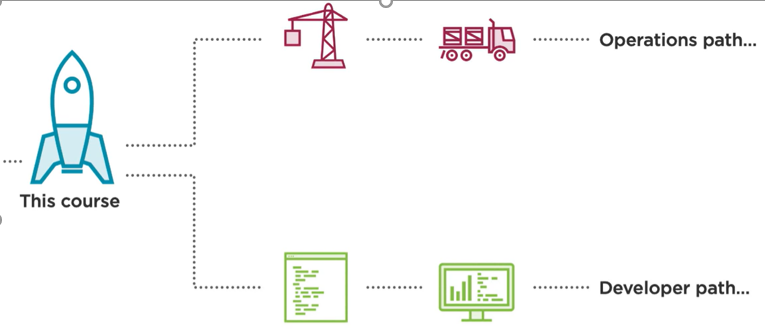
**DOCKER (Nigel Poulton)**

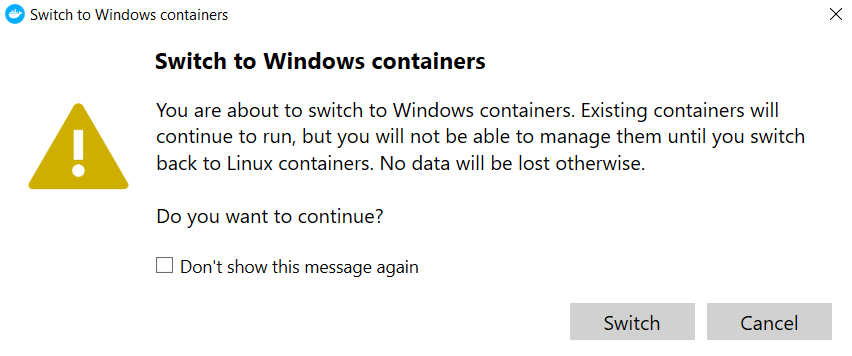
It is a devOps tools which is helpful for both developer and operations team.





Docker desktop on windows can run linux as well as windows container which means we can use our windows laptop to develop and test linux and windows apps.

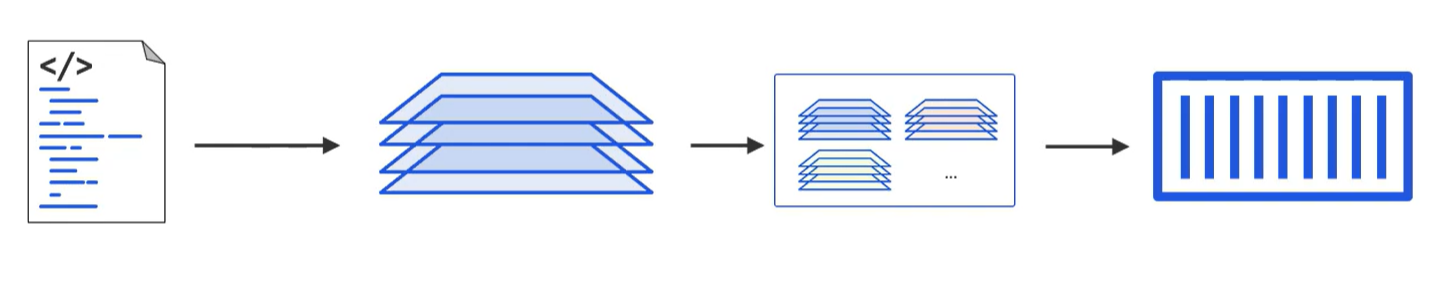
Whereas docker desktop running on mac can run only linux container.



IF WE CAN’t INSTALL DOCKER DESKTOP OR SOME ISSUES WITH PERMISSION TO INSTALL THEN ALSO WE CAN LEARN DOCKER USING **PLAY WITH DOCKER**

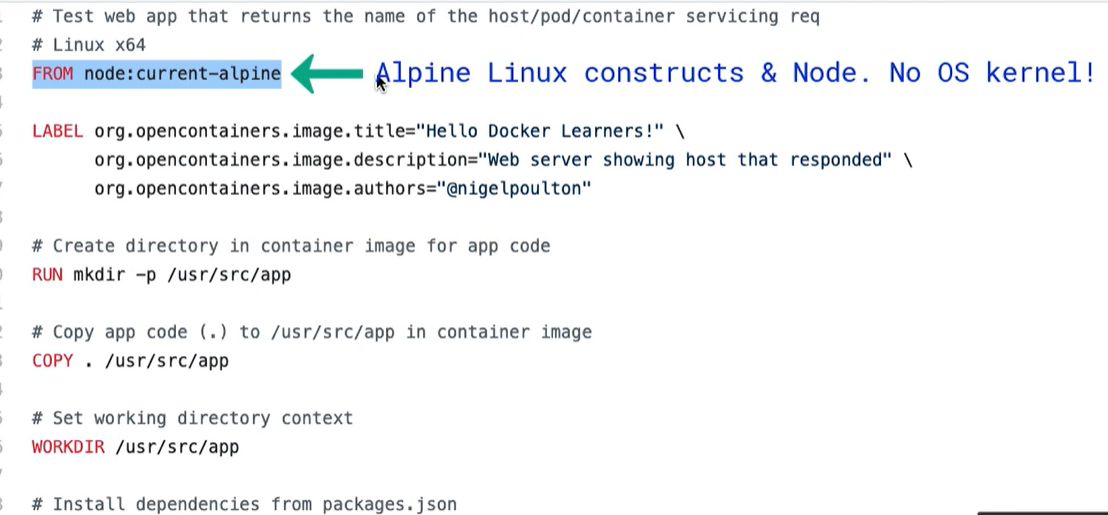
<https://labs.play-with-docker.com/>

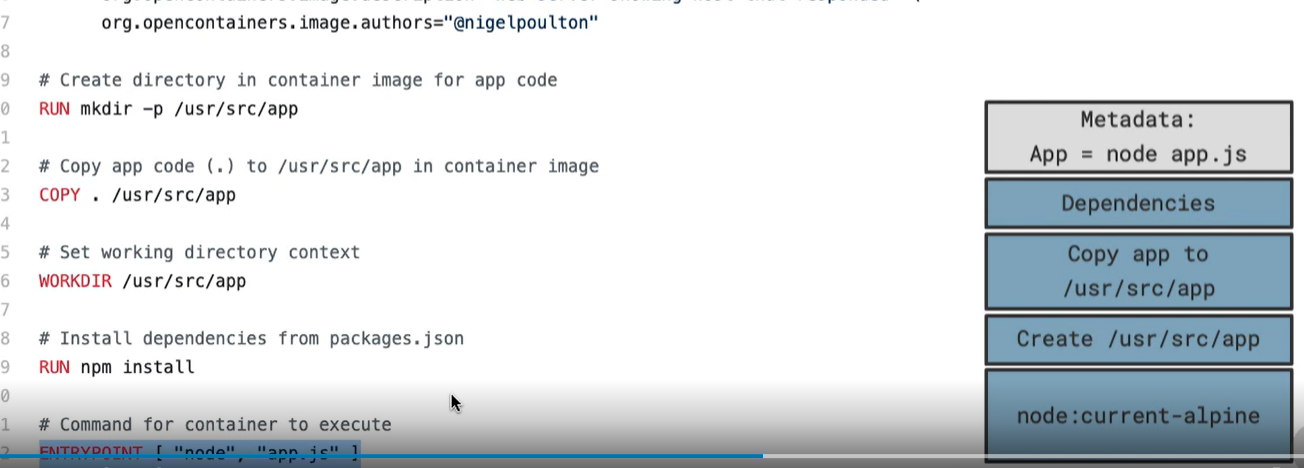
We need a docker hub account to login to the above lab.



