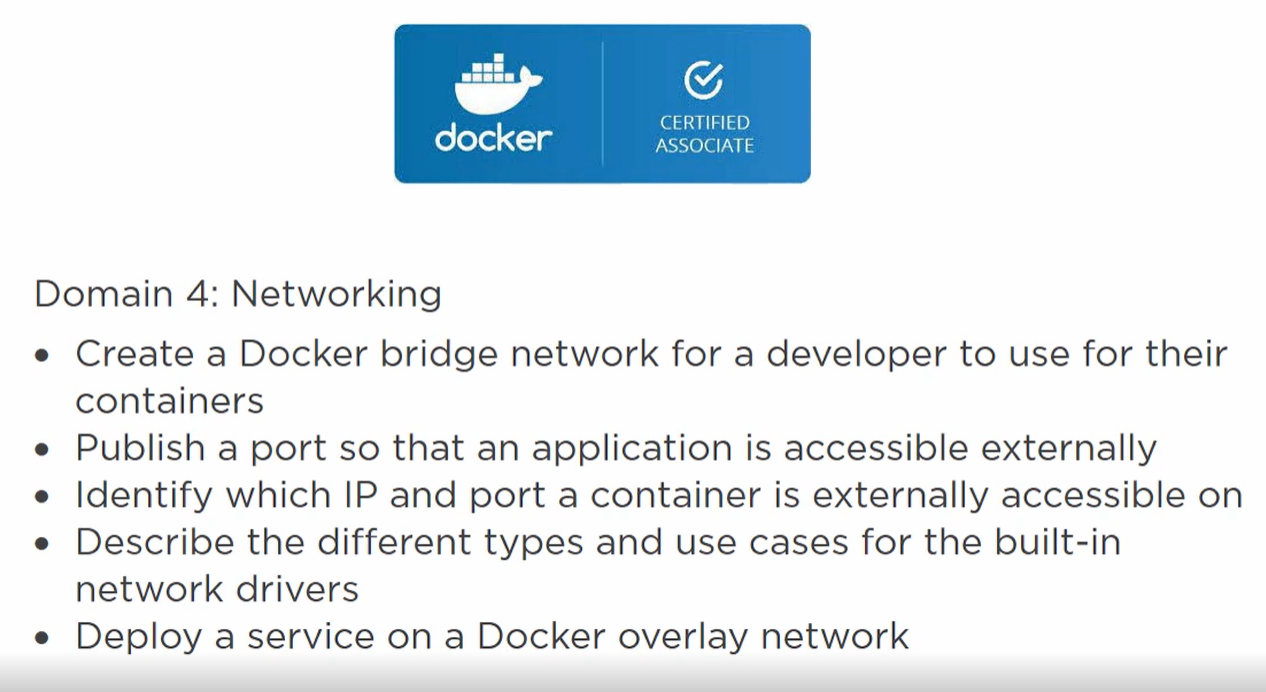
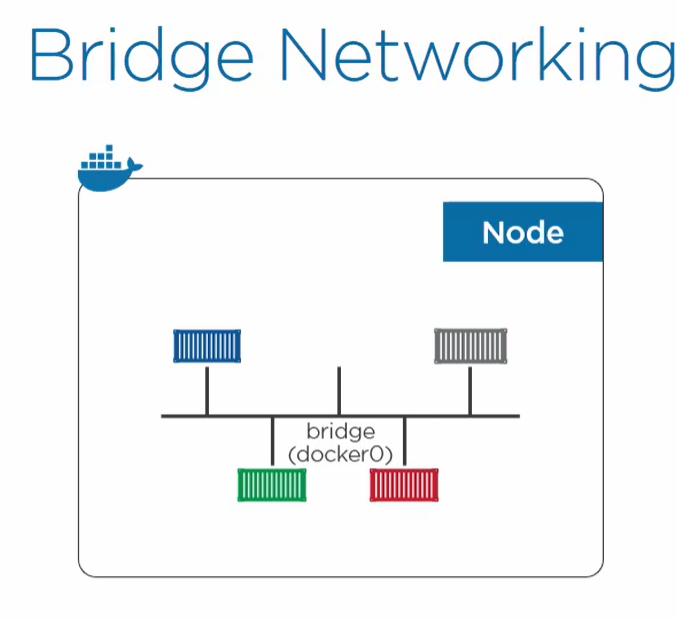
**NETWORKING**



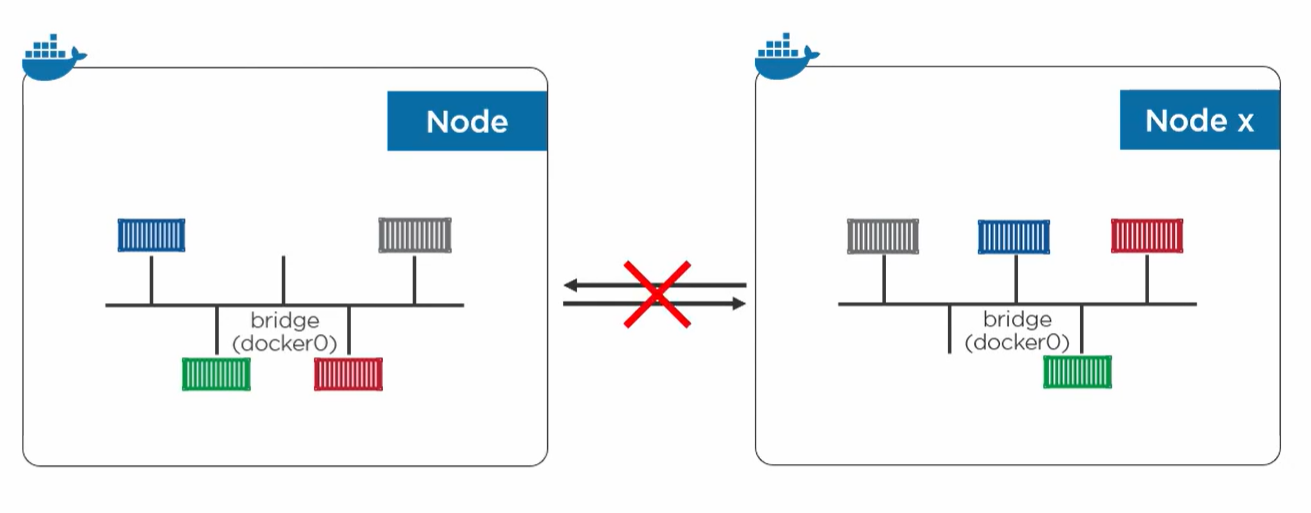
Containers needs to talk to each other and sometime it needs to talk to VM, physical as well as internet.

1. **BRIDGE NETWORKING** (Single host network)

* In linux its called Bridge driver
* In windows called nat driver
* It is turned on by default



We have a host in which multiple containers are running, so a bridge network is created and each container is given an IP address. Containers can talk to each other.



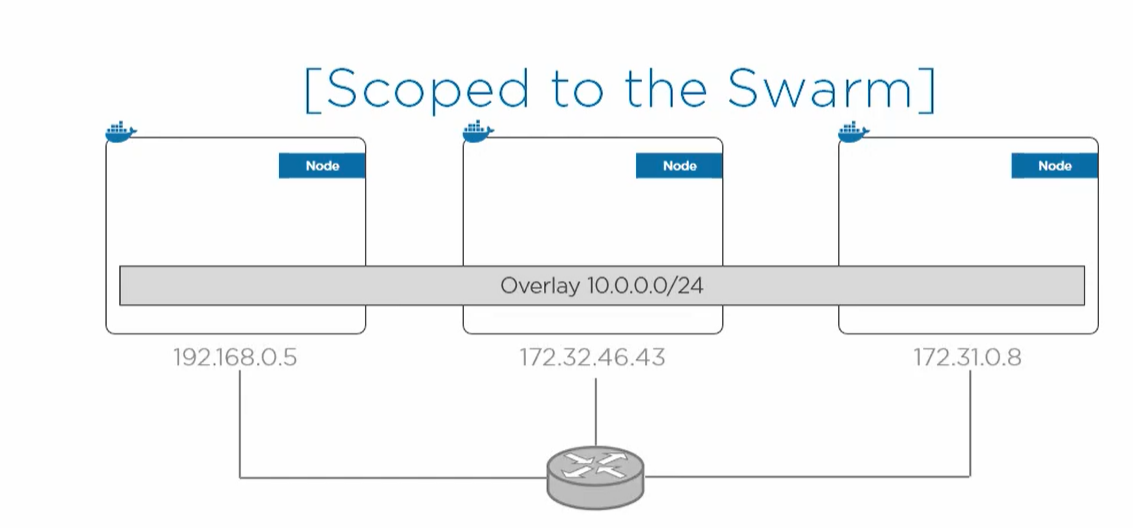
Each of the bridge is isolated. So container of one host cannot talk to container of other host with IP to IP connection.

The only way to connect is **port mapping**.

