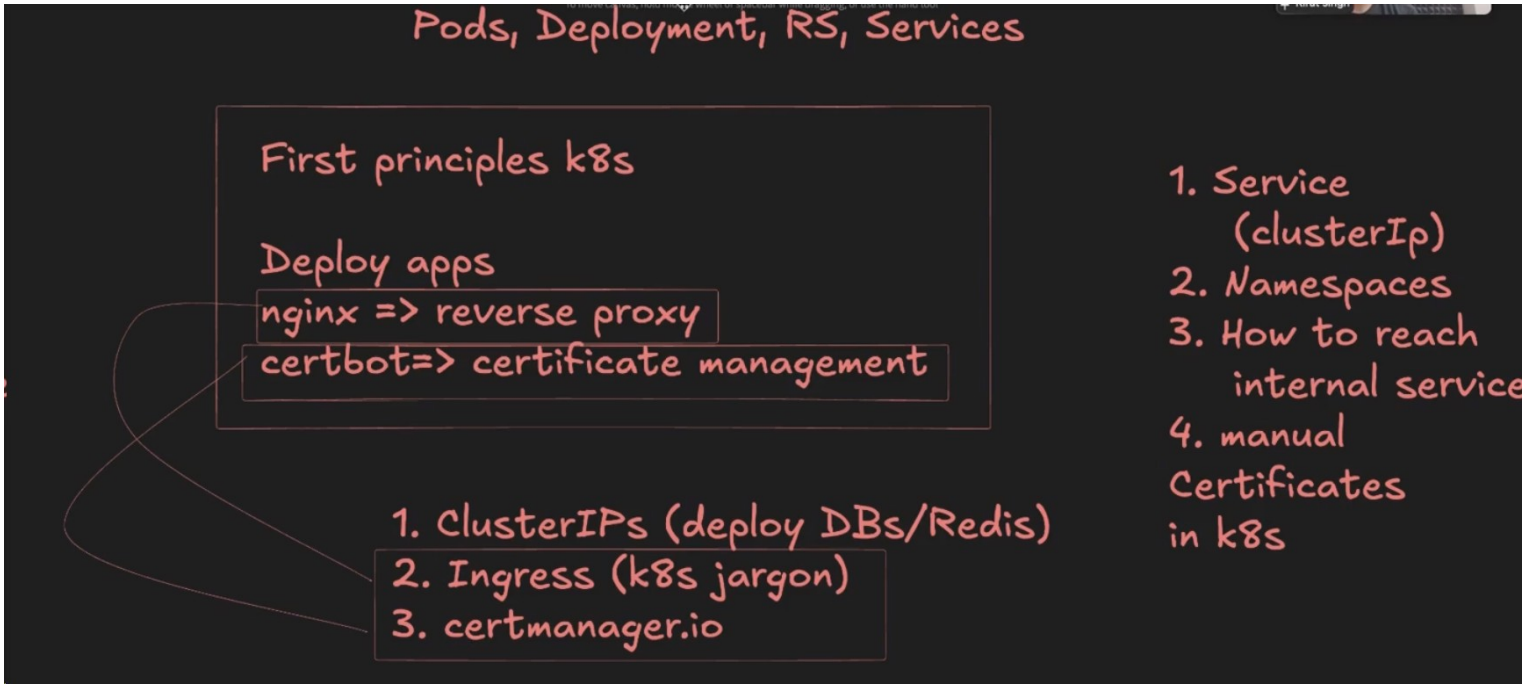


Currently trying to do everything from the first principle for understanding but mainly we will be using the ingress, certmanager.io these things only in the production
you can also use civo.com for the k8s cluster

WHAT IS NAMESPACE?
It just helps to create a hirerachy in the system
In Kubernetes, a namespace is a way to divide cluster resources between multiple users/teams. Namespaces are intended for use in environments with many users spread across multiple teams, or projects, or environments like development, staging, and production.
Like be people will see different pods , fe people will see difference pods buts all the pods are running in one cluster only

kubectl get pods
it gets you the pods in the default namespace

now we will create a k8s cluster and a lb in the DO, vulter very difficult in aws

currently creating a kubernetes cluster

creating a kubernetes cluster keep a name(backend-week-35)

wait until the cluster is provisined

then download the config file and change in the .kube folder

kubectl get pods(shows pods of all the team)

if I want to find like the pods of just the backend pod it will be very difficult to read the only backend team pods for that we use the namespace to segregate based on the team or whatever we want

AND WHENEVER YOU DO KUBECTL GET PODS(YOU GET THE PODS IN THE DEFAULT NAMESPACE)

Create a new namespace

kubectl create namespace backend-team
kubectl get namespaces(return all namespaces)

we can add pod directly to a namespace using the manifest.yml file

```yaml
apiVersion: v1
kind: Pod
metadata:
    name: nginx
    namespace: backend-team
spec:
    containers:
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
```

in this we have mentioned the namespace so when we will do

kubectl get pods (It will not show nginx here)

kubectl get pods -n backend-team (this will show this pod)

```
> ~ kubectl get pods
No resources found in default namespace.
> ~ kubectl create namespace backend-team
namespace/backend-team created
> ~ kubectl get namespace
NAME              STATUS   AGE
backend-team      Active   7s
default           Active   9m27s
kube-node-lease   Active   9m27s
kube-public       Active   9m27s
kube-system       Active   9m27s
> ~ kubectl get pods -n backend-team
No resources found in backend-team namespace.
> ~ kubectl apply -f manifest.yml
pod/nginx created
> ~ kubectl get pods
No resources found in default namespace.
> ~ kubectl get pods -n backend-team
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          27s
> ~ kubectl get pods -n default
No resources found in default namespace.
```

and now you cannot directly delete a pod kubectl delete pod nginx (NOPE bcoz default namespace mai nhi hai)

```
> ~ ⎈ kubectl delete pod nginx -n default
Error from server (NotFound): pods "nginx" not found
> ~ ⎈ kubectl delete pod nginx -n backend-team
pod "nginx" deleted from backend-team namespace
> ~
```

now to delete we have to add athe namespace
kubectl delete pod nginx -n backend-team

```
> ~ kubectl get pods -n backend-team
No resources found in backend-team namespace.
> ~ kubectl apply -f manifest.yml
deployment.apps/nginx-deployment created
> ~ kubectl get deployment
No resources found in default namespace.
> ~
> ~
> ~ kubectl get deployment -n backend-team
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3            3           17s
> ~ kubectl get pods -n backend-team
NAME                                READY   STATUS    RESTARTS
AGE
nginx-deployment-59c4c87bc6-6znf5   1/1     Running   0
22s
nginx-deployment-59c4c87bc6-n667s   1/1     Running   0
22s
nginx-deployment-59c4c87bc6-rf9p2   1/1     Running   0
22s
> ~
```

same thing for the deployment also
if you create that that cannont be accessed without the name the yml file is like this mentioned in which we mentioned the namespace

manifest.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: backend-team
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

NOW TO CHANGE THE DEFAULT NAMESPACE TO THE BACKEND-TEAM NAME SPACE
CMD IS

kubectl config set-context --current --namespace=backend-team
(NOW THE DEFAULT IS K GET PODS -N BACKEND-TEAM)

now kubectl get pods will show the backend-team pods directly without using the -n thing

```
> ~ ⎈ kubectl create namespace frontend-team
namespace/frontend-team created
> ~ kubectl get pods
NAME                                READY   STATUS    RESTARTS
AGE
nginx-deployment-59c4c87bc6-6znf5   1/1     Running   0
5m16s
nginx-deployment-59c4c87bc6-n667s   1/1     Running   0
5m16s
nginx-deployment-59c4c87bc6-rf9p2   1/1     Running   0
5m16s
> ~ kubectl get pods -n frontend-team
No resources found in frontend-team namespace.
> ~ kubectl get pods -n default
No resources found in default namespace.
```

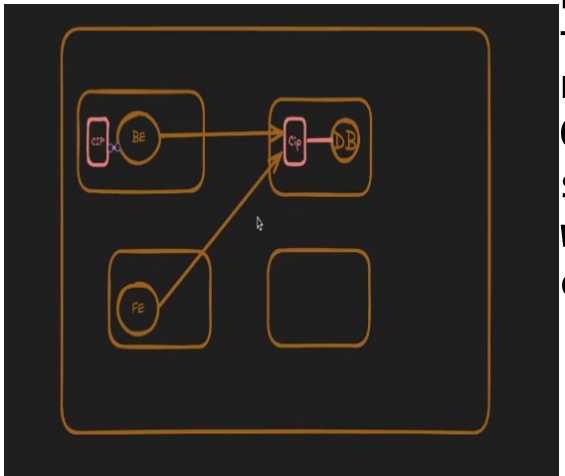
revert back the kubectl config
kubectl config set-context --current --namespace=default

k8s, vulter, civo all are the same one only

now we cloned the repo which contains the code and the docker file so we are pushing that to docker hub

docker build --platform=linux/amd64 -t pdgamersg/backend-pg:2 .

WHAT IS CLUSTER IP IN THE K8S?
It helps to export the ports just inside the pods not outside the cluster eg db can talk to the fe , be inside the pods
for now the be is exposed to the other nodes using the CLUSTERIP
see in this we have written the type as cluster ip



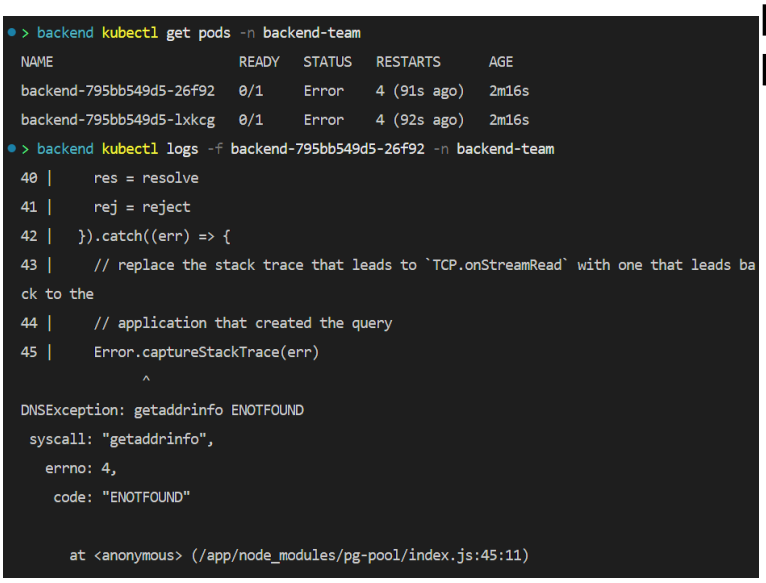

```yaml
apiVersion: v1
kind: Service
metadata:
  name: backend
  namespace: backend-team
spec:
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 3000
      targetPort: 3000
  type: ClusterIP
```

PODS WHICH ARE MADE ARE ACCESSIBLE NOT TO ANY OTHER PODS AND NOT EVEN THE OUTSIDE WORLD REMEMBER THIS

TO MAKE THEM ACCESSIBLE WE USE THE CLUSTERIP, LB, NODEPORT

kubectl apply -f .\manifest.yml
the repo mainfest file we are applying

to debug if any issue in the code we can use this kubectl logs -f pod_name -n backend-team

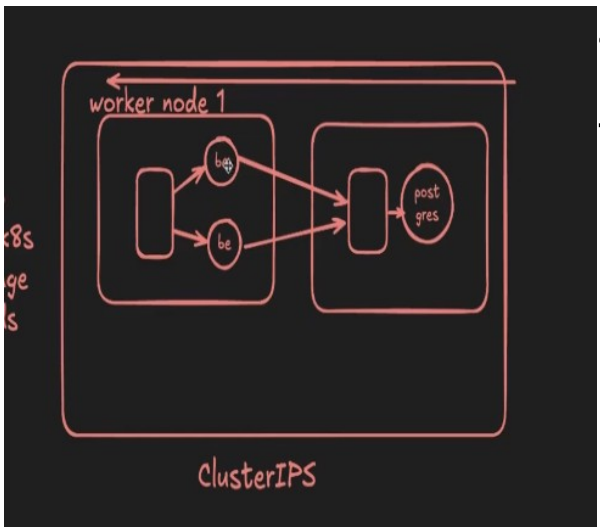


this error was happenning because there was a pg db code in the file and the pg db was not started

