Virtualization

two types:

1. type 1 or bare metal virtualization- hypervisor sits on top of hardware(Microsoft VM, VMWare ESXi, Kernel Virtual Machine(KVM))
2. type 2 or the hosted virtualization- hypervisor sits on top of a layer of host OS,which sits on top of hardware(Oracle VM, VMWare workstations)

The downside of hosted hypervisors is that latency is higher than bare-metal hypervisors. This is because communication between the hardware and the hypervisor must pass through the extra layer of the OS.

Hypervisors and containers are used for different purposes. Hypervisors are used to create and run virtual machines (VMs), which each have their own complete operating systems, securely isolated from the others. In contrast to VMs, containers package up just an app and its related services. This makes them more lightweight and portable than VMs, so they are often used for fast and flexible application development and movement.

Level at which virtualization happens:

Virtual machines are Hardware level virtualizations.

Containers are Operating System level virtualizations

Type of Isolations

Containers are process level isolation, they cannot talk with each other,neither share resources, they share same OS, and each container assumes that they have a seperate OS with only dependencies of applications installed on the OS.

Whereas in Virtualization we achieve Machine level Isolation, we make several machines on a one physical machine,each machine is relatively isolated with each other.

How are resources accessed?

In VMs , we create different machines, hypervisor manages resources to different machines

whereas in containers namespaces for isolation of process from other processes and cgroups(control groups) of linux help in controlling resources that are given to containers.

Flexibility v/s Portability

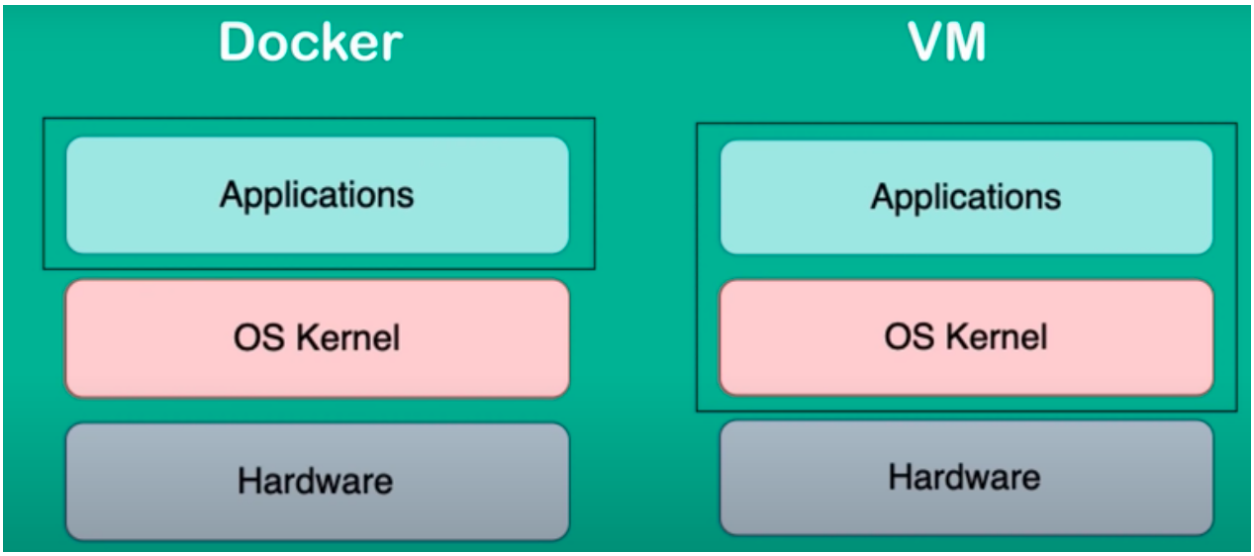
VMs have infinite flexibility as a hardware, we are flexible about RAM, storage and processor(Virtualbox, Parallels)

whereas in Container we have a container file(docker file) which tells how to build a container, dependencies and everything, so we have infinite portability with Containers.

docker tags are pointers to the images, and they are mutable

Docker virtualizes on application layer, while VM virtualizes both Applications and OS kernel, In other words, docker images use host's OS kernel, whereas VMs use their own kernel. hence:

1. Docker images are lighter(less space) than VMs
2. Docker images are faster in execution
3. Docker images created with linux kernel won't work on windows machine, as the images aren't compatible with windows kernel.



Docker Hands-on

The Docker CLI has a command called *run* which will start a container based on a Docker Image. The structure is *docker run <options> <image-name>*.

By default, Docker will run a command in the foreground. To run in the background, the option *-d* needs to be specified.(here d stands for detached mode)

`docker run -d redis`

The launched container is running in the background, the `docker ps` command lists all running containers, the image used to start the container and uptime.

The command `docker inspect <friendly-name|container-id>` provides more details about a running container, such as IP address.

The command `docker logs <friendly-name|container-id>` will display messages the container has written to standard error or standard out.

ports are bound when containers are started using *-p <host-port>:<container-port>* option. Jane also discovers that it's useful to define a name when starting the container,

Jane finds the best way to solve her problem of running *Redis* in the background, with a name of *redisHostPort* on port *6379* is using the following command `docker run -d --name redisHostPort -p 6379:6379 redis:latest`

The problem with running processes on a fixed port is that you can only run one instance. Jane would prefer to run multiple *Redis* instances and configure the application depending on which port Redis is running on.

After experimenting, Jane discovers that just using the option *-p 6379* enables her to expose *Redis* but on a randomly available port. She decides to test her theory using `docker run -d --name redisDynamic -p 6379 redis:latest`

While this works, she now doesn't know which port has been assigned. Thankfully, this is discovered via `docker port redisDynamic 6379`

Jane also finds that listing the containers displays the port mapping information, `docker ps`

After working with containers for a few days, Jane realises that the data stored keeps being removed when she deletes and re-creates a container. Jane needs the data to be persisted and reused when she recreates a container.

Containers are designed to be stateless. Binding directories (also known as volumes) is done using the option *-v <host-dir>:<container-dir>*. When a directory is mounted, the files which exist in that directory on the host can be accessed by the container

and any data changed/written to the directory inside the container will be stored on the host. This allows you to upgrade or change containers without losing your data.

Using the Docker Hub documentation for Redis, Jane has investigated that the official Redis image stores logs and data into a /data directory.

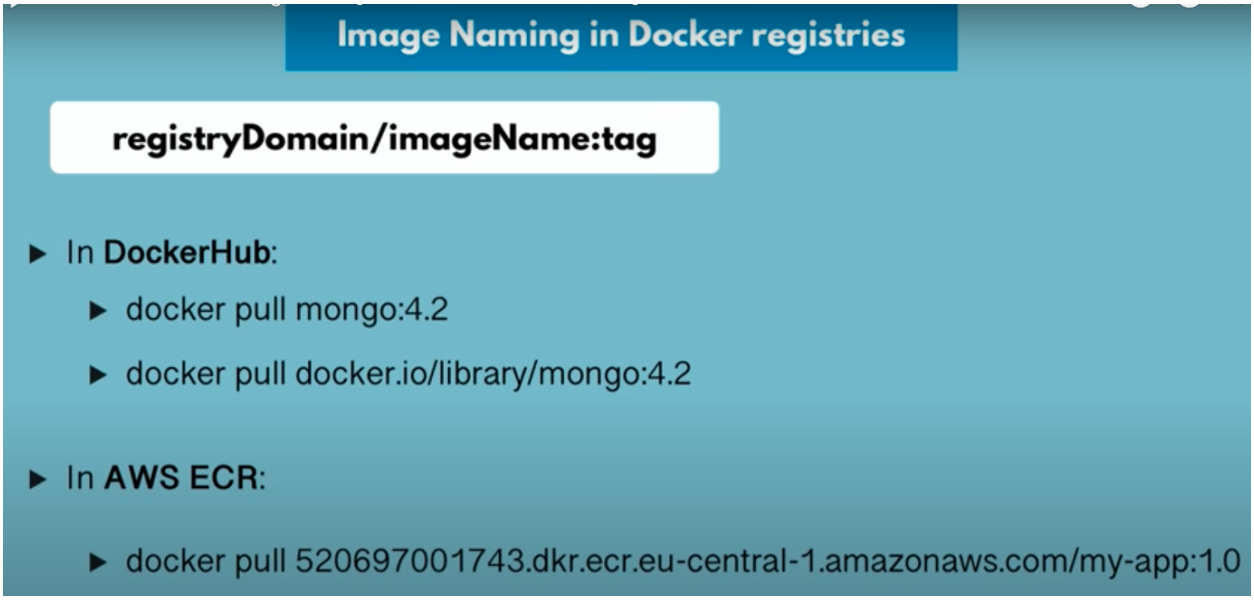
Any data which needs to be saved on the Docker Host, and not inside containers, should be stored in */opt/docker/data/redis*.

The complete command to solve the task is `docker run -d --name redisMapped -v /opt/docker/data/redis:/data redis`

Docker allows you to use $PWD as a placeholder for the current directory.