DockerFile Image Put it on a Registry Run a container

1. We are building the image of a linux app with the help of Dockerfile**. DOCKER IS LANGUAGE INDEPENDENT**. Take the source code and build it on to an image.
2. Docker look at the Dockerfile and build the image.
3. Dockerfile is a set of instruction or steps for docker to follow to build the app and its dependency into an image.





**EVERY CONTAINER WHEN IT IS RUNNING USE THE KERNEL OF THE HOST.**

**STEP 1: Building Dockerfile**

**docker build -t abhi14563200/repo:img\_name .**

-t = it says we are going to tag the image

Abhi14563200 - dockerhub username

Repo – repo on docker hub

. – this indicates everything that docker needs to build image is in current directory

**DOCKERFILE**

# Test web app that returns the name of the host/pod/container servicing req

# Linux x64

FROM node:current-alpine

LABEL org.opencontainers.image.title="Hello Docker Learners!" \

org.opencontainers.image.description="Web server showing host that responded" \

org.opencontainers.image.authors="@nigelpoulton"

# Create directory in container image for app code

RUN mkdir -p /usr/src/app

# Copy app code (.) to /usr/src/app in container image

COPY . /usr/src/app

# Set working directory context

WORKDIR /usr/src/app

# Install dependencies from packages.json

RUN npm install

# Command for container to execute

ENTRYPOINT [ "node", "app.js" ]

**AFTER EXECUTING BUILD LOGS ARE**

=> [internal] load build definition from Dockerfile 0.2s

=> => transferring dockerfile: 714B 0.0s

=> [internal] load .dockerignore 0.2s

=> => transferring context: 2B 0.0s

=> [internal] load metadata for docker.io/library/node:current-alpine 10.8s

=> [internal] load build context 0.1s

=> => transferring context: 106.31kB 0.0s

=> [1/5] FROM docker.io/library/node:current-alpine@sha256:3ed634e0f15d3e05a1918c3949963508f7ed56350cf94156e6d804ae577849fa 21.4s

=> => resolve docker.io/library/node:current-alpine@sha256:3ed634e0f15d3e05a1918c3949963508f7ed56350cf94156e6d804ae577849fa 0.0s

=> => sha256:d0b02b1ec5534efb43a926069915c982aec745a8eb0611ebcffc4cafaa4e4a74 1.16kB / 1.16kB 0.0s

=> => sha256:f01bef8b5e9265c7d0f283d1fd9d0cdcb84f75fd7c3546a7c1da5c2049064285 6.44kB / 6.44kB 0.0s

=> => sha256:c158987b05517b6f2c5913f3acef1f2182a32345a304fe357e3ace5fadcad715 3.37MB / 3.37MB 3.3s

=> => sha256:ab5501735aae1e75c640bc26624c2daff7ea894126e950ca633ed8b9d9b0eb6a 2.35MB / 2.35MB 2.0s

=> => sha256:3ed634e0f15d3e05a1918c3949963508f7ed56350cf94156e6d804ae577849fa 1.43kB / 1.43kB 0.0s

=> => sha256:27ae9ce8b069146ce8506e0aa367ca24753a64b743e4b6e9bd8d1f876051d03a 47.73MB / 47.73MB 18.5s

=> => sha256:9ce0a5833ea2e805996e2300f45658a1e45304ffef38db5ed83901896a174093 450B / 450B 3.0s

=> => extracting sha256:c158987b05517b6f2c5913f3acef1f2182a32345a304fe357e3ace5fadcad715 0.1s

=> => extracting sha256:27ae9ce8b069146ce8506e0aa367ca24753a64b743e4b6e9bd8d1f876051d03a 1.9s

=> => extracting sha256:ab5501735aae1e75c640bc26624c2daff7ea894126e950ca633ed8b9d9b0eb6a 0.1s

=> => extracting sha256:9ce0a5833ea2e805996e2300f45658a1e45304ffef38db5ed83901896a174093 0.0s

=> [2/5] RUN mkdir -p /usr/src/app 1.0s

=> [3/5] COPY . /usr/src/app 0.1s

=> [4/5] WORKDIR /usr/src/app 0.2s

=> [5/5] RUN npm install 11.1s

=> exporting to image 0.4s

=> => exporting layers 0.4s

=> => writing image sha256:0d43842f9a20639f6ef5b93e9b30396277f4ce3624251d435a89381b96eff7bb 0.0s

=> => naming to docker.io/abhi14563200/pluralsight\_learn:first\_img 0.0s

**STEP 2: Hosting on Registry**

Once the image is build from dockerfile and if we want external world to use it we need to register on a hub.

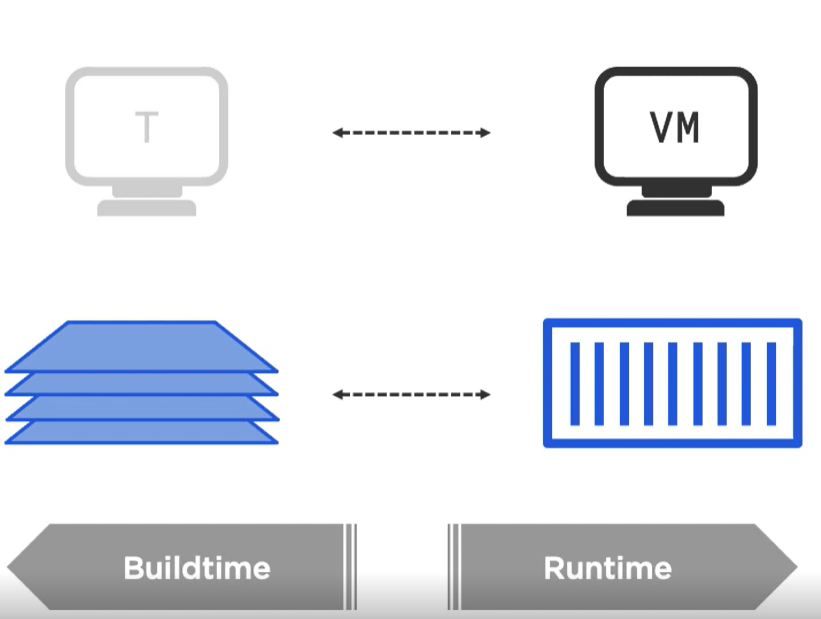
Docker login

**Docker push** **abhi14563200/repo:img\_name**

**STEP 3 : Running a container**

Container packages the code and dependencies so that application runs quickly and reliably in different computing env.

Running instance of an image like template VM and VM



**Docker run -d –name first\_container -p 8000:8080 imagename**

**Imagename = abhi14563200/repo:img\_name**

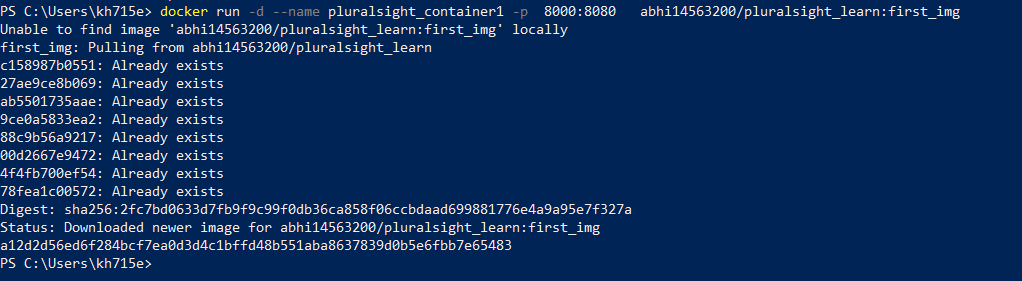
**-p 8000** = means docker desktop is running on 8000 port of our system

**8080** = means application inside container is running on 8080 port

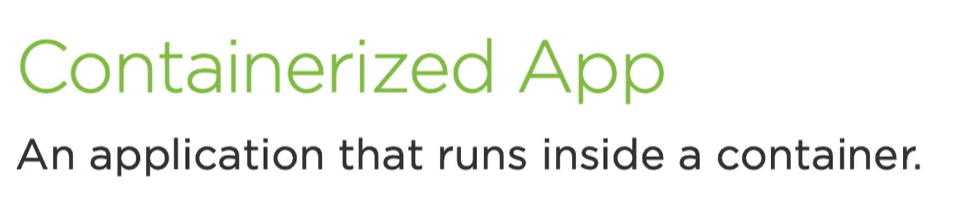
So whenever request is send for 8000 it is mapped to 8080 port inside container.