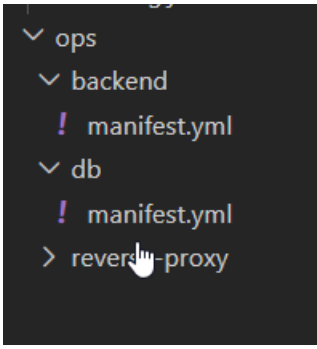
connectionString:
"postgres://postgres:postgres@db.default.svc.cluster.local:5432/postgres",
});

id pass db name of the service(default)
svc = service
5432 par exposed hai postgres
this string in present in the index.ts and the postgres and postgress is the id and pass we choose in the db ka

manifest.yml file mai tabhi usko yaa likhenge and phele db chalu hoga



actually andar yeh hoga ki joh be hai dono ka and postgress hai yeh teeno exposed rhenge



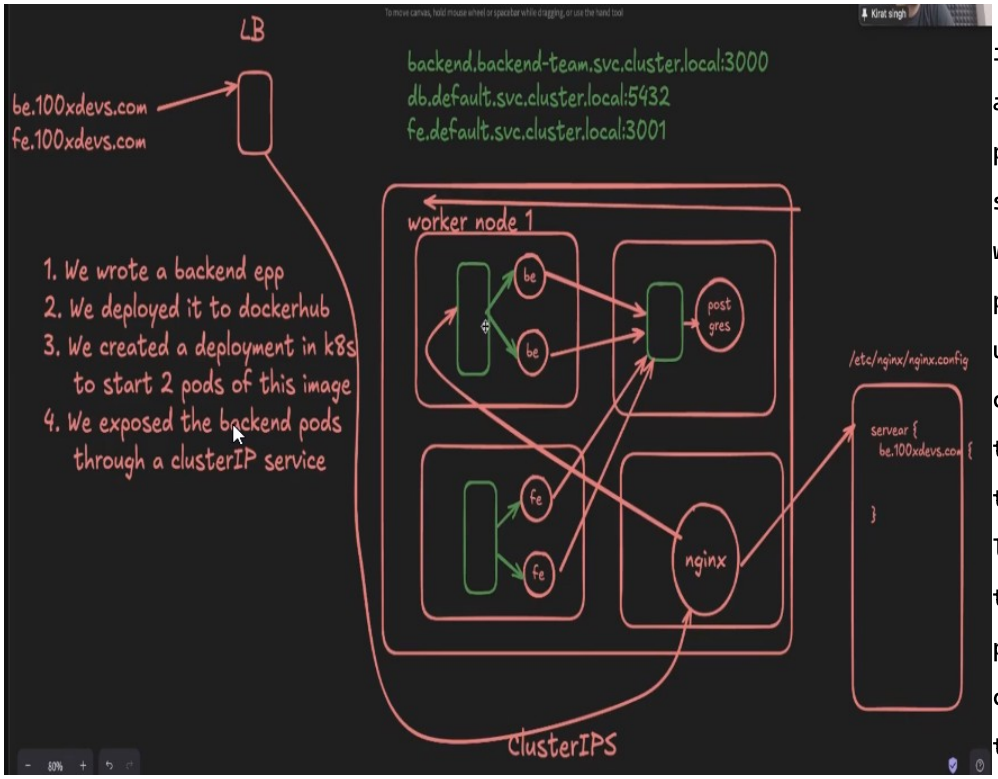
as we have 3 yml file in the ops we started first the db manifest file and then the backend manifest file

so its like before the k8s pods were trying to find the be but they couldn’t so for that we started the db and now we have to force restart the backend pods
scale the pods to 0
kubectl scale deployment backend -n backend-team --replicas 0

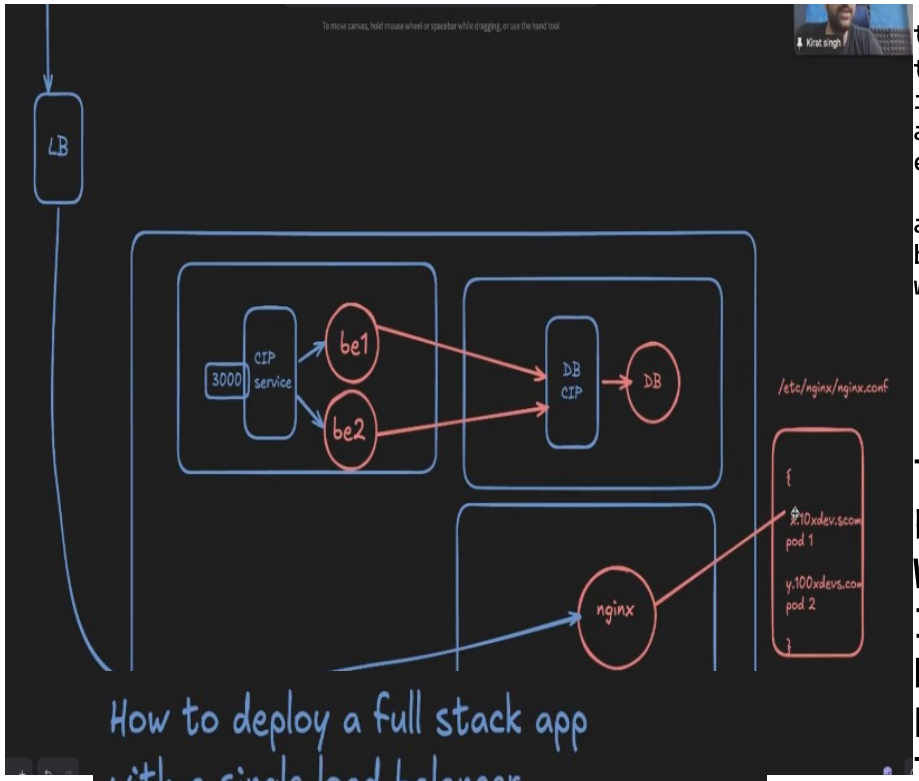
kubectl get pods -n backend-team (this will show they are terminating)
kubectl scale deployment backend -n backend-team --replicas 2
now they will again start running

kubectl logs -f backend-795bb549d5-6nzcw -n backend-team
OUTPUT:
Server is running on port 3000

ENJOY



in this what I understood is we have a some pods running be, fe and postgres as db and also one as nginx so we are using clusterIP in this in which fe and be and talk to the postgres db so the issue before using the nginx was that we like created one lb for the be to handle the request then we created one for the fe and then if we later added like the ws then we made one for that so to fix that we made one more pod in which we will run the nginx code which will have to logic of if the request is to fe.100xddevs.com



then it will go to this route and if the request is to this be route then it will go to this router and later if any ws server is added just we have to edit the nginx conf file

also that there is now just a 1 lb between the nginx and the user which will send the request to the nginx

THIS WAS BASCIALLY A FIRST PRINICPLE APPROACH WHEN THERE WAS NO INGRESS WHICH UNDER THE HOOD DOES THE SAME STUFF BUT MAKES IT EASY FOR THE DEV

THIS WAS OUR TASK WE DID I UNDERSTOOD SOME PART OF IT BUT NOT THE ALL BUT YAA I ENJOYED TILL THE END

Recapping how to run this locally

Creating a cluster

- Create a kind.yml file locally

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 30007
    hostPort: 30007
- role: worker
  extraPortMappings:
  - containerPort: 30007
    hostPort: 30008
- role: worker
```

- Run the cluster locally

```
kind create cluster --config kind.yml --name local2
```

- Run docker ps to confirm that the cluster is running

```
docker ps
```

Creating a pod

- Create a pod manifest (pod.yml)

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

- Apply the pod manifest

```
kubectl apply -f pod.yml
```

- Check if the pod exists now

```
kube get pods
```

- Check the logs

```
kubectl logs -f nginx
```

- Delete the pod

```
kubectl delete pod nginx
```

Creating a replicaset

- Create the replicaset manifest

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

- Apply the replicaset manifest

```
kubectl apply -f rs.yml
```

- Check the number of pods running now

```
kubectl get pods
```

- Try deleting a pod, and ensure it gets restarted
- Delete the replicaset

```
kubectl delete rs nginx-replicaset
```

Creating a Deployment

- Create a deployment manifest (deployment.yml)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

- Apply the manifest

```
kubectl apply -f deployment.yml
```

- Check the rs that exist now

```
kubectl get rs
```

- Check the pods that exist now

```
kubectl get pods
```

- Try creating a new deployment with a wrong image name

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
```

```
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx2:latest
        ports:
        - containerPort: 80
```

- Ensure that the old pods are still running

kubectl get pods

Keep the deployment, it’ll come in handy in the 4th slide

FOR LONG RUNNING PROCESS WE ALWAYS MAKE DEPLOYMENT RATHER THAN MAKING A POD

CURRENTLY STARTING A CLUSTER AND WE CAN ALSO USE THE KIND.YML TO START THE CLUSTER

created a k8s cluster on the DO and then copied the config file to C:\Users\Administrator\.kube\config file

cd ~/.kube (we can do this directly to go to the folder)

to copy we use
cp new_config config
created the manifest file

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

kubectl apply -f .\manifest.yml

kubectl get deployment(shows the status of the pods)
kubectl get pods

kubectl logs -f nginx-deployment-59c4c87bc6-79f68(to get the logs of the pod)

