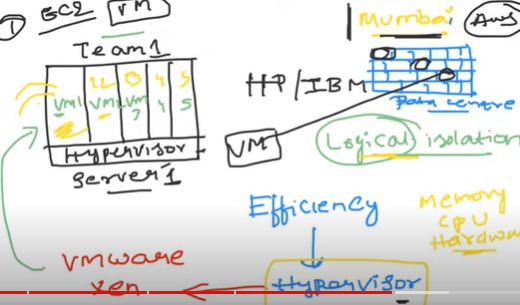
**Virtual Machine**

Breaking the physical server resources in to logical parts with the help of hypervisor (vm ware, xen) and making them as virtual machines for efficiancy.

-> Every VM has its own OS, cpu, memory, ram and one wont depend on other VM’s.

-> This is called logical Isolation.

-> The world of virtual machines managed by hypervisor either it is aws, gcp, azure.



->VM is advancement to physical server.

**Containers**

**(** [**https://github.com/iam-veeramalla/Docker-Zero-to-Hero/tree/main**](https://github.com/iam-veeramalla/Docker-Zero-to-Hero/tree/main) **)**

**->** docker container is advancement to VM’s

-> Docker is light weight because they don’t have complete OS like VM. Docker uses the host OS. That doesn’t mean the container runs without OS. They have minimal OS or BASE IMAGE.

-> A container is a package of application, app libraries, and system dependencies and any other shred libraries will be used from the host OS.

-> The snapshot or image of docker is less in size compare to VM and also containers are very easy to shift and transfer to other platforms like Kubernetes.

-> Docker engine convert the docker file in to docker image and image to container.

**Docker file -> docker image -> docker container (** docker engine does all these )

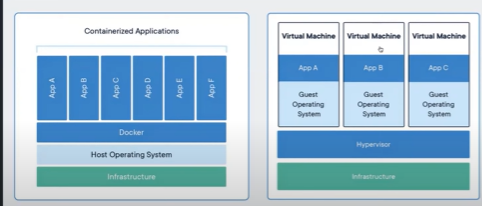
-> Docker is entirely dependent on docker engine, if engine goes down then no container will work which is known as single point of failure. To solve this we have BUILDA which creates the docker image.

-> Docker needs some system dependencies to form logical isolation between one container to another container. If there is no logical isolation between containers, one can enter in to all containers which brings security issues.

Basic folders in containers like /bin, /usr, /etc, /lib, /sbin, /var, /root form a logical isolation from one container to another container. A container cannot share these files and folders with any other container. These are not from kernel or OS but these are from container image.

-> application running on a container uses these files from the image we have and all the other folders will be taken from kernel or host OS. The size of a service or any tool is way less in image compare to its original one which is on the internet.

->we can run any number of containers in a server if the resources are left with host OS.



**Docker Commands**

**docker run -d ubuntu sleep 1000** -> to run the container in detach mode

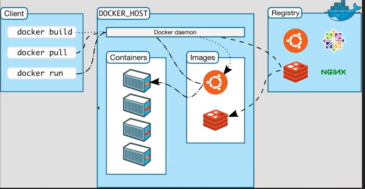
**docker run attach cont\_id** -> to bring back the running container from background.  
**docker stop cont\_id ->** To stop any running container

-> even after deleting the container if we want to save and persist the data inside the application which is running on the docker container, we map the data path to the external directory which is created outside the container and inside the docker host.

**Docker**

Containerization is concept of technology and Docker implements containerization.

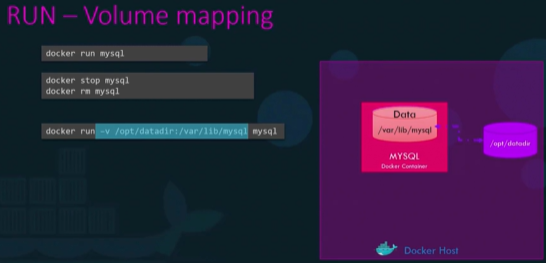
Docker is a **containerization platform** that provides easy way to **containerize your applications**, which means, using Docker you can build container images, run the images to create containers and also push these containers to container registries such as Docker Hub, Quay.io and so on.