1. **OVERLAY NETWORK (Container only)**

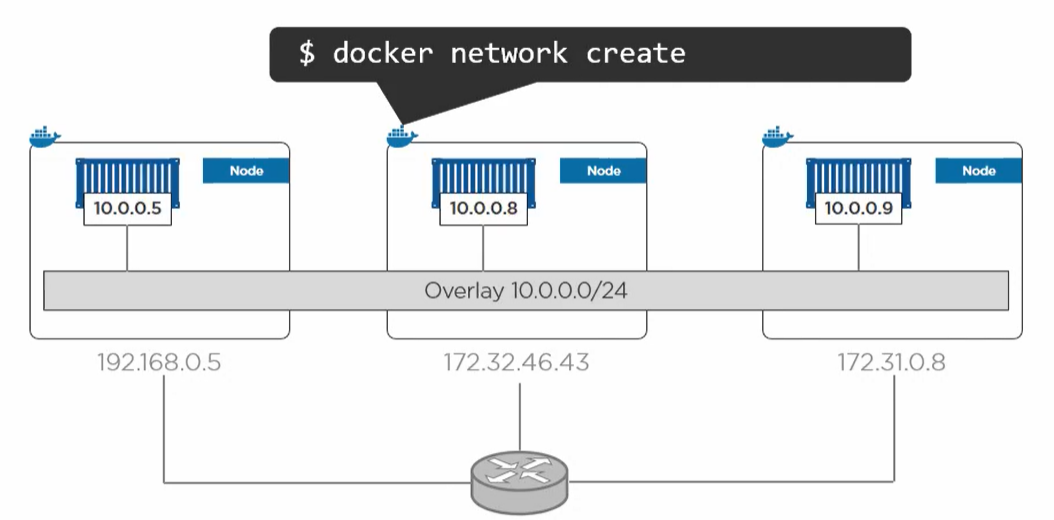
Instead of isolated bridges scope to single host in bridge network**.**

**It is a single layer2 network spanning multiple host.**

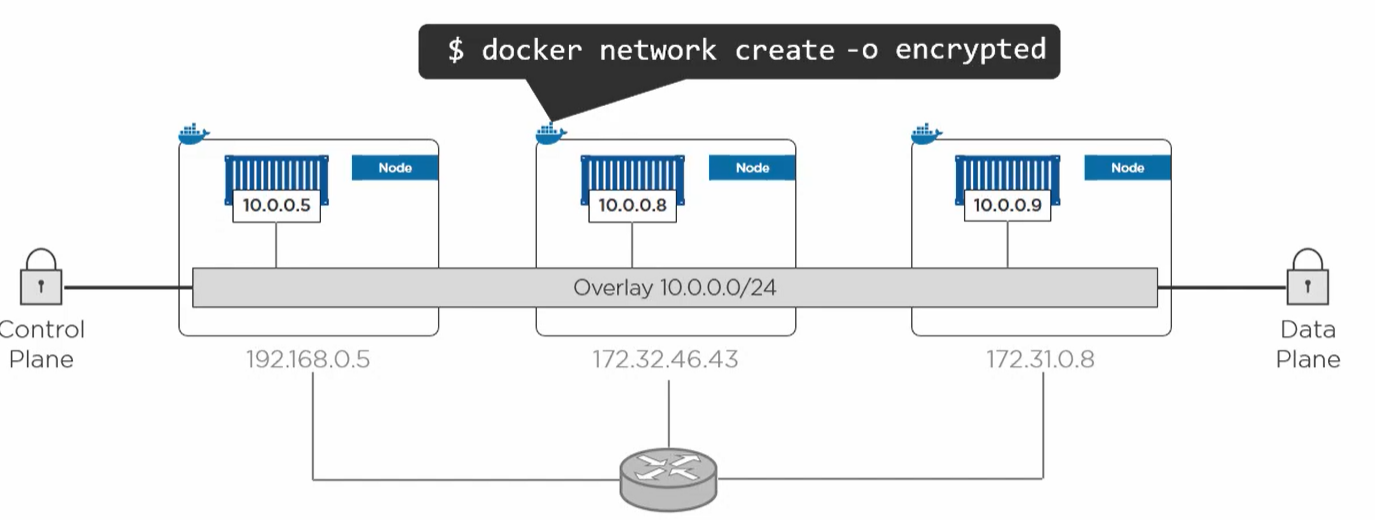


**$ docker network create**

**A single command to create a network and then we can attach our container to it.**



**Control plane and Data plane both are encrypted.**



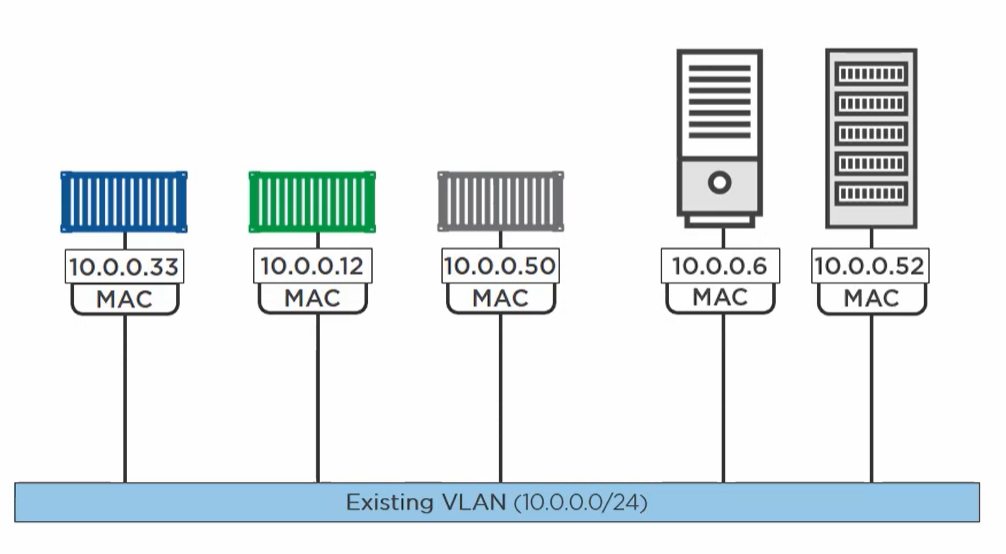
1. **MACVLAN OR Transparent driver windows**

If container needs to connect to VM and physicals then we use **MACVLAN**.

It gives container its own IP address and MAC address on existing network.

But it requires **promiscuous mode** on host network.

**If we are working on cloud the providers don’t allow promiscuous mode.**



**BRIDGE EXAMPLE**

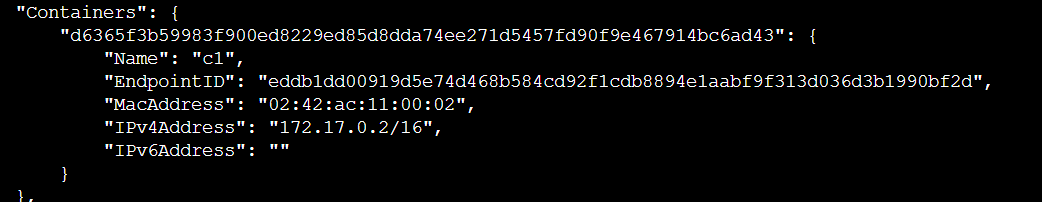
Suppose we have a swarm of 3 manager node and when we do

**Docker network ls** (in any one manager)

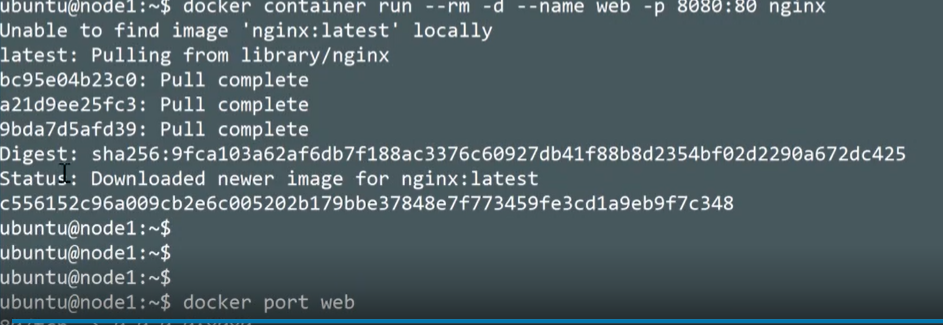


We see a bridge driver network so if we create a container in this node it will go to this network.

If we create a container by default this container will run in this bridge. **Docker inspect bridge**



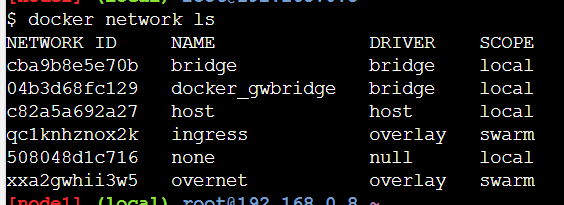
But this container is exposed to this bridge only. In order to make it available to outside world we need to do **port mapping** in case of bridge network.



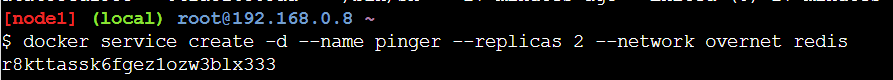
That means 80 inside a container is mapped to 8080 of all instances of swarm.