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: LoadBalancer
```

then we addded the loadbalancer logic and ran the manifest file again
kubectl get svc we can see the lb is loading and running

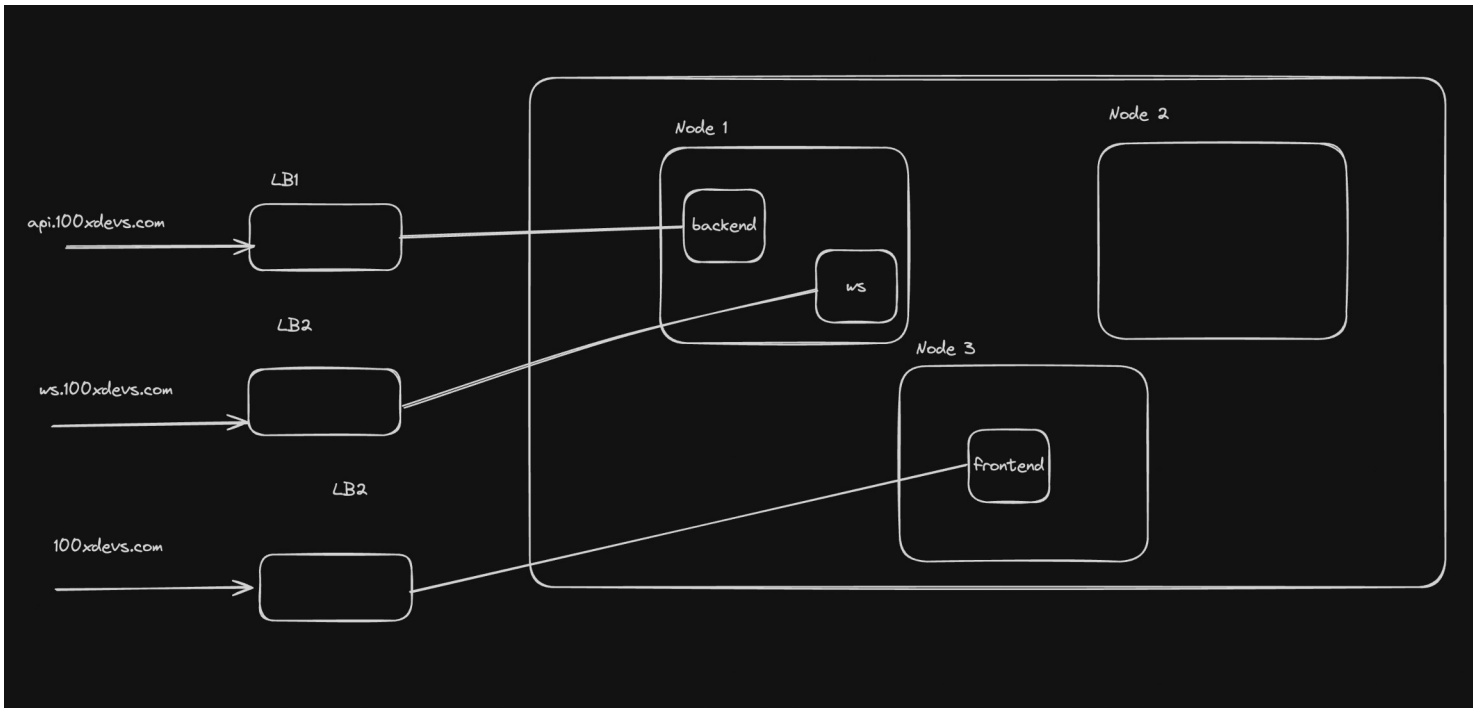
just the problem in using the service loadbalancer is you have to create a new lb for each server like for the fe, be , ws etc….

Downsides of services

Services are great, but they have some downsides -

Scaling to multiple apps

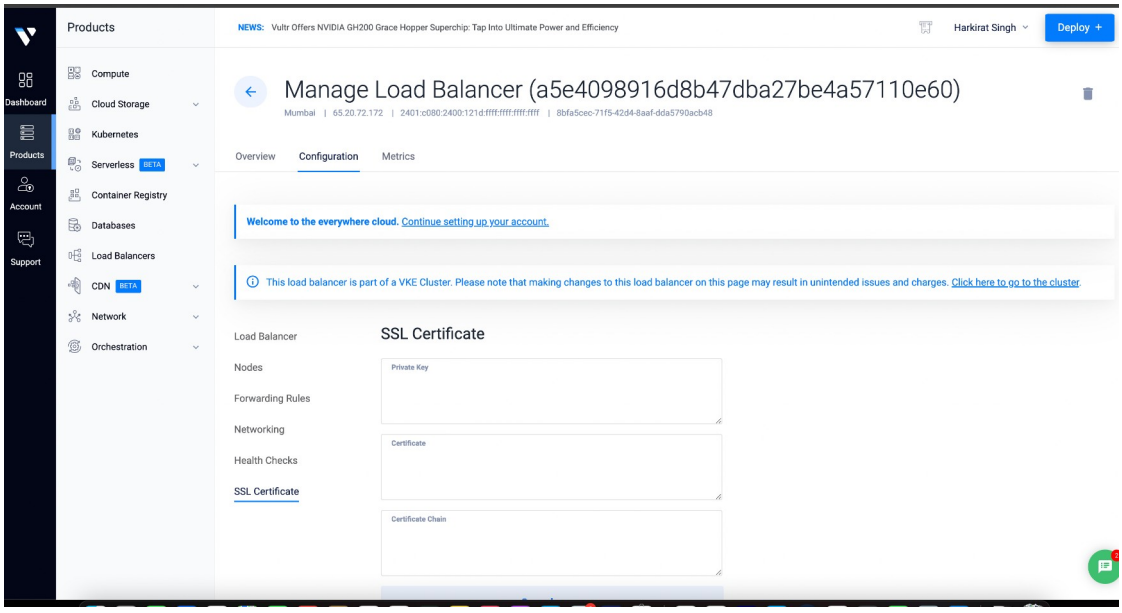
1. If you have three apps (frontend, backend, websocket server), you will have to create 3 saparate services to route traffic to them. There is no way to do centralized traffic management (routing traffic from the same URL/Path-Based Routing)
2. There are also limits to how many load balancers you can create



Multiple certificates for every route

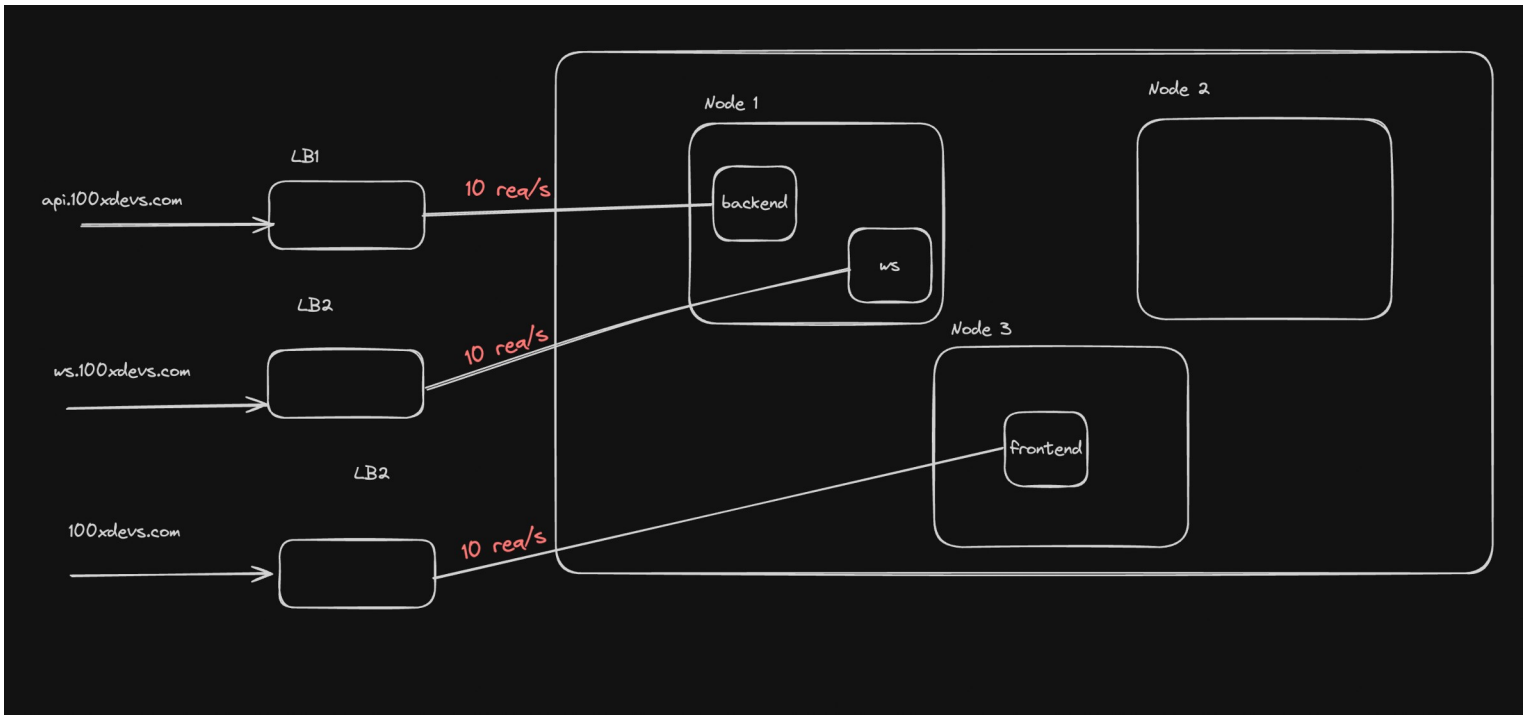
You can create certificates for your load balancers but you have to maintain them outside the cluster and create them manually

You also have to update them if they ever expire



No centralized logic to handle rate limitting to all services(REQ WE ARE SENDING TO A LB)

Each load balancer can have its own set of rate limits, but you cant create a single rate limitter for all your services.



FOR SOLVING THE ISSUE OF LB WE USE INGRESS

Ingress and Ingress Controller

Ref - https://kubernetes.io/docs/concepts/services-networking/ingress/

An API object that manages external access to the services in a cluster, typically HTTP.

Ingress may provide load balancing, SSL termination(THIS MEANS BAHAR SAE JOH REQUEST AA RHI HAI WOH HOGI HTTPS WALI FHIR LB CHECK KAREGA AND THEN ANDAR FHIR HTTP REQUEST HII CHALEGI) and name-based virtual hosting.

What is Ingress?

Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

Here is a simple example where an Ingress sends all its traffic to one Service:

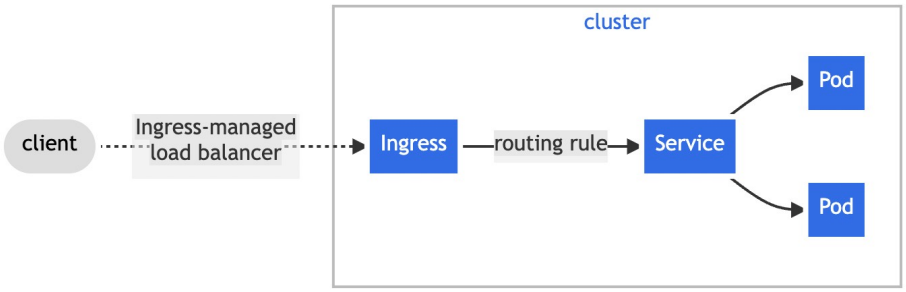
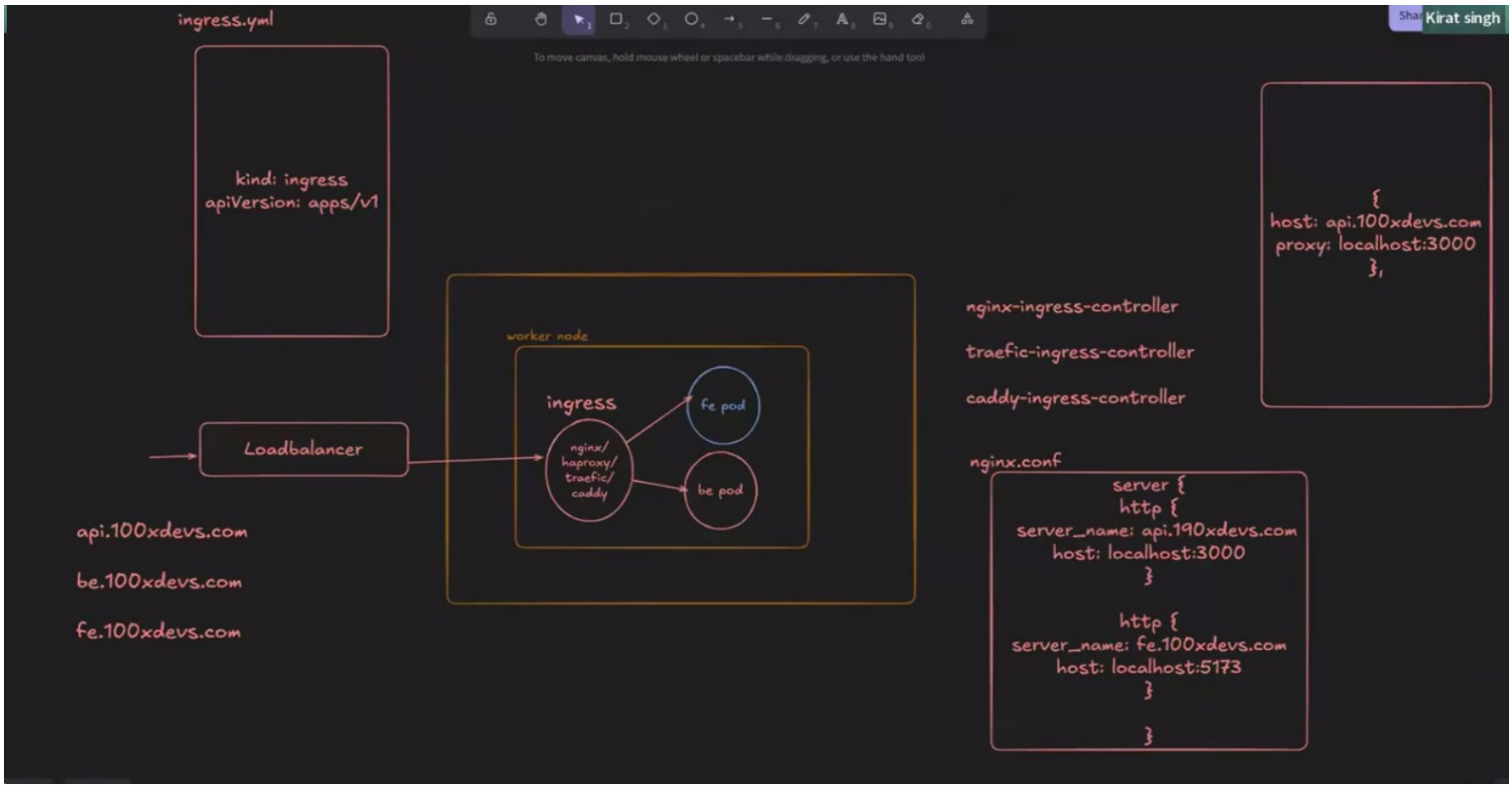


Figure. Ingress

An Ingress does not expose arbitrary ports or protocols. Exposing services other than HTTP and HTTPS to the internet typically uses a service of type Service.Type=NodePort or Service.Type=LoadBalancer.

LAST WEEK WE DID IT FIRST PRINCIPLE BASED BUT NOW ITS LIKE THE SAME THING BUT WE DONT HAVE TO CREATE PODS OURSELF

INGRESS MAINLY HELPS IN CREATING ONE ENTRY POINT FOR THE SYSTEM



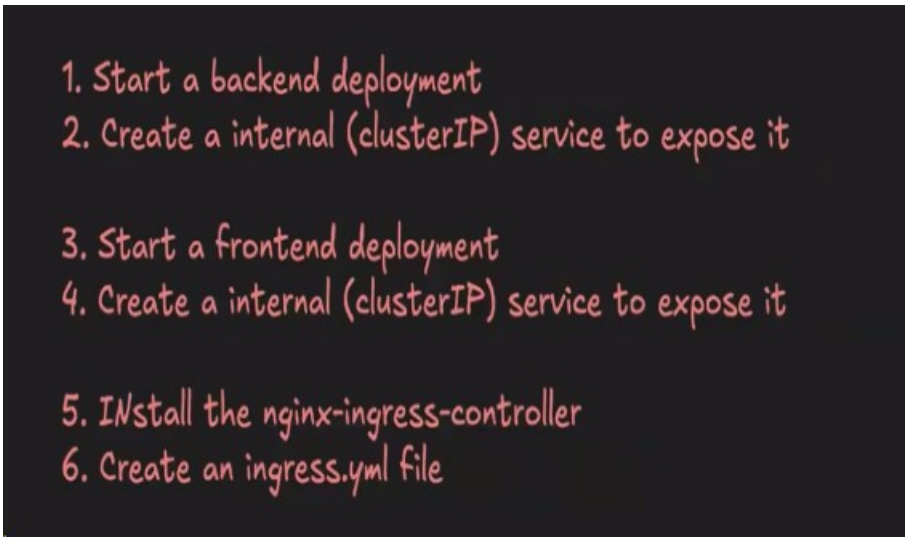
SO IT WAS VERY DIFFICULT FOR K8S TO CHOOSE WHICH REVERSE PROXY TO USE FOR THERE AS THE MAIN ONE SO THEY CREATED INGRESS IN WHICH WE CAN SPECIFY WHICH ONE WE WANT TO USE AND THEN MAKE THE CONF FILE ACCORDING TO THAT
BUT FOR THE NGINX, HAPROXY, TRAFIC, CADDY TO BRING TO THE CODE WE HAVE TO FIRST BRING THE CONTROLLER

like these proxies have different use case like in bolt we need to have different url for different users which the nginx cannot do for that we need to use traffic, caddy etc

CADDY IS THE BEST ONE FOR THE DYNAMIC ROUTE CONFIG

FOR EACH INGREES YOU WANT TO USE YOU HAVE TO START A CONTROLLER

TASK

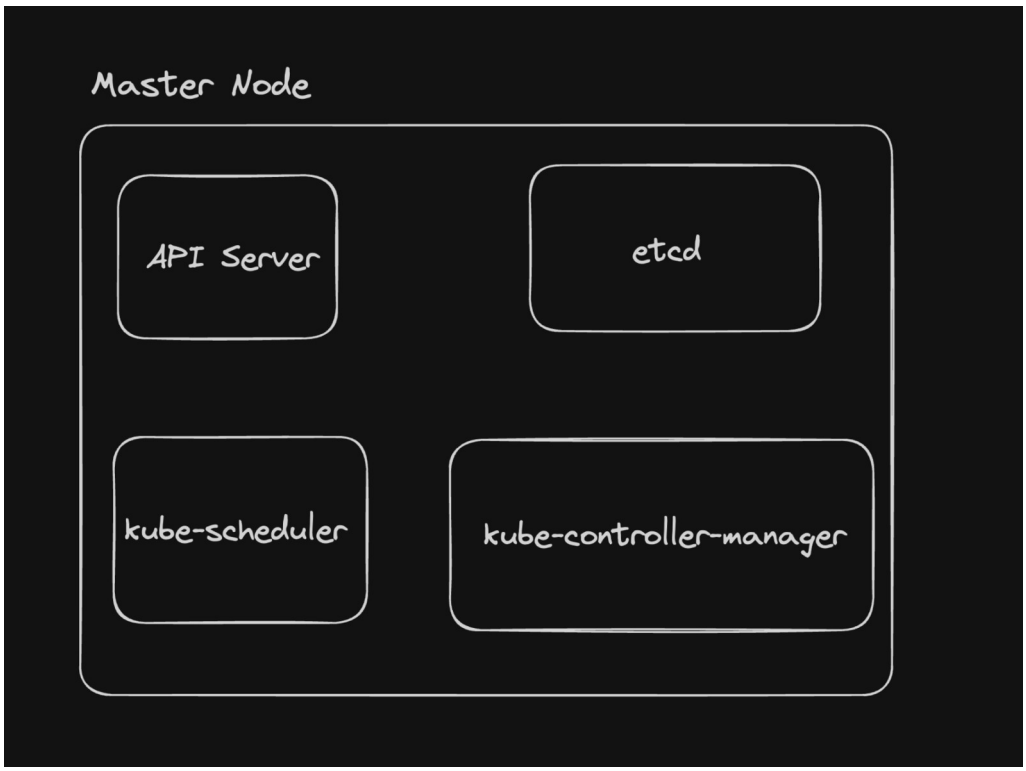


FIRST WE WILL INSTALL THE NGINX INGRESS CONTROLLER

Ingress controller

If you remember from last week, our control plane had a controller manager running.

Ref - https://projects.100xdevs.com/tracks/kubernetes-1/Kubernetes-Part-1-3



The kube-controller-manager runs a bunch of controllers like

1. Replicaset controller
2. Deployment controller

etc
If you want to add an ingress to your kubernetes cluster, you need to install an ingress controller manually. It doesn't come by default in k8s

Famous k8s ingress controllers

- The NGINX Ingress Controller for Kubernetes works with the NGINX webserver (as a proxy).
- HAProxy Ingress is an ingress controller for HAProxy.
- The Traefik Kubernetes Ingress provider is an ingress controller for the Traefik proxy.

Full list - https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.12.1/deploy/static/provider/cloud/deploy.yaml
on this url full nginx yaml file is present

so just do
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.12.1/deploy/static/provider/cloud/deploy.yaml

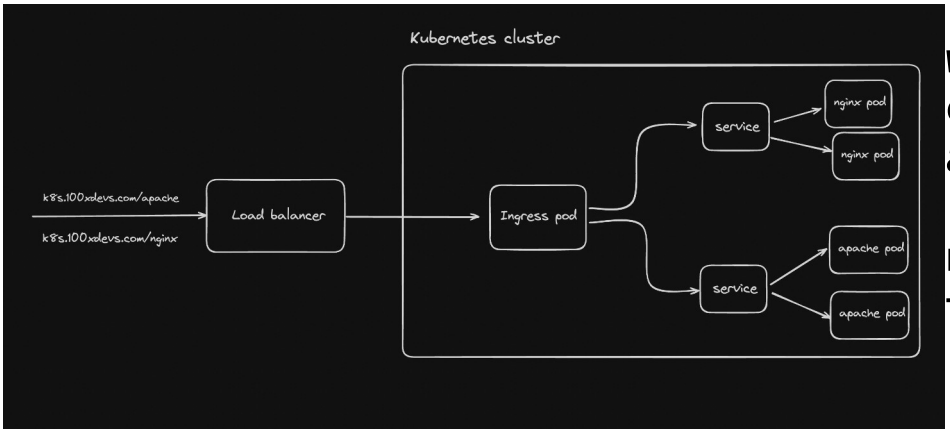
for installing the nginx ingress controller

whenver we start the controller it start it in a namespace

