

Basic intro1

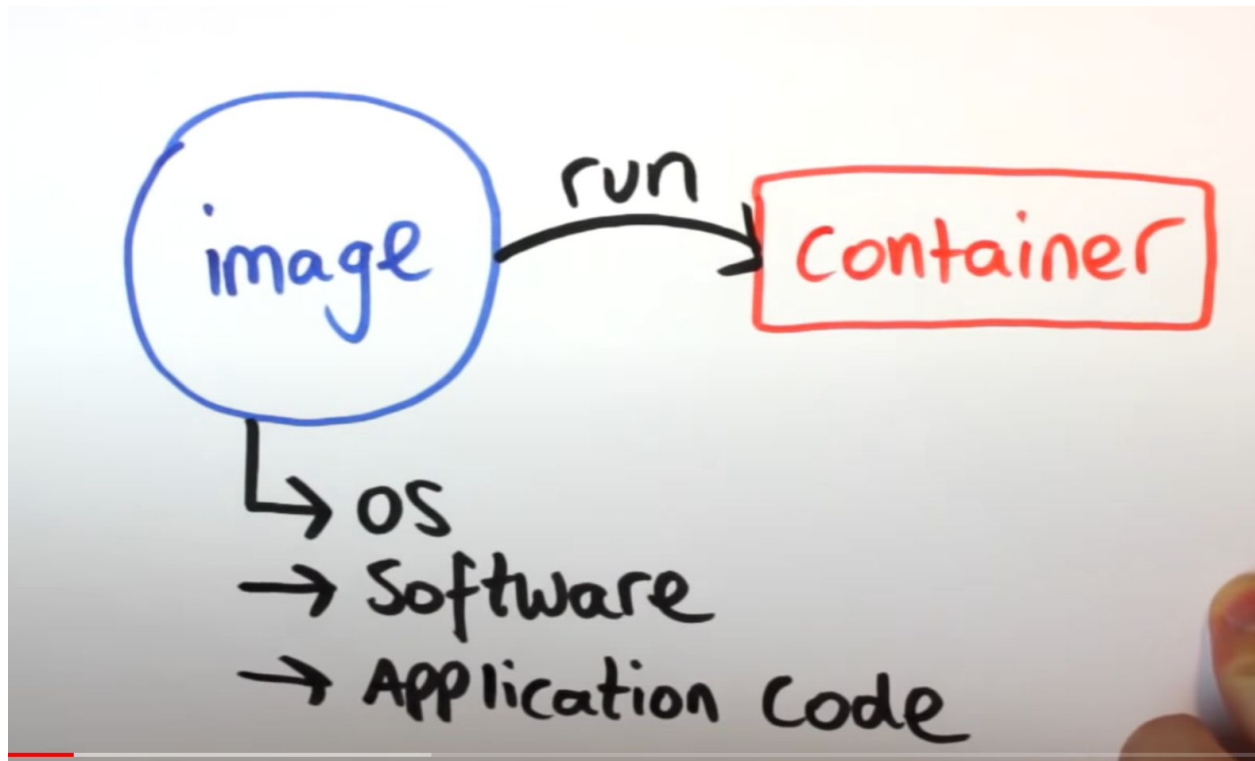
1. What is Docker
2. Virtual Machines vs. Docker
3. Introduction to Dockerfiles, images and containers
4. The Docker Hub
5. Writing a Dockerfile
6. Building an image
7. Running a container
8. Mounting volumes
9. One process per container

REF: <https://youtu.be/YFI2mCHdv24>

CONTAINER: is a running instance of an IMAGE

IMAGE - template for creating environment you wanted to run (its snapshot), a 'package template plan'.

image: has OS, software, application code.

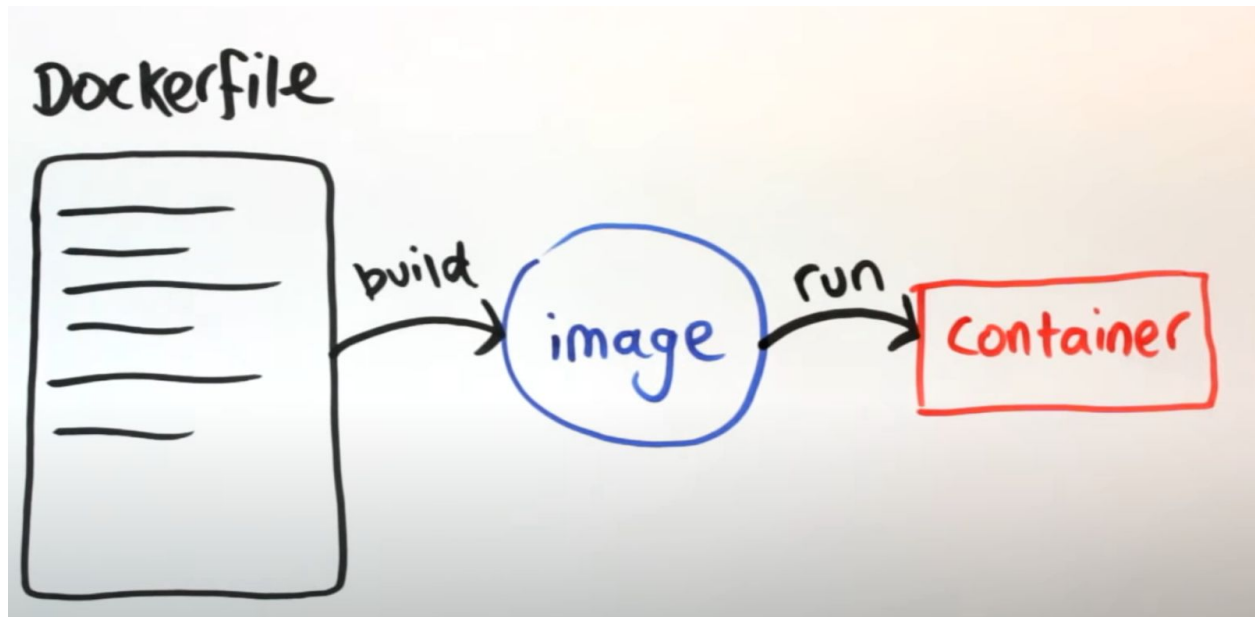


Images are 'defined' using DOCKERFILE (a text file), with a list of steps to perform, to create that image.

Use 'Dockerfile' to build the 'Image', which is run to get 'Container'.

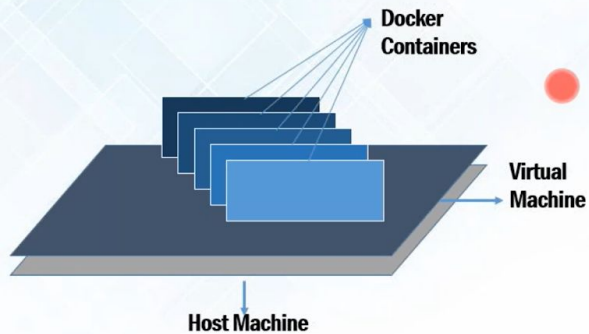
-p for forward port 80 on host (the computer) to port 80 in container ().

-- " docker run -p 80:80 <container_name> "

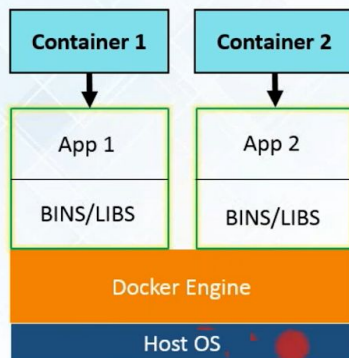


- to reflect changes made: rebuild the image and spin up new container from updated image

You can run several microservices in the same VM by running various Docker containers for each microservice.



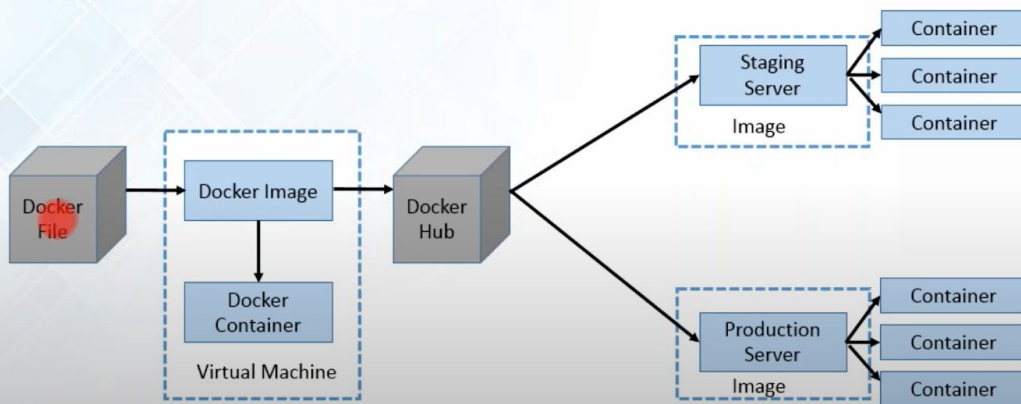
Provides a consistent computing environment throughout the whole SDLC.



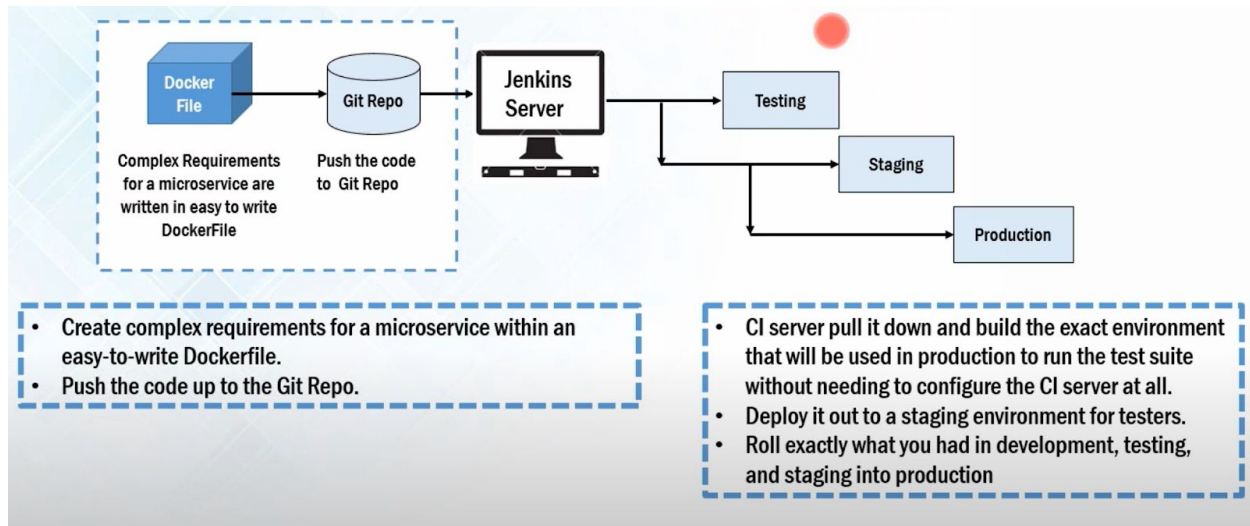
- Docker is a tool designed to make it easier to create, deploy, and run applications by using containers.
- Docker containers are lightweight alternatives to Virtual Machines and it uses the host OS.
- You don't have to pre-allocate any RAM in containers.

docker

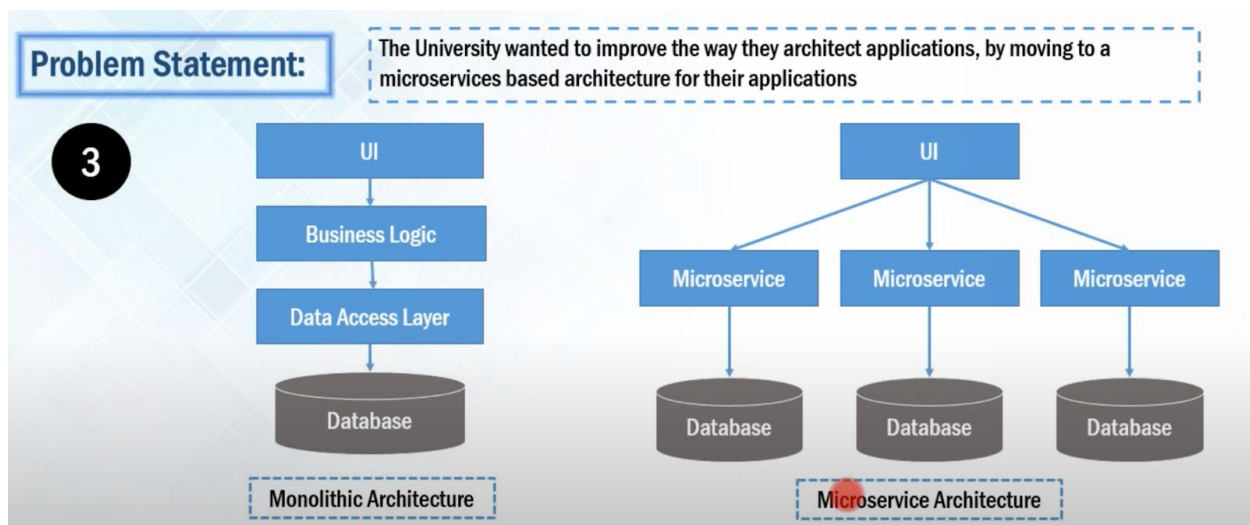
- Docker file builds a Docker image and that image contains all the project's code
- You can run that image to create as many Docker containers as you want
- Then this Image can be uploaded on Docker hub, from Docker hub any one can pull the image and build a container