-> **Docker Daemon** is heart of docker and is a process which listens to docker API’s and takes all the docker cli commands and executes. Daemon is the one which listens to your containers, networks, volumes.

-> Docker reduces the workflow of the application maintainence inside the server where we need to download, patching, config setting and many other manual things. It basically reduces the complexity of the workflow.

-> Docker registery is a hub where all the images are stored.



-> To display the prompt and give standard input from the user inside the docker container. Because docker container doesn’t have the terminal and it is un-interactive mode of communication.



**i = input**

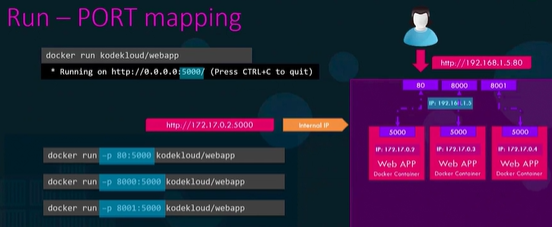
**t = terminal**

\*\* **port Mapping \*\***

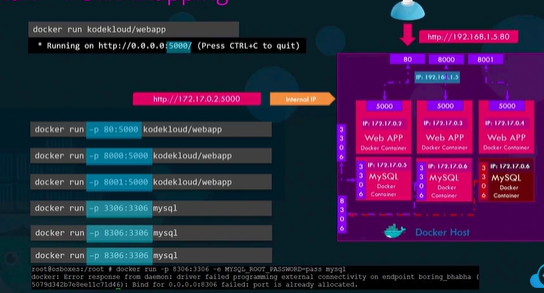
-> every docker container gets an ip by default and this is an internal ip which is accessed inside the docker host. Docker host ip and docker container ip are different.

-> every application has its own port for listening.

-> if user wants to access the application inside the container, we map the port of the docker host to the port of the application. We can run multiple instances of our application and map them with different ports on the docker host.



-> we cannot map same port on the docker host more than once.



\*\* **Advance docker features \*\***

**- > docker run ubuntu cat /etc/\*release\*** -> to see the version after running the container.

-> **docker run ubuntu:17.10** -> to run specific version.

-> docker run Jenkins/Jenkins

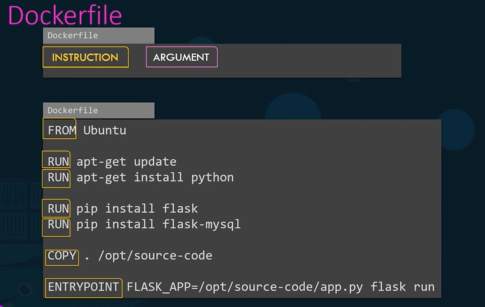
docker run -p 8080:8080 jenkins/jenkins

docker run -p 8080:8080 -p 50000:50000 --restart=on-failure -v jenkins\_home:/var/jenkins\_home jenkins/jenkins:lts-jdk11

This will automatically create a 'jenkins\_home' [docker volume](https://docs.docker.com/storage/volumes/) on the host machine. Docker volumes retain their content even when the container is stopped, started, or deleted.

**\*\* Docker Images \*\***

**->** docker file written in instruction and argument format.

****

Every docker image must be based on another image either OS or other image that was created based on OS.

-> All docker files must start with **FROM** instruction.

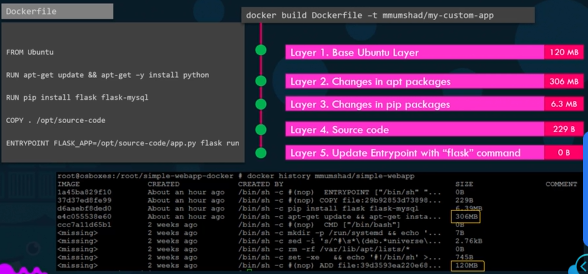
-> **copy** instruction copies source code file from local machine to docker image.

-> **ENTRYPOINT** allows us to specify a command that will be run when the image is run as a container.