```
> week-36-k8s-manifest kubectl get pods -n ingress-nginx
NAME                                        READY   STATUS      RESTARTS   A
GE
ingress-nginx-admission-create-j2nhf        0/1     Completed   0          4
4s
ingress-nginx-admission-patch-8tsj6         0/1     Completed   0          4
4s
ingress-nginx-controller-6c47c5c5cc-x9qc5   1/1     Running     0          4
4s
> week-36-k8s-manifest kubectl get svc -n ingress-nginx
NAME                                 TYPE           CLUSTER-IP      EXTERNAL
-IP      PORT(S)                      AGE
ingress-nginx-controller             LoadBalancer   10.109.22.192   129.212.
244.59   80:30232/TCP,443:32088/TCP   113s
ingress-nginx-controller-admission   ClusterIP      10.109.16.244   <none>
         443/TCP                      112s
> week-36-k8s-manifest
```

also the controller also starts some services, pods etc

after this we have to start some fe and be and its service before that we closed the deployment and the svc which were running earlier



write now we have created the ingress pod, and lb only

now we have to create the fe and be

kubectl get svc -n ingress-nginx(this shows nginx wali ingress abhi bhi chal rhi hai)

```
> week-36-k8s-manifest kubectl get svc -n ingress-nginx
NAME                                 TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S
)                    AGE
ingress-nginx-controller             LoadBalancer   10.109.22.192   129.212.244.59   80:302
32/TCP,443:32088/TCP   15m
ingress-nginx-controller-admission   ClusterIP      10.109.16.244   <none>           443/TC
```

this shows ingress nginx still works

now according to the question we created 2 manifest file one for the be and one for the fe

be one

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
```

```
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80

apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

fe one

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: apache
  template:
    metadata:
      labels:
        app: apache
    spec:
      containers:
      - name: apache
        image: apache
        ports:
        - containerPort: 80

apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
```

```
  selector:
    app: apache
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

and they are also exposed inside only because of the clusterIP

ingress.yml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: web-apps-ingress
  namespace: default
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - host: week-36.pallabdas.me
    http:
      paths:
      - path: /backend
        pathType: Prefix
        backend:
          service:
            name: backend-service
            port:
              number: 80
      - path: /frontend
        pathType: Prefix
        backend:
          service:
            name: frontend-service
            port:
              number: 80
```

now see we changed the path and the host url and the classname changes on the basic of which controller you import in the starting

```
kubectl apply -f .\manifest1.yml
kubectl apply -f .\manifest2.yml
kubectl apply -f .\ingress.yml
```

this will run all the files and also to the url point the lb

week-36.pallabdas.me
to get the ip of the lb kubectl get svc -n ingress-nginx
from this get the ip of the ingress nginx controller

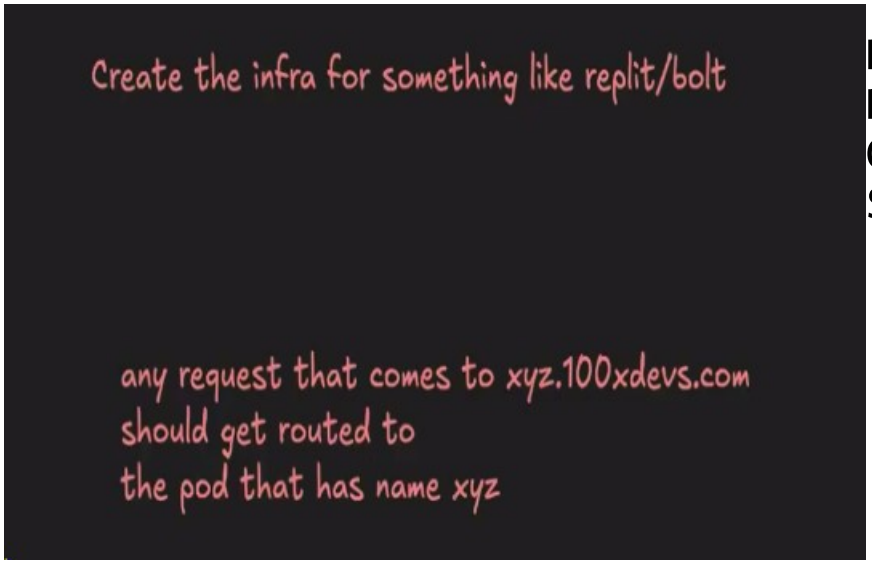
GREAT HACK LIKE YOU CAN TAKE THE IP OF THE LB AND IN THE HOSTS FILE WE CAN JUST WRITE THIS IP POINTS TO THE WEEK-36.PALLABDAS.ME
GOOD HACK THO
THIS IS CALLED AS DNS SPOFFING

IF THE OUTPUT SHOWS 503 MEANS SOMEHTING IS WRONG

--- are very important in the yml file

now on the http://week-36.pallabdas.me/backend it shows the nginx
http://week-36.pallabdas.me/frontend it shows the httpd file

WHICH IS IT WORKS



MOBILE MAGIC HARKIRAT WALE MAI HE HAS USED CADDY CONTROLLER FOR THE NGINX SO I CAN CHECK THAT OUT

ALWAYS REMEMBER CLOSE ALL THE PODS, LB AND SERVICES

Secrets and configmaps

Kubernetes suggests some standard configuration practises. These include things like