**OVERLAY example**

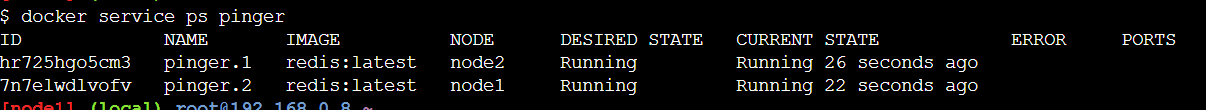
Docker network create -d overlay overnet



Scope swarm means this network is available to every host in swarm.



So we are creating a service which has 2 replica container running over overnet network on node 1 and node 2.



So we can inspect the network on node 1 and node2 we will have containers on both node.

Both containers can communicate with one another. We can ping one container inside another.

**NETWORK SERVICES**



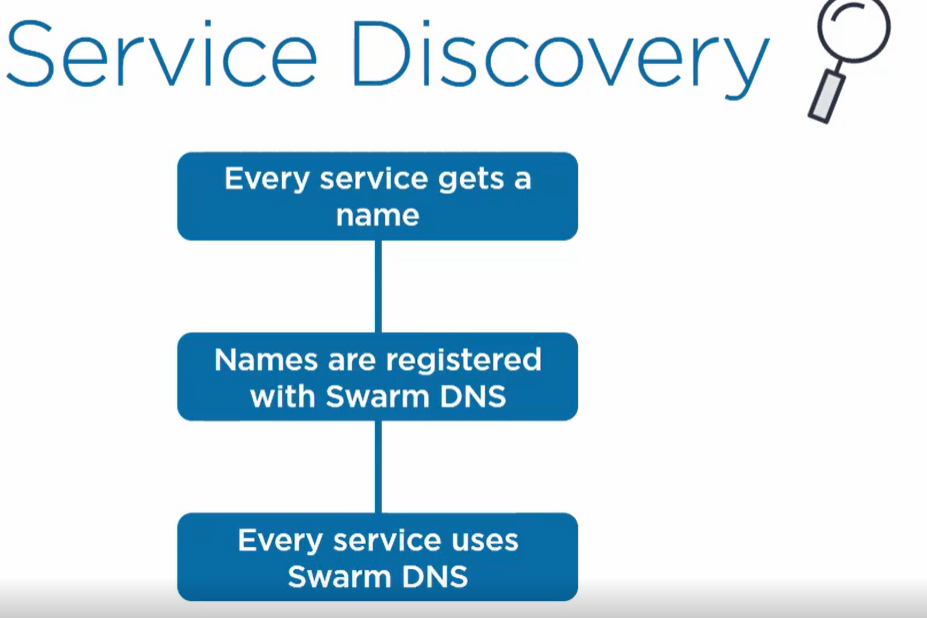
**Service discovery** – Locate services in a swarm.

**Load Balancing** - Access service from anywhere in the swarm even if service is not hosted on the node.

Service Discovery –

* Every service gets a name
* The name is registered with DNS on the swarm
* Containers in the service uses swarm DNS.

All services on a same network are pingable by name.



Example:

We create a swarm with 3 manager nodes:

1. First, we create an overlay network called overnet

**Docker network create -d overlay overnet**

1. Once we create an overlay network that will be shared across all nodes in the swarm.
2. Now after network creation we create a service of redis with 3 replicas over overnet network.

Docker service create -d –name ping –network overnet –replicas 3 redis sleep 1d

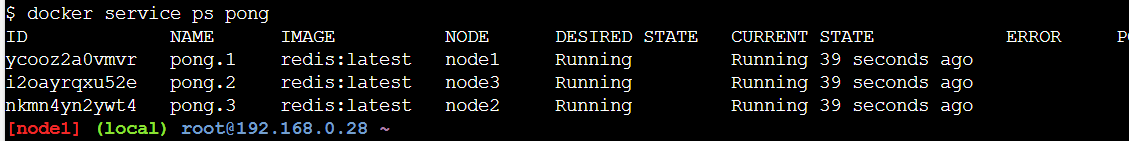
1. It will create a service with 3 container load balanced among all 3 nodes







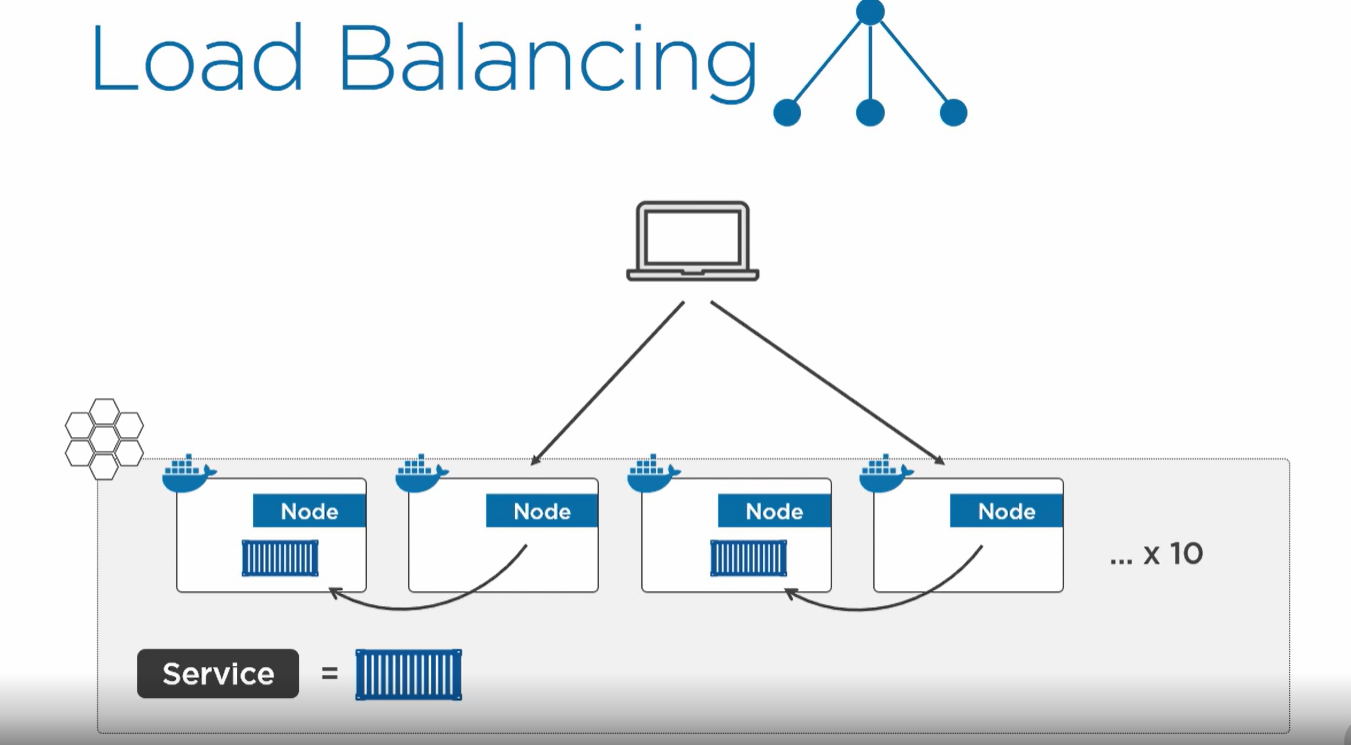
1. We create another service name pong



1. Now when we exec inside a container **we can ping a service by its name**

Ping pong

As these both services are on same overlay network they can find each other but services on another network can’t find each other.



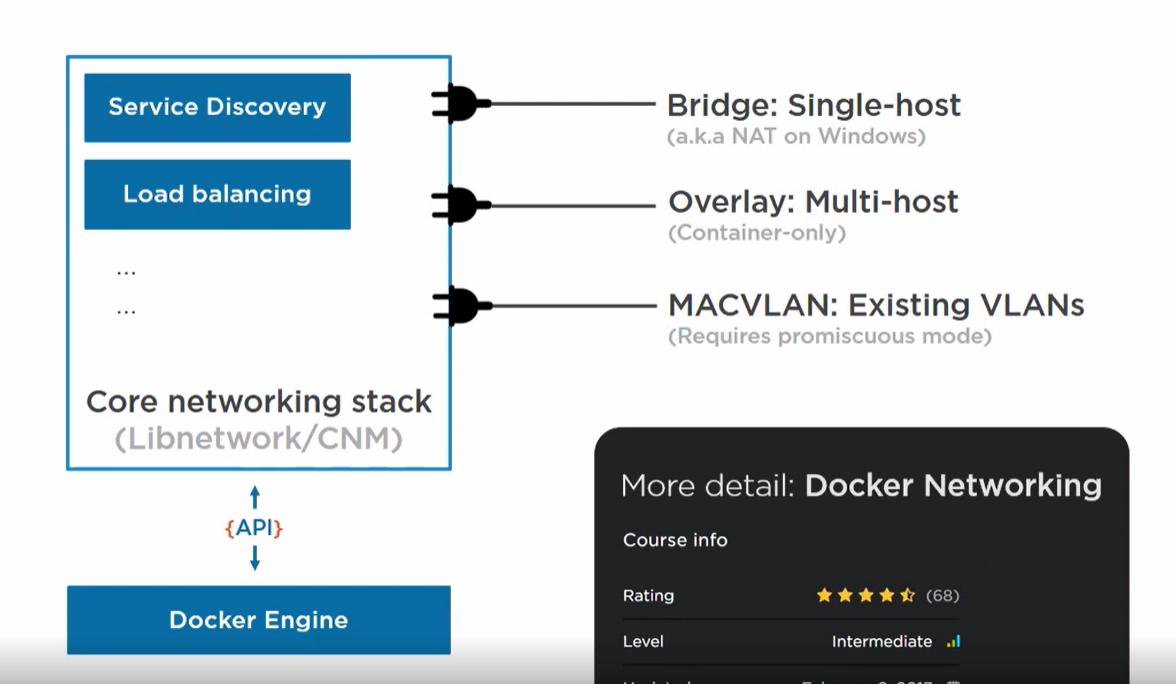
Even if we are hitting nodes which don’t have service replica it can access the service.