**-d** = means run this container in background detached from terminal

If the image is not in local it will look in the hub for images:



Once we run the above container then when we go on <http://locahost:8000> the application opens.



Docker run -d –name container1 imageName

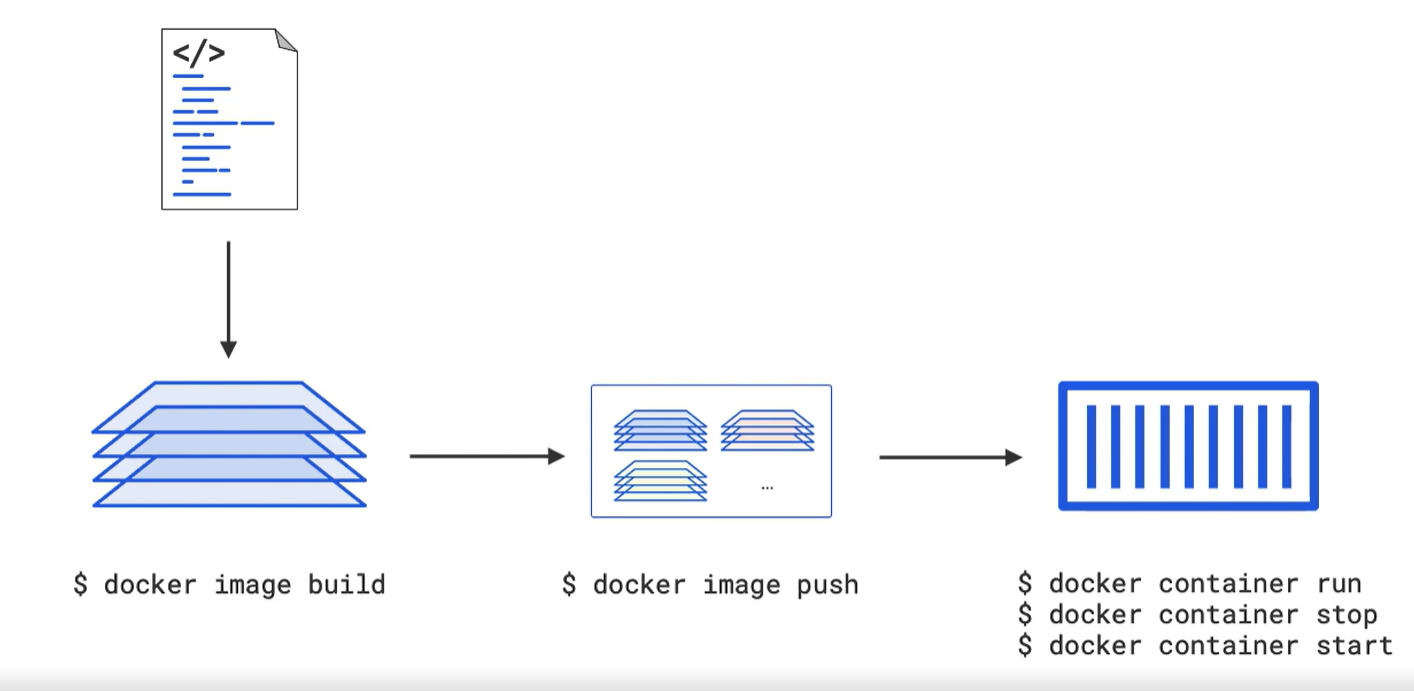
When we run with -d then container will run detact from terminal which means to go inside container we need to do

Docker exec -it containername sh

-it stands for interactive terminal

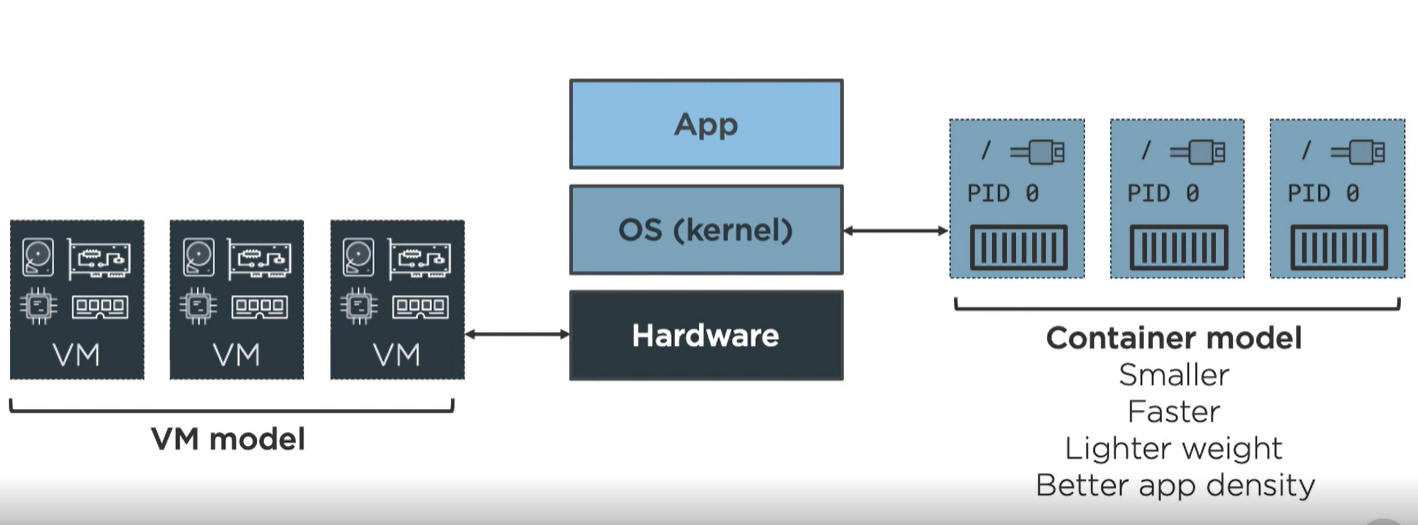
Docker run -it –name container1 imagename sh