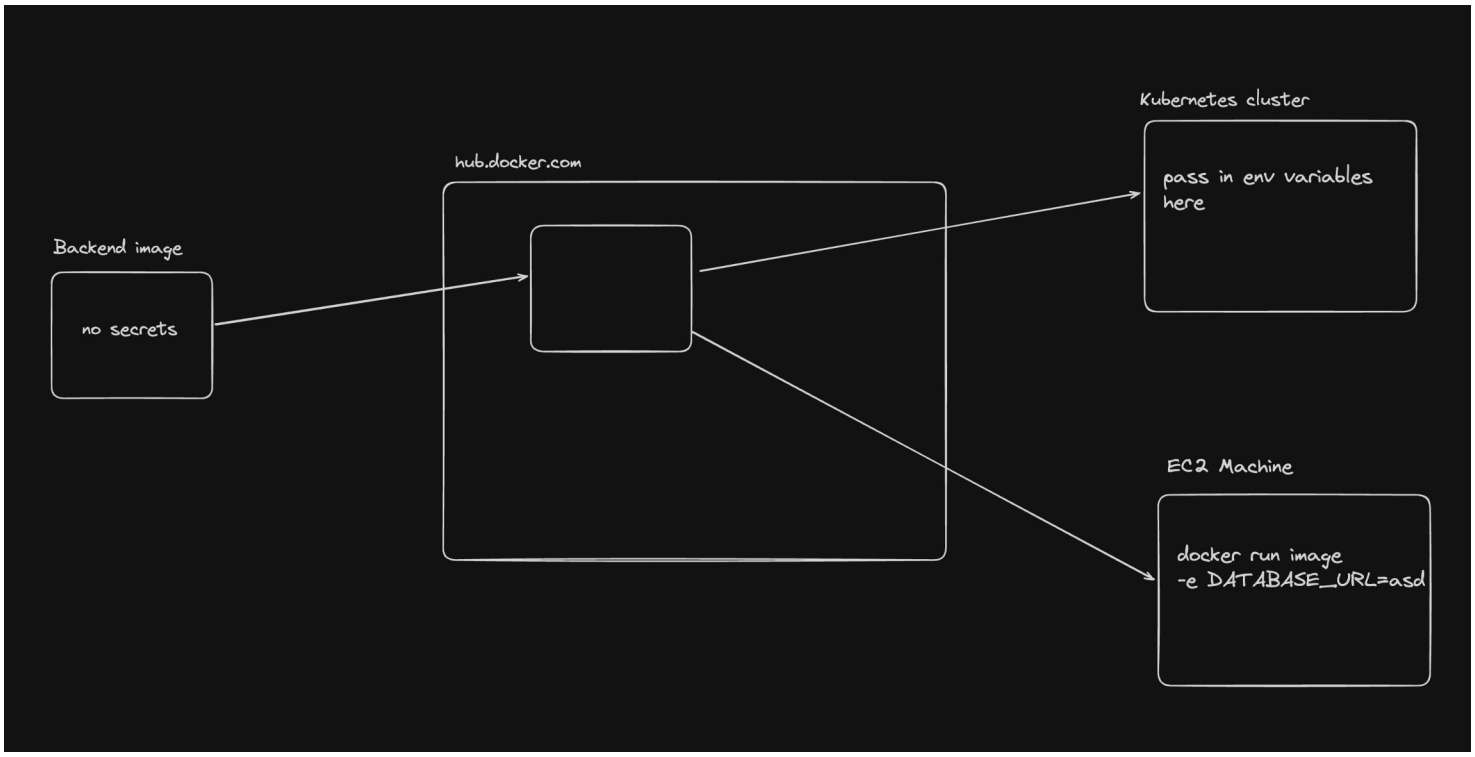
1. You should always create a deployment rather than creating naked pods
2. Write your configuration files using YAML rather than JSON
3. Configuration files should be stored in version control before being pushed to the cluster

Kubernetes v1 API also gives you a way to store configuration of your application outside the image/pod
This is done using

1. ConfigMaps
2. Secrets

Rule of thumb

Don't bake your application secrets in your docker image
Pass them in as environment variables whenever you're starting the container



ConfigMaps

Ref - https://kubernetes.io/docs/concepts/configuration/configmap/
A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.
A ConfigMap allows you to decouple environment-specific configuration from your container images, so that your applications are easily portable.

CREATING A CLUSTER LOCALLY
cat clusters.yml
kind create cluster --config clusters.yml -n local
kind delete cluster -n local
kubectl get pods(will show if any pod is running)
$env:PORT="3000";
$env:DATABASE_URL="postgres://postgres:password@localhost:5342"; bun index.ts
in linux you can run that without the $env thingy

PORT= 3000
DATABASE_URL=postgres://postgres:password@localhost:5342 bun index.ts

or you can do
for terminal
$env:PORT = 3000
$env:DATABASE_URL = postgres
bun index.ts

for the linux
export PORT = 3000
export DATABASE_URL = postgres
bun index.ts

built a docker file and a .dockerignore which included node_modules

Dockerfile content
FROM oven/bun:1

```
WORKDIR /app
COPY ./ ./

RUN bun install

EXPOSE 3000

CMD ["bun","index.ts"]
```

docker build --platform=linux/amd64 -t pdgamersg/todo-app-test:1 .
(the last dot is to access the current directory)

docker push pdgamersg/todo-app-test:1
https://hub.docker.com/r/pdgamersg/todo-app-test
at the above url anyone can access the docker file
docker run -e DATABASE_URL=postgres123 -e PORT=3000 -p 3000:3000 pdgamersg/todo-app-test:1
this command to run the docker file in the system
Now creating the manifest files

CREATED A CM.YML
WHICH HELPS US TO PASS THE CREDS TO THE PODS

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: backend-config
data:
  PORT: "3000"
```

```
kubectl apply -f .\ops\cm.yml
kubetcl get cm(to get the configmaps)
```

configmap is just a key value pair which we are using in the yml file currently we are doing like creating a configmap storing them in that and accessing that using the other file which is not idle because eventually we will upload that also but this will just store the dummy urls and the secrets will store the actual urls

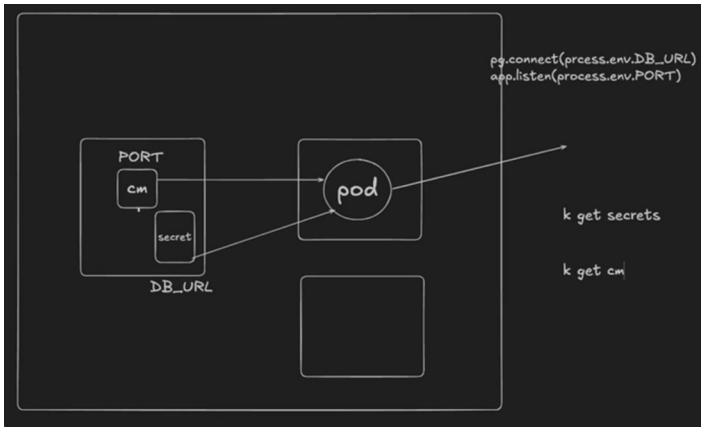
```
kubectl apply -f .\ops\cm.yml
kubectl apply -f .\ops\deployment.yml
```

after the changes always do this

what we are doing is adding secrets in the cm.yml file and accessing it via docker command or through the deployment file

now to push the secrets we use something called SEALED SECRETS which is like the secrets which we can use and is like key value paired thingy first signed and kept as a encrypted key and then sent to the k8s and is decrypted and can be used by the pod
WE WILL STUDY THAT LATER

SO WHAT I UNDERSTAND IS STANDARD PORTS AND ALL ARE STORED IN THE CONFIGMAP AND THE SECRETS STORES THE DB URLS AND ALL
THIS IS JUST A CONVENTION WHICH IS USED BOTH CAN BE ACCESSED EASILY



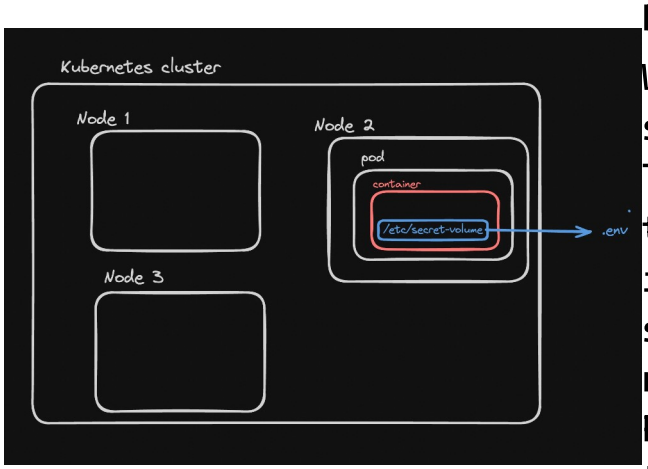
Secrets

Secrets are also part of the kubernetes v1 api. They let you store passwords / sensitive data which can then be mounted on to pods as environment variables. Using a Secret means that you don't need to include confidential data in your application code.
Ref - https://kubernetes.io/docs/concepts/configuration/secret/

Using a secret

- Create the manifest with a secret and pod (secret value is base64 encoded) (https://www.base64encode.org/)

```
apiVersion: v1
kind: Secret
metadata:
  name: dotfile-secret
data:
  database_url: "postgres://username:password@neon.tech/postgres"
```



Base64 encoding

Whenever you're storing values in a secret, you need to base64 encode them. They can still be decoded , and hence this is not for security purposes. This is more to provide a standard way to store secrets, incase they are binary in nature.
For example, TLS (https) certificates that we'll be storing as secrets eventually can have non ascii characters. Converting them to base64 converts them to ascii characters.

THE SECRET WHICH IS WRITTEN INSIDE THE SECRET.YML FILE MUST BE IN FORMAT LIKE BASE64, BASE58 ETC

WE CANNOT WRITE THAT IN PLAIN TEXT, THIS IS NOT FOR SECURTIY PURPOSE AS THAT CAN BE CONVERTED BACK TO ORIGINAL TEXT
THIS IS JUST DONE TO LIKE REMOVE THE /N AND ALL OR THE NEXT LINE IN THE DB URL OR ANY SECRET

cG9zdGdyZXM6Ly91c2VybmFtZTpwYXNzd29yZEBuZW9uLnRlY2gvcG9zdGdyZXM=
this we replace in the secret file instead of the db url

now we changed the secret.yml file

```
apiVersion: v1
kind: Secret
metadata:
  name: backend-secret
data:
  database_url:
"cG9zdGdyZXM6Ly91c2VybmFtZTpwYXNzd29yZEBuZW9uLnRlY2gvcG9zdGdyZXM="
```

and also the deployment.yml file below lines to get the config data from this secret.yml file

secret.yml | deployment.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ecom-backend-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ecom-backend
  template:
    metadata:
      labels:
        app: ecom-backend
    spec:
      containers:
      - name: todo-backend
        image: pdgamersg/todo-app-test:1
        ports:
        - containerPort: 3000
        env:
        - name: PORT
          valueFrom:
            configMapKeyRef:
              name: backend-config
              key: port
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: backend-secret
              key: database_url
```

see in this below we have added secrey key ref which refers to the above file and now when we do
kubectl get pods and then use the pods name to see the logs

kubectl logs -f ecom-backend-deployment-9f7556589-4nzfb

OUTPUT
postgres://
username:password@neon.tech/
postgres
3000

NOW CURRENTLY THE DB URL IS EXPOSED AND ANYONE CAN USE IT SO TO FIX THAT WE USE SEALED SECRET WHICH IS USED TO STORE THE SECRETS
AND THIS WE STUDY IN GITOPS

ConfigMaps vs Secrets

- Creating a ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-config
data:
  key1: value1
  key2: value2
```

- Creating a Secret