Another Docker Example:



Case study example:

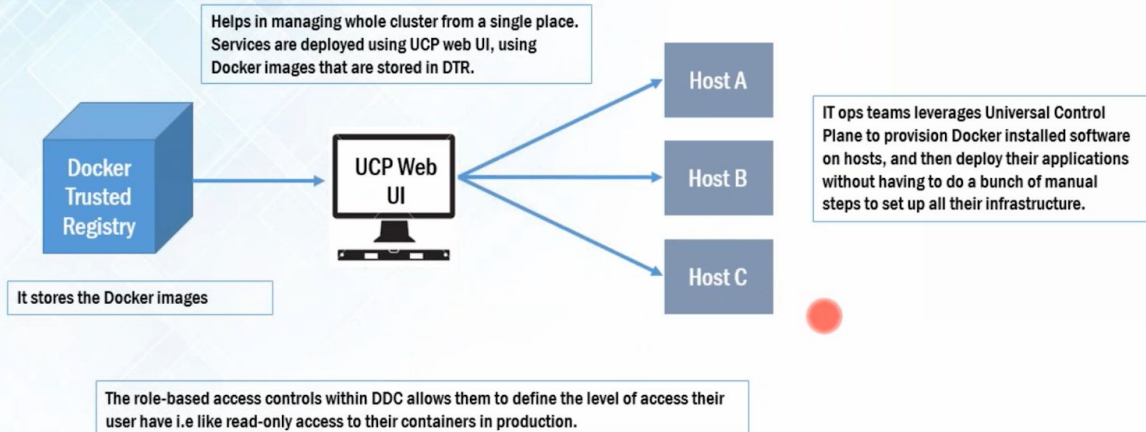


Solution: Docker Data Center (DDC)



Source: <https://www.docker.com/customers/indiana-university-delivers-state-art-it-115000-students-docker-datacenter>

Solution: Docker Data Center (DDC)



Source: <https://www.docker.com/customers/indiana-university-delivers-state-art-it-115000-students-docker-datacenter>


DOCKER COMPONENTS:

Docker Registry:

Docker Registry

Press **Esc** to exit full screen

- Docker Registry is a storage component for Docker Images
- We can store the Images in either Public / Private repositories
- [Docker Hub](#) is Docker's very own cloud repository



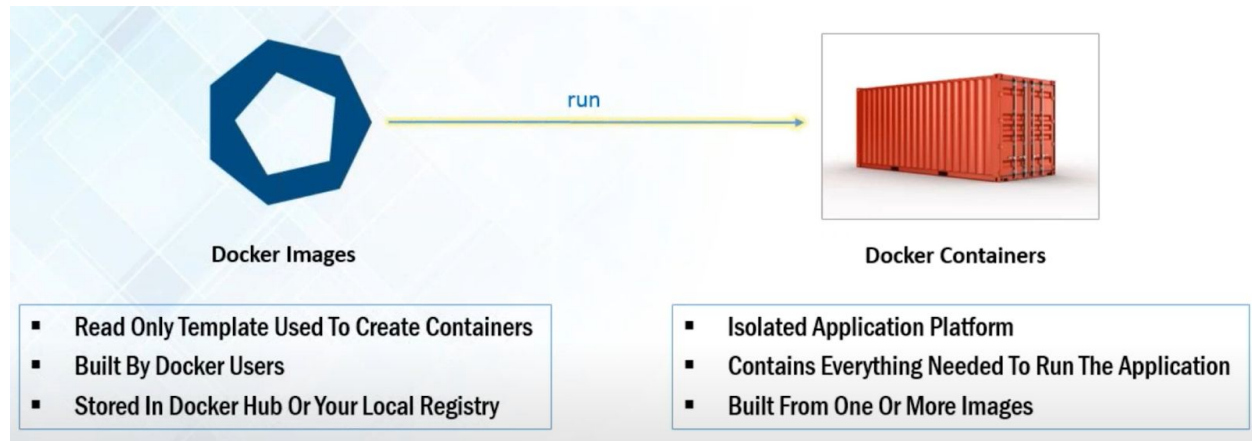
Why Use Docker Registries?

- Control where your images are being stored
- Integrate image storage with your in-house development workflow

Docker Hub:

Docker Images:

Docker Containers:



Deviation

STEPS:

Start docker server"

Cmd: " sudo service docker start "

Pull (centos) image from docker hub:

Cmd: " sudo docker pull centos "

Run (centos) container:

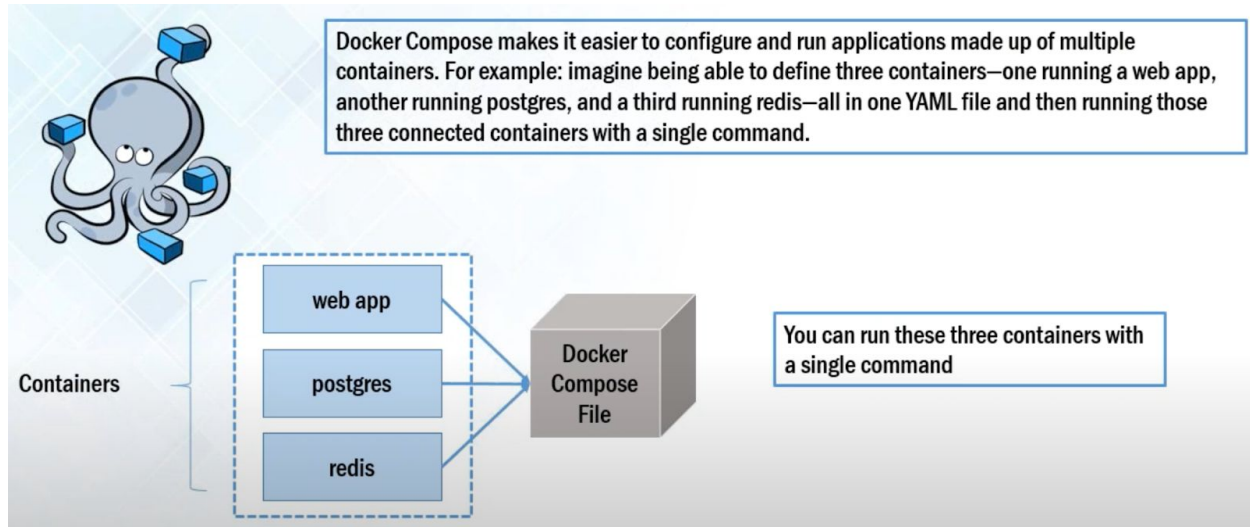
Cmd: " sudo docker run -it centos "

Exit container:

Cmd: " exit "

##

Docker Compose:



Cmd use of compose steps:

“ sudo apt-get install python-pip ”

“ sudo pip install docker-compose ”

Make a directory named wordpress:

“ mkdir wordpress ”

“ cd wordpress/ “

Edit yml file using gedit:

“ sudo gedit docker -compose.yml ”

Open a new doc and copy the yml code, and paste it into the docker-compose.yml file.

**While linking containers the word after colon , i.e., “mysql” is the name we give.

“ sudo docker-compose up -d ” - this will pull 3 images and build containers.

“ ”

REF: <https://youtu.be/lcQfQRDAMpQ?t=1509>

#####

=====

Kodekloud youtube: Full basic Docker course:

REF: <https://youtu.be/zJ6WbK9zFpI>

Docker For Beginners

[2:37](#) Docker Overview

[16:55](#) Docker Installation

[20:00](#) Docker Commands

[42:06](#) Docker Environment variables

[44:05](#) Docker Images

[51:36](#) Docker CMD vs Entrypoint

[58:30](#) Docker Networking

[1:03:57](#) Docker Storage

[1:16:19](#) Docker Compose

[1:34:39](#) Docker Registry

[1:39:30](#) Docker Engine

[1:46:06](#) Docker on Windows

[1:52:06](#) Docker on Mac

[1:54:39](#) Container Orchestration

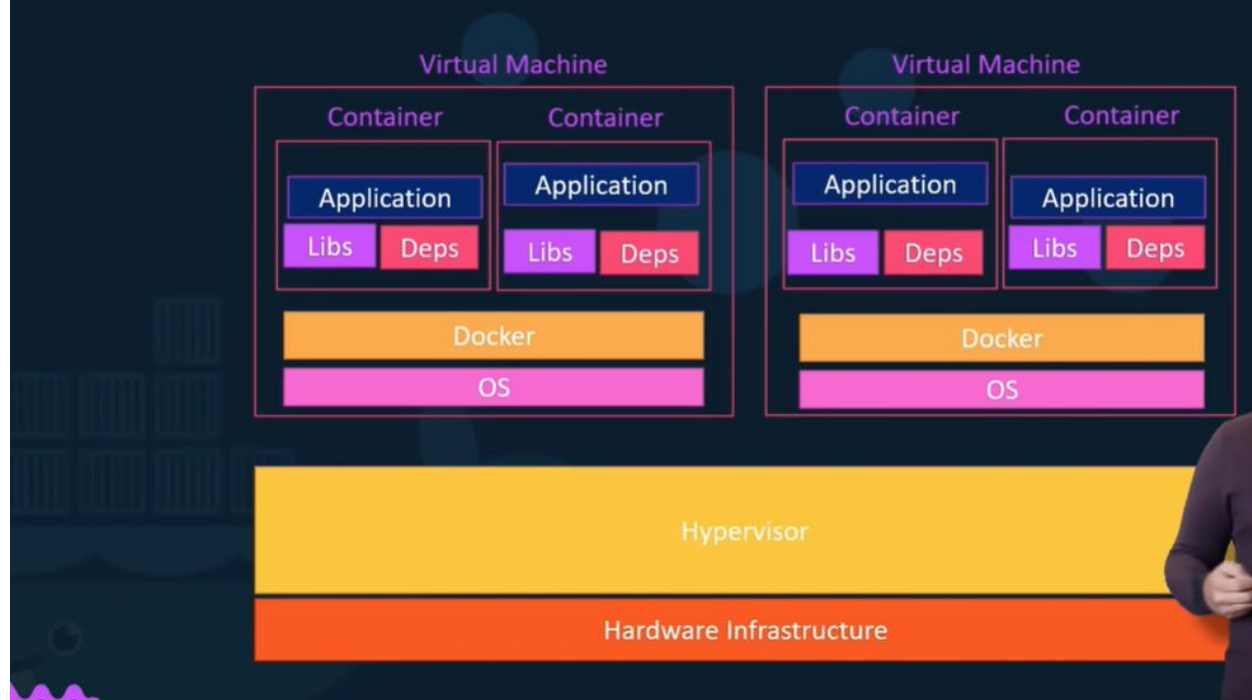
[1:58:53](#) Docker Swarm

[02:02:35](#) Kubernetes

[2:08:40](#) Conclusion

The 2 can be used together in integration:

Containers & Virtual Machines



DOCKER COMMANDS:

run - will run a container , either by using local image or pulling it from hub.

ps - list all running containers. Eg: “ docker ps ”

To see all containers: “ docker ps -a ”

stop - stop a container. Used with container name. Eg: “ docker stop [silly_sammer](#) ”

rm - remove an exited or stopped container permanently.

images - list all images.eg: “ docker images ”

rmi - remove an image. Eg: “ docker rmi [nginx](#) ”

****Make sure to delete all dependent containers in order to remove/delete an image!**

pull - to only pull/download an image (and not run). Eg: “ docker pull nginx ”

****A container only lives as long as the process inside it is alive.**

Eg: If the service inside the container dies/closes/crashes, then the container ‘exists’.

exec - Execute a command on a running docker container.

