🡪 main🡪resources🡪application.properties

this file will be in our artifact, Tomcat artifact, sorry,

the vprofile artifact will be in the Tomcat,

and that application, vprofile application,

will connect to the backend service like MySQL

1. Services and volumes are listed together. In this case, "volumes" indicates the start of a new section.
2. Under "vprodbdata," we use curly braces to create a volume within Docker Engine itself.
3. Moving on to "vproappdata," when you run "Docker Compose up," it initially creates these volumes. Then, as it builds the containers, it links them to these volumes when specified like this.

# BUILD & RUN

Don’t care below lines

Instructor: I've saved all these files or this repository in the same container folder named "docker-engine" where I created the Vagrant file for the VM. You can see the "vprofile-project" folder, right? If your VM is already running, I recommend rebooting it with the command "vagrant reload." If it's not created yet, go ahead and create it.

Now, let's discuss how to create the VM. If you're using an EC2 instance, copy all your saved data to it. Commit the data to your GitHub repository and then pull it into the EC2 instance or clone it within the EC2 instance. If you're using a VM like me, you've probably already saved it in the VM folder. However, if you saved it elsewhere, just copy that folder to the VM folder.

Next, bring up the VM. If the VM is running, reboot it. Once the VM is up, log in using "vagrant ssh." Become the root user with "sudo -i," clear the screen, and navigate to "cd /vagrant." Inside that, you'll find the repository folder.

Inside the repository folder, you have the "docker-compose.yml" file and the Docker files. Check the "docker-compose.yml" file for correct indentation.

Now, it's time to test. Use the command "docker-compose build" to build all the images specified in the docker-compose file. This command will read your docker-compose file and start building the images from the specified Dockerfiles. It may take some time. You can pause the screen and resume when it's done.

After building the images, check them with "docker images." You should see the expected images.

To ensure everything is working, run the containers with "docker-compose up -d." This command will start all the containers defined in the docker-compose file. It will also build the images if they haven't been built yet.

To access the web container that routes to Tomcat, find the IP address of your VM with "ip addr show" and use it in a web browser.

The instructor demonstrates how the containers are working by logging in and checking various components. All containers seem to be validated, showing that the system is functioning correctly.

Now, you can push the images to Docker Hub. To do this, use "docker login" with your Docker Hub username and password. After logging in, use "docker push" to push your images to your Docker Hub account.

For cleanup, stop and remove the containers with "docker-compose down." If you want to clean up even more, use "docker system prune -a" to remove stopped containers, networks, images, and cache build images.

You're also encouraged to commit your source code and push it to your GitHub repository. This step isn't mandatory but is recommended.

In summary, the instructor guides you through the process of containerizing a project using Docker and Docker Compose. He emphasizes the importance of getting setup documentation from the developers and explains that Docker files and Docker Compose files are key components of the containerization process. He encourages you to practice this multiple times to become proficient and, when done, to shut down your instance or VM.

If you have already using VM and made a changes ex: docker images etc the try to reload the **vm or ec2 instances**

Sudo -i

Cd /

Cd vagrant/vprofile-project

Vim docker-compose.yml

**docker compose build 🡪 it will build all the images**

**docker images**

**docker compose up -d 🡪 even if you skip the “docker compose build ” 🡪 it will first build the images and start the containers**

**docker ps**

**ip addr show**

**docker login**

**docker images**

**docker push Gpurandhar/vprofileapp**

**docker push Gpurandhar/vprofiledb**

**docker push Gpurandhar/vprofileweb**

**docker compose ps 🡪 to list the running containars**

**docker compose stop**

**docker compose rm**

**docker compose down 🡪 it used to stop & remove the contianer**

**docker system prune -a 🡪 it’s going to stop the container , remove the networks, images, cache build**

**end of the day 🡪 containerizing is two thing docker file and docker compose**

Right, so whenever you get into the project

the first step is knowing about that project

which you will get to know from the developers

how to build it, how to run it.