```
apiVersion: v1
kind: Secret
metadata:
  name: example-secret
data:
  password: cGFzc3dvcmQ=
  apiKey: YXBpa2V5
```

Key differences

- **Purpose and Usage:**
  - **Secrets**: Designed specifically to store sensitive data such as passwords, OAuth tokens, and SSH keys.
  - **ConfigMaps**: Used to store non-sensitive configuration data, such as configuration files, environment variables, or command-line arguments.
- **Base64 Encoding:**
  - **Secrets**: The data stored in Secrets is base64 encoded. This is not encryption but simply encoding, making it slightly obfuscated. This encoding allows the data to be safely transmitted as part of JSON or YAML files.
  - **ConfigMaps**: Data in ConfigMaps is stored as plain text without any encoding.
- **Volatility and Updates:**
  - **Secrets**: Often, the data in Secrets needs to be rotated or updated more frequently due to its sensitive nature.

    - **ConfigMaps:** Configuration data typically changes less frequently compared to sensitive data.

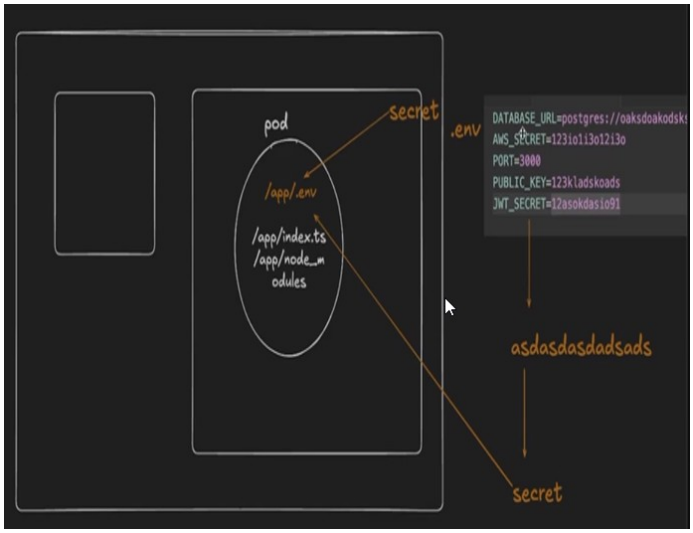
- **Kubernetes Features:**
    - **Secrets:** Kubernetes provides integration with external secret management systems and supports encryption at rest for Secrets when configured properly. Ref https://secrets-store-csi-driver.sigs.k8s.io/concepts.html#provider-for-the-secrets-store-csi-driver
    - **ConfigMaps:** While ConfigMaps are used to inject configuration data into pods, they do not have the same level of support for external management and encryption.

VERY COMMON PRACTICE TO KEEP RECYCLING THE DB PASSWORDS SO THAT ITS NOT PERSISTENT ENOUGH AND NO ONE SHOULD USE THAT OFTEN

NOW TO PROTECT THE SECRETS WE WILL USE SOMETHING CALLED AS VOLUMES WHICH WILL HELP US TO GET THE VOLUMES FROM THE OUTSIDE AND MOUNT THEM IN K8S WHENEVER REQUIRED

WE WILL HAVE A .ENV FILE WHICH WE WILL MOUNT IN THE SECRET.YML FILE

now we will encode the whole .env file and then paste in the secret.yml file



and phele haam aek aek karke mount kar rhe the deployment wali file mai abh haam puri ki puri env file mount kar denge

for that hame 2 chije likhni hoti hai volume and volume mounts

volume helps to mount in the deployment.yml file and volume mount helps to actually use it

index.ts

```
require("dotenv").config({
    file:"./secret/.env"
})

import express  from "express";

const app = express();

console.log(process.env.DATABASE_URL);
console.log(process.env.PORT);

app.get("/",(req,res) =>{
    res.json({
        db:process.env.DATABASE_URL
    })
})
app.listen(process.env.PORT);
```

deployment.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ecom-backend-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ecom-backend
  template:
    metadata:
      labels:
        app: ecom-backend
    spec:
      containers:
      - name: todo-backend
        image: pdgamersg/todo-app-test:1
        ports:
        - containerPort: 3000
        volumeMounts:
        - name: env-file
          readOnly: true
          mountPath: "/app/secret"
      volumes:
      - name: env-file
        secret:
          secretName: backend-secret
```

secrets.yml

```
apiVersion: v1
kind: Secret
metadata:
  name: backend-secret
data:
  .env:
"REFUQUJBU0VfVVJMPXBvc3RncmVzOi8vbGFzbGRrZmpzZGxmawpBV1NfU0VDUkVUPTQwOTgxMjM0a2oyMzRoClBPUlQ9MzAwMApQVUJMSUNfS0VZPTEyMzRsa2poMjM0CkpXVF9TRUNSVQ9MjM0a2oyMzRsa2oyMzRsa2oK"
```

cm.yml is the same and dockerfile is also same

Volumes in docker

Pretext

The following docker image runs a Node.js app that writes peridically to the filesystem - https://hub.docker.com/r/100xdevs/write-random
Nodejs Code

Run it in docker

Try running the image above in your local machine

```
docker run 100xdevs/write-random
```

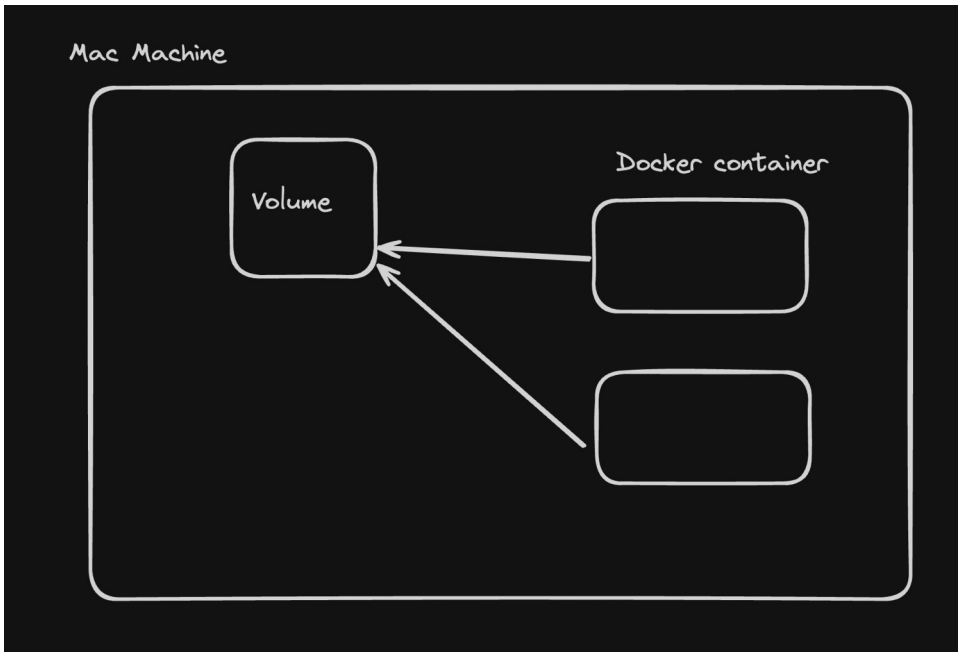
Try going to the container and seeing the contents of the container

```
docker exec -it container_id /bin/bash
cat randomData.txt
```

Where is this file being stored?

The data is stored in the `docker runtime filesystem` . When the container dies, the data dies with it. **This is called ephemeral storage(TEMPEROARY STORAGE)**

Volumes in docker



If you want to persist data across container stops and starts, you can use Volumes in Docker

Bind mounts (THESE MOUNTS STORE THE FILE IN THE SYSTEM AND WHILE USING THE SYSTEM IT MOUNTS THAT FOLDER FROM THE SYSTEM)

Replace the mount on the left with a folder on your own machine

```
docker run -v
/Users/harkiratsingh/Projects/100x/mount:/usr/src/app/generated
100xdevs/write-random
```

Volume Mounts(IN THIS THE FOLDER IS INSIDE THE DOCKER FILE WHICH IT CAN ONLY ACCESS AND USE IT AND IF WE KILL THE DOCKER THEN ALSO THE DOCKER VOLUME PERSIST)

- Create a volume

```
docker volume create hello
```

- Mount data to volume

```
docker run -v hello:/usr/src/app/generated 100xdevs/write-random
```

If you stop the container in either case, the `randomFile.txt` file persists

Volumes in kubernetes
Ref - https://kubernetes.io/docs/concepts/storage/volumes/

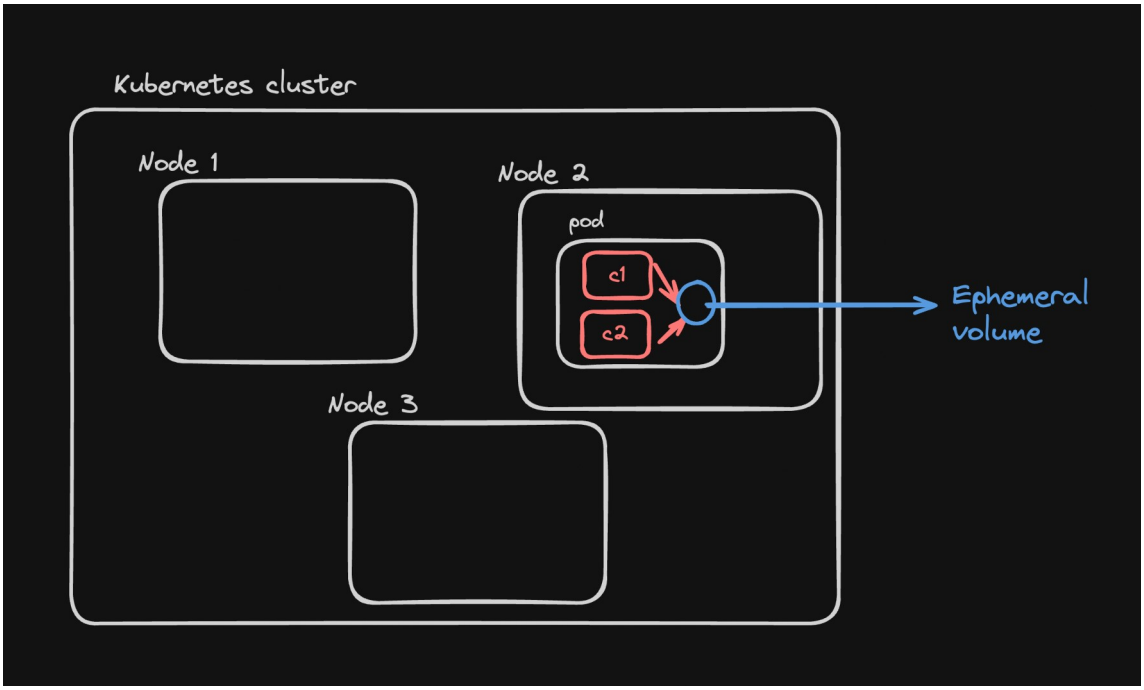
Volumes

In Kubernetes, a Volume is a directory, possibly with some data in it, which is accessible to a Container as part of its filesystem. Kubernetes supports a variety of volume types, such as EmptyDir, PersistentVolumeClaim, Secret, ConfigMap, and others.

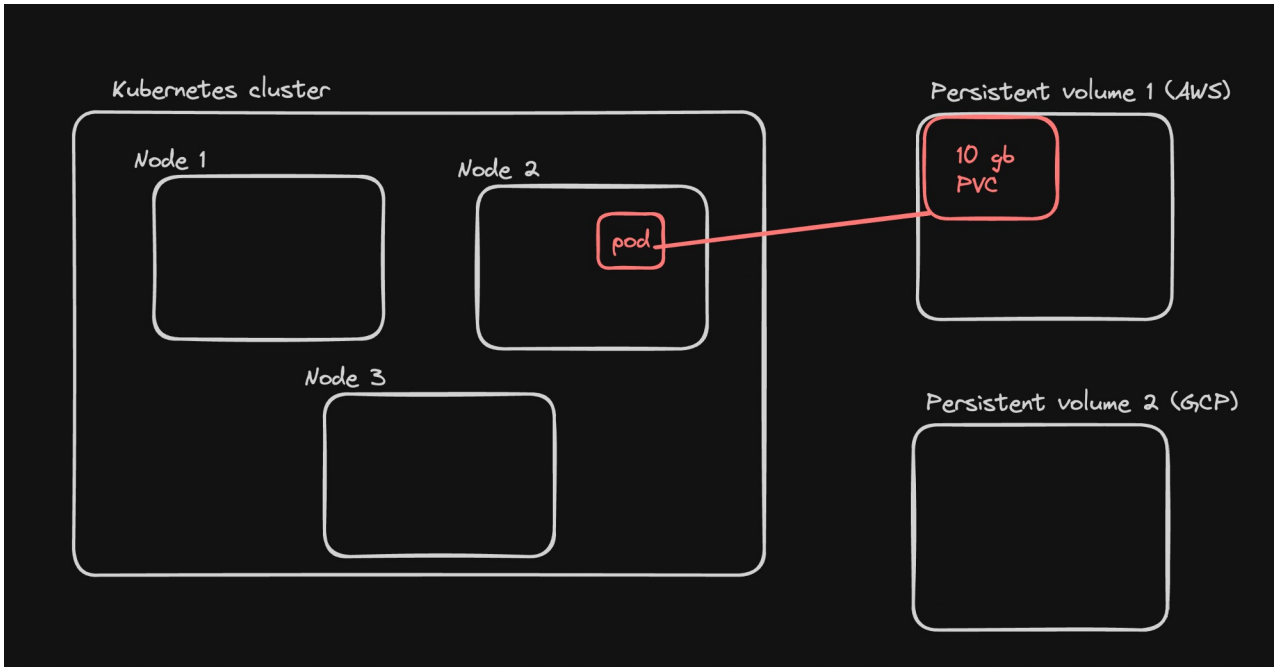
NOW LIKE IF YOU NEED LIKE 2 CLUSTERS WHICH ARE PRESENT IN A POD TO SHARE ONE COMMON ENV FILE IN BOTH FOR THAT WE HAVE TO USE THE EPHIMERAL VOLUME WHICH WILL LET US USE THE VOLUME IN BOTH THE PODS AS CURRENTLY THE PODS WHICH ARE CREATED CAN ONLY ACCESS EACH OTHER ON THE LOCALHOST IF THEY ARE IN A SINGLE POD

Why do you need volumes?