-d - Run docker container in 'detach' mode. Eg: " docker run -d thefolder/simple-webapp "

******run the '*docker ps*' command to view the container.

attach - to attach to a running (detached) container. Eg: " docker attach [name/ID_of_container](#) "

******can only just provide the first few chars. of the ID of container, in case of ID.

******Then can use " control+C" to detach.

tag -

Eg: Specifying version: " docker run redis:4.0 "

Default tag is always " :latest "

****** IMP: by default docker doesn't listen to STDIN.

-i - for interactive mode. Eg: " docker run -i [my_folder/simple-prompt-docker](#) "

-t - stands for sudo terminal (to get the prompt as well). Eg: " docker run -it [my_folder/simple-prompt-docker](#) " - attached to terminal and in interactive mode.

Port Mapping:

The underlying host is called docker host or docker engine. We run a contain

What IP to use to access the app running in docker container (listening on port, say, 5000) from a web browser?

2 options:

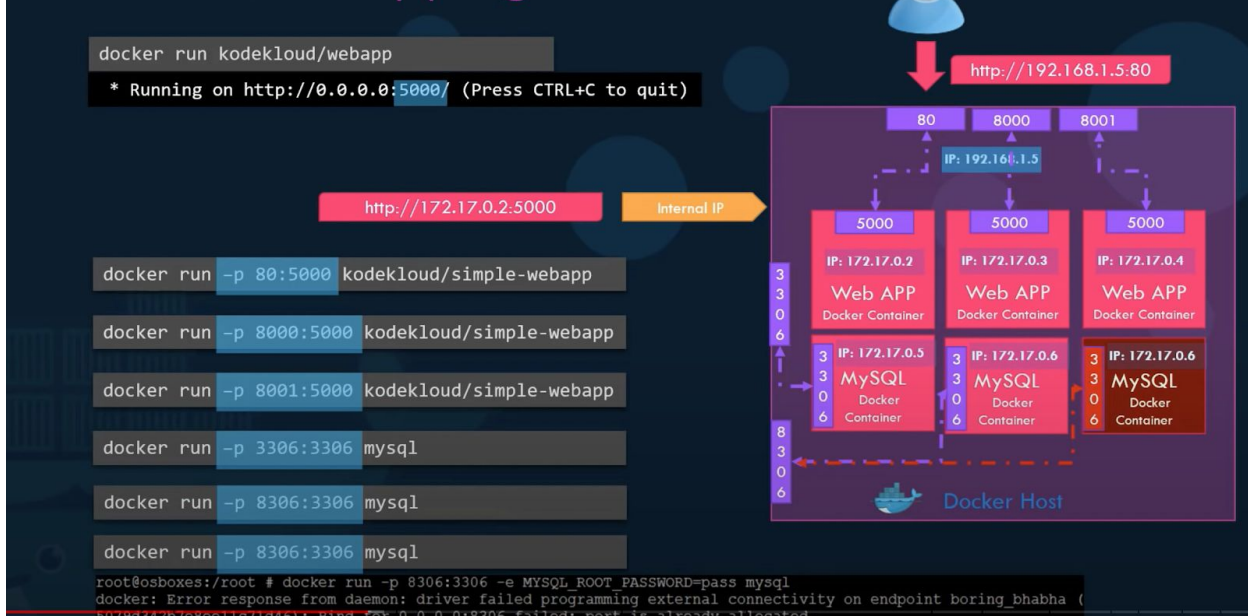
1)

To use IP of the docker container. Every docker container gets an IP assigned by default. This is an internal IP and only accessible within the docker host.

To access we can use the IP of the docker host. For that to work, the port inside the Docker container should be mapped to a free port on the docker host.

Port mapping - " docker run -p 80:5000 [my_folder/simple-prompt-docker](#) "

Run – PORT mapping



VOLUME MAPPING: how data is persisted in a container.

Docker container has its own isolated file system. Located at: “ /var/lib/container_name “
If the Container is deleted, all stored data there is gone.

To keep the data persistent after container is deleted, map a dir outside the container on docker host to a dir inside container.

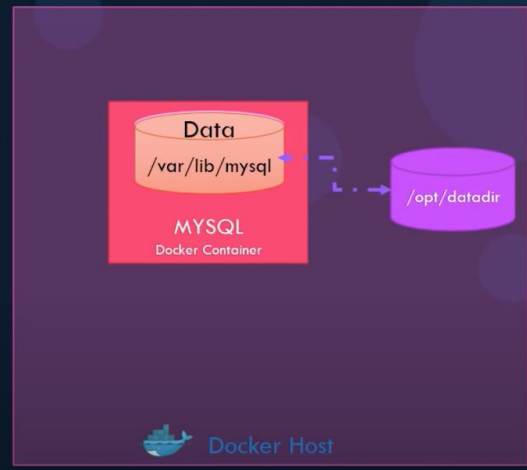
-v - “ docker run -v dir_on_docker_host:dir_inside_docker_container docker_container ”

RUN – Volume mapping

```
docker run mysql
```

```
docker stop mysql  
docker rm mysql
```

```
docker run -v /opt/datadir:/var/lib/mysql mysql
```



Inspect Container: see additional details of container.

“docker inspect *container_name*” : returns all details in json format.

Container **Logs**:

“docker logs *container_name*”

Docker Environment Variables:

-e -

Environment variables are found under the “config” section on the json return from the inspect container command.

Assign variable to a system variable: “color = os.environ.get('APP_COLOR')” then

“docker run -e APP_COLOR=blue *container_name*”

Docker IMAGES:

How to create an Image?:

Manual broad steps for deployment:

1. OS - Ubuntu

2. Update apt repo

3. Install dependencies using apt

4. Install Python dependencies using pip

5. Copy source code to /opt folder

6. Run the web server using “flask” command

DOCKERFILE:

Dockerfile

Dockerfile

INSTRUCTION

ARGUMENT

Dockerfile

FROM Ubuntu

RUN apt-get update

RUN apt-get install python

RUN pip install flask

RUN pip install flask-mysql

COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run

Dockerfile

INSTRUCTION

ARGUMENT

Dockerfile

FROM Ubuntu

RUN apt-get update

RUN apt-get install python

RUN pip install flask

RUN pip install flask-mysql

COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run

Start from a base OS or another image

Install all dependencies

Copy source code

Specify Entrypoint

Dockerfile

```
FROM Ubuntu
RUN apt-get update && apt-get -y install python
RUN pip install flask flask-mysql
COPY . /opt/source-code
ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run
```

docker build Dockerfile -t mmumshad/my-custom-app

Layer 1. Base Ubuntu Layer

120 MB

Layer 2. Changes in apt packages

306 MB

Layer 3. Changes in pip packages

6.3 MB

Layer 4. Source code

229 B

Layer 5. Update Entrypoint with "flask" command

0 B

```
root@osboxes:/root/simple-webapp-docker # docker history mmumshad/simple-webapp
IMAGE          CREATED          CREATED BY          SIZE      COMMENT
1a45ba829f10   About an hour ago /bin/sh -c #(nop)  ENTRYPOINT ["/bin/sh" "... 0B
```

It's build in layer on top of layer format.

Failure of any particular layer: it rebuilds the remainder of layers from using previous layers from cache.

CMD vs ENTRYPOINT:

Who defines what process is run inside a container?

**** BASH:** bash is not exactly a 'process' like a web server or DB server. It is a 'shell' that listens for input from a terminal. If it cannot find a terminal, it exits.

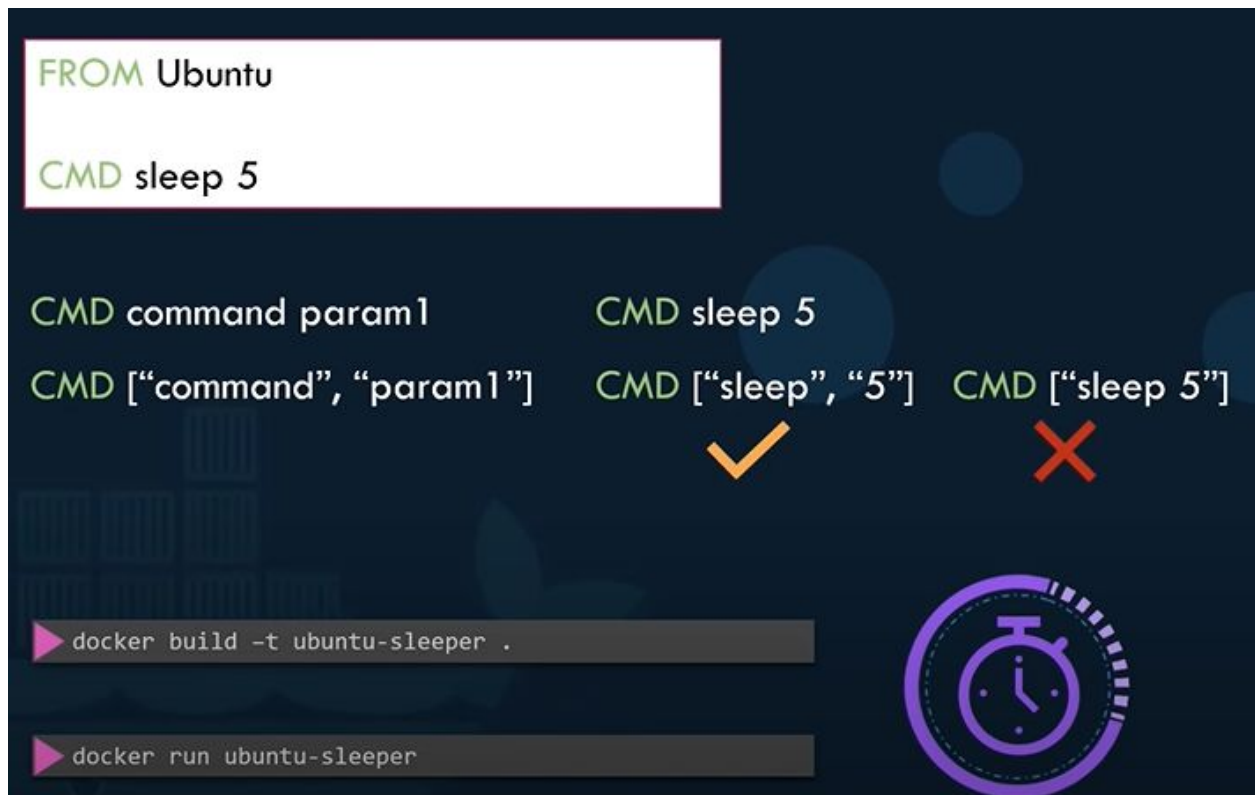
By default, docker doesn't attach a terminal to a container when it is run.

How to specify a different command to start a container than the default CMD (containing bash).?

One way: append a command to docker run to override the default one.
“ docker run ubuntu [COMMAND] ” . eg: “ docker run ubuntu sleep 5 ”

Next, how to make this change permanent?

Create own image from base image (eg: base ubuntu image)



How to change the parameter of the *sleep* command, say?

One way is to change value while calling:
“ docker run ubuntu-sleeper sleep 10 ”

Using ‘Entrypoint’ instructions:

In case of CMD, commandline params passed will be replaced entirely, while in case of 'entrypoint' the commandline params will get appended.

How to configure a default value for the command: use both, CMD and ENTRYPOINT.

If we still wanna change, it can be overwritten.

The diagram illustrates the configuration of a Docker container named 'ubuntu-sleeper' using both `ENTRYPOINT` and `CMD`. It shows three scenarios of how the container's command is executed and the resulting output.