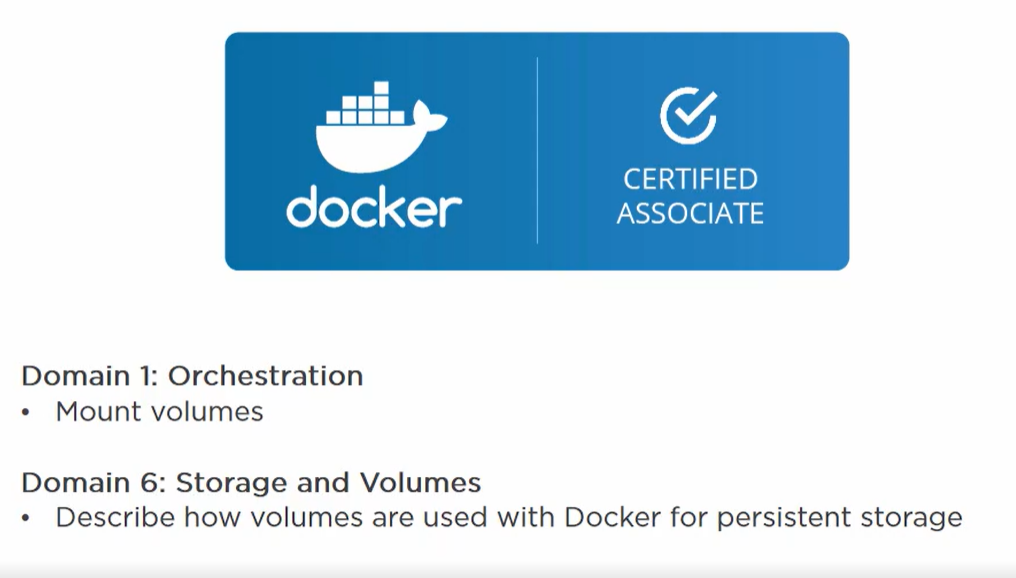
Even if the service is on node1 we can access it on node 2 and node 3 using 8080 port



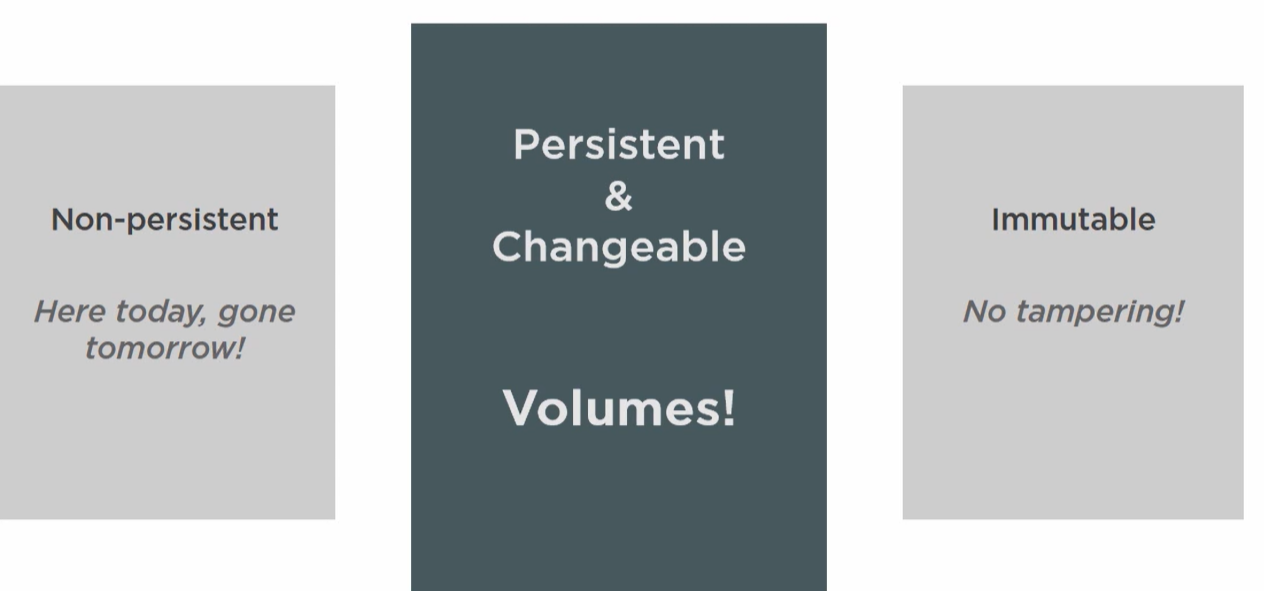
**Every node in a swarm knows about all service.**



**VOLUMES AND PERSISTENT DATA**

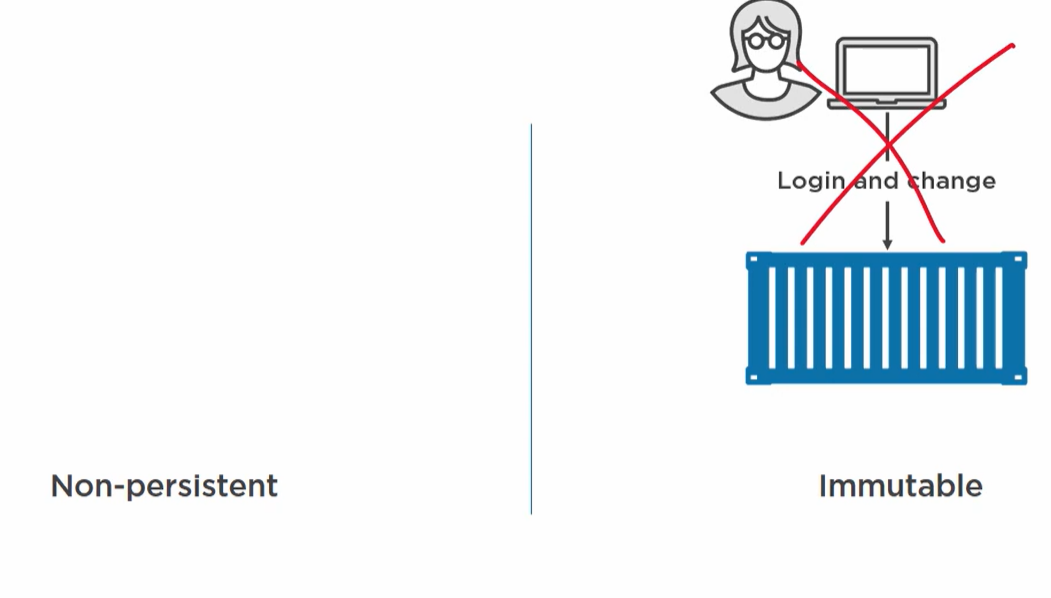


* Containers are **non-persistent**, we start it , run it and then exit it or delete it. We don’t persist any container.
* They are **immutable** we hardly change anything in same container we just create a new one.



To persist any data, we use volumes. **Volumes are de-coupled from container**.

* Containers are great for non-persistent stuff.
* It is immutable so no one can change the existing container once deployed. If we want to deploy new stuff we create a new one and deploy it.

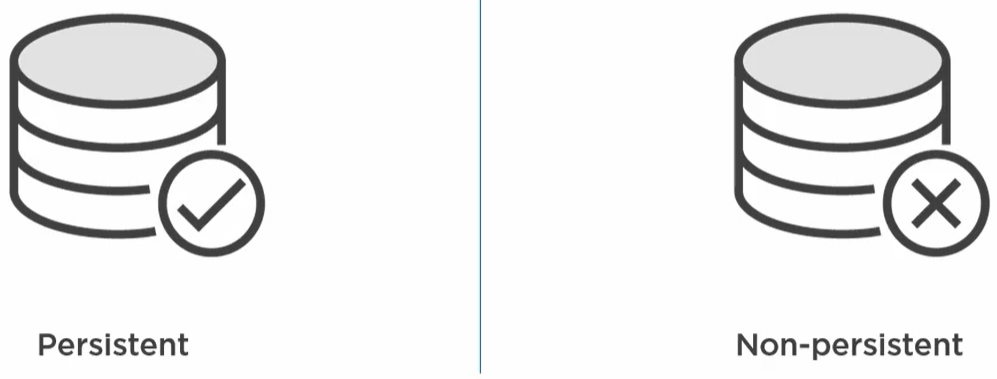


Never change existing container.