The command `docker run ubuntu ps` launches an Ubuntu container and executes the command *ps* to view all the processes running in a container.

Using `docker run -it ubuntu bash` allows Jane to get access to a bash shell inside of a container.



deploy a static html website

Docker Images start from a base image

This base image is defined as an instruction in the Dockerfile. Docker Images are built based on the contents of a Dockerfile. The Dockerfile is a list of instructions describing how to deploy your application.

In this example, our base image is the Alpine version of Nginx. This provides the configured web server on the Linux Alpine distribution.

Create your *Dockerfile* for building your image by copying the contents below into the editor.

```
FROM nginx:alpine
COPY . /usr/share/nginx/html
```

The first line defines our base image. The second line copies the content of the current directory into a particular location inside the container.

The Dockerfile is used by the Docker CLI *build* command. The *build* command executes each instruction within the Dockerfile. The result is a built Docker Image that can be launched and run your configured app.

The build command takes in some different parameters. The format is *docker build -t <build-directory>*. The *-t* parameter allows you to specify a friendly name for the image and a tag, commonly used as a version number. This allows you to track built images and

be confident about which version is being started.

Build our static HTML image using the build command below.

```
docker build -t webserver-image:v1 .
```

You can view a list of all the images on the host using `docker images`.

The built image will have the name *webserver-image* with a tag of *v1*.

Launch our newly built image providing the friendly name and tag. As it's a web server, bind port 80 to our host using the *-p* parameter.

```
docker run -d -p 80:80 webserver-image:v1
```

Once started, you'll be able to access the results of port 80 via `curl docker`

Docker images should be designed that they can be transferred from one environment to the other without making any changes or requiring to be rebuilt. By following this pattern you can be confident that if it works in one environment, such as staging, then it will work in another, such as production.

With Docker, environment variables can be defined when you launch the container. For example with Node.js applications, you should define an environment variable for *NODE_ENV* when running in production.

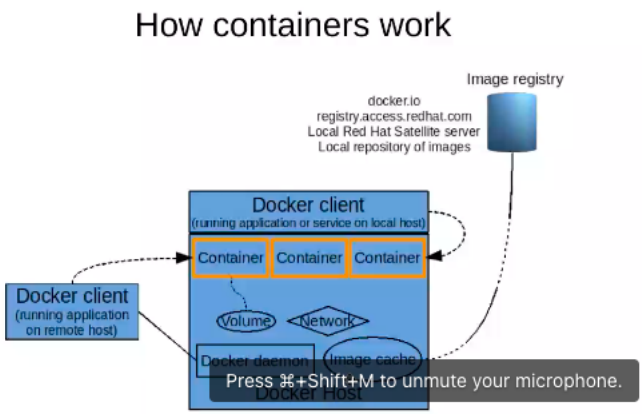
Using *-e* option, you can set the name and value as *-e NODE_ENV=production*

Example

```
docker run -d --name my-production-running-app -e NODE_ENV=production -p 3000:3000 my-nodejs-app
```

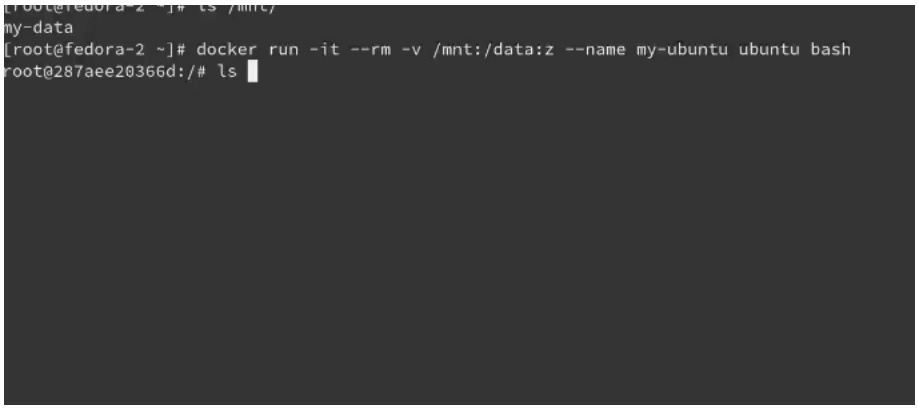
CMD and ENTRYPOINT are almost same, except CMD can be overwritten

devops



`docker run --it --rm ubuntu bash` the —rm will remove the container once job is done.

docker by default runs with ephemeral storage, so if we want persistent



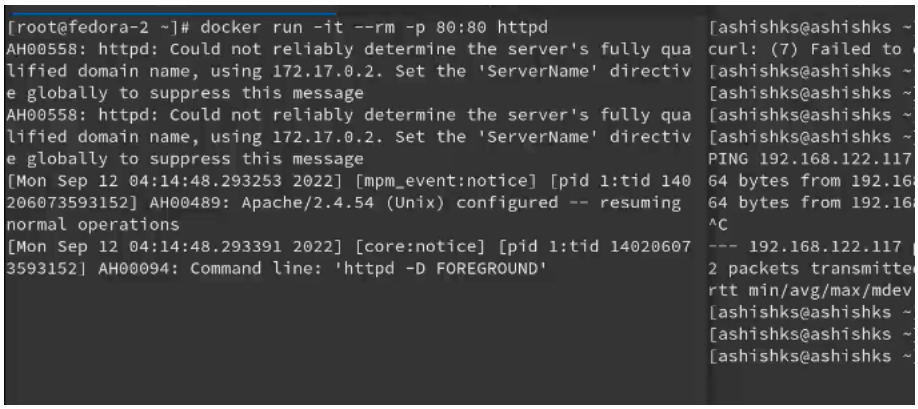
-v is volume mount, /mnt is host volume which will be mount to /data on the container, :z is for the permission thing

running webserver

docker run httpd

it displays a ip address,

try pinging to that ip addrs



port mapping, mapping 80 on host to 80 on container

docker commit to create an image

docker save

All the changes done in docker images are in form of layers

ALL COMMANDS

sudo apt install docker

systemctl enable docker.service

docker run hello-world

docker start gracious_shamir # did not work; this was "name" of docker

docker ps -a # docker ps does not show anything

docker run ubuntu

docker ps -a

docker run -it ubuntu bash# interactive terminal

#inside docker now

uname -a

shows that you are on a FEDORA kernel!

#boot directory is empty on docker

exit # inside docker

#from outside docker, another terminal

docker ps -l

ps aux | grep docker

#shows docker daemon

docker images

docker rm <container-id>/<container_name>

removes this from output of docker ps -a

docker rmi hello-world

#removes image

docker run --it --rm ubuntu bash

#removes the docker entry from "docker ps -a" automatically after you exit docker

docker run --it --rm --name myubuntu1 ubuntu bash

#to get a name that you like

#if you create a file in a docker and exit it, and restart docker, then the file is gone. to preserve the work you have to attach a volume to docker

docker run --it --rm -v /mnt:/data:z --name myubuntu1 ubuntu bash

#inside docker now

touch /data/mydata

you can see the file under /mnt in host now

docker run httpd

#will show you IP address also , e.g 172.17.0.2

curl <ip-addreess-above> # failed in his case

docker run --it --rm -p 80:80 httpd

#this will map port 80 in docker with port 80 in host
-p <host port>:<container port>

NOTE: 2 different containers can listen on same port, however the host port has to be unique for a container.
#now you can do

curl <ip-address-of-host> # and it connects to docker's web server

#commands to change an existin image

docker run --it --rm fedora

#inside docker now

mkdir mydata

cd mydata

touch myfile

vi /bin/mycode

change a+x /bin/mycode

#outside dokcer now

docker commit <docker-name>

docker image

#shows new image with image id but no name

docker tag <image-id> <image-name>

docker save <image-name> -o something.tar

#you can now expect the .tar file

#the new image has 2 layers, the original image has 1 layer

#see layer.tar inside the un-tarred folder

#if you have a paid account on docker-hub then you can push the image on docker-hub. Not sure about this.

docker rmi my-new-fedora-image # removes image

#but you have it in .tar file now

docker load -i my-new-fedora-image.tar

you can share data from host to one container and map another container to use the same volume as well using the —volumes-from option

DOCKERFILE

```
from fedora
run echo “Docker demo”
CMD /bin/bash
```

`docker build -t myapp2 .` to build image from dockerfile

difference between run and CMD is that run is used to install packages/ update/ mounting volumes and so on. Final thing which you want to run from a container is specified in CMD

check logs of container as:

`docker logs <container-id>`

docker run -ti fedora fedora bash

ps aux I grep docker shows 2 entries, 1 of dockercli and 1 dockerd

docker-compose is used to run multiple containers specified in a yaml file together.

`docker-compose -f mongo.yaml up`

docker-compose also creates a network between all the containers specified in the yaml file.