Configuration:

```
FROM Ubuntu  
ENTRYPOINT ["sleep"]  
CMD ["5"]
```

Scenario 1:

```
docker run ubuntu-sleeper
```

Output: `sleep: missing operand
Try 'sleep --help' for more information.`

Scenario 2:

```
docker run ubuntu-sleeper 10
```

Output: `Command at Startup: sleep 5`

Scenario 3:

```
docker run --entrypoint sleep2.0 ubuntu-sleeper 10
```

Output: `Command at Startup: sleep 10`

Scenario 4:

```
docker run --entrypoint sleep2.0 ubuntu-sleeper 10
```

Output: `Command at Startup: sleep2.0 10`

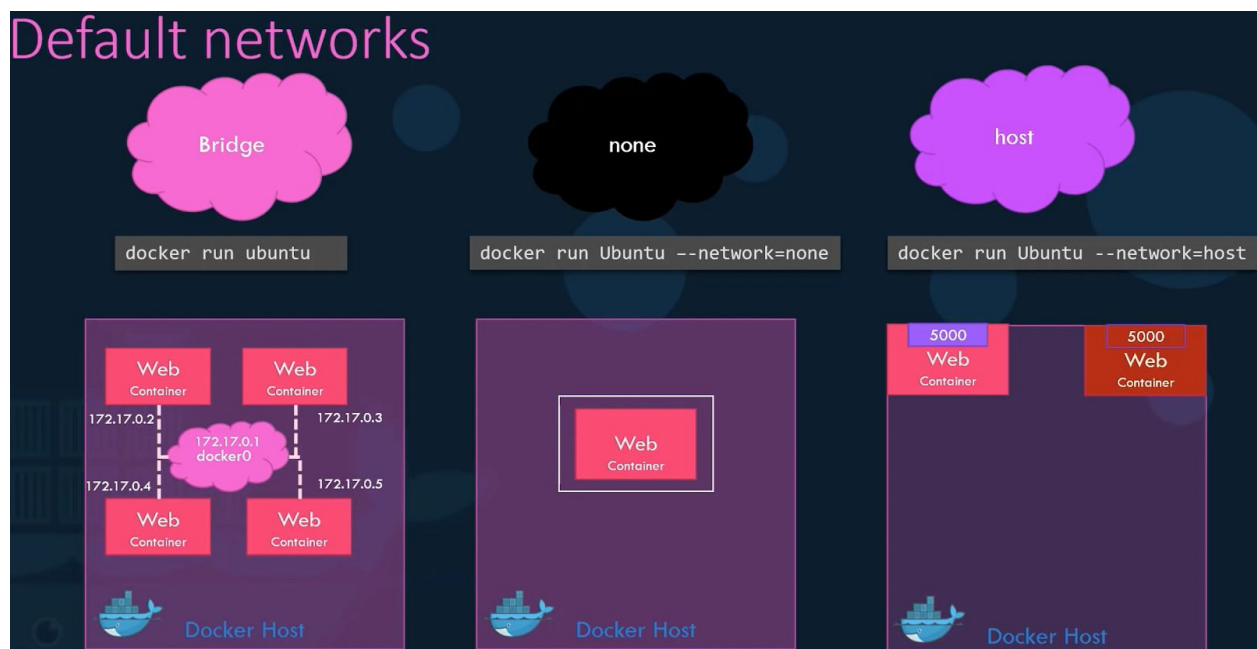
Docker NETWORKING:

When installed, Docker creates 3 networks automatically .

BRIDGE: default.

NONE: run in an isolated network

HOST: container share the same port as the docker host, so only one port can be used for only one container.



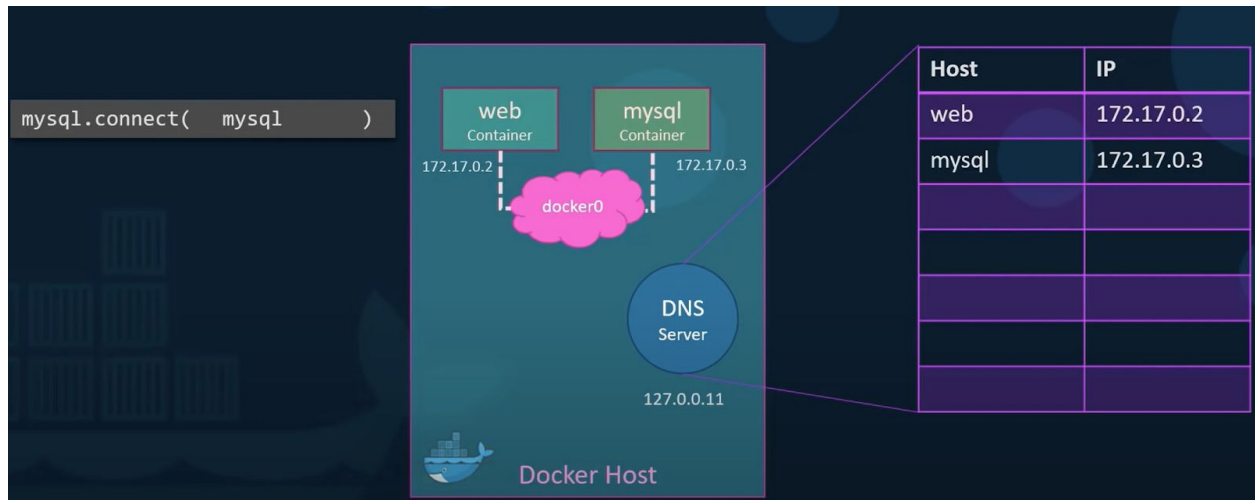
COMMANDS:

ls - To list all networks: " `docker network ls` "

Inspect network: " `docker inspect id_or_name_of_container` " - will have a section on 'network settings'

Embedded DNS:

** All container in a docker host can resolve (reach) each other with the container name (with help of docker DNS server), as their IP might not be same each time the system boots.
Docker has a built-in DNS server. It always runs at 127.0.0.11



Docker STORAGE:

File system



Docker builds images in a Layered Architecture.

Layered architecture

Dockerfile

```
FROM Ubuntu

RUN apt-get update && apt-get -y install python

RUN pip install flask flask-mysql

COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask
run
```

```
docker build Dockerfile -t mmumshad/my-custom-app
```

Layer 1. Base Ubuntu Layer	120 MB
Layer 2. Changes in apt packages	306 MB
Layer 3. Changes in pip packages	6.3 MB
Layer 4. Source code	229 B
Layer 5. Update Entrypoint	0 B

Dockerfile2

```
FROM Ubuntu

RUN apt-get update && apt-get -y install python

RUN pip install flask flask-mysql

COPY app2.py /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app2.py flask
run
```

```
docker build Dockerfile2 -t mmumshad/my-custom-app-2
```

Layer 1. Base Ubuntu Layer	0 MB
Layer 2. Changes in apt packages	0 MB
Layer 3. Changes in pip packages	0 MB
Layer 4. Source code	229 B
Layer 5. Update Entrypoint	0 B

Storage drivers and file systems. : Layered Architecture

Uses 'copy on write' mechanism.

COPY-ON-WRITE



VOLUME MOUNTING:

How to **persist** the data (in the read write above)?

Volumes:

First, Create a 'volume' : " docker volume create *data_volume* "

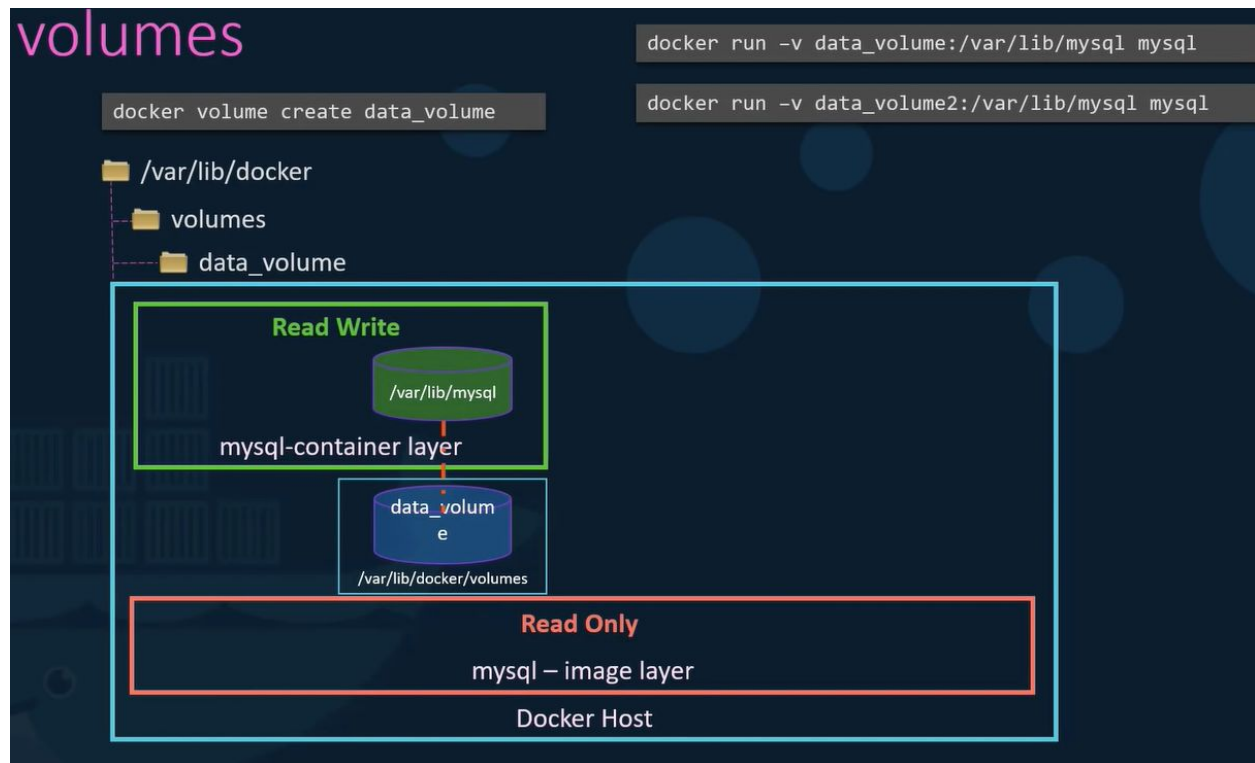
TWO types of mounting:

1) Volume mount:

Then mount this volume inside the container while running it:

“ `docker run -v data_volume:/var/lib/mysql mysql` ” { `:/var/lib/mysql` is the location inside container where container stores data by default }

If not specifically created volume before mounting while running container, docker will create a new volume directory and mount it automatically.



2) Bind mount: mounts dir from any location on docker host

data on already at **another present location** volume on docker host:

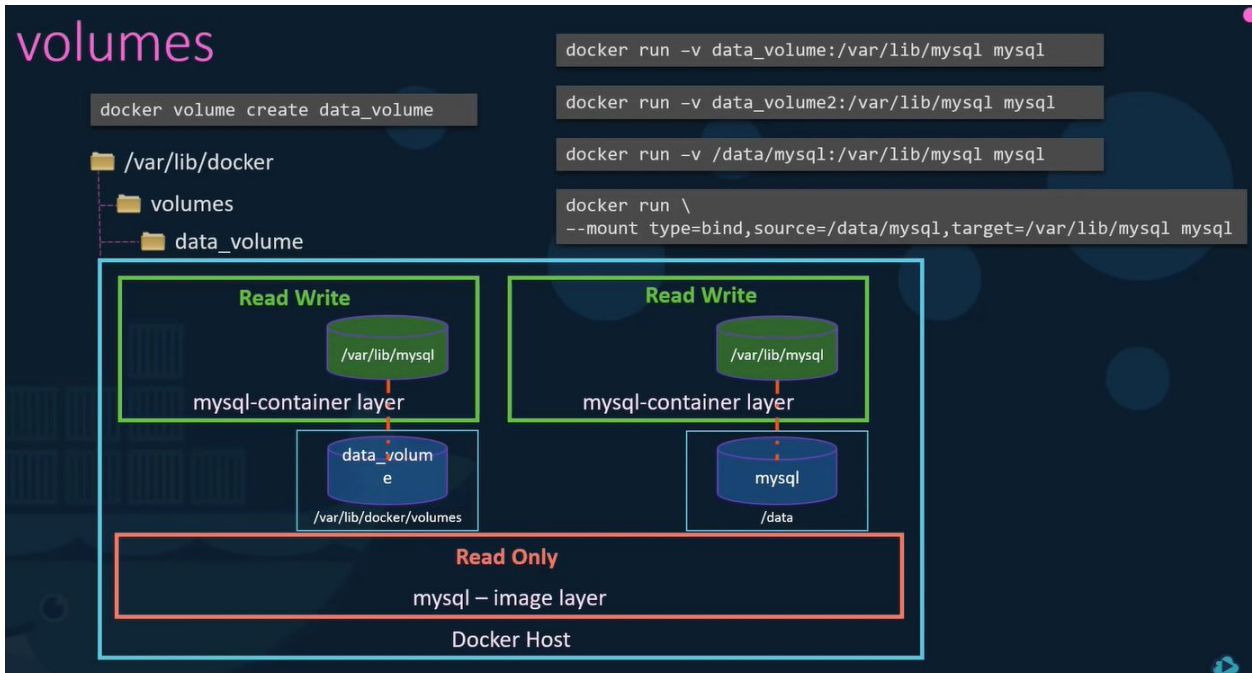
Use “ -v ”, but provide complete path to the folder/location we want to mount:

“ `docker run -v /data/mysql:/var/lib/mysql mysql` ”