**To come out from shell ctrl+P+Q, don’t do exit it will stop the container**



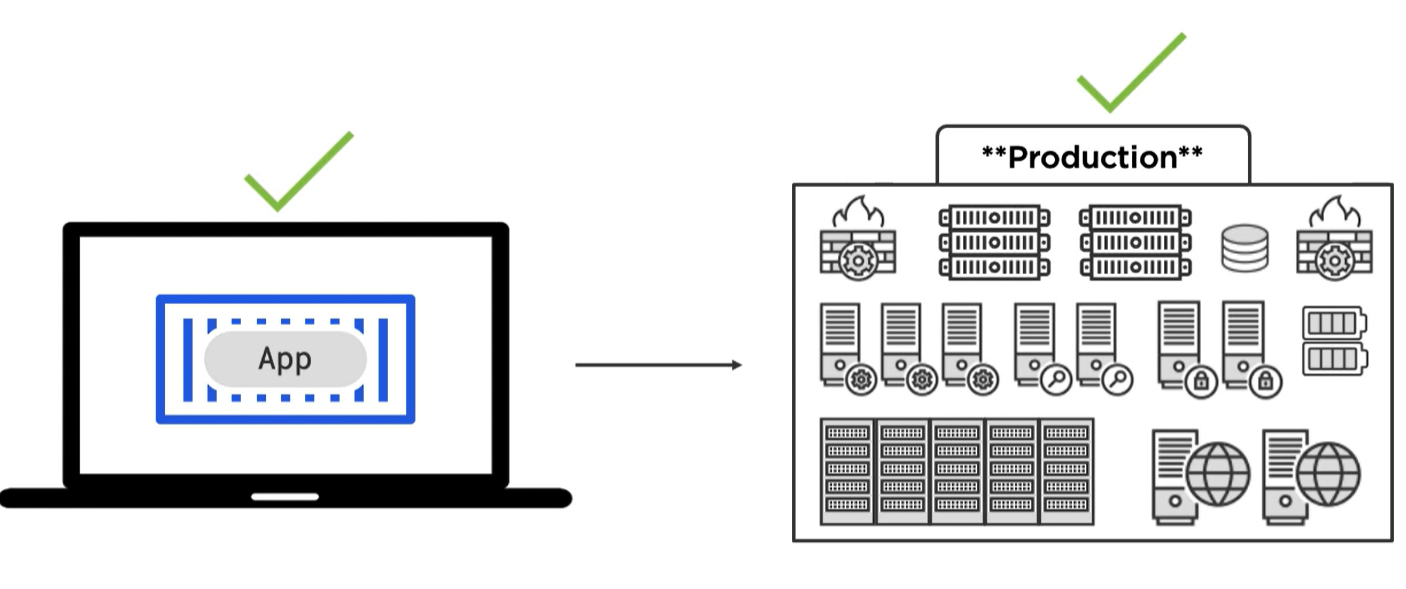
**CONTAINERS ARE virtualization 2.0**

* Hypervisors virtualizes hardware, like virtual CPU, RAM ,networks etc.
* Every VM on a host shares same hardware.
* **Containers virtualizes OS** – each container has a virtual operating system. It has its own process stream , root file system etc.
* Every container on a host shares same OS kernel.
* Because there is only **single OS kernel** in container model it is lightweight, smaller .



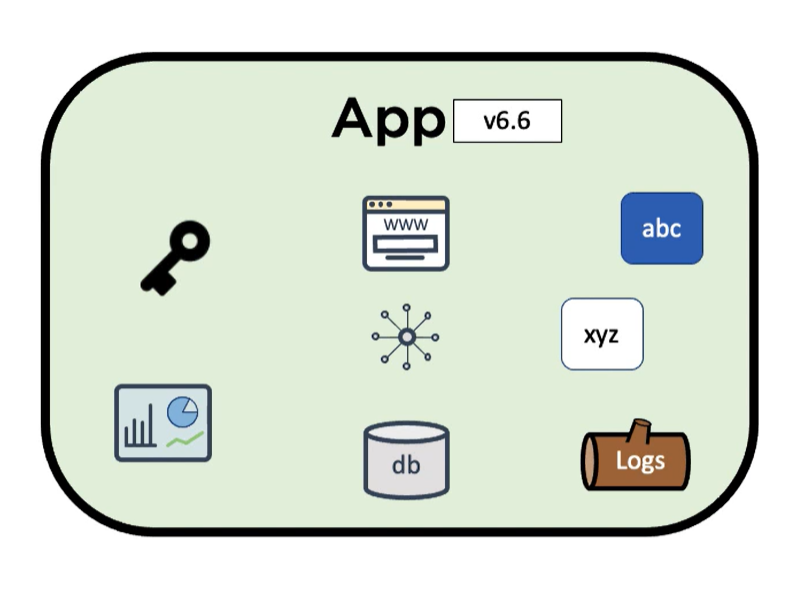
**Advantage for a developer.**

**We are packaging code as well as all dependency together in a container so if the app is working in our system it can run in any environment outside.** Gone are days where apps worked in our system but in production as it has to look for some library at different place so it is broken. All library is shipped with the images.