**Layered Architecture:**

-> When docker builds the images, it builds in layered architecture. Each of the instruction in docker image with just the changes from the previous layer. Since each layer stores the changes from the previous layer it reflects in the size as well.

**docker history image\_name** -> to see the size of the layers from image.



All the layers built are cached. Layered architecture helps u to build the docker file from that particular layer in case if it fails or if u want to add new steps in the image u don’t want to build from the start. This way re-building your image is faster and no need to build the image from start each time we make any changes.

-> we can containerise almost everything like browsers, utilities, applications and all.



**\*\* Building a docker image \*\***

1. create a separate directory for storing the application and docker file.

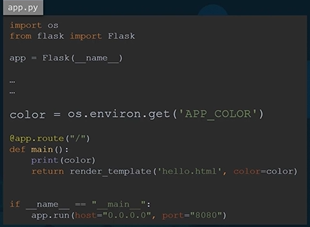
2. create a docker file

3. docker build . -t nikhilit/image\_name ( image will be created )

4. docker login

5. docker push nikhilit/image\_name

\*\* **env variables \*\***

****

****

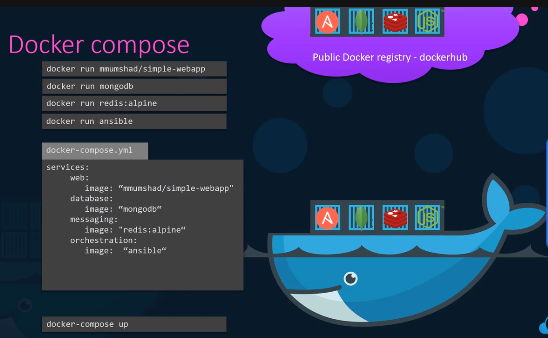
**$docker inspect cont\_name ­**-> to see env on the docker file

\*\* **Docker Compose \*\***

**->** for setting up complex application running multiple services, we use docker compose where we create configuration file in yaml format putting those services in this file.

-> However these are applicable running containers in single docker host.

-> **Links** is a command line option which links 2 containers together

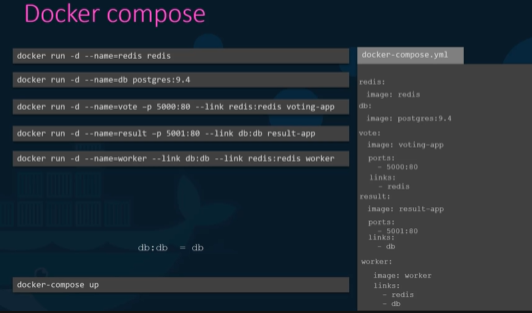


**Docker Links:**

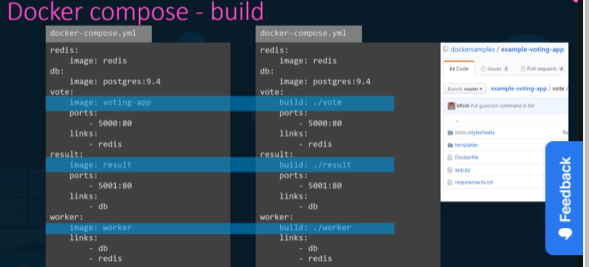
****

Each service running on a diff container. Every service has dependency on each other to make this application work. We have to link all the services which are running in the containers to other containers.

**Docker compose**

****

**-> Docker compose up** to bring the application stack to work.

****

When docker image is not available in the hub registry we can make build the image directly inside the compose file.

**Docker compose versions:**

**Version 1 :** we cannot specify deploy the containers in other networks other than the default bridge network in this version.

If we have any dependency situation where db container should run first before the app, we cant specify that here.

**Version 2:** we can specify all stack information under services section.

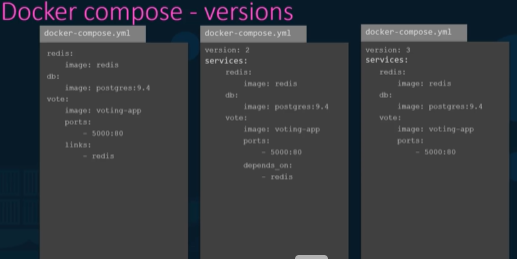
-> docker compose automatically creates a dedicated bridged network for this application and then attaches all containers to that new network. All containers are communicate well with each other with service names.