** ‘ -v ’ is the old method. New one is “ -mount ” : needs to specify in ‘key=value’ format:

“ `docker run \`

`--mount type=bind, source=/data/mysql, target=/var/lib/mysql mysql` ”



```
docker run -v data_volume:/var/lib/mysql mysql
```

```
docker run -v data_volume2:/var/lib/mysql mysql
```

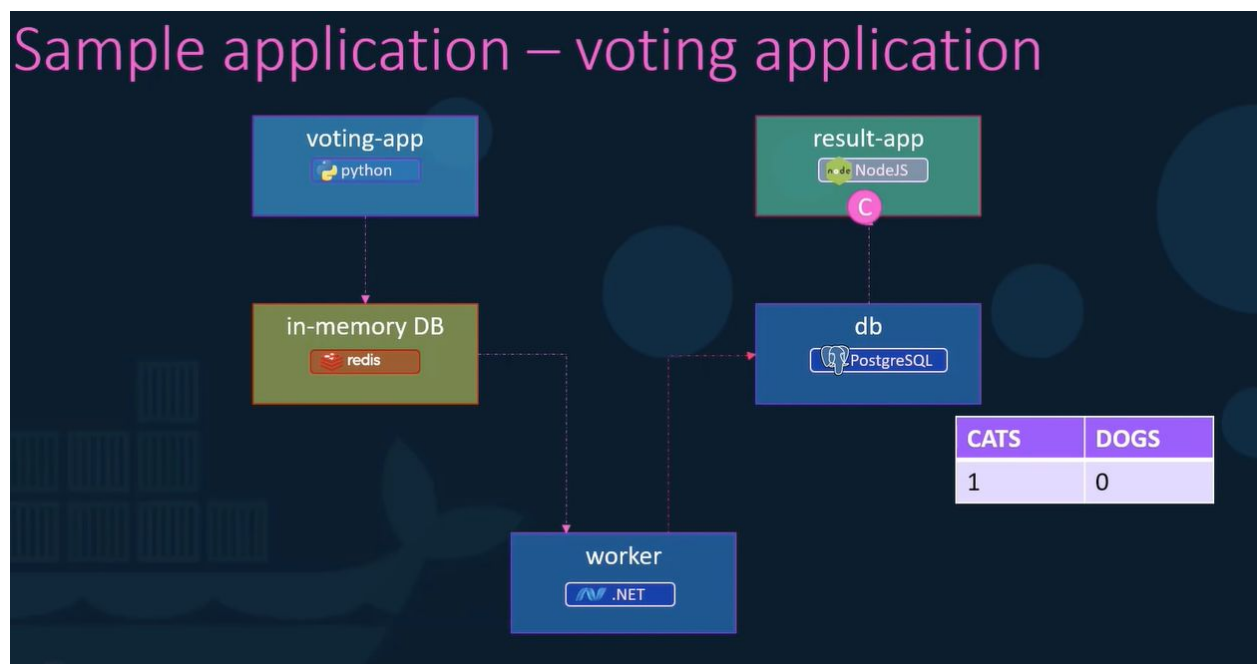
```
docker run -v /data/mysql:/var/lib/mysql mysql
```

```
docker run \--mount type=bind,source=/data/mysql,target=/var/lib/mysql mysql
```

Storage drivers do all this work. Docker uses the best available option based on the OS.

Docker COMPOSE:

- Use DOCKER COMPOSE: to Run an application using multiple services.
- Configurations in “.yaml” files.



First approach: using ‘Docker run’

Assume that all images are built and available.

Naming the containers is important.
First, Run each container in the background:

docker run --links

```
docker run -d --name=redis redis
docker run -d --name=db postgres:9.4 result-app
docker run -d --name=vote -p 5000:80 voting-app
docker run -d --name=result -p 5001:80
docker run -d --name=worker worker
```

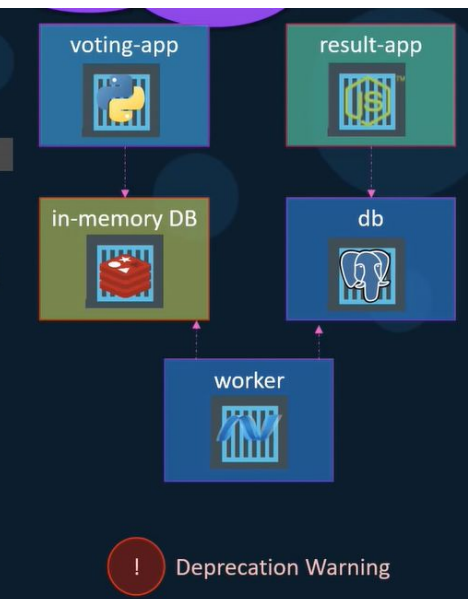


Then, connect the containers to each other using 'links'

```
docker run -d --name=redis redis
docker run -d --name=db postgres:9.4 --link db:db result-app
docker run -d --name=vote -p 5000:80 --link redis:redis voting-app
docker run -d --name=result -p 5001:80
docker run -d --name=worker --link db:db --link redis:redis worker
```

```
try {
  Jedis redis = connectToRedis("redis");
  Connection dbConn = connectToDB("db");

  System.err.println("Watching vote queue");
}
```



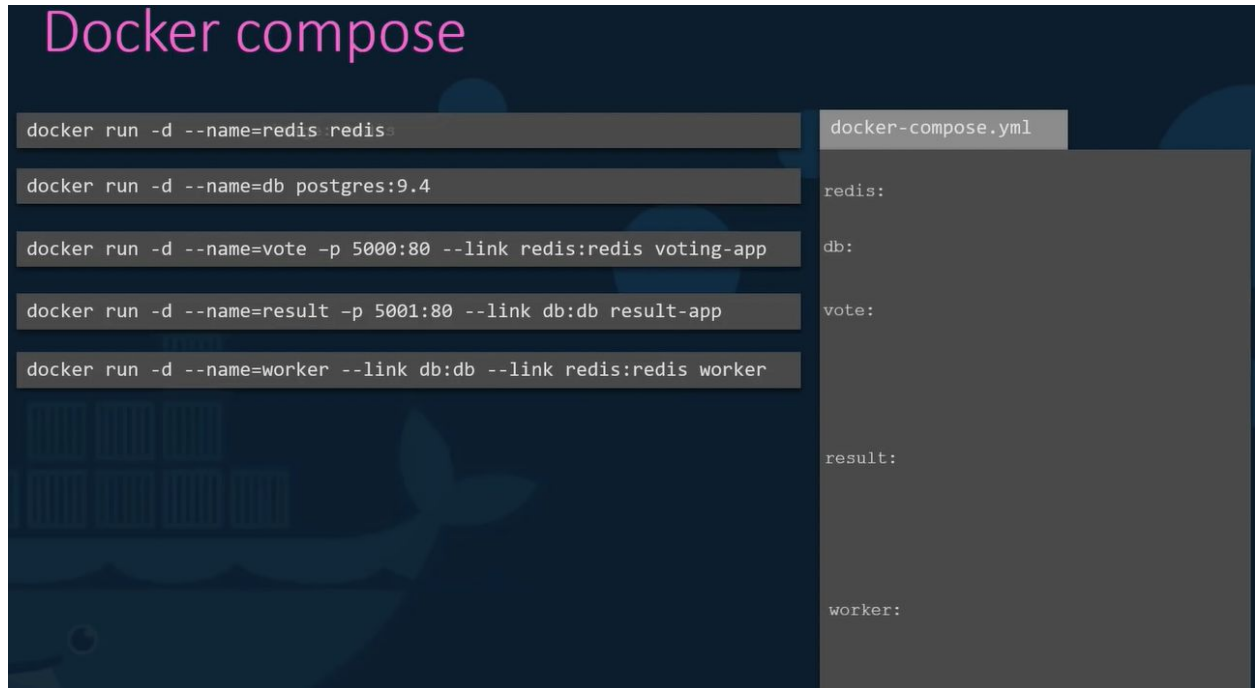
! Deprecation Warning

For link, `--link name_of_redis_container:name of the host voting app` is looking for
**Using ' `--link` ' is deprecated, and may be removed.

After 'run commands' are tested and ready: DOCKER COMPOSE from it.

“ Docker-compose.yml “

Start with dictionary of container names:



under each key is a nested dictionary starting with a key-value pair of image and their names.

Next, move to other options used in the run docker approach.

Eg:

- 1) Ports: 2) Links 3)

```
docker run -d --name=redis redis
docker run -d --name=db postgres:9.4
docker run -d --name=vote -p 5000:80 --link redis:redis voting-app
docker run -d --name=result -p 5001:80 --link db:db result-app
docker run -d --name=worker --link db:db --link redis:redis worker
```

```
db:db = db
```

```
docker-compose.yml
redis:
  image: redis
db:
  image: postgres:9.4
vote:
  image: voting-app
  ports:
    - 5000:80
  links:
    - redis
result:
  image: result-app
  ports:
    - 5001:80
  links:
    - db
worker:
  image: worker
  links:
    - redis
    - db
```

After compose file is ready:

To bring up the entire stack : “ docker-compose up ”

DOCKER COMPOSE - BUILT:

Building the images used in the docker compose above:

If we need docker compose to run a docker built instead of trying to pull an image:

Replace the ‘image’ line with ‘built’ line and Provide location with dir containing application code and a docker file with instructions to build a docker image.

Docker compose - build

The image displays two versions of a `docker-compose.yml` file side-by-side, illustrating the evolution of Docker Compose syntax. The left version uses the `image` property to specify pre-built images, while the right version uses the `build` property to define services from local source code. A third panel on the right shows a GitHub repository for `example-voting-app`, which contains the source files for the services defined in the compose files.

docker-compose.yml (Left)	docker-compose.yml (Right)
<code>redis:</code>	<code>redis:</code>
<code> image: redis</code>	<code> image: redis</code>
<code>db:</code>	<code>db:</code>
<code> image: postgres:9.4</code>	<code> image: postgres:9.4</code>
<code>vote:</code>	<code>vote:</code>
<code> image: voting-app</code>	<code> build: ./vote</code>
<code> ports:</code>	<code> ports:</code>
<code> - 5000:80</code>	<code> - 5000:80</code>
<code> links:</code>	<code> links:</code>
<code> - redis</code>	<code> - redis</code>
<code>result:</code>	<code>result:</code>
<code> image: result</code>	<code> build: ./result</code>
<code> ports:</code>	<code> ports:</code>
<code> - 5001:80</code>	<code> - 5001:80</code>
<code> links:</code>	<code> links:</code>
<code> - db</code>	<code> - db</code>
<code>worker:</code>	<code>worker:</code>
<code> image: worker</code>	<code> build: ./worker</code>
<code> links:</code>	<code> links:</code>
<code> - db</code>	<code> - db</code>
<code> - redis</code>	<code> - redis</code>

Repository: dockersamples / example-voting-app

Branch: master

Files:

- static/stylesheets
- templates
- Dockerfile
- app.py
- requirements.txt

Docker compose - VERSIONS:

Docker has had many versions over the years and versions of development, with changing formats. So, docker-compose files can look very different sometimes.

Docker compose - versions

docker-compose.yml

```
redis:
  image: redis
db:
  image: postgres:9.4
vote:
  image: voting-app
ports:
  - 5000:80
links:
  - redis
```

version: 1

docker-compose.yml

```
version: 2
services:
  redis:
    image: redis
  db:
    image: postgres:9.4
  vote:
    image: voting-app
    ports:
      - 5000:80
    depends_on:
      - redis
```

version: 2

docker-compose.yml

```
version: 3
services:
  redis:
    image: redis
  db:
    image: postgres:9.4
  vote:
    image: voting-app
    ports:
      - 5000:80
```

version: 3

Docker Compose: NETWORKS:

Use separate frontend and backend networks to reduce traffic.