Deploy new one



Containers are short lived, immutable and don’t generate data we want to keep.

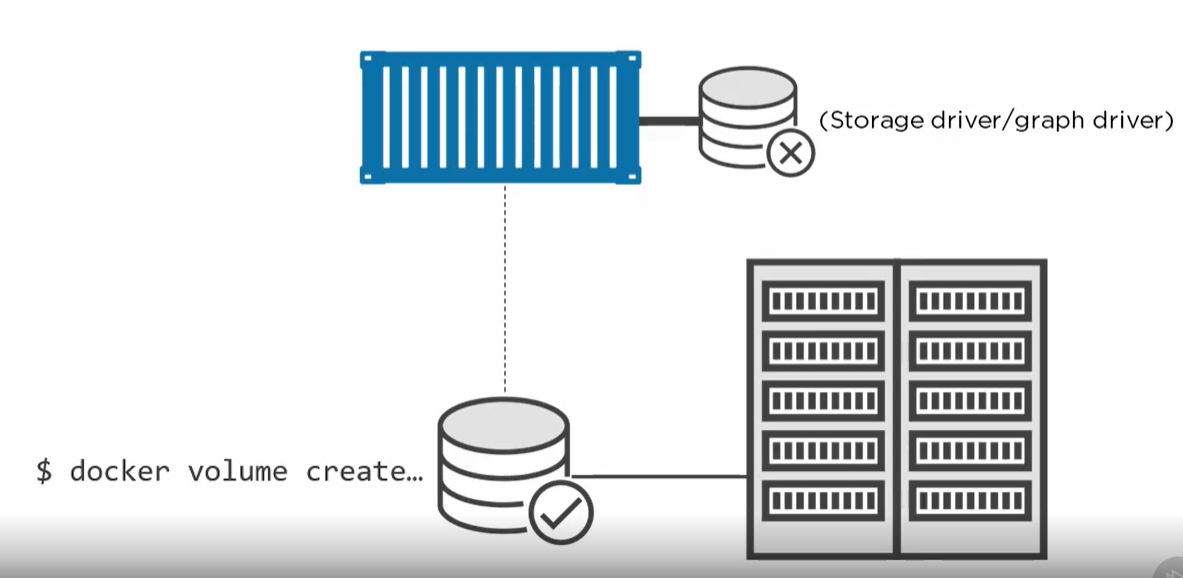


Data can be persistent and non-persistent.

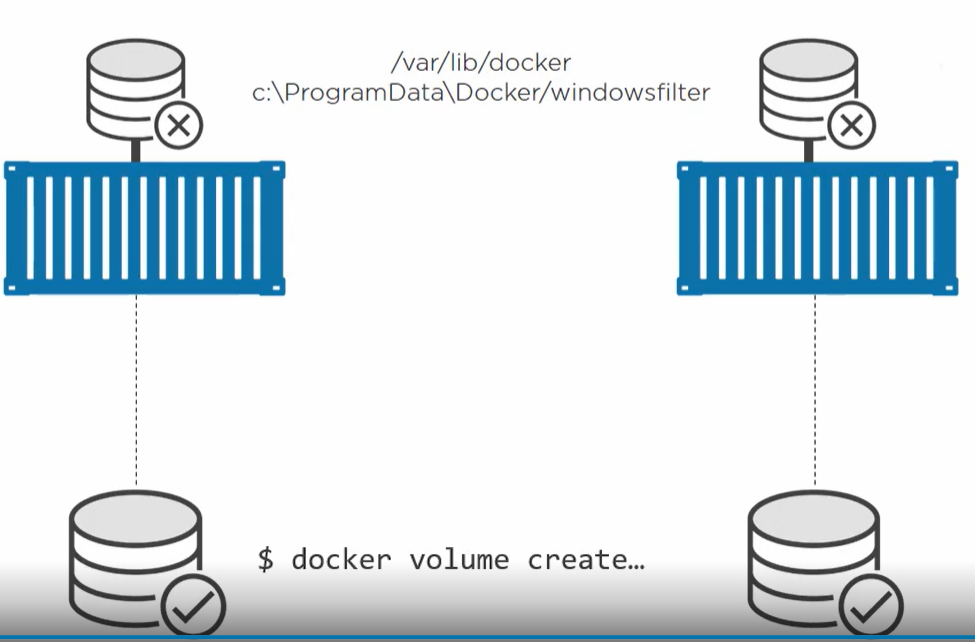
Containers are ideal fit for non-persistent data.

Every container has its non-persistent storage manage by storage drivers like overlay2,AUFS etc.

Every container gets its local graph driver storage. It store all image layer and writable layer in this storage. It is attached to the image so once container is deleted the driver storage is deleted.



* When comes to persistent storage we need to create a **volume** which is outside our union file system.
* Docker volume can connect to any database using docker volume driver.
* Or it is an **object in docker** which is managed by its own, separate from container.
* We can mount volume on container.



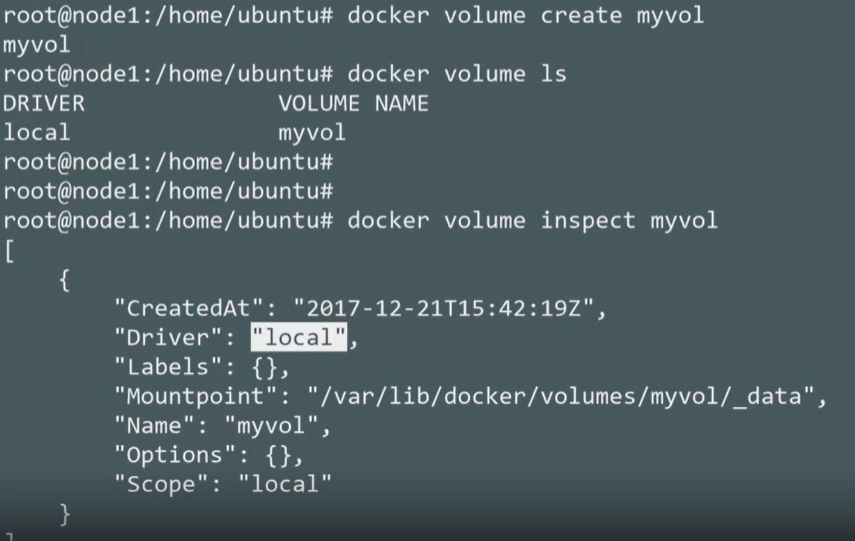
When we create a container, non-persistent storage is created at some location using storage driver.

It is tied to the lifecycle of container.

Volume data is created independently of container.

We can mount volume data on a container.

Even if we delete a container volume data exists.



We created volumes without any container. Means they are independent.

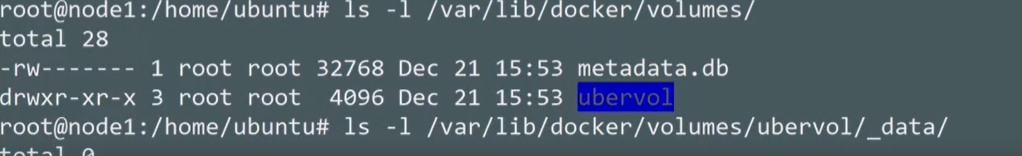
**Attaching a volume to a container**

**We don’t have any volume created when we create the container specifying any volume, docker will create one.**

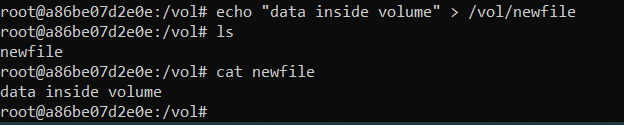
*docker run -dit --name volumeContainer --mount source=volumeNew,target=/vol imageName*

*source = name of volume*

*target = location inside container to mount volume*

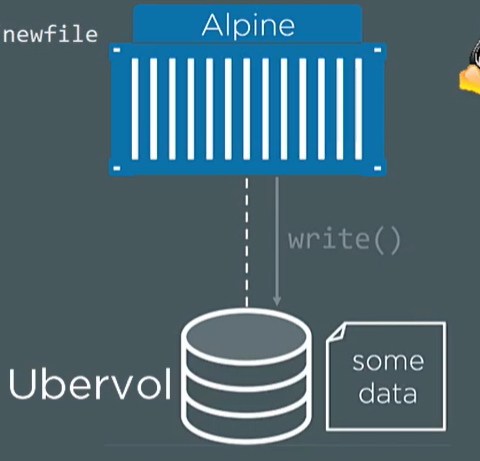


**Volume data directory** *var/lib/docker/volumes/volumeName/\_data*



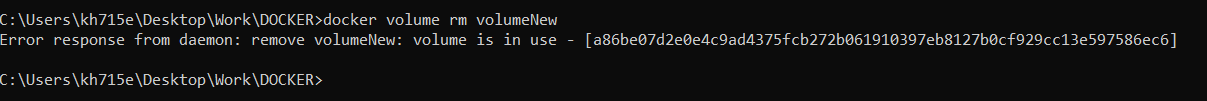
We created a new file inside /vol directory. This will map directly to our \_data in volume storage location.