`docker attach <container-id>` to go inside a dettached container

`docker cp <file-path> <container-id>:/path` to copy file from computer to container

`docker diff <container-id>` to see diff

Kubernetes

Components of Kubernetes cluster

Master node - manages the cluster setup

- Etcd

Etcd is a key-value store database. Configuration data and information about the state of the cluster lives in etcd,

Fault-tolerant and distributed, etcd is designed to be the ultimate source of truth about your cluster.

- kube-scheduler

The scheduler considers the resource needs of a pod, such as CPU or memory, along with the health of the cluster. Then it schedules the pod to an appropriate compute node.

- Controller manager

Controllers take care of actually running the cluster, and the Kubernetes controller-manager contains several controller functions. For Example:

- Node controller - Manages node related tasks like onboarding nodes, managing node failures, etc
- Replication controller - Manages container. Make sure the desired number of containers run all the time.
- Kube-apiserver

Facilitates the interaction with the Kubernetes cluster.

The is the front end of the Kubernetes control plane, handling internal and external requests. The API server determines if a request is valid and, if it is, processes it. You can access the API through REST calls, through the kubectl command-line interface, or through other command-line tools such as kubeadm.

Worker node - Rus the container workload in the cluster setup

- Container runtime Engine

To run the containers, each compute node has a container runtime engine. Docker is one example, but Kubernetes supports other Open Container Initiative-compliant runtimes as well, For example:

- Docker
- Containerd
- Rocket
- CRI-O
- Kubelet

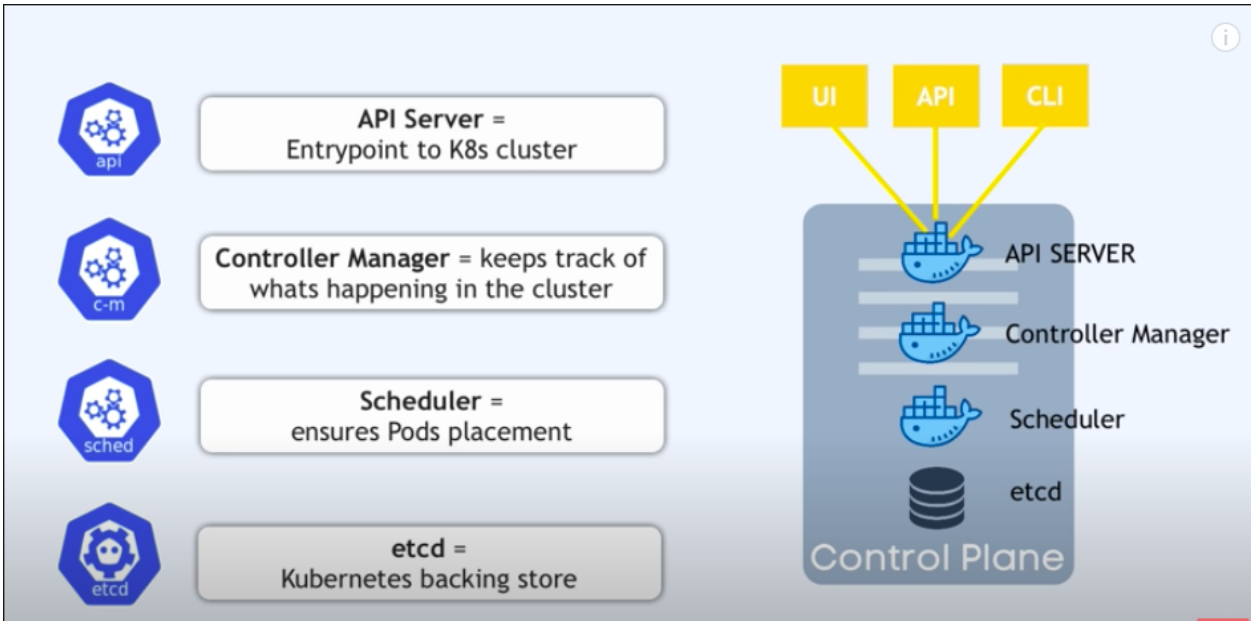
Each compute node contains a kubelet, a tiny application that communicates with the control plane. The kublet makes sure containers are running in a pod. When the control plane needs something to happen in a node, the kubelet executes the action.

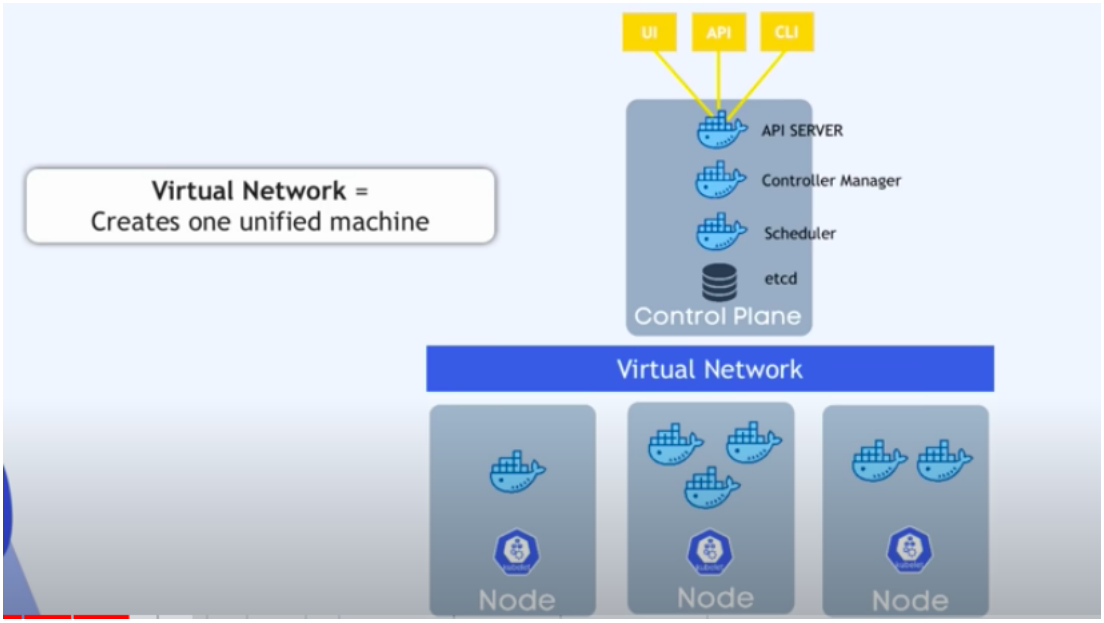
- Kube-proxy

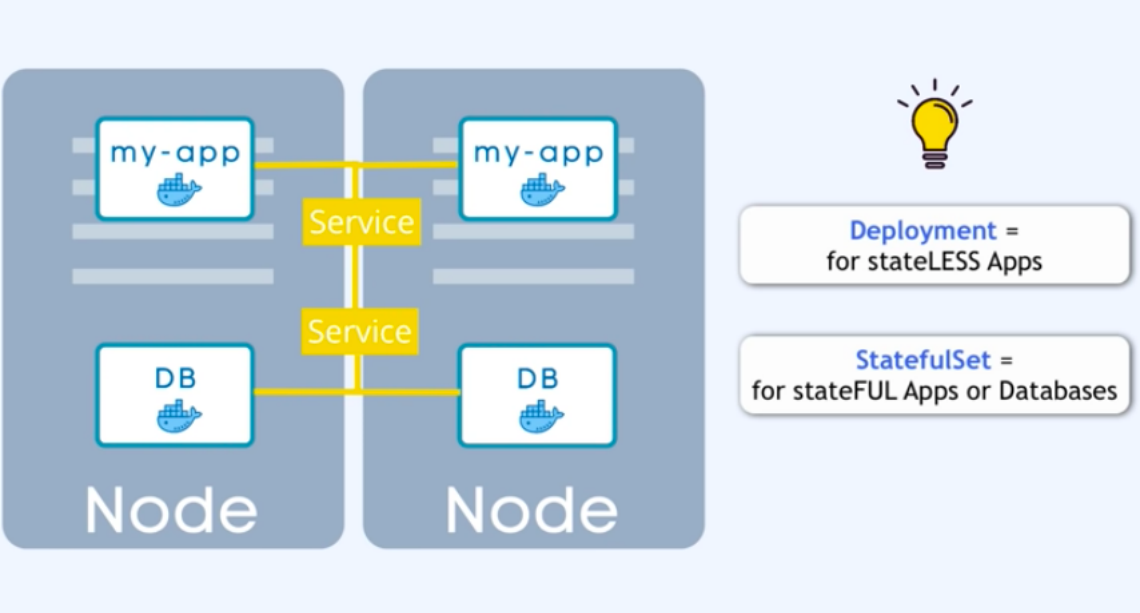
Each compute node also contains kube-proxy, a network proxy for facilitating Kubernetes networking services. The kube-proxy handles network communications inside or outside of your cluster—relying either on your operating system's packet filtering layer, or forwarding the traffic itself.