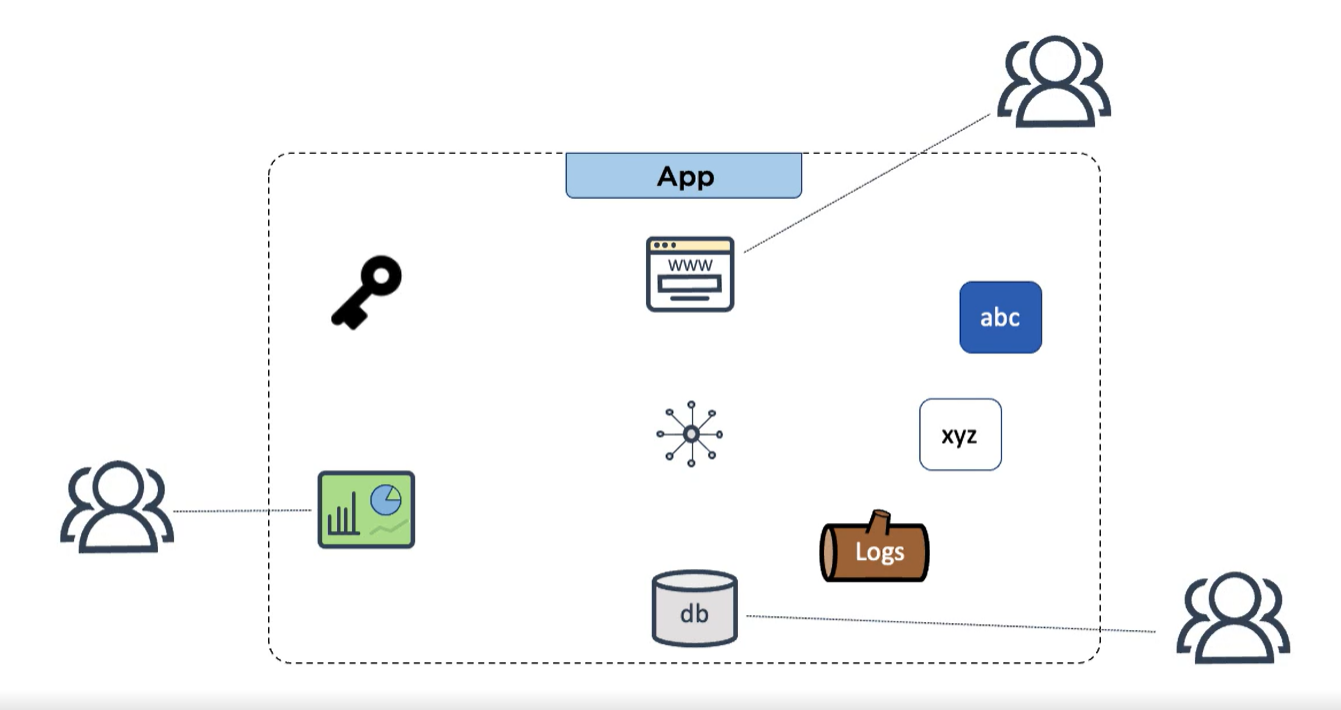


Monolathic service



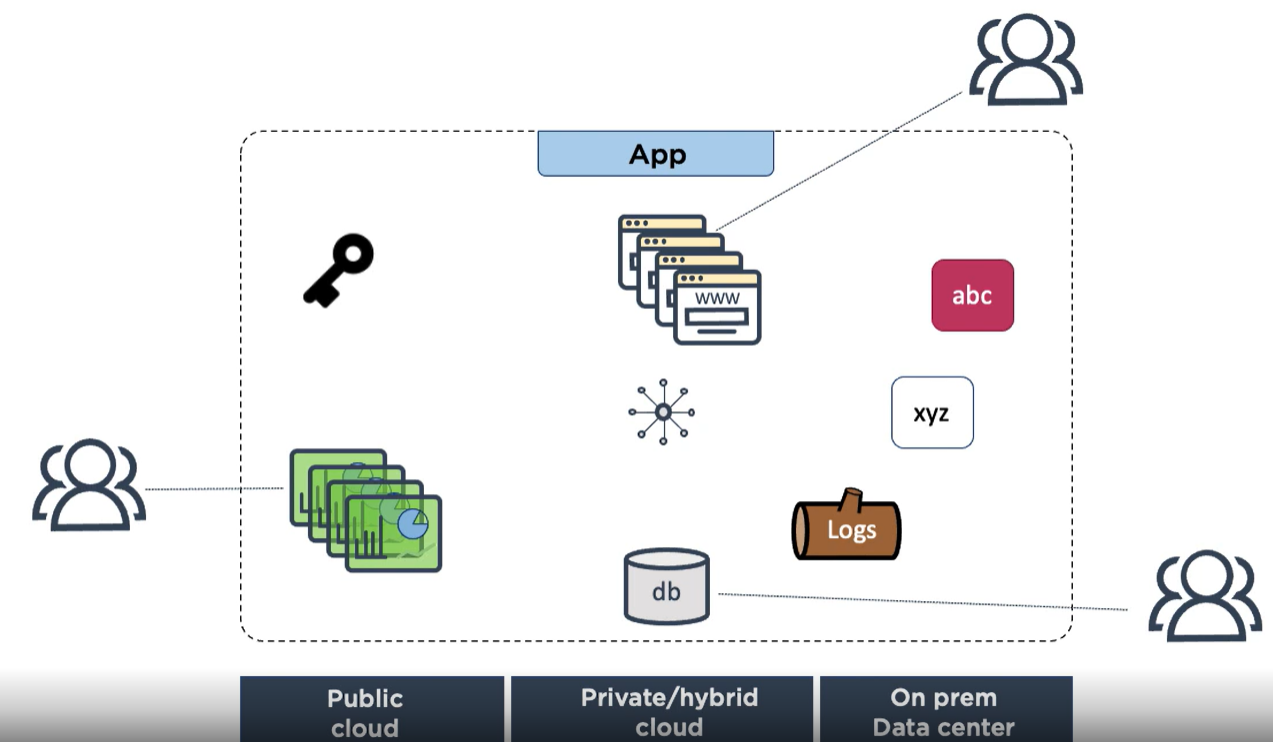
Multiple services and features are coupled together and build and deployed as a single unit. If we add any feature to one service entire application needs to be down and build. If we want to scale an individual service we need to scale entire application.

**Microservice**



Each feature or service is build and deployed independently and by different teams.

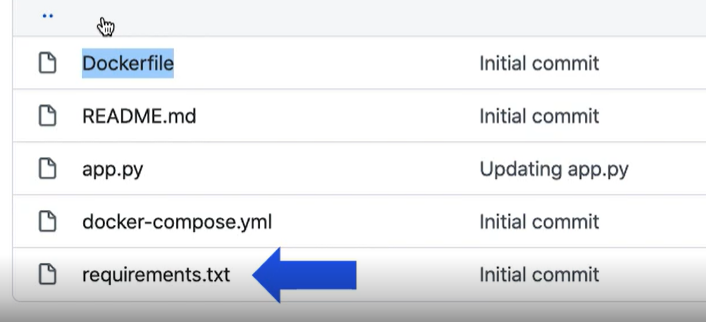
**Cloud native features** like patching, dynamic updating and scaling can be done on an individual microservice without disturbing other. Application can run on private, public or in-house cloud. Anywhere we have docker or kubernetes.



**Docker compose**

Docker compose is a tool which builds and runs multiple containers application. It contains a config yml file which **defines the application state and services** and with single command we can run all containers.

**DECLARATIVE APPROACH**

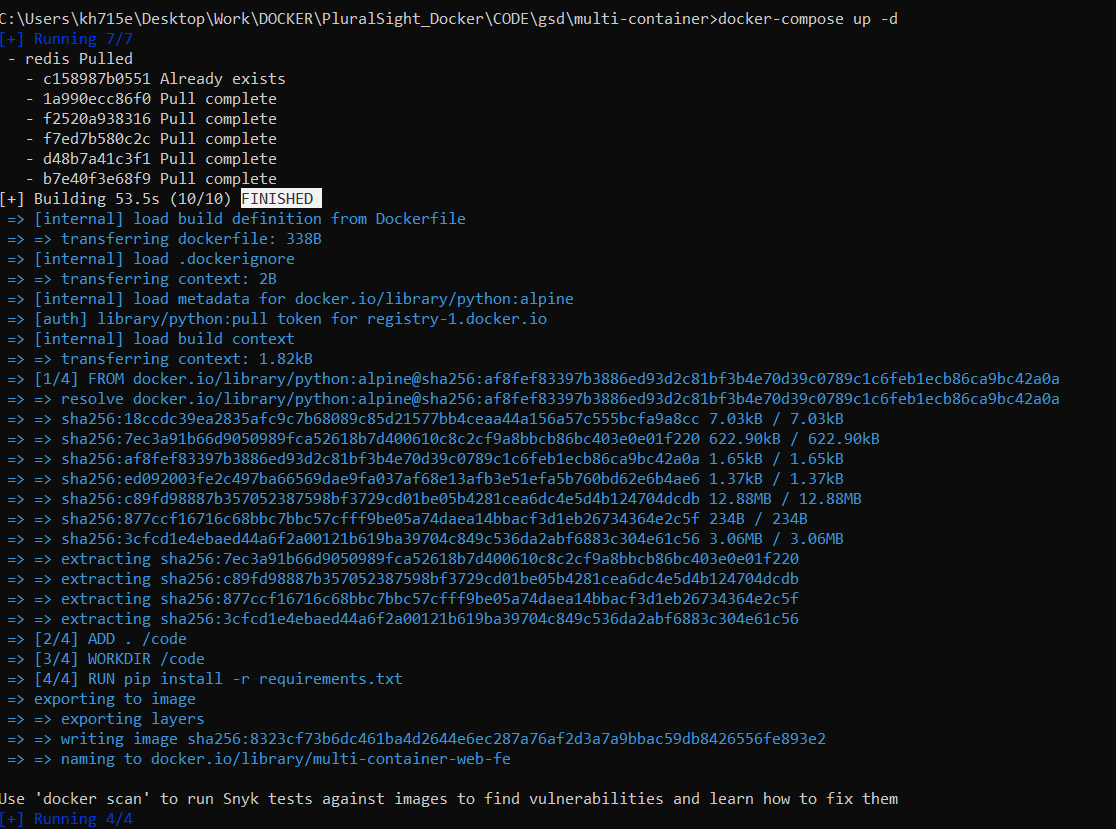


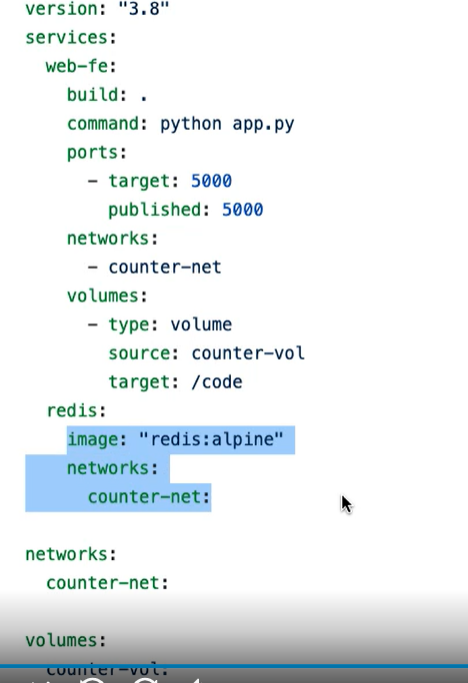
**We have a app.py which is an app code requires flask and talks to redis cache.**