Even if we delete the container then also \_data will contain the data.

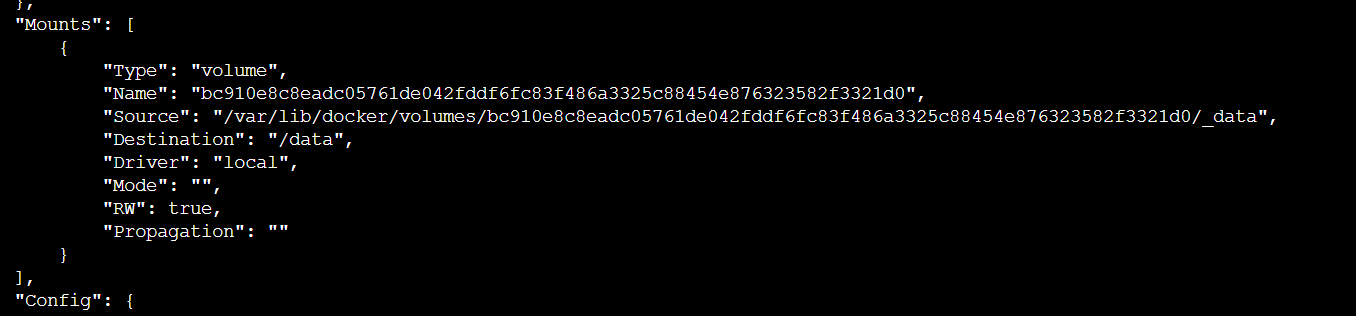


But if any volume data is mounted to any container then it will not allow us to delete the volume.

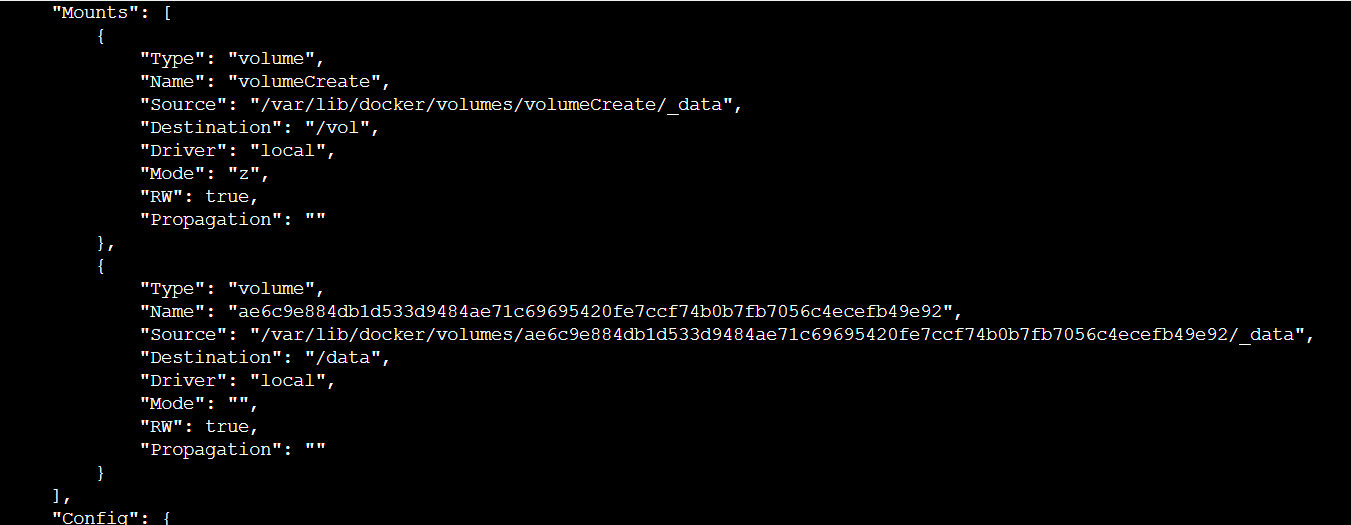
So first delete the container and then delete the volume.



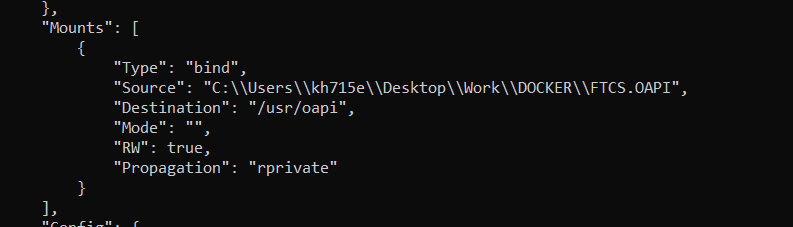
1. When we create a container **without manually** creating any volume into it. It will create a default volume and mount on container



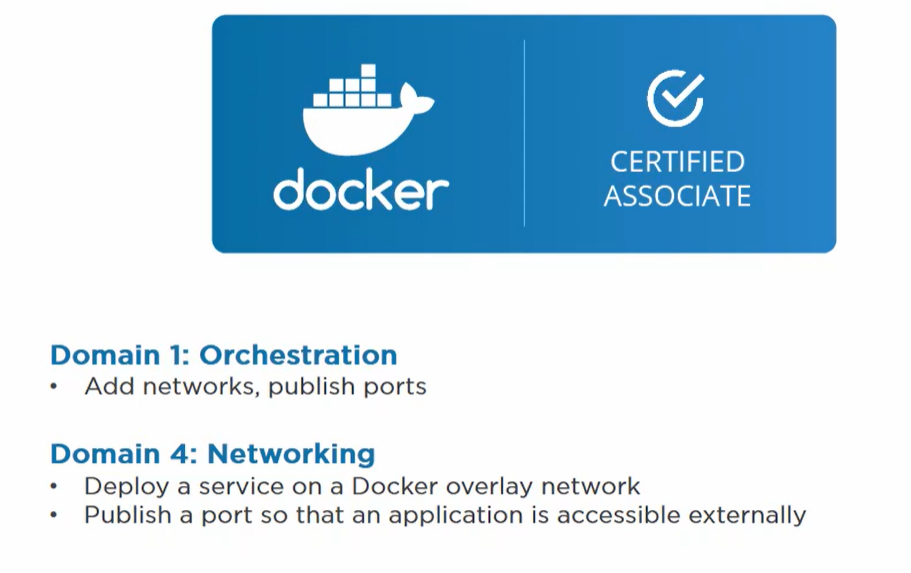
1. When we create a **volume manually** then container can mount to 2 volumes one in /data and /vol



We can have mounts without creating volume also using **bind from our local directory**



**SECRETS**



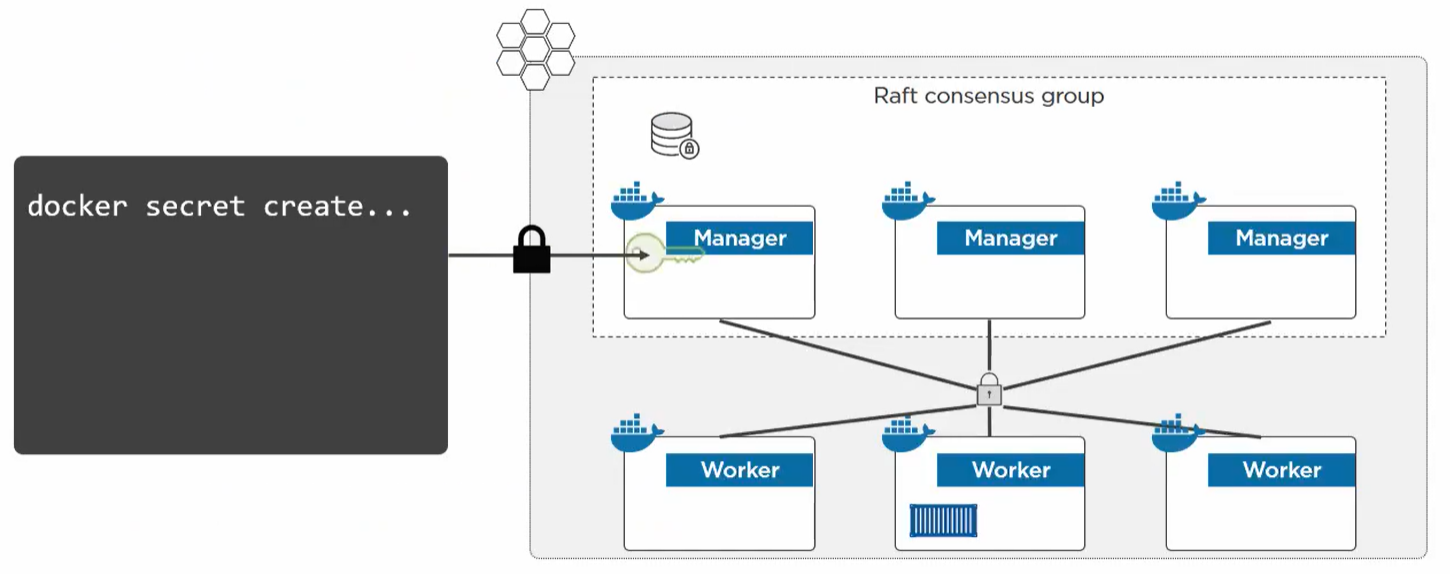
Anything in our apps which is a vulnerable to security threat ex : password, certificates, names of services etc use secrets.