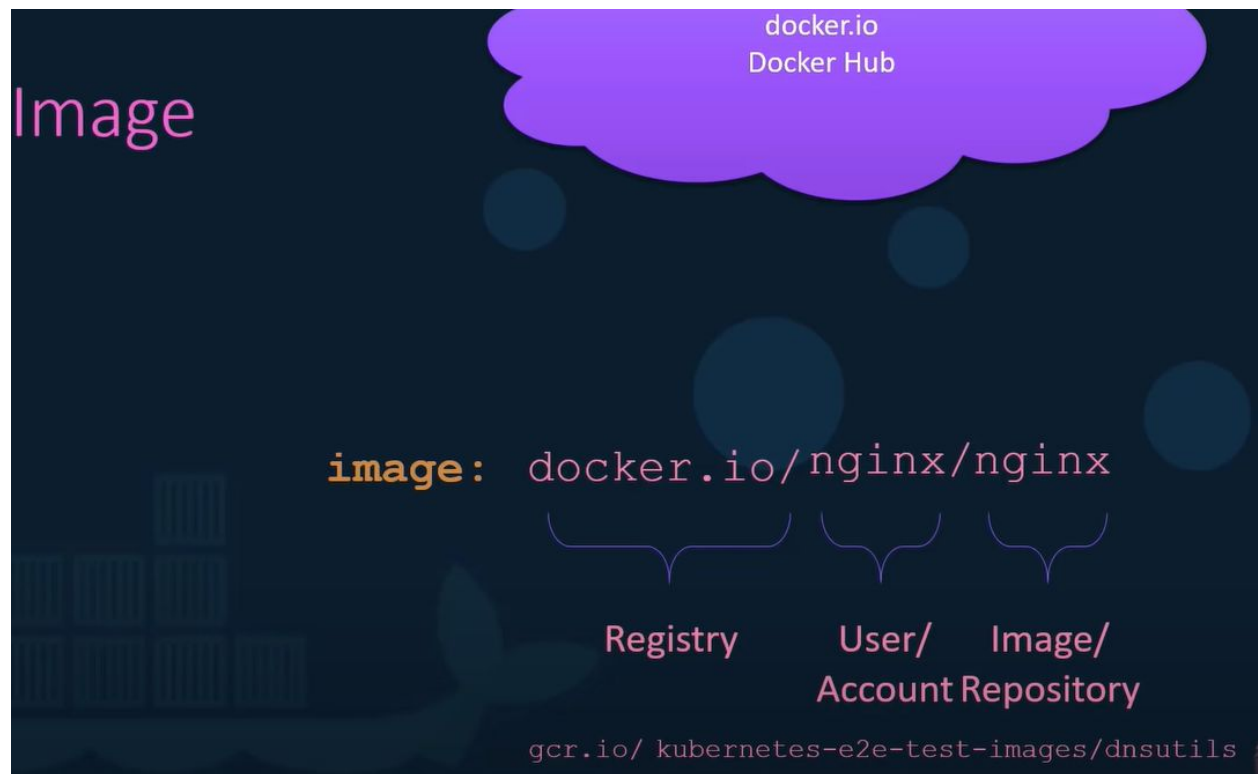
*omitted content from yml file here

Docker REGISTRY:

What is a registry?

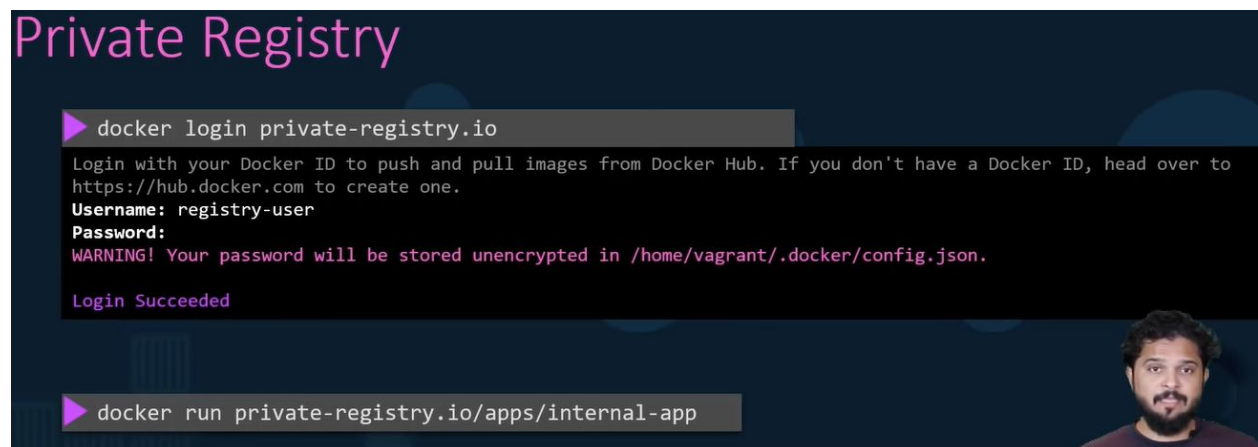
A registry is a storage and content delivery system, holding named Docker images, available in different tagged versions.

A registry is a storage and content delivery system, holding named Docker images, available in different tagged versions.



PRIVATE REGISTRY:

*always login before pulling or pushing to a private registry. Otherwise, 'image not found' error.



DEPLOY:

Running applications on premise and don't have a private registry.

Then, how to **deploy** (your own private registry) within your own Organization?:

Docker Registry is itself an application and available as an image named registry, exposes the API on port 5000.

image tag - tag the image with a private registry URL in it.

“ docker image tag my-image localhost:5000/my-image ”

Then, Push my image to the local private registry. “ docker push localhost:5000/my-image ”

The, can be pulled anywhere.

Deploy Private Registry

```
▶ docker run -d -p 5000:5000 --name registry registry:2
```

```
▶ docker image tag my-image localhost:5000/my-image
```

```
▶ docker push localhost:5000/my-image
```

```
▶ docker pull localhost:5000/my-image
```

```
▶ docker pull 192.168.56.100:5000/my-image
```

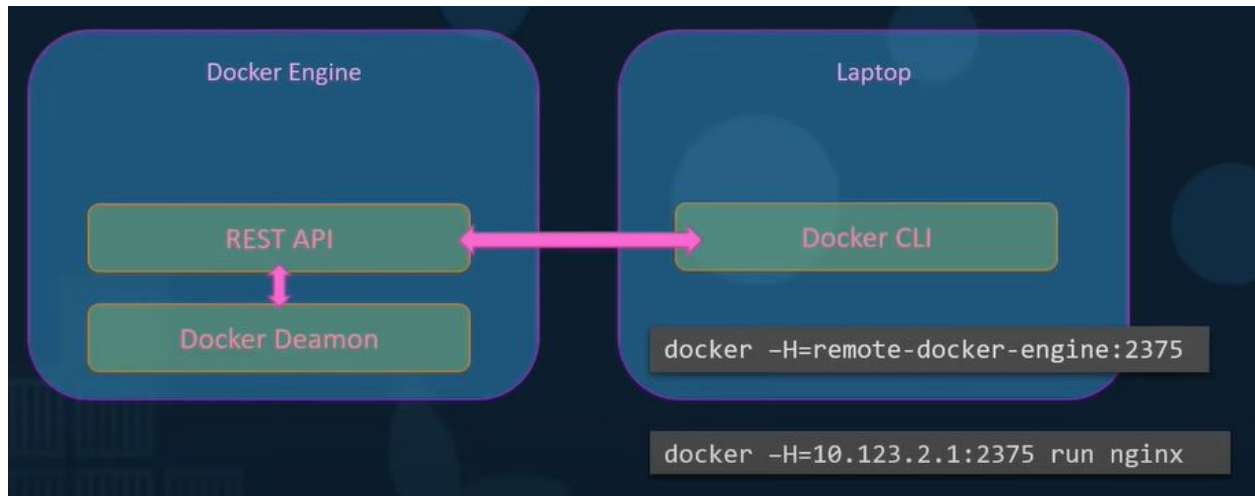
Docker ENGINE:

Simple the host with docker installed on it.

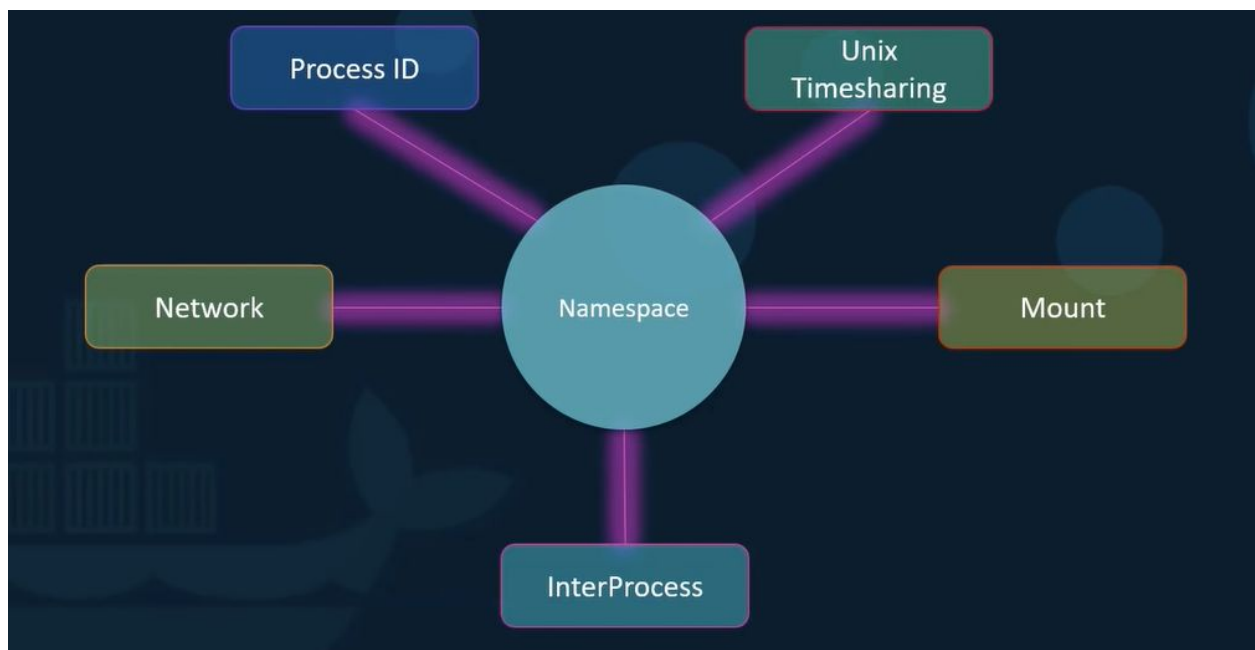


Architecture

CLI can be on a different system:

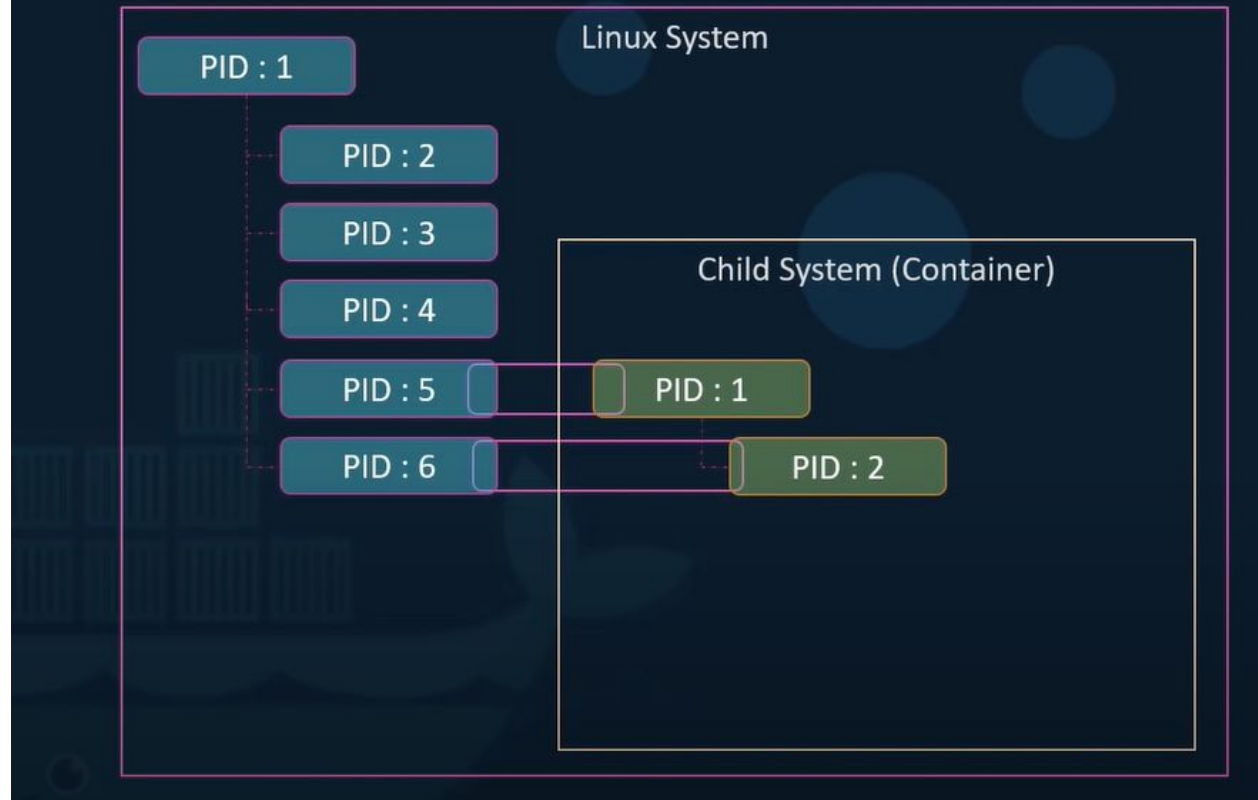


Docker uses namespaces to isolate workspace.