**So 2 dependency flask and redis.**

**Docker-compose.yml file defining 2 app microservice.**

**After running docker compose file**



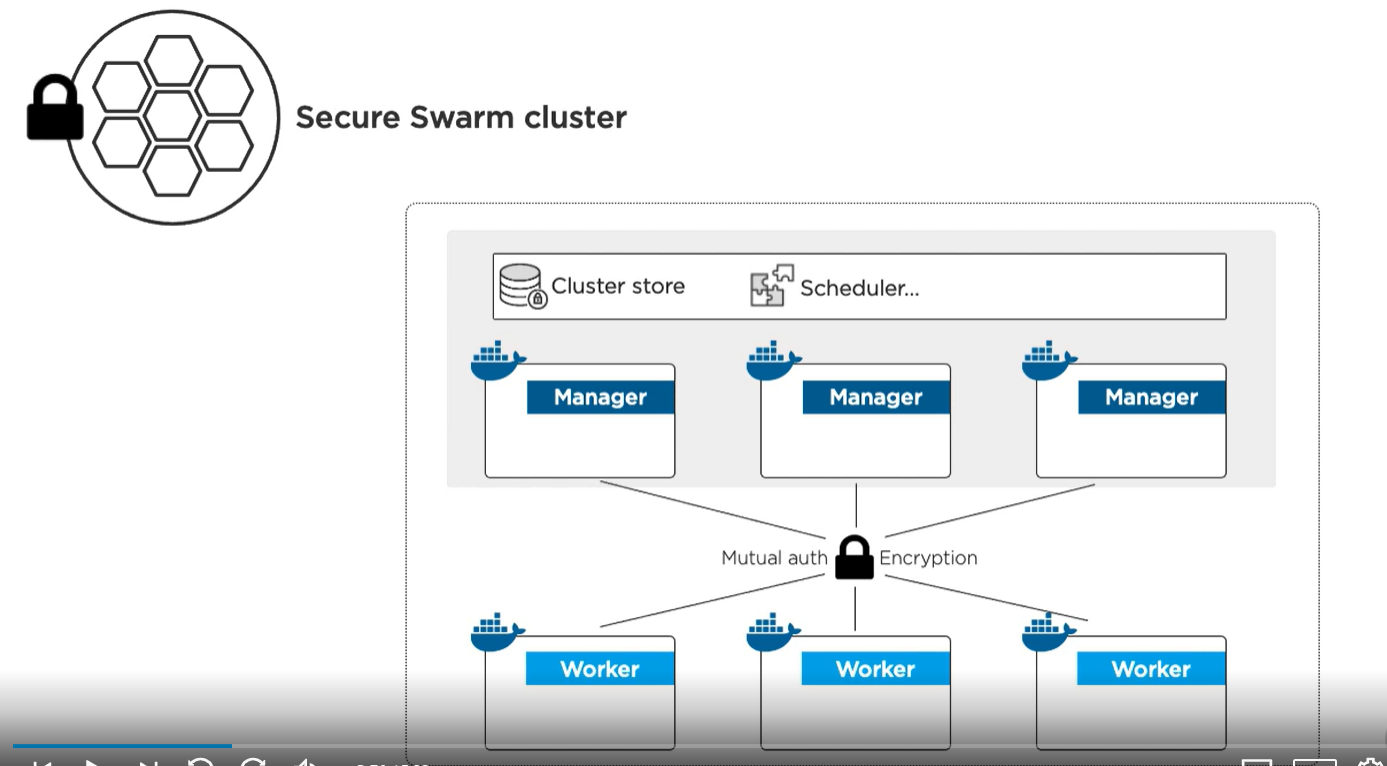


**Docker-compose up -d**

**DOCKER SWARM**

Docker swarm mode **clusters multiple docker host** into secure highly available cluster.

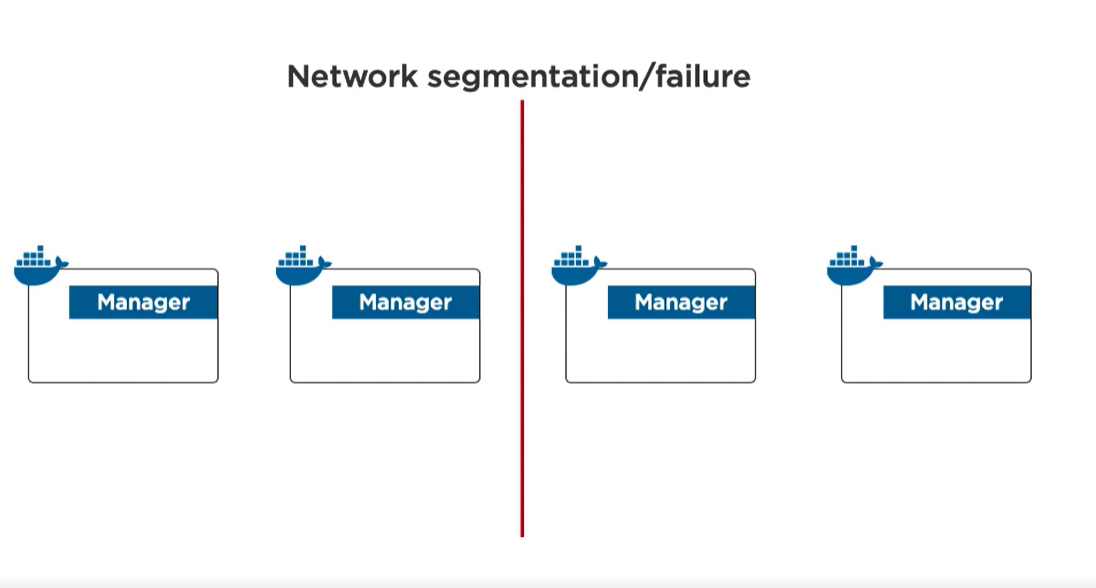
The cluster comprises of many managers and workers nodes.



Managers control the scheduling and state of cluster and apps its hosting.

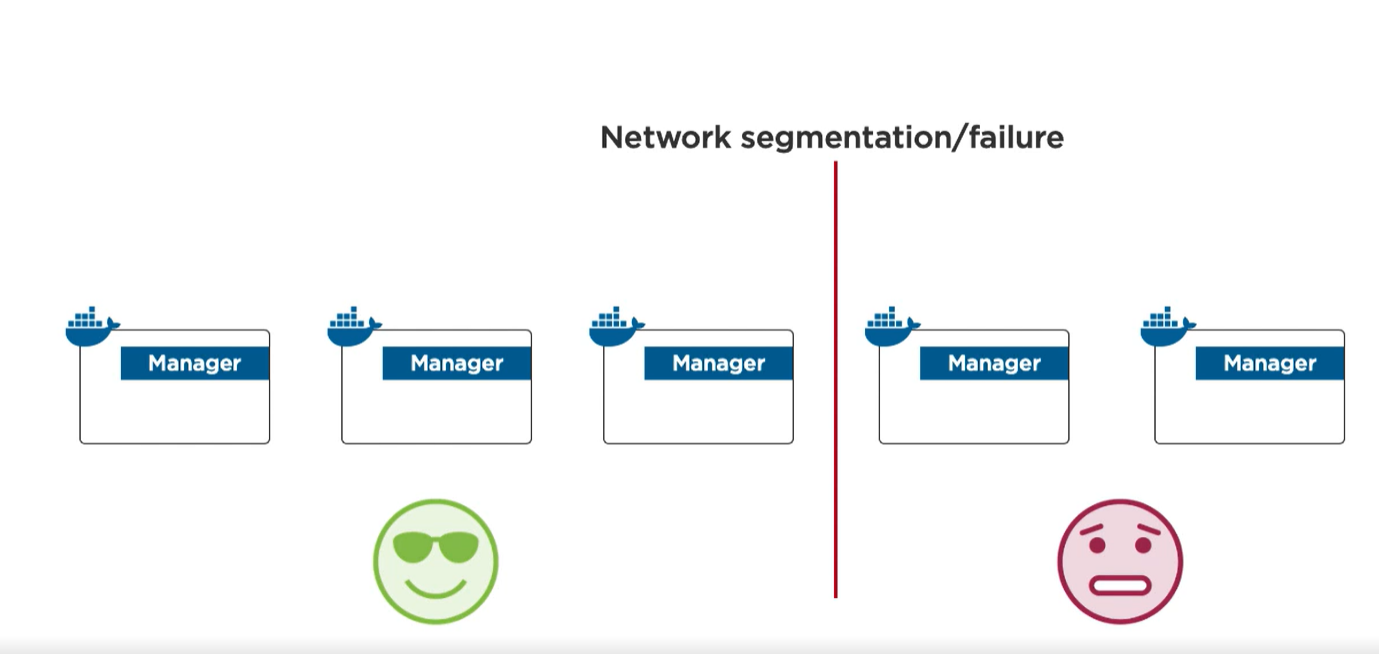
Recommendation is to have odd number of manager nodes.