* Docker secret is a text blob upto 500k.
* Use case- Telling a web front end the password of backend persistent store.
* It runs in swarm mode.

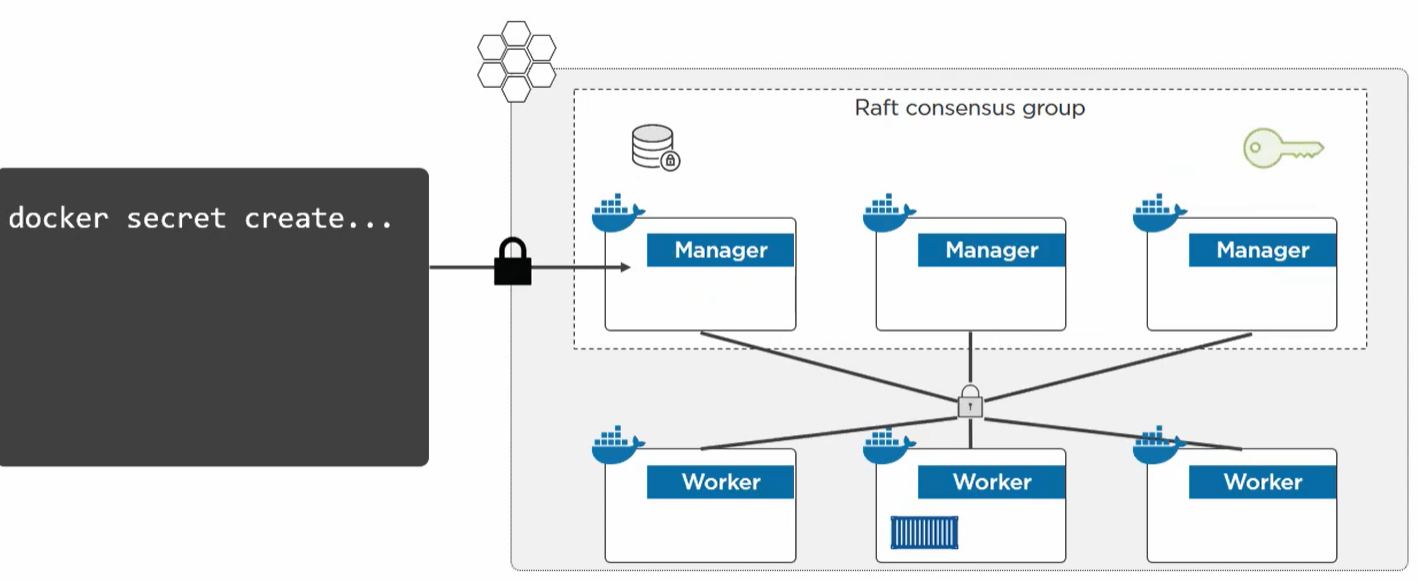


Secret Steps:

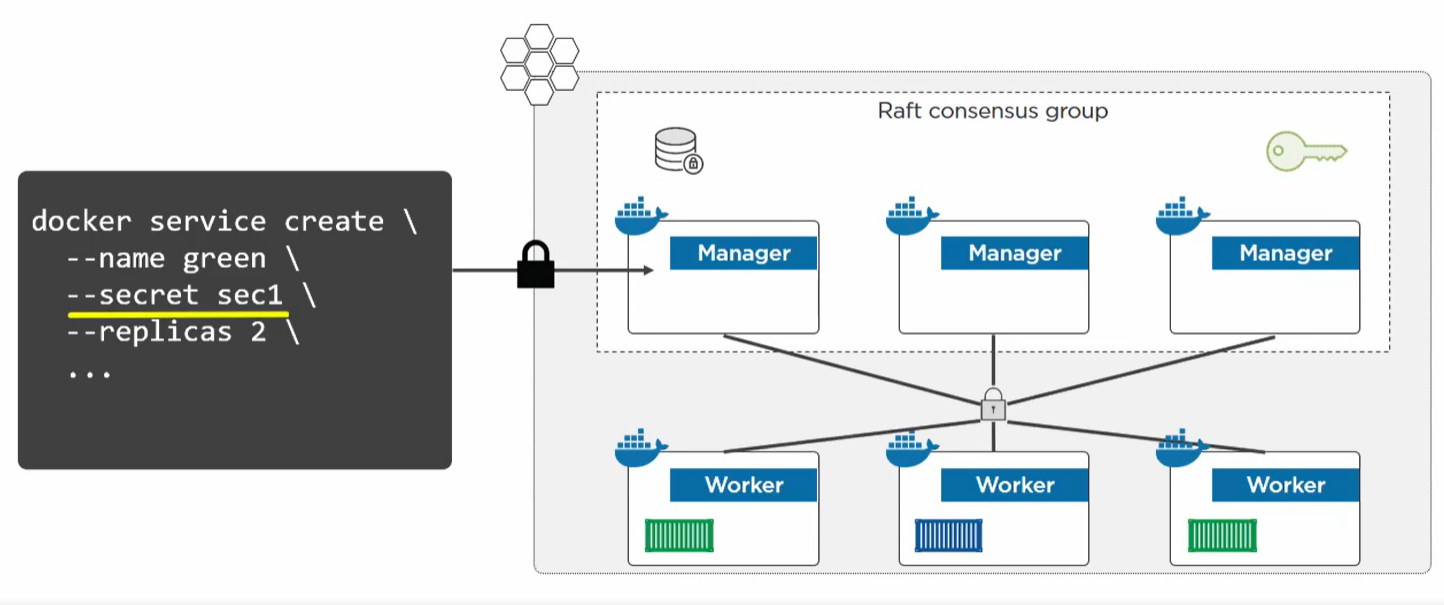
1. We have a swarm with manager and workers. We create a secret and docker client send to a Manager over a secure connection.



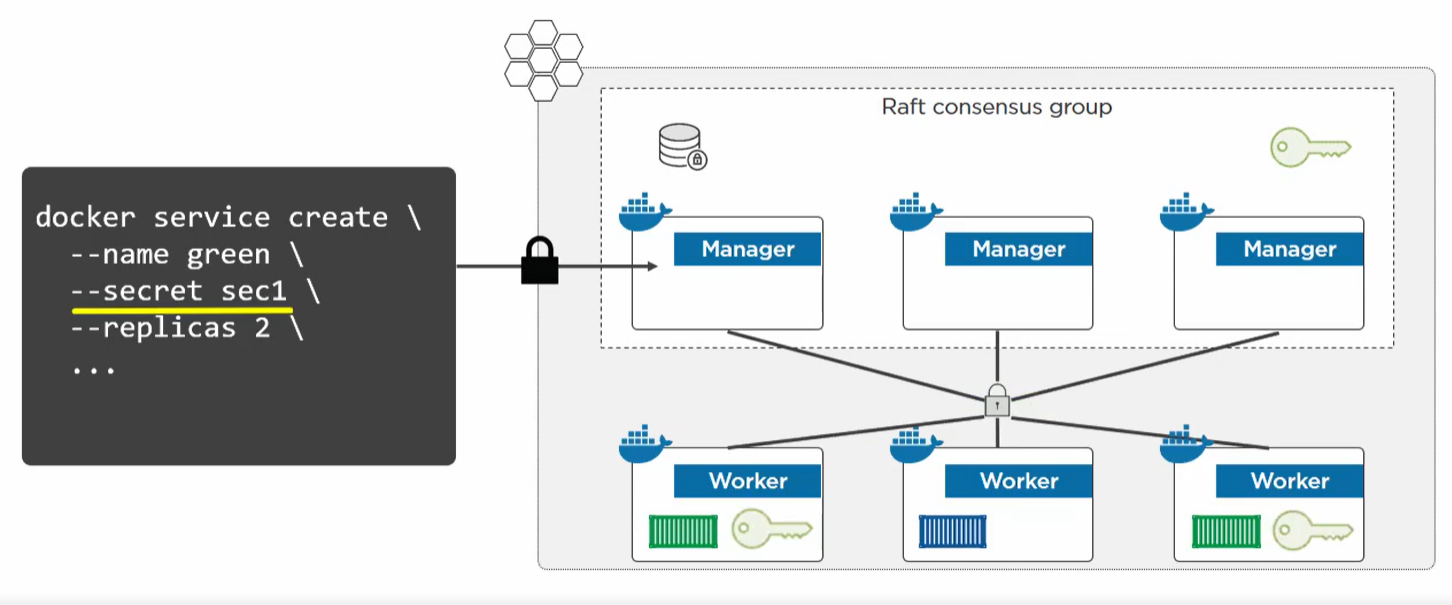
1. Manager keeps the secret in a cluster store where it is encrypted at rest.