During the request for creating pod:

1. **kubectl** generates authentication request to **kube-apiserver**
2. **Kube-apiserver** validates the request
3. **kube-apiserver** creates a pod object without assigning it to a node
4. **kube-apiserver** updates the pod info in **etcd** database
5. **Kube-apiserver** updates the user that pod has been created
6. **kube-scheduler** continuously monitors the **kube-apiserver** and realizes there is new pod with no node assigned
7. **kube-scheduler** identifies the right node for the pod and communicated that back to **kube-apiserver**
8. **kube-apiserver** updates the info in **etcd**
9. **kube-apiserver** passes that info to the **kubelet** in the appropriate worker node
10. **kubelet** creates the pod in the node and instructs the container runtime engine to deploy the application image
11. once done **kubelet** updates the status back to the **kube-apiserver**
12. **kube-apiserver** updates the data back in the **etcd**


api
API Server = Entrypoint to K8s cluster
c-m
Controller Manager = keeps track of whats happening in the cluster
sched
Scheduler = ensures Pods placement
etcd
etcd = Kubernetes backing store
UI
API
CLI
API SERVER
Controller Manager
Scheduler
etcd
Control Plane


Virtual Network = Creates one unified machine
UI
API
CLI
API SERVER
Controller Manager
Scheduler
etcd
Control Plane
Virtual Network
Node
Node
Node



why do we need multiple containers of the same application?

vertical scaling is adding resources to the same system, increasing ram,processing speed

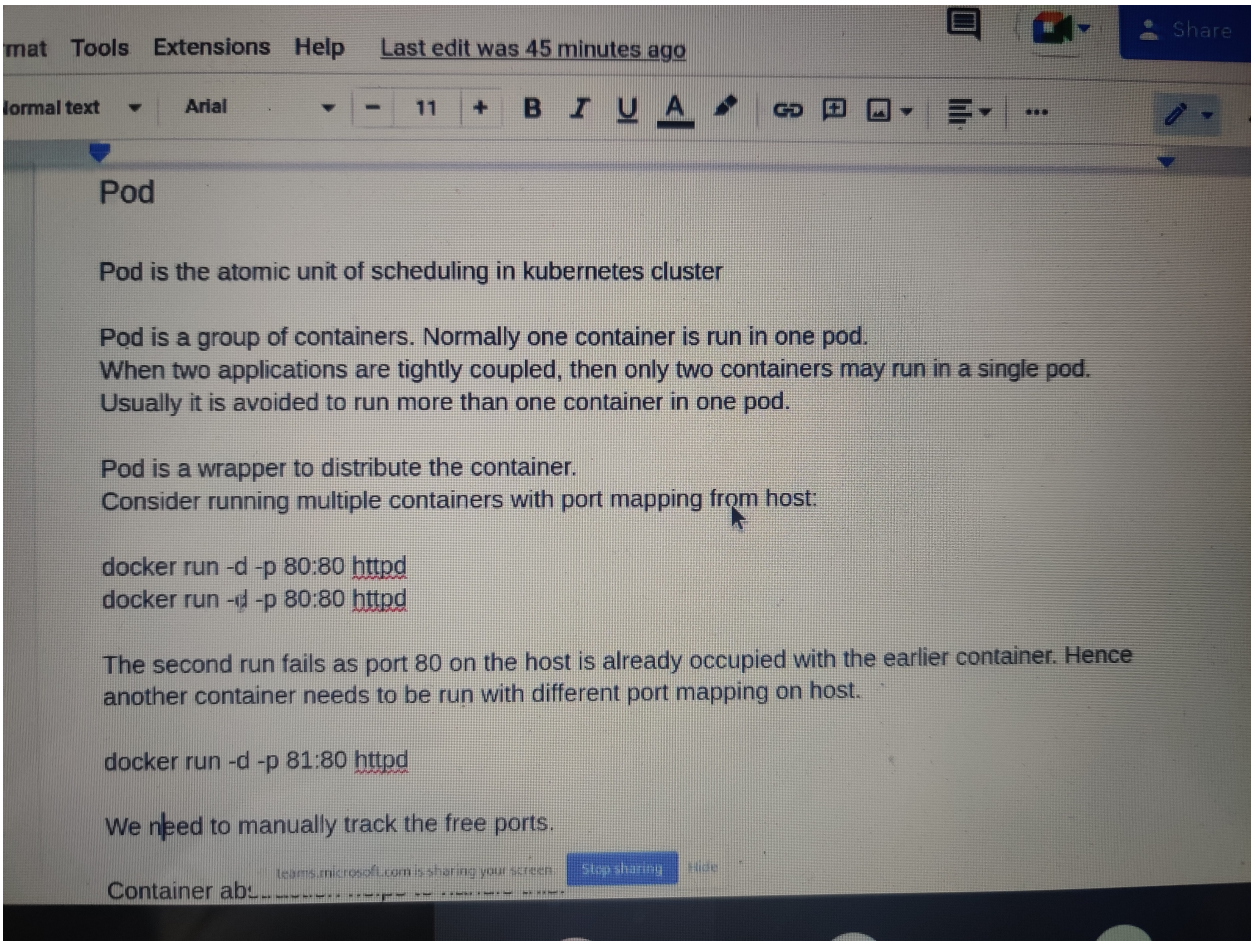
horizontal scaling is adding another instance to serve the workload, along with loadbalancer that distributes the request into multiple instance.

another reason is fault tolerance, if you run only 1 container, and it is down then whole system is down.