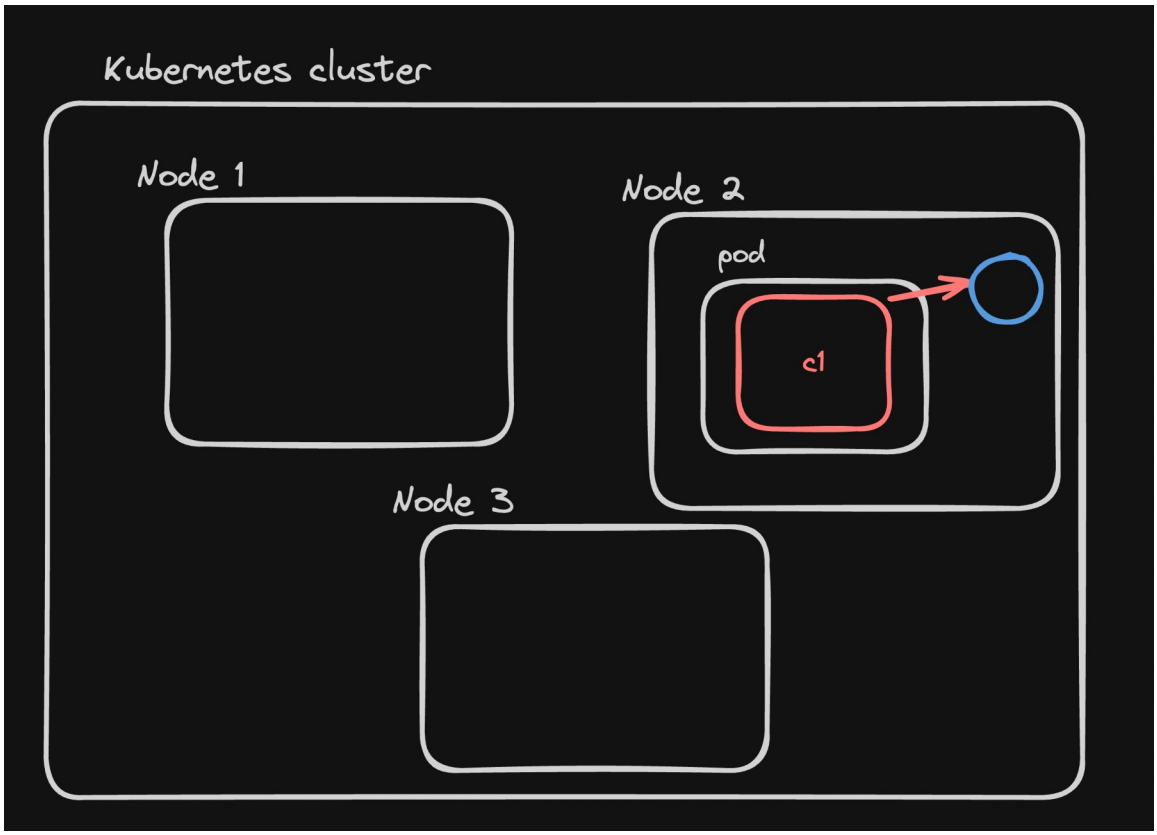
- If two containers in the same pod want to share data/fs.



- If you want to create a database that persists data even when a container restarts (creating a DB)



- Your pod just needs extra space during execution (for caching lets say) but doesnt care if it persists or not.



Types of volumes

Ephemeral Volume

Temporary volume that can be shared amongst various containers of a pod. When the pods dies, the volume dies with it.
For example –

1. ConfigMap
2. Secret
3. emptyDir(THIS ONE HELP THE CONTAINERS TO SHARE THE FILES AMONG EACH OTHER)

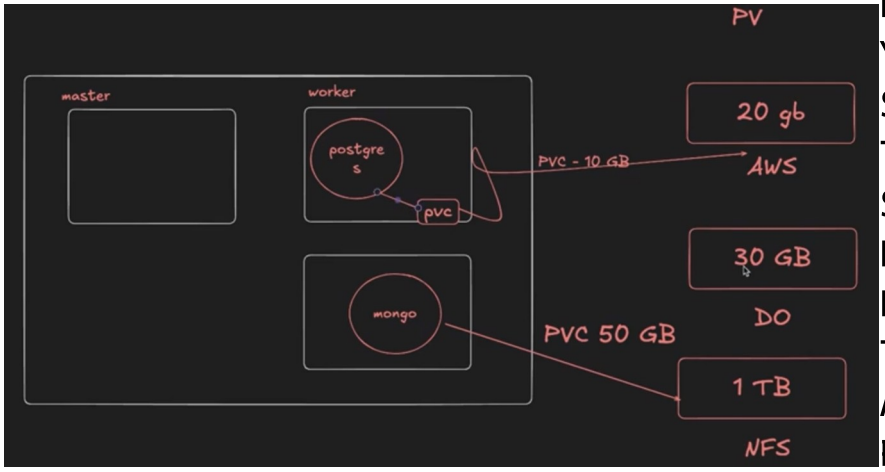
Persistent Volume

A Persistent Volume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes but

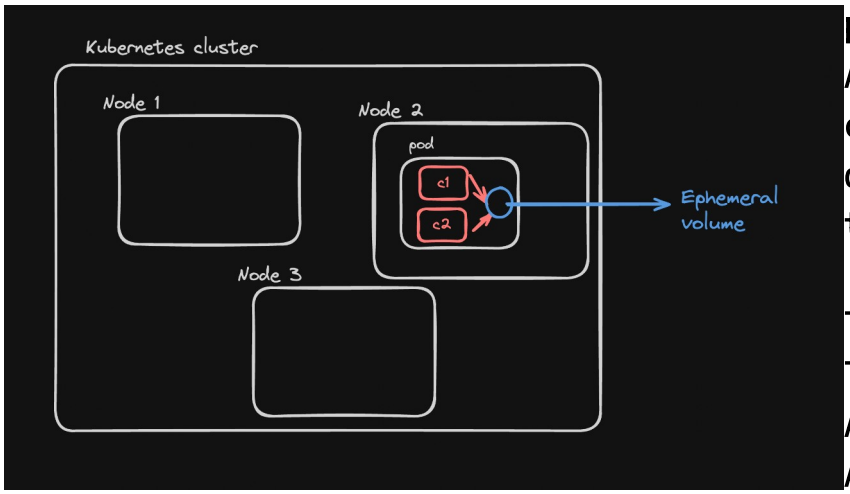
have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system. (WE CAN STORE THE DATA LIKE PERMENTALY ALSO IN DO WHICH IS CALLED AS VOLUME BLOCK STORAGE AND ALSO IN NAS ALSO)

Persistent volume claim

A Persistent Volume Claim (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., can be mounted once read/write or many times read-only).



PERSISTENT VOLUME CLAIM MEANS YOU ARE CLAIMING THE DATA SPACE IN THE AWS OR DO OR NFS THAT I WILL BE USING THAT SPACE FOR THE DATA STORAGE BCOZ IF WE DONT DO THAT THE POSTGRES DATA WILL BE GONE IF THE CONTAINER SHUTS DOWN
A POD NEED TO GET A PVC PERM FROM THE PV



Ephemeral volumes(EMPTY DIR)
A lot of times you want two containers in a pod to share data. But when the pods dies, then the data can die with it.

THIS IS NOT USED THAT MUCH AS THIS HELPS IN LIKE IF THE PODS ARE RUNNING THEY SHOULD HAVE ACCESS TO ONE STORAGE WHICH AN BE ACCESSED BY BOTH THE CONTAINERS

CREATING A YML FILE TO TEST THIS
NOW IN THIS WE HAVE 2 CONATINERS RUNNING AND A VOLUME
NEW THING CALLED AS **BUSYBOX** WHICH IS LIKE A INF RUNNING CODE

SO WHAT WE ARE DOING IN THIS YML FILE IS THERE ARE 2 CONTAINTER IN WHICH ONE CONTAINER IS CREATING A TXT FILE WITH SOME DATA AND OTHER FILE IS ACCESSING THAT AND USING THAT DATA AND THAT SHARING OF STORAGE BETWEEN THE VOLUME IS DONE USING THE EPHEMERAL VOLUME

VOLUMES = JOH JOH VOLUME HAAM USE KAR SKTE HAI WOH MENTION KI HAI
VOLUME MOUNT = JOH VOLUME IS POD MAI HAME USE KARNI HAI