Make sure you get the setup document

from the developers, okay?

That will be your starting point, okay?

Get all the details, versions, packages,

what services running, how you're building it,

how you're running it.

# Summarize

Let's review our source code once more.

To start, we initially focused on creating the Docker files, correct?

Now, let's recap by examining the docker-compose file, as it provides a comprehensive view of our containerization project.

If you revisit it, you'll easily grasp the docker-compose file, thanks to the data we've gathered thus far.

This conclusion stems from the information we've collected, which is a crucial first step in any project—gathering information.

You begin by understanding how to construct and deploy the project, working closely with the developer to comprehend the development, deployment, and setup steps of the application.

You also familiarize yourself with the various services used in the project. For example, in the vprofile-project, we employ a database, Memcached, RabbitMQ, Tomcat, and Nginx.

From the outset, you can create a basic structure for a docker-compose file. Define your services and specify where the Dockerfile is located.

Speaking of Dockerfiles, they contain information about how the application is deployed, including the setup steps.

If you need Maven, you can utilize OpenJDK or the Maven image available on Docker Hub. You might want to include additional steps, such as cloning the source code. Alternatively, you can mount the current repository folder into the container image.

To do this, you can use the "ADD" command and specify the folder path. For instance, "dot dot slash dot dot" represents the Docker-files folder, followed by the vprofile-project folder, where your source code resides.

Now, with your source code mounted, you can proceed to build or execute other tasks like switching branches if necessary.

You can even choose to clone the source code if that suits your project requirements.

Remember to keep your build image separate from the app image. After building the artifact, copy it to the app image.

For database images, such as MySQL or Redis, setting them up is usually straightforward. Specify the schemas, usernames, and database names as needed.

Returning to the docker-compose file, ensure you gather information about port numbers, environment variables, and other configuration details.

With the information you've gathered, containerizing additional images will be mostly a matter of copy-pasting and making adjustments.

In the end, you'll primarily work with Docker-compose and Dockerfile. Build and test, then push to Docker Hub. Once that's successful, you can commit your source code.

After preparing your source code, the rest is straightforward. Fetch the source code, run docker-compose up -d, and it will build the image and launch the container.

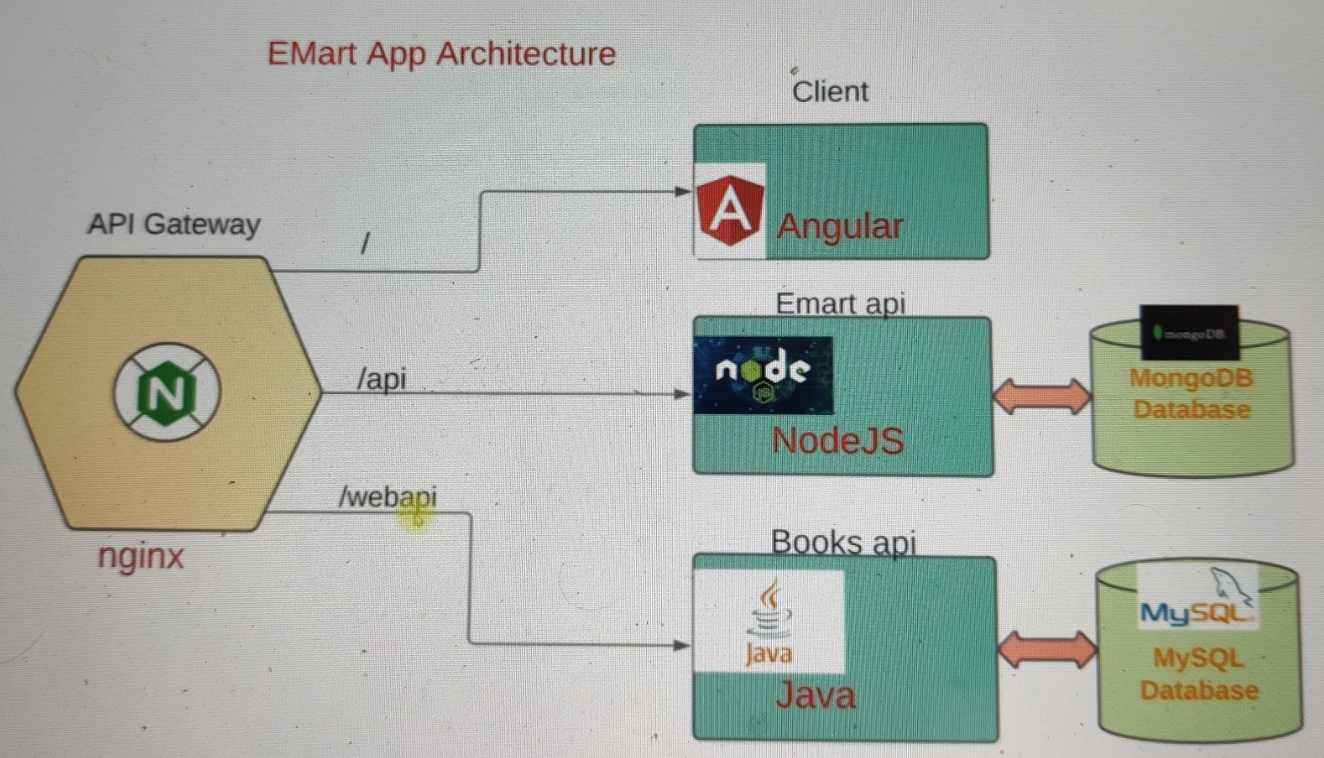
To summarize the steps:

1. Understand the setup steps for your stack on local or remote VMs.
2. Know the setup steps for each service in your stack.
3. Find the appropriate Base image from Docker Hub.
4. Customize your image using a Dockerfile.
5. Use a docker-compose file to manage and test multiple containers.
6. Once satisfied, host the image on Docker Hub.

This process covers what we've done in this project, from cloning the source code to building the Dockerfile, using Docker-compose for testing, and ultimately pushing it to Docker Hub.

I hope this project has provided valuable insights and experience in containerizing your applications. Thank you for watching, and I look forward to the next one.

# Containerizing microservices project



In software development, microservice architecture is highly effective for delivering features quickly and efficiently. This approach offers numerous advantages and is expected to remain prominent. As a DevOps professional, you'll encounter microservice applications and should know how to containerize them. This lecture focuses on that topic.

Previously, we discussed this application, which comprises four services: NGINX (the API gateway), a client microservice (Angular-based frontend), an emart API (NodeJS-based backend), and a books API (Java-based). These services rely on MongoDB and MySQL databases. You can find the source code for this e-commerce application on a public GitHub repository called Emartapp, which is a mono repo containing all the microservices' source code. While this setup can be advantageous, it's also common to split the code into separate repositories for individual services, enabling distinct CI/CD pipelines.

In the future, I'll provide a separate course on GitOps. However, this lecture focuses on containerizing the microservices stack. We'll clone the source code, open it in Visual Studio Code, and explore the Dockerfile and Docker Compose files.

Please note that I am using Visual Studio Code, but you can use your preferred IDE.

To begin, clone the source code, create a directory, and open the repository in Visual Studio Code.

The microservices we're containerizing include a client, an API, web API, NGINX, MongoDB, and MySQL. Docker Compose is essential to manage and run these containers together.

Each service has its own Dockerfile, responsible for building the respective container image. In the Docker Compose file, you define the services, their dependencies, and configurations, ensuring they work harmoniously.

Containerizing microservices involves understanding the build process and hosting method, which varies for different technologies. If you're new to this, collaborate closely with developers to grasp the build processes and hosting details. Be patient, as errors are common when running containers. Troubleshooting, checking logs, and verifying configurations are key skills.

In summary, containerizing microservices involves writing Dockerfiles and Docker Compose files to run multiple containers together. It's essential to understand the build process and hosting method, and collaboration with developers can be beneficial."