**ISSUE WITH EVEN NUMBER MANAGERS**



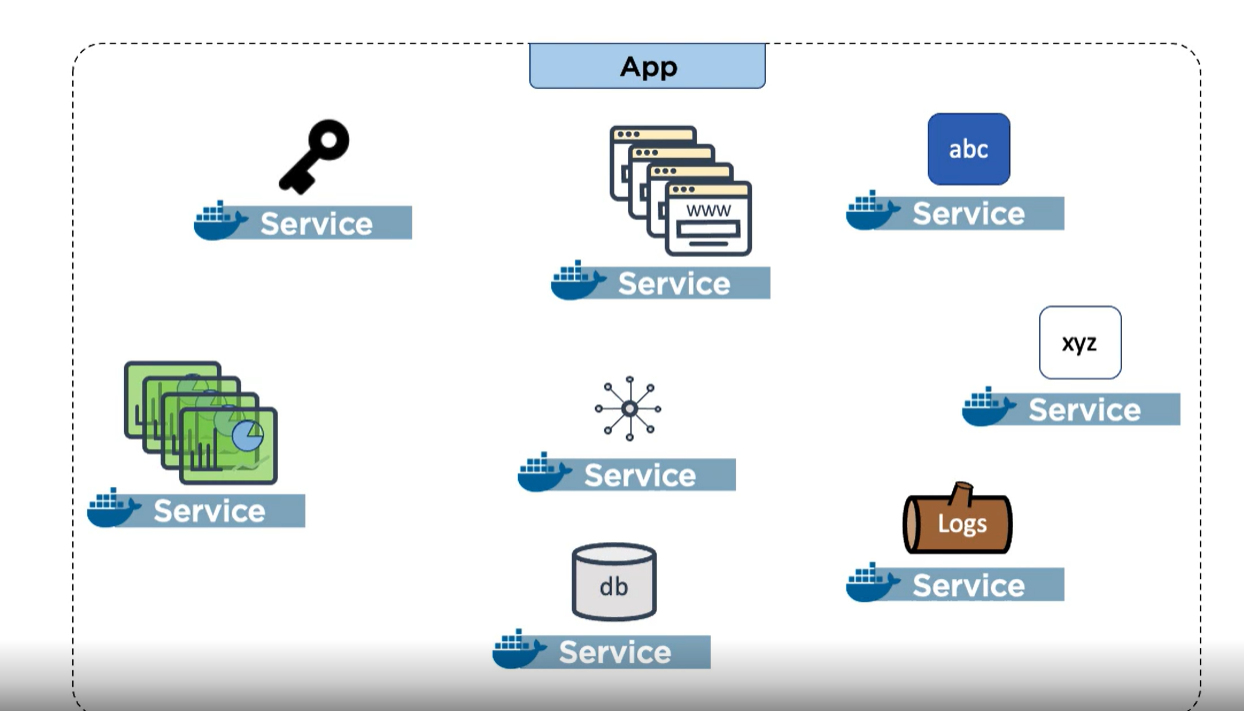
**Split brain condition-** If there is a network issue and nodes can’t communicate with each other and we have equal number of nodes on both side so no one has majority so cluster will be frozen.

**ODD MANAGERS**



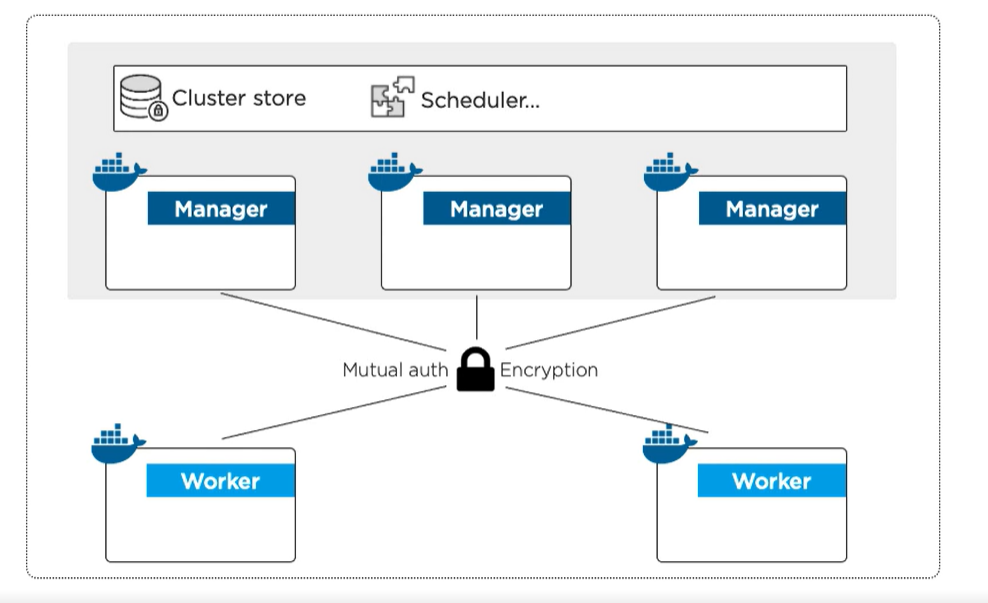
Here we have 3 nodes and has majority so it keeps the cluster open for updates.

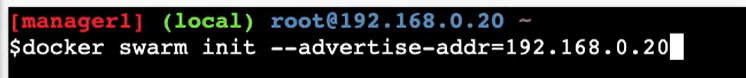
* The managers and worker nodes can be present anywhere on premise, on cloud etc .
* The only condition is they have docker installed and can communicate with each other over reliable network.
* If we are working on docker desktop then we can run swarm but our manager nodes will be 1.
* Docker swarm also unlocks docker services which can directly maps to microservices.



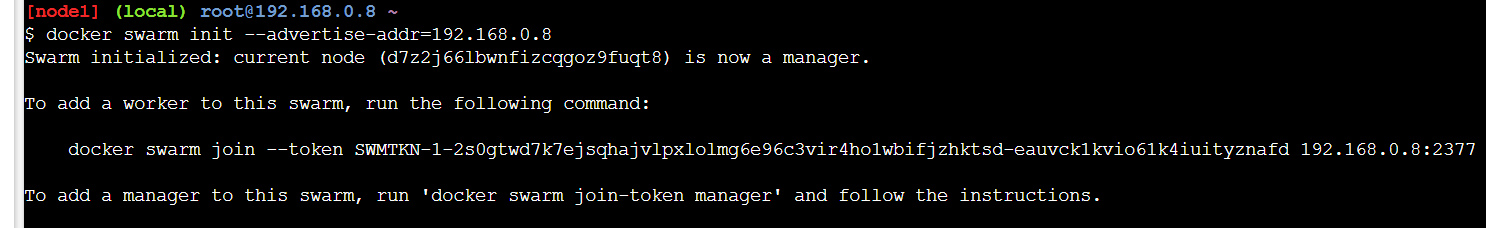
* **Docker swarm init** when we run this on docker desktop we will get 1 manager node