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: shared-volume-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: shared-volume-app
  template:
    metadata:
      labels:
        app: shared-volume-app
    spec:
      containers:
      - name: writer
        image: busybox
        command: ["/bin/sh", "-c", "echo 'Hello from Writer Pod' > /data/hello.txt; sleep 3600"]
        volumeMounts:
        - name: shared-data
          mountPath: /data
      - name: reader
        image: busybox
        command: ["/bin/sh", "-c", "cat /data/hello.txt; sleep 3600"]
        volumeMounts:
        - name: shared-data
          mountPath: /data
      volumes:
      - name: shared-data
        emptyDir: {}
```

to run it use
kubectl apply -f manifest.yml

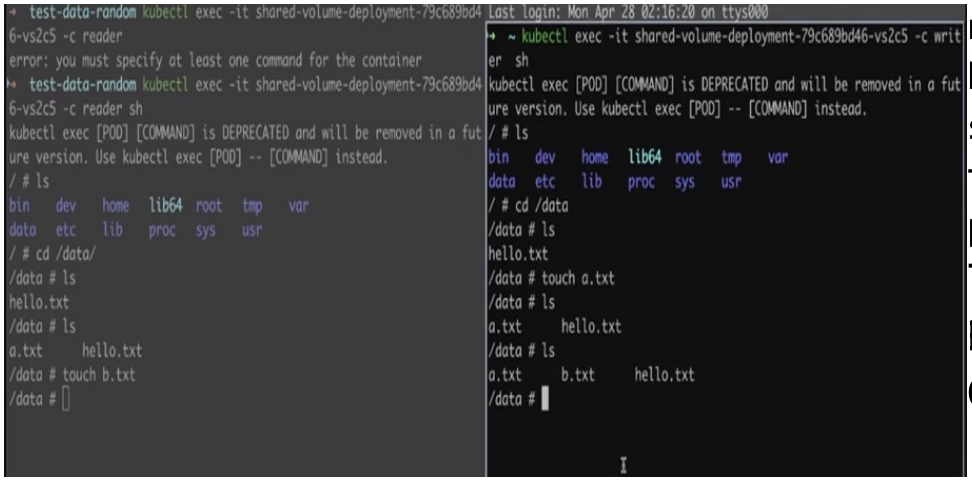
to see the deployment running
kubectl get deployment

kubectl get pods
you will see the pod running

now to see the logs use this
kubectl logs -f pod_name
Output:
Defaulted container "writer" out of: writer, reader

this is asking konse container kae logs dikhau abhi default writer kae dikha rha hai
to choose that use
kubectl logs -f pod_name -c reader
reader and writer are names joh hamne yml mai mention kiya tha
OUTPUT
Hello from Writer Pod
MEANS THEY ARE ABLE TO SHARE THE VOLUMES

TO GET THE SHELL ACCESS OF THE POD USE THIS
kubectl exec -it POD_NAME -c writer sh



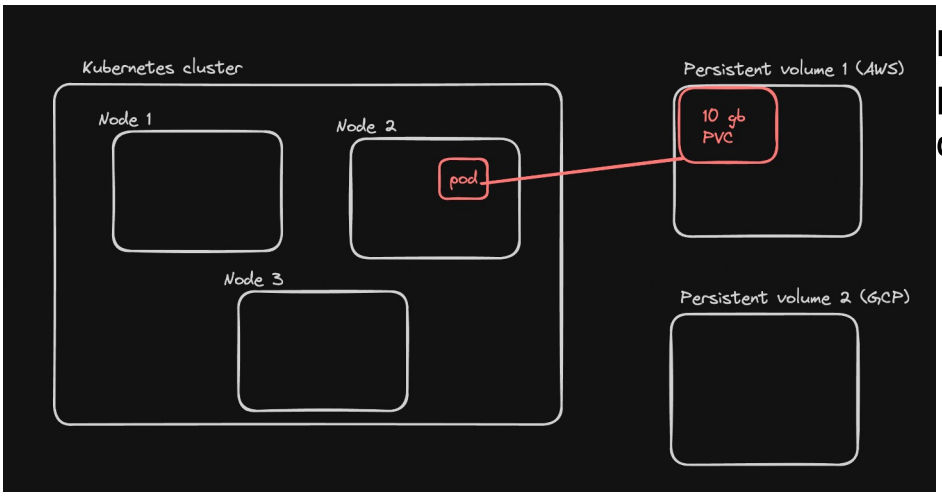
now you can see in this both the pods share the same storage as they have the same files in both the pods
THIS WAS ALL ABOUT THE EMPTY DIR EPHIMERAL CONTAINERS

VERY RARELY WE WILL USE THIS EMPTY DIR EPHIMERAL CONTAINERS

PERSISTENT VOLUMES

Just like our kubernetes cluster has nodes where we provision our pods.

We can create peristent volumes where our pods can claim (ask for) storage



Persistent volumes can be provisioned statically or dynamically.

Static persistent volumes(CREATING A NAS TYPE SHIT)
Creating a NFS(NETWORK FILE SYSTEM)

NFS is one famous implementation you can use to deploy your own persistent volume

I'm running one on my aws server -

NOW WE CREATED A DROPLET SSH INTO IT AND THEN INSTALLED DOCKER IN THAT MACHINE USING THE STEP 3 OF DOCKER INSTALL IN UBUNTU

AND THEN INSTALL DOCKER COMPOSE ALSO

NOW CREATING THE docker-compose.yml file

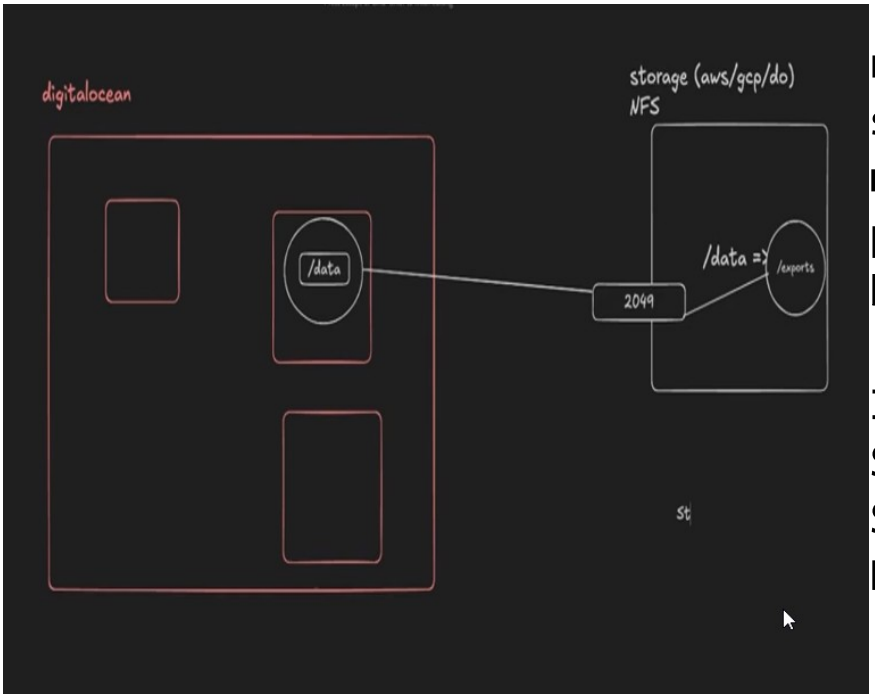
```
version: '3.7'

services:
  nfs-server:
    image: itsthenetwork/nfs-server-alpine:latest
    container_name: nfs-server
    privileged: true
    environment:
      SHARED_DIRECTORY: /exports
    volumes:
      - ./data:/exports:rw
    ports:
      - "2049:2049"
    restart: unless-stopped
```

and this requires a data folder so create it
mkdir data

and then run the docker-compose file
docker-compose up

now on the machine ip:2049(mentioned in the yml file something is running)
now this accept the connection for the storage



now we have created a storage and now we have to make a pvc and pv in the pod to make it accessible by it

ITS JUST LIKE TELLING K8S SOMETHING IS RUNNING SOMEWHERE WHICH HAS SOME PERSISTENT VOLUME

CREATING PV AND PVC

NOW WE ARE CREATING THE PERSISTED VOLUME
THIS IS THE YML FILE FOR THAT
pv.yml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteMany(read write access allowed)
  storageClassName: nfs(this the network file storage system)
  nfs:
    path: /exports(this is the folder path which we mentioned in the docker-compose.yml file)
    server: 52.66.197.168(volume ip)
```

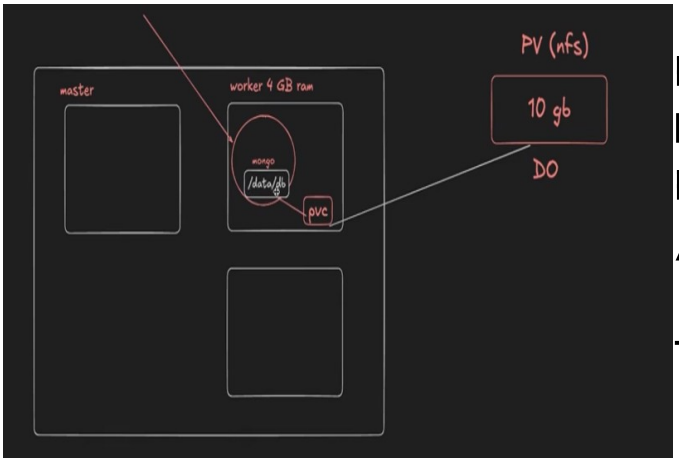
and now do

kubectl apply -f pv.yml
kubectl get pv
NOW WE HAVE TO CLAIM THE STORAGE USING PVC(PERSISTENT VOLUME CLAIM)
THIS IS PVC.YML

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  accessModes:
    - ReadWriteMany(THIS ACCESS SHOULD MATCH THE PC ACCESS ONLY THEN ONLY THIS CLAIM CAN BE )
  resources:
    requests:
      storage: 10Gi(THIS IS THE STORAGE WHICH IS MADE AND SHOULD BE LESS OR EQUAL TO THE PV MADE)
  storageClassName: nfs
```

THEN DO
kubectl apply -f pvc.yml
kubectl get pvc(STATUS SHOWS BOUND)
FOR THE PVC TO BE BOUND IT SHOULD SHOW IN THE PV ALSO
NOW WE HAVE A PERSISTENT VOLUME PV AND A PERSISTENT VOLUME CLAIM PVC AND NOW WE HAVE TO JUST MOUNT THAT IN THE POD

```
apiVersion: v1
kind: Pod
metadata:
  name: mongo-pod
spec:
  containers:
  - name: mongo
    image: mongo:4.4
    command: ["mongod", "--bind_ip_all"]
    ports:
    - containerPort: 27017(PORT ON WHICH THE MONGO WILL BE EXPOSED)
    volumeMounts:
    - mountPath: "/data/db"(PATH WHERE THE DATA WILL BE STORED IN THE DB)
      name: nfs-volume
  volumes:
  - name: nfs-volume(JUST A NAME TO BE USE IN THE MOUNT)
    persistentVolumeClaim:
      claimName: nfs-pvc(POD CAN ONLY TALK TO PVC THATS WHY THE PVC NAME IS MENTIONED HERE SO THAT IT CAN USE IT)
```



FOR MONGO THE FOLDER FOR STORING DATA IS /data/db
FOR POSTGRES ITS /var/lib/postgresql/data

THE CURRENT ARCHITECUTRE

CONTAINER IS MOUNT TO PVC AND PVC IS MOUNT TO THE PV
NOW JUST SAVE THE POD.YML FILE AND DO
kubectl apply -f pod.yml

IN THE DB MACHINE WE HAVE TO CREATE A EXPORTS FOLDER ALSO
ALSO WHENEVER SOMETHING IS NOT WORKING ALWAYS CHECK THE LOGS

now to check if the data is persisted
kubectl exec -it mongo-pod sh
now go to data/db (to check for the files)

same now go to the DO machine to check for the same folder
SAME FILES ARE SHOWN
BEAUTIFULLLLLLL…..

Try it out

- Put some data in mongodb

```
kubectl exec -it mongo-pod -- mongo
use mydb
db.mycollection.insert({ name: "Test", value: "This is a test" })
exit
```

- Delete and restart the pod

```
kubectl delete pod mongo-pod
kubectl apply -f mongo.yml
```

- Check if the data persists

```
kubectl exec -it mongo-pod -- mongo
use mydb
db.mycollection.find()
```

```
---
> use mydb
switched to db mydb
> db.mycollection.find()
{ "_id" : ObjectId("66659b1845d3d75115c37228"), "name" : "Test", "value" : "This is a test" }
>
```

STILL THE DATA IS PERSISTED IN THE DB
FOR NOW ITS WORKING
BUT ITS SLOW
ITS COMPLICATED ALSO
ITS EXPOSED OVER THE INTERNET
BUT ITS CHEAPER THAN THE BLOCK STORAGE IN DO

BUT NO ONE DOES THIS EVERYONE USES

Automatic pv creation(THIS IS LIKE WE JUST CREATE THE PVC AND THE PV IS AUTOMATICALLY CREATED)
THIS METHOD IS LIKE JUST CREATING THE PVC FILE AND THE PV IS AUTOMATICALLY CREATE DUMB DEV TYPE THING BUT IT STILL WORKS

Ref - https://docs.vultr.com/how-to-provision-persistent-volume-claims-on-vultr-kubernetes-engine

- Create a persistent volume claim with storageClassName set to `vultr-block-storage-hdd`

PVC.YML

JUST ADD THE NAME OF THE STORAGE CLASS NAME

THIS USES **VOLUME BLOCK STORAGE**

FOR MANUAL ITS NFS FOR DO = do-block-storage

for vultr = vultr-block-storage-hdd

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: csi-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 40Gi
  storageClassName: vultr-block-storage-hdd
```

THIS AUTOMATICALLY CREATED THE PERSISTENT VOLUME
NOW IF YOU SEE kubectl get pvc(IT WILL SHOW ITS BOUNDED)

kubectl get pv(THIS WILL SHOW THAT THE STORAGE WAS CREATED)

pod manifest.yml file

```
apiVersion: v1
kind: Pod
metadata:
  name: mongo-pod
spec:
  containers:
  - name: mongo
    image: mongo:4.4
    command: ["mongod", "--bind_ip_all"]
    ports:
    - containerPort: 27017
    volumeMounts:
    - name: mongo-storage
      mountPath: /data/db
  volumes:
  - name: mongo-storage
    persistentVolumeClaim:
      claimName: csi-pvc
```

Explore the resources created

```
kubectl get pv
kubectl get pvc
kubectl get pods
```

- **Put some data in mongodb**

```
kubectl exec -it mongo-pod -- mongo
use mydb
db.mycollection.insert({ name: "Test", value: "This is a
test" })
exit
```

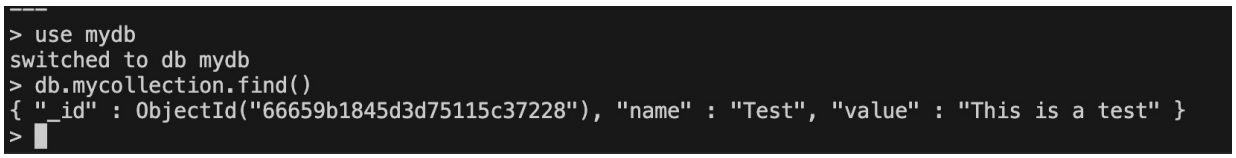
- **Delete and restart the pod**

```
kubectl delete pod mongo-pod
kubectl apply -f mongo.yml
```

- **Check if the data persists**

```
kubectl exec -it mongo-pod -- mongo
```

use mydb
db.mycollection.find()



THIS STORED ALL THE DB DATA
LIKE IN THE 100XDEVS WEBSITE IT STORES THE ALL THE CREDS INSIDE THE DO

THE LAST THING WE USE THE MOST AND LEARN THAT PROPERLY