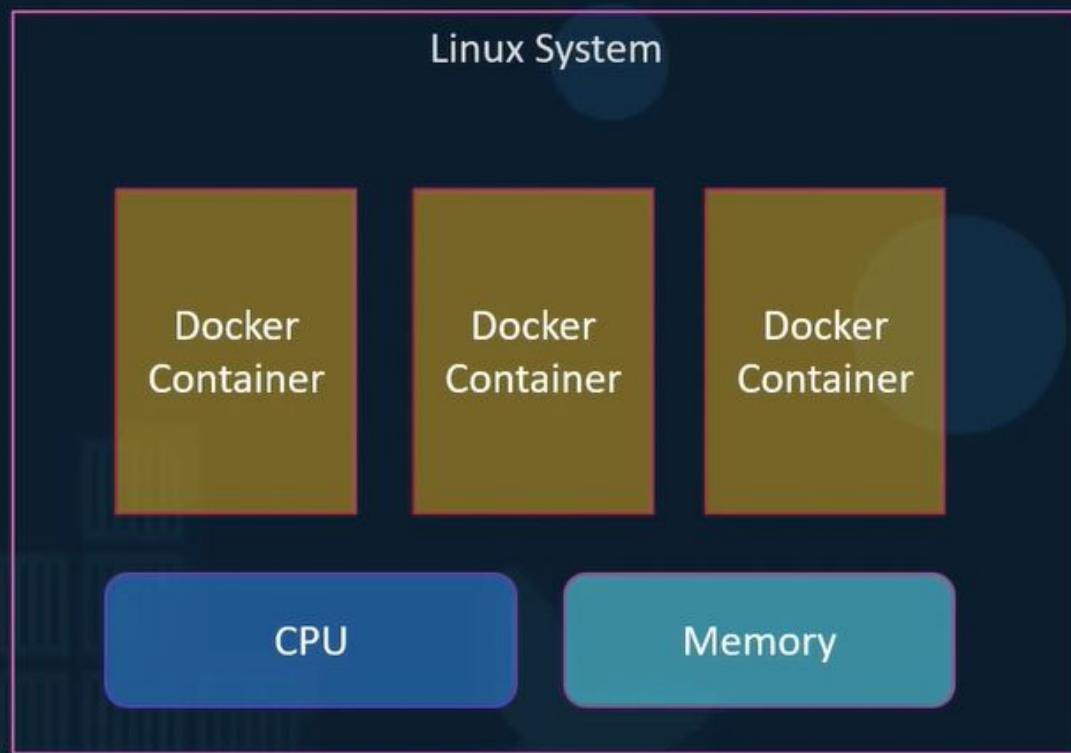
Namespace isolation technique example:

Namespace - PID



Resource sharing for the host, containers etc can be preset using "cgroups":
No restriction on resource use by default.

cgroups



```
docker run --cpus=.5 ubuntu
```

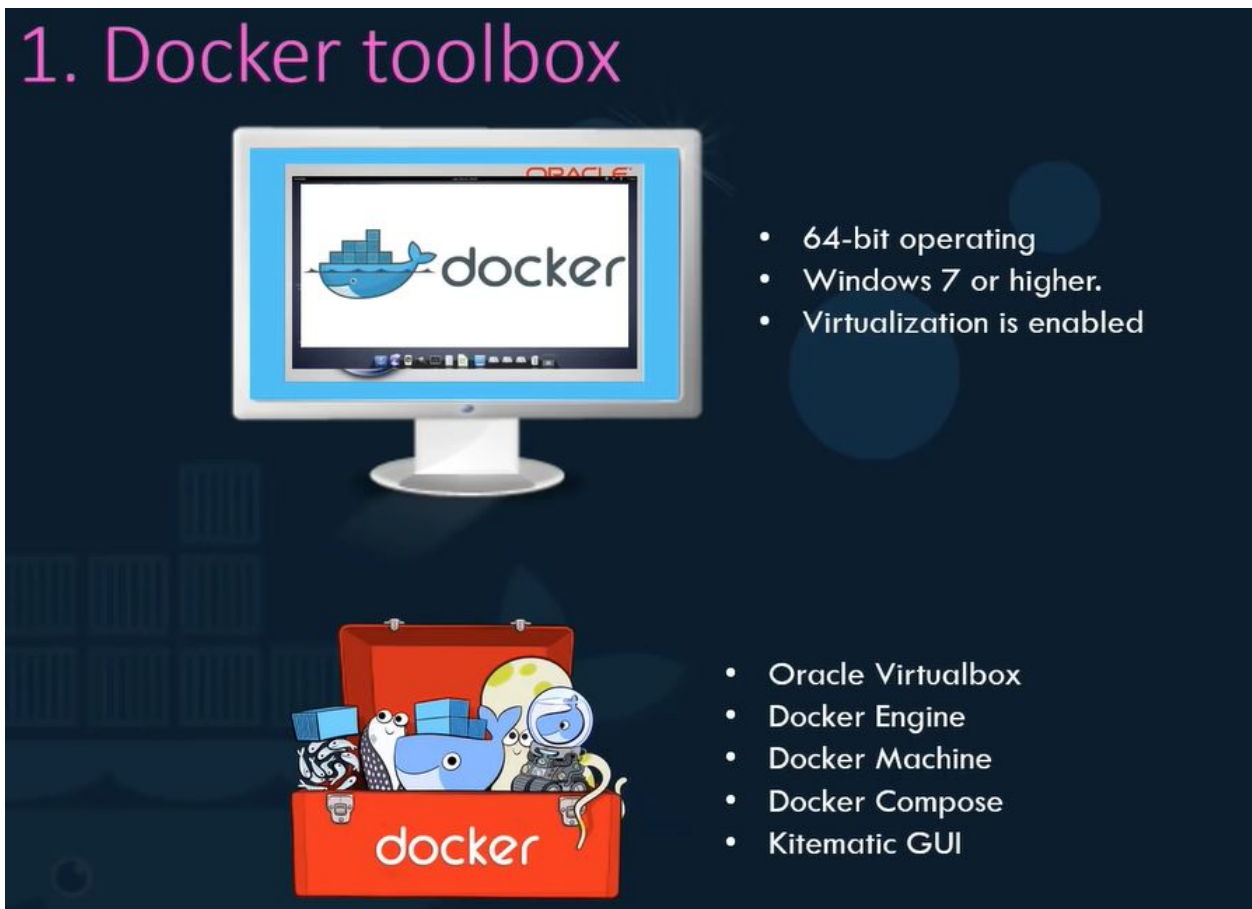
```
docker run --memory=100m ubuntu
```

Docker on WINDOWS:

TWO options

- 1) Docker toolbox and 2) Docker desktop for windows.

1. Docker toolbox



- 64-bit operating
- Windows 7 or higher.
- Virtualization is enabled

- Oracle Virtualbox
- Docker Engine
- Docker Machine
- Docker Compose
- Kitematic GUI

One is the old option (legacy) and for mainly old Windows that don't meet new requirements

****all this is for linux containers, linux applications packaged into linux images. Not talking about windows applications, windows images, or windows containers.**

This is only for: "Linux containers on windows host".

2016: MS support for windows containers for 1st time.

*Now, package Windows applications into windows docker containers and run on windows docker host, using 'Docker Desktop for Windows'.

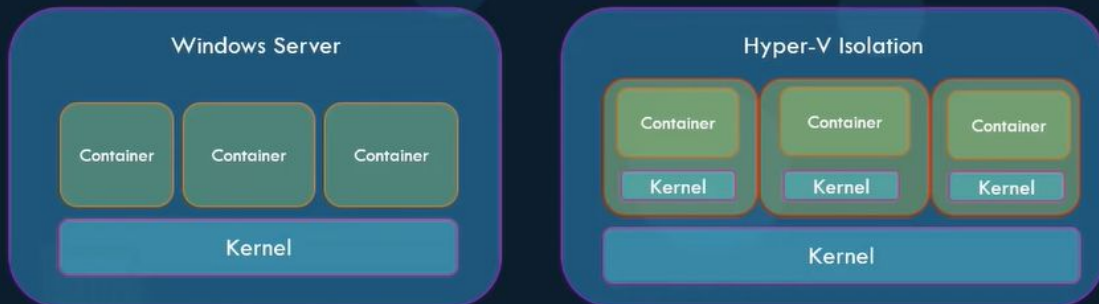


**The default docker for windows installation works with linux containers. Explicitly configure docker to switch to using windows containers.

WINDOWS CONTAINERS: Types:

Windows containers

Container Types:



Base Images:

- Windows Server Core
- Nano Server

Support

- Windows Server 2016
- Nano Server
- Windows 10 Professional and Enterprise (Hyper-V Isolated Containers)

*Windows 10 P and E only supports hyper-v isolated containers.

*VirtualBox and Hyper-V cannot coexist on same windows host

Docker on MAC:

<https://youtu.be/zJ6WbK9zFpl?t=6769>

Container Orchestration:



Docker SWARM: easy to set-up and get started. But lacks some of advanced autoscaling features.

MESOS: difficult to set up.

KUBERNETES: most popular. Really good features.

Solutions



Docker Swarm



kubernetes



MESOS

Docker SWARM:

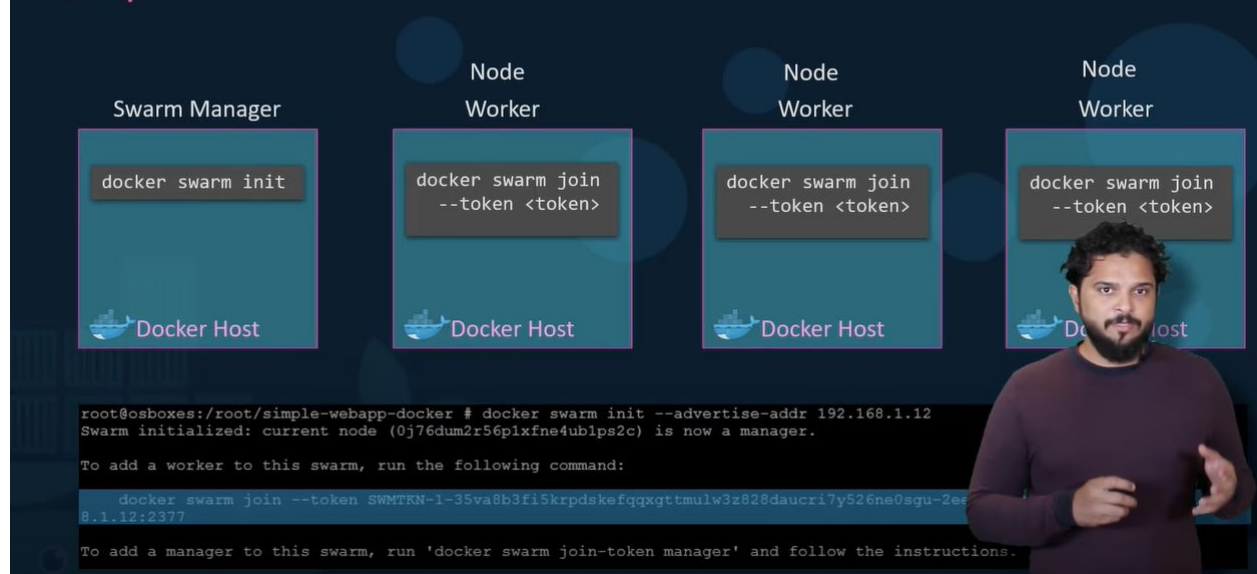
Can combine multiple docker machines together, into a single cluster.

Docker swarm



High availability and load balancing.