**TO VISUALIZE DOCKER SWARM WE WILL USE PLAY WITH DOCKER AS IT ALLOWS MULTIPLE DOCKER HOST.**

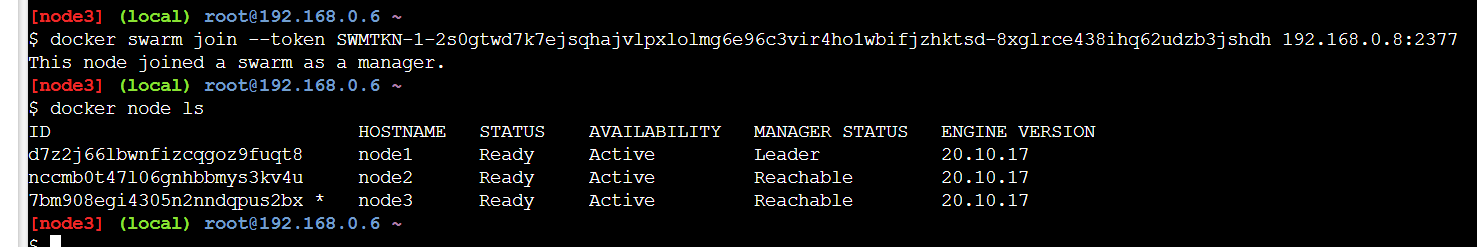




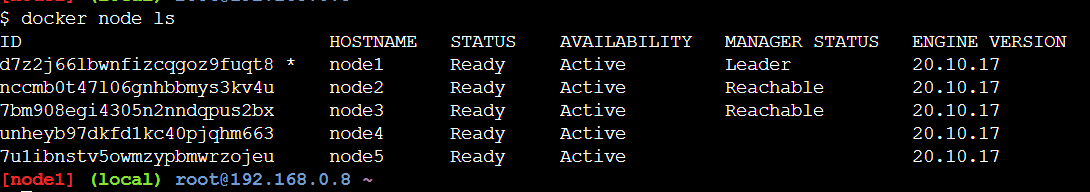
* **Creating 1st manager, --advertise-addr** tells docker which host IP to use for cluster communication.



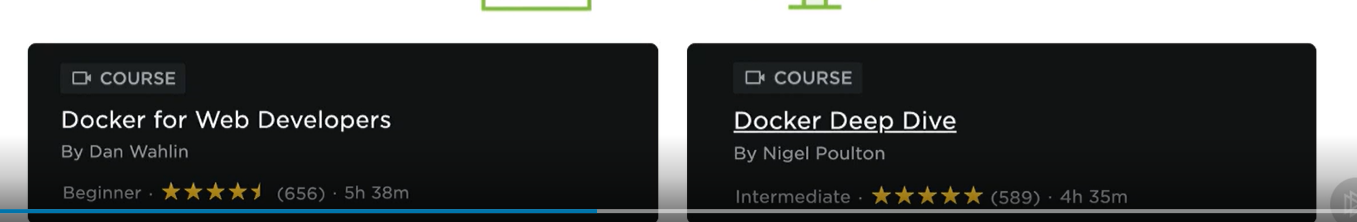
* Then run **docker swarm join-token manager** command to get join token command which we can run in another instance to create 2nd manager and join it to first.
* Paste this join command in different instance to create 2 more managers --- (docker swarm join --token SWMTKN-1-2s0gtwd7k7ejsqhajvlpxlolmg6e96c3vir4ho1wbifjzhktsd-8xglrce438ihq62udzb3jshdh 192.168.0.8:2377)
* docker swarm join --token SWMTKN-1-2i9s7spf1ptaxs15nqkwkkyh2ciy99pr5idf78d195j8s3f815-91xb5wuyw5j3trfiuvrso6u8d 192.168.0.8:2377
* docker swarm join --token SWMTKN-1-2i9s7spf1ptaxs15nqkwkkyh2ciy99pr5idf78d195j8s3f815-1875fjha10fldnh50f8dolqx4 192.168.0.7:2377

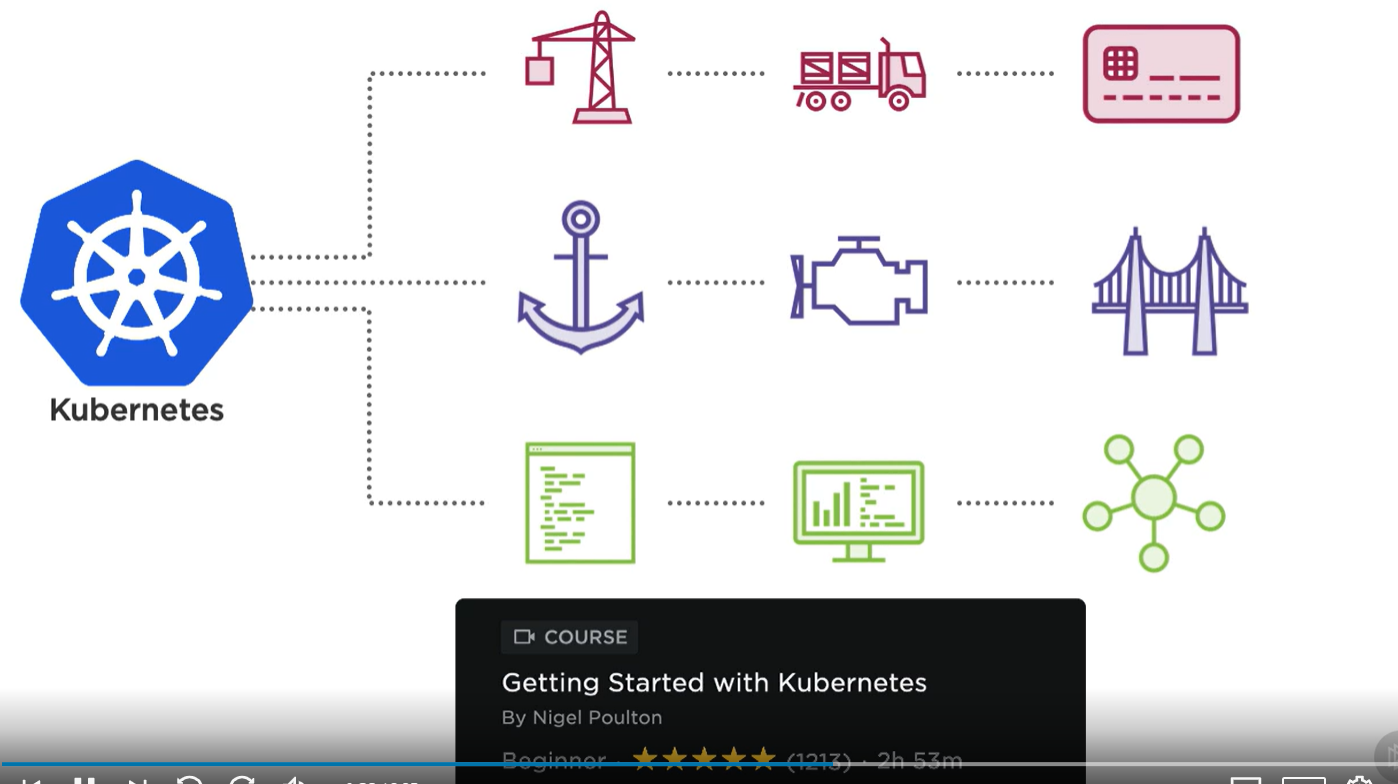


* Check nodes by docker nodes ls
* **TO add workers run docker swarm join-token workers** in the 1st node and then paste the join command in another instance to get 2 workers.
* **FINALLY WE HAVE 3MANAGERS AND 2Workers**



**FUTURE COURSES:**





**BOEING \_ LINKS TO DOCKER**

[**https://git.web.boeing.com/container/boeing-images**](https://git.web.boeing.com/container/boeing-images) **(git lab)**