-> we don’t need to use links in version 2 of docker compose.

-> **depends on** feature helps the voting-app to start only after the starting of the redis service.

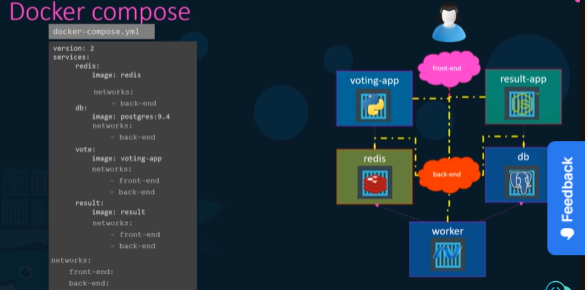


**Version 3:**

****

User generated traffic is front-end network and application generated traffic is backend network.

Networks section is added in the end of the compose file.

****

**\*\* Docker Engine \*\***

When we install docker on linux host , we are installing 3 components( docker CLI, docker rest API, docker aemon )

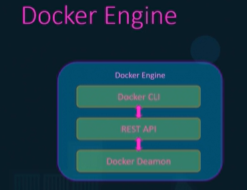
**Docker Daemon** is a background process that manages docker objects such as images, containers, volumes and networks.

**Docker Rest API** server is the API interface that programs can use to talk to the daemon and provide

Instructions.

**Docker CLI** is a command line we use for running containers, images, stopping them etc.

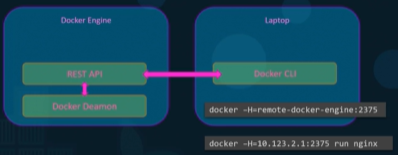
It uses the rest API to talk with docker daemon



( <https://docs.docker.com/engine> )

( <https://docs.docker.com/engine/api/> )

-> Docker CLI can be on a diff system and still can run a container on docker engine which is on diff system with port and IP address.( **Docker -H=10.61.76.169:2356 run nginx )**

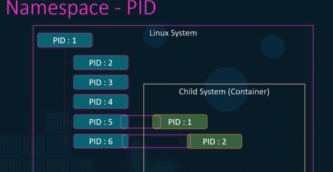


**Containeraization:**

Docker uses namespaces to isolate workspace. Namespaces like PID,network,mount,…all these are created in their own namespace there by providing isolation between containers.

Eg: Namespace PID:

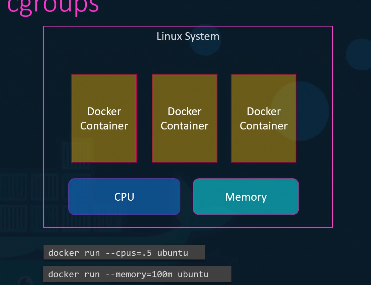
-> processes running on linux host starts with process 1. Same like process running on a container starts with pid 1. But on a linux system no two processes should have same PID. So to make separate PID for both in container and linux, with Namespaces each process can have multiple process ID’s associated with it. PID 1 in the container is just another process on the linux host. But for container it’s the first process. PID of a service inside a container will be diff with the PID in docker host.

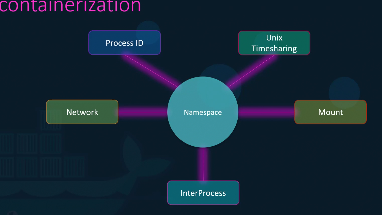


**Cgroups:**

**->**the docker host and containers share the same system resources such as cpu and memory.by default there is no restriction on how much resource a container can use. So a single container may end up using all the resources of underlying host. To restrict the amount of CPU and memory for a container , docker used Cgroup to restrict amount of hardware resources for each each container.