Setup swarm



Key component of swarm orchestration: “**Docker Services**”

*docker services are one or more instances of a single application or service that runs across nodes in the swarm cluster. Eg:

Docker service

```
docker run my-web-server
```

```
docker service create --replicas=3 --network frontend my-web-server
```

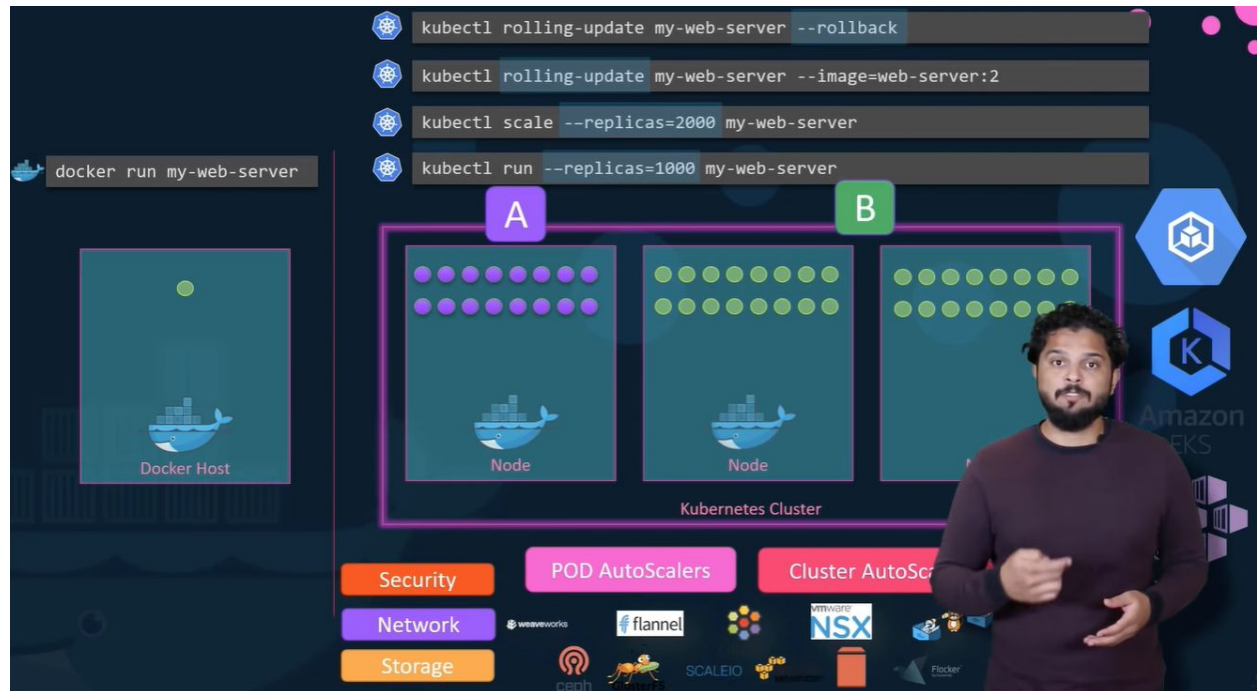
```
docker service create --replicas=3 -p 8080:80 my-web-server
```

```
docker service create --replicas=3 my-web-server
```

The diagram illustrates the Docker service architecture. On the left, a single 'Docker Host' is shown with a 'Web Server' container. On the right, a 'Docker Swarm' cluster is shown, consisting of three 'Worker Node' containers, each containing a 'Web Server'. A presenter is visible on the right side of the Swarm cluster.

Docker service command is run on the manager node, not on the service node.

KUBERNETES:

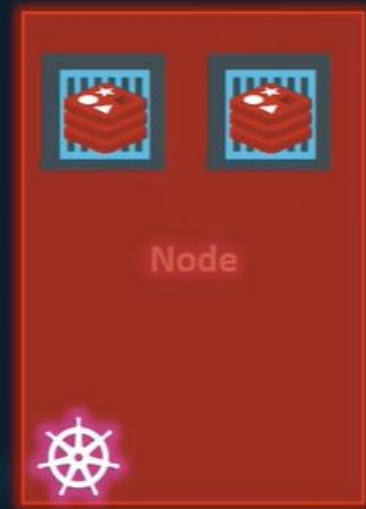


Kubernetes Architecture:

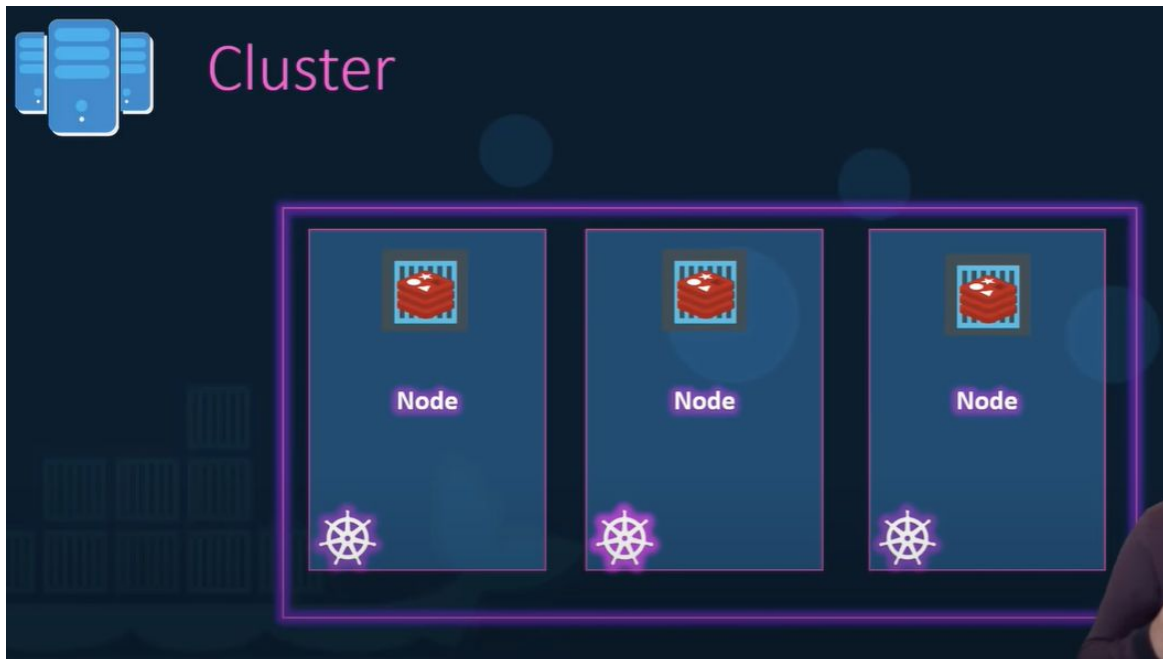
Nodes: a physical machine where kubernetes sits



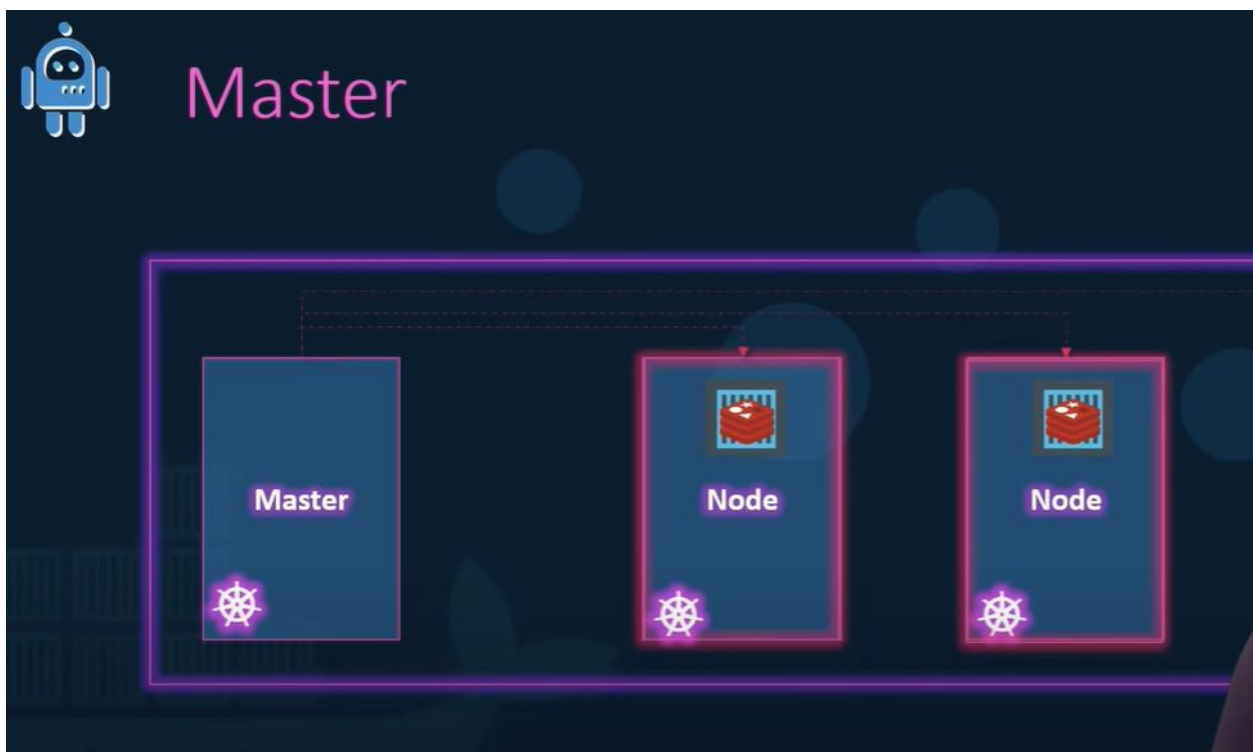
Nodes (Minions)



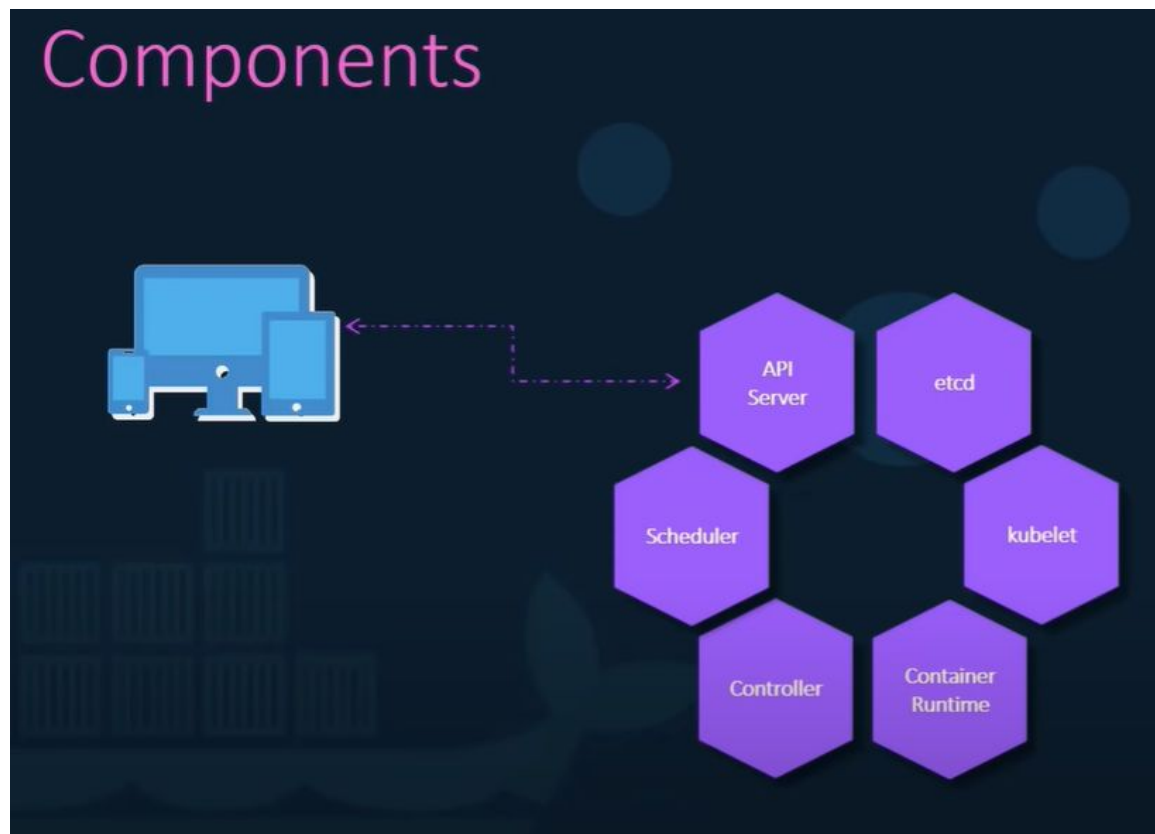
Cluster: a set of nodes:



Master:



Components:



Command line utility: “ kubectl ”



REF (same as beginning): <https://youtu.be/zJ6WbK9zFpl>

=====

#####