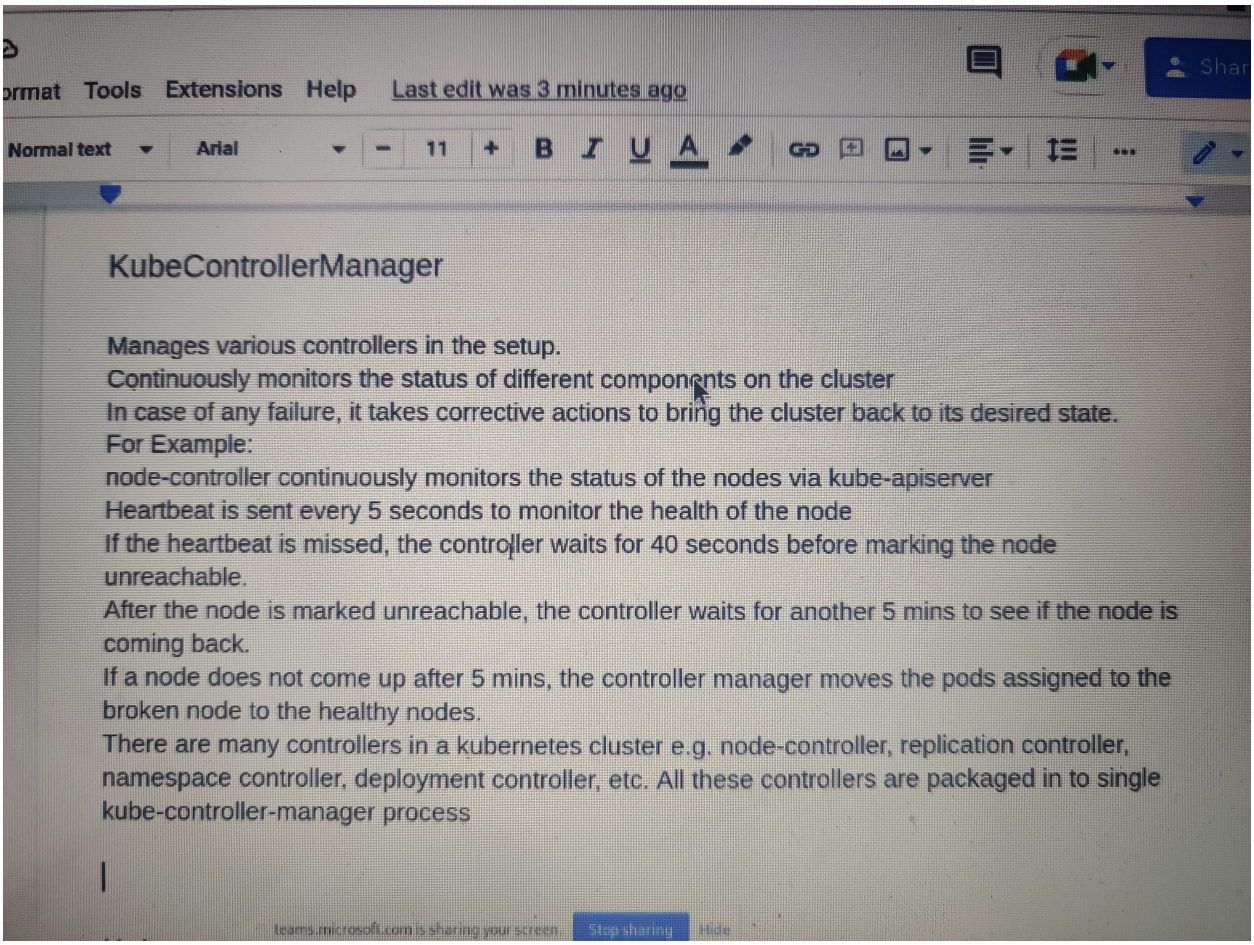
Pod is atomic unit of k8s

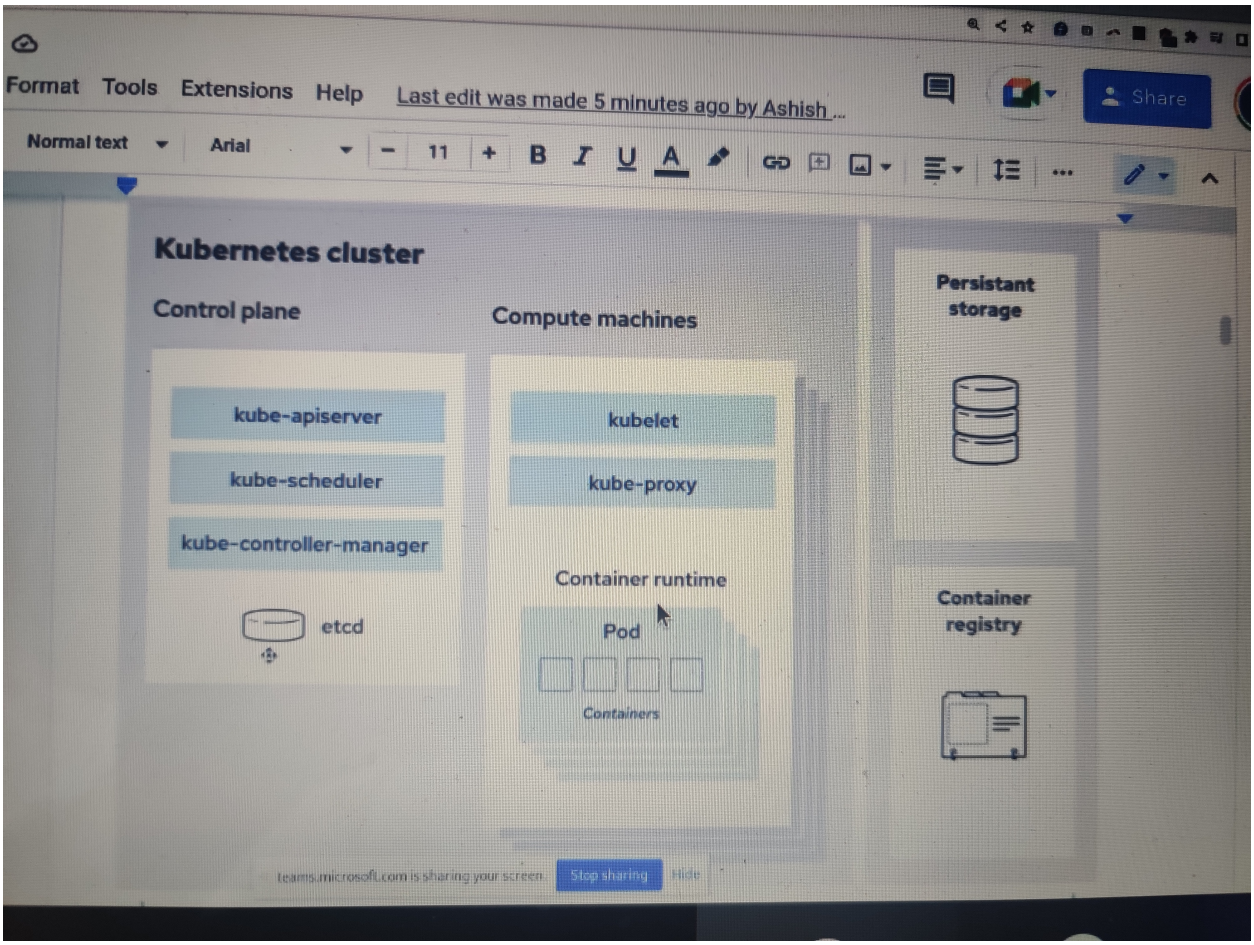
pod is group of containers, usually 1 container is run in one pod.

Pod is wrapper to distribute the container



etcd is central database


rmat Tools Extensions Help Last edit was 3 minutes ago
Normal text
Arial
11
KubeControllerManager
Manages various controllers in the setup.
Continuously monitors the status of different components on the cluster
In case of any failure, it takes corrective actions to bring the cluster back to its desired state.
For Example:
node-controller continuously monitors the status of the nodes via kube-apiserver
Heartbeat is sent every 5 seconds to monitor the health of the node
If the heartbeat is missed, the controller waits for 40 seconds before marking the node unreachable.
After the node is marked unreachable, the controller waits for another 5 mins to see if the node is coming back.
If a node does not come up after 5 mins, the controller manager moves the pods assigned to the broken node to the healthy nodes.
There are many controllers in a kubernetes cluster e.g. node-controller, replication controller, namespace controller, deployment controller, etc. All these controllers are packaged in to single kube-controller-manager process
teams.microsoft.com is sharing your screen.
Stop sharing
Hide



when you need k8s?

for ecommerce website, you need webserver, database, caching, all related stuff.

if all these services is containerized, management is easier.

if we are running at as standalone, if one service fails it would make the whole system fail.

if we are using k8s, it respawns the failing containers so that atleast n number of containers are running.

```
kartik@kartik-MacBookAir:~/Desktop$ kubectl get pods
No resources found in default namespace.
kartik@kartik-MacBookAir:~/Desktop$ kubectl get nodes
NAME       STATUS   ROLES           AGE   VERSION
minikube   Ready    control-plane   29m   v1.25.0
kartik@kartik-MacBookAir:~/Desktop$ kubectl get pods -A
NAMESPACE     NAME                               READY   STATUS    RESTARTS      AGE
kube-system   coredns-565d847f94-6ctcm           1/1     Running   0             29m
kube-system   etcd-minikube                      1/1     Running   0             30m
kube-system   kube-apiserver-minikube            1/1     Running   0             30m
kube-system   kube-controller-manager-minikube   1/1     Running   0             30m
kube-system   kube-proxy-bxwvg                   1/1     Running   0             29m
kube-system   kube-scheduler-minikube            1/1     Running   0             30m
kube-system   storage-provisioner                1/1     Running   1 (29m ago)   29m
kartik@kartik-MacBookAir:~/Desktop$ kubectl create deployment myapp --image=nginx
deployment.apps/myapp created
kartik@kartik-MacBookAir:~/Desktop$ kubectl get pods
NAME                     READY   STATUS              RESTARTS   AGE
myapp-7679744c4d-zqnnk   0/1     ContainerCreating   0          11s
kartik@kartik-MacBookAir:~/Desktop$ vi pod.yml
kartik@kartik-MacBookAir:~/Desktop$ vi pod.yml
```

```
kartik@kartik-MacBookAir:~/Desktop$ gedit pod.yml
kartik@kartik-MacBookAir:~/Desktop$ #kubectl create -f pod.yml
```

Node is a single server or single machine.

All 5 bare metal server will form 1 cluster. The nodes could be control node or compute node.

Master node is same as the control plane.

Worker node is compute plane.

Kubectl controls the entire cluster.

Minikube creates a single node cluster. It runs all in one setup in 1 machine. Minikube runs etcd, Proxy and all such containerized services.

`kubectl edit pod <pod-name>` to edit pod's setting dynamically.

if we have a replicaSet of 3 replicas,which uses selector: "type:frontend", now can I create a pod separately having the same label, which may increase the count of pods to 4, or it will still be 3 pods(since same label,so replicaSet will maintain it)? It will maintain it to 3 pods always.

`kubectl apply -f replica.yml`

kubectl edit is on running container, it is used to do changes on running container.

Replicaset:

It always mantains the number of replica of the pod always.even if you manually delete pod, ot will create new.