( <https://docs.docker.com/config/containers/runmetrics/#control-groups> )





( <https://medium.com/@kasunmaduraeng/docker-namespace-and-cgroups-dece27c209c7#:~:text=Docker%20uses%20namespaces%20of%20various,Process%20ID> )

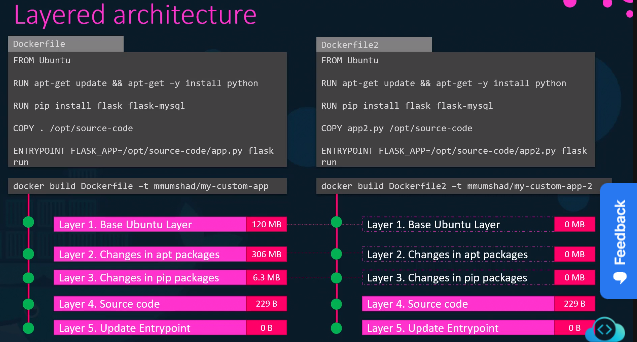
**Docker Storage: (** where and how docker stores the data and how it manages the file systems of conatainers )

-> on installing docker on a system it creates a folder structure **/var/lib/docker** where all the files related to images, containers, and volumes running on the docker host are stored here.



**Layered Architecture**

The advantages of docker layered architecture is, it re-uses the layers which built in other images from cache and quicky builds the layers which are new.



Here the first 3 lines of both docker files are same. While building the image of the second docker file, docker takes the layers from the cache which are built in first docker image and uses in this file which saves time and disk space.

**Copy on write Mechanism**

-> Files in the **image layer** read-only. once u build the image we cannot modify the files inside image layers. So, the single image can be used by multiple containers. If we need any modification of files we can do it in **container layer ( read-write ).** But before we save the modified file, docker saves the copy of the image and it will be stored in the **image layer**. Now we can make modifications in container layer. And all the future modifications are done in **read-write layer.** The image in the image layer remains the same until we rebuild it again with ” docker build “.



So, after modification of files in the container layer, they will be deleted once the containers is destroyed. So, to persists these changes we use **volumes.** if we wish to persists this data generated by the container, we can add **persistent volume** to the **container.**

**There are 2 types of mounts : 1) Volume Mounting 2) Bind Mounting**

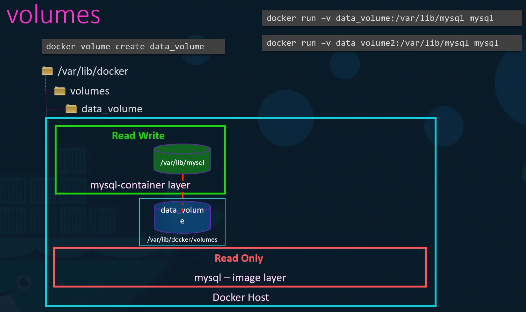
**(** [**https://docs.docker.com/storage/volumes/**](https://docs.docker.com/storage/volumes/) **)**

**(** [**https://docs.docker.com/storage/**](https://docs.docker.com/storage/) **)**

**Volume Mounting**

**-> Docker volume create data\_volume** -> creates a new volume named data\_volume

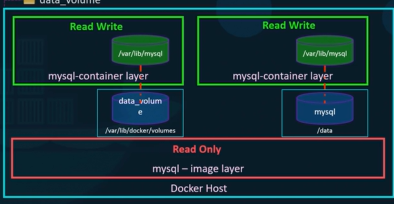
-> **docker run -v data\_volume:/var/lib/mysql mysql** -> creates a container and mounts the volume to the container.



Now the data which will be generating inside the container will be added to the volume which we created in the docker host and it will be persisted when after docker container is destroyed.

**Bind Mounting**

If our data is already at another external storage on docker host and if we want store our data from the container on that volume and not in default which /var/lib/docker, we run a container with the external storage path on the docker host ( **docker run -v /data/mysql :/var/lib/mysql mysql )** -> old style

****

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

**source=** location of the volume on host

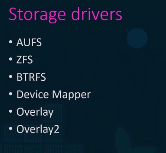
target=location of the volume in container

**Storage Drivers**

**(** [**https://docs.docker.com/storage/storagedriver/**](https://docs.docker.com/storage/storagedriver/) **)**

-> Storage drivers are responsible for doing all of these operations like **maintaining the layered architecture, creating a writable layer, moving files across the layers to enable copy and write** etc..