This simplifies and condenses the original text while maintaining the key information.

Explain about terminology

1. **Microservice Architecture in Software Development**:
   * Microservice architecture is a beneficial approach in software development.
   * It allows for efficient and quick delivery of features to users.
2. **Advantages of Microservice Architecture**:
   * There are several advantages to using microservice architecture, but we won't detail them here.
3. **Longevity of Microservice Architecture**:
   * Microservice architecture is expected to remain a significant part of software development for a long time.
4. **Relevance for DevOps Professionals**:
   * If you're a DevOps professional, you will likely work with microservice applications.
   * You should understand how to containerize them.
5. **Purpose of the Lecture**:
   * The lecture you're currently engaged in is focused on containerizing microservice applications.
   * This is particularly valuable for DevOps professionals.
6. **Introduction to the Application**:
   * The text describes an application with four main services:
     + NGINX: Serves as the API gateway and receives user requests.
     + Client Microservice: Written in Angular, it handles the frontend of the website.
     + emart API: A NodeJS-based service that manages the back-end data.
     + Books API: A Java-based service that uses a MySQL database.
   * MongoDB is also used as a NoSQL database for certain aspects of the application.
7. **Source Code Repository**:
   * The application's source code is stored in a public GitHub repository named "Emartapp."
8. **Mono Repo vs. Separate Repositories**:
   * Initially, all microservices' source code is in one repository, known as a "mono repo."
   * However, in some cases, it's beneficial to have separate repositories for different services, which simplifies the management of CI/CD pipelines🡪 this promote more GitOps.
9. **GitOps**:
   * The text briefly mentions GitOps, a set of practices for managing infrastructure and applications using Git and automated pipelines.
10. **Containerizing Microservices**:
    * The primary focus of the lecture is on the process of containerizing microservices.
11. **Overview of the Process**:
    * This process involves writing Dockerfiles and Docker Compose files.
12. **Recommendation for Beginners**:
    * For those new to this process, it is suggested to work closely with developers to understand the build processes and hosting methods specific to the technologies used in your microservices.
13. **Key Skills**:
    * Troubleshooting is a crucial skill when working with containers. If issues arise, examining logs and configurations can help identify and resolve problems.
14. **Docker Compose for Managing Containers**:
    * Docker Compose is a tool used to define and run multi-container applications. It simplifies the process of managing multiple containers and their interactions.
15. **Collaboration with Developers**:
    * Effective collaboration with developers is important for understanding the intricacies of building and hosting microservices, as different technologies have different requirements.
16. **Summary**:
    * In summary, the process of containerizing microservices involves writing Dockerfiles for individual services and creating a Docker Compose file to manage and run them together. Understanding the unique build processes and hosting methods for each service is critical, and collaboration with developers can be highly beneficial in this endeavor.

This detailed explanation should make the original text more understandable and accessible.

## repo

[**https://github.com/devopshydclub/emartapp.git**](https://github.com/devopshydclub/emartapp.git)

**mkdir -p /f/microsvc**

**git clone** [**https://github.com/devopshydclub/emartapp.git**](https://github.com/devopshydclub/emartapp.git) **if it not work use ssh url**

**ls**

**cd emartapp.git**

**code . 🡪 it will directly open in vscode**

**please go though emartapp repo 🡪 Docker-compose 🡪 inside follow the dockerfile & services**

# Build & Run Microservice App

**Introduction:**

* The text begins by welcoming you to a lecture where Docker and Docker Compose are being used for managing microservices.

**Setting Up Environment:**

* The lecturer mentions that they will be using a Docker Compose file to build and run multiple microservices together.
* You're given two options for running this setup: on an EC2 instance or a Vagrant VM. The lecturer prefers an EC2 instance due to better performance.