In order to delete

Kubectl delete replicasets.apps myapp-ha

`kubectl scale replicaset my-replicaset --replicas=5` to change the number of replicas of a running replicaset.

limiting amount of memory a pod uses:

requests:

memory: “8Mi”

limits:

memory: “16Mi”

it means 8MB to allocate initially, while the max memory to allocate is 16MB

to execute a pod use kubectl

kubectl exec -it testpod bash

kubernetes deployments and rollouts

lets create a deployment:

Deployment enables declarative updates for Pods and ReplicaSets. [Same config as replicaset except kind: Deployment]

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-new
  labels:
    app: myapp
    type: front-end
spec:
  template:
    metadata:
      name: myapp
      labels:
        app: myapp
        type: frontend
    spec:
      containers:
        - name: container-1
          image: nginx:1.21
  replicas: 4
  selector:
    matchLabels:
      type: frontend
```

`kubectl apply -f deployment1.yml`

this command creates a deployment.

`kubectl get deployments` to see deployment

`kubectl get pods` shows that 4 pods are created in this deployment.

`kubectl rollout history deployment myapp-new` shows that only 1 revision of history is present so far.

`kubectl get replicaset.apps` shows 1 replicaset for myapp-new deployment.

now `vi deployment1.yml` change nginx version to **1.20** from **1.18**.

`kubectl apply -f deployment1.yml`

this will create a new replicaset as well as we can see a new entry in the history

`kubectl rollout history deployment myapp-new` shows that 2 revision of history is present so far.

`kubectl get replicaset.apps` shows 2 replicaset for myapp-new deployment.

`kubectl get pods` → `kubectl describe pods <pod-name>` shows that nginx version is changed from **1.18** to **1.20**

```
NOTE: `kubectl rollout status deployment myapp-new` run this command after apply to see the status of deployment change
```

now `vi deployment1.yml` change nginx version to **1.21** from **1.20**.

`kubectl apply -f deployment1.yml`

this will create a new replicaset as well as we can see a new entry in the history

`kubectl rollout history deployment myapp-new` shows that 3 revision of history is present so far.

`kubectl get replicaset.apps` shows 3 replicaset for myapp-new deployment.

`kubectl get pods` → `kubectl describe pods <pod-name>` shows that nginx version is changed from **1.20** to **1.21**

Now,

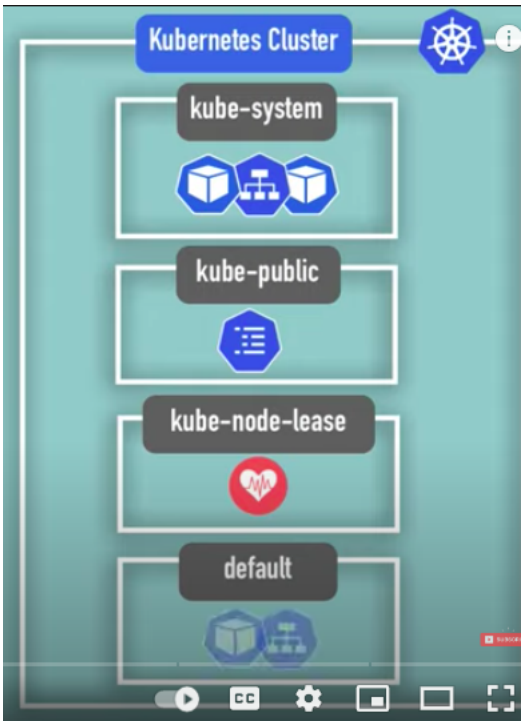
`kubectl rollout undo deployment myapp-new` will move pods from new replicaset to the previous one, and now we can see that the pods are using nginx:1.20 instead of 1.21.

we have rolled back our update.

```
NOTE: if you again run the undo deployment command, this time it will rollback to version 1.21 again, instead of going to 1.18.
So in order to go to version 1.18 or some specific versionn in history use command:
`kubectl rollout undo deployment myapp-new --to-revision <revision-number>`.
`kubectl rollout history deployment myapp-new` to get revision-number.
Example:
`kubectl rollout undo deployment myapp-new --to-revision 1`
```

namespaces in K8s

there are 4 namespaces in kubernetes present already:



Namespace provides a scope for Names. It provides isolation, so for example you may want ot keep dev and prod setup separate so we can use namespaces in that case.

`kubectl get all -A` to see namespace of pods, replicasets, deployments

`kubectl create namespace myspace` to create new namespace in k8s

`kubectl get namespaces` or `kubectl get ns` to see all namespaces.

`kubectl apply -f pod.yml --namespace myspace` to create pod in that namespace

`kubectl get pods --namespace myspace` to get all pods in that namespace

`kubectl get pods --all-namespaces`

`kubectl config current-context` to see current context

`kubectl config set-context minikube --namespace=myspace` to change active namespace to myspace from default.

resourceQuota in k8s

you can set limit to resources in a namespace

```
Run the pod with custom image

cat pod-limits.yml
---
apiVersion: v1
kind: Pod
metadata:
  name: testpod
spec:
  containers:
  - name: app
    image: ashishkshah/myrepo:test
    resources:
      requests:
        memory: "8Mi"
      limits:
        memory: "16Mi"

Connect to the pod via bash shell (two sessions) and execute memory consumption utility /usr/local/bin/mem
Keep increasing the program's memory consumption and observe it in another session.
$ watch “ps aux | grep mem | grep -v mem”

Watch the program getting terminated upon hitting limit

A request is the amount of that resource that the system will guarantee for the container, and Kubernetes will use this value to decide on
A limit is the maximum amount of resources that Kubernetes will allow the container to use.

Resource limit on namespace
cat resource-quota.yml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota
spec:
  hard:
    cpu: "2"
    memory: "32Mi"
    pods: "4"
    replicationcontrollers: "2"
    resourcequotas: "1"
    services: "3"

kubectl describe quota

kubectl --namespace=myspace describe quota

# kubectl apply -f resource-quota.yml --namespace=myspace

kubectl --namespace=myspace describe quota

kubectl config set-context minikube --namespace=myspace
kubectl config get-contexts
kubectl describe quota