-> Docker uses storage drivers to enable layered architecture.

 The selection of the storage drivers depends on underlying OS. Docker chooses the best storage driver based on the OS.

-> **docker run -d --name mysql-db -e MYSQL\_ROOT\_PASSWORD=db\_pass123 mysql** -> to set the password for db and run a container named mysql-db

**Docker Networking**

When we install docker it creates 3 networks automatically. **1) Bridge 2) none 3) host**

-> bridge network is the default one which a container gets attached to.

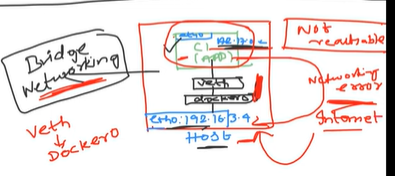
**Docker run ubuntu --network=host ->** command to associate with other network.

**Bridge Network**

**(** [**https://www.youtube.com/watch?v=xrUGEoUpa3s&list=PLdpzxOOAlwvLjb0vTD9BXLOwwLD\_GWCmC&index=6**](https://www.youtube.com/watch?v=xrUGEoUpa3s&list=PLdpzxOOAlwvLjb0vTD9BXLOwwLD_GWCmC&index=6) **)**

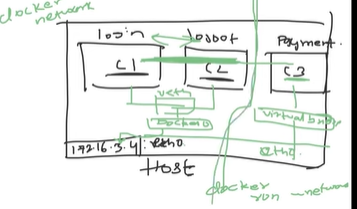
-> It is a default networking in docker. The container having a diff subnet and host having a diff subnet and both can talk with a bridge which is called virtual ethernet( **veth->docker0)**. For user to access the application running inside the container, the host and container should be connected.

-> Bridge network is a private internal network created by docker on the host, and all containers attached to this network by default. These containers can access each other with internal ip’s.



**Custom bridge networking**

Our application will be having multiple feature applications which we run each single one in different containers. If we want to isolate these containers from each one we can create a custom bridge netwotk where one part of our application is isolated from other.



**$docker network create secure-network**

**Host Network**

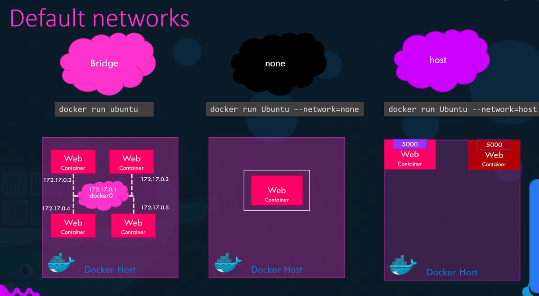
-> In host networking, by default docker connects containers to host ip range and container can access host by default. Here both cidr ranges of containers and host are same. But this is not secure as access having to host will also be having the access to the containers.

-> To access these containers externally from the outside world, map the ports of these containers to the ports of the docker host. When we map containers to docker host the isolation b/w container and host is lost.

-> once we map container to the host, u cant run multiple container on the same port and same host as the port is common to all container in host network.

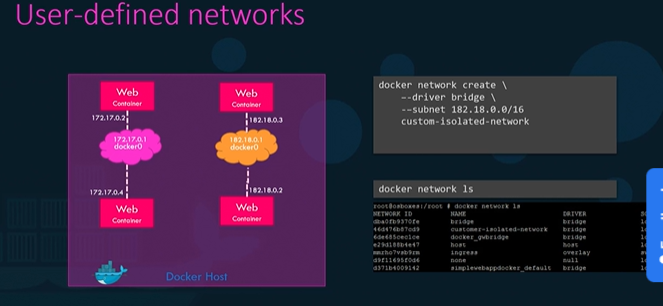
**None network**

-> with none network, the containers are not attached to any network and doesn’t have any access to external network and other containers. They run in isolated network.