**EC2 Instance Setup:**

* The lecturer provides details about setting up an EC2 instance:
  + The recommended instance type is "t3.medium" due to better performance.
  + There will be some charges, but they will be minimal since you only need the instance for less than an hour.
  + The lecturer names the instance "Docker engine" and specifies that they need an Ubuntu server (version 20) as the operating system.
  + For better performance, the lecturer chooses the "t3.medium" instance type, which has 4 GB of RAM and 2 CPUs. Internet speed will also be better on this instance.
  + They create a key pair called "docker key" for SSH access to the instance.
  + The lecturer mentions that you should allow SSH access from your IP address.
  + Additionally, HTTP access is allowed because the NGINX container (API gateway) will be accessed via a web browser.
  + The lecturer specifies a minimum volume size of 15 GB (or optionally, 20 GB) for storage.
* After configuring these settings, the lecturer explains that a script for installing Docker Engine and Docker Compose will be provided. This script is intended for automation but can be executed manually if needed.

**Logging into the EC2 Instance:**

* The lecturer proceeds to log into the EC2 instance using SSH with the provided key pair and the instance's IP address.

**Environment Verification:**

* After logging in, the lecturer runs an **id** command to verify that the user is in the Docker group. This is crucial for Docker operations.
* The lecturer checks that Docker and Docker Compose are installed and functional on the instance.

**Cloning Repository:**

* The next step is to clone a repository from a Git URL. This repository likely contains the source code for the microservices.

**Building Microservices:**

* The Docker Compose file included in the repository is used for building and running the microservices.
* The lecturer starts by explaining the "build" process, which is time-consuming because it involves multiple microservices.
* A visual interface (e.g., Docker Desktop) can be used to initiate the build process, or you can watch the progress.
* The lecture then pauses and plans to continue after the build process is complete.

**Building Process Completion:**

* After some time, the build process is completed. As a result, images for the microservices are created. These images include client, Web API, and eMart API.

**Running Microservices:**

* To run the microservices, the lecturer uses the command **docker-compose up -d**, which starts the containers in the background.
* Optionally, you can run the containers in the foreground if you want to see their output.

**Accessing the Application:**

* You can access the application through an API gateway (NGINX) by connecting to the instance's IP on port 80. This will load the front end, where you can register and log in.
* The lecturer demonstrates this by opening a web browser, entering the IP address, and accessing the front end.
* It's highlighted that this is a sample application, and all services are functioning correctly.

**Stopping and Cleaning Up:**

* The lecturer stops all containers with **Ctrl+C**, and then runs **docker-compose up -d** to restart them in the background.
* Commands like **docker ps**, **docker-compose ps**, and **docker-compose top** are mentioned as useful for managing containers.
* To clean up the environment, you can use **docker-compose down**, which stops and removes all containers. This is crucial to avoid incurring extra charges.
* The lecturer also mentions that the build process will be faster after the initial setup since Docker caches image layers.

**Conclusion:**

* The lecture concludes by advising you to take your time, get comfortable with the processes, and explore the provided code.
* Once you're done with the environment, you should stop or terminate the EC2 instance to avoid additional charges.

In summary, this text outlines the process of setting up a Docker-based microservices environment on an EC2 instance, from instance creation to application access and clean-up procedures. The lecturer emphasizes the importance of getting comfortable with these steps and automating as much as possible through CI/CD pipelines.

Provision is three steps 🡪 are you can do manually

#!/bin/bash

# Install docker on Ubuntu

sudo apt-get update

sudo apt-get install \

ca-certificates \

curl \

gnupg \

lsb-release -y

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

echo \

"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu \

$(lsb\_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# Install docker-compose

sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io -y

sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose

# Add ubuntu user into docker group

sudo usermod -a -G docker ubuntu

id

docker-compose –version

**git clone** [**https://github.com/devopshydclub/emartapp.git**](https://github.com/devopshydclub/emartapp.git)

**ls**

**docker-compose build**

**docker images**

**docker-compose up -d (# -d is running background, if you want to see the logs then remove -d)**

**if you want to see the logs to individual 🡪 docker logs container id/name**

**get the ip address and run it**