cat pod-limits-nginx.yml
---
apiVersion: v1
kind: Pod
metadata:
  name: testpod
spec:
  containers:
  - name: app
    image: nginx
    resources:
      requests:
        memory: "8Mi"
        cpu: 2
      limits:
        memory: "16Mi"
        cpu: 2
```

Services in K8s

ClusterIP is used for inter pod communication between

ClusterIP is the default service type.

This is used for inter service communication within the cluster.

Consider two deployment sets, front-end and back-end.

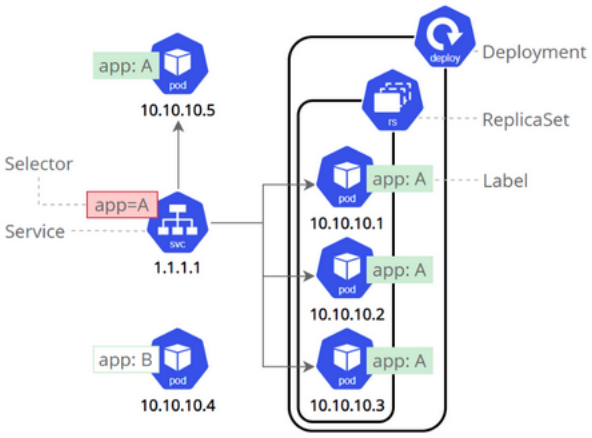
Communication between these two deployment sets is established using clusterip service.

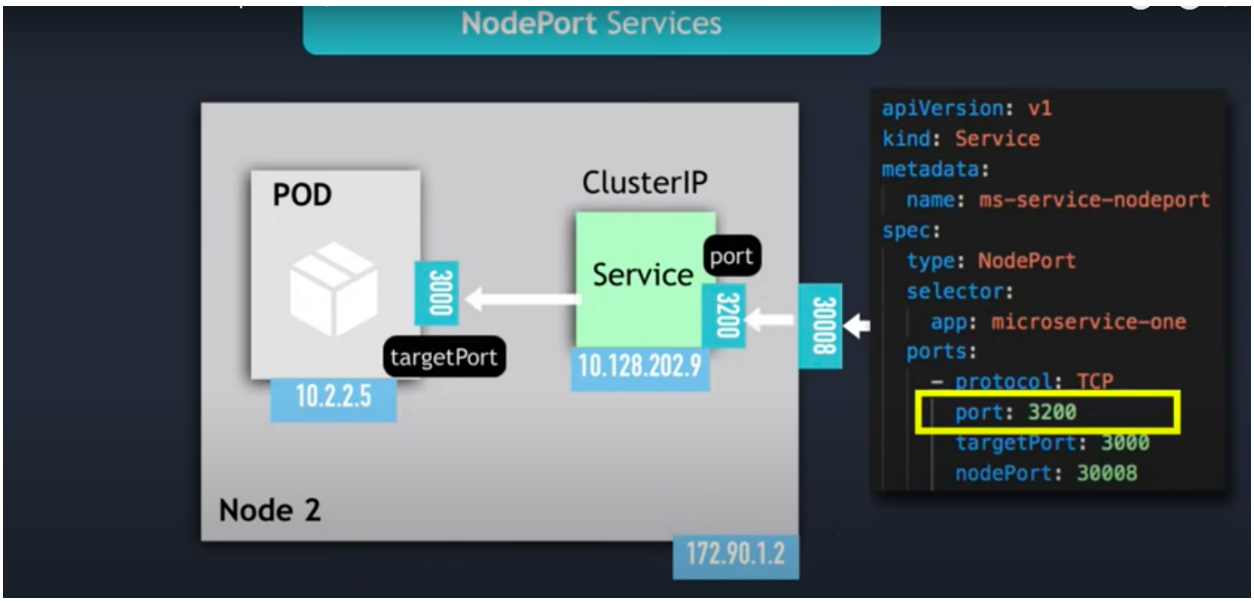
Why do we need it? or why do we prefer this for inter pod communication within the cluster?

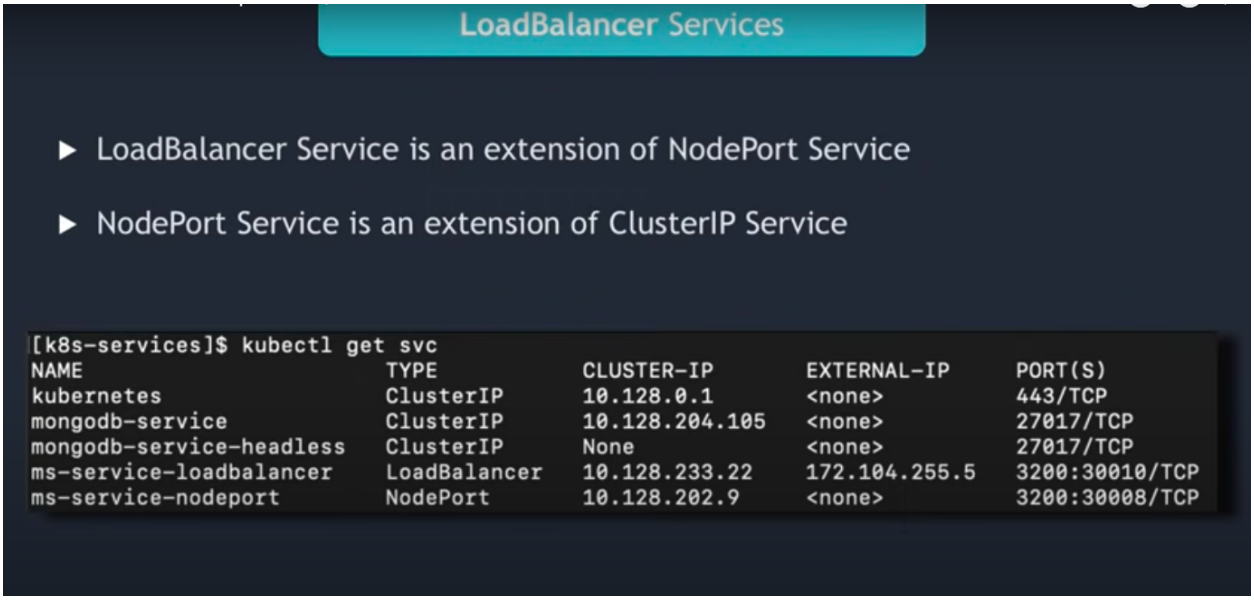
If we are using a pod's ip address for communication among the pods, when any pod is deleted or crashed due to some reason, the deployment set or replica set will spawn another pod to maintain the replica count.

The new pod spawned will have different ip addresses and hence it will not be easy to maintain the communication among the pods with new pods running with different ips.

each node has a set of ip addresses, while pods in a service are identified using the selector label. For ex: In this diagram, app=A is the label selector for service having ip 1.1.1.1, hence all the pods having label app:A are part of this service.

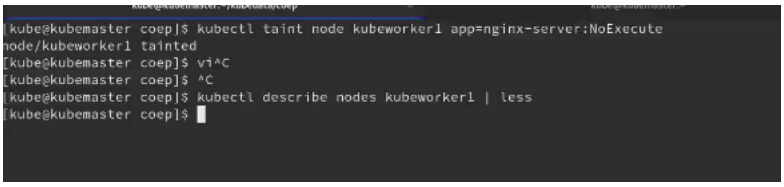




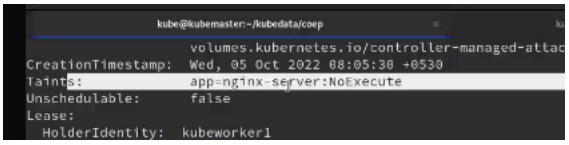


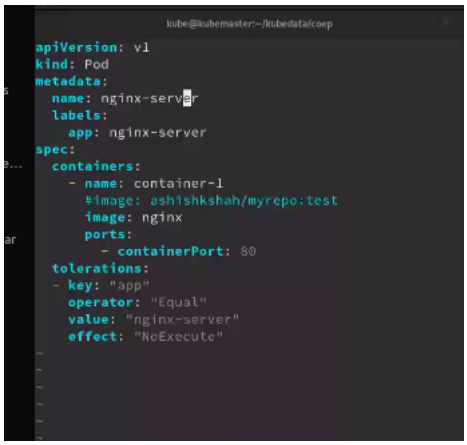
scheduling and taints,tolerances in kubernetes

Manual scheduling on node using nodeName property



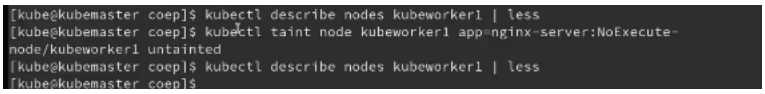
tolerations is kind of exception



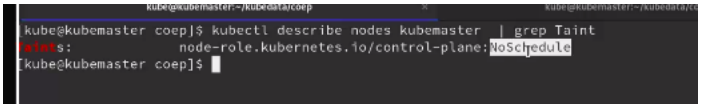


this pod will still be scheduled on kubeworker1, having that taint

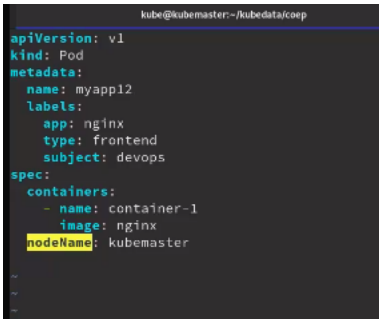
REMOVING TAINT



master node has a by default taint of NoSchedule.



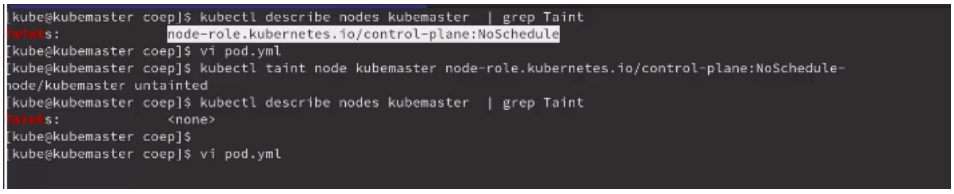
So we can run a pod manually on kubemaster, as:



Untainting master node:

`kubectl taint node kubemaster node-role.kubernetes.io/control-plane:NoSchedule-`

here '-' sign is important.

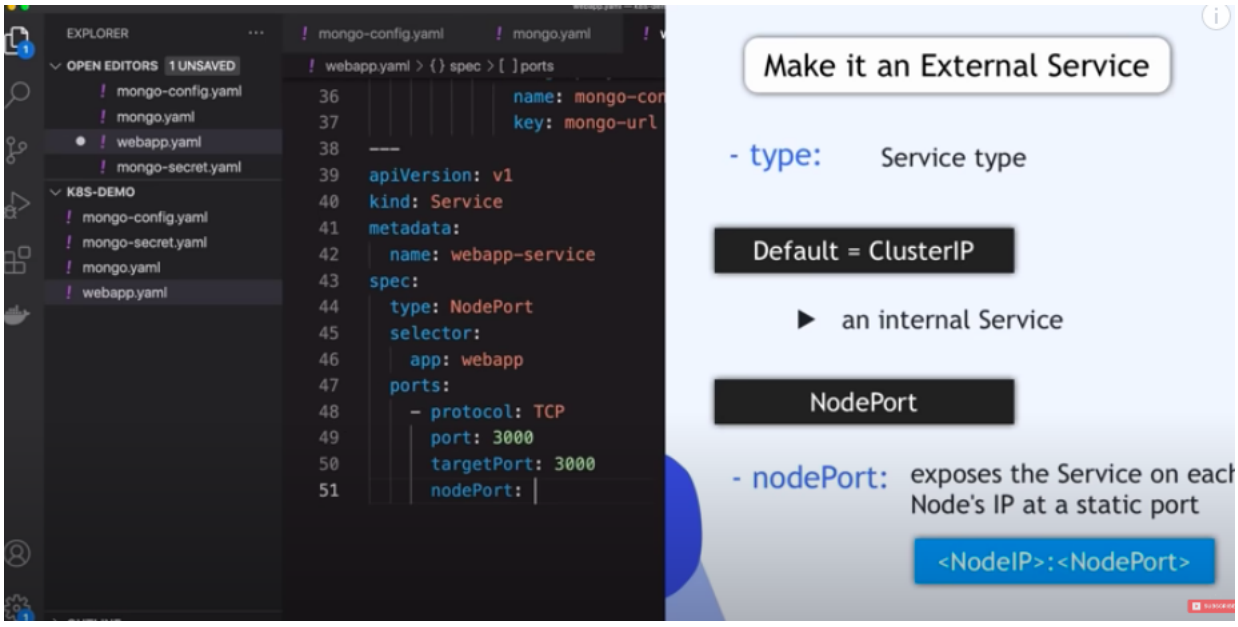


here we see pods are scheduled on kubeworker1 and kubeworker2 nodes, now we taint those nodes to NoExecute and run `kubectl get pods -o wide` and see all the pods are deleted due to above taints, then we create a new pod, which is now scheduled on kubemaster(which was untained from NoSchedule before running these commands)