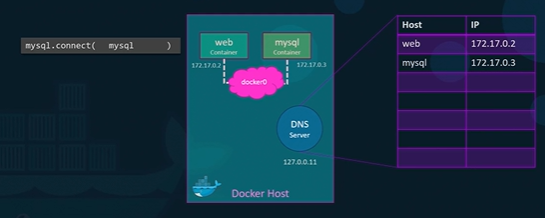
**User-defined network ( custom bridge network )**

-> if we want to isolate the containers within the same docker host. By default docker creates default bridge network. We can create our own internal network and map the containers to it.



-> To see the ip address and network assigned to the existing container, **docker inspect cont\_name**

**->** Containers can reach each other with ip addresses and their names. But ip addresses will change once the system reboots. So we can connect with other container with their names. All containers in the docker can resolve each other by their names. Docker has built-in DNS server which runs on 127.0.0.11



-> Docker containers are isolated with each other in host, because docker uses network name spaces that creates separate name spaces for each container. It then uses virtual ethernet pairs to connect containers together.

**Advance Docker networking**

-> Each docker host has its own private internal bridge network allowing the containers running on each host to communicate with each other. Containers across the host has no way of communicating with each other unless we publish the port on them and set up routing. We can do it by overlay network.

**Overlay network**

-> with docker swarm we can crate a new network type called overlay which creates an internal private network that spans across all the hosts participitaing in the swarm cluster. We can then attach the containers or services to this network.

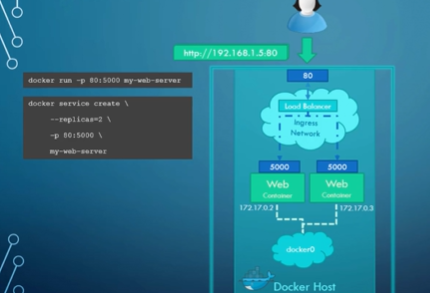


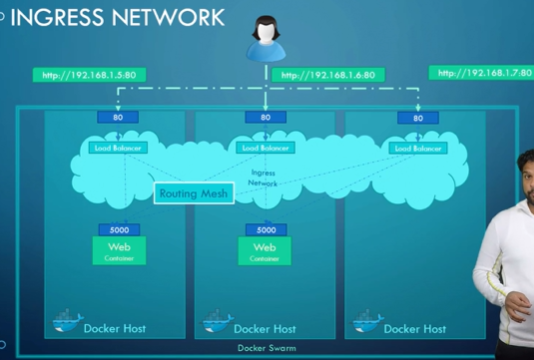
**Ingress Networking**

**$docker run -p 80:5000 my-web-server** -> to map the port of service inside container to the docker host.

-> when we are working with swarm cluster where we will be having multiple nodes or hosts. We cannot have multiple containers attaching to same port but, Ingress networking has builtin load balancer which redirects the traffic to the mapped ports of the host.

-> we we create docker swarm, ingress network will create automatically.





-> if we have cluster with 3 nodes and our services are running in 2 nodes and 3rd node is empty. Since it is cluster, user can access any node with ip of the container since all containers are part of the of the cluster. But user cant access the 3rd node as there is no service running. But with ingress networking, it receives any request made to the node and redirects to the nodes where the service is running which creates routing mesh. Routing mesh helps in routing the user traffic that is received on a node that isn’t even running the service to the node where the service is running.

**Commands**

**Docker network ls** -> to check no of networks exixts.

**Docker inspect cont\_name -**> to check network type of container

**docker network inspect bridge** -> to check about bridge network

**docker run --name alpine-2 --network=none alpine** -> to run container with image and attach it to none network.

**docker network create --driver bridge --subnet 182.18.0.1/24 --gateway 182.18.0.1 wp-mysql-network** -> To Create a new network named wp-mysql-network using the bridge driver. Allocate subnet 182.18.0.1/24. Configure Gateway 182.18.0.1

**docker network inspect wp-mysql-network** -> to inspect the created network

**\*\*\* Commands and Arguments in docker \*\*\***

-> unlike virtual machines, docker doesn’t host operation system to be alive all the time. They just depend and use OS as long as the container runs with an application active. Containers are meant to run a specific task or a process. Once the request is processed, container exits. **CMD** defines the program that will run on a container when it starts.

-> To start a container when it exists, we need to append a command to the **docker run** command.