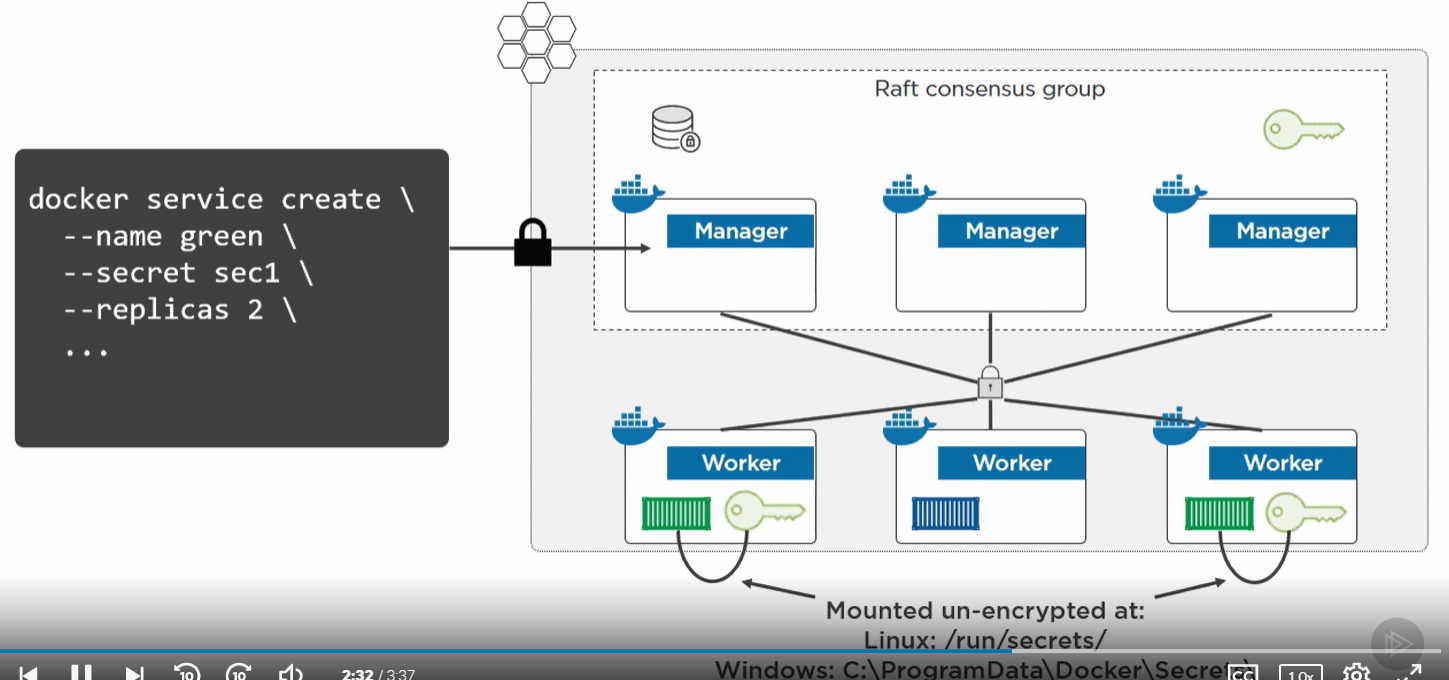
1. Then we create a service or update one with replicas. We explicitly grant the service access to secrets. So, the nodes in which this service replica is running will get the secret.



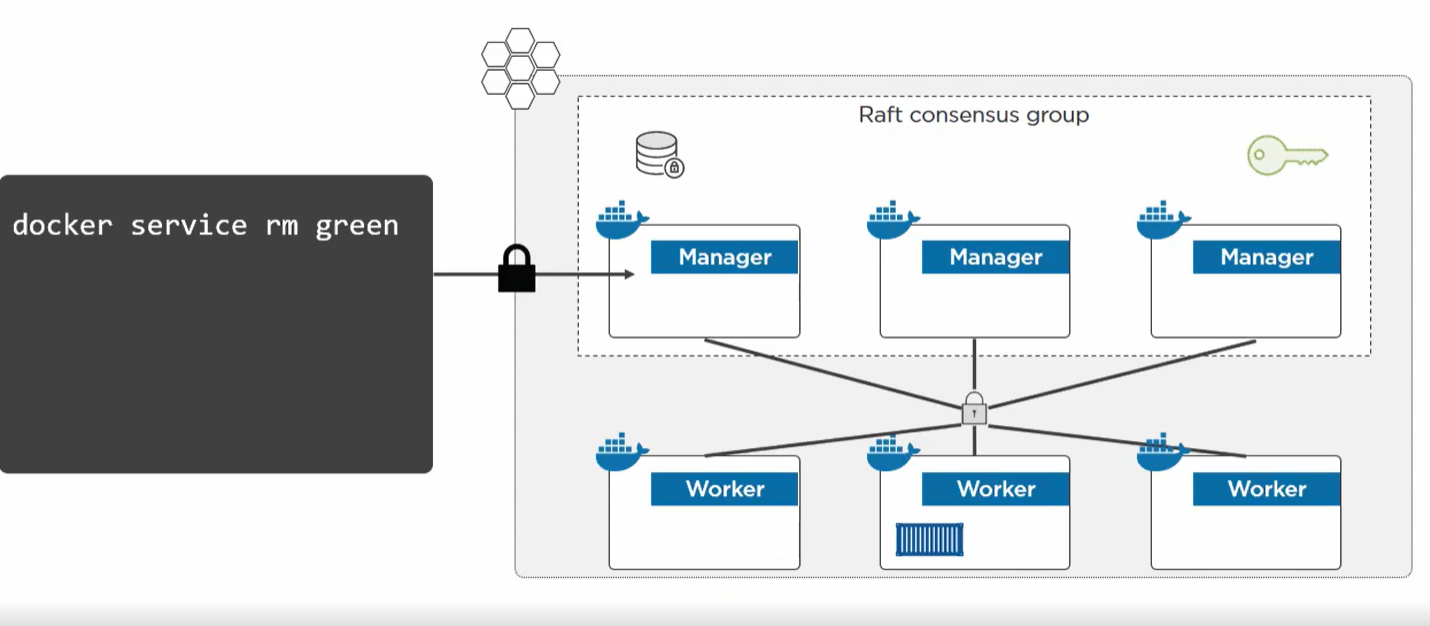
1. Manager then sends the secret over secure connection to the nodes in which replica of service which is authorized is running.



1. Once the node receives the secret then it is mounted inside the service task in its unencrypted form. It is stored in tmp location(in memory) in linux so that it is not persisted.



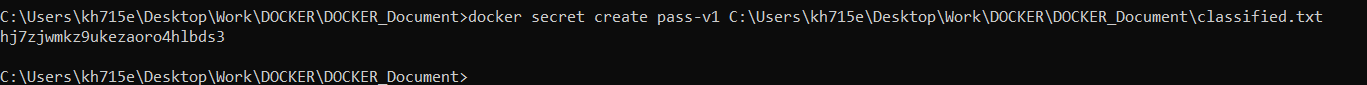
1. Once the service or container is terminated the worker node is instructed to flush the secrets



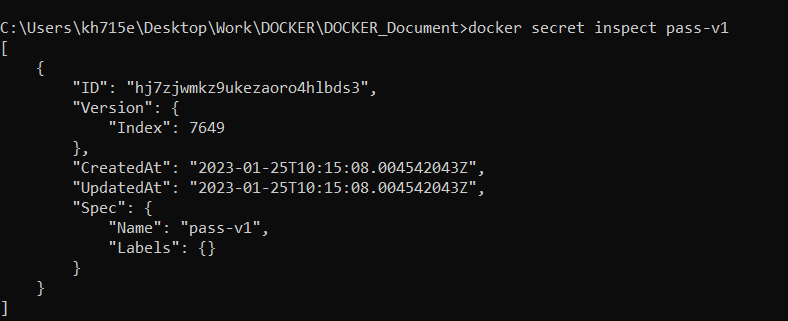
**Docker secrets only work for services in swarm mode. If we have just a container running in swarm mode secrets won’t work it will work with only services.**

**DEMO**

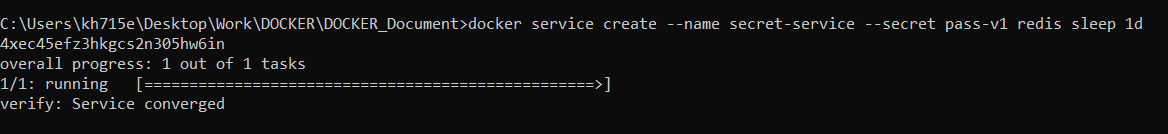
1. We have swarm running in docker desktop.
2. Create a file and then create a secret using the file



1. Once we create a secret then our docker client has send the secret to swarm manager daemon over a secure channel and secret securely stored in swarm raft ( cluster store).



1. Even we inspect we cannot see the content of secret. Only way to see it is create a service and grant access to this secret.
2. Now we create a service and grant access to the secret.



1. When we inspect the service we see secret inside the service



In windows container we can see secrets stored in **C:/programData/Docker/secrets**. In linux its **run/secret**

1. We can’t delete a secret in use. Only when service is terminated the secrets are flushed.

We can use UNIVERSAL CONTROL PANEL TO DEMO SECRETS