```
[kube@kubemaster coep]$ kubectl get pods -o wide
NAME     READY   STATUS              RESTARTS   AGE   IP            NODE          NOMINATED NODE   READINESS GATES
nginx    1/1     Running             0          60s   10.85.0.136   kubeworker1   <none>           <none>
nginx1   1/1     Running             0          50s   10.85.0.159   kubeworker2   <none>           <none>
nginx3   1/1     Running             0          42s   10.85.0.160   kubeworker2   <none>           <none>
nginx4   1/1     Running             0          34s   10.85.0.137   kubeworker1   <none>           <none>
nginx5   0/1     ContainerCreating   0          1s    <none>        kubeworker1   <none>           <none>
[kube@kubemaster coep]$
[kube@kubemaster coep]$
[kube@kubemaster coep]$
[kube@kubemaster coep]$
[kube@kubemaster coep]$ kubectl taint node kubeworker1 app=nouse:NoExecute
node/kubeworker1 tainted
[kube@kubemaster coep]$ kubectl taint node kubeworker2 app=nouse:NoExecute
node/kubeworker2 tainted
[kube@kubemaster coep]$ kubectl get pods -o wide
No resources found in default namespace.
[kube@kubemaster coep]$ kubectl run --image nginx nginx
pod/nginx created
[kube@kubemaster coep]$ kubectl get pods -o wide
NAME    READY   STATUS              RESTARTS   AGE   IP       NODE         NOMINATED NODE   READINESS GATES
nginx   0/1     ContainerCreating   0          2s    <none>   kubemaster   <none>           <none>
```

here worker nodes have taint of noschedule, and a pod is already running on master, then we set taint of noschedule on master which shows that nginx2 isn't scheduled on any node(<none>) then now if we untain any node, and check the status then it will be scheduled on untained node

```
[kube@kubemaster coep]$ kubectl taint node kubemaster node-role.kubernetes.io/control-plane:NoSchedule
node/kubemaster tainted
[kube@kubemaster coep]$ kubectl describe nodes kubemaster  | grep Taint
Taints:             node-role.kubernetes.io/control-plane:NoSchedule
[kube@kubemaster coep]$ kubectl run --image nginx nginx2
pod/nginx2 created
[kube@kubemaster coep]$ kubectl get pods -o wide
NAME     READY   STATUS    RESTARTS   AGE   IP            NODE         NOMINATED NODE   READINESS GATES
nginx    1/1     Running   0          41s   10.85.1.115   kubemaster   <none>           <none>
nginx2   0/1     Pending   0          2s    <none>        <none>       <none>           <none>
[kube@kubemaster coep]$ kubectl get pods -o wide
NAME     READY   STATUS    RESTARTS   AGE   IP            NODE         NOMINATED NODE   READINESS GATES
nginx    1/1     Running   0          57s   10.85.1.115   kubemaster   <none>           <none>
nginx2   0/1     Pending   0          18s   <none>        <none>       <none>           <none>
[kube@kubemaster coep]$
```



In command, `kubectl exec myapp -- bash`, what does empty "--" signify in the command?

The double dash (`--`) separates the arguments you want to pass to the command from the kubectl arguments.

Storage in Kubernetes:

Persistent Volumes:

Developer creates PersistentVolumeClaim file, and pod file. While kubernetes administrator creates PersistentVolume kind of file.

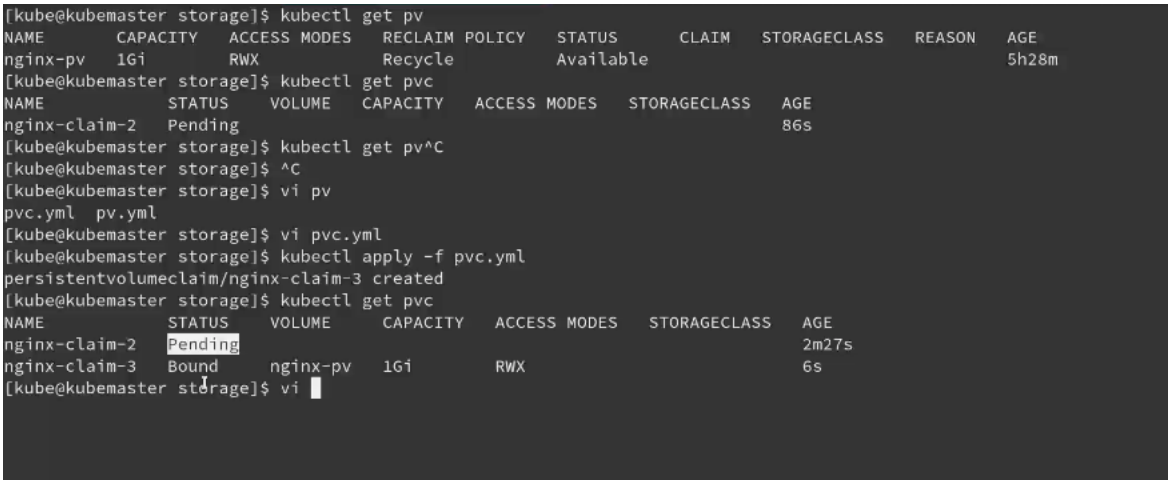
`kubectl get pvc` to see all the persistent volume claims to volumes

`kubectl get pv`

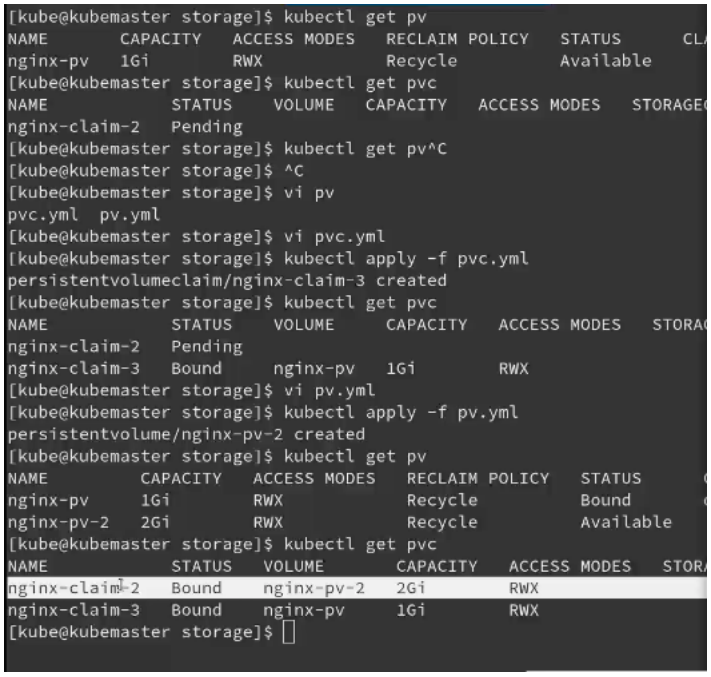
If we create a claim of more than the volume available, then the claim remains in pending state, that can be seen using `kubectl get pvc`

here nginx-claim-2 tried to get 2gb from volume nginx-pv(which has only 1gb space) hence it is in pending state.

then we create nginx-claim-3 which requests smaller space and it is bound



Then we create nginx-pv-2 with larger available volume and see that claim-2,which was pending is also bound.


[kube@kubemaster storage]$ kubectl get pv
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CL
nginx-pv 1Gi RWX Recycle Available
[kube@kubemaster storage]$ kubectl get pvc
NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGE
nginx-claim-2 Pending
[kube@kubemaster storage]$ kubectl get pv^C
[kube@kubemaster storage]$ ^C
[kube@kubemaster storage]$ vi pv
pvc.yml pv.yml
[kube@kubemaster storage]$ vi pvc.yml
[kube@kubemaster storage]$ kubectl apply -f pvc.yml
persistentvolumeclaim/nginx-claim-3 created
[kube@kubemaster storage]$ kubectl get pvc
NAME STATUS VOLUME CAPACITY ACCESS MODES STORA
nginx-claim-2 Pending
nginx-claim-3 Bound nginx-pv 1Gi RWX
[kube@kubemaster storage]$ vi pv.yml
[kube@kubemaster storage]$ kubectl apply -f pv.yml
persistentvolume/nginx-pv-2 created
[kube@kubemaster storage]$ kubectl get pv
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS
nginx-pv 1Gi RWX Recycle Bound
nginx-pv-2 2Gi RWX Recycle Available
[kube@kubemaster storage]$ kubectl get pvc
NAME STATUS VOLUME CAPACITY ACCESS MODES STORA
nginx-claim-2 Bound nginx-pv-2 2Gi RWX
nginx-claim-3 Bound nginx-pv 1Gi RWX
[kube@kubemaster